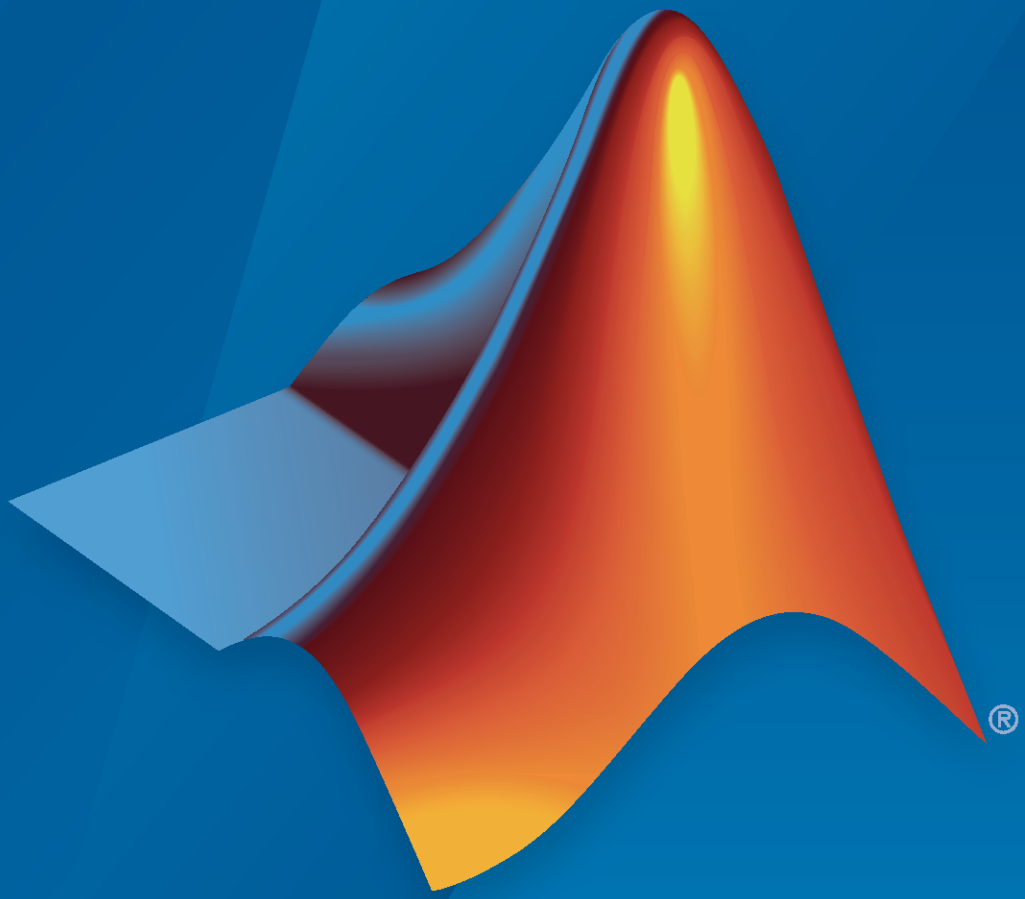
Communications Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2021b

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™ User's Guide

Revision History

April 2011	First printing	New for Version 5.0 (Release 2011a)
September 2011	Online only	Revised for Version 5.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.3 (Release 2012b)
March 2013	Online only	Revised for Version 5.4 (Release 2013a)
September 2013	Online only	Revised for Version 5.5 (Release 2013b)
March 2014	Online only	Revised for Version 5.6 (Release 2014a)
October 2014	Online only	Revised for Version 5.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release 2016b)
March 2017	Online only	Revised for Version 6.4 (Release 2017a)
September 2017	Online only	Revised for Version 6.5 (Release 2017b)
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)
March 2021	Online only	Revised for Version 7.5 (Release 2021a)
September 2021	Online only	Revised for Version 7.6 (Release 2021b)

Contents

1 Shared comm_simrf Examples

2 Shared spc_channel Examples (comm/antenna/phased)

3 Bluetooth Toolbox Examples

4 Shared deeplearning_shared Examples (comm/deeplearning)

5 Shared phased_comm Examples (comm/phased)

6 HDL Coder Featured Examples

Communications Toolbox Library for ZigBee and UWB - Featured Examples

7

Communications Toolbox Featured Examples

8

9 Input, Output, and Display

10 Data and Signal Management

11 Digital Modulation

12 Various User Guide Topic Examples

13 Bluetooth Tutorials

14 Equalization

15 Equalizer Examples (new & old)

16 System Design

17 OFDM Modulation

20 Galois Fields

21 Error Detection and Correction

22 Channel Modeling and RF Impairments

23 Measurements

24 Filtering Section

25 Visual Analysis

26 C Code Generation

27 HDL Code Generation

28 Simulation Acceleration

29 Wireless Waveform Generator App

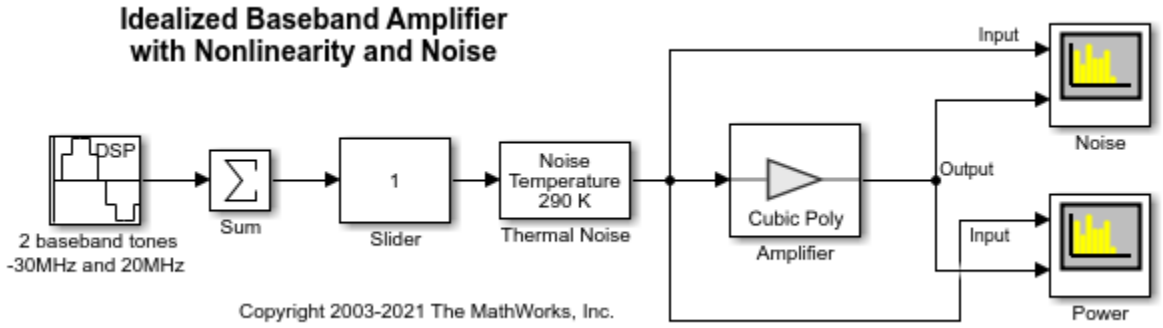
30 RF Propagation

31 Guidance for Discouraged Features

Shared comm_simrf Examples

Idealized Baseband Amplifier with Nonlinearity and Noise

The example shows how to use the idealized baseband library Amplifier block to amplify a signal with nonlinearity and noise. The Amplifier uses the `Cubic Polynomial` model with a `Linear power gain` of 10 dB, an `Input IP3` nonlinearity of 30 dBm, and a `Noise figure` of 3 dB.



System Architecture

The DSP Sine Wave block inputs two complex baseband tones with a power level of -20 dBm and -25 dBm at frequencies of -30 MHz and 20 MHz. In this block you can also:

- Increase the samples per frame to increase the simulation speed.
- Use output complexity and phase offset to control the I-Q relationship of each baseband signal
- Control the bandwidth of the scopes using the inverse of the sample time parameter.

The Amplifier block only accepts a vector input. The Sum block combines the two baseband signals into a vector length equal to the samples per frame in the DSP Sine Wave block.

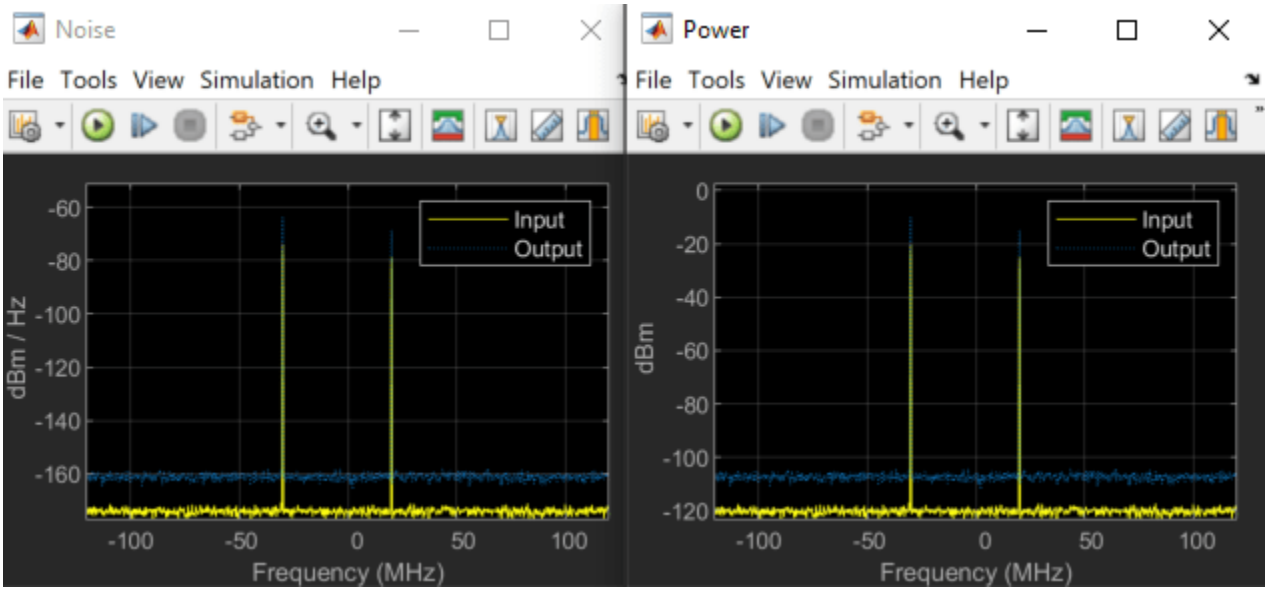
The Thermal Noise block creates a thermal noise floor input of -174 dBm/Hz.

Simulation Analysis

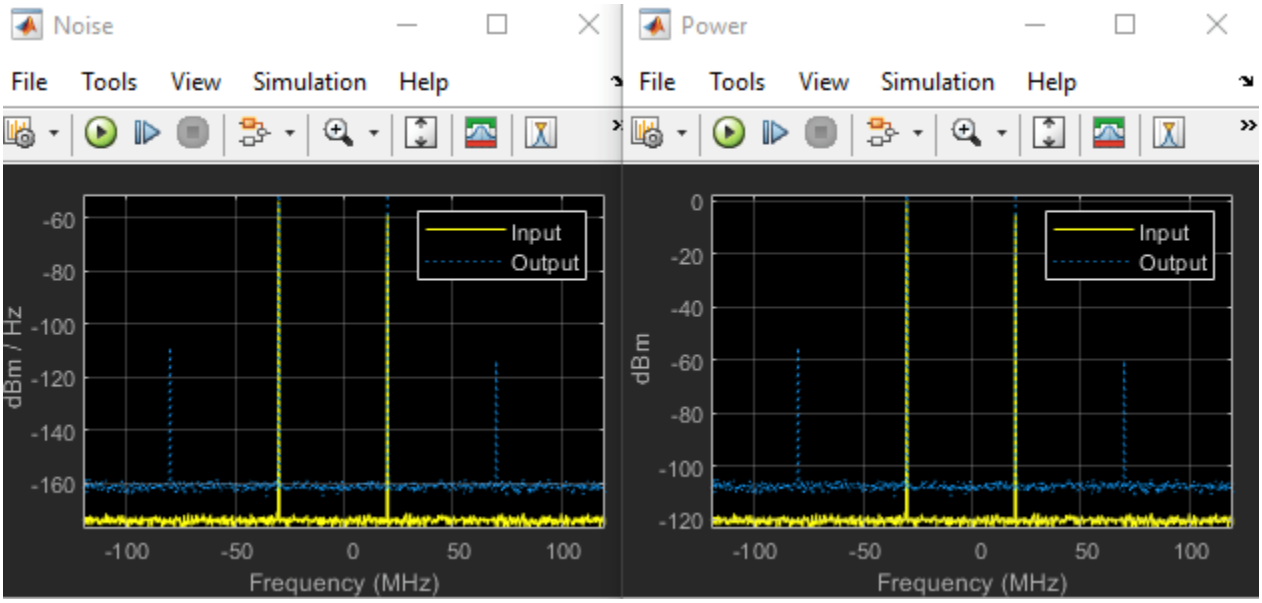
The Amplifier block with `Linear power gain` of 10 dB outputs tone with magnitude -10 dBm and -15 dBm as seen in the Power plot. The Amplifier also increases the thermal noise floor to -161 dBm/Hz. You can calculate the output thermal noise using this equation:

$$InputNoiseFloor + linearpowergain + NoiseFigure = -174dBm/Hz + 10dBm + 3dBm = -161dBm/Hz$$

The following plots illustrate the differences in the input and output noise floors. The spurs appear at 70 MHz (2*20 MHz + 30 MHz) and -80 MHz (2*(-30 MHz) - 20 MHz). This shows the third order intercept nature of the spurs.



Increasing the Slider value from 1 to 10, shows nonlinear effects in the plots. These are the Noise and Power plots when the gain of the Slider is 10.



See Also

Amplifier

Related Examples

- “Impact of RF Effects on Communication System Performance” on page 8-32
- “Impact of Thermal Noise on Communication System Performance” on page 1-42

Power Amplifier Characterization

This example shows how to characterize a power amplifier (PA) using measured input and output signals of an NXP Airfast PA. Optionally, you can use a hardware test setup including an NI PXI chassis with a vector signal transceiver (VST) to measure the signals at run time.

You can use the characterization results to simulate the PA using the `comm.MemorylessNonlinearity` System object or Memoryless Nonlinearity block. For a PA model with memory, you can use Power Amplifier (RF Blockset) block. You can use these models to design digital predistortion (DPD) using `comm.DPD` and `comm.DPDCoefficientEstimator` System objects or DPD and DPD Coefficient Estimator blocks. For more information, see “Digital Predistortion to Compensate for Power Amplifier Nonlinearities” on page 1-31.

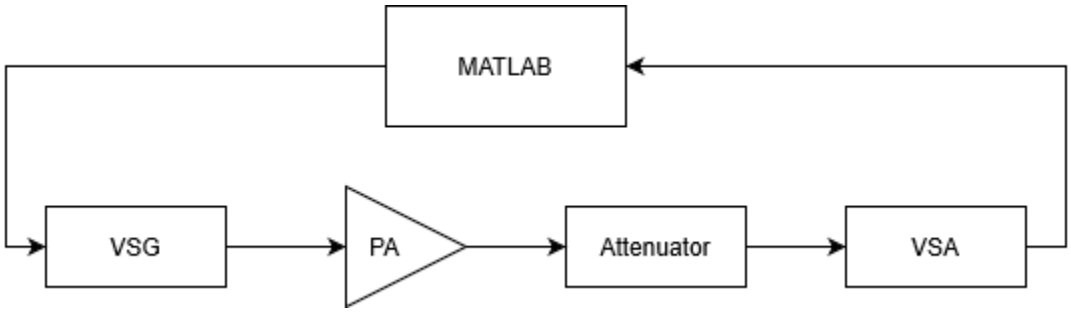
Optional Hardware and Software

This example can run on an NI PXI chassis with a VST to measure PA input and output signals during run time. The VST is a high-bandwidth RF instrument that combines a Vector Signal Generator (VSG) with a Vector Signal Analyzer (VSA). The following NI PXI chassis configuration was used to capture the saved signal:

- NI PXIe-5840 Vector Signal Transceiver (VST)
- NI PXIe-4139 Source Measure Unit (SMU)
- NI PXIe-4145 SMU
- NI RFmx SpecAn software
- NI-RFSG software
- NI-RFSG Playback Library software

As the device under test (DUT), this example uses an NXP Airfast LDMOS Doherty PA with operating frequency 3.6-3.8 GHz and 29 dB gain. This PA requires 29V, 5V, 3 V, 1.6V and 1.4V DC bias, which are provided using PXIe-4139 and PXIe-4145 SMUs.

Install MATLAB® on the NI PXI controller to run this example with the hardware setup, which is illustrated in the following figure. MATLAB, running on the PXI controller, generates test waveform and downloads the waveform to the VSG. The VSG transmits this test waveform to the PA and the VSA receives the impaired waveform at the PA output. MATLAB collects the PA output from the VSA and performs PA characterization.



Set dataSource variable to `"Hardware"` to run a test signal though the PA using the hardware setup described above. The test signal can be either a 5G-like OFDM waveform or two tones, as described in the following section. Set dataSource variable to `"From file"` to use prerecorded data.

```
dataSource = From file;
```

Generate Test Signals

If `testSignal` is `"OFDM"`, this example uses a 5G-like OFDM waveform with 64-QAM modulated signals for each subcarrier. If `testSignal` is `"Tones"`, this example uses two tones at 1.8 MHz and 2.6 MHz, to test the intermodulation caused by the PA.

```
testSignal = OFDM;
switch testSignal
  case "OFDM"
    bw = 100 MHz;
    [txWaveform,sampleRate,numFrames] = helperPACharGenerateOFDM(bw);
  case "Tones"
    bw = 3e6;
    [txWaveform,sampleRate,numFrames] = helperPACharGenerateTones();
end
```

To identify high order nonlinearities, the test signal must be oversampled at least by the amount of expected order of nonlinearity. In this example, we run a grid search up to nonlinearity order of seven. Upsample by seven to cover possible seventh order nonlinearities. Also, normalize the waveform amplitude.

```
overSamplingRate = 7;
filterLength = 6*70;
lowpassfilter = firpm(filterLength, [0 8/70 10/70 1], [1 1 0 0]);
firInterp = dsp.FIRInterpolator(overSamplingRate, lowpassfilter);
txWaveform = firInterp([txWaveform; zeros(filterLength/overSamplingRate/2,1)]);
txWaveform = txWaveform((filterLength/2)+1:end,1);      % Remove transients
txWaveform = txWaveform/max(abs(txWaveform));   % Normalize the waveform
sampleRate = sampleRate * overSamplingRate;
```

Hardware Test

If the `dataSource` variable is set to `"From file"`, load the prerecorded data. If the `dataSource` variable is set to `"Hardware"`, run the test signal through the PA using the VST. Create a helperVSTDriver object to communicate with the VST device. Set the resource name to the resource name assigned to the VST device. This example uses `'VST_01'`. For NI devices, you can find the resource name using the NI Measurement & Automation Explorer (MAX) application.

```
if strcmp(dataSource, "Hardware")
  VST = helperVSTDriver('VST_01');
```

Set the expected gain values of the DUT and the attenuator. Since PA output is connected to a 30 dB attenuator, set VSA external attenuation to 30. Set the expected gain of the DUT to 29 dB and gain accuracy to 1 dB. Set the acquisition time to a value that will result in about 40k samples. Set the target input power to 8 dBm. You can increase this value to drive the PA more into the non-linear region.

```
  VST.DUTExpectedGain     = 29;     % dB
  VST.ExternalAttenuation = 30;     % dB
  VST.AcquisitionTime     = 0.9e-3*(53.76e6/sampleRate); % seconds
  VST.DUTTargetInputPower = 8;  % dBm
  VST.CenterFrequency     = 3.7e9   % Hz
```

Download the test waveform to the VSG. Measure PA output.

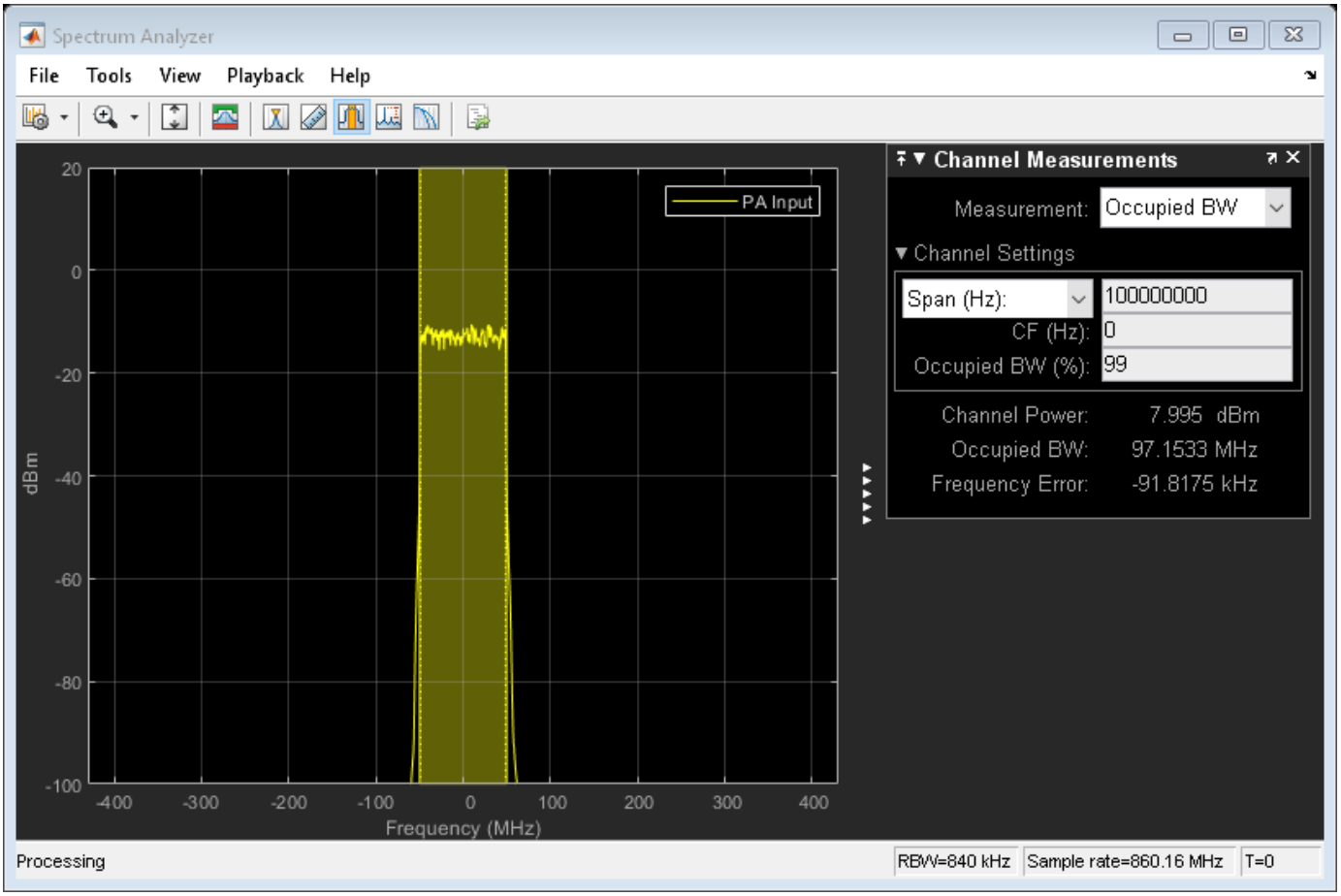
```
  writeWaveform(VST,txWaveform,sampleRate,testSignal)
  results = runPAMeasurements(VST);
  release(VST)
else
  % Load the prerecorded results from VST
  switch testSignal
    case "OFDM"
      dataFileName = sprintf("helperPACharSavedData%dMHz",bw/1e6);
    case "Tones"
      dataFileName = "helperPACharSavedDataTones";
  end
  load(dataFileName,"results","sampleRate","overSamplingRate","testSignal","numFrames")
end
```

Map results into local variables.

```
referencePower = results.ReferencePower;
measuredAMToAM = results.MeasuredAMToAM;
paInput = results.InputWaveform;
paOutput = results.OutputWaveform;
linearGaindB = results.LinearGain;
```

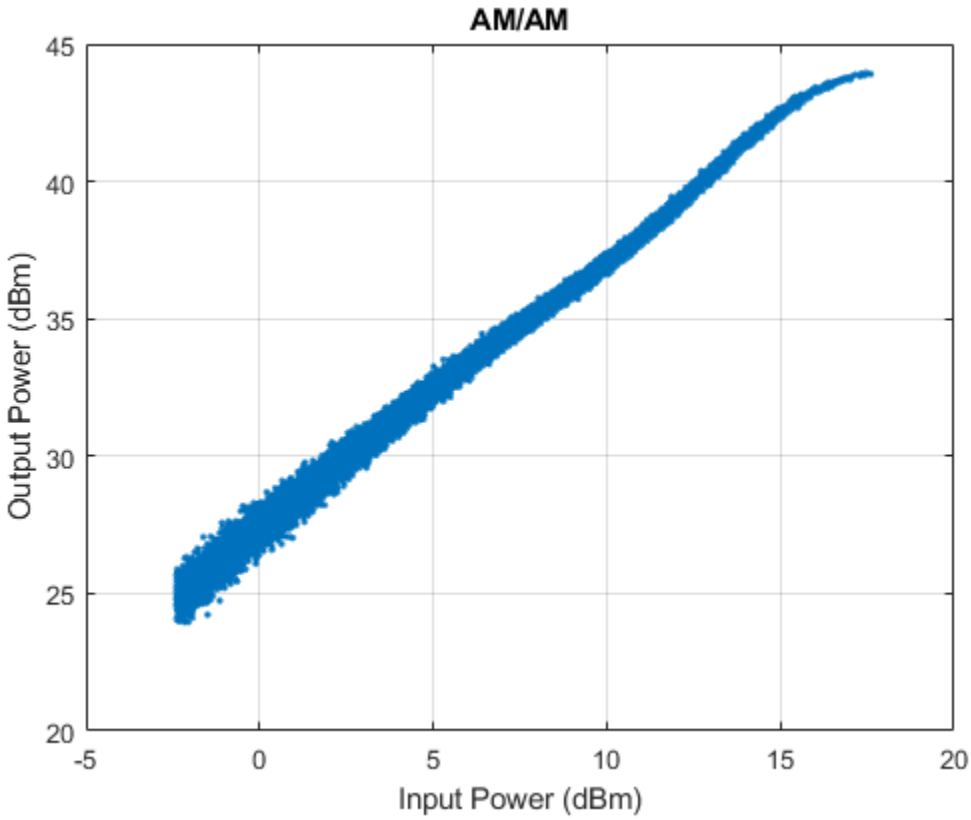
Plot the spectrum of the test signal using `dsp.SpectrumAnalyzer` System object.

```
saInput = helperPACharPlotInput(paInput, sampleRate, testSignal, bw);
```



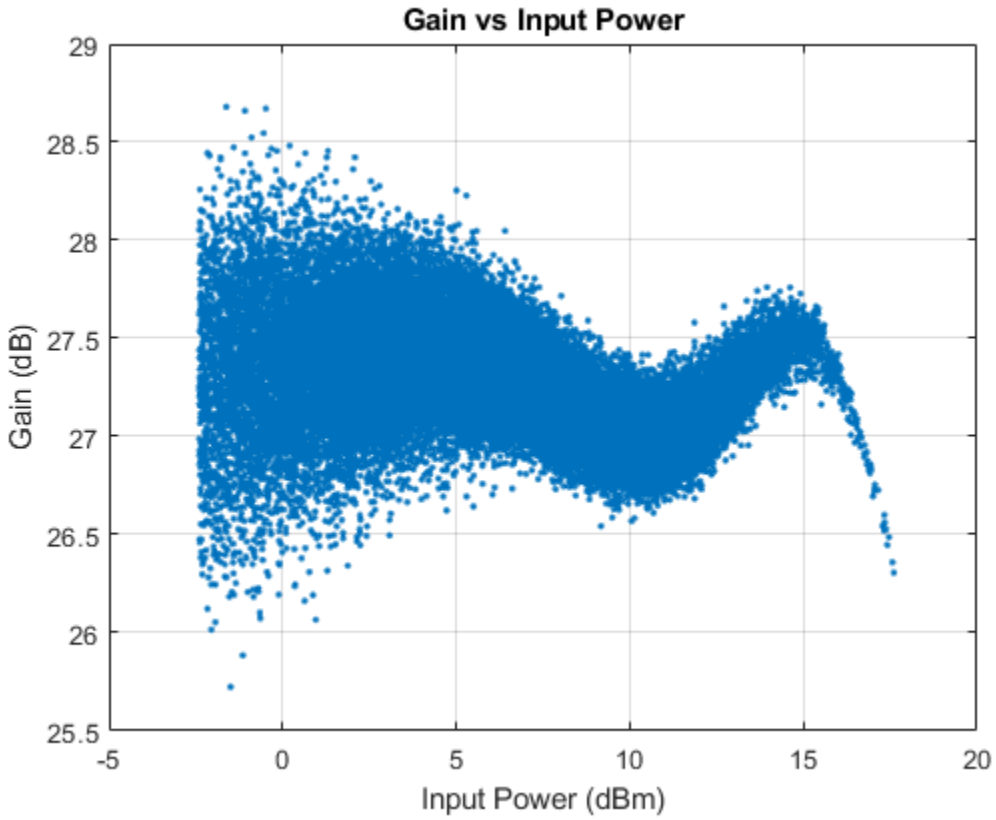
Plot the AM/AM characteristics of the PA.

```
helperPACharPlotSpecAnAMAM(referencePower, measuredAMToAM)
```



For a better view, focus on gain vs input power instead of output power vs input power and plot again.

```
helperPACharPlotSpecAnGain(referencePower, measuredAMToAM)
```



The PA is mostly linear of the input power range -1 to 17 dBm, with only about 1dB variation over that range. The width of the gain curve is due to the memory effects of the PA.

PA Characterization

Use the measured PA input and output data to model the PA. Then, you can use this model to simulate a system that contains this PA and fine tune the parameters. This example considers three models: memoryless nonlinearity, memory polynomial and memory polynomial with cross terms.

Memoryless Nonlinearity Model

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). The `comm.MemorylessNonlinearity` System object and Memoryless Nonlinearity block implements several such distortions. Use the PA input and output data to create a lookup table to use with this object or block.

To characterize the AM/AM transfer function, calculate the average output power for a range of input power values. Measurements are in volts over an overall 100 ohm impedance, split between the transmitter and receiver. Convert the measured baseband samples to power values in dBm. The +30 dB term is for dBW to dBm conversion and the -20 dB term is for the 100 ohm impedance.

```
paInputdBm  = mag2db(abs(paInput)) + 30 - 20;
paOutputdBm  = mag2db(abs(paOutput)) + 30 - 20;
```

Partition the input power values into bins. The `edges` variable contains the bin edges, and the `idx` variable contains the index of the bin values for each input power value.

```
[N,edges,idx] = histcounts(paInputdBm, 'BinWidth', 0.5);
```

For each bin, calculate the midpoint of the bin, average output power and average phase shift. Do not include any input power value that is less than 20 dB below the maximum input power. Store the results in a three-column matrix where the first column is the input power in dBm, second column is the output power in dBm and last column is the phase shift.

```
minInPowerdBm = max(paInputdBm) - 20;
minIdx = find(edges < minInPowerdBm, 1, 'last');
tableLen = length(edges)-minIdx-1;
inOutTable = zeros(tableLen,2);
for p = minIdx+1:length(edges)-1
  inOutTable(p-minIdx,1) = mean(paInputdBm(idx == p));   % Average input power for current bin
  inOutTable(p-minIdx,2) = mean(paOutputdBm(idx == p));  % Average output power for current bin
  inOutTable(p-minIdx,3) = mean(angle(paOutput(idx == p)./paInput(idx == p)));  % Average phase s
end
```

Use the table in the comm.MemorylessNonlinearity System object to model the PA. Compare the estimated output with the actual output.

```
pa = comm.MemorylessNonlinearity('Method','Lookup table','Table',inOutTable,'ReferenceImpedance',
```

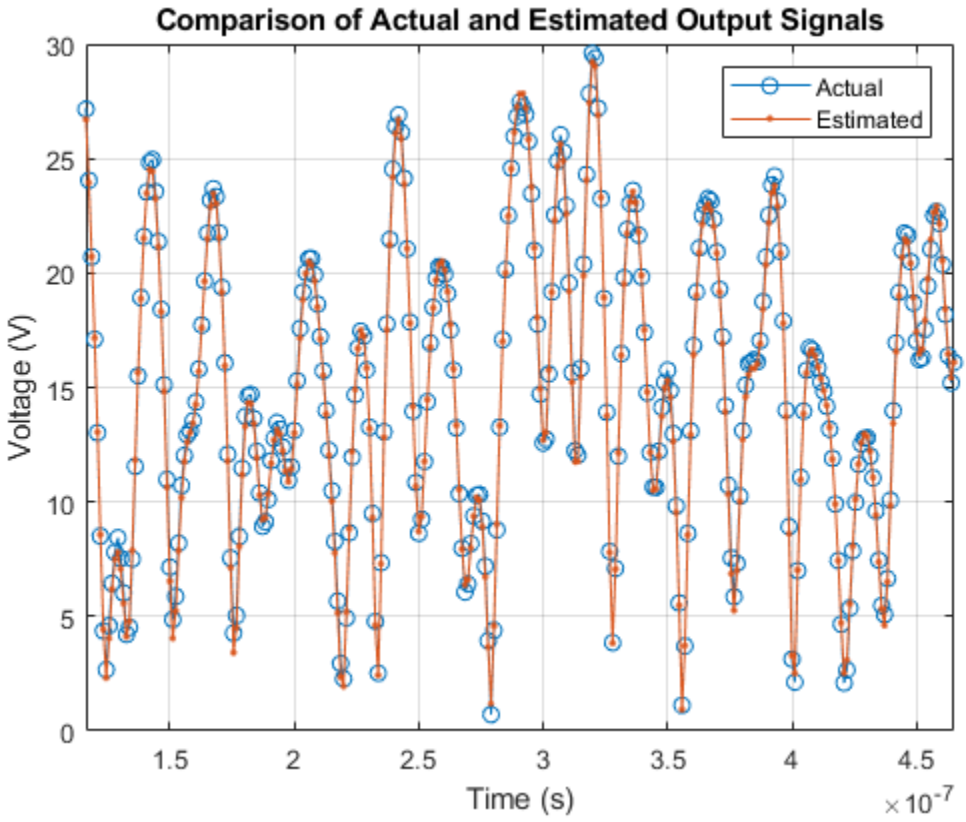
```
pa =
  comm.MemorylessNonlinearity with properties:

               Method: 'Lookup table'
                Table: [40x3 double]
    ReferenceImpedance: 100
```

```
paOutputFitMemless = pa(paInput);
err = abs(paOutput - paOutputFitMemless)./abs(paOutput);
rmsErrorMemless = rms(err)*100;
disp(['Percent RMS error in time domain is ' num2str(rmsErrorMemless) '%'])
```

```
Percent RMS error in time domain is 12.1884%
```

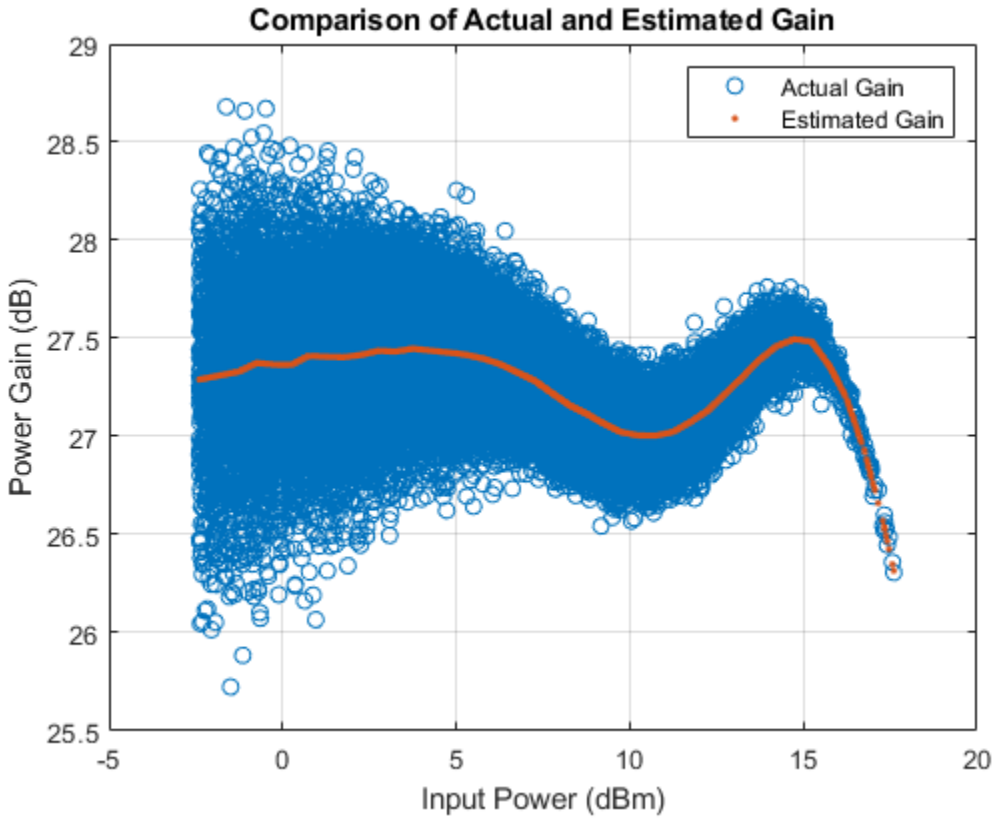
To visualize both the measured output signal and the fitted output signal, plot the actual and fitted time-domain output voltages.

```
helperPACharPlotTime(paOutput, paOutputFitMemless, sampleRate)
```



Plot the magnitude of the gain.

```
helperPACharPlotGain(paInput, paOutput, paOutputFitMemless)
```



Memory Polynomial Model

The memory polynomial model includes the memory effects of the PA in addition to the nonlinear gain. Use the multipurpose helper function helperPACharMemPolyModel to determine the complex coefficients of a memory polynomial model for the amplifier characteristics. Set the model type to `'Memory Polynomial'`.

`modType =` Memory Polynomial ;

Perform a grid search as shown in Appendix Grid Search for Memory Length and Polynomial Order on page 1-0 . Based on this grid search results, the best fit is obtained when memory length and polynomial degree values are as follows:

```
memLen = 5;
degLen = 5;
```

Perform the fit and RMS error calculation for these values. Only half of the data is used to compute the fitting coefficients, as the whole data set will be used to compute the relative error. The helper function helperPACharMemPolyModel calculates the coefficients of the model.

```
numDataPts = length(paInput);
halfDataPts = round(numDataPts/2);
```

The helper function helperPACharMemPolyModel is editable for custom modifications, and to return the desired matrix. The PA model has some zero valued coefficients, which results in a rank deficient matrix.

```
fitCoefMatMem = helperPACharMemPolyModel('coefficientFinder',              ...
  paInput(1:halfDataPts),paOutput(1:halfDataPts),memLen,degLen,modType);
```

```
Warning: Rank deficient, rank = 24, tol =  1.870573e-01.
```

```
disp(abs(fitCoefMatMem))
```

```
   23.1553    8.8536   17.8391   13.3033    3.2171
         0   11.7675   26.4648   23.1902    5.5469
   20.9748   16.8511   25.7274   22.1880    5.0680
   32.6202    8.4028    9.4851   10.6957    2.5609
   15.3879    2.3639    2.0886    2.9343    0.7370
```

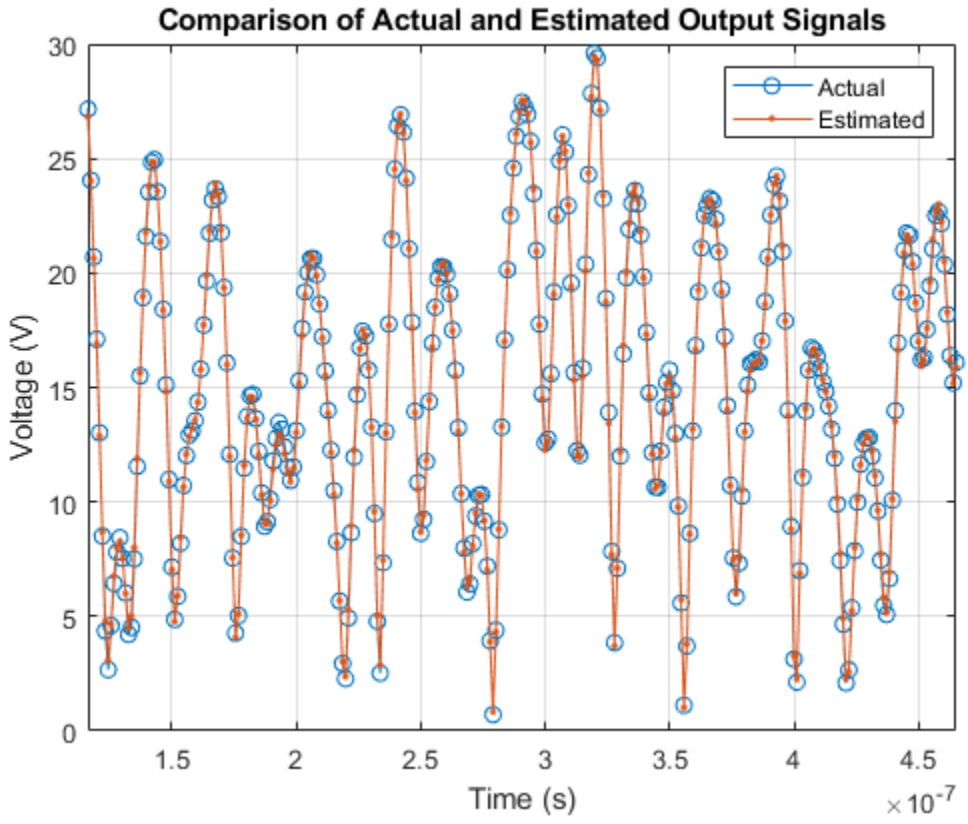
To validate the fitting, use the helper function to compute percent RMS error with respect to the measured signal.

```
rmsErrorTimeMem = helperPACharMemPolyModel('errorMeasure', ...
  paInput, paOutput, fitCoefMatMem, modType);
disp(['Percent RMS error in time domain is ' num2str(rmsErrorTimeMem) '%'])
```

```
Percent RMS error in time domain is 6.1057%
```

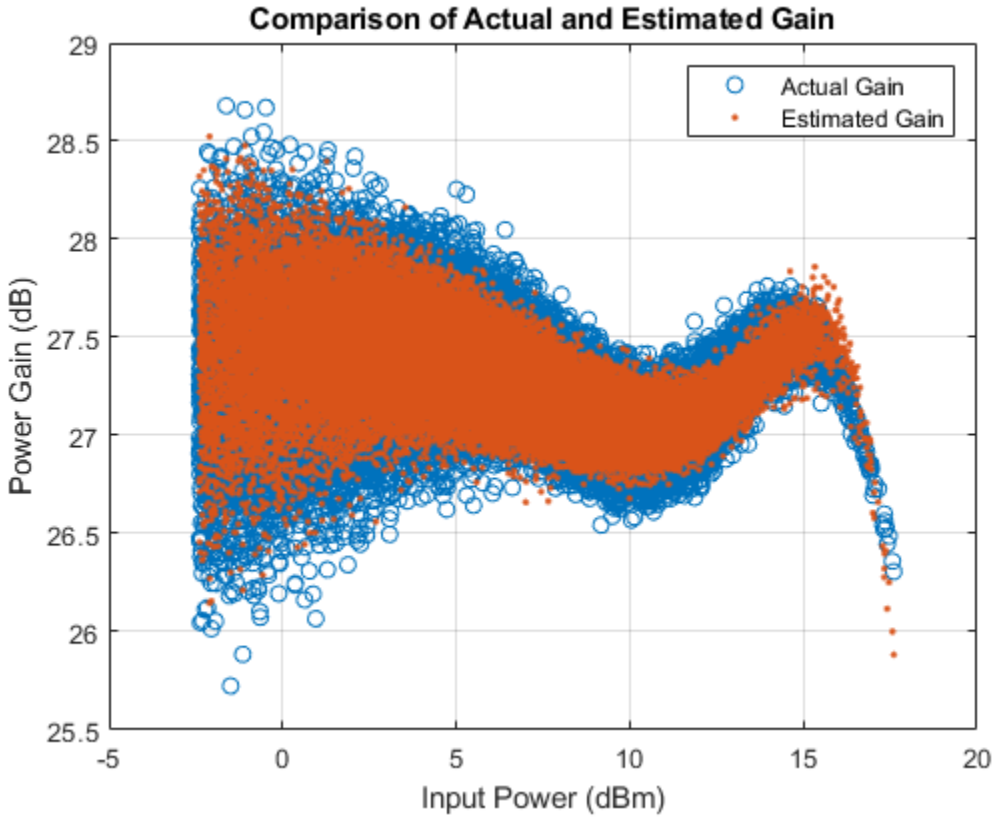
To visualize both the measured output signal and the fitted output signal, plot the actual and fitted time-domain output voltages.

```
paOutputFitMem = helperPACharMemPolyModel('signalGenerator', ...
  paInput, fitCoefMatMem, modType);
helperPACharPlotTime(paOutput, paOutputFitMem, sampleRate)
```



Plot the magnitude of the gain.

```
helperPACharPlotGain(paInput, paOutput, paOutputFitMem)
```

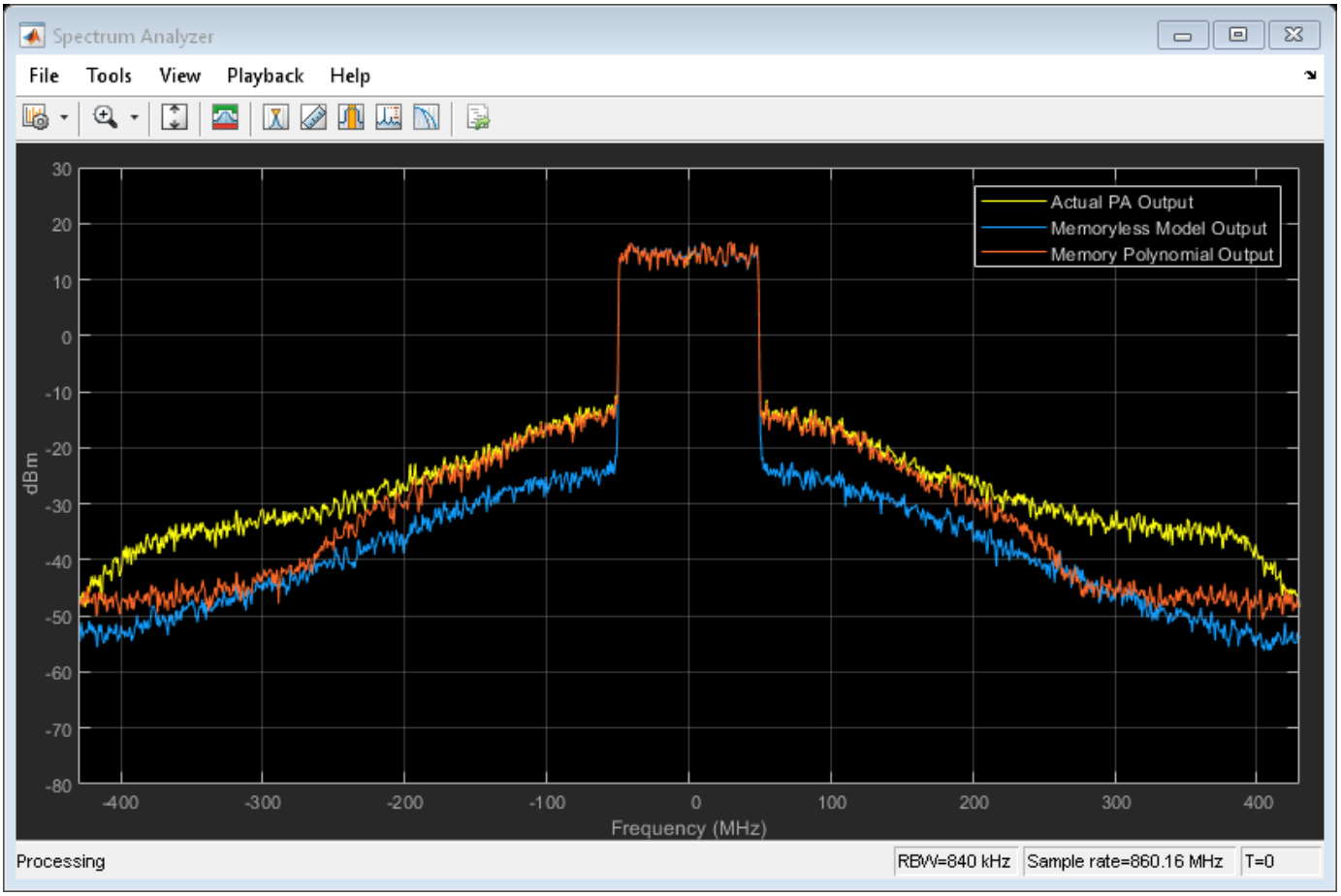


Discussions

The percent RMS estimation error in time domain for the memoryless nonlinearity model, which is between 9% and 13%, is about 3 to 4 times more than the error for the memory polynomial model is, which is between 2% and 6%, for the OFDM signals with different bandwidths.

Check the estimation error in frequency domain by plotting the spectrum of the actual PA output together with the spectrum of the estimated PA output for all three models. The memoryless nonlinearity table lookup model is not able to simulate the spectral growth seen in the measured PA output. For this PA, memory polynomial model provides a good approximation of the PA characteristics.

```
sa = helperPACharPlotSpectrum(...
  [paOutput paOutputFitMemless paOutputFitMem],...
  {'Actual PA Output','Memoryless Model Output', ...
  'Memory Polynomial Output'},...
  sampleRate,testSignal);
```



The helper function helperPACharMemPolyModel can also use the memory polynomial with cross terms model, which includes the leading and lagging memory cross terms in addition to the memory effects of the PA and the nonlinear gain. Set the model type to `'Cross-Term Memory'` to explore this model.

For further exploration, try different memory length and polynomial degree combinations. Modify the oversampling factor and explore its effect on the PA model performance. Modify the helper function helperPACharMemPolyModel to try different PA models.

Using PA Model for DPD Testing

Save the coefficient matrix of the PA model to be used in the Power Amplifier (RF Blockset) block for simulation at the system-level in the “Digital Predistortion to Compensate for Power Amplifier Nonlinearities” on page 1-31.

```
frameSize = floor(length(paInput)/numFrames);
paIn.signals.values = double(reshape(paInput(1:frameSize*numFrames,1),numFrames,frameSize));
paIn.signals.dimensions = frameSize;
paIn.time = [];
save('PAcoefficientsAndInput.mat','modType','fitCoefMatMem','memLen','degLen','paIn','linearGain
```

Appendix: Grid Search for Memory Length and Polynomial Order

Uncomment following lines to perform the grid search when the cost function is the percent RMS error in time. First choose the model type.

```
modType = Memory Polynomial;
% rmsErrorTime = helperPACharGridSearchTime(paInput,paOutput,modType,overSamplingRate)
```

Repeat the search when the cost function is the percent RMS error in frequency.

```
% rmsErrorFreq = helperPACharGridSearchFrequency(paInput,paOutput,modType,overSamplingRate)
```

Top-Down Design of an RF Receiver

This example designs an RF receiver for a ZigBee®-like application using a top-down methodology. It verifies the BER of an impairment-free design, then analyzes BER performance after the addition of impairment models. The example uses the `RF Budget Analyzer` App to rank the elements contributing to the noise and nonlinearity budget.

Design specifications:

- Data rate = 250 kbps
- OQPSK modulation with half sine pulse shaping, as specified in IEEE® 802.15.4 for the physical layer of ZigBee
- Direct sequence spread spectrum with chip rate = 2 Mchips/s
- Sensitivity specification = -100 dBm
- Bit Error Rate (BER) specification = 1e-4
- Analog to digital converter (ADC) with 10 bits and 0 dBm saturation power

To create fully standard-compliant ZigBee waveforms, you can use the *Communications Toolbox Library for the ZigBee Protocol* Add-on.

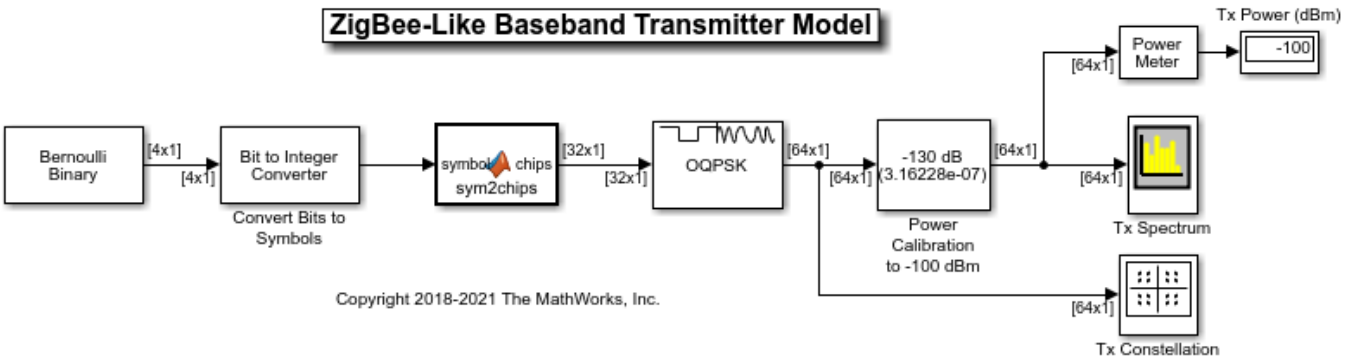
This example guides you through the following steps:

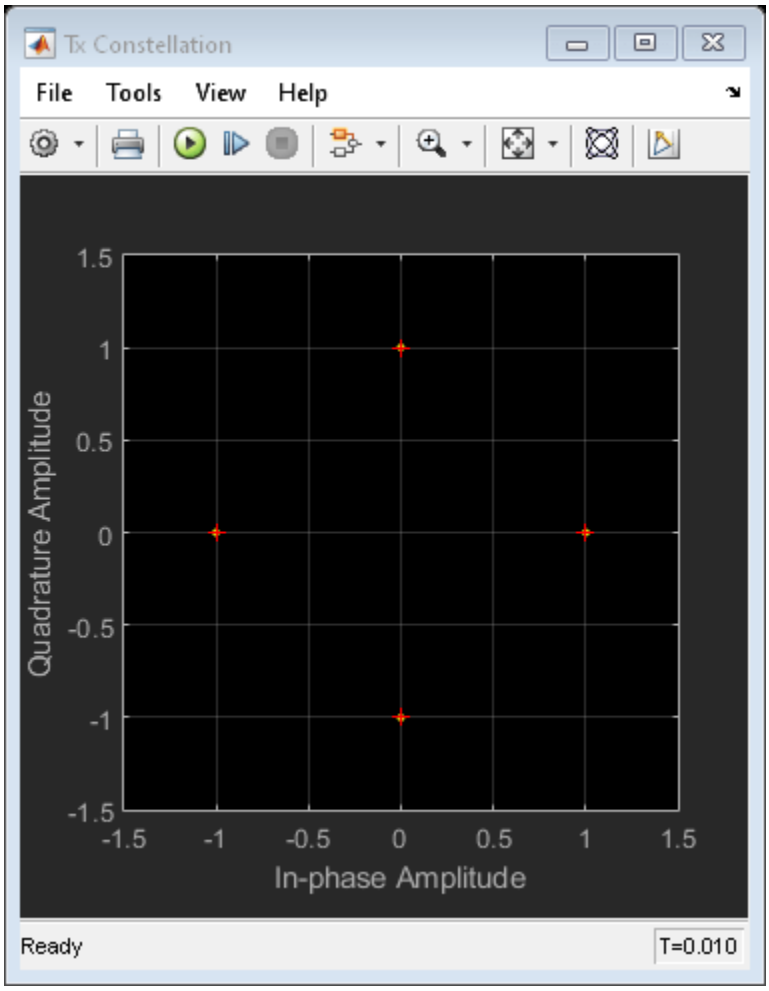
- Develop the baseband transmitter model for waveform generation
- Determine SNR specification to achieve the 1e-4 BER from a link-level idealized baseband model
- Derive RF subsystem specifications from equivalent-baseband model of RF receiver and ADC
- Derive direct conversion specifications from circuit envelope model of RF receiver
- Perform multi-carrier simulation including interfering signals and derive the specifications of the DC offset compensation algorithm

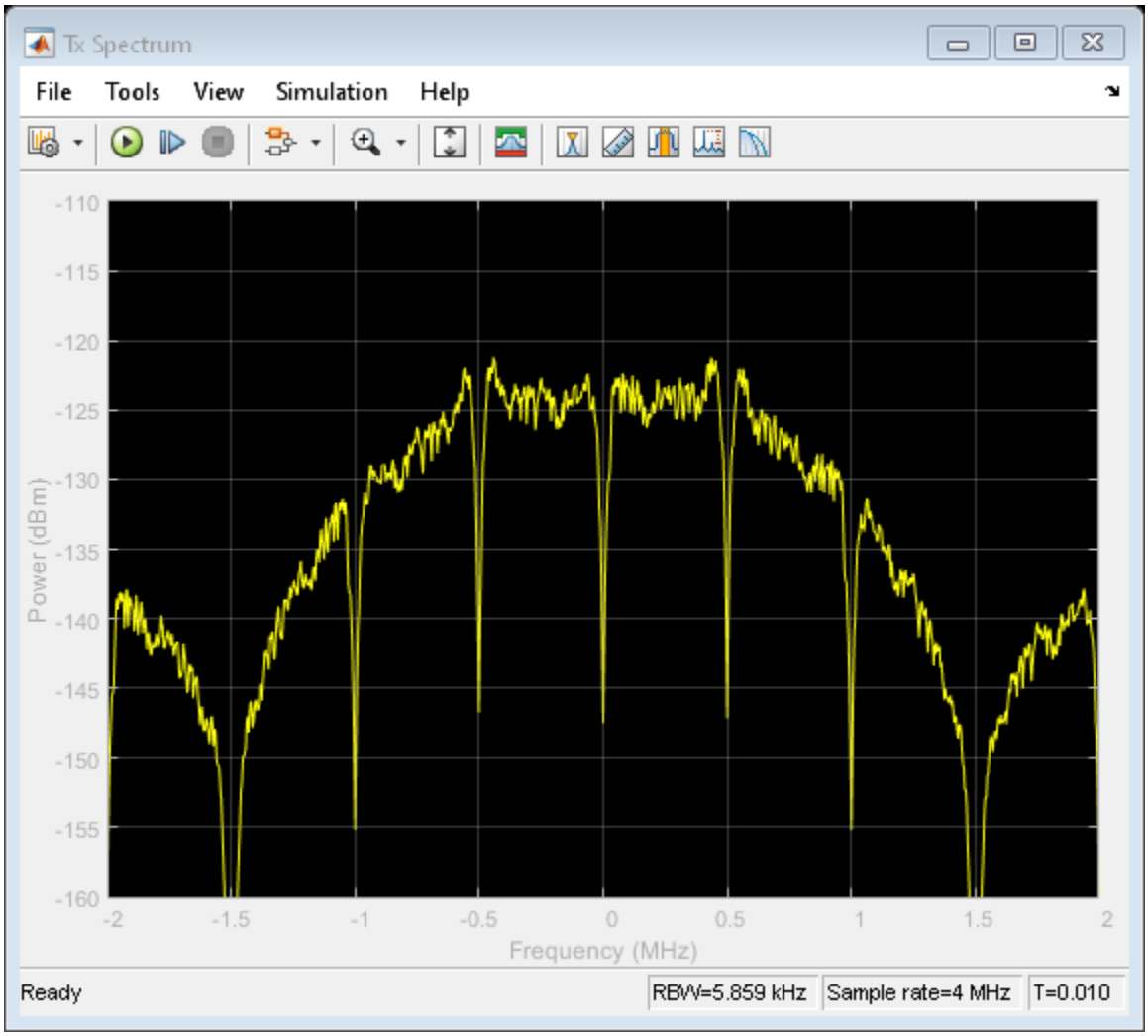
Design and Verify Baseband Transmitter

To evaluate the performance of the RF receiver design, it is necessary and sufficient to use a signal spectrally representative of an 802.15.4 waveform.

The baseband transmitter model creates and illustrates a spectrally representative ZigBee waveform in the spectral and constellation domains. This model and all the subsequent models use callbacks to create MATLAB workspace variables that parameterize the systems.



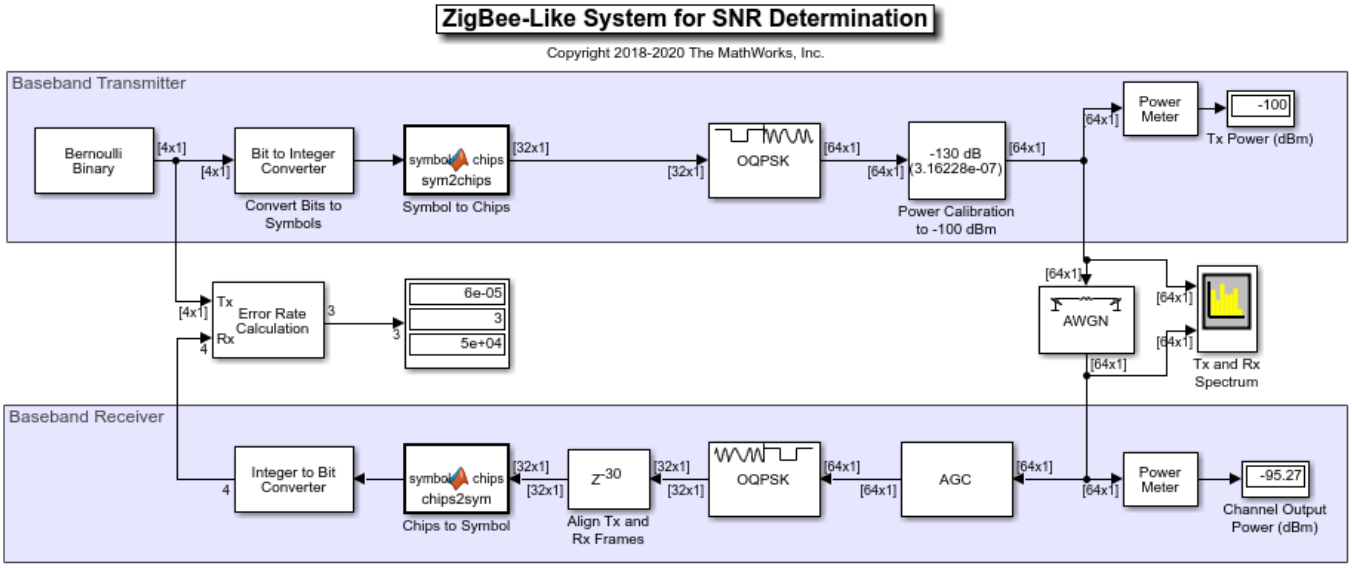

Tx Constellation
File
Tools
View
Help
1.5
1
0.5
0
-0.5
-1
-1.5
Quadrature Amplitude
-1.5
-1
-0.5
0
0.5
1
1.5
In-phase Amplitude
Ready
T=0.010

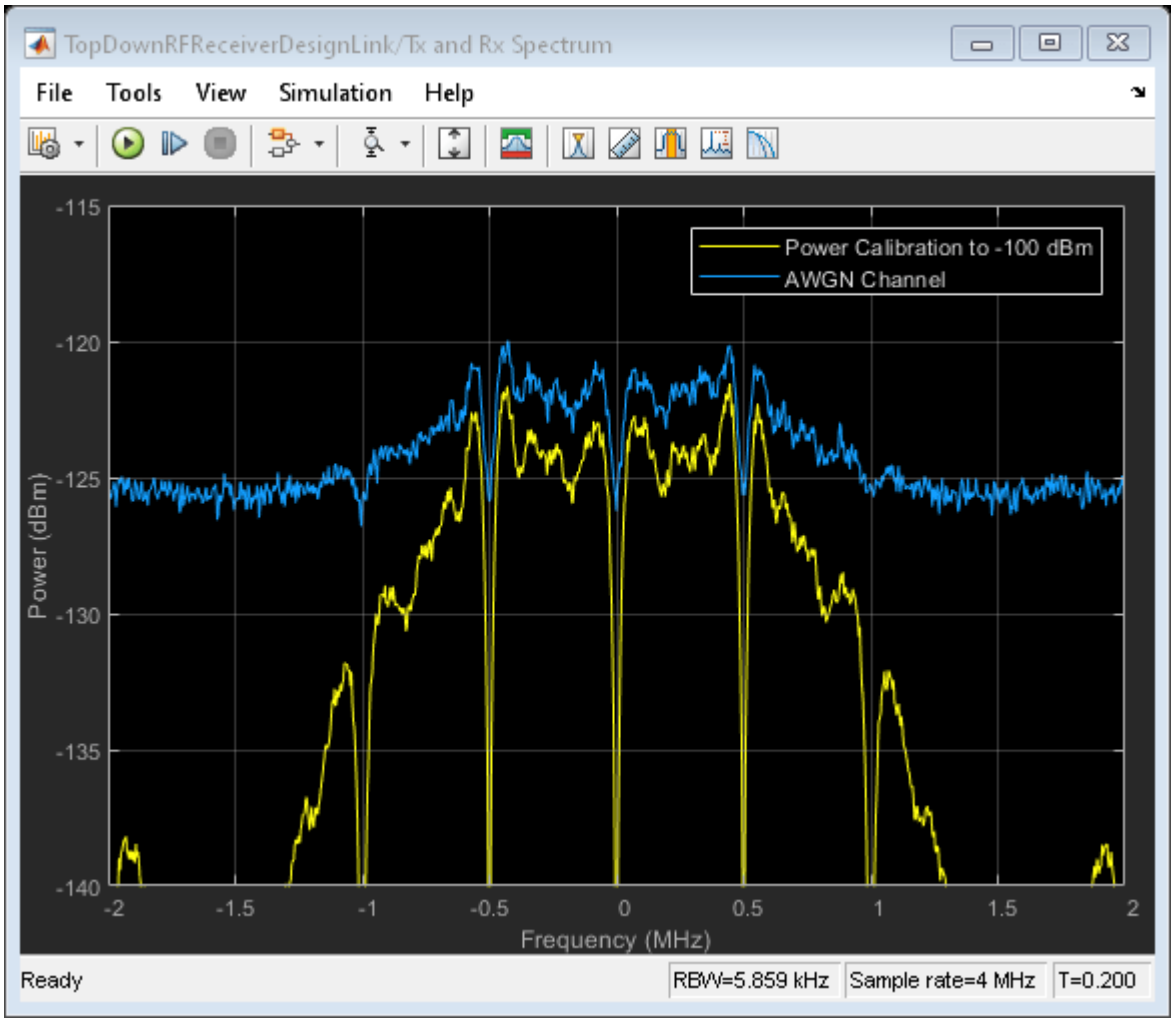


Determine Receiver SNR Requirement

To design the receiver, first determine the SNR needed to achieve the specified BER less than 1e-4. calculated in the simulation bandwidth of 4 MHz. Run the link-level model to simulate the receiver processing required to achieve the target BER.

Computing the BER accurately requires alignment of the transmit and receive signals. The simulation must compensate for a two-sample delay of the received signal compared to the transmitted signal. Also, to ensure correct chip-to-symbol-to-bit mapping, the simulation must align the chips to frame boundaries at the input to the Chips to Symbol block on a frame boundary. Accounting for the receive signal delay and the frame boundary alignment requires addition of a **Delay** block set to a 32-2=30 delay on the receiver branch before recovering the received symbols.


ZigBee-Like System for SNR Determination
Copyright 2018-2020 The MathWorks, Inc.
Baseband Transmitter
Bernoulli Binary
Bit to Integer Converter
Convert Bits to Symbols
sym2chips
Symbol to Chips
OQPSK
-130 dB (3.16228e-07)
Power Calibration to -100 dBm
Power Meter
-100
Tx Power (dBm)
Tx
Rx
Error Rate Calculation
6e-05
3
5e+04
AWGN
Tx and Rx Spectrum
Baseband Receiver
Integer to Bit Converter
chips2sym
Chips to Symbol
Z-30
Align Tx and Rx Frames
OQPSK
AGC
Power Meter
-95.27
Channel Output Power (dBm)


TopDownRFReceiverDesignLink/Tx and Rx Spectrum
File
Tools
View
Simulation
Help
Power Calibration to -100 dBm
AWGN Channel
Power (dBm)
-115
-120
-125
-130
-135
-140
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Frequency (MHz)
Ready
RBW=5.859 kHz
Sample rate=4 MHz
T=0.200

The model achieves a 1e-4 BER at an SNR of -2.7 dB, which can be verified by collecting 100 bit errors.

In the link-level model, the AWGN block accounts for the overall channel and RF receiver SNR budget.

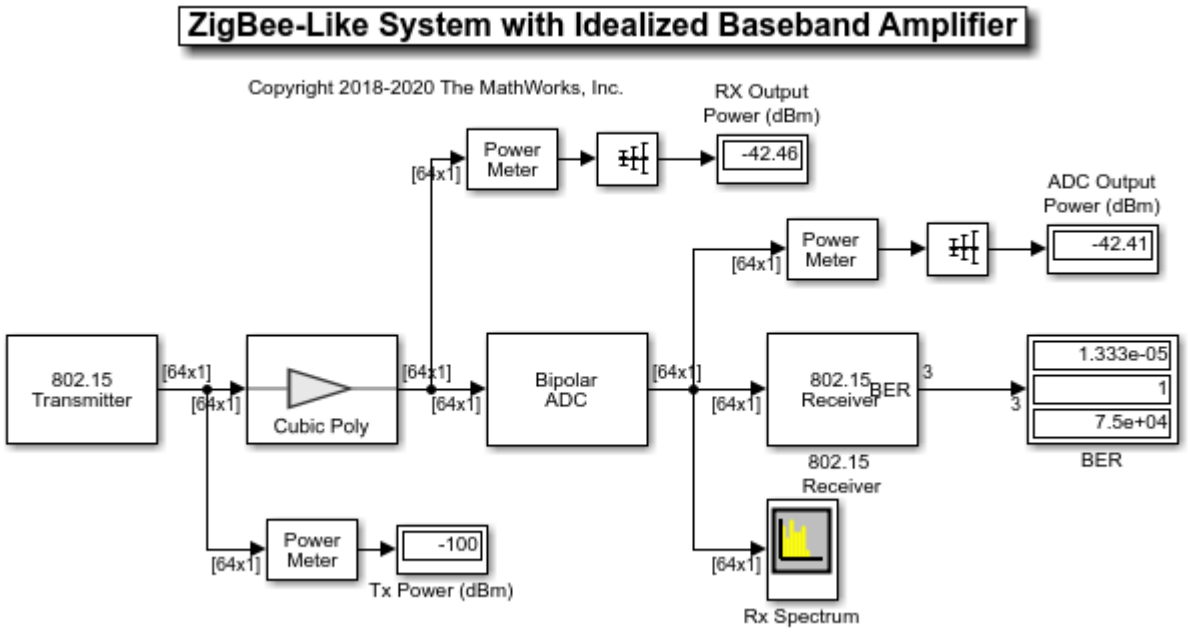
Add ADC and Determine Receiver Total Gain and Noise Figure (NF)

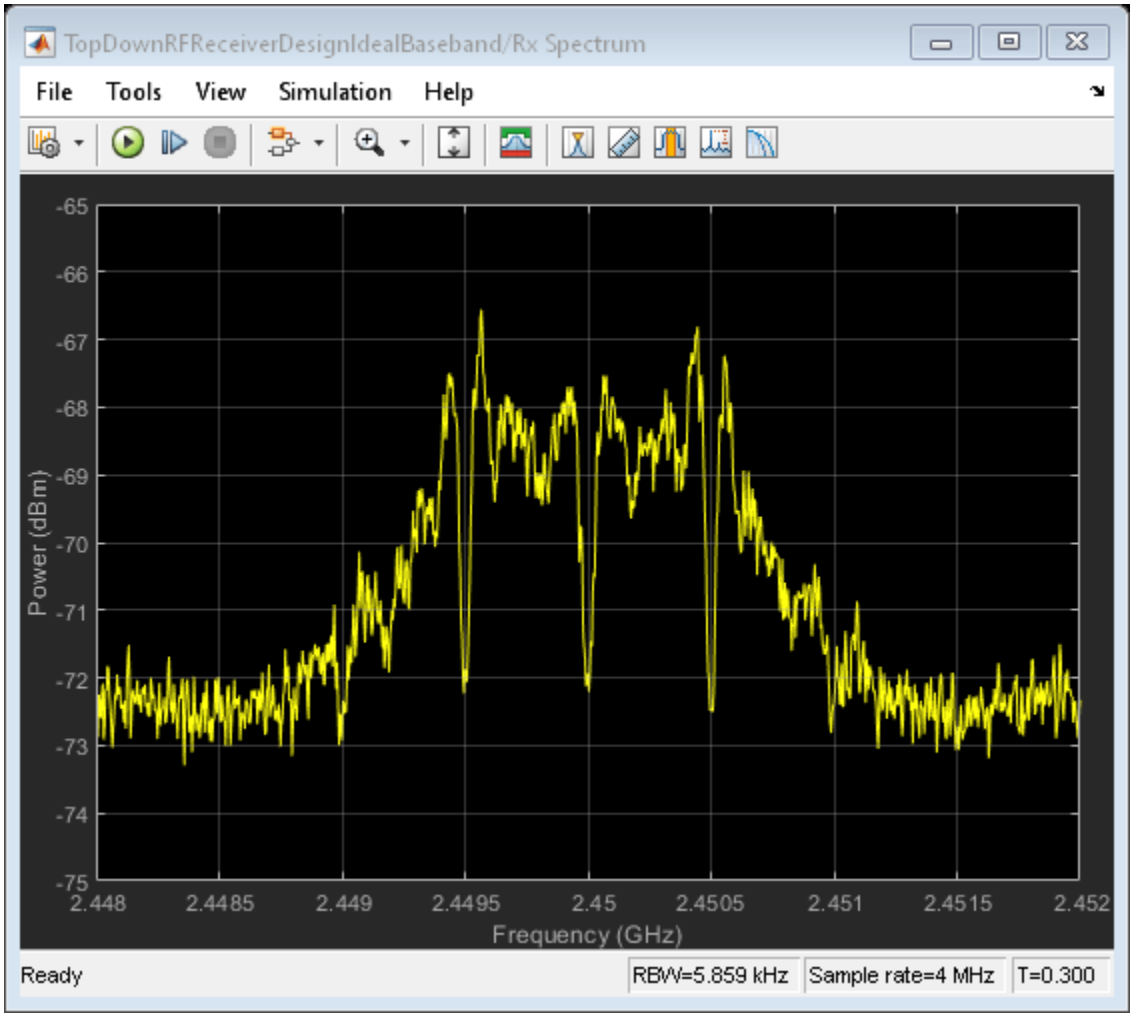
This section uses traditional heuristic derivations to determine the high-level specifications of the RF receiver and ADC.

- B = 4 MHz = simulation bandwidth = simulation sampling frequency
- kT = 174 dBm/Hz = thermal noise floor power
- Sensitivity = -100 dBm = receiver sensitivity
- SNR = -2.7 dB
- Noise power in simulation bandwidth = Pn = sensitivity-SNR = -100 dBm - (-2.7 dB) = -97.3 dBm

Simulating an idealized baseband model of the RF Receiver, verify the preliminary RF receiver specifications (NF = 10.7 dB and receiver gain = 53.4 dB). This can be done by collecting 100 errors.

The spectrum analyzer shows that the received spectrum with the ADC is roughly identical in shape to the spectrum of the previous section, without the ADC.





Refine Architectural Description of RF Receiver

In this section the RF receiver, and its noise figure and gain budget specifications, are modelled by using four discrete subcomponents with these characteristics:

- SAW Filter: Noise Figure = 2.5 dB, Gain = -3 dB
- LNA: Noise Figure = 6 dB, Gain = 22 dB
- Passive Mixer: Noise Figure = 10 dB, Gain = -5 dB
- VGA: Noise Figure = 14 dB, Gain = 40 dB

The SAW filter performance is derived from a Touchstone file that specifies S-parameters characteristics. You can verify the gain by visualizing the S21 parameter in the X-Y plane at the operating frequency of 2.45 GHz. You can verify the noise figure by visualizing the NF parameter in the X-Y plane at the operating frequency of 2.45 GHz. Typically, an LNA with low noise and high gain follows the SAW filter, which greatly reduces the impact of the noise figure of the components after the LNA. Also, the passive mixer is specified with a high IP2. Similar to the SAW filter, you can verify the mixer gain by visualizing the S21 parameter in the X-Y plane over a user-specified frequency range of [2e9 3e9].

An equivalent baseband model simulates the refined RF receiver.

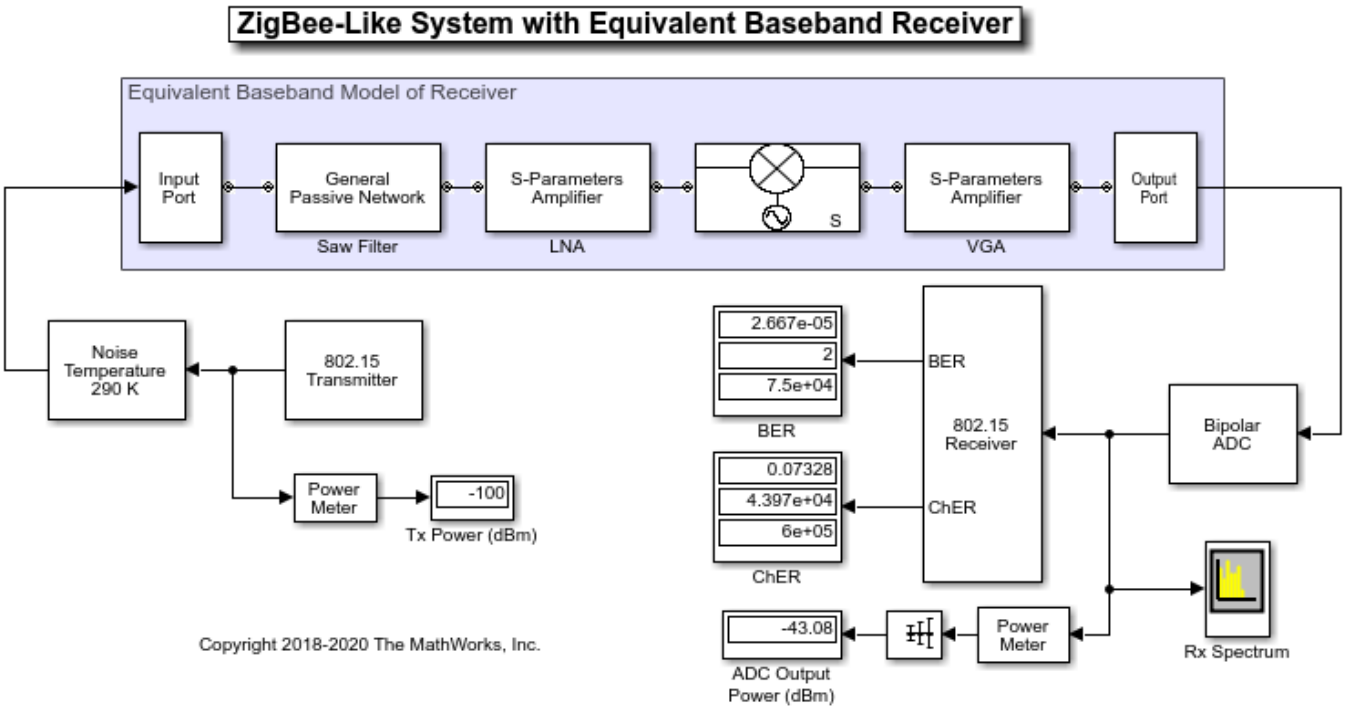
Run the simulation and verify the RF receiver link budget by using the output port visualization pane. The total noise figure and gain across the four stages has been divided according to the following budget:

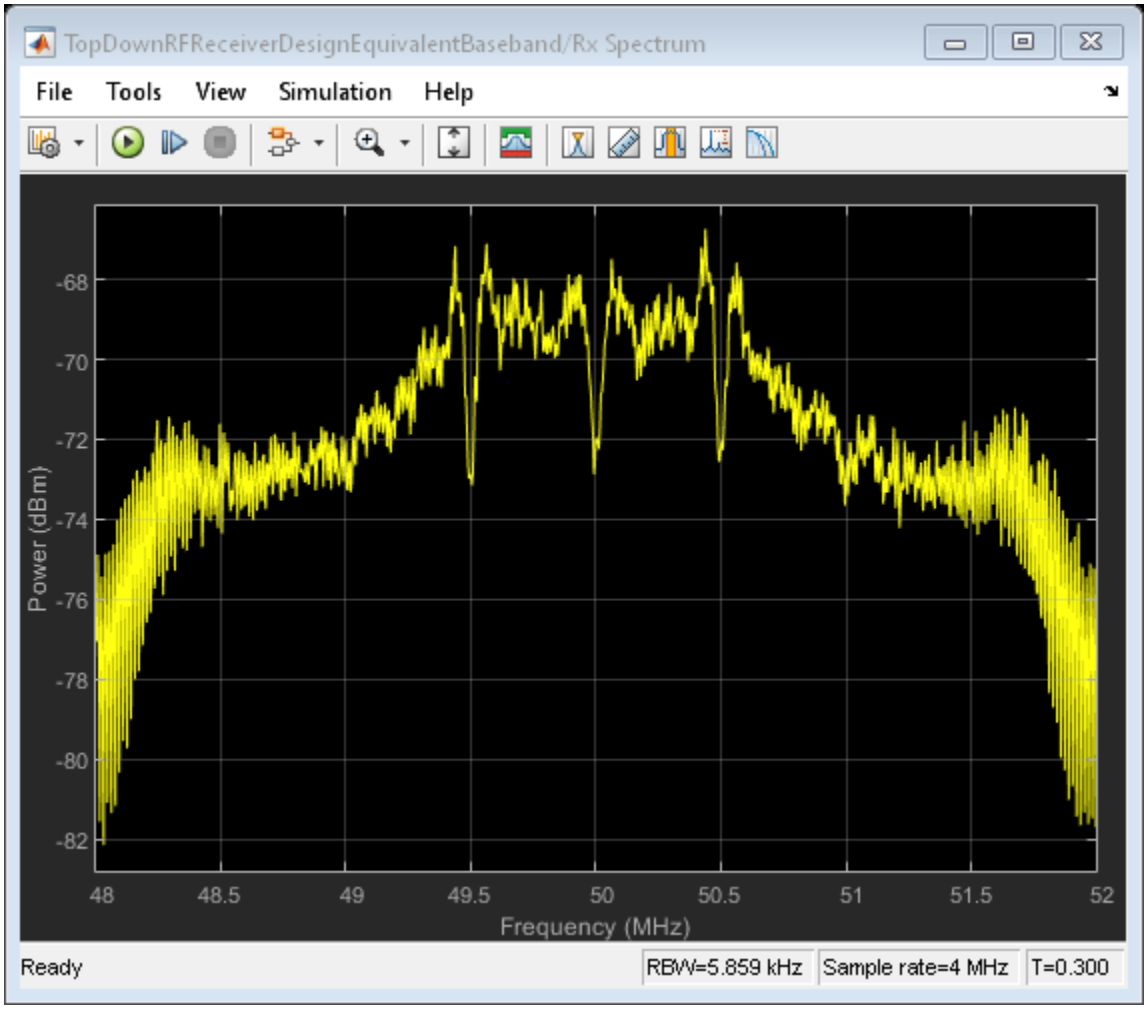
- Component NF (dB) = [2.5, 6, 10, 14]
- Component noise factor F (linear) = 10^(NF/10) = [1.78 3.98 10.0 25.1]
- Power gain (dB) = [-3, 22, -5, 40] = 54 dB > 53.4 dB
- Voltage gain VG (linear) = 10^(Power gain/20) = [0.71 12.59 0.56 100.0]
- System noise factor Fsys (linear) =

$$1 + [F(1) - 1] + \frac{[F(2) - 1]}{VG(1)} + \frac{[F(3) - 1]}{VG(1) \times VG(2)} + \frac{[F(4) - 1]}{VG(1) \times VG(2) \times VG(3)} = 11.8$$

- System noise figure NFsys (dB) = 10*log10(Fsys) = 10.7 dB

With this model you can verify that a BER < 1e-4 corresponds to a Chip Error Rate (ChER) around 7%. By computing ChER, you can run the subsequent models for less time and still collect accurate BER statistics.





Use Circuit Envelope to Simulate Additional RF Impairments

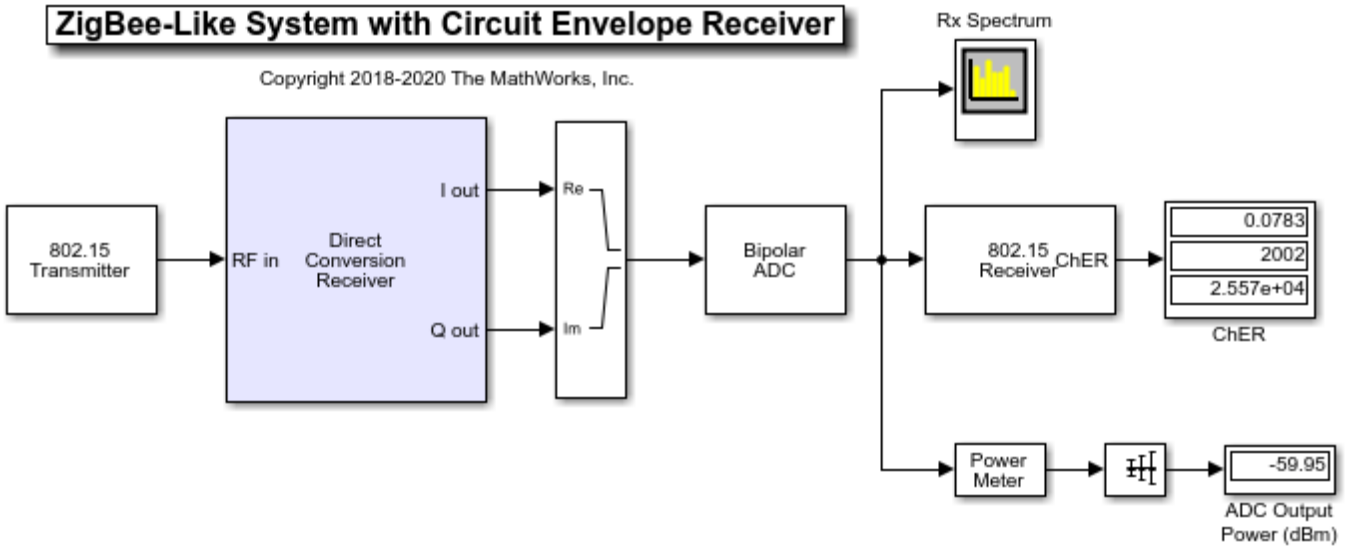
The equivalent baseband modeling technique used in the previous section cannot model a true direct conversion receiver. That model used a mixer with an input frequency of 2.45 GHz and an LO frequency of 2.4 GHz, which led to a spectrum analyzer center frequency of 50 MHz. This modeling limitation motivates a change to the circuit envelope method.

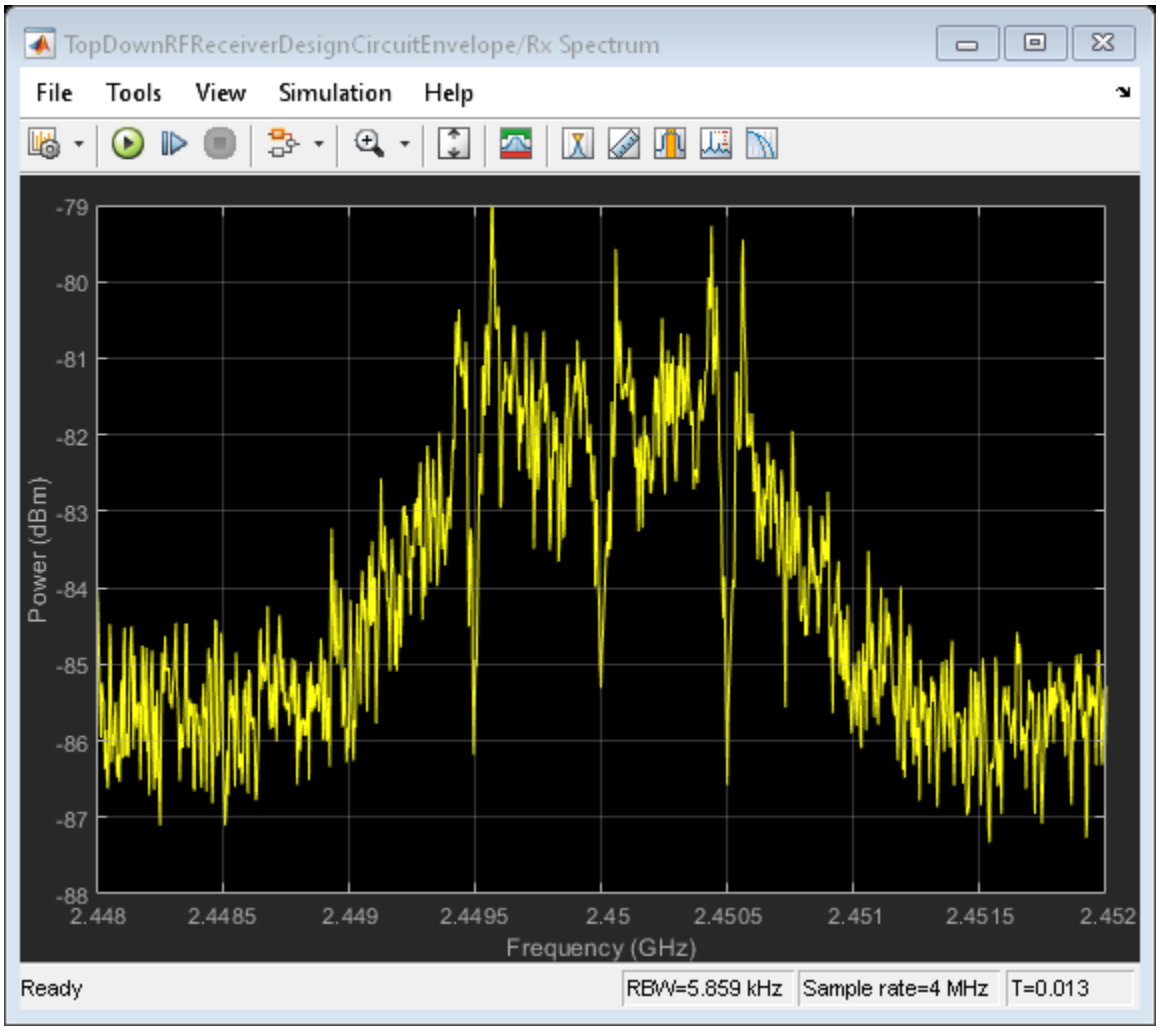
Using the circuit envelope modeling approach, continue refining the RF receiver architecture by adding more realistic impairments.

The circuit envelope model of the RF Receiver differs from the equivalent baseband model as it:

- Replaces the equivalent baseband mixer with a quadrature modulator, consisting of parameterizable I and Q mixers and phase shifter block, and an LO with impairments
- Uses broadband impedances (50 ohm) to explicitly model the power transfer between blocks

Comparing spectra, power measurements, and ChER to the equivalent baseband model, there are no significant performance differences. However, with the circuit envelope model, you can include even order nonlinearity effects, I/Q imbalance, and specifications of colored noise distributions for each of the components.

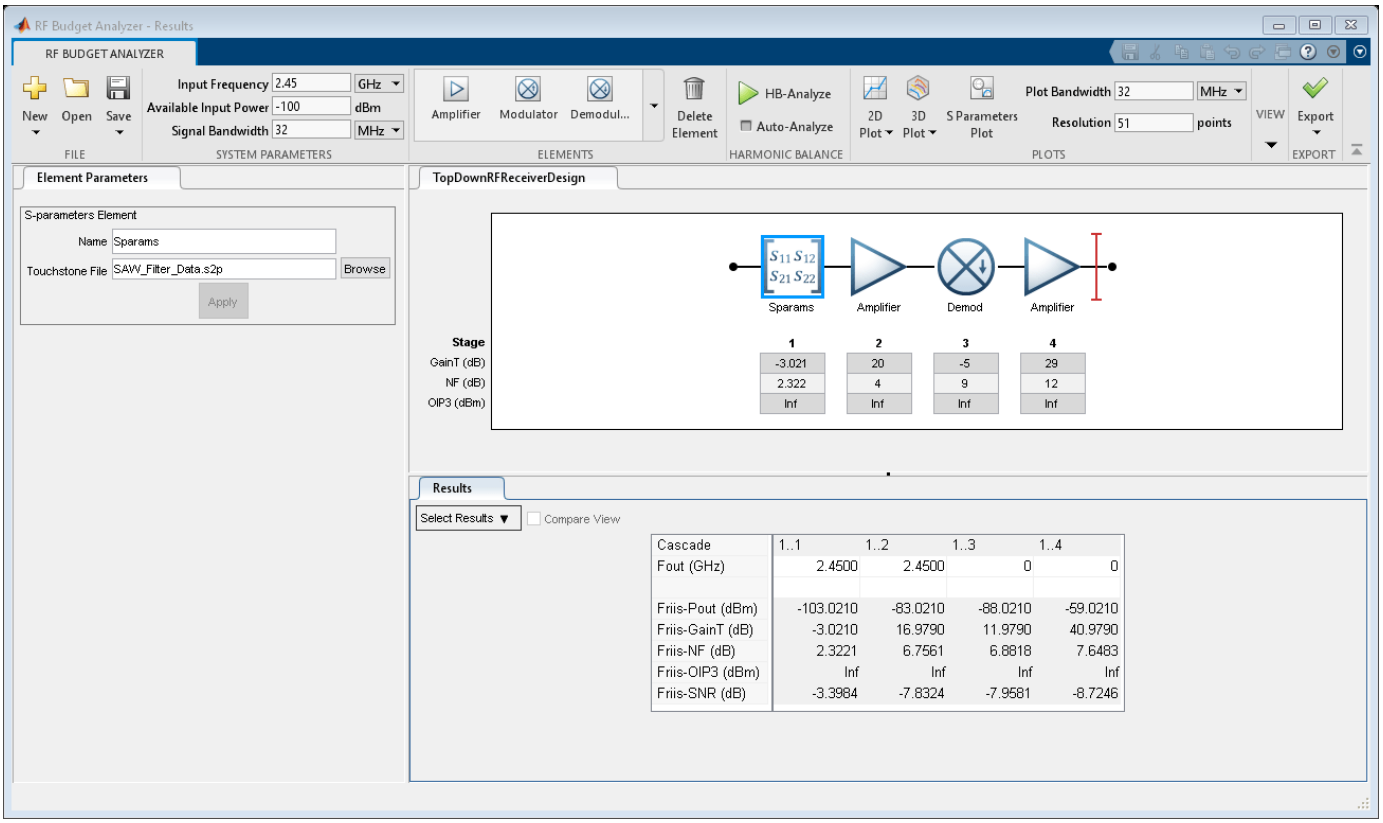

ZigBee-Like System with Circuit Envelope Receiver
Copyright 2018-2020 The MathWorks, Inc.
Rx Spectrum
802.15
Transmitter
RF in
Direct
Conversion
Receiver
I out
Q out
Re
Im
Bipolar
ADC
802.15
Receiver
ChER
0.0783
2002
2.557e+04
ChER
Power
Meter
-59.95
ADC Output
Power (dBm)


TopDownRFReceiverDesignCircuitEnvelope/Rx Spectrum
File
Tools
View
Simulation
Help
Power (dBm)
-79
-80
-81
-82
-83
-84
-85
-86
-87
-88
2.448
2.4485
2.449
2.4495
2.45
2.4505
2.451
2.4515
2.452
Frequency (GHz)
Ready
RBW=5.859 kHz
Sample rate=4 MHz
T=0.013

You can manually build the circuit envelope model of the RF Receiver by using blocks from the Circuit Envelope library, or it can be automatically generated using the RF Budget Analyzer App.

The RF Budget Analyzer App

- Uses Friis equations to determine the noise, gain, and nonlinearity budget of an RF chain
- Allows you to explore the receiver design space and determine how to break down the specifications across the elements of the chain
- Helps you determine which element has the largest contribution to the noise and nonlinearity budget
- Can generate an RF receiver model with which you can perform multi-carrier simulation and further modify.



Add Wideband Interference, LO Leakage, and DC Offset Cancellation

This section modifies the circuit envelope model to create this circuit envelope with interferer model. The circuit envelope with interferer model includes a wideband interfering signal and these impairments:

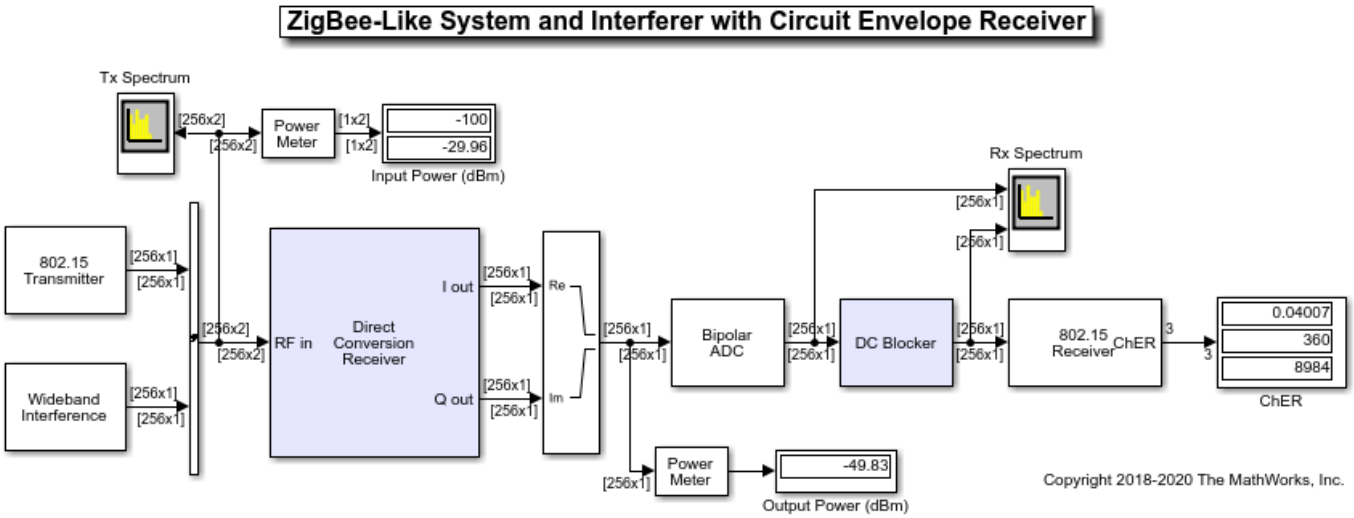
- LO-RF isolation of 90 dB in the quadrature demodulator
- OIP2 equal to 70 dBm in the quadrature demodulator
- WCDMA-like blocker of -30 dBm at 2500 MHz

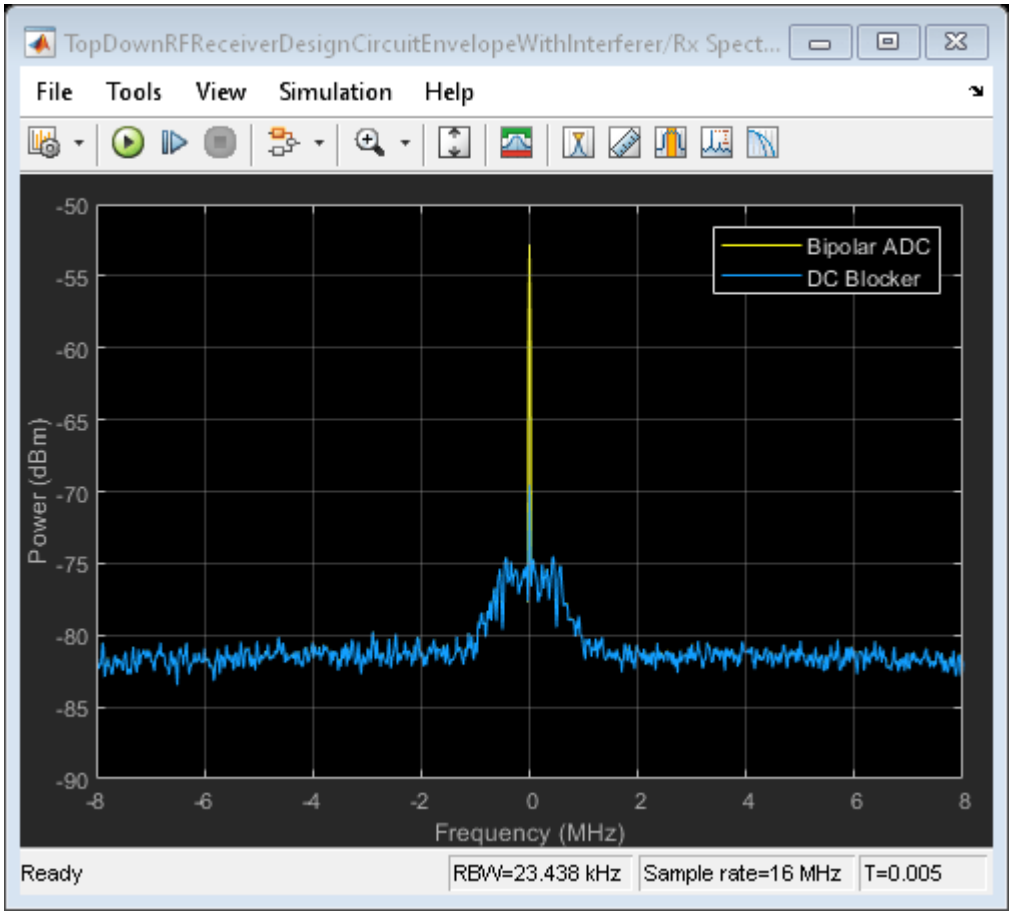
This simulation models a non-standard-compliant interfering signal that has power and spectral distribution characteristics realistic for a WCDMA signal. The simulation of the wideband interfering

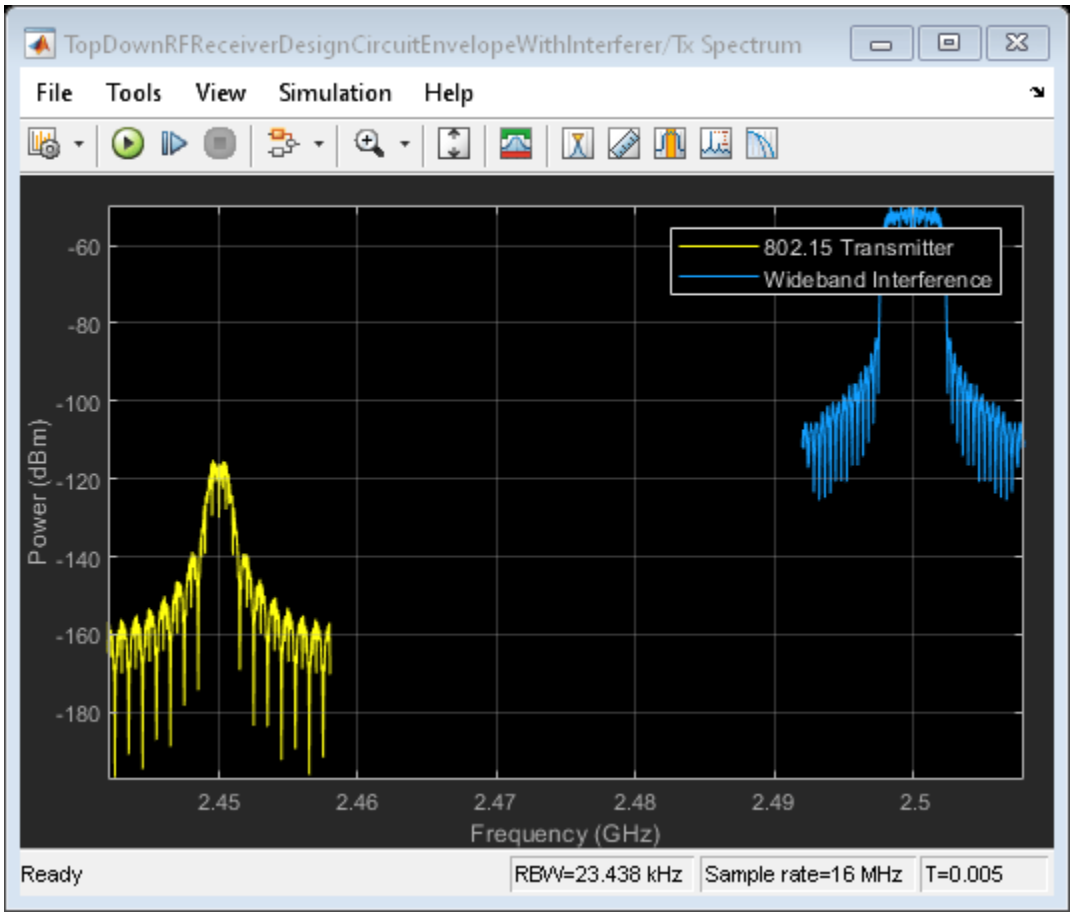
signal requires a larger simulation bandwidth of 16MHz. Therefore the 1 MHz OQPSK signal is oversampled by 16, and the Circuit Envelope simulation bandwidth is also increased to 16 MHz.

The design requires a DC offset compensation algorithm to achieve the desired ChER due to the DC offset that results from the LO leakage and the nonlinearity in the demodulator caused by the high out-of-band interfering signal power. In this case you include a very selective filter, that introduces a long latency with corresponding computation delay increases in the ChER measurement block.

The spectrum centered at 0 Hz shows the DC offset compensation reducing the DC offset. As you run the model, note that the DC offset is eventually completely removed.

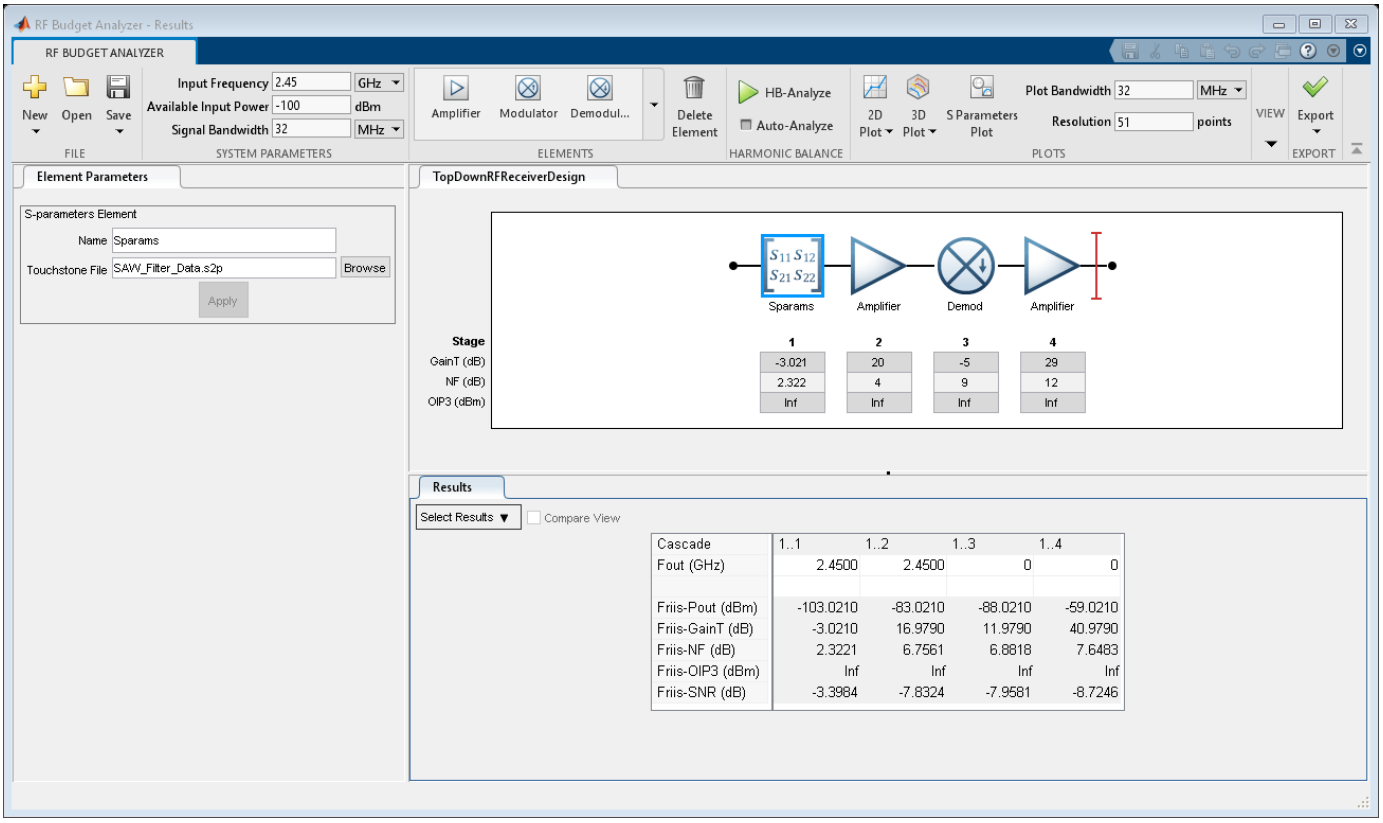



TopDownRFReceiverDesignCircuitEnvelopeWithInterferer/Rx Spect...
File
Tools
View
Simulation
Help
Bipolar ADC
DC Blocker
Power (dBm)
-50
-55
-60
-65
-70
-75
-80
-85
-90
-8
-6
-4
-2
0
2
4
6
8
Frequency (MHz)
Ready
RBW=23.438 kHz
Sample rate=16 MHz
T=0.005



Conclusion

Following a top-down design methodology, RF receiver components specifications were derived. Impairment, interferer, and RF receiver subcomponent models were iteratively refined to increase fidelity and validated at each stage to confirm overall system performance goals were achieved.


RF Budget Analyzer - Results
RF BUDGET ANALYZER
New
Open
Save
FILE
Input Frequency 2.45 GHz
Available Input Power -100 dBm
Signal Bandwidth 32 MHz
SYSTEM PARAMETERS
Amplifier
Modulator
Demodul...
Delete Element
ELEMENTS
HB-Analyze
Auto-Analyze
HARMONIC BALANCE
2D Plot
3D Plot
S Parameters Plot
Plot Bandwidth 32 MHz
Resolution 51 points
PLOTS
VIEW
Export
EXPORT
Element Parameters
S-parameters Element
Name Sparams
Touchstone File SAW_Filter_Data.s2p
Browse
Apply
TopDownRFReceiverDesign
Sparams
Amplifier
Demod
Amplifier
Stage
GainT (dB)
NF (dB)
OIP3 (dBm)
1 2 3 4
-3.021 20 -5 29
2.322 4 9 12
Inf Inf Inf Inf
Results
Select Results
Compare View
Cascade 1..1 1..2 1..3 1..4
Fout (GHz) 2.4500 2.4500 0 0
Friis-Pout (dBm) -103.0210 -83.0210 -88.0210 -59.0210
Friis-GainT (dB) -3.0210 16.9790 11.9790 40.9790
Friis-NF (dB) 2.3221 6.7561 6.8818 7.6483
Friis-OIP3 (dBm) Inf Inf Inf Inf
Friis-SNR (dB) -3.3984 -7.8324 -7.9581 -8.7246

Digital Predistortion to Compensate for Power Amplifier Nonlinearities

This example shows how to use digital predistortion (DPD) in a transmitter to offset the effects of nonlinearities in a power amplifier. We use models of a power amplifier that were obtained using the “Power Amplifier Characterization” on page 1-4 example to simulate two cases. In the first simulation, the RF transmitter sends two tones. In the second simulation, the RF transmitter sends a 5G-like OFDM waveform with 100 MHz bandwidth.

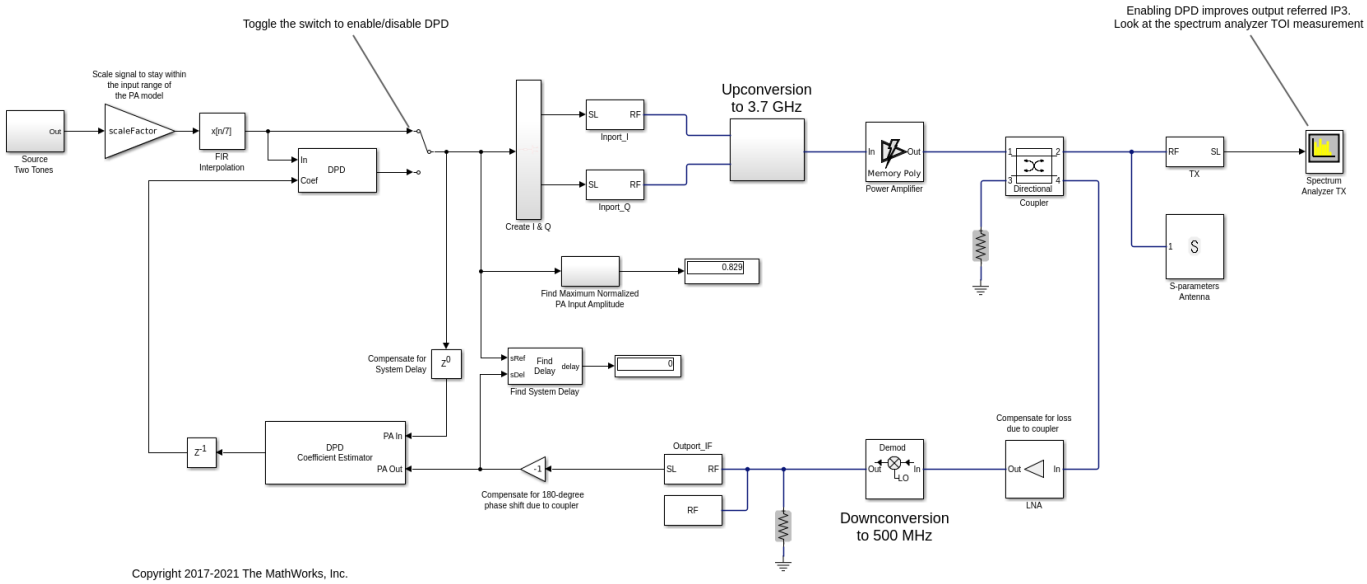
DPD with Two Sinusoidal Test Signals

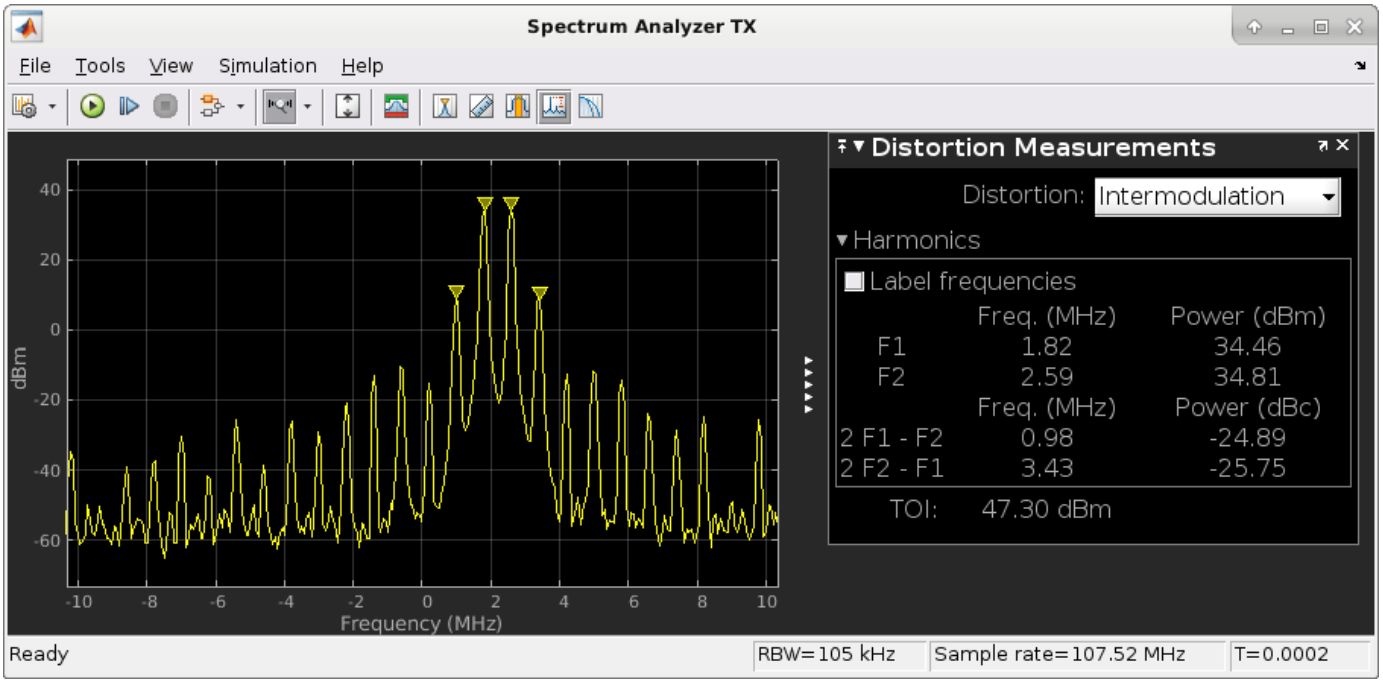
Open the Simulink RF Blockset model: System-level model PA + DPD with two tones.

The model includes a two-tone signal generator that is used for testing the output-referred third-order intercept point of the system. The model includes upconversion to RF frequency using an I-Q modulator, the PA model, a coupler to sniff the output of the PA, and an S-parameter block representing the antenna loading effect. The receiver chain performs downconversion to low intermediate frequency. Notice that the simulation bandwidth of this system is 107.52 MHz.

The model can be simulated without DPD when the toggle switch is in the up position.

```
model = 'simrfV2_powamp_dpd';
open_system(model)
sim(model)
```





The manual switch is toggled to enable the DPD algorithm. When toggled, the TOI (third-order intercept point) is improved significantly. Inspect the distortion measurement in the Spectrum Analyzer to validate these results and see how the power of the harmonics is reduced thanks to the DPD linearization.

Before the two-tone signal enters the DPD block or the power amplifier, it goes through an FIR interpolator, the same FIR interpolator used during PA characterization. This is necessary because the power amplifier model was obtained for the sample rate after interpolation, not the original sample rate of the two-tone signal, and oversampling the signal is required for modeling high order nonlinearities introduced by the power amplifier.

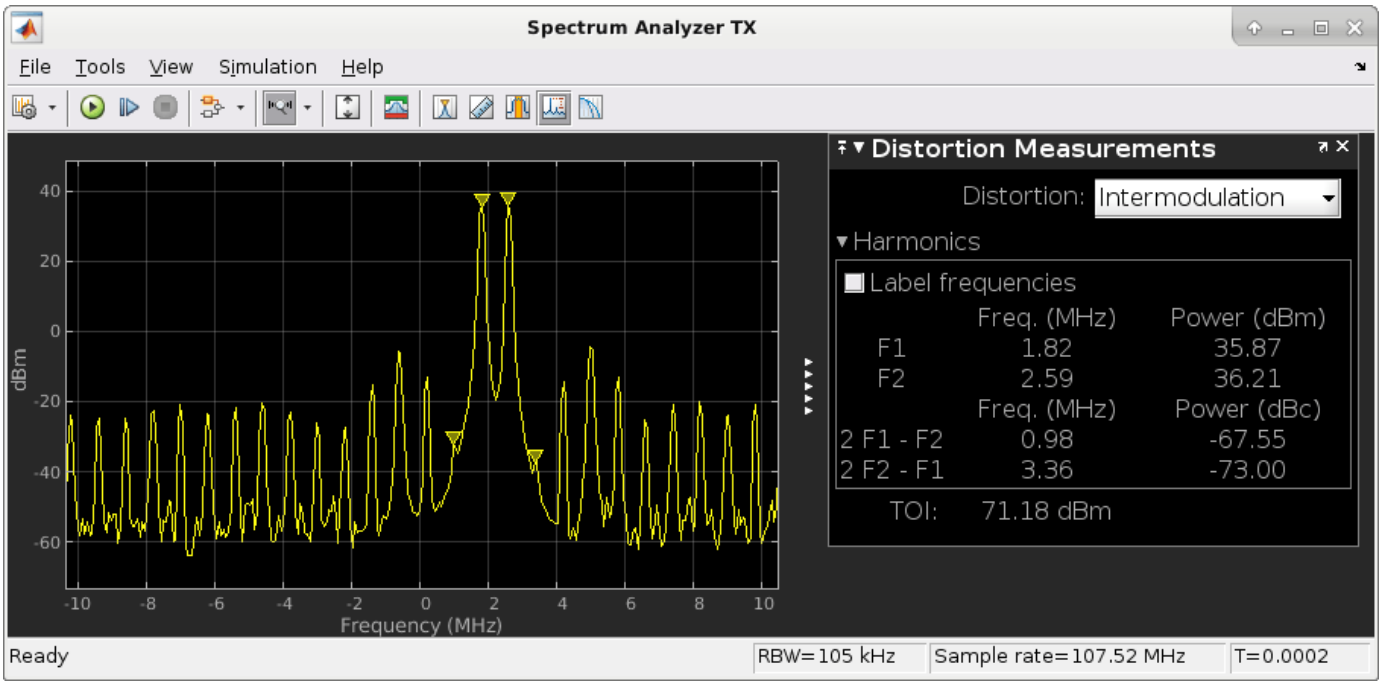
The desired amplitude gain of the DPD Coefficient Estimator is set based on the expected gain of the power amplifier (obtained during PA characterization), because in addition to linearization, the overall goal is to make the combined gain from the DPD input to the power amplifier output as close to the expected gain as possible. To estimate the DPD coefficients correctly, the input signals to the DPD Coefficient Estimator block, PA In and PA Out, must be aligned in the time domain. This is verified by the Find Delay block which shows that the delay introduced by the RF system is 0. Moreover, PA In and PA Out must be accurate baseband representations of the power amplifier input signal and output signal, i.e. no extra gain or phase shift. Otherwise, the DPD Coefficient Estimator block would not observe the power amplifier correctly and would not produce the right DPD coefficients. This is done by ensuring that both the upconversion and downconversion steps have a gain of 1 and the loss and phase shift due to the coupler are properly compensated for before the feedback signal reaches PA Out.

The purpose of the scale factor in front of the FIR interpolator is to help utilize the linearized power amplifier effectively. Even with DPD enabled, two undesirable scenarios may occur. The two-tone signal may be very small with respect to the input range of the linearized system, hence under-utilizing the amplification capability of the linearized system. Or the two-tone signal may be so large that the power amplifier model operates outside the range observed during PA characterization and therefore the power amplifier model may not be an accurate model of the physical device. We use the following heuristic approach to set the scale factor.

Assuming that the DPD block perfectly linearizes the power amplifier to achieve the expected amplitude gain, then the maximum input amplitude allowed by the DPD block should be the maximum power amplifier output amplitude observed during PA characterization divided by the expected amplitude gain. The scale factor before the DPD block should then be the maximum input amplitude allowed by the DPD block divided by the maximum amplitude of the interpolated signal observed during PA characterization.

The system model has a block that calculates the maximum normalized PA input amplitude. If it is equal to 1, it means that the baseband signal entering the RF system has a maximum amplitude equal to the maximum PA input amplitude observed during PA characterization. Therefore, if the maximum normalized PA input amplitude is smaller than 1, the scale factor set by the heuristic approach above may be increased. If the maximum normalized PA input amplitude is greater than 1, the scale factor should be reduced.

```
set_param([model '/Manual Switch'], 'action', '1')
sim(model)
```



By changing the degree and the memory depth defined in the DPD Coefficient Estimator block, you can find the most suitable tradeoff between performance and implementation cost.

```
close_system(model,0)
close all; clear
```

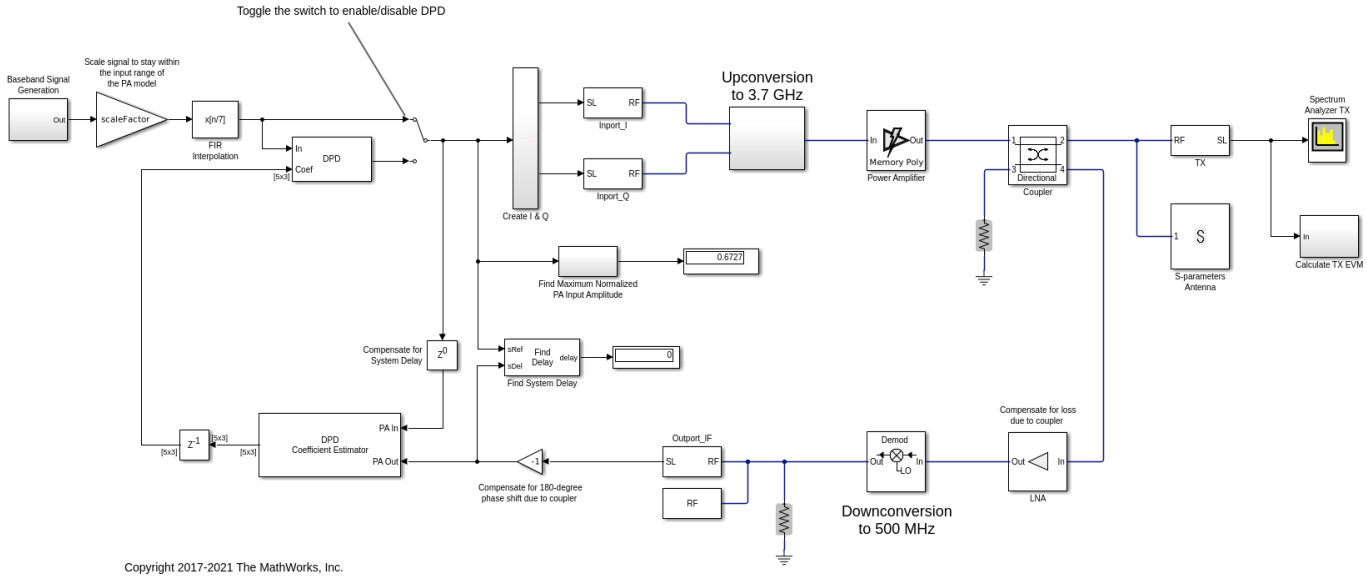
DPD with a 5G-like OFDM Waveform

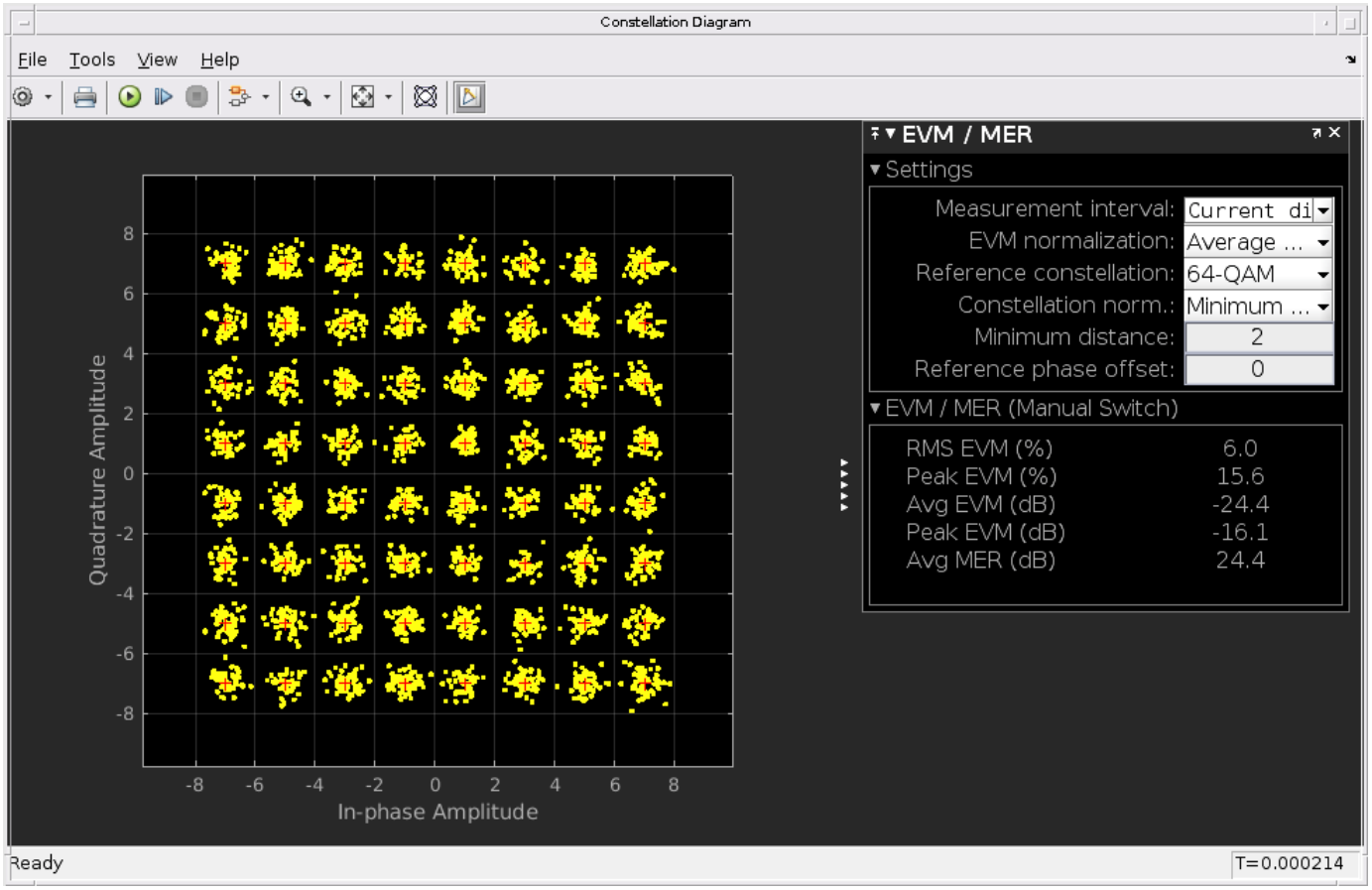
Open the Simulink RF Blockset model: System-level model PA + DPD with a 5G-like OFDM waveform.

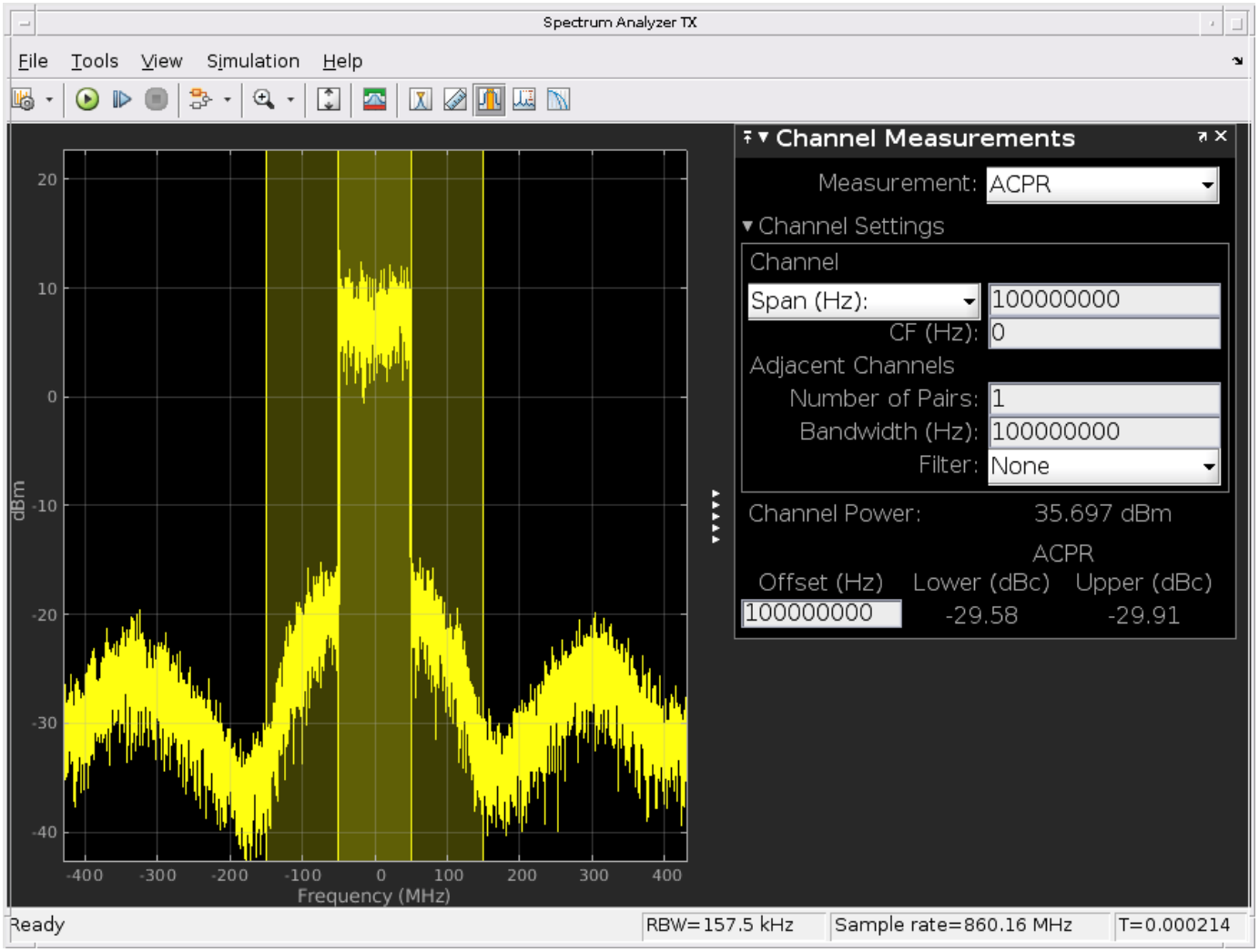
The structure of this Simulink model is the same as that of the previous Simulink model. The signal being amplified is now a 5G-like OFDM waveform, rather than a two-tone signal. The spectrum analyzer measures ACPR instead of TOI and we add a subsystem to measure the EVM and MER of the amplified OFDM waveform.

Without DPD linearization, the system achieves an average Modulation Error Ratio of 24.4 dB, as seen from the constellation plot measurement.

```
model = 'simrfV2_powamp_dpd_comms';
open_system(model)
sim(model)
```

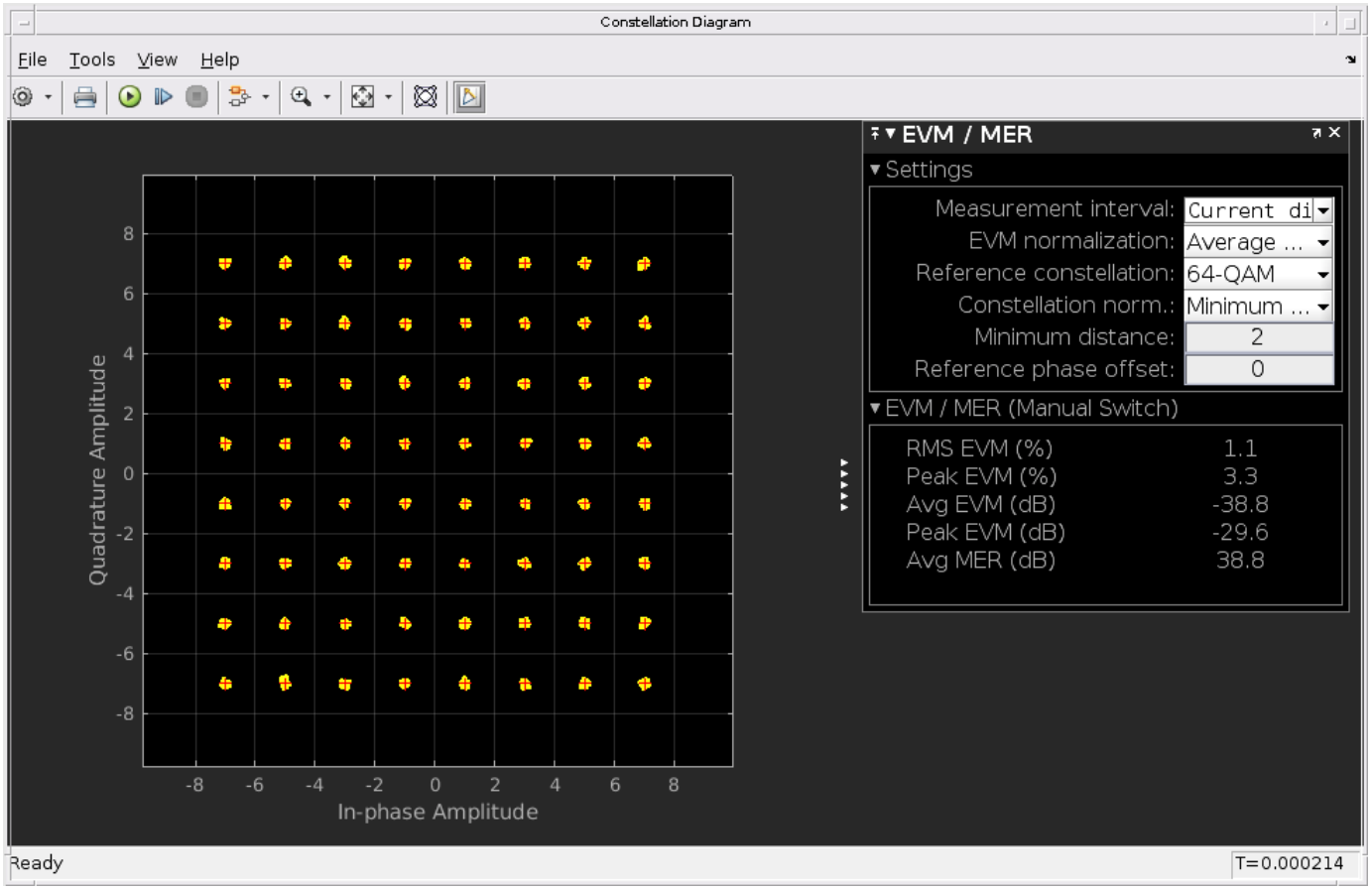


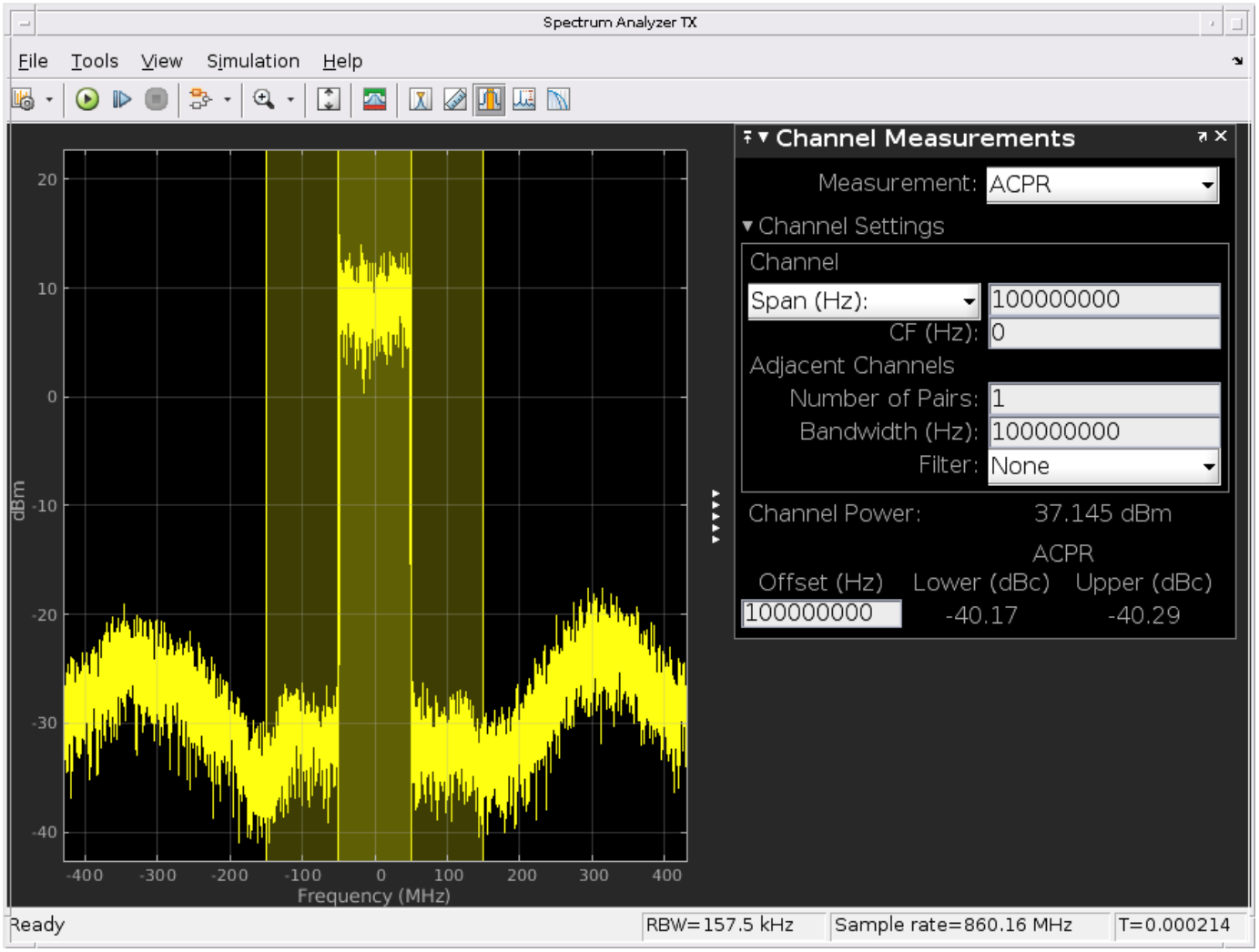

Constellation Diagram
File Tools View Help
EVM / MER
Settings
Measurement interval: Current di
EVM normalization: Average ...
Reference constellation: 64-QAM
Constellation norm.: Minimum ...
Minimum distance: 2
Reference phase offset: 0
EVM / MER (Manual Switch)
RMS EVM (%) 6.0
Peak EVM (%) 15.6
Avg EVM (dB) -24.4
Peak EVM (dB) -16.1
Avg MER (dB) 24.4
Quadrature Amplitude
In-phase Amplitude
8
6
4
2
0
-2
-4
-6
-8
Ready
T=0.000214



The manual switch is toggled to enable the DPD algorithm. When toggled, the average MER is improved significantly.

```
set_param([model '/Manual Switch'], 'action', '1')
sim(model)
```


Constellation Diagram
File Tools View Help
EVM / MER
Settings
Measurement interval: Current di
EVM normalization: Average ...
Reference constellation: 64-QAM
Constellation norm.: Minimum ...
Minimum distance: 2
Reference phase offset: 0
EVM / MER (Manual Switch)
RMS EVM (%) 1.1
Peak EVM (%) 3.3
Avg EVM (dB) -38.8
Peak EVM (dB) -29.6
Avg MER (dB) 38.8
Quadrature Amplitude
In-phase Amplitude
8
6
4
2
0
-2
-4
-6
-8
Ready
T=0.000214




```
close_system(model,0)
close all; clear
```

Selected Bibliography

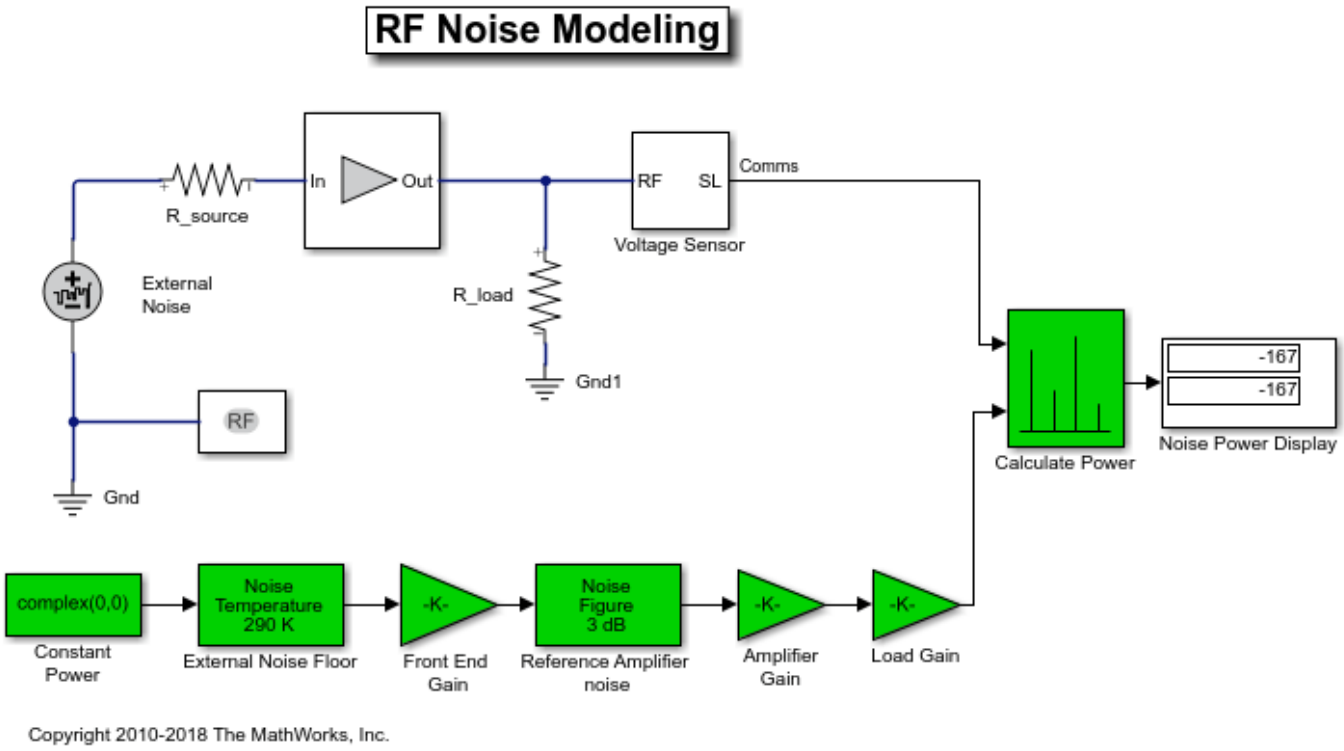
1 Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A Generalized Memory Polynomial Model for Digital Predistortion of Power Amplifiers." *IEEE® Transactions on Signal Processing*. Vol. 54, No. 10, October 2006, pp. 3852–3860.

2 Gan, Li, and Emad Abd-Elrady. "Digital Predistortion of Memory Polynomial Systems Using Direct and Indirect Learning Architectures." In *Proceedings of the Eleventh IASTED International Conference on Signal and Image Processing (SIP)* (F. Cruz-Roldán and N. B. Smith, eds.), No. 654-802. Calgary, AB: ACTA Press, 2009.

RF Noise Modeling

This example shows how to use the RF Blockset™ Circuit Envelope library to simulate noise and calculate noise power. Results are compared against theoretical calculations and a Communications Toolbox™ reference model.

The model is shown below.



System Architecture

The RF system, shown in white, consists of:

- A `Configuration` block, which sets global simulation parameters for the RF Blockset system. With the **Simulate Noise** option checked, noise is included in the simulation.
- An `External Noise` source with a power spectral density of $4kT_sR$ applied at the input. In this equation, k is the Boltzmann constant, T_s is the temperature of the source, and R is the noise reference impedance. The calculated noise level of -174 dBm/Hz is used in this example. The `External Noise` source is an explicit signal.
- An `Amplifier` block with a specified power gain and noise figure.
- A `Voltage Sensor` (i.e. `Outport`) block, with the **Source type** parameter set to **Voltage**.
- Source and load resistors.

The Communications Toolbox reference system, shown in green, consists of:

- `Gain` blocks that model amplifier gain and loading effects.
- Two `Receiver Thermal Noise` blocks that model the external noise and the amplifier noise, respectively.

The `Calculate Power` block computes RMS noise power. Note that the Communications Toolbox signal is referenced to 1 ohm, while RF Blockset power is computed for the actual load R_load .

The example model defines variables for block parameters using a callback function. To access model callbacks, select **MODELING > Model Settings > Model Properties** and click the Callbacks tab in the Model Properties window.

Running the Example

1. Type `open_system('RFNoiseExample')` at the Command Window prompt.
2. Select **Simulation** > **Run**.

The `Noise Power Display` block verifies that the RF Blockset and Communications Toolbox noise models are equivalent.

Computing RF System Noise

To enable noise in the RF Blockset circuit envelope environment:

- In the `Configuration` block dialog, select **Simulate noise**.
- Specify a **Temperature**. RF Blockset uses this value to calculate the equivalent noise temperature inside the amplifier.
- Specify the **Noise figure (dB)** parameter of any amplifiers or mixers in the system.

In the example, for a specified LNA gain of 4 dB and noise figure of 3 dB, the output noise is calculated using the following equations:

$$G_1 = 2.5119 \text{ (4 dB)}$$

$$F_1 = 1.9953 \text{ (3 dB)}$$

The next equation converts the noise factor to an equivalent noise temperature. T is the **Temperature** parameter of the RF Blockset `Configuration` block.

$$T_e = (F_1 - 1) * T = 288.63$$

The final equation calculates the output noise power. T_s is the temperature of the SimRF™ `External Noise` block and the Communications Toolbox `External Noise Floor` block.

$$N_{out,sys} = 10 \log_{10} \left(k(T_s + T_e)G_1 \right) + 30 = -166.97 \text{ dBm/Hz}$$

The available noise power is the power that can be supplied by a resistive source when it is feeding a noiseless resistive load equal to the source resistance. The green `External Noise Floor` block generates an available power referenced to 50 ohms.

The `Front End Gain` block models the voltage divider due to the source resistance and the input impedance of the amplifier.

The green `Reference Amplifier Noise` and `Amplifier Gain` blocks model the noise added by the amplifier and the amplifier gain, respectively.

The output of the Communications Toolbox `Amplifier Gain` block is equal to the voltage across the RF Blockset `R_load` block.

Impact of Thermal Noise on Communication System Performance

This example shows how to use the RF Blockset™ Circuit Envelope library to model thermal noise in a super-heterodyne RF receiver and measure its effects on a communications system noise figure (NF) and bit error rate (BER). A Communications Toolbox™ reference model with parameters computed using Friis equations and a RF Blockset Noise Testbench are used to verify the results.

RF Receiver System Architecture

The Modulator and Channel subsystems consist of Communications Toolbox blocks that model:

- A QPSK-modulated waveform of random bits
- A raised cosine pulse-shaping filter for spectral limiting
- free-space path loss

The RF receiver subsystem, shown in light purple, consists of RF Blockset blocks:

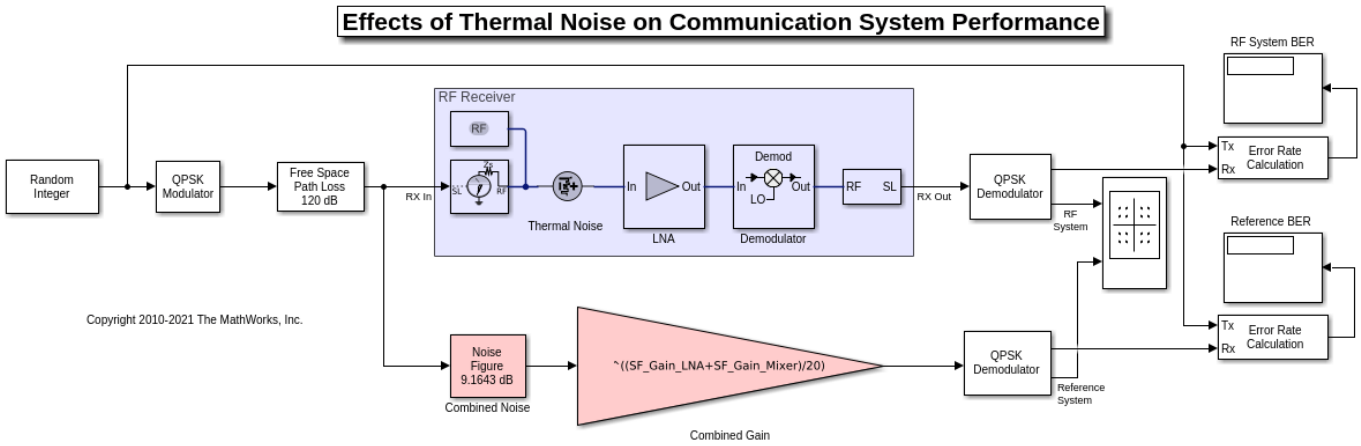
- An Inport block converts the complex input waveform to available power in the RF system with reference impedance equal to the **Source impedance** and assigns the input modulation waveform to a `2.1` GHz RF carrier.
- A noise source to set the RF system noise floor for all simulation carrier frequencies. The block performs this action when **White** is selected for the mask Noise distribution option. To set the Noise power spectral density level, a value of `4*K*T*50` is used (`K` is Boltzmann's constant, `T` is set to a room temperature of `290` kelvin, and `50` ohms is the system reference impedance).
- Cascaded RF amplifier and RF demodulator blocks with specified noise figure and gain. These blocks only enable noise impairments. The Demodulator block's image reject filter is enabled using a mask checkbox and defines with other mask parameters a bandpass filter whose edges are `2.0` and `2.2` GHz. This filter prevents the down-conversion of thermal noise centered around 2.6 GHz or folding of other carrier frequencies with noise into the intermediate frequency (IF) defined as the absolute difference of the RF and LO frequencies. If the image rejection filter is removed, the noise contribution on the IF increases above the estimation provided by Friis equations and the BER will deteriorate.
- An Outport block, with the parameter **Sensor type** is set to `Power` , **Carrier frequencies** set to the IF frequency, and **Output** parameter is set to `Complex baseband` . These block settings enable the RF system to supply a complex baseband communication signal to the ensuing Communication Toolbox system blocks.
- A Configuration block to set model conditions for simulation. Since the model's RF Blockset section has only included noise impairments, an accurate simulation can be achieved by setting the Configuration block **Fundamental tones** to the Inport Carrier (RF), `5e8` Hz and Demodulator Local oscillator (LO), `1.6e9` Hz frequencies and the **Harmonic order** `1` . Use the Configuration blocks **View** button to explore simulation carrier frequencies.
- All blocks in the RF receiver are matched to 50 ohms. To understand the effects of impedance mismatch on noise simulation see, `“RF Noise Modeling” (RF Blockset)`.

The reference system, shown in red, consists of:

- A Communications Toolbox Receiver Thermal Noise block that includes both the thermal noise floor along with the amplifier and demodulator block noise. The Friis Equation is used to correctly combined noise contributed by the amplifier and demodulator blocks. You can find the calculation in the model's pre-load callback function.

- A Simulink Gain block that models the combined gain of the RF receiver.
- Baseband filters and demodulators process the received signal.

Circuit Envelope Simulation of RF Receiver



Select **Simulation** > **Run** .

Error Rate Calculation blocks compute the BER for the system and reference. To observe the BER as it approaches steady state, increase the total simulation time. For this example, the steady-state bit error rate is approximately `1e-4` .

Computing RF Receiver Noise Figure and Gain

To model noise and gain in the RF Blockset circuit envelope environment:

- In the Configuration block dialog, select **Simulate noise** .
- Specify the **Noise figure (dB)** parameter of RF Amplifier and RF Mixer blocks in your system. The following specifications for the RF receiver in this example produce a combined noise figure of `9.16` dB (as per the Friis Equation): LNA gain of `20` dB, LNA noise figure of `9` dB, Demodulator gain of `-5` dB and RF Demodulator noise figure of `15` dB.

$$G_1 = 100 \text{ (20 dB)}$$

$$G_2 = .316 \text{ (-5 dB)}$$

$$F_1 = 7.94 \text{ (9 dB)}$$

$$F_2 = 31.62 \text{ (15 dB)}$$

$$F_{sys} = F_1 + \frac{F_2 - 1}{G_1} = 8.25$$

$$NF_{sys} = 10 \log_{10} F_{sys} = 9.16 \text{ dB}$$

$$G_{sys} = G_1 \text{ dB} + G_2 \text{ dB} = 15 \text{ dB}$$

RF Blockset Noise Figure Testbench

The RF Blockset Noise Figure Testbench simplifies the measurement of system noise figure. To setup a noise figure test system, insert an RF Noise Figure Testbench in a new model. Copy the settings found in the Model Properties Callbacks `PreLoadFcn` to the new models Model Properties Callback `InitFcn` .

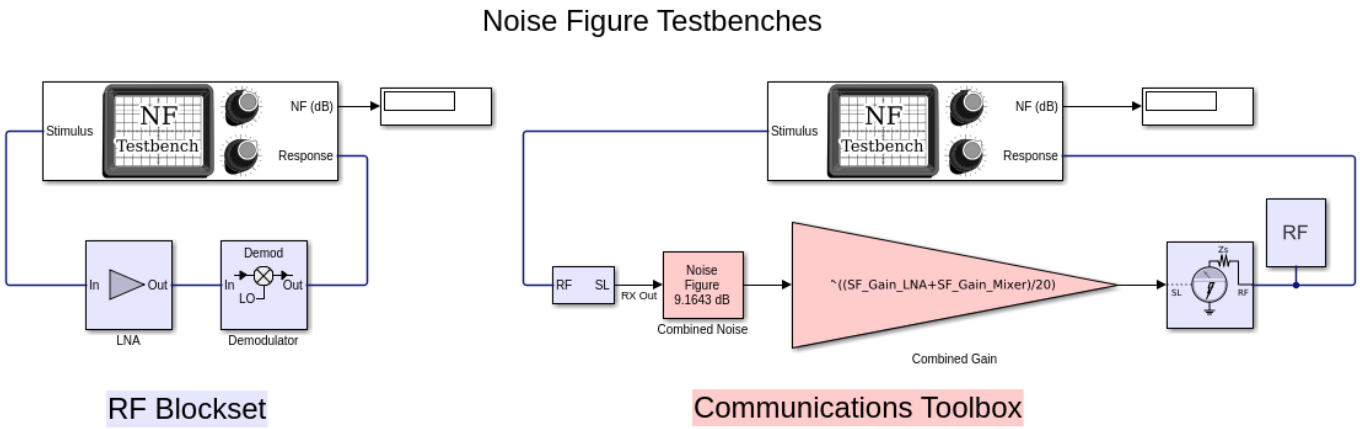
For the system composed of RF Blockset blocks in the above model, copy the LNA and Demodulator blocks with previously set parameters to the new model. The Testbench includes a Noise source that sets the noise floor.

- Connect the Stimulus terminal of the testbench to the In terminal of the LNA and the Out terminal of the Demodulator to testbench Response terminal. A Display block can be connected to the testbench NF terminal to display the measured Noise figure.
- Set the Testbench mask parameters. The RF **Input frequency (Hz)** is `2.1` GHz and the IF **Output frequency (Hz)** is `.5` GHz as in the previous example. A `10e6` Hz **Baseband bandwidth (Hz)** was chosen for this example. The mask instructions provide additional information for configuring the testbench.

For the Communications Friis system in the above model, copy the Combined Noise and Gain blocks with previously set parameters to the new model. The Combined Noise block's **Add 290K antenna noise** checkbox needs to be deselected since the Testbench includes a Noise source that sets the noise floor.

- Three RF Blockset blocks are included: an Outport, an Inport and a Configuration since the testbench expects RF Blockset blocks at its connection points. The type setting for the Inport and Outport blocks is Power. Since the Communication branch is agnostic to carrier frequencies, these blocks Carrier frequencies and Fundamental tones need to be the same and are set to `2.1` GHz. The **Output** parameter of the Outport is `Complex Baseband`. For accuracy, the configuration block **Step size** needs an Envelope bandwidth (**Step size** of `1/80e6` s) at least 8 times larger than the 10 MHz Baseband bandwidth of the testbench.

Run Noise Figure Testbench



Select **Simulation** > **Run** .

Exploring Example

You can include additional RF model impairments using RF block mask selections: Impedance mismatch, nonlinearities or LO isolation.

See Also

Amplifier

Related Examples

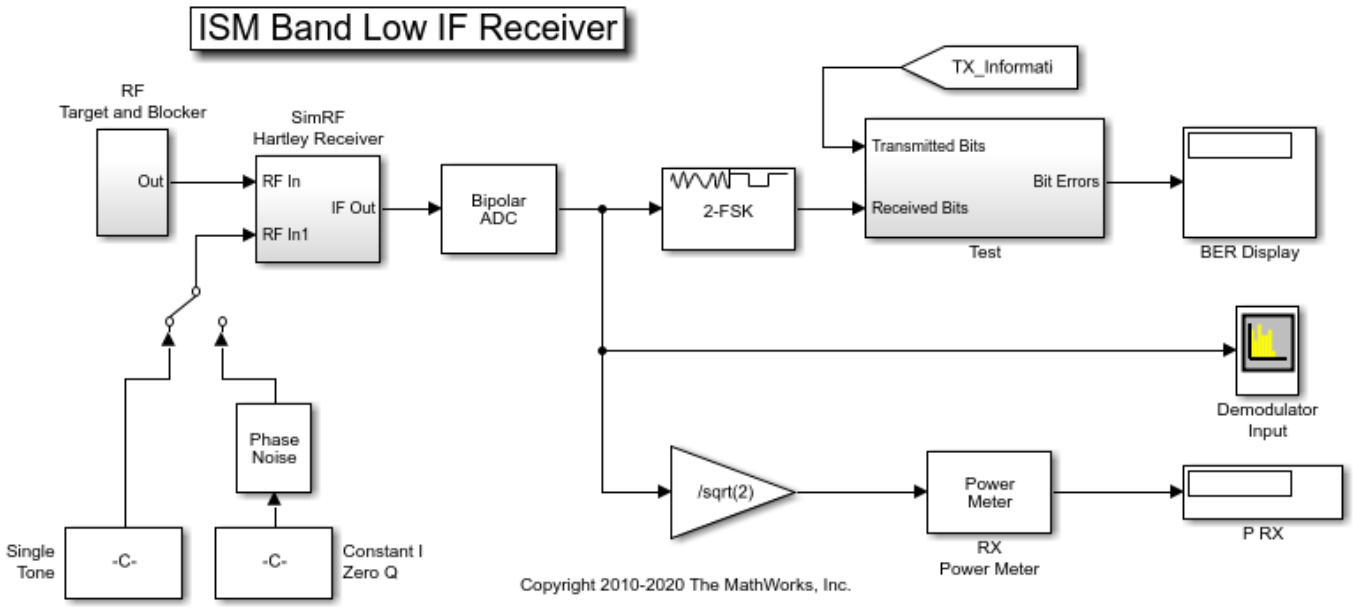
- “Impact of RF Effects on Communication System Performance” on page 8-32

Architectural Design of a Low IF Receiver System

This example shows how to use the RF Blockset™ Circuit Envelope library to simulate the performance of a Low IF architecture with the following RF impairments:

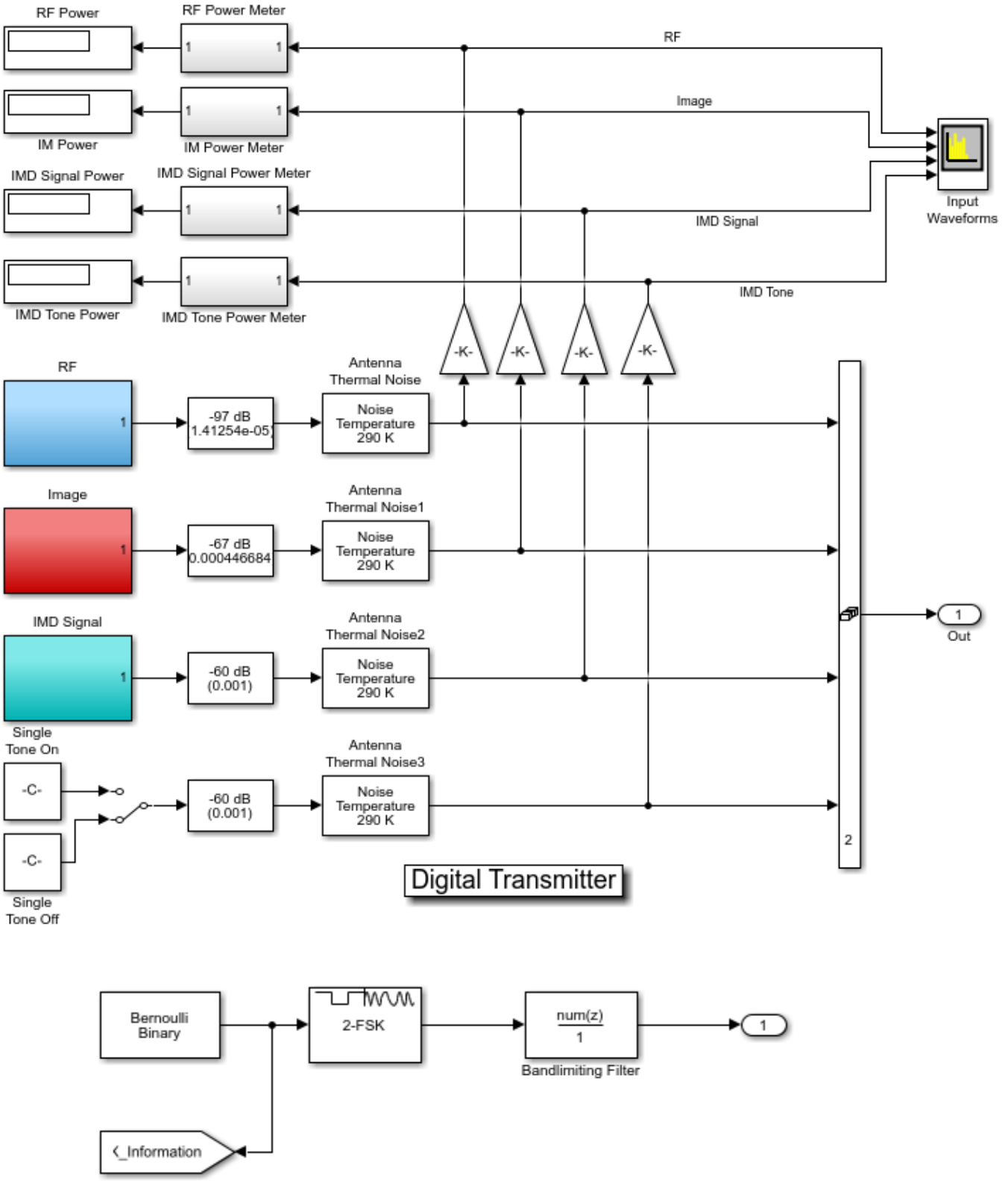
- Component noise
- Interference from blocker signals
- LO phase noise
- Analog-to-digital converter (ADC) dynamic range
- Component mismatch

Design variables in the RF portion of the model include explicit specification of gain, noise figure, IP3, input/output impedance, LO phase offset, and LO phase noise. Carrier frequencies for waveforms entering RF Blockset subsystems are specified in the Inport blocks. Design variables for the transmitter side of the RF interface include carrier frequency, modulation scheme, signal power, and blocker power level. Baseband design variables are number of bits and full scale range of the ADC.

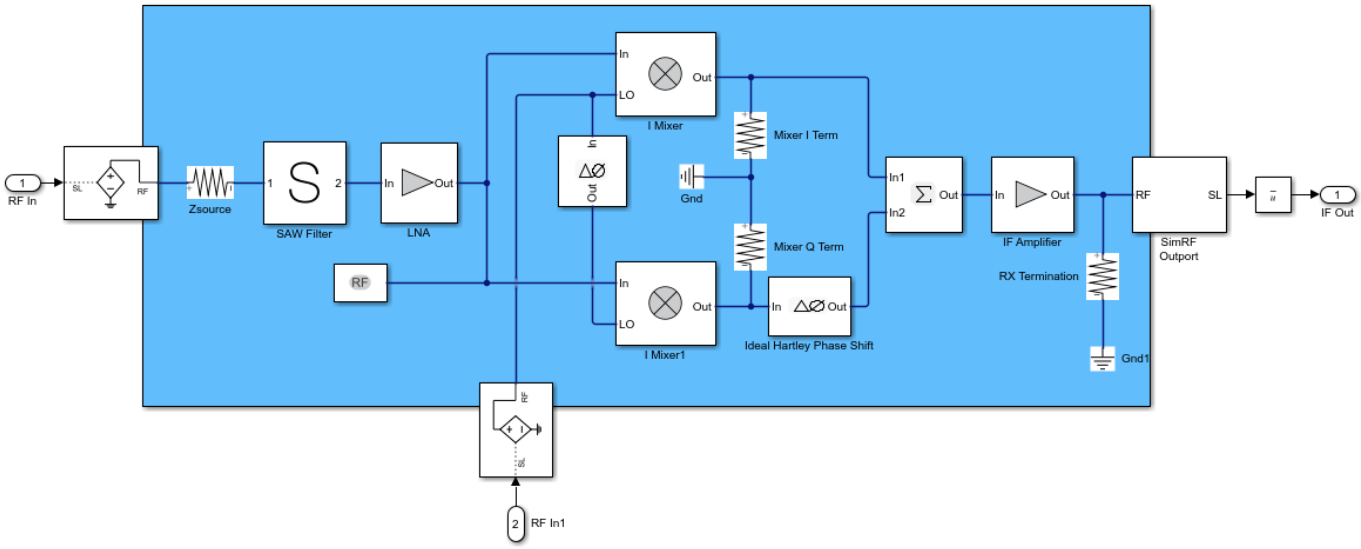


System Architecture:

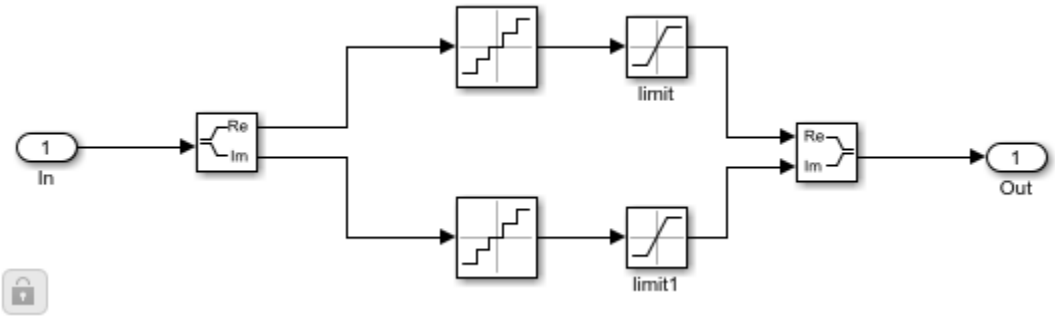
This model illustrates the design and simulation of an ISM Band Receiver. Primary subsystems include a digital transmitter, an RF receiver, an ADC, a phase noise block for noisy LO modeling, and a digital receiver. The remaining blocks are used for analysis.


RF Power
RF Power Meter
RF
IM Power
IM Power Meter
Image
IMD Signal Power
IMD Signal Power Meter
IMD Signal
Input Waveforms
IMD Tone Power
IMD Tone Power Meter
IMD Tone
-K-
RF
Antenna Thermal Noise
-97 dB 1.41254e-05
Noise Temperature 290 K
Image
Antenna Thermal Noise1
-67 dB 0.000446684
IMD Signal
Antenna Thermal Noise2
-60 dB (0.001)
Single Tone On
-C-
Antenna Thermal Noise3
Single Tone Off
Out
Digital Transmitter
Bernoulli Binary
2-FSK
num(z)
1
Bandlimiting Filter
Information

The digital transmitter consists of three FSK modulated waveforms and a high power tone. The three FSK waveform generators use a bandlimiting filter that suppresses the FSK sidebands below the expected thermal noise level. The target waveform at 2450 MHz has a 1 ohm referenced passband power level of approximately -70 dBm. Similarly defined image and intermodulation distortion (IMD) blocker waveforms have passband powers of approximately -40 dBm and -33 dBm, respectively. The IMD tone that couples with the IMD blocker to generate in-band IM3 products has a passband power of -33 dBm. Since the baseband processing defines the complex envelope waveforms, computing passband power requires the insertion of 1/sqrt(2) gain as shown in the design. An IF of 2 MHz can be inferred by inspecting the demodulator input signal spectrum, where a 2 MHz offset is specified for the display.



The Low IF receiver is comprised of a receive band SAW filter, a frequency conversion stage, an image rejection stage, and two gain stages. Resistors are used to model input and output impedances. Each nonlinear block has a noise figure specification. Power nonlinearities in the low noise amplifier (LNA), IF amplifier and mixers are specified by IP3. Image rejection is accomplished with a Hartley design, and single LO and phase shift blocks provide cosine and sine terms to mix with the I and Q branches, respectively. The summation block recombines the signals on the I branch and the phase-shifted Q branch. Image rejection quality can be controlled directly by setting a non-ideal phase offset in the Phase Shift block. To capture the RF, Image, IMD Signal and IMD Tone waveforms/spectra, choose the **Fundamental tones** to be 2450 MHz, 1 MHz and the **Harmonic Order** as 1 for the first tone and 8 for the second tone within the Configuration block. To model a thermal noise floor in the RF Blockset environment, the **Temperature** within the System Parameters section in the Configuration block is set to a noise temperature of 290.0 K.



The ADC is modeled using an a 12-bit quantizer. The quantizer takes into account the full-scale and dynamic ranges of the ADC, properly modeling its quantization noise floor.

A digital receiver demodulates the waveform for bit error rate calculation. This noncoherent FSK receiver assumes perfect timing synchronization, such that each FSK pulse is integrated over one and only one symbol.

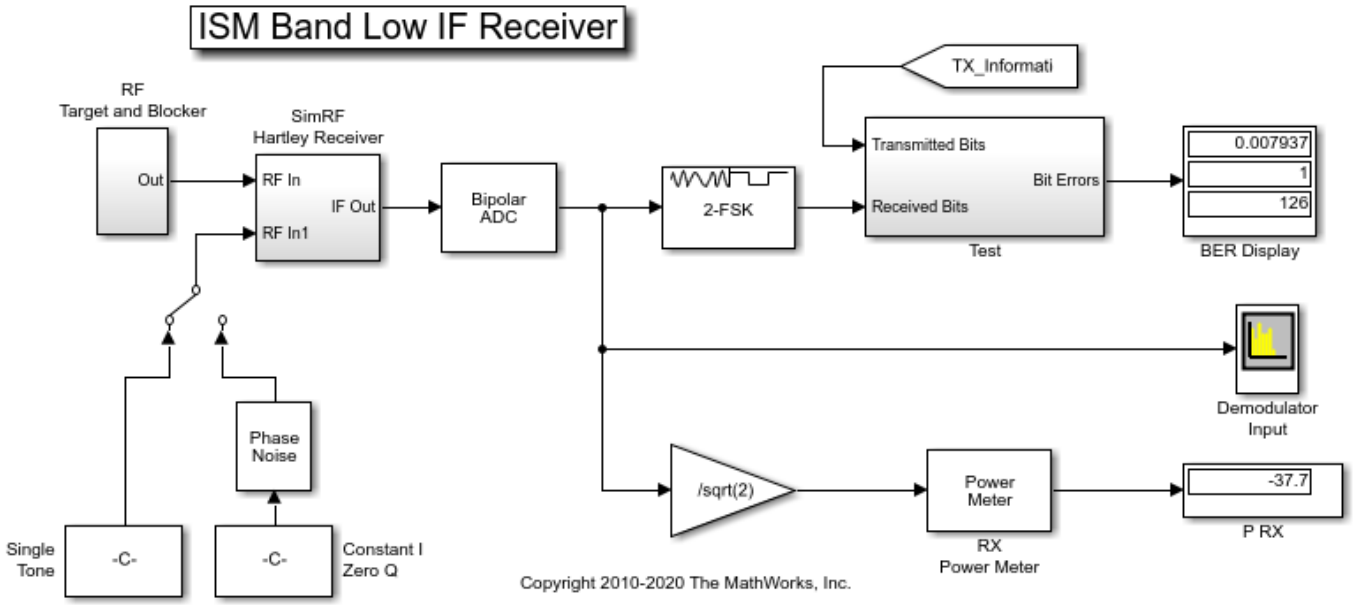
Running the Example

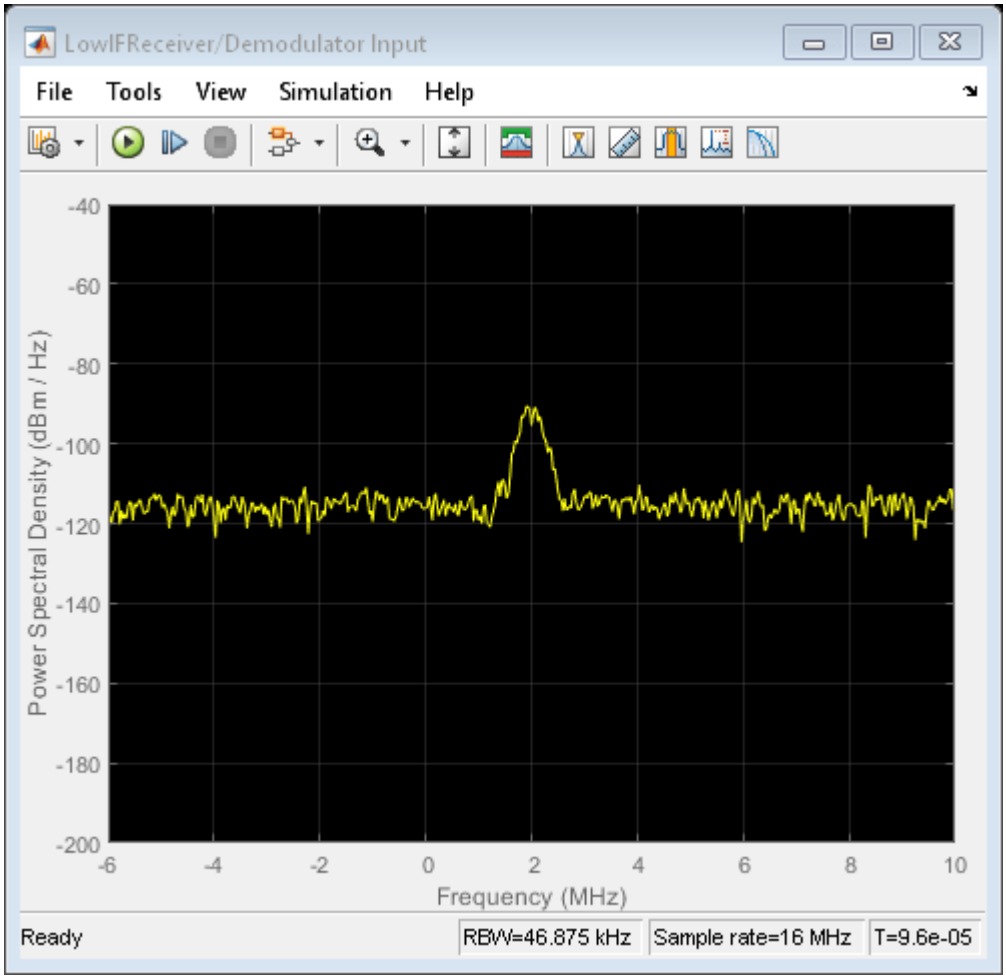
Running the example simulates a design that meets an uncoded BER spec of less than 1%. Modifications to the signals and component specifications in the receiver and ADC have a direct impact on the receiver performance. Manual switches enable you to:

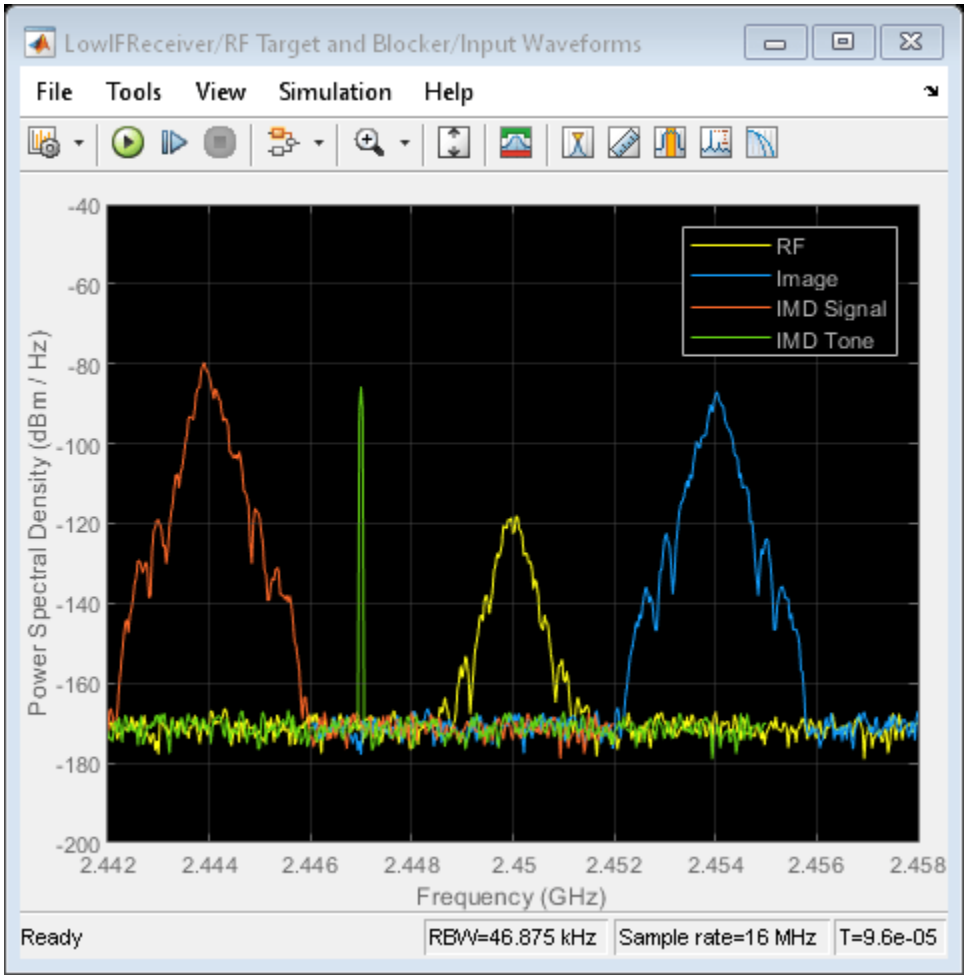
1. Select a power level for the IMD blocker tone of -33 dBm or -45 dBm
2. Select an ideal or noisy LO.

Other possible changes to the design include:

- Image rejection ratio (IRR) of the Hartley design. The IRR of the present design (dPhi=0.01 degrees) is -40 dB. For more information on calculating IRR, see the example `“Measuring Image Rejection Ratio in Receivers” (RF Blockset)` Measuring Image Rejection Ratio in Receivers>.
- Modulation schemes
- Baseband filtering options
- Signal power levels
- Signal carrier frequencies
- Noise figures
- Non-linear gain parameters
- Interstage matching
- ADC bit length and full scale range


ISM Band Low IF Receiver
RF
Target and Blocker
Out
SimRF
Hartley Receiver
RF In
IF Out
RF In1
Bipolar
ADC
2-FSK
TX_Informati
Transmitted Bits
Bit Errors
Received Bits
Test
0.007937
1
126
BER Display
Demodulator
Input
Phase
Noise
/sqrt(2)
Power
Meter
-37.7
P RX
RX
Power Meter
Single
Tone
-C-
-C-
Constant I
Zero Q
Copyright 2010-2020 The MathWorks, Inc.


LowIFReceiver/Demodulator Input
File
Tools
View
Simulation
Help
Power Spectral Density (dBm / Hz)
-40
-60
-80
-100
-120
-140
-160
-180
-200
-6
-4
-2
0
2
4
6
8
10
Frequency (MHz)
Ready
RBW=46.875 kHz
Sample rate=16 MHz
T=9.6e-05



See Also

Related Examples

- “RF Noise Modeling” on page 1-39
- “Top-Down Design of an RF Receiver” on page 1-17

More About

- “Circuit Envelope Simulation” (RF Blockset)

Shared spc_channel Examples (comm/antenna/phased)

RF Propagation and Visualization

RF propagation models describe the behavior of signals as they travel through the environment. You can display transmitter sites, receiver sites, and RF propagation visualizations by using Site Viewer, an interactive 3-D viewer. Site Viewer enables you to visualize propagation models in both outdoor and indoor environments.

Visualize Outdoor Wireless Coverage

Display transmitter and receiver sites on a 3-D globe, calculate the distance and angles between the sites, and analyze the signal strength of the transmitter at the receiver site. Display a communication link, a coverage map, and a signal-to-interference-plus-noise ratio (SINR) map.

Display Sites

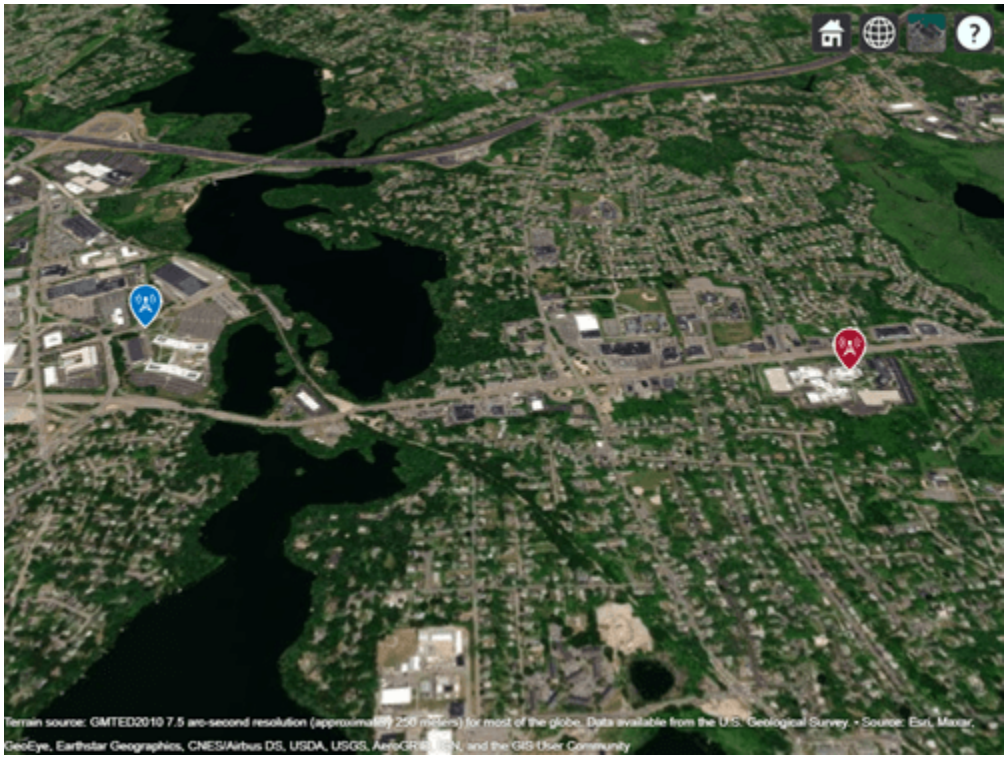
Create a transmitter site and a receiver site. Specify the position using geographic coordinates in degrees.

```
tx = txsite("Latitude",42.3001,"Longitude",-71.3504);
rx = rxsite("Latitude",42.3021,"Longitude",-71.3764);
```

Display the sites in Site Viewer. Site Viewer displays geographic sites on an interactive 3-D globe. You can customize the propagation environment of the 3-D globe by using DTED terrain and OpenStreetMap® buildings.

```
show(tx)
show(rx)
```

Pan the map by clicking and dragging. Zoom out by using the scroll wheel.



Find Distance and Angles

Calculate the distance between the sites in meters. By default, the `distance` function calculates the distance along a straight line between the sites. This straight-line path is called the Euclidean path and ignores all obstructions, including the Earth.

```
dm = distance(tx,rx)
```

```
dm = 2.1556e+03
```

You can also calculate distance using a great circle path, which considers the curvature of the Earth.

Calculate the azimuth and elevation angles between the sites. For geographic sites, the `angle` function returns the azimuth angle in degrees, measured counterclockwise from the east. The `angle` function returns the elevation angle in degrees from the horizontal plane.

```
[az,el] = angle(tx,rx)
```

```
az = 174.0753
```

```
el = -0.7267
```

Analyze Signal Strength

The signal strength of a transmitter at a receiver site is given by the following equation:

$$P_{rx} = P_{tx} + G_{tx} + G_{rx} - \text{pathloss}$$

where:

- P_{rx} is the power available at the receiver.
- P_{tx} is the transmitter output power.
- G_{tx} is the transmitter gain.
- G_{rx} = is the receiver gain.
- pathloss is the RF attenuation suffered by the transmitter signal when it arrives at the receiver.

Calculate the signal strength at the desk receiver site. By default, the `sigstrength` function calculates signal strength in power units (dBm). You can also calculate the signal strength in electric field strength units (dBμV/m).

```
ss = sigstrength(rx,tx)
```

```
ss = -67.0767
```

The link margin measures the robustness of the communication link. Calculate the link margin by subtracting the required receiver sensitivity from the signal strength.

```
margin = abs(rx.ReceiverSensitivity - ss)
```

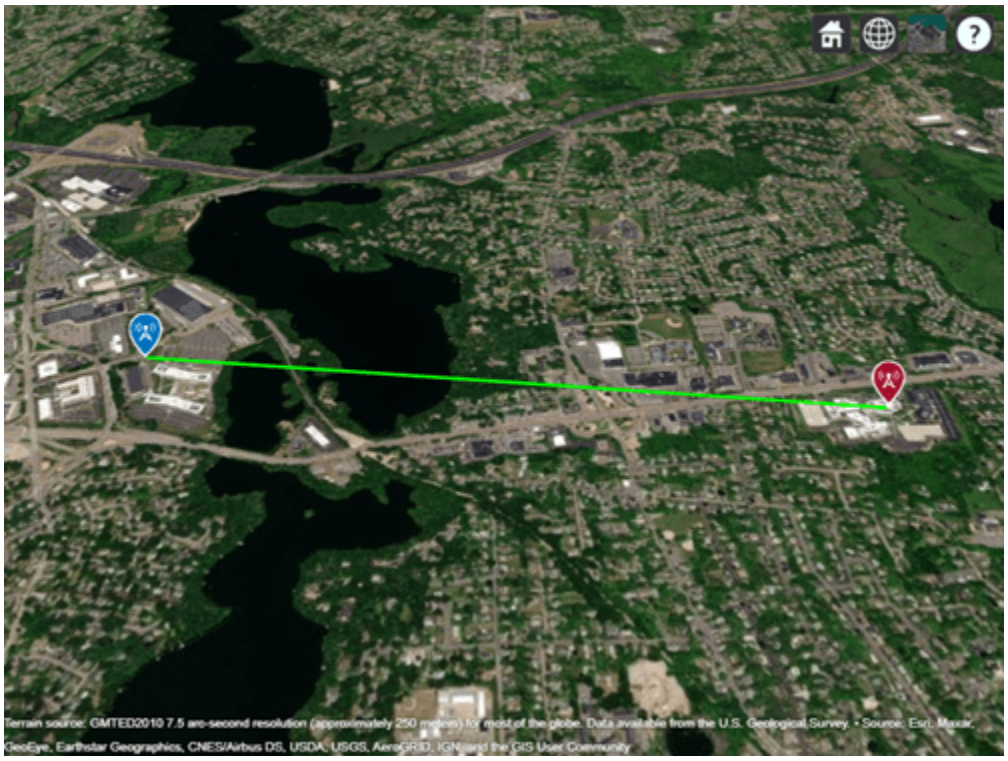
```
margin = 32.9233
```

Display Communication Link

Display the communication link status between the sites. The success of the link depends on the power received by the receiver from the transmitter. By default, a green line indicates that the

received power meets or exceeds the receiver sensitivity. A red line indicates unsuccessful communication.

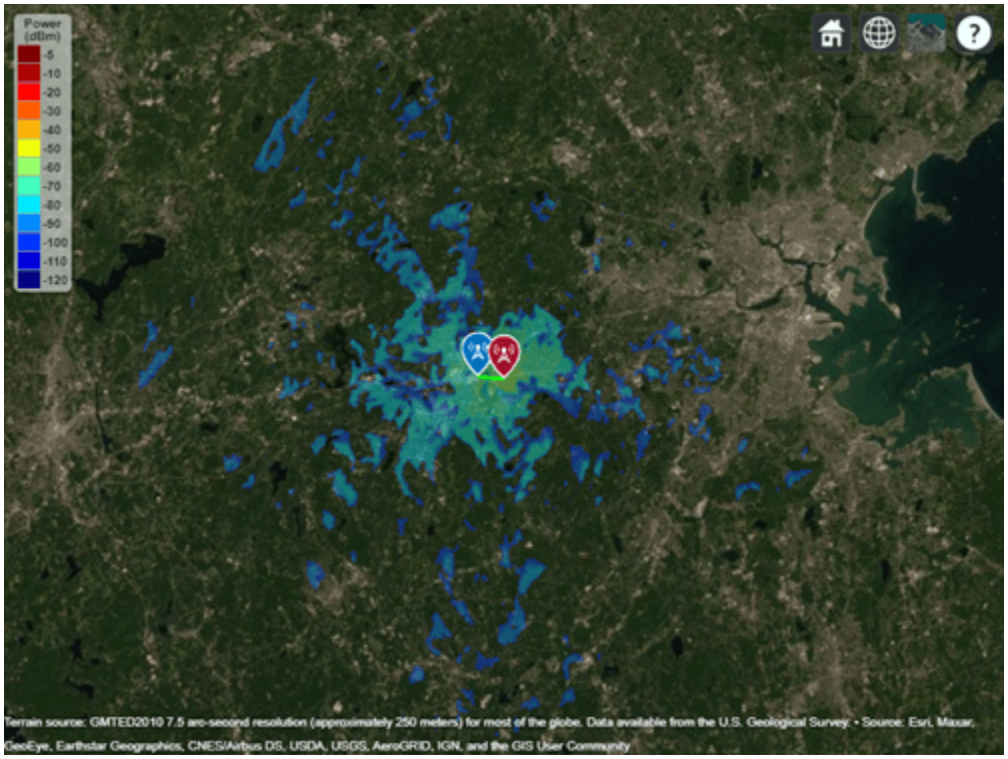
```
link(rx,tx)
```



Display Coverage Map

Display the coverage map of the transmitter. A coverage map visualizes the service area of the transmitter, which is where the received signal strength for a reference receiver meets its sensitivity. You can create coverage maps that depict signal strength as either a power quantity (typically dBm) or a voltage quantity (typically dBμV/m).

```
coverage(tx,"SignalStrengths",-100:5:-60)
```



Find New Transmitter Site

Create and display a new transmitter site that is 1 km north of the existing transmitter site. Specify the antenna height as 30 m.

```
[lat,lon] = location(tx,1000,90);
tx2 = txsite("Latitude",lat,"Longitude",lon,"AntennaHeight",30);
show(tx2)
```

Calculate SINR

Calculate the SINR in decibels. The SINR of a receiver is given by the following equation:

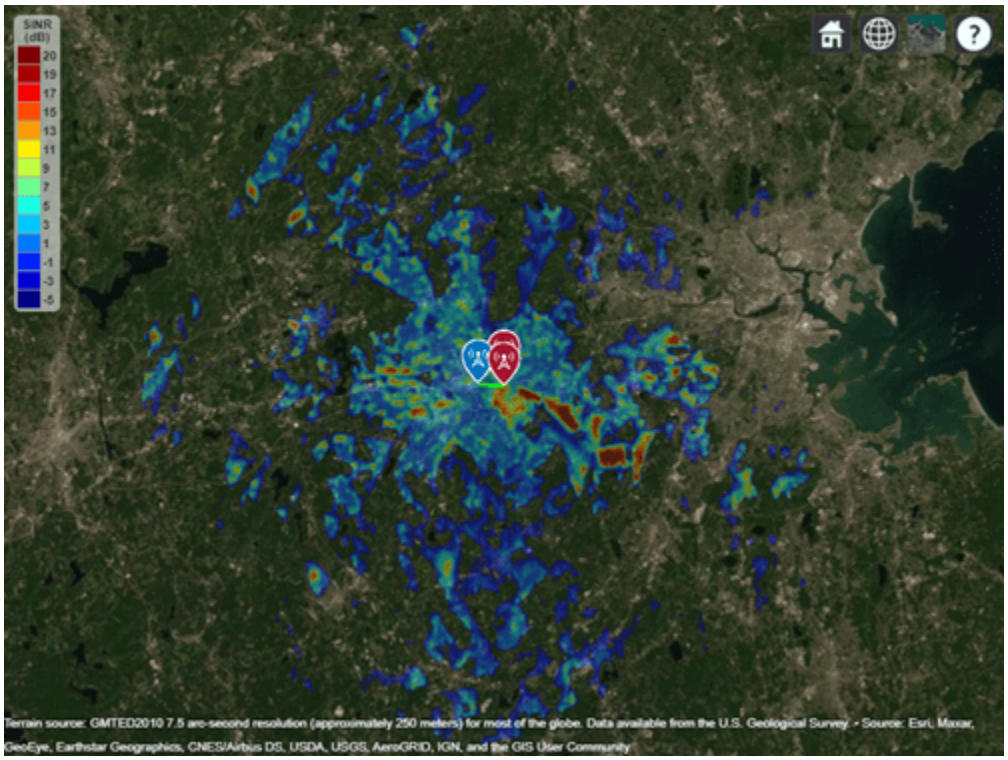
$$\text{SINR} = \frac{S}{I + N}$$

where:

- S is the received power of the signal of interest.
- I is the received power of interfering signals in the network.
- N is the total received noise power.

When Site Viewer has terrain data, the `sinr` function incorporates the terrain into the calculations.

```
sinr([tx,tx2])
```



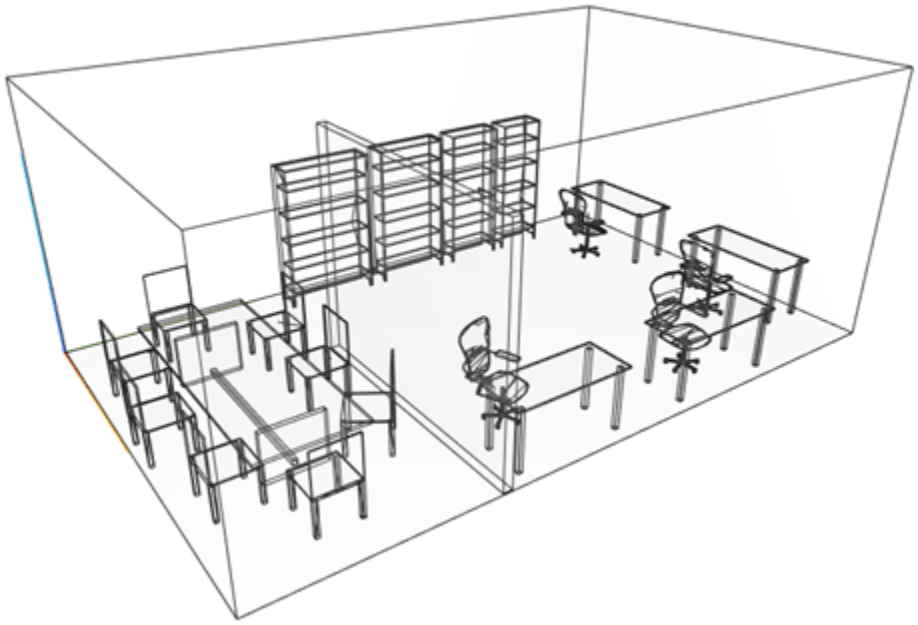
Visualize Indoor Propagation Paths

Import a 3-D scene model of a conference room. Display sites and find propagation paths between the sites.

Import Scene

Import and view an STL file. The file models an indoor office with a conference room and open space separated by a partial wall. STL files contain geometry information and do not contain information about colors, surfaces, or textures.

```
viewer = siteviewer("SceneModel","office.stl");
```



Display Sites

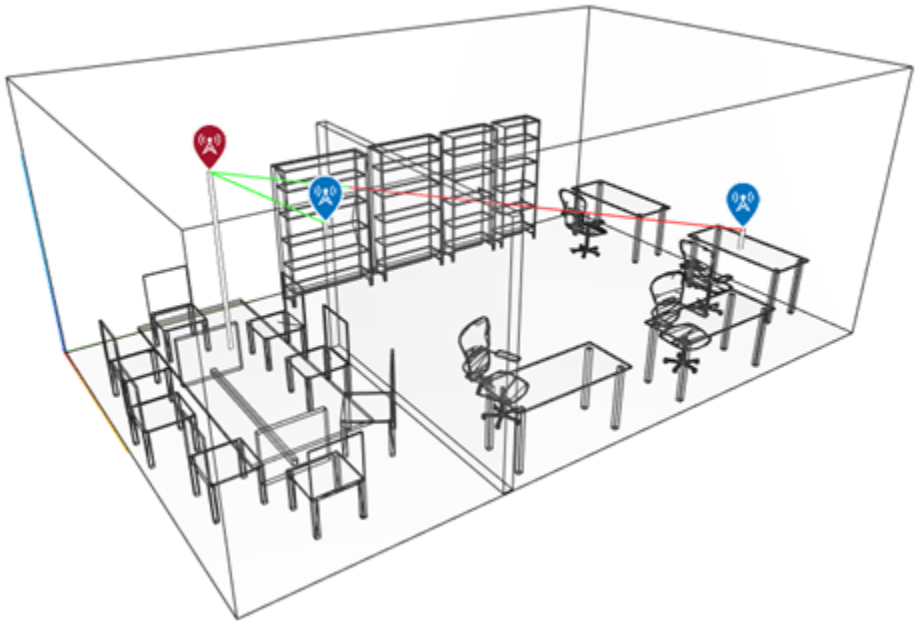
Place one transmitter near the ceiling in the conference room. Place one receiver on a desk in the open space and another receiver on a shelf. Specify the position using Cartesian coordinates in meters.

```
tx = txsite("cartesian","AntennaPosition",[2; 1.3; 2.5]);
rx_desk = rxsite("cartesian","AntennaPosition",[3.6; 7.5; 1]);
rx_shelf = rxsite("cartesian","AntennaPosition",[0.4; 3.3; 1]);
```

Display the receivers and the line-of-sight paths.

```
los(tx,[rx_desk rx_shelf])
```

Pan the scene by left-clicking, zoom by right-clicking or by using the scroll wheel, and rotate by clicking the middle button and dragging or by pressing **Ctrl** and left-clicking and dragging.



The path to the shelf receiver is clear and the path to the desk receiver is obstructed.

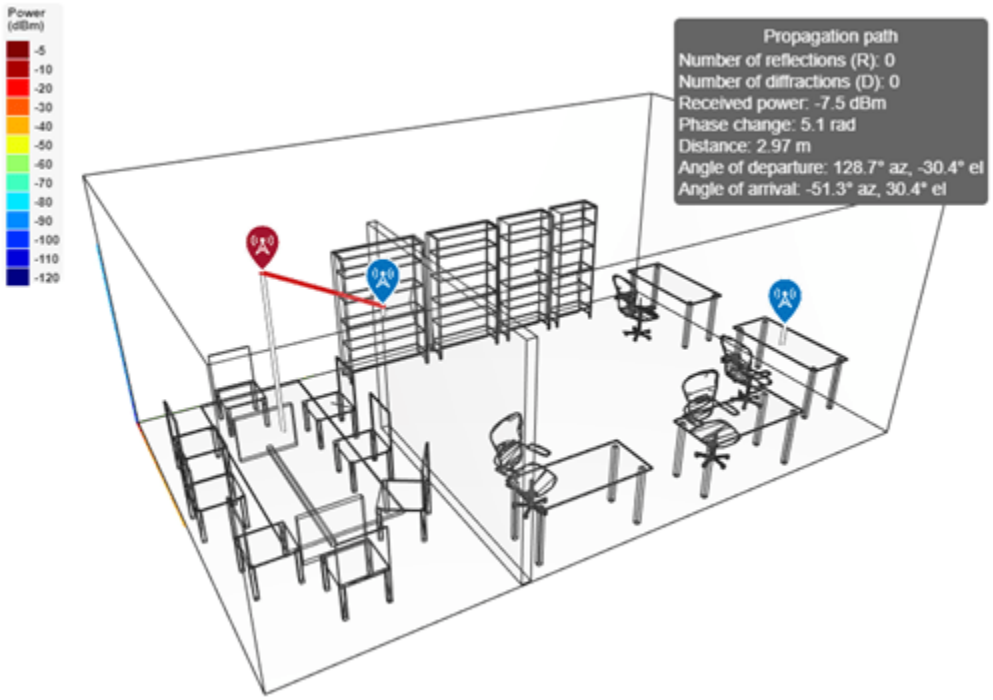
Display Propagation Paths

Create a ray tracing propagation model that uses the shooting and bouncing rays (SBR) method. Specify the surface material as wood.

```
pm = propagationModel("raytracing", ...
    "CoordinateSystem","cartesian", ...
    "Method","sbr", ...
    "SurfaceMaterial","wood");
```

Display propagation paths that are within the line of sight by setting the `MaxNumReflections` property to `0`. Unlike the `los` function, the `raytrace` function does not show obstructed paths.

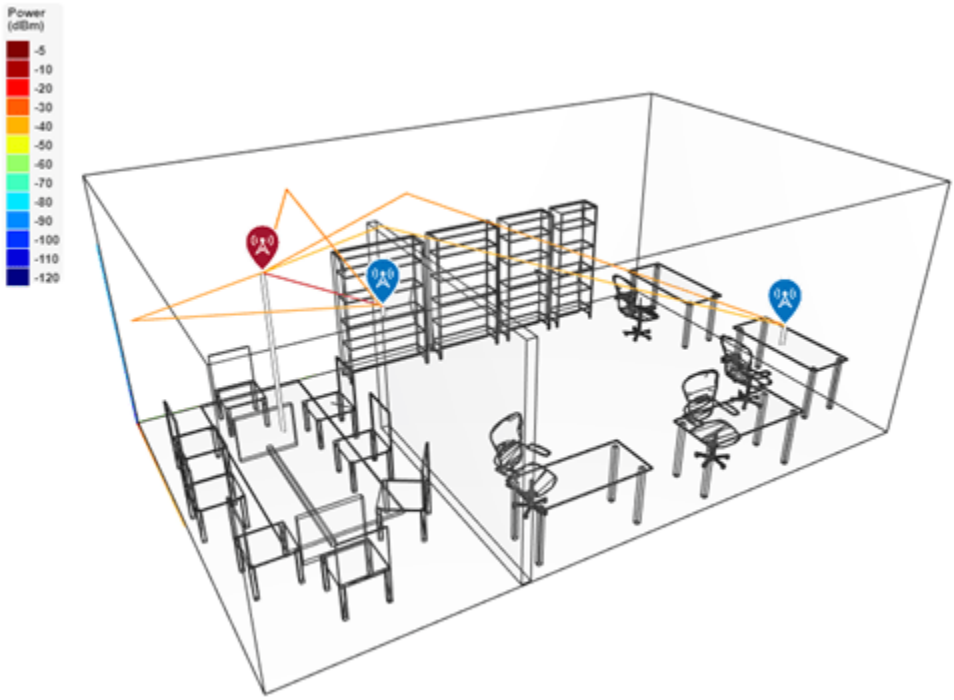
```
pm.MaxNumReflections = 0;
clearMap(viewer)
raytrace(tx,[rx_desk rx_shelf],pm)
```



The `raytrace` function finds one line-of-sight path. You can view information about the path, such as the received power, by clicking on the path.

Display propagation paths with up to one reflection.

```
pm.MaxNumReflections = 1;
raytrace(tx,[rx_desk rx_shelf],pm)
```



The updated model calculates additional paths.

See Also

Functions
`coverage` | `sigstrength` | `link` | `sinr` | `raytrace`

Objects
`siteviewer` | `txsite` | `rxsite`

More About

- “Visualize Antenna Coverage Map and Communication Links” on page 2-12
- “Urban Link and Coverage Analysis Using Ray Tracing” on page 2-21
- “Indoor MIMO-OFDM Communication Link Using Ray Tracing” on page 8-9

Visualize Antenna Coverage Map and Communication Links

This example shows how to calculate and visualize signal strength between a transmitter and multiple receivers. The visualizations include an area coverage map and colored communication links. The example also shows selection of a directional antenna in order to achieve a communication link to a specific location.

Define Transmitter Site

```
% Define transmitter site at MathWorks (3 Apple Hill Dr, Natick, MA)
fq = 6e9; % 6 GHz
tx = txsite("Name","MathWorks", ...
    "Latitude",42.3001, ...
    "Longitude",-71.3504, ...
    "Antenna",design(dipole,fq), ...
    "AntennaHeight",60, ...        % Units: meters
    "TransmitterFrequency",fq, ... % Units: Hz
    "TransmitterPower",15);        % Units: Watts
```

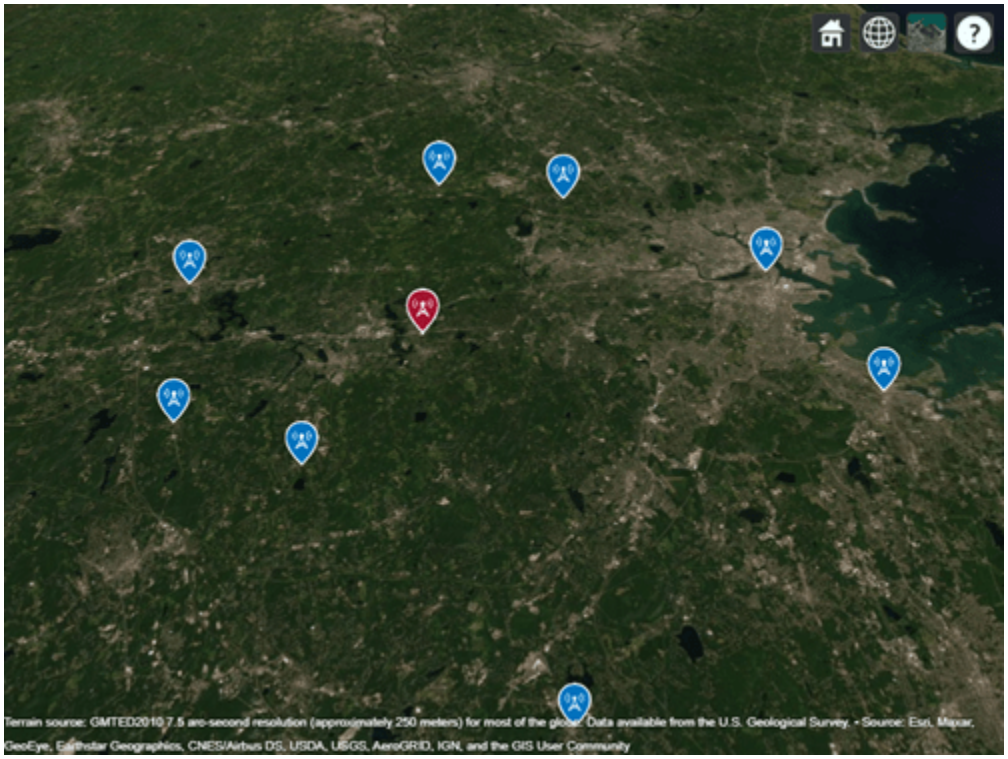
Define Receiver Sites

```
% Define receiver sites in several surrounding towns and cities
rxNames = [...
   "Boston, MA","Lexington, MA","Concord, MA","Marlborough, MA", ...
   "Hopkinton, MA","Holliston, MA","Foxborough, MA","Quincy, MA"];

rxLocations = [...
   42.3601 -71.0589; ... % Boston
   42.4430 -71.2290; ... % Lexington
   42.4604 -71.3489; ... % Concord
   42.3459 -71.5523; ... % Marlborough
   42.2287 -71.5226; ... % Hopkinton
   42.2001 -71.4245; ... % Holliston
   42.0654 -71.2478; ... % Foxborough
   42.2529 -71.0023];    % Quincy

% Define receiver sensitivity. Sensitivity is the minimum signal strength in
% power that is necessary for the receiver to accurately detect the signal.
rxSensitivity = -90; % Units: dBm

rxs = rxsite("Name",rxNames, ...
    "Latitude",rxLocations(:,1), ...
    "Longitude",rxLocations(:,2), ...
    "Antenna",design(dipole,tx.TransmitterFrequency), ...
    "ReceiverSensitivity",rxSensitivity); % Units: dBm
```

Show Sites on a Map

Show transmitter and receiver sites on a map. Site markers may be clicked to display site information.

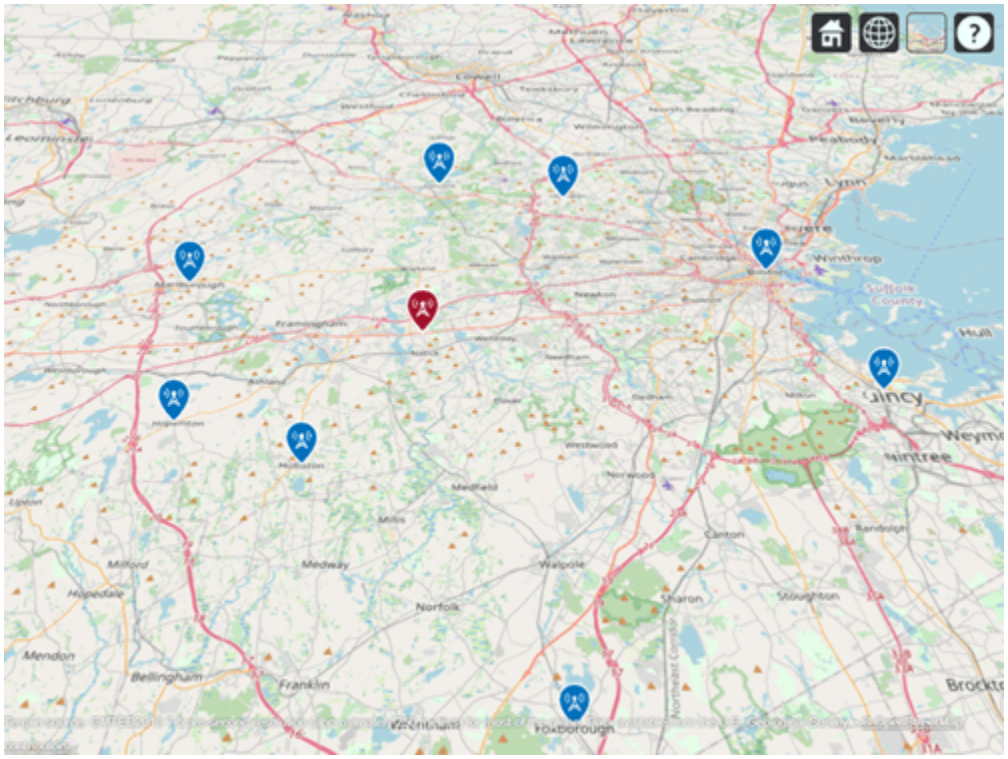
```
viewer = siteviewer;
show(tx)
show(rxs)
```



Customize Site Viewer

Set the map imagery using the `Basemap` property. Alternatively, open the map imagery picker in Site Viewer by clicking the second button from the right. Select "OpenStreetMap" to see streets and labels on the map. Rotate the view to show an overhead perspective.

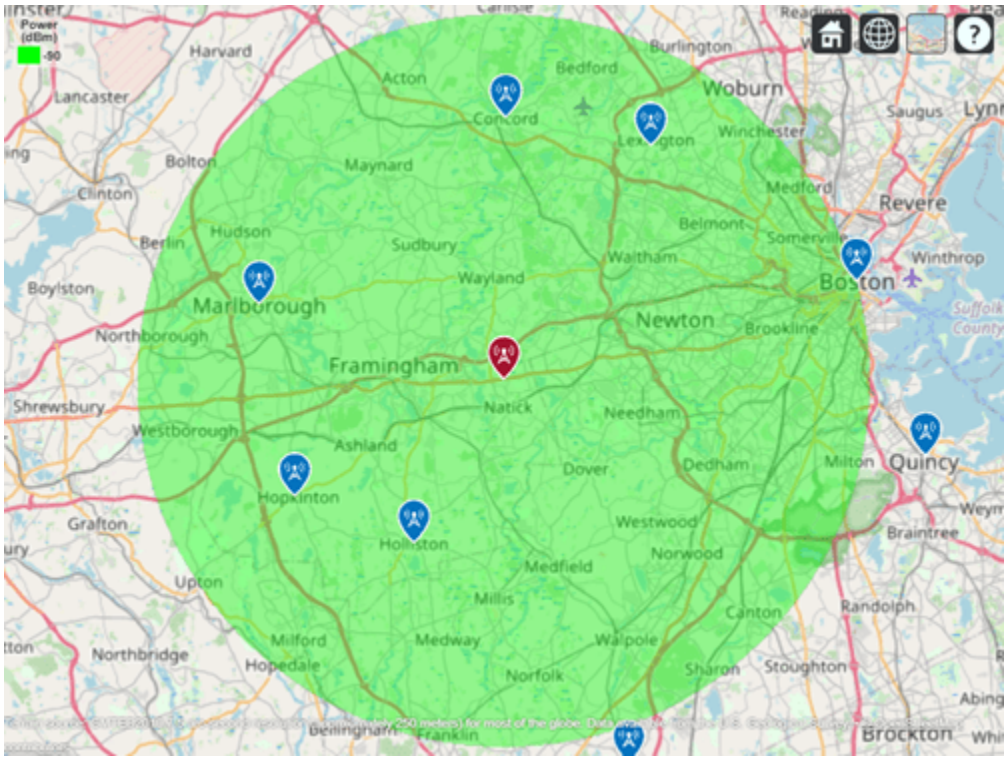
```
viewer.Basemap = "openstreetmap";
```



Display Idealized Coverage Map using Dipole Antenna

Display coverage map. A coverage map shows the geographic area where a receiver will obtain good reception, which is where transmitted signal strength meets or exceeds the receiver"s sensitivity. Transmitted signal strength in power (dBm) is computed using a free-space propagation model, which disregards terrain, obstacles, and atmospheric effects. As a result, the coverage map shows idealized coverage area in the absence of any path loss impairments beyond free space loss.

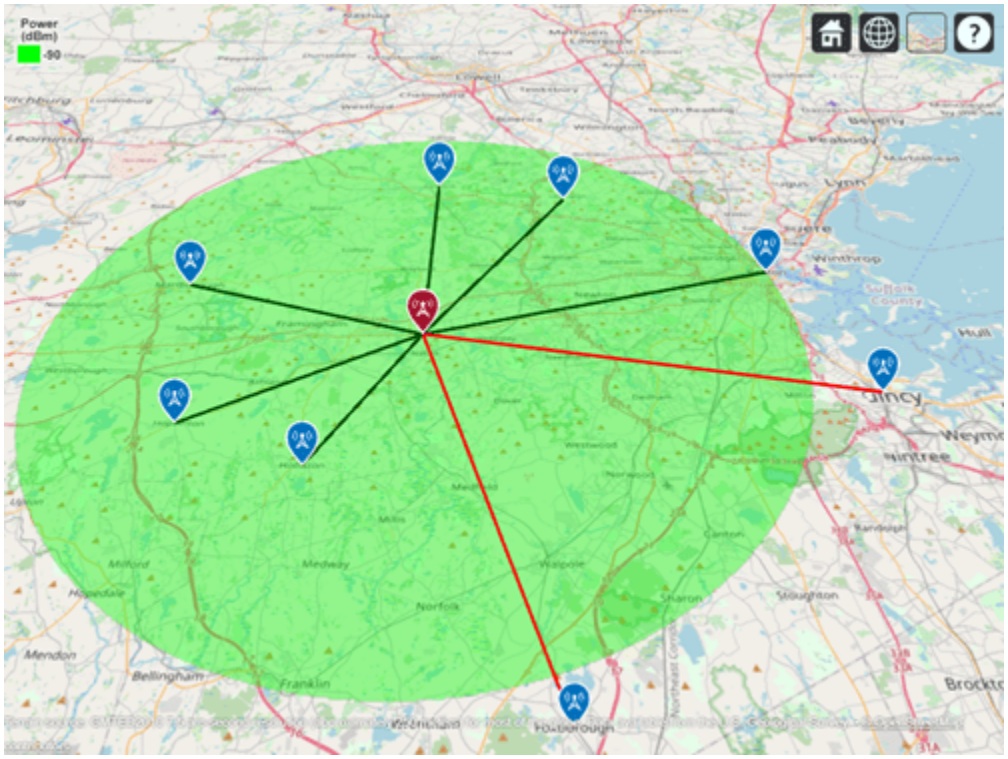
```
coverage(tx,"freespace", ...
    "SignalStrengths",rxSensitivity)
```



Plot Communication Links using Dipole Antenna

Plot communication links on the map. Red links appear where the receiver is outside of the coverage zone, and green links appear where the receiver is within the coverage zone. Link lines may be clicked to display link statistics. To contrast the colors of the coverage zone and successful links, specify the color of successful links as dark green.

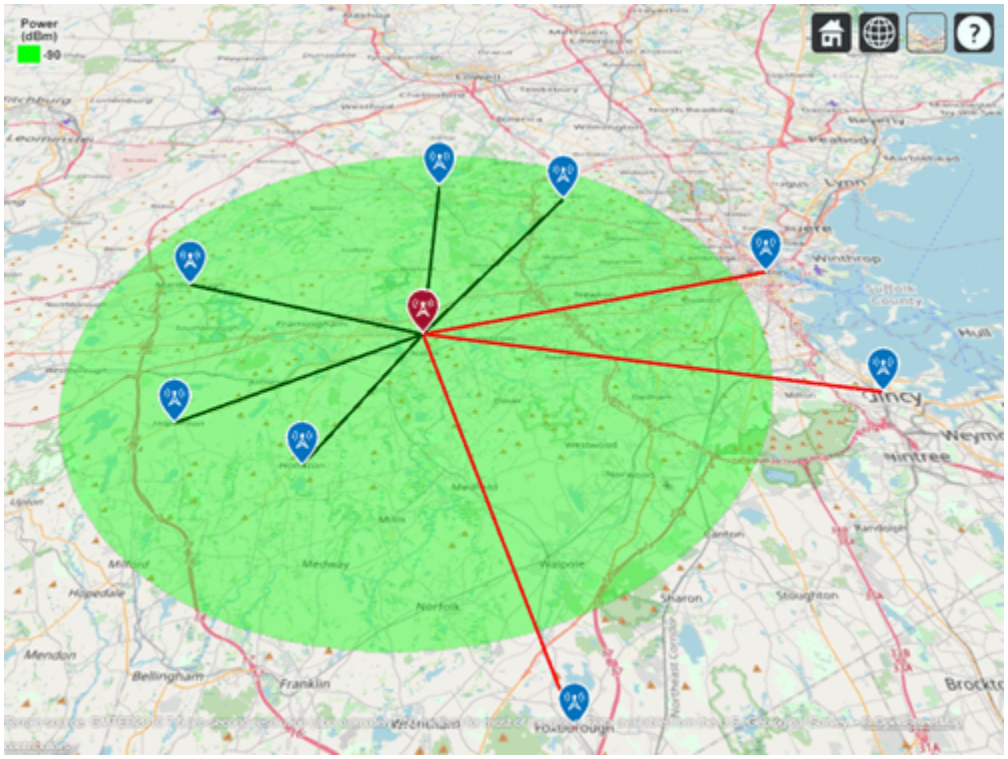
```
sc = [0 0.3 0];
link(rxs,tx,"freespace","SuccessColor",sc)
```



Use Rain Propagation Model

Update the coverage map and links to include path loss due to rain. Note that Boston, MA is no longer inside the coverage zone.

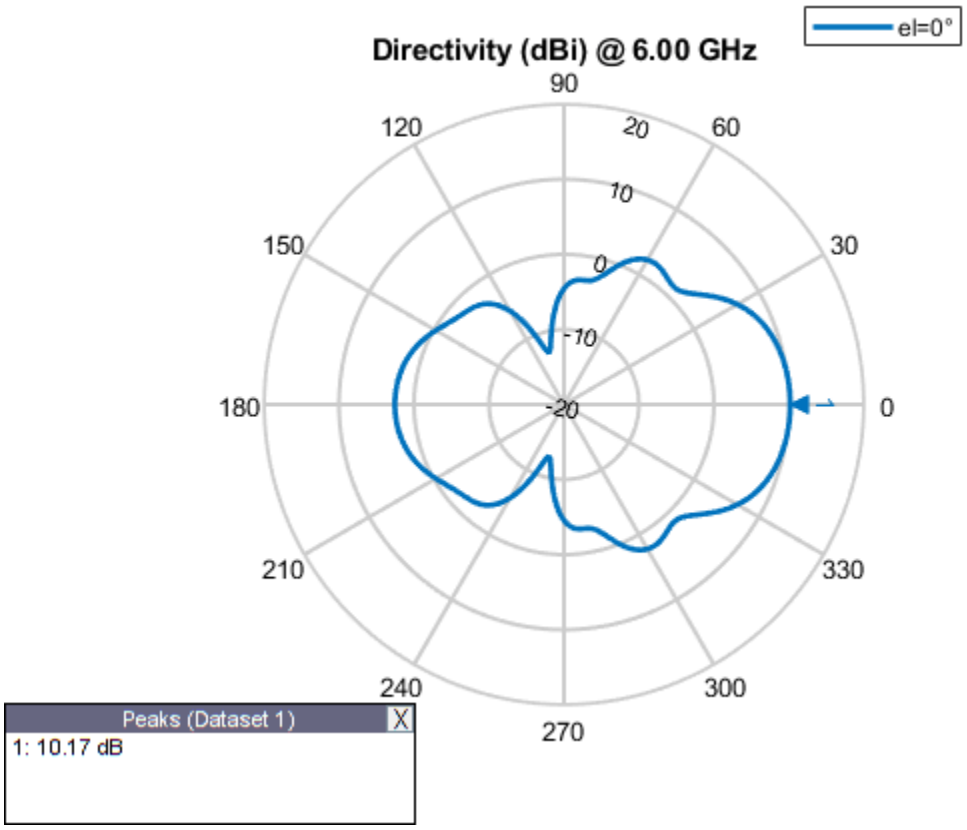
```
coverage(tx,"rain","SignalStrengths", rxSensitivity)
link(rxs,tx,"rain","SuccessColor",sc)
```



Define Directional Antenna

The dipole antenna transmitter results in a few receiver sites outside of the coverage zone, including the receiver in Boston, MA. Now assume a requirement of the transmitter is to achieve a communication link with Boston. Define a directional antenna that can increase antenna gain in that direction.

```
% Define Yagi-Uda antenna designed for transmitter frequency
yagiAnt = design(yagiUda,tx.TransmitterFrequency);

% Tilt antenna to direct radiation in XY-plane (i.e. geographic azimuth)
yagiAnt.Tilt = 90;
yagiAnt.TiltAxis = "y";

f = figure;

% Show directivity pattern
patternAzimuth(yagiAnt,tx.TransmitterFrequency)
```

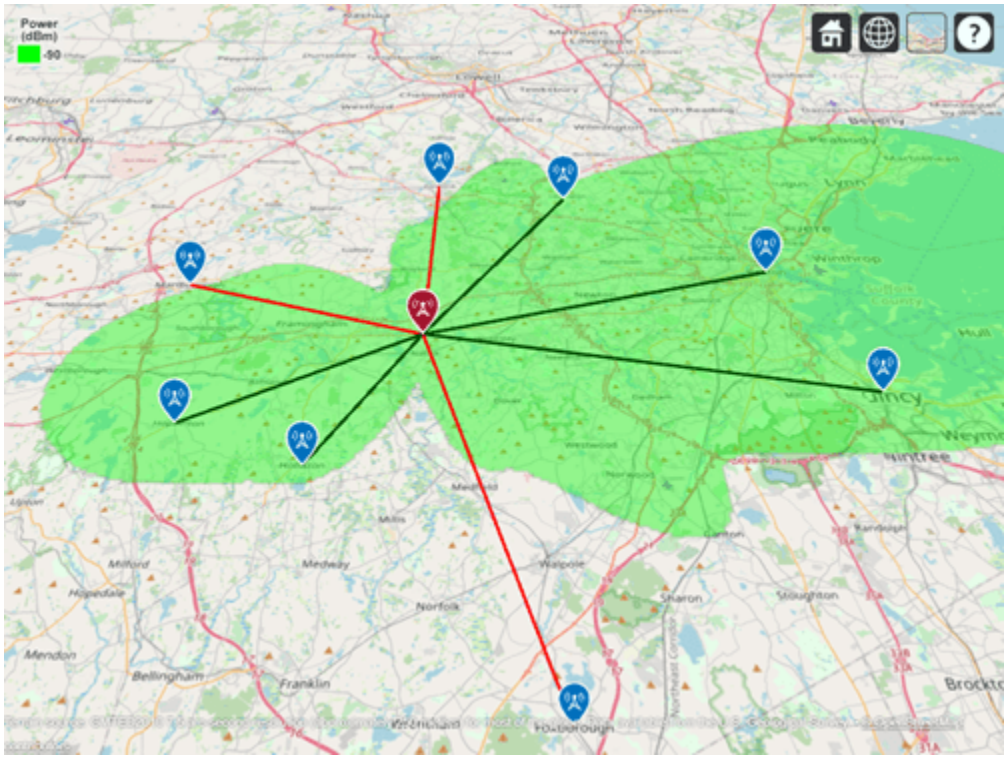



```
%Close the previous figure
if (isvalid(f))
    close(f);
end
```

Display Coverage Map using Yagi-Uda Antenna

Update the coverage map and links. Boston is now within the coverage zone, but communication links with receivers in other directions are lost.

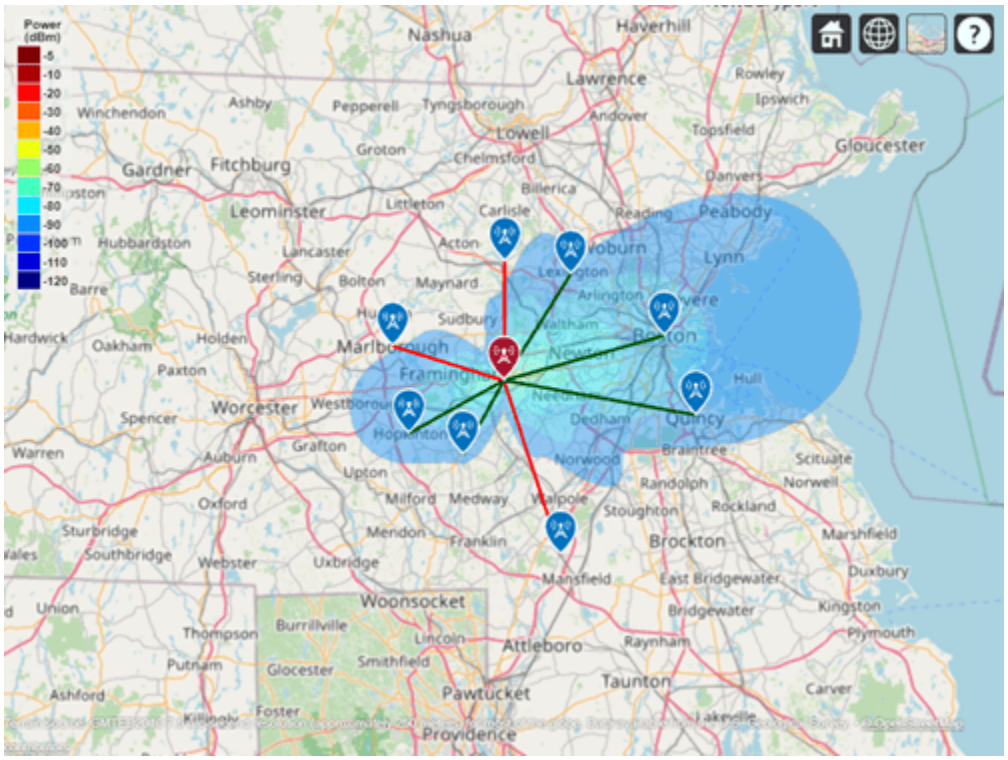
```
% Update transmitter antenna
tx.Antenna = yagiAnt;

% Point main beam toward Boston, MA by assigning azimuth angle between
% transmitter location and Boston receiver location
tx.AntennaAngle = angle(tx, rxs(1));

% Update visualizations, using "rain" propagation model
coverage(tx,"rain","SignalStrengths",rxSensitivity)
link(rxs,tx,"rain","SuccessColor",sc)
```



Display Contoured Coverage Map using Multiple Signal Strengths

When a single signal strength is specified, the coverage map is green for the coverage region. Specify multiple signal strengths to generate a coverage map with contours for different signal levels.

```
% Define signal strengths from sensitivity to -60 dB
sigStrengths = rxSensitivity:5:-60;

% Update coverage map
coverage(tx,"rain","SignalStrengths",sigStrengths)
```



See Also

Functions
`coverage` | `link` | `design`

Objects
`txsite` | `rxsite` | `siteviewer`

Related Examples

- “Urban Link and Coverage Analysis Using Ray Tracing” on page 2-21

Urban Link and Coverage Analysis Using Ray Tracing

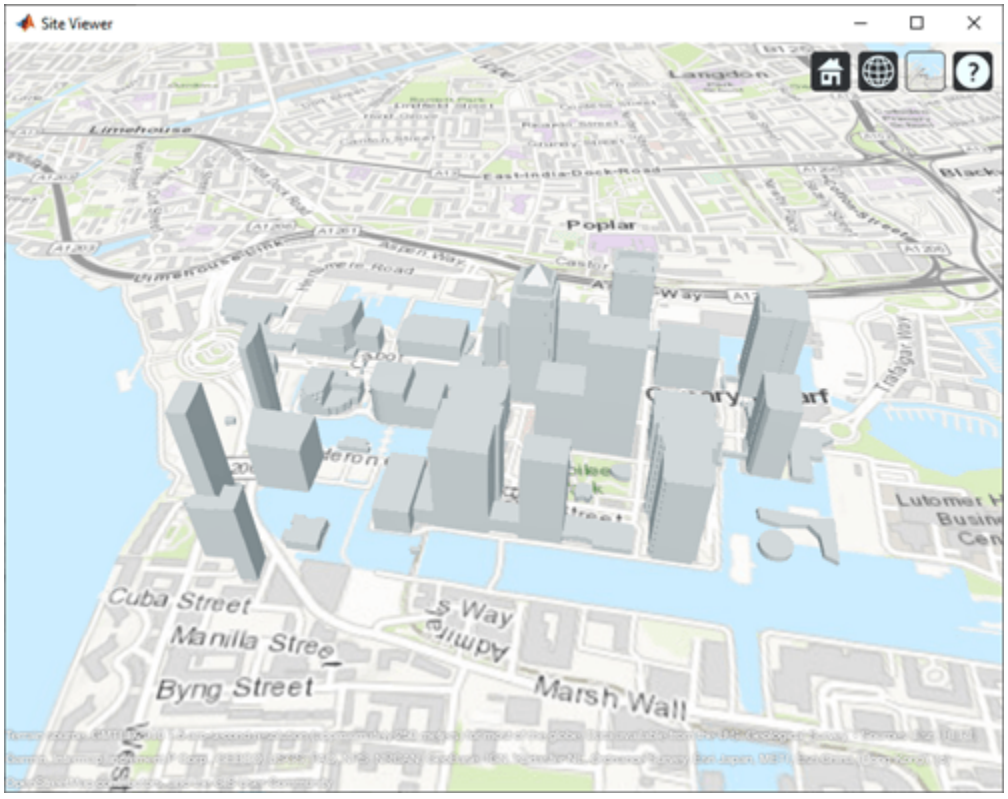
This example shows how to use ray tracing to analyze communication links and coverage areas in an urban environment. Within the example:

- Import and visualize 3-D buildings data into Site Viewer
- Define a transmitter site and ray tracing propagation model corresponding to a 5G urban scenario
- Analyze a link in non-line-of-sight conditions
- Visualize coverage using the shooting and bouncing rays (SBR) ray tracing method with different numbers of reflections and launched rays
- Optimize a non-line-of-sight link using beam steering and Phased Array System Toolbox™

Import and Visualize Buildings Data

Import an OpenStreetMap (.osm) file corresponding to Canary Wharf in London, UK. The file was 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/. The buildings information contained within the OpenStreetMap file is imported and visualized in Site Viewer.

```
viewer = siteviewer("Buildings","canarywharf.osm","Basemap","topographic");
```

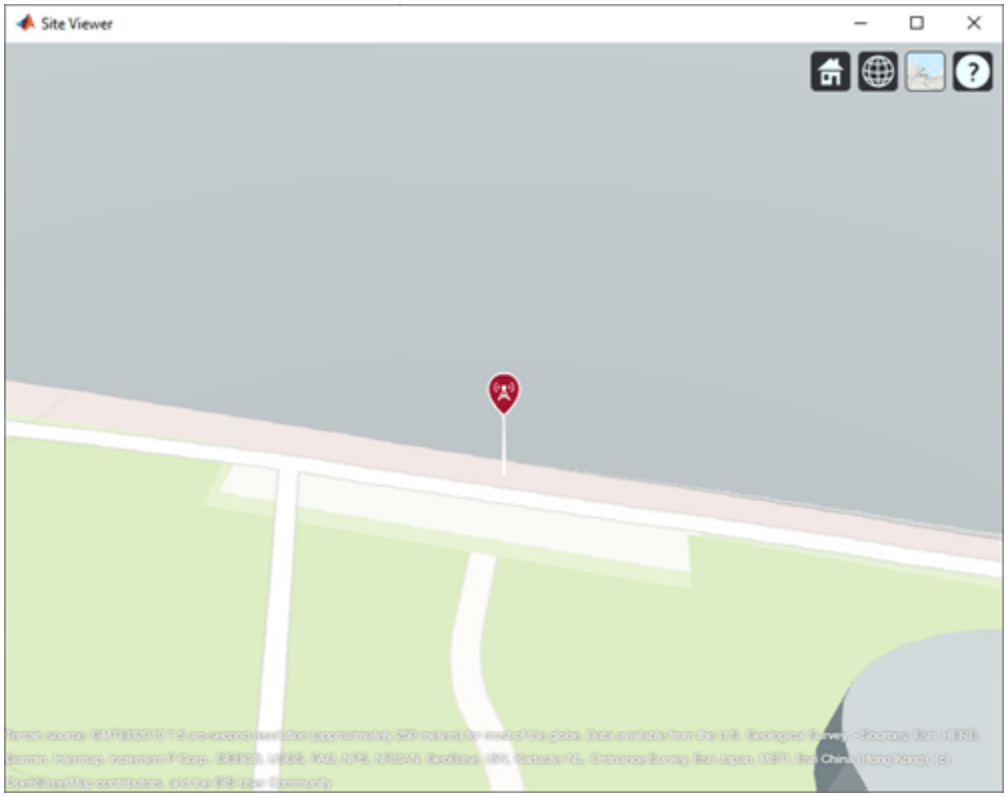


Define Transmitter Site

Define a transmitter site to model a small cell scenario in a dense urban environment. The transmitter site represents a base station that is placed on a pole servicing the surrounding area

which includes a neighboring park. The transmitter uses the default isotropic antenna, and operates at a carrier frequency of 28 GHz with a power level of 5 W.

```
tx = txsite("Name","Small cell transmitter", ...
    "Latitude",51.50375, ...
    "Longitude",-0.01843, ...
    "AntennaHeight",10, ...
    "TransmitterPower",5, ...
    "TransmitterFrequency",28e9);
show(tx)
```



View Coverage Map for Line-of-Sight Propagation

Create a ray tracing propagation model using the shooting and bouncing ray (SBR) method. The SBR propagation model uses ray tracing analysis to compute propagation paths and their corresponding path losses. Path loss is calculated from free-space loss and reflection loss due to material, and antenna polarization loss.

Set the maximum number of reflections to 0 in order to limit the initial analysis to line-of-sight propagation paths only. Set the building and terrain material types to model perfect reflection.

```
rtpm = propagationModel("raytracing", ...
    "Method","sbr", ...
    "MaxNumReflections",0, ...
    "BuildingsMaterial","perfect-reflector", ...
    "TerrainMaterial","perfect-reflector");
```

View the corresponding coverage map for a maximum range of 250 meters from the base station. The coverage map shows received power for a receiver at each ground location but is not computed for building tops or sides.

```
coverage(tx,rtpm, ...
    "SignalStrengths",-120:-5, ...
    "MaxRange",250, ...
    "Resolution",3, ...
    "Transparency",0.6)
```

Define Receiver Site in Non-Line-of-Sight Location

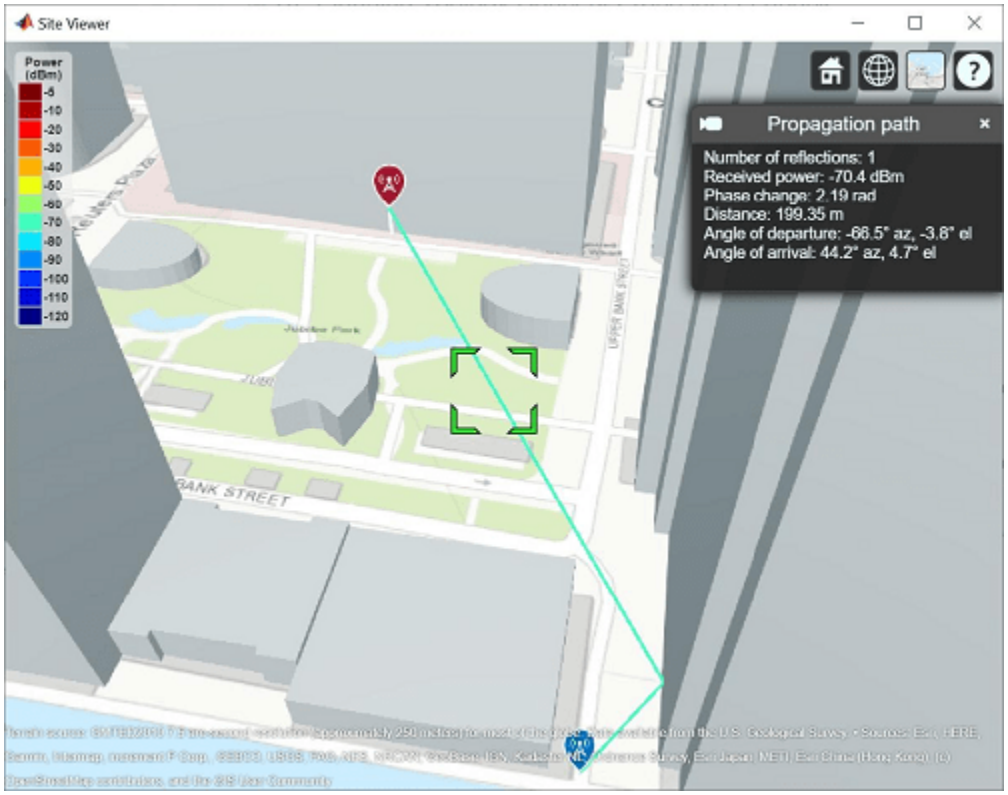
The coverage map for line-of-sight propagation shows shadowing due to obstructions. Define a receiver site to model a mobile receiver in an obstructed location. Plot the line-of-sight path to show the obstructed path from the transmitter to the receiver.

```
rx = rxsite("Name","Small cell receiver", ...
    "Latitude",51.50216, ...
    "Longitude",-0.01769, ...
    "AntennaHeight",1);

los(tx,rx)
```

Plot Propagation Path using Ray Tracing

Adjust the ray tracing propagation model to include single-reflection paths, and plot the rays. The result shows signal propagation along a single-reflection path. The path does not end exactly at the receiver site because the SBR ray tracing method computes approximate paths. Select the plotted path to view the corresponding propagation characteristics, which include received power, phase change, distance, and angles of departure and arrival.

```
rtpm.MaxNumReflections = 1;
clearMap(viewer)
raytrace(tx,rx,rtpm)
```



Analyze Signal Strength and Effect of Materials

Compute the received power using the propagation model which was previously configured to model perfect reflection. Then assign a more realistic material type and re-compute the received power. Update the rays shown in Site Viewer. The use of realistic material reflection results in about 8 dB of power loss compared to perfect reflection.

```
ss = sigstrength(rx,tx,rtpm);
disp("Received power using perfect reflection: " + ss + " dBm")
```

```
Received power using perfect reflection: -70.3924 dBm
```

```
rtpm.BuildingsMaterial = "concrete";
rtpm.TerrainMaterial = "concrete";

raytrace(tx,rx,rtpm)
ss = sigstrength(rx,tx,rtpm);
disp("Received power using concrete materials: " + ss + " dBm")
```

```
Received power using concrete materials: -78.9591 dBm
```

Include Weather Loss

Adding weather impairments to the propagation model and re-computing the received power results in another 1.5 dB of loss.

```
rtPlusWeather = ...
    rtpm + propagationModel("gas") + propagationModel("rain");
raytrace(tx,rx,rtPlusWeather)
```

```
ss = sigstrength(rx,tx,rtPlusWeather);
disp("Received power including weather loss: " + ss + " dBm")
```

```
Received power including weather loss: -80.4766 dBm
```

Plot Propagation Paths including Two Reflections

Expand the point-to-point analysis to include two-reflection paths and choose a smaller angular separation between launched rays for the SBR method. The visualization shows two clusters of propagation paths and the total received power increases by approximately 3 dB compared to the single-reflection paths.

```
rtPlusWeather.PropagationModels(1).MaxNumReflections = 2;
rtPlusWeather.PropagationModels(1).AngularSeparation = "low";

ss = sigstrength(rx, tx, rtPlusWeather);
disp("Received power with two-reflection paths: " + ss + " dBm")
```

```
Received power with two-reflection paths: -77.1445 dBm
```

```
clearMap(viewer)
raytrace(tx,rx,rtPlusWeather);
```

View Coverage Map with Single-Reflection Paths

Use the configured propagation model and re-generate a coverage map including single-reflection paths and weather impairments. Code to re-generate the coverage results is included but commented out. The results, producible by running the code, are loaded from file to save several minutes of computation time in the example presentation. The resultant coverage map shows received power in the area around the non-line-of-site receiver analyzed above.

```
rtPlusWeather.PropagationModels(1).MaxNumReflections = 1;
clearMap(viewer)
```

Load coverage results and plot. Coverage results were generated using commented coverage call below, which takes a few minutes to complete.

```
show(tx)

coverageResults = load("coverageResults.mat");
contour(coverageResults.propDataSingleRef, ...
    "Type","power", ...
    "Transparency",0.6)

% coverage(tx,rtPlusWeather, ...
%     "SignalStrengths",-120:-5, ...
%     "MaxRange", 250, ...
%     "Resolution",2, ...
%     "Transparency",0.6);
```

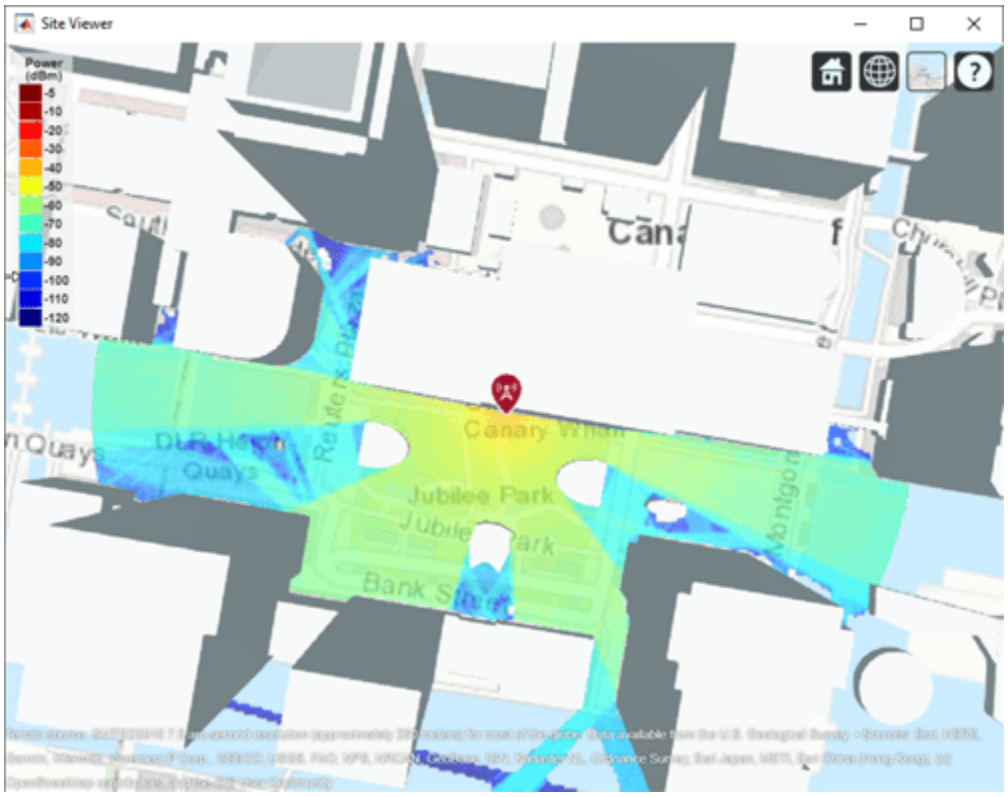
View Coverage Map with Four-Reflection

Account for more propagation paths and generate a more accurate coverage map by increasing the maximum number of reflections for the ray tracing analysis to 4. Visualize a pre-computed coverage map again which shows nearly full coverage for the area around the transmitter site.

```
rtPlusWeather.PropagationModels(1).MaxNumReflections = 4;
clearMap(viewer)
```

Use pre-loaded coverage results to plot. Coverage results were generated using commented coverage call below, which may take a few hours to complete depending on the computer hardware.

```
show(tx)
contour(coverageResults.propDataFourRef, ...
    "Type","power", ...
    "Transparency",0.6)

% coverage(tx,rtPlusWeather, ...
%     "SignalStrengths",-120:-5, ...
%     "MaxRange", 250, ...
%     "Resolution",2, ...
%     "Transparency",0.6);
```

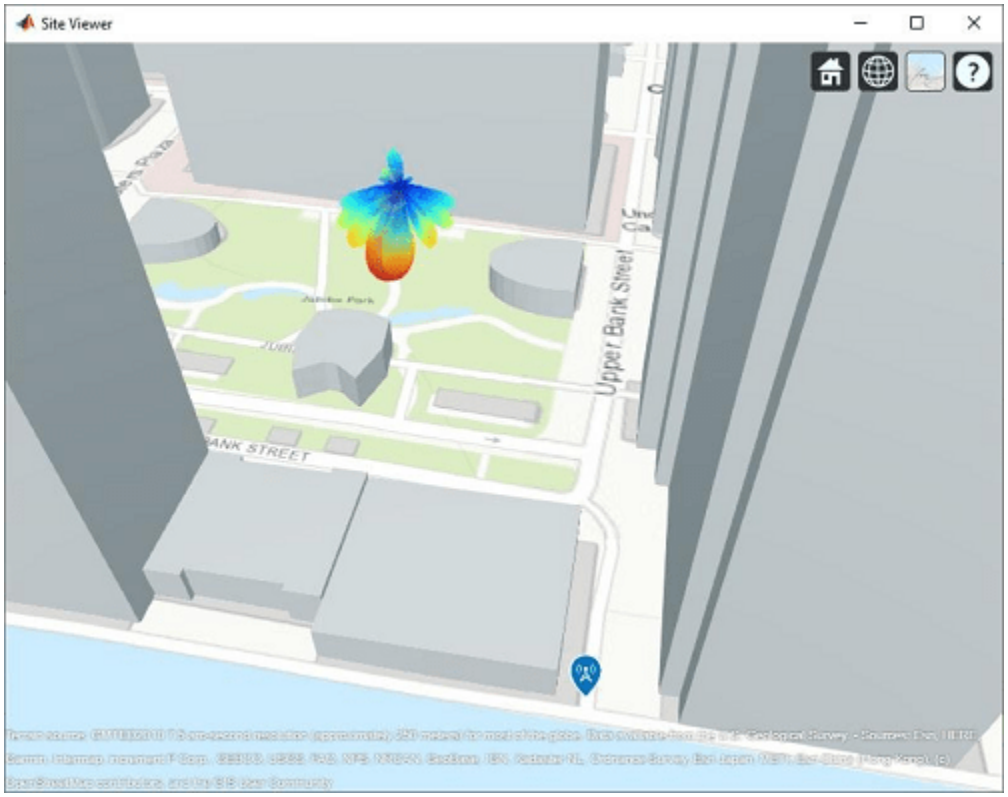


Use Beam Steering to Enhance Received Power

Many modern communications systems use techniques to steer the transmitter antenna to achieve optimal link quality. This section uses Phased Array System Toolbox™ to optimally steer a beam to maximize received power for a non-line-of-sight link.

Define a custom antenna from Report ITU-R M.2412 [1] on page 2-0 for evaluating 5G radio technologies. Create an 8-by-8 uniform rectangular array from the element pattern defined in Section 8.5 of the report, point it south, and view the radiation pattern.

```
tx.Antenna = helperM2412PhasedArray(tx.TransmitterFrequency);
tx.AntennaAngle = -90;

clearMap(viewer)
show(rx)
pattern(tx,"Transparency",0.6)
hide(tx)
```



Call `raytrace` with an output to access the rays that were computed. The returned `comm.Ray` objects include both the geometric and propagation-related characteristics of each ray.

```
rtPlusWeather.PropagationModels(1).MaxNumReflections = 1;
ray = raytrace(tx,rx,rtPlusWeather);
disp(ray{1})
```

```
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
           Interactions: [1×1 struct]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
                PathLoss: 117.4546
             PhaseShift: 3.8170

   Read-only properties:
       PropagationDelay: 6.6489e-07
    PropagationDistance: 199.3293
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
        NumInteractions: 1
```

Get the angle-of-departure for the single-reflection path and apply this angle to steer the antenna in the optimal direction to achieve higher received power. The angle-of-departure azimuth is offset by

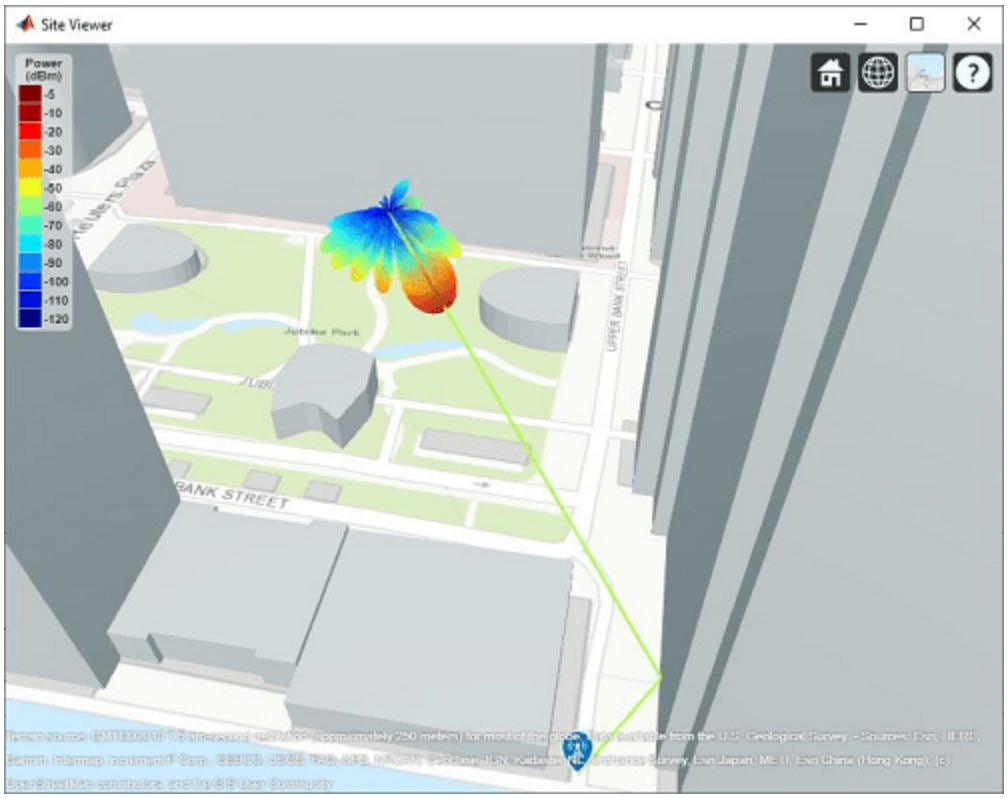
the physical antenna angle azimuth to convert it to the steering vector azimuth defined in the local coordinate system of the phased array antenna.

```
aod = ray{1}.AngleOfDeparture;
steeringaz = wrapTo180(aod(1)-tx.AntennaAngle(1));
steeringVector = phased.SteeringVector("SensorArray",tx.Antenna);
sv = steeringVector(tx.TransmitterFrequency,[steeringaz;aod(2)]);
tx.Antenna.Taper = conj(sv);
```

Plot the radiation pattern to show the antenna energy directed along the propagation path. The new received power increases by about 20 dB. The increased received power corresponds to the peak gain of the antenna.

```
pattern(tx,"Transparency",0.6)
raytrace(tx,rx,rtPlusWeather);
hide(tx)

ss = sigstrength(rx, tx, rtPlusWeather);
disp("Received power with beam steering: " + ss + " dBm")
```

```
Received power with beam steering: -57.5126 dBm
```



Conclusion

This example used ray tracing for link and coverage analysis in an urban environment. The analysis shows:

- How to use ray tracing analysis to predict signal strength for non-line-of-sight links where reflected propagation paths exist

- Analysis with realistic materials has a significant impact on the calculated path loss and received power
- Analysis with higher number of reflections results in increased computation time but reveals additional areas of signal propagation
- Usage of a directional antenna with beam steering significantly increases the received power for receivers, even if they are in non-line-of-sight locations

This example analyzed received power and path loss for links and coverage. To see how to use ray tracing to configure a channel model for link-level simulation, see the “Indoor MIMO-OFDM Communication Link Using Ray Tracing” on page 8-9 example.

References

[1] Report ITU-R M.2412, "Guidelines for evaluation of radio interface technologies for IMT-2020", 2017. https://www.itu.int/pub/R-REP-M.2412

See Also

Functions

`propagationModel` | `raytrace` | `coverage` | `contour` | `pattern`

Objects

`siteviewer` | `txsite` | `rxsite`

Related Examples

- “Ray Tracing for Wireless Communications” on page 30-12

Bluetooth Toolbox Examples

Evaluate the Performance of Bluetooth QoS Traffic Scheduling with WLAN Signal Interference

This example shows how to evaluate the performance of the Bluetooth® scheduler by implementing multiple applications with different quality-of-service (QoS) requirements (throughput and latency) in a use-case scenario. Using this example, you can:

- Create and configure a use-case scenario of a home environment showing multiple Bluetooth applications in a piconet with WLAN interference.
- Emulate and configure the application traffic pattern by using the generic On-Off traffic model.
- Implement round-robin (RR) and QoS-based priority schedulers to schedule application traffic.
- Add your own custom scheduler.
- Specify the source of WLAN interference by adding the WLAN signal using the features of WLAN Toolbox™ or from a baseband file.
- Evaluate the performance of each Slave in the presence of a synchronous connection-oriented (SCO) link and by varying the scheduler.

The example supports adaptive frequency hopping (AFH) by classifying channels as good or bad based on the packet error rate (PER) of each channel. Visualize the power spectral density of Bluetooth waveforms with a WLAN signal interference using the Spectrum Analyzer.

Bluetooth Logical Transports and Application Profiles

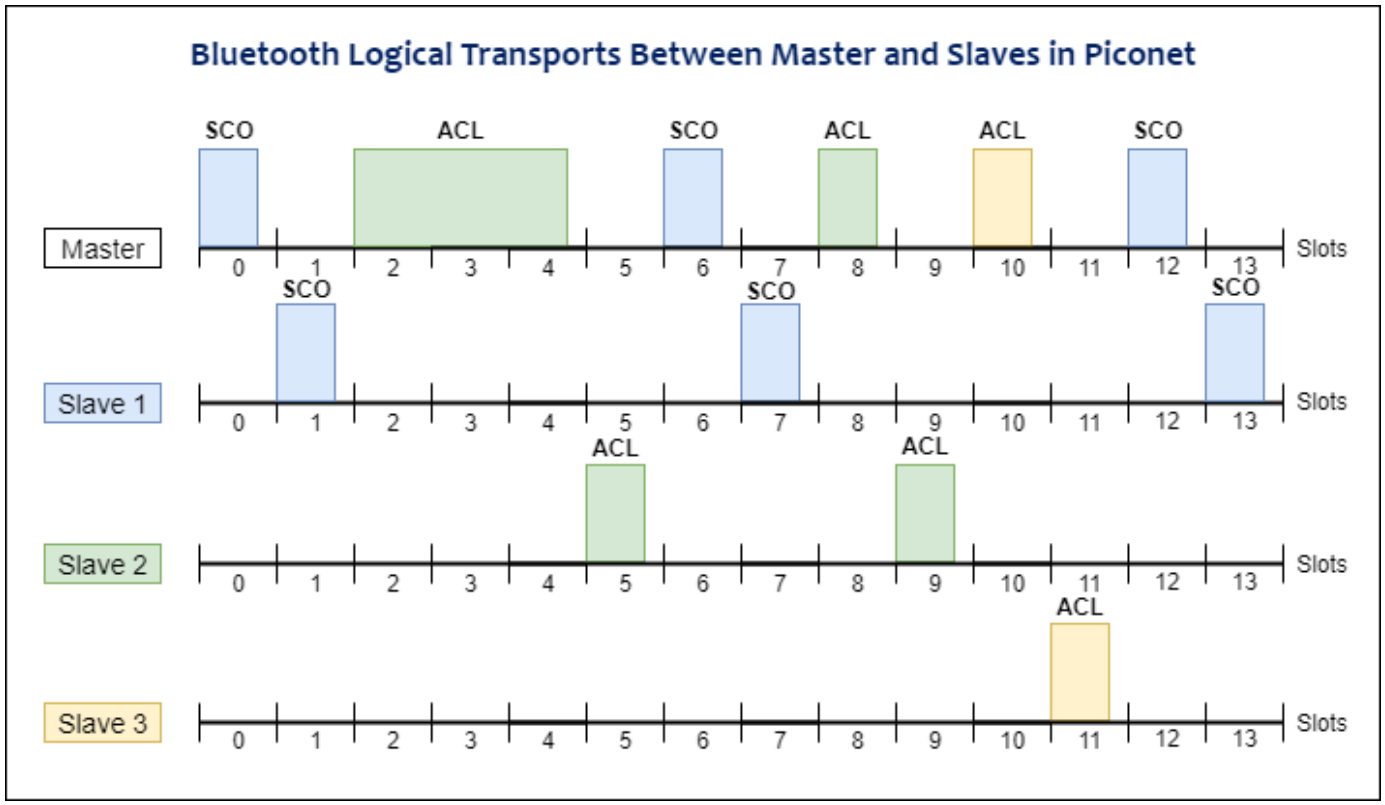
"What Is Bluetooth?" on page 13-2 supports communication over multiple logical transports with different applications running on it. These applications include audio streaming, gaming controls, wireless peripherals, and file transfer applications. Because multiple applications can exist in a Bluetooth piconet, different types of application traffic flow from the higher layers to the baseband.

Logical Transports

In a Bluetooth piconet, the Master and Slave exchange data over multiple logical transports. These logical transports are:

- Asynchronous connection-oriented (ACL)
- SCO
- Extended synchronous connection-oriented (eSCO)
- Active slave broadcast (ASB)
- Connectionless slave broadcast (CSB)

This figure shows the communication between a Master and three Slaves in a piconet over ACL and SCO logical transports. Because Bluetooth is a Master-driven time division duplex (TDD) system, the channel access in the piconet is controlled by the Master. The Slave can respond to only a transmission from the Master in the previous Tx slot. This process is called *polling*.



The Master polls a Slave with a poll packet (if no data exists) or a data packet in a Tx slot, and the Slave responds to the polling. The Master can poll any Slave of the SCO or ACL logical transport. For SCO links, the Master reserves the slots for the dedicated SCO Slave. The Master polls the ACL Slaves in the remaining slots. The Slave responds to the Master with a data packet or a null packet.

Application Profiles

Bluetooth profiles (often called application profiles) are definitions of possible applications and specify general behaviors that Bluetooth-enabled devices use to communicate with other Bluetooth devices. The Bluetooth Special Interest Group (SIG) [2] on page 3-0 defines these profiles and the possible applications of each profile.

In general, Bluetooth application traffic can be categorized into these three classes.

- Streaming - These applications have latency and bandwidth requirements. For example, a headphone or a laptop.
- Human interface devices (HID) - These applications have low latency requirements. For example, a keyboard or a joystick.
- Best-effort traffic - These applications have no latency requirements. For example, file transfer using Bluetooth.

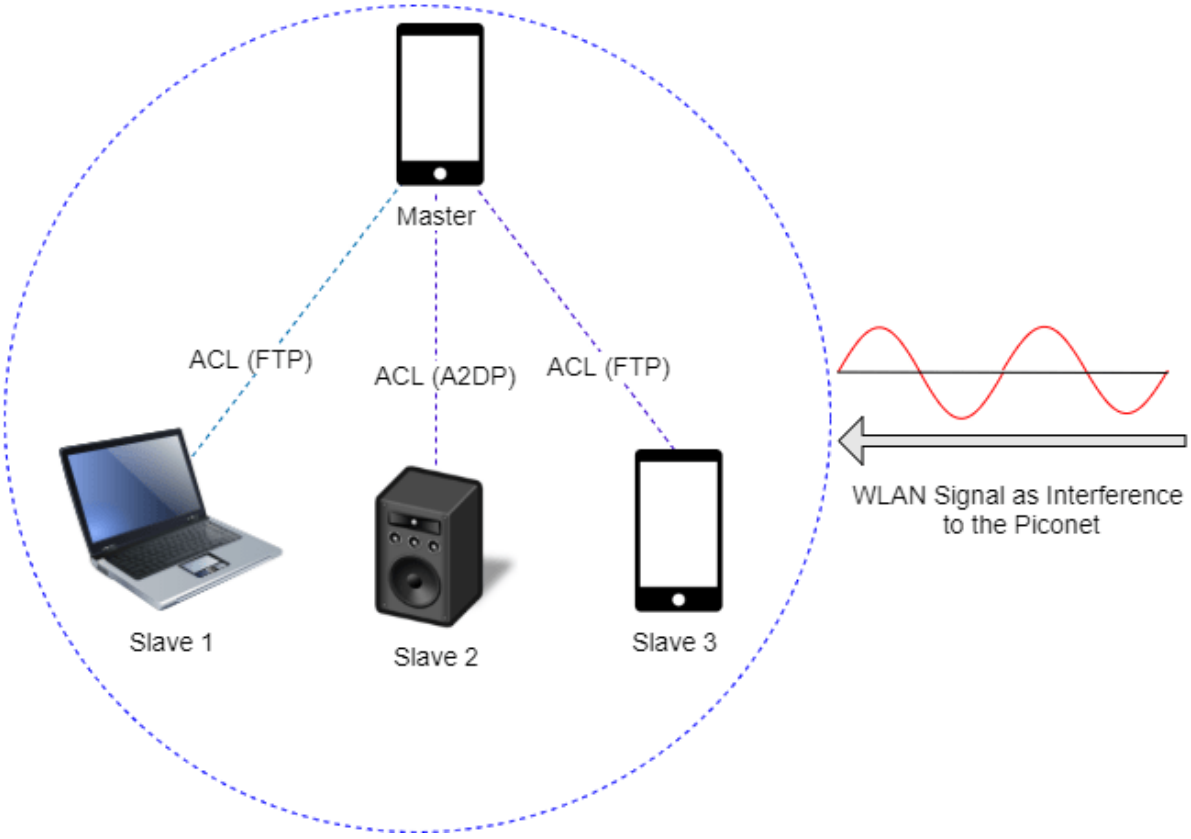
Each Slave corresponds to a specific application profile. Typically, the amount of traffic flow in each application profile varies. Also, the throughput and latency performance requirements of each application profile are different. In such scenarios, if the implementation uses an RR scheduler for polling the ACL Slaves, and if the polled Slaves do not have data to transmit, this results in low bandwidth utilization by Slaves. To help improve performance, prioritize the ACL Slaves for polling based on the QoS requirements.

In this example, the Bluetooth nodes operate with the basic rate (BR) physical layer (PHY) and communicate with each other simultaneously by transmitting the application data packets (on the ACL link). This example simulates a use-case scenario consisting of multiple Bluetooth devices that communicate by emulating the traffic pattern of FTP and A2DP application profiles. For more details, see Use-Case Scenario on page 3-0 .

Use-Case Scenario

This example shows a use-case scenario of a home environment, where a smartphone (Master) connects to a laptop (Slave 1), wireless speaker (Slave 2), and smartphone (Slave 3). All of the devices are Bluetooth-BR-enabled. You can enable or disable presence of static WLAN signal interference in the vicinity of the Bluetooth piconet.

Use Case Scenario Showing Multiple Bluetooth Applications in a Piconet With WLAN Signal Interference



In the preceding figure:

- The Master transfers a file to the laptop (Slave 1) on the ACL link by emulating the FTP traffic pattern.
- The Master streams music in the wireless speaker (Slave 2) on the ACL link by emulating the A2DP traffic pattern.

- The Master transfers another file smartphone (Slave 3) on the ACL link by emulating the FTP traffic pattern.

In the preceding scenario, the performance of each Slave degrades due to these reasons.

- Concurrent communication by various Bluetooth applications
- Presence of WLAN signal interference in the wireless medium (if enabled)

This example shows how to simulate this use-case scenario and how to measure the communication performance. To communicate with the laptop (Slave 1), wireless speaker (Slave 2), and smartphone (Slave 3) on the ACL links, the Master uses the RR scheduling mechanism. The RR scheduling mechanism provides equal transmission and reception opportunities for Slave 1, Slave 2, and Slave 3. Because, Slave 2 has QoS requirements (related to throughput and latency), Slave 2 must be prioritized. The RR scheduling mechanism fails to prioritize the communication opportunities of Slave 2, resulting in the degradation of communication performance. To mitigate the performance degradation at Slave 2 and help improve performance the example uses the QoS-based priority scheduling mechanism at the Master. The example also shows how to add a custom scheduling algorithm, enabling you to schedule application traffic with specific performance requirements.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed.
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

Configure the simulation parameters for the Bluetooth piconet, the application traffic, the wireless channel, and the WLAN signal interference.

Configure Bluetooth Piconet

The `NumSlaves` parameter specifies the number of Slaves in the Bluetooth piconet. The `LinkTraffic` parameter specifies the type of traffic over Bluetooth logical transports between a Master and the respective Slave. This table maps `LinkTraffic` to different logical transports. If the Master communicates with multiple Slaves, `LinkTraffic` must be a vector.

`linkTraffic` Value	Logical Transport
1	ACL
2	SCO
3	ACL and SCO

```
% Set the simulation time in microseconds
simulationTime = 2*1e6;

% Specify to enable or disable the visualization (true or false, respectively)
enableVisualization = true;

simulationParameters = struct;
```

```
% Configure the number of Slaves in the piconet
simulationParameters.NumSlaves = 3;

% Configure the logical links between the Master and Slaves. Each element
% represents the logical link between the Master and the respective Slave.
% If the Master is connected to multiple Slaves, this value must be a row
% vector.
simulationParameters.LinkTraffic = [1 1 1];

% Specify the positions of Bluetooth nodes in the form of an n-by-3 array,
% where n is the number of nodes in the piconet. Each row specifies the
% cartesian coordinates of a node starting from the Master and followed by
% the Slaves.
simulationParameters.NodePositions = [10 0 0; 20 0 0; 30 0 0; 40 0 0];

% Configure the frequency hopping sequence as Connection adaptive (for
% enabling AFH) or Connection basic
simulationParameters.SequenceType = Connection adaptive;
```

Configure Application Traffic Pattern

This example shows how to use a generic On-Off model for emulating the application traffic pattern. To emulate the application traffic pattern of the ACL Slaves (Slave 1, Slave 2, and Slave 3), use the `helperBluetoothNetworkTraffic`. To model the traffic pattern, specify the application token rate (bit rate), frame size, access latency, and the values for `OnTime` and `OffTime` properties of the object. If you specify SCO Slaves, model the application traffic pattern based on the packet type. For each transmission, the data (random bits) is made available at the baseband layer. In the majority of the scenarios involving ACL links, the traffic flow is present at the source, and the sink sends only the acknowledgement. In this example, all of the sources are present at the Master.

```
% Load the Bluetooth application traffic pattern configuration
load('bluetoothTrafficConfig.mat');

% Specify the parameters for SCO application. To enable an SCO logical
% transport, set the linkTraffic value at the Slave index to 2 or 3.
% Specify the SCO packet type as 'HV1', 'HV2', or 'HV3' for the respective
% Slave that has SCO link traffic. Index 1 represents the Slave number, and
% index 2 represents the SCO packet type to be used by the Slave.
simulationParameters.SCOPacketType = {};

% Compute the number of ACL Slaves
numACLSlaves = nnz(simulationParameters.LinkTraffic ~= 2);
aclApplications = repmat({bluetoothTrafficConfig}, 1, numACLSlaves);

% Update the configuration at the first ACL application (for example, a
% laptop). You can observe the variation in throughput based on the
% configured data rate and packet size.
aclApplications{1}.SlaveNumber = 1;                % Slave LT address
aclApplications{1}.TrafficPattern = 'FTP';         % FTP traffic pattern
aclApplications{1}.PacketSize = 152;               % In bytes
aclApplications{1}.DataRateKbps = 224;             % In Kbps
aclApplications{1}.ApplicationOnTime = 2;          % In milliseconds
aclApplications{1}.ApplicationOffTime = 0;         % In milliseconds
aclApplications{1}.AccessLatency = 500;            % In milliseconds
aclApplications{1}.Role = 'Master';                % Install the application at 'Master', 'Slave
```

```
% Update the configuration at the second ACL application (for example, a
% wireless speaker).
aclApplications{2}.SlaveNumber = 2;                 % Slave LT address
aclApplications{2}.TrafficPattern = 'A2DP';         % A2DP traffic pattern
aclApplications{2}.PacketSize = 328;                % In bytes
aclApplications{2}.DataRateKbps = 237;              % In Kbps
aclApplications{2}.ApplicationOnTime = 2;           % In milliseconds
aclApplications{2}.ApplicationOffTime = 0;          % In milliseconds
aclApplications{2}.AccessLatency = 60;              % In milliseconds
aclApplications{2}.Role = 'Master';                 % Install the application at 'Master', 'Slave

% Update the configuration at the third ACL application (for example, a
% smartphone).
aclApplications{3}.SlaveNumber = 3;                 % Slave LT address
aclApplications{3}.TrafficPattern = 'FTP';          % FTP traffic pattern
aclApplications{3}.PacketSize = 164;                % In bytes
aclApplications{3}.DataRateKbps = 172;              % In Kbps
aclApplications{3}.ApplicationOnTime = 2;           % In milliseconds
aclApplications{3}.ApplicationOffTime = 0;          % In milliseconds
aclApplications{3}.AccessLatency = 500;             % In milliseconds
aclApplications{3}.Role = 'Master';                 % Install the application at 'Master', 'Slave

% Configure the application traffic objects
for appIdx = 1:numACLSlaves
    appCfg = aclApplications{appIdx};
    app = helperBluetoothNetworkTraffic( ...
        'PacketSize',appCfg.PacketSize, ...
        'DataRate',appCfg.DataRateKbps, ...
        'OnTime',appCfg.ApplicationOnTime, ...
        'OffTime',appCfg.ApplicationOffTime);
    aclApplications{appIdx}.AppTrafficPattern = app;
end
simulationParameters.ACLApplications = aclApplications;
```

Configure Scheduler

Configure the scheduler to be used at the baseband layer. To configure the RR and priority scheduler, use the `helperBluetoothRRScheduler` and `helperBluetoothPriorityScheduler` helper objects, respectively. You can also add a custom scheduler at the baseband layer. For information about how to implement and integrate a custom scheduler, see Compare Performance of Slaves by varying Scheduling Algorithm on page 3-0 .

```
% Specify the scheduling scheme as 'RR' or 'Priority'
simulationParameters.Scheduler = Priority;
```

Configure Wireless Channel and WLAN Signal Interference

Configure the wireless channel by using the `helperBluetoothChannel` helper object and set the `SIR` value at each node. You can set the `EbNo` value for the AWGN channel. The AWGN is present throughout the simulation.

To generate the WLAN signal interference, use the `helperBluetoothGenerateWLANWaveform` helper function. Specify the sources of WLAN interference by using the `WLANInterference` parameter. Use one of these options to specify the source of the WLAN interference.

- `'Generated'` - To add a WLAN toolbox™ signal to interfere the communication between Bluetooth nodes, select this option. For details on how to add this signal, follow the steps shown in Add WLAN Signal Using WLAN Toolbox™ Features on page 3-0 .
- `'BasebandFile'` - To add a WLAN signal from a baseband file (`.bb`) to interfere the communication between Bluetooth nodes, select this option. You can specify the file name using the `WLANBBFilename` input argument. If you do not specify the `.bb` file, the example uses the default `.bb` file, `'WLANNonHTDSSS.bb'`, to add the WLAN signal.

The `'None'` option specifies that no WLAN signal is added to the Bluetooth signals, and the example uses this option by default.

```
% Configure wireless channel parameters
simulationParameters.EbNo = 22; % Ratio of energy per bit (Eb) to spectral noise density (No) in

% Configure the WLAN interference. Specify the WLAN interference as
% Generated, BasebandFile, or None. To use the wlanBBFilename property, set
% wlanInterference to BasebandFile.
simulationParameters.WLANInterference = None;
simulationParameters.WLANBBFilename =  'WLANNonHTDSSS.bb';

% Specify the signal to interference ratio, in dB, at each node. Specify
% this value as an n-element vector, where n is the number of nodes in the
% piconet, starting from the Master and followed by the Slaves. Each value
% indicates the SIR at a node.
simulationParameters.SIR = [-12 -6 -10 -8];
```

Create Bluetooth Piconet

Create a Bluetooth piconet of nodes with an L2CAP layer, baseband layer, PHY, and channel. To configure the Bluetooth piconet from the configured parameters, use the `helperBluetoothCreatePiconet` helper function.

```
% Set the random number generator with a default seed
rng('default');

% Specify the Tx power in dBm
simulationParameters.TxPower = 20;

% Specify the Bluetooth node receiver range (in meters).
simulationParameters.ReceiverRange = 40;

% Configure the channel classification parameters
simulationParameters.PERThreshold = 40;
simulationParameters.ClassificationInterval = 3000;
simulationParameters.RxStatusCount = 10;
simulationParameters.MinRxCountToClassify = 4;
simulationParameters.PreferredMinimumGoodChannels = 20;

% Set the total number of nodes in the piconet (one Master and multiple
% Slaves)
numNodes = simulationParameters.NumSlaves + 1;

% Configure the Bluetooth piconet
btNodes = helperBluetoothCreatePiconet(simulationParameters);
```

Visualize the Bluetooth waveforms by using the `dsp.SpectrumAnalyzer` System object™.

```
if enableVisualization
    btSpectrumAnalyzer = dsp.SpectrumAnalyzer( ...
        'Name','Bluetooth Full Duplex Communication', ...
        'ViewType','Spectrum and spectrogram', ...
        'TimeResolutionSource','Property', ...
        'TimeResolution',0.0005, ...
        'SampleRate',btNodes{1}.PHY.SamplesPerSymbol*1e6, ...
        'TimeSpanSource','Property', ...
        'TimeSpan',0.05, ...
        'FrequencyResolutionMethod','WindowLength', ...
        'WindowLength',512, ...
        'AxesLayout','Horizontal', ...
        'FrequencyOffset',2441*1e6, ...
        'ColorLimits',[-20 15]);
end
```

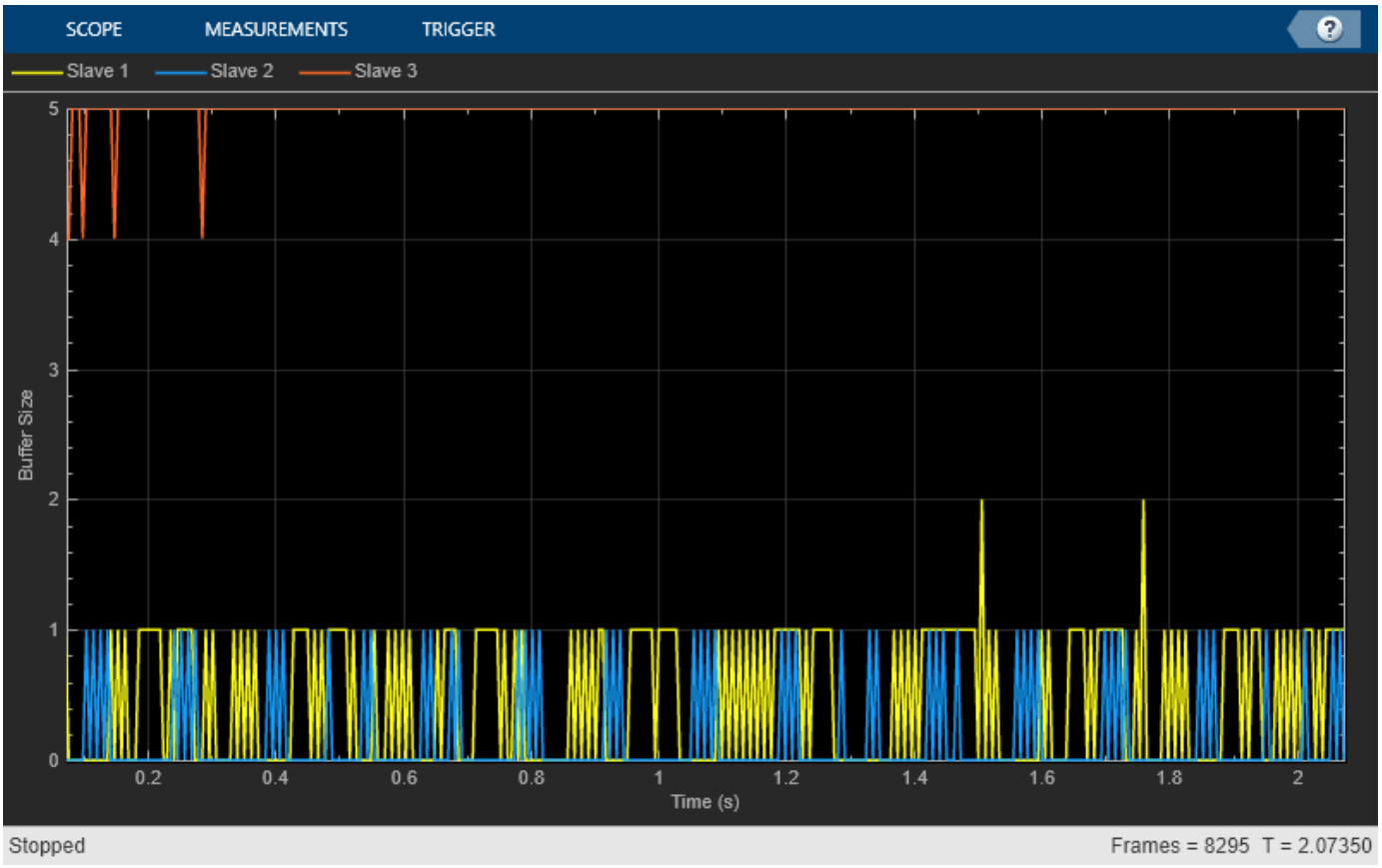
Simulation

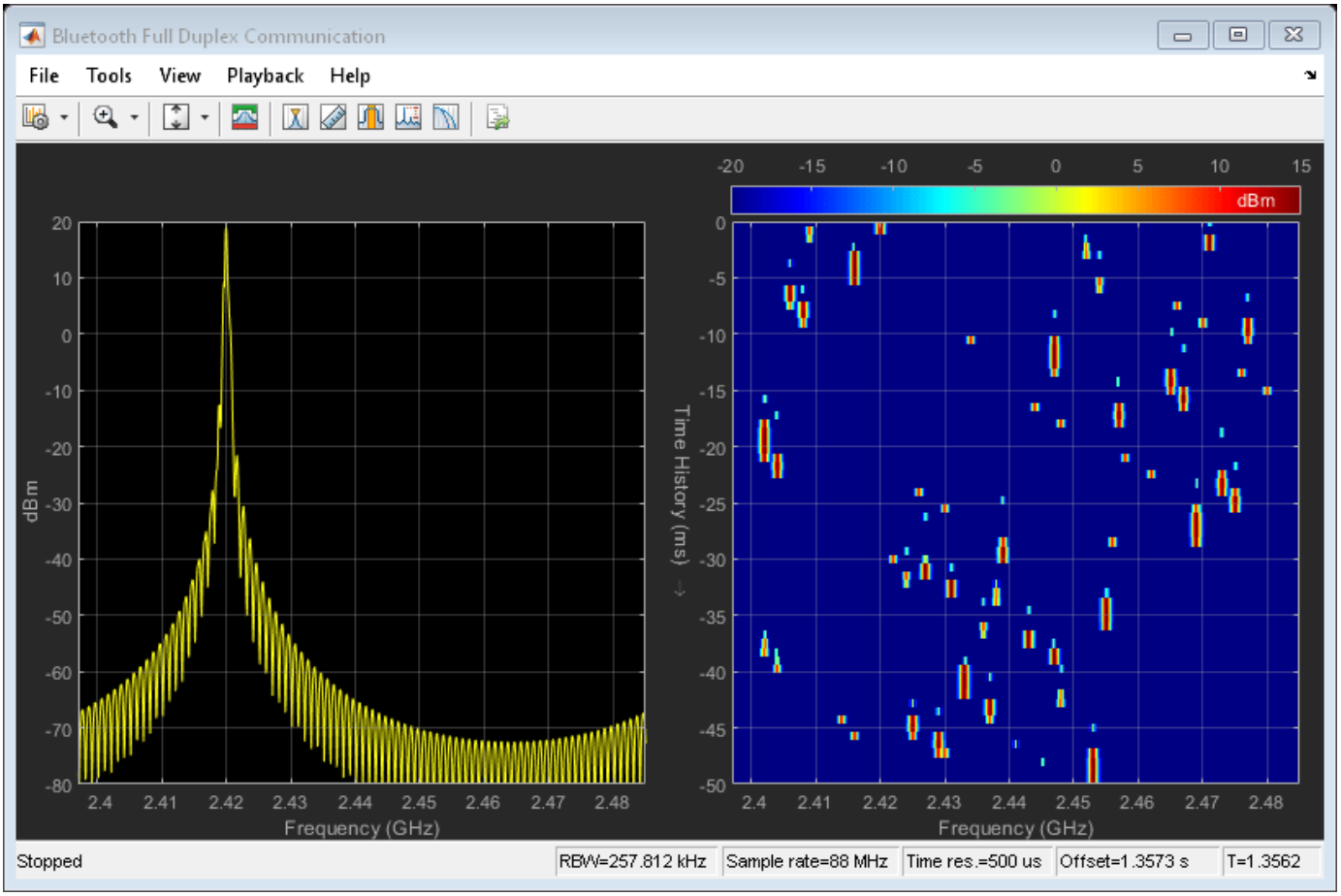
Run the Bluetooth piconet simulation by calling each node instance. Distribute the Tx packets from each node to the Rx buffer of the other nodes, and then advance the simulation time to the next event of a node. Update the Spectrum Analyzer visualization.

```
% Specify the current simulation time, elapsed time, and next invoke times
% for all of the nodes in microseconds
curTime = 0;
elapsedTime = 0;
nextInvokeTimes = zeros(1, numel(btNodes));
slotPerSec = 1600;

% View buffer size at each ACL Slave
if enableVisualization
    channelNames = cell(1, numACLSlaves);
    for idx=1:numACLSlaves
        channelNames{idx} = btNodes{idx+1}.NodeName;
    end
    bufferSizeVisualization = timescope( ...
        'Name','Buffer Size of each ACL Slave at Master', ...
        'SampleRate',slotPerSec*1.25*simulationTime/1e6, ...
        'AxesScaling','auto', ...
        'ShowLegend',true, ...
        'TimeSpanSource','property', ...
        'TimeSpan',simulationTime/1e6, ...
        'YLabel','Buffer Size', ...
        'YLimits',[0 5], ...
        'ChannelNames',channelNames);
    bufferSizeVisualization(zeros(1, numACLSlaves));
    if ~strcmpi(simulationParameters.WLANInterference, 'None')
        % Generate the WLAN waveform for visualization
        wlanWaveform = helperBluetoothGenerateWLANWaveform(...
            simulationParameters.WLANInterference, simulationParameters.WLANBBFilename);
    end
end

% Run the simulation
while(curTime < simulationTime)
    % Simulate the Bluetooth nodes
    for nodeIdx = 1:numel(btNodes)
        % Run the Bluetooth node instance
```

```
            nextInvokeTimes(nodeIdx) = runNode(btNodes{nodeIdx}, elapsedTime);

            % Update the visualization
            if enableVisualization
                % Update and plot the buffer status at the Master
                if nodeIdx == 1
                    [~, emptySlots] = btNodes{1}.Baseband.aclLinkAddress;
                    bufferSize = helperBluetoothLogicalTransports.MaxQueueSize - emptySlots;
                    bufferSizeVisualization(bufferSize);
                end
            end
        end

        % Advance the current time by the elapsed time
        curTime = curTime + elapsedTime;

        % Distribute any transmitted packets from each node into the receiving
        % buffers of the other nodes
        [isPacketDistributed, txBuffer] = helperBluetoothDistributePackets(btNodes);
        % Update visualization
        if enableVisualization
            for txIdx = 1:numel(txBuffer)
                channelWaveform = txBuffer{txIdx}.Waveform(1:txBuffer{txIdx}.NumSamples);
                if ~strcmpi(simulationParameters.WLANInterference, 'None')
                    % Add WLAN interference to the channel waveform.
                    channelWaveform = channelWaveform + wlanWaveform(1:txBuffer{txIdx}.NumSamples);
                end
                % Plot the Bluetooth waveform.
                btSpectrumAnalyzer(channelWaveform);
            end
        end

        % If packets are distributed to the receiver nodes, run the nodes to
        % check the packet reception buffer
        if isPacketDistributed
            elapsedTime = 0;
            % Advance the simulation time to the next event at a node
        else
            elapsedTime = min(nextInvokeTimes(nextInvokeTimes ~= -1));
        end
    end

ans = 1x79 logical array

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

ans = 1x79 logical array

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

ans = 1x79 logical array

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1


% Release the System objects
```

```
if enableVisualization
    release(bufferSizeVisualization);
    release(btSpectrumAnalyzer);
end
```





The preceding Spectrum Analyzer plot shows the spectrum of the Bluetooth waveform distorted with WLAN signal interference (in the frequency domain) and passed through the AWGN channel. The right-side plot shows the overlapping of Bluetooth packets with the interfering WLAN signal. The WLAN waveform in the plot is present throughout the simulation.

Simulation Results

At each node, the simulation measures these metrics.

- Throughput at application (L2CAP)
- PER
- Bit error rate (BER)
- Packet statistics at the PHY, baseband layer, and L2CAP layer

In addition to those metrics, at each Slave, the simulation measures these metrics.

- Latency (only for ACL Slaves)
- Fairness of scheduler

These metrics are generated during the simulation and stored in the `masterStats and slavesStats` table. For more information on these statistics, refer `helperBluetoothSchedulingStatistics`. Get the metrics of each Blutooth Slave in the piconet.

```
[masterStats, slavesStats] = helperBluetoothSchedulingStatistics(btNodes, simulationParameters.Li
    simulationTime)
```

```
masterStats=1×11 table
              Throughput (Kbps)    PER    BER    TotalTxPackets    TotalRxPackets    ValidRxPacke
              _________________    ___    ___    ______________    ______________    ____________

    Master         569.41           0      0          712               711               711


slavesStats=3×17 table
                    Traffic Pattern    Application Data Rate (Kbps)    Throughput (Kbps)    Later
                    _______________    ____________________________    _________________    _____

    Slave1 (ACL)        "FTP"                     224                     223.74              4
    Slave2 (ACL)        "A2DP"                    237                     237.47              4
    Slave3 (ACL)        "FTP"                     172                      104.3             61
```

Further Exploration

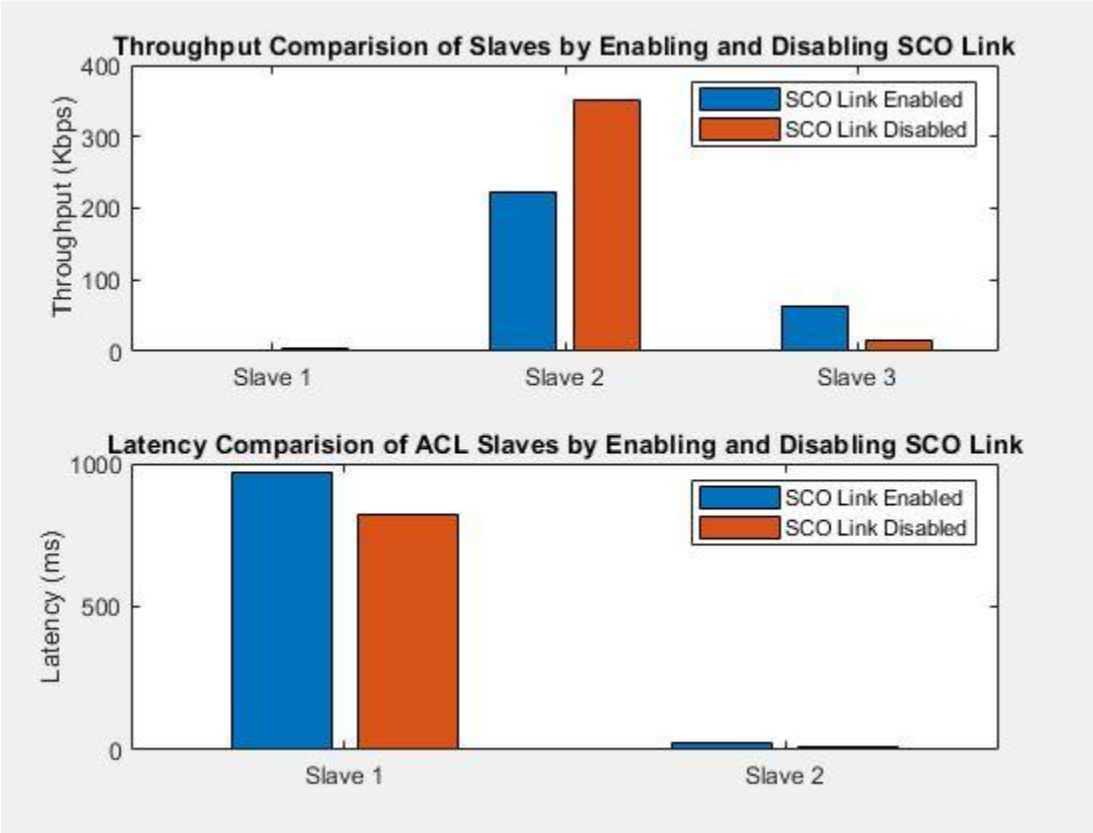
Because the presence of an SCO Slave in the piconet reserves the slots, the bandwidth for ACL Slaves decreases. Consequently, the presence of an SCO Slave results in the suspension of most of the ongoing ACL transmissions. In the default configuration, as the simulation uses the priority scheduler, wireless speaker (Slave 2) is prioritized over laptop (Slave 1) and smartphone (Slave 2). In this case, audio streaming has a better throughput and lower latency than the file transfer.

You can further evaluate and compare the performance of Slaves:

- In the presence of an SCO link
- By varying the scheduling algorithm

Compare Performance of Slaves in Presence of SCO Link

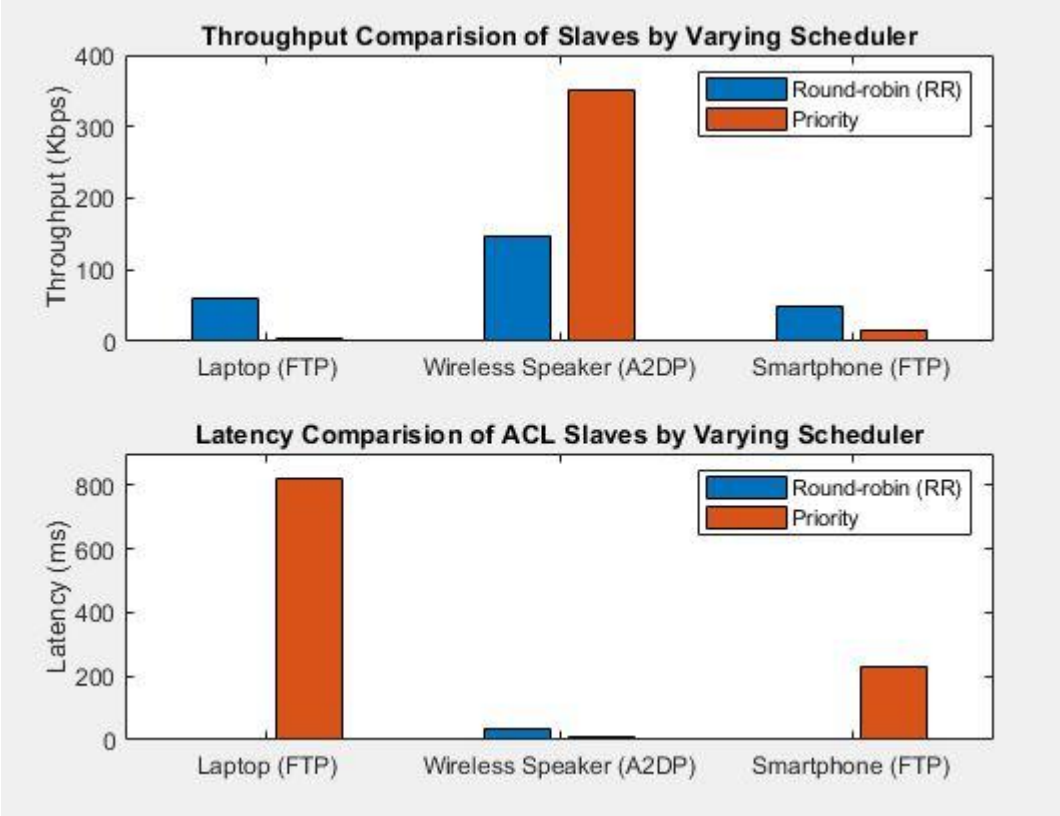
Simulate the scenario for 3 seconds with a QoS-based scheduler and with WLAN signal interference enabled. Compare the throughput and latency by enabling and then disabling the SCO link at Slave 3.



The presence of the SCO link degrades the throughput and latency performance of ACL Slaves. And also, the wireless speaker (Slave 2) cannot achieve the required throughput if an SCO Slave in the piconet exists, even if you use a QoS-based priority scheduler. Because acknowledgement for SCO packets is not required, the latency is calculated only for the ACL Slaves.

Compare Performance of Slaves by varying Scheduling Algorithm

Simulate the scenario for 3 seconds with the RR and QoS-based priority scheduler. Enable WLAN signal interference for both scenarios throughout the simulation.



For the RR scheduler, the throughput and latency values are approximately same for the laptop, wireless speaker, and smartphone. The QoS-based priority scheduler achieves the desired throughput and low latency of the wireless speaker (Slave 2).

You can use parameters such as PER, BER, level of WLAN interference, and channel map to create your own custom scheduler and evaluate its performance using this simulation. For more details on experimenting with channel classification, see “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51. Follow these steps to implement and integrate a custom Bluetooth scheduler.

1. Use the `helperBluetoothScheduler` object to create a new scheduler object.
2. Implement the algorithm by defining the `runScheduler` function. Modify the `helperBluetoothBaseband` object to update and retrieve the newly added parameters required by the algorithm.
3. Update the logic to attach the scheduler to the node in the `helperBluetoothCreatePiconet` function based on the algorithm.

Add WLAN Signal Using WLAN Toolbox™ Features

To add a WLAN signal using WLAN Toolbox™ features, set the value of `wlanInterference` to `Generated`. Use this code to add the generated WLAN signal as static signal interference to Bluetooth. Use this sample code snippet in WLAN signal generation using WLAN Toolbox™ features.

```
% % Create a WLAN waveform to interfere with Bluetooth waveforms,
% % which can be modified by using the features of the WLAN Toolbox.
% psduLength = 1000;
```

```
%
% % Create a configuration object for generating the WLAN waveform (802.11b)
% cfgNHT = wlanNonHTConfig('Modulation','DSSS', ...
%     'PSDULength', psduLength);
%
% % Create a random payload
% payload = randi([0 1], cfgNHT.PSDULength*8, 1);
%
% % Generate the WLAN waveform
% wlanWaveform = wlanWaveformGenerator(payload, cfgNHT);
```

You can add your custom signal generation code in the `helperBluetoothGenerateWLANWaveform` function. You can also write the respective signal WLAN spectrum masks and register to the `WLANSpectrum` property of the `helperBluetoothChannel` as a function pointer.

This example simulates a home environment use-case scenario and demonstrates how to schedule Bluetooth FTP and A2DP application traffic on ACL link. The example uses the RR and QoS-based priority scheduling algorithms to schedule the application traffic. The results show that the QoS-based priority scheduling algorithm prioritizes A2DP traffic and gives a better throughput and latency performance of the wireless speaker (Slave 2). The results from further explorations indicate that the performance of ACL Slaves decreases in the presence of SCO Slaves.

Appendix

The example uses these helpers:

- helperBluetoothNode: Configure and simulate Bluetooth node
- helperBluetoothBaseband: Configure and simulate Bluetooth baseband layer
- helperBluetoothLogicalTransports: Configure logical transports between Bluetooth nodes
- helperBluetoothSlotTimer: Manage Bluetooth clock and the timing of slots
- helperBluetoothPHY: Configure and simulate Bluetooth PHY layer
- helperBluetoothChannel: Configure and simulate Bluetooth wireless channel
- helperBluetoothGenerateWLANWaveform: Generate WLAN waveform to be added as an interference to Bluetooth waveforms
- helperBluetoothWLANDSSSSpectrumMask: Calculate adjacent channel interference power using the WLAN 802.11b (DSSS) spectrum masks
- helperBluetoothCreatePiconet: Create Bluetooth piconet using the Bluetooth nodes
- helperBluetoothSchedulingStatistics: Return statistics of each Bluetooth node in the Bluetooth piconet
- helperBluetoothDistributePackets: Distribute Tx packets in the piconet
- helperBluetoothL2CAP: Create and process Bluetooth L2CAP channels
- helperBluetoothL2CAPFrame: Generate Bluetooth L2CAP frame
- helperBluetoothL2CAPFrameDecode: Decode Bluetooth L2CAP frame
- helperBluetoothRRScheduler: Create object for Bluetooth Round-robin scheduler
- helperBluetoothPriorityScheduler: Create object for Bluetooth priority scheduler

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com.

2 Bluetooth Special Interest Group (SIG). "Traditional Profile Specifications." . https://www.bluetooth.com.

See Also

More About

- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96
- “Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH” on page 13-101
- “Packet Distribution in Bluetooth Piconet” on page 13-106
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76
- “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51

End-to-End Bluetooth BR/EDR PHY Simulation Using Path Loss Model, RF Impairments, and AWGN

This example uses Communications Toolbox™ Library for the Bluetooth® Protocol to perform end-to-end simulation for the Bluetooth basic rate/enhanced data rate (BR/EDR) physical layer (PHY) transmission modes in the presence of the path loss model, radio front-end (RF) impairments, and additive white Gaussian noise (AWGN). The simulation results show the estimated value of the bit error rate (BER), path loss, and the impact of path loss on the spectrum of the waveform.

Path Loss Modeling in Bluetooth BR/EDR Network

Bluetooth is a short-range Wireless Personal Area Network (WPAN) technology, operating in the globally unlicensed industrial, scientific, and medical (ISM) band in the frequency range of 2.4 GHz to 2.485 GHz.

The Bluetooth Core Specification [1 on page 3-0] specifies these PHY modes.

Basic rate (BR) - Mandatory mode, uses Gaussian frequency shift keying (GFSK) modulation with a data rate of 1 Mbps.

Enhanced data rate (EDR) - Optional mode, uses phase shift keying (PSK) modulation with these two variants:

- **EDR2M:** Uses pi/4-DQPSK with a data rate of 2 Mbps
- **EDR3M:** Uses 8-DPSK with a data rate of 3 Mbps

For more information about Bluetooth BR/EDR protocol stack, see “Bluetooth Protocol Stack” on page 13-7.

For more information about Bluetooth BR/EDR packet structures, see “Bluetooth Packet Structure” on page 13-23.

Path loss or path attenuation is the decline in the power density of a given signal as it propagates from the transmitter to receiver through space. This reduction in power density occurs naturally over the distance and is impacted by the obstacles present in the environment in which the signal is being transmitted. The path loss is generally expressed in decibels (dB) and is calculated as:

$\mathrm{PL}_{\mathrm{dB}} = P_t - P_r$.

In this equation,

- $\mathrm{PL}_{\mathrm{dB}}$ is the path loss in dB.
- P_t is the transmitted signal power in dB.
- P_r is the received signal power in dB.

Path loss models describe the signal attenuation between the transmitter and receiver based on the propagation distance and other parameters such as frequency, wavelength, path loss exponent, and antenna gains. The example considers these path loss models:

- Free-space [3] on page 3-0
- Log-distance [3] on page 3-0

- Log-normal shadowing [3] on page 3-0
- Two-ray ground reflection [3] on page 3-0
- NIST PAP 02-Task 6 [4] on page 3-0

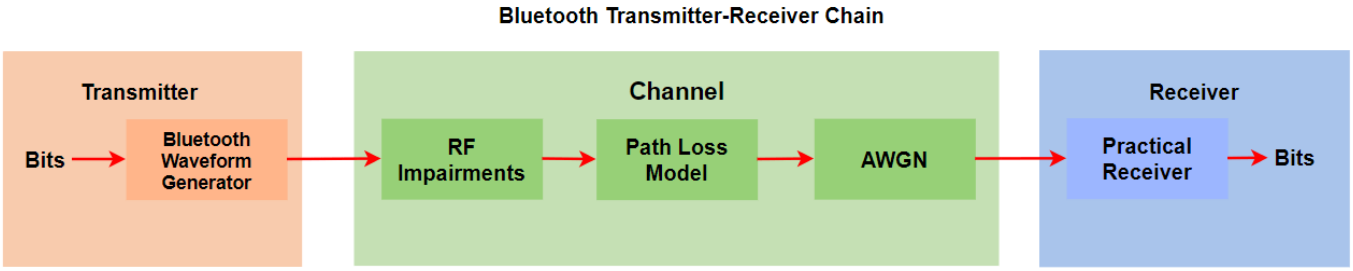
For more information about these path loss models, see “End-to-End BLE PHY Simulation Using Path Loss Model, RF Impairments, and AWGN” on page 3-90 example.

This example estimates the BER and the path loss between the transmitter and receiver by considering a specific path loss model with RF impairments and AWGN added to the transmission packets.

End-to-End Bluetooth BR/EDR Simulation Procedure

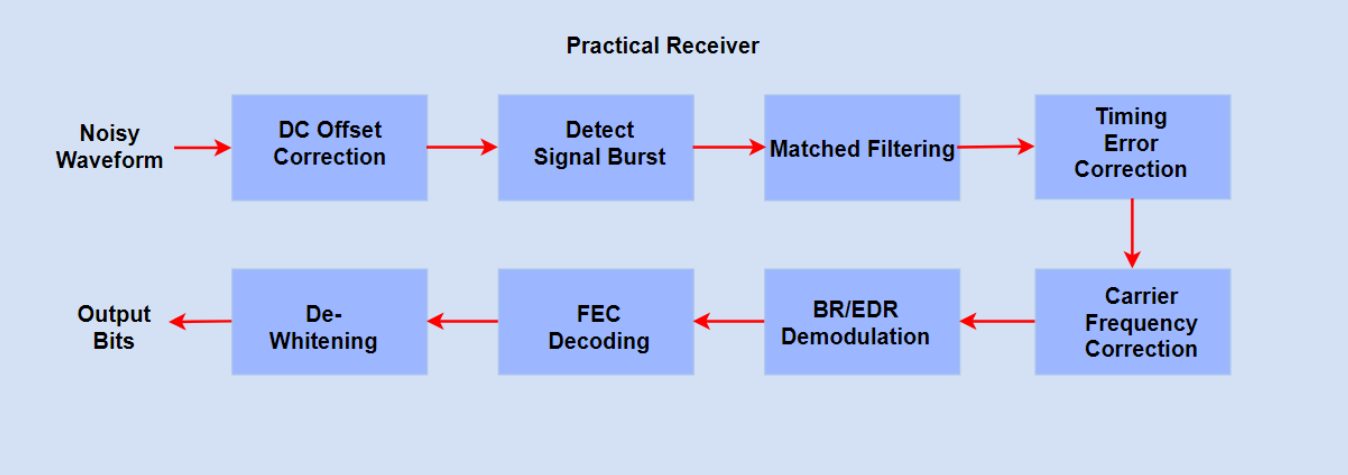
Specify the simulation parameters on page 3-0 . To perform the end-to-end simulation in the presence of path loss, implement these steps.

1. Generate random bits
2. Generate a Bluetooth BR/EDR waveform
3. Add impairments
4. Attenuate the waveform based on the path loss
5. Add AWGN
6. Display the spectrum of the transmitted and received waveforms



Pass the distorted and noisy waveforms through a practical receiver and perform these operations.

1. Remove DC offset
2. Detect the signal bursts
3. Perform matched filtering
4. Estimate and correct the timing offset
5. Estimate and correct the carrier frequency offset
6. Demodulate BR/EDR waveform
7. Perform forward error correction (FEC) decoding
8. Perform data dewhitening
9. Perform header error check (HEC) and cyclic redundancy check (CRC)
10. Outputs decoded bits



To estimate the bit error rate, compare the transmitted bits with the decoded bits.

Check for the Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed or not.

```
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

Specify the path loss model and the distance between the transmitter and receiver. Set the PHY transmission mode and the type of Bluetooth BR/EDR packet to be generated. Configure the RF impairments.

```
% Configure parameters related to the communication link between the transmitter and receiver

pathLossModel = Free Space; % Path loss model
distance = 5;               % Distance between transmitter and receiver, in meters

% Configure parameters for waveform generation

phyMode = BR;                % PHY transmission mode
bluetoothPacket = 'FHS';     % Type of Bluetooth BR/EDR packet. This value ca
                             % 'NULL','POLL','FHS','HV1','HV2','HV3','DV','EV
                             % 'EV4','EV5','AUX1','DM3','DM1','DH1','DM5','DH
                             % 'DH5','2-DH1','2-DH3','2-DH5','2-DH1','2-DH3',
                             %  '2-DH5','2-EV3','2-EV5','3-EV3','3-EV5'}
samplesPerSymbol = 8;        % Samples per symbol

% Configure RF impairments

frequencyOffset = 6000;% In Hz
timingOffset = 0.5;    % Offset within the symbol, in samples
timingDrift = 2;       % In parts per million
dcOffset = 2;          % Percentage with respect to maximum amplitude
EbNo = 25;             % Eb/No in dB
```

Generate Bluetooth BR/EDR Waveform

Generate Bluetooth BR/EDR waveform based on the physical layer transmission mode, packet type, and samples per symbol.

```
% Create a Bluetooth BR/EDR waveform configuration object
txCfg = bluetoothWaveformConfig('Mode',phyMode,'PacketType',bluetoothPacket,...
    'SamplesPerSymbol',samplesPerSymbol);
if strcmp(bluetoothPacket,'DM1')
    txCfg.PayloadLength = 17;       % Maximum length of DM1 packets in bytes
end
dataLen = getPayloadLength(txCfg);  % Length of the payload

% Generate Bluetooth BR/EDR waveform
bitsPerByte = 8;                              % Number of bits per byte
txBits = randi([0 1],dataLen*bitsPerByte,1); % Generate data bits
txWaveform = bluetoothWaveformGenerator(txBits,txCfg);
```

Add RF Impairments, Path Loss, and AWGN

Distort the generated Bluetooth BR/EDR waveform by adding the RF impairments.

```
% Create timing offset object
timingDelayObj = dsp.VariableFractionalDelay;

% Create frequency offset object
symbolRate = 1e6; % Symbol rate, in Hz
frequencyDelay = comm.PhaseFrequencyOffset('SampleRate',symbolRate*samplesPerSymbol);

% Add Frequency Offset
frequencyDelay.FrequencyOffset = frequencyOffset;
txWaveformCFO = frequencyDelay(txWaveform);

% Add Timing Delay
[packetDuration,~] = helperBluetoothPacketDuration(bluetoothPacket,phyMode,dataLen);
totalTimingDrift = zeros(length(txWaveform),1);
timingDriftRate = (timingDrift*1e-6)/(packetDuration*samplesPerSymbol);        % Timing drift ra
timingDriftVal = timingDriftRate*(0:1:((packetDuration*samplesPerSymbol)-1))'; % Timing drift
totalTimingDrift(1:(packetDuration*samplesPerSymbol)) = timingDriftVal;
timingDelay = (timingOffset*samplesPerSymbol)+totalTimingDrift;                % Static timing o
txWaveformTimingCFO = timingDelayObj(txWaveformCFO,timingDelay);

% Add DC Offset
dcValue = (dcOffset/100)*max(txWaveformTimingCFO);
txImpairedWaveform = txWaveformTimingCFO + dcValue;
```

To obtain the path loss value, use helperBluetoothEstimatePathLoss.m function. To attenuate the Bluetooth BR/EDR waveform, add the path loss value to it.

```
% Obtain the path loss value in dB
[plLinear,pldB] = helperBluetoothEstimatePathLoss(pathLossModel,distance);

% Attenuate Bluetooth BR/EDR waveform
txAttenWaveform  = txImpairedWaveform./plLinear;
```

Add AWGN to the attenuated Bluetooth BR/EDR waveform.

```
% Set code rate based on packet type
if any(strcmp(bluetoothPacket,{'FHS','DM1','DM3','DM5','HV2','DV','EV4'}))
```

```
    codeRate = 2/3;
elseif strcmp(bluetoothPacket,'HV1')
    codeRate = 1/3;
else
    codeRate = 1;
end

% Set number of bits per symbol based on the PHY transmission mode
bitsPerSymbol = 1+ (strcmp(phyMode,'EDR2M'))*1 + (strcmp(phyMode,'EDR3M'))*2;

% Get SNR from EbNo values
snr = EbNo + 10*log10(codeRate) + 10*log10(bitsPerSymbol) - 10*log10(samplesPerSymbol);

% Add AWGN
rxWaveform = awgn(txAttenWaveform,snr,'measured');
```

Receiver Processing

To retrieve the data bits, pass the attenuated, AWGN-distorted Bluetooth BR/EDR waveform through the practical receiver.

```
% Get PHY configuration properties
rxCfg = getPhyConfigProperties(txCfg);

% Receiver Module
[rxBits,~,~] = helperBluetoothPracticalReceiver(rxWaveform,rxCfg);
```

Simulation Results

Estimate BER based on the retrieved and transmitted bits.

```
% Calculate BER
ber = [];
if ((~any(strcmp(bluetoothPacket,{'ID','NULL','POLL'})))&&((length(txBits) == length(rxBits))))
    ber = (sum(xor(txBits,rxBits))/length(txBits));
    berDisplay = num2str(ber);
else % BER is not applicable either when packet is lost or when packet type is ID, NULL, POLL pac
    berDisplay = 'Not applicable';
end
```

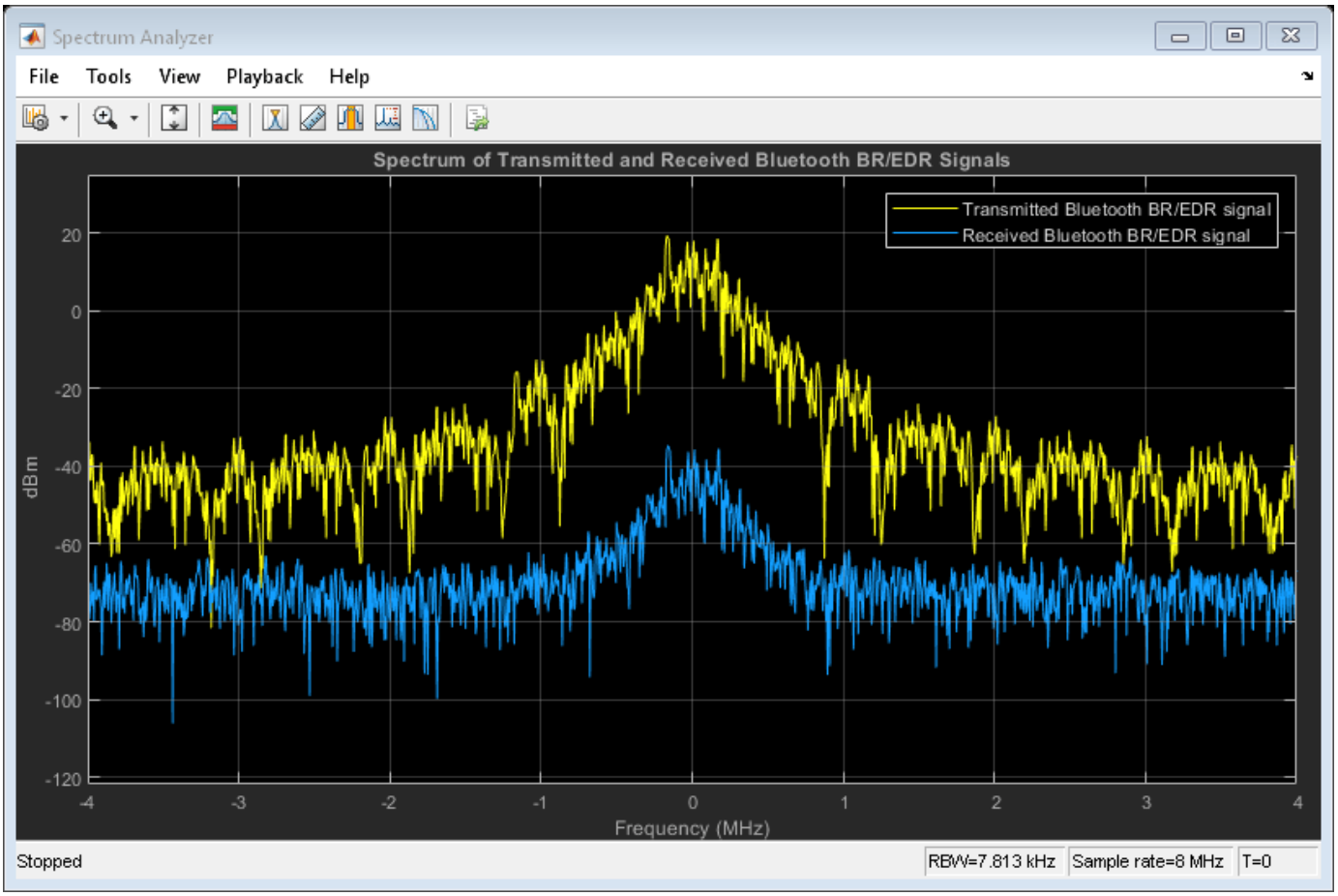
Display the estimated BER results. Plot the spectrum of the transmitted and received Bluetooth BR/EDR waveform.

```
% Display the estimated BER and distance between the transmitter and the receiver.
disp(['Input configuration: ', newline , '    PHY transmission mode: ', phyMode,....
    newline,'    Path loss model: ', pathLossModel, newline ,...
    '    Distance between the transmitter and receiver: ', num2str(distance), ' m', newline ,...
    '    Eb/No: ', num2str(EbNo), ' dB']);
```

```
Input configuration:
    PHY transmission mode: BR
    Path loss model: Free space
    Distance between the transmitter and receiver: 5 m
    Eb/No: 25 dB
```

```
disp(['Estimated outputs: ', newline , '    Path loss : ', num2str(pldB), ' dB'....
    newline, '    BER: ', berDisplay]);
```

```
Estimated outputs:
    Path loss : 54.0326 dB
    BER: 0

% Plot the spectrum of the transmitted and received Bluetooth BR/EDR waveform
specAnalyzer = dsp.SpectrumAnalyzer('NumInputPorts',2,'SampleRate',symbolRate*samplesPerSymbol,...
    'Title','Spectrum of Transmitted and Received Bluetooth BR/EDR Signals',...
    'ShowLegend',true,'ChannelNames',{'Transmitted Bluetooth BR/EDR signal','Received Bluetooth B
specAnalyzer(txWaveform(1:packetDuration*samplesPerSymbol),rxWaveform(1:packetDuration*samplesPer
release(specAnalyzer);
```



This example demonstrates an end-to-end Bluetooth BR/EDR simulation by considering the path loss model, distance between transmitter and receiver, RF impairments, and AWGN. The obtained simulation results display the estimated path loss and BER. The spectrum of the transmitted and received Bluetooth BR/EDR waveforms is visualized by using a spectrum analyzer.

Appendix

The example uses these helper functions:

- helperBluetoothEstimatePathLoss.m: Estimates the path loss between the transmitter and receiver based on the path loss model and the distance between the transmitter and receiver.
- helperBluetoothPracticalReceiver.m: Detects, synchronizes, and decodes the received Bluetooth BR/EDR waveform.

- helperBluetoothPacketDuration.m: Estimates the duration of a Bluetooth packet.

Selected Bibliography

[1] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.2. https://www.bluetooth.com.

[2] *Path Loss Models Used in Bluetooth Range Estimator*. Bluetooth Special Interest Group (SIG). https://www.bluetooth.com.

[3] Rappaport, Theodore. *Wireless Communication – Principles and Practice*. Prentice Hall, 1996.

[4] *NIST Smart Grid Interoperability Panel Priority Action Plan 2: Guidelines for Assessing Wireless Standards for Smart Grid Applications*. National Institute of Standards and Technology, U.S. Department of Commerce, 2014, https://nvlpubs.nist.gov/.

See Also

More About

- “End-to-End BLE PHY Simulation Using Path Loss Model, RF Impairments, and AWGN” on page 3-90

Bluetooth BR/EDR Power and Spectrum Test Measurements

This example shows how to perform radio frequency (RF) physical layer (PHY) transmitter tests specific to power and spectrum on Bluetooth® basic rate (BR) and enhanced data rate (EDR) transmitted waveforms by using Communications Toolbox™ Library for the Bluetooth Protocol features. The example also verifies whether these test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Objectives of Bluetooth RF-PHY tests

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by the Bluetooth Special Interest Group (SIG) includes RF-PHY tests for transmitters and receivers. The objectives of these RF-PHY tests are to:

- Ensure interoperability between all Bluetooth devices.
- Verify that a basic level of system performance is guaranteed for all Bluetooth products.

Each test case has a specific test procedure and an expected outcome, which must be met by the implementation under test (IUT).

Power and Spectrum Tests

This example shows how to perform power and spectrum test measurements on Bluetooth BR/EDR transmitted waveforms according to the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

- **Power Tests:** These tests verify whether the peak power, average power, power density, and power control of the transmitted Bluetooth signals are within the limits specified in the Bluetooth RF-PHY Test Specifications [1 on page 3-0]. For more information about these tests, see sections 4.5.1, 4.5.2, 4.5.3, 4.5.10, and 4.5.14 of the Bluetooth RF-PHY Test Specifications.
- **Spectrum Tests:** These tests verify whether the signal emissions are within the operating frequency range specified in the Bluetooth RF-PHY Test Specifications. For more information about these tests, see sections 4.5.4, 4.5.5, 4.5.6, and 4.5.13 of the Bluetooth RF-PHY Test Specifications.

This table shows various RF-PHY transmitter tests used in this example.

Conformance Tests	Test Case ID	Test Purpose	Required Packet Types	Center Frequency (MHz)
Power Tests	RF/TRM/CA/BV-01-C	Verifies the maximum peak and average RF-output power	'DH1', 'DH3', 'DH5', 'DM1', 'DM3', 'DM5'	2402, 2441, 2480
	RF/TRM/CA/BV-02-C	Verifies the maximum RF-power density	'DH1', 'DH3', 'DH5', 'DM1', 'DM3', 'DM5'	2441
	RF/TRM/CA/BV-03-C	Verifies the transmitter power control	'DH1'	2402, 2441, 2480
	RF/TRM/CA/BV-10-C	Verifies whether the difference in average transmit power during Gaussian frequency shift keying (GFSK) modulated and differential phase shift keying (DPSK) modulated portions of a packet is within an acceptable range	For pi/4-DQPSK, '2-DHx', '2-EVx' For 8DPSK, '3-DHx', '3-EVx' Where x = 1,3,5	2402, 2441, 2480
	RF/TRM/CA/BV-14-C	Verifies the transmitter power control	'2-DH1', '3-DH1'	2402, 2441, 2480
Spectrum Tests	RF/TRM/CA/BV-04-C	Verifies whether the emissions are within the limits of the operating frequency range	'DH1', 'DH3', 'DH5', 'DM1', 'DM3', 'DM5'	2402, 2480
	RF/TRM/CA/BV-05-C	Verifies whether the emissions for 20 dB bandwidth are within the limits	'DH1', 'DH3', 'DH5', 'DM1', 'DM3', 'DM5'	2402, 2441, 2480
	RF/TRM/CA/BV-06-C	Verifies whether the adjacent channel power emissions are within the limits specified	'DH-1'	2402, 2441, 2480
	RF/TRM/CA/BV-13-C	Verifies whether the level of unwanted signals from the DPSK transmitter are within the range	'2-DH1', '2-DH3', '2-DH5', '2-EV3', '2-EV5'	2402, 2441, 2480

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed.
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

Specify the test case ID, center frequency, packet type, samples per symbol, and output power.

```
% Select test case ID to perform power and spectrum tests
testCaseID = RF/TRM/CA/BV-01-C ;

% Select the frequency of operation for the IUT as shown in this table
%  ---------------------------
%  | Operating | Frequency in |
%  | Frequency |    MHz       |
%  ---------------------------
%  | Low       |    2402      |
%  | Mid       |    2440      |
%  | High      |    2480      |
%  ---------------------------
% Specify the type of center frequency required to perform the test case
%  -----------------------------------------------------
%  |    Test Case ID    |   Type of Center Frequency   |
%  -----------------------------------------------------
%  |                    |                              |
%  | RF/TRM/CA/BV-01-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-02-C  |            'Mid'             |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-03-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-04-C  |         'Low','High'         |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-05-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-06-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-10-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-13-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  | RF/TRM/CA/BV-14-C  |      'Low','Mid','High'      |
%  |                    |                              |
%  |                    |                              |
%  -----------------------------------------------------
centerFrequency = 'Low';

% Specify the type of packet required to perform the test case
packetType = 'DH1';
sps = 8;                                % Number of samples per symbol

outputPower =  20                       ; % Output power in dBm (must be in the range [-20,20])
```

```
stepSize = 2 ;      % Step size in dB (for power control tests
                    % {'RF/TRM/CA/BV-03-C',
                    % 'RF/TRM/CA/BV-14-C'})
```

Configure Test Parameters and Generate Bluetooth Test Waveform

Configure the test parameters and generate the Bluetooth test waveform, by using the helperBluetoothPowerTestConfig.m helper function.

```
% Get test parameters and test waveform
[configParams,txWaveformBaseBand] = helperBluetoothPowerTestConfig(testCaseID,centerFrequency,pa

% Interpolation factor for upconversion
interpFactor = ceil(2*configParams.StopFreq/configParams.SampleRate);

% Create a digital upconverter System object
upConv = dsp.DigitalUpConverter( ...
    'InterpolationFactor',interpFactor, ...
    'SampleRate',configParams.SampleRate, ...
    'Bandwidth',6e6, ...
    'CenterFrequency',configParams.CenterFreq);
dBmConvFactor = 30;
scalingFactor = 10^((outputPower-dBmConvFactor)/20);

% Upconvert the baseband waveform to passband and scale the waveform to
% required power
txWaveform = scalingFactor*upConv(txWaveformBaseBand);
```

Perform Spectrum Analysis on the Waveform

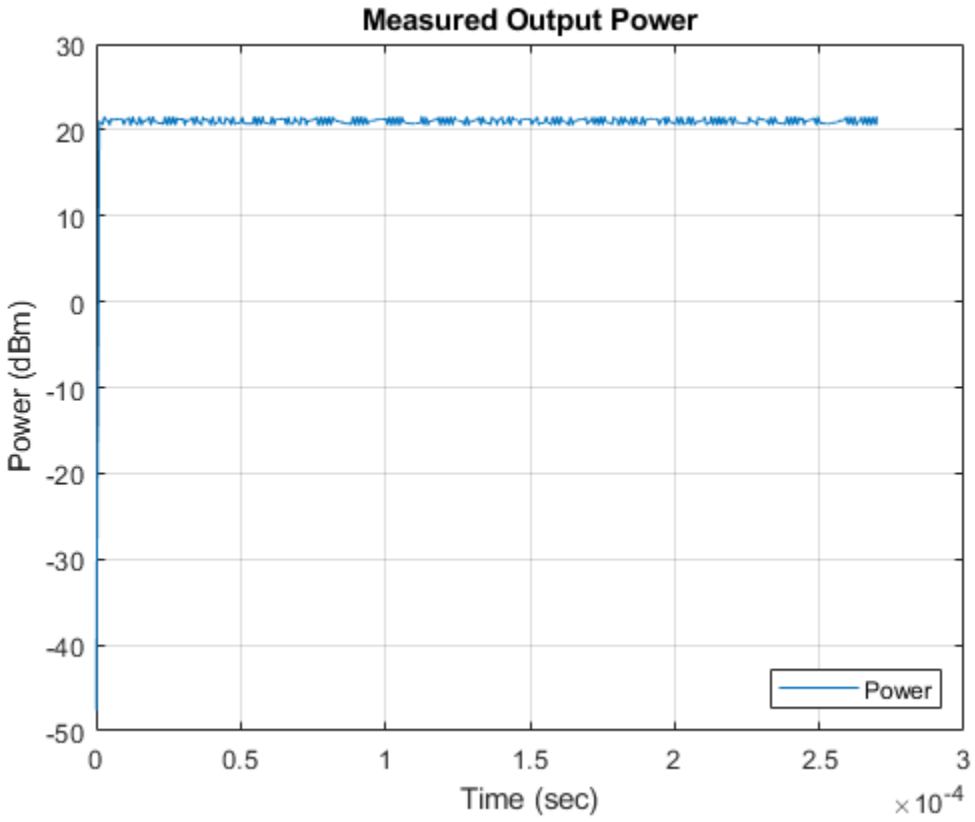
Perform the spectrum analysis using the helperBluetoothPowerTestAnalysis.m helper function. The function returns the output power along with the frequency, time, or step size data from the spectrum.

```
[outPower,testMeas] = helperBluetoothPowerTestAnalysis(testCaseID,configParams,txWaveform,interpF
```

Calculate Test Measurements

Calculate the test measurements for each test based on the spectrum analysis outputs by using the helperBluetoothPowerTestMeasurements.m helper function.

```
output = helperBluetoothPowerTestMeasurements(testCaseID,centerFrequency,outPower,testMeas);
```



Simulation Results

Validate the test results and display the verdict using the helperBluetoothPowerTestVerdict.m helper function.

```
helperBluetoothPowerTestVerdict(testCaseID,output);
```

```
Measured average power - 21.043922 dBm
Measured peak power - 21.546605 dBm
Verdict - Output power test passed
```

This example demonstrates the Bluetooth BR/EDR RF-PHY transmitter test measurements specific to power and spectrum. The simulation results verify that the computed test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

The example uses these helpers:

- helperBluetoothPowerTestConfig.m: Configure test parameters and generate Bluetooth test waveform
- helperBluetoothPowerTestAnalysis.m: Perform analysis of Bluetooth test waveforms in time and frequency domain
- helperBluetoothPowerTestMeasurements.m: Calculate test measurements
- helperBluetoothPowerTestVerdict.m: Validate test results and displays the verdict

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). “Bluetooth RF-PHY Test Specifications”, v1.2/2.0/2.0, EDR/2.1/2.1, EDR/3.0/3.0, HS (), RF.TS/3.0.H.1, Section 4.5. 2009. https://www.bluetooth.com.

2 Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". Bluetooth Core Specification. Version 5.2, Volume 2. https://www.bluetooth.com.

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151
- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62
- “Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift” on page 3-71

BLE RF-PHY Receiver Tests for IQC and IQDR

This example shows how to perform Bluetooth® low energy (BLE) radio frequency (RF) physical layer (PHY) receiver tests specific to in-phase quadrature samples coherency (IQC) and IQ samples dynamic range (IQDR) by using Communications Toolbox™ Library for the Bluetooth Protocol. The tests compute relative phase, reference phase deviation, and amplitudes of IQ samples at each antenna in an antenna array. This example also verifies whether these test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specification [1 on page 3-0].

Objectives of Bluetooth RF-PHY Tests

The Bluetooth RF-PHY Test Specification [1 on page 3-0] defined by the Bluetooth Special Interest Group (SIG) includes RF-PHY tests for transmitter and receiver. The objectives of these RF-PHY tests are to:

- Ensure interoperability between all Bluetooth devices
- Ensure a basic level of system performance for all Bluetooth products

Each test case has a specific test procedure and an expected outcome, which must be met by the implementation under test (IUT).

IQC and IQDR Tests

The Bluetooth Core Specification 5.1 [2 on page 3-0] introduced angle of arrival (AoA) and angle of departure (AoD) direction finding features. For more information about direction finding services in BLE, see “Bluetooth Low Energy Based Positioning Using Direction Finding” on page 3-38 and “Bluetooth Location and Direction Finding” on page 13-37. The Bluetooth RF-PHY Test Specification [1 on page 3-0] specifies the tests for direction finding transmitted waveforms with constant tone extension (CTE). This example includes AoA and AoD receiver tests specific to IQC and IQDR.

- **IQ sample coherency**: This test verifies the relative phase and reference phase deviation values derived from the I and Q values sampled on AoA or AoD receiver.
- **IQ sample dynamic range**: This test verifies the I and Q values sampled on AoA or AoD receiver by varying the dynamic range of the CTE.

This table shows various RF-PHY AoA and AoD receiver tests performed in this example.

Conformance Test	Test Case ID	PHY Mode	CTE Type	Number of Antenna Elements
IQ Samples Coherency	RF-PHY/RCV/IQC/BV-01-C	LE1M	[0;1] (2 µs)	4
	RF-PHY/RCV/IQC/BV-02-C	LE1M	[1;0] (1 µs)	
	RF-PHY/RCV/IQC/BV-03-C	LE2M	[0;1] (2 µs)	
	RF-PHY/RCV/IQC/BV-04-C	LE2M	[1;0] (1 µs)	
	RF-PHY/RCV/IQC/BV-05-C	LE1M	[0;0] (2 µs)	2 to 4
	RF-PHY/RCV/IQC/BV-06-C	LE2M	[0;0] (2 µs)	
IQ Samples Dynamic Range	RF-PHY/RCV/IQDR/BV-07-C	LE1M	[0;1] (2 µs)	4
	RF-PHY/RCV/IQDR/BV-08-C	LE1M	[1;0] (1 µs)	
	RF-PHY/RCV/IQDR/BV-09-C	LE2M	[0;1] (2 µs)	
	RF-PHY/RCV/IQDR/BV-10-C	LE2M	[1;0] (1 µs)	
	RF-PHY/RCV/IQDR/BV-11-C	LE1M	[0;0] (2 µs)	2 to 4
	RF-PHY/RCV/IQDR/BV-12-C	LE2M	[0;0] (2 µs)	

Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

Specify the receiver test ID, array size, samples per symbol, and element spacing between the antenna elements.

```
rxTestID = RF-PHY/RCV/IQC/...;  % Receiver test case ID
arraySize = 2; % Array size, must be 4 or [2 2] for AoD receiver tests and 2
               % for AoA receiver tests
sps = 8;               % Samples per symbol
elementSpacing = 0.5;  % Normalized spacing between the antenna elements with respect
```

Generate RF-PHY Test Parameters

Generate test parameters based on the receiver test ID, array size, and samples per symbol. To generate the PHY mode, CTE type, slot duration, test switching pattern, number of packets to

transmit and input power to the receiver, use the helperBLEIQCIQDRTestConfig function. Create and configure `comm.ThermalNoise` System object™ to add thermal noise.

```
[phyMode,cteType,slotDuration,switchingPattern,numPackets,rxPower] = ...
    helperBLEIQCIQDRTestConfig(rxTestID,arraySize,sps);

% The CTEInfo field position is same for the LE test packet and data
% packet, so consider dfPacketType as ConnectionCTE for CTE based RF-PHY
% tests
dfPacketType = 'ConnectionCTE';

% Create and configure BLE angle estimation configuration object
cfg = bleAngleEstimateConfig('ArraySize',arraySize,'SlotDuration',slotDuration, ...
                     'SwitchingPattern',switchingPattern,'ElementSpacing',elementSpacing);
numElements = getNumElements(cfg); % Number of elements in the array

% Create a thermal noise System object
NF = 12; % Noise figure (dB)
symRate = 1e6 + 1e6*(strcmp(phyMode,'LE2M')); % Symbol rate in Hz based on PHY transmission mode
sampleRate = symRate*sps; % Sampling rate in Hz
thNoise = comm.ThermalNoise('NoiseMethod','Noise figure', ...
                            'SampleRate',sampleRate, ...
                            'NoiseFigure',NF);
```

Simulate IQ Coherency or Dynamic Range Tests

To simulate the IQC and IQDR tests, perform these steps.

1. Generate BLE test packet waveform.
2. Perform waveform steering and antenna switching.
3. Add thermal noise.
4. Perform demodulation, decoding, and IQ sampling on the noisy waveform.
5. Perform IQC and IQDR test measurements.

Based on the receiver test, the helperBLEIQCIQDRTest function returns these values.

Tests	Test Results
• RF-PHY/RCV/IQC/BV-01-C • RF-PHY/RCV/IQC/BV-02-C • RF-PHY/RCV/IQC/BV-03-C • RF-PHY/RCV/IQC/BV-04-C • RF-PHY/RCV/IQC/BV-05-C • RF-PHY/RCV/IQC/BV-06-C	Relative phase (RP), mean relative phase (MRP), summation in the formula for mean relative phase, reference phase deviation (RPD), mean reference phase deviation (MRPD), summation in the formula for mean reference phase deviation
• RF-PHY/RCV/IQDR/BV-07-C • RF-PHY/RCV/IQDR/BV-08-C • RF-PHY/RCV/IQDR/BV-09-C • RF-PHY/RCV/IQDR/BV-10-C • RF-PHY/RCV/IQDR/BV-11-C • RF-PHY/RCV/IQDR/BV-12-C	Amplitudes of IQ samples for each antenna

```
% Initialize the number of outputs based on the receiver test ID
numOutputs = 2 + 4*any(strcmp(rxTestID,{'RF-PHY/RCV/IQC/BV-01-C','RF-PHY/RCV/IQC/BV-02-C', ...
                'RF-PHY/RCV/IQC/BV-03-C','RF-PHY/RCV/IQC/BV-04-C','RF-PHY/RCV/IQC/BV-05-C', ...
                'RF-PHY/RCV/IQC/BV-06-C'}));
[iqcIQDROutputs,iqcIQDROutputsConc] = deal(cell(1,numOutputs));
% Generate BLE test waveform
payloadLength = 0; % Empty payload for the considered receiver test IDs
cteLength = 160; % CTE length must be 160 microseconds for the considered receiver test IDs
payloadType = randsrc(1,1,1:7); % Payload type can be any value as the payload length is zero
bleWaveform = helperBLETestWaveform(payloadType,payloadLength,sps,phyMode,cteLength,cteType);

% Loop over the number of packets
for i = 1:numPackets
    % Generate random angle(s) between -90 to 90 degrees
    angles = randsrc(2,1,-90:90);

    % Perform steering and switching between the antennas
    dfWaveform = helperBLESwitchAntenna(bleWaveform,angles, ...
        phyMode,sps,dfPacketType,payloadLength,cteLength,cfg);

    % Attenuate the waveform according to the given received power
    dfWaveformAtt = dfWaveform.*10.^(rxPower/20);

    % Add thermal noise to the waveform
    noisyWaveform = thNoise(dfWaveformAtt);

    % Pass the noisy waveform to the BLE ideal receiver and get the IQ
    % samples
    [~,~,iqSamples] = bleIdealReceiver(noisyWaveform,'Mode',phyMode, ...
                'SamplesPerSymbol',sps,'DFPacketType',dfPacketType, ...
                'SlotDuration',slotDuration,'WhitenStatus','Off');

    % Perform IQC and IQDR test measurements based on the receiver test ID
    [iqcIQDROutputs{:}] = helperBLEIQCIQDRTest(rxTestID,iqSamples,numElements);

    % Concatenate the outputs over the number of packets
```

```
    for j = 1:numOutputs
        iqcIQDROutputsConc{j} = [iqcIQDROutputsConc{j}; iqcIQDROutputs{j}];
    end
end
```

Test Verdict

Verify whether the IQC and IQDR test measurements are within the specified limits and display the test verdict.

```
if any(strcmp(rxTestID,{'RF-PHY/RCV/IQC/BV-01-C','RF-PHY/RCV/IQC/BV-02-C', ...
        'RF-PHY/RCV/IQC/BV-03-C','RF-PHY/RCV/IQC/BV-04-C', ...
        'RF-PHY/RCV/IQC/BV-05-C','RF-PHY/RCV/IQC/BV-06-C'})) % IQC test
    % For each nonreference antenna, Am, where m is in the range [0, number
    % of antenna elements-1], used in the switching pattern, the results of
    % the summations in the formulae for MRP(m) and MRPD must be nonzero
    disp ('Expected summations in the formulae for MRP(m) and MRPD must be non-zero.');
    if all(all(iqcIQDROutputsConc{2}~=0)) && all(iqcIQDROutputsConc{5}~=0)
        disp('Result: Pass');
    else
        disp('Result: Fail');
    end

    % For each nonreference antenna, Am, used in the switching pattern, 95%
    % of the values, v, in the set must be -0.52<=principal(v-MRP(m))<=0.52
    mrpRep = kron(iqcIQDROutputsConc{3},ones(length(iqcIQDROutputsConc{1})/length(iqcIQDROutputsC
    subMRP = iqcIQDROutputsConc{1} - mrpRep;
    if size(subMRP,2) == 3 && any(strcmp(rxTestID,{'RF-PHY/RCV/IQC/BV-01-C',...
        'RF-PHY/RCV/IQC/BV-03-C','RF-PHY/RCV/IQC/BV-05-C','RF-PHY/RCV/IQC/BV-06-C'}))
        subMRP(3:3:end,3) = 0;
    end
    subMRPPrincipal = helperBLEPrincipalAngle(subMRP);
    subMRPRange = sum(subMRPPrincipal<=0.52 & subMRPPrincipal>=-0.52);
    disp ('Expected 95% of the values v in the set RP(m) must meet -0.52<=principal(v-MRP(m))<=0
    if all(subMRPRange>0.95*length(subMRPPrincipal))
        disp('Result: Pass');
    else
        disp('Result: Fail');
    end

    % MRPD must be in the range -1.125 to 1.125
    disp ('Expected MRPD in the range [-1.125, 1.125] radians.');
    if all(iqcIQDROutputsConc{6}<=1.125) && all(iqcIQDROutputsConc{6}>=-1.125)
        disp('Result: Pass');
    else
        disp('Result: Fail');
    end
else  % IQDR test

    % The mean of amplitudes of IQ samples measured at each antenna follows
    % the equation mean(ANT3)<mean(ANT2)<mean(ANT0)<mean(ANT1)
    meanA1 = mean(iqcIQDROutputsConc{1});
    meanA = mean(iqcIQDROutputsConc{2});
    if length(meanA) == 1
        disp('The mean of amplitudes must follow mean(ANT0)<mean(ANT1).');
        conditionCheck = meanA1<meanA(1);
    elseif length(meanA) == 2
        disp('The mean of amplitudes must follow mean(ANT2)<mean(ANT0)<mean(ANT1).');
```

```
        conditionCheck = meanA1<meanA(1) && meanA1>meanA(2);
    else
        disp('The mean of amplitudes must follow mean(ANT3)<mean(ANT2)<mean(ANT0)<mean(ANT1).');
        conditionCheck = meanA1<meanA(1) && meanA1>meanA(2) && meanA(3)<meanA(2);
    end
    if conditionCheck
       disp('Result: Pass');
    else
        disp('Result: Fail');
    end
end
```

```
Expected summations in the formulae for MRP(m) and MRPD must be non-zero.

Result: Pass

Expected 95% of the values v in the set RP(m) must meet -0.52<=principal(v-MRP(m))<=0.52.

Result: Pass

Expected MRPD in the range [-1.125, 1.125] radians.

Result: Pass
```

This example demonstrates the BLE receiver test measurements specific to IQC and IQDR test measurements. The simulation results verify that the computed test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

The example uses these helpers:

- helperBLEIQCIQDRTestConfig: Configure test parameters specific to IQC and IQDR test measurements
- helperBLETestWaveform: Generate BLE test waveform
- helperBLESwitchAntenna: Perform antenna steering and switching
- helperBLEPrincipalAngle: Compute principal angle in radians
- helperBLEIQCIQDRTest: Perform IQC and IQDR test measurements

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). “Bluetooth RF-PHY Test Specification”, RF-PHY.TS.5.1.0, Section 4.5. <https://www.bluetooth.com>.

2 Bluetooth Special Interest Group (SIG). "Core System Package [Low Energy Controller Volume]". Bluetooth Core Specification. Version 5.1, Volume <https://www.bluetooth.com>.

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151

- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62
- “Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift” on page 3-71

Bluetooth Low Energy Based Positioning Using Direction Finding

This example shows how to calculate the 2-D or 3-D position of a Bluetooth® low energy (BLE) node by implementing Bluetooth direction finding features and the triangulation-based location estimation technique using the Communication Toolbox™ Library for the Bluetooth Protocol. The Bluetooth Core Specification 5.1 [2 on page 3-0] introduced angle of arrival (AoA) and angle of departure (AoD) direction finding features to support centimeter-level accuracy in BLE location finding. This example shows how to calculate the distance between the estimated and actual BLE node positions in an additive white Gaussian noise (AWGN) channel to determine the positioning accuracy with respect to the bit energy to noise density ratio (Eb/No).

BLE Localization

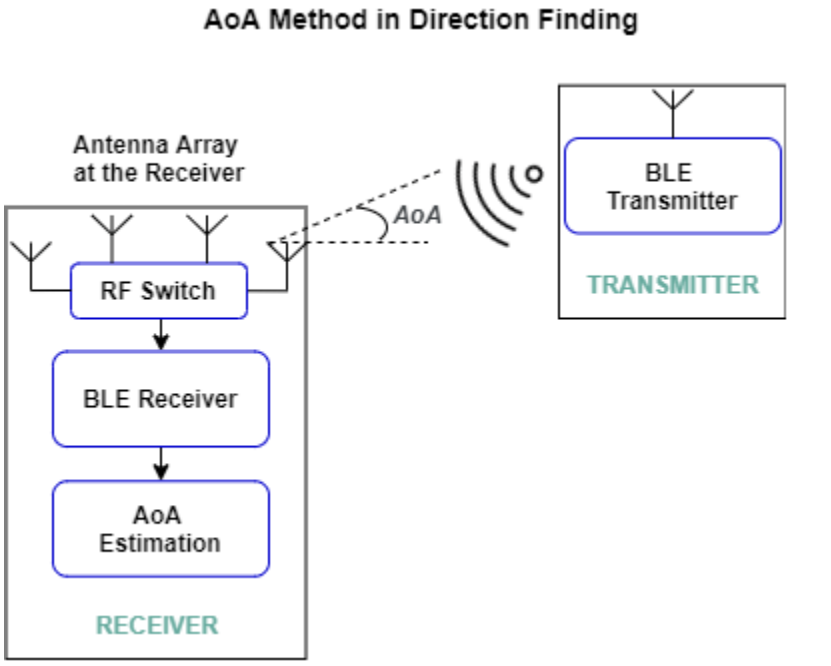
Bluetooth technology provides different types of location based services [1 on page 3-0]. On a high level, these services can be split into two categories.

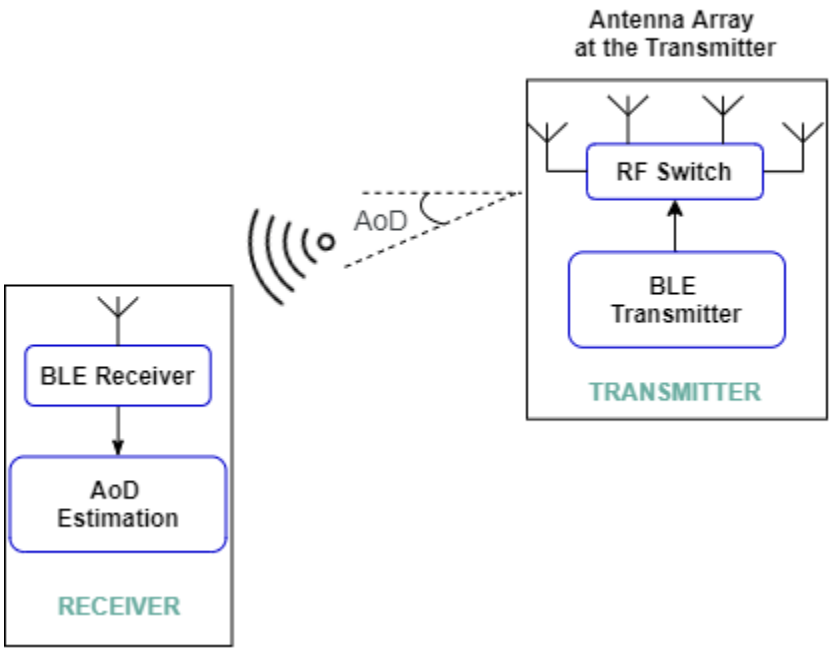
- Proximity Solutions: To estimate the distance between two devices, the Bluetooth proximity solutions previously used received signal strength indication (RSSI) measurements.
- Positioning Systems: To estimate the position of device, the Bluetooth positioning systems use trilateration based on several RSSI measurements to estimate the position of the device.

Previous versions of Bluetooth provide only meter-level accuracy in estimating the device location. The Bluetooth Core Specification 5.1 [2 on page 3-0] introduced new direction finding features that support centimeter-level accuracy in estimating the location of a device. For more information about direction finding services in BLE, see “Bluetooth Location and Direction Finding” on page 13-37.

Direction Finding Methods

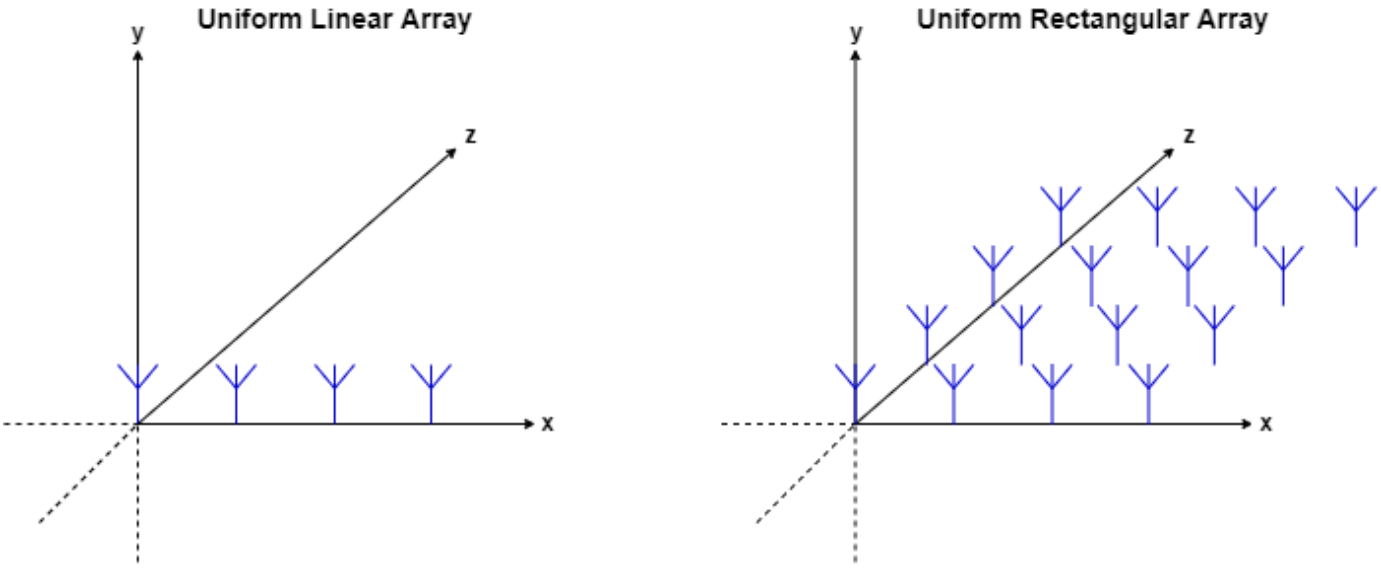
Bluetooth direction finding provides two distinct methods each of which exploits the same underlying basis. These direction finding methods are AoA and AoD. Each of these techniques require one of the two communicating devices to have an array of multiple antennas. In the AoA and AoD techniques, the antenna array is present at the receiver and transmitter, respectively.



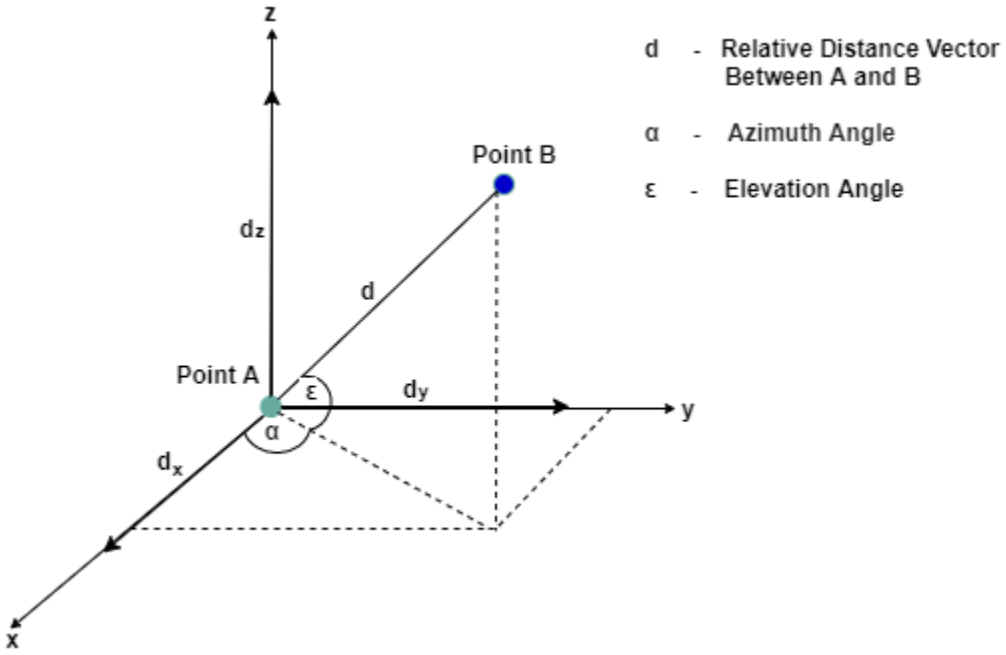


Antenna Arrays

Use a uniform linear array (ULA) or uniform rectangular array (URA) to calculate the direction of a signal. Simple linear designs like ULAs enable you to calculate only azimuth angle from a signal. Two dimensional arrays like URAs enable you to calculate both the azimuth and elevation angles in the 3-D half space. For more information about antenna arrays, see “Bluetooth Location and Direction Finding” on page 13-37.



Calculating the elevation and azimuth angles of the signal relative to a reference plane is common in these antenna arrays. This figure shows the concept of azimuth and elevation angles.



- **Azimuth angle**: This angle 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.
- **Elevation angle**: This 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.

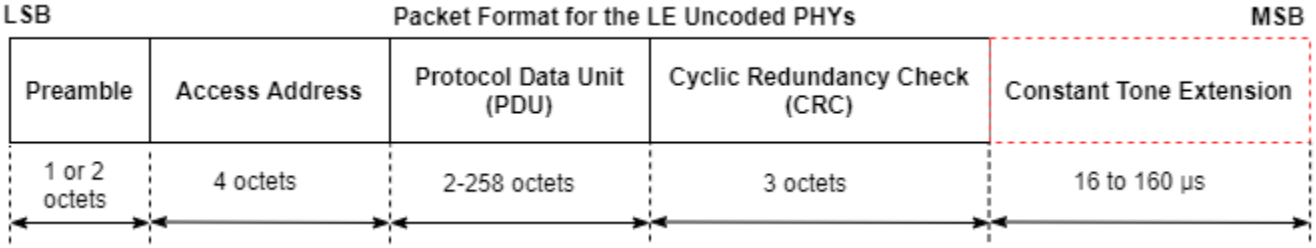
Direction Finding Signals

The communication in BLE is realized using one of these two distinct physical layers (PHYs).

- **LE Uncoded**: This PHY is further segregated into the LE1M PHY and LE2M PHY. LE1M is the default PHY and provides a symbol rate of 1 Msym/s. LE2M provides a symbol rate of 2 Msym/s.
- **LE Coded**: This PHY is equipped for long range communication and provides a symbol rate of 1 Msym/s. It has the potential to quadruple the range that can be achieved whilst reducing the data rate.

Bluetooth direction finding can use either the LE1M or LE2M PHY, but not the LECoded PHY.

The Bluetooth Core Specification 5.1 [2 on page 3-0] specifies additional data in the protocol data unit (PDU) packet structure, known as the constant tone extension (CTE) for direction finding. This figure shows the CTE appended at the end of LE uncoded PHY packet.



Use the CTE in any of these communication types.

- **Connection-oriented communication**: It specifies the CTE using the new LL_CTE_RSP PDUs that are sent over the connection in response to the LL_CTE_REQ PDUs.
- **Connectionless communication**: It appends the CTE to the existing periodic advertising PDUs, AUX_SYNC_IND, for direction finding.

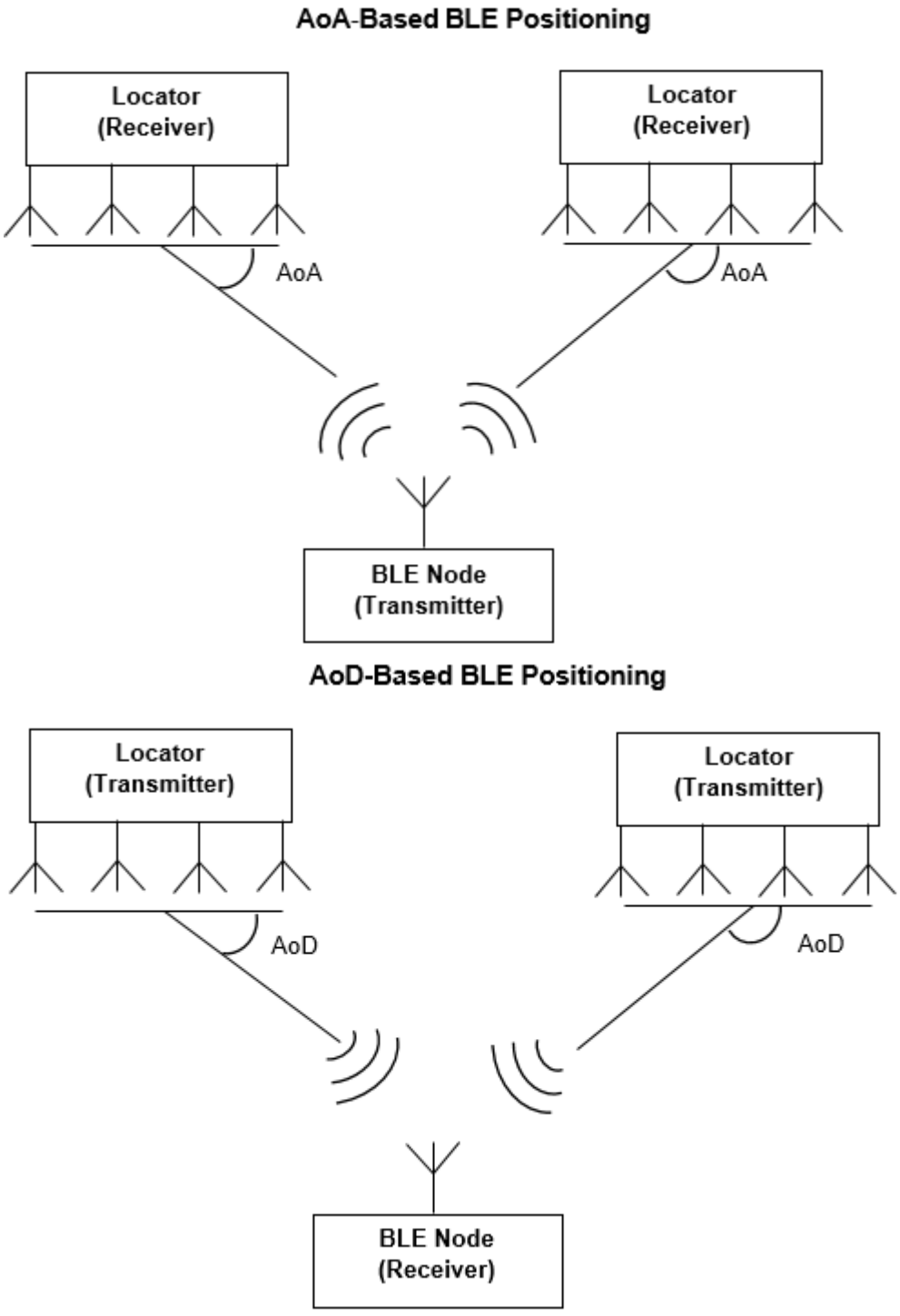
In connection-oriented and connectionless communication, the CTE is appended at the end of the PDU. For information about Bluetooth packet structures, see “Bluetooth Packet Structure” on page 13-23. For more information about the CTE, see volume 6, Part B, Section 2.5.1 of the Bluetooth Core Specification 5.1 [2 on page 3-0].

AoA and AoD Based BLE Positioning

This example uses these terms:

- *BLE node* - Specifies the device whose location is to be determined.
- *Locator* - Specifies the receiving device (in the AoA calculation) and transmitting device (in the AoD calculation).

This figure shows how to estimate the position of a BLE node using the AoA and AoD methods.



In the AoA method, the transmitter (BLE node) transmits a direction finding signal using single antenna. The receiving device (locator), equipped with an antenna array, takes the IQ samples while switching between the antennas present in the array. The locator uses the IQ samples to calculate the AoA.

In the AoD method, the transmitting device (locator) is equipped with an antenna array. The transmitting device transmits the signals while switching between the antennas in the array. The receiving device, consisting of a single antenna, collects the IQ samples and calculates the AoD.

To estimate the position of a BLE node in 2-D or 3-D, the device requires at least two or three locators in the network, respectively. Based on the estimated angles and the known BLE locator positions, estimate the position of a BLE node using the triangulation technique.

Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Simulation Parameters

Specify the dimension in which the BLE node position needs to be determined and the number of BLE locators. To estimate the 2-D or 3-D position of a BLE node, specify at least two or three locators, respectively.

```
numDimensions = 2;     % Dimension of BLE devices position in a network
numLocators = 3;       % Number of locators
```

Specify the Eb/No range and the number of iterations to simulate each Eb/No point.

```
EbNo = 6:2:16;                % Eb/No in dB
numIterations = 200;          % Number of iterations to average the position error
```

Specify the direction finding method, the direction finding packet type, and the PHY transmission mode.

```
dfMethod = AoA;               % Direction finding method
dfPacketType = ConnectionCTE; % Direction finding packet type
phyMode = LE1M;               % PHY transmission mode, must be LE1M or LE2M (for Con
```

Specify the antenna array parameters.

```
arraySize = 16;               % Antenna array size, must be a scalar (represents ULA
elementSpacing = 0.5;         % Normalized spacing between the antenna elements with
switchingPattern = 1:prod(arraySize); % Antenna switching pattern, must be a 1xM row vector
```

Specify the waveform generation parameters.

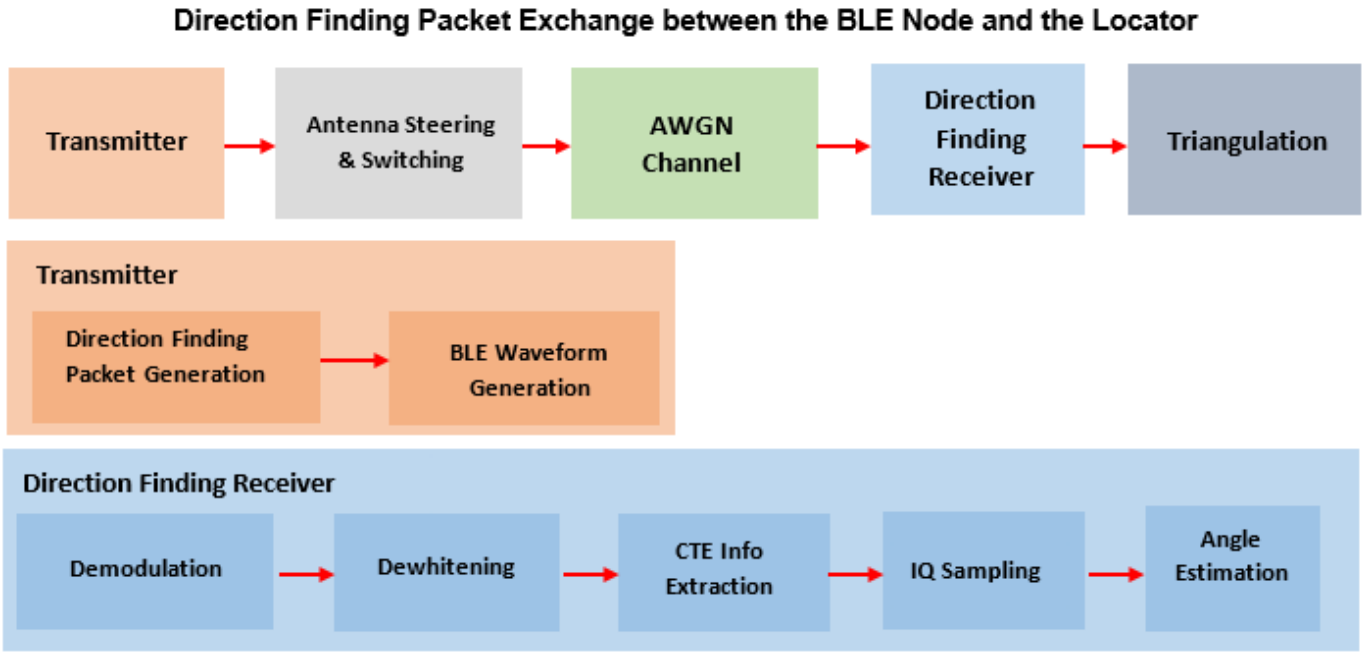
```
slotDuration = 2;             % Slot duration in microseconds
cteLength = 160;              % Length of CTE in microseconds, must be in the range
sps = 8;                      % Samples per symbol
```

```
chanIndex = 17;    % Channel index
crcInit = '555551';          % CRC initialization
accAddress = '01234567';     % Access address
payloadLength = 1;           % Payload length in bytes
```

Direction Finding and Position Estimation Procedure

Follow these steps to estimate the BLE node position.

1. Position the BLE node at the origin. Place the locators randomly in the 2-D or 3-D space.
2. Model the direction finding packet exchange between the BLE node and each locator to estimate the angles between them.



a. Generate a direction finding packet for connection or connectionless communication.

b. Generate BLE waveform.

c. Perform waveform steering and antenna switching.

d. Add AWGN to the waveform.

e. Perform demodulation, decoding, and IQ sampling on the noisy waveform.

f. Estimate the angle(s) between the BLE node and each locator using the IQ samples.

3. Estimate the BLE node position by performing triangulation using the known locator positions and the estimated angles.

For each iteration, assign random positions to the locators over a range of Eb/No points, and then repeat the preceding steps.

To increase the speed of the simulation, use a `parfor` loop instead of a `for` loop. The `parfor` loop processes each Eb/No point in parallel to reduce the total simulation time. To enable parallel computing for increased speed, comment-out the `for` statement and uncomment the `parfor` statement in this code. If “Parallel Computing Toolbox”™ is not installed, the `parfor` statement switches to the `for` statement by default.

Validate the simulation parameters.

```
minLocators = numDimensions; % Minimum number of locators
if numDimensions == 2 && size(arraySize,2) ~= 1
    error('The arraySize must be a scalar for 2-D position estimation');
end
if numDimensions == 3 && size(arraySize,2) ~= 2
    error('The arraySize must be a 1-by-2 vector for 3-D position estimation');
end
if numLocators < minLocators
    error(['The numLocators must be greater than or equal to ' num2str(minLocators) ' for ' num2s
end
if strcmp(dfPacketType,'ConnectionCTE') && payloadLength ~= 1
    error('The payloadLength must be 1 byte for direction finding packet type of ConnectionCTE')
elseif strcmp(dfPacketType,'ConnectionlessCTE') && payloadLength < 3
    error('The payloadLength must be greater than or equal to 3 bytes for direction finding packe
end
```

Create and configure BLE angle estimation configuration object. Derive the type of CTE based on the slot duration and the direction finding method.

```
if numDimensions == 3 && isscalar(elementSpacing)
    elementSpacing = [elementSpacing elementSpacing]; % Element spacing must be a vector to perfo
end
cfg = bleAngleEstimateConfig('ArraySize',arraySize,'SlotDuration',slotDuration,'SwitchingPattern
                                    switchingPattern,'ElementSpacing',elementSpacing);
validateConfig(cfg);
if strcmp(dfMethod,'AoA')
    cteType = [0;0];
else
    cteType = [0;1];
    if slotDuration == 1
        cteType = [1;0];
    end
end

% Convert access address in hexadecimal to binary
accessAddBits = de2bi(hex2dec(accAddress),32)';

% Initialize the variables to be used outside the parfor loop
numEbNo = numel(EbNo); % Number of Eb/No points
posNode = zeros(numDimensions,numEbNo);
posLocator = zeros(numDimensions,numLocators,numEbNo);
angleEst = zeros(numLocators,numDimensions-1,numEbNo);
posNodeEst = zeros(numDimensions,numEbNo);
validResult = zeros(1,numEbNo);
avgPositionError = zeros(1,numEbNo);
```

Model the direction finding packet exchange between the BLE node and each locator to estimate the angles between them.

```
% parfor iEbNo = 1:numEbNo % Use 'parfor' to speed up the simulation
for iEbNo = 1:numEbNo % Use 'for' to debug the simulation

    % Set the random substream index to ensure that each iteration uses a
    % repeatable set of random numbers
    stream = RandStream('combRecursive','Seed',12345);
    stream.Substream = iEbNo;
    RandStream.setGlobalStream(stream);

    % Loop over the number of iterations to average the positioning error.
    % If the successful links between the locators and node are less than
    % the minimum number of locators required to perform triangulation, then
    % the iteration fails.
    posErr = zeros(1,numIterations);
    iterationFailCount = 0;
    for iterCount = 1:numIterations

        % For each iteration, generate random positions for the locators
        [tempPosNode,tempPosLocator,ang] = helperBLEGeneratePositions(numLocators,numDimensions)
        posNode(:,iEbNo) = tempPosNode;
        posLocator(:,:,iEbNo) = tempPosLocator;

        % Loop over the number of locators
        tempAngleEst = zeros(numLocators,numDimensions-1);
        idx = [];
        linkFailFlag = zeros(numLocators,1);
        for i=1:numLocators

            % Generate direction finding packet
            data = helperBLEGenerateDFPDU(dfPacketType,cteLength,cteType,payloadLength,crcInit);

            % Generate BLE waveform
            bleWaveform = bleWaveformGenerator(data,'Mode',phyMode,'SamplesPerSymbol',sps,...
                'ChannelIndex',chanIndex,'DFPacketType',dfPacketType,'AccessAddress',accessAddBi

            % Perform steering and switching between the antennas
            dfWaveform = helperBLESwitchAntenna(bleWaveform,ang(i,:),...
                phyMode,sps,dfPacketType,payloadLength,cteLength,cfg);

            % Pass the waveform through AWGN channel
            snr = EbNo(iEbNo) - 10*log10(sps); % Signal to noise ratio (SNR)
            noiseWaveform = awgn(dfWaveform,snr,'measured');

            % Pass the noisy waveform to bleIdealReceiver which returns
            % the IQ samples
            [~, ~, iqSamples] = bleIdealReceiver(noiseWaveform,'Mode',phyMode,...
                'SamplesPerSymbol',sps,'ChannelIndex',chanIndex,'DFPacketType',...
                dfPacketType,'SlotDuration',slotDuration);

            % Estimate the angle(s) using the IQ samples. A packet is
            % detected successfully when the minimum number of non-zero IQ
            % samples are present.
            refSampleLength = 8; % Reference samples length
            % IQ samples must contain at least eight samples from reference period and
            % one sample from each antenna
            minIQSamples = refSampleLength+getNumElements(cfg)-1;
            if length(nonzeros(iqSamples)) >= minIQSamples % If packet detection is successful
                tempAngleEst(i,:) = bleAngleEstimate(iqSamples,cfg);
```

```
            else
                linkFailFlag(i) = 1; % If packet detection fails, enable the link fail flag
                idx = [idx i]; %#ok<AGROW> % Store the indices corresponding to the locator-node
            end
        end

        % If the successful node-locator links are greater than the minimum
        % number of locators and all the locator angles are not same, then
        % estimate the node position using triangulation and compute the
        % position error
        if (numLocators-nnz(linkFailFlag)) >= minLocators && ~isequal(tempAngleEst(:,1),repmat(te
            % If any link fails, then assign NaN to the corresponding angle
            % estimates
            posLocatorEbNo = posLocator(:,:,iEbNo);
            tempAngEstTri = tempAngleEst;
            if any(linkFailFlag == 1)
                posLocatorEbNo(:,idx) = [];
                tempAngleEst(idx,:) = NaN(numel(idx),numDimensions-1);
                tempAngEstTri(idx,:) = [];
            end

            % Estimate the node position using triangulation
            posNodeEst(:,iEbNo) = helperBLETriangulation(posLocatorEbNo,tempAngEstTri);
            angleEst(:,:,iEbNo) = tempAngleEst;

            % Compute the position error
            posErr(iterCount) = sqrt(sum((posNodeEst(:,iEbNo)-posNode(:,iEbNo)).^2));
        else
            iterationFailCount = iterationFailCount + 1; % Count the number of failed links used
        end
    end

    if(iterationFailCount == numIterations) % If all the links fail at a given Eb/No value
        disp(['At Eb/No = ',num2str(EbNo(iEbNo)),' dB, all direction finding packet transmissions
        validResult(iEbNo) = 0; % Disable plot flag for failed links
    else
        avgPositionError(iEbNo) = sum(posErr)/(numIterations-iterationFailCount);
        disp(['At Eb/No = ',num2str(EbNo(iEbNo)),' dB, positioning error in meters = ', num2str(a
        validResult(iEbNo) = 1; % Enable plot flag for successful links
    end
end
```

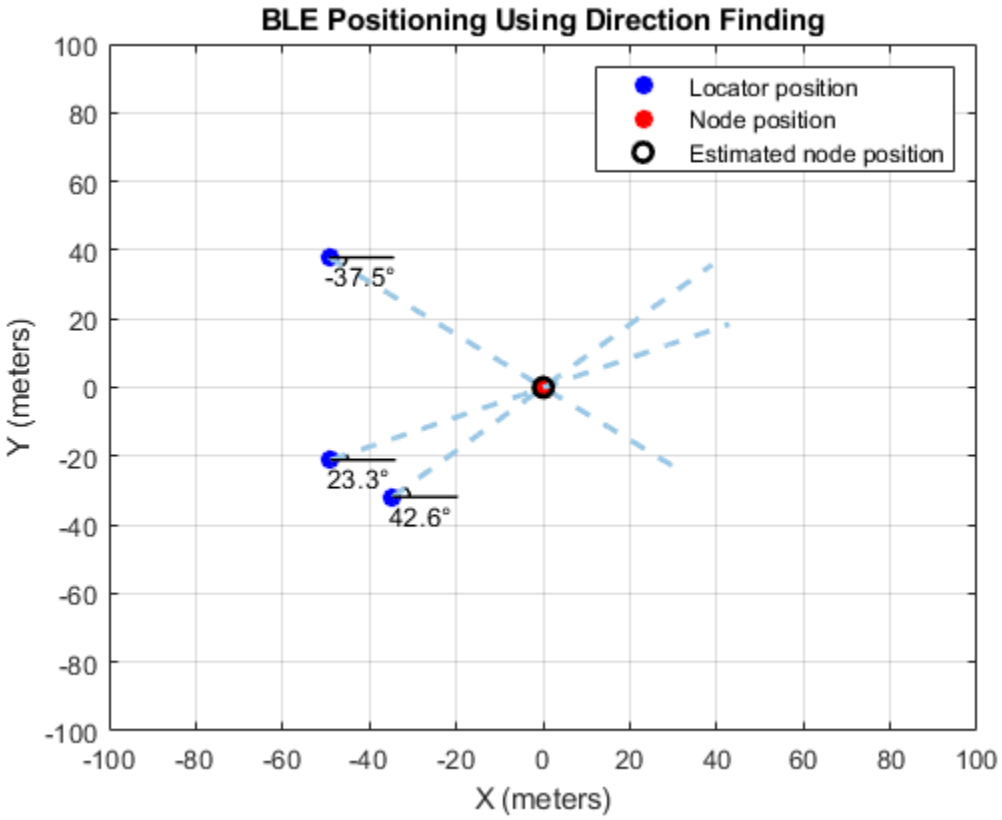
```
At Eb/No = 6 dB, positioning error in meters = 0.5594
At Eb/No = 8 dB, positioning error in meters = 0.43461
At Eb/No = 10 dB, positioning error in meters = 0.31708
At Eb/No = 12 dB, positioning error in meters = 0.27091
At Eb/No = 14 dB, positioning error in meters = 0.21464
At Eb/No = 16 dB, positioning error in meters = 0.17115
```

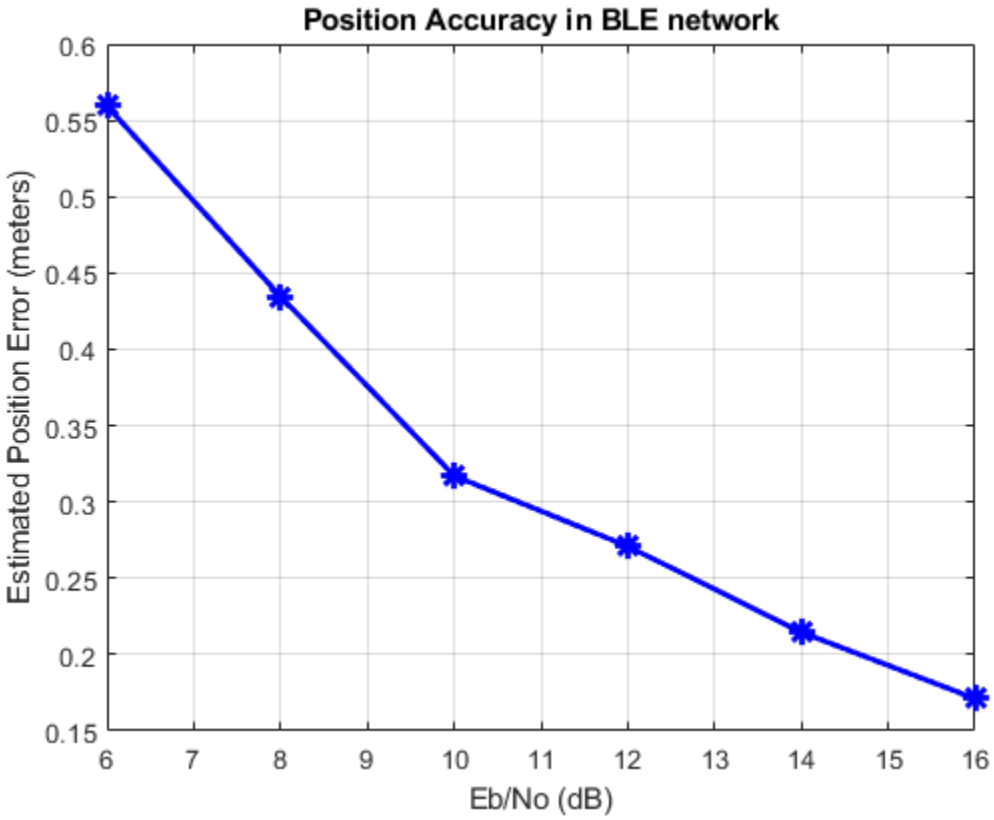
Simulation Results

If the direction finding packet transmission is successful, the example displays these plots.

- BLE node position, locators positions, and the estimated node positions in 2-D or 3-D. The plot also shows the triangulation lines for the maximum Eb/No value and the last iteration.
- Average position error (in meters) versus Eb/No (in dB).

```
[~,validIdx] = find(validResult==1);
if ~isempty(validIdx)
    EbNoValid = EbNo(validIdx);
    [~,EbNoIdx] = max(EbNoValid);
    EbNoValidIdx = validIdx(EbNoIdx);
    helperBLEVisualizePosition(posLocator(:,:,EbNoValidIdx),posNode(:,EbNoValidIdx),...
                        angleEst(:,:,EbNoValidIdx),posNodeEst(:,EbNoValidIdx));
    figure
    plot(EbNoValid,avgPositionError(validIdx),'-b*','LineWidth',2, ...
            'MarkerEdgeColor','b','MarkerSize',10)
    grid on
    xlabel('Eb/No (dB)')
    ylabel('Estimated Position Error (meters)')
    title('Position Accuracy in BLE network')
end
```





This example enables you to estimate the 2-D or 3-D position of a BLE node by using the BLE direction finding functionality. The example shows how to implement the triangulation method to calculate the angles between the locators and the BLE node. The example also shows how to measure the positioning accuracy related to the Eb/No value by computing the distance between the estimated and actual node positions in an AWGN channel.

Appendix

The example uses these helpers:

- helperBLEGeneratePositions: Generate locators and node positions
- helperBLEGenerateDFPDU: Generate direction finding packet PDU
- helperBLESwitchAntenna: Perform antenna steering and switching
- helperBLETriangulation: Estimate the node position using triangulation
- helperBLEVisualizePosition: Generate BLE position visualization

Selected Bibliography

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). "Core System Package [Low Energy Controller Volume]". Bluetooth Core Specification. Version 5.1, Volume https://www.bluetooth.com.

See Also

Functions
`bleAngleEstimate` | `bleWaveformGenerator` | `bleIdealReceiver`

Objects
`bleAngleEstimateConfig`

More About

- “Bluetooth Location and Direction Finding” on page 13-37

Bluetooth Full Duplex Data and Voice Transmission in MATLAB

This example shows how to model a full duplex communication in a Bluetooth® piconet having WLAN interference and supporting the adaptive frequency hopping (AFH) functionality using the Communications Toolbox™ Library for the Bluetooth Protocol. The Bluetooth nodes operating with basic rate (BR) physical layer (PHY) communicate with each other simultaneously by transmitting data packets (over an asynchronous connection-oriented (ACL) logical transport) and voice packets (over a synchronous connection-oriented (SCO) logical transport) as random bits. The supported data and voice packets are:

- Data packet types: DM1, DH1, DM3, DH3, DM5, and DH5
- Voice packet types: HV1, HV2, and HV3

This example enables AFH by classifying channels as *good* or *bad* based on the packet error rate (PER) of each channel. You can add your own classification algorithm to analyze the simulation results. The simulation results show a plot of the packet error rate (PER) for each Bluetooth node. The power spectral density of Bluetooth waveforms with WLAN interference is visualized using the spectrum analyzer.

Bluetooth Specifications

Bluetooth technology operates in 2.4 GHz industrial, scientific, and medical (ISM) band and shares it with other wireless technologies like ZigBee and WLAN. The Bluetooth Core Specification [1 on page 3-0] defined by the Special Interest Group (SIG) specifies two PHY modes: the mandatory BR and the optional enhanced data rate (EDR). The Bluetooth BR/EDR radio implements a 1600 hops/s frequency hopping spread spectrum (FHSS) technique. The radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth channel has a bandwidth of 1 MHz. Each channel is centered at (2402 + k) MHz, where k = 0, 1, ... 78. The modulation technique on the payload for BR and EDR mode is Gaussian frequency shift-keying (GFSK) and differential phase shift-keying (DPSK), respectively. The baud rate is 1 MSymbols/s. The Bluetooth BR/EDR radio uses a time-division duplex (TDD) scheme in which data transmission occurs in one direction at one time. The transmission alternates in two directions, one after the other.

Bluetooth and WLAN radios often operate in the same physical scenario and in the same device. Therefore, Bluetooth and WLAN transmissions can interfere with each other, thus impacting the performance and reliability of both the networks. To mitigate this interference, the IEEE 802.15.2 Task Group [2 on page 3-0] recommends using the AFH technique. To study AFH and the coexistence of Bluetooth with WLAN, see “Bluetooth-WLAN Coexistence” on page 13-60.

For more information about the Bluetooth BR/EDR radio and the protocol stack, see “Bluetooth Protocol Stack” on page 13-7. For more information about Bluetooth BR/EDR packet structures, see “Bluetooth Packet Structure” on page 13-23.

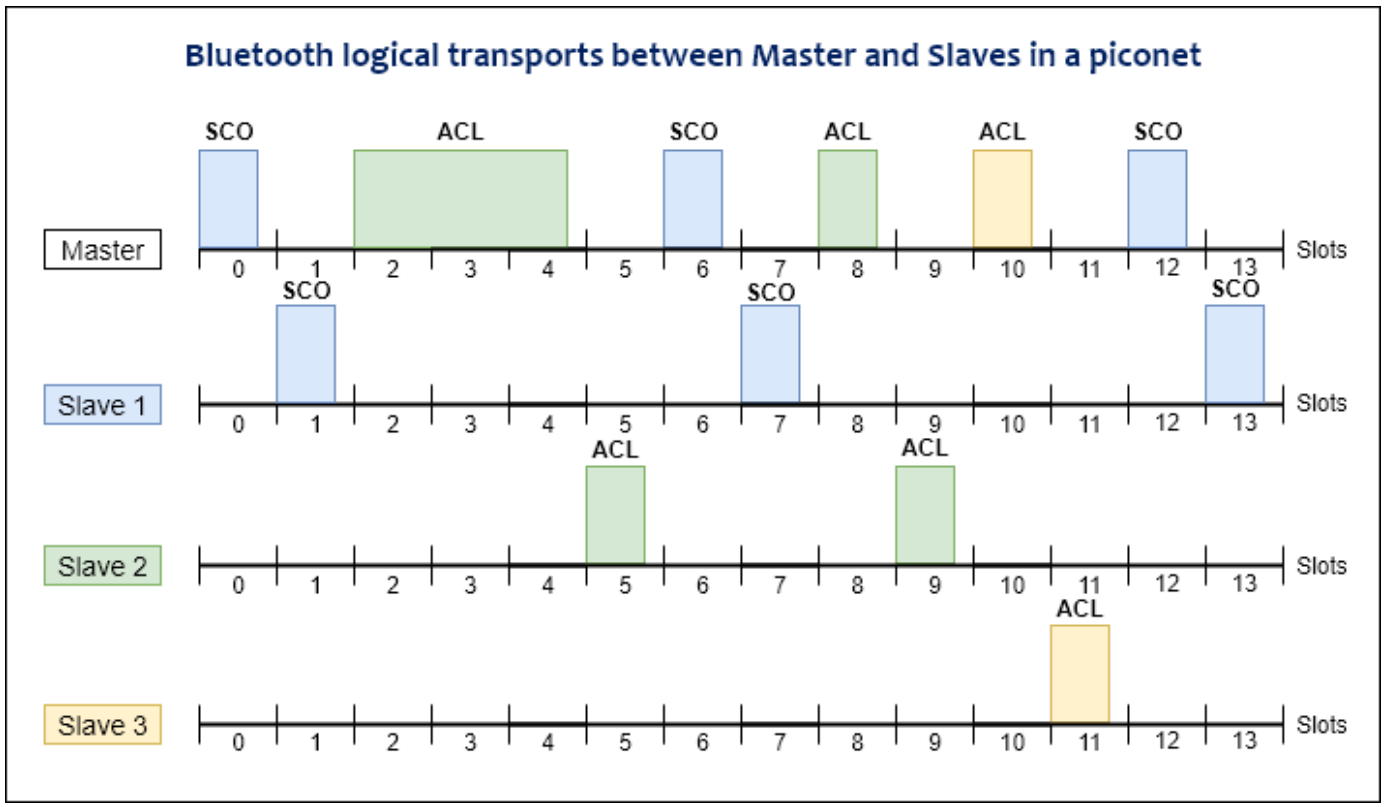
Logical Transports

The Bluetooth system supports point-to-point or point-to-multipoint connections called as *piconets*. Each piconet consists of a node in the role of Master, with other nodes in the Slave role. The Master and Slave exchange data over multiple logical transports. These logical transports are:

- *SCO*: The Master and Slave exchange SCO packets at regular intervals in the reserved slots. The Bluetooth nodes use SCO logical transport to exchange periodic data such as audio streaming. This logical transport does not support retransmissions.

- *Extended synchronous connection-oriented (eSCO)*: The Bluetooth nodes use eSCO to exchange periodic data such as audio streaming. This logical transport supports retransmissions.
- *ACL*: The Bluetooth nodes use ACL to exchange asynchronous data such as a file transfer protocol (FTP). During each poll interval, the Master polls the ACL logical transport of a Slave at least once.
- *Active slave broadcast (ASB):* The Bluetooth nodes use ASB logical transport to send messages from the Master to all of the Slaves in a piconet. This logical transport supports unidirectional traffic with no acknowledgments.
- *Connectionless slave broadcast (CSB):* The Master node uses a CSB logical transport to send profile broadcast data to multiple Slaves. This logical transport supports unidirectional traffic with no acknowledgments.

This example supports ACL and SCO logical transports between a Master and Slaves by considering data as random bits of 0s and 1s. This figure shows the communication between a Master and three Slaves in a piconet over ACL and SCO logical transports.



Bluetooth uses reserved time slots for communication between the nodes. The duration of each slot is 625 microseconds. The Master node initiates the transmission in even slots and extends the transmission to odd slots when transmitting a multislot packet. The Slave node initiates the transmission in odd slots and extends the transmission to even slots when transmitting a multislot packet.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

This section shows how to configure the simulation parameters for the Bluetooth piconet, the wireless channel, and WLAN interference.

Bluetooth Piconet

The `NumSlaves` parameter specifies the number of Slaves in the Bluetooth piconet. The `LinkTraffic` parameter specifies the type of traffic over Bluetooth logical transports between a Master and the respective Slave. This table maps `LinkTraffic` to different logical transports.

linkTraffic Value	Logical Transport
1	ACL
2	SCO
3	ACL and SCO

If the Master communicates with multiple Slaves, `LinkTraffic` must be a vector.

The `SequenceType` parameter specifies the type of frequency hopping algorithm that the Bluetooth node uses. When you set `SequenceType` to `'Connection adaptive'`, the Bluetooth channels are classified as *good* or *bad* periodically based on the PER of each Bluetooth channel. To classify the Bluetooth channels, you can use the classifyChannels object function.

```
% Set the simulation time in microseconds
simulationTime = 3*1e6;

% Enable or disable the visualization in the example
enableVisualization = true;

simulationParameters = struct;
% Configure the number of Slaves in the piconet
simulationParameters.NumSlaves = 1 ;

% Specify the positions of Bluetooth nodes in the form of n-by-3 array.
% where n represents the number of nodes in the piconet. Each row specifies
% the cartesian coordinates of a nodes starting from Master and followed by
% Slaves.
simulationParameters.NodePositions = [10 0 0; 20 0 0];

% Configure the logical links between the Master and Slaves

% Each element represents the logical link between the Master and the
% respective Slave. If the Master is connected to multiple Slaves, this
% value must be a row vector.
simulationParameters.LinkTraffic = 1;

% Configure the frequency hopping sequence as 'Connection basic' or
% 'Connection adaptive'
simulationParameters.SequenceType = Connection adaptive ;
```

```
% To enable an ACL logical transport, set linkTraffic to 1 or 3. Specify
% the ACL packet type as 'DM1', 'DH1', 'DM3', 'DH3', 'DM5', or 'DH5'.
simulationParameters.ACLPacketType = DM1;

% To enable a SCO logical transport, set linkTraffic to 2 or 3. Specify the
% SCO packet type as 'HV1', 'HV2', or 'HV3' for the respective Slave that
% has SCO link traffic. Here, 1 represents the Slave number, and 'HV3'
% represents the corresponding SCO packet type used by Slave 1.
simulationParameters.SCOPacketType = {1, 'HV3'};
```

Wireless Channel and WLAN Interference

Configure the wireless channel by using the helperBluetoothChannel helper object. You can set the `EbNo` value for the AWGN channel. To generate the WLAN signal interference, use the helperBluetoothGenerateWLANWaveform helper function. Specify the sources of WLAN interference by using the `WLANInterference` parameter. Use one of these options to specify the source of the WLAN interference.

- `'Generated'`: To add a WLAN Toolbox™ signal, select this option. Perform the steps shown in further exploration on page 3-0 to add the signal from the WLAN Toolbox™.
- `'BasebandFile'`: To add a WLAN signal from a baseband file (`.bb`), select this option. You can specify the file name using the `WLANBBFilename` input argument. If you do not specify the `.bb` file, the example uses the default `.bb` file, `'WLANNonHTDSSS.bb'`, to add the WLAN signal.

The `'None'` option implies that no WLAN signal is added. AWGN is present throughout the simulation.

```
% Configure wireless channel parameters
simulationParameters.EbNo = 22; % Ratio of energy per bit (Eb) to spectral noise density (No) in

% Configure the WLAN interference

% Specify the WLAN interference as 'Generated', 'BasebandFile', or 'None'.
% To use the 'wlanBBFilename' option, set wlanInterference to
% 'BasebandFile'.
simulationParameters.WLANInterference = BasebandFile;
simulationParameters.WLANBBFilename =  'WLANNonHTDSSS.bb';

% Signal to interference ratio in dB
simulationParameters.SIR = [-15 -16];
```

Channel Classification Parameters

Classify the Bluetooth channels as good or bad by using the helperBluetoothChannelClassification object only when the `SequenceType` is `'Connection adaptive'`. The example classifies the Bluetooth channels by using these parameters.

- `PERThreshold`: PER threshold
- `ClassificationInterval`: Periodicity (in slots) of channel classification
- `RxStatusCount`: Maximum number of received packets status maintained for each channel
- `MinRxCountToClassify`: Minimum number of received packets status for each channel to classify a channel as good or bad

- `PreferredMinimumGoodChannels`: Preferred number of good channels required to communicate between the Master and Slaves

You can add your own classification algorithm by customizing the classifyChannels method of the helperBluetoothChannelClassification object.

```
simulationParameters.PERThreshold = 50;                 % Packet error
simulationParameters.ClassificationInterval = 3000;   % In slots
simulationParameters.RxStatusCount = 10;                % Maximum Rx pa
simulationParameters.MinRxCountToClassify = 4;          % Minimum packe
simulationParameters.PreferredMinimumGoodChannels = 20; % Preferred num
```

Create Bluetooth Piconet

Specify the total number of Bluetooth nodes in the piconet. Set the role of the nodes as Master or Slave. To create a Bluetooth piconet from the configured parameters, use the helperBluetoothCreatePiconet helper function.

```
% Reset the random number generator
rng('default');

% Specify Tx power, in dBm
simulationParameters.TxPower = 20;

% Specify the Bluetooth node receiver range (in meters)
simulationParameters.ReceiverRange = 40;

% Set the total number of nodes in the piconet (one Master and multiple
% Slaves)
numNodes = simulationParameters.NumSlaves + 1;

% Create a Bluetooth piconet
btNodes = helperBluetoothCreatePiconet(simulationParameters);
```

To visualize the Bluetooth waveforms, create the `dsp.SpectrumAnalyzer` System™ object.

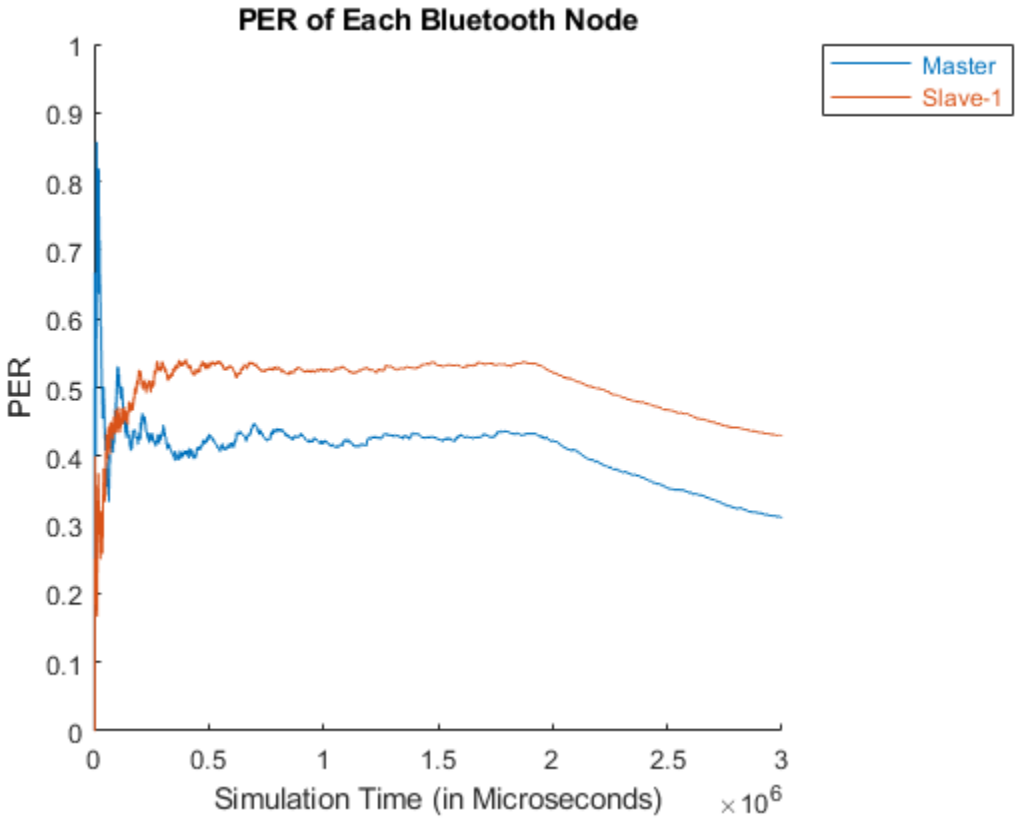
```
% View the Bluetooth waveforms using the spectrum analyzer
spectrumAnalyzer = dsp.SpectrumAnalyzer(...
    'Name','Bluetooth Full Duplex Communication', ...
    'ViewType','Spectrum and spectrogram', ...
    'TimeResolutionSource','Property', ...
    'TimeResolution',0.0005, ...
    'SampleRate',btNodes{1}.PHY.SamplesPerSymbol*1e6, ...
    'TimeSpanSource','Property', ...
    'TimeSpan',0.05, ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',512, ...
    'AxesLayout','Horizontal', ...
    'FrequencyOffset',2441*1e6, ...
    'ColorLimits',[-20 15]);
```

Simulation

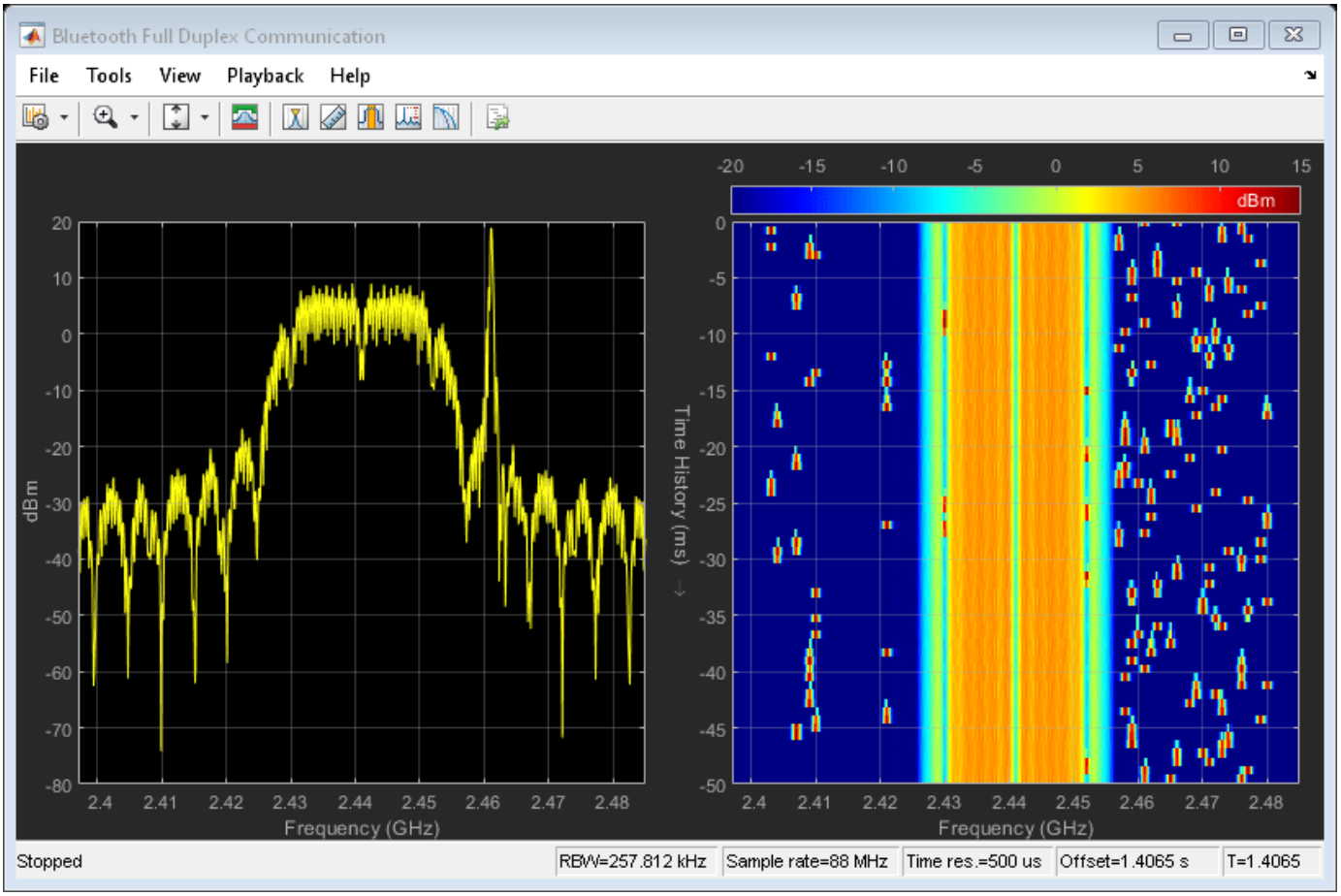
Simulate the Bluetooth piconet using the configured parameters. Visualize the plot of the PER of each Bluetooth node in the piconet. Visualize the power spectral density of the Bluetooth waveforms by using the `dsp.SpectrumAnalyzer` System object. You can also calculate the baseband layer statistics (total transmitted packets, total received packets, and total dropped packets) and channel classification statistics at each Bluetooth node. When the sequence type is set to `'Connection adaptive'`, the Bluetooth node updates the channel classification statistics.

```
% Current simulation time in microseconds
curTime = 0;

% Elapsed time in microseconds
elapsedTime = 0;

% Next invoke times of all of the nodes in microseconds
nextInvokeTimes = zeros(1, numNodes);

if enableVisualization
    % Plot the PER
    perFigure = figure("Name","PER of Each Bluetooth Node",'Tag','BluetoothPERPlot');
    perAxes = axes(perFigure);
    % Add annotations to the figure
    ylim(perAxes, [0 1]);
    xlabel(perAxes, 'Simulation Time (in Microseconds)');
    ylabel(perAxes, 'PER');
    title(perAxes, 'PER of Each Bluetooth Node');
    % Plot the PER line for each Bluetooth node
    [perPlots, legendStr] = deal(cell(1, numNodes));
    for plotIdx = 1:numNodes
        hold on
        perPlots{plotIdx} = plot(perAxes, curTime, 0);
        if plotIdx == 1
            legendStr{1} = ['\color[rgb]{' num2str(perPlots{plotIdx}.Color) '} Master'];
        else
            legendStr{plotIdx} = ['\color[rgb]{' num2str(perPlots{plotIdx}.Color) '} Slave-' num2
        end
    end
    % Add a legend to the figure
    legend(perAxes, legendStr,'Location','northeastoutside','Box','on');
    if ~strcmpi(simulationParameters.WLANInterference, 'None')
        % Generate the WLAN waveform for visualization
        wlanWaveform = helperBluetoothGenerateWLANWaveform(...
            simulationParameters.WLANInterference, simulationParameters.WLANBBFilename);
    end
end

% Run the simulation
while(curTime < simulationTime)
    % Simulate the Bluetooth nodes
    for nodeIdx = 1:numNodes
        % Push the data into the node
        pushData(btNodes{nodeIdx}, ...
            simulationParameters.ACLPacketType, simulationParameters.SCOPacketType);

        % Run the Bluetooth node instance
        nextInvokeTimes(nodeIdx) = runNode(btNodes{nodeIdx}, elapsedTime);
```

```
            if enableVisualization
                if ~isempty(findobj('Tag', 'BluetoothPERPlot'))
                    % Update and plot the PER
                    perPlots{nodeIdx}.XData = [perPlots{nodeIdx}.XData curTime];
                    if isempty(btNodes{nodeIdx}.PHY.PER)
                        per = 0;
                    else
                        per = btNodes{nodeIdx}.PHY.PER;
                    end
                    perPlots{nodeIdx}.YData = [perPlots{nodeIdx}.YData per];
                end
            end
        end

        % Update the current simulation time
        curTime = curTime + elapsedTime;

        % Distribute any transmitted packets from each node into the receiving
        % buffers of the other nodes
        [isPacketDistributed, txBuffer] = helperBluetoothDistributePackets(btNodes);
        % Update visualization
        if enableVisualization
            for txIdx = 1:numel(txBuffer)
                channelWaveform = txBuffer{txIdx}.Waveform(1:txBuffer{txIdx}.NumSamples);
                if ~strcmpi(simulationParameters.WLANInterference, 'None')
                    % Add WLAN interference to the channel waveform
                    channelWaveform = channelWaveform + wlanWaveform(1:txBuffer{txIdx}.NumSamples);
                end
                % Plot the Bluetooth waveform
                spectrumAnalyzer(channelWaveform);
            end
        end

        % If packets are distributed to the receiver nodes, run the nodes to
        % check the packet reception buffer.
        if isPacketDistributed
            elapsedTime = 0;
        % Advance the simulation time to the next event at a node
        else
            elapsedTime = min(nextInvokeTimes(nextInvokeTimes ~= -1));
        end
    end

ans = 1x79 logical array

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




```
release(spectrumAnalyzer);
```



The preceding spectrum analyzer plot shows the spectrum of the Bluetooth waveform distorted with WLAN interference (in the frequency domain) and passed through the AWGN channel. The right-side plot shows the overlapping of Bluetooth packets with the interfering WLAN signal.

The plot "PER of Each Bluetooth Node" shows the PER of each node in the Bluetooth piconet with respect to the simulation time.

To see the baseband layer statistics for each Bluetooth node, inspect the `statisticsAtEachNode` variable. To see the channel classification statistics for each Master-Slave pair, inspect the `classificationStats` variable. The channel classification statistics are valid when `sequenceType` is set to `'Connection adaptive'`. Get the baseband layer and channel classification statistics of each Bluetooth node in the piconet.

```
% Get the baseband layer and channel classification statistics of each Bluetooth node in the pico
[statisticsAtEachNode, classificationStats] = helperBluetoothFullDuplexStatistics(btNodes)
```

```
statisticsAtEachNode=2×19 table
              TotalRxPackets    TotalTxPackets    TxACLPackets    TxACLOneSlotPackets    TxACLTh
              ______________    ______________    ____________    ___________________    _______

    Master          1435              2401        {7x2 double}        {7x2 double}           {7x2
    Slave1          2379              1444        {7x2 double}        {7x2 double}           {7x2
```

```
classificationStats=1×3 table
    LTAddress    ClassificationCount                                    ChannelMap
    _________    ___________________    ____________________________________________________________

        1                1            {[0 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
```

This example enables you to configure and simulate a full duplex data and voice communication in a Bluetooth piconet with WLAN interference. The example uses AFH to mitigate WLAN interference by classifying channels as *good* or *bad* based on the PER value. The performance of full duplex communication is analyzed by visualizing the impact of WLAN interference on each Bluetooth node.

Further Exploration

To add the WLAN signal using the features of the WLAN Toolbox™, set the value of wlanInterference to 'Generated'. Use this code to add the WLAN signal.

Add the WLAN signal generated using the features of the WLAN Toolbox™ as static signal interference to Bluetooth. To enable this, set `wlanInterference` value as 'Generated', add your custom signal generation code in function. Use this sample code snippet in WLAN signal generation using features of WLAN Toolbox™.

```
% % WLAN waveform to interfere with Bluetooth waveforms can be
% % modified by using the features of the WLAN Toolbox(TM).
% psduLength = 1000;
%
% % Create configuration object for WLAN waveform (802.11b)
% cfgNHT = wlanNonHTConfig('Modulation','DSSS', ...
%     'PSDULength', psduLength);
%
% % Create random PSDU
% psdu = randi([0 1], cfgNHT.PSDULength*8, 1);
%
% % Generate WLAN waveform
% wlanWaveform = wlanWaveformGenerator(psdu, cfgNHT);
```

You can add your custom signal generation code in the helperBluetoothGenerateWLANWaveform function. You can also write the respective signal WLAN spectrum masks and register to the `WLANSpectrum` property of the helperBluetoothChannel as a function pointer.

Appendix

The example uses this object:

- `bluetoothFrequencyHop`: Selects Bluetooth BR/EDR channel index for frequency hopping

The example uses these helpers:

- helperBluetoothFullDuplexNode: Configures and simulates Bluetooth node
- helperBluetoothBaseband: Configures and simulates Bluetooth baseband layer
- helperBluetoothLogicalTransports: Configures logical transports between Bluetooth nodes
- helperBluetoothSlotTimer: Manages Bluetooth clock and the timing of slots
- helperBluetoothPHY: Configures and simulates Bluetooth PHY layer
- helperBluetoothChannel: Configures and simulates Bluetooth wireless channel

- helperBluetoothGenerateWLANWaveform: Generates WLAN waveform to be added as an interference to Bluetooth waveforms
- helperBluetoothWLANDSSSSpectrumMask: Calculates adjacent channel interference power using the WLAN 802.11b (DSSS) spectrum masks
- helperBluetoothCreatePiconet: Creates Bluetooth piconet using the Bluetooth nodes
- helperBluetoothFullDuplexStatistics: Returns statistics of each Bluetooth node in the Bluetooth piconet
- helperBluetoothQueue: Create an object for Bluetooth queue functionality

Selected Bibliography

1. Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com.
2. IEEE® Standard 802.15.2™. "Coexistence of Wireless Personal Area Networks with Other Wireless Devices Operating in Unlicensed Frequency Bands." *IEEE Recommended Practice for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements; IEEE Computer Society.*

See Also

More About

- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96
- “Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH” on page 13-101
- “Packet Distribution in Bluetooth Piconet” on page 13-106
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76

Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability

This example shows how to perform Bluetooth® enhanced data rate (EDR) radio frequency (RF) physical layer (PHY) transmitter tests specific to modulation accuracy and carrier frequency stability using the Communications Toolbox™ Library for the Bluetooth Protocol. The test measurements compute the initial frequency offset, root mean square (RMS) differential error vector magnitude (DEVM), and peak DEVM values. This example also verifies whether these test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Objectives of Bluetooth RF-PHY Tests

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by the Bluetooth Special Interest Group (SIG) includes RF-PHY tests for the transmitter and receiver. The objectives of these RF-PHY tests are to:

- Ensure interoperability between all of the Bluetooth devices.
- Ensure a basic level of system performance for all of the Bluetooth products.

Each test case has a specific test procedure and an expected outcome, that must be achieved by the implementation under test (IUT).

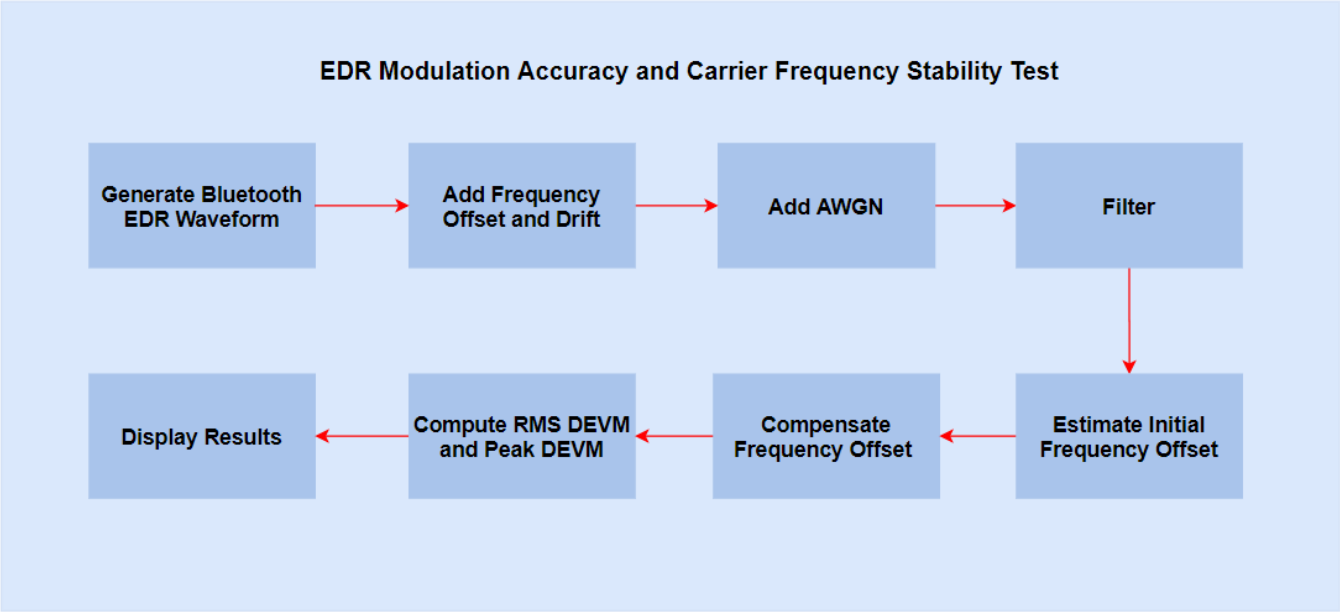
RF-PHY Transmitter Tests

The main goal of the transmitter test measurements is to ensure that the transmitter characteristics are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0]. This example includes transmitter tests relevant to EDR modulation accuracy and carrier frequency stability. This table shows various RF-PHY transmitter tests performed in this example.

Conformance Test	Test Case ID	Test Purpose
Modulation accuracy	RF/TRM/CA/BV-11-C	This test verifies the modulation accuracy of Bluetooth EDR waveforms.
Carrier frequency stability	RF/TRM/CA/BV-11-C	This test verifies the carrier frequency stability of Bluetooth EDR waveforms.

RF-PHY Transmitter Test Procedure

This block diagram summarizes the test procedure for transmitter tests relevant to EDR modulation accuracy and carrier frequency stability of Bluetooth EDR waveforms.



- Generate DH or EV packets by using pseudorandom sequences of these lengths.

PHY Mode	Packet Type	Maximum Payload Length (Bytes)
$\pi/4-$ DQPSK	2-DH1	31
	2-DH3	356
	2-DH5	656
	2-EV3	58
	2-EV5	358
8-DPSK	3-DH1	11
	3-DH3	88
	3-DH5	986
	3-EV3	88
	3-EV5	538

- Pass the payload bits through the `bluetoothWaveformGenerator` function to generate Bluetooth EDR test waveforms.
- Add a carrier frequency offset and drift.
- Add additive white Gaussian noise (AWGN).
- Estimate the initial frequency offset using the basic rate (BR) portion of the waveform.
- Compensate the EDR portion with the estimated initial frequency offset.

- Perform square root raised cosine filtering using the filter whose coefficients are generated based on the Bluetooth RF-PHY Test Specifications [1 on page 3-0].
- Divide the EDR portion into blocks of length 50 microseconds each.
- For each block, delay the compensated sequence by 1 microsecond and differentiate the delay with actual compensated sequence to get the error sequence.
- Compute the RMS DEVM and peak DEVM based on the error sequence and compensated sequence.
- Get the test verdict and display the results.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

To specify PHY transmission mode, packet type, initial frequency offset, maximum frequency drift, and samples per symbol, set `phyMode`, `packetType`, `initialFreqOffset`, `maxFreqDrift`, and `sps` respectively.

```
phyMode = EDR2M;                          % PHY transmission mode
packetType = DH1;                         % EDR packet type
initialFreqOffset = 40000;                % Initial frequency offset (Hz)
maxFreqDrift = 0;                         % Maximum frequency drift (Hz), must be in t
sps = 8;                                  % Samples per symbol
```

Generate Test Parameters

Use the preceding configured parameters to generate the test parameters. To get all of the test parameters, use the helperEDRModulationTestConfig.m helper function. To add frequency offset and thermal noise, create and configure `comm.PhaseFrequencyOffset` and `comm.ThermalNoise` System objects, respectively.

```
[edrTestParams,waveformConfig,filtCoeff] = helperEDRModulationTestConfig(phyMode,packetType,sps)

% Create frequency offset System object
frequencyDelay = comm.PhaseFrequencyOffset('SampleRate',edrTestParams.sampleRate);

% Create thermal noise System object
NF = 12; % Noise figure (dB)
thNoise = comm.ThermalNoise('NoiseMethod','Noise figure', ...
                            'SampleRate',edrTestParams.sampleRate, ...
                            'NoiseFigure',NF);
```

Simulate Transmitter Tests

Using the preceding RF-PHY transmitter test procedure, simulate the transmitter tests.

```
% Initialize variables
symDEVM = zeros(1,edrTestParams.requiredBlocks*edrTestParams.blockLength);
[blockRMSDEVM,estimatedBlockFreqDrifts] = deal(zeros(1,edrTestParams.requiredBlocks));
```

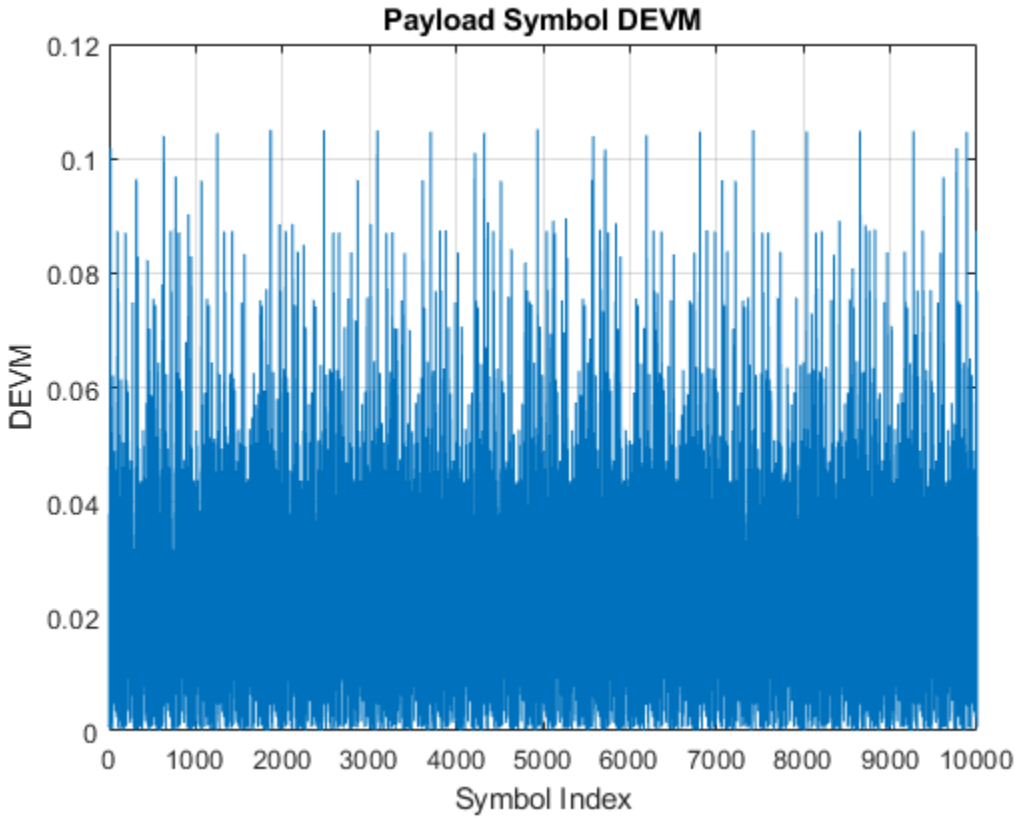
```
estimatedInitFreqOff = zeros(1,edrTestParams.NumPackets);
blockCount = 0;

% Generate 200 blocks of data as specified in Bluetooth RF-PHY Test Specifications
for packetCount = 1:edrTestParams.NumPackets

    % Generate random bits
    payload = edrTestParams.pnSeq();

    % Generate Bluetooth EDR waveform
    txWaveform = bluetoothWaveformGenerator(payload,waveformConfig);

    % Generate ideal EDR symbols from waveform
    packetDuration = helperBluetoothPacketDuration(packetType,phyMode,edrTestParams.numBytes);
    txWaveform1 = txWaveform(1:(packetDuration+edrTestParams.span)*sps);
    idealTxEDRWaveform = txWaveform1((edrTestParams.startIndex)*sps+1:end);

    % Perform matched filtering
    rxFilt = upfirdn(idealTxEDRWaveform,filtCoeff,1,sps);

    % Remove delay and normalize filtered signal
    idealEDRSymbols = rxFilt(edrTestParams.span+1:end,1)/sqrt(sps);

    % Add frequency offset
    driftRate = maxFreqDrift/((packetDuration+edrTestParams.span)*sps); % Drift rate
    freqDrift = driftRate*(0:1:((packetDuration+edrTestParams.span)*sps-1))';% Frequency drift fo
    frequencyDelay.FrequencyOffset = freqDrift + initialFreqOffset; % Frequency offset, includes
    transWaveformCFO = frequencyDelay(txWaveform(1:(packetDuration+edrTestParams.span)*sps));

    % Add thermal noise
    noisyWaveform = thNoise(transWaveformCFO);

    % Compute initial frequency offset specified in Bluetooth RF-PHY Test Specifications
    estimatedInitFreqOff(packetCount) = helperEstimateInitialFreqOffset(noisyWaveform,sps);

    % Compensate initial frequency offset in the received waveform
    pfOffset = comm.PhaseFrequencyOffset('SampleRate',edrTestParams.sampleRate,'FrequencyOffset'
    freqTimeSyncRcv = pfOffset(noisyWaveform);

    % Remove access code, packet header, and guard time from packet
    rxEDRWaveform = freqTimeSyncRcv((edrTestParams.startIndex)*sps+1:end);

    % Perform matched filtering
    rxFilt = upfirdn(rxEDRWaveform,filtCoeff,1,sps);
    receivedEDRSymbols = rxFilt(edrTestParams.span+1:end,1)/sqrt(sps);

    % Compute DEVM values
    [rmsDEVM,rmsDEVMSymbol,samplingFreq] = ...
        helperEDRModulationTestMeasurements(receivedEDRSymbols,idealEDRSymbols,edrTestParams);

    % Accumulate measured values for 200 blocks as specified in Bluetooth RF-PHY Test Specificati
    blockCount = blockCount + edrTestParams.numDEVMBlocks;
    symDEVM(((packetCount-1)*edrTestParams.numDEVMBlocks*edrTestParams.blockLength)+1:(packetCou
        *edrTestParams.blockLength) = rmsDEVMSymbol(1:edrTestParams.numDEVMBlocks*edrTestParams.
    blockRMSDEVM(((packetCount-1)*edrTestParams.numDEVMBlocks)+1:((packetCount)*edrTestParams.nu
        rmsDEVM(1:edrTestParams.numDEVMBlocks);
    estimatedBlockFreqDrifts(((packetCount-1)*edrTestParams.numDEVMBlocks)+1:((packetCount)*edrTe
```

```
        samplingFreq(1:edrTestParams.numDEVMBlocks);
end
```

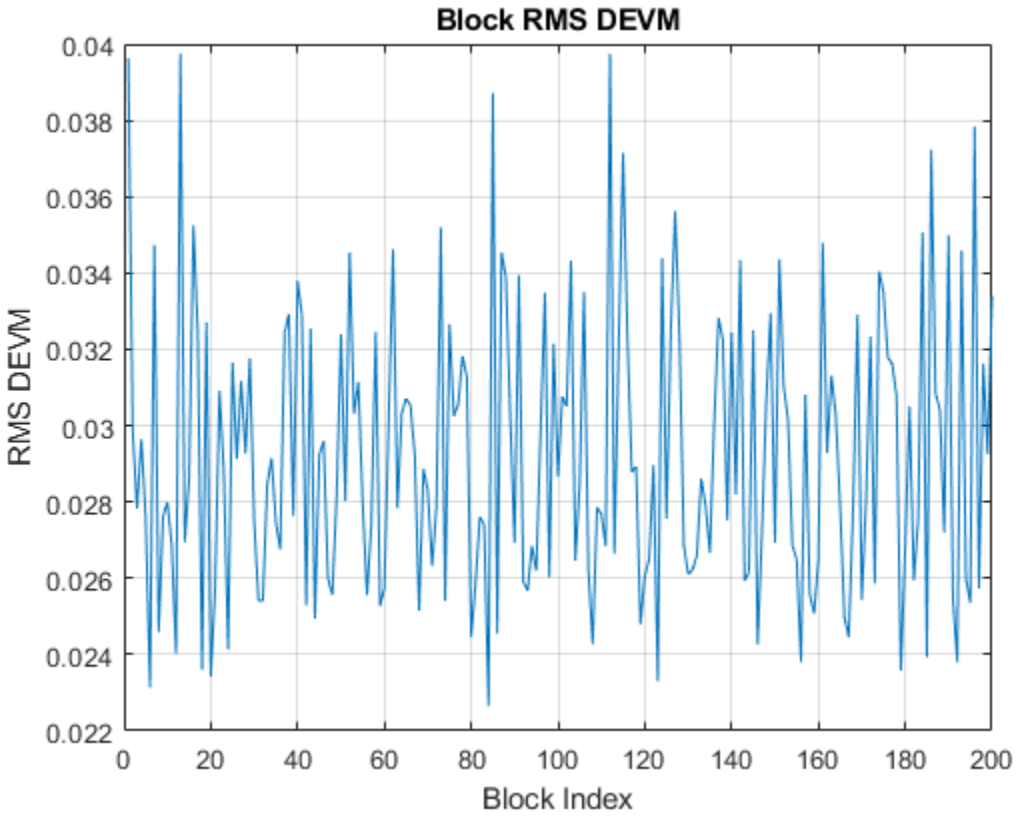
Use the helperEDRModulationTestVerdict.m helper function to verify whether the measurements are within the specified limits and display the verdict.

```
helperEDRModulationTestVerdict(phyMode, ...
    edrTestParams,estimatedInitFreqOff,symDEVM,blockRMSDEVM,estimatedBlockFreqDrifts)
```

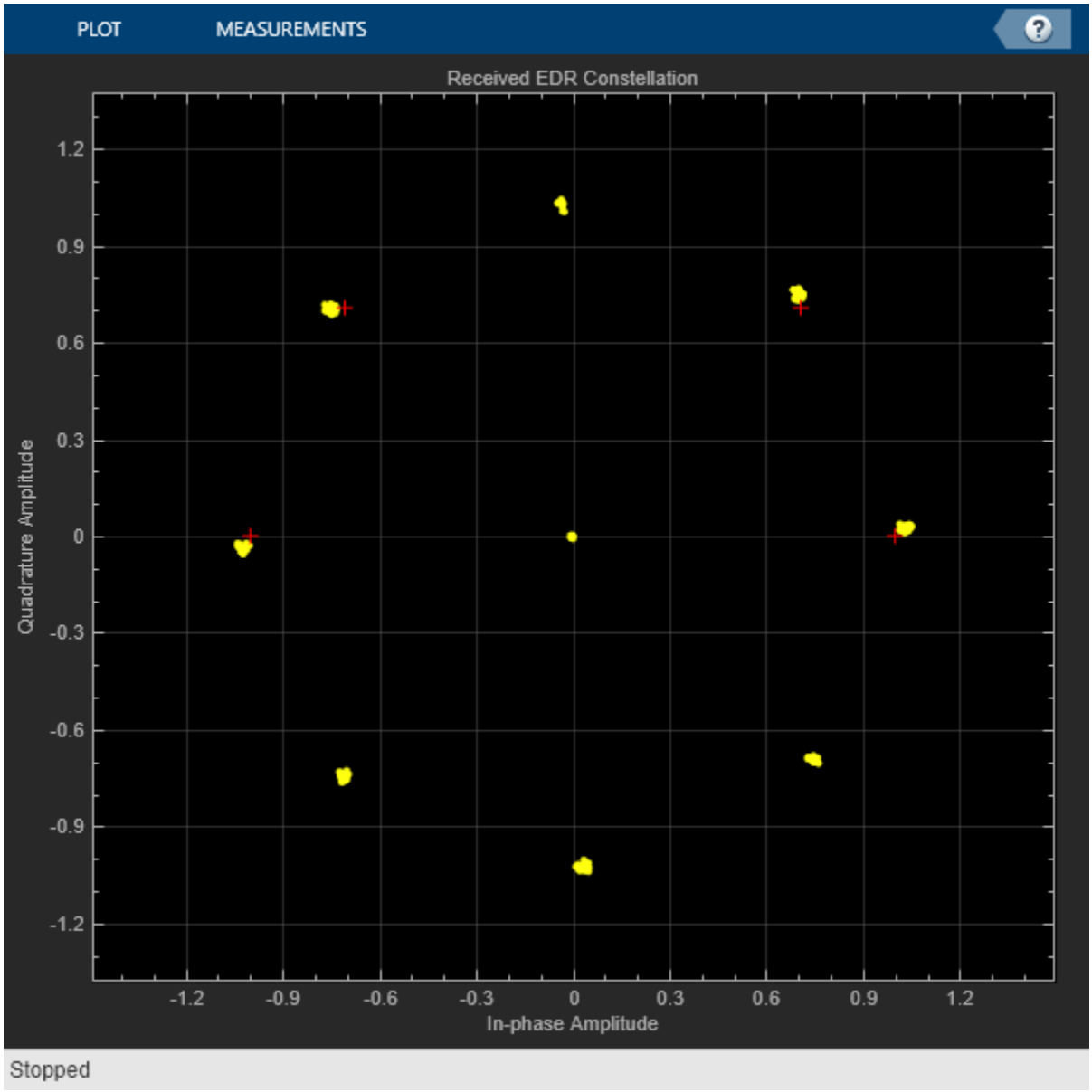
```
Modulation Accuracy Test Results:
```




```
Expected peak DEVM for all pi/4-DQPSK symbols is less than or equal to 0.35
Result: Pass
Percentage of pi/4-DQPSK symbols with DEVM less than or equal to 0.3 is 100
Expected percentage of pi/4-DQPSK symbols with DEVM less than or equal to 0.3 is 99 %
Result: Pass
```




```
       Expected RMS DEVM for all pi/4-DQPSK blocks is less than or equal to 0.2
       Result: Pass
Carrier Frequency Stability Test Results:
       Expected initial frequency offset range: [-75 kHz, 75 kHz]
       Do estimated initial frequency offsets for all the packets fall under expected values?
       Result: Yes
       Expected sampling frequencies range: [-10 kHz, 10 kHz]
       Do estimated sampling frequencies for all the blocks fall under expected values?
       Result: Yes

% Plot the constellation diagram
if strcmp(phyMode,'EDR2M')
    refSymbols = dpskmod(0:edrTestParams.M-1,edrTestParams.M,pi/4,'gray'); % Perform pi/4-DQPSK m
else
    refSymbols = dpskmod(0:edrTestParams.M-1,edrTestParams.M,0,'gray'); % Perform 8-DPSK modulati
end
constDiag = comm.ConstellationDiagram('ReferenceConstellation',refSymbols, ...
    'Title','Received EDR Constellation');
constDiag(receivedEDRSymbols);
release(constDiag);
```



This example demonstrates the Bluetooth EDR transmitter test measurements specific to modulation accuracy and carrier frequency stability. The simulation results verify that these computed test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

The example uses these helpers:

- helperEDRModulationTestConfig.m: Configure Bluetooth test parameters
- helperEstimateInitialFreqOffset.m: Estimate initial frequency offset
- helperEDRModulationTestMeasurements.m: Compute all DEVM measurements required for test
- helperEDRModulationTestVerdict.m: Validate test measurements and display result

Selected Bibliography

1 - Bluetooth Special Interest Group (SIG). “Bluetooth RF-PHY Test Specification”, v1.2/2.0/2.0, EDR/2.1/2.1, EDR/3.0/3.0, HS (), RF.TS/3.0.H.1, Section 4.5. 2009. https://www.bluetooth.com/

2 - Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". Bluetooth Core Specification. Version 5.2, Volume 2. https://www.bluetooth.com/

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151
- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift” on page 3-71

Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift

This example shows how to perform Bluetooth® basic rate (BR) radio frequency (RF) physical layer (PHY) transmitter tests specific to modulation characteristics, carrier frequency offset, and drift using the Communications Toolbox™ Library for the Bluetooth Protocol. The test measurements compute frequency deviation, carrier frequency offset, and drift values. This example also verifies whether these test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Objectives of Bluetooth RF-PHY Tests

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by the Bluetooth Special Interest Group (SIG) include RF-PHY tests for transmitters and receivers. The objectives of these RF-PHY tests are to:

- Ensure interoperability between all Bluetooth devices.
- Ensure a basic level of system performance for all Bluetooth products.

Each test case has a specified test procedure. The expected outcome must be met by the implementation under test (IUT).

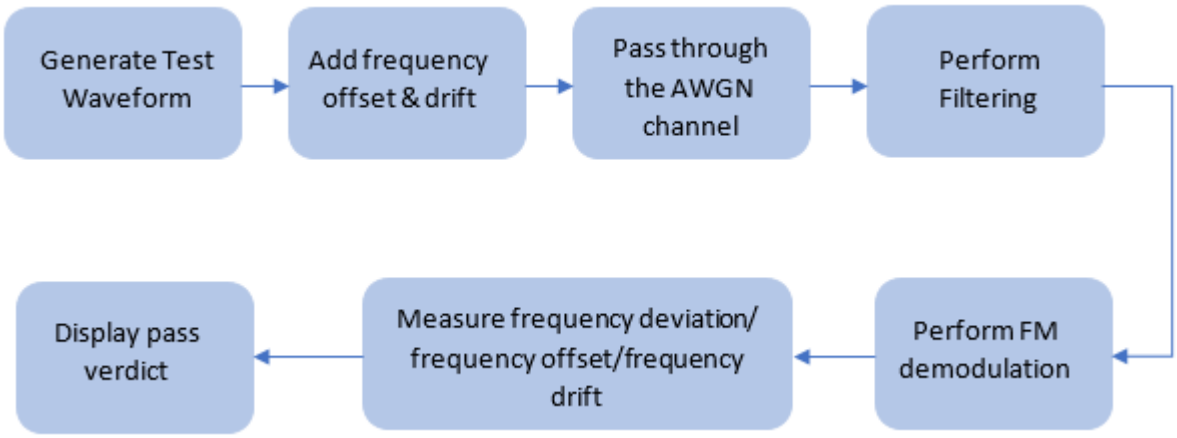
RF-PHY Transmitter Tests

The main goal of the transmitter test measurements is to ensure that the transmitter characteristics are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0]. This example includes transmitter tests relevant to modulation characteristics, carrier frequency offset, and drift. This table shows various RF-PHY transmitter tests performed in this example.

Conformance Test	Test Case ID	Test Purpose
Modulation characteristics	RF/TRM/CA/BV-07-C	Verification of the modulation index
Initial carrier frequency tolerance	RF/TRM/CA/BV-08-C	Verification of the transmitter carrier frequency
Carrier frequency drift	RF/TRM/CA/BV-09-C	Verification of the transmitter center frequency drift within a packet

RF-PHY Transmitter Test Procedure

This block diagram summarizes the test procedure for transmitter tests relevant to modulation characteristics, carrier frequency offset, and drift.



Generate DH/DM packets and pass through the `bluetoothWaveformGenerator` function to generate Bluetooth test waveforms. This table shows the test waveforms and packet type(s) required for different test IDs:

Test Case ID	Test Waveforms	Packet Type
RF/TRM/CA/BV-07-C	Generate two Bluetooth DM or DH packets with the repetitive sequences of 11110000b and 10101010b in transmission order	'DH1', 'DH3', 'DH5', 'DM1', DM3', 'DM5'
RF/TRM/CA/BV-08-C	Generate one Bluetooth DH1 packet with a repetitive sequence of PRBS9	'DH1'
RF/TRM/CA/BV-09-C	Generate one Bluetooth DH1 packet with a repetitive sequence of 10101010b	'DH1', 'DH3', 'DH5'

Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

You can change the `txTestID`, `packetType`, `initFreqOffset`, `maxFreqDrift` and `sps` parameters based on the transmitter test, packet type, initial frequency offset, maximum frequency drift, and samples per symbol, respectively.

```
txTestID = 'RF/TRM/CA/BV-07-C';
packetType = 'DH1';              % Select packet type as per transmitter test cas
initFreqOffset = 0;              % Initial frequency offset in Hz
maxFreqDrift = 0;                % Maximum frequency drift in Hz, [-25e3, 25e3] f
                                 % [-40e3, 40e3] for three and five slot packets
sps = 32;                        % Minimum of 4 samples per symbol as per test spec
```

Generate Test Parameters

Test parameters are generated based on transmitter test, packet type, initial frequency offset, maximum frequency drift and samples per symbol. To generate payload, packet duration, and waveform configuration parameters, use helperBRModulationTestPacketConfig.m function. To design channel filter based on the sample rate, use helperModulationTestFilterDesign.m function. To add frequency offset and thermal noise, create and configure`comm.PhaseFrequencyOffset` and `comm.ThermalNoise` System objects, respectively.

```
[payload,packetDuration,cfg] = helperBRModulationTestPacketConfig(txTestID,packetType,sps);
filtDesign = helperModulationTestFilterDesign('BR',sps); % Design channel filter
driftRate = maxFreqDrift/(packetDuration*sps);           % Drift rate
freqDrift = driftRate*(0:1:(packetDuration*sps-1))';     % Frequency drift for the packet
freqOffset = freqDrift + initFreqOffset;                 % Frequency offset, includes initial fre

% Create a phase frequency offset System object
sampleRate = sps*1e6;                                    % Sample rate in Hz
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',freqOffset,'SampleRate',sampleRate);

% Create a thermal noise System object
NF = 12;                                                 % Noise figure (dB)
thNoise = comm.ThermalNoise('NoiseMethod','Noise figure', ...
                            'SampleRate',sampleRate, ...
                            'NoiseFigure',NF);
```

Simulate Transmitter Tests

To simulate the transmitter tests, follow these steps:

1. Generate a Bluetooth BR waveform for the selected packet type and waveform configuration.
2. Add frequency offset, which includes initial frequency offset and drift to the waveform.
3. Add noise to the waveform.
4. Perform filtering on the noisy waveform.
5. Perform FM demodulation on the filtered waveform.
6. Perform test measurements and display the pass verdict.

```
filtWaveform = zeros(packetDuration*sps,size(payload,2)); % Initialization
for i = 1:size(payload,2)
    txWaveform = bluetoothWaveformGenerator(payload(:,i),cfg);
    txWaveformValid = txWaveform(1:packetDuration*sps);
    wfmFreqOffset = pfo(txWaveformValid);
    wfmChannel = thNoise(wfmFreqOffset);
    filtWaveform(:,i) = conv(wfmChannel,filtDesign.Coefficients.','same');
end
```

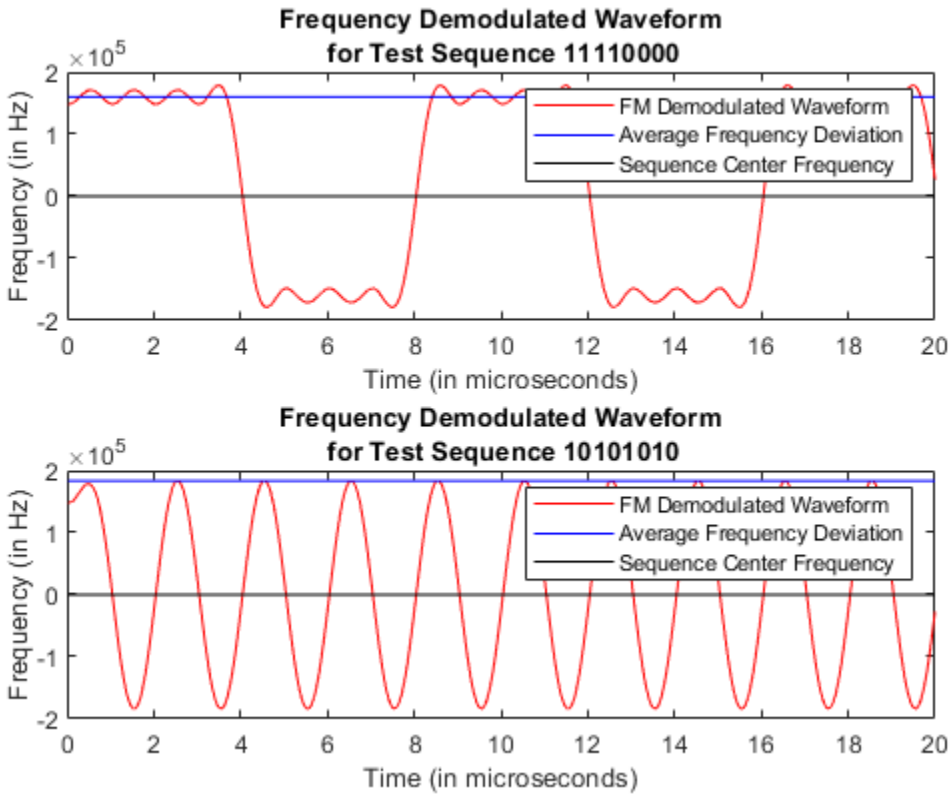
Based on the transmitter test, the helperBRModulationTestMeasurements.m function performs FM demodulation and returns these values:

- RF/TRM/CA/BV-07-C: Returns the frequency deviations and center frequencies for the two test sequences, `freq1`, `freq2`, respectively and maximum frequency deviation for the second test sequence, `freq3`.
- RF/TRM/CA/BV-08-C: Returns the initial frequency offset, `freq1`.
- RF/TRM/CA/BV-09-C: Returns the initial frequency offset and frequency drift, `freq1` and `freq2`, respectively.

```
[waveform,freq1,freq2,freq3] = helperBRModulationTestMeasurements(filtWaveform,txTestID,sps,packe
```

The helperBRModulationTestVerdict.m function verifies whether the test measurements are within the specified limits and displays the verdict.

```
helperBRModulationTestVerdict(waveform,txTestID,sps,freq1,freq2,freq3)
```




```
Test sequence: 11110000
    Measured average frequency deviation: 160 kHz
    Expected average frequency deviation range: [140 kHz, 175 kHz]
    Result: Pass
Test sequence: 10101010
    Expected 99.9% of all maximum frequency deviation greater than 115 kHz
    Result: Pass
Ratio of frequency deviations in the two test sequences: 1.1462
Expected Ratio greater than 0.8
    Result: Pass
```

This example demonstrates the Bluetooth BR transmitter test measurements specific to modulation characteristics, carrier frequency offset, and drift. The simulation results verify that these computed test measurement values are within the limits specified by Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

The example uses these helpers:

- helperBRModulationTestPacketConfig.m: Configures Bluetooth transmitter test parameters
- helperModulationTestFilterDesign.m: Designs channel filter
- helperBRModulationTestMeasurements.m: Measures frequency deviation, carrier frequency offset and drift
- helperModulationCharacteristicsTest.m: Performs modulation characteristics test
- helperBRModulationTestVerdict.m: Validates test measurement values and displays the result
- helperBluetoothPacketDuration.m: Returns the packet duration

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). “Bluetooth RF-PHY Test Specification”, v1.2/2.0/2.0, EDR/2.1/2.1, EDR/3.0/3.0, HS (), RF.TS/3.0.H.1, Section 4.5. 2009. https://www.bluetooth.com.

2 Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". Bluetooth Core Specification. Version 5.2, Volume 2. https://www.bluetooth.com.

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151
- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62

End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping

This example presents an end-to-end simulation to demonstrate how adaptive frequency hopping (AFH) alters the frequency hopping sequence in Bluetooth® basic rate (BR) and enhanced data rate (EDR) physical layer (PHY) and minimizes the impact of WLAN interference using the Communications Toolbox™ Library for the Bluetooth® Protocol. Unlike basic frequency hopping, AFH excludes the Bluetooth channels that are sources of interference. By excluding these channels from the list of available channels, AFH reassigns the transmission and reception of packets on channels with relatively less interference. The simulation results in the example show that the packet error rate (PER) and bit error rate (BER) values of the Bluetooth PHY simulation with WLAN interference are less with AFH as compared with basic frequency hopping. Also, the example shows the selected channel index per slot for basic frequency hopping and AFH. The power spectral density of Bluetooth BR/EDR waveforms with WLAN interference is visualized using the spectrum analyzer.

Bluetooth BR/EDR PHY

The Bluetooth standard specifies two PHY modes: BR and EDR. The Communications Toolbox™ Library for the Bluetooth Protocol support package enables you to model Bluetooth BR/EDR communication system links, as specified in the Bluetooth Core Specification [1] on page 3-0 .

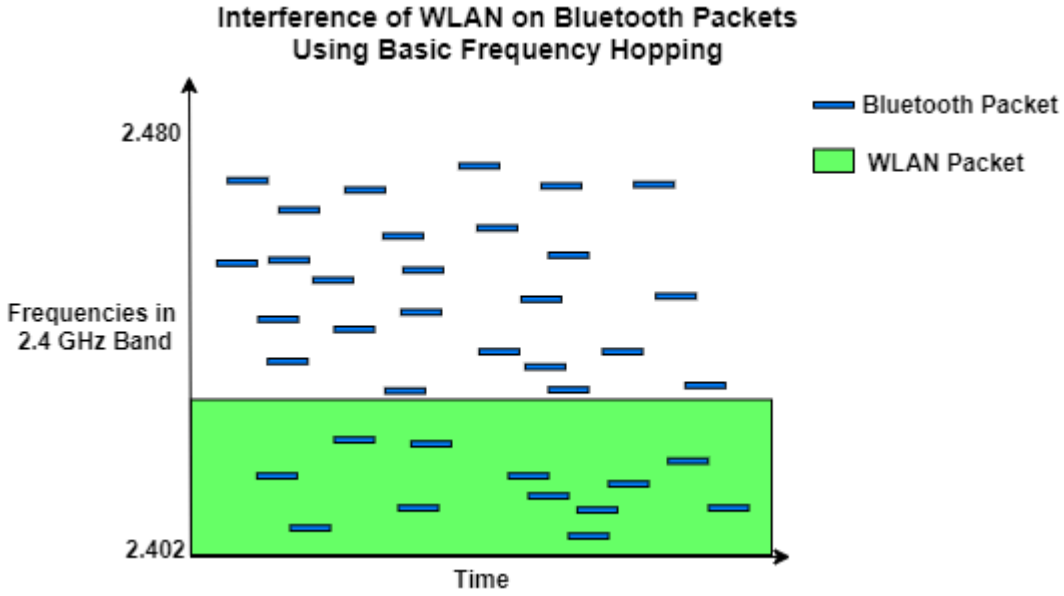
The Bluetooth BR mode is mandatory, whereas the EDR mode is optional. The Bluetooth BR/EDR radio implements a 1600 hops/s frequency hopping spread spectrum (FHSS) technique. The radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth channel has a bandwidth of 1 MHz. Each channel is centered at (2402 + k) MHz, where k = 0, 1, ..., 78. The modulation technique on BR and EDR mode payloads is Gaussian frequency shift-keying (GFSK) and differential phase shift-keying (DPSK), respectively. The baud rate is 1 MSymbols/s. The Bluetooth BR/EDR radio uses time division duplex (TDD) in which data transmission occurs in one direction at one time. The transmission alternates in two directions, one after the other.

For more information about the Bluetooth BR/EDR radio and the protocol stack, see “Bluetooth Protocol Stack” on page 13-7. For more information about Bluetooth BR/EDR packet structures, see “Bluetooth Packet Structure” on page 13-23.

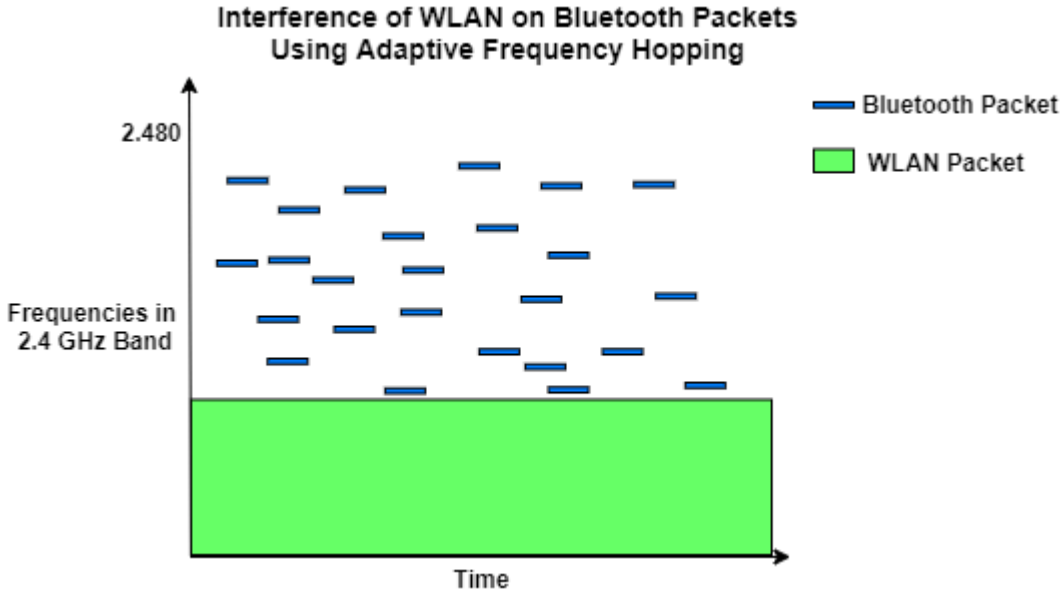
Adaptive Frequency Hopping

The objective of using FHSS in Bluetooth is to provide diversity that allows to minimize BER even if the interfering networks or the physical environment renders some channels unusable. Frequency hopping techniques can either implement a fixed sequence of channel hops such as with basic frequency hopping or adapt its hopping sequence dynamically with AFH to varying interference conditions.

Prior to AFH capability, Bluetooth devices implemented the basic frequency hopping scheme. In this approach, a Bluetooth radio hops in a pseudo-random way at the rate of 1600 hops/s. When another wireless device operating in the same 2.4 GHz band comes into the environment, the basic frequency hopping scheme results in occasional collisions. For example, Bluetooth and WLAN are two such networks that operate in the 2.4 GHz frequency band. Bluetooth and WLAN radios often operate in the same physical scenario and on the same device. In these cases, Bluetooth and WLAN transmissions can interfere with each other. This interference impacts the performance and reliability of both networks. This figure shows a scenario in which Bluetooth and WLAN packet transmissions interfere with each other.



AFH enables Bluetooth to minimize collisions by avoiding sources of interference and excluding them from the list of available channels.This figure shows the previous scenario with AFH enabled.



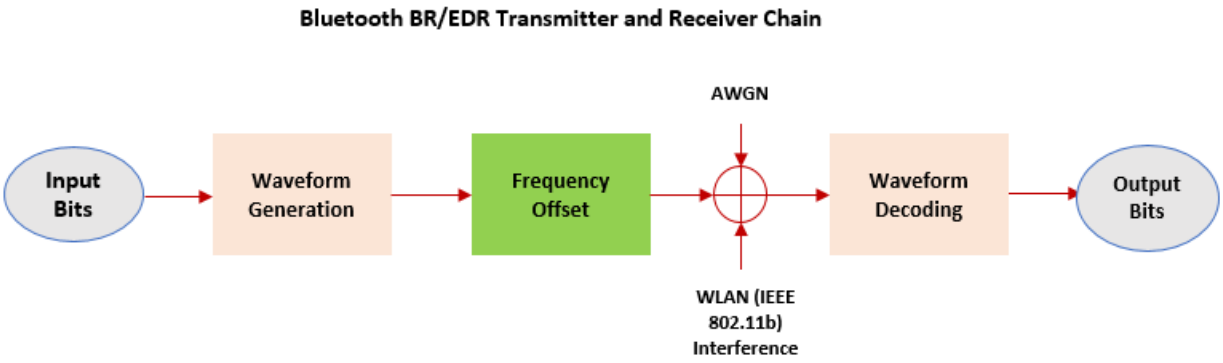
This procedure of remapping involves reducing the number of channels to be used by Bluetooth. The Bluetooth Core Specifications [1] on page 3-0 require at least 20 channels for Bluetooth transmissions.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed.
commSupportPackageCheck('BLUETOOTH');
```

Bluetooth BR/EDR Transmitter and Receiver Chain

This example demonstrates an end-to-end Bluetooth BR/EDR waveform processing by using the frequency hopping mechanism defined in the Bluetooth Core Specification [1] on page 3-0 . The generated Bluetooth BR/EDR waveform is frequency modulated and then distorted with WLAN interference. This flowchart shows the Bluetooth transmitter and receiver chain.



Transmitter chain

1. Select a channel index for the transmission.
2. Generate random input bits.
3. Generate a Bluetooth BR/EDR waveform.
4. Apply a frequency offset based on the selected channel index.

Receiver chain

1. Select a channel index for reception.
2. Apply a frequency offset based on the selected channel index.
3. Decode the Bluetooth BR/EDR waveform to get the output bits.

Wireless Channel

1. Add WLAN (IEEE 802.11b) interference to Bluetooth BR/EDR waveform.
2. Add AWGN to Bluetooth BR/EDR waveform.

Results

The example displays these results for basic frequency hopping and AFH.

- The PER and BER for the simulations performed under an additive white gaussian noise (AWGN) channel for a given bit energy to noise density ratio (Eb/No) value
- The received signal spectrum and the spectrogram of the channel
- A plot displaying the selected channel index per transmission or reception slot

Configure Simulation Parameters

Configure the desired Bluetooth packet type, payload length, PHY mode, and simulation time.

```
simulationTime = 2*1e6; % Simulation time in microseconds
packetType     = DM1; % Specify baseband packet type
mode           = BR; % Specify PHY mode ('BR', 'EDR2M', 'EDR3M')
payloadLength  = 10; % Length of baseband packet in bytes
```

Configure Frequency Hopping

Use the `bluetoothFrequencyHop` object to select a channel index for the transmission and reception of Bluetooth BR/EDR waveforms.

```
% Bluetooth frequency hopping
frequencyHop = bluetoothFrequencyHop;
frequencyHop.SequenceType = 'Connection Adaptive';
```

Configure Bluetooth PHY

Use the helperBluetoothPHY helper object to model the Bluetooth BR/EDR waveform transmission and reception.

```
% Configure Bluetooth PHY transmission
phyTx = helperBluetoothPHY;
phyTx.Mode = mode;

% Configure Bluetooth PHY reception
phyRx = helperBluetoothPHY;
phyRx.Mode = mode;
```

Configure Channel and WLAN Interference:

Use helperBluetoothChannel object to configure the wireless channel. You can set the `EbNo` value for the AWGN channel. To generate the interfering WLAN waveform, use the helperBluetoothGenerateWLANWaveform function. Specify the sources of WLAN interference by using `wlanInterferenceSource` parameter. The WLAN signal is present between -10 to 10 MHz throughout the simulation. Use one of these options to specify the source of WLAN interference.

- ‘Generated’: To add WLAN signal from the WLAN Toolbox™, select this option.
- ‘BasebandFile’: To add a WLAN signal from a baseband file (`.bb`), select this option and specify the baseband file name in the `WLANBBFilename` property. If you do not specify the `.bb` file, the example uses the default `.bb` file, `WLANNonHTDSSS.bb`, to add the WLAN signal.
- ‘None’: No WLAN signal is added.

AWGN is present throughout the simulation.

```
% Specify as one of 'Generated' | 'BasebandFile' | 'None'
wlanInterferenceSource = BasebandFile;
wlanBBFilename = 'WLANNonHTDSSS.bb'; % Default baseband file
% Configure wireless channel
channel = helperBluetoothChannel;
channel.EbNo = 22; % Ratio of energy per bit (Eb) to the spectral noise density (No) in dB
channel.SIR = -20; % Signal to interference ratio in dB
```

```
if ~strcmpi(wlanInterferenceSource, 'None')
    % Generate the WLAN waveform
    wlanWaveform = helperBluetoothGenerateWLANWaveform(wlanInterferenceSource, wlanBBFilename);
    % Add the WLAN interference to Bluetooth channel
    addWLANWaveform(channel, wlanWaveform);
end
```

Simulation Setup

Initialize parameters to perform the end-to-end Bluetooth BR/EDR simulation.

```
slotTime = 625; % Bluetooth slot duration in microseconds
% Simulation time in terms of slots
numSlots = floor(simulationTime/slotTime);
% Slot duration, including transmission and reception
slotValue = phyTx.slotsRequired(packetType)*2;
% Number of Master transmission slots
numMasterTxSlots = floor(numSlots/slotValue);
% Total number of Bluetooth physical channels
numBtChannels = 79;
% errorsBasic and errorsAdaptive store relevant bit and packet error
% information per channel. Each row stores the channel index, bit errors,
% packet errors, total bits, and BER per channel. errorsBasic and
% errorsAdaptive arrays store these values for basic frequency hopping
% and AFH, respectively.
[errorsBasic, errorsAdaptive] = deal(zeros(numBtChannels,5));
% Initialize first column with channel numbers
[errorsBasic(:,1), errorsAdaptive(:,1)] = deal(0:78);
% Initialize variables for calculating PER and BER
[berBasic, berAdaptive, bitErrors] = deal(0);
badChannels = zeros(1,0);
totalTransmittedPackets = numMasterTxSlots;
% Number of bits per octet
octetLength = 8;
% Sample rate and input clock used in PHY processing
samplePerSymbol = 88;
symbolRate = 1e6;
sampleRate = symbolRate*samplePerSymbol;
inputClock = 0;
% Store hop index
hopIndex = zeros(1, numMasterTxSlots);
% Index to hop index vector
hopIdx = 1;
% Baseband packet structure
basebandData = struct(...
    'LTAddr',1,             ... % Logical transport address
    'PacketType',packetType,... % Packet type
    'Payload',zeros(1,phyTx.MaxPayloadSize), ... % Payload
    'PayloadLength',0,  ... % Payload length
    'LLID',[0; 0],      ... % Logical link identifier
    'SEQN',0,           ... % Sequence number
    'ARQN',1,           ... % Acknowledgment flag
    'IsValid',false);   ... % Flag to identify the status of
    % cyclic redundancy check (CRC) and
% header error control (HEC)

% Bluetooth signal structure
bluetoothSignal = struct(...
```

```
    'PacketType',packetType, ... % Packet type
    'Waveform',[],           ... % Waveform
    'NumSamples',[],         ... % Number of samples
    'SampleRate',sampleRate, ... % Sample rate
    'SamplesPerSymbol',samplePerSymbol,      ... % Samples per symbol
    'Payload',zeros(1,phyTx.MaxPayloadSize), ... % Payload
    'PayloadLength',0, ... % Payload length
    'SourceID',0,      ... % Source identifier
    'Bandwidth',1,     ... % Bandwidth
    'NodePosition',[0 0 0], ... % Node position
    'CenterFrequency',centerFrequency(phyTx), ... % Center frequency
    'StartTime',0, ... % Waveform start time
    'EndTime',0,   ... % Waveform end time
    'Duration',0); ... % Waveform duration

% Clock ticks(one slot is 2 clock ticks)
clockTicks = slotValue*2;
```

To visualize the Bluetooth BR/EDR waveforms, create a `dsp.SpectrumAnalyzer` System object™.

```
% Spectrum analyzer for basic frequency hopping
spectrumAnalyzerBasic = dsp.SpectrumAnalyzer(...
    'Name','Bluetooth Basic Frequency Hopping', ...
    'ViewType','Spectrum and spectrogram', ...
    'TimeResolutionSource','Property', ...
    'TimeResolution',0.0005, ...
    'SampleRate',sampleRate, ...
    'TimeSpanSource','Property', ...
    'TimeSpan', 0.05, ...
    'FrequencyResolutionMethod', 'WindowLength', ...
    'WindowLength', 512, ...
    'AxesLayout', 'Horizontal', ...
    'FrequencyOffset',2441*1e6, ...
    'ColorLimits',[-20 15]);

% Spectrum analyzer for AFH
spectrumAnalyzerAdaptive = dsp.SpectrumAnalyzer(...
    'Name','Bluetooth Adaptive Frequency Hopping', ...
    'ViewType','Spectrum and spectrogram', ...
    'TimeResolutionSource','Property', ...
    'TimeResolution',0.0005, ...
    'SampleRate',sampleRate, ...
    'TimeSpanSource','Property', ...
    'TimeSpan',0.05, ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',512, ...
    'AxesLayout','Horizontal', ...
    'FrequencyOffset',2441*1e6, ...
    'ColorLimits',[-20 15]);
```

Simulations

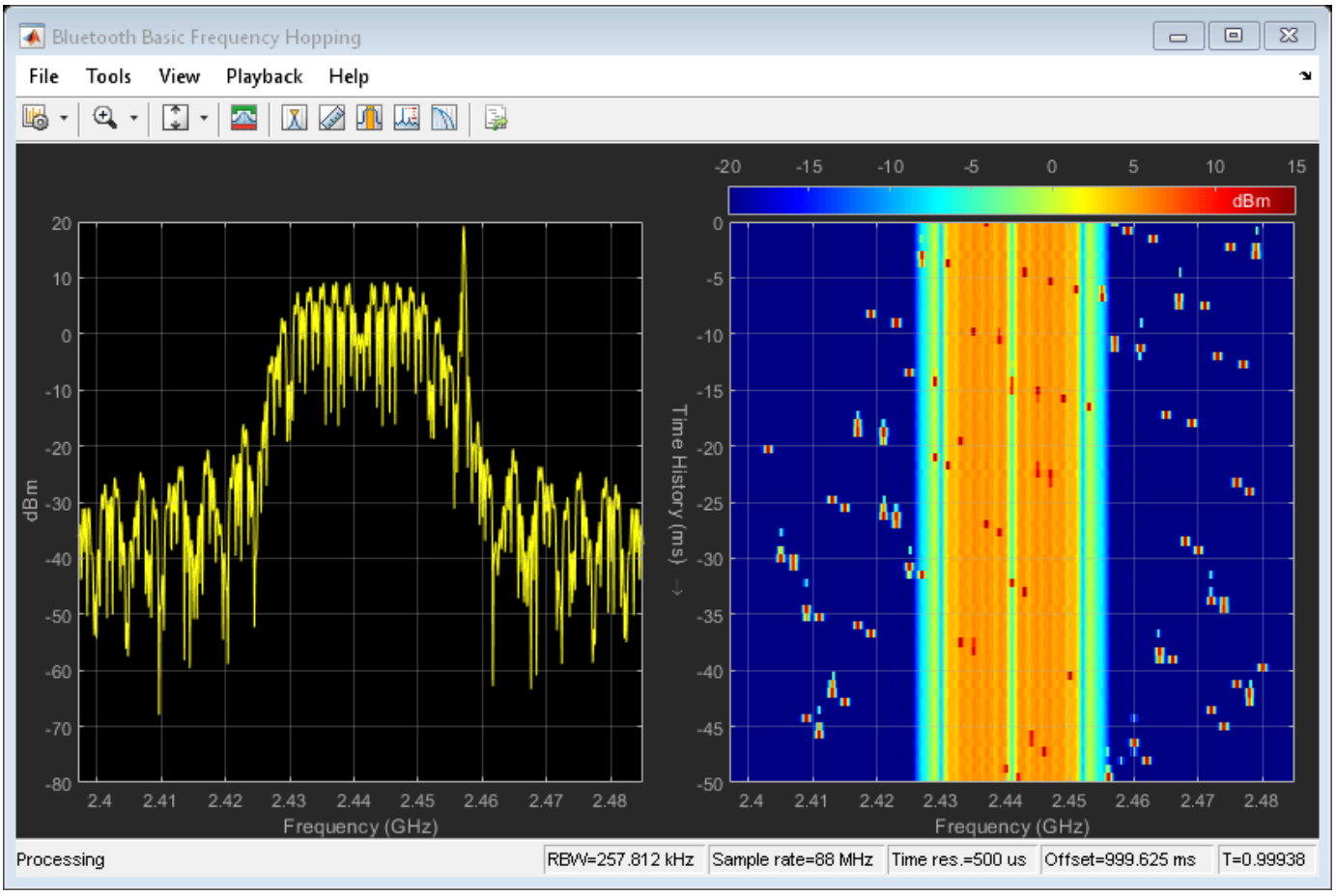
The Bluetooth transmitter and receiver chain is simulated using basic frequency hopping and AFH. Using per channel PER and BER results of basic frequency hopping, derive a list of used channels. The list of used channels is fed as an input to the simulation using AFH.

Basic Frequency Hopping

The simulation runs for all the specified number of Master transmission slots. Simulates the transmitter chain, receiver chain, and channel for each slot. At the end of the simulation, the example computes the PER and BER for all the Bluetooth BR/EDR waveforms.

```
sprev = rng('default'); % Set random number generator seed
for slotIdx = 0:slotValue:numSlots-slotValue
    % Update clock
    inputClock = inputClock + clockTicks;

    % Frequency hopping
    [channelIndex,~] = nextHop(frequencyHop,inputClock);
    % PHY transmission
    stateTx = 1; % Transmission state
    TxBits = randi([0 1],payloadLength*octetLength,1);
    basebandData.Payload = TxBits;
    basebandData.PayloadLength = payloadLength;
    % Generate whiten initialization vector from clock
    clockBinary = comm.internal.utilities.de2biBase2RightMSB(inputClock,28);
    whitenInitialization = [clockBinary(2:7)'; 1];
    % Update the PHY with request from the baseband layer
    updatePHY(phyTx,stateTx,channelIndex,whitenInitialization,basebandData);
    % Initialize and pass elapsed time as zero
    elapsedTime = 0;
    [nextTxTime,btWaveform] = run(phyTx,elapsedTime); % Run PHY transmission
    run(phyTx, nextTxTime); % Update next invoked time

    % Channel
    bluetoothSignal.Waveform = btWaveform;
    bluetoothSignal.NumSamples = numel(btWaveform);
    bluetoothSignal.CenterFrequency = centerFrequency(phyTx);
    channel.ChannelIndex = channelIndex;
    bluetoothSignal = run(channel,bluetoothSignal,mode);
    distortedWaveform = bluetoothSignal.Waveform;

    % PHY reception
    stateRx = 2; % Reception state
    % Update the PHY with request from the baseband layer
    updatePHY(phyRx,stateRx,channelIndex,whitenInitialization);
    [nextRxTime,~] = run(phyRx,elapsedTime,bluetoothSignal);
    bluetoothSignal.NumSamples = 0;
    run(phyRx,nextRxTime,bluetoothSignal); % Run PHY reception
    chIdx = channelIndex + 1;

    % Calculate error rate upon successful decoding the packet
    if phyRx.Decoded
        rxBitsLength = phyRx.DecodedBasebandData.PayloadLength*octetLength;
        RxBits = phyRx.DecodedBasebandData.Payload(1:rxBitsLength);
        % BER calculation
        txSymLength = length(TxBits);
        rxSymLength = length(RxBits);
        minSymLength = min(txSymLength,rxSymLength);
        if minSymLength > 0
            bitErrors = sum(xor(TxBits(1:minSymLength),RxBits(1:minSymLength)));
            totalBits = minSymLength;
            % Bit errors found in channel
            errorsBasic(chIdx,2) = errorsBasic(chIdx,2) + bitErrors;
            % Total bits transmitted in channel
            errorsBasic(chIdx,4) = errorsBasic(chIdx,4) + totalBits;
```

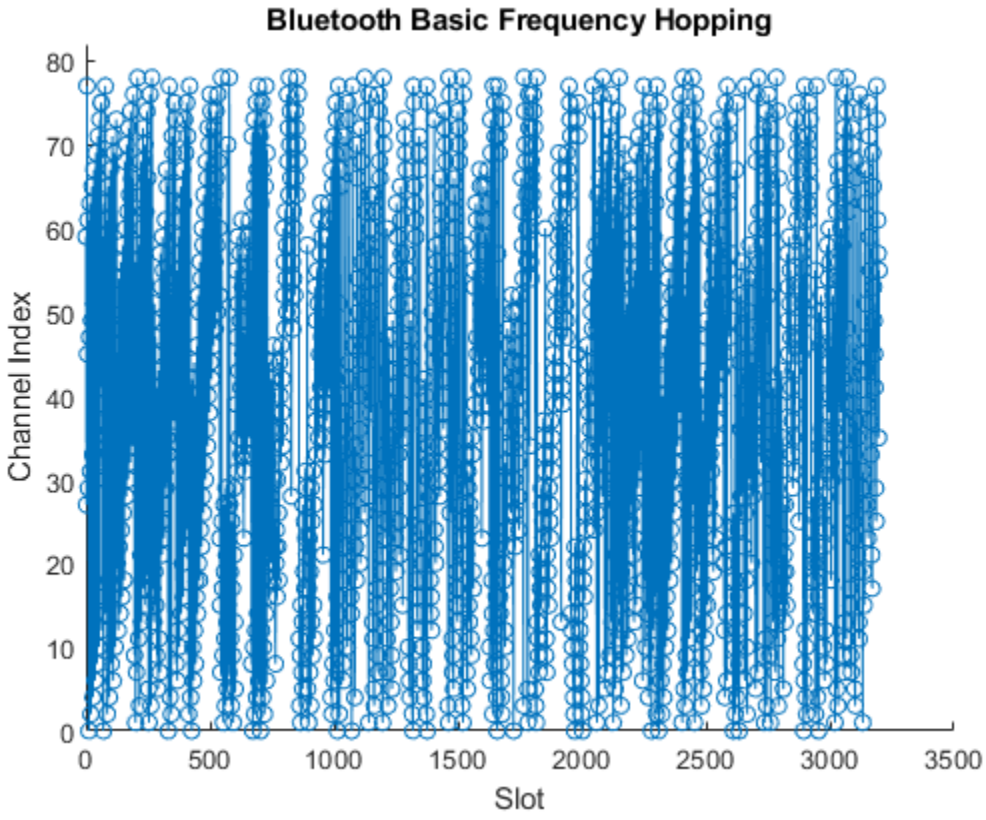
```
            end
            if ~phyRx.DecodedBasebandData.IsValid || bitErrors
                % Packet errors found in channel
                errorsBasic(chIdx,3) = errorsBasic(chIdx,3) + 1;
            end
        else
            % Packet errors found in channel
            errorsBasic(chIdx,3) = errorsBasic(chIdx,3) + 1;
        end
        hopIndex(hopIdx) = channelIndex;
        hopIdx = hopIdx + 1;

        % Plot spectrum
        spectrumAnalyzerBasic(btWaveform + wlanWaveform(1:numel(btWaveform)));
        pause(0.01);
    end
```



In the previous figure, the plot on the left shows the spectrum of the Bluetooth BR/EDR waveform distorted with WLAN interference in the frequency domain and passed through the AWGN channel. The plot on the right shows that the WLAN signal is present from –10 to 10 MHz. The results show that Bluetooth packets with the interfering WLAN signal overlap.

```
% Plot selected channel index per slot
figBasic = figure('Name','Basic frequency hopping');
axisBasic = axes(figBasic);
```

```
xlabel(axisBasic,'Slot');
ylabel(axisBasic,'Channel Index');
ylim(axisBasic,[0 numBtChannels+3]);
title(axisBasic,'Bluetooth Basic Frequency Hopping');
hold on;
plot(axisBasic,0:slotValue:numSlots-slotValue,hopIndex,'-o');
```



This plot displays the selected channel index per transmission or reception slot using basic frequency hopping. In this case all of the channels are used channels.

Channel Classification

The channels with more packet errors are marked as bad channels. Based on the bit and packet errors collected from the above simulation, calculate the PER and BER.

```
% Select 25 channels with highest packet errors as bad channels
[~,indexes] = sort(errorsBasic(:,3),'descend');
badChannelIdx = min(nnz(errorsBasic(:,3)),25);
if badChannelIdx ~= 0
    badChannels = indexes(1:badChannelIdx) - 1;
end
usedChannels = setdiff(0:numBtChannels-1,badChannels);

% BER per channel calculation
errorsBasic(:,5) = errorsBasic(:,2)./errorsBasic(:,4);
errorsBasic(:,5) = fillmissing(errorsBasic(:,5),'constant',0);
ber = nonzeros(errorsBasic(:,5));
```

```
if ~isempty(ber)
    berBasic = mean(ber);
end

% PER calculation
packetErrorsBasic = sum(errorsBasic(:,3));
perBasic = packetErrorsBasic/totalTransmittedPackets;

% Reset
hopIdx = 1;
[inputClock,bitErrors] = deal(0);
fprintf('PER of Bluetooth BR/EDR waveforms using basic frequency hopping: %.4f\n',perBasic);
```

```
PER of Bluetooth BR/EDR waveforms using basic frequency hopping: 0.2787
```

```
fprintf('BER of Bluetooth BR/EDR waveforms using basic frequency hopping: %.4f\n',berBasic);
```

```
BER of Bluetooth BR/EDR waveforms using basic frequency hopping: 0.0550
```

Adaptive Frequency Hopping

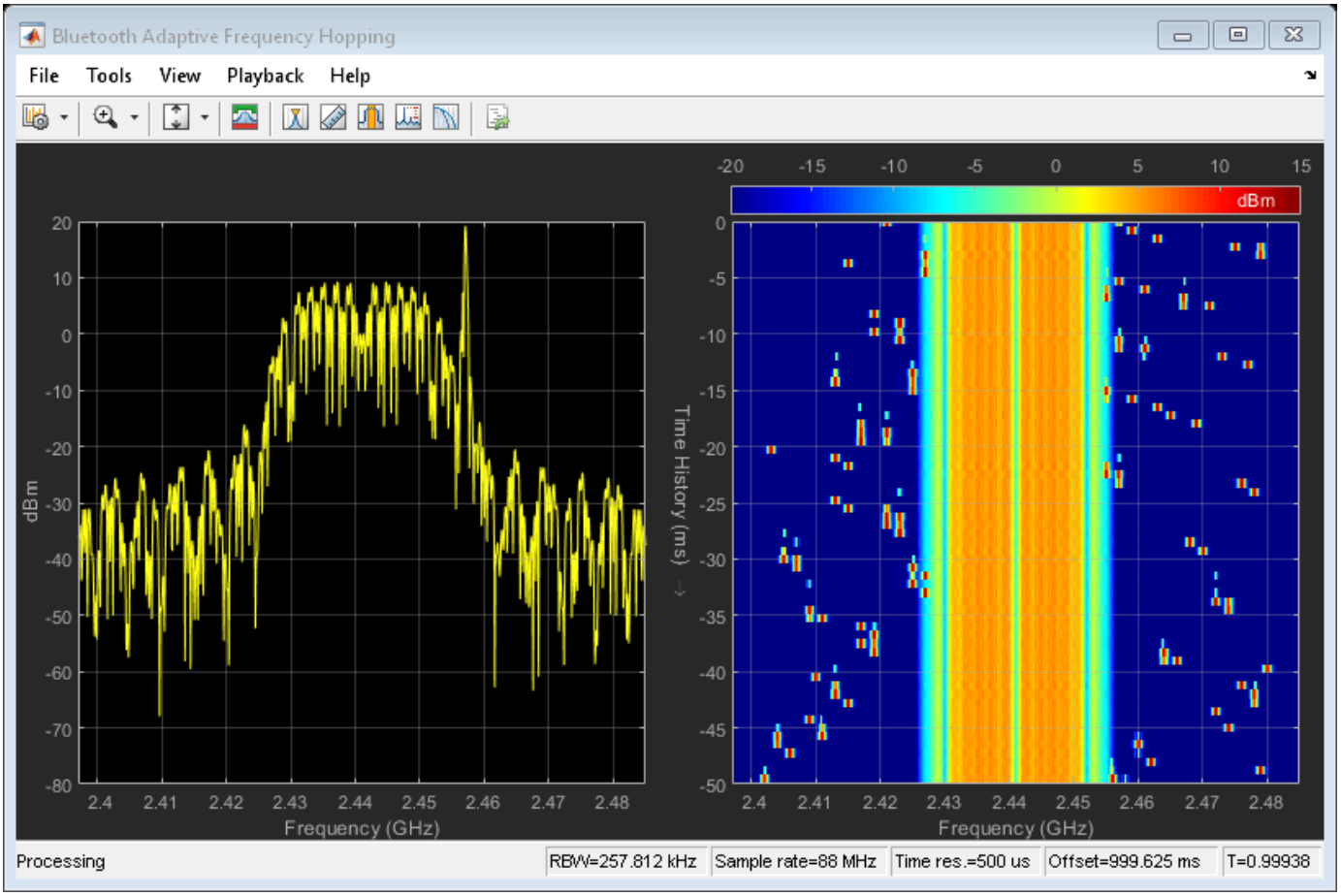
Set the value of the sequence type as `'Connection adaptive'` and specify the classified used channels.

```
frequencyHop.SequenceType = 'Connection adaptive';
frequencyHop.UsedChannels = usedChannels;
```

Run the simulation using AFH and compute the PER and BER.

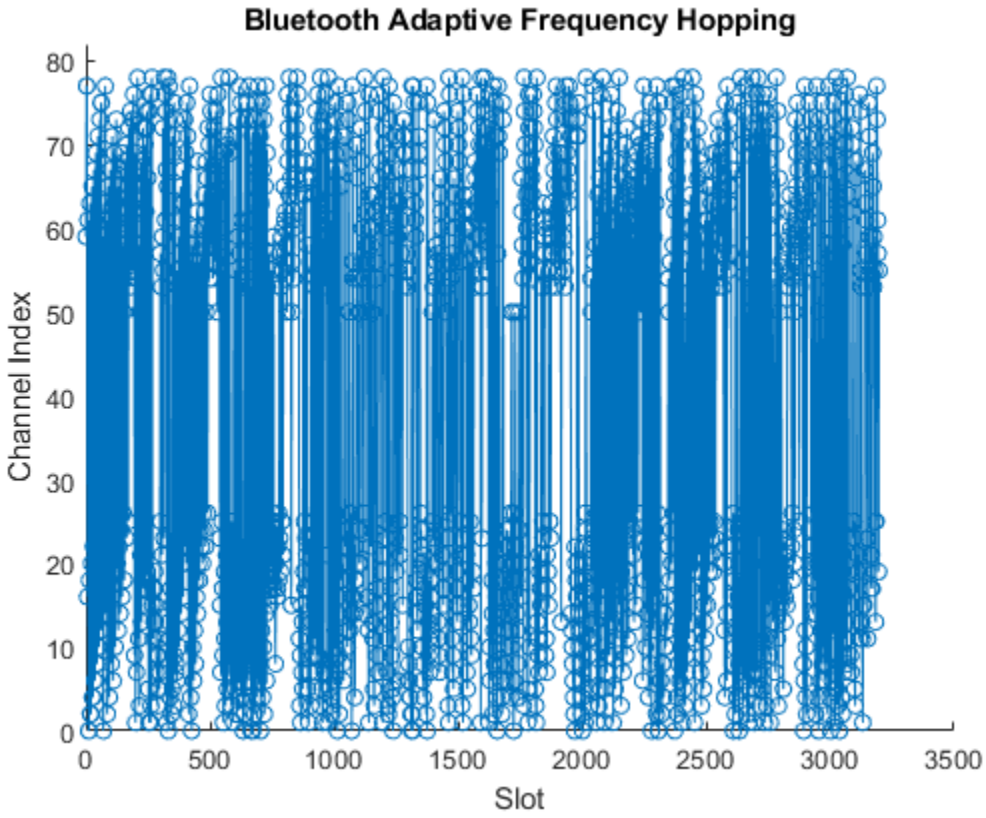
```
%rng('default'); % Set random number generator seed
for slotIdx = 0:slotValue:numSlots-slotValue
    % Update clock
    inputClock = inputClock + clockTicks;

    % Frequency hopping
    [channelIndex,~] = nextHop(frequencyHop,inputClock);

    % PHY transmission
    stateTx = 1; % Transmission state
    TxBits = randi([0 1],payloadLength*octetLength,1);
    basebandData.Payload = TxBits;
    basebandData.PayloadLength = payloadLength;
    % Generate whiten initialization vector from clock
    clockBinary = comm.internal.utilities.de2biBase2RightMSB(inputClock,28);
    whitenInitialization = [clockBinary(2:7)'; 1];
    % Update the PHY with request from the baseband layer
    updatePHY(phyTx,stateTx,channelIndex,whitenInitialization,basebandData);
    % Initialize and pass elapsed time as zero
    elapsedTime = 0;
    [nextTxTime,btWaveform] = run(phyTx,elapsedTime); % Run PHY transmission
    run(phyTx,nextTxTime); % Update next invoked time

    % Channel
    bluetoothSignal.Waveform = btWaveform;
    bluetoothSignal.NumSamples = numel(btWaveform);
    bluetoothSignal.CenterFrequency = centerFrequency(phyTx);
    channel.ChannelIndex = channelIndex;
    bluetoothSignal = run(channel,bluetoothSignal,mode);
    distortedWaveform = bluetoothSignal.Waveform;
```

```
        % PHY reception
        stateRx = 2; % Reception state
        % Update the PHY with request from the baseband layer
        updatePHY(phyRx,stateRx,channelIndex,whitenInitialization);
        [nextRxTime,~] = run(phyRx,elapsedTime,bluetoothSignal);
        bluetoothSignal.NumSamples = 0;
        run(phyRx,nextRxTime,bluetoothSignal); % Run PHY reception
        chIdx = channelIndex + 1;

        % Calculate error rate upon successful decoding the packet
        if phyRx.Decoded
            rxBitsLength = phyRx.DecodedBasebandData.PayloadLength*octetLength;
            RxBits = phyRx.DecodedBasebandData.Payload(1:rxBitsLength);
            % BER calculation
            txSymLength = length(TxBits);
            rxSymLength = length(RxBits);
            minSymLength = min(txSymLength,rxSymLength);
            if minSymLength > 0
                bitErrors = sum(xor(TxBits(1:minSymLength),RxBits(1:minSymLength)));
                totalBits = minSymLength;
                % Bit errors found in channel
                errorsAdaptive(chIdx,2) = errorsAdaptive(chIdx,2) + bitErrors;
                % Total bits transmitted in channel
                errorsAdaptive(chIdx,4) = errorsAdaptive(chIdx,4) + totalBits;
            end

            if ~phyRx.DecodedBasebandData.IsValid || bitErrors
                % Packet errors found in channel
                errorsAdaptive(chIdx, 3) = errorsAdaptive(chIdx, 3) + 1;
            end
        else
            % Packet errors found in channel
            errorsAdaptive(chIdx, 3) = errorsAdaptive(chIdx, 3) + 1;
        end
        hopIndex(hopIdx) = channelIndex;
        hopIdx = hopIdx + 1;

        % Plot spectrum
        spectrumAnalyzerAdaptive(btWaveform + wlanWaveform(1:numel(btWaveform)));
        pause(0.01);
    end
```



In the previous plot, you can observe that the transmission of Bluetooth packets does not overlap with the WLAN signal. AFH excludes the channels that are sources of WLAN interference and reassigns the transmission of Bluetooth packets on channels with relatively less interference.

```
% Plot selected channel index per slot
figAdaptive = figure('Name','Adaptive Frequency Hopping');
axisAdaptive = axes(figAdaptive);
xlabel(axisAdaptive,'Slot');
ylabel(axisAdaptive,'Channel Index');
title(axisAdaptive,'Bluetooth Adaptive Frequency Hopping');
ylim(axisAdaptive,[0 numBtChannels+3]);
hold on;
plot(axisAdaptive,0:slotValue:numSlots-slotValue,hopIndex,'-o');
```



This plot displays the selected channel index per transmission or reception slot using AFH. To minimize the packet and bit errors in the wireless channel, AFH selects only used channels for transmission or reception of Bluetooth BR/EDR waveforms

Compute the PER and BER of the Bluetooth BR/EDR waveforms with AFH.

```
% BER per channel calculation
errorsAdaptive(:,5) = errorsAdaptive(:,2)./errorsAdaptive(:,4);
errorsAdaptive(:,5) = fillmissing(errorsAdaptive(:,5),'constant',0);
ber = nonzeros(errorsAdaptive(:,5));
if ~isempty(ber)
    berAdaptive = mean(ber);
end

% PER calculation
packetErrorsAdaptive = sum(errorsAdaptive(:,3));
perAdaptive = packetErrorsAdaptive/totalTransmittedPackets;
fprintf('PER of Bluetooth BR/EDR waveforms using adaptive frequency hopping: %.4f\n',perAdaptive)

PER of Bluetooth BR/EDR waveforms using adaptive frequency hopping: 0.0625

fprintf('BER of Bluetooth BR/EDR waveforms using adaptive frequency hopping: %.4f\n',berAdaptive)

BER of Bluetooth BR/EDR waveforms using adaptive frequency hopping: 0.0020

% Restore previous setting of random number generation
rng(sprev);
```

The PER and BER values of the Bluetooth BR/EDR waveforms are less with AFH as compared with basic frequency hopping.

This example simulates an end-to-end transmitter-receiver chain to study how AFH mitigates interference between the Bluetooth BR/EDR and WLAN signals. The simulation results verify that the PER and the BER of the Bluetooth BR/EDR waveforms with WLAN interference is less with AFH as compared to basic frequency hopping.

Appendix

The example uses this feature.

- `bluetoothFrequencyHop`: Bluetooth BR/EDR channel index for frequency hopping

The example uses these helpers:

- helperBluetoothPHY: Configure and simulate Bluetooth PHY
- helperBluetoothChannel: Configure and simulate wireless channel
- helperBluetoothGenerateWLANWaveform: Generates WLAN waveform to be added as an interference to Bluetooth waveforms
- helperBluetoothWLANDSSSSpectrumMask: Calculates adjacent channel interference power using the WLAN 802.11b (DSSS) spectrum masks
- helperBluetoothPacketDuration: Calculate duration of Bluetooth packet

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification" Version 5.2.https://www.bluetooth.com.

See Also

More About

- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96
- “Packet Distribution in Bluetooth Piconet” on page 13-106
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51

End-to-End BLE PHY Simulation Using Path Loss Model, RF Impairments, and AWGN

This example uses Communications Toolbox™ Library for the Bluetooth® Protocol to perform end-to-end Bluetooth low energy (BLE) simulation for different BLE physical layer (PHY) transmission modes in the presence of the path loss model, radio front-end (RF) impairments, and additive white Gaussian noise (AWGN). The simulation results show the estimated value of the bit error rate (BER), path loss, and distance between the transmitter and receiver.

Path Loss Modeling in BLE Network

The Bluetooth Core Specifications [1] on page 3-0 defined by the Bluetooth Special Interest Group (SIG) introduced BLE to enable low-power short-range communication. BLE devices operate in the globally unlicensed industrial, scientific, and medical (ISM) band in a frequency range from 2.4 GHz to 2.485 GHz. BLE specifies a channel spacing of 2 MHz, resulting in 40 RF channels. The prominent applications of BLE include direction finding services and building intelligent internet of things (IoT) solutions to facilitate home, commercial, and industrial automation. For more information about direction finding services in BLE, see the “Bluetooth Location and Direction Finding” on page 13-37 topic.

In past few years, there has been a significant increase in designing BLE networks for a plethora of use case scenarios. To achieve high performance and quality in the BLE network, studying the propagation of the BLE signal along the link between the transmitter and the receiver is recommended. This example shows an end-to-end BLE simulation considering these factors that impact the propagation of BLE signals along the communication link between the transmitter and receiver.

- Receiver sensitivity
- Path loss model
- Transmit power
- Antenna gain

Receiver Sensitivity

Receiver sensitivity is the measure of minimum signal strength at which the receiver can detect, demodulate, and decode the waveform. The reference sensitivity level specified in the Bluetooth Core Specifications [1] on page 3-0 is -70 dBm. However, the actual sensitivity level for the receiver as per the Bluetooth Core Specifications [1] on page 3-0 is defined as the receiver input level for which the BER specified in this table is achieved.

Maximum Supported Payload Length (bytes)	BER (%)
1 to 37	0.1
38 to 63	0.064
64 to 127	0.034
128 to 255	0.017

This table shows the actual sensitivity level of the receiver for a given PHY transmission mode.

PHY Transmission Modes	Receiver Sensitivity (dBm)
LE Uncoded PHYs	≤ -70
LE Coded PHY with S = 2 coding	≤ -75
LE Coded with S = 8 coding	≤ -82

Path Loss Model

Path loss or path attenuation is the decline in the power density of a given signal as it propagates from the transmitter to receiver through space. This reduction in power density occurs naturally over the distance and is impacted by the obstacles present in the environment in which the signal is being transmitted. The path loss is generally expressed in decibels (dB) and is calculated as:

$\mathrm{PL}_{\mathrm{dB}} = P_t - P_r$.

In this equation,

- $\mathrm{PL}_{\mathrm{dB}}$ is the path loss in dB.
- P_t is the transmitted signal power in dB.
- P_r is the received signal power in dB.

Path loss models describe the signal attenuation between the transmitter and receiver based on the propagation distance and other parameters such as frequency, wavelength, path loss exponent, and antenna gains. The example considers these path loss models:

- Free-space [3] on page 3-0
- Log-distance [3] on page 3-0
- Log-normal shadowing [3] on page 3-0
- Two-ray ground reflection [3] on page 3-0
- NIST PAP 02-Task 6 [4] on page 3-0

Free-Space Path Loss Model

Free-space path loss is the attenuation of signal strength between the transmitter and receiver along the line of sight (LoS) path through free space (usually air), excluding the effect of the obstacles in the path. The free-space path loss is calculated as:

$$\mathrm{PL_{dB}} = 20\log\left(\frac{4\pi d}{\lambda}\right).$$

In this equation,

- d is the distance between the transmitter and receiver.
- λ is the signal wavelength.

Log-Distance Path Loss Model

A log-distance path loss model reflects the path loss that a signal encounters in an indoor environment such as a building. It is computed as:

$$\mathrm{PL_{dB}} = \mathrm{PL_0} + 10\gamma\log\left(\frac{d}{d_0}\right).$$

In this equation,

- $\mathrm{PL_0}$ is the path loss at the reference distance d_0.
- d is the distance between the transmitter and receiver.
- d_0 is the reference distance.
- γ is the path loss exponent.

Log-Normal Shadowing Path Loss Model

The log-normal shadowing model is an extension of log-distance path loss model. Unlike the log-distance model, the log-normal shadowing model considers the fact that the surrounding environment clutter can be vastly different at two different locations having the same transmitter-receiver separation. Measurements show that at any transmitter-receiver distance, d, the path loss at a particular location is random and distributed log normally (in dB) about the mean distance dependent value. The path loss is calculated as:

$$\mathrm{PL_{dB}}(d) = \mathrm{PL_{dB}}(d_0) + 10\gamma\log\left(\frac{d}{d_0}\right) + X_\sigma.$$

In this equation,

- $PL_{dB}(d_0)$ is the path loss at the reference distance d_0.
- d is the distance between the transmitter and receiver.
- d_0 is the reference distance.
- γ is the path loss exponent.
- X_σ is the normal or Gaussian random variable with zero mean, reflecting the attenuation caused by the flat fading.

Two-Ray Ground Reflection Model

The two-ray ground reflection model is a radio propagation model that estimates the path loss between the transmitter and receiver by considering these two signal components: LoS and the component reflected from the ground. When the transmitter and receiver antenna heights are approximately similar and the distance between the antennas is very large relative to the height of the antennas, then the path loss is calculated as:

$$PL_{\text{linear scale}} = \frac{G\,h_t^2 h_r^2}{d^4}.$$

The path loss in logarithmic scale is calculated as:

$$PL_{dB} = 40\log10(d) - 10\log10\left(G\,h_t^2 h_r^2\right).$$

In this equation,

- d is the distance between the transmitter and receiver.
- G is the product of antenna gains.
- h_t is the height of the transmitter.
- h_r is the height of the receiver.

NIST PAP02-Task 6 Model

The National Institute of Standards and Technology (NIST) conducted studies for indoor to indoor, outdoor to outdoor, and outdoor to indoor propagation paths and derived these equations for calculating the path loss:

$$PL_d = PL_0 + 10(n_0)\log10\left(\frac{d}{d_0}\right). \qquad \text{for } d \le d_1$$

$$PL_d = PL_0 + 10(n_0)\log10\left(\frac{d}{d_0}\right) + 10(n_1)\log10\left(\frac{d}{d_1}\right). \qquad \text{for } d > d_1$$

In these equations,

- PL_0 is the path loss at the reference distance d_0.
- n_0,n_1 are the path loss exponents.
- d is the distance between the transmitter and receiver.
- d_0 is the reference distance, assumed to be 1 meter in simulations.
- d_1 is the breakpoint where the path loss exponent adjusts from n_0 to n_1.

The example considers these values for different environments.

Environment	PL_0 (dB)	n_0	d_1 (m)	n_1
Home	12.5	4.2	11	7.6
Office	26.8	4.2	10	8.7

Most of these measurements for the NIST PAP02 Task 6 channel model were taken with transmitters and receivers located in hallways with distances ranging from 5 m to 45 m.

Transmit Power

Transmit power is the power of the radio frequency signal generated by the transmitter. Increasing the transmit power increases the likelihood that the signal can be transmitted over longer distances. Bluetooth supports transmit power from -20 dBm (0.01 mW) to 20 dBm (100 mW).

Antenna Gain

Antenna gain is the factor by which the antenna improves the total radiated power. Bluetooth designers can choose to implement a variety of antenna options. Bluetooth devices typically achieve an antenna gain in the range from -10 dBi to 10 dBi.

End-to-End BLE Simulation Procedure

The end-to-end BLE PHY simulations estimate the BER and the distance between the transmitter and receiver by considering a specific path loss model with RF impairments and AWGN added to the transmission packets.

For a given set of simulation parameters, obtain the signal-to-noise ratio (SNR) at the receiver by assuming a fixed noise figure. For the obtained value of SNR including the path loss, generate the BLE waveform using `bleWaveformGenerator` function. Distort the generated waveform with RF impairments and AWGN. Each packet is distorted by these RF impairments:

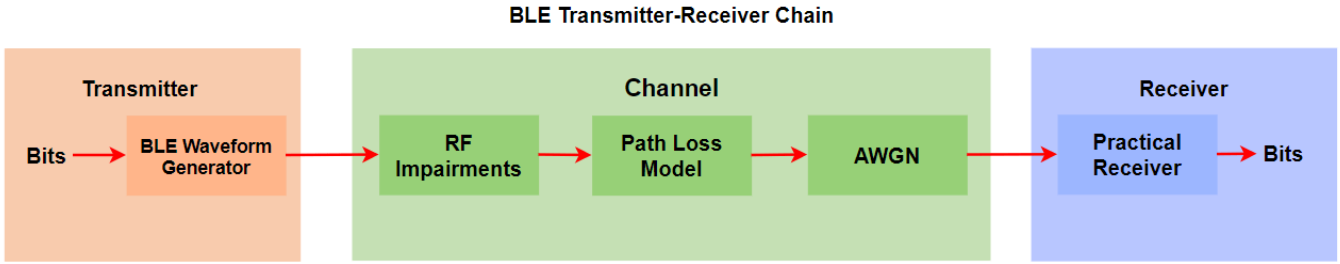
- DC offset
- Carrier frequency offset
- Carrier phase offset
- Timing drift

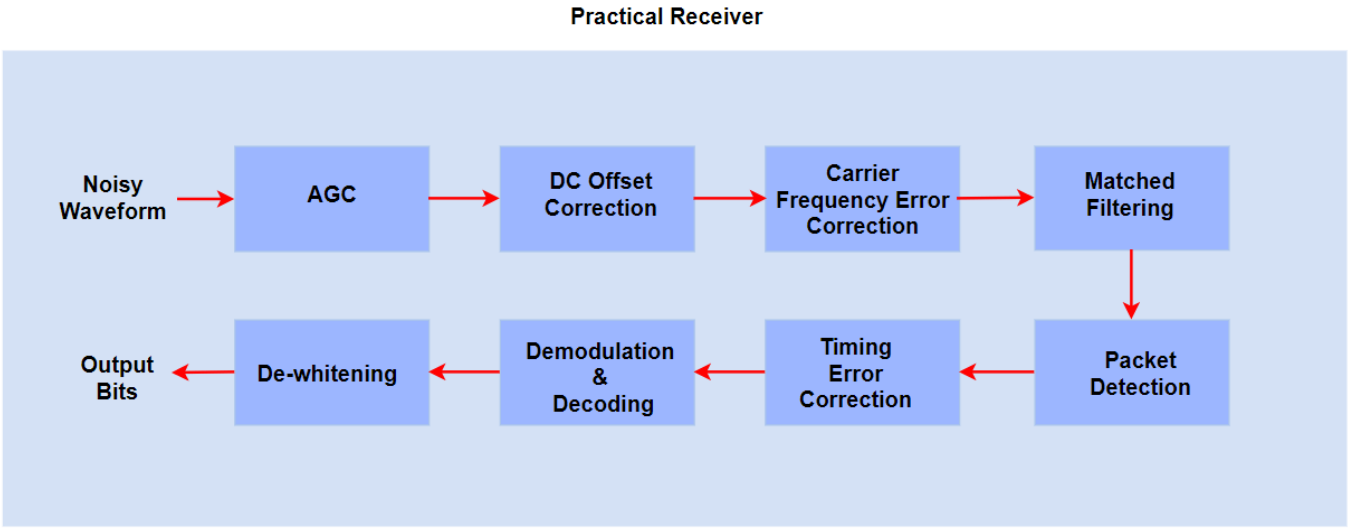
The noisy packets are processed through a practical BLE receiver that performs these operations:

1. Automatic gain control (AGC)
2. DC removal
3. Carrier frequency offset correction
4. Matched filtering
5. Packet detection

6 Timing error detection
7 Demodulation and decoding
8 De-whitening

The end-to-end example chain is summarized in these block diagrams





The BER is obtained by comparing the transmitted and recovered data bits.

Check for the Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed or not.

```
commSupportPackageCheck('BLUETOOTH');
```

Configure Simulation Parameters

In this example, the distance between the transmitter and receiver is estimated based on the environment and the power levels of the signal at the transmitter and receiver.

Configure parameters related to the communication link between the transmitter and receiver

```
pathLossModel = 'Free Space';          % Path loss model
rxSensitivity = -70;                   % Receiver sensitivity in dBm
txPower = 0;                           % Transmit power in dBm
txAntennaGain = 0;                     % Transmitter antenna gain in dB
rxAntennaGain = 0;                     % Receiver antenna gain in dB
linkMargin = 15;                       % Link margin(dB) assumed in the simulation
```

Configure parameters for waveform generation

```
samplesPerSymbol = 8;                  % Samples per symbol
dataLen = 254;                         % Data length in bytes
phyMode = 'LE1M';                      % PHY transmission mode

% Default access address for periodic advertising channels
accessAdd = [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]';

% Random data bits generation
txBits = randi([0 1],dataLen*8,1,'int8');

% Random channel index
chanIndex = 37;
fc = (2*chanIndex+2402)*1e6; % Center frequency in Hz

% Generate BLE waveform
txWaveform = bleWaveformGenerator(txBits,'Mode',phyMode,...
                    'SamplesPerSymbol',samplesPerSymbol,...
                    'ChannelIndex',chanIndex,...
                    'AccessAddress',accessAdd);
```

Configure noise and signal power at the receiver

The noise floor of the receiver is simulated with thermal noise. The height of the noise floor determines the SNR at the receiver. The noise figure of the receiver determines the level of noise floor.

```
NF = 6;             % Noise figure (dB)
T = 290;            % Ambient temperature (K)
dBm2dBFactor = 30; % Factor for converting dBm to dB

% Symbol rate based on the PHY transmission mode
symbolRate = 1e6;
if strcmp(phyMode,'LE2M')
    symbolRate = 2e6;
end
BW = samplesPerSymbol*symbolRate; % Bandwidth (Hz)
k = 1.3806e-23;                   % Boltzmann constant (J/K)
noiseFloor = 10*log10(k*T*BW)+NF; % Nosie floor in dB
```

```
% Measure signal power at the receiver based on the receiver sensitivity and
% assumed link margin
measuredPowerVector = rxSensitivity - dBm2dBFactor+linkMargin;
snrdB = measuredPowerVector - noiseFloor; % SNR in dB
```

Distort BLE Waveform

Distort the generated BLE waveform using RF impairments, path loss, and AWGN.

Add RF Impairments

The RF impairments are generated randomly and added to the BLE waveform.

```
% Create and configure the System objects for impairments
initImp = helperBLEImpairmentsInit(phyMode,samplesPerSymbol);

% Configure RF impairments
initImp.pfo.FrequencyOffset = 5800 ; % Frequency offset in Hz
initImp.pfo.PhaseOffset = 5 ;         % Phase offset in degrees
initoff = 0.15*samplesPerSymbol; % Static timing offset
stepsize = 20*1e-6;              % Timing drift in ppm, Max range is +/- 50 ppm
initImp.vdelay = (initoff:stepsize:initoff+stepsize*(length(txWaveform)-1))';
initImp.dc = 20; % Percentage related to maximum amplitude value

% Pass generated BLE waveform through RF impairments
txImpairedWfm = helperBLEImpairmentsAddition(txWaveform,initImp);
```

Attenuate Impaired BLE Waveform

Obtain the path loss value and attenuate the impaired BLE waveform.

```
% Obtain the path loss value in dB
pldB = txPower-dBm2dBFactor+rxAntennaGain+txAntennaGain-measuredPowerVector;
plLinear = 10^(pldB/20); % Convert from dB to linear scale

% Attenuate BLE waveform
attenWaveform  = txImpairedWfm./plLinear;
```

Add AWGN

Add AWGN to the attenuated BLE waveform.

```
% Add WGN to the attenuated BLE waveform
rxWaveform = awgn(attenWaveform,snrdB,'measured');
```

Simulation Results

Estimate and display the BER and the distance between the transmitter and the receiver by processing the distorted BLE waveform through the practical receiver.

Receiver Processing

To retrieve the data bits, pass the attenuated, AWGN-distorted BLE waveform through the practical receiver.

```
% Create and configure the receiver System objects
initRxParams = helperBLEReceiverInit(phyMode,samplesPerSymbol,accessAdd);
```

```
% Recover data bits using practical receiver
[rxBits,accessAddress] = helperBLEPracticalReceiver(rxWaveform,initRxParams,chanIndex);
```

Estimate BER

Estimate value of the BER based on the retrieved and the transmitted data bits.

```
% Obtain BER by comparing the transmitted and recovered bits
ber = [];
if(length(txBits) == length(rxBits))
    ber = (sum(xor(txBits,rxBits))/length(txBits));
end
```

Estimate Distance

Estimate the distance between the transmitter and the receiver.

```
% Estimate the distance between the transmitter and the receiver based on the path loss value and
if any(strcmp(pathLossModel,{'Free space','Log distance','Log normal shadowing'}))

    % Center frequency is required only for these path loss models
    distance = helperBluetoothEstimateDistance(pathLossModel,pldB,fc);
else
    distance = helperBluetoothEstimateDistance(pathLossModel,pldB);
end
```

Display Results

Display the estimated results and plot the spectrum of the transmitted and received BLE waveform.

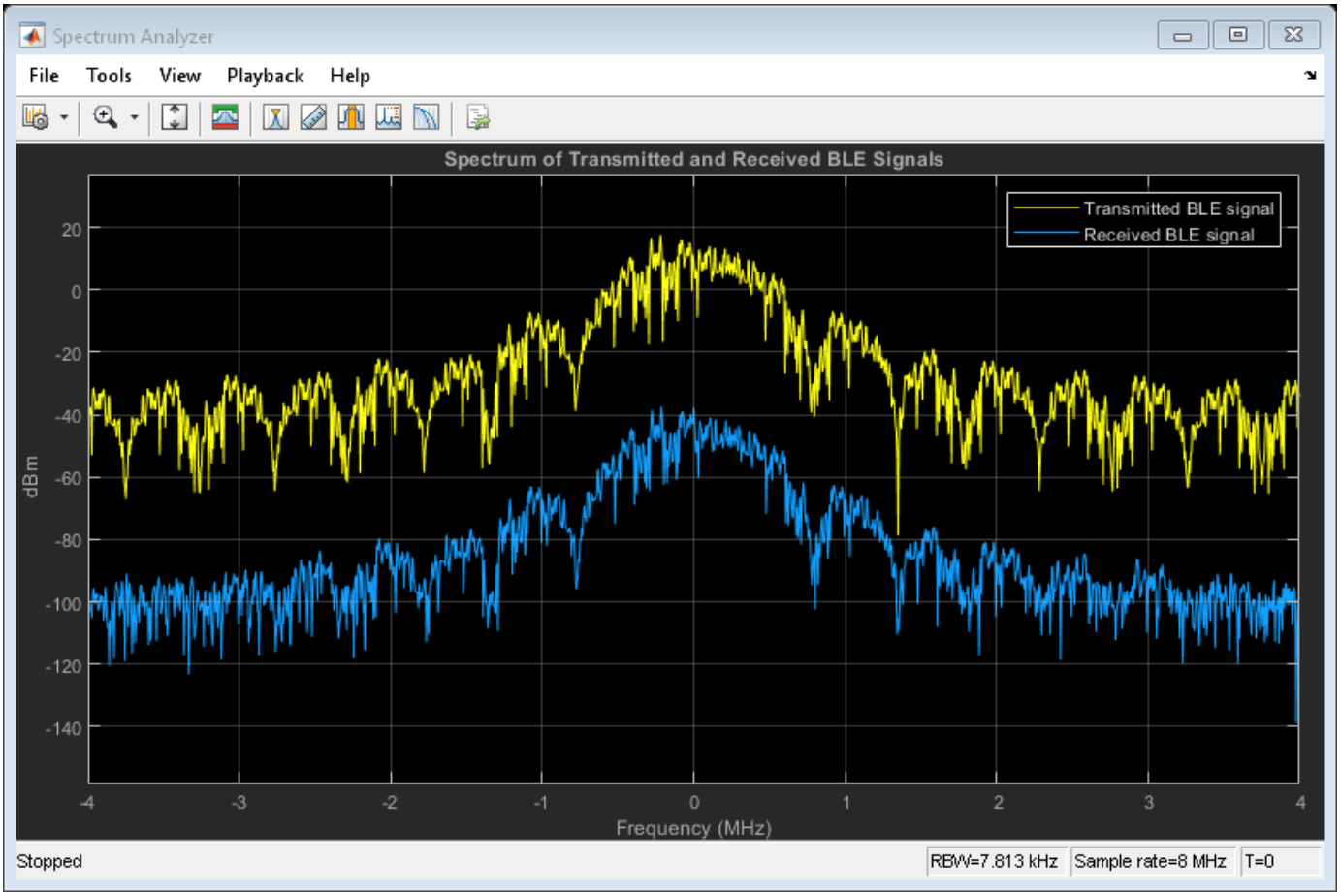
```
% Display estimated BER and distance between the transmitter and the receiver.
disp(['Input configuration: ', newline , '    PHY transmission mode: ', phyMode,....
    newline,'    Path loss model: ', pathLossModel]);
```

```
Input configuration:
    PHY transmission mode: LE1M
    Path loss model: Free space
```

```
disp(['Estimated outputs: ', newline , '    Path loss : ', num2str(pldB), ' dB'....
    newline,'    Distance between the transmitter and receiver: ', num2str(distance), ' m', newli
    '    BER: ', num2str(ber)]);
```

```
Estimated outputs:
    Path loss : 55 dB
    Distance between the transmitter and receiver: 5.422 m
    BER: 0
```

```
% Plot the spectrum of the transmitted and received BLE waveform
specAnalyzer = dsp.SpectrumAnalyzer('NumInputPorts',2,'SampleRate',symbolRate*samplesPerSymbol,.
    'Title','Spectrum of Transmitted and Received BLE Signals',...
   'ShowLegend',true,'ChannelNames',{'Transmitted BLE signal','Received BLE signal'});
specAnalyzer(txWaveform,rxWaveform);
release(specAnalyzer);
```



This example demonstrates an end-to-end BLE simulation for different PHY transmission modes by considering the path loss model, RF impairments, and AWGN. The obtained simulation results display the path loss, estimated distance between the transmitter and receiver, and BER. The spectrum of the transmitted and received BLE waveform is visualized by using a spectrum analyzer.

Appendix

The example uses these helper functions:

- helperBluetoothEstimateDistance.m: Calculates the distance between the transmitter and receiver based on the obtained path loss and the assumed environment.
- helperBLEImpairmentsAddition.m: Adds RF impairments to the BLE waveform.
- helperBLEPracticalReceiver.m: Demodulates and decodes the received BLE waveform.
- helperBLEReceiverInit.m: Initializes BLE receiver parameters.
- helperBLEImpairmentsInit.m: Initializes RF impairment parameters.

Selected Bibliography

[1] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.2. https://www.bluetooth.com.

[2] *Path Loss Models Used in Bluetooth Range Estimator*. Bluetooth Special Interest Group (SIG). https://www.bluetooth.com.

[3] Rappaport, Theodore. *Wireless Communication – Principles and Practice*. Prentice Hall, 1996.

[4] *NIST Smart Grid Interoperability Panel Priority Action Plan 2: Guidelines for Assessing Wireless Standards for Smart Grid Applications*. National Institute of Standards and Technology, U.S. Department of Commerce, 2014, https://nvlpubs.nist.gov/.

Bluetooth BR/EDR Waveform Reception by Using SDR

This example shows how to capture and decode Bluetooth® BR/EDR waveforms by using the Communications Toolbox™ Library for the Bluetooth Protocol. You can either capture the Bluetooth BR/EDR waveforms by using the ADALM-PLUTO radio or load IQ samples corresponding to the Bluetooth BR/EDR waveforms from a baseband file (*.bb). To generate and transmit the Bluetooth BR/EDR waveforms, refer to “Bluetooth BR/EDR Waveform Generation and Transmission Using SDR” on page 3-129 and configure your test environment with:

- Two SDR platforms connected to the same host computer, and run two MATLAB® sessions on the single host computer.
- Two SDR platforms connected to two host computers, and run one MATLAB session on each host computer.

To configure your host computer to work with the Support Package for ADALM-PLUTO Radio, refer to “Guided Host-Radio Hardware Setup” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Required Hardware

To capture signals in real time, you need ADALM-PLUTO radio and the corresponding support package add-on:

- Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of communications toolboxes supported by SDR platforms, refer to the Supported Hardware section of the Software Defined Radio (SDR) discovery page.

Bluetooth BR/EDR Radio Specifications

Bluetooth is a short-range Wireless Personal Area Network (WPAN) technology, operating in the globally unlicensed industrial, scientific, and medical (ISM) band in the frequency range of 2.4 GHz to 2.485 GHz. In Bluetooth technology, data is divided into packets and each packet is transmitted on one of the 79 designated Bluetooth channels. Each channel has a bandwidth of 1 MHz. As there are different types of wireless networks operating in the same unlicensed frequency band, it is possible for two different networks to interfere with each other. To mitigate the interference, Bluetooth implements the frequency-hopping spread spectrum (FHSS) scheme to switch a carrier between multiple frequency channels by using a pseudorandom sequence known to both transmitter and receiver.

The Bluetooth standard specifies these physical layer (PHY) modes:

Basic rate (BR) - Mandatory mode, uses gaussian frequency shift keying (GFSK) modulation with a data rate of 1 Mbps

Enhanced data rate (EDR) - Optional mode, uses phase shift keying (PSK) modulation with these two variants:

- EDR2M: Uses pi/4-DQPSK with a data rate of 2 Mbps
- EDR3M: Uses 8-DPSK with a data rate of 3 Mbps

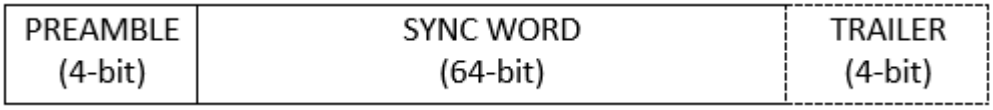
Bluetooth BR/EDR Packet Formats

The air interface packet formats for PHY modes include these fields:

Access Code: Each packet starts with an access code. If a packet header follows, the access code is 72 bits long. Otherwise, the length of the access code is 68 bits and referred to as a shortened access code. The access code consists of these fields:

- Preamble: The preamble is a fixed zero-one pattern of four symbols.
- Sync Word: The sync word is a 64-bit code word derived from the 24-bit lower address part (LAP) of the Bluetooth device address.
- Trailer: The trailer is a fixed zero-one pattern of four symbols.

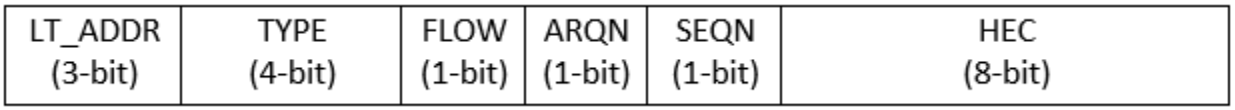
Access Code Format



Packet Header: The header includes link control information and consists of these fields:

- LT_ADDR: 3-bit logical transport address
- TYPE: 4-bit type code, which specifies the packet type used for transmission. The value of this field can be ID, NULL, POLL, FHS, HV1, HV2, HV3, DV, EV3, EV4, EV5, 2-EV3, 2-EV5, 3-EV3, 3-EV5, DM1, DH1, DM3, DH3, DM5, DH5, AUX1, 2-DH1, 2-DH3, 2-DH5, 3-DH1, 3-DH3 and 3-DH5. This field determines the number of slots the current packet occupies.
- FLOW: 1-bit flow control over the asynchronous connection-oriented logical (ACL) transport
- ARQN: 1-bit acknowledgement indication
- SEQN: 1-bit sequence number
- HEC: 8-bit header error check

Header Format



Payload: Payload includes an optional payload header, a payload body, and an optional CRC.

Payload Format

Guard: For EDR packets, guard time allows the Bluetooth BR/EDR radio to prepare for the change in modulation from GFSK to DPSK. The guard time must be between 4.75 to 5.25 microseconds.

Sync: For EDR packets, the synchronization sequence contains one reference symbol and ten DPSK symbols.

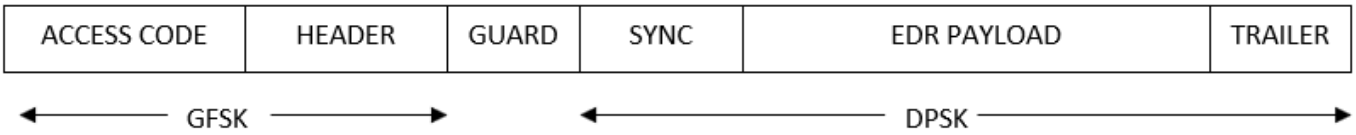
Trailer: For EDR packets, the trailer bits must be all zero pattern of two symbols, {00,00} for pi/4-DQPSK and {000,000} for 8DPSK.

Packet format for BR mode is shown in this figure.

Basic Rate Packet Format

Packet format for EDR mode is shown in this figure.

Enhanced Data Rate Packet Format



Check for Support Package Installation

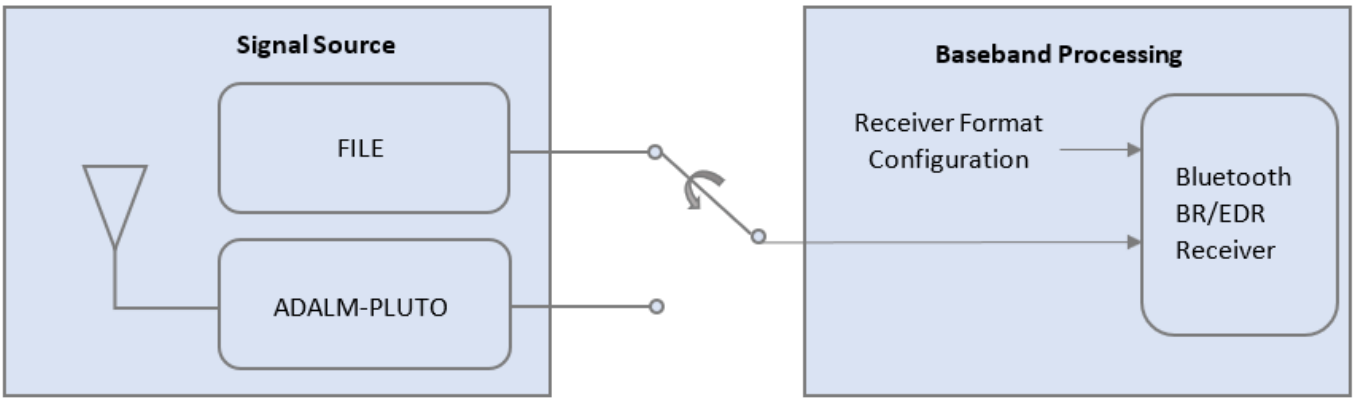
Check if the Communications Toolbox Library for the Bluetooth Protocol support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Decode Bluetooth BR/EDR Waveforms

This example shows how to decode Bluetooth BR/EDR waveforms either captured by using ADALM-PLUTO or by reading IQ samples from a baseband file.

Bluetooth BR/EDR Receiver



The general structure of the Bluetooth receiver example is:

1. Initialize the receiver parameters.
2. Specify the signal source.
3. Capture the Bluetooth BR/EDR waveforms.
4. Process Bluetooth BR/EDR waveforms at receiver.

Initialize the Receiver Parameters

To configure Bluetooth BR/EDR parameters, use `bluetoothPhyConfig` object.

```
cfg = bluetoothPhyConfig;
cfg.Mode = BR; % Mode of transmission as one of BR, EDR2M and EDR3M
cfg.WhitenInitialization = [0;0;0;0;0;1;1]; % Whiten initialization
```

Specify the Signal Source

Specify the signal source as File or ADALM-PLUTO.

- **File**: Uses the `comm.BasebandFileReader` to read a file that contains a previously captured over-the-air signal.
- **ADALM-PLUTO**: Uses the`sdrrx` (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) System object to receive a live signal from the SDR hardware.

If you assign **ADALM-PLUTO** as the signal source, the example searches your computer for the **ADALM-PLUTO** radio at radio address 'usb:0' and uses it as the signal source.

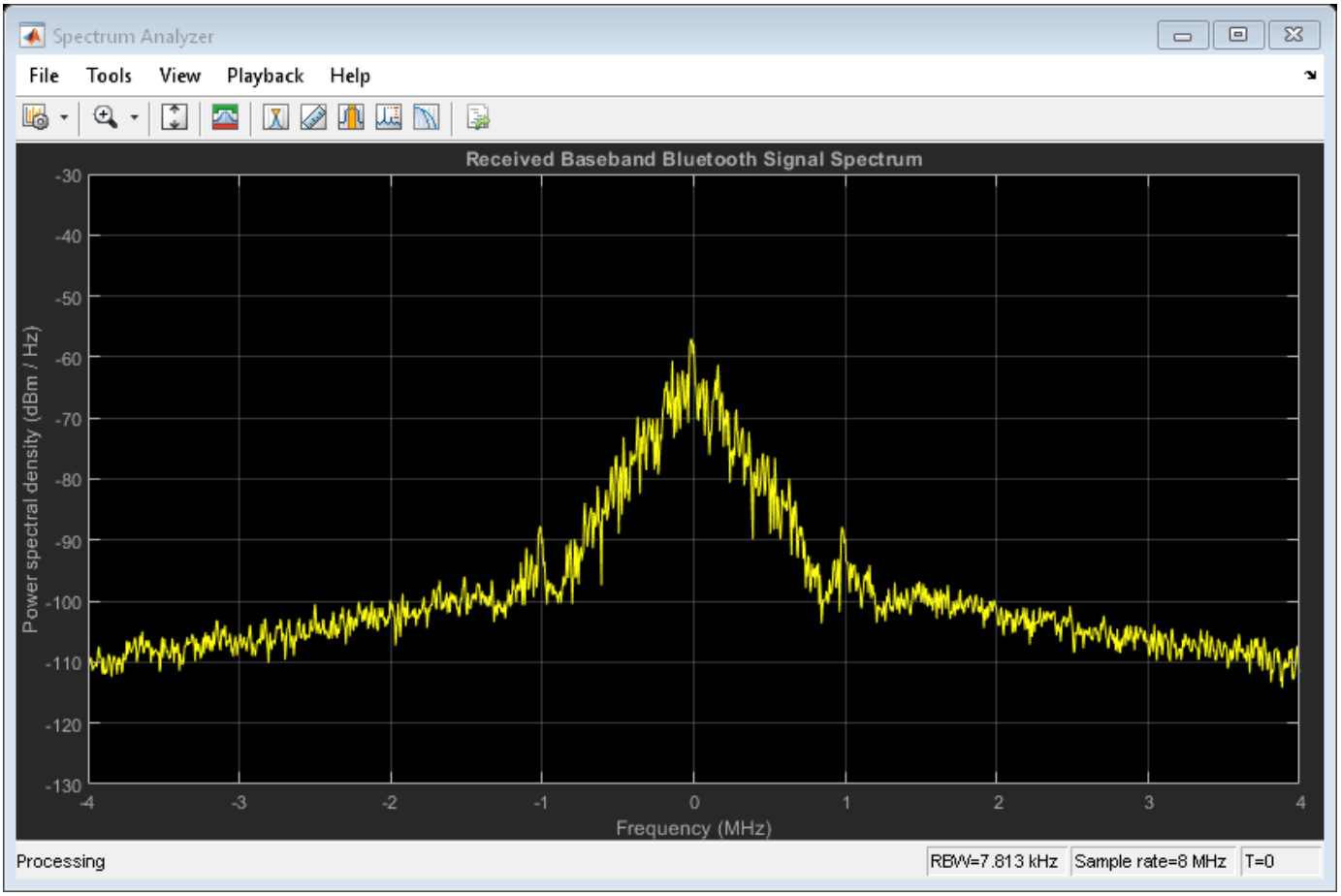
```
% The default signal source is 'File'
signalSource = File;

bbSymbolRate = 1e6; % 1 MSps
if strcmp(signalSource,'File')
    switch cfg.Mode
        case 'BR'
            bbFileName = 'bluetoothCapturesBR.bb';
        case 'EDR2M'
            bbFileName = 'bluetoothCapturesEDR2M.bb';
        case 'EDR3M'
            bbFileName = 'bluetoothCapturesEDR3M.bb';
    end
    sigSrc = comm.BasebandFileReader(bbFileName);
    sigSrcInfo = info(sigSrc);
    bbSampleRate = sigSrc.SampleRate;
    sigSrc.SamplesPerFrame = sigSrcInfo.NumSamplesInData;
    cfg.SamplesPerSymbol = bbSampleRate/bbSymbolRate;

else

    % Check if the pluto Hardware Support Package (HSP) is installed
    if isempty(which('plutoradio.internal.getRootDir'))
        error(message('comm_demos:common:NoSupportPackage', ...
                      'Communications Toolbox Support Package for ADALM-PLUTO Radio',...
                      ['<a href="https://www.mathworks.com/hardware-support/' ...
                      'adalm-pluto-radio.html">ADALM-PLUTO Radio Support From Communications Too
    end
    connectedRadios = findPlutoRadio; % Discover ADALM-PLUTO radio(s) connected to your computer
    radioID = connectedRadios(1).RadioID;
    rxCenterFrequency = 2445000000;  % In Hz, choose between 2.402e9 to
    bbSampleRate = bbSymbolRate * cfg.SamplesPerSymbol;
    sigSrc = sdrrx('Pluto',...
        'RadioID',             radioID,...
        'CenterFrequency',     rxCenterFrequency,...
```

```
        'BasebandSampleRate',  bbSampleRate,...
        'SamplesPerFrame',     1e7,...
        'GainSource',          'Manual',...
        'Gain',                25,...
        'OutputDataType',      'double');
end
```

Capture the Bluetooth BR/EDR Waveforms

Capture the IQ samples corresponding to Bluetooth BR/EDR waveforms either by using ADALM-PLUTO or baseband file as signal source. Visualize the spectrum of the received Bluetooth waveforms by using a spectrum analyzer.

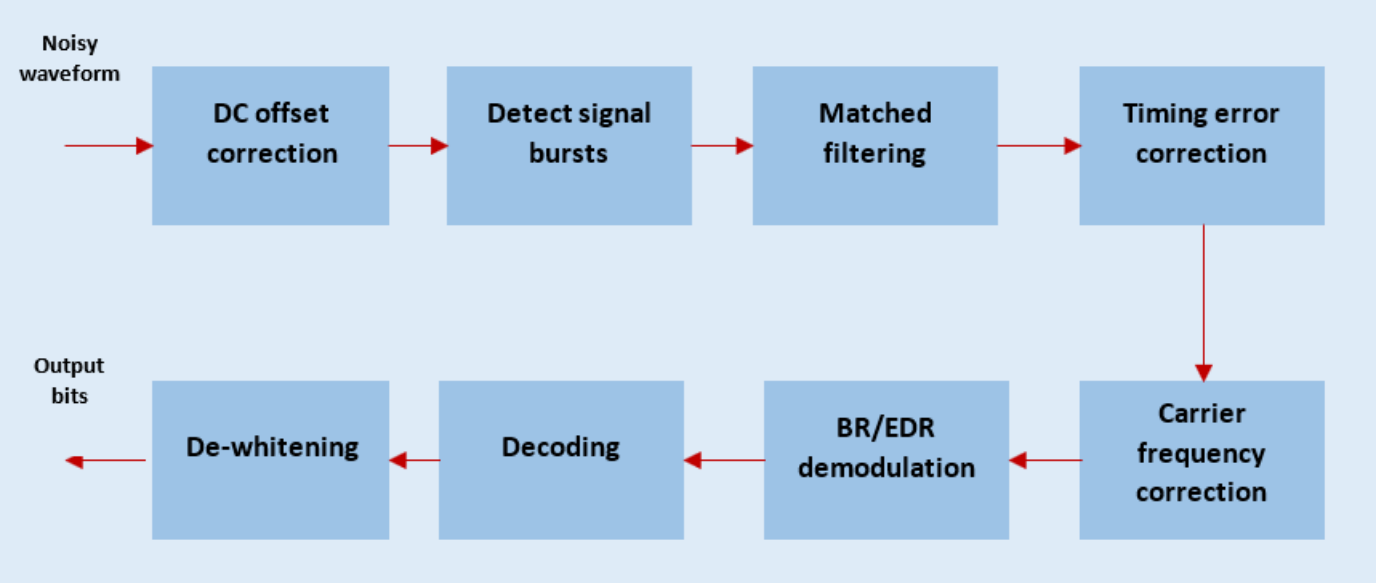
```
% The transmitted waveforms are captured as a burst
dataCaptures = sigSrc();

% Setup spectrum viewer
spectrumScope = dsp.SpectrumAnalyzer( ...
    'SampleRate',       bbSampleRate,...
    'SpectrumType',     'Power density', ...
    'SpectralAverages', 10, ...
    'YLimits',          [-130 -30], ...
    'Title',            'Received Baseband Bluetooth Signal Spectrum', ...
    'YLabel',           'Power spectral density');

% Show power spectral density of the received waveform
spectrumScope(dataCaptures);
```



Process Bluetooth BR/EDR Waveforms at Receiver

To decode the packet header, payload header information, and raw message bits, the receiver processess the baseband samples received from the signal source. This figure shows the receiver processing.

Bluetooth Practical Receiver



The Bluetooth practical receiver performs these functions:

1. Remove DC offset
2. Detect the signal bursts
3. Perform matched filtering
4. Estimate and correct the timing offset
5. Estimate and correct the carrier frequency offset
6. Demodulate BR/EDR waveform
7. Perform forward error correction (FEC) decoding
8. Perform data dewhitening
9. Perform header error check (HEC) and cyclic redundancy check (CRC)
10. Outputs decoded bits and decoded packet statistics based on decoded lower address part (LAP), HEC and CRC

```
% Bluetooth practical receiver
[decBits,decodedInfo,pktStatus] = helperBluetoothPracticalReceiver(dataCaptures,cfg);

% Get the number of detected packets
pktCount = length(pktStatus);
disp(['Number of Bluetooth packets detected: ' num2str(pktCount)])
```

```
Number of Bluetooth packets detected: 2
```

```
% Get the decoded packet statistics
displayFlag = true; % set true, to display the decoded packet statistics
if(displayFlag && (pktCount~=0))
    decodedInfoPrint = decodedInfo;
    for ii = 1:pktCount
        if(pktStatus(ii))
            decodedInfoPrint(ii).PacketStatus = 'Success';
        else
```

```
            decodedInfoPrint(ii).PacketStatus = 'Fail';
        end
    end
    packetInfo = struct2table(decodedInfoPrint,'AsArray',1);
    fprintf('Decoded Bluetooth packet(s) information: \n \n')
    disp(packetInfo);
end
```

```
Decoded Bluetooth packet(s) information:

        LAP          PacketType    LogicalTransportAddress    HeaderControlBits    PayloadLength
    _____________    __________    _______________________    _________________    _____________

    {24x1 double}     {'FHS'}           {3x1 double}            {3x1 double}            18
    {24x1 double}     {'FHS'}           {3x1 double}            {3x1 double}            18
```

```
% Get the packet error rate performance metrics
if(pktCount)
    pktErrCount = sum(~pktStatus);
    pktErrRate = pktErrCount/pktCount;
    disp(['Simulated Mode: ' cfg.Mode ', '...
        'Packet error rate: ',num2str(pktErrRate)])
end
```

```
Simulated Mode: BR, Packet error rate: 0
```

```
% Release the signal source
release(sigSrc);
```

This example enables you to decode Bluetooth BR/EDR waveforms either captured by using ADALM-PLUTO or by reading IQ samples from a baseband file. Visualize the spectrum of the received Bluetooth waveforms by using a spectrum analyzer. The packet error rate is computed based on the decoded packet information.

Further Exploration

You can use this example to receive EDR packets by changing the PHY transmission mode. To generate the Bluetooth waveforms in this example, refer to “Bluetooth BR/EDR Waveform Generation and Transmission Using SDR” on page 3-129.

Troubleshooting

General tips for troubleshooting SDR hardware and the Communications Toolbox Support Package for ADALM-PLUTO Radio can be found in “Common Problems and Fixes” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Appendix

This example uses this helper function:

- helperBluetoothPracticalReceiver.m: Practical receiver for Bluetooth physical layer

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". *Bluetooth Core Specification*. Version 5.1, Volume 2 <https://www.bluetooth.com/>

See Also

More About

- “Bluetooth BR/EDR Waveform Generation and Transmission Using SDR” on page 3-129

End-to-End Bluetooth BR/EDR PHY Simulations with RF Impairments and Corrections

This example shows an end-to-end simulation to measure the bit error rate (BER) and packet error rate (PER) for different Bluetooth® BR/EDR physical layer (PHY) packet types by using the Communications Toolbox™ Library for the Bluetooth® Protocol. These PHY packets are distorted by adding radio front-end (RF) impairments and the additive white Gaussian noise (AWGN). The distorted Bluetooth BR/EDR waveforms are processed at the practical receiver to get the BER and PER values. The obtained simulation results show the plots of BER and PER as a function of energy-to-noise density ratio (Eb/No).

Bluetooth BR/EDR Radio Specifications

Bluetooth is a short-range Wireless Personal Area Network (WPAN) technology, operating in the globally unlicensed industrial, scientific, and medical (ISM) band in the frequency range of 2.4 GHz to 2.485 GHz. In Bluetooth technology, data is divided into packets. Each packet is transmitted on one of the 79 designated Bluetooth channels. The bandwidth of each channel is 1 MHz. Bluetooth implements the frequency-hopping spread spectrum (FHSS) scheme to switch a carrier between multiple frequency channels by using a pseudorandom sequence known to the transmitter and the receiver.

The Bluetooth standard specifies these PHY modes:

Basic rate (BR) - Mandatory mode, uses Gaussian frequency shift keying (GFSK) modulation with a data rate of 1 Mbps.

Enhanced data rate (EDR) - Optional mode, uses phase shift keying (PSK) modulation with these two variants:

- **EDR2M:** Uses pi/4-DQPSK with a data rate of 2 Mbps
- **EDR3M:** Uses 8-DPSK with a data rate of 3 Mbps

This end-to-end Bluetooth BR/EDR PHY simulation determines BER and PER performance of one Bluetooth packet that has RF impairments and AWGN. Each packet is generated over a loop of a vector equal to length of the energy-to-noise density ratio (Eb/No) using the `bluetoothWaveformGenerator` function by configuring the `bluetoothWaveformConfig` object.

To accumulate the error rate statistics, the generated waveform is altered with RF impairments and AWGN before passing through the receiver.

These RF impairments are used to distort the packet:

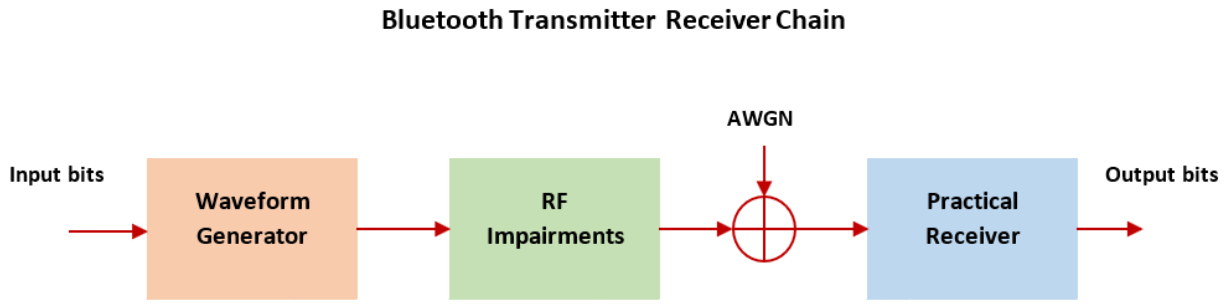
- DC offset
- Carrier frequency offset
- Static timing offset
- Timing drift

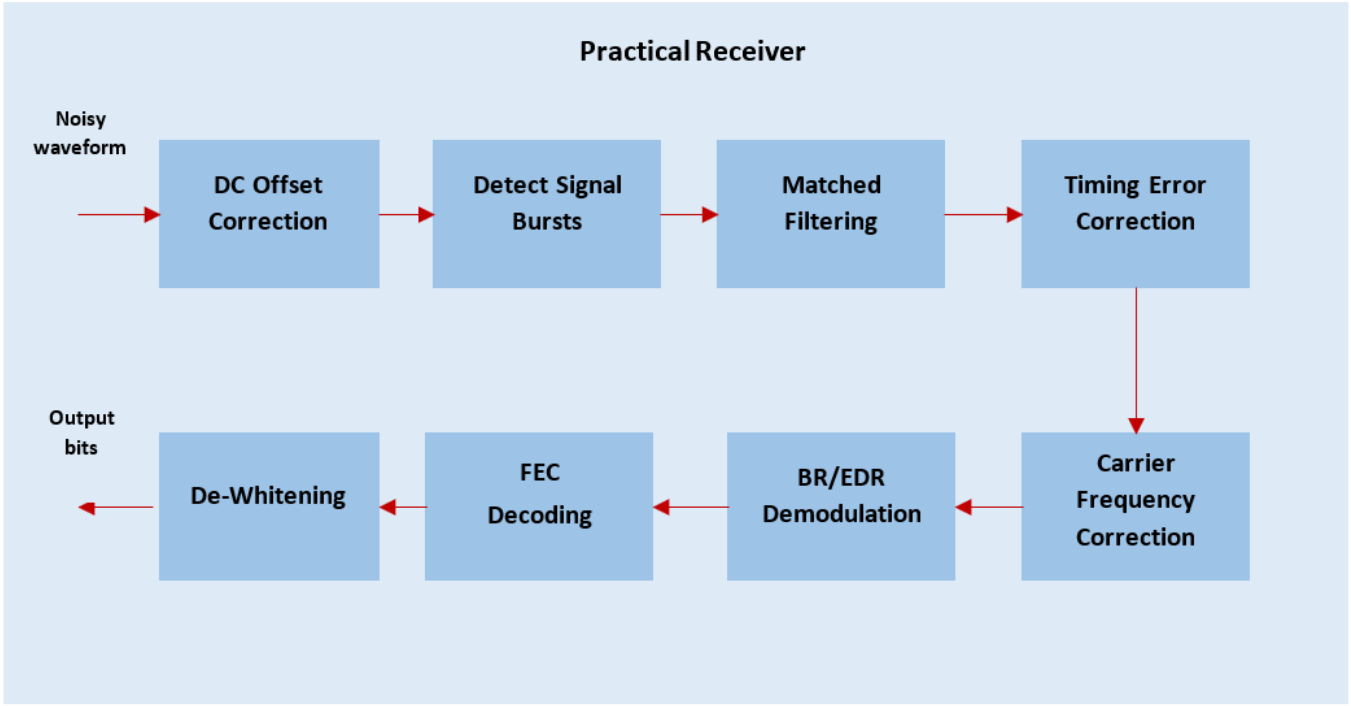
White Gaussian noise is added to the generated Bluetooth BR/EDR waveforms. The distorted and noisy waveforms are processed through a practical Bluetooth receiver performing these operations:

- Remove DC offset

- Detect the signal bursts
- Perform matched filtering
- Estimate and correct the timing offset
- Estimate and correct the carrier frequency offset
- Demodulate BR/EDR waveform
- Perform forward error correction (FEC) decoding
- Perform data dewhitening
- Perform header error check (HEC) and cyclic redundancy check (CRC)
- Outputs decoded bits and decoded packet statistics based on decoded lower address part (LAP), HEC, and CRC

This block diagram illustrates the processing steps for each Bluetooth BR/EDR PHY packet.





To determine the BER and the PER, compare the recovered output bits with the transmitted data bits.

Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Initialize Simulation Parameters

```
% Eb/No in dB
EbNo = 2:2:14;
% Maximum number of bit errors simulated at each Eb/No point
maxNumErrs = 100;
% Maximum number of bits accumulated at each Eb/No point
maxNumBits = 1e6;
% Maximum number of packets considered at each Eb/No point
maxNumPkts = 1000;
```

In this example, the values for maxNumErrs, maxNumBits, and maxNumPkts are selected for a short simulation time.

Configure Bluetooth BR/EDR Waveform

The Bluetooth BR/EDR waveform is configured by using the `bluetoothWaveformConfig` object. Configure the properties of the `bluetoothWaveformConfig` object as per your requirements. In this example, the PHY mode of transmission, the Bluetooth packet type, and the number of samples per symbol are configured.

```
phyMode = 'BR';                    % PHY transmission mode
bluetoothPacket = 'FHS';           % Type of Bluetooth packet, this value can be: {'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','3-EV5'}
sps = 8;                           % Samples per symbol, must be greater than 1
```

Configure RF Impairments

Set frequency, time, and DC offset parameters to distort the Bluetooth BR/EDR waveform.

```
frequencyOffset = 6000;% In Hz
timingOffset = 0.5;    % In samples, less than 1 microsecond
timingDrift = 2;       % In parts per million
dcOffset = 2;          % Percentage w.r.t maximum amplitude value
symbolRate = 1e6;      % Symbol Rate

% Create timing offset object
timingDelayObj = dsp.VariableFractionalDelay;

% Create frequency offset object
frequencyDelay = comm.PhaseFrequencyOffset('SampleRate',symbolRate*sps);
```

Process Eb/No Points

For each Eb/No point, packets are generated and processed through these steps:

- Generate random bits
- Generate Bluetooth BR/EDR waveform
- Pass generated waveform through AWGN channel
- Add frequency offset
- Add timing offset
- Add DC offset
- Pass distorted waveform through practical receiver
- Calculate BER and PER

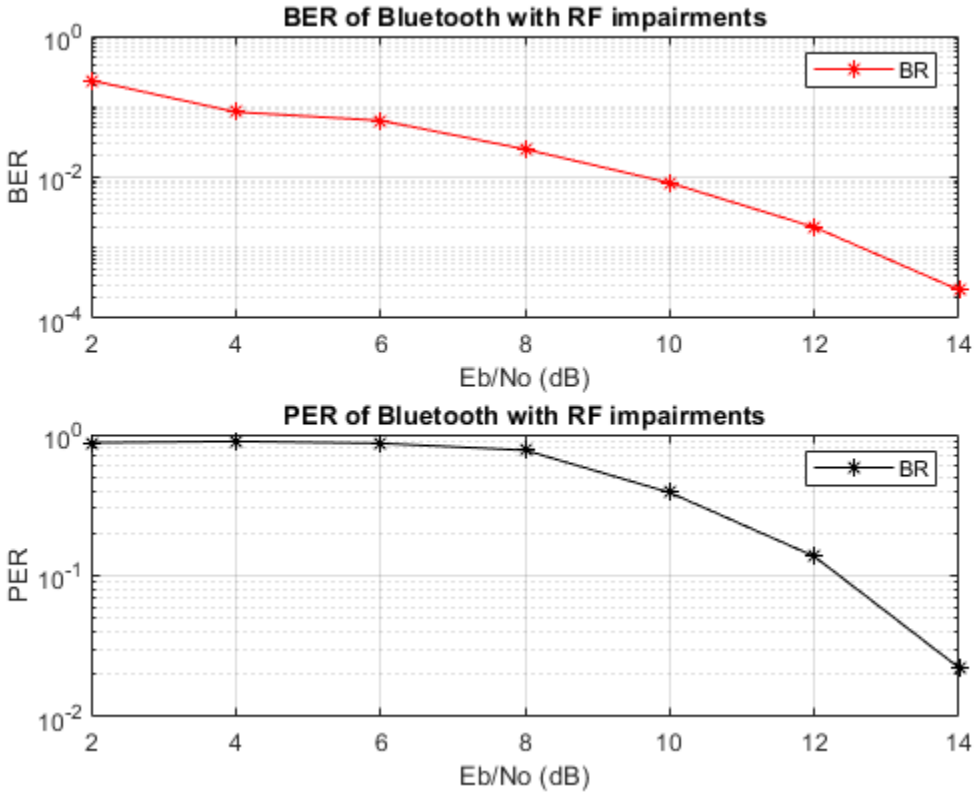
```
ber = zeros(1,length(EbNo));       % BER results
per = zeros(1,length(EbNo));       % PER results
bitsPerByte = 8;                   % Number of bits per byte
% Set code rate based on packet
if any(strcmp(bluetoothPacket,{'FHS','DM1','DM3','DM5','HV2','DV','EV4'}))
    codeRate = 2/3;
elseif strcmp(bluetoothPacket,'HV1')
    codeRate = 1/3;
else
    codeRate = 1;
end
% Set number of bits per symbol based on the PHY transmission mode
bitsPerSymbol = 1+ (strcmp(phyMode,'EDR2M'))*1 +(strcmp(phyMode,'EDR3M'))*2;

% Get SNR from EbNo values
snr = EbNo + 10*log10(codeRate) + 10*log10(bitsPerSymbol) - 10*log10(sps);
% Create a Bluetooth BR/EDR waveform configuration object
txCfg = bluetoothWaveformConfig('Mode',phyMode,'PacketType',bluetoothPacket,...
                                'SamplesPerSymbol',sps);
if strcmp(bluetoothPacket,'DM1')
    txCfg.PayloadLength = 17; % Maximum length of DM1 packets in bytes
end
dataLen = getPayloadLength(txCfg);  % Length of the payload
% Get PHY properties
rxCfg = getPhyConfigProperties(txCfg);

for iSnr = 1:length(snr)
        rng default
        % Initialize error computation parameters
        errorCalc = comm.ErrorRate;
        berVec = zeros(3,1);
        pktCount = 0;  % Counter for number of detected Bluetooth packets
        loopCount = 0; % Counter for number of packets at each SNR value
        pktErr = 0;
         while((berVec(2) < maxNumErrs) && (berVec(3) < maxNumBits) && (loopCount < maxNumPkts))
            txBits = randi([0 1],dataLen*bitsPerByte,1); % Data bits generation
            txWaveform = bluetoothWaveformGenerator(txBits,txCfg);

            % Add Frequency Offset
            frequencyDelay.FrequencyOffset = frequencyOffset;
            transWaveformCFO = frequencyDelay(txWaveform);
```

```
            % Add Timing Delay
            timingDriftRate = (timingDrift*1e-6)/(length(txWaveform)*sps);% Timing drift rate
            timingDriftVal = timingDriftRate*(0:1:(length(txWaveform)-1))';% Timing drift
            timingDelay = (timingOffset*sps)+timingDriftVal;   % Static timing offset and timing
            transWaveformTimingCFO = timingDelayObj(transWaveformCFO,timingDelay);

            % Add DC Offset
            dcValue = (dcOffset/100)*max(transWaveformTimingCFO);
            txImpairedWaveform = transWaveformTimingCFO + dcValue;

             % Add AWGN
            txNoisyWaveform = awgn(txImpairedWaveform,snr(iSnr),'measured');

            % Receiver Module
            [rxBits,decodedInfo,pktStatus]...
                                = helperBluetoothPracticalReceiver(txNoisyWaveform,rxCfg);
            numOfSignals = length(pktStatus);
            pktCount = pktCount+numOfSignals;
            loopCount = loopCount+1;

            % BER and PER Calculations
            L1 = length(txBits);
            L2 = length(rxBits);
            L = min(L1,L2);
            if(~isempty(L))
               berVec = errorCalc(txBits(1:L),rxBits(1:L));
            end
            pktErr = pktErr+sum(~pktStatus);
        end
        % Average of BER and PER
        per(iSnr) = pktErr/pktCount;
        ber(iSnr) = berVec(1);
        if ((ber(iSnr) == 0) && (per(iSnr) == 1))
            ber(iSnr) = 0.5; % If packet error rate is 1, consider average BER of 0.5
        end
        if ~any(strcmp(bluetoothPacket,{'ID','NULL','POLL'}))
            disp(['Mode ' phyMode ', '...
                'Simulated for Eb/No = ', num2str(EbNo(iSnr)), ' dB' ', '...
                'obtained BER:',num2str(ber(iSnr)),' obtained PER: ',...
                   num2str(per(iSnr))]);
        else
            disp(['Mode ' phyMode ', '...
                'Simulated for Eb/No = ', num2str(EbNo(iSnr)), ' dB' ', '...
                'obtained PER: ',num2str(per(iSnr))]);
        end
end

Mode BR, Simulated for Eb/No = 2 dB, obtained BER:0.23611 obtained PER: 0.875
Mode BR, Simulated for Eb/No = 4 dB, obtained BER:0.084028 obtained PER: 0.89474
Mode BR, Simulated for Eb/No = 6 dB, obtained BER:0.063492 obtained PER: 0.86667
Mode BR, Simulated for Eb/No = 8 dB, obtained BER:0.025 obtained PER: 0.77419
Mode BR, Simulated for Eb/No = 10 dB, obtained BER:0.0083333 obtained PER: 0.38824
Mode BR, Simulated for Eb/No = 12 dB, obtained BER:0.0019597 obtained PER: 0.13699
Mode BR, Simulated for Eb/No = 14 dB, obtained BER:0.00025304 obtained PER: 0.022267
```

Simulation Results

This section presents the BER and PER results with respect to the input Eb/No range for the considered PHY mode.

```
figure,
if any(strcmp(bluetoothPacket,{'ID','NULL','POLL'}))
    numOfPlots = 1; % Plot only PER
else
    numOfPlots = 2; % Plot both BER and PER
    subplot(numOfPlots,1,1),semilogy(EbNo,ber.','-r*');
    xlabel('Eb/No (dB)');
    ylabel('BER');
    legend(phyMode);
    title('BER of Bluetooth with RF impairments');
    hold on;
    grid on;
end
subplot(numOfPlots,1,numOfPlots),semilogy(EbNo,per.','-k*');
xlabel('Eb/No (dB)');
ylabel('PER');
legend(phyMode);
title('PER of Bluetooth with RF impairments');
hold on;
grid on;
```



Appendix

The example uses this helper function:

- helperBluetoothPracticalReceiver.m: Detects, synchronizes, and decodes the received Bluetooth BR/EDR waveform.

This example shows an entire end-to-end procedure to generate a Bluetooth BR/EDR packet. The generated Bluetooth BR/EDR waveform is distorted by adding RF impairments and AWGN. To get the BER and PER values, the distorted Bluetooth BR/EDR waveform is synchronized, demodulated, and decoded from the practical receiver.

Selected Bibliography

- Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". *Bluetooth Core Specification*. Version 5.1, Volume 2. www.bluetooth.com/

See Also

More About

- “Generate BLE Waveform and Add RF Impairments” on page 13-103
- “End-to-End BLE PHY Simulation Using Path Loss Model, RF Impairments, and AWGN” on page 3-90
- “End-to-End Bluetooth Low Energy PHY Simulation with RF Impairments and Corrections” on page 3-275

Estimate Packet Delivery Ratio in Bluetooth Mesh Network

This example models a multi-node Bluetooth® mesh network discrete event simulation (DES) by using the Communications Toolbox™ Library for the Bluetooth® Protocol. DES is the process of simulating the behavior of a system as an ordered and discrete sequence of well-defined events occurring in the time domain. DES allows you to model events in a system that occur in microsecond granularity. Moreover, DES also results in low simulation time thus making it viable to support large-scale system-level simulations. The multi-node mesh network simulated in this example models the complete Bluetooth mesh stack over the advertising bearer. The example aims to accurately model the asynchronous transmissions by using DES. The simulation results include packet delivery ratio (PDR), node-related statistics, and a plot displaying the visual representation of the mesh network.

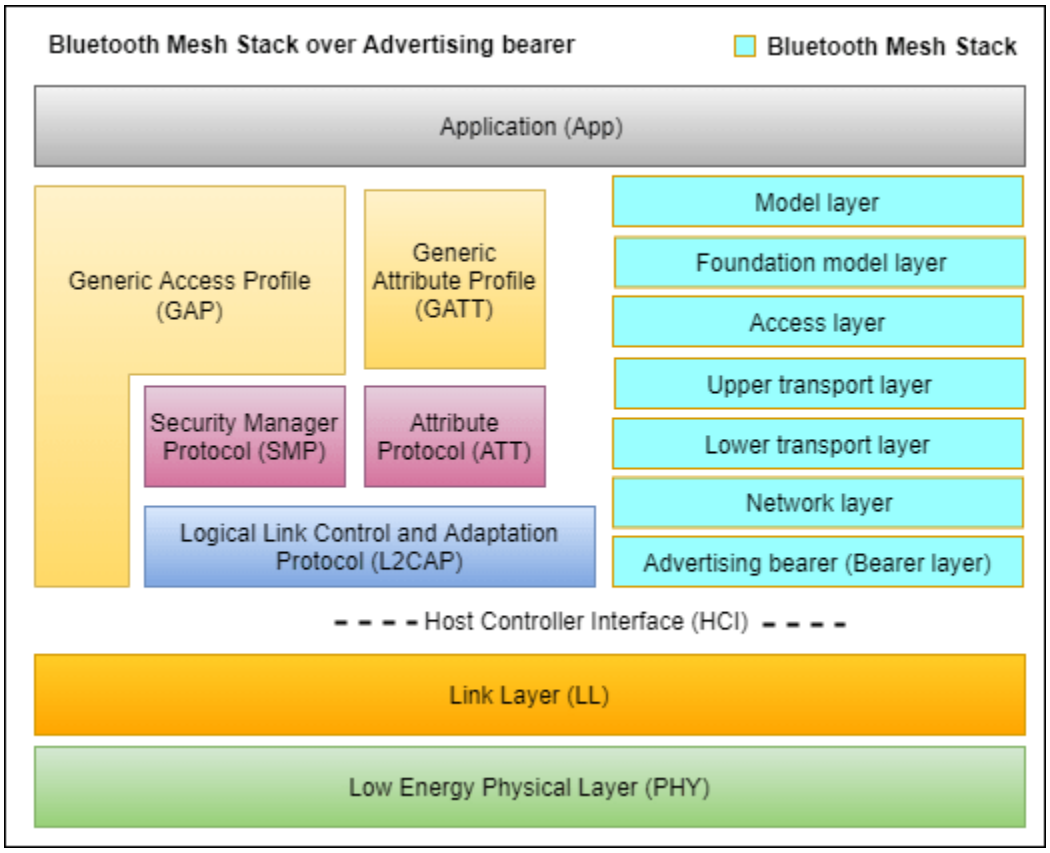
Bluetooth Mesh Stack

The Bluetooth Core Specification [1] includes a low energy version for low-rate wireless personal area networks, referred as Bluetooth low energy (BLE) or Bluetooth Smart. The BLE stack consists of the: generic attribute profile (GATT), attribute protocol (ATT), security manager protocol (SMP), logical link control and adaptation protocol (L2CAP), link layer (LL) and physical layer (PHY). BLE was added to the standard for low energy devices generating small amounts of data, such as the notification alerts used in applications like home automation, healthcare, fitness, and the Internet of Things (IoT).

The Bluetooth Mesh Profile [2] defines the fundamental requirements to implement mesh networking solutions for BLE. The mesh stack is located on top of the Bluetooth Core Specification and consists of the: model layer, foundation model layer, access layer, upper transport layer, lower transport layer, network layer and bearer layer. Bluetooth mesh networking enables end-to-end communication in large-scale networks to support applications like smart lighting, industrial automation, sensor networking, asset tracking, and many other IoT solutions.

Mesh Stack

This figure shows the Bluetooth mesh stack over the advertising bearer.



- Model layer: This layer defines the models, messages, and states required to create user scenarios. For example, to change the state of a light to On or Off, use the 'Generic onOff set' message from the 'Generic onOff' model.
- Foundation model layer: This layer defines the models, messages, and states required to configure and manage the mesh network. This layer configures the element, the publish and the subscription addresses of the node.
- Access layer: This layer defines the interface to the upper transport layer and the format of the application data. This layer also controls the encryption and decryption of the application data in the upper transport layer.
- Upper transport layer: The functionality of the upper transport layer includes encryption, decryption and authentication of the application data and provides confidentiality of the access messages. This layer also generates the transport control messages (Friendship and Heartbeat) and transmits them to the peer upper transport layer.
- Lower transport layer: The functionality of lower transport layer includes segmentation and reassembly of upper transport layer messages into multiple lower transport layer messages. This layer helps to deliver large upper transport layer messages to other nodes in the network. It also defines the Friend queue used by the Friend node to store the lower transport layer messages for a Low Power node.
- Network layer: This layer defines encryption, decryption, and authentication of the lower transport layer messages. It transmits the lower transport layer messages over the bearer layer and relays the mesh messages when the 'Relay' feature is enabled. It also defines the message

cache containing all the recently seen network messages. If the received message is in the cache, then it is discarded. The message cache is used by the relay nodes (nodes in which the 'Relay' feature is enabled).

- Bearer layer: This layer is an interface between the Bluetooth mesh stack and the BLE core stack. This layer is also responsible for creating a mesh network by provisioning the devices. Here, provisioning implies authenticating and providing basic information to a device. A device must be provisioned to become a node. This example assumes all the nodes are already provisioned into a mesh network. The two types of bearers supported by the Bluetooth mesh are advertising bearer and GATT bearer. This example uses only the advertising bearer.

BLE Core Stack

This example models these layers of the BLE core stack:

- Generic access profile: This profile defines advertising data (AD) types for carrying mesh messages over the advertising bearer. This example supports 'Mesh message' AD type, which is used for exchanging network layer messages between mesh nodes.
- Link layer: This layer defines Broadcaster and Observer roles for message exchange between the nodes within the Bluetooth mesh network. In a Broadcaster role, a node always advertises. Whereas in an Observer role, the node always scans for the advertisers. Each node in the mesh network switches between these two roles to serve as a Bluetooth mesh node.
- Physical layer: This layer transmits and receives the waveforms for exchanging messages between the nodes within the Bluetooth mesh network. This layer models channel impairments such as free-space path loss, range propagation loss, and interference.

Discrete Event Simulation

DES is a type of simulation that models the functioning of a system as a discrete sequence of events in the time domain. Each event occurs at a specific time epoch and subsequently marks a change of state in the system. As a result, the simulation can directly jump from event to event in the time domain. The fundamental advantages of using DES in this example are:

- Its flexibility in time handling to suppress or expand, allowing the simulation to speed-up or slow-down the phenomena under investigation. This property of DES is used to model asynchronous transmissions in a multi-node Bluetooth network, resulting in accurate modeling of collisions.
- DES improves the simulation time performance and thus makes it feasible to support large-scale system-level simulations. For accurate modeling in a MATLAB implementation, simulations might need to run in microsecond steps. This will not only increase the simulation time but will also impact the network scalability. An increase in the step time might not allow you to capture or schedule events that occur in the microsecond granularity. DES enables you to address this issue by modeling events in discrete points in time.

Check for Support Package Installation

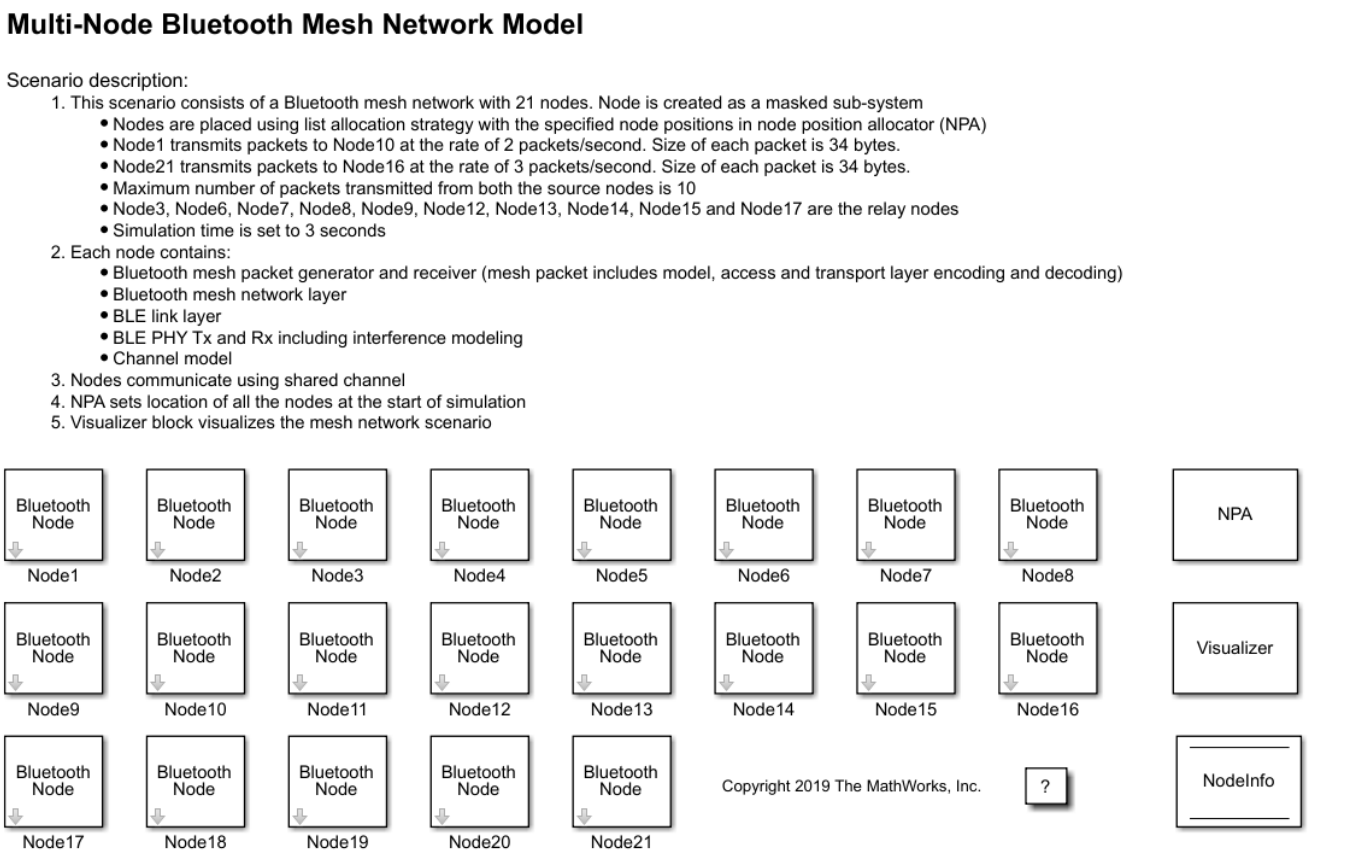
```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Multi-Node Bluetooth Mesh Network Model

This example models a Bluetooth mesh network with 21 nodes. The model outputs PDR of the network along with different statistics such as the number of transmitted, received, and dropped

packets at physical, link, and network layers, and also a plot visualizing the network scenario. The modeling includes:

- Multiple nodes, where each node contains a Bluetooth mesh packet generator and receiver (mesh packet includes model, access, and transport layer encoding and decoding), network layer, link layer, and physical layer
- A shared channel, which is simulated with these channel impairment options: range propagation loss, free-space path loss, and interference
- Packets transmitted over the shared channel
- A node position allocator (NPA) that configures the position of nodes in the network. NPA supports linear, grid, and list allocation strategies
- A visualizer that visualizes the mesh network scenario

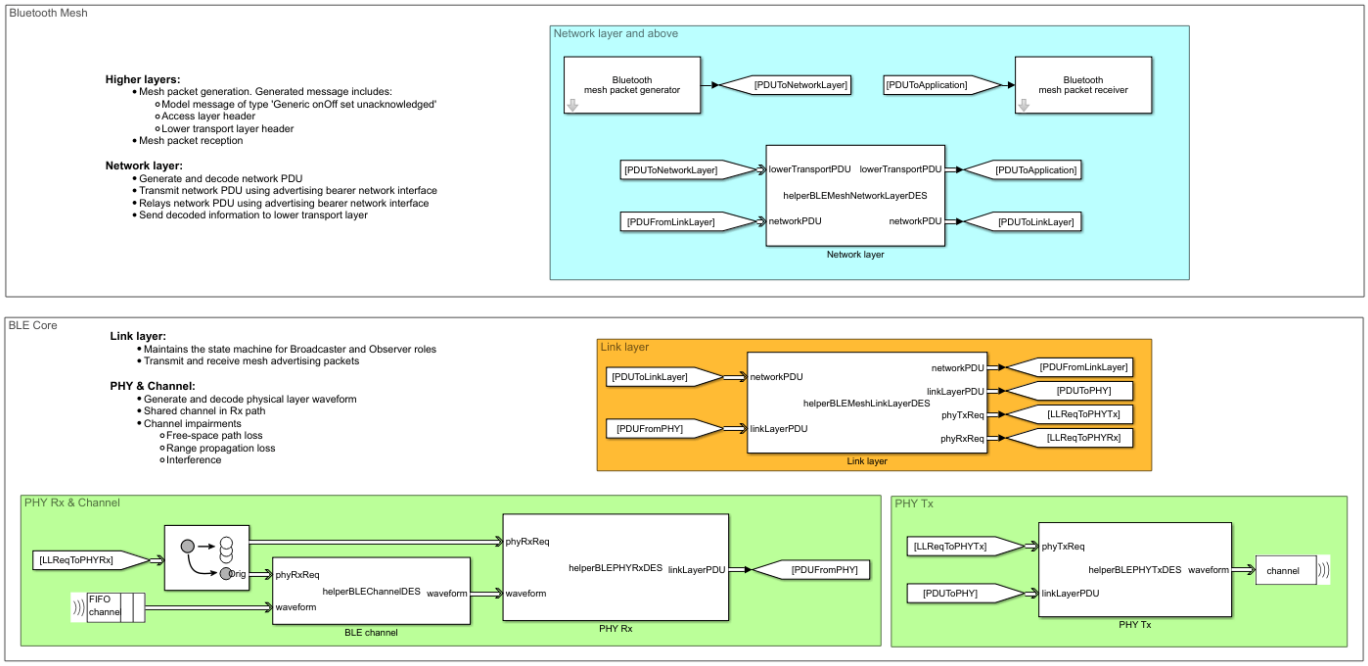


To configure a specific scenario, do one of these:

- Update the default configuration parameters for each node in the preceding model
- Specify the configuration as an input to helperBLEMeshCreateNetworkModel for creating a mesh network model

Bluetooth node

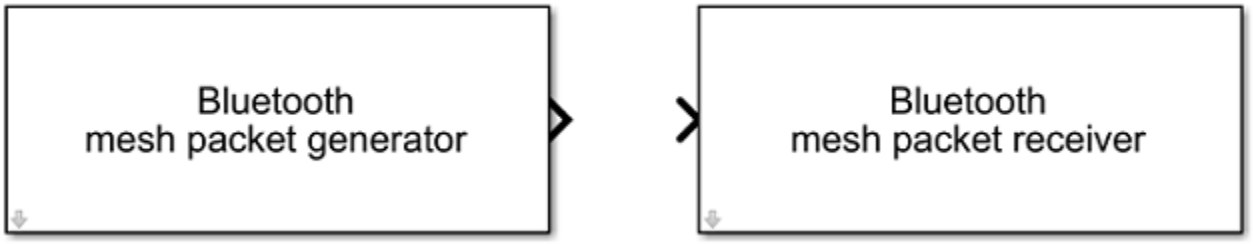
Each node is modeled as a subsystem with a network stack, which includes the Bluetooth mesh packet generator and receiver, network layer, LL, and PHY.



- The application layer generates packets by using the Entity Generator (SimEvents) block
- The MATLAB Discrete-Event System (SimEvents) block is used to model the network layer, LL, and PHY
- In each node, the shared channel is modeled in the receive path

Application layer

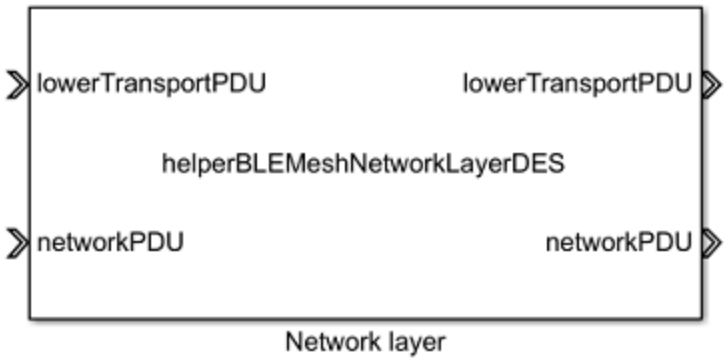
The application layer is implemented to generate and receive application traffic. It is divided into two sub-blocks:



- **Bluetooth mesh packet generator** This block uses the SimEvents Entity Generator block to generate lower transport data protocol data unit (PDU). The generated PDU contains the model layer message of type 'Generic onOff set unacknowledged' appended with higher layer headers. This PDU is passed to the network layer. You can configure the application state (On/Off), name of the destination node, source rate (in packets/second), and maximum number of packets that can be transmitted from source to destination by using this block. The block stops generating the packets once it has generated the maximum number of packets configured.
- **Bluetooth mesh packet receiver** This block uses the SimEvents Entity Terminator block to receive the output from the network layer

Network layer

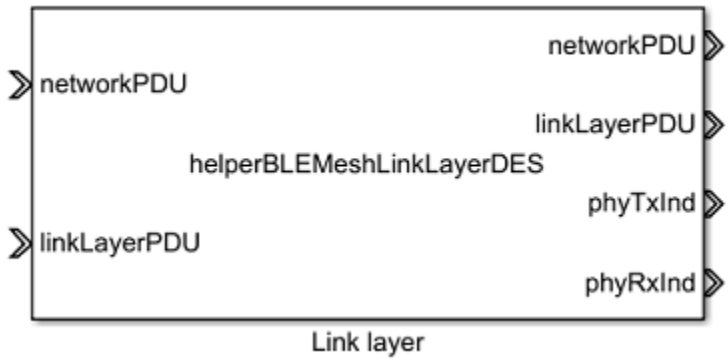
The network layer is modeled as a DES block. This block is responsible for transmitting the lower transport layer messages over the advertising bearer and relaying the mesh messages when the 'Relay' feature is enabled. When a network PDU is received, this block decodes the received PDU. If the PDU is decoded successfully, then the decoded information is passed to the lower transport layer.



You can configure the relay feature, network transmit interval, network transmit count, relay retransmit interval, and relay retransmit count by using mask parameters of the Network layer block.

Link layer

The link layer is modeled as a DES block. This block maintains a state machine for LL Broadcaster and Observer roles. This block is responsible for transmitting and receiving the mesh advertising packets by using `bleLLAdvertisingChannelPDU` and `bleLLAdvertisingChannelPDUDecode` functions.



You can configure scan and advertising intervals by using mask parameters of the Link layer block.

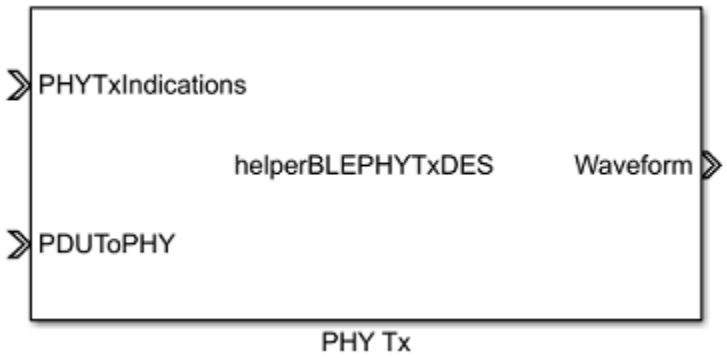
Physical layer

The PHY functionality includes:

- **Transmit chain**

LL initiates packet transmission by sending an LL packet and Tx indication to the PHY Tx block. This block generates a waveform for the received LL packet by using the `bleWaveformGenerator` function. It also scales the samples of the BLE waveform with the configured Tx power (assuming Tx

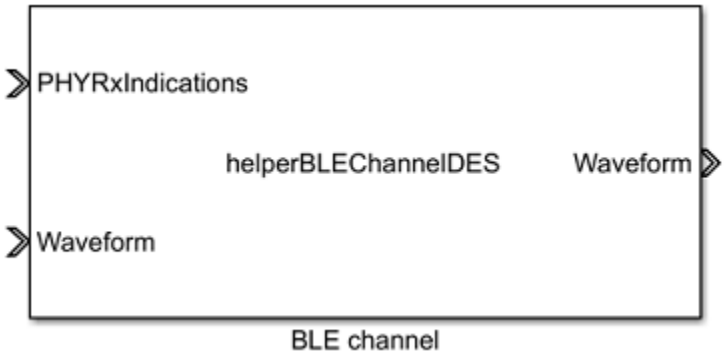
gain is 0). The generated BLE waveform is transmitted through the shared channel. The shared channel is modeled by using the SimEvents Multicast Queue.



You can configure the Tx power (dBm) by using mask parameters of the PHY Tx block.

- **Channel impairments modeling**

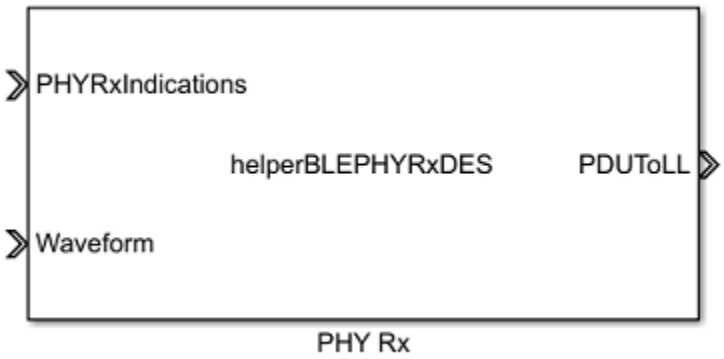
The free-space path loss model is added to the transmitted BLE waveform as channel impairments. You can choose to enable or disable this impairment. In addition to this impairment model, the signal reception range can also be limited by using an optional range propagation loss model. To model any of these channel impairment options, the channel model must contain the position of both the sender and the receiver. The channel is modeled inside each receiving node, before passing the BLE waveform to the PHY Rx block.



You can configure channel impairments by using mask parameters of the BLE channel block.

- **Receive chain**

This block applies thermal noise and interference to the received BLE waveform (assuming Rx gain is 0). Thermal noise is modeled by using the `comm.ThermalNoise` function with the configured value of the noise figure. Interference is modeled by adding the IQ samples of both the interfered and the actual signals. After applying thermal noise and interference, PHY Rx block decodes the resultant waveform. If the LL packet is decoded successfully, then it is passed to the LL.



You can configure the noise figure (in dB) using mask parameters of the PHY Rx block.

Node position allocator (NPA) Assigns the location of nodes in the mesh network. This block supports linear, grid, and list position allocation strategies.

- **Linear position allocation** Places nodes uniformly in a straight line on a 2D grid
- **Grid position allocation** Places nodes in a grid format specified by the grid properties
- **List position allocation** Assigns node positions from a list [[x1, y1, z1] [x2, y2, z2] ... [xn, yn, zn]] such that (xk, yk, zk) is the position of the kth node for all k in (1, 2, ..., n)

Visualizer This block is used to visualize the mesh network scenario in the simulation. You can configure this block to visualize the specified configuration. You can enable or disable visualization by using the mask parameters of this block.

Simulation Results

The results obtained in this simulation are:

- **Packet delivery ratio (PDR)**

The PDR is the ratio of number of received packets at the destination to the number of packets transmitted by the source and is given by:

$$\frac{\sum N_{\text{rx,destination}}}{\sum N_{\text{tx,source}}}$$

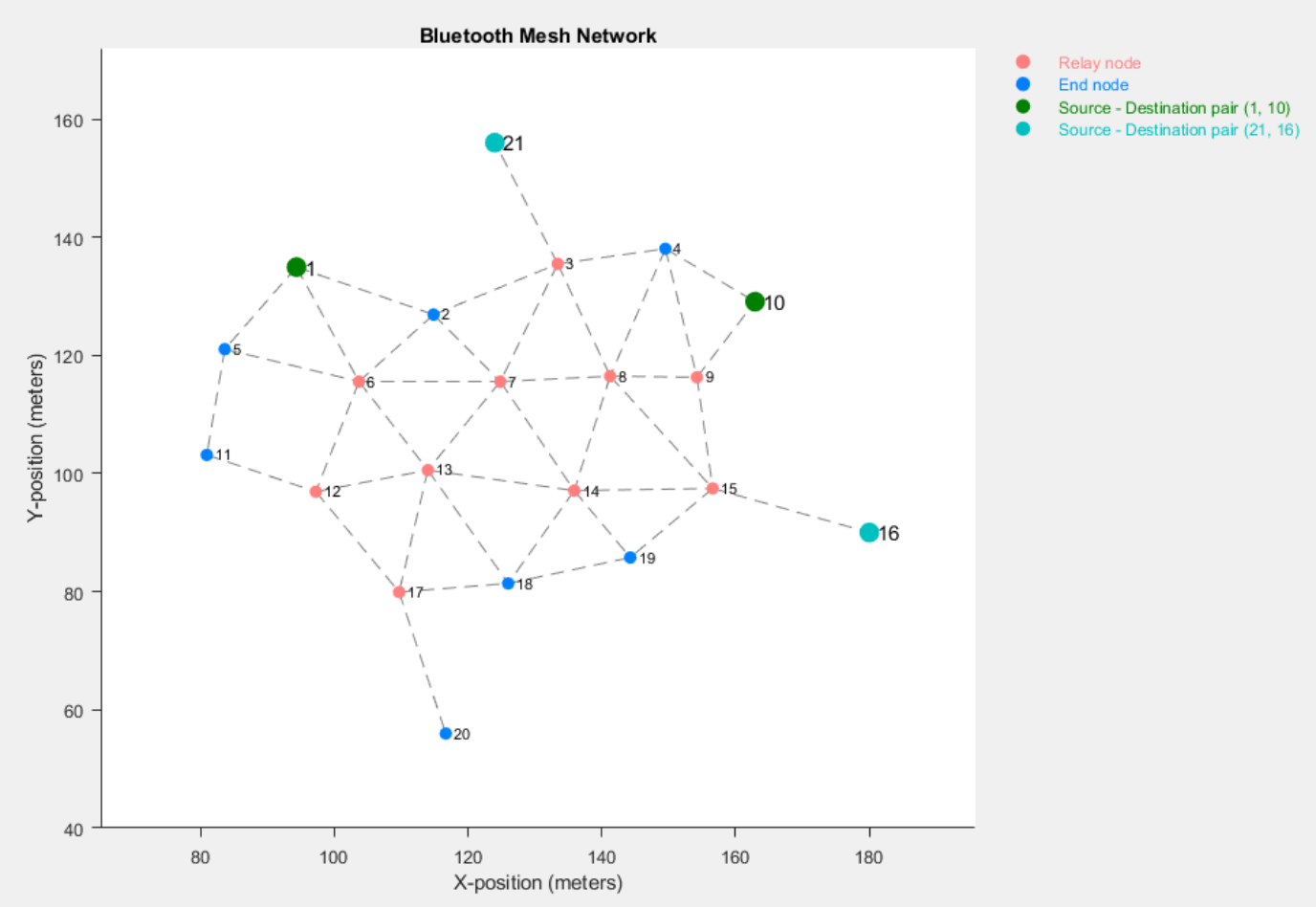
This model outputs PDR for this multi-node mesh network and is saved to a base workspace variable named `PDR`.

- **Statistics at each node**

This model outputs statistics of each node in the workspace variable `statisticsAtEachNode`. The statistics captured at each node are:

- Number of transmitted and received messages at the PHY
- Number of transmitted and received messages at the LL
- Number of messages received with CRC failures
- Number of transmitted, received, and dropped messages at the network layer
- Number of messages relayed at the network layer
- Number of received application messages at the network layer
- **Network visualization**

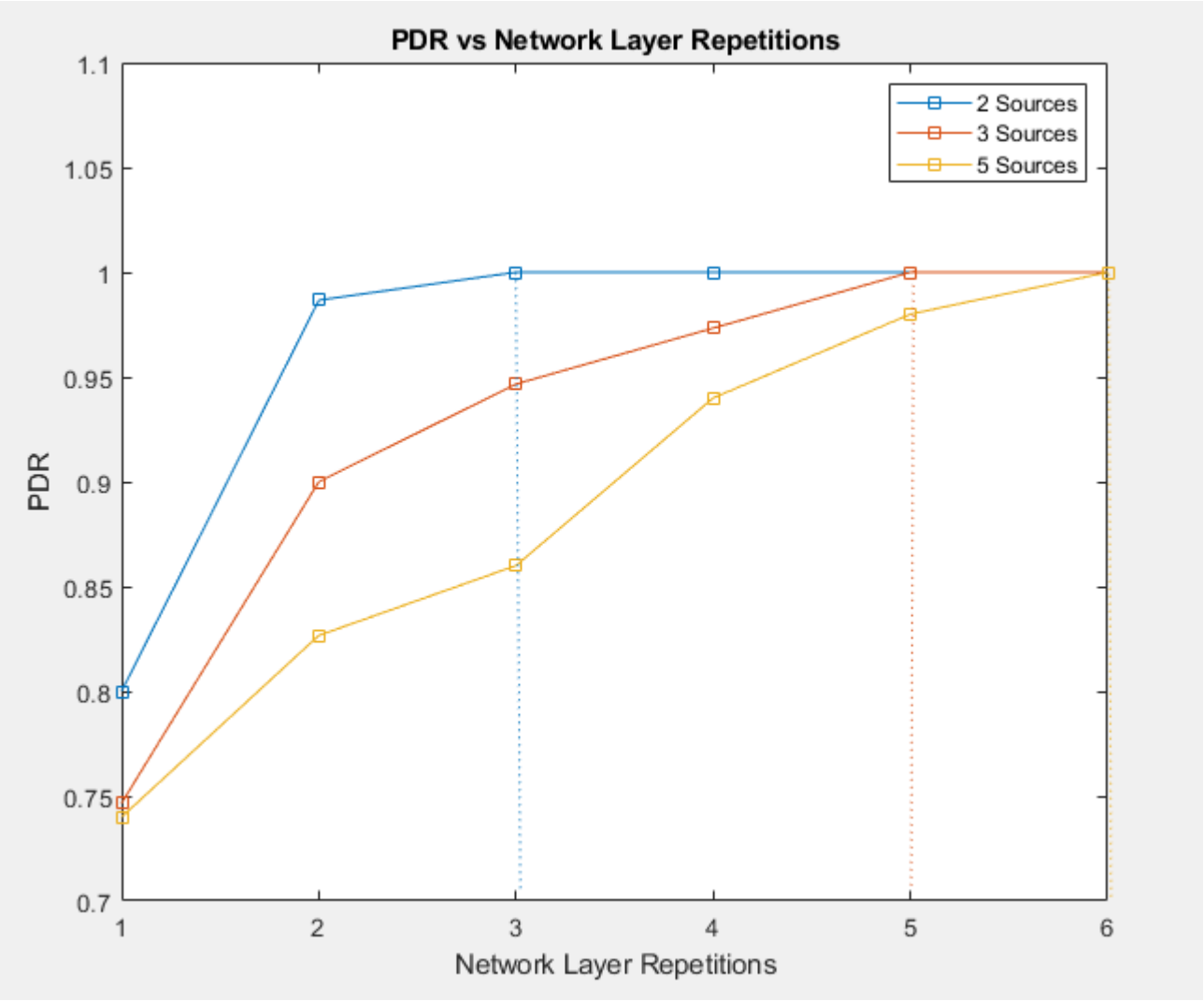
A plot with visual representation of the mesh network scenario is shown in the simulation. You can see the statistics of each node by placing your cursor over it.



This example shows how to configure and simulate a multi-node Bluetooth mesh network by using DES. The mesh network model in this example outputs PDR as a workspace variable with a visual representation of the mesh network.

Further Exploration

To observe the variation in the network PDR, you can vary the configuration parameters at the mesh packet generator, the network layer, LL and PHY. In these simulation results, you can see the impact of network layer repetitions (NLR) on the network PDR.



The NLR includes the repetitions of both the network messages and the relayed messages. The working principle of flood-based networks ensures that the message reaches the destination node. Therefore, it is important to retransmit the network and the relay messages. The number of NLR is dependent on the network configuration of the given network topology. Increasing the number of NLR ensures that the likelihood of the messages reaching the desired destination node is high. However, specifying a high value of the NLR can have adverse effects on the network performance parameters such as the overhead, energy consumption, and the duty cycle. As a result, it is essential to tune the value of NLR for a given network topology and achieve an efficient tradeoff between the PDR and network performance.

In the preceding figure you can see that the PDR increases with the NLR and decreases with the number of source nodes in the network. For a specific value of the NLR, the PDR value reaches 1 and thereafter it stabilizes. This specific value of the NLR might vary based on the network configuration parameters such as the total number of nodes, location of the nodes, number of source nodes, number of relay nodes, and so on. You can run helperBLEMeshDESPDRCalculation to reproduce these results by using three source nodes. Set the number of source nodes to two and five to get the corresponding results. You can run the simulations for any custom network scenario and get the optimal value of the NLR.

Apart from the NLR, the PDR varies with respect to multiple configuration parameters stated in helperBLEMeshDESPDRCalculation. You can further explore the mesh network model by varying any of these parameters.

Appendix

The example uses these features:

- `bleLLAdvertisingChannelPDUConfig`: Create a configuration object for the BLE Link Layer advertising channel PDU
- `bleLLAdvertisingChannelPDU`: Generate BLE Link Layer advertising channel PDU
- `bleLLAdvertisingChannelPDUDecode`: Decode BLE Link Layer advertising channel PDU
- `bleWaveformGenerator`: Generate BLE waveform

The example uses these helpers:

- helperBLEMeshAppGenericPDU: Generate Bluetooth mesh generic PDU
- helperBLEMeshAccessPDU: Generate Bluetooth mesh access PDU
- helperBLEMeshTransportDataMessage: Generate Bluetooth mesh transport data message
- helperBLEMeshNetworkLayer: Create an object for Bluetooth mesh network layer functionality
- helperBLEMeshNetworkLayerDES: Model Bluetooth mesh network layer
- helperBLEMeshNetworkPDU: Generate Bluetooth mesh network PDU
- helperBLEMeshNetworkPDUDecode: Decode Bluetooth mesh network PDU
- helperBLEMeshLLGAPBearer: Create an object for BLE LL advertising bearer functionality
- helperBLEMeshLinkLayerDES: Model Bluetooth mesh link layer
- helperBLEMeshGAPDataBlock: Generate advertising data with Bluetooth mesh network PDU
- helperBLEMeshGAPDataBlockDecode: Decode advertising data with Bluetooth mesh network PDU
- helperBLEPHYTransmitter: Create an object for BLE PHY transmitter
- helperBLEPHYTxDES: Generate and transmit the BLE waveform
- helperBLEChannel: Create an object for BLE channel model
- helperBLEChannelDES: Apply channel model on the received BLE waveform
- helperBLEPHYReceiver: Create an object for BLE PHY receiver
- helperBLEPHYRxDES: Receive and decode the BLE waveform
- helperBLEPracticalReceiver: Demodulate and decode the received signal
- helperBluetoothQueue: Create an object for Bluetooth queue functionality
- helperBLEMeshRetransmissions: Create an object for retransmissions in Bluetooth mesh
- helperBLEMeshVicinityNodes: Obtain the vicinity nodes of a given node
- helperBLEMeshGraphCursorCallback: Display the node statistics on mouse hover action
- helperBLEMeshVisualizeNetwork: Create an object for Bluetooth mesh network visualization
- helperBLEMeshAssignNodeIDs: Assigns node IDs to all the nodes in the model
- helperBLEMeshGetNodeNamesList: Get the list of nodes in the model
- helperBLEMeshCreateNetworkModel: Create a Bluetooth mesh network with given configuration
- helperBLEMeshUpdateStatistics: Create and update statistics in a Bluetooth mesh network simulation

References

1 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.0. https://www.bluetooth.com/.

2 Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Profile". Version 1.0. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Mesh Networking” on page 13-46
- “Create, Configure, and Visualize BLE Mesh Network” on page 13-93
- “Bluetooth Mesh Flooding in Wireless Sensor Networks” on page 3-159
- “Estimate Packet Delivery Ratio in Bluetooth Mesh Network” on page 3-117
- “Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks” on page 3-136

Bluetooth BR/EDR Waveform Generation and Transmission Using SDR

This example shows how to generate and transmit Bluetooth® BR/EDR waveforms using the Communications Toolbox™ Library for the Bluetooth Protocol. You can either transmit the Bluetooth BR/EDR waveforms by using the ADALM-PLUTO radio or write to a baseband file (*.bb).

To receive the transmitted Bluetooth BR/EDR waveform, see the Bluetooth BR/EDR Receiver example and implement any one of these setups:

- Two SDR platforms connected to the same host computer which runs two MATLAB sessions.
- Two SDR platforms connected to two host computers which runs two separate MATLAB sessions.

To configure your host computer to work with the Support Package for ADALM-PLUTO Radio, refer “Guided Host-Radio Hardware Setup” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Required Hardware

To transmit signals in real time, you need ADALM-PLUTO radio and the corresponding support package:

- Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of the Software Defined Radio (SDR) discovery page.

Bluetooth BR/EDR Radio Specifications

Bluetooth [1 on page 3-0] is a short-range Wireless Personal Area Network (WPAN) technology, operating in the globally unlicensed industrial, scientific, and medical (ISM) band in the frequency range 2.4 GHz to 2.485 GHz. In Bluetooth technology, data is divided into packets. Each packet is transmitted on one of the 79 designated Bluetooth channels. Each channel has a bandwidth of 1 MHz. As there are different types of wireless networks operating in the same unlicensed frequency band, it is possible for two different networks to interfere with each other. To mitigate the interference, Bluetooth implements the frequency-hopping spread spectrum (FHSS) scheme to switch a carrier between multiple frequency channels by using a pseudorandom sequence known to both the transmitter and receiver.

The Bluetooth standard specifies these physical layer (PHY) modes:

Basic rate (BR) - Mandatory mode, uses Gaussian frequency shift keying (GFSK) modulation with a data rate of 1 Mbps.

Enhanced data rate (EDR) - Optional mode, uses phase shift keying (PSK) modulation with these two variants:

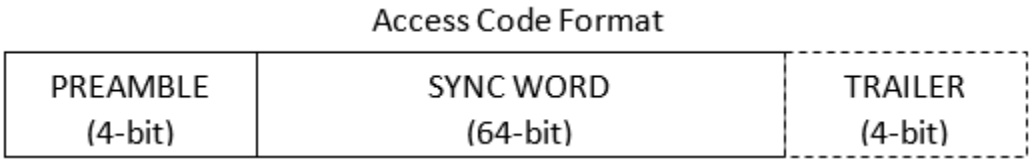
- EDR2M: Uses pi/4-DQPSK with a data rate of 2 Mbps.
- EDR3M: Uses 8-DPSK with a data rate of 3 Mbps.

Packet Formats

The air interface packet formats for PHY modes include these fields:

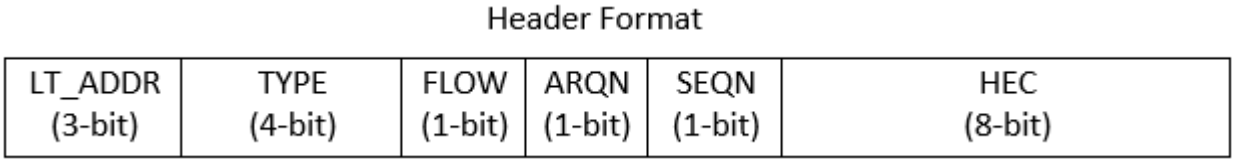
Access Code: Each packet starts with an access code. If a packet header follows, the access code is 72 bits long, otherwise the access code is 68 bits long. The access code consists of these fields:

- Preamble: The preamble is a fixed zero-one pattern of four symbols.
- Sync Word: The sync word is a 64-bit code word derived from 24-bit lower address part (LAP) of the Bluetooth device address.
- Trailer: The trailer is a fixed zero-one pattern of four symbols.

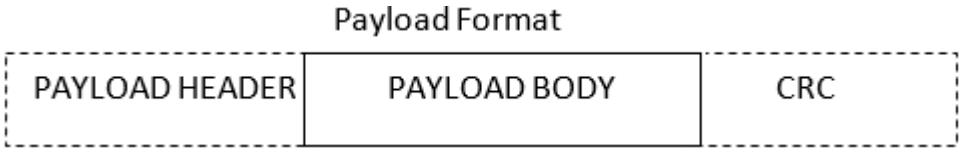


Packet Header: The header includes link control information and consists of these fields:

- LT_ADDR: 3-bit logical transport address.
- TYPE: 4-bit type code, which specifies the packet type used for transmission. It can be one of {ID, NULL, POLL, FHS, HV1, HV2, HV3, DV, EV3, EV4, EV5, 2-EV3, 2-EV5, 3-EV3, 3-EV5, DM1, DH1, DM3, DH3, DM5, DH5, AUX1, 2-DH1, 2-DH3, 2-DH5, 3-DH1, 3-DH3, 3-DH5}.
- FLOW: 1-bit flow control.
- ARQN: 1-bit acknowledgement indication.
- SEQN: 1-bit sequence number.
- HEC: 8-bit header error check.



Payload: Payload includes an optional payload header, a payload body, and an optional CRC.

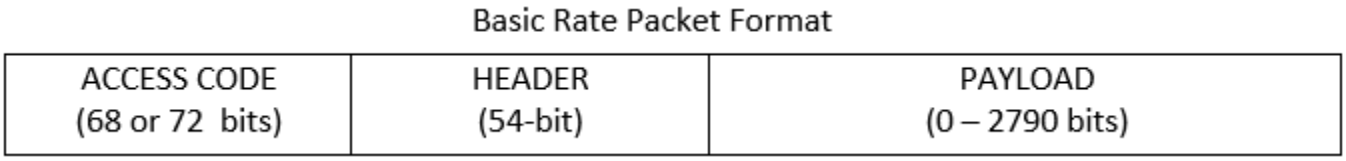


Guard: For EDR packets, guard time allows the Bluetooth radio to prepare for the change in modulation from GFSK to DPSK. The guard time must be between 4.75 to 5.25 microseconds.

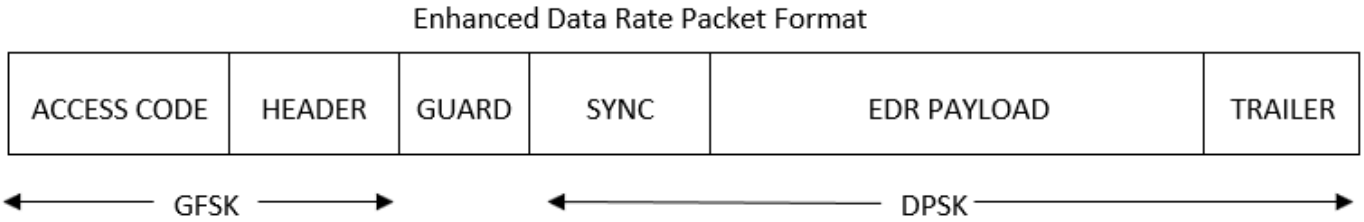
Sync: For EDR packets, the synchronization sequence contains one reference symbol and ten DPSK symbols.

Trailer: For EDR packets, the trailer bits must be all zero pattern of two symbols, {00,00} for pi/4-DQPSK and {000,000} for 8DPSK.

This figure shows the packet format for BR mode

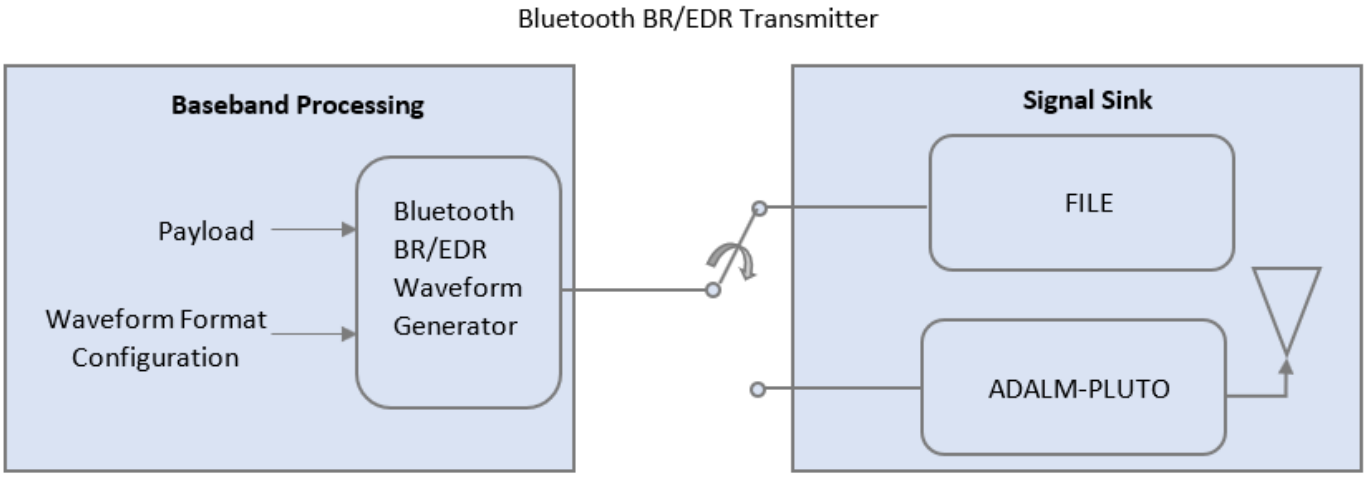


This figure shows the packet format for EDR mode



Bluetooth BR/EDR Waveform Generation and Transmission

This example shows how to generate Bluetooth BR/EDR waveforms according to the Bluetooth specification. The spectrum and spectrogram of the generated Bluetooth BR/EDR waveforms are visualized by using the spectrum analyzer. You can transmit the generated waveforms by using the ADALM-PLUTO radio or by writing them to a baseband file (*.bb).



Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed.
commSupportPackageCheck('BLUETOOTH');
```

Bluetooth BR/EDR Baseband Waveform Generation and Visualization

To configure the Bluetooth waveform generator for basic rate transmission, use the `bluetoothWaveformConfig` object.

```
cfg = bluetoothWaveformConfig;
cfg.Mode = BR; % Mode of transmission as one of BR, EDR2M and EDR3M
cfg.PacketType = 'FHS';     % Packet type
```

```
cfg.SamplesPerSymbol = 60; % Samples per symbol
cfg.WhitenInitialization = [0;0;0;0;0;1;1]; % Whiten initialization
```

To generate the Bluetooth BR/EDR waveforms, use the `bluetoothWaveformGenerator` function. Use `getPayloadLength` to determine the required payload length for the given configuration. Then use the payload length to create a random payload for transmission.

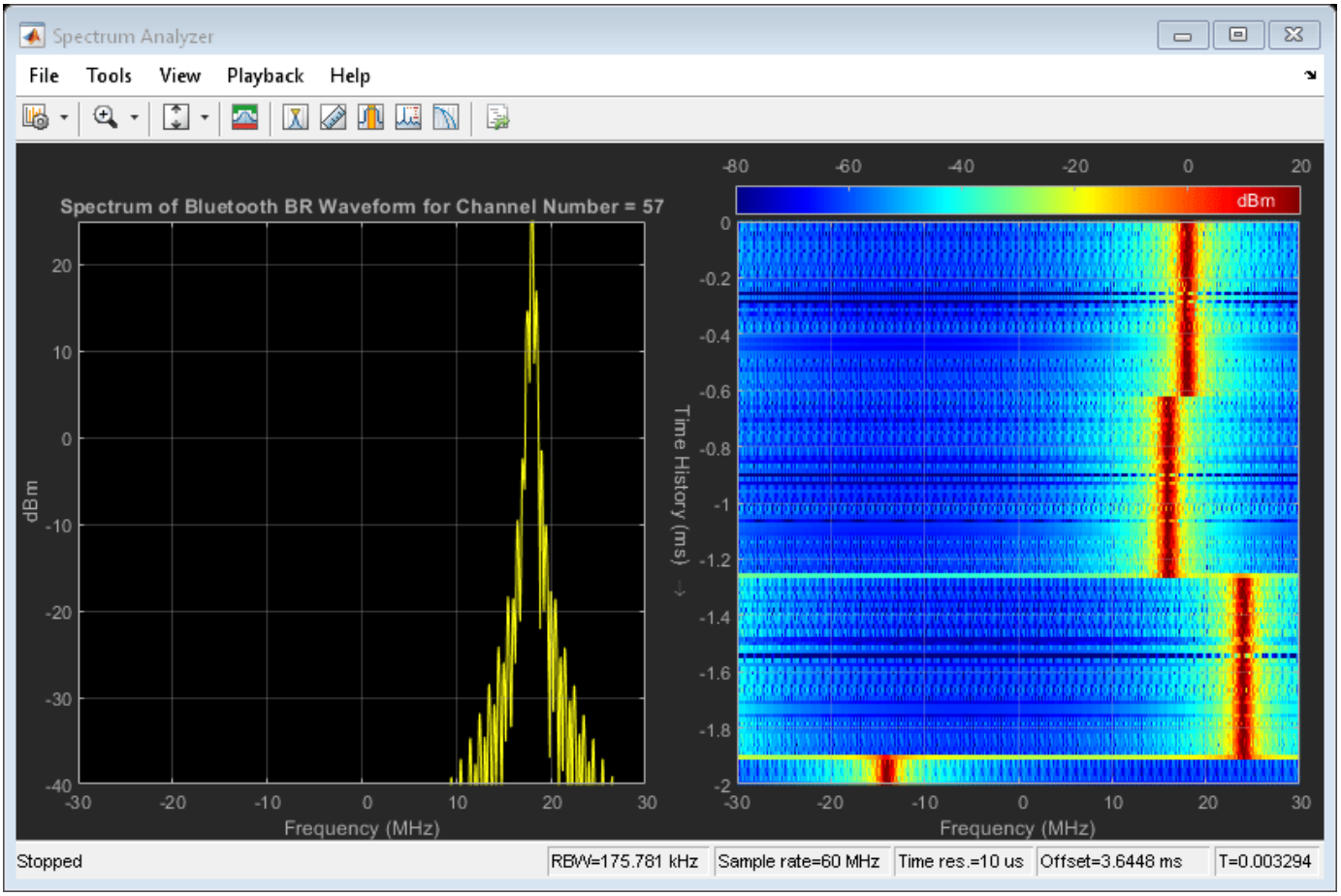
```
payloadLength = getPayloadLength(cfg); % Payload length in bytes
octetLength = 8;
dataBits = randi([0 1],payloadLength*octetLength,1); % Generate random payload bits
txWaveform = bluetoothWaveformGenerator(dataBits,cfg); % Create Bluetooth waveform
```

You can configure the function helperBluetoothPacketDuration.m to derive Bluetooth packet duration corresponding to the generated Bluetooth symbols.

```
packetDuration = helperBluetoothPacketDuration(cfg.PacketType,cfg.Mode,payloadLength);
```

The `comm.PhaseFrequencyOffset` System object is used to perform a frequency shift for Bluetooth BR/EDR waveforms based on the channel number. In this example, the waveform is visualized by using the `dsp.SpectrumAnalyzer` System object that selects a random channel number from the range 0 to 60 as sample rate used in this example is 60 MHz.

```
symbolRate = 1e6; % Symbol rate
sampleRate = symbolRate * cfg.SamplesPerSymbol;
numChannels = 10; % Number of channels

% Create and configure frequency offset System object
pfo = comm.PhaseFrequencyOffset('SampleRate',sampleRate);
% Create and configure spectrum analyzer System object
scope = dsp.SpectrumAnalyzer('ViewType','Spectrum and spectrogram',...
                'TimeResolutionSource','Property','TimeResolution',1e-5,...
                'SampleRate',sampleRate,'TimeSpanSource','Property',...
                'TimeSpan',2e-3,'FrequencyResolutionMethod','WindowLength',...
                'WindowLength',512,'AxesLayout','Horizontal','YLimits',[-40 25]);
% Loop over the number of channels to visaulize the frequency shift
for packetIdx = 1:numChannels
    channelNum = randsrc(1,1,0:60); % Generate random channel number
    freqIndex = channelNum - 39; % To visualize as a two sided spectrum
    pfo.FrequencyOffset = freqIndex*symbolRate; % Frequency shift
    hoppedWaveform = pfo(txWaveform(1:packetDuration*cfg.SamplesPerSymbol));
    scope.Title = ['Spectrum of Bluetooth ',cfg.Mode,...
                   ' Waveform for Channel Number = ', num2str(channelNum)];
    scope(hoppedWaveform);
end
% Release the System objects
release(scope);
```




```
release(pfo);
```

Transmitter Processing

Specify the signal sink as 'File' or 'ADALM-PLUTO'.

- **File**: Uses the `comm.BasebandFileWriter` System object to write a baseband file.
- **ADALM-PLUTO**: Uses the `sdrtx` (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) function to create a `comm.SDRTxPluto` (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) System object to transmit a live signal from the SDR hardware.

```
% Initialize the parameters required for signal sink

txCenterFrequency = 2445000000 ;  % In Hz, varies between 2.402e9 to 2.48
txFrameLength     = length(txWaveform);
txNumberOfFrames  = 1e4;
bbFileName        = 'bluetoothBRCaptures.bb';

% The default signal sink is 'File'

signalSink = File;

if strcmp(signalSink,'File')
    sigSink = comm.BasebandFileWriter('CenterFrequency',txCenterFrequency,...
```

```
        'Filename',bbFileName,...
        'SampleRate',sampleRate);
    sigSink(txWaveform); % Writing to a baseband file 'bluetoothBRCaptures.bb'
else % For 'ADALM-PLUTO'
    % Check if the pluto Hardware Support Package (HSP) is installed
    if isempty(which('plutoradio.internal.getRootDir'))
        error(message('comm_demos:common:NoSupportPackage', ...
                      'Communications Toolbox Support Package for ADALM-PLUTO Radio',...
                      ['<a href="https://www.mathworks.com/hardware-support/' ...
                      'adalm-pluto-radio.html">ADALM-PLUTO Radio Support From Communications Too
    end
    connectedRadios = findPlutoRadio; % Discover ADALM-PLUTO radio(s) connected to your computer
    radioID = connectedRadios(1).RadioID;
    sigSink = sdrtx('Pluto',...
        'RadioID',           radioID,...
        'CenterFrequency',   txCenterFrequency,...
        'Gain',              0,...
        'SamplesPerFrame',   txFrameLength,...
        'BasebandSampleRate',sampleRate);
    % The transfer of baseband data to the SDR hardware is enclosed in a
    % try/catch block. This implies that if an error occurs during the
    % transmission, the hardware resources used by the SDR System
    % object are released.
    currentFrame = 1;
    try
        while currentFrame <= txNumberOfFrames
            % Data transmission
            sigSink(txWaveform);
            % Update the counter
            currentFrame = currentFrame + 1;
        end
    catch ME
        release(sigSink);
        rethrow(ME);
    end
end

% Release the signal sink
release(sigSink);
```

In this example, you can generate and transmit Bluetooth BR/EDR waveforms by using ADALM-PLUTO or by writing the waveforms to a baseband file. The spectrum and spectogram of the generated Bluetooth BR/EDR waveforms is visualized by using a spectrum analyzer.

Further Exploration

You can use this example to transmit EDR packets by changing the mode of transmission. The example uses the helperBluetoothPacketDuration.m helper function to return the Bluetooth packet duration.

To decode the Bluetooth BR/EDR waveform generated in this example, refer to the Bluetooth BR/EDR Receiver example.

Troubleshooting

General tips for troubleshooting SDR hardware and the Communications Toolbox Support Package for ADALM-PLUTO Radio can be found in “Common Problems and Fixes” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Core System Package [BR/EDR Controller Volume]". *Bluetooth Core Specification*. Version 5.1, Volume 2. www.bluetooth.com

See Also

More About

- “Bluetooth BR/EDR Waveform Reception by Using SDR” on page 3-101

Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks

This example shows energy profiling of different types of nodes in a Bluetooth® mesh network using the Communications Toolbox™ Library for the Bluetooth® Protocol. Energy is computed based on the time profiled by End nodes, Low Power nodes (LPNs), Friend nodes, and Relay nodes in transmission, listen, sleep, and idle state. Using this example, you can:

- Create and configure a Bluetooth mesh network
- Visualize the impact of mesh message exchange on the energy performance of End node, LPN, Friend node, and Relay node
- Observe the energy consumption of mesh nodes by varying the number of source-destination pair, Friend node-LPN pair, and the application traffic.
- Estimate the node lifetime based on the hardware-specific energy parameters
- Modify the hardware-specific energy parameters to suit your requirements
- Explore the impact of poll timeout and receive window size on the node lifetime

The simulation calculates the lifetime of an LPN with the specified configuration and the hardware-specific energy parameters. The results validate that LPN always consume less energy by spending more time in sleep, resulting in energy conservation and increased lifetime.

Bluetooth Mesh Stack

The Bluetooth Core Specification [1] includes a Low Energy version for low-rate wireless personal area networks, referred to as Bluetooth low energy (BLE) or Bluetooth Smart. The BLE stack consists of generic attribute profile (GATT), attribute protocol (ATT), security manager protocol (SMP), logical link control and adaptation protocol (L2CAP), link layer (LL) and physical layer. The Special Interest Group (SIG) added BLE to the Bluetooth standard for low energy devices which generate small amounts of data such as notification alerts used in such applications as home automation, health-care, fitness, and Internet of Things (IoT). For more information about BLE protocol stack, see “Bluetooth Protocol Stack” on page 13-7.

The Bluetooth mesh profile [2] defines the fundamental requirements to implement a mesh networking solution for BLE. The mesh stack is located on top of the BLE core specification and consists of model layer, foundation model layer, access layer, upper transport layer, lower transport layer, network layer and bearer layer. Bluetooth mesh networking enables large-scale device networks in the applications such as smart lighting, industrial automation, sensor networking, asset tracking, and many other IoT solutions. For more information about Bluetooth mesh stack, see “Bluetooth Mesh Networking” on page 13-46.

Each Bluetooth mesh node can possess some optional features enabling them to acquire additional, special capabilities. These features include the Relay, Proxy, Friend, and the Low Power features. The Bluetooth mesh nodes possessing these features are known as Relay nodes, Proxy nodes, Friend nodes, and Low Power nodes (LPNs), respectively. To reduce the duty cycles of the LPN and conserve energy, the LPN must establish a *Friendship* with a mesh node supporting the Friend feature. This *Friendship* between the LPN and the Friend nodes (mesh nodes supporting the Friend feature) enables the Friend node to store and forward messages addressed to the LPN. Forwarding by the Friend node occurs only when the LPN wakes up and polls the Friend node for messages awaiting delivery. This mechanism enables all of the LPNs to conserve energy and operate for longer durations.

For more information about devices, nodes, and the Friendship in Bluetooth mesh network, see “Bluetooth Mesh Networking” on page 13-46.

The main objectives of this example are:

1. Create and configure a Bluetooth mesh network
2. Visualize message flooding
3. Analyze the behavior of Friendship in the Bluetooth mesh network
4. Profile the energy consumed by each node in the Bluetooth mesh network

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Bluetooth Mesh Energy Profiling Simulation

In the simulation, a source node initiates and relays a sample mesh message to a destination node. To relay mesh messages to multiple destination nodes, the source nodes transmits the messages to a common group address. During the simulation, the Friend nodes and LPNs exchange Friendship messages. Each node computes the time spent in various states (transmission, listen, idle and sleep) and calculates its lifetime.

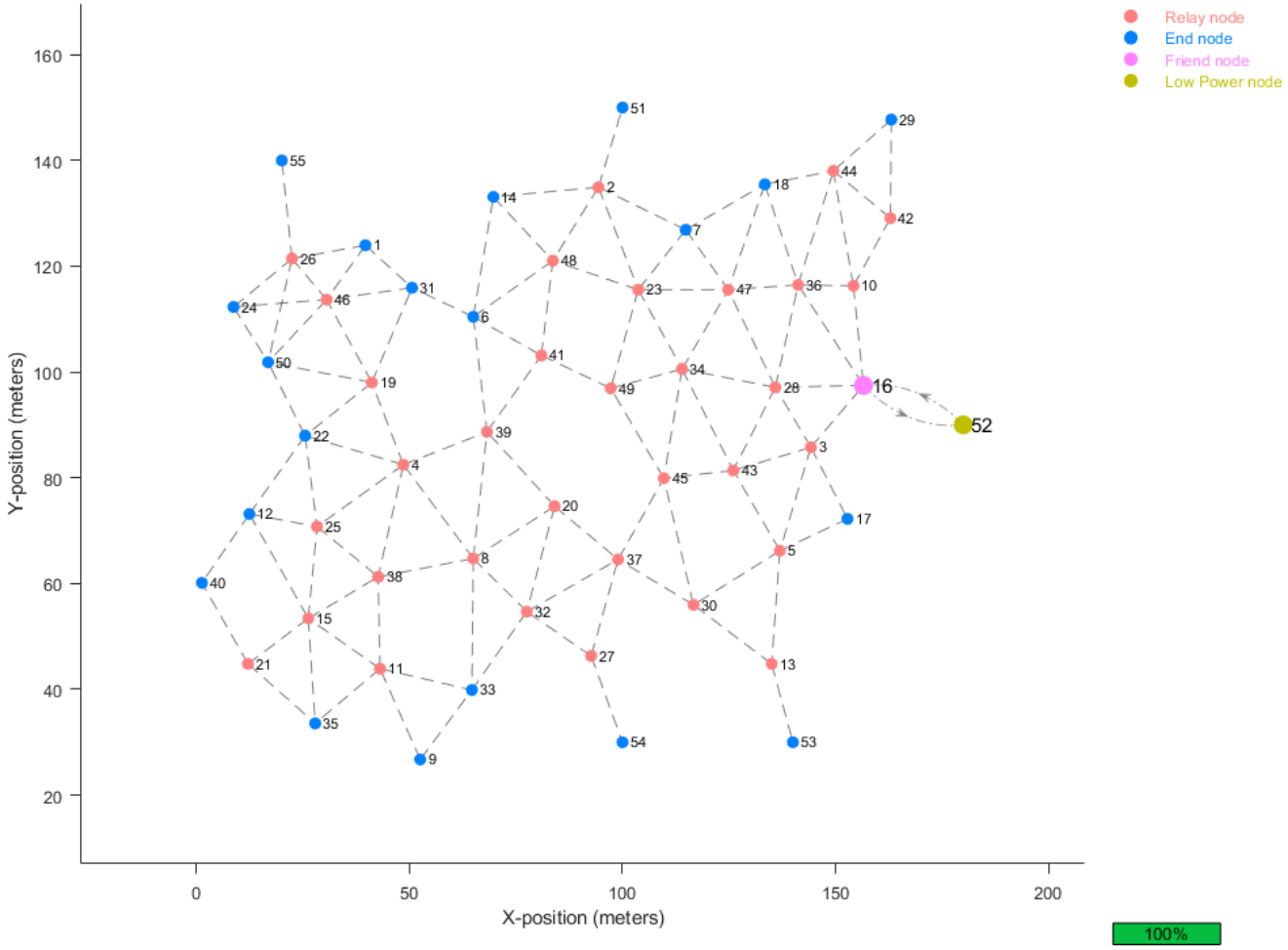
To create and visualize the mesh network, use helperBLEMeshNode and helperBLEMeshVisualizeNetwork classes. Specify the number of nodes (`NumberofNodes`) and the type of node position (`NodePositionType`) in helperBLEMeshVisualizeNetwork function. The default type of node position is 'Grid'. To specify your own network, set the value of `NodePositionType` to 'UserInput' and node positions to `Positions`.

```
% Set random number generator seed to 'default'
sprev = rng('default');

% Specify the number of nodes in the mesh network
totalNodes = 55;

% Initialize 'bleMeshNodes' vector with objects of type helperBLEMeshNode
meshNodes(1, totalNodes) = helperBLEMeshNode();

% Configure each mesh node with unique identifier
for nodeIdx = 1:totalNodes
    meshNode = helperBLEMeshNode();
    meshNode.Identifier = nodeIdx;
    meshNodes(nodeIdx) = meshNode;
end

% Load node positions from the MAT file
load('bleMeshNodesPositions.mat');

% Create and Configure the visualization object for Bluetooth mesh network
meshNetworkGraph = helperBLEMeshVisualizeNetwork();
meshNetworkGraph.NumberOfNodes = totalNodes;

% Set the type of the node position allocation as 'Grid' or 'UserInput'
meshNetworkGraph.NodePositionType = 'UserInput';
```

```
% Set node positions based on number of nodes (applicable for 'UserInput'),
% in meters
meshNetworkGraph.Positions = bleMeshNodesPositions;

% Set vicinity range (in meters) based on node positions, in meters
meshNetworkGraph.VicinityRange = 25;

% Set title to the network visualization
meshNetworkGraph.Title = 'Energy Profiling in Bluetooth Mesh Network';
```

Specify the number of source and destination pairs by using `sourceDestinationPairs` variable. To specify Friend node and LPN pairs, use `friendLowPowerPairs` variable. To specify the Relay nodes in the network, use `relayNodeIDs` variable. Configure the mesh node objects related to each mesh node. The `paths` variable store the paths obtained for each source and destination pair.

```
% Specify the simulation time (in milliseconds)
simulationTime = 6000;

% Enable or disable visualization
enableVisualization = true;

% Enable or disable the animation in the visualization. If
% "enableVisualization" is set to false, the simulation does not considers
% "enableAnimation".
enableAnimation = false;

% Specify the source and destination pairs. Source node transmits sample
% mesh message to destination node.
sourceDestinationPairs = [1 52; 1 17; 12 7; 6 53; 54 51; 9 33; 18 52; ...
    29 52; 31 7; 12 9; 54 53; 55 1; 9 17; 18 35];

% Specify the time to live (TTL) value (in the range [0, 127]) for each
% source and destination pair
ttl = [20 23 35 21 23 30 22 20 23 35 21 23 30 22];

% Specify the Friend node and LPN
friendLowPowerPairs = [16 52];

% Specify the receive window (in milliseconds) for each Friend and LPN
% pair. This value is in the range [120, 255]
receiveWindow = 180;

% Specify the poll timeout (in seconds) for each Friend and LPN pair. The
% value is in the range [2 seconds, 95.9 hours].
pollTimeout = 20;

% Specify the relay nodes
relayNodeIDs = [3 4 5 8 10 11 15 19 20 21 23 25 28 30 32 34 36 37 38 39 41 ...
    42 43 44 45 46 47 48 49 26 2 16 13 27];

% Simulate the Bluetooth mesh network
[meshNodes, paths] = helperBLEMeshSimulation(meshNodes, totalNodes, meshNetworkGraph, ...
    simulationTime, sourceDestinationPairs, ttl, friendLowPowerPairs, receiveWindow, ...
    pollTimeout, relayNodeIDs, enableVisualization, enableAnimation);

% Restore the previous setting of random number generation
rng(sprev);
```



Simulation Results

At each mesh node, the simulation captures these statistics.

- Time spent in transmission state
- Time spent in listening state
- Time spent in sleep state
- Time spent in idle state
- Number of messages transmitted from the node
- Number of messages received by the node
- Number of messages relayed by the node
- Number of messages dropped at the node
- Number of messages received with cyclic redundancy check (CRC) failures

The workspace variable, `statisticsAtEachNode`, contains the cumulative value of the preceding statistics for all the nodes in the network. For a given simulation run, you can view the statistics for first five nodes. The network statistics for the first five nodes in the network are:

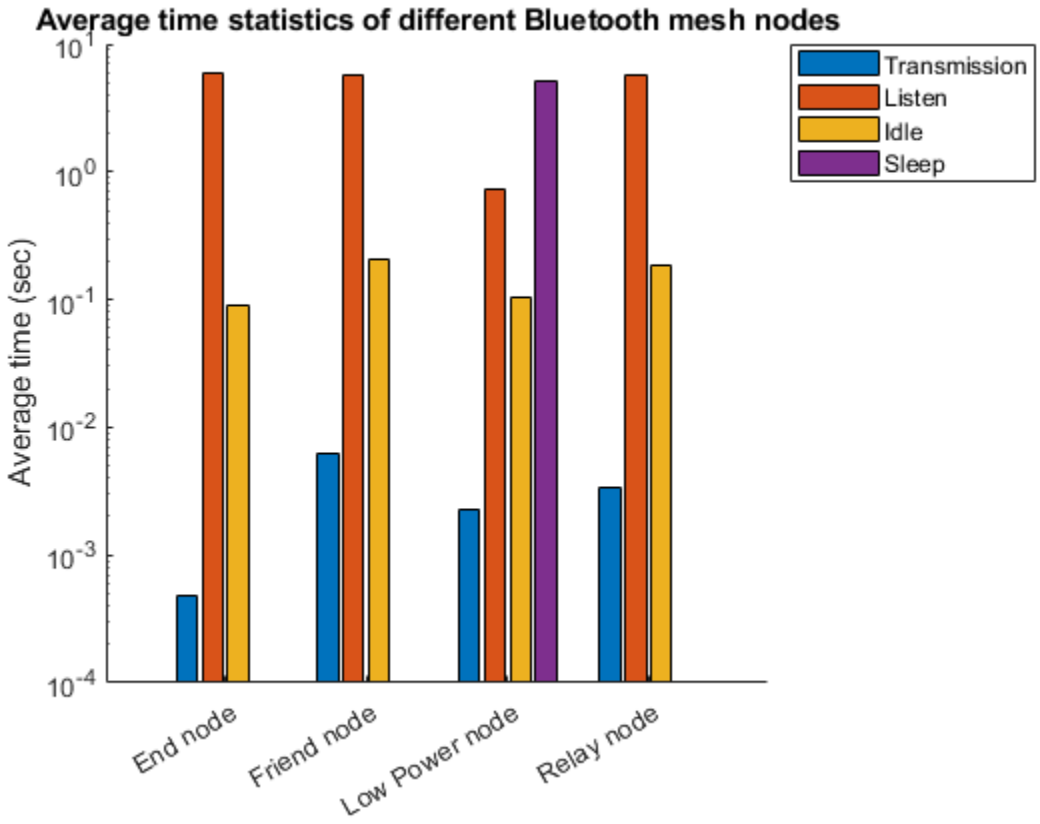
```
% Statistics for first five nodes
statisticsAtEachNode = helperBLEMeshNodesStatistics(meshNodes);
statisticsForFirstFiveNodes = statisticsAtEachNode(1:min(totalNodes, 5), :)
```

```
statisticsForFirstFiveNodes =

  5x14 table

             NodeType    TransmittedMsgs    ReceivedMsgs    ReceivedMsgsFromLPN    ReceivedAppli
             ________    _______________    ____________    ___________________    _____________

    Node_1     End              6               10                 0                        1
    Node_2     Relay           15                6                 0                        0
    Node_3     Relay           15               24                 0                        0
    Node_4     Relay           12               20                 0                        0
    Node_5     Relay           12               10                 0                        0
```

This plot shows the average time spent by different type of mesh nodes in different states. The results conclude that the LPN spend most of the time in sleep state, resulting in energy conservation and increased lifetime.

```
fprintf('Average time statistics of different Bluetooth mesh nodes are:\n');
meshNodesAvgStats = helperBLEMeshNodeAverageTime(meshNodes)
```

```
Average time statistics of different Bluetooth mesh nodes are:

meshNodesAvgStats =

  4x5 table

    Type of Bluetooth mesh node    Transmission time (milliseconds)    Listen time (milliseconds)
    ___________________________    ____________________________________    __________________________

          Low Power node                      2.304                              720
          Friend node                         6.192                           5771.5
          Relay node                         3.3869                           5801.4
          End node                           0.4836                           5907.4
```



The simulation consists of single message transmission from source node to destination node. Configure the traffic between the mesh nodes by using the pushModelMessage function periodically. The transmission time at the End node depends on the application traffic. The transmission time at the LPN depends on the poll timeout value.

Further Exploration

Calculate Lifetime of LPN:

Use helperBLEMeshNodeLifetime function to calculate the lifetime of a node in the Bluetooth mesh network at the end of simulation. To compute node lifetime, the `simulationTime` and the mesh node object of type helperBLEMeshNode is given as an input to the helperBLEMeshNodeLifetime function. The node lifetime is calculated by using the energy parameters that are hardware dependent. To update these hardware parameters, use helperBLEMeshNodeLifetime function.

```
% Fetch one of the Low Power nodes for calculating the lifetime
meshNode = meshNodes(52);
lifeTime = helperBLEMeshNodeLifetime(meshNode, simulationTime);
fprintf('Lifetime of node %d is %.4f days.\n', meshNode.Identifier, lifeTime);
```

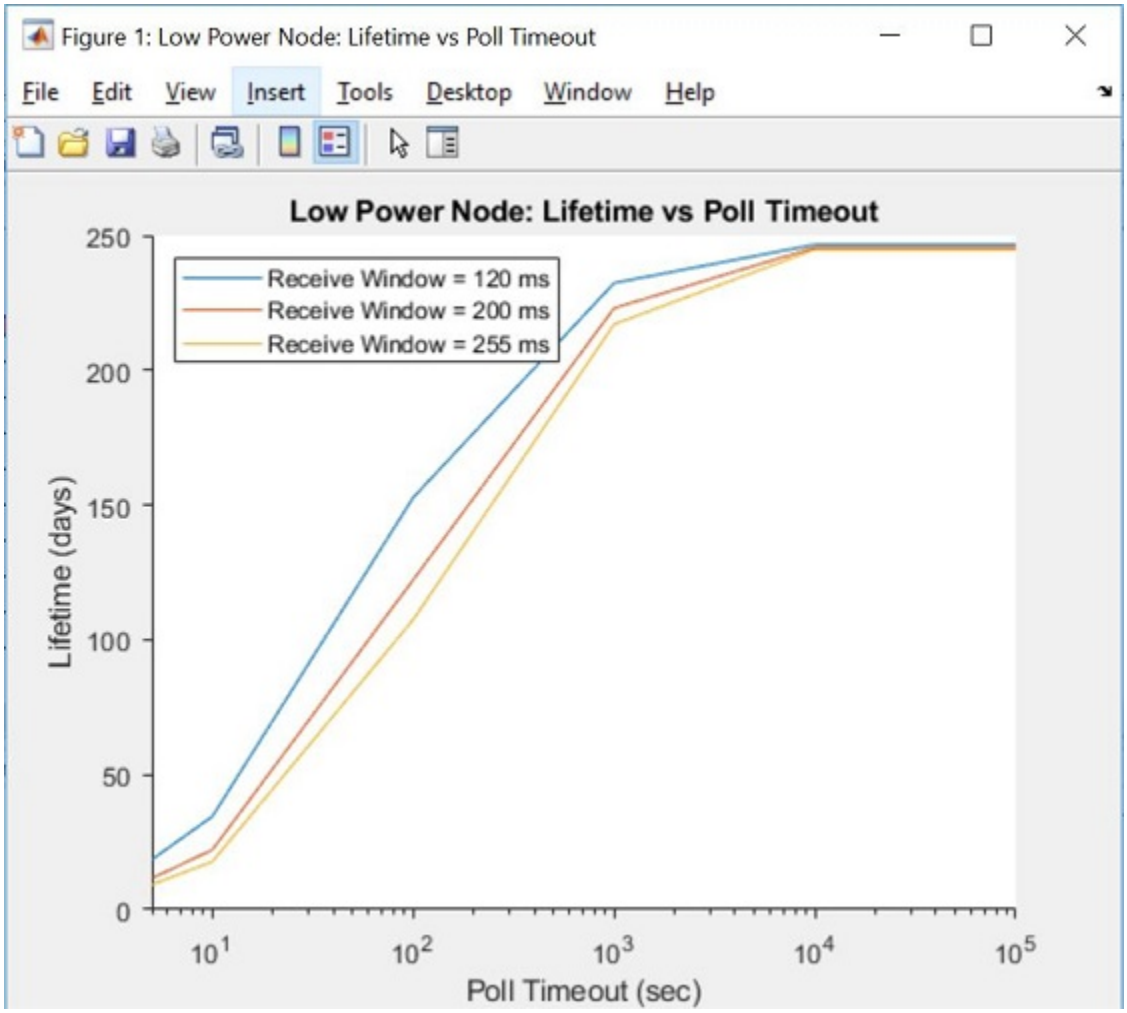
```
Configured hardware parameters for a 1200 mAh battery are:

hardwareParameters =

  7x2 table

      Hardware parameters        Configured values (mA)
```

```
    ____________________________    ____________________

    Self-discharge                        0.0013699
    Transmission on channel 37                 7.57
    Transmission on channel 38                 7.77
    Transmission on channel 39                  7.7
    Listening                                  10.3
    Sleep                                       0.2
    Idle                                       1.19

Statistics at node 52 are:

statisticsAtNode =

  4x2 table

     Time variables      Time (milliseconds)
    ________________    ____________________

    Transmission time            2.304
    Listen time                    720
    Sleep time                  5166.5
    Idle time                      103

Lifetime of node 52 is 34.8927 days.
```

Lifetime of LPN by Varying Poll Timeout

The lifetime of a LPN depends on the time for which the node is in the listen state. In a given poll timeout, a LPN is in listen or sleep state for most of the time. The receive window for each poll request of a LPN determines the time spent in listen state. The time spent in transmission state is negligible.

Visualize the impact of the poll timeout and receive window on the lifetime of LPN by using the helperBLEMeshLPNLifetimeVSPolltimeout function.



The preceding plot concludes that the lifetime of LPN is directly proportional to the poll timeout. The poll timeout specifies the maximum time between two consecutive requests from an LPN to Friend node. As the poll timeout increases, the LPN spends more time in sleep state that results in increasing the lifetime of the LPN.

This example shows how to create and configure a multinode Bluetooth mesh network and analyze the message exchange in the network. This example also enables you to analyze the behavior and the advantages of the Friendship between Friend node and LPN. To calculate the time spent by each node on different states the Bluetooth mesh node is simulated with multiple Friend and Low Power node pairs. The plot of the average time spent by each node in different states show that the LPNs always consume less energy by spending more time in sleep state. You can further explore the energy profiling of LPN by varying the poll timeout and receive window values.

Appendix

The example uses these features:

- `bleLLAdvertisingChannelPDUConfig`: Create a configuration object for BLE Link Layer advertising channel PDU
- `bleLLAdvertisingChannelPDU`: Generate BLE Link Layer advertising channel PDU

- `bleLLAdvertisingChannelPDUDecode`: Decode BLE Link Layer advertising channel PDU

The example uses these helpers:

- helperBLEMeshNode: Create an object for Bluetooth mesh node
- helperBLEMeshAccessLayer: Create an object for Bluetooth mesh access layer functionality
- helperBLEMeshNetworkLayer: Create an object for Bluetooth mesh network layer functionality
- helperBLEMeshTransportLayer: Create an object for Bluetooth mesh transport (upper and lower) layer functionality
- helperBLEMeshLowPowerNode: Create an object for Bluetooth mesh Low Power node functionality
- helperBLEMeshFriendNode: Create an object for Bluetooth mesh Friend node functionality
- helperBLEMeshFriendTimer: Create an object for Bluetooth mesh friend timer
- helperBLEMeshLLGAPBearer: Create an object for BLE LL advertising bearer functionality
- helperBLEMeshAppGenericPDU: Generate Bluetooth mesh generic PDU
- helperBLEMeshAppGenericPDUDecode: Decode Bluetooth mesh generic PDU
- helperBLEMeshLightnessPDU: Generate Bluetooth mesh lightness PDU
- helperBLEMeshLightnessPDUDecode: Decode Bluetooth mesh lightness PDU
- helperBLEMeshAccessPDU: Generate Bluetooth mesh access PDU
- helperBLEMeshAccessPDUDecode: Decode Bluetooth mesh access PDU
- helperBLEMeshNetworkPDU: Generate Bluetooth mesh network PDU
- helperBLEMeshNetworkPDUDecode: Decode Bluetooth mesh network PDU
- helperBLEMeshTransportControlMessage: Generate Bluetooth mesh transport control message
- helperBLEMeshTransportControlMessageDecode: Decode Bluetooth mesh transport control message
- helperBLEMeshTransportDataMessage: Generate Bluetooth mesh transport data message
- helperBLEMeshTransportDataMessageDecode: Decode Bluetooth mesh transport data message
- helperBLEMeshGAPDataBlock: Generate advertising data with Bluetooth mesh network PDU
- helperBLEMeshGAPDataBlockDecode: Decode advertising data with Bluetooth mesh network PDU
- helperBluetoothQueue: Create an object for Bluetooth queue functionality
- helperBLEMeshRetransmissions: Create an object for retransmissions in Bluetooth mesh
- helperBLEMeshNetworkChannelMessage: Receive message from Bluetooth mesh network channel
- helperBLEMeshPath: Return the path between source and destination within Bluetooth mesh network
- helperBLEMeshVicinityNodes: Obtain the vicinity nodes of a given node
- helperBLEMeshGraphCursorCallback: Display the node statistics on mouse hover action
- helperBLEMeshVisualizeNetwork: Create an object for Bluetooth mesh network visualization
- helperBLEMeshSimulation: Simulate a Bluetooth mesh network
- helperBLEMeshNodesStatistics: Collect statistics at each node into a table
- helperBLEMeshNodeLifetime: Compute lifetime of a Bluetooth mesh node

- helperBLEMeshNodeAverageTime: Compute average time spent in various states by the Bluetooth mesh nodes
- helperBLEMeshLPNLifetimeVSPolltimeout: Script to compute the lifetime of Bluetooth mesh Low Power node for different poll timeout and receive window values
- helperBLEPrependAccessAddress: Prepend the PDU with the access address

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.0. https://www.bluetooth.com/.

2 Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Profile". Version 1.0. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Mesh Networking” on page 13-46
- “Create, Configure, and Visualize BLE Mesh Network” on page 13-93
- “Bluetooth Mesh Flooding in Wireless Sensor Networks” on page 3-159
- “Estimate Packet Delivery Ratio in Bluetooth Mesh Network” on page 3-117

BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements

This example shows how to perform Bluetooth® low energy (BLE) radio frequency (RF) physical layer (PHY) transmitter tests specific to modulation characteristics, carrier frequency offset, and drift using Communications Toolbox™ Library for the Bluetooth Protocol. The test measurements compute frequency deviation, carrier frequency offset, and drift values. This example also verifies whether these test measurement values are within the limits specified by the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Objectives of BLE RF-PHY Tests

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by Bluetooth Special Interest Group (SIG) includes RF-PHY tests for both transmitter and receiver. The objectives of these RF-PHY tests are to:

- Ensure interoperability between all Bluetooth devices.
- Ensure a basic level of system performance for all Bluetooth products.

Each test case has a specified test procedure and an expected outcome, which must be met by the implementation under test (IUT).

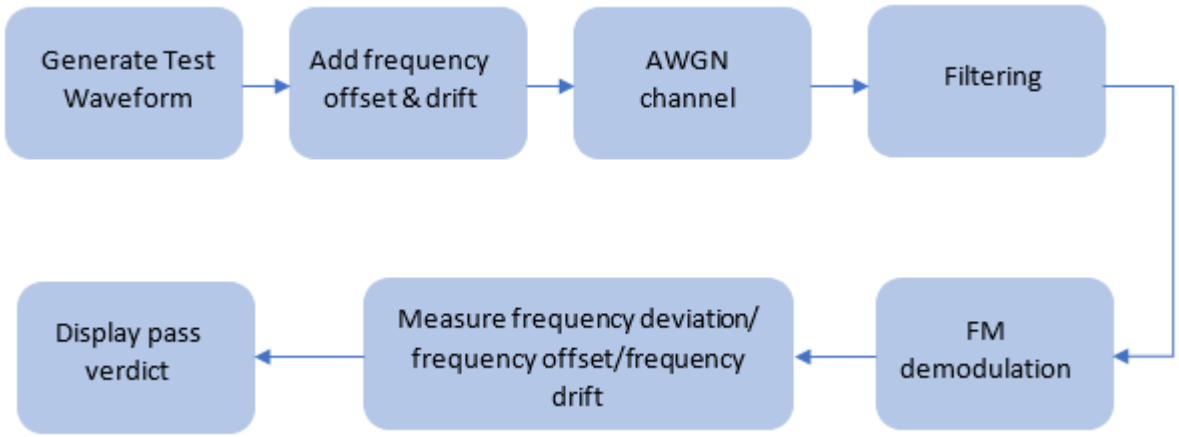
RF-PHY Transmitter Tests

The main aim of transmitter test measurements is to ensure that the transmitter characteristics are within the specified limits as specified in the test specifications [1 on page 3-0]. This example includes transmitter tests relevant to modulation characteristics, carrier frequency offset, and drift. This table shows various RF-PHY transmitter tests performed in this example.

Conformance Test	Test Case ID	Transmission Mode
Modulation Characteristics	RF-PHY/TRM/BV-09-C	LE 1M, uncoded data at 1 Mbps.
	RF-PHY/TRM/BV-11-C	LE 2M, uncoded data at 2 Mbps.
	RF-PHY/TRM/BV-13-C	LE 125K, coded data at 125 Kbps.
Carrier Frequency Offset and Drift	RF-PHY/TRM/BV-06-C	LE 1M, uncoded data at 1 Mbps.
	RF-PHY/TRM/BV-12-C	LE 2M, uncoded data at 2 Mbps.
	RF-PHY/TRM/BV-14-C	LE 125K, coded data at 125 Kbps.

Transmitter Test Procedure

This block diagram summarizes the test procedure for transmitter tests relevant to modulation characteristics, carrier frequency offset, and drift.



Generate test packets and pass them through `bleWaveformGenerator` to generate BLE test waveforms. The test waveforms required for different test IDs are:

Test Case ID	Test Waveforms
RF-PHY/TRM/BV-09-C & RF-PHY/TRM/BV-11-C	Generate two BLE test packets with repetitive sequence of 11110000b & 10101010b in transmission order.
RF-PHY/TRM/BV-13-C & RF-PHY/TRM/BV-14-C	Generate one BLE test packet with repetitive sequence of 11111111b.
RF-PHY/TRM/BV-06-C & RF-PHY/TRM/BV-12-C	Generate one BLE test packet with repetitive sequence of 10101010b.

Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed.

```
commSupportPackageCheck('BLUETOOTH');
```

Configure the Test Parameters

Initialize variables such as transmitter test ID, number of samples per symbol, payload length, and maximum carrier frequency drift. The function,helperBLEModulationTestConfig.m, can be configured to generate test parameters.

```
txTestID = RF-PHY/TRM/BV-0...;
payloadLen = 240;    % Payload length in bytes, must be in the range [37
sps = 32;            % Number of samples per symbol, minimum of 32 samples per
                     % symbol as per the test specifications

% Frequency offset and drift for the tests: RF-PHY/TRM/BV-06-C,
% RF-PHY/TRM/BV-12-C, RF-PHY/TRM/BV-14-C.
maxFreqDrift = 0;     % In Hz, must be in the range [-50e3,50e3]
initFreqOffset = 23000;   % In Hz, must be in the range [-100e3,100e3]
testParams = helperBLEModulationTestConfig(txTestID,sps); % Generate test parameters
```

Simulate Transmitter Tests

To simulate the transmitter tests, perform these steps:

1. Generate BLE test packet waveform using helperBLETestWaveform.
2. Add frequency offset, which includes initial frequency offset, and drift to the waveform using `comm.PhaseFrequencyOffset`.
3. Add thermal noise using `comm.ThermalNoise`.
4. Perform filtering on the noisy waveform using helperModulationTestFilterDesign.
5. Perform FM demodulation on the filtered waveform.
6. Perform test measurement and display the pass verdict.

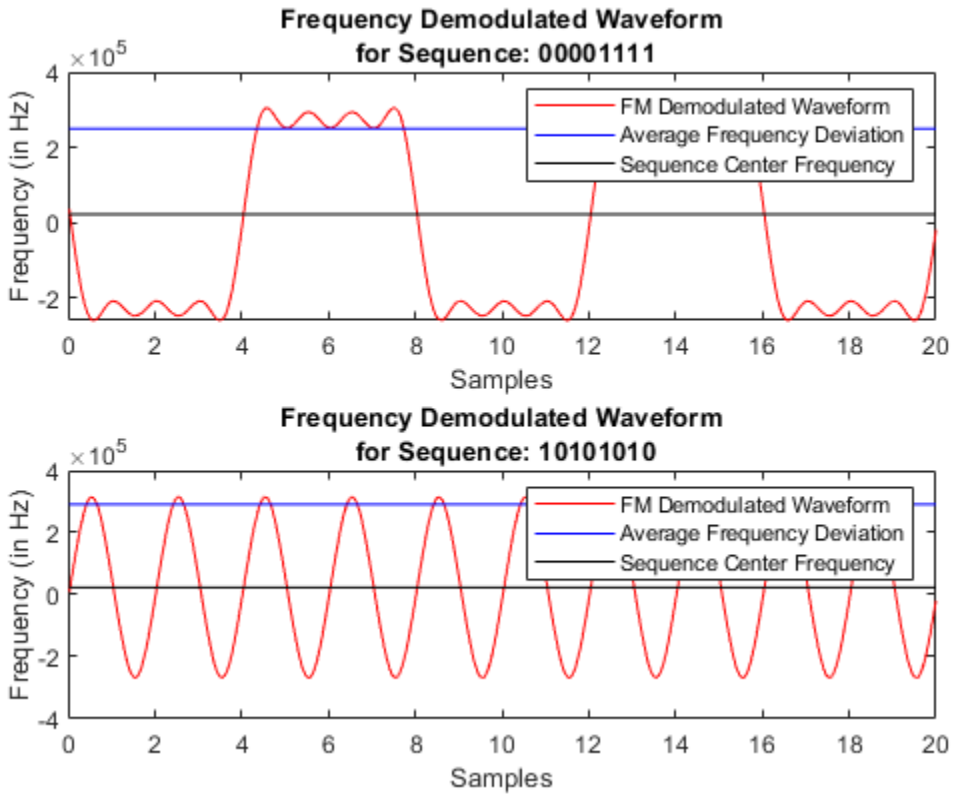
```
testWfmLen = (testParams.nonPDULen+testParams.codingFactor*payloadLen*testParams.bitsPerByte)*sps
driftRate = maxFreqDrift/length(testWfmLen);% Drift rate
freqDrift = driftRate*(0:1:(length(testWfmLen)-1))';% Frequency drift
freqOffset = freqDrift+initFreqOffset;% Frequency offset and frequency drift
% Create a phase frequency offset System object
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',freqOffset,'SampleRate',testParams.sampleRate)
% Create a thermal noise System object
NF = 12; % Noise figure (dB)
thNoise = comm.ThermalNoise('NoiseMethod','Noise figure',...
                            'SampleRate',testParams.sampleRate,...
                            'NoiseFigure',NF);
filtDesign = helperModulationTestFilterDesign(testParams.phyMode,sps);
filtTestWfm = zeros(testWfmLen,testParams.numOfTestSeqs);
for wfmIdx = 1:testParams.numOfTestSeqs
    % Generate BLE test waveforms
    testWfm = helperBLETestWaveform(testParams.testSeqIds(wfmIdx),...
                                    payloadLen,sps,testParams.phyMode);
    wfmFreqOffset = pfo(testWfm);
    wfmChannel = thNoise(wfmFreqOffset);
    filtTestWfm(:,wfmIdx) =  conv(wfmChannel,filtDesign.Coefficients.','same'); % Perform filteri
end
```

The function, helperBLEModulationTestMeasurements.m, performs FM demodulation and computes either frequency deviation, or frequency drift and initial frequency offset based on the provided test case ID.

```
[waveformDiffFreq,fOut1,fOut2,fOut3] = helperBLEModulationTestMeasurements(filtTestWfm,txTestID,
```

The function, helperBLEModulationTestVerdict.m, verifies whether the measurements are within the specified limits, and displays the verdict on the command window.

```
helperBLEModulationTestVerdict(waveformDiffFreq,txTestID,testParams,fOut1,fOut2,fOut3)
```




```
Test sequence: 00001111
    Measured average frequency deviation = 250 kHz
    Expected average frequency deviation = 247.5 kHz to 252.5 kHz
    Result: Pass
Test sequence: 10101010
    Expected 99.9% of all maximum frequency deviation > 185000 kHz
    Result: Pass
Ratio of frequency deviations between two test sequences = 1.163
Expected Ratio > 0.8
    Result: Pass
```

This example demonstrated the BLE transmitter test measurements specific to modulation characteristics, carrier frequency offset and, drift. The simulation results verify that these computed test measurement values are within the limits specified by Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

The helpers used in this example are:

- helperBLETestWaveform.m: Generates BLE test packet waveform
- helperBLEModulationTestConfig.m: Configures BLE transmitter test parameters
- helperBLEModulationTestMeasurements.m: Measures frequency deviation, carrier frequency offset and drift
- helperBLEModulationTestVerdict.m: Validates test measurement values and displays the result
- helperModulationCharacteristicsTest.m: Performs modulation characteristics test
- helperModulationTestFilterDesign.m: Designs channel filter

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). “Bluetooth RF-PHY Test Specification”, Revision: RF-PHY.TS.5.1.0, Section 4.4. 2018. https://www.bluetooth.com.

2 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.2. https://www.bluetooth.com.

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151
- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62

BLE Output Power and In-Band Emissions Test Measurements

This example shows how to perform transmitter test measurements specific to output power and in-band emissions on Bluetooth® Low Energy (BLE) transmitted waveforms as per the Bluetooth RF-PHY Test Specifications [1 on page 3-0] using Communications Toolbox™ Library for the Bluetooth® Protocol.

Objectives of BLE RF-PHY tests

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by Bluetooth Special Interest Group (SIG) includes RF-PHY tests for both transmitter and receiver. The objectives of these RF-PHY tests are to ensure interoperability between all BLE devices and to verify that a basic level of system performance is guaranteed for all BLE products. Each test case has a specified test procedure and an expected outcome, which must be met by the implementation under test (IUT).

RF-PHY Transmitter Tests

This example performs output power and in-band emissions test measurements according to the Bluetooth RF-PHY Test Specifications [1 on page 3-0]. The output power measurement is designed to ensure that power levels are high enough to maintain interoperability with other Bluetooth devices and low enough to minimize interference within the ISM band. The in-band emission test is to verify that the level of unwanted signals within the frequency range from the transmitter do not exceed the specified limits. The test case IDs corresponding to the tests considered in this example are as follows:

Output Power:

RF-PHY/TRM/BV-01-C: This test verifies the maximum peak and average power emitted from the IUT are within limits.

In-band Emissions:

- *RF-PHY/TRM/BV-03-C*: This test verifies that the in-band emissions are within limits when the transmitter is operating with uncoded data at 1 Ms/s.
- *RF-PHY/TRM/BV-08-C*: This test verifies that the in-band emissions are within limits when the transmitter is operating with uncoded data at 2 Ms/s.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Configure the Test Parameters

You can change `phyMode`, `Fc`, `outputPower` and `numDominantFreq` parameters based on the PHY transmission mode, frequency of operation, output power and number of dominant frequencies, respectively.

```
% Select PHY transmission mode {'LE1M','LE2M'} as per Bluetooth RF-PHY Test
% Specifications
phyMode = 'LE1M';
% Select frequency of operation for IUT based on the generic access profile
% (GAP) role(s) as shown in the table below.
```

```
%  ------------------------------------------------------------------------------
% | Operating |  Peripheral & Central Devices    |  Broadcaster & Observer Devices |
% | Frequency |                                  |                                 |
% | (MHz)     |----------------------------------|---------------------------------|
% |           | Output Power | In-band Emissions | Output Power | In-band Emissions|
% |           |     Test     |        Test       |     Test     |        Test      |
% |-----------|--------------|-------------------|--------------|------------------|
% | Lowest    |    2402      |       2406        |     2402     |       2402       |
% | Middle    |    2440      |       2440        |     2426     |       2440       |
% | Highest   |    2480      |       2476        |     2480     |       2480       |
%  ------------------------------------------------------------------------------
Fc = 2440e6; % Frequency of operation in Hz

payloadLength = 37                                ; % Payload length in bytes, must be in the range [37
sps = 32; % Number of samples per symbol, minimum of 32 sps as per the test specifications

outputPower = 20                                  ; % Output power in dBm, must be in the range [-20,20]

numDominantFreq = 6                               ; % Select number of dominant frequencies for in-ban
% be in the range [1,78] and [1,74] for LE1M and LE2M modes, respectively.

% The number of dominant frequencies represents the number of test
% frequencies near the operating frequency at which the in-band emissions
% test is to be performed. The number chosen in this example leads to a
% short simulation. For performing complete in-band emissions test, change
% the |numDominantFreq| parameter to maximum number of dominant frequencies
% as specified in the Section 4.4.2 of the Bluetooth RF-PHY Test
% Specifications.
```

Generate BLE Test Waveforms

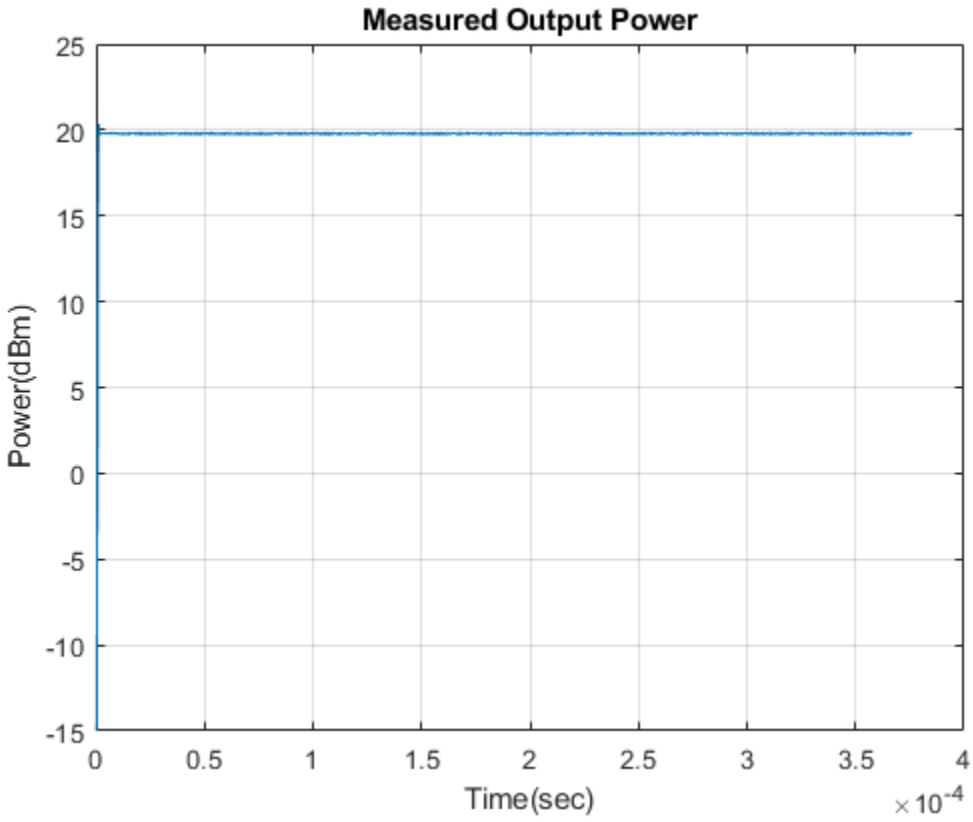
The function, helperBLETestWaveform.m, is configured to generate a BLE test waveform as per the Bluetooth specifications [2 on page 3-0].

```
payloadType = 0; % Payload type for PRBS9 sequence
waveform =  helperBLETestWaveform(payloadType,payloadLength,sps,phyMode);

% Calculate sampling rate in Hz based on PHY transmission mode
Rsym = 1e6;
if strcmp(phyMode,'LE2M')
    Rsym = 2e6;
end
Fs = Rsym*sps;

% Apply frequency upconversion to obtain a passband signal for the
% specified frequency of operation.
maxFreq = 2485e6; % in Hz
interpFactor = ceil(2*maxFreq/Fs); % Interpolation factor for upconversion to
                                   % cover BLE RF frequency band (2400e6 to 2485e6)

% Change the stopband frequency in Hz based on the PHY transmission mode
stopbandFreq = 2e6;
if strcmp(phyMode,'LE2M')
    stopbandFreq = 4e6;
end

% Create a digital upconverter System object
upConv = dsp.DigitalUpConverter(...
```

```
                'InterpolationFactor', interpFactor,...
                'SampleRate', Fs,...
                'Bandwidth', 2e6,...
                'StopbandAttenuation', 44,...
                'PassbandRipple',0.5,...
                'CenterFrequency',Fc,...
                'StopbandFrequencySource','Property',...
                'StopbandFrequency', stopbandFreq);

% Upconvert the baseband waveform to passband
dBdBmConvFactor = 30;
scalingFactor = 10^((outputPower-dBdBmConvFactor)/20);
upConvWaveform = scalingFactor*upConv(waveform);
```

Perform Output Power Test Measurement

```
rbwOutputPower = 3e6; % Resolution bandwidth, in Hz
% Frequency span must be zero so that the power measurement is performed in
% the time domain. Span 0 can be replicated by taking power values from the
% spectrogram at frequency of operation (Fc). Frequency limits are
% considered starting from frequency of operation (Fc) up to maximum
% frequency in the frequency band.
[P,F,T] = pspectrum(upConvWaveform,interpFactor*Fs,'spectrogram',...
                                'TimeResolution',1/rbwOutputPower,...
                                'FrequencyLimits',[Fc,maxFreq]);
powerAtFc = P(1,:); % Extract power values at Fc (F(1) = Fc)
% Calculate average power, AVGPOWER over at least 20% to 80% of the
% duration of the burst as specified in Section 4.4.1 of the Bluetooth
% RF-PHY Test Specifications.
powerAvgStartIdx = floor(0.2*length(powerAtFc));
powerAvgStopIdx = floor(0.8*length(powerAtFc));
avgPower = 10*log10(mean(powerAtFc(powerAvgStartIdx:powerAvgStopIdx)))+dBdBmConvFactor;
% Calculate peak power, PEAKPOWER
peakPower = 10*log10(max(powerAtFc))+dBdBmConvFactor;

% Plot power vs time
powerAtFcdBm = 10*log10(powerAtFc) + dBdBmConvFactor;
figure,plot(T,powerAtFcdBm)
grid on;
xlabel('Time(sec)');
ylabel('Power(dBm)');
title('Measured Output Power');
```




```
% Pass verdict - All measured values shall fulfill the following conditions:
%
% * Peak power <= (Average power + 3 dB)
% * -20dBm <= Average power <= 20dBm
%
fprintf('Measured average power and peak power are %f dBm and %f dBm, respectively.\n',avgPower,p
```

```
Measured average power and peak power are 19.792338 dBm and 20.362792 dBm, respectively.
```

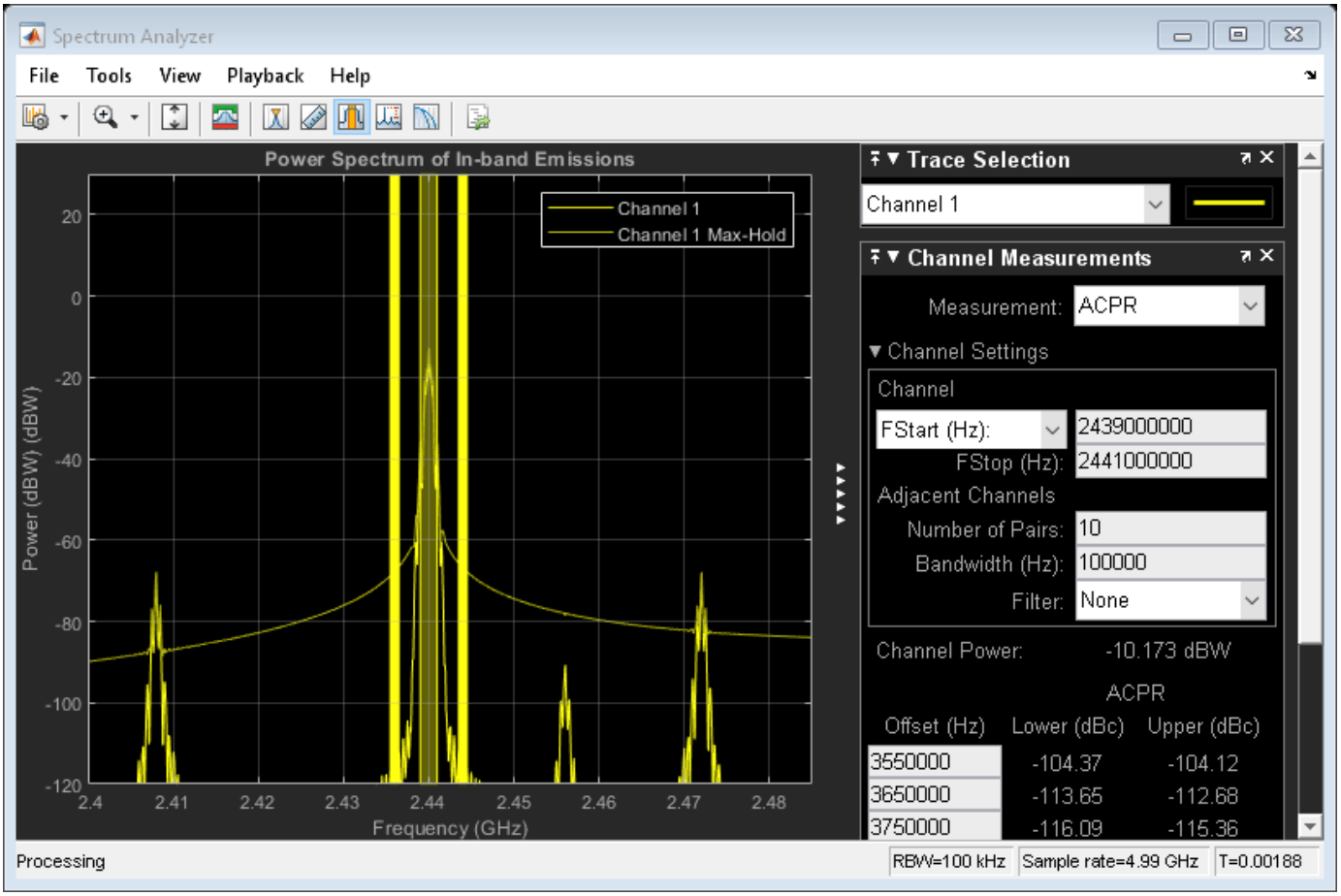
```
if (-20 <= avgPower <= 20) && (peakPower <= (avgPower+3))
    fprintf('Output power test passed.\n');
else
    fprintf('Output power test failed.\n');
end
```

```
Output power test passed.
```

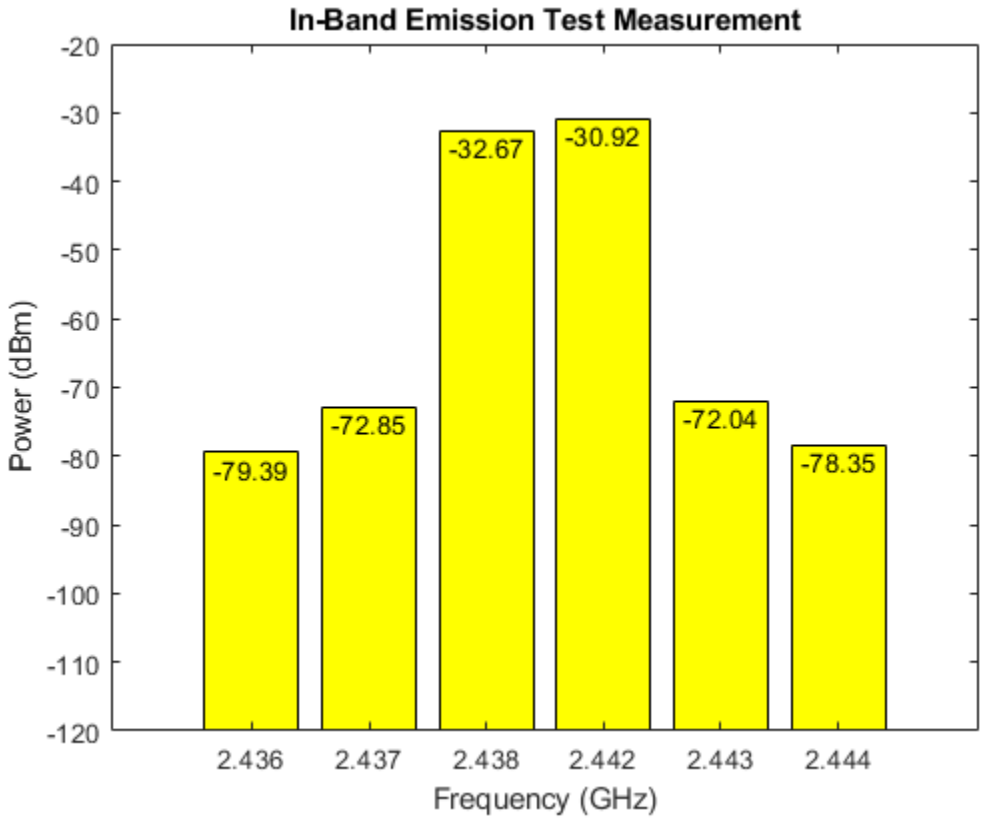
Perform In-band Emissions Test Measurement

```
% The function, <matlab:edit('helperBLEInbandEmissionsParams.m')
% helperBLEInbandEmissionsParams.m>, is configured to generate dominant
% test frequency parameters.
[testFreq,idx1,idx2] = helperBLEInbandEmissionsParams(Fc,numDominantFreq,phyMode);

% For each test frequency measure the power levels at the following 10
% frequencies.
numOffsets = 10;
```

```
freqOffset = -450e3+(0:numOffsets-1)*100e3;
adjChannelFreqOffsets = (freqOffset+testFreq-Fc).';

% Create and configure a spectrum analyzer for the waveform sampling rate
% and a resolution bandwidth of 100 kHz as specified in Section 4.4.2 of
% the Bluetooth RF-PHY Test Specifications.
rbw = 100e3; % Resolution bandwidth, in Hz
spectrumScope = dsp.SpectrumAnalyzer( ...
                'SampleRate',            Fs*interpFactor,...
                'SpectralAverages',      10, ...
                'YLimits',               [-120 30], ...
                'Title',                 'Power Spectrum of In-band Emissions',...
                'YLabel',                'Power (dBW)',...
                'SpectrumUnits',         'dBW',...
                'ShowLegend',            true,...
                'FrequencySpan',         'Start and stop frequencies',...
                'StartFrequency',         2400e6,...
                'StopFrequency',          maxFreq,...
                'RBWSource',             'Property',...
                'RBW',                    rbw,...
                'PlotMaxHoldTrace',       true,...
                'PlotAsTwoSidedSpectrum', false);

spectrumScope.ChannelMeasurements.Enable = true;
spectrumScope.ChannelMeasurements.Algorithm = 'ACPR';
spectrumScope.ChannelMeasurements.CenterFrequency = Fc;
spectrumScope.ChannelMeasurements.Span = 2e6; % Main channel bandwidth
spectrumScope.ChannelMeasurements.AdjacentBW = 1e5; % Adjacent channel bandwidth
spectrumScope.ChannelMeasurements.NumOffsets = numOffsets;

% Compute adjacent channel power ratio (ACPR) for the transmitted waveform
acpr = zeros(numOffsets,numDominantFreq);
for i = 1:numDominantFreq
    % Assign the 10 frequency offsets at each test frequency to ACPR Offsets
    spectrumScope.ChannelMeasurements.ACPROffsets = adjChannelFreqOffsets(:,i);

    % Estimate the power spectrum of the transmitted waveform using the spectrum analyzer
    spectrumScope(upConvWaveform);

    % Compute ACPR
    data = getMeasurementsData(spectrumScope); % Get the measurements data
    mainChannelPower = data.ChannelMeasurements.ChannelPower; % Main channel power at Fc
    acpr(:,i) = data.ChannelMeasurements.ACPRUpper; % Extract the ACPR values
end
```




```
% Power levels at 10 frequency offsets at each test frequency are
% calculated by adding main channel power to ACPR.
adjChannelPower = acpr(:,1:numDominantFreq) + mainChannelPower;

% Compute the power at each test frequency by adding all the powers
% measured at 10 frequency offsets.
adjPowerAtTestFreq = 10*log10(sum(10.^(adjChannelPower(:,1:numDominantFreq)/10))) + dBdBmConvFact

% Plot the adjacent channel powers
tick = 1:numel(adjPowerAtTestFreq);
ticklabel = testFreq/1e9;
figure;
bar(adjPowerAtTestFreq, 'BaseValue', -120, 'FaceColor', 'yellow');
set(gca, 'XTick', tick, 'XTickLabel', ticklabel, 'YLim', [-120 -20]);
for i = tick
    text(i, adjPowerAtTestFreq(i), sprintf('%0.2f',adjPowerAtTestFreq(i)), ...
        'HorizontalAlignment', 'Center', 'VerticalAlignment', 'Top');
end
title('In-Band Emission Test Measurement');
xlabel('Frequency (GHz)');
ylabel('Power (dBm)');
```




```
% Pass verdict- All measured values shall fulfill the following conditions:
% For LE1M PHY transmission mode
%
% * powerAtTestFreq <= -20 dBm for testFreq = Fc ± 2 MHz
% * powerAtTestFreq <= -30 dBm for testFreq = Fc ± [3+n] MHz; where
%   n=0,1,2,...
%
% For LE2M PHY transmission mode
%
% * powerAtTestFreq <= -20 dBm for testFreq = Fc ± 4 MHz AND testFreq = Fc
% ± 5 MHz
% * powerAtTestFreq <= -30 dBm for testFreq = Fc ± [6+n] MHz; where
%   n=0,1,2,...
%
for i = 1:numDominantFreq
    fprintf('Measured power at test frequency (Fc%+de6) is %.3f dBm.\n',(Fc-testFreq(i))*1e-6,adj
end

Measured power at test frequency (Fc+4e6) is -79.393 dBm.
Measured power at test frequency (Fc+3e6) is -72.852 dBm.
Measured power at test frequency (Fc+2e6) is -32.670 dBm.
Measured power at test frequency (Fc-2e6) is -30.924 dBm.
Measured power at test frequency (Fc-3e6) is -72.038 dBm.
Measured power at test frequency (Fc-4e6) is -78.351 dBm.

if (all(adjPowerAtTestFreq(idx1) <= -20)||isempty(idx1)) && (all(adjPowerAtTestFreq(idx2) <= -30
    fprintf('In-band emissions test passed.\n');
```

```
else
    fprintf('In-band emissions test failed.\n');
end
```

```
In-band emissions test passed.
```

This example demonstrated the transmitter test measurements specific to output power and in-band emissions on BLE transmitted waveforms as per the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

Appendix

This example uses the following helper functions:

- helperBLETestWaveform.m
- helperBLEInbandEmissionsParams.m

Selected Bibliography

1 Bluetooth RF-PHY Test Specification.

2 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Blocking, Intermodulation and Carrier to Interference Performance Tests” on page 3-168
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62
- “Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift” on page 3-71

Bluetooth Mesh Flooding in Wireless Sensor Networks

This example demonstrates network layer flooding in a Bluetooth® mesh network using Communication Toolbox Library™ for the Bluetooth® Protocol. Using this example, you can:

- Create and configure a Bluetooth mesh network by positioning the nodes in a grid.
- Specify your own network by configuring the node positions and the type of node position allocation.
- Classify and configure the mesh nodes as source, destination, relay, and end nodes and observe how network layer flooding helps in communication between the source and destination even after disabling few intermediate relay nodes.
- Visualize the flow of packets from source to destination.

The example also shows how to perform Monte Carlo simulations on the Bluetooth mesh network to obtain numerical results (like number of relay nodes required, critical relay nodes between the source and destination) averaged over multiple iterations.

Bluetooth Mesh Stack

The Bluetooth Core Specification [1] includes a Low Energy version for low-rate wireless personal area networks, referred to as Bluetooth low energy (BLE) or Bluetooth Smart. The BLE stack consists of generic attribute profile (GATT), attribute protocol (ATT), security manager protocol (SMP), logical link control and adaptation protocol (L2CAP), link layer (LL) and physical layer. BLE was added to the standard for low energy devices generating small amounts of data, such as notification alerts used in such applications as home automation, health-care, fitness, and Internet of things (IoT). For more information about BLE protocol stack, see “Bluetooth Protocol Stack” on page 13-7.

The Bluetooth mesh profile [2] defines the fundamental requirements to implement a mesh networking solution for BLE. The mesh stack is located on top of the BLE core specification and consists of model layer, foundation model layer, access layer, upper transport layer, lower transport layer, network layer and bearer layer. Bluetooth mesh networking enables large-scale device networks in the applications such as smart lighting, industrial automation, sensor networking, asset tracking, and many other IoT solutions. For more information about Bluetooth mesh stack, see “Bluetooth Mesh Networking” on page 13-46.

The Bluetooth mesh network layer performs these primary operations.

- Transmit upper layer messages over the network using the bearer layer
- Relay mesh messages
- Implement managed flooding to optimize network flooding
- Assign network addresses
- Configure network layer security

For more information about these mesh network layer operations, see “Bluetooth Mesh Networking” on page 13-46.

This example uses the advertising bearer to demonstrate Bluetooth mesh flooding in a wireless sensor network.

The main objectives of this example are:

1. Create and configure a Bluetooth mesh network
2. Visualize message flooding
3. Derive path between selected source and destination
4. Display statistics (refer Network Layer Statistics at Each Node) at each node

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Create and Configure Bluetooth Mesh Network Scenarios

This example enables you to create and configure two Bluetooth mesh network scenarios. Each scenario is a 50-node network. The nodes in the network are classified as relays, source, destination, and end nodes. Specify the corresponding time to live (TTL) values of source and destination nodes. In the first scenario, the example identifies the paths between the source and destination nodes. You can visualize the message flow in the network with network layer statistics. In the second scenario, the example disables some relay nodes and end nodes. In this case, the simulations show that the network has the likelihood of establishing a path between the specified source and destination pair.

To create and visualize the mesh network, use helperBLEMeshNetworkNode and helperBLEMeshVisualizeNetwork functions. Specify number of nodes (`totalNodes`) and the type of node position (`NodePositionType`) in helperBLEMeshVisualizeNetwork function. The default type of node position is 'Grid'. To specify your own network, set the value of `NodePositionType` to 'UserInput' and node positions to `Positions`.

```
% Set random number generator seed to 'default'
sprev = rng('default');

% Specify the number of nodes in the mesh network
totalNodes = 50;

% Initialize 'bleMeshNodes' vector with objects of type
% helperBLEMeshNetworkNode
bleMeshNodes(1, totalNodes) = helperBLEMeshNetworkNode;

% Configure each mesh node with unique identifier
for idx = 1:totalNodes
    meshNode = helperBLEMeshNetworkNode;
    meshNode.Identifier = idx;
    meshNode.NetworkLayer.ElementAddresses = dec2hex(idx, 4);
    bleMeshNodes(idx) = meshNode;
end

% Load node positions from the MAT file
load('bleMeshNetworkNodePositions.mat');

% Number of scenarios simulated in this examples
numberOfScenarios = 2;

% Initialize 'meshNetworkPlots' vector with objects of type
% helperBLEMeshVisualizeNetwork
meshNetworkPlots(1, numberOfScenarios) = helperBLEMeshVisualizeNetwork;

for idx = 1:numberOfScenarios
```

```
    meshNetworkPlots(idx) = helperBLEMeshVisualizeNetwork;
    meshNetworkPlots(idx).NumberOfNodes = totalNodes;
    % Set the type of the node position allocation as 'Grid' or
    % 'UserInput'
    meshNetworkPlots(idx).NodePositionType = 'UserInput';

    % Set node positions based on number of nodes (applicable for
    % 'UserInput'), in meters
    meshNetworkPlots(idx).Positions = bleMeshNetworkNodePositions;

    % Set vicinity range based on node positions, in meters
    meshNetworkPlots(idx).VicinityRange = 25;

    % Set title to the network visualization
    meshNetworkPlots(idx).Title = ...
        ['Scenario ' num2str(idx) ': Bluetooth Mesh Flooding'];
end
```

Specify the number of source and destination pairs in the mesh network using `srcDstPairs` parameter. Specify the TTL values for the packet originated at each source node.

```
% Specify the simulation time in milliseconds
simulationTime = 600;

% Enable or disable visualization
enableVisualization = true;

% Specify the source and destination pairs
srcDstPairs = [1 7; 13 29];

% Specify TTL values for packet originated at each source node
ttl = [25; 25];
```

Simulations

To run the simulation and get the results, use helperBLEMeshFloodingSimulation and helperBLEMeshFloodingSimulationResults functions, respectively.

- **Scenario 1:** In this scenario, all the fifty nodes in the network are active. Some of these nodes are selected as relays and there are no failed nodes in this scenario.

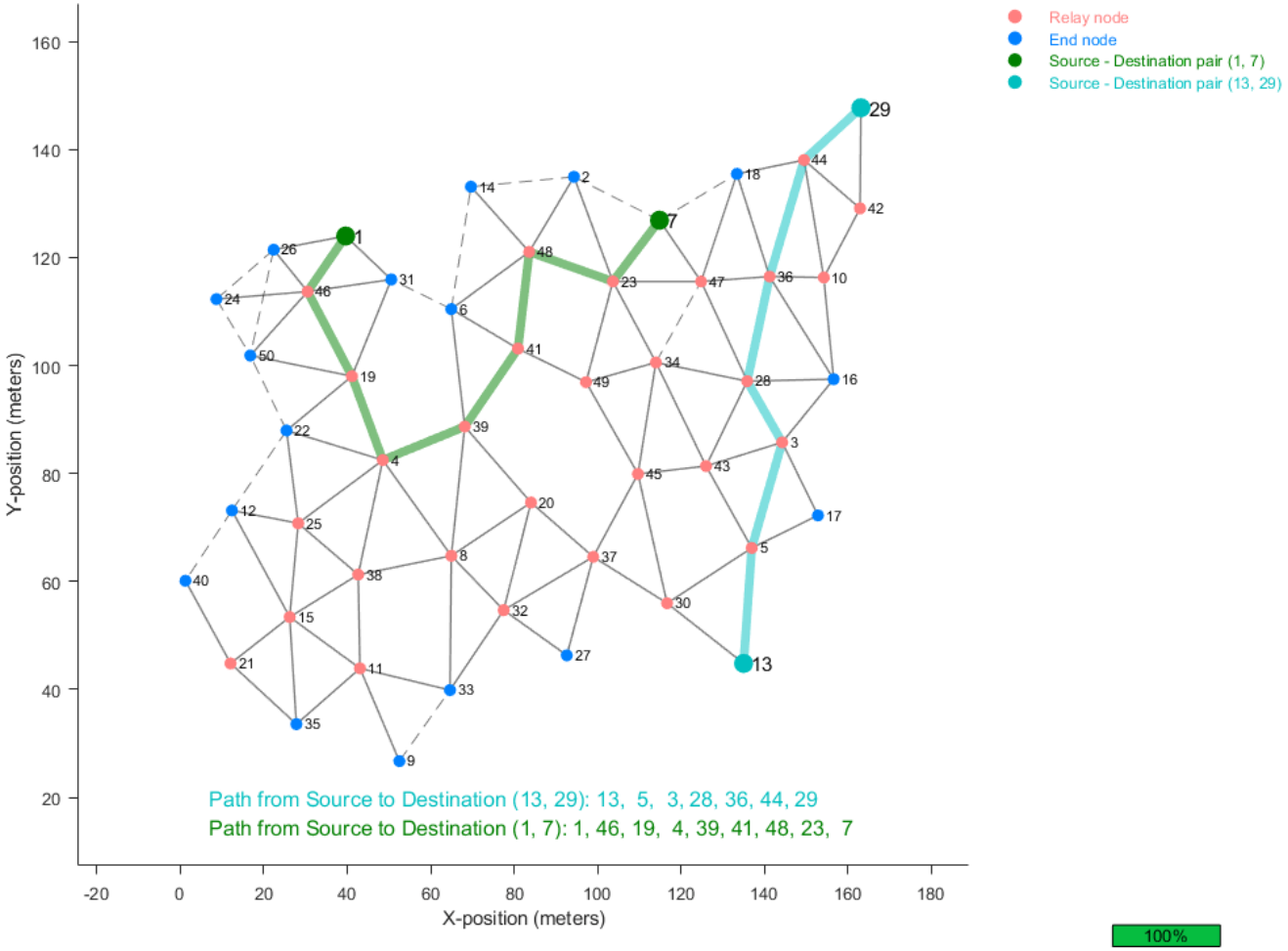
```
% Specify the relay nodes
relayNodeIDs = [3 4 5 8 10 11 15 19 20 21 23 25 28 30 32 34 36 37 38 39 41 ...
    42 43 44 45 46 47 48 49];

% Specify the failed nodes (nodes that are out of network)
failedNodeIDs = [];
```

This plot shows the corresponding paths between each source and destination pair. The `scenarioOneResults` workspace variable stores the results containing the obtained paths in scenario 1.

```
% Run the simulation with scenario 1 configuration
pathScenarioOne = helperBLEMeshFloodingSimulation(totalNodes, bleMeshNodes, meshNetworkPlots(1),
    simulationTime, srcDstPairs, ttl, relayNodeIDs, failedNodeIDs, ...
    enableVisualization);
```

```
% Display the results of the scenario 1
scenarioOneResults = helperBLEMeshFloodingSimulationResults(srcDstPairs, pathScenarioOne)
```

```
scenarioOneResults =

  2x4 table

    Source    Destination                Path                 NumberOfHops
    ______    ___________    ____________________________    ____________

      1            7         {[1 46 19 4 39 41 48 23 7]}          8
     13           29         {[     13 5 3 28 36 44 29]}          6
```



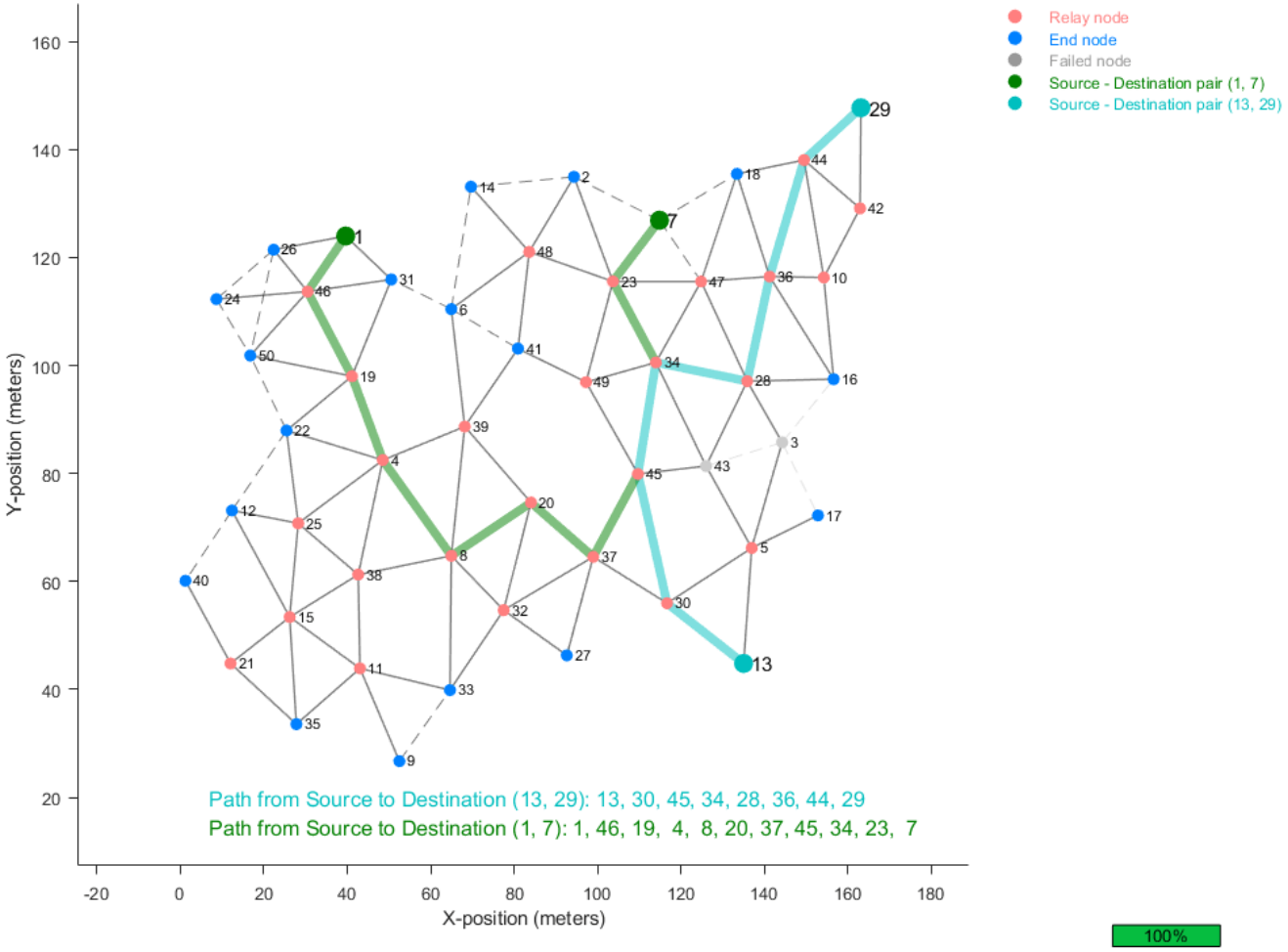
- **Scenario 2:** In this scenario, disable the relay feature of Node 41. Remove Node 3 and Node 43 from the network.

```
% Specify the relay nodes
relayNodeIDs = [4 5 8 10 11 15 19 20 21 23 25 28 30 32 34 36 37 38 39 42 ...
    44 45 46 47 48 49];

% Specify the failed nodes (nodes that are out of network)
failedNodeIDs = [3, 43];
```

This plot shows the corresponding paths between each source and destination pair. The `scenarioTwoResults` workspace variable stores the results containing the obtained paths in scenario 2.

```
% Run the simulation with scenario 2 configuration
pathScenarioTwo = helperBLEMeshFloodingSimulation(totalNodes, bleMeshNodes, meshNetworkPlots(2),
    simulationTime, srcDstPairs, ttl, relayNodeIDs, failedNodeIDs, ...
    enableVisualization);

% Display the results of the scenario 2
scenarioTwoResults = helperBLEMeshFloodingSimulationResults(srcDstPairs, pathScenarioTwo)
```

```
scenarioTwoResults =

  2x4 table

    Source    Destination                  Path                    NumberOfHops
    ______    ___________    ____________________________________    ____________

       1           7         {[1 46 19 4 8 20 37 45 34 23 7]}          10
      13          29         {[     13 30 45 34 28 36 44 29]}           7
```



Network Layer Statistics

At each node, the example captures these network layer statistics.

- Number of messages transmitted by the node
- Number of messages received by the node
- Number of application messages received
- Number of messages relayed by the node
- Number of messages dropped at the node

The `statisticsAtEachNode` workspace variable contains cumulative network statistics of all the nodes in scenario 1 and scenario 2. For a specific simulation run, you can see the network statistics for only first five nodes. These are the network statistics for first five nodes in the network.

```
statisticsAtEachNode = helperBLEMeshFloodingSimulationResults(bleMeshNodes);
statisticsForFirstFiveNodes = statisticsAtEachNode(1:min(5, totalNodes), :)
```

```
statisticsForFirstFiveNodes =

  5x6 table

    NodeID    TotalTxMsgs    TotalRxMsgs    TotalAppRxMsgs    TotalRelayedMsgs    TotalDroppedMsg
    ______    ___________    ___________    ______________    ________________    _______________

      1            2              4               0                 0                  4
      2            0              8               0                 0                  8
      3            0              5               0                 2                  3
      4            0             10               0                 4                  6
      5            0              9               0                 4                  5
```

Further Exploration

To obtain numerical results averaged over multiple simulations, the example implements the Monte Carlo method [3]. To analyze the probability of message delivery from the source node to the destination node after enabling or disabling the relay nodes in the mesh network, use helperBLEMeshMonteCarloSimulations script. Each simulation run follows these steps.

1. Uses a new seed to generate a random number.
2. Randomly disables the relay nodes until only one path exist between the source and destination nodes.
3. Stores the path.

The Monte Carlo simulations outputs these statistics.

- Probability of a message delivery from source to destination when relay nodes are randomly disabled in the network
- Average hop count between the source and destination nodes
- Critical relays required to ensure packet delivery from the source to destination

The example performs Monte Carlo simulations by using these configuration parameters.

```
% Source and destination nodes
srcDstPair = [16 12];

% TTL value for the message originated at the above source node
ttl = 25;

% Relay nodes
relayNodeIDs = [21 15 25 11 38 19 46 8 39 20 37 32 30 5 45 49 43 3 28 36 47 ...
    34 23 48 41 44 42 10 4];

% Failed nodes (nodes that are out of network)
failedNodeIDs = [];
```

The example performs 10,000 simulations by using the above configuration. To view the simulation results, see `bleMeshMonteCarloResults.mat` MAT file.

```
load('bleMeshMonteCarloResults.mat');
disp(['Probability of having a path between nodes Node ' num2str(srcDstPair(1)) ...
    ' and Node ' num2str(srcDstPair(2)) ' is ' ...
    num2str(probabilityOfSuccess) '%.']);
```

```
disp(['Average hop count between nodes Node ' num2str(srcDstPair(1)) ' and Node ' ...
    num2str(srcDstPair(2)) ' is ' ...
    num2str(averageHopCount) '.']);
disp(['Critical relay nodes required to derive a path between Node ' num2str(srcDstPair(1)) ...
    ' and Node ' num2str(srcDstPair(2)) ' are [' num2str(criticalRelaysInfo{1:5, 1}') ...
    '].']);
% Restore the previous setting of random number generation
rng(sprev);
```

```
Probability of having a path between nodes Node 16 and Node 12 is 88.6428%.
Average hop count between nodes Node 16 and Node 12 is 8.
Critical relay nodes required to derive a path between Node 16 and Node 12 are [39  37   8  38
```

To perform Monte Carlo simulations for custom configuration parameters, modify and run the helperBLEMeshMonteCarloSimulations script.

This example enables you to create and configure a multinode Bluetooth mesh network and analyze the network layer flooding. To study the flooding behavior, the example considers two simulation scenarios. In the first scenario, the path between source and destination nodes is identified and visualized by selecting some intermediate nodes as relay nodes. In the second scenario, some nodes (Relay and End) are dropped, and the relay feature for some of the relay nodes is disabled. The obtained results show that there exists a path between the source and destination nodes even if nodes (Relay and End) fail randomly in the network.

This example enables you to create your own Bluetooth mesh network and visualize mesh flooding and network statistics. To obtain numerical results averaged over multiple iterations, you can perform Monte Carlo simulations on the Bluetooth mesh network.

Appendix

The example uses these helpers:

- helperBLEMeshNetworkNode: Create an object for Bluetooth mesh node
- helperBLEMeshNetworkLayer: Create an object for Bluetooth mesh network layer functionality
- helperBLEMeshNetworkPDU: Generate Bluetooth mesh network PDU
- helperBLEMeshNetworkPDUDecode: Decode Bluetooth mesh network PDU
- helperBluetoothQueue: Create an object for Bluetooth queue functionality
- helperBLEMeshRetransmissions: Create an object for retransmissions in Bluetooth mesh node
- helperBLEMeshChannelMessage: Receive message from Bluetooth mesh network channel
- helperBLEMeshPath: Derive path between source and destination within Bluetooth mesh network
- helperBLEMeshVicinityNodes: Get vicinity nodes of a given node
- helperBLEMeshGraphCursorCallback: Display the node statistics on mouse hover action
- helperBLEMeshVisualizeNetwork: Create an object for Bluetooth mesh network visualization
- helperBLEMeshFloodingSimulation: Simulate a Bluetooth mesh network
- helperBLEMeshFloodingSimulationResults: Bluetooth mesh network simulation results
- helperBLEMeshMonteCarloSimulations: Bluetooth mesh network Monte Carlo simulations

Selected Bibliography

1 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.0. https://www.bluetooth.com/.

2 Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Profile". Version 1.0. https://www.bluetooth.com/.

3 Metropolis, Nicholas, and S. Ulam. "The Monte Carlo Method." *Journal of the American Statistical Association* 44, no. 247 (September 1949): 335–41. https://doi.org/10.1080/01621459.1949.10483310.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Mesh Networking” on page 13-46
- “Create, Configure, and Visualize BLE Mesh Network” on page 13-93
- “Estimate Packet Delivery Ratio in Bluetooth Mesh Network” on page 3-117
- “Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks” on page 3-136

BLE Blocking, Intermodulation and Carrier to Interference Performance Tests

This example shows how to model Bluetooth® low energy (BLE) RF-PHY receiver tests specific to blocking, intermodulation and carrier to interference (C/I) performance as per the Bluetooth RF-PHY Test Specifications [1 on page 3-0] using Communications Toolbox™ Library for the Bluetooth Protocol.

Background

The Bluetooth RF-PHY Test Specifications [1 on page 3-0] defined by Bluetooth special interest group (SIG) includes RF-PHY tests for both transmitter and receiver. The objectives of these RF-PHY tests are to ensure interoperability between all BLE devices and to verify that a basic level of system performance is guaranteed for all BLE products. Each test case has a specified test procedure and an expected outcome, which must be met by the implementation under test (IUT).

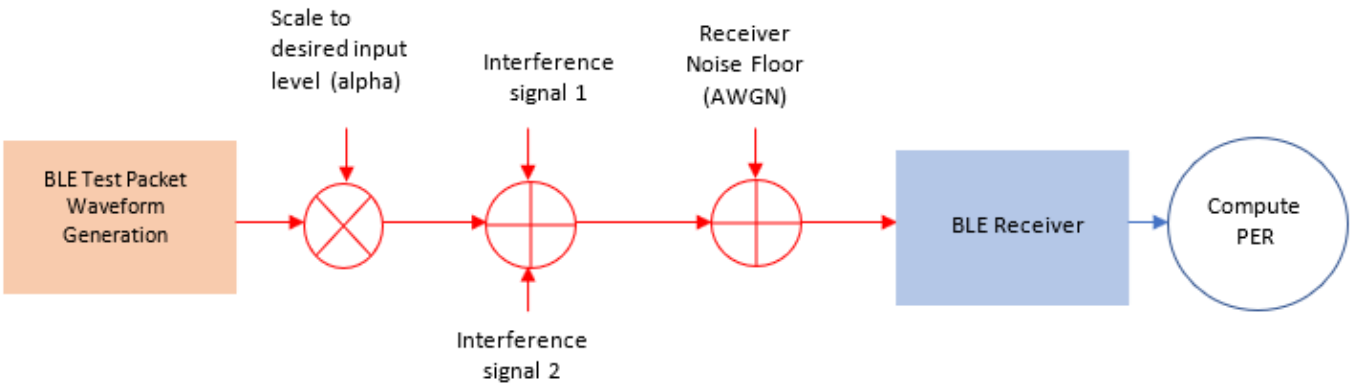
Introduction

The Bluetooth receiver tests are designed to ensure that the IUT can receive data over a range of conditions where the transmitted signal has high power, and in presence of both in-band and out-of-band interference with a defined packet error rate (PER). This example covers three BLE RF-PHY receiver tests for blocking, intermodulation and C/I performance as per the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

- **Blocking Performance**: The blocking performance test verifies the receiver performance in the presence of out-of-band interfering signals i.e. operating outside the 2400 MHz - 2483.5 MHz band.
- **Intermodulation Performance**: The intermodulation performance test verifies the receiver performance in presence of unwanted signals nearby in frequency.
- **C/I Performance**: The C/I performance test verifies the receiver performance in presence of adjacent and co-channel interfering signals.

All the above RF-PHY tests are necessary because the wanted signal often will not be the only signal transmitting in the given frequency range.

The following block diagram summarizes the example flow.



1. Generate test packets and pass through bleWaveformGenerator to generate BLE test waveform.
2. Perform frequency upconversion to obtain a passband signal.
3. Scale the transmitted signal to a desired input level.
4. Add the interference signal(s) depending on the performance test.
5. Add white gaussian noise based on receiver noise floor.
6. At the receiver, down convert the signal and then demodulate, decode and perform CRC check.
7. Measure the PER based on CRC check and then compare it with the reference PER.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Initialize the Simulation Parameters

You can change `rxPerformanceTest`, `phyMode` and `Fc` parameters based on the receiver performance test, PHY transmission mode and frequency of operation, respectively.

```
rxPerformanceTest = 'Intermodulation'; % Select one from the set {'C/I', 'Blocking', 'Interm
% Select PHY transmission mode as per Bluetooth RF-PHY Test Specifications
phyMode = 'LE1M'; % {LE1M, LE2M, LE500K, LE125K} for C/I
                  % {LE1M, LE2M} for blocking and intermodulation
% Select frequency of operation for IUT based on the performance test and
% generic access profile (GAP) role(s) as shown in the table below.
% ------------------------------------------------------------------------------
% Operating | Peripheral & Central Devices    | Broadcaster & Observer Devices   |
% Frequency |                                 |                                  |
% (MHz)     |---------------------------------|----------------------------------|
%           | C/I  | Blocking |Intermodulation| C/I  | Blocking | Intermodulation|
% ----------|------|----------|---------------|------|----------|----------------|
%   Lowest  | 2406 |    -     |    2402       | 2402 |    -     |    2402        |
%   Middle  | 2440 |    2426  |    2440       | 2426 |    2426  |    2426        |
%   Highest | 2476 |    -     |    2480       | 2480 |    -     |    2480        |
% ------------------------------------------------------------------------------
Fc = 2426e6; % Frequency of operation in Hz
payloadLength = 37; % Payload length in bytes, must be in the range [37
sps = 40; % Number of samples per symbol

% Calculate sampling rate in Hz based on PHY transmission mode
Rsym = 1e6;
if strcmp(phyMode,'LE2M')
    Rsym = 2e6;
end
Fs = Rsym*sps;
```

Generate Baseband Waveforms

The function, helperBLETestWaveform.m, can be configured to generate a BLE test packet waveform as per the Bluetooth specifications [2 on page 3-0]. In this example, wanted and interference baseband waveforms can be generated by changing the payload type parameter.

```
% Generate a wanted signal which is always a modulated carrier with a PRBS9
% payload
```

```
payloadTypeWanted = 0; % Payload type for PRBS9 sequence
wantedWaveform =  helperBLETestWaveform(payloadTypeWanted,payloadLength,sps,phyMode);

% Generate an interference signal #1 which is a modulated carrier with a
% PRBS15 payload
payloadTypeInterference = 3; % Payload type for PRBS15 sequence
interferenceWaveform1 =  helperBLETestWaveform(payloadTypeInterference,payloadLength,sps,phyMode
```

Frequency Upconversion

Apply frequency upconversion to obtain a passband signal for the specified frequency of operation.

```
% Interpolation factor for upconversion to cover BLE RF frequency band
% (2400e6 to 2485e6)
interpFactor = ceil(2*2485e6/Fs);

% Create a digital upconverter System object
upConv = dsp.DigitalUpConverter(...
              'InterpolationFactor',interpFactor,...
              'SampleRate',Fs,...
              'Bandwidth',2e6,...
              'StopbandAttenuation',44,...
              'PassbandRipple',0.5,...
              'CenterFrequency',Fc);

% Upconvert the baseband waveform to passband
wantedWaveformUp = upConv([wantedWaveform;zeros(8*sps,1)]);
```

Generate Test Parameters

Test parameters are generated based on performance test, frequency of operation and PHY transmission mode. The function, helperBLETestParamGenerate.m, is used to generate all the interference frequencies and corresponding scaling factors (alpha, beta, gamma) for selected receiver performance test.

```
[alpha,beta,gamma,interferenceFreq1,interferenceFreq2] = ...
               helperBLETestParamGenerate(rxPerformanceTest,Fc,phyMode);
```

Repeat test parameters based on the number of packets used for simulation.

```
pktCnt = 10; % Number of packets
maxInterferenceParams = min(length(interferenceFreq1),pktCnt); % Maximum number of interference p

% Repeat all the interference parameters such that PER can be averaged over
% the entire range of interference frequencies for selected receiver
% performance test.
repFact = ceil(pktCnt/maxInterferenceParams); % Repetition factor
betaRep = repmat(beta,repFact,1);
gammaRep = repmat(gamma,repFact,1);
interferenceFreq1Rep = repmat(interferenceFreq1,repFact,1);
interferenceFreq2Rep = repmat(interferenceFreq2,repFact,1);
```

Test Simulation

In this example, all the three BLE RF-PHY performance tests are simulated as follows:

- For Blocking performance, there will be only one interference signal i.e. interference signal #2. So, the scaling factor (beta) for interference signal #1 is zero.

- For Intermodulation performance, there will be two interference signals.
- For C/I performance, there will be only one interference signal i.e. interference signal #1. So, the scaling factor (gamma) for interference signal #2 is zero.

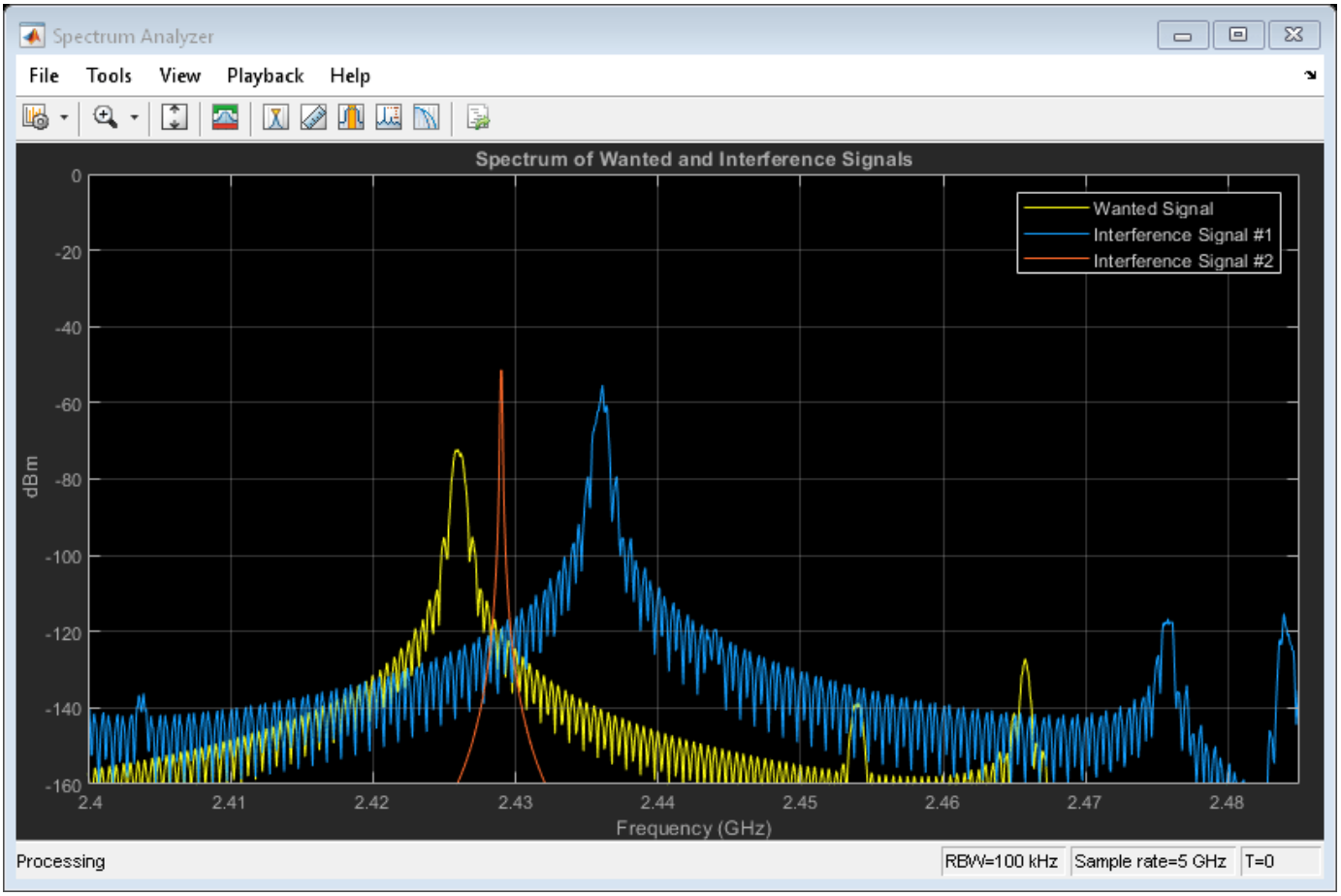
```
% Upconvert and store the interference waveform #1 based on buffer
% size, so that the stored interference waveforms can be reused if
% the packet count exceeds the buffer size.
interferenceWaveform1Up = zeros(length(wantedWaveformUp),maxInterferenceParams);
if any(strcmp(rxPerformanceTest,{'C/I','Intermodulation'}))
    for i=1:maxInterferenceParams
        release(upConv)
        upConv.CenterFrequency = interferenceFreq1Rep(i);
        interferenceWaveform1Up(:,i) = upConv([interferenceWaveform1;zeros(8*sps,1)]);
    end
end

% Initialize a variable for reusing the interference waveform #1
j = rem(1:pktCnt,maxInterferenceParams);
j(j == 0) = maxInterferenceParams;

% Create a digital down converter System object
downConv = dsp.DigitalDownConverter(...
              'DecimationFactor',interpFactor,...
              'SampleRate',Fs*interpFactor,...
              'Bandwidth',2e6,...
              'StopbandAttenuation',44,...
              'PassbandRipple',0.5,...
              'CenterFrequency',Fc);

% Create automatic gain control System object
agc = comm.AGC('DesiredOutputPower',1);

% Create a thermal noise System object
NF = 12; % Noise figure (dB)
thNoise = comm.ThermalNoise('NoiseMethod','Noise figure',...
                            'SampleRate',interpFactor*Fs,...
                            'NoiseFigure',NF);

% Time vector to generate sinusoidal unmodulated interference signal i.e.
% interference signal #2.
t = (0:(length(wantedWaveformUp)-1))'/(interpFactor*Fs);
pktLost = 0; % Initialize counter
for i=1:pktCnt

    % Generate an interference waveform #2 which is a sinusoidal
    % unmodulated signal. The sqrt(2) factor ensures that the power of the
    % sinusoidal signal is normalized.
    interferenceWaveform2 = sqrt(2)*sin(2*pi*interferenceFreq2Rep(i)*t);

    % Add the interference signals to wanted signal
    rxWaveform = alpha*wantedWaveformUp + betaRep(i)*interferenceWaveform1Up(:,j(i)) + gammaRep(i
    chanOut = thNoise(complex(rxWaveform)); % Add thermal noise to the signal
    downConvOut = downConv(real(chanOut)); % Perform frequency down conversion
    agcOut = agc(downConvOut); % Apply AGC
    [payload,accessAddr] = bleIdealReceiver(agcOut,'Mode',phyMode,...
                                'SamplesPerSymbol',sps,'WhitenStatus','Off'); % Extract message i
    [crcFail,pdu] = helperBLETestPacketValidate(payload,accessAddr); % Validate the BLE test pack
```

```
    pktLost = pktLost + crcFail;
end

% Determine the PER
per = pktLost/pktCnt;
```

Spectrum Visualization

Create and configure a spectrum analyzer and show the spectrum of last transmitted wanted signal and interference signal(s) based on the receiver performance test.

```
% Setup spectrum viewer
spectrumScope = dsp.SpectrumAnalyzer( ...
                    'SampleRate',            interpFactor*Fs,...
                    'SpectralAverages',      10,...
                    'YLimits',               [-160 0], ...
                    'Title',                 'Spectrum of Wanted and Interference Signals',...
                    'SpectrumUnits',         'dBm',...
                    'NumInputPorts' ,        2,...
                    'ChannelNames',          {'Wanted Signal','Interference Signal'},...
                    'ShowLegend',            true,...
                    'FrequencySpan',         'Start and stop frequencies',...
                    'StartFrequency',        2400e6,...
                    'StopFrequency',         2485e6,...
                    'RBWSource',             'Property',...
                    'RBW',                   1e5,...
                    'PlotAsTwoSidedSpectrum',false);

if strcmp(rxPerformanceTest,'C/I')
    spectrumScope(alpha*wantedWaveformUp,betaRep(end)*interferenceWaveform1Up(:,end))
elseif strcmp(rxPerformanceTest,'Blocking')
    spectrumScope.StartFrequency = 30e6;
    spectrumScope(alpha*wantedWaveformUp,gammaRep(end)*interferenceWaveform2)
else
    spectrumScope.NumInputPorts = 3;
    spectrumScope.ChannelNames = {'Wanted Signal','Interference Signal #1','Interference Signal #
    spectrumScope(alpha*wantedWaveformUp,betaRep(end)*interferenceWaveform1Up(:,end),gammaRep(end
end
```



Reference Results

This section generates the reference PER values for each PHY transmission mode based on the payload length as specified in section 6.4 of the Bluetooth RF-PHY Test Specifications [1 on page 3-0].

```
berTable = [0.1 0.064 0.034 0.017]*0.01;
if(payloadLength <= 37)
    refBER = berTable(1);
elseif(payloadLength <= 63)
    refBER = berTable(2);
elseif(payloadLength <= 127)
    refBER = berTable(3);
else
    refBER = berTable(4);
end
accessAddLen = 4; % Access address length in bytes
crcLengthBytes = 3; % CRC length in bytes
pduHeaderLen = 2; % Header length in bytes
refPER = 1-(1-refBER)^((payloadLength+accessAddLen+pduHeaderLen+crcLengthBytes)*8);
fprintf('Measured PER and reference PER for payload length of %d bytes are %f, %f respectively.\n
```

```
Measured PER and reference PER for payload length of 37 bytes are 0.000000, 0.308010 respectively
```

```
if per <= refPER
    fprintf('%s performance test passed.\n',rxPerformanceTest);
else
    fprintf('%s performance test failed.\n',rxPerformanceTest);
end
```

```
Intermodulation performance test passed.
```

Appendix

This example uses the following helper functions:

- helperBLETestWaveform.m: Generates BLE test waveform
- helperBLETestParamGenerate.m: Generates BLE test parameters specific to blocking, intermodulation, and C/I
- helperBLETestPacketValidate.m: Validates BLE test packets

Selected Bibliography

1 Bluetooth RF-PHY Test Specification.

2 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

See Also

More About

- “BLE Modulation Characteristics, Carrier Frequency Offset and Drift Test Measurements” on page 3-146
- “BLE Output Power and In-Band Emissions Test Measurements” on page 3-151
- “Bluetooth EDR RF-PHY Transmitter Tests for Modulation Accuracy and Carrier Frequency Stability” on page 3-62
- “Bluetooth BR RF-PHY Transmitter Tests for Modulation Characteristics, Carrier Frequency Offset, and Drift” on page 3-71

BLE Coexistence Model with WLAN Signal Interference

This example shows how to simulate Bluetooth® low energy (BLE) coexistence with WLAN interference using the Communications Toolbox™ Library for the Bluetooth Protocol and the WLAN Toolbox™. Coexistence mechanisms are used to minimize the interference of WLAN on BLE network. In this example, the collision probability and interference level of each WLAN network is used to corrupt the BLE signals. The simulation results generated in this example conclude that for high collision probability and interference level of a WLAN channel, the achieved success rate of the respective BLE channel is low.

BLE-WLAN Coexistence Mechanism

As there are different types of wireless networks operating in the same unlicensed frequency band, it is possible for two different networks to interfere with each other. This interference can cause transmission failures in both the networks. There is no standardized algorithm to achieve coexistence of two different wireless networks. However, the IEEE® 802.15.2™ standard [1] specifies some recommended practices to achieve the coexistence of wireless personal area networks (WPAN) with other wireless devices operating in unlicensed frequency bands.

This example illustrates a BLE coexistence model with WLAN signal interference. WLAN communication requires a minimum of 20 MHz bandwidth, while BLE devices require only 2 MHz bandwidth. WLAN uses a channel access mechanism called carrier-sense multiple access with collision avoidance (CSMA/CA), while BLE devices use frequency hopping. Interference occurs when the operating frequency of BLE and WLAN devices overlap. To minimize the interference, coexistence mechanisms are used.

Coexistence mechanisms are broadly classified into these two categories [1]:

- **Collaborative:** This mechanism requires a communication link between the BLE and WLAN networks. Since these two networks can communicate with each other, one of these networks pauses its transmission while the other is using the channel. This mechanism is used when the WLAN and BLE devices are embedded into the same physical device.
- **Non-Collaborative:** This mechanism does not require any communication link between the BLE and WLAN networks. Since these two networks cannot communicate with each other, they use their own methods to detect the interference of the other network. This mechanism is used when the WLAN and BLE devices are not embedded into the same physical device.

This example illustrates a non-collaborative coexistence mechanism for BLE devices with WLAN.

BLE Coexistence with WLAN - Model Description

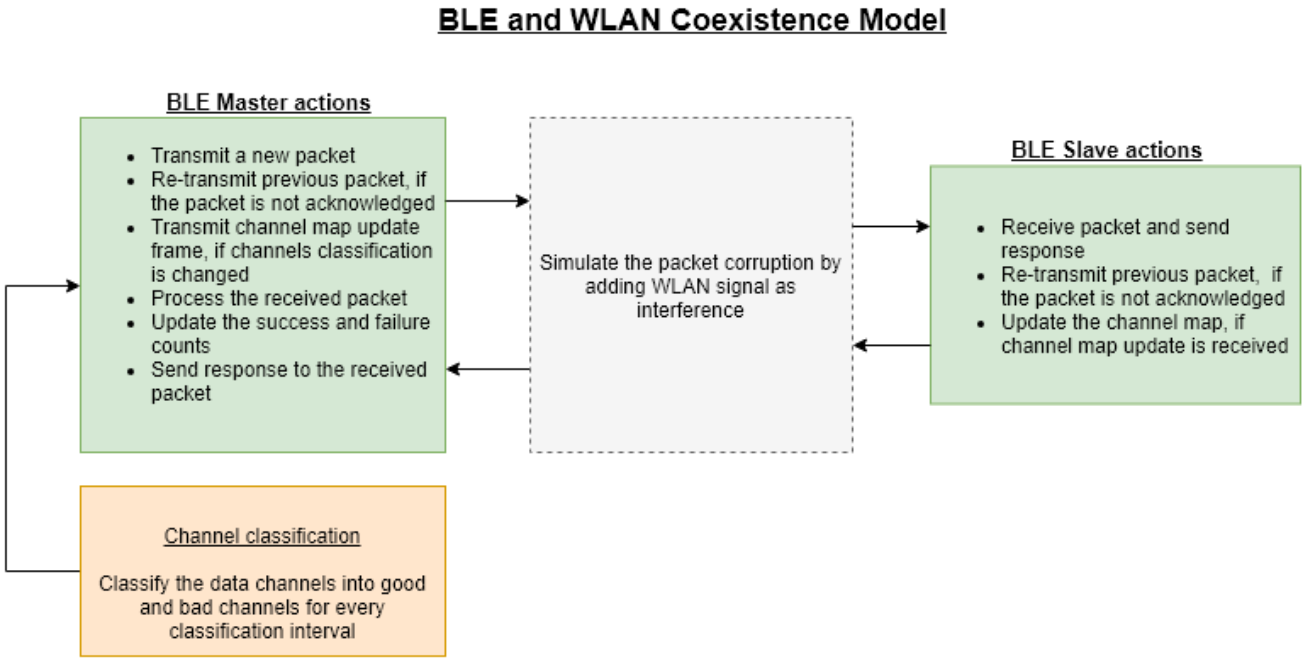
This section elaborates the data communication in BLE, WLAN interference and coexistence algorithm used for avoiding the interference in this example.

Communication in BLE: BLE defines two major roles at the Link Layer, namely the Master and the Slave. Master initiates the data communication and Slave responds to the Master. In this example, BLE packet exchange is modeled between one Master and multiple (configurable up to 5) Slaves. In BLE [2], data communication occurs only during connection events. A connection event is a recurring (at regular intervals called connection interval) sequence of data packets exchange between a Master and a Slave. All the packets within a connection event are transmitted on the same data channel. At the start of every connection event, the Master initiates communication with the respective Slave. Thereafter, the Slave responds to the Master with a data packet. If there is no data to send, the Slave responds with an empty packet. In this example, only one transaction is modeled

per connection event. A new connection event uses a new data channel. The new channel is selected based on adaptive channel hopping. A channel map indicating good or bad channels is used while selecting a new channel, thus showing the adaptiveness in channel hopping.

WLAN traffic: WLAN traffic is dynamically added to, or removed from, the model according to the specified start and end times. Each WLAN network is configured with an individual collision probability. Additionally, WLAN interference level is configured for each WLAN network to corrupt the BLE signals in the respective channel. For every transmission, a random number between 0 and 1 is generated. If the generated random number is less than the collision probability, then the transmitting BLE signal is corrupted by adding the WLAN signals in that channel. The generated WLAN traffic can be modified for IEEE® 802.11ax™ [3] or 802.11n [4] using the wlanTraffic function. However, this example uses only 20 MHz WLAN channels.

BLE coexistence with WLAN: If the selected BLE channel is significantly impacted by the WLAN interference based on collision probability, then the transmitted BLE signal is interfered by WLAN signals in that channel. The Master device periodically classifies the Slave channels as 'good channels' or 'bad channels', based on packet failures in that channel. The channel classification information is stored in the form of a bitmap called channel map. The bitmap is an array of 1's and 0's defining the classification of the channel (either 'good' or 'bad'). The classifyChannels function classifies the BLE channels and stores the generated bitmap. The Master maintains a different channel map for each Slave. The updated channel map is sent to the Slave. The periodicity of channel classification is configured by setting the property `ClassificationInterval` in helperBLEChannelClassification object. BLE devices in idle state, calculate channel busy time for all the 'bad channels' by performing energy detection (ED) of the received signals. If the current number of good channels is less than the preferred number of good channels, the bad channels are classified again. This classification is based on the channel busy time when the `BadChannelClassificationMethod` property is set to '*Using energy indications*'. If the `BadChannelClassificationMethod` property is set to '*Reset all bad channels*', then all the bad channels are reset to good channels.



Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed or not.

```
commSupportPackageCheck('BLUETOOTH');
```

BLE Configuration Parameters

This section adds a BLE Master device and the specified number of Slave devices to the BLE network. Since the Master is responsible for updating channel map for each Slave in a BLE network, the channel classification parameters are configured at the Master device using helperBLEChannelClassification. The helperBLEDeviceModel object is used to model the BLE coexistence with WLAN.

```
% The number of BLE Slaves in connection with the Master
slavesCount = 1;

% Create the BLE Master device capable of connecting with "slavesCount"
% number of Slaves
master = helperBLEDeviceModel('Role','Master', ...
    'SlavesCount',slavesCount);

% Initialize the channel classification parameters to classify the BLE
% channels into good or bad channels. PERThreshold:                 Packet
% error rate (PER) threshold value ClassificationInterval:
% Periodicity of channel classification RxStatusCount:
% Maximum number of received packets status MinRxCountToClassify:
% Minimum number of received packets status BadChannelClassificationMethod:
% Method for bad channels classification PreferredMinimumGoodChannels:
% Preferred number of good channels
```

```
channelClassification = helperBLEChannelClassification(...
    'PERThreshold',40, ...
    'ClassificationInterval',150, ...
    'RxStatusCount',50, ...
    'MinRxCountToClassify',4, ...
    'BadChannelClassificationMethod','Reset all the bad channels', ...
    'PreferredMinimumGoodChannels',30);

% Assign channel classification parameters to the Master device
master.ChannelClassification = channelClassification;

% Initialize "slavesCount" number of Slaves
slaves(1, slavesCount) = helperBLEDeviceModel;

% Create "slavesCount" number of Slave devices
for idx = 1:slavesCount
    slaves(idx) = helperBLEDeviceModel('Role','Slave');
end

% Create "slavesCount" connections between the "Master" and "Slaves". This
% function creates a Link Layer connection by sharing the common connection
% parameters such as connection interval, access address for each
% Master-Slave connection pair.
[master, slaves] = helperBLECreateLLConnection(master, slaves);
```

Model WLAN Traffic

This section models the WLAN traffic using specified configuration.

Configuration Parameters

The configuration parameters for each WLAN network includes collision probability, interference level, interference start time and interference end time in the specified WLAN channel. The helperBLEWLANSignalTrafficConfig object is used to model the WLAN traffic.

```
% Set number of WLAN networks interfering with the BLE network
wlanNetworksCount = 6;

% Set of WLAN channels (in the range [1, 14]) used by each WLAN network
wlanChannels = [1, 5, 6, 12, 9, 8];
% Probability of collisions of each WLAN network with BLE network
collisionProbabilities = [0.75, 0.68, 0.76, 0.80, 0.78, 0.64];
% Start and end times (in milliseconds) of transmission in each WLAN
% network
wlanInterferencePeriod = [0, inf; ...
    0, inf; ...
    0, 2100; ...
    0, inf; ...
    200, 2800; ...
    150, inf];

% Ratio of WLAN signal power level relative to BLE signal power level
wlanInterferenceLevel = [1.20, 0.90, 0.85, 0.95, 0.70, 1.15];
```

Model WLAN Traffic

This section configures the interference to each Slave by adding WLAN traffic with the specified configuration. WLAN traffic (non-HT waveforms) is added in all specified WLAN channels using wlanTraffic function.

```
% Create a configuration object for WLAN traffic
wlanTrafficConfig = helperBLEWLANSignalTrafficConfig();

% Configure WLAN traffic with the specified WLAN network parameters
wlanTraffic(wlanTrafficConfig, wlanNetworksCount, wlanChannels, collisionProbabilities, ...
    wlanInterferencePeriod, wlanInterferenceLevel);
```

Coexistence Simulation

This section illustrates the communication between Master and Slave devices while interfering with WLAN signals.

Initialize Simulation Parameters

The simulation parameters required for the BLE coexistence with WLAN signal interference are initialized in this code.

```
% Initialize simulation parameters

% Reset the random number generator seed
sprev = rng('default');

% To enable the visualization of BLE coexistence with WLAN, set the
% "enableVisualization" to true. To disable the visualization of BLE
% coexistence with WLAN set the "enableVisualization" to false.
enableVisualization = true;

% To enable the visualization of channel hopping sequence, set the
% "enableHoppingVisualization" to true. To disable the visualization of
% channel hopping sequence, set the "enableHoppingVisualization" to false.
% If the "enableVisualization" is set to false, then
% "enableHoppingVisualization" is not considered.
enableHoppingVisualization = true;

% Total simulation time in milliseconds
simulationTime = 4000;

% One step time is considered as 0.025 milliseconds. All the timing
% parameters (connection interval, scan interval, advertising interval,
% etc.) in BLE specification are multiple of 0.625 milliseconds. The
% minimum packet size used in this example is 9 octets (72 bits). The
% packet transmission time in different PHY modes are: 0.072 milliseconds
% (in LE1M), 0.036 milliseconds (in LE2M), 0.144 milliseconds (in LE500K)
% and 0.288 milliseconds (in LE125K). Therefore, the step time is
% considered as 0.025 milliseconds (0.625 is multiple of 0.025) to achieve
% a trade-off between the simulation time and accuracy.
timeStep = 0.025;

% Parameters for generating BLE transmission mode
phyMode = 'LE1M';   % Mode can be 'LE2M' | 'LE1M' | 'LE500K' | 'LE125K'
EbNo = 16;          % Eb/No value in dB

% Initialize PHY parameters sps:              Samples per symbol bleSNR:
% BLE signal to noise ratio initImpairments:  System object for BLE PHY
```

```
% impairments
[sps, bleSNR, initImpairments] = helperBLEInitPHYParameters(EbNo, phyMode);

% Create structure for an empty packet to initialize the output of Master
% and Slaves LLPDU:        Generated Link Layer Protocol Data Unit (PDU)
% appended with
%               Cyclic Redundancy Check (CRC)
% RateIndex:    String representing the rate at which the packet will be
%               transmitted. It contains one of 'LE2M' | 'LE1M' | 'LE500K'
%               | 'LE125K'
% AccessAddress:Unique address for each Master-Slave connection pair
% ChannelIndex: Channel on which the packet is transmitted
emptyPacket = struct('LLPDU',[], ...
    'AccessAddress','', ...
    'RateIndex','', ...
    'ChannelIndex',-1);

% Initialize the Slave output
slaveOutput = emptyPacket;

% Preallocate the buffers to store the Slave outputs
slaveOutputs = cell(1, slavesCount);
```

Simulation

This section simulates the exchange of packets between a BLE Master and Slave devices for a specified amount of time.

- **Master (Transmission or Reception)**: In each connection event, BLE Master initiates the communication with the respective Slave by transmitting a BLE waveform generated for the Link Layer packet on a data channel. The WLAN signal interferes with the generated BLE waveform in the respective BLE channel. After transmission, the Master waits for the response from the Slave.
- **Slave (Transmission or Reception)**: In each connection event, BLE Slave receives the interfered waveform from the Master on a data channel. Thereafter, the Slave responds to the Master on the same data channel by transmitting a Link Layer packet after the generating BLE waveform. The generated BLE waveform is interfered by the WLAN signal in the respective BLE channel.

Before adding the WLAN interference, the transmitted BLE signal is passed through the following RF impairments.

- DC offset
- Carrier frequency offset
- Carrier phase offset
- Timing drift

Use the helperBLEImpairments function to configure the RF impairments.

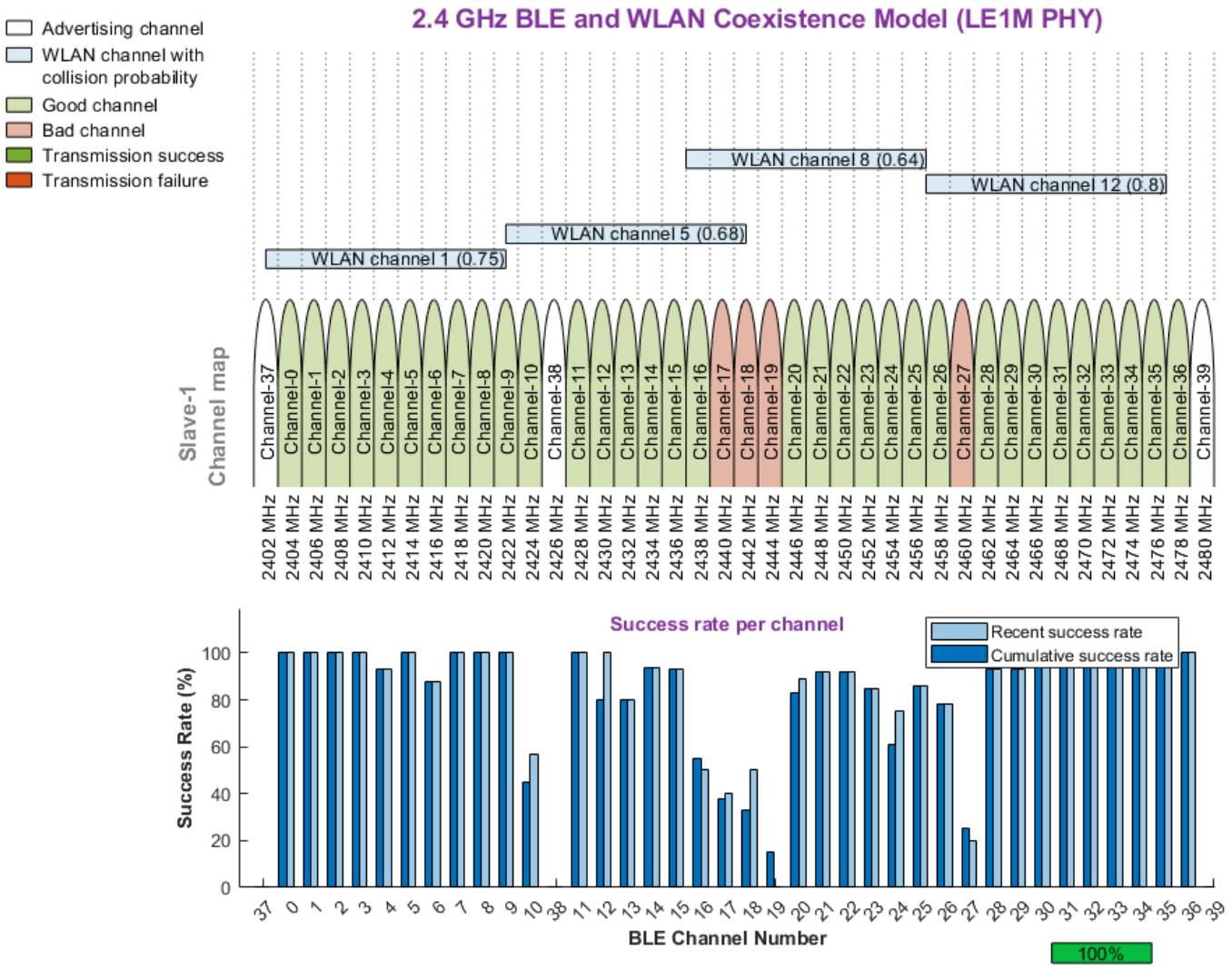
The run function of helperBLEDeviceModel is used for communication between BLE Master and Slave devices. The addInterference function adds the WLAN signals to corrupt the BLE signals. White gaussian noise (WGN) is added to the interfered BLE waveforms. The helperBLEVisualizeCoexistence visualizes the simulation of BLE coexistence with WLAN signals.

```
% Initialize figures for visualization of coexistence model for each Slave.
% This visualization shows the WLAN channels along with their collision
```

```
% probabilities and also shows the channel hopping for the communication
% between BLE Master and Slave devices. It also shows the status (good or
% bad) of each BLE channel along with the success rate in the respective
% channel.
coexistenceModel = ...
    helperBLEVisualizeCoexistence(...
    'Action','Initialize', ...
    'SlaveCount',slavesCount, ...
    'WLANChannelList',wlanChannels, ...
    'PERThreshold',master.ChannelClassification.PERThreshold, ...
    'ClassificationInterval',master.ChannelClassification.ClassificationInterval, ...
    'ChannelBusyCountThreshold',master.ChannelClassification.ChannelBusyCountThreshold, ...
    'PreferredMinimumGoodChannels',master.ChannelClassification.PreferredMinimumGoodChannels, ..
    'ConnectionInterval',master.LLConnectionConfigs(1).ConnectionInterval, ...
    'Stoptime',simulationTime, ...
    'PHYMode',phyMode, ...
    'EnableVisualization',enableVisualization, ...
    'EnableHoppingVisualization',enableHoppingVisualization);
coexistenceModel.initializeVisualization();
viewModel(coexistenceModel);
master.CoexistenceVisualization = coexistenceModel;

% Run simulation
for simulationTimer = 0:timeStep:simulationTime
    % Stop the simulation, if all the Slaves are disconnected from the
    % Master due to interference. If the PER of the BLE channels in which
    % they are communicating with each other is high, then the Master and
    % the Slave are disconnected. The PER of the channel is high because of
    % the high collision probability in the respective channel.
    if numel(master.ActiveConnectionIdxs(master.ActiveConnectionIdxs ~= -1)) == 0
        fprintf('Simulation terminated as all Slaves are disconnected from the Master device.\n'
        break;
    end

    % Update WLAN traffic in visualization
    helperBLEUpdateWLANTraffic(slavesCount, wlanChannels, wlanTrafficConfig, simulationTimer, mas

    % MASTER: Transmitting or Receiving mode
    if (master.ActiveChannel == slaveOutput.ChannelIndex) && ...
            ~isempty(slaveOutput.LLPDU)
        masterOutput = run(master, slaveOutput);
    else
        masterOutput = run(master, emptyPacket);
    end

    if ~(isempty(masterOutput.LLPDU))
        % Generate PHY waveform
        masterOutput.RateIndex = phyMode;
        masterWaveformTx = helperBLEPHYTx(masterOutput, sps);

        % Add impairments
        masterWaveformTx = helperBLEImpairments(initImpairments, masterWaveformTx, sps);

        % Add WLAN interference
        masterWaveformTx = addInterference(wlanTrafficConfig, ...
            masterOutput.ChannelIndex, simulationTimer, masterWaveformTx);

        % Pass the transmitted waveform through AWGN channel
```

```
        masterWaveformRx = awgn(masterWaveformTx, bleSNR);

        % Decode PHY waveform after adding impairments and interference
        [decodedMasterPacket, decodedMasterAccessAddress] = helperBLEPHYRx(masterWaveformRx, ...
            phyMode, sps, masterOutput.AccessAddress, masterOutput.ChannelIndex);

        masterOutput.LLPDU = decodedMasterPacket;
        % Access address becomes empty when the BLE PHY receiver fails to
        % detect a valid BLE packet due to high interference level or
        % impairments or noise level.
        if ~isempty(decodedMasterAccessAddress)
            masterOutput.AccessAddress = dec2hex(bi2de(decodedMasterAccessAddress'), 8);
        end
    end

    % Update current simulation time
    master.CoexistenceVisualization.CurrentTime = simulationTimer;
    master.CoexistenceVisualization.Action = 'Simulation Progress';

    % SLAVE: Transmitting or Receiving mode
    for idx = 1:slavesCount
        % Pass the "MasterOutput" to the Slave listening in the same
        % frequency
        if (slaves(idx).ActiveChannel == masterOutput.ChannelIndex) && ...
                ~isempty(masterOutput.LLPDU)
            slaveOutputs{idx} = run(slaves(idx), masterOutput);
            % Pass an empty packet to all other Slaves
        else
            slaveOutputs{idx} = run(slaves(idx), emptyPacket);
        end

        % Update simulation progress for each Slave
        master.CoexistenceVisualization.SlaveNumber = idx;
        viewModel(master.CoexistenceVisualization)
    end

    slaveOutput = emptyPacket;

    % Get the active Slave output (At any time instance only one Slave is
    % active)
    for idx = 1:slavesCount
        if ~isempty(slaveOutputs{idx}.LLPDU)
            slaveOutput = slaveOutputs{idx};
            break
        end
    end

    if ~(isempty(slaveOutput.LLPDU))
        % Generate PHY waveform
        slaveOutput.RateIndex = phyMode;
        slaveWaveformTx = helperBLEPHYTx(slaveOutput, sps);

        % Add BLE impairments
        slaveWaveformTx = helperBLEImpairments(initImpairments, slaveWaveformTx, sps);

        % Add WLAN interference
        slaveWaveformTx = addInterference(wlanTrafficConfig, ...
            slaveOutput.ChannelIndex, simulationTimer, slaveWaveformTx);
```

```
            % Pass the transmitted waveform through AWGN channel
            slaveWaveformRx = awgn(slaveWaveformTx, bleSNR);

            % Decode PHY waveform after adding impairments and interference
            [decodedSlavePacket, decodedSlaveAccessAddress] = helperBLEPHYRx(slaveWaveformRx, ...
                phyMode, sps, slaveOutput.AccessAddress, slaveOutput.ChannelIndex);

            slaveOutput.LLPDU = decodedSlavePacket;
            % Access address becomes empty when the BLE PHY receiver fails to
            % detect a valid BLE packet due to high interference level or
            % impairments or noise level.
            if ~isempty(decodedSlaveAccessAddress)
                slaveOutput.AccessAddress = dec2hex(bi2de(decodedSlaveAccessAddress'), 8);
            end
        end
    end

    % Update the simulation progress for each Slave
    for idx = 1:slavesCount
        master.CoexistenceVisualization.SlaveNumber = idx;
        master.CoexistenceVisualization.Action = 'Simulation Progress';
        viewModel(master.CoexistenceVisualization)
    end

    % Log the statistics of this example to
    % |bleCoexistenceWithWLANSignalStatistics.mat| file
    helperBLELogCoexistenceStats(master, slaves, ...
        'bleCoexistenceWithWLANSignalStatistics.mat');

    % Restore the previous setting of random number generation
    rng(sprev)
```



Simulation Results

The simulation of this example generates:

1. A run-time plot for each Master-Slave connection pair depicting the status (good or bad) and the cumulative, recent success rates of each channel is displayed
2. A MAT file *bleCoexistenceWithWLANSignalStatistics.mat* with detailed statistics such as number of packets received, number of packets corrupted on each channel and status (good or bad) of the channel for each classification interval is obtained

Further Exploration

You can further explore this example by:

- Using other variants of WLAN formats such as non-HT direct-sequence spread spectrum (DSSS) or high throughput (HT) in the wlanTraffic function

- Corrupting the BLE signal in the addInterference function by varying the interference present at different stages of the BLE signal

You can also explore “Statistical Modeling of WLAN Interference on BLE Network” on page 3-198.

This example enables you to analyze the BLE coexistence with WLAN signal interference. Collision probability and interference level of each WLAN network is used to corrupt the BLE signals. The BLE Master and Slave devices use good channels to communicate with each other to avoid packet loss. The success rate is calculated at each BLE channel. This example concludes that for high collision probability and interference level of a WLAN channel, the achieved success rate of the respective BLE channel is low. Therefore, these channels are not used for communication between BLE Master and Slave devices.

Appendix

The example uses these features:

- `bleChannelSelection`: Select a BLE channel index
- `bleLLDataChannelPDUConfig`: Create a configuration object for BLE Link Layer data channel PDU
- `bleLLDataChannelPDU`: Generate BLE Link Layer data channel PDU
- `bleLLDataChannelPDUDecode`: Decode BLE Link Layer data channel PDU

The example uses these helpers:

- helperBLEChannelClassification: Create an object BLE channel classification
- helperBLEWLANSignalTrafficConfig: Create a configuration object for WLAN signal traffic
- helperBLEDeviceModel: Create an object for BLE device
- helperBLELLConnectionEvent: Create an object for BLE Link Layer connection events
- helperBLELLConnectionEventStatus: Enumeration to indicate the status of BLE Link Layer connection events
- helperBLEConnectionStateModel: Create an object for a BLE Link Layer connection
- helperBLECreateLLConnection: Create a connection between BLE Master and BLE Slave devices
- helperBLEUpdateWLANTraffic: Update WLAN traffic in the visualization at the simulation timer
- helperBLEVisualizeCoexistence: Create an object to visualization the coexistence model
- helperBLELogCoexistenceStats: Log the coexistence statistics to MATLAB workspace
- helperBLEInitPHYParameters: Initialize BLE PHY parameters
- helperBLEPHYTx: Generate BLE PHY waveform
- helperBLEImpairments: Add impairments to the BLE waveform
- helperBLEPHYRx: BLE PHY waveform receiver
- helperBLEImpairmentsAddition: Add RF impairments to the BLE waveform
- helperBLEImpairmentsInit: Initialize RF impairment parameters
- helperBLEPracticalReceiver: Demodulate and decode the received signal
- helperBLEReceiverInit: Initialize BLE PHY receiver parameters

Selected Bibliography

1. IEEE® Standard 802.15.2™. "Coexistence of Wireless Personal Area Networks with Other Wireless Devices Operating in Unlicensed Frequency Bands". *IEEE Recommended Practice for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements; IEEE Computer Society*
2. Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.0. https://www.bluetooth.com/
3. IEEE P802.11ax™/D3.1. "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 6: Enhancements for High Efficiency WLAN". *Draft Standard for Information technology - Telecommunications and information exchange between systems Local and metropolitan area networks - Specific requirements; LAN/MAN Standards Committee of the IEEE Computer Society*
4. IEEE Std 802.11™. "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications". *IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements; LAN/MAN Standards Committee of the IEEE Computer Society*

See Also

More About

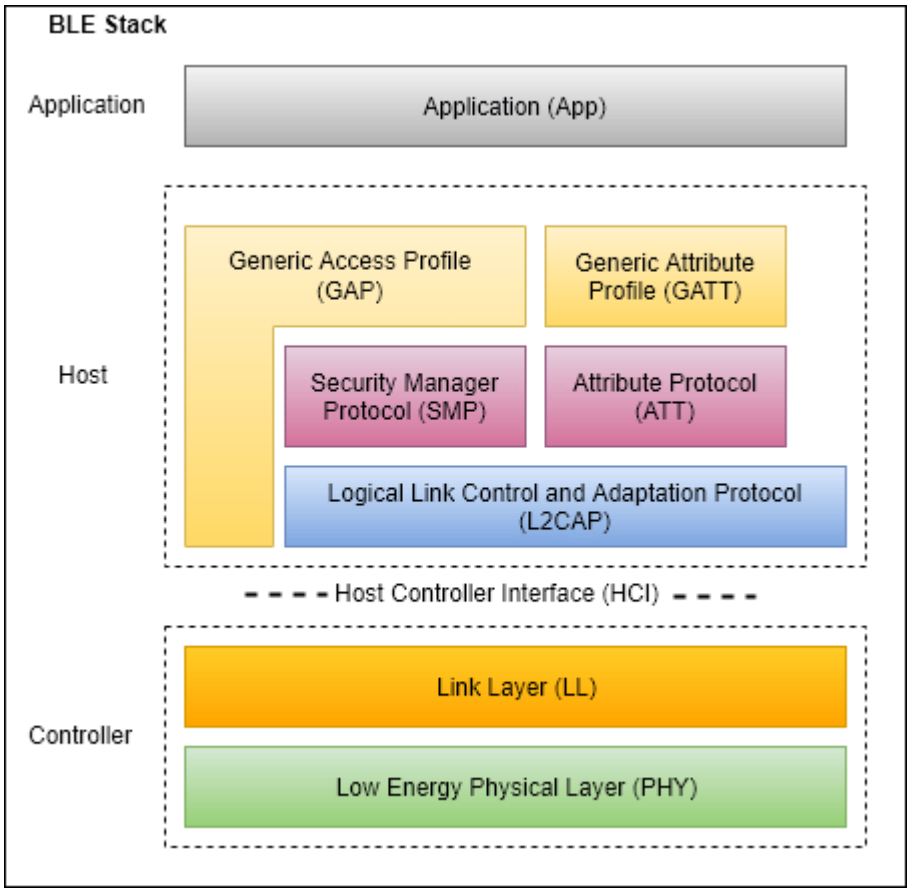
- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96
- “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51
- “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76

Link Layer State Machine for BLE Devices Using Stateflow

This example shows how to model link layer state machine to establish a connection between Bluetooth® low energy (BLE) devices using the Communications Toolbox™ Library for the Bluetooth Protocol and Stateflow®. The working mechanism of the link layer is described in terms of a state machine consisting of these five states: Standby, Advertising, Scanning, Initiating, and Connection. Based on these states, the Bluetooth devices can either be Advertisers, Scanners, Initiators, Master or Slave. This example presents a model to demonstrate the connection establishment process between a Master and a Slave through transitions between different states of the link layer state machine. The simulation results display the time taken to establish a connection between a Master and a Slave. Moreover, this example also provides plots showing the variation in connection establishment time as a function of configuration parameters such as advertising interval and interference.

BLE Stack

The Bluetooth core specification [1] includes a low energy (LE) version for low-rate wireless personal area networks, referred as BLE or Bluetooth Smart. The BLE stack consists of: Generic Attribute Profile (GATT), Attribute Protocol (ATT), Security Manager Protocol (SMP), Logical Link Control and Adaptation Protocol (L2CAP), Link Layer (LL) and Physical layer (PHY). BLE was added to the Bluetooth Special Interest Group (SIG) [1] standard for low energy devices generating small amount of data such as notification alerts used in applications like home automation, health-care, fitness, and Internet of Things (IoT).



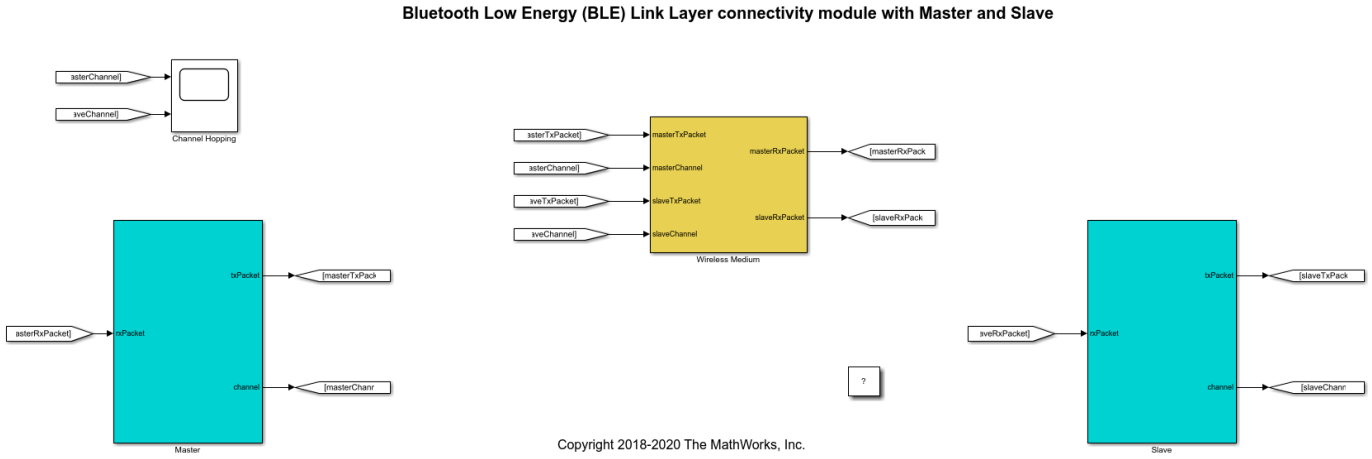
BLE operates in the 2.4 GHz ISM band at 2400 - 2483.5 MHz. It uses 40 RF channels and each channel is 2 MHz wide.

BLE classifies these 40 RF channels into three advertising channels (channel indices: 37, 38, 39) and 37 data channels (channel indices: 0 - 36). Advertising channels are mainly used for creating connection between the BLE devices by transmitting advertising packets, scan request/response packets and connection indication packets. Data channels are mainly used after connection establishment for exchanging data packets.

Check for Support Package Installation

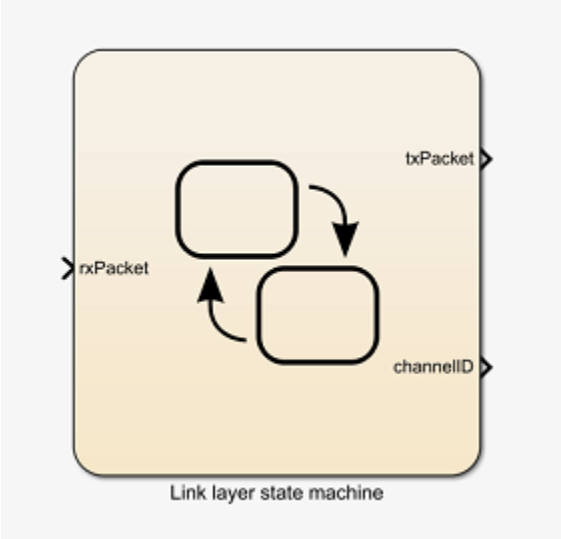
BLE Link Layer State Machine Model

The model in this example demonstrates the connection establishment process between two BLE devices, where one device acts as a Master and the other acts as a Slave. The Master device starts scanning for the advertising packets on the advertising channels. The Slave device starts sending advertising packets on advertising channels. After selecting an advertiser, the Master initiates a connection with it using the 'Connection indication' packet. The connection is established when the Slave receives this packet. Thereafter, the Master and Slave devices can exchange the data packets.



Link Layer States

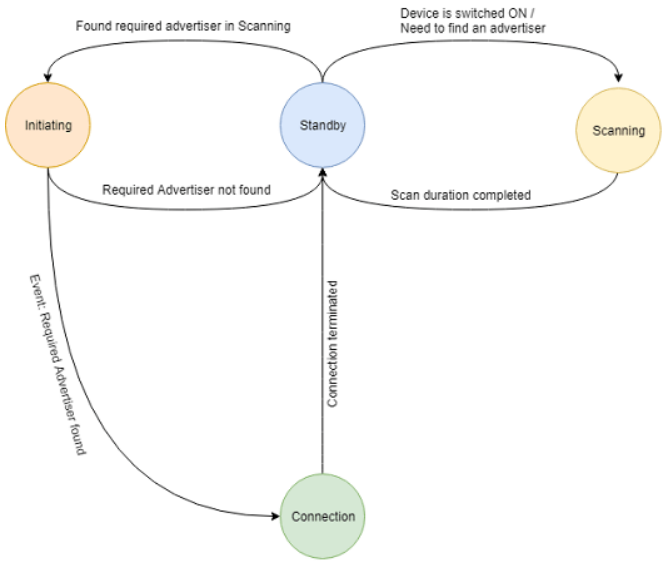
The link layer state machine is implemented using Stateflow® as shown below.



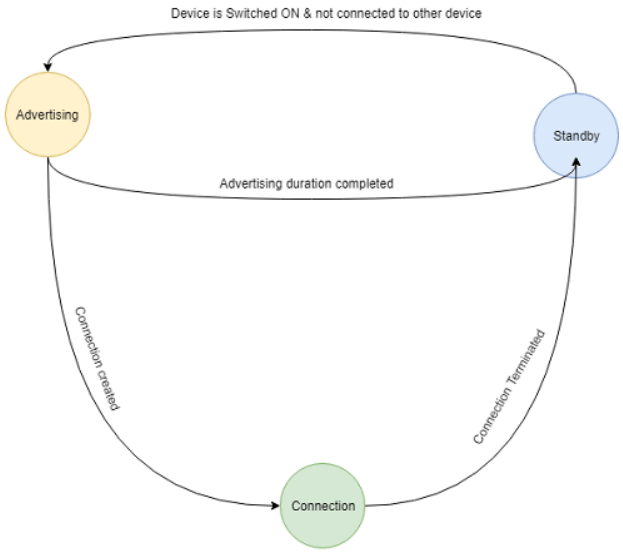
The link layer maintains a state machine consisting of these five states:

1. **Standby** : This is the default state in the link layer. The device does not send or receive packets in this state. It may leave the Standby state to enter one of the Advertising, Scanning or Initiating states.
2. **Advertising** : In this state, the device transmits advertising packets in periodic intervals called advertising events. These devices are called advertisers.
3. **Scanning** : In this state, the device listens for advertisers on the advertising channels and transits to the Standby state for selecting an advertiser to initiate a connection.
4. **Initiating** : In this state, the device listens for a specific advertiser and initiates a connection with it. The device will move to Connection state after receiving the advertising packet from the selected advertiser. These devices are called initiators.
5. **Connection** : In this state, the device transmits data packets in periodic intervals called connection events.

After the connection is established, the initiator becomes the Master (Central) device and the advertiser becomes the Slave (Peripheral) device.



State Machine for Central (Master) Device



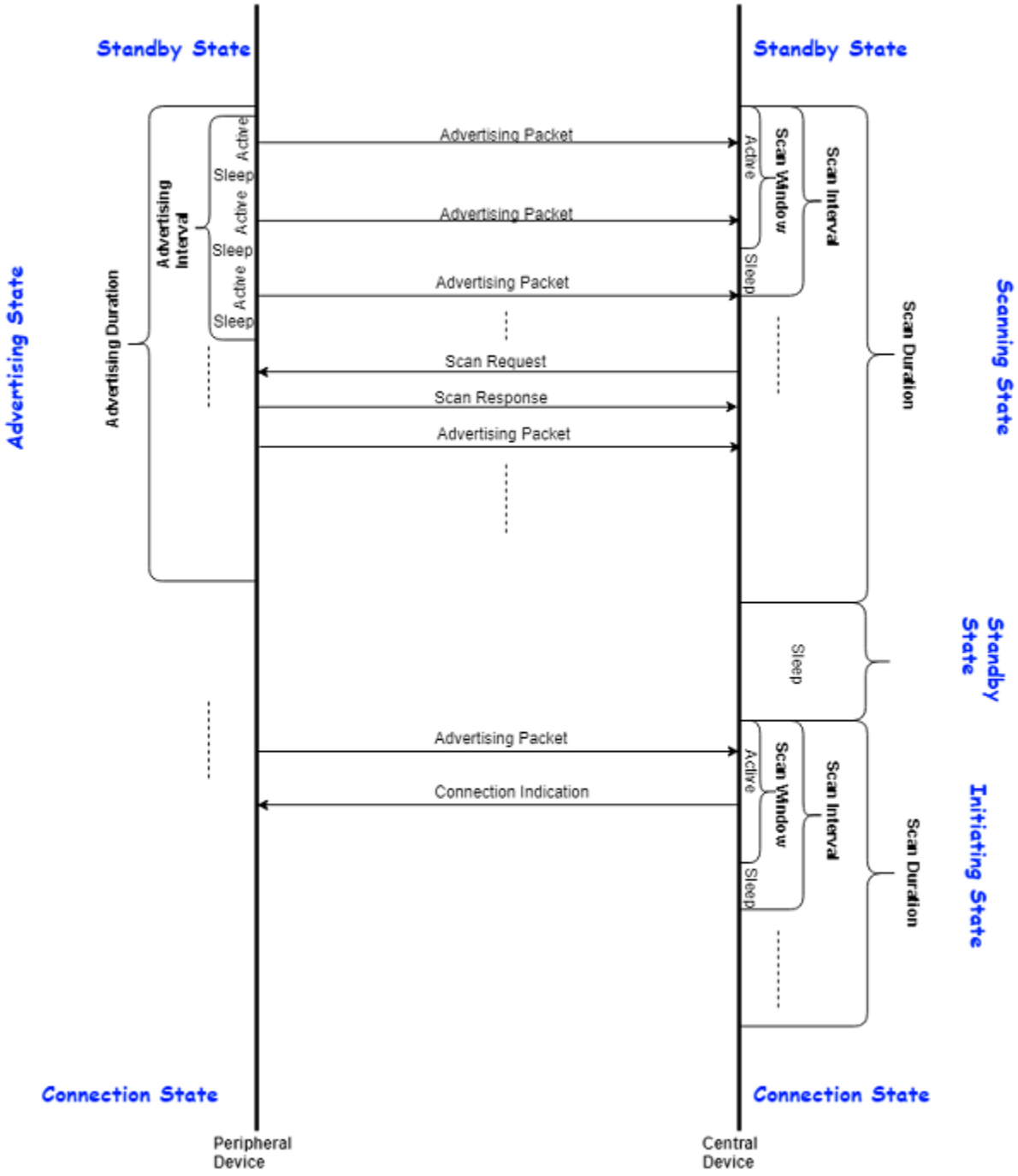
State Machine for Peripheral (Slave) Device

Link Layer Events

An advertising device transmits advertising packets on the three advertising channels in a cyclic manner (starting from channel index 37). Each cycle of advertising is called an advertising event. The same procedure is used by the scanning or the initiating device that listens on the three advertising channels in a cyclic manner called scan event.

A connected device changes to a new data channel for every connection event. A connection event is a time frame for exchanging a sequence of data packets between two connected devices, which repeats at regular intervals. All the packets within a connection event are transmitted on the same data channel. A new connection event uses a new data channel. Two alternate channel selection algorithms are specified by the Bluetooth core specification (refer Section 4.5.8, Part-B, Vol-6 of [1]). These algorithms are used to select the data channel for each connection event.

This figure illustrates the connection establishment process between two BLE devices using link layer events:



Model Configuration

The link layer state machine model demonstrated in this example use these two library blocks: Device and Wireless Medium.

You can update the device configuration through the mask parameters of the Device block in the model. Based on the selected GAP role (Central/ Peripheral), the mask will be updated with relevant properties.

You can also configure the interference statistically for each advertising channel (37, 38 and 39) in the Wireless Medium block.

Simulation Results

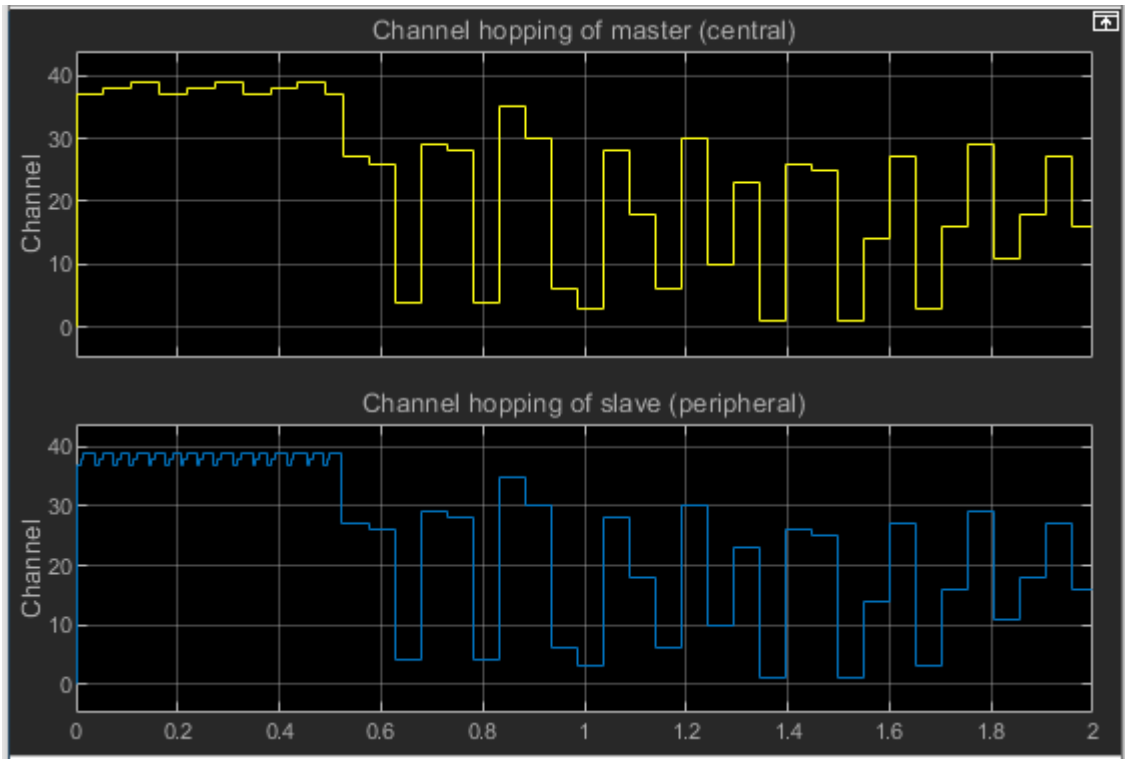
The results obtained in this simulation are:

- **Connection establishment time**:

This model outputs the time taken, in milliseconds, for establishing the connection. This time value is saved to a base workspace variable `timeTakenToEstablishConnection`.

- **Channel hopping sequence**:

The sequence of channel hops used by the Master and Slave devices is separately shown in a scope diagram.



- **Packet exchange between devices**

The packets exchanged between the Master and the Slave are captured in a PCAP format file with the name **BLEPacketExchange.pcap**. This PCAP file can be opened with any third-party packet analyzing software.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
2	0.000001	d3:e6:2c:27:f0:91	eb:13:f6:11:cd:be	LE LL	21	SCAN_REQ
3	0.000002	eb:13:f6:11:cd:be	Broadcast	LE LL	40	SCAN_RSP
4	0.000003	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
5	0.000004	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
6	0.000005	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
7	0.000006	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
8	0.000007	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
9	0.000008	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
10	0.000009	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
11	0.000010	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
12	0.000011	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
13	0.000012	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
14	0.000013	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
15	0.000014	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
16	0.000015	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
17	0.000016	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
18	0.000017	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
19	0.000018	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
20	0.000019	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
21	0.000020	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
22	0.000021	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
23	0.000022	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
24	0.000023	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
25	0.000024	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
26	0.000025	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
27	0.000026	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
28	0.000027	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
29	0.000028	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
30	0.000029	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
31	0.000030	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
32	0.000031	eb:13:f6:11:cd:be	Broadcast	LE LL	18	ADV_IND
33	0.000032	d3:e6:2c:27:f0:91	eb:13:f6:11:cd:be	LE LL	43	CONNECT_REQ
34	0.000033	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
35	0.000034	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
36	0.000035	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
37	0.000036	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
38	0.000037	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
39	0.000038	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU
40	0.000039	Unknown_0xabcd1234	Unknown_0xabcd1234	LE LL	9	Empty PDU

- **Diagnostic log for device operations**

You can enable or disable the message log using the settings of the Device block.

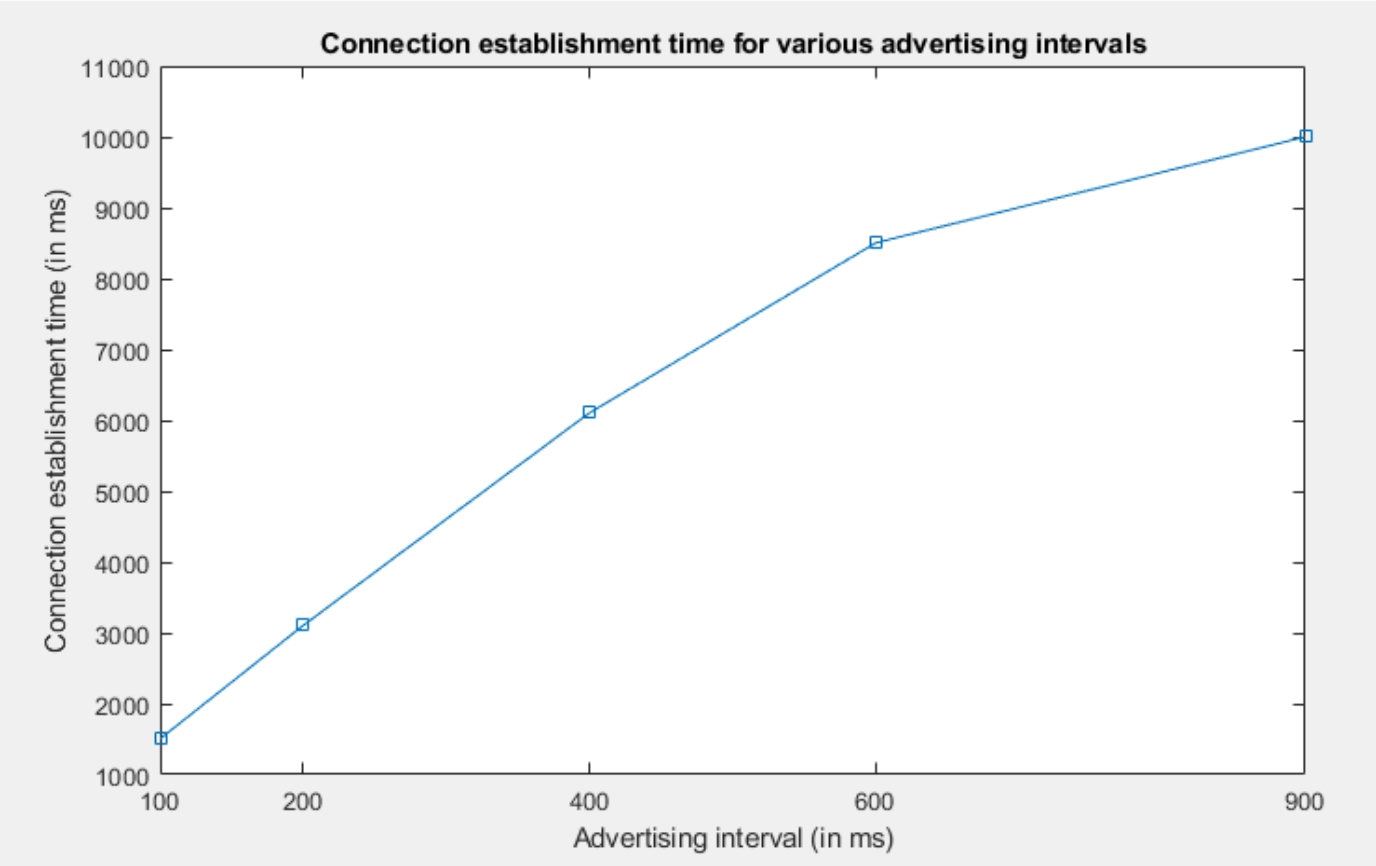
```
Collector :: Central (ABCDEF654321) :: Advertising channel switched to 37
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 37
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 37
Collector :: Central (ABCDEF654321) :: Sent "Scan request" to BLE Device (ABCDEF123459) in "Scanning" State
HeartRateSensor :: Peripheral (ABCDEF123459) :: Received "Scan request" from BLE Device (ABCDEF654321) in "Advertising" State
HeartRateSensor :: Peripheral (ABCDEF123459) :: Sent "Scan response" in "Advertising" State
Collector :: Central (ABCDEF654321) :: Received "Scan response" from BLE Device (ABCDEF123459) in "Scanning" state
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 38
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 38
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 39
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 39
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 37
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 37
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 38
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 38
HeartRateSensor :: Peripheral (ABCDEF123459) :: Advertising channel switched to 39
HeartRateSensor :: Peripheral (ABCDEF123459) :: Send "Advertising indication" on Advertising channel 39
Collector :: Central (ABCDEF654321) :: Advertising channel switched to 38
```

Further Exploration

In the simulation, you can vary the intervals and the interference to observe the variation of the connection establishment time. The simulation results are shown below:

- **Simulation results without interference**:

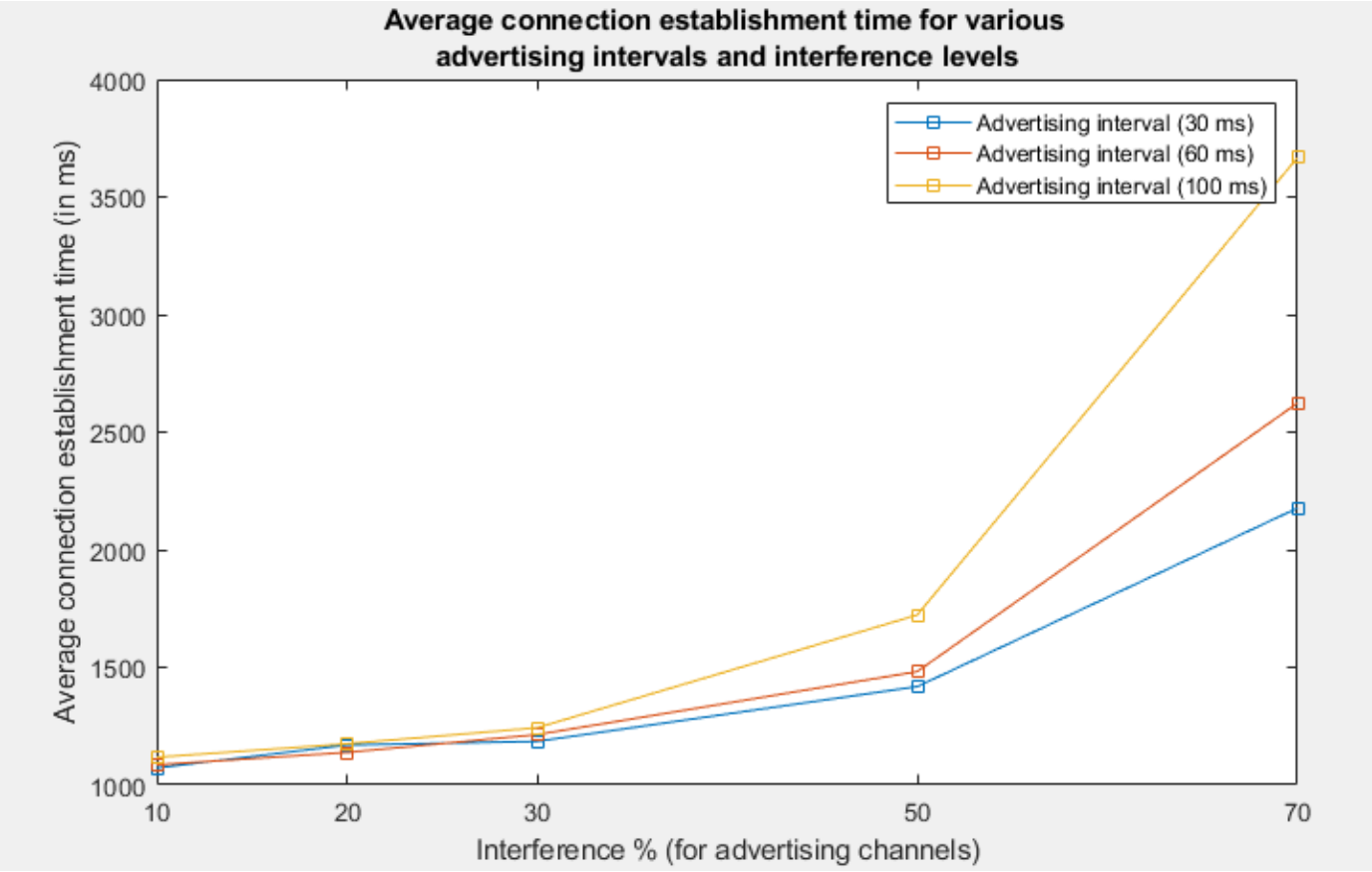
This plot shows how the connection establishment time varies with the advertising interval. Here, scan window is 10 milliseconds with a scan interval of 100 milliseconds i.e. 10% of the active period in scanning.



Run helperBLEPlotConnectionTimeWithoutInterference script (long run simulation) for generating the above results.

- **Simulation results with interference**:

This plot shows how the average connection establishment time varies with different levels of interference. The plot also shows the variation for multiple values of advertising intervals. Here, scan window is 100 milliseconds with a scan interval of 100 milliseconds i.e. 100% of active period in scanning. These connection establishment time values are averaged over 100 simulation runs for different values of advertising intervals and interference levels.



Run helperBLEPlotConnectionTimeWithInterference script (long run simulation) for generating the above results.

Apart from advertising interval, the connection establishment time varies with respect to multiple parameters such as scan interval, scan window, scan duration, advertising duration and interference. You can further explore by varying any of these parameters. Both the above simulations are run with scan and advertising duration of 1 and 10 seconds respectively.

Conclusion

This example illustrates a link layer state machine model to establish a connection between two BLE devices namely: Master and Slave. This model gives detailed information about different link layer states and events. The state transition diagrams presented in this example clearly explain the process of connection establishment between the Master and the Slave. The derived simulation results display the time taken to establish a connection between the BLE devices. Furthermore, the derived plots also show that the connection establishment time varies with the configuration parameters such as advertising interval and interference.

Appendix

The example uses these features:

- `bleLLAdvertisingChannelPDU`: Generate LL advertising channel PDU
- `bleLLAdvertisingChannelPDUDecode`: Decode LL advertising channel PDU

- `bleLLAdvertisingChannelPDUConfig`: Create a configuration object for generation and decoding of LL advertising channel PDU
- `bleLLDataChannelPDU`: Generate LL data channel PDU
- `bleLLDataChannelPDUDecode`: Decode LL data channel PDU
- `bleLLDataChannelPDUConfig`: Create a configuration object for generation and decoding of LL data channel PDU
- `bleLLControlPDUConfig`: Create a sub-configuration object used in generation and decoding of data channel PDU
- `bleGAPDataBlock`: Generate GAP advertising and scan response data
- `bleGAPDataBlockDecode`: Decode GAP advertising and scan response data
- `bleGAPDataBlockConfig`: Create a configuration object for generation and decoding of GAP data block
- `bleChannelSelection`: Create a System object used for selecting a new data channel to transmit the data packet
- `blePCAPWriter`: Create BLE PCAP or PCAPNG file writer object

The example uses these helpers:

- helperBLELLConnectionEvent: Create an object for BLE LL connection events
- helperBLELLConnectionEventStatus: Enumeration to indicate the status of BLE LL connection events
- helperBLELLStateMachineEnumeration: Enumeration to indicate strings used in BLE LL state machine model
- helperBLEGenerateRandomDeviceAddress: Generate random device address for a Bluetooth device
- helperBLEGetDurationInSteps: Calculate the number of steps required for timing parameters in the model based on stepunit
- helperBLEPlotConnectionTimeWithoutInterference: Run multiple simulations varying advertising interval without interference
- helperBLEPlotConnectionTimeWithInterference: Run multiple simulations varying advertising interval with interference

Selected Bibliography

1 Bluetooth® Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed July 8, 2020. https://www.bluetooth.com/.

2 "Development/LibpcapFileFormat - The Wireshark Wiki." Accessed July 8, 2020. https://wiki.wireshark.org/Development/LibpcapFileFormat.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7

Statistical Modeling of WLAN Interference on BLE Network

This example shows how to simulate the statistical modeling of WLAN interference on Bluetooth® low energy (BLE) network using the Communications Toolbox™ Library for the Bluetooth Protocol. Coexistence mechanisms are used to minimize the interference of WLAN on BLE network. In this example, the collision probability of each WLAN network is used to corrupt the BLE signals. The simulation results generated in this example conclude that for high collision probability of a WLAN channel, the achieved success rate of the respective BLE channel is low.

BLE-WLAN Coexistence Mechanism

As there are different types of wireless networks operating in the same unlicensed frequency band, it is possible for two different networks to interfere with each other. This interference can cause transmission failures in both the networks. There is no standardized algorithm to achieve coexistence of two different wireless networks. However, the IEEE® 802.15.2™ standard [1] specifies some recommended practices to achieve the coexistence of wireless personal area networks (WPAN) with other wireless devices operating in unlicensed frequency bands.

This example illustrates a statistical modeling of WLAN interference on BLE network. WLAN communication requires a minimum of 20 MHz bandwidth, while BLE devices require only 2 MHz bandwidth. WLAN uses a channel access mechanism called carrier-sense multiple access with collision avoidance (CSMA/CA), while BLE devices use frequency hopping. Interference occurs when the operating frequency of BLE and WLAN devices overlap. To minimize the interference, coexistence mechanisms are used.

Coexistence mechanisms are broadly classified into these two categories [1]:

- **Collaborative:** This mechanism requires a communication link between the BLE and WLAN networks. Since these two networks can communicate with each other, one of these networks pauses its transmission while the other is using the channel. This mechanism is used when the WLAN and BLE devices are embedded into the same physical device.
- **Non-Collaborative:** This mechanism does not require any communication link between the BLE and WLAN networks. Since these two networks cannot communicate with each other, they use their own methods to detect the interference of the other network. This mechanism is used when the WLAN and BLE devices are not embedded into the same physical device.

This example illustrates a non-collaborative coexistence mechanism for BLE devices with WLAN.

BLE Coexistence with WLAN - Model Description

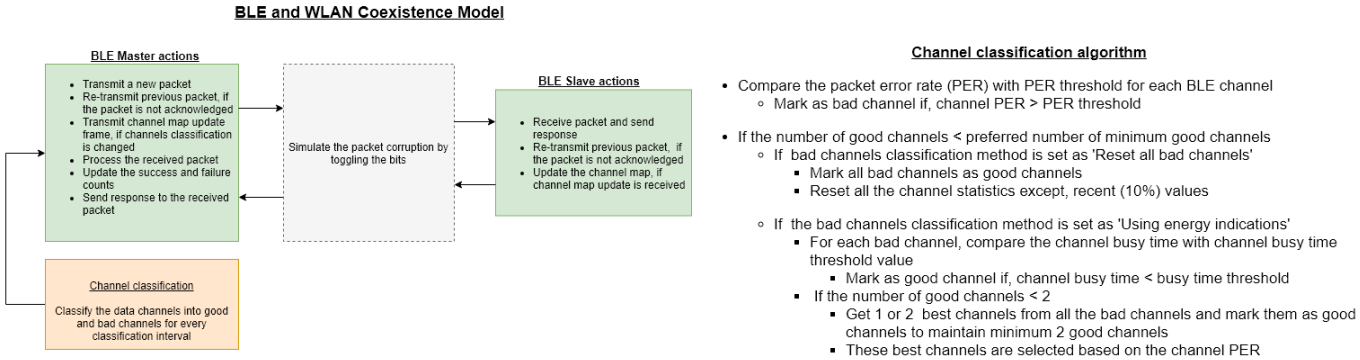
This section elaborates the data communication in BLE, WLAN interference and coexistence algorithm used for avoiding the interference in this example.

Communication in BLE: BLE defines two major roles at the Link Layer, namely the Master and the Slave. Master initiates the data communication and Slave responds to the Master. In this example, BLE packet exchange is modeled between one Master and multiple (configurable up to 5) Slaves. In BLE [2], data communication occurs only during connection events. A connection event is a recurring (at regular intervals called connection interval) sequence of data packets exchange between a Master and a Slave. All the packets within a connection event are transmitted on the same data channel. At the start of every connection event, the Master initiates communication with the respective Slave. Thereafter, the Slave responds to the Master with a data packet. If there is no data to send, the Slave responds with an empty packet. In this example, only one transaction is modeled per connection event. A new connection event uses a new data channel. The new channel is selected

based on adaptive channel hopping. A channel map indicating good or bad channels is used while selecting a new channel, thus showing the adaptiveness in channel hopping.

WLAN traffic: WLAN traffic is dynamically added to, or removed from, the model according to the specified start and end times. Each WLAN network is configured with an individual collision probability. For every transmission, a random number between 0 and 1 is generated. If the generated random number is less than the collision probability, then the transmitting frame is corrupted.

BLE coexistence with WLAN: If the selected BLE channel is significantly impacted by the WLAN interference based on collision probability, then the transmitted BLE packet will undergo corruption. The Master device periodically classifies the Slave channels as 'good channels' or 'bad channels', based on packet failures in that channel. The channel classification information is stored in the form of a bitmap called channel map. The bitmap is an array of 1's and 0's defining the classification of the channel (either 'good' or 'bad'). The classifyChannels function classifies the BLE channels and stores the generated bitmap. The Master maintains a different channel map for each Slave. The updated channel map is sent to the Slave. The periodicity of channel classification is configured by setting the property `ClassificationInterval` in helperBLEChannelClassification object. BLE devices in idle state, calculate channel busy time for all the 'bad channels' by performing energy detection (ED). If the current number of good channels is less than the preferred number of good channels, the bad channels are classified again. This classification is based on the channel busy time when the `BadChannelClassificationMethod` property is set to '*Using energy indications*'. If the `BadChannelClassificationMethod` property is set to '*Reset all bad channels*', all the bad channels are reset to good channels.



Check for Support Package Installation

Check if the 'Communications Toolbox Library for the Bluetooth Protocol' support package is installed or not.

```
commSupportPackageCheck('BLUETOOTH');
```

BLE Configuration Parameters

This section adds a BLE Master device and the specified number of Slave devices to the BLE network. Since the Master is responsible for updating channel map for each Slave in a BLE network, the channel classification parameters are configured at the Master device using helperBLEChannelClassification. The helperBLEDeviceModel object is used to model the BLE coexistence with WLAN.

```
% The number of BLE Slaves in connection with the Master
slavesCount = 1;

% Create the BLE Master device capable of connecting with "slavesCount"
% number of Slaves
master = helperBLEDeviceModel('Role','Master', ...
    'SlavesCount',slavesCount);

% Initialize the channel classification parameters to classify the BLE
% channels into good or bad channels. PERThreshold:                  Packet
% error rate (PER) threshold value ClassificationInterval:
% Periodicity of channel classification RxStatusCount:
% Maximum number of received packets status MinRxCountToClassify:
% Minimum number of received packets status BadChannelClassificationMethod:
% Method for bad channels classification PreferredMinimumGoodChannels:
% Preferred number of good channels
channelClassification = helperBLEChannelClassification(...
    'PERThreshold',60, ...
    'ClassificationInterval',150, ...
    'RxStatusCount',50, ...
    'MinRxCountToClassify',4, ...
    'BadChannelClassificationMethod','Using energy detections', ...
    'PreferredMinimumGoodChannels',20);

% Assign channel classification parameters to the Master device
master.ChannelClassification = channelClassification;

% Initialize "slavesCount" number of Slaves
slaves(1, slavesCount) = helperBLEDeviceModel;

% Create "slavesCount" number of Slave devices
for idx = 1:slavesCount
    slaves(idx) = helperBLEDeviceModel('Role','Slave');
end

% Create "slavesCount" connections between the "Master" and "Slaves". This
% function creates a Link Layer connection by sharing the common connection
% parameters such as connection interval, access address for each
% Master-Slave connection pair.
[master, slaves] = helperBLECreateLLConnection(master, slaves);
```

Model WLAN Traffic

This section models the WLAN traffic using specified configuration.

Configuration Parameters

The configuration parameters for each WLAN network includes collision probability, interference start time and interference end time in the specified WLAN channel. The helperBLEWLANStatisticalTrafficConfig object is used to model the WLAN traffic.

```
% Set number of WLAN networks interfering with the BLE network
wlanNetworksCount = 6;

% Set of WLAN channels (in the range [1, 14]) used by each WLAN network
wlanChannels = [1, 5, 6, 12, 9, 8];
% Probability of collisions of each WLAN network with BLE network
collisionProbabilities = [0.35, 0.48, 0.26, 0.60, 0.28, 0.34];
```

```
% Start and end times (in milliseconds) of transmission in each WLAN
% network
wlanInterferencePeriod = [0, inf; ...
    0, inf; ...
    0, 2100; ...
    0, inf; ...
    200, 2800; ...
    150, inf];
```

Model WLAN Traffic

This section configures the interference to each Slave by adding WLAN traffic with the specified configuration. WLAN network is added in all specified WLAN channels using wlanTraffic function.

```
% Create a configuration object for WLAN traffic
wlanTrafficConfig = helperBLEWLANStatisticalTrafficConfig();

% Configure WLAN traffic with the specified WLAN network parameters
wlanTraffic(wlanTrafficConfig, wlanNetworksCount, wlanChannels, ...
    collisionProbabilities, wlanInterferencePeriod);
```

Coexistence Simulation

This section illustrates the communication between Master and Slave devices while the WLAN is interfering statistically.

Initialize Simulation Parameters

The simulation parameters required for the statistical modeling of WLAN interference on BLE network are initialized in this code.

```
% Initialize simulation parameters

% Reset the random number generator seed
sprev = rng('default');

% To enable the visualization of BLE coexistence with WLAN, set the
% "enableVisualization" to true. To disable the visualization of BLE
% coexistence with WLAN set the "enableVisualization" to false.
enableVisualization = true;

% To enable the visualization of channel hopping sequence, set the
% "enableHoppingVisualization" to true. To disable the visualization of
% channel hopping sequence, set the "enableHoppingVisualization" to false.
% If the "enableVisualization" is set to false, then
% "enableHoppingVisualization" is not considered.
enableHoppingVisualization = true;

% Total simulation time in milliseconds
simulationTime = 4000;

% One step time is considered as 0.025 milliseconds. All the timing
% parameters (connection interval, scan interval, advertising interval,
% etc.) in BLE specification are multiple of 0.625 milliseconds. The
% maximum packet size used in this example is 33 octets (264 bits). The
% packet transmission time in different PHY modes are: 0.264 milliseconds
% (in LE1M), 0.132 milliseconds (in LE2M), 0.528 milliseconds (in LE500K)
% and 1.056 milliseconds (in LE125K). Therefore, the step time is
```

```
% considered as 0.025 milliseconds (0.625 is multiple of 0.025) to achieve
% a trade-off between the simulation time and accuracy.
timeStep = 0.025;

% Create structure for an empty packet to initialize the output of Master
% and Slaves LLPDU:        Generated Link Layer Protocol Data Unit (PDU)
% appended with
%               Cyclic Redundancy Check (CRC)
% RateIndex:    String representing the rate at which the packet will be
%               transmitted. It contains one of 'LE2M' | 'LE1M' | 'LE500K'
%               | 'LE125K'
% AccessAddress:Unique address for each Master-Slave connection pair
% ChannelIndex: Channel on which the packet is transmitted
emptyPacket = struct('LLPDU',[], ...
    'AccessAddress','', ...
    'RateIndex','', ...
    'ChannelIndex',-1);

% Initialize the Slave output
slaveOutput = emptyPacket;

% Preallocate the buffers to store the Slave outputs
slaveOutputs = cell(1, slavesCount);
```

Simulation

This section simulates the exchange of packets between a BLE Master and Slave devices for a specified amount of time.

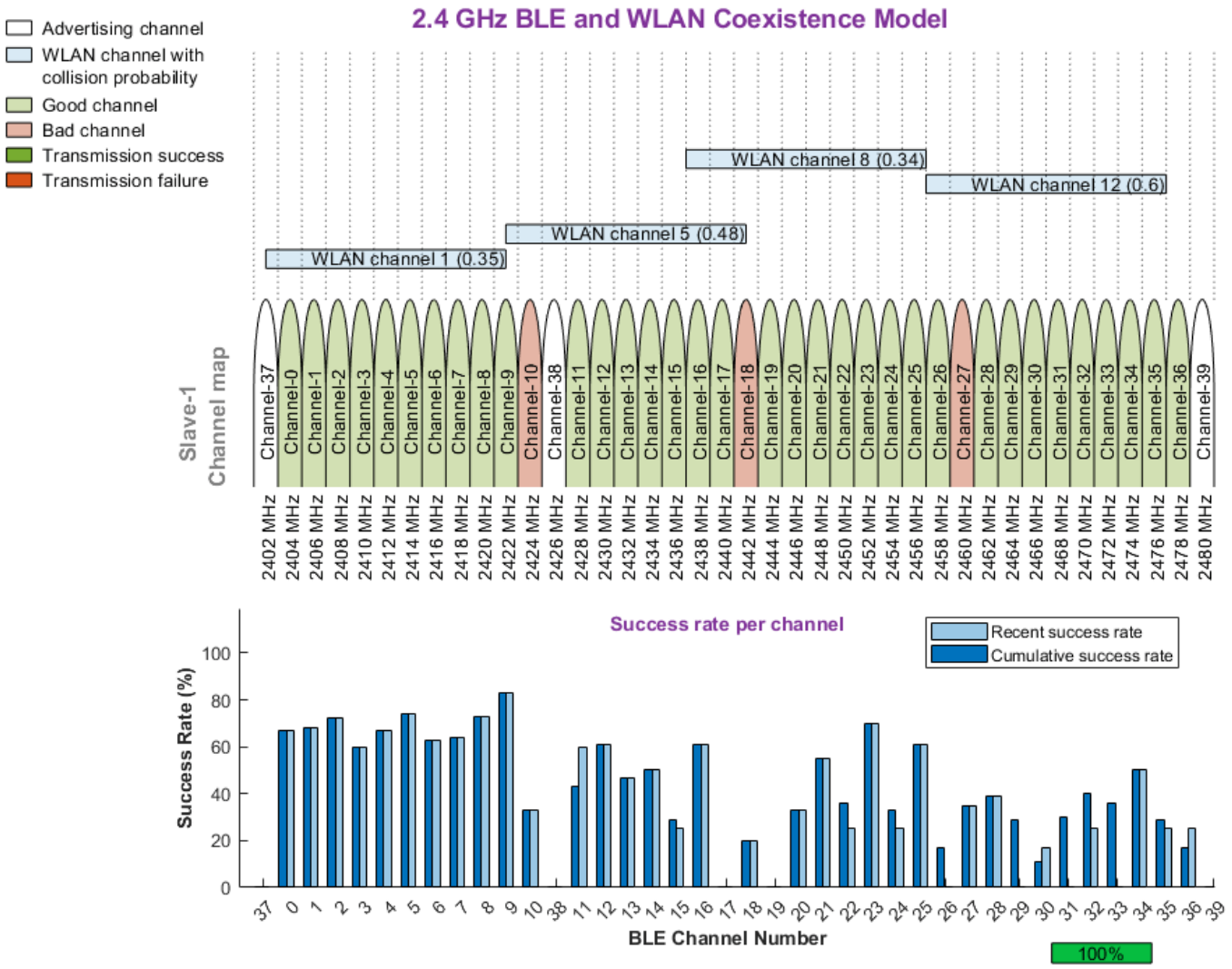
- **Master (Transmission or Reception)**: In each connection event, BLE Master initiates the communication with the respective Slave by transmitting a Link Layer packet on a data channel. The generated BLE packet is corrupted based on the WLAN collision probability of the respective channel. After transmission, the Master waits for the response from the Slave.
- **Slave (Transmission or Reception)**: In each connection event, BLE Slave receives the interfered packet from the Master on a data channel. Thereafter, the Slave responds to the Master on the same data channel by transmitting a Link Layer packet. The generated BLE packet is corrupted based on the WLAN collision probability of the respective channel.

The run function of helperBLEDeviceModel is used for communication between BLE Master and Slave devices. The getInterferenceLevel function verifies whether or not the BLE channel is significantly interfered by the WLAN traffic. The helperBLEVisualizeCoexistence visualizes the simulation of BLE coexistence with WLAN traffic.

```
% Initialize figures for visualization of coexistence model for each Slave.
% This visualization shows the WLAN channels along with their collision
% probabilities and also shows the channel hopping for the communication
% between BLE Master and Slave devices. It also shows the status (good or
% bad) of each BLE channel along with the success rate in the respective
% channel.
coexistenceModel = ...
    helperBLEVisualizeCoexistence(...
    'Action','Initialize', ...
    'SlaveCount',slavesCount, ...
    'WLANChannelList',wlanChannels, ...
    'PERThreshold',master.ChannelClassification.PERThreshold, ...
    'ClassificationInterval',master.ChannelClassification.ClassificationInterval, ...
```

```
        'ChannelBusyCountThreshold',master.ChannelClassification.ChannelBusyCountThreshold, ...
        'PreferredMinimumGoodChannels',master.ChannelClassification.PreferredMinimumGoodChannels, ...
        'ConnectionInterval',master.LLConnectionConfigs(1).ConnectionInterval, ...
        'Stoptime',simulationTime, ...
        'EnableVisualization',enableVisualization, ...
        'EnableHoppingVisualization',enableHoppingVisualization);
    coexistenceModel.initializeVisualization();
    viewModel(coexistenceModel);
    master.CoexistenceVisualization = coexistenceModel;

    % Run simulation
    for simulationTimer = 0:timeStep:simulationTime
        % Stop the simulation, if all the Slaves are disconnected from the
        % Master due to interference. Master and Slave are disconnected when
        % the PER of the BLE channels in which they are communicating with each
        % other is high.
        if numel(master.ActiveConnectionIdxs(master.ActiveConnectionIdxs ~= -1)) == 0
            fprintf('Simulation terminated as all Slaves are disconnected from the Master device.\n')
            break;
        end

        % Update WLAN traffic in visualization
        helperBLEUpdateWLANTraffic(slavesCount, wlanChannels, wlanTrafficConfig, ...
            simulationTimer, master);

        % MASTER: Transmitting or Receiving mode
        if (master.ActiveChannel == slaveOutput.ChannelIndex) && ...
                strcmpi(master.ActiveAccessAddress, slaveOutput.AccessAddress)
            masterOutput = run(master, slaveOutput);
        else
            masterOutput = run(master, emptyPacket);
        end

        if ~(isempty(masterOutput.LLPDU))
            interferenceEffect = getInterferenceLevel(wlanTrafficConfig, ...
                masterOutput.ChannelIndex, simulationTimer);
            % Corrupt the packet, if the interference effect is 1
            if (interferenceEffect == 1)
                masterOutput.LLPDU(1:2) = ~masterOutput.LLPDU(1:2);
                % Drop the packet, if the interference effect is 2
                % (interference is too high)
            elseif (interferenceEffect == 2)
                masterOutput = emptyPacket;
            end
        end

        % Update current simulation time
        coexistenceModel.CurrentTime = simulationTimer;
        coexistenceModel.Action = 'Simulation Progress';

        % SLAVE: Transmitting or Receiving mode
        for idx = 1:slavesCount
            % Pass the "MasterOutput" to the Slave listening in the same
            % frequency and matched access address
            if (slaves(idx).ActiveChannel == masterOutput.ChannelIndex) && ...
                    strcmpi(slaves(idx).ActiveAccessAddress, masterOutput.AccessAddress)
                slaveOutputs{idx} = run(slaves(idx), masterOutput);
                % Pass an empty packet to all other Slaves to update the timers
```

```
            else
                slaveOutputs{idx} = run(slaves(idx), emptyPacket);
            end

            % Update simulation progress for each Slave
            coexistenceModel.SlaveNumber = idx;
            viewModel(coexistenceModel)
        end

        slaveOutput = emptyPacket;

        % Get the active Slave output (At any time instance only one Slave is
        % active)
        for idx = 1:slavesCount
            if ~isempty(slaveOutputs{idx}.LLPDU)
                slaveOutput = slaveOutputs{idx};
                break;
            end
        end

        if ~(isempty(slaveOutput.LLPDU))
            interferenceEffect = getInterferenceLevel(wlanTrafficConfig, ...
                slaveOutput.ChannelIndex, simulationTimer);
            % Corrupt the packet, if the interference effect is 1
            if (interferenceEffect == 1)
                slaveOutput.LLPDU(1:2) = ~slaveOutput.LLPDU(1:2);
                % Drop the packet, if the interference effect is 2
                % (interference is too high)
            elseif (interferenceEffect == 2)
                slaveOutput = emptyPacket;
            end
        end
    end

    % Update the simulation progress for each Slave
    for idx = 1:slavesCount
        master.CoexistenceVisualization.SlaveNumber = idx;
        master.CoexistenceVisualization.Action = 'Simulation Progress';
        viewModel(master.CoexistenceVisualization)
    end

    % Log the statistics of this example to
    % |bleCoexistenceWithStatisticalWLANStatistics.mat| file
    helperBLELogCoexistenceStats(master, slaves, ...
        'bleCoexistenceWithStatisticalWLANStatistics.mat');

    % Restore the previous setting of random number generation
    rng(sprev);
```



Simulation results

The simulation of this example generates:

1. A run-time plot for each Master-Slave connection pair depicting the status (good or bad) and the cumulative, recent success rates of each channel is displayed
2. A MAT file *bleCoexistenceWithStatisticalWLANStatistics.mat* with detailed statistics such as number of packets received, number of packets corrupted on each channel and status (good or bad) of the channel for each classification interval is obtained

This example enables you to analyze the BLE coexistence with WLAN statistical interference. Collision probability of each WLAN network is used to corrupt the BLE packets. The BLE Master and Slave devices use good channels to communicate with each other to avoid packet loss. The success rate is calculated at each BLE channel. This example concludes that for high collision probability of a WLAN channel, the achieved success rate of the respective BLE channel is low. Therefore, these channels are not used for communication between BLE Master and Slave devices.

Appendix

The example uses these features:

- `bleChannelSelection`: Select a BLE channel index
- `bleLLDataChannelPDUConfig`: Create a configuration object for BLE Link Layer data channel PDU
- `bleLLDataChannelPDU`: Generate BLE Link Layer data channel PDU
- `bleLLDataChannelPDUDecode`: Decode BLE Link Layer data channel PDU

The example uses these helpers:

- helperBLEChannelClassification: Create an object BLE channel classification
- helperBLEWLANStatisticalTrafficConfig: Create a configuration object for WLAN signal traffic
- helperBLEDeviceModel: Create an object for BLE device
- helperBLELLConnectionEvent: Create an object for BLE Link Layer connection events
- helperBLELLConnectionEventStatus: Enumeration to indicate the status of BLE Link Layer connection events
- helperBLEConnectionStateModel: Create an object for a BLE Link Layer connection
- helperBLECreateLLConnection: Create a connection between BLE Master and BLE Slave devices
- helperBLEUpdateWLANTraffic: Update WLAN traffic in the visualization at the simulation timer
- helperBLEVisualizeCoexistence: Create an object to visualization the coexistence model
- helperBLELogCoexistenceStats: Log the coexistence statistics to MATLAB workspace

Selected Bibliography

1 IEEE® Standard 802.15.2™. "Coexistence of Wireless Personal Area Networks with Other Wireless Devices Operating in Unlicensed Frequency Bands". *IEEE Recommended Practice for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements; IEEE Computer Society*

2 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.0. https://www.bluetooth.com/

See Also

More About

- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76

Bluetooth Low Energy Transmitter

This example shows how to implement a Bluetooth® Low Energy (BLE) transmitter using the Communications Toolbox™ Library for the Bluetooth Protocol. You can either transmit BLE signals using the ADALM-PLUTO radio or write to a baseband file (*.bb). The transmitted BLE signal can be received by the companion example, “Bluetooth Low Energy Receiver” on page 3-212, with any one of the following setup: (i) Two SDR platforms connected to the same host computer which runs two MATLAB sessions (ii) Two SDR platforms connected to two computers which run separate MATLAB sessions.

Refer to the “Guided Host-Radio Hardware Setup” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) documentation for details on how to configure your host computer to work with the Support Package for ADALM-PLUTO Radio.

Required Hardware and Software

To run this example, you need the following software:

- Communications Toolbox Library for the Bluetooth Protocol

To transmit signals in real time, you also need ADALM-PLUTO radio and the corresponding support package Add-On:

- Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of the Software Defined Radio (SDR) discovery page.

Background

The Bluetooth Special Interest Group (SIG) introduced BLE for low power short range communications. The Bluetooth standard [1] specifies the **Link** layer which includes both **PHY** and **MAC** layers. BLE applications include image and video file transfers between mobile phones, home automation, and the Internet of Things (IoT).

Specifications of BLE:

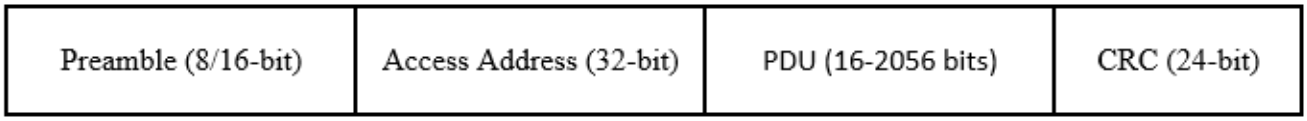
- **Transmission frequency range**: 2.4-2.4835 GHz
- **RF channels** : 40
- **Symbol rate** : 1 Msym/s, 2 Msym/s
- **Modulation** : Gaussian Minimum Shift Keying (GMSK)
- **PHY transmission modes** : (i) LE1M - Uncoded PHY with data rate of 1 Mbps (ii) LE2M - Uncoded PHY with data rate of 2 Mbps (iii) LE500K - Coded PHY with data rate of 500 Kbps (iv) LE125K - Coded PHY with data rate of 125 Kbps

The Bluetooth standard [1] specifies air interface packet formats for all the four PHY transmission modes of BLE using the following fields:

- **Preamble**: The preamble depends on PHY transmission mode. LE1M mode uses an 8-bit sequence of alternate zeros and ones, '01010101'. LE2M uses a 16-bit sequence of alternate zeros and ones, '0101...'. LE500K and LE125K modes use an 80-bit sequence of zeros and ones obtained by repeating '00111100' ten times.

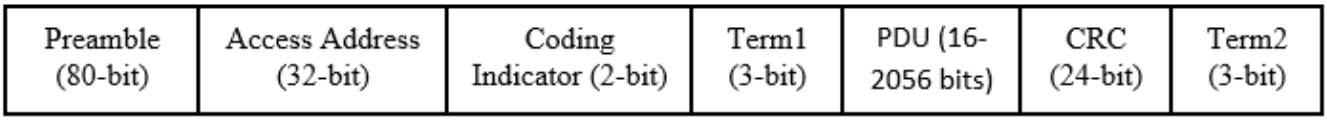
- **Access Address**: Specifies the connection address shared between two BLE devices using a 32-bit sequence.
- **Coding Indicator**: 2-bit sequence used for differentiating coded modes (LE125K and LE500K).
- **Payload**: Input message bits including both protocol data unit (PDU) and cyclic redundancy check (CRC). The maximum message size is 2080 bits.
- **Termination Fields**: Two 3-bit vectors of zeros, used in forward error correction encoding. The termination fields are present for coded modes (LE500K and LE125K) only.

Packet format for uncoded PHY (LE1M and LE2M) modes is shown in the figure below:



BLE Uncoded PHY Packet Format

Packet format for coded PHY (LE500K and LE125K) modes is shown in the figure below:



BLE Coded PHY Packet Format

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Example Structure

The general structure of the BLE transmitter example is described as follows:

1. Generate link layer PDUs
2. Generate baseband IQ waveforms
3. Transmitter processing

Generate Link Layer PDUs

Link layer PDUs can be either advertising channel PDUs or data channel PDUs. You can configure and generate advertising channel PDUs using `bleLLAdvertisingChannelPDUConfig` and `bleLLAdvertisingChannelPDU` functions respectively. You can configure and generate data channel PDUs using `bleLLDataChannelPDUConfig` and `bleLLDataChannelPDU` functions respectively.

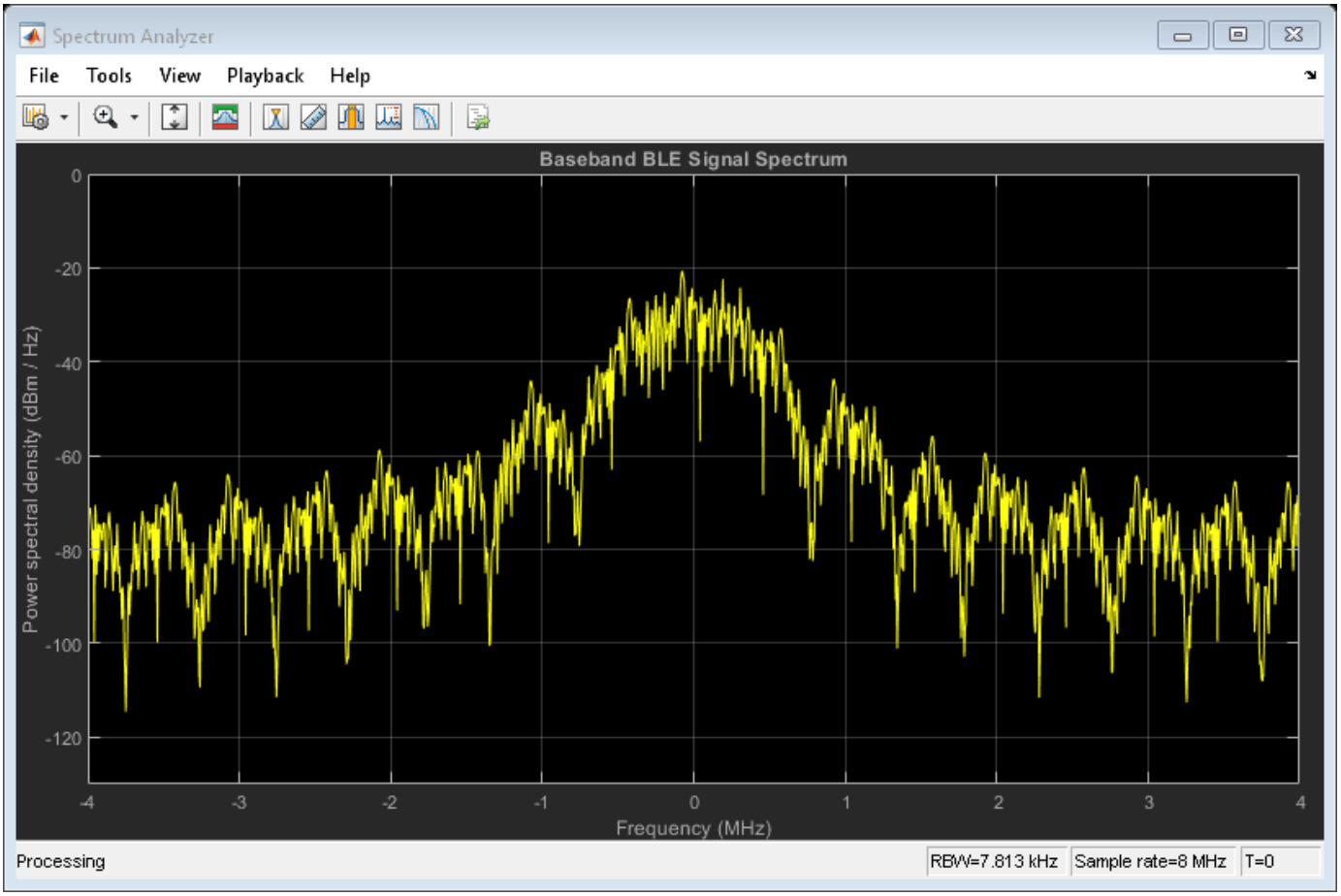
```
% Configure an advertising channel PDU
cfgLLAdv = bleLLAdvertisingChannelPDUConfig;
```

```
cfgLLAdv.PDUType         = 'Advertising indication';
cfgLLAdv.AdvertisingData = '0123456789ABCDEF';
cfgLLAdv.AdvertiserAddress = '1234567890AB';

% Generate an advertising channel PDU
messageBits = bleLLAdvertisingChannelPDU(cfgLLAdv);
```

Generate Baseband IQ Waveforms

You can use the `bleWaveformGenerator` function to generate standard-compliant waveforms.

```
phyMode = 'LE1M'; % Select one mode from the set {'LE1M','LE2M','LE500K','LE125K'}
sps = 8;          % Samples per symbol
channelIdx = 37;  % Channel index value in the range [0,39]
accessAddLen = 32;% Length of access address
accessAddHex = '8E89BED6';  % Access address value in hexadecimal
accessAddBin = de2bi(hex2dec(accessAddHex),accessAddLen)'; % Access address in binary

% Symbol rate based on |'Mode'|
symbolRate = 1e6;
if strcmp(phyMode,'LE2M')
    symbolRate = 2e6;
end

% Generate BLE waveform
txWaveform = bleWaveformGenerator(messageBits,...
    'Mode',            phyMode,...
    'SamplesPerSymbol',sps,...
    'ChannelIndex',    channelIdx,...
    'AccessAddress',   accessAddBin);

% Setup spectrum viewer
spectrumScope = dsp.SpectrumAnalyzer( ...
    'SampleRate',       symbolRate*sps,...
    'SpectrumType',     'Power density', ...
    'SpectralAverages', 10, ...
    'YLimits',          [-130 0], ...
    'Title',            'Baseband BLE Signal Spectrum', ...
    'YLabel',           'Power spectral density');

% Show power spectral density of the BLE signal
spectrumScope(txWaveform);
```



Transmitter Processing

Specify the signal sink as 'File' or 'ADALM-PLUTO'.

- **File**:Uses the `comm.BasebandFileWriter` to write a baseband file.
- **ADALM-PLUTO**: Uses the `sdrtx` (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) System object to transmit a live signal from the SDR hardware.

```
% Initialize the parameters required for signal source
txCenterFrequency       = 2.402e9;  % Varies based on channel index value
txFrameLength           = length(txWaveform);
txNumberOfFrames        = 1e4;
txFrontEndSampleRate    = symbolRate*sps;

% The default signal source is 'File'
signalSink = 'File';

if strcmp(signalSink,'File')

    sigSink = comm.BasebandFileWriter('CenterFrequency',txCenterFrequency,...
        'Filename','bleCaptures.bb',...
        'SampleRate',txFrontEndSampleRate);
    sigSink(txWaveform); % Writing to a baseband file 'bleCaptures.bb'
```

```
elseif strcmp(signalSink,'ADALM-PLUTO')

    % First check if the HSP exists
    if isempty(which('plutoradio.internal.getRootDir'))
        error(message('comm_demos:common:NoSupportPackage', ...
                      'Communications Toolbox Support Package for ADALM-PLUTO Radio',...
                      ['<a href="https://www.mathworks.com/hardware-support/' ...
                      'adalm-pluto-radio.html">ADALM-PLUTO Radio Support From Communications Too
    end
    connectedRadios = findPlutoRadio; % Discover ADALM-PLUTO radio(s) connected to your computer
    radioID = connectedRadios(1).RadioID;
    sigSink = sdrtx( 'Pluto',...
        'RadioID',           radioID,...
        'CenterFrequency',   txCenterFrequency,...
        'Gain',              0,...
        'SamplesPerFrame',   txFrameLength,...
        'BasebandSampleRate',txFrontEndSampleRate);
    % The transfer of baseband data to the SDR hardware is enclosed in a
    % try/catch block. This means that if an error occurs during the
    % transmission, the hardware resources used by the SDR System
    % object(TM) are released.
    currentFrame = 1;
    try
        while currentFrame <= txNumberOfFrames
            % Data transmission
            sigSink(txWaveform);
            % Update the counter
            currentFrame = currentFrame + 1;
        end
    catch ME
        release(sigSink);
        rethrow(ME)
    end
else
    error('Invalid signal sink. Valid entries are File and ADALM-PLUTO.');
end

% Release the signal sink
release(sigSink)
```

Further Exploration

The companion example “Bluetooth Low Energy Receiver” on page 3-212 can be used to decode the waveform transmitted by this example. You can also use this example to transmit the data channel PDUs by changing channel index, access address and center frequency values in both the examples.

Troubleshooting

General tips for troubleshooting SDR hardware and the Communications Toolbox Support Package for ADALM-PLUTO Radio can be found in “Common Problems and Fixes” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Selected Bibliography

1 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

Bluetooth Low Energy Receiver

This example shows how to implement a Bluetooth® Low Energy (BLE) receiver using the Communications Toolbox™ Library for the Bluetooth Protocol. You can either use captured signals or receive signals in real time using the ADALM-PLUTO Radio. A suitable signal for reception can be generated by simulating the companion example, “Bluetooth Low Energy Transmitter” on page 3-207, with any one of the following setup: (i) Two SDR platforms connected to the same host computer which runs two MATLAB sessions (ii) Two SDR platforms connected to two computers which run separate MATLAB sessions.

Refer to the “Guided Host-Radio Hardware Setup” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) documentation for details on how to configure your host computer to work with the Support Package for ADALM-PLUTO Radio.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Communications Toolbox Library for the Bluetooth Protocol

To receive signals in real time, you also need an ADALM-PLUTO radio and the corresponding support package Add-On:

- Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of the Software Defined Radio (SDR) discovery page.

Background

The Bluetooth Special Interest Group (SIG) introduced BLE for low power short range communications. The Bluetooth standard [1] specifies the **Link** layer which includes both **PHY** and **MAC** layers. BLE applications include image and video file transfers between mobile phones, home automation, and the Internet of Things (IoT).

Specifications of BLE:

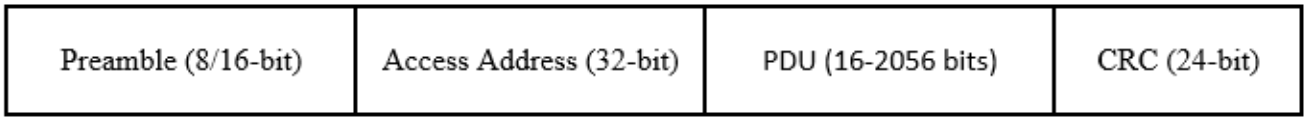
- **Transmission frequency range**: 2.4-2.4835 GHz
- **RF channels** : 40
- **Symbol rate** : 1 Msym/s, 2 Msym/s
- **Modulation** : Gaussian Minimum Shift Keying (GMSK)
- **PHY transmission modes** : (i) LE1M - Uncoded PHY with data rate of 1 Mbps (ii) LE2M - Uncoded PHY with data rate of 2 Mbps (iii) LE500K - Coded PHY with data rate of 500 Kbps (iv) LE125K - Coded PHY with data rate of 125 Kbps

The Bluetooth standard [1] specifies air interface packet formats for all the four PHY transmission modes of BLE using the following fields:

- **Preamble**: The preamble depends on PHY transmission mode. LE1M mode uses an 8-bit sequence of alternate zeros and ones, '01010101'. LE2M uses a 16-bit sequence of alternate zeros and ones, '0101...'. LE500K and LE125K modes use an 80-bit sequence of zeros and ones obtained by repeating '00111100' ten times.

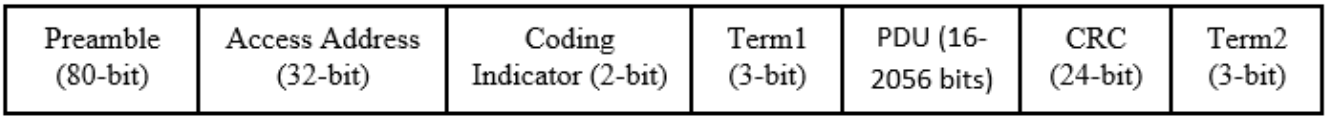
- **Access Address**: Specifies the connection address shared between two BLE devices using a 32-bit sequence.
- **Coding Indicator**: 2-bit sequence used for differentiating coded modes (LE125K and LE500K).
- **Payload**: Input message bits including both protocol data unit (PDU) and cyclic redundancy check (CRC). The maximum message size is 2080 bits.
- **Termination Fields**: Two 3-bit vectors of zeros, used in forward error correction encoding. The termination fields are present for coded modes (LE500K and LE125K) only.

Packet format for uncoded PHY (LE1M and LE2M) modes is shown in the figure below:



BLE Uncoded PHY Packet Format

Packet format for coded PHY (LE500K and LE125K) modes is shown in the figure below:



BLE Coded PHY Packet Format

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Example Structure

The general structure of the BLE receiver example is described as follows:

1. Initialize the receiver parameters
2. Signal source
3. Capture the BLE packets
4. Receiver processing

Initialize the Receiver Parameters

The helperBLEReceiverConfig.m script initializes the receiver parameters. You can change `phyMode` parameter to decode the received BLE waveform based on the PHY transmission mode. `phyMode` can be one from the set: {'LE1M','LE2M','LE500K','LE125K'}.

```
phyMode = 'LE1M';
bleParam = helperBLEReceiverConfig(phyMode);
```

Signal Source

Specify the signal source as 'File' or 'ADALM-PLUTO'.

- **File**:Uses the `comm.BasebandFileReader` to read a file that contains a previously captured over-the-air signal.
- **ADALM-PLUTO**: Uses the `sdrrx` (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio) System object to receive a live signal from the SDR hardware.

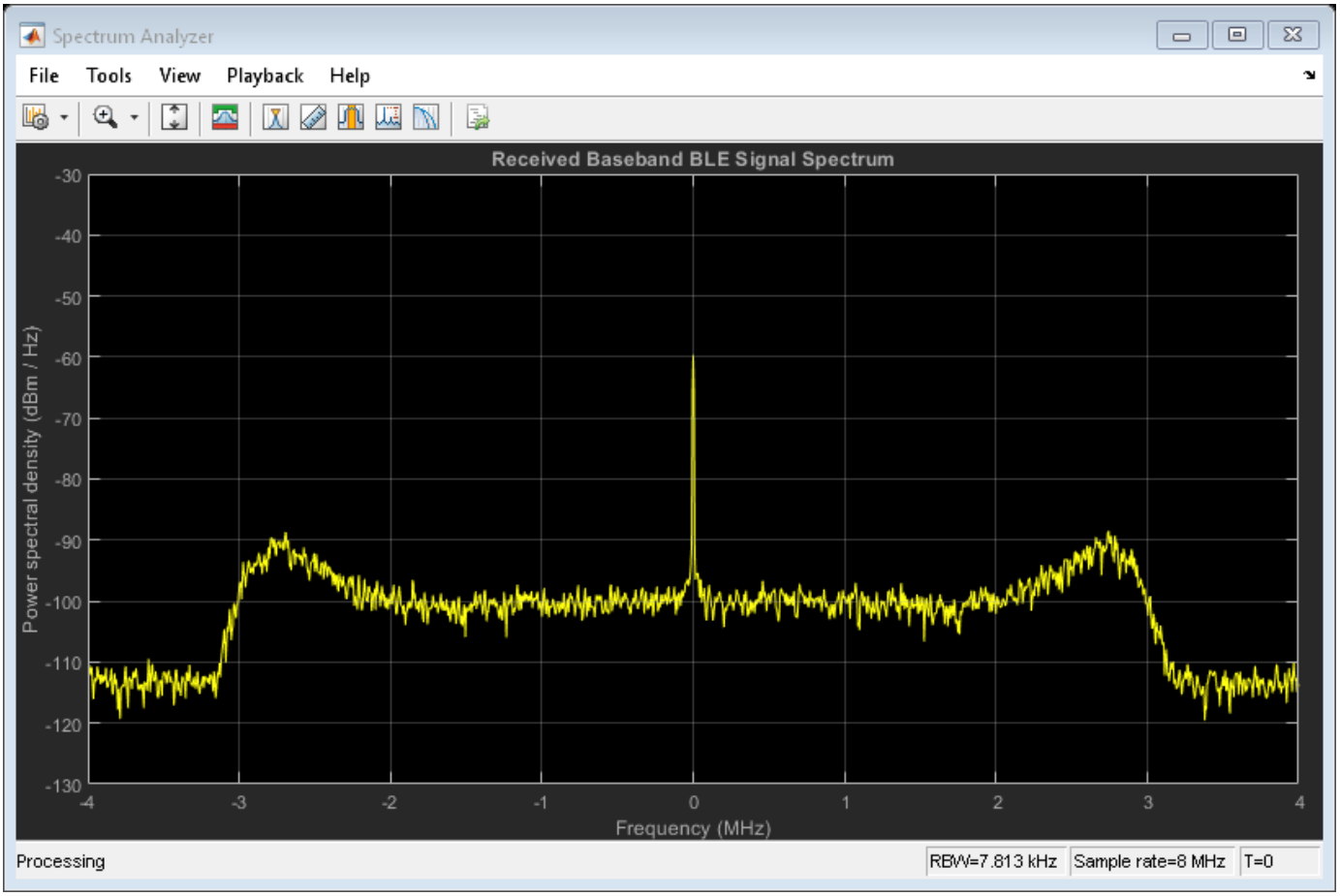
If you assign ADALM-PLUTO as the signal source, the example searches your computer for the ADALM-PLUTO radio at radio address 'usb:0' and uses it as the signal source.

```
signalSource = 'File'; % The default signal source is 'File'

if strcmp(signalSource,'File')
    switch bleParam.Mode
        case 'LE1M'
            bbFileName = 'bleCapturesLE1M.bb';
        case 'LE2M'
            bbFileName = 'bleCapturesLE2M.bb';
        case 'LE500K'
            bbFileName = 'bleCapturesLE500K.bb';
        case 'LE125K'
            bbFileName = 'bleCapturesLE125K.bb';
        otherwise
            error('Invalid PHY transmission mode. Valid entries are LE1M, LE2M, LE500K and LE125K
    end
    sigSrc = comm.BasebandFileReader(bbFileName);
    sigSrcInfo = info(sigSrc);
    sigSrc.SamplesPerFrame = sigSrcInfo.NumSamplesInData;
    bbSampleRate = sigSrc.SampleRate;
    bleParam.SamplesPerSymbol = bbSampleRate/bleParam.SymbolRate;

elseif strcmp(signalSource,'ADALM-PLUTO')

    % First check if the HSP exists
    if isempty(which('plutoradio.internal.getRootDir'))
        error(message('comm_demos:common:NoSupportPackage', ...
            'Communications Toolbox Support Package for ADALM-PLUTO Radio',...
            ['<a href="https://www.mathworks.com/hardware-support/' ...
            'adalm-pluto-radio.html">ADALM-PLUTO Radio Support From Communications Toolbox</a>']))
    end

    bbSampleRate = bleParam.SymbolRate * bleParam.SamplesPerSymbol;
    sigSrc = sdrrx('Pluto',...
        'RadioID',              'usb:0',...
        'CenterFrequency',      2.402e9,...
        'BasebandSampleRate',   bbSampleRate,...
        'SamplesPerFrame',      1e7,...
        'GainSource',          'Manual',...
        'Gain',                 25,...
        'OutputDataType',      'double');
else
    error('Invalid signal source. Valid entries are File and ADALM-PLUTO.');
end

% Setup spectrum viewer
```

```
spectrumScope = dsp.SpectrumAnalyzer( ...
    'SampleRate',       bbSampleRate,...
    'SpectrumType',     'Power density', ...
    'SpectralAverages', 10, ...
    'YLimits',          [-130 -30], ...
    'Title',            'Received Baseband BLE Signal Spectrum', ...
    'YLabel',           'Power spectral density');
```

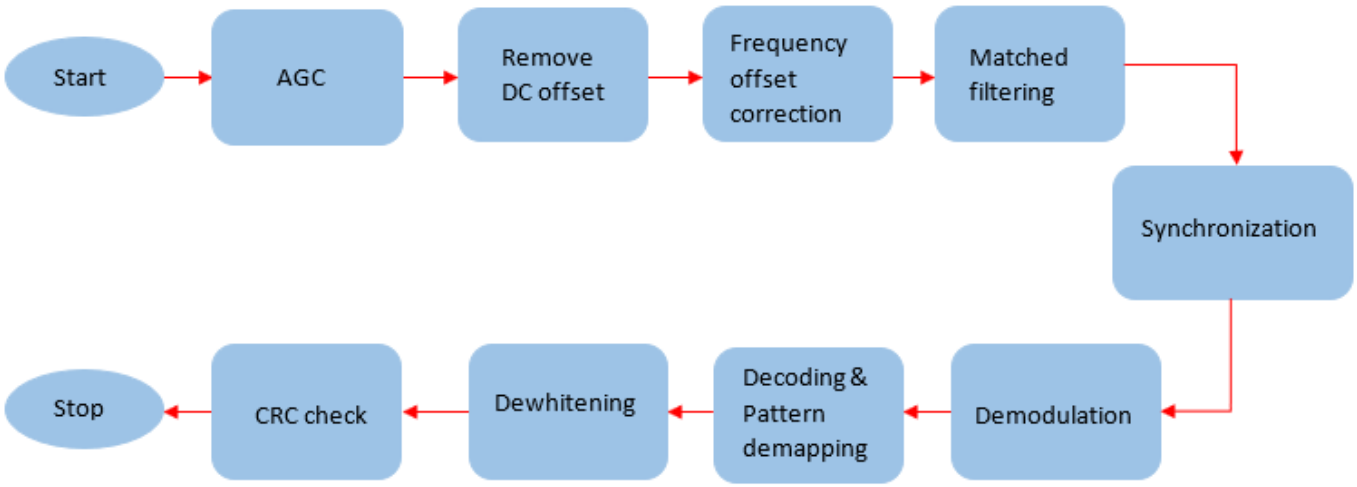
Capture the BLE Packets

```
% The transmitted waveform is captured as a burst
dataCaptures = sigSrc();

% Show power spectral density of the received waveform
spectrumScope(dataCaptures);
```



Receiver Processing

The baseband samples received from the signal source are processed to decode the PDU header information and raw message bits. The following diagram shows the receiver processing.



1. Perform automatic gain control (AGC)
2. Remove DC offset
3. Estimate and correct for the carrier frequency offset
4. Perform matched filtering with gaussian pulse
5. Timing synchronization
6. GMSK demodulation
7. FEC decoding and pattern demapping for LECoded PHYs (LE500K and LE125K)
8. Data dewhitening
9. Perform CRC check on the decoded PDU
10. Compute packet error rate (PER)

```
% Initialize System objects for receiver processing
agc = comm.AGC('MaxPowerGain',20,'DesiredOutputPower',2);

freqCompensator = comm.CoarseFrequencyCompensator('Modulation','OQPSK', ...
    'SampleRate',bbSampleRate,...
    'SamplesPerSymbol',2*bleParam.SamplesPerSymbol,...
    'FrequencyResolution',100);

prbDet = comm.PreambleDetector(bleParam.RefSeq,'Detections','First');

% Initialize counter variables
pktCnt = 0;
crcCnt = 0;
displayFlag = false; % true if the received data is to be printed

% Loop to decode the captured BLE samples
while length(dataCaptures) > bleParam.MinimumPacketLen

    % Consider two frames from the captured signal for each iteration
    startIndex = 1;
    endIndex = min(length(dataCaptures),2*bleParam.FrameLength);
    rcvSig = dataCaptures(startIndex:endIndex);
```

```
        rcvAGC = agc(rcvSig); % Perform AGC
        rcvDCFree = rcvAGC - mean(rcvAGC); % Remove the DC offset
        rcvFreqComp = freqCompensator(rcvDCFree); % Estimate and compensate for the carrier frequency
        rcvFilt = conv(rcvFreqComp,bleParam.h,'same'); % Perform gaussian matched filtering

        % Perform frame timing synchronization
        [~, dtMt] = prbDet(rcvFilt);
        release(prbDet)
        prbDet.Threshold = max(dtMt);
        prbIdx = prbDet(rcvFilt);

        % Extract message information
        [cfgLLAdv,pktCnt,crcCnt,remStartIdx] = helperBLEPhyBitRecover(rcvFilt,...
            prbIdx,pktCnt,crcCnt,bleParam);

        % Remaining signal in the burst captures
        dataCaptures = dataCaptures(1+remStartIdx:end);

        % Display the decoded information
        if displayFlag && ~isempty(cfgLLAdv)
            fprintf('Advertising PDU Type: %s\n',cfgLLAdv.PDUType);
            fprintf('Advertising Address: %s\n',cfgLLAdv.AdvertiserAddress);
        end

        % Release System objects
        release(freqCompensator)
        release(prbDet)
    end

    % Release the signal source
    release(sigSrc)

    % Determine the PER
    if pktCnt
        per = 1-(crcCnt/pktCnt);
        fprintf('Packet error rate for %s mode is %f.\n',bleParam.Mode,per);
    else
        fprintf('\n No BLE packets were detected.\n')
    end
```

```
Packet error rate for LE1M mode is 0.000000.
```

Further Exploration

The companion example “Bluetooth Low Energy Transmitter” on page 3-207 can be used to transmit a standard-compliant BLE waveform which can be decoded by this example. You can also use this example to transmit the data channel PDUs by changing channel index, access address and center frequency values in both the examples.

Troubleshooting

General tips for troubleshooting SDR hardware and the Communications Toolbox Support Package for ADALM-PLUTO Radio can be found in “Common Problems and Fixes” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio).

Appendix

This example uses these helper functions:

* helperBLEReceiverConfig.m: Configures BLE receiver parameters * helperBLEPhyBitRecover.m: Recovers the payload bits

Selected Bibliography

Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.2, Volume 6. https://www.bluetooth.com.

See Also

More About

- “Bluetooth Low Energy Receiver” on page 3-212
- “Bluetooth Low Energy Waveform Generation and Visualization” on page 3-270

Bluetooth Low Energy Bit Error Rate Simulation

This example shows how the Communications Toolbox™ Library for the Bluetooth® Protocol can be used to measure the bit error rate (BER) for different modes of Bluetooth Low Energy (BLE) [1] using an end-to-end physical layer simulation.

Introduction

In this example, an end-to-end simulation is used to determine the BER performance of BLE [1] under an additive white gaussian noise (AWGN) channel for a range of bit energy to noise density ratio (Eb/No) values. At each Eb/No point, multiple BLE packets are transmitted through a noisy channel with no other radio front-end (RF) impairments. Assuming perfect synchronization, an ideal receiver is used to recover the data bits. These recovered data bits are compared with the transmitted data bits to determine the BER. BER curves are generated for the four PHY transmission throughput modes supported in BLE specification [1] as follows:

- Uncoded PHY with data rate of 1 Mbps (LE1M)
- Uncoded PHY with data rate of 2 Mbps (LE2M)
- Coded PHY with data rate of 500 Kbps (LE500K)
- Coded PHY with data rate of 125 Kbps (LE125K)

The following diagram summarizes the simulation for each packet.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Initialize the Simulation Parameters

```
EbNo = -2:2:8;                          % Eb/No range in dB
sps = 4;                                % Samples per symbol
dataLen = 2080;                         % Data length in bits
simMode = {'LE1M','LE2M','LE500K','LE125K'};
```

The number of packets tested at each Eb/No point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of bit errors simulated at each Eb/No point. When the number of bit errors reaches this limit, the simulation at this Eb/No point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each Eb/No point and limits the length of the simulation if the bit error limit is not reached.

The numbers chosen for `maxNumErrors` and `maxNumPackets` in this example will lead to a very short simulation. For statistically meaningful results we recommend increasing these numbers.

```
maxNumErrors = 100; % Maximum number of bit errors at an Eb/No point
maxNumPackets = 10; % Maximum number of packets at an Eb/No point
```

Simulating for Each Eb/No Point

This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each Eb/No point to speed up a simulation. `parfor`, as part of the “Parallel Computing Toolbox”, executes processing for each Eb/No point in parallel to reduce the total simulation time. To enable the use of parallel computing for increased speed, comment out the 'for' statement and uncomment the 'parfor' statement below. If Parallel Computing Toolbox™ is not installed, 'parfor' will default to the normal 'for' statement.

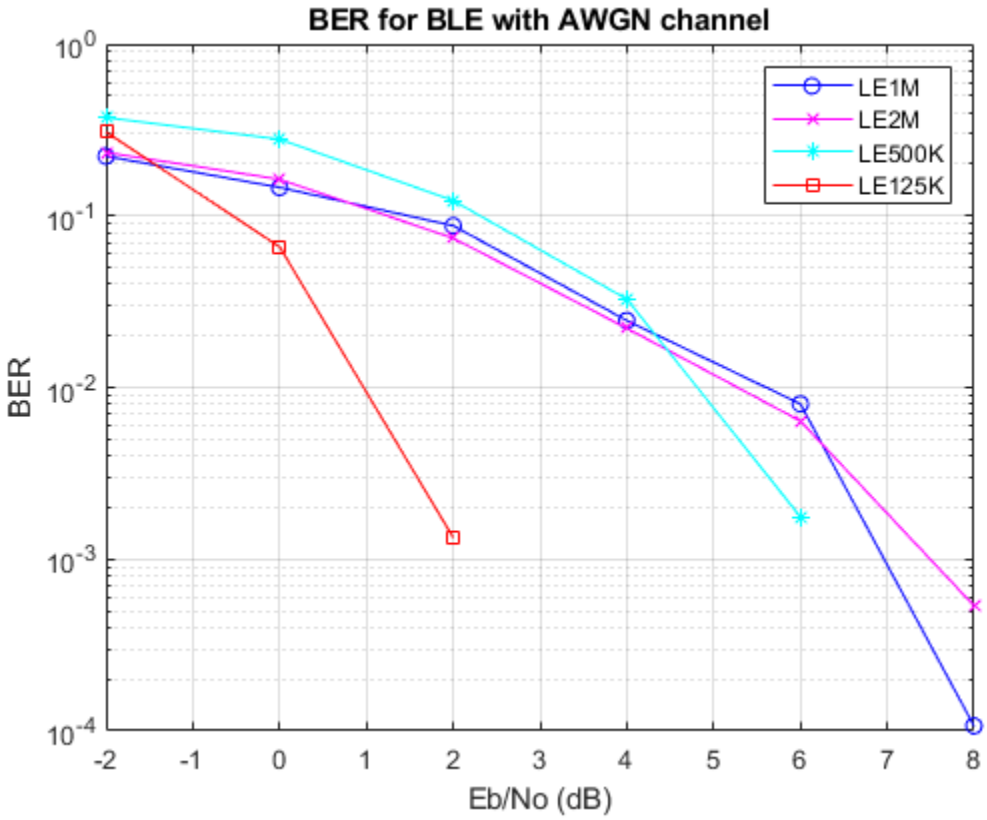
```
numMode = numel(simMode);          % Number of modes
ber = zeros(numMode,length(EbNo)); % Pre-allocate to store BER results

for iMode = 1:numMode

    phyMode = simMode{iMode};
    % Set signal to noise ratio (SNR) points based on mode
    % For Coded PHY's (LE500K and LE125K), the code rate factor is included
    % in SNR calculation as 1/2 rate FEC encoder is used.
    if any(strcmp(phyMode,{'LE1M','LE2M'}))
        snrVec = EbNo - 10*log10(sps);
    else
        codeRate = 1/2;
        snrVec = EbNo + 10*log10(codeRate) - 10*log10(sps);
    end

%     parfor iSnr = 1:length(snrVec)  % Use 'parfor' to speed up the simulation
    for iSnr = 1:length(snrVec)       % Use 'for' to debug the simulation

        % Set random substream index per iteration to ensure that each
        % iteration uses a repeatable set of random numbers
        stream = RandStream('combRecursive','Seed',0);
        stream.Substream = iSnr;
        RandStream.setGlobalStream(stream);

        % Create an instance of error rate
        errorRate = comm.ErrorRate('Samples','Custom','CustomSamples',1:(dataLen-1));

        % Loop to simulate multiple packets
        numErrs = 0;
        numPkt = 1; % Index of packet transmitted
        while numErrs < maxNumErrors && numPkt < maxNumPackets

            % Generate BLE waveform
            txBits = randi([0 1],dataLen,1,'int8'); % Data bits generation
            chanIndex = randi([0 39],1,1); % Random channel index value for each packet
            if chanIndex <=36
                % Random access address for data channels
                % Ideally, this access address value should meet the requirements specified in
                % Section 2.1.2, Part-B, Vol-6 of Bluetooth specification.
                accessAdd = [1 0 0 0 1 1 1 0 1 1 0 0 1 ...
                             0 0 1 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 0]';
            else
                % Default access address for periodic advertising channels
                accessAdd = [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]';
```

```
            end
            txWaveform = bleWaveformGenerator(txBits,'Mode',phyMode,...
                                              'SamplesPerSymbol',sps,...
                                              'ChannelIndex',chanIndex,...
                                              'AccessAddress',accessAdd);

            % Pass the transmitted waveform through AWGN channel
            rxWaveform = awgn(txWaveform,snrVec(iSnr));

            % Recover data bits using ideal receiver
            rxBits = bleIdealReceiver(rxWaveform,'Mode',phyMode,...
                                      'SamplesPerSymbol',sps,...
                                      'ChannelIndex',chanIndex);

            % Determine the BER
            errors = errorRate(txBits,rxBits);
            ber(iMode,iSnr) = errors(1);
            numErrs = errors(2);
            numPkt = numPkt + 1;
        end
    disp(['Mode ' phyMode ', '...
        'Simulating for Eb/No = ', num2str(EbNo(iSnr)), ' dB' ', '...
        'BER:',num2str(ber(iMode,iSnr))])
    end
end

Mode LE1M, Simulating for Eb/No = -2 dB, BER:0.22222
Mode LE1M, Simulating for Eb/No = 0 dB, BER:0.14622
Mode LE1M, Simulating for Eb/No = 2 dB, BER:0.087542
Mode LE1M, Simulating for Eb/No = 4 dB, BER:0.024531
Mode LE1M, Simulating for Eb/No = 6 dB, BER:0.0080167
Mode LE1M, Simulating for Eb/No = 8 dB, BER:0.00010689
Mode LE2M, Simulating for Eb/No = -2 dB, BER:0.23377
Mode LE2M, Simulating for Eb/No = 0 dB, BER:0.16306
Mode LE2M, Simulating for Eb/No = 2 dB, BER:0.074074
Mode LE2M, Simulating for Eb/No = 4 dB, BER:0.022126
Mode LE2M, Simulating for Eb/No = 6 dB, BER:0.0063733
Mode LE2M, Simulating for Eb/No = 8 dB, BER:0.00053444
Mode LE500K, Simulating for Eb/No = -2 dB, BER:0.37326
Mode LE500K, Simulating for Eb/No = 0 dB, BER:0.27898
Mode LE500K, Simulating for Eb/No = 2 dB, BER:0.12266
Mode LE500K, Simulating for Eb/No = 4 dB, BER:0.032708
Mode LE500K, Simulating for Eb/No = 6 dB, BER:0.0017637
Mode LE500K, Simulating for Eb/No = 8 dB, BER:0
Mode LE125K, Simulating for Eb/No = -2 dB, BER:0.30736
Mode LE125K, Simulating for Eb/No = 0 dB, BER:0.065897
Mode LE125K, Simulating for Eb/No = 2 dB, BER:0.0013361
Mode LE125K, Simulating for Eb/No = 4 dB, BER:0
Mode LE125K, Simulating for Eb/No = 6 dB, BER:0
Mode LE125K, Simulating for Eb/No = 8 dB, BER:0
```

Plot BER vs Eb/No Results

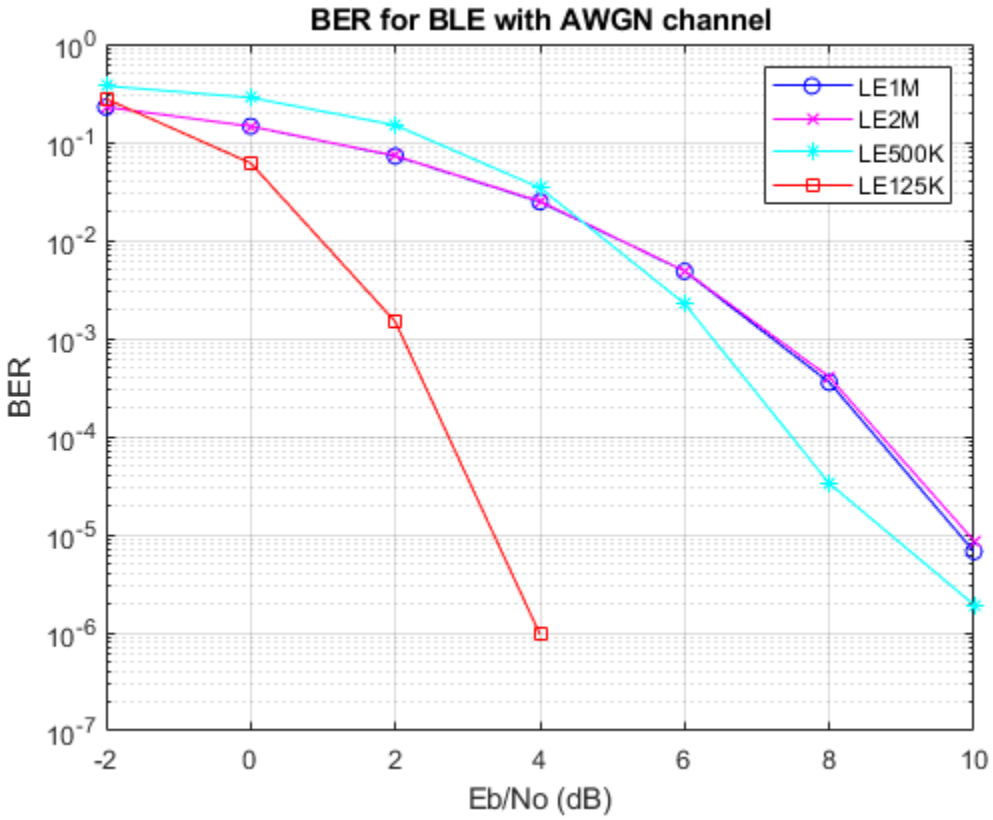
```
markers = 'ox*s';
color = 'bmcr';
dataStr = {zeros(numMode,1)};
figure;
for iMode = 1:numMode
```

```
    semilogy(EbNo,ber(iMode,:).',['-' markers(iMode) color(iMode)]);
    hold on;
    dataStr(iMode) = simMode(iMode);
end
grid on;
xlabel('Eb/No (dB)');
ylabel('BER');
legend(dataStr);
title('BER for BLE with AWGN channel');
```



Further Exploration

The number of packets tested at each Eb/No point is controlled by `maxNumErrors` and `maxNumPackets` parameters. For statistically meaningful results these values should be larger than those presented in this example. The figure below was created by running the example for longer with `maxNumErrors = 1e3`, `maxNumPackets = 1e4`, for all the four modes.



Summary

This example simulates a BLE physical layer link over an AWGN channel. It shows how to generate BLE waveforms, demodulate and decode bits using an ideal receiver and compute the BER.

Selected Bibliography

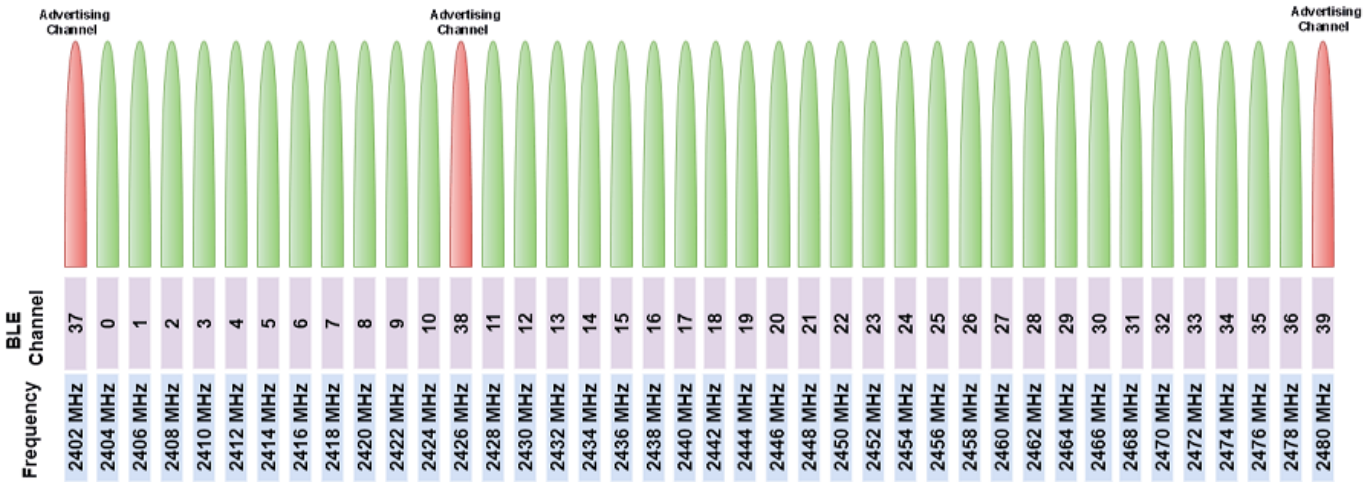
1 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

BLE Channel Selection Algorithms

This example shows how to select a channel index using the channel selection algorithms specified in the Bluetooth® Low Energy (BLE) core specification [1] using the Communications Toolbox™ Library for the Bluetooth® Protocol.

BLE channels

The BLE system operates in the 2.4 GHz ISM band at 2400 - 2483.5 MHz. It uses forty RF channels (each channel is 2 MHz wide). The figure below shows the mapping between the frequencies and BLE channels. Each of these RF channels is allocated a unique channel index (labelled as "BLE Channel" in the figure).



BLE classifies these forty RF channels into three advertising channels (channel indices: 37, 38, 39) and thirty-seven data channels (channel indices: 0 to 36). Note that the advertising channels are spread across the 2.4 GHz spectrum. The purpose of this wide spacing is to avoid interference from other devices operating in the same spectrum, such as WLAN. Advertising channels are mainly used for transmitting advertising packets, scan request/response packets and connection indication packets. Data channels are mainly used for exchanging data packets.

Channel Hopping

Channel hopping is used in Bluetooth to reduce interference and improve throughput. The Bluetooth standard defines rules for switching between channels and algorithms used when performing channel hopping.

Use of the unlicensed 2.4GHz ISM band by several wireless technologies causes increased interference and results in retransmissions to correct errors in received packets. Since BLE is a low energy oriented protocol, it is more susceptible to interference. BLE uses channel hopping to combat the impact of interference. When one channel is completely blocked due to interference, devices can still continue to communicate with each other on other channels.

In classic Bluetooth, channel hopping is restricted to 1600 frequency hops/sec. For BLE, the channel hopping specification has been revised. Different rules apply for advertising and connected devices, and two channel selection algorithms are defined.

An advertising device transmits advertising packets on the three advertising channels in a cyclic manner (starting from channel index 37). The same procedure is used by the scanning/initiating device, listening on the three advertising channels in a cyclic manner.

A connected device changes to a new data channel for every connection event. A connection event is a sequence of data packet exchanges between two connected devices. The connection events occur periodically with an interval called connection interval. All the packets within a connection event are transmitted on the same data channel. A new connection event uses a new data channel.

Two alternative channel selection algorithms are specified by the Bluetooth core specification (see Section 4.5.8, Part-B, Vol-6 of [1]) can be used to select data channels for each connection event:

1. Algorithm #1
2. Algorithm #2

The two channel selection algorithms avoid channels that are prone to transmission errors. A channel map is exchanged between the master and slave devices. This map indicates the good and bad data channels. The classification of good and bad data channels is implementation dependent and can be done based on various parameters like SNR (Signal-To-Noise Ratio), PER (Packet Error Rate), etc. Only the good data channels are used for communication between devices. The channel map will be updated by the master device if it recognizes any bad data channels. The two channel selection algorithms use the channel map to determine whether the selected data channel is good to use. If the selected data channel turns out to be bad, a new data channel is selected using channel remapping procedure (see Section 4.5.8, Part-B, Vol-6 of [1]), which remaps the bad data channel to one of the good data channels. Each algorithm has a remapping procedure of its own.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Simulating Algorithm #1

You can use `bleChannelSelection` System object to select a new channel index. This System object configures the fields required for selecting a channel index.

Create a System object for 'Algorithm #1'

To select a channel index, create a `bleChannelSelection` System object with `Algorithm` set to 1.

```
csa = bleChannelSelection('Algorithm', 1);
```

Configure the fields.

- The `HopIncrement` property defines the hop increment count to be used. The default value is 5. This property is applicable for 'Algorithm #1'.
- The `UsedChannels` property defines the list of used (good) data channels.

```
csa.HopIncrement = 8;
csa.UsedChannels = [0, 5, 13, 9, 24, 36]
```

```
csa =
```

```
  bleChannelSelection with properties:

      Algorithm: 1
    HopIncrement: 8
    UsedChannels: [0 5 9 13 24 36]
    ChannelIndex: 0
    EventCounter: 0
```

- `ChannelIndex` is a read-only property that indicates the current channel being used.
- `EventCounter` is a read-only property that indicates the number of connection events occurred until now. It is incremented for every new selected channel.

Select a channel index for next hop

Call the object `csa` as a function to determine the next channel hop and to select a new channel for each new connection event.

```
nextChannel = csa();
fprintf('Selected channel for connection event %d using ''Algorithm #1'' is: %d\n', csa.EventCou
```

```
Selected channel for connection event 0 using 'Algorithm #1' is: 9
```

Simulating Algorithm #2

You can use `bleChannelSelection` System object to select a new channel index. This System object configures the fields required for selecting a channel index.

Create a System object for 'Algorithm #2'

To select a channel index, Create a `bleChannelSelection` System object with `Algorithm` set to 2.

```
csa = bleChannelSelection('Algorithm', 2);
```

Configure the fields.

- The `AccessAddress` property defines the 32-bit unique connection address between two devices. The default value is '8E89BED6'. This property is applicable for 'Algorithm #2'.
- The `UsedChannels` property defines the list of used (good) data channels.

```
csa.AccessAddress = 'E89BED68';
csa.UsedChannels = [9, 10, 21, 22, 23, 33, 34, 35, 36]
```

```
csa =

  bleChannelSelection with properties:

                 Algorithm: 2
             AccessAddress: 'E89BED68'
    SubeventChannelSelection: 0
              UsedChannels: [9 10 21 22 23 33 34 35 36]
              ChannelIndex: 0
              EventCounter: 0
```

Select a channel index for next hop

Call the object `csa` as a function to determine the next channel hop and to select a new channel for each new connection event.

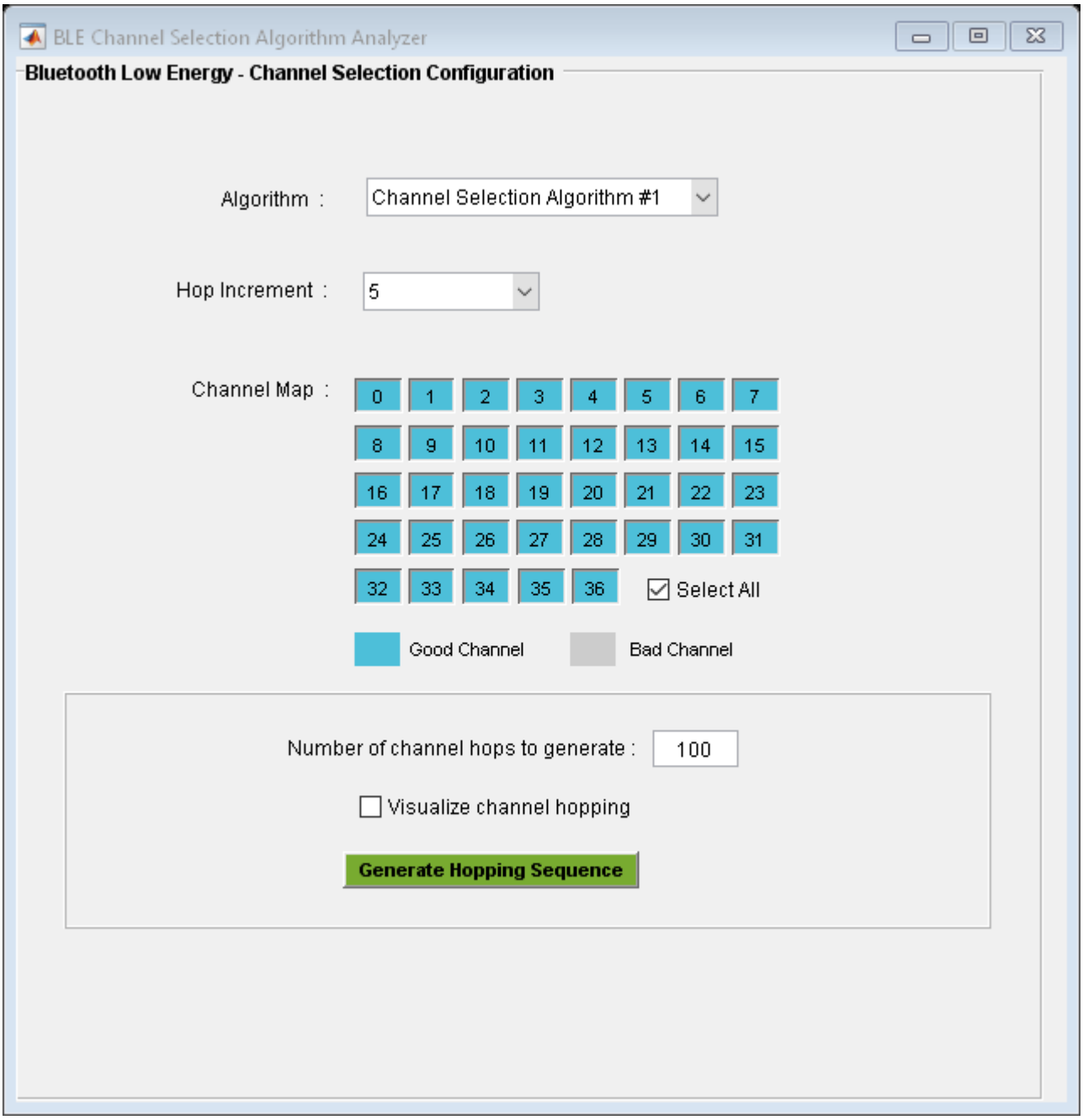
```
nextChannel = csa();
fprintf('Selected channel for connection event %d using ''Algorithm #2'' is: %d\n', csa.EventCou
```

```
Selected channel for connection event 0 using 'Algorithm #2' is: 22
```

GUI for analyzing Channel Selection Algorithms

The function helperBLEChannelHopSelectionUI provides a graphical user interface to generate desired number of channel hops for analyzing the algorithm. Both channel selection algorithms can be analyzed using this GUI. It can be used to plot the channel hopping pattern of an algorithm and also plots the corresponding histogram.

```
helperBLEChannelHopSelectionUI()
```



Algorithm verification with sample data

Sample data is provided to verify Algorithm #2 (see Section 3, Vol 6, Part C in [1]). However, there is no sample data available for verifying Algorithm #1.

Sample data 1 (thirty-seven good data channels)

1. Access Address = 8E89BED6
2. Used Channels = [0:36]

When the above inputs are used, the Algorithm #2 is expected to select the following channels according to Section 3.1, Part-B, Vol-6 of [1]

EventCounter	Channel
1	20
2	6
3	21

The following code selects three channels for the first three connection events.

```
% Create a System object for 'Algorithm #2'
csa = bleChannelSelection('Algorithm', 2);
```

Configure the fields with sample data #1.

```
% Connection access address
csa.AccessAddress = '8E89BED6';
% Use 37 good data channels as used channels according to the sample data
csa.UsedChannels = (0:36);
```

Select channel indices for first 3 connection events. Verify the generated outputs with the table mentioned above.

```
numConnectionEvents = 4;
for i = 1:numConnectionEvents
    channel = csa();
    fprintf('Event Counter: %d, selected Channel: %d\n', csa.EventCounter, channel);
end
```

```
Event Counter: 0, selected Channel: 25
Event Counter: 1, selected Channel: 20
Event Counter: 2, selected Channel: 6
Event Counter: 3, selected Channel: 21
```

Sample data 2 (nine good data channels)

1. Access Address = 8E89BED6
2. Used Channels = [9, 10, 21, 22, 23, 33, 34, 35, 36]

When the above inputs are used, the Algorithm #2 is expected to select the following channels according to Section 3.2, Part-B, Vol-6 of [1]. Since the channel map contains bad channels, the channel remapping procedure used in the algorithm is also verified.

EventCounter	Channel
6	23
7	9
8	34

The following code selects eight channels for the first eight connection events.

```
% Create a System object for 'Algorithm #2'
csa = bleChannelSelection('Algorithm', 2);
```

Configure the fields with sample data #2.

```
% Connection access address
csa.AccessAddress = '8E89BED6';
% Use 9 good data channels as used channels according to the sample data
csa.UsedChannels = [9, 10, 21, 22, 23, 33, 34, 35, 36];
```

Select channel indices for first 8 connection events. Verify the generated outputs with the table mentioned above.

```
numConnectionEvents = 9;
for i = 1:numConnectionEvents
    channel = csa();
    fprintf('Event Counter: %d, selected Channel: %d\n', csa.EventCounter, channel);
end
```

```
Event Counter: 0, selected Channel: 35
Event Counter: 1, selected Channel: 9
Event Counter: 2, selected Channel: 33
Event Counter: 3, selected Channel: 21
Event Counter: 4, selected Channel: 34
Event Counter: 5, selected Channel: 36
Event Counter: 6, selected Channel: 23
Event Counter: 7, selected Channel: 9
Event Counter: 8, selected Channel: 34
```

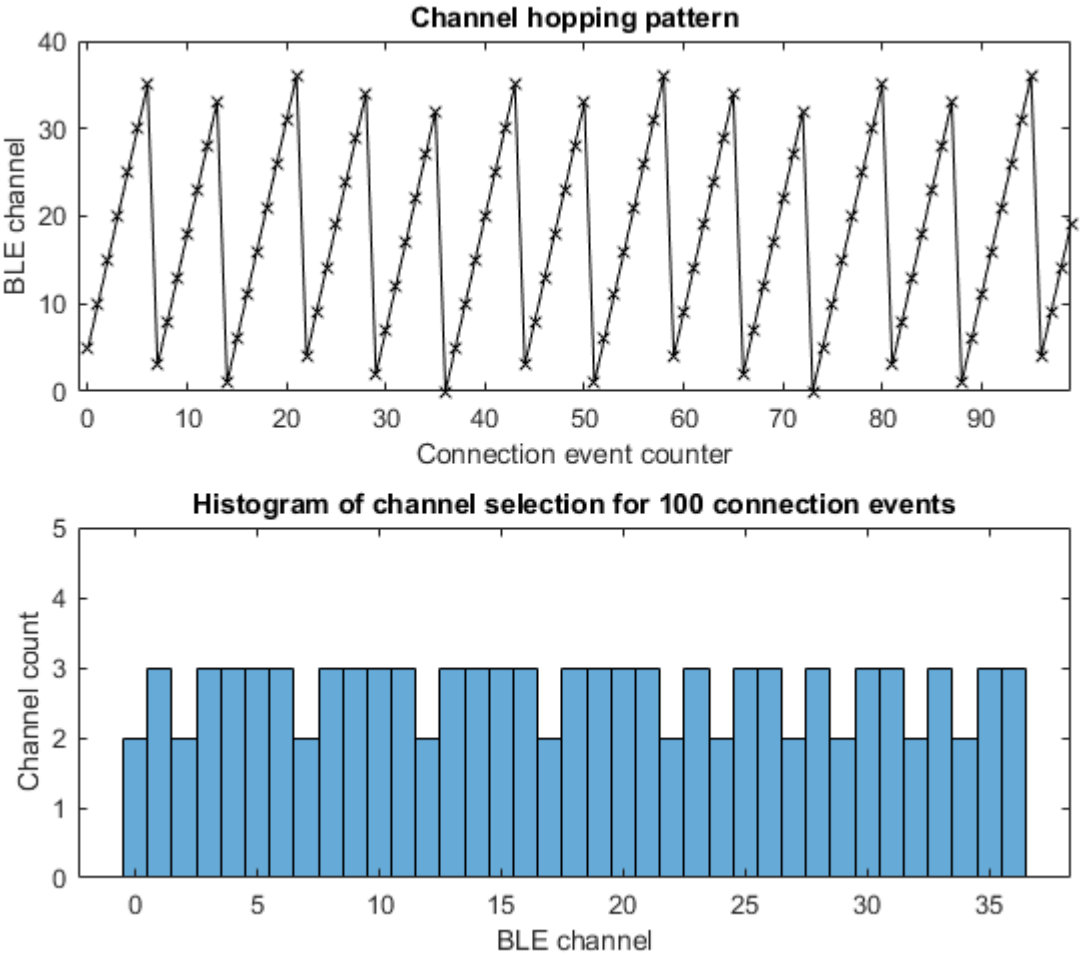
Plot and analyze the hopping pattern - Algorithm #1 and Algorithm #2

The following code selects channel indices for the first hundred connection events using 'Algorithm #1'. The selected channels are plotted and compared with those of 'Algorithm #2'.

```
% Channel selection algorithm System object for 'Algorithm #1'
csa = bleChannelSelection;
% For 100 connection events
numConnectionEvents = 100;
hopSequence = zeros(1, numConnectionEvents);
% Generate channel hop sequence for 100 connection events
for i = 1:numConnectionEvents
    hopSequence(i) = csa();
end
```

The helperBLEPlotChannelHopSequence function plots the hopping pattern and also outputs a histogram of the selected channels.

```
helperBLEPlotChannelHopSequence(csa, hopSequence);
```

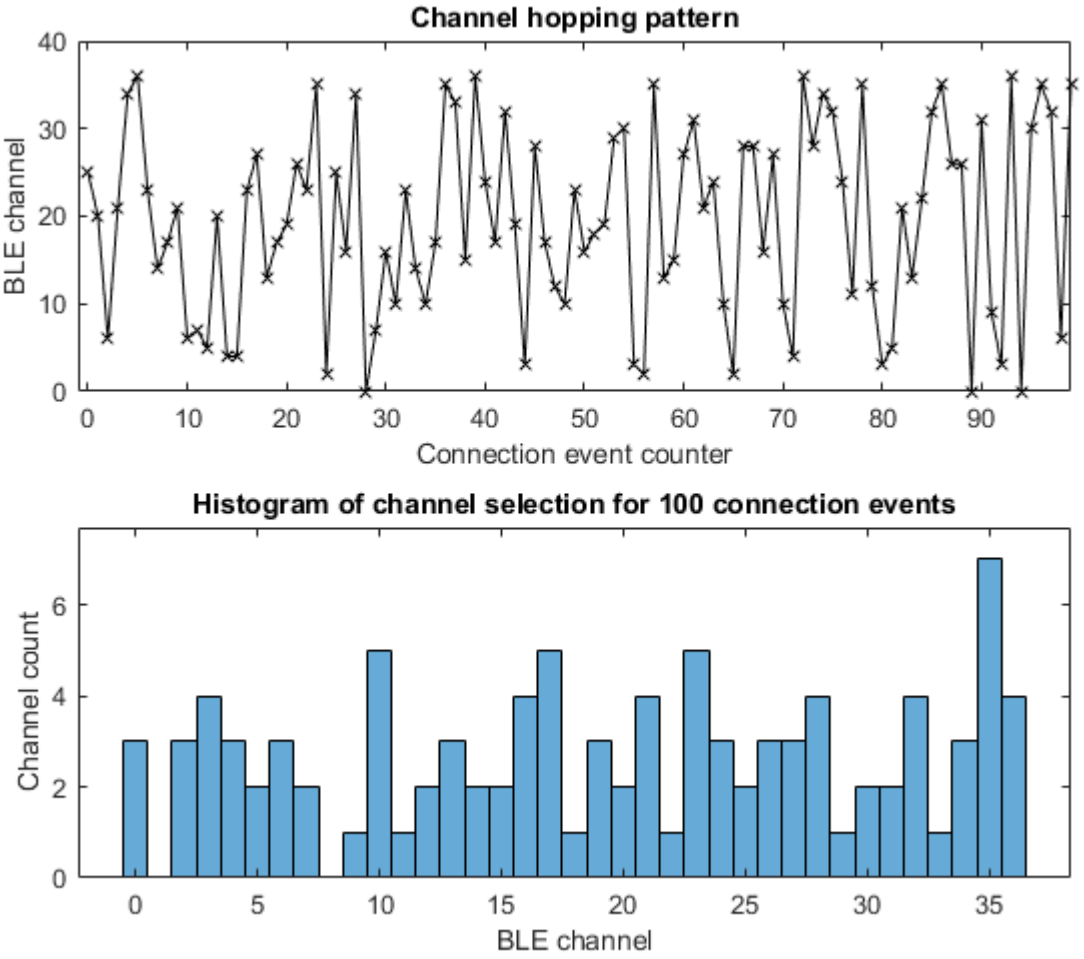


The following code generates channel indices for the first hundred connection events using 'Algorithm #2'. The selected channels are plotted and compared with those of 'Algorithm #1'.

```
% Channel selection algorithm System object for 'Algorithm #2'
csa = bleChannelSelection('Algorithm', 2);
% For 100 connection events
numConnectionEvents = 100;
hopSequence = zeros(1, numConnectionEvents);
% Generate channel hop sequence for 100 connection events
for i = 1:numConnectionEvents
    hopSequence(i) = csa();
end
```

The helperBLEPlotChannelHopSequence function plots the hopping pattern and also outputs a histogram of the selected channels.

```
helperBLEPlotChannelHopSequence(csa, hopSequence);
```



Algorithm #1 vs Algorithm #2

The above plots show the difference between the two algorithms.

- Algorithm #1 is a simple incremental algorithm that produces a uniform sequence of channels. There is no randomization involved in the process of selecting a new channel.
- Algorithm #2 was introduced in version 5.0 of the Bluetooth Core Specification [1]. Compared to Algorithm #1, this is more complex and produces a randomized sequence of channels.

Conclusion

This example demonstrated the behavior of the channel selection algorithms specified in the Bluetooth core specification [1].

Appendix

The example uses this feature:

- `bleChannelSelection`: Create a System object used for selecting a new data channel to transmit the data packet

The example uses these helpers:

- helperBLEChannelHopSelectionUI: Script for helperBLEChannelHopSelectionUI figure
- helperBLEPlotChannelHopSequence: Plot the channel hopping sequence for a given algorithm

Selected Bibliography

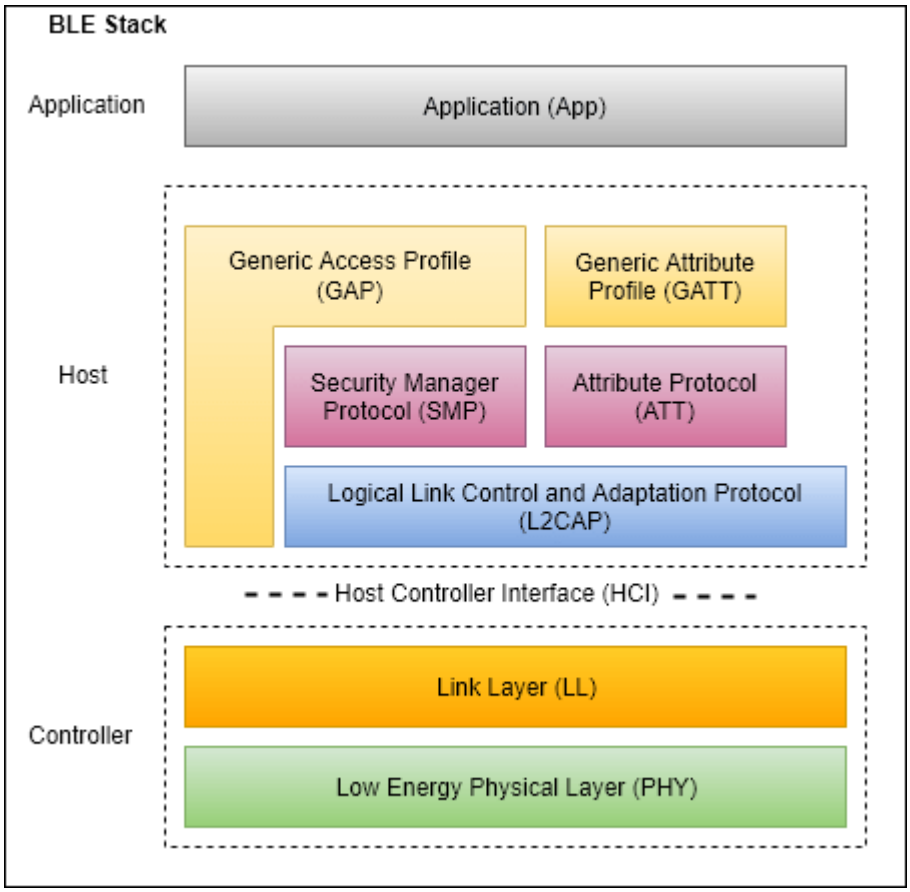
1 Bluetooth® Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed July 8, 2020. https://www.bluetooth.com/.

Modeling of BLE Devices with Heart Rate Profile

This example shows the modeling of Bluetooth® Low Energy devices with Heart Rate Profile using the Communications Toolbox™ Library for the Bluetooth® Protocol.

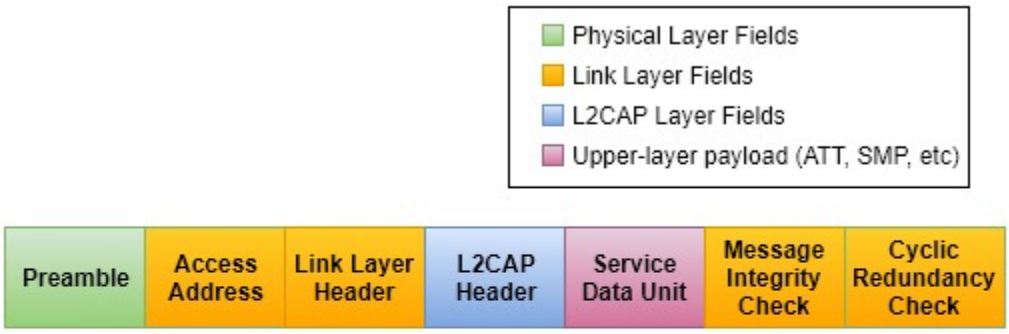
Background

The Bluetooth core specification [1] includes a Low Energy version for low-rate wireless personal area networks, that is referred to as Bluetooth Low Energy (BLE) or Bluetooth Smart. The BLE stack consists of: Generic Attribute Profile (GATT), Attribute Protocol (ATT), Security Manager Protocol (SMP), Logical Link Control and Adaptation Protocol (L2CAP), Link Layer (LL) and Physical layer (PHY). BLE was added to the standard for low energy devices generating small amounts of data, such as notification alerts used in such applications as home automation, health-care, fitness, and Internet of Things (IoT).



Attribute Protocol

The ATT is built on top of the L2CAP layer of BLE. ATT defines a set of Protocol Data Units (PDUs) that are used for data exchange in GATT-based profiles.



Format of BLE (Bluetooth Low Energy) frame

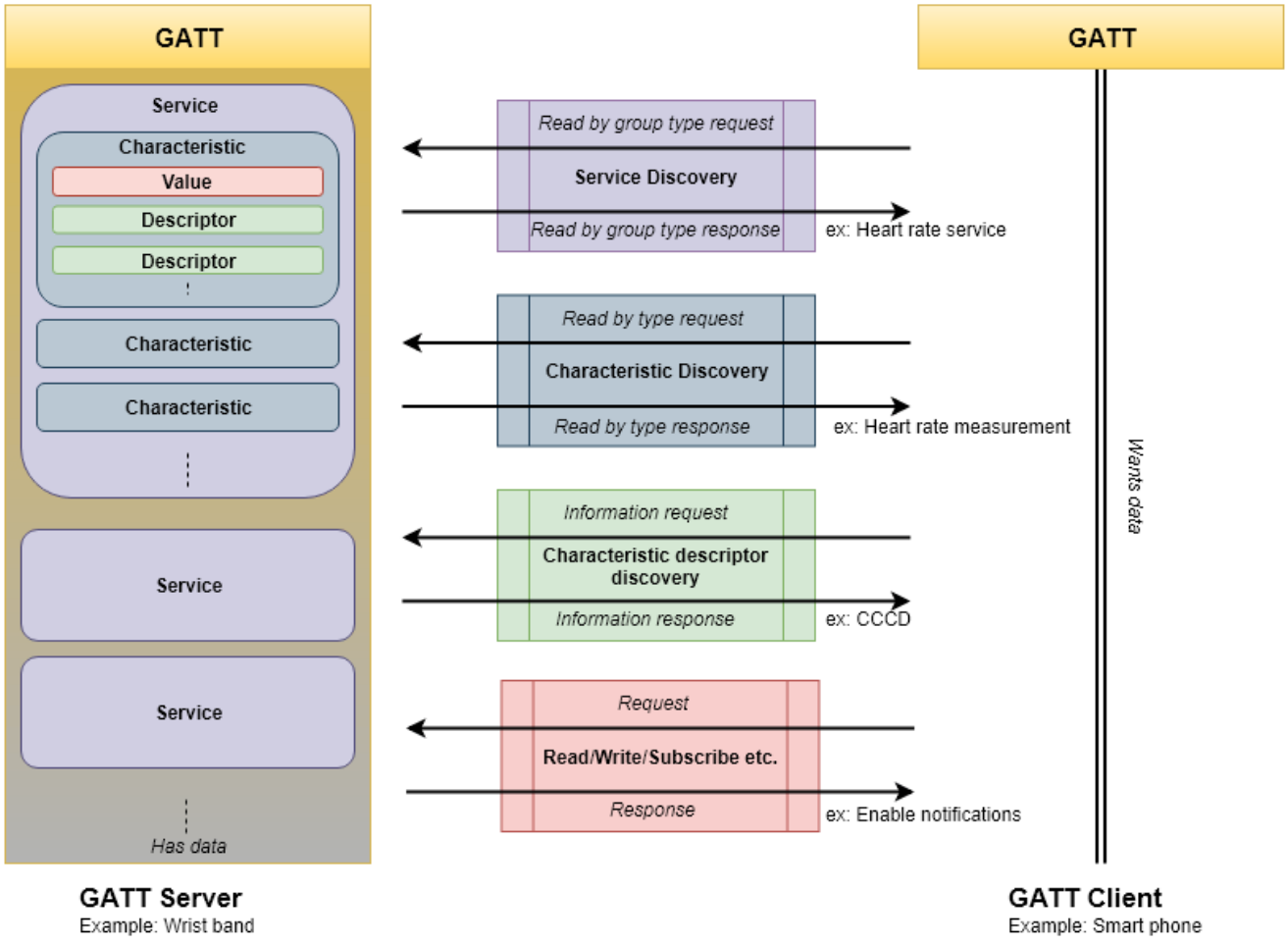
Service Data Unit - Attribute Protocol(ATT) PDU format

Generic Attribute Profile

The GATT is a service framework built using ATT. GATT handles the generation of requests or responses based on application data from the higher layers or ATT PDU received from the lower layer. It stores the information in the form of services, characteristics, and characteristic descriptors. It uses a client-server architecture.

GATT Terminology:

- **Service:** A service is a collection of data and associated behaviors to accomplish a particular function or feature. Example: A heart rate service that allows measurement of a heart rate.
- **Characteristic:** A characteristic is a value used in a service along with its permissions. Example: A heart rate measurement characteristic contains information about the measured heart rate value.
- **Characteristic descriptor:** Descriptors of the characteristic behavior. Example: A Client Characteristic Configuration Descriptor (CCCD), describes whether or not the server has to notify the client in a response containing the characteristic value.
- **GATT-Client:** Initiates commands and requests to the server, and receives responses, indications and notifications sent by the server.
- **GATT-Server:** Accepts incoming commands and requests from a client, and sends responses, indications, and notifications to the client.



BLE GATT Client-Server model

Heart Rate Profile

Heart Rate Profile (HRP) [2] is a GATT-based low energy profile defined by the Bluetooth Special Interest Group (SIG). The HRP defines the communication between a GATT-server of a heart rate sensor device, such as a wrist band, and a GATT-client, such as a smart phone or tablet. The HRP is widely used in fitness applications to collect heart rate measurements.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

BLE HRP Client-Server Scenario

In this scenario, the GATT-server is a wrist band with a heart rate sensor and the GATT-client is a smart phone.

```
% Create objects for GATT-server and GATT-client devices.
gattServer = helperBLEGATTServer;
gattClient = helperBLEGATTClient;
```

Initially, the HRP client discovers the services, characteristics, characteristic descriptors defined at the server. After discovery, the client subscribes for heart rate measurement notifications.

Service Discovery

Clients perform a *service discovery* operation to get information about the available services. In service discovery, the client invokes `'Discover all primary services'` by sending a *Read by group type request* ATT PDU. The server responds with the available services and their associated handles by sending a 'Read by group type response' ATT PDU. A *handle* is a unique identifier of an attribute that are dynamically assigned by the server.

Client request for services at Server

The generateATTPDU function generates an ATT PDU corresponding to the given sub-procedure as specified in the Bluetooth core specification.

```
% Preallocate a variable to store the generated link layer packets.
pcapPackets = cell(1, 9);
count = 1;

% Configure a GATT client to discover services available at the server.
gattClient.SubProcedure = 'Discover all primary services';
serviceDiscReqPDU = generateATTPDU(gattClient);

% Transmit the application data (|serviceDiscReqPDU|) to the server through
% PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(serviceDiscReqPDU);
count = count+1;
```

Receive Client request at Server

The server receives a *Read by group type request* from the client and sends the list of available services in a *Read by group type response* ATT PDU.

The receiveData function decodes the incoming PDU as a GATT-server and returns the corresponding ATT PDU configuration object and the appropriate response PDU.

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode the received ATT PDU and generate response PDU, if applicable.
[attServerRespPDU, serviceDiscReqCfg, gattServer] = receiveData(gattServer, receivedPDU);

fprintf("Received service discovery request at the server:\n")
serviceDiscReqCfg

% Transmit the application response data (|attServerRespPDU|) to the client
% through PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(attServerRespPDU);
count = count+1;
```

```
Received service discovery request at the server:
```

```
serviceDiscReqCfg =

  bleATTPDUConfig with properties:

           Opcode: 'Read by group type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'

   Read-only properties:
    No properties.
```

Receive Server response at Client

The receiveData function decodes the incoming PDU as a GATT-client and returns the corresponding ATT PDU configuration object and the appropriate response PDU, if applicable.

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[~, serviceDiscRespCfg] = receiveData(gattClient, receivedPDU);
gattClient.StartHandle = serviceDiscRespCfg.StartHandle;
gattClient.EndHandle = serviceDiscRespCfg.EndHandle;

% Expected response from the server: |'Read by group type response'| or
% |'Error response'|.
if strcmp(serviceDiscRespCfg.Opcode, 'Error response')
    fprintf("Received error response at the client:\n")
    serviceDiscRespCfg
    serviceDiscRespMsg = ['Error response(''' serviceDiscRespCfg.ErrorMessage ''')'];
else
    fprintf("Received service discovery response at the client:\n")
    serviceDiscRespCfg
    service = helperBluetoothID.getBluetoothName(serviceDiscRespCfg.AttributeValue);
    serviceDiscRespMsg = ['Service discovery response(''' service ''')'];
end

Received service discovery response at the client:

serviceDiscRespCfg =

  bleATTPDUConfig with properties:

            Opcode: 'Read by group type response'
       StartHandle: '0001'
         EndHandle: '0006'
    AttributeValue: [2x2 char]

   Read-only properties:
    No properties.
```

Characteristics Discovery

A service consists of multiple characteristics. For each service, there are information elements exchanged between a client and server. Each information element may contain descriptors of its

behavior. A characteristic contains a value and its associated descriptors. After discovering the service, clients perform *characteristics discovery* to learn about the characteristics defined in the service. In characteristic discovery, the client invokes `'Discover all characteristics of service'` by sending 'Read by type request' ATT PDU. The server responds with the available characteristics and their associated handles by sending a 'Read by type response' ATT PDU.

Client request for characteristics at Server

```
% Configure a GATT client to discover all the available characteristics at
% the server.
gattClient.SubProcedure = 'Discover all characteristics of service';
chrsticDiscReqPDU = generateATTPDU(gattClient);

% Transmit the application data (|chrsticDiscReqPDU|) to the server through
% PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(chrsticDiscReqPDU);
count = count+1;
```

Receive Client request at Server

Decodes the received request and return the list of available characteristics in a *Read by type response* ATT PDU.

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[chrsticDiscRespPDU, chrsticDiscReqCfg, gattServer] = receiveData(gattServer, receivedPDU);

fprintf("Received characteristic discovery request at the server:\n")
chrsticDiscReqCfg

% Transmit the application response data (|chrsticDiscRespPDU|) to the
% client through PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(chrsticDiscRespPDU);
count = count+1;
```

```
Received characteristic discovery request at the server:

chrsticDiscReqCfg =

  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: '0006'
    AttributeType: '2803'

   Read-only properties:
    No properties.
```

Receive Server response at Client

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
```

```
[~, chrsticDiscRespCfg] = receiveData(gattClient, receivedPDU);

% Expected response from the server: |'Read by type response'| or |'Error
% response'|.
if strcmp(chrsticDiscRespCfg.Opcode, 'Error response')
    fprintf("Received error response at the client:\n")
    chrsticDiscRespCfg
    chrsticDescRespMsg = ['Error response(''' chrsticDiscRespCfg.ErrorMessage ''')'];
else
    fprintf("Received characteristic discovery response at the client:\n")
    attributeValueCfg = helperBLEDecodeAttributeValue(...
        chrsticDiscRespCfg.AttributeValue, 'Characteristic');
    attributeValueCfg
    chrsticDescRespMsg = ['Characteristic discovery response(''' attributeValueCfg.Characteristi
end
```

```
Received characteristic discovery response at the client:

attributeValueCfg =

  helperBLEAttributeValueConfig with properties:

                   AttributeType: 'Characteristic'
                   BroadcastFlag: 'False'
                        ReadFlag: 'False'
         WriteWithoutResponseFlag: 'False'
                       WriteFlag: 'False'
                      NotifyFlag: 'True'
                    IndicateFlag: 'False'
    AuthenticatedSignedWritesFlag: 'False'
           ExtendedPropertiesFlag: 'False'
        CharacteristicValueHandle: '0003'
              CharacteristicType: 'Heart rate measurement'

   Read-only properties:
    No properties.
```

Characteristic Descriptor Discovery

A characteristic may consists of multiple characteristic descriptors. After discovering the characteristic, clients perform *characteristic descriptors discovery* to learn about the list of descriptors and their handles. In characteristic descriptor discovery, the client invokes `'Discover all descriptors'` by sending 'Information request' ATT PDU. The server responds with the available characteristic descriptors and their associated handles by sending a 'Information response' ATT PDU.

Client request for characteristic descriptors at Server

```
% Configure a GATT client to discover all the available characteristic
% descriptors at the server.
gattClient.SubProcedure = 'Discover all descriptors';
gattClient.StartHandle = dec2hex(hex2dec(chrsticDiscRespCfg.AttributeHandle)+1, 4);
chrsticDescDiscReqPDU = generateATTPDU(gattClient);

% Transmit the application data (|chrsticDescDiscReqPDU|) to the client
% through PHY.
```

```
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(chrsticDescDiscReqPDU);
count = count+1;
```

Receive Client request at Server

Decodes the received request and returns the list of available characteristic descriptors in a *Information response* ATT PDU.

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[chrsticDescDiscRespPDU, chrsticDescDiscReqCfg, gattServer] = receiveData(gattServer, receivedPDU

fprintf("Received characteristic descriptor discovery request at the server:\n")
chrsticDescDiscReqCfg

% Transmit the application response data (|chrsticDescDiscRespPDU|) to the
% client through PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(chrsticDescDiscRespPDU);
count = count+1;
```

```
Received characteristic descriptor discovery request at the server:

chrsticDescDiscReqCfg =

  bleATTPDUConfig with properties:

         Opcode: 'Information request'
    StartHandle: '0003'
      EndHandle: '0006'

   Read-only properties:
    No properties.
```

Receive Server response at Client

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[~, chrsticDescDiscRespCfg] = receiveData(gattClient, receivedPDU);

% Expected response from the server: |'Information response'| or |'Error
% response'|.
if strcmp(chrsticDescDiscRespCfg.Opcode, 'Error response')
    fprintf("Received error response at the client:\n")
    chrsticDescDiscRespCfg
    chrsticDescDiscRespMsg = ['Error response(''' chrsticDescDiscRespCfg.ErrorMessage ''')'];
else
    fprintf("Received characteristic descriptor discovery response at the client:\n")
    chrsticDescDiscRespCfg
    descriptor = helperBluetoothID.getBluetoothName(chrsticDescDiscRespCfg.AttributeType);
    chrsticDescDiscRespMsg = ['Characteristic descriptor discovery response(''' descriptor ''')'
end
```

```
Received characteristic descriptor discovery response at the client:
```

```
chrsticDescDiscRespCfg =

  bleATTPDUConfig with properties:

            Opcode: 'Information response'
            Format: '16 bit'
    AttributeHandle: '0004'
      AttributeType: '2902'

   Read-only properties:
    No properties.
```

Subscribe for Notifications

After discovering the characteristic descriptors, the client may enable or disable *notifications* for its characteristic value. To enable notifications, the client must set the notification bit (first bit) of *Client Characteristic Configuration Descriptor (CCCD)* value by invoking `'Write characteristic value'` sub-procedure.

Client subscribe for notifications at Server

```
% Configure a GATT client to enable the notifications of Heart rate
% measurement characteristic.
gattClient.SubProcedure = 'Write characteristic value';
gattClient.AttributeHandle = chrsticDescDiscRespCfg.AttributeHandle;
gattClient.AttributeValue = '0100';
enableNotificationReqPDU = generateATTPDU(gattClient);

% Transmit the application data (|enableNotificationReqPDU|) to the client
% through PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(enableNotificationReqPDU);
count = count+1;
```

Receive Client request at Server

Decodes the received request and sends the response in a *Write response* ATT PDU.

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[enableNotificationRespPDU, enableNotificationReqCfg, gattServer] = receiveData(gattServer, rece

fprintf("Received enable notification request at the server:\n")
enableNotificationReqCfg

% Transmit the application response data (|enableNotificationRespPDU|) to
% the client through PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(enableNotificationRespPDU);
count = count+1;
```

```
Received enable notification request at the server:

enableNotificationReqCfg =

  bleATTPDUConfig with properties:
```

```
            Opcode: 'Write request'
    AttributeHandle: '0004'
     AttributeValue: [2x2 char]

   Read-only properties:
    No properties.
```

Receive Server response at Client

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[~, enableNotificationRespCfg] = receiveData(gattClient, receivedPDU);

% Expected response from the server: |'Write response'| or |'Error
% response'|.
if strcmp(enableNotificationRespCfg.Opcode, 'Error response')
    fprintf("Received error response at the client:\n")
    enableNotificationRespCfg
    enableNotificRespMsg = ['Error response(''' enableNotificationRespCfg.ErrorMessage ''')'];
else
    fprintf("Received enable notification response at the client:\n")
    enableNotificationRespCfg
    enableNotificRespMsg = 'Notifications enabled(''Heart rate measurement '')';
end
```

```
Received enable notification response at the client:

enableNotificationRespCfg =

  bleATTPDUConfig with properties:

    Opcode: 'Write response'

   Read-only properties:
    No properties.
```

Notifying the Heart Rate Measurement Value to the Client

When a client enables notifications for a characteristic, the server periodically notifies the value of characteristic (*Heart rate measurement*) to the client.

The HRP server notifies heart rate measurement to the client after its subscription.

Server sends notifications to Client

The notifyHeartRateMeasurement function generates notification PDU as specified in the Bluetooth core specification.

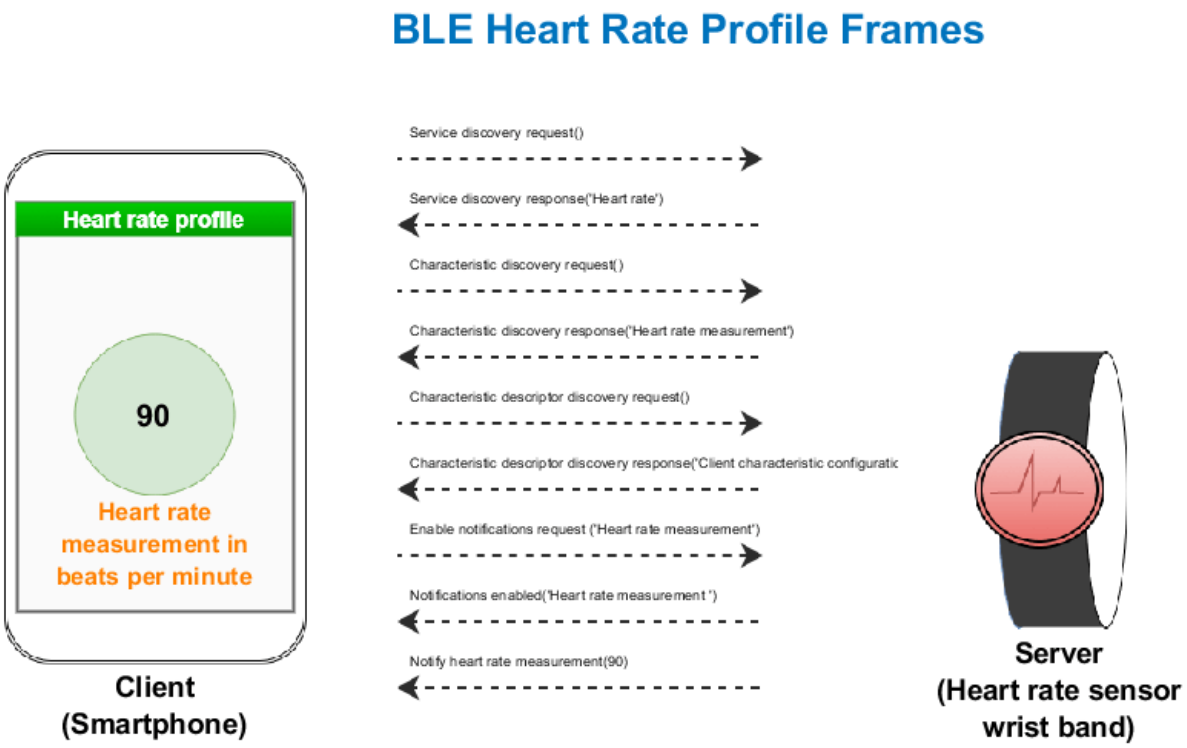
```
% Reset the random number generator seed.
rng default

% Measure heart rate value using sensor (generate a random number for heart
% rate measurement value).
heartRateMeasurementValue = randi([65 95]);
```

```
% Notify the heart rate measurement.
[gattServer, notificationPDU] = notifyHeartRateMeasurement(gattServer, ...
    heartRateMeasurementValue);

% Transmit the application data (|notificationPDU|) to the client through
% PHY.
[bleWaveform, pcapPackets{count}] = helperBLETransmitData(notificationPDU);
count = count+1;
```

Receive Server notifications at Client

```
% Decode the received BLE waveform and retrieve the application data.
receivedPDU = helperBLEDecodeData(bleWaveform);

% Decode received ATT PDU and generate response PDU, if applicable.
[~, notificationCfg] = receiveData(gattClient, receivedPDU);

fprintf("Received notification at the client:\n")

% Decode the received heart rate measurement characteristic value.
heartRateCharacteristicValue = helperBLEDecodeAttributeValue(...
    notificationCfg.AttributeValue, 'Heart rate measurement');
heartRateCharacteristicValue

heartRateMeasurementValue = heartRateCharacteristicValue.HeartRateValue;

% Visualize the BLE GATT Client-Server model.
helperBLEVisualizeHRPFrame(serviceDiscRespMsg, chrsticDescRespMsg, ...
    chrsticDescDiscRespMsg, enableNotificRespMsg, heartRateMeasurementValue);
```

```
Received notification at the client:

heartRateCharacteristicValue =

  helperBLEAttributeValueConfig with properties:

             AttributeType: 'Heart rate measurement'
       HeartRateValueFormat: 'UINT8'
        SensorContactStatus: 'Contact detected'
    EnergyExpendedFieldFlag: 'Present, Units: Kilo Joules'
        RRIntervalFieldFlag: 'Present'
             HeartRateValue: 90
             EnergyExpended: 100
                 RRInterval: 10

   Read-only properties:
    No properties.
```



Exporting to a PCAP File

This example uses `blePCAPWriter` object to export the generated PDUs to a file with .pcap extension or .pcapng extension. To analyze and visualize this file, use a third part packet analyzer such as Wireshark.

Create an object of type `blePCAPWriter` and specify the packet capture file name.

```
% Create the BLE PCAP Writer file object
pcapObj = blePCAPWriter("FileName", "bleHRP");
```

Use the `write` function to write all the BLE LL PDUs to a PCAP file. The constant `timestamp` specifies the capture time of a PDU. In this example, the capture time is same for all the PDUs.

```
timestamp = 124800; % timestamp (in microseconds)

% Write all the LL PDUs to the PCAP file
for idx = 1:numel(pcapPackets)
    write(pcapObj, pcapPackets{idx}, timestamp, "PacketFormat", "bits");
end

% Clear the object
clear pcapObj;
```

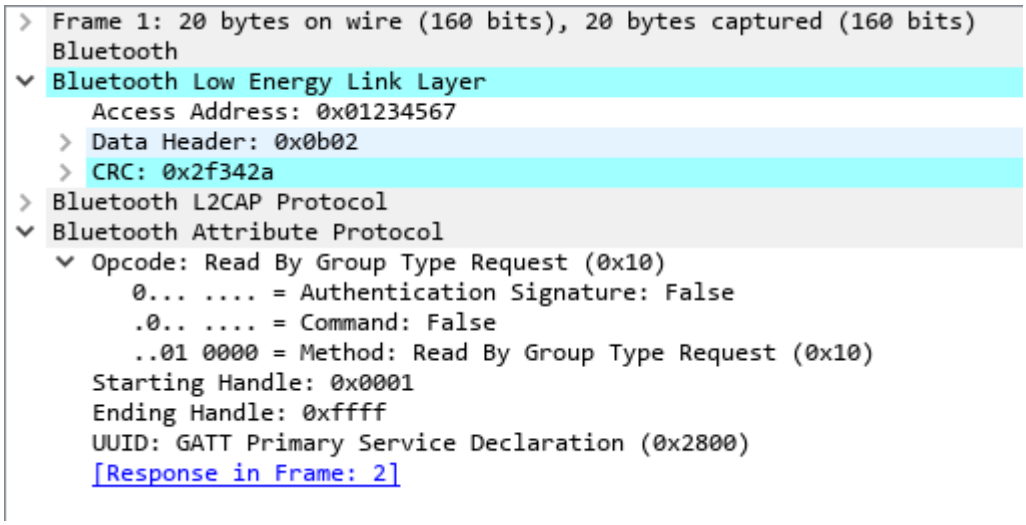
```
fprintf("Open generated pcap file 'bleHRP.pcap' in a protocol analyzer to view the generated fra
```

```
Open generated pcap file 'bleHRP.pcap' in a protocol analyzer to view the generated frames.
```

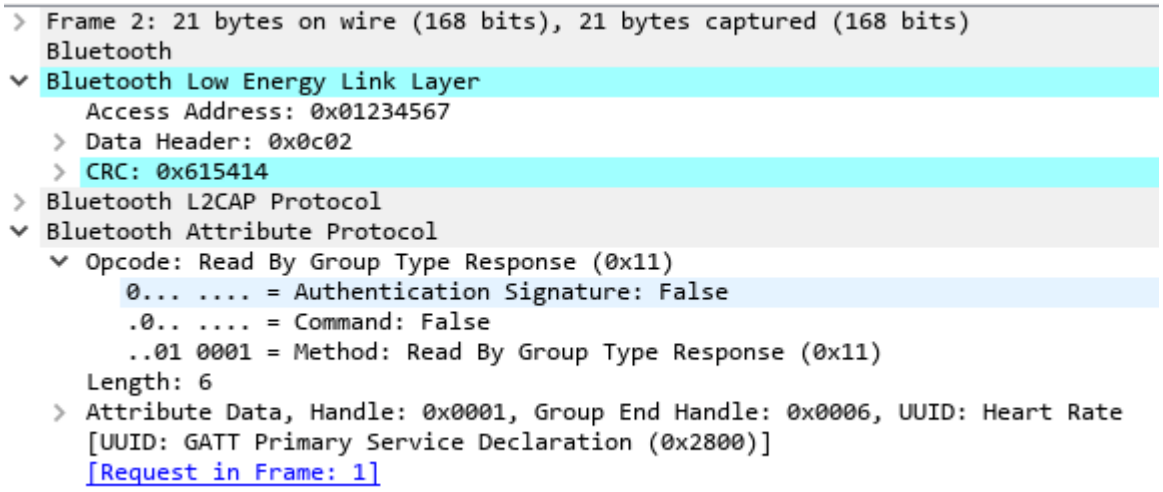
Visualization of the Generated ATT PDUs

Since the generated heart rate profile packets are compliant with the Bluetooth standard, you can open, analyze and visualize the PCAP file using a third party packet analyzer such as Wireshark [3]. The data shown in these figures uses the heart rate profile packets generated in this example.

- **Service discovery request**



- **Service discovery response**



- **Notifying heart rate measurement value**

```
> Frame 9: 22 bytes on wire (176 bits), 22 bytes captured (176 bits)
  Bluetooth
v Bluetooth Low Energy Link Layer
    Access Address: 0x01234567
  > Data Header: 0x0d02
  > CRC: 0xaa0ddb
> Bluetooth L2CAP Protocol
v Bluetooth Attribute Protocol
  v Opcode: Handle Value Notification (0x1b)
      0... .... = Authentication Signature: False
      .0.. .... = Command: False
      ..01 1011 = Method: Handle Value Notification (0x1b)
  v Handle: 0x0003 (Heart Rate: Heart Rate Measurement)
      [Service UUID: Heart Rate (0x180d)]
      [UUID: Heart Rate Measurement (0x2a37)]
  v Flags: 0x1e, RR Interval, Energy Expended, Sensor Support, Sensor Contact
      000. .... = Reserved: 0x0
      ...1 .... = RR Interval: True
      .... 1... = Energy Expended: True
      .... .1.. = Sensor Support: True
      .... ..1. = Sensor Contact: True
      .... ...0 = Value is UINT16: False
    Value: 90
    Energy Expended: 100
  v RR Intervals [count =  1]
      RR Interval: 10
```

Conclusion

This example demonstrated the modeling of BLE devices with Heart Rate Profile using the GATT client-server scenario as specified in the Bluetooth core specification [1]. You can use a packet analyzer to view the generated frames.

Appendix

The example uses these features:

- `bleATTPDUConfig`: Create configuration object for a BLE ATT PDU
- `bleATTPDU`: BLE ATT PDU generation
- `bleATTPDUDecode`: BLE ATT PDU decoding
- `blePCAPWriter`: Create BLE PCAP or PCAPNG file writer object

The example uses these helpers:

- helperBLEGATTClient: Provides methods to the Generic Attribute profile
- helperBLEGATTServer: Creates a GATT server object
- helperBLEAttributeValueConfig: Create configuration object for a BLE attribute value
- helperBLEGenerateAttributeValue: BLE attribute value generation
- helperBLEDecodeAttributeValue: BLE attribute value decoder
- helperBLEDecodeData: Decodes the received waveform and retrieve the application data
- helperBLEPrependAccessAddress: Prepends the link layer PDU with the access address

- helperBLETransmitData: Transmit application data by generating BLE waveform
- helperBluetoothID: Bluetooth identifiers and their names assigned by the Bluetooth Special Interest Group (SIG)
- helperBLEVisualizeHRPFrame: Visualize the Heart Rate Profile (HRP) frames exchanged in HRP Example
- helperBLEPlotHRPFrames: Plot the data frame exchange between the Heart rate profile server and client

Selected Bibliography

1 Bluetooth® Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed July 8, 2020. https://www.bluetooth.com/.

2 "Development/LibpcapFileFormat - The Wireshark Wiki." Accessed July 8, 2020. https://wiki.wireshark.org/Development/LibpcapFileFormat.

3 Group, The Tcpdump. "Tcpdump/Libpcap Public Repository." Accessed July 8, 2020. https://www.tcpdump.org.

See Also

More About

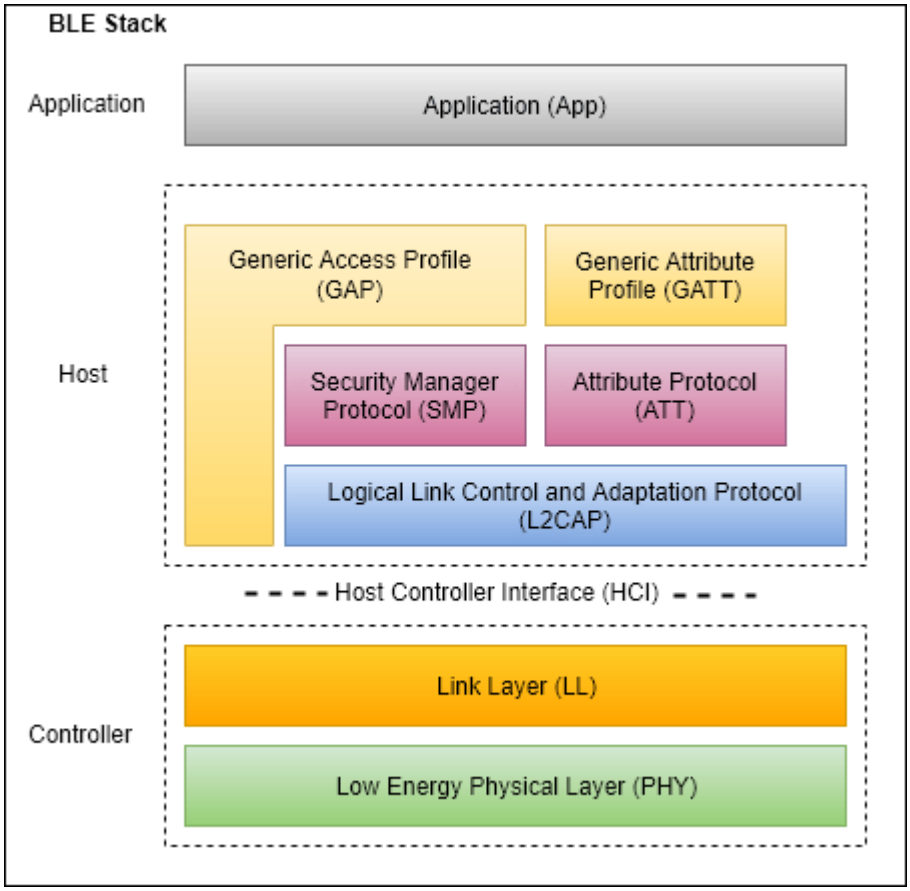
- “Bluetooth Protocol Stack” on page 13-7

BLE L2CAP Frame Generation and Decoding

This example shows how to generate and decode Bluetooth® Low Energy L2CAP frames using the Communications Toolbox™ Library for the Bluetooth® Protocol.

Background

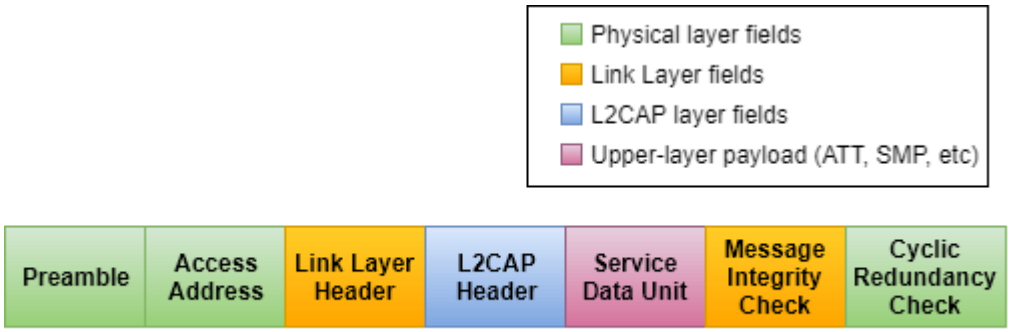
The Bluetooth Core Specification [1] includes a Low Energy (LE) version for low-rate wireless personal area networks, that is referred to as Bluetooth Low Energy (BLE) or Bluetooth Smart. The BLE stack consists of: Generic Attribute Profile (GATT), Attribute Protocol (ATT), Security Manager Protocol (SMP), Logical Link Control and Adaptation Protocol (L2CAP), Link layer and Physical layer. BLE was added to the standard for low energy devices generating small amounts of data, such as notification alerts used in such applications as home automation, health-care, fitness, and Internet of Things (IoT).



The L2CAP layer in BLE corresponds to the higher sub-layer i.e. Logical Link Control (LLC) of the Data Link Layer in the OSI reference model. The L2CAP is above the PHY and Link Layer of BLE. The BLE specification optimized and simplified the L2CAP when compared to classic Bluetooth.

L2CAP in BLE is responsible for: *(i)* logical connection establishment *(ii)* protocol multiplexing *(iii)* segmentation and reassembly *(iv)* flow control per 'dynamic' L2CAP channel.

The L2CAP layer adds an L2CAP basic header to the higher-layer payload and passes the Protocol Data Unit (PDU) to the Link Layer below it.



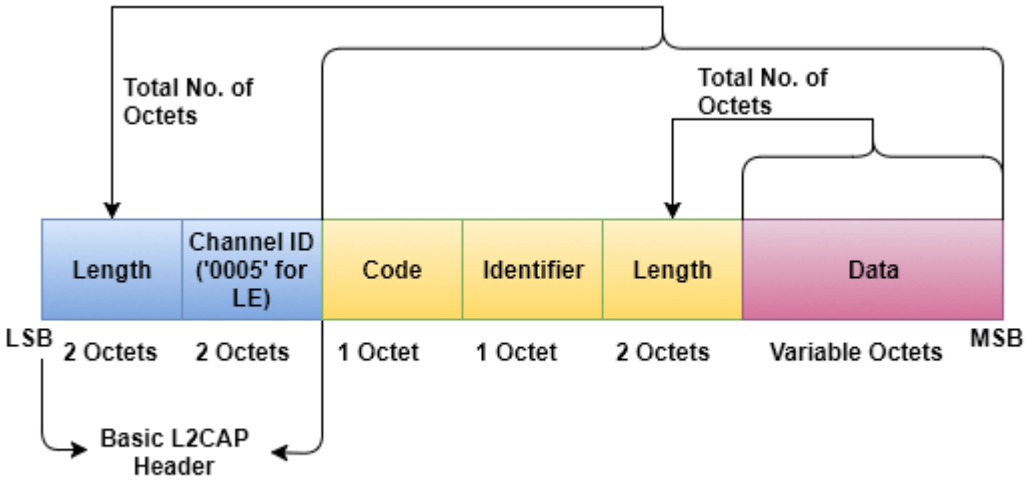
Format of BLE (Bluetooth Low Energy) frame

L2CAP Frames

L2CAP Frames consist of two sub-categories: **Data frames** and **Signaling frames**. There are different types of frames within these two categories of frames. The *Data frames* are again sub-categorized into *B-frame* (Basic information frame) and *LE-frame* (Low Energy information frame). Each frame type has its own format.

A **channel identifier (CID)** is the local name representing a logical channel endpoint on the device. For the protocols, such as the ATT and SMP, these channels are *fixed* by the Bluetooth Special Interest Group (SIG). For application specific profiles, such as Internet Protocol Support Profile (IPSP) and Object Transfer Profile (OTP), these channels are *dynamically* allocated.

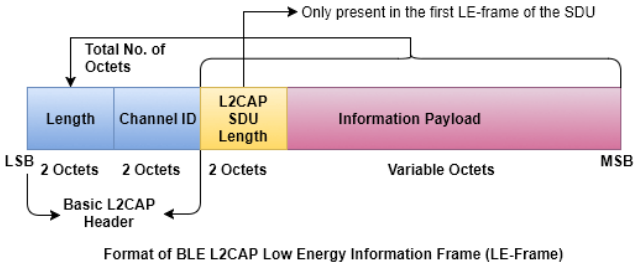
Signaling frames are used with a fixed logical channel called signaling channel (**'0005'**) and used for logical connection establishment between peer devices using the LE credit based flow control mechanism. These signaling frames are also used for updating the connection parameters (Slave latency, Connection timeout, Minimum connection interval and Maximum connection interval) when connection parameters request procedure is not supported in the Link Layer.



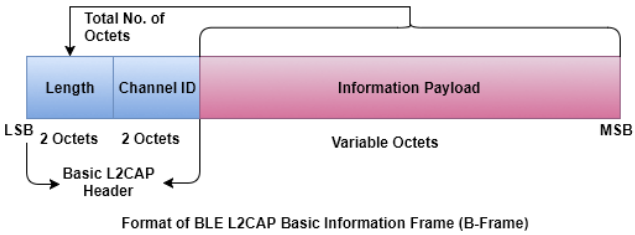
Format of BLE L2CAP Signaling Frame

Data frames (B-frames and LE-frames) carry the upper-layer payload as 'Information Payload' in its frame format. *B-Frames* are used to carry fixed channels (ATT and SMP with fixed logical channels

'0004' and **'0006'** respectively) payload. *LE-frames* are used to carry payload through dynamically created logical channels for application specific profiles, such as IPSP and OTP.



Format of BLE L2CAP Low Energy Information Frame (LE-Frame)



Format of BLE L2CAP Basic Information Frame (B-Frame)

This example illustrates generation and decoding of the following frames. For a list of other signaling frames supported, see the `CommandType` property of `bleL2CAPFrameConfig` object.

1. Flow control credit: This signaling frame is sent to create and configure an L2CAP logical channel between two devices.

2. B-frames over fixed channels (ATT, SMP, etc.): This frame is used for carrying fixed channels payload in basic L2CAP mode.

3. LE-frames over dynamic channels (profiles like IPSP, OTP, etc.): This frame is used for carrying dynamic channels payload in LE credit based flow control mode.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

L2CAP Frames Generation

You can use the `bleL2CAPFrame` function to generate an L2CAP frame. This function accepts a configuration object `bleL2CAPFrameConfig`. This object configures the fields required for generating an L2CAP frame.

Signaling frame generation

To generate a signaling frame, create a `bleL2CAPFrameConfig` object with `ChannelIdentifier` set to `'0005'`.

```
cfgL2CAP = bleL2CAPFrameConfig('ChannelIdentifier', '0005');
```

Configure the fields:

```
% Command type
cfgL2CAP.CommandType = 'Flow control credit';
% Source channel identifier
cfgL2CAP.SourceChannelIdentifier = '0041';
% LE credits
cfgL2CAP.Credits = 25
```

```
cfgL2CAP =

  bleL2CAPFrameConfig with properties:
```

```
          ChannelIdentifier: '0005'
                CommandType: 'Flow control credit'
             SignalIdentifier: '01'
    SourceChannelIdentifier: '0041'
                    Credits: 25

   Read-only properties:
    No properties.
```

Generate a 'Flow control credit' command.

```
sigFrame = bleL2CAPFrame(cfgL2CAP);
```

B-frame generation

To generate a B-frame (carrying ATT PDU), create a `bleL2CAPFrameConfig` object with `ChannelIdentifier` set to `'0004'` (ATT channel ID).

```
cfgL2CAP = bleL2CAPFrameConfig('ChannelIdentifier', '0004')
```

```
cfgL2CAP =

  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'

   Read-only properties:
    No properties.
```

A B-frame is used to transmit payload from the ATT upper-layer. A 5-byte ATT PDU is used as payload in this example.

```
payload = ['04';'01';'00';'FF';'FF'];
```

Generate an L2CAP B-frame using the payload and configuration.

```
bFrame = bleL2CAPFrame(cfgL2CAP, payload);
```

LE-frame generation

To generate an LE-frame, create a `bleL2CAPFrameConfig` object with `ChannelIdentifier` set to `'0035'`.

```
cfgL2CAP = bleL2CAPFrameConfig('ChannelIdentifier', '0035')
```

```
cfgL2CAP =

  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0035'

   Read-only properties:
    No properties.
```

An LE-frame is used to transmit the payload of dynamic channels. A 2-byte payload is used in this example.

```
payload = ['01';'02'];
```

Generate an L2CAP LE-frame using the payload and configuration.

```
leFrame = bleL2CAPFrame(cfgL2CAP, payload);
```

Decoding L2CAP Frames

You can use the `bleL2CAPFrameDecode` function to decode an L2CAP frame. This function outputs the following information:

1. `status`: An enumeration of type `blePacketDecodeStatus`, which indicates whether or not the L2CAP decoding was successful.
2. `cfgL2CAP`: An L2CAP frame configuration object of type `bleL2CAPFrameConfig`, which contains the decoded L2CAP properties.

This function accepts a BLE L2CAP frame as the input.

Decoding Signaling frame

```
[sigFrameDecodeStatus, cfgL2CAP] = bleL2CAPFrameDecode(sigFrame);

% Observe the outputs

% Decoding is successful
if strcmp(sigFrameDecodeStatus, 'Success')
    fprintf('L2CAP decoding status is: %s\n\n', sigFrameDecodeStatus);
    fprintf('Received L2CAP signaling frame configuration is:\n');
    cfgL2CAP
% Decoding failed
else
    fprintf('L2CAP decoding status is: %s\n', sigFrameDecodeStatus);
end
```

```
L2CAP decoding status is: Success

Received L2CAP signaling frame configuration is:

cfgL2CAP =

  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Flow control credit'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0041'
                    Credits: 25

   Read-only properties:
    No properties.
```

Decoding B-frame

```
[bFrameDecodeStatus, cfgL2CAP, payload] = bleL2CAPFrameDecode(bFrame);
```

```
% Observe the outputs

% Decoding is successful
if strcmp(bFrameDecodeStatus, 'Success')
    fprintf('L2CAP decoding status is: %s\n\n', bFrameDecodeStatus);
    fprintf('Received L2CAP B-frame configuration is:\n');
    cfgL2CAP
    fprintf('Payload carried by L2CAP B-frame is:\n');
    payload
% Decoding failed
else
    fprintf('L2CAP decoding status is: %s\n', bFrameDecodeStatus);
end
```

```
L2CAP decoding status is: Success

Received L2CAP B-frame configuration is:

cfgL2CAP =

  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'

   Read-only properties:
    No properties.

Payload carried by L2CAP B-frame is:

payload =

  5x2 char array

    '04'
    '01'
    '00'
    'FF'
    'FF'
```

Decoding LE-frame

```
[leFrameDecodeStatus, cfgL2CAP, payload] = bleL2CAPFrameDecode(leFrame);

% Observe the outputs

% Decoding is successful
if strcmp(leFrameDecodeStatus, 'Success')
    fprintf('L2CAP decoding status is: %s\n\n', leFrameDecodeStatus);
    fprintf('Received L2CAP LE-frame configuration is:\n');
    cfgL2CAP
    fprintf('Payload carried by L2CAP LE-frame is:\n');
    payload
% Decoding failed
else
    fprintf('L2CAP decoding status is: %s\n', leFrameDecodeStatus);
end
```

```
L2CAP decoding status is: Success

Received L2CAP LE-frame configuration is:

cfgL2CAP =

  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0035'

   Read-only properties:
    No properties.

Payload carried by L2CAP LE-frame is:

payload =

  2x2 char array

    '01'
    '02'
```

Exporting to a PCAP File

This example uses `blePCAPWriter` object to export the generated PDUs to a file with .pcap extension or .pcapng extension. To analyze and visualize this file, use a third part packet analyzer such as Wireshark.

The PCAP format expects L2CAP frame to be enclosed within Link Layer packet and also expects the generated packet to be prepended with the access address. The helperBLEPrependAccessAddress helper function prepends the access address to the generated packet. The following commands generate a PCAP file for the L2CAP frames generated in this example.

```
% Create a cell array of L2CAP frames
l2capFrames = {sigFrame, bFrame, leFrame};
llPackets = cell(1, numel(l2capFrames));
for i = 1:numel(llPackets)
    % Add Link Layer header to the generated L2CAP frame
    cfgLLData = bleLLDataChannelPDUConfig('LLID', 'Data (start fragment/complete)');
    llDataPDU = bleLLDataChannelPDU(cfgLLData, l2capFrames{i});
    % Prepend access address. A 4-byte access address is used in this example
    llPackets{i} = helperBLEPrependAccessAddress(llDataPDU, '01234567');
end
```

Export to a PCAP file

Create an object of type `blePCAPWriter` and specify the packet capture file name.

```
% Create the BLE PCAP Writer file object
pcapObj = blePCAPWriter("FileName", "BLEL2CAPFrames");
```

Use the `write` function to write all the BLE LL PDUs to a PCAP file. The constant `timestamp` specifies the capture time of a PDU. In this example, the capture time is same for all the PDUs.

```
timestamp = 124800; % timestamp (in microseconds)

% Write all the LL PDUs to the PCAP file
```

```
for idx = 1:numel(llPackets)
    write(pcapObj, llPackets{idx}, timestamp, "PacketFormat", "bits");
end

% Clear the object
clear pcapObj;
```

Visualization of the Generated L2CAP Frames

You can open the PCAP file containing the generated L2CAP frames in a packet analyzer. The L2CAP frames decoded by the packet analyzer match the standard compliant L2CAP frames generated by the Communications Toolbox™ Library for the Bluetooth Protocol. The captured analysis of the L2CAP frames is shown below.

- **Signaling frame (flow control credit)**

```
Frame 1: 21 bytes on wire (168 bits), 21 bytes captured (168 bits)
Bluetooth
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
    Length: 8
    CID: Low Energy L2CAP Signaling Channel (0x0005)
  ⌄ Command: LE Flow Control Credit
        Command Code: LE Flow Control Credit (0x16)
        Command Identifier: 0x01
        Command Length: 4
        CID: Dynamically Allocated Channel (0x0041)
        Credits: 25
```

- **B-frame (carrying ATT PDU)**

```
Frame 2: 18 bytes on wire (144 bits), 18 bytes captured (144 bits)
Bluetooth
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
    Length: 5
    CID: Attribute Protocol (0x0004)
Bluetooth Attribute Protocol
  ⌄ Opcode: Find Information Request (0x04)
        0... .... = Authentication Signature: False
        .0.. .... = Command: False
        ..00 0100 = Method: Find Information Request (0x04)
    Starting Handle: 0x0001
    Ending Handle: 0xffff
```

- **LE-frame (carrying dynamic channel payload)**

```
Frame 3: 17 bytes on wire (136 bits), 17 bytes captured (136 bits)
Bluetooth
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
    Length: 4
    CID: Reserved (0x0035)
    Payload: 02000102
```

Conclusion

This example demonstrated generation and decoding of L2CAP frames specified in the Bluetooth [1] standard. You can use a packet analyzer to view the generated L2CAP frames.

Appendix

The example uses these features:

- `bleL2CAPFrameConfig`: Create configuration object for a BLE L2CAP frame
- `bleL2CAPFrame`: BLE L2CAP frame generation
- `bleL2CAPFrameDecode`: BLE L2CAP frame decoder
- `blePCAPWriter`: Create BLE PCAP or PCAPNG file writer object

The example uses this helper:

- helperBLEPrependAccessAddress: Prepends the link layer PDU with the access address

Selected Bibliography

1 Bluetooth® Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed July 8, 2020. https://www.bluetooth.com/.

2 "Development/LibpcapFileFormat - The Wireshark Wiki." Accessed July 8, 2020. https://wiki.wireshark.org/Development/LibpcapFileFormat.

3 Group, The Tcpdump. "Tcpdump/Libpcap Public Repository." Accessed July 8, 2020. https://www.tcpdump.org.

See Also

More About

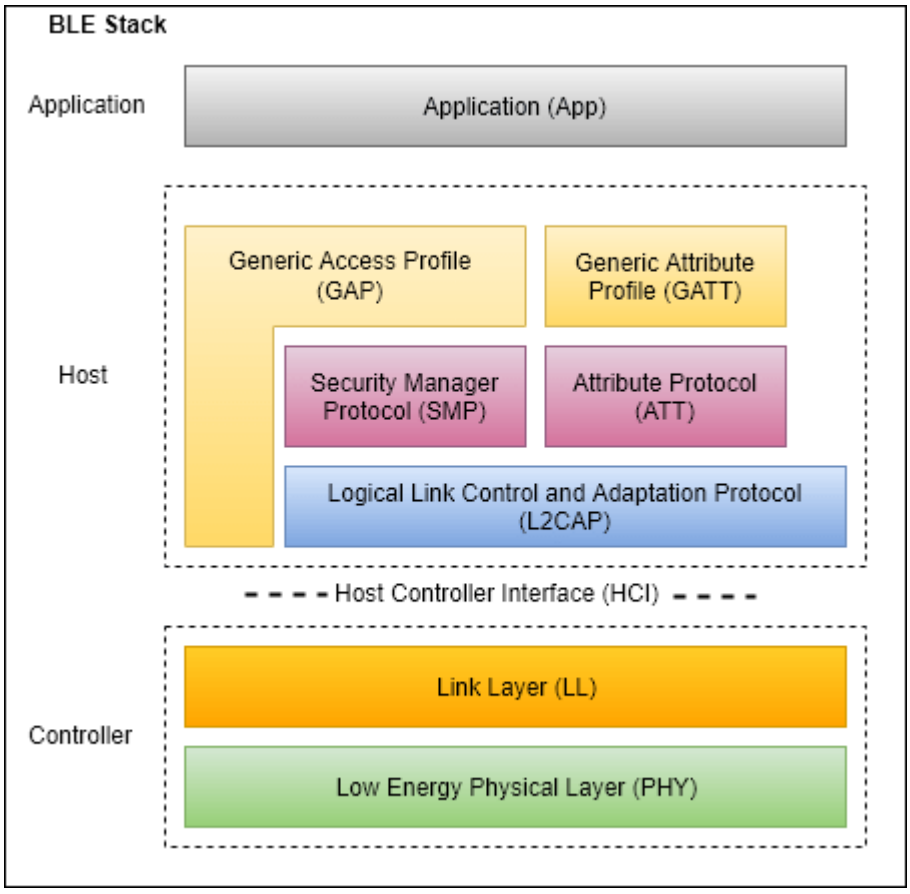
- “Bluetooth Protocol Stack” on page 13-7

BLE Link Layer Packet Generation and Decoding

This example shows how to generate and decode Bluetooth® Low Energy (BLE) link layer packets using the Communications Toolbox™ Library for the Bluetooth® Protocol.

Background

The Bluetooth Core Specification [1] includes a Low Energy version for low-rate wireless personal area networks, that is referred to as Bluetooth Low Energy (BLE) or Bluetooth Smart. The BLE stack consists of: Generic Attribute Profile (GATT), Attribute Protocol (ATT), Security Manager Protocol (SMP), Logical Link Control and Adaptation Protocol (L2CAP), link layer (LL) and physical layer. BLE was added to the standard for low energy devices generating small amounts of data, such as notification alerts used in such applications as home automation, health-care, fitness, and Internet of Things (IoT).



Packet Formats

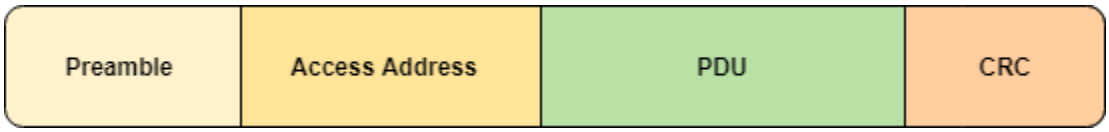
Bluetooth core specification [1] defines two kinds of PHYs for BLE. Each PHY has its own packet format.

(i) Uncoded PHYs (1 Mbps and 2 Mbps)

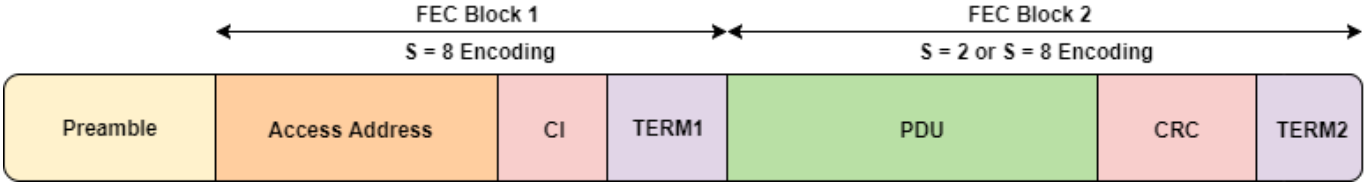
(ii) Coded PHYs (125 Kbps and 500 Kbps)

Coded PHYs use Forward Error Correction (FEC) encoding (with coding scheme S = 8 or S = 2) for the packets. The figures show the uncoded and coded PHY packet formats.

Format of LE Air Interface Packet for Uncoded PHY



Format of LE Air Interface Packet for Coded PHY



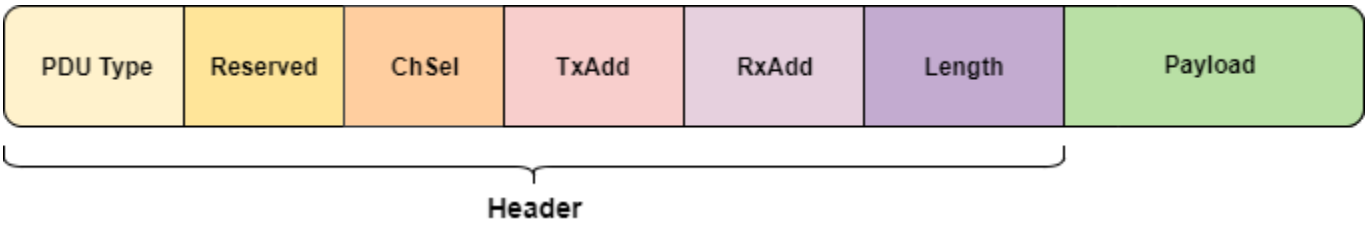
The Communications Toolbox™ Library for the Bluetooth Protocol generates LL packets that consist of Protocol Data Unit (PDU) and the Cyclic Redundancy Check (CRC) shown in the PHY packet.

BLE classifies 40 RF channels into three advertising channels (Channel indices: 37, 38, 39) and thirty-seven data channels (Channel indices: 0 to 36). BLE link layer defines two categories of PDUs, advertising channel PDUs and data channel PDUs. There are different PDU types within these two categories of PDUs. The access address field in the air interface packet format differentiates between a data channel PDU and an advertising channel PDU. Each category of PDU has its own format.

Advertising Channel PDUs

The advertising channel PDUs (see Section 2.3, Part-B, Vol-6 in [1]) are used before a LL connection is created. These PDUs are transmitted only on the advertising channels and used in establishing the LL connection. The advertising channel PDU has a 16-bit header and a variable size payload.

The advertising channel PDU has the following packet format:



This example illustrates generation and decoding of advertising indication PDU. For a list of other advertising channel PDUs supported, see the `PDUType` property of `bleLLAdvertisingChannelPDUConfig` object.

Advertising indication: The advertising indication PDU is used when a device wants to advertise itself. This PDU contains the advertising data related to the application profile of the device.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Advertising Channel PDUs Generation

You can use the `bleLLAdvertisingChannelPDU` function to generate an advertising channel PDU. This function accepts a configuration object `bleLLAdvertisingChannelPDUConfig`. This object configures the fields required for generating an advertising channel PDU.

Advertising Indication Generation

To generate an 'Advertising indication' PDU, create a `bleLLAdvertisingChannelPDUConfig` object with `PDUType` set to `'Advertising indication'`.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig('PDUType', ...
    'Advertising indication');
```

Configure the fields:

```
% Advertiser address
cfgLLAdv.AdvertiserAddress = '012345ABCDEF';
% Advertising data
cfgLLAdv.AdvertisingData = '0201060D094261747465727920563 12E30'
```

```
cfgLLAdv =

  bleLLAdvertisingChannelPDUConfig with properties:

                 PDUType: 'Advertising indication'
        ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '012345ABCDEF'
          AdvertisingData: [17x2 char]

   Read-only properties:
    No properties.
```

Generate an 'Advertising indication' PDU.

```
llAdvPDU = bleLLAdvertisingChannelPDU(cfgLLAdv);
```

Decoding Advertising Channel PDUs

You can use the `bleLLAdvertisingChannelPDUDecode` function to decode an advertising channel PDU. This function outputs the following information:

1. `status`: An enumeration of type `blePacketDecodeStatus`, specifying whether the LL decoding was successful.
2. `cfgLLAdv`: A LL advertising channel PDU configuration object of type `bleLLAdvertisingChannelPDUConfig`, which contains the decoded LL properties.

Provide the advertising channel PDU and an optional name-value pair specifying the format of the input data PDU to the bleLLAdvertisingChannelPDUDecode function. Default input format is 'bits'.

Decoding Advertising Indication

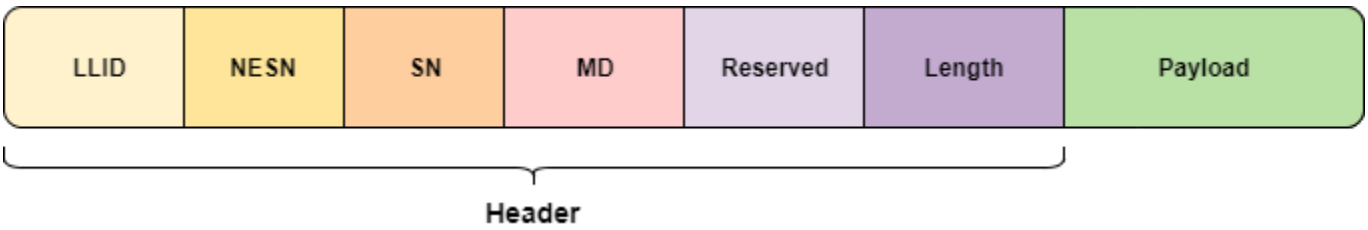
```
[llAdvDecodeStatus, cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(llAdvPDU);

% Observe the outputs

% Decoding is successful
if strcmp(llAdvDecodeStatus, 'Success')
    fprintf('Link layer decoding status is: %s\n\n', llAdvDecodeStatus);
    fprintf('Received Advertising channel PDU configuration is:\n');
    cfgLLAdv
% Decoding failed
else
    fprintf('Link layer decoding status is: %s\n', llAdvDecodeStatus);
end
```

```
Link layer decoding status is: Success

Received Advertising channel PDU configuration is:

cfgLLAdv =

  bleLLAdvertisingChannelPDUConfig with properties:

                 PDUType: 'Advertising indication'
       ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '012345ABCDEF'
          AdvertisingData: [17x2 char]

   Read-only properties:
    No properties.
```

Data Channel PDUs

The data channel PDUs (see Section 2.4, Part-B, Vol-6 in [1]) are used after a LL connection is created. The data channel PDUs consist of two sub-categories: **LL data PDUs** and **LL control PDUs**. The LL control PDUs are used for managing the LL connection and the LL data PDUs are used to carry the upper-layer data. The data channel PDU has a 16-bit header and a variable size payload.

The data channel PDUs have the following packet format:



This example illustrates generation and decoding of the following PDUs. For a list of other control PDU types and data PDU types supported see `Opcode` and `LLID` properties of `bleLLControlPDUConfig` and `bleLLDataChannelPDUConfig` objects, respectively.

1 *Channel map indication*: This LL control PDU is used to update the channel map at the peer device. This PDU contains the updated channel map indicating good and bad channels.

1 *Data (start fragment/complete)*: This LL data PDU is used to carry L2CAP data to the peer device.

Data Channel PDUs Generation

You can use the `bleLLDataChannelPDU` function to generate a data channel PDU. This function accepts a configuration object `bleLLDataChannelPDUConfig`, which configures the fields required for generating a data channel PDU. The `bleLLControlPDUConfig` is a sub-configuration object within `bleLLDataChannelPDUConfig`, control PDU payload fields are populated using the settings of this configuration object.

Data channel PDUs use the CRC initialization value obtained in the 'Connection indication' packet. The CRC initialization value used in the generation and decoding of packets.

```
% CRC initialization value
crcInit = 'ED321C';
```

LL Data PDU Generation

To generate a data PDU, create a `bleLLDataChannelPDUConfig` object with `LLID` set to `'Data (start fragment/complete)'`.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', ...
    'Data (start fragment/complete)');
```

Configure the fields:

```
% CRC initialization value
cfgLLData.CRCInitialization = crcInit;
% Sequence number
cfgLLData.SequenceNumber = 0;
% Next expected sequence number
cfgLLData.NESN = 1
```

```
cfgLLData =

  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (start fragment/complete)'
                NESN: 1
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: 'ED321C'

   Read-only properties:
    No properties.
```

A data PDU is used to transmit a payload from upper-layer. A 18-byte payload is used in this example.

```
% Payload
payload = '0E00050014010A001F004000170017000000';
```

Generate a data PDU using payload and configuration.

```
llDataPDU = bleLLDataChannelPDU(cfgLLData, payload);
```

LL Control PDU Generation

To generate a control PDU, create a `bleLLDataChannelPDUConfig` object with `LLID` set to `'Control'`.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', 'Control');
```

Configure the fields:

```
% CRC initialization value
cfgLLData.CRCInitialization = crcInit
```

```
cfgLLData =

  bleLLDataChannelPDUConfig with properties:

                LLID: 'Control'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: 'ED321C'
        ControlConfig: [1x1 bleLLControlPDUConfig]

   Read-only properties:
    No properties.
```

You can configure the contents of an LL control PDU using `bleLLControlPDUConfig`.

Create a control PDU configuration object with `Opcode` set to `'Channel map indication'`.

```
cfgControl = bleLLControlPDUConfig('Opcode', 'Channel map indication');
```

Configure the fields:

```
% Used channels
cfgControl.UsedChannels = [9, 10, 12, 24, 28, 32];
% Connection event instant
cfgControl.Instant = 245
```

```
cfgControl =

  bleLLControlPDUConfig with properties:

         Opcode: 'Channel map indication'
        Instant: 245
    UsedChannels: [9 10 12 24 28 32]

   Read-only properties:
    No properties.
```

Assign the updated control PDU configuration object to the `ControlConfig` property in the data channel PDU configuration object.

```
% Update the data channel PDU configuration
cfgLLData.ControlConfig = cfgControl;
```

Generate a control PDU with the updated configuration.

```
llControlPDU = bleLLDataChannelPDU(cfgLLData);
```

Decoding Data Channel PDUs

You can use the `bleLLDataChannelPDUDecode` function to decode a data channel PDU. This function outputs the following information:

1. `status`: An enumeration of type `blePacketDecodeStatus`, specifying whether the LL decoding was successful.
2. `cfgLLData`: An LL data channel PDU configuration object of type `bleLLDataChannelPDUConfig`, which contains the decoded LL properties.
3. `payload`: An n-by-2 character array representing the upper-layer payload carried by LL data PDUs.

Provide the data channel PDU, CRC initialization value and an optional name-value pair specifying the format of the input data PDU to the `bleLLDataChannelPDUDecode` function. Default input format is 'bits'.

Decoding LL Data PDU

```
[llDataDecodeStatus, cfgLLData, payload] = ...
    bleLLDataChannelPDUDecode(llDataPDU, crcInit);

% Observe the outputs

% Decoding is successful
if strcmp(llDataDecodeStatus, 'Success')
    fprintf('Link layer decoding status is: %s\n\n', llDataDecodeStatus);
    fprintf('Received Data channel PDU configuration is:\n');
    cfgLLData
    fprintf('Size of the received upper-layer payload is: %d\n', ...
        numel(payload)/2);
% Decoding failed
else
    fprintf('Link layer decoding status is: %s\n', llDataDecodeStatus);
end
```

```
Link layer decoding status is: Success

Received Data channel PDU configuration is:

cfgLLData =

  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (start fragment/complete)'
                NESN: 1
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: '012345'

   Read-only properties:
    No properties.

Size of the received upper-layer payload is: 18
```

Decoding LL Control PDU

```
[llControlDecodeStatus, cfgLLData] = ...
    bleLLDataChannelPDUDecode(llControlPDU, crcInit);

% Observe the outputs

% Decoding is successful
if strcmp(llControlDecodeStatus, 'Success')
    fprintf('Link layer decoding status is: %s\n\n', llControlDecodeStatus);
    fprintf('Received Data channel PDU configuration is:\n');
    cfgLLData
    fprintf('Received control PDU configuration is:\n');
    cfgControl = cfgLLData.ControlConfig
% Decoding failed
else
    fprintf('Link layer decoding status is: %s\n', llControlDecodeStatus);
end
```

```
Link layer decoding status is: Success

Received Data channel PDU configuration is:

cfgLLData =

  bleLLDataChannelPDUConfig with properties:

                LLID: 'Control'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: '012345'
        ControlConfig: [1x1 bleLLControlPDUConfig]

   Read-only properties:
    No properties.

Received control PDU configuration is:

cfgControl =

  bleLLControlPDUConfig with properties:

         Opcode: 'Channel map indication'
        Instant: 245
    UsedChannels: [9 10 12 24 28 32]

   Read-only properties:
    No properties.
```

Exporting to a PCAP File

This example uses `blePCAPWriter` object to export the generated PDUs to a file with .pcap extension or .pcapng extension. To analyze and visualize this file, use a third part packet analyzer such as Wireshark.

Prepend access address

The PCAP format expects access address to be prepended to the LL packet. The helper function helperBLEPrependAccessAddress prepends access address to the generated LL packet.

```
% Advertising channel PDUs use the default access address
advAccessAddress = '8E89BED6';
% Data channel PDUs use the access address obtained from 'Connection
% indication' packet. A random access address is used for this example
connAccessAddress = 'E213BC42';
% Prepend access address
llPkts{1} = helperBLEPrependAccessAddress(llAdvPDU, advAccessAddress);
llPkts{2} = helperBLEPrependAccessAddress(llDataPDU, connAccessAddress);
llPkts{3} = helperBLEPrependAccessAddress(llControlPDU, connAccessAddress);
```

Export to a PCAP file

Create an object of type `blePCAPWriter` and specify the packet capture file name.

```
% Create the BLE PCAP Writer file object
pcapObj = blePCAPWriter("FileName", "bleLLPackets");
```

Use the `write` function to write all the BLE LL PDUs to a PCAP file. The constant `timestamp` specifies the capture time of a PDU. In this example, the capture time is same for all the PDUs.

```
timestamp = 124800; % timestamp (in microseconds)

% Write all the LL PDUs to the PCAP file
for idx = 1:numel(llPkts)
    write(pcapObj, llPkts{idx}, timestamp, 'PacketFormat', 'bits');
end

% Clear the object
clear pcapObj;
```

Visualization of the Generated Link Layer Packets

You can open the PCAP file containing the generated LL packets in a packet analyzer. The packets decoded by the packet analyzer match the standard compliant LL packets generated by the Communications Toolbox™ Library for the Bluetooth Protocol. The captured analysis of the packets is shown below.

- **Advertising indication**

```
Frame 1: 32 bytes on wire (256 bits), 32 bytes captured (256 bits)
Bluetooth
   [Source: SonyMobi_ab:cd:ef (01:23:45:ab:cd:ef)]
   [Destination: Broadcast (ff:ff:ff:ff:ff:ff)]
Bluetooth Low Energy Link Layer
   Access Address: 0x8e89bed6
 ⌄ Packet Header: 0x1740 (PDU Type: ADV_IND, ChSel: #1, TxAdd: Random)
      .... 0000 = PDU Type: ADV_IND (0x0)
      ...0 .... = RFU: 0
      ..0. .... = Channel Selection Algorithm: #1
      .1.. .... = Tx Address: Random
      0... .... = Reserved: False
      Length: 23
   Advertising Address: SonyMobi_ab:cd:ef (01:23:45:ab:cd:ef)
 ⌄ Advertising Data
    › Flags
    › Device Name: Battery V1.0
   CRC: 0xeebc04
```

- **LL data PDU (carrying L2CAP payload)**

```
Frame 2: 27 bytes on wire (216 bits), 27 bytes captured (216 bits)
Bluetooth
Bluetooth Low Energy Link Layer
   Access Address: 0xe213bc42
 ⌄ Data Header: 0x1206
      .... ..10 = LLID: Start of an L2CAP message or a complete L2CAP message with no fragmentation (0x2)
      .... .1.. = Next Expected Sequence Number: 1
      .... 0... = Sequence Number: 0
      ...0 .... = More Data: False
      000. .... = RFU: 0
      Length: 18
 › CRC: 0xbee203
Bluetooth L2CAP Protocol
   Length: 14
   CID: Low Energy L2CAP Signaling Channel (0x0005)
 ⌄ Command: LE Credit Based Connection Request
      Command Code: LE Credit Based Connection Request (0x14)
      Command Identifier: 0x01
      Command Length: 10
      LE PSM: Fixed, SIG Assigned (0x001f)
      [PSM: ATT (0x001f)]
      Source CID: Dynamically Allocated Channel (0x0040)
      MTU: 23
      MPS: 23
      Initial Credits: 0
```

- **LL control PDU (channel map indication)**

```
Frame 3: 17 bytes on wire (136 bits), 17 bytes captured (136 bits)
Bluetooth
Bluetooth Low Energy Link Layer
   Access Address: 0xe213bc42
 ⌄ Data Header: 0x0803
      .... ..11 = LLID: Control PDU (0x3)
      .... .0.. = Next Expected Sequence Number: 0
      .... 0... = Sequence Number: 0
      ...0 .... = More Data: False
      000. .... = RFU: 0
      Length: 8
   Control Opcode: LL_CHANNEL_MAP_REQ (0x01)
 ⌄ Channel Map: 0016001101
      .... ...0 = RF Channel 1 (2404 MHz - Data - 0): False
      .... ..0. = RF Channel 2 (2406 MHz - Data - 1): False
      .... .0.. = RF Channel 3 (2408 MHz - Data - 2): False
      .... 0... = RF Channel 4 (2410 MHz - Data - 3): False
      ...0 .... = RF Channel 5 (2412 MHz - Data - 4): False
      ..0. .... = RF Channel 6 (2414 MHz - Data - 5): False
      .0.. .... = RF Channel 7 (2416 MHz - Data - 6): False
      0... .... = RF Channel 8 (2418 MHz - Data - 7): False
      .... ...0 = RF Channel 9 (2420 MHz - Data - 8): False
      .... ..1. = RF Channel 10 (2422 MHz - Data - 9): True
      .... .1.. = RF Channel 11 (2424 MHz - Data - 10): True
      .... 0... = RF Channel 13 (2428 MHz - Data - 11): False
      ...1 .... = RF Channel 14 (2430 MHz - Data - 12): True
      ..0. .... = RF Channel 15 (2432 MHz - Data - 13): False
      .0.. .... = RF Channel 16 (2434 MHz - Data - 14): False
      0... .... = RF Channel 17 (2436 MHz - Data - 15): False
      .... ...0 = RF Channel 18 (2438 MHz - Data - 16): False
      .... ..0. = RF Channel 19 (2440 MHz - Data - 17): False
      .... .0.. = RF Channel 20 (2442 MHz - Data - 18): False
      .... 0... = RF Channel 21 (2444 MHz - Data - 19): False
      ...0 .... = RF Channel 22 (2446 MHz - Data - 20): False
      ..0. .... = RF Channel 23 (2448 MHz - Data - 21): False
      .0.. .... = RF Channel 24 (2450 MHz - Data - 22): False
      0... .... = RF Channel 25 (2452 MHz - Data - 23): False
      .... ...1 = RF Channel 26 (2454 MHz - Data - 24): True
      .... ..0. = RF Channel 27 (2456 MHz - Data - 25): False
      .... .0.. = RF Channel 28 (2458 MHz - Data - 26): False
      .... 0... = RF Channel 29 (2460 MHz - Data - 27): False
      ...1 .... = RF Channel 30 (2462 MHz - Data - 28): True
      ..0. .... = RF Channel 31 (2464 MHz - Data - 29): False
      .0.. .... = RF Channel 32 (2466 MHz - Data - 30): False
      0... .... = RF Channel 33 (2468 MHz - Data - 31): False
      .... ...1 = RF Channel 34 (2470 MHz - Data - 32): True
      .... ..0. = RF Channel 35 (2472 MHz - Data - 33): False
      .... .0.. = RF Channel 36 (2474 MHz - Data - 34): False
      .... 0... = RF Channel 37 (2476 MHz - Data - 35): False
      ...0 .... = RF Channel 38 (2478 MHz - Data - 36): False
      ..0. .... = RF Channel 0 (2402 MHz - Reserved for Advertising - 37): False
      .0.. .... = RF Channel 12 (2426 MHz - Reserved for Advertising - 38): False
      0... .... = RF Channel 39 (2480 MHz - Reserved for Advertising - 39): False
   Instant: 245
 › CRC: 0x563473
```

Conclusion

This example demonstrated generation and decoding of LL packets specified in the Bluetooth standard [1]. You can use a packet analyzer to view the generated LL packets.

Appendix

The example uses these feature:

- `bleLLAdvertisingChannelPDU`: Generate LL advertising channel PDU
- `bleLLAdvertisingChannelPDUDecode`: Decode LL advertising channel PDU
- `bleLLAdvertisingChannelPDUConfig`: Create a configuration object for generation and decoding of LL advertising channel PDU
- `bleLLDataChannelPDU`: Generate LL data channel PDU
- `bleLLDataChannelPDUDecode`: Decode LL data channel PDU
- `bleLLDataChannelPDUConfig`: Create a configuration object for generation and decoding of LL data channel PDU
- `bleLLControlPDUConfig`: Create a sub-configuration object used in generation and decoding of data channel PDU
- `blePCAPWriter`: Create BLE PCAP or PCAPNG file writer object

This example uses this helper:

- helperBLEPrependAccessAddress: Prepends the link layer PDU with the access address

Selected Bibliography

1. Bluetooth® Technology Website. "Bluetooth Technology Website | The Official Website of Bluetooth Technology." Accessed July 8, 2020. https://www.bluetooth.com/.
2. "Development/LibpcapFileFormat - The Wireshark Wiki." Accessed July 8, 2020. https://wiki.wireshark.org/Development/LibpcapFileFormat.
3. Group, The Tcpdump. "Tcpdump/Libpcap Public Repository." Accessed July 8, 2020. https://www.tcpdump.org.

See Also

More About

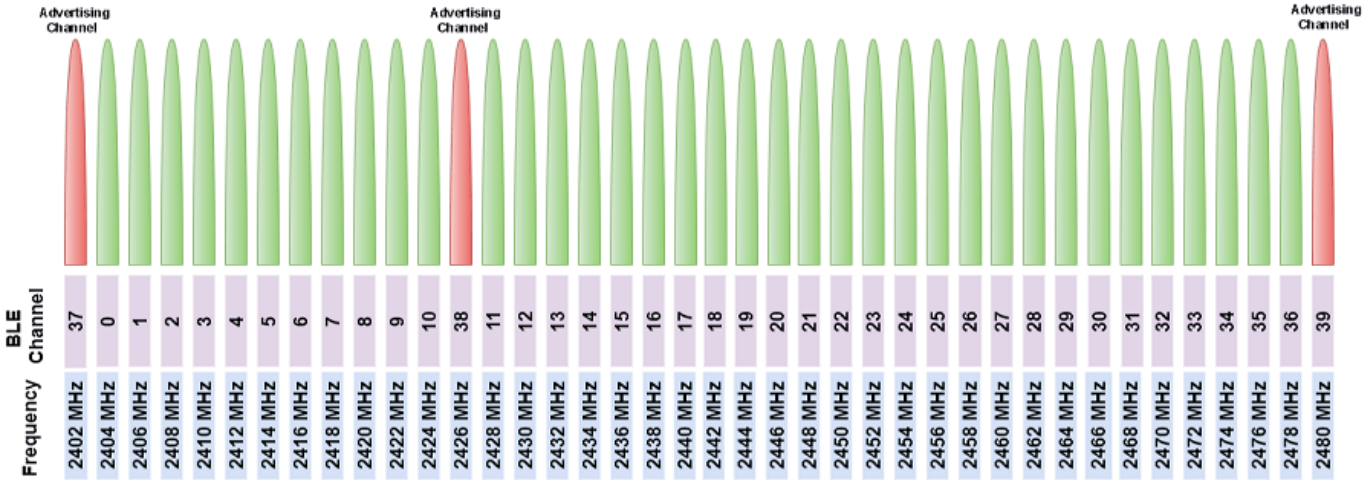
- “Bluetooth Protocol Stack” on page 13-7

Bluetooth Low Energy Waveform Generation and Visualization

This example shows how the Communications Toolbox™ Library for the Bluetooth® Protocol can be used to generate waveforms for different modes of Bluetooth Low Energy (BLE) physical layer (PHY) [1].

Background

Bluetooth special interest group (SIG) introduced BLE for low power short range communications. BLE devices operate in the globally unlicensed industrial, scientific and medical (ISM) band in the frequency range 2.4 GHz to 2.485 GHz. BLE specifies a channel spacing of 2 MHz, which results in 40 RF channels as shown in the figure below. The BLE standard [1] specifies the **Link** layer which includes both **PHY** and **MAC** layers. BLE finds applications in transfer of files such as images and MP3 between mobile phones, home automation and internet of things (IoT) trend.



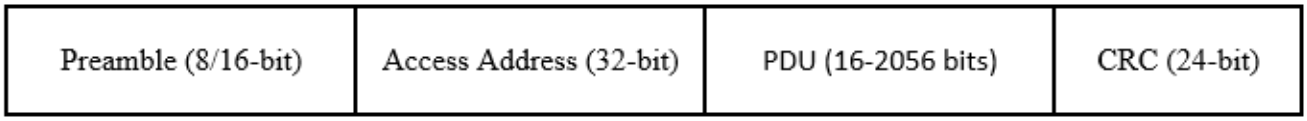
The Bluetooth standard [1] specifies the following physical layer modes:

- **LE1M** - Uncoded PHY with data rate of 1 Mbps
- **LE2M** - Uncoded PHY with data rate of 2 Mbps
- **LE500K** - Coded PHY with data rate of 500 Kbps
- **LE125K** - Coded PHY with data rate of 125 Kbps

The air interface packet formats for these modes include the following fields:

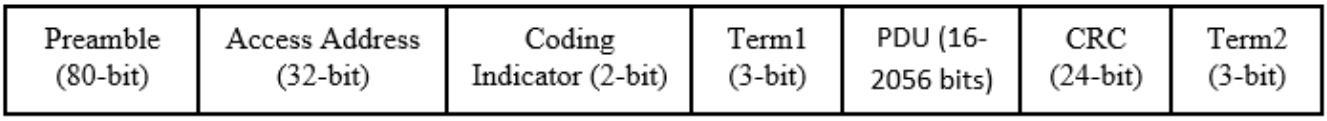
- **Preamble**: The preamble depends on which PHY mode is used. LE1M mode uses an 8-bit sequence of alternate zeros and ones, '01010101'. LE2M uses a 16-bit sequence of alternate zeros and ones, '0101...'. LE500K and LE125K modes use an 80-bit sequence of zeros and ones obtained by repeating '00111100' ten times.
- **Access Address**: Specifies the connection address shared between two BLE devices using a 32-bit sequence.
- **Coding Indicator**: 2-bit sequence used for differentiating two coded modes (LE125K, LE500K).
- **Payload**: Input message bits including both PDU and CRC. The maximum message size is 2080 bits.
- **Termination Fields**: Two 3-bit vectors of zeros, used in Forward Error Correction encoding. The termination fields are present for coded modes (LE500K and LE125K) only.

Packet format for uncoded PHY (LE1M and LE2M) modes is shown in the figure below:



BLE Uncoded PHY Packet Format

Packet format for coded PHY (LE500K and LE125K) modes is shown in the figure below:



BLE Coded PHY Packet Format

Introduction

This example shows how to generate BLE waveforms for all the physical layer modes as per the Bluetooth specification [1]. The generated BLE waveforms are visualized in both time-domain and frequency-domain using time scope and spectrum analyzer respectively.

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Initialize Parameters for Waveform Generation

```
% Specify the input parameters for generating BLE waveform
numPackets = 10;    % Number of packets to generate
sps = 16;           % Samples per symbol
messageLen = 2000;  % Length of message in bits
phyMode = 'LE1M';   % Select one mode from the set {'LE1M','LE2M','LE500K','LE125K'};
channelBW = 2e6;    % Channel spacing (Hz) as per standard
% Define symbol rate based on the PHY mode
if any(strcmp(phyMode,{'LE1M','LE500K','LE125K'}))
    symbolRate = 1e6;
else
    symbolRate = 2e6;
end
```

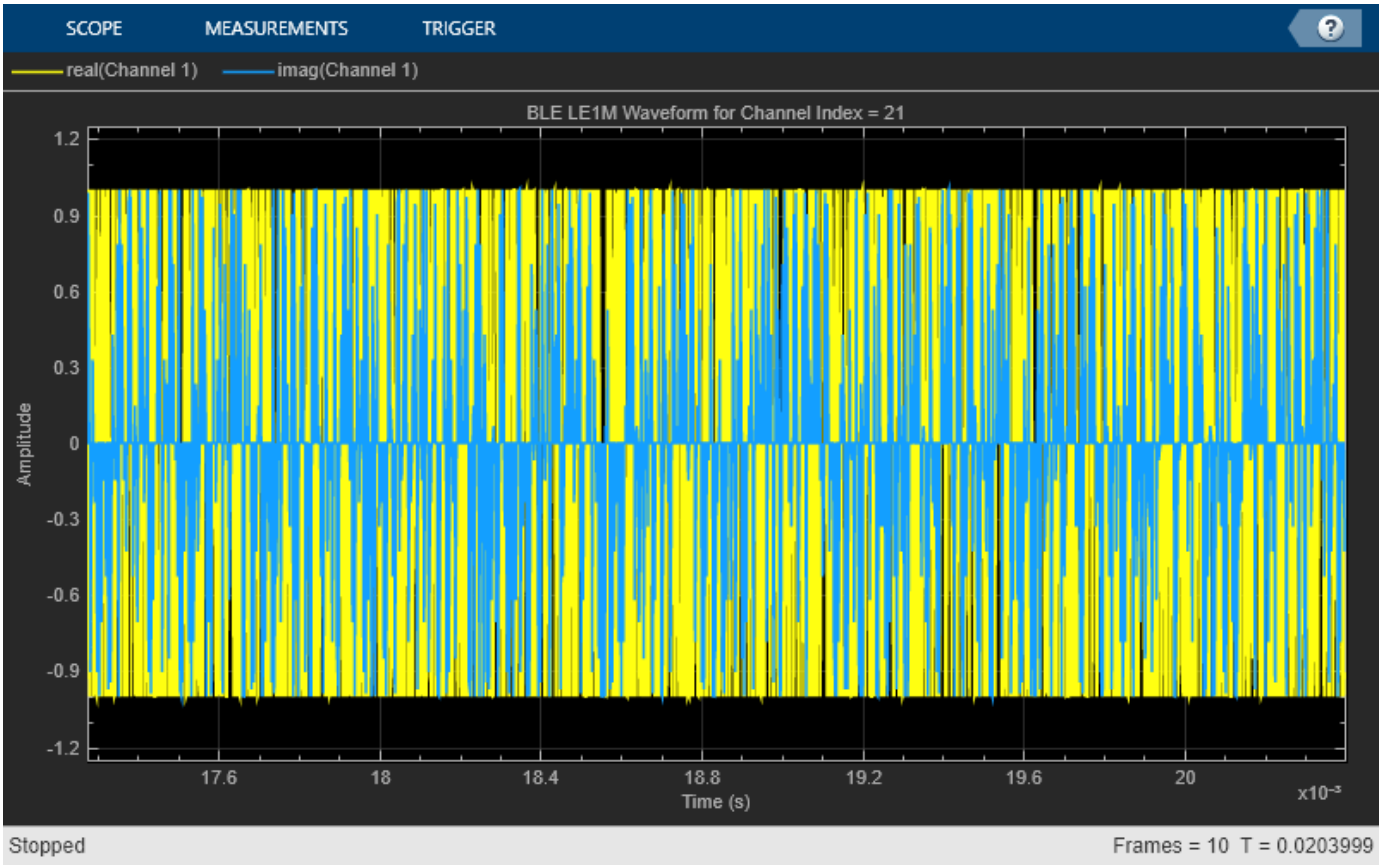
Create Objects for Visualization

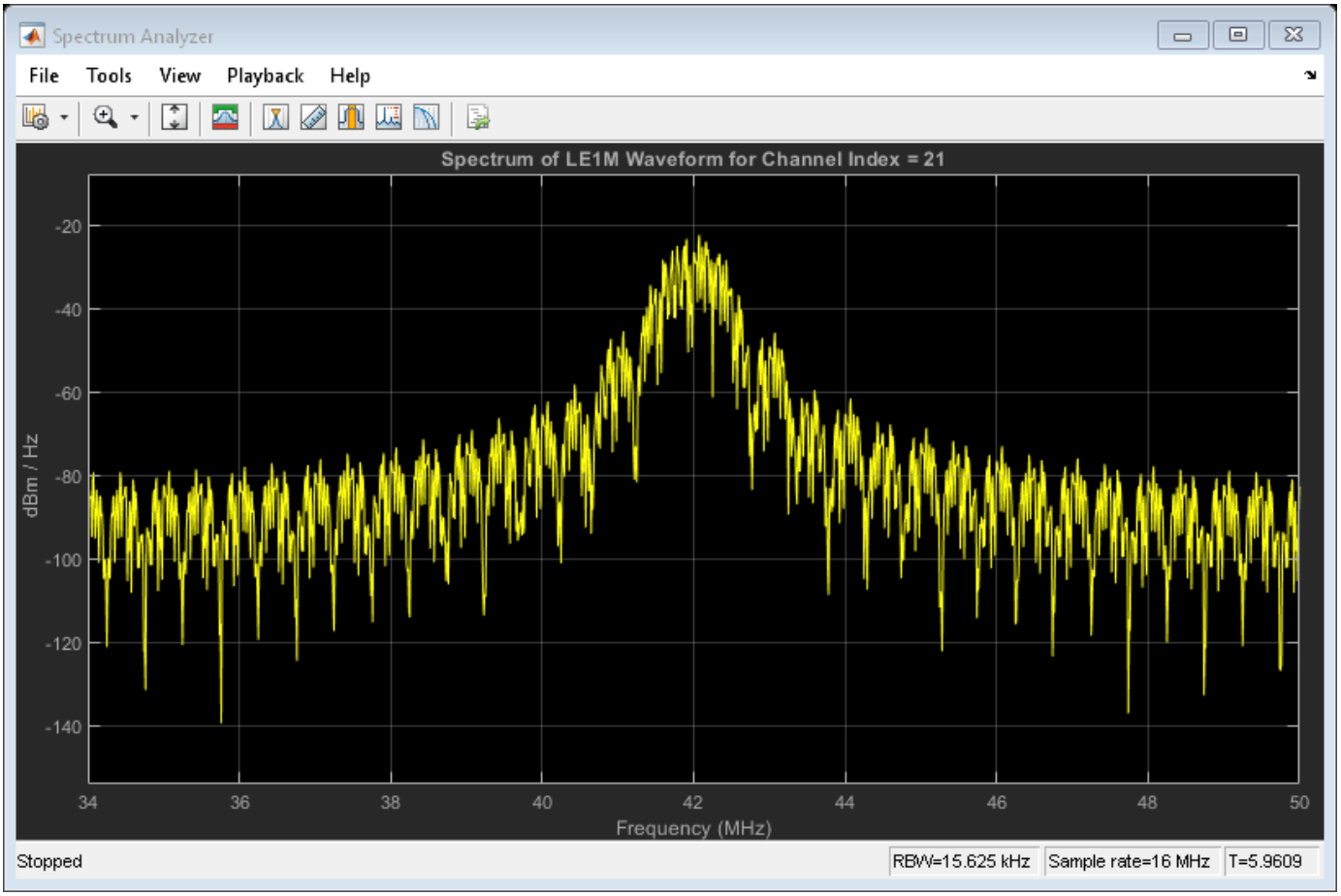
```
% Create a spectrum analyzer object
specAn = dsp.SpectrumAnalyzer('SpectrumType','Power density');
specAn.SampleRate = symbolRate*sps;

% Create a time scope object
```

```
timeScope = timescope('SampleRate', symbolRate*sps,'TimeSpanSource','Auto',...
     'ShowLegend',true);
```

Waveform Generation and Visualization

```
% Loop over the number of packets, generating a BLE waveform and plotting
% the waveform spectrum
rng default;
for packetIdx = 1:numPackets
    message = randi([0 1],messageLen,1);    % Message bits generation
    chanIndex = randi([0 39],1,1);          % Channel index decimal value

    if(chanIndex >=37)
        % Default access address for periodic advertising channels
        accessAdd = [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]';
    else
        % Random access address for data channels
        % Ideally, this access address value should meet the requirements
        % specified in Section 2.1.2 of volume 6 of the Bluetooth Core
        % Specification.
        accessAdd = [0 0 0 0 0 0 0 1 0 0 1 0 0 ...
            0 1 1 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1]';
    end

    waveform = bleWaveformGenerator(message,...
                                    'Mode',phyMode,...
                                    'SamplesPerSymbol',sps,...
                                    'ChannelIndex',chanIndex,...
                                    'AccessAddress',accessAdd);

    specAn.FrequencyOffset = channelBW*chanIndex;
    specAn.Title = ['Spectrum of ',phyMode,' Waveform for Channel Index = ', num2str(chanIndex)];

    tic
    while toc < 0.5 % To hold the spectrum for 0.5 seconds
        specAn(waveform);
    end

    % Plot the generated waveform
    timeScope.Title = ['BLE ',phyMode,' Waveform for Channel Index = ', num2str(chanIndex)];
    timeScope(waveform);
end

% Release objects
release(specAn);
release(timeScope);
```


SCOPE
MEASUREMENTS
TRIGGER
real(Channel 1)
imag(Channel 1)
BLE LE1M Waveform for Channel Index = 21
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Amplitude
17.6
18
18.4
18.8
19.2
19.6
20
Time (s)
x10⁻³
Stopped
Frames = 10 T = 0.0203999



Appendix

The feature used in this example is:

- `bleWaveformGenerator`: Generates BLE physical layer waveform

Selected Bibliography

1 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

End-to-End Bluetooth Low Energy PHY Simulation with RF Impairments and Corrections

This example shows how the Communications Toolbox™ Library for the Bluetooth® Protocol can be used to measure the bit error rate (BER) and packet error rate (PER) for different modes of Bluetooth Low Energy (BLE) [1] physical layer (PHY) packet transmissions that have radio front-end (RF) impairments and additive white gaussian noise (AWGN) added to them.

Introduction

Bluetooth special interest group (SIG) introduced BLE for low power short range communications. BLE devices operate in the globally unlicensed industrial, scientific and medical (ISM) band in the frequency range 2.4 GHz to 2.485 GHz. BLE specifies a channel spacing of 2 MHz, which results in 40 RF channels. The BLE standard specifies the **Link** layer which includes both **PHY** and **MAC** layers. BLE applications include image and video file transfers between mobile phones, home automation, and the internet of things (IoT).

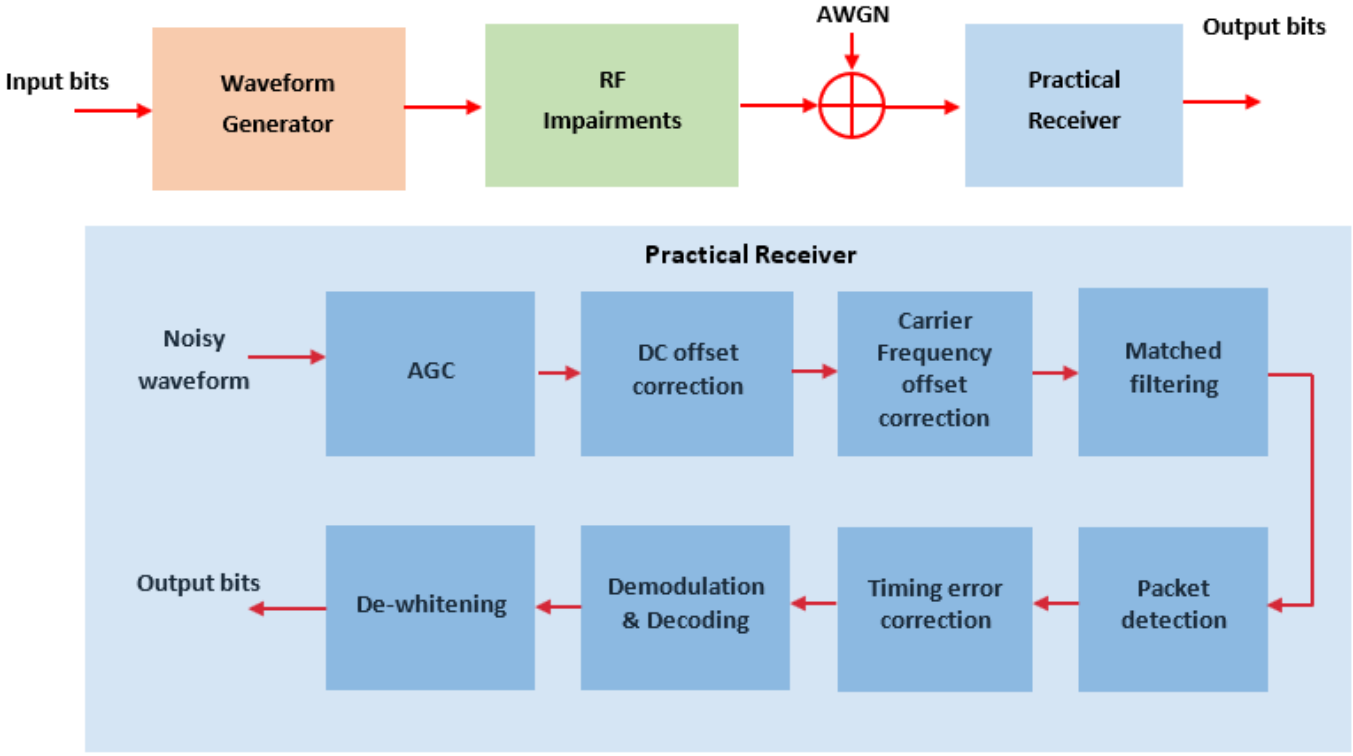
This end-to-end BLE PHY simulation determines BER and PER performance of the four BLE PHY transmission modes with RF impairments and AWGN added to the transmission packets. Nested for loops are used to compute error rates for each transmission mode at several bit energy to noise density ratio (Eb/No) settings. Inside the Eb/No loop, multiple transmission packets are generated using the `bleWaveformGenerator` function and altered with RF impairments and AWGN to accumulate the error rate statistics. Each packet is distorted by these RF impairments:

- DC offset
- Carrier frequency offset
- Carrier phase offset
- Timing drift

White gaussian noise is added to the transmitted BLE waveforms. The noisy packets are processed through a practical BLE receiver that performs the following operations:

- Automatic gain control (AGC)
- DC removal
- Carrier frequency offset correction
- Matched filtering
- Packet detection
- Timing error correction
- Demodulation and decoding
- Dewhitening

The processing steps for each packet are summarized in the following diagram:



The synchronized packets are then demodulated and decoded to recover the data bits. These recovered data bits are compared with the transmitted data bits to determine the BER and PER. BER and PER curves are generated for the following four PHY transmission throughput modes supported in BLE:

- Uncoded PHY with data rate of 1 Mbps (LE1M)
- Uncoded PHY with data rate of 2 Mbps (LE2M)
- Coded PHY with data rate of 500 Kbps (LE500K)
- Coded PHY with data rate of 125 Kbps (LE125K)

Check for Support Package Installation

```
% Check if the 'Communications Toolbox Library for the Bluetooth Protocol'
% support package is installed or not.
commSupportPackageCheck('BLUETOOTH');
```

Initialize the Simulation Parameters

```
EbNo = 2:4:10;                          % Eb/No in dB
sps = 4;                                % Samples per symbol, must be greater than 1
dataLen = 42;                           % Data length in bytes, includes header, payload and CRC
simMode = {'LE1M','LE2M','LE500K','LE125K'}; % PHY modes considered for the simulation
```

The number of packets tested at each Eb/No point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of bit errors simulated at each Eb/No point. When the number of bit errors reaches this limit, the simulation at this Eb/No point is complete.

2 `maxNumPackets` is the maximum number of packets simulated at each Eb/No point and limits the length of the simulation if the bit error limit is not reached.

The numbers chosen for `maxNumErrors` and `maxNumPackets` in this example will lead to a very short simulation. For meaningful results we recommend increasing these numbers.

```
maxNumErrors = 100; % Maximum number of bit errors at an Eb/No point
maxNumPackets = 10; % Maximum number of packets at an Eb/No point
```

Simulating for Each Eb/No Point

This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each Eb/No point to speed up a simulation. `parfor`, as part of the “Parallel Computing Toolbox”, executes processing for each Eb/No point in parallel to reduce the total simulation time. To enable the use of parallel computing for increased speed, comment out the 'for' statement and uncomment the 'parfor' statement below. If Parallel Computing Toolbox (TM) is not installed, 'parfor' will default to the normal 'for' statement.

```
numMode = numel(simMode);          % Number of modes
ber = zeros(numMode,length(EbNo)); % Pre-allocate to store BER results
per = zeros(numMode,length(EbNo)); % Pre-allocate to store PER results
bitsPerByte = 8;                   % Number of bits per byte

% Fixed access address Ideally, this access address value should meet the
% requirements specified in Section 2.1.2 of the Bluetooth specification.
accessAdd = [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]';
for iMode = 1:numMode

    phyMode = simMode{iMode};

    % Set signal to noise ratio (SNR) points based on mode.
    % For Coded PHYs (LE500K and LE125K), the code rate factor is included
    % in SNR calculation as 1/2 rate FEC encoder is used.
    if any(strcmp(phyMode,{'LE1M','LE2M'}))
        snrVec = EbNo - 10*log10(sps);
    else
        codeRate = 1/2;
        snrVec = EbNo + 10*log10(codeRate) - 10*log10(sps);
    end

%     parfor iSnr = 1:length(snrVec)  % Use 'parfor' to speed up the simulation
    for iSnr = 1:length(snrVec)   % Use 'for' to debug the simulation

        % Set random substream index per iteration to ensure that each
        % iteration uses a repeatable set of random numbers
        stream = RandStream('combRecursive','Seed',0);
        stream.Substream = iSnr;
        RandStream.setGlobalStream(stream);

        % Create an instance of error rate
        errorRate = comm.ErrorRate('Samples','Custom',...
                        'CustomSamples',1:(dataLen*bitsPerByte-1));

        % Create and configure the System objects for impairments
        initImp = helperBLEImpairmentsInit(phyMode,sps);
```

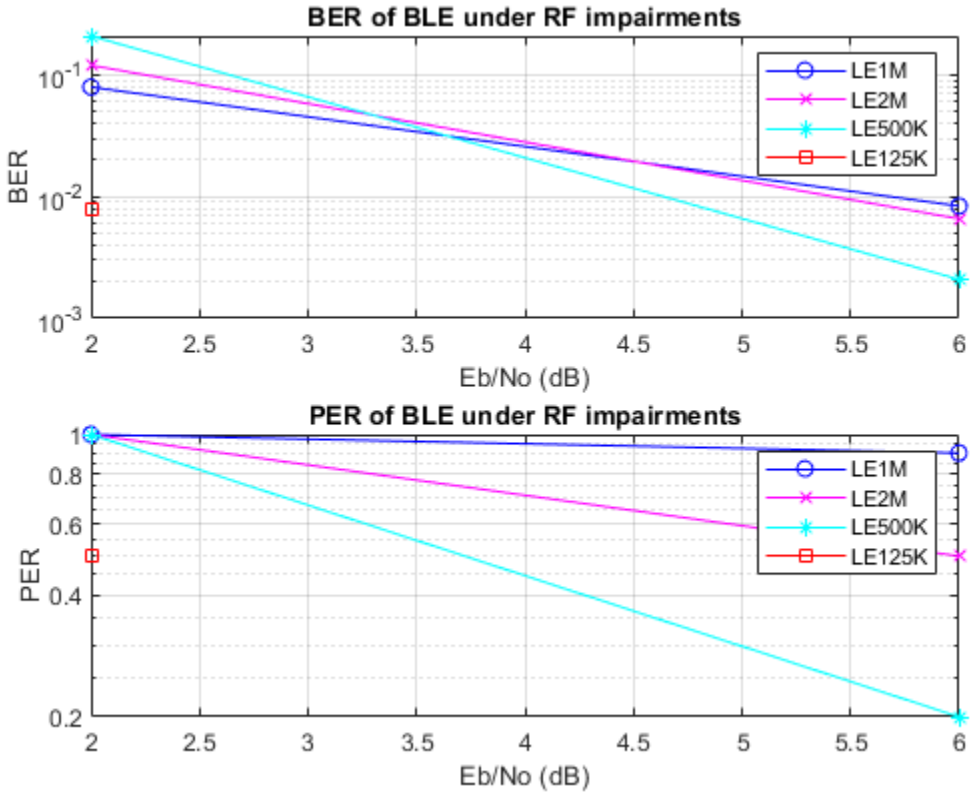
```
            % Create and configure the receiver System objects
            initRxParams = helperBLEReceiverInit(phyMode,sps,accessAdd);

            % Initialize error computation parameters
            [numErrs,perCnt] = deal(0);
            numPkt = 1;

            % Loop to simulate multiple packets
            while numErrs <= maxNumErrors && numPkt <= maxNumPackets

                % Generate BLE waveform
                txBits = randi([0 1],dataLen*bitsPerByte,1,'int8'); % Data bits generation
                chanIndex = randi([0 39],1,1); % Random channel index value for each packet
                txWaveform = bleWaveformGenerator(txBits,'Mode',phyMode,...
                                                  'SamplesPerSymbol',sps,...
                                                  'ChannelIndex',chanIndex,...
                                                  'AccessAddress',accessAdd);

                % Define the RF impairment parameters
                initImp.pfo.FrequencyOffset = randsrc(1,1,-50e3:10:50e3); % In Hz, Max range is +/- 1
                initImp.pfo.PhaseOffset = randsrc(1,1,-10:5:10);          % In degrees
                initoff = 0.15*sps; % Static timing offset
                stepsize = 20*1e-6; % Timing drift in ppm, Max range is +/- 50 ppm
                initImp.vdelay = (initoff:stepsize:initoff+stepsize*(length(txWaveform)-1))'; % Varia
                initImp.dc = 20;     % Percentage w.r.t maximum amplitude value

                % Pass the generated waveform through RF impairments
                txImpairedWfm = helperBLEImpairmentsAddition(txWaveform,initImp);

                % Pass the transmitted waveform through AWGN channel
                rxWaveform = awgn(txImpairedWfm,snrVec(iSnr));

                % Recover data bits using practical receiver
                [rxBits,accessAddress] = helperBLEPracticalReceiver(rxWaveform,initRxParams,chanIndex

                % Determine the BER and PER
                if(length(txBits) == length(rxBits))
                    errors = errorRate(txBits,rxBits); % Outputs the accumulated errors
                    ber(iMode,iSnr) = errors(1);       % Accumulated BER
                    currentErrors = errors(2)-numErrs; % Number of errors in current packet
                    if(currentErrors) % Check if current packet is in error or not
                        perCnt  = perCnt + 1;          % Increment the PER count
                    end
                    numErrs = errors(2);               % Accumulated errors
                    numPkt = numPkt + 1;
                end
            end
            per(iMode,iSnr) = perCnt/(numPkt-1);

        disp(['Mode ' phyMode ', '...
            'Simulating for Eb/No = ', num2str(EbNo(iSnr)), ' dB' ', '...
            'BER:',num2str(ber(iMode,iSnr)), ', '...
            'PER:',num2str(per(iMode,iSnr))])
        end
    end

    Mode LE1M, Simulating for Eb/No = 2 dB, BER:0.079104, PER:1
    Mode LE1M, Simulating for Eb/No = 6 dB, BER:0.0083582, PER:0.9
```

```
Mode LE1M, Simulating for Eb/No = 10 dB, BER:0, PER:0
Mode LE2M, Simulating for Eb/No = 2 dB, BER:0.1194, PER:1
Mode LE2M, Simulating for Eb/No = 6 dB, BER:0.0065672, PER:0.5
Mode LE2M, Simulating for Eb/No = 10 dB, BER:0, PER:0
Mode LE500K, Simulating for Eb/No = 2 dB, BER:0.20746, PER:1
Mode LE500K, Simulating for Eb/No = 6 dB, BER:0.0020896, PER:0.2
Mode LE500K, Simulating for Eb/No = 10 dB, BER:0, PER:0
Mode LE125K, Simulating for Eb/No = 2 dB, BER:0.0077612, PER:0.5
Mode LE125K, Simulating for Eb/No = 6 dB, BER:0, PER:0
Mode LE125K, Simulating for Eb/No = 10 dB, BER:0, PER:0
```

Simulation Results

This section presents the BER and PER results w.r.t the input Eb/No range for the considered PHY modes

```
markers = 'ox*s';
color = 'bmcr';
dataStr = {zeros(numMode,1)};
for iMode = 1:numMode
    subplot(2,1,1),semilogy(EbNo,ber(iMode,:).',['-' markers(iMode) color(iMode)]);
    hold on;
    dataStr(iMode) = simMode(iMode);

    subplot(2,1,2),semilogy(EbNo,per(iMode,:).',['-' markers(iMode) color(iMode)]);
    hold on;
    dataStr(iMode) = simMode(iMode);
end
subplot(2,1,1),
grid on;
xlabel('Eb/No (dB)');
ylabel('BER');
legend(dataStr);
title('BER of BLE under RF impairments');

subplot(2,1,2),
grid on;
xlabel('Eb/No (dB)');
ylabel('PER');
legend(dataStr);
title('PER of BLE under RF impairments');
```



Reference Results

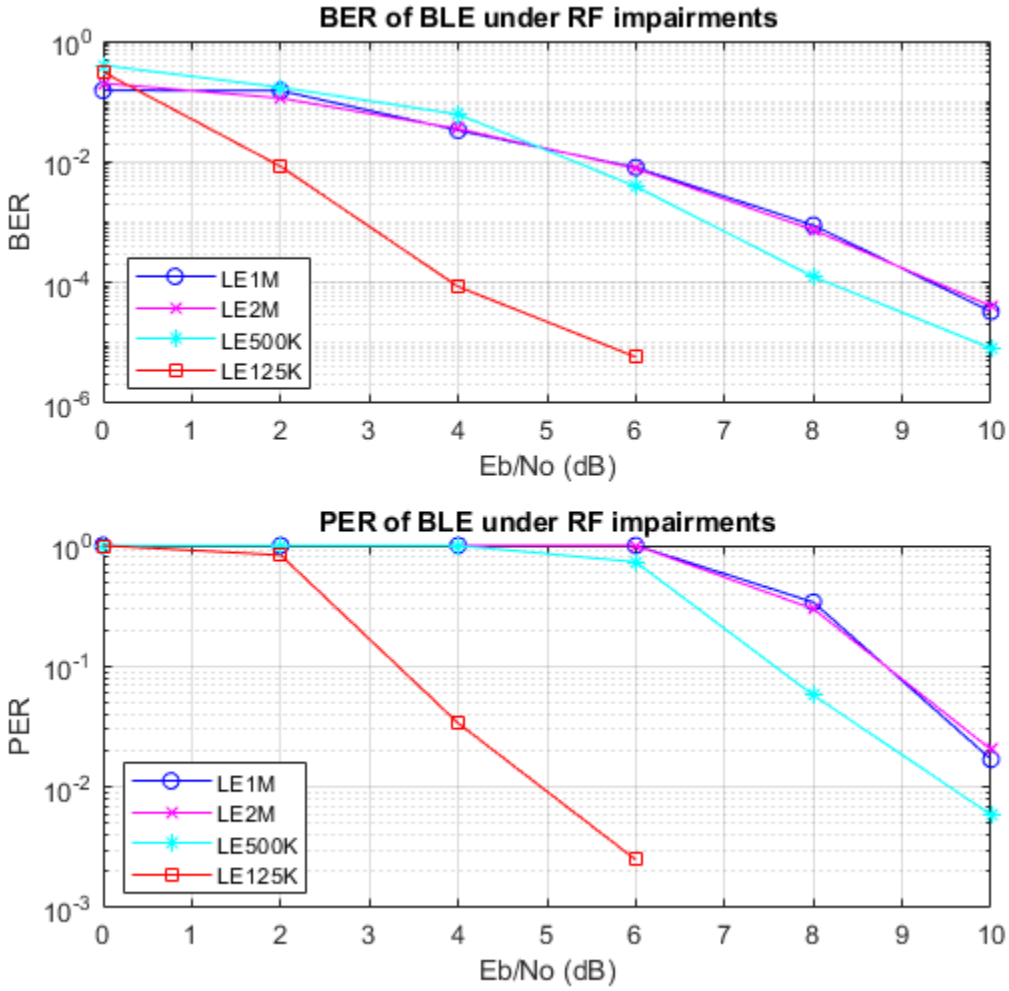
This section generates the reference BER, PER and Eb/No values for each PHY mode based on the receiver sensitivity and corresponding BER as specified in section 4.1 of the Bluetooth specification [1].

```
[refBER,refPER,refEbNo] = deal(zeros(numMode,1));
headerLen = 2; % Header length in bytes
crcLen = 3; % CRC length in bytes
payloadLen = dataLen-headerLen-crcLen; % Payload length in bytes
for iMode = 1:numMode
    [refBER(iMode),refPER(iMode),refEbNo(iMode)] = ...
                        helperBLEReferenceResults(simMode(iMode),payloadLen);
    disp(['Mode ' simMode{iMode} ', '...
        'Reference Eb/No = ', num2str(refEbNo(iMode)), ' dB' ', '...
        'BER = ',num2str(refBER(iMode)), ', '...
        'PER = ',num2str(refPER(iMode)), ', '...
        'for payload length of ',num2str(payloadLen), ' bytes'])
end
```

```
Mode LE1M, Reference Eb/No = 34.919 dB, BER = 0.001, PER = 0.30801, for payload length of 37 byte
Mode LE2M, Reference Eb/No = 34.919 dB, BER = 0.001, PER = 0.30801, for payload length of 37 byte
Mode LE500K, Reference Eb/No = 31.9087 dB, BER = 0.001, PER = 0.30801, for payload length of 37 b
Mode LE125K, Reference Eb/No = 31.9087 dB, BER = 0.001, PER = 0.30801, for payload length of 37 b
```

Further Exploration

The number of packets tested at each Eb/No value is controlled by `maxNumErrors` and `maxNumPackets` parameters. For statistically meaningful results these values should be larger than those presented in this example. To generate the figure below, the simulation ran using a data length of 128 bytes, `maxNumErrors = 1e3`, and `maxNumPackets = 1e4` for all the four transmission modes.



The figure shows that the reference BER and PER can be obtained at lower Eb/No points compared to the reference Eb/No value given in the Bluetooth specification. In this example, only the following impairments are added and passed through AWGN channel.

- DC offset
- Carrier frequency offset
- Carrier phase offset
- Timing drift

The reference Eb/No values generated based on the BLE specification include margin for RF impairments and fading channel conditions that are not modeled in this simulation. As a result, the simulation results here outperform the standard reference results. If you modify this example to include additional impairments such as frequency drift, fading, and interference in the simulation the BER and PER curves will move right towards the reference Eb/No values generated based on the BLE standard receiver characteristics in [1], Volume 6, Section 4.1.

Appendix

The helpers used in this example are:

- helperBLEImpairmentsAddition.m: Adds RF impairments to the BLE waveform
- helperBLEPracticalReceiver.m: Demodulate and decodes the received signal
- helperBLEReceiverInit.m: Initializes BLE receiver parameters
- helperBLEImpairmentsInit.m: Initializes RF impairment parameters
- helperBLEReferenceResults.m: Generates reference BER, PER and Eb/No values

Summary

This example simulates a BLE PHY packet transmissions that have RF impairments and AWGN added to them. It shows how to generate BLE waveforms, demodulate and decode data bits using practical receiver and compute the BER and PER.

Selected Bibliography

1 Volume 6 of the Bluetooth Core Specification, Version 5.0 Core System Package [Low Energy Controller Volume].

See Also

More About

- “Generate BLE Waveform and Add RF Impairments” on page 13-103
- “Bluetooth Low Energy Transmitter” on page 3-207
- “Bluetooth Low Energy Receiver” on page 3-212

Shared deeplearning_shared Examples (comm/deeplearning)

Spectrum Sensing with Deep Learning to Identify 5G and LTE Signals

This example shows how to train a semantic segmentation network using deep learning for spectrum monitoring. One of the uses of spectrum monitoring is to characterize spectrum occupancy. The neural network in this example is trained to identify 5G NR and LTE signals in a wideband spectrogram.

Introduction

Computer vision uses the semantic segmentation technique to identify objects and their locations in an image or a video. In wireless signal processing, the objects of interest are wireless signals, and the locations of the objects are the frequency and time occupied by the signals. In this example we apply the semantic segmentation technique to wireless signals to identify spectral content in a wideband spectrogram.

In the following, you will:

1. Generate training signals.
2. Apply transfer learning to a semantic segmentation network to identify 5G NR and LTE signals in time and frequency.
3. Test the trained network with synthetic signals.
4. Use an SDR to test the network with over the air (OTA) signals.

Generate Training Data

One advantage of wireless signals in the deep learning domain is the fact that the signals are synthesized. Also, we have highly reliable channel and RF impairment models. As a result, instead of collecting and manually labeling signals, you can generate 5G NR signals using 5G Toolbox™ and LTE signals using LTE Toolbox™ functions. You can pass these signals through standards-specified channel models to create the training data.

Train the network with frames that contain only 5G NR or LTE signals and then shift these signals in frequency randomly within the band of interest. Each frame is 40 ms long, which is the duration of 40 subframes. The network assumes that the 5G NR or LTE signal occupies the same band for the whole frame duration. To test the network performance, create frames that contain both 5G NR and LTE signals on distinct random bands within the band of interest.

Use a sampling rate of 61.44 MHz. This rate is high enough to process most of the latest standard signals and several low-cost software defined radio (SDR) systems can sample at this rate providing about 50 MHz of useful bandwidth. To monitor a wider band, you can increase the sample rate, regenerate training frames and retrain the network.

Use the `helperSpecSenseTrainingData` function to generate training frames. This function generates 5G NR signals using the `helperSpecSenseNRSignal` function and LTE signals using the `helperSpecSenseLTESignal` function. This table lists 5G NR variable signal parameters.

5G NR Parameter	Value	Units
Bandwidth	[10 15 20 25 30 40 50]	MHz
Sub-Carrier Spacing (SCS)	[15 30]	kHz
SSB Block Pattern	["Case A" "Case B"]	
SSB Period	[20]	ms

This table lists LTE variable signal parameters.

LTE Parameter	Value	Units
Reference Channel	["R.2","R.6","R.8","R.9"}	
Bandwidth	[10 5 15 20]	MHz
Duplex Mode	FDD	

Use the `nrCDLChannel` (5G Toolbox) and the `lteFadingChannel` (LTE Toolbox) functions to add channel impairments. For details of the channel configurations, see the `helperSpecSenseTrainingData` function. This table lists channel parameters.

Channel Parameter	Value	Units
SNR	[40 50 100]	dB
Doppler	[0 10 500]	Hz

The `helperSpecSenseTrainingData` function uses the `helperSpecSenseSpectrogramImage` function to create spectrogram images from complex baseband signals. Calculate the spectrograms using an FFT length of 4096. Generate 256 by 256 RGB images. This Image size allows a large enough batch of images to fit in memory during training while providing enough resolution in time and frequency. If your GPU does not have sufficient memory, you can resize the images to smaller sizes or reduce the training batch size.

The `generateTrainData` variable determines whether training data is to be downloaded or generated. Choosing "Use downloaded data" sets the `generateTrainData` variable to `false`. Choosing "Generate training data" sets the `generateTrainData` variable to `true` to generate the training data from scratch. Data generation may take several hours depending on the configuration of your computer. Using a PC with Intel® Xeon® W-2133 CPU @ 3.60GHz and creating a parallel pool with six workers with the Parallel Computing Toolbox, training data generation takes about an hour. Choose "Train network now" to train the network. This process takes about 20 minutes with the same PC and NVIDIA® Titan V GPU. Choose "Use trained network" to skip network training. Instead, the example downloads the trained network.

Use 900 frames from each set of signals: 5G NR only, LTE only and 5G NR and LTE both. If you increase the number of possible values for the system parameters, increase the number of training frames .

```
imageSize = [256 256];    % pixels
sampleRate = 61.44e6;     % Hz
numSubFrames = 40;        % corresponds to 40 ms
frameDuration = numSubFrames*1e-3;    % seconds
trainDir = fullfile(pwd,'TrainingData');
```

```
generateTrainData = Use downloaded data ;
trainNow = Use trained network ;
if ~generateTrainData || ~trainNow
  helperSpecSenseDownloadData()
end
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/SpectrumSensing/SpectrumSenseTrainingDataNetwork.
Download complete. Extracting files.
Extract complete.
```

```
if generateTrainData
  numFramesPerStandard = 900;
  helperSpecSenseTrainingData(numFramesPerStandard,imageSize,trainDir,numSubFrames,sampleRate);
end
```

Load Training Data

Use the `imageDatastore` function to load training images with the spectrogram of 5G NR and LTE signals. The `imageDatastore` function enables you to efficiently load a large collection of images from disk. Spectrogram images are stored in .png files.

```
imds = imageDatastore(trainDir,'IncludeSubfolders',false,'FileExtensions','.png');
```

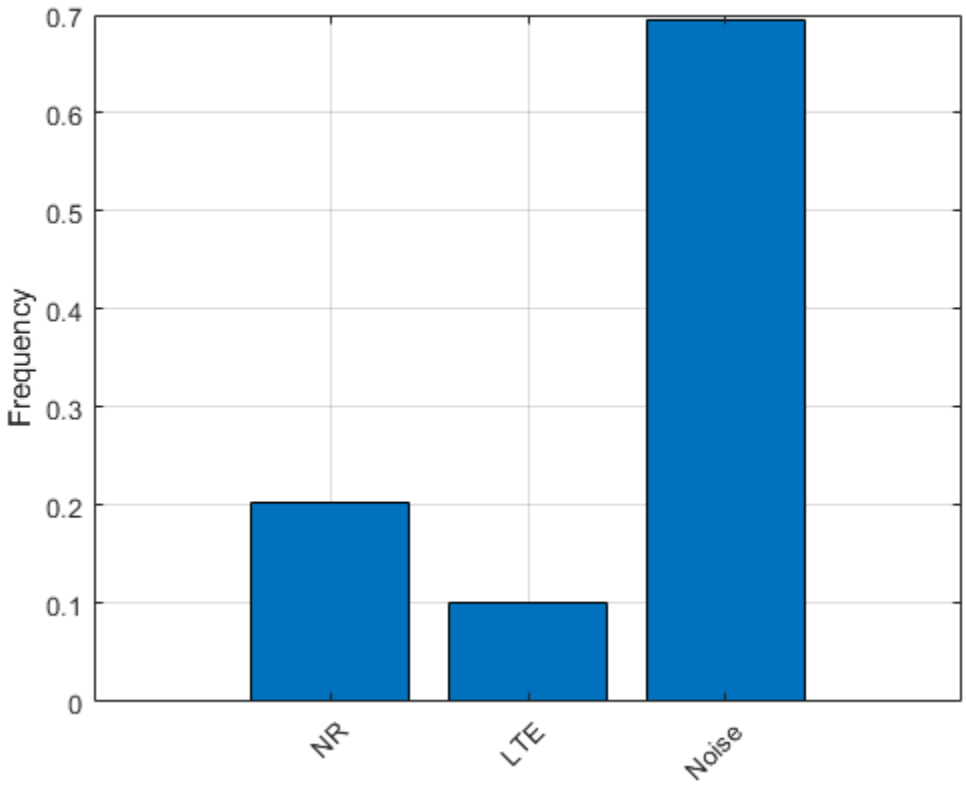
Use the `pixelLabelDatastore` (Computer Vision Toolbox) function to load spectrogram pixel label image data. Each pixel is labeled as one of "NR", "LTE" or "Noise". A pixel label datastore encapsulates the pixel label data and the label ID to a class name mapping. Pixel labels are stored in .hdf files.

```
classNames = ["NR" "LTE" "Noise"];
pixelLabelID = [127 255 0];
pxdsTruth = pixelLabelDatastore(trainDir,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
```

Analyze Dataset Statistics

To see the distribution of class labels in the training dataset, use the `countEachLabel` (Computer Vision Toolbox) function to count the number of pixels by class label, and plot the pixel counts by class.

```
tbl = countEachLabel(pxdsTruth);
frequency = tbl.PixelCount/sum(tbl.PixelCount);
figure
bar(1:numel(classNames),frequency)
grid on
xticks(1:numel(classNames))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes would have an equal number of observations. However, with wireless signals it is common for the classes in the training set to be imbalanced. 5G NR signals may have larger bandwidth than LTE signals, and noise fills the background. Because the learning is biased in favor of the dominant classes, imbalance in the number of observations per class can be detrimental to the learning process. In the Balance Classes Using Class Weighting on page 4-0 section, class weighting is used to mitigate bias caused by imbalance in the number of observations per class.

Prepare Training, Validation, and Test Sets

The deep neural network uses 80% of the single signal images from the dataset for training and, 20% of the images for validation. The `helperSpecSensePartitionData` function randomly splits the image and pixel label data into training and validation sets.

```
[imdsTrain,pxdsTrain,imdsVal,pxdsVal] = helperSpecSensePartitionData(imds,pxdsTruth,[80 20]);
cdsTrain = pixelLabelImageDatastore(imdsTrain,pxdsTrain,'OutputSize',imageSize);
cdsVal = pixelLabelImageDatastore(imdsVal,pxdsVal,'OutputSize',imageSize);
```

Train Deep Neural Network

Use the `deeplabv3plusLayers` (Computer Vision Toolbox) function to create a semantic segmentation neural network. Choose `resnet50` (Deep Learning Toolbox) as the base network and specify the input image size (number of pixels used to represent time and frequency axes) and the number of classes. If the Deep Learning Toolbox™ Model *for ResNet-50 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the

installation is successful by typing `resnet50` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
baseNetwork = resnet50;
lgraph = deeplabv3plusLayers(imageSize,numel(classNames),baseNetwork);
```

Balance Classes Using Class Weighting

To improve training when classes in the training set are not balanced, you can use class weighting to balance the classes. Use the pixel label counts computed earlier with the `countEachLabel` function and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;
```

Specify the class weights using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights
lgraph = replaceLayer(lgraph,"classification",pxLayer);
```

Select Training Options

Configure training using the `trainingOptions` (Deep Learning Toolbox) function to specify the stochastic gradient descent with momentum (SGDM) optimization algorithm and the hyper-parameters used for SGDM. To get the best performance from the network, you can use the Experiment Manager (Deep Learning Toolbox) to optimize training options.

```
opts = trainingOptions("sgdm",...
  MiniBatchSize = 40,...
  MaxEpochs = 20, ...
  LearnRateSchedule = "piecewise",...
  InitialLearnRate = 0.02,...
  LearnRateDropPeriod = 10,...
  LearnRateDropFactor = 0.1,...
  ValidationData = cdsVal,...
  ValidationPatience = 5,...
  Shuffle="every-epoch",...
  OutputNetwork = "best-validation-loss",...
  Plots = 'training-progress')
```

```
opts =
  TrainingOptionsSGDM with properties:

                      Momentum: 0.9000
                InitialLearnRate: 0.0200
               LearnRateSchedule: 'piecewise'
             LearnRateDropFactor: 0.1000
             LearnRateDropPeriod: 10
                L2Regularization: 1.0000e-04
         GradientThresholdMethod: 'l2norm'
               GradientThreshold: Inf
                       MaxEpochs: 20
                   MiniBatchSize: 40
                         Verbose: 1
                VerboseFrequency: 50
                  ValidationData: [1×1 pixelLabelImageDatastore]
             ValidationFrequency: 50
```

```
              ValidationPatience: 5
                         Shuffle: 'every-epoch'
                  CheckpointPath: ''
             ExecutionEnvironment: 'auto'
                      WorkerLoad: []
                       OutputFcn: []
                           Plots: 'training-progress'
                  SequenceLength: 'longest'
             SequencePaddingValue: 0
         SequencePaddingDirection: 'right'
             DispatchInBackground: 0
          ResetInputNormalization: 1
    BatchNormalizationStatistics: 'population'
                   OutputNetwork: 'best-validation-loss'
```

Train the network using the combined training data store, `cdsTrain.` The combined training data store contains single signal frames and true pixel labels.

```
if trainNow
  [net,trainInfo] = trainNetwork(cdsTrain,lgraph,opts); %#ok<UNRCH>
else
  load specSenseTrainedNet net
end
```

Test with Synthetic Signals

Test the network signal identification performance using signals that contain both 5G NR and LTE signals. Use the `semanticseg` (Computer Vision Toolbox) function to get the pixel estimates of the spectrogram images in the test data set. Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute various metrics to evaluate the quality of the semantic segmentation results.

```
dataDir = fullfile(trainDir,'LTE_NR');
imds = imageDatastore(dataDir,'IncludeSubfolders',false,'FileExtensions','.png');
pxdsResults = semanticseg(imds,net,"WriteLocation",tempdir);
```

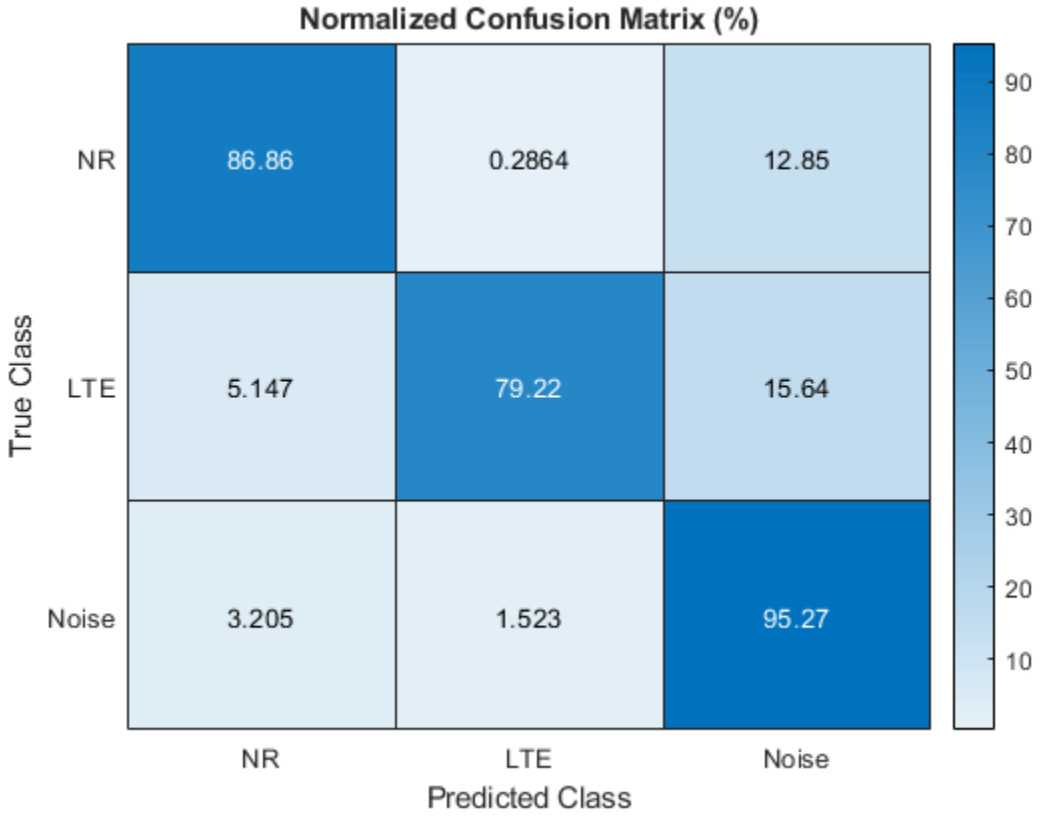
```
Running semantic segmentation network
-------------------------------------
* Processed 900 images.
```

```
pxdsTruth = pixelLabelDatastore(dataDir,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTruth);
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 900 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.88609           0.87117       0.79066      0.79601        0.65624
```

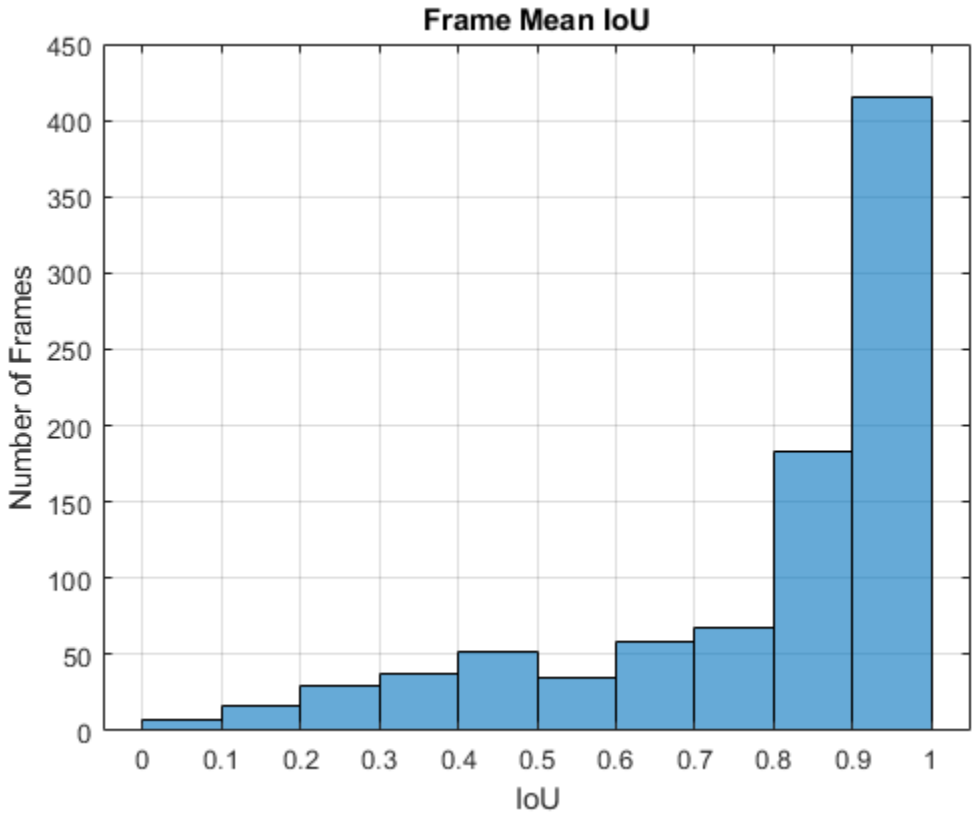
Plot the normalized confusion matrix for all test frames as a heat map.

```
normConfMatData = metrics.NormalizedConfusionMatrix.Variables;
figure
h = heatmap(classNames,classNames,100*normConfMatData);
h.XLabel = 'Predicted Class';
h.YLabel = 'True Class';
h.Title = 'Normalized Confusion Matrix (%)';
```



Plot the histogram of the per-image intersection over union (IoU). For each class, IoU is the ratio of correctly classified pixels to the total number of ground truth and predicted pixels in that class.

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
grid on
xlabel('IoU')
ylabel('Number of Frames')
title('Frame Mean IoU')
```



Inspecting low SNR frames shows that the spectrogram images do not contain visual features that can help the network identify the low SNR frames correctly. Repeat the same process, considering only the frames with average SNR of 50dB or 100dB and ignoring the frames with average SNR of 40dB.

```
files = dir(fullfile(dataDir,'*.mat'));
dataFiles = {};
labelFiles = {};
for p=1:numel(files)
  load(fullfile(files(p).folder,files(p).name),'params');
  if params.SNRdB > 40
    [~,name] = fileparts(files(p).name);
    dataFiles = [dataFiles; fullfile(files(p).folder,[name '.png'])]; %#ok<AGROW>
    labelFiles = [labelFiles; fullfile(files(p).folder,[name '.hdf'])]; %#ok<AGROW>
  end
end
imds = imageDatastore(dataFiles);
pxdsResults = semanticseg(imds,net,"WriteLocation",tempdir);
```

```
Running semantic segmentation network
-------------------------------------
* Processed 608 images.
```

```
pxdsTruth = pixelLabelDatastore(labelFiles,classNames,pixelLabelID);
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTruth);
```

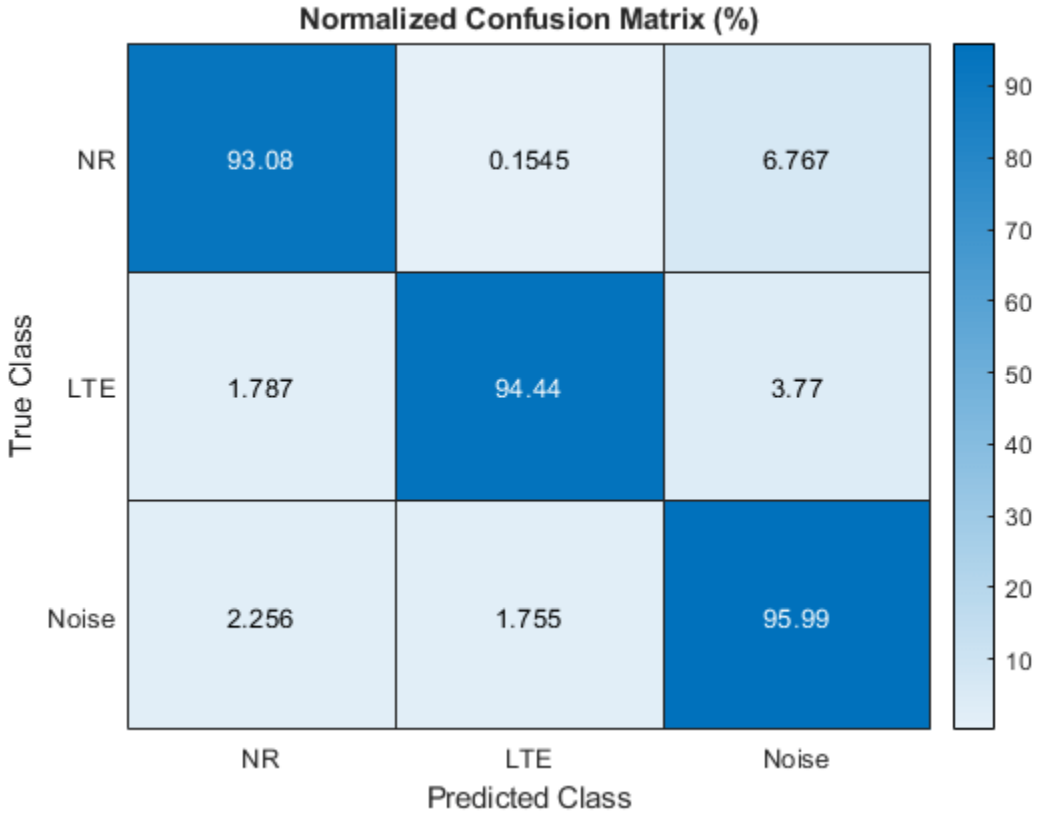
```
Evaluating semantic segmentation results
----------------------------------------
```

```
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 608 images.
* Finalizing... Done.
* Data set metrics:
```

GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU	MeanBFScore
0.94487	0.94503	0.89799	0.89582	0.74699

Considering only the set of frames with higher SNR, replot the normalized confusion matrix and observe the improved network accuracy.

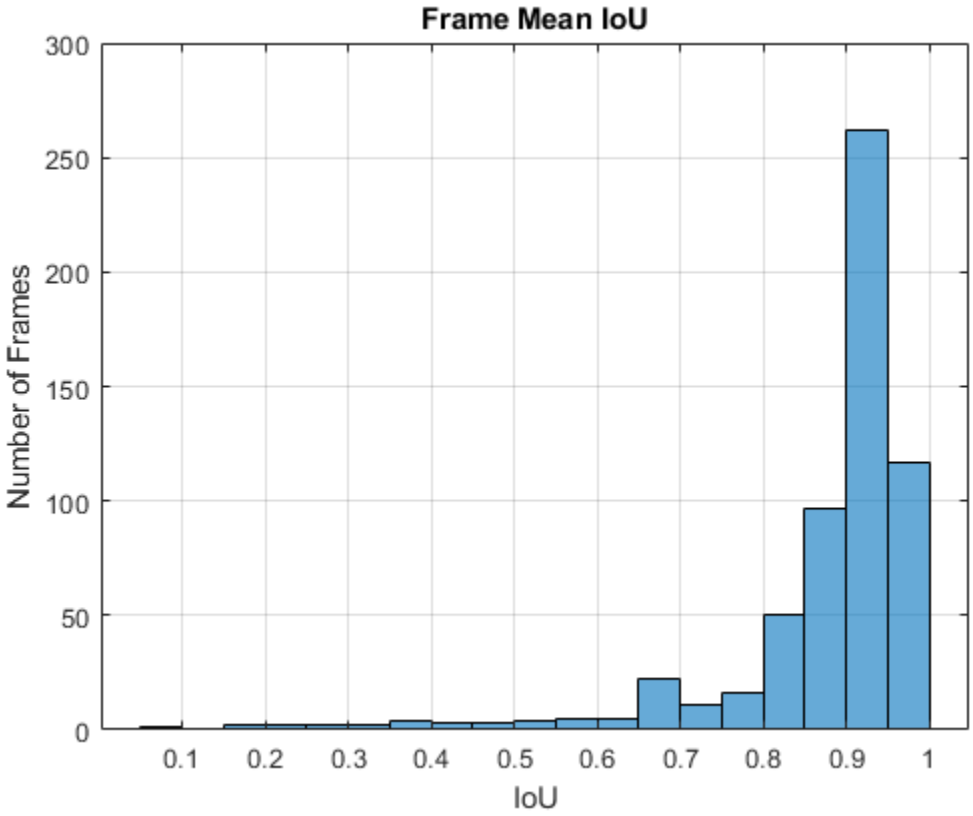
```
normConfMatData = metrics.NormalizedConfusionMatrix.Variables;
figure
h = heatmap(classNames,classNames,100*normConfMatData);
h.XLabel = 'Predicted Class';
h.YLabel = 'True Class';
h.Title = 'Normalized Confusion Matrix (%)';
```



Considering only the set of frames with higher SNR, replot the per-image IoU histogram and observe the improved distribution.

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
grid on
xlabel('IoU')
```

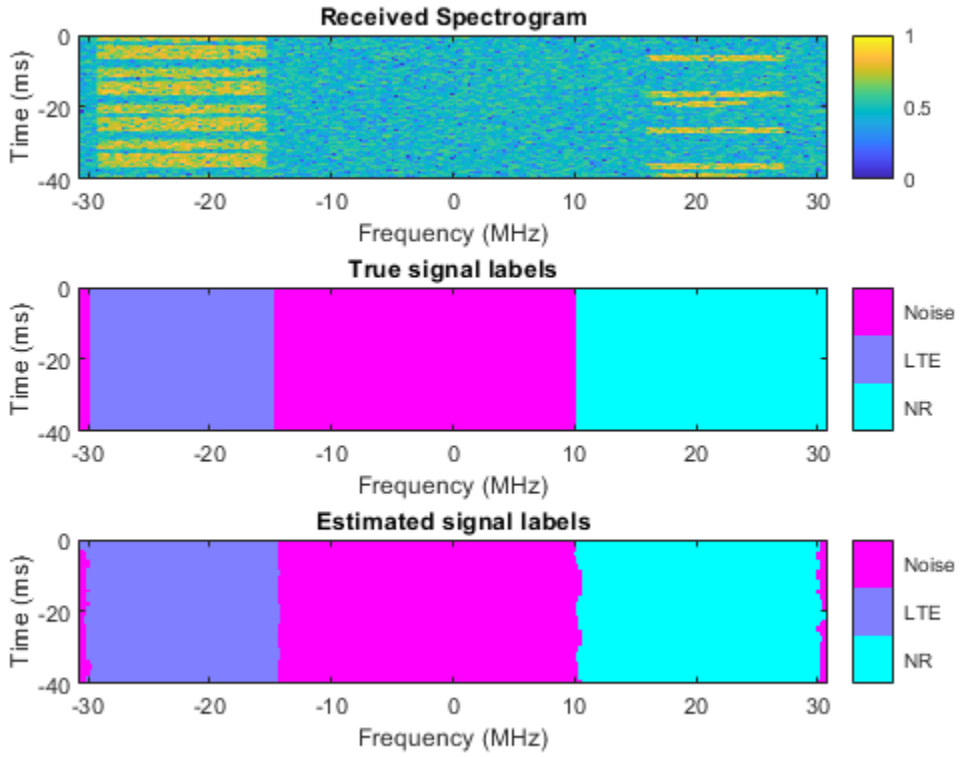
```
ylabel('Number of Frames')
title('Frame Mean IoU')
```



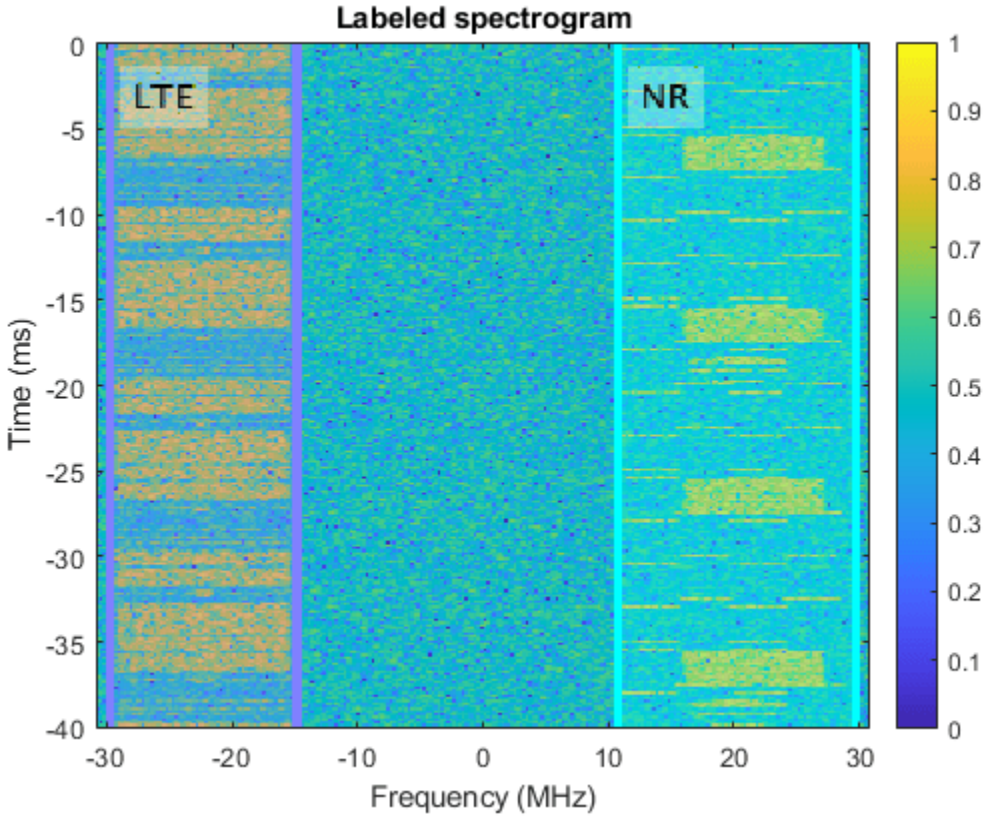
Identify 5G NR and LTE Signals in Spectrogram

Visualize the received spectrum, true labels, and predicted labels for the image with index 602.

```
imgIdx = 602;
rcvdSpectrogram = readimage(imds,imgIdx);
trueLabels = readimage(pxdsTruth,imgIdx);
predictedLabels = readimage(pxdsResults,imgIdx);
figure
helperSpecSenseDisplayResults(rcvdSpectrogram,trueLabels,predictedLabels, ...
  classNames,sampleRate,0,frameDuration)
```




```
figure
helperSpecSenseDisplayIdentifiedSignals(rcvdSpectrogram,predictedLabels, ...
  classNames,sampleRate,0,frameDuration)
```



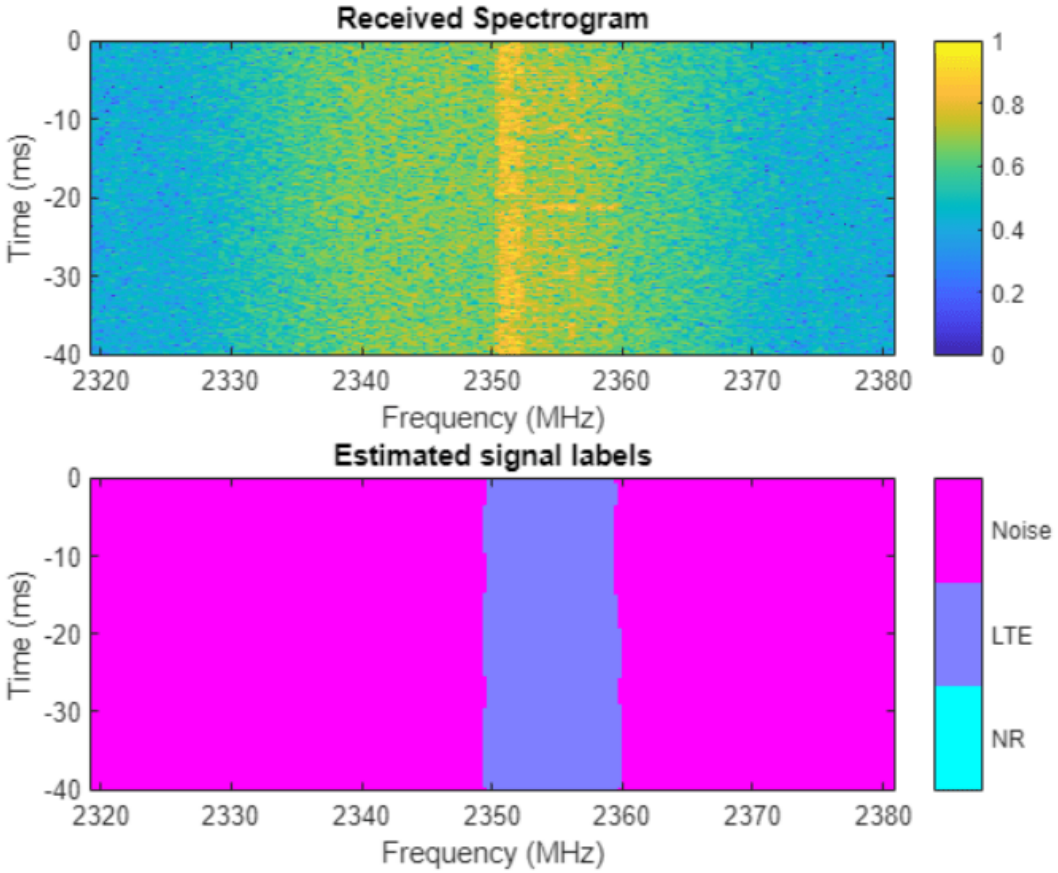
Test with Over-the-Air Signals

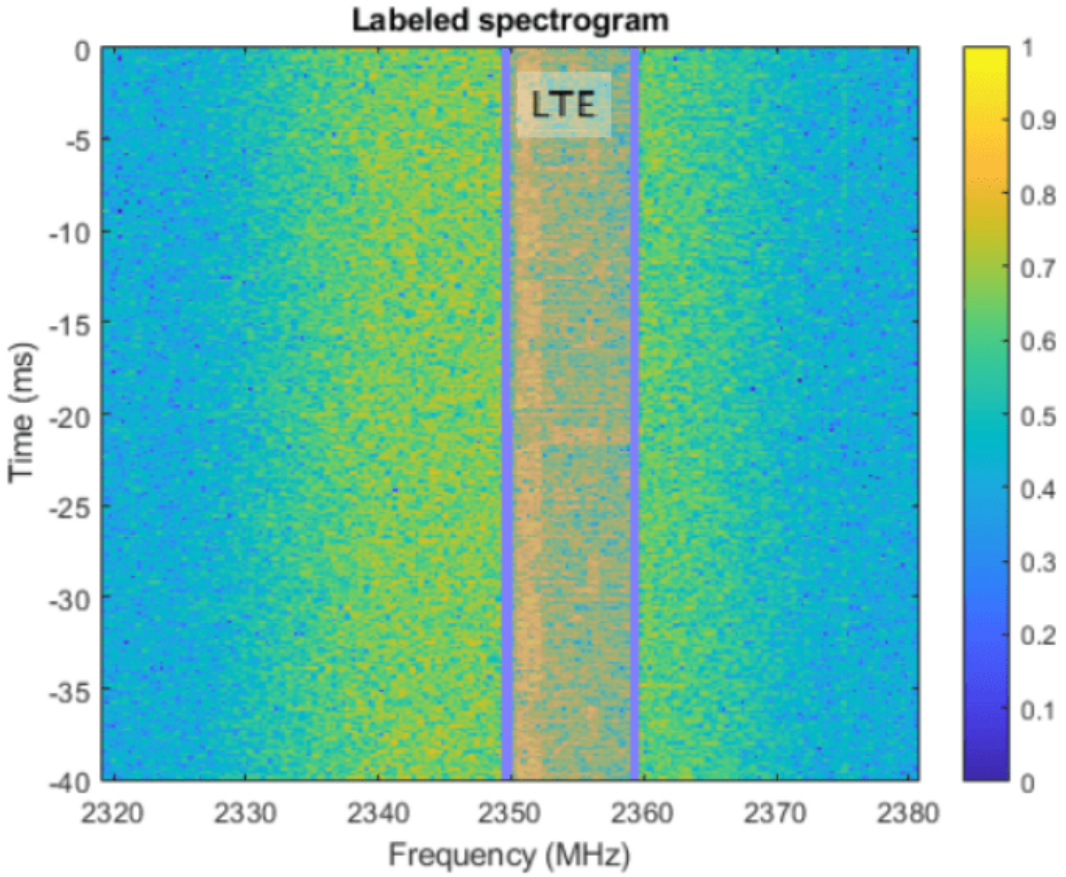
Test the performance of the trained network using over-the-air signal captures. Find a nearby base station and tune the center frequency of your radio to cover the band of the signals you want to identify. This example sets the center frequency to 2.35 GHz. If you have at least one ADALM-PLUTO radio and have installed Communication Toolbox Support Package for ADALM-PLUTO Radio, you can run this section of the code. In case you do not have access to an ADALM-PLUTO radio, this example shows results of a test conducted using captured signals.

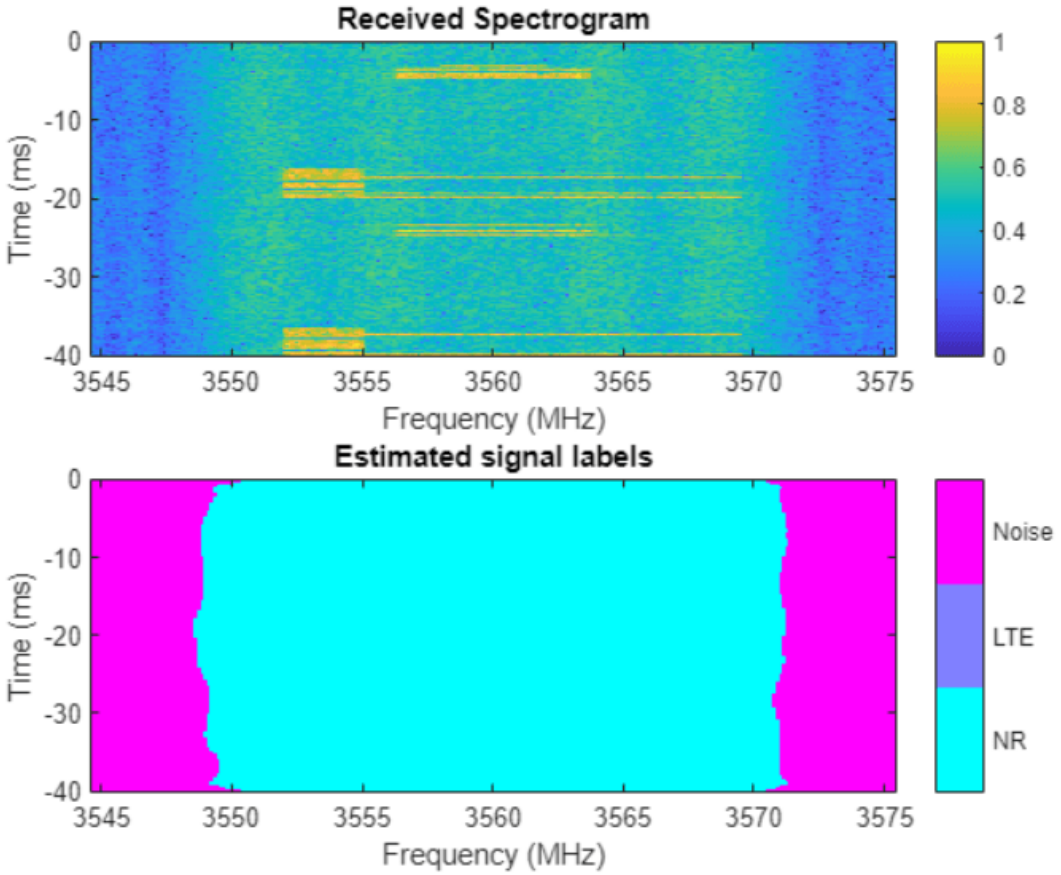
```
runSDRSection = false;
if helperIsPlutoSDRInstalled()
  radios = findPlutoRadio();
  if length(radios) >= 1
    runSDRSection = true;
  else
    disp("At least one ADALM-PLUTO radios is needed. Skipping SDR test.")
  end
else
    disp("Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found."
    disp("Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package."
    disp("Skipping SDR test.")
end
```

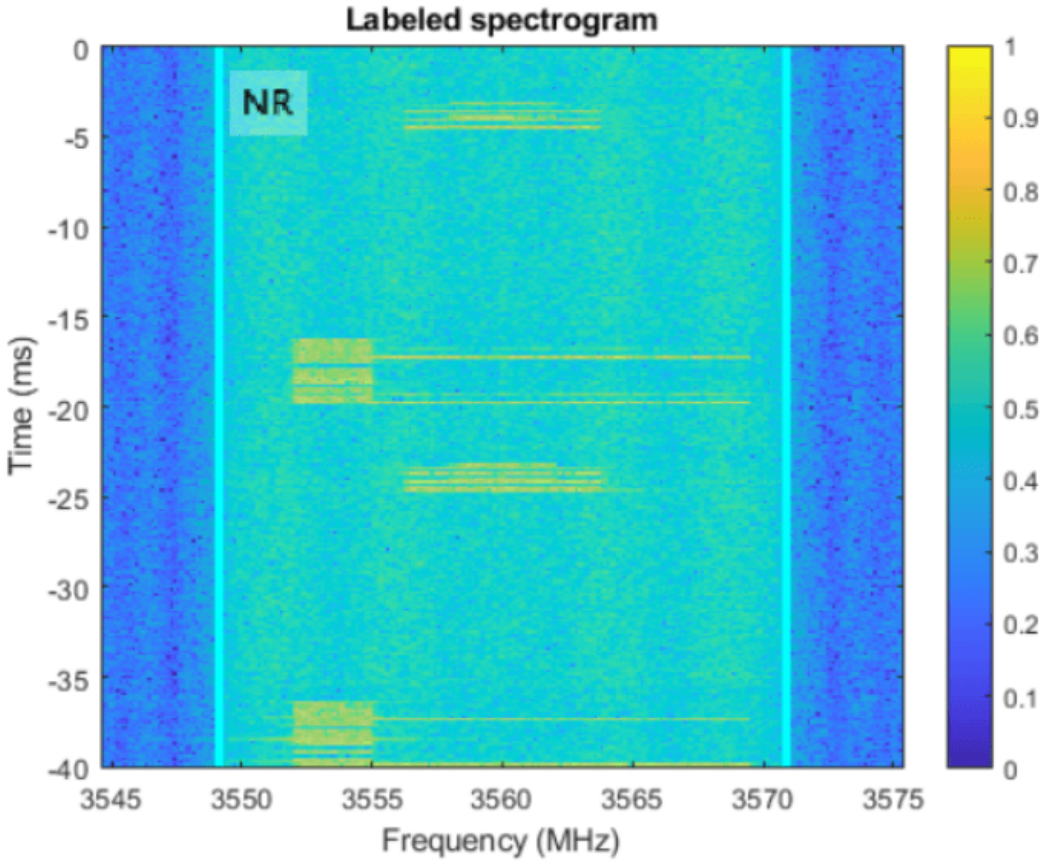
```
Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found.

Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package.

Skipping SDR test.
```

```
if runSDRSection
  % Set up PlutoSDR receiver
  rx = sdrrx('Pluto');
  rx.CenterFrequency = 2.35e9;
  rx.BasebandSampleRate = sampleRate;
  rx.SamplesPerFrame = frameDuration*rx.BasebandSampleRate;
  rx.OutputDataType = 'single';
  rx.EnableBurstMode = true;
  rx.NumFramesInBurst = 1;
  Nfft = 4096;
  overlap = 10;

  meanAllScores = zeros([imageSize numel(classNames)]);
  segResults = zeros([imageSize 10]);
  for frameCnt=1:10
    rxWave = rx();
    rxSpectrogram = helperSpecSenseSpectrogramImage(rxWave,Nfft,sampleRate,imageSize);

    [segResults(:,:,frameCnt),scores,allScores] = semanticseg(rxSpectrogram,net);
    meanAllScores = (meanAllScores*(frameCnt-1) + allScores) / frameCnt;
  end
  release(rx)

  [~,predictedLabels] = max(meanAllScores,[],3);
  figure
  helperSpecSenseDisplayResults(rxSpectrogram,[],predictedLabels,classNames,...
    sampleRate,rx.CenterFrequency,frameDuration)
  figure
  freqBand = helperSpecSenseDisplayIdentifiedSignals(rxSpectrogram,predictedLabels,...
    classNames,sampleRate,rx.CenterFrequency,frameDuration)
else
  figure
  imshow('lte_capture_result1.png')
  figure
  imshow('lte_capture_result2.png')
  figure
  imshow('nr_capture_result1.png')
  figure
  imshow('nr_capture_result2.png')
end
```


Received Spectrogram
Time (ms)
0
-10
-20
-30
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
1
0.8
0.6
0.4
0.2
0
Estimated signal labels
Time (ms)
0
-10
-20
-30
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
Noise
LTE
NR


Labeled spectrogram
LTE
Time (ms)
0
-5
-10
-15
-20
-25
-30
-35
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0


Received Spectrogram
Time (ms)
0
-10
-20
-30
-40
3545
3550
3555
3560
3565
3570
3575
Frequency (MHz)
1
0.8
0.6
0.4
0.2
0
Estimated signal labels
Time (ms)
0
-10
-20
-30
-40
3545
3550
3555
3560
3565
3570
3575
Frequency (MHz)
Noise
LTE
NR



Conclusions and Further Exploration

The trained network can distinguish 5G NR and LTE signals including two example captures from real base stations. The network may not be able to identify every captured signal correctly. In such cases, enhance the training data either by generating more representative synthetic signals or capturing over-the-air signals and including these in the training set.

You can use the LTE “Cell Search, MIB and SIB1 Recovery” (LTE Toolbox) and the “NR Cell Search and MIB and SIB1 Recovery” (5G Toolbox) examples to identify LTE and 5G NR base stations manually to capture training data, respectively.

If you need to monitor wider bands of spectrum, increase the `sampleRate`, regenerate the training data and retrain the network.

See Also

`classificationLayer` | `featureInputLayer` | `fullyConnectedLayer` | `reluLayer` | `softmaxLayer` | `pixelLabelDatastore` | `countEachLabel` | `pixelClassificationLayer`

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Autoencoders for Wireless Communications

This example shows how to model an end-to-end communications system with an autoencoder to reliably transmit information bits over a wireless channel.

Introduction

A traditional autoencoder is an unsupervised neural network that learns how to efficiently compress data, which is also called encoding. The autoencoder also learns how to reconstruct the data from the compressed representation such that the difference between the original data and the reconstructed data is minimal.

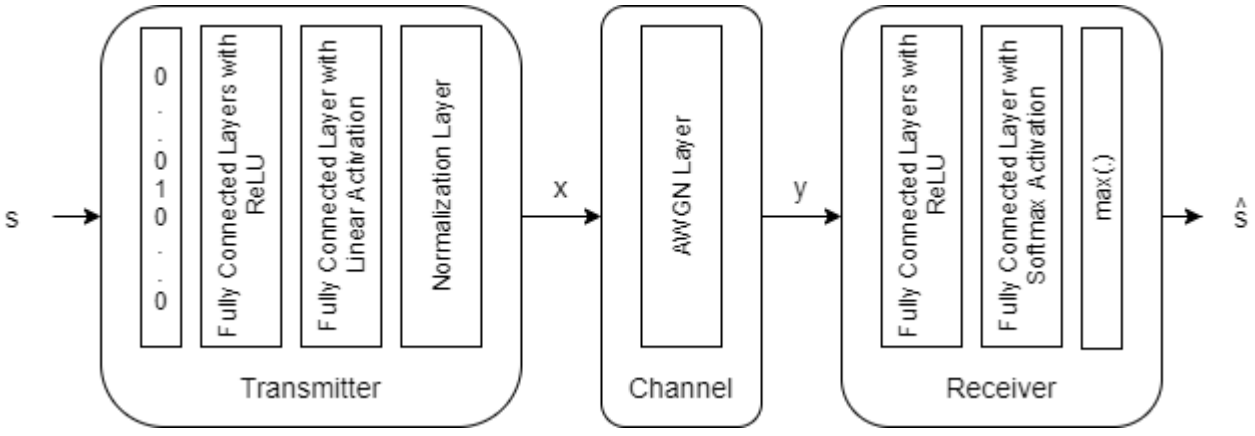
Traditional wireless communication systems are designed to provide reliable data transfer over a channel that impairs the transmitted signals. These systems have multiple components such as channel coding, modulation, equalization, synchronization, etc. Each component is optimized independently based on mathematical models that are simplified to arrive at closed form expressions. On the contrary, an autoencoder jointly optimizes the transmitter and the receiver as a whole. This joint optimization has the potential of providing a better performance than the traditional systems [1] on page 4-0 ,[2] on page 4-0 .

Traditional autoencoders are usually used to compress images, in other words remove redundancies in an image and reduce its dimension. A wireless communication system on the other hand uses channel coding and modulation techniques to add redundancy to the information bits. With this added redundancy, the system can recover the information bits that are impaired by the wireless channel. So, a wireless autoencoder actually adds redundancy and tries to minimize the number of errors in the received information for a given channel while learning to apply both channel coding and modulation in an unsupervised way.

Basic Autoencoder System

The following is the block diagram of a wireless auto encoder system. The encoder (transmitter) first maps k information bits into a message s such that $s \in \{1, \ldots, M\}$, where $M = 2^k$. Then message s is mapped to n real number to create $\mathbf{x} = f(s) \in \mathbb{R}^n$. The last layer of the encoder imposes constraints on $\mathbf{x}$ to further restrict the encoded symbols. The following are possible such constraints and are implemented using the normalization layer:

- Energy constraint: $\|x\|_2^2 \le n$
- Average power constraint: $\mathbb{E}[|x_i|^2] \le 1, \forall i$



Define the communication rate of this system as $R = k/n$ [bits/channel use], where (n,k) means that the system sends one of $M = 2^k$ messages using n channel uses. The channel impairs encoded (i.e. transmitted) symbols to generate $\mathbf{y} \in \mathbb{R}^n$. The decoder (i.e. receiver) produces an estimate, $\hat{s}$, of the transmitted message, s.

The input message is defined as a one-hot vector $\mathbf{1}_s \in \mathbb{R}^M$, which is defined as a vector whose elements are all zeros except the s^{th} one. The channel is additive white Gaussian noise (AWGN) that adds noise to achieve a given energy per bit to noise power density ratio, E_b/N_o.

Define a (7,4) autoencoder network with energy normalization and a training E_b/N_o of 3 dB. In [1] on page 4-0 , authors showed that two fully connected layers for both the encoder (transmitter) and the decoder (receiver) provides the best results with minimal complexity. Input layer (`featureInputLayer` (Deep Learning Toolbox)) accepts a one-hot vector of length M. The encoder has two fully connected layers (`fullyConnectedLayer` (Deep Learning Toolbox)). The first one has M inputs and M outputs and is followed by an ReLU layer (`reluLayer` (Deep Learning Toolbox)). The second fully connected layer has M inputs and n outputs and is followed by the normalization layer (helperAEWNormalizationLayer.m). The encoder layers are followed by the AWGN channel layer (helperAEWAWGNLayer.m). The output of the channel is passed to the decoder layers. The first decoder layer is a fully connected layer that has n inputs and M outputs and is followed by an ReLU layer. Second fully connected layer has M inputs and M outputs and is followed by a softmax layer (`softmaxLayer` (Deep Learning Toolbox)), which outputs the probability of each M symbols. The classification layer (`classificationLayer` (Deep Learning Toolbox)) outputs the most probable transmitted symbol from 0 to M-1.

```
k = 4;    % number of input bits
M = 2^k;  % number of possible input symbols
n = 7;    % number of channel uses
EbNo = 3; % Eb/No in dB

wirelessAutoencoder = [
  featureInputLayer(M,"Name","One-hot input","Normalization","none")

  fullyConnectedLayer(M,"Name","fc_1")
  reluLayer("Name","relu_1")

  fullyConnectedLayer(n,"Name","fc_2")

  helperAEWNormalizationLayer("Method", "Energy", "Name", "wnorm")

  helperAEWAWGNLayer("Name","channel",...
    "NoiseMethod","EbNo",...
    "EbNo",EbNo,...
    "BitsPerSymbol",2,...
    "SignalPower",1)

  fullyConnectedLayer(M,"Name","fc_3")
  reluLayer("Name","relu_2")

  fullyConnectedLayer(M,"Name","fc_4")
  softmaxLayer("Name","softmax")

  classificationLayer("Name","classoutput")]
```

```
wirelessAutoencoder =
  11×1 Layer array with layers:
```

```
     1   'One-hot input'   Feature Input            16 features
     2   'fc_1'            Fully Connected          16 fully connected layer
     3   'relu_1'          ReLU                     ReLU
     4   'fc_2'            Fully Connected          7 fully connected layer
     5   'wnorm'           Wireless Normalization   Energy normalization layer
     6   'channel'         AWGN Channel             AWGN channel with EbNo = 3
     7   'fc_3'            Fully Connected          16 fully connected layer
     8   'relu_2'          ReLU                     ReLU
     9   'fc_4'            Fully Connected          16 fully connected layer
    10   'softmax'         Softmax                  softmax
    11   'classoutput'     Classification Output    crossentropyex
```

The `helperAEWTrainWirelessAutoencoder.m` function defines such a network based on the (n,k), normalization method and the E_b/N_o values. The Wireless Autoencoder Training Function section on page 4-0 shows the contents of the `helperAEWTrainWirelessAutoencoder.m` function.

Train Autoencoder

Run the `helperAEWTrainWirelessAutoencoder.m` function to train a (2,2) autoencoder with energy normalization. This function uses the `trainingOptions` (Deep Learning Toolbox) function to select

- Adam (adaptive moment estimation) optimizer,
- Initial learning rate of 0.01,
- Maximum epochs of 15,
- Minibatch size of 20*M,
- Piecewise learning schedule with drop period of 10 and drop factor of 0.1.

Then, the `helperAEWTrainWirelessAutoencoder.m` function runs the `trainNetwork` (Deep Learning Toolbox) function to train the autoencoder network with the selected options. Finally, this function separates the network into encoder and decoder parts. Encoder starts with the input layer and ends after the normalization layer. Decoder starts after the channel layer and ends with the classification layer. A feature input layer is added at the beginning of the decoder.

Train the autoencoder with an E_b/N_o value that is low enough to result in some errors but not too low such that the training algorithm cannot extract any useful information from the received symbols, y. Set E_b/N_o to 3 dB.

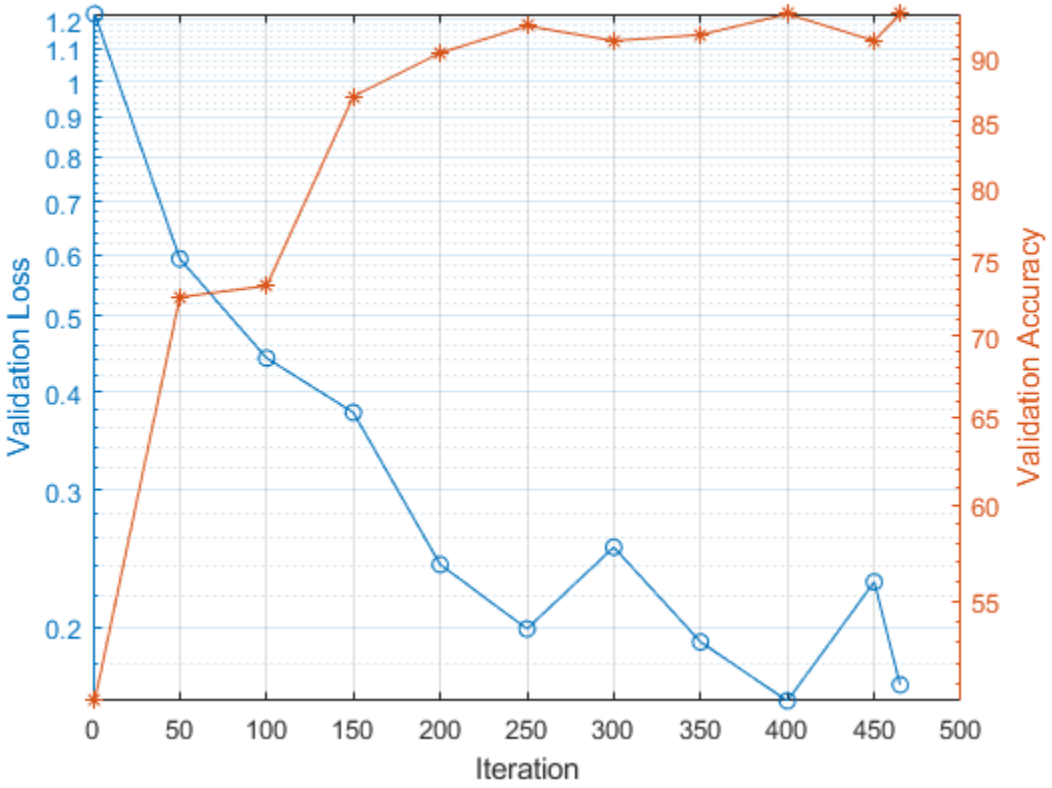
Training an autoencoder may take several minutes. Set `trainNow` to false to use saved networks.

```
trainNow = false; %#ok<*NASGU>

n = 2;                      % number of channel uses
k = 2;                      % bits per data symbol
EbNo = 3;                   % dB
normalization = "Energy";   % Normalization "Energy" | "Average power"

if trainNow
  [txNet22e,rxNet22e,info22e,wirelessAutoEncoder22e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo); %#ok<*UNRCH>
else
  load trainedNet_n2_k2_energy txNet rxNet info trainedNet
  txNet22e = txNet;
```

```
  rxNet22e = rxNet;
  info22e = info;
  wirelessAutoEncoder22e = trainedNet;
end
```

Plot the traning progress. The validation accuracy quickly reaches more than 90% while the validation loss keeps slowly decreasing. This behavior shows that the training E_b/N_o value was low enough to cause some errors but not too low to avoid convergence. For definitions of validation accuracy and validation loss, see “Monitor Deep Learning Training Progress” (Deep Learning Toolbox) section.

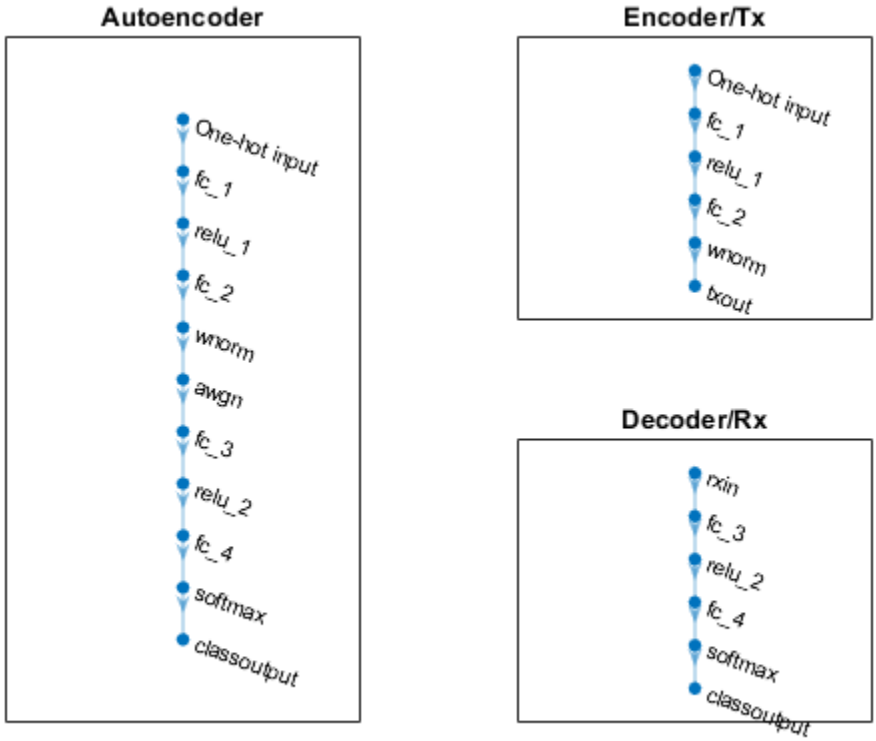
```
figure
helperAEWPlotTrainingPerformance(info22e)
```



Use the plot object function of the trained network objects to show the layer graphs of the full autoencoder, the encoder network, i.e. the transmitter, and the decoder network, i.e. the receiver.

```
figure
tiledlayout(2,2)
nexttile([2 1])
plot(wirelessAutoEncoder22e)
title('Autoencoder')
nexttile
plot(txNet22e)
title('Encoder/Tx')
nexttile
```

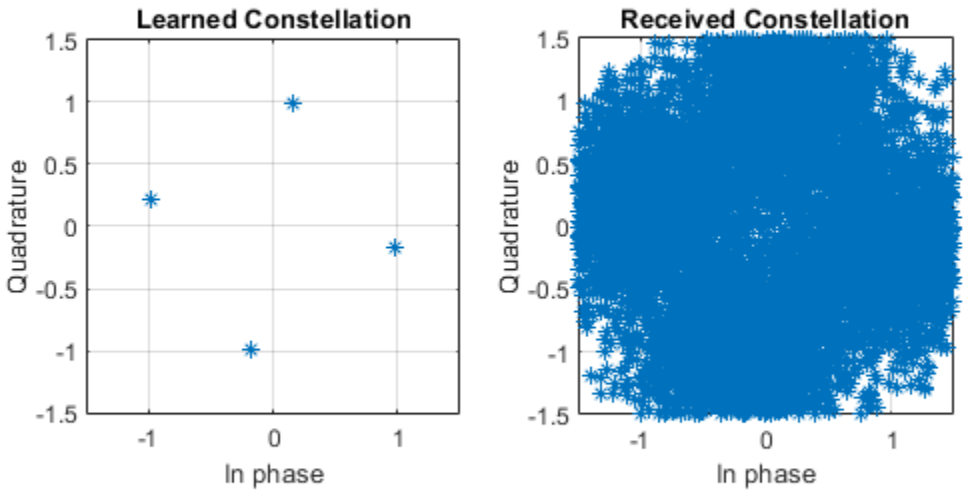
```
plot(rxNet22e)
title('Decoder/Rx')
```



Plot Transmitted and Received Constellation

Plot the constellation learned by the autoencoder to send symbols through the AWGN channel together with the received constellation. For a (2,2) configuration, autoencoder learns a QPSK ($M = 2^k = 4$) constellation with a phase rotation. The received constellation is basically the activation values at the output of the channel layer obtained using the `activations` (Deep Learning Toolbox) function and treated as interleaved complex numbers.

```
subplot(1,2,1)
helperAEWPlotConstellation(txNet22e)
title('Learned Constellation')
subplot(1,2,2)
helperAEWPlotReceivedConstellation(wirelessAutoEncoder22e)
title('Received Constellation')
```



Simulate BLER Performance

Simulate the block error rate (BLER) performance of the (2,2) autoencoder. Setup simulation parameters.

```
simParams.EbNoVec = 0:0.5:8;
simParams.MinNumErrors = 10;
simParams.MaxNumFrames = 300;
simParams.NumSymbolsPerFrame = 10000;
simParams.SignalPower = 1;
```

Generate random integers in the [0 M-1] range that represents k random information bits. Encode these information bits into complex symbols with `helperAEWEncode.m` function. The `helperAEWEncode` function runs the encoder part of the autoencoder then maps the real valued $\mathbf{x}$ vector into a complex valued $\mathbf{x_c}$ vector such that the odd and even elements are mapped into the in-phase and the quadrature component of a complex symbol, respectively, where $\mathbf{x}_c = \mathbf{x}(1:2:end) + j\mathbf{x}(2:2:end)$. In other words, treat the $\mathbf{x}$ array as an interleaved complex array.

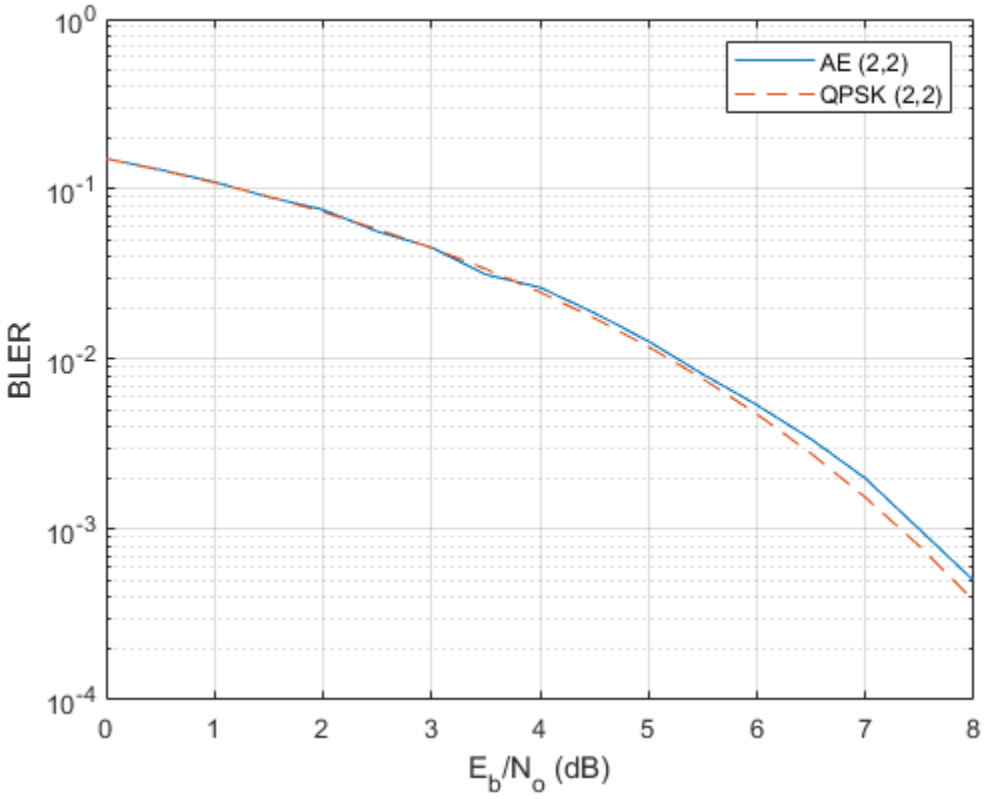
Pass the complex symbols through an AWGN channel. Decode the channel impaired complex symbols with the `helperAEWDecode.m` function. The following code runs the simulation for each E_b/N_o point for at least 10 block errors. To obtain more accurate results, increase minimum number of errors to at least 100. If Parallel Computing Toolbox™ is installed and a license is available, the simulation will run on a parallel pool. Compare the results with that of an uncoded QPSK system with block length 2.

```
EbNoVec = simParams.EbNoVec;
R = k/n;
```

```
M = 2^k;
BLER = zeros(size(EbNoVec));
parfor EbNoIdx = 1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx) + 10*log10(R);
  chan = comm.AWGNChannel("BitsPerSymbol",2, ...
    "EbNo", EbNo, "SamplesPerSymbol", 1, "SignalPower", 1);

  numBlockErrors = 0;
  frameCnt = 0;
  while (numBlockErrors < simParams.MinNumErrors) ...
      && (frameCnt < simParams.MaxNumFrames) %#ok<PFBNS>

    d = randi([0 M-1],simParams.NumSymbolsPerFrame,1);    % Random information bits
    x = helperAEWEncode(d,txNet22e);                      % Encoder
    y = chan(x);                                          % Channel
    dHat = helperAEWDecode(y,rxNet22e);                   % Decoder

    numBlockErrors = numBlockErrors + sum(d ~= dHat);
    frameCnt = frameCnt + 1;
  end
  BLER(EbNoIdx) = numBlockErrors / (frameCnt*simParams.NumSymbolsPerFrame);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
figure
semilogy(simParams.EbNoVec,BLER,'-')
hold on
qpsk22BLER = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^2;
semilogy(simParams.EbNoVec,qpsk22BLER,'--')
hold off
ylim([1e-4 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('AE (2,2)','QPSK (2,2)')
```



The well formed constellation together with the BLER results show that training for 15 epochs is enough to get a satisfactory convergence.

Compare Constellation Diagrams

Compare learned constellations of several autoencoders normalized to unit energy and unit average power. Train (2,4) autoencoder normalized to unit energy.

```
n = 2;      % number of channel uses
k = 4;      % bits per data symbol
EbNo = 3;   % dB
normalization = "Energy";
if trainNow
  [txNet24e,rxNet24e,info24e,wirelessAutoEncoder24e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n2_k4_energy txNet rxNet info trainedNet
  txNet24e = txNet;
  rxNet24e = rxNet;
  info24e = info;
  wirelessAutoEncoder24e = trainedNet;
end
```

Train (2,4) autoencoder normalized to unit average power.

```
n = 2;      % number of channel uses
k = 4;      % bits per data symbol
```

```
EbNo = 3;   % dB
normalization = "Average power";
if trainNow
  [txNet24p,rxNet24p,info24p,wirelessAutoEncoder24p] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n2_k4_power txNet rxNet info trainedNet
  txNet24p = txNet;
  rxNet24p = rxNet;
  info24p = info;
  wirelessAutoEncoder24p = trainedNet;
end
```

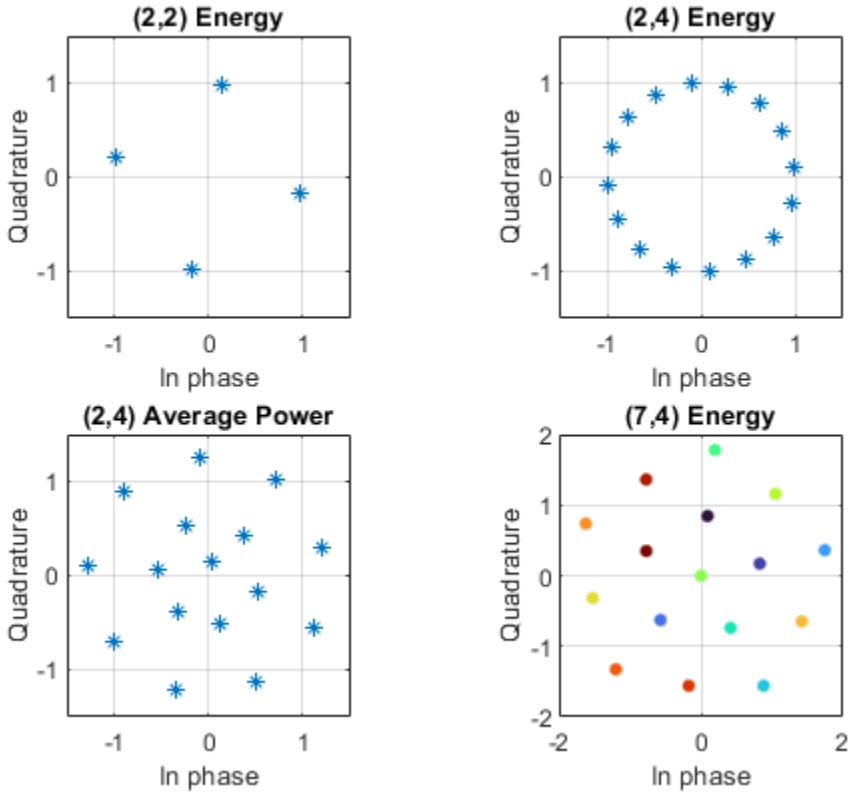
Train (7,4) autoencoder normalized to unit energy.

```
n = 7;      % number of channel uses
k = 4;      % bits per data symbol
EbNo = 3;   % dB
normalization = "Energy";
if trainNow
  [txNet74e,rxNet74e,info74e,wirelessAutoEncoder74e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n7_k4_energy txNet rxNet info trainedNet
  txNet74e = txNet;
  rxNet74e = rxNet;
  info74e = info;
  wirelessAutoEncoder74e = trainedNet;
end
```

Plot the constellation using the `helperAEWPlotConstellation.m` function. The trained (2,2) autoencoder converges on a QPSK constellation with a phase shift as the optimal constellation for the channel conditions experienced. The (2,4) autoencoder with energy normalization converges to a 16PSK constellation with a phase shift. Note that, energy normalization forces every symbol to have unit energy and places the symbols on the unit circle. Given this constraint, best constellation is a PSK constellation with equal angular distance between symbols. The (2,4) autoencoder with average power normalization converges to a three-tier constellation of 1-6-9 symbols. Average power normalization forces the symbols to have unity average power over time. This constraint results in an APSK constellation, which is different than the conventional QAM or APSK schemes. Note that, this network configuration may also converge to a two-tier constellation with 7-9 symbols based on the random initial condition used during training. The last plot shows the 2-D mapping of the 7-D constellation generated by the (7,4) autoencoder with energy constraint. 2-D mapping is obtained using the t-Distributed Stochastic Neighbor Embedding (t-SNE) method (see `tsne` (Statistics and Machine Learning Toolbox) function).

```
figure
subplot(2,2,1)
helperAEWPlotConstellation(txNet22e)
title('(2,2) Energy')
subplot(2,2,2)
helperAEWPlotConstellation(txNet24e)
title('(2,4) Energy')
subplot(2,2,3)
helperAEWPlotConstellation(txNet24p)
title('(2,4) Average Power')
subplot(2,2,4)
```

```
helperAEWPlotConstellation(txNet74e,'t-sne')
title('(7,4) Energy')
```

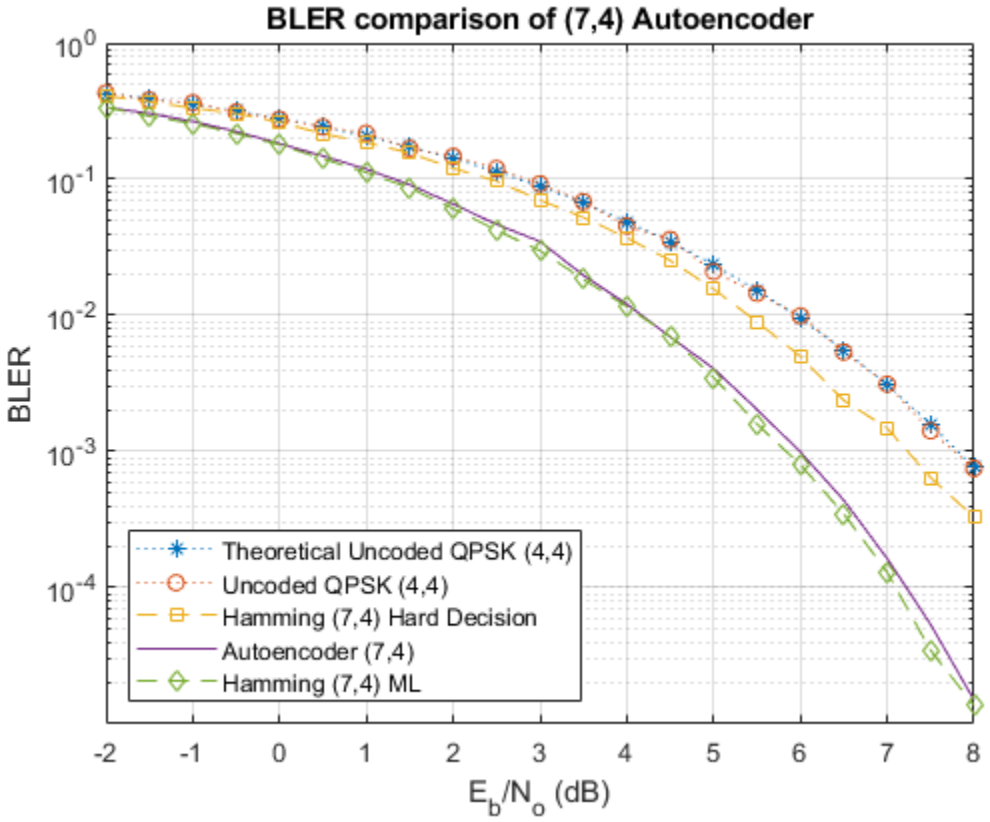


Compare BLER Performance of Autoencoders with Coded and Uncoded QPSK

Simulate the BLER performance of a (7,4) autoencoder with that of (7,4) Hamming code with QPSK modulation for both hard decision and maximum likelihood (ML) decoding. Use uncoded (4,4) QPSK as a baseline. (4,4) uncoded QPSK is basically a QPSK modulated system that sends blocks of 4 bits and measures BLER. The data for the following figures is obtained using helperAEWSimulateBLER.mlx and helperAEWPrepareAutoencoders.mlx files.

```
load codedBLERResults.mat
figure
qpsk44BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^4;
semilogy(simParams.EbNoVec,qpsk44BLERTh,':*')
hold on
semilogy(simParams.EbNoVec,qpsk44BLER,':o')
semilogy(simParams.EbNoVec,hammingHard74BLER,'--s')
semilogy(simParams.EbNoVec,ae74eBLER,'-')
semilogy(simParams.EbNoVec,hammingML74BLER,'--d')
hold off
ylim([1e-5 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('Theoretical Uncoded QPSK (4,4)','Uncoded QPSK (4,4)','Hamming (7,4) Hard Decision',...
```

```
  'Autoencoder (7,4)','Hamming (7,4) ML','Location','southwest')
title('BLER comparison of (7,4) Autoencoder')
```

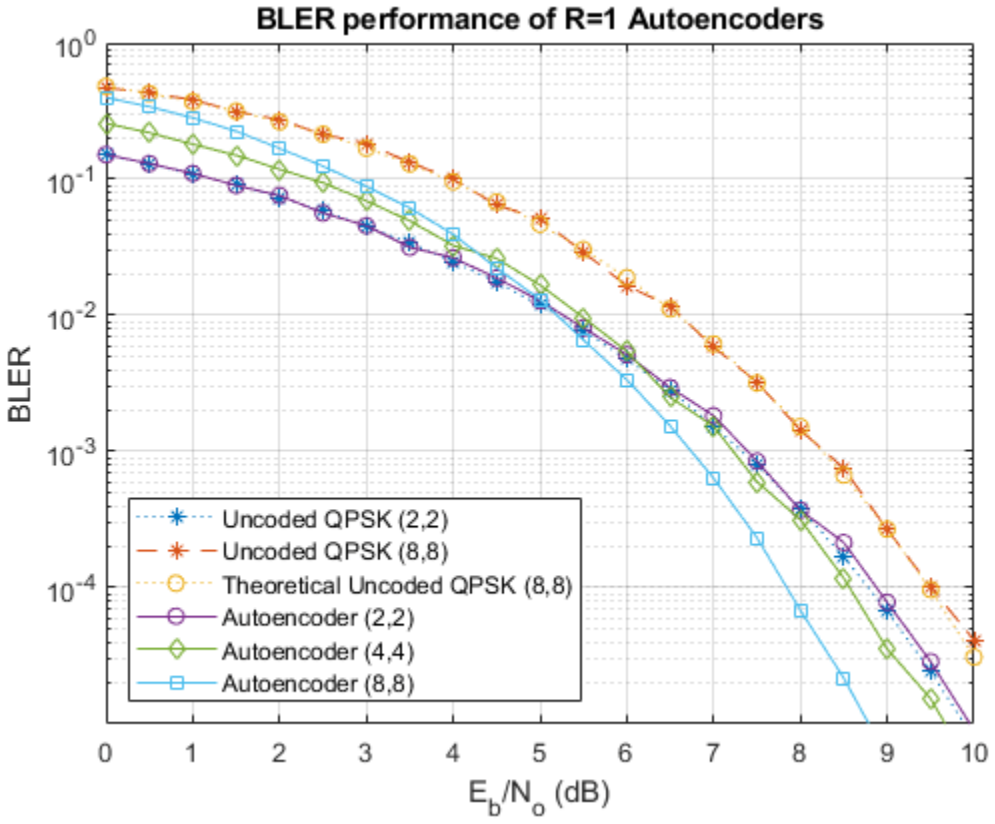


As expected, hard decision (7,4) Hamming code with QPSK modulation provides about 0.6 dB E_b/N_o advantage over uncoded QPSK, while the ML decoding of (7,4) Hamming code with QPSK modulation provides another 1.5 dB advantage for a BLER of 10^{-3}. The (7,4) autoencoder BLER performance approaches the ML decoding of (7,4) Hamming code, when trained with 3 dB E_b/N_o. This BLER performance shows that the autoencoder is able to learn not only modulation but also channel coding to achieve a coding gain of about 2 dB for a coding rate of R=4/7.

Next, simulate the BLER performance of autoencoders with R=1 with that of uncoded QPSK systems. Use uncoded (2,2) and (8,8) QPSK as baselines. Compare BLER performance of these systems with that of (2,2), (4,4) and (8,8) autoencoders.

```
load uncodedBLERResults.mat
qpsk22BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^2;
semilogy(simParams.EbNoVec,qpsk22BLERTh,':*')
hold on
semilogy(simParams.EbNoVec,qpsk88BLER,'--*')
qpsk88BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^8;
semilogy(simParams.EbNoVec,qpsk88BLERTh,':o')
semilogy(simParams.EbNoVec,ae22eBLER,'-o')
semilogy(simParams.EbNoVec,ae44eBLER,'-d')
semilogy(simParams.EbNoVec,ae88eBLER,'-s')
hold off
ylim([1e-5 1])
```

```
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('Uncoded QPSK (2,2)','Uncoded QPSK (8,8)','Theoretical Uncoded QPSK (8,8)',...
  'Autoencoder (2,2)','Autoencoder (4,4)','Autoencoder (8,8)','Location','southwest')
title('BLER performance of R=1 Autoencoders')
```



Bit error rate of QPSK is the same for both (8,8) and (2,2) cases. However, the BLER depends on the block length, n, and gets worse as n increases as given by $BLER = 1 - (1 - BER)^n$. As expected, BLER performance of (8,8) QPSK is worse than the (2,2) QPSK system. The BLER performance of (2,2) autoencoder matches the BLER performance of (2,2) QPSK. On the other hand, (4,4) and (8,8) autoencoders optimize the channel coder and the constellation jointly to obtain a coding gain with respect to the corresponding uncoded QPSK systems.

Effect of Training Eb/No on BLER Performance

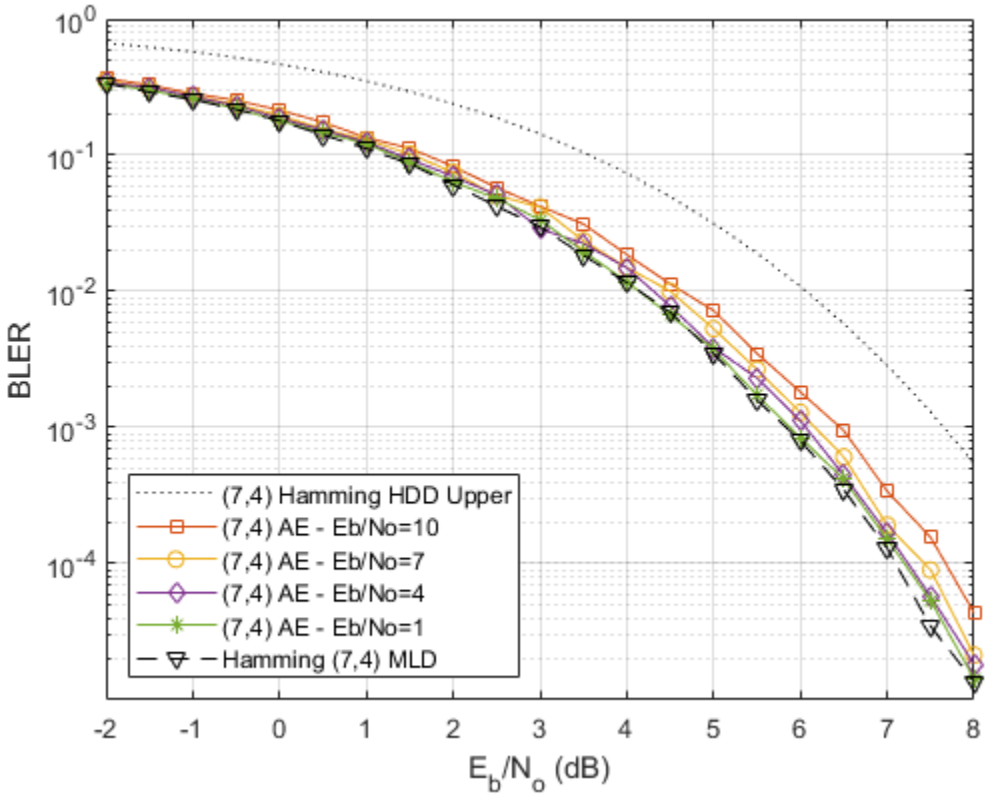
Train the (7,4) autoencoder with energy normalization under different E_b/N_o values and compare the BLER performance.

```
n = 7;
k = 4;
normalization = 'Energy';

EbNoVec = 1:3:10;
if trainNow
  for EbNoIdx = 1:length(EbNoVec)
    EbNo = EbNoVec(EbNoIdx);
```

```
    [txNetVec{EbNoIdx},rxNetVec{EbNoIdx},infoVec{EbNoIdx},trainedNetVec{EbNoIdx}] = ...
      helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
    BLERVec{EbNoIdx} = helperAEWAutoencoderBLER(txNetVec{EbNoIdx},rxNetVec{EbNoIdx},simParams);
  end
else
  load ae74TrainedEbNo1to10 BLERVec trainParams simParams txNetVec rxNetVec infoVec trainedNetVe
end
```

Plot the BLER performance together with theoretical upper bound for hard decision decoded Hamming (7,4) code and simulated BLER of maximum likelihood decoded (MLD) Hamming (7,4) code. The BLER performance of the (7,4) autoencoder gets closer to the Hamming (7,4) code with MLD as the training E_b/N_o decreases from 10 dB to 1 dB, at which point it almost matches the MLD Hamming (7,4) code.

```
berHamming = bercoding(simParams.EbNoVec,'hamming','hard',7);
blerHamming = 1-(1-berHamming).^7;
load codedBLERResults hammingML74BLER
figure
semilogy(simParams.EbNoVec,blerHamming,':k')
hold on
linespec = {'-*','-d','-o','-s',};
for EbNoIdx=length(EbNoVec):-1:1
  semilogy(simParams.EbNoVec,BLERVec{EbNoIdx},linespec{EbNoIdx})
end
semilogy(simParams.EbNoVec,hammingML74BLER,'--vk')
hold off
ylim([1e-5 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('(7,4) Hamming HDD Upper','(7,4) AE - Eb/No=10','(7,4) AE - Eb/No=7',...
  '(7,4) AE - Eb/No=4','(7,4) AE - Eb/No=1','Hamming (7,4) MLD','location','southwest')
```



Conclusions and Further Exploration

The BLER results show that it is possible for autoencoders to learn joint coding and modulation schemes in an unsupervised way. It is even possible to train an autoencoder with R=1 to obtain a coding gain as compared to traditional methods. The example also shows the effect of hyperparameters such as E_b/N_o on the BLER performance.

The results are obtained using the following default settings for training and BLER simulations:

```
trainParams.Plots = 'none';
trainParams.Verbose = false;
trainParams.MaxEpochs = 15;
trainParams.InitialLearnRate = 0.01;
trainParams.LearnRateSchedule = 'piecewise';
trainParams.LearnRateDropPeriod = 10;
trainParams.LearnRateDropFactor = 0.1;
trainParams.MiniBatchSize = 20*2^k;

simParams.EbNoVec = -2:0.5:8;
simParams.MinNumErrors = 100;
simParams.MaxNumFrames = 300;
simParams.NumSymbolsPerFrame = 10000;
simParams.SignalPower = 1;
```

Vary these parameters to train different autoencoders and test their BLER performance. Experiment with different n, k, normalization and E_b/N_o values. See the help for

helperAEWTrainWirelessAutoencoder.m, helperAEWPrepareAutoencoders.mlx and helperAEWAutoencoderBLER.m for more information.

List of helper functions

- helperAEWAWGNLayer.m
- helperAEWNormalizationLayer.m
- helperAEWEncode.m
- helperAEWDecode.m
- helperAEWTrainWirelessAutoencoder.m
- helperAEWPlotConstellation.m
- helperAEWPlotTrainingPerformance.m
- helperAEWAutoencoderBLER.m
- helperAEWPrepareAutoencoders.mlx
- helperAEWSimulateBLER.mlx
- helperAEWPlotReceivedConstellation.m

Wireless Autoencoder Training Function

This section shows the content of the helperAEWTrainWirelessAutoencoder function. To open the runnable version of the function in the MATLAB editor, click helperAEWTrainWirelessAutoencoder.m.

```
type helperAEWTrainWirelessAutoencoder
```

```
function [txNet,rxNet,info,trainedNet] = ...
  helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo,varargin)
%helperAEWTrainWirelessAutoencoder Train wireless autoencoder
%   [TX,RX,INFO,AE] = helperAEWTrainWirelessAutoencoder(N,K,NORM,EbNo)
%   trains an autoencoder, AE, with (N,K), where K is the number of input
%   bits and N is the number of channel uses. The autoencoder employs NORM
%   normalization. NORM must be one of 'Energy' and 'Average power'. The
%   channel is an AWGN channel with Eb/No set to EbNo. TX and Rx are the
%   encoder and decoder parts of the autoencoder that can be used in the
%   helperAEWEncoder and helperAEWDecoder functions, respectively. INFO is
%   the training information that can be used to check the convergence
%   behavior of the training process.
%
%   [TX,RX,INFO,AE] = helperAEWTrainWirelessAutoencoder(...,TP) provides
%   training parameters as follows:
%     TP.Plots: Plots to display during network training defined as one of
%               'none' (default) or 'training-progress'.
%     TP.Verbose: Indicator to display training progress information
%               defined as 1 (true) (default) or 0 (false).
%     TP.MaxEpochs: Maximum number of epochs defined as a positive integer.
%               The default is 15.
%     TP.InitialLearnRate: Initial learning rate as a floating point number
%               between 0 and 1. The default is 0.01;
%     TP.LearnRateSchedule: Learning rate schedule defined as one of
%               'piecewise' (default) or 'none'.
%     TP.LearnRateDropPeriod: Number of epochs for dropping the learning
%               rate as a positive integer. The default is 10.
%     TP.LearnRateDropFactor: Factor for dropping the learning rate,
%               defined as a scalar between 0 and 1. The default is 0.1.
```

```
%     TP.MiniBatchSize: Size of the mini-batch to use for each training
%               iteration, defined as a positive integer. The default is
%               20*M.
%
%   See also AutoencoderForWirelessCommunicationsExample, helperAEWEncode,
%   helperAEWDecode, helperAEWNormalizationLayer, helperAEWAWGNLayer.

%   Copyright 2020 The MathWorks, Inc.

% Derived parameters
M = 2^k;
R = k/n;

if nargin > 4
  trainParams = varargin{1};
else
  % Set default training options. Set maximum epochs to 15. SGD requires a
  % representative mini-batch that has enough symbols to achieve
  % convergence. Therefore, increase the mini-batch size with M. Set the
  % initial learning rate to 0.01 and reduce the learning rate by a factor
  % of 10 every 10 epochs. Do not plot or print training progress.
  trainParams.MaxEpochs = 15;
  trainParams.MiniBatchSize = 20*M;
  trainParams.InitialLearnRate = 0.01;
  trainParams.LearnRateSchedule = 'piecewise';
  trainParams.LearnRateDropPeriod = 10;
  trainParams.LearnRateDropFactor = 0.1;
  trainParams.Plots = 'none';
  trainParams.Verbose = false;
end

% Convert Eb/No to channel Eb/No values using the code rate
EbNoChannel = EbNo + 10*log10(R);

% As the number of possible input symbols increase, we need to increase the
% number of training symbols to give the network a chance to experience a
% large number of possible input combinations. The same is true for number
% of validation symbols.
numTrainSymbols = 2500 * M;
numValidationSymbols = 100 * M;

% Define autoencoder network. Input is a one-hot vector of length M. The
% encoder has two fully connected layers. The first one has M inputs and M
% outputs and is followed by an ReLU layer. The second fully connected
% layer has M inputs and n outputs and is followed by the normalization
% layer. Normalization layer imposes constraints on the encoder output and
% available methods are energy and average power normalization. The encoder
% layers are followed by the AWGN channel layer. Set BitsPerSymbol to 2
% since two output values are mapped onto a complex symbol. Set the signal
% power to 1 since the normalization layer outputs signals with unity
% power. The output of the channel is passed to the decoder layers. The
% first decoder layer is a fully connected layer that has n inputs and M
% outputs and is followed by an ReLU layer. Second fully connected layer
% has M inputs and M outputs and is followed by a softmax layer. The output
% of the decoder is chosen as the most probable transmitted symbol from 0
% to M-1.
wirelessAutoEncoder = [
  featureInputLayer(M,"Name","One-hot input","Normalization","none")
```

```
  fullyConnectedLayer(M,"Name","fc_1")
  reluLayer("Name","relu_1")

  fullyConnectedLayer(n,"Name","fc_2")

  helperAEWNormalizationLayer("Method", normalization)

  helperAEWAWGNLayer("NoiseMethod","EbNo",...
    "EbNo",EbNoChannel,...
    "BitsPerSymbol",2,...
    "SignalPower",1)

  fullyConnectedLayer(M,"Name","fc_3")
  reluLayer("Name","relu_2")

  fullyConnectedLayer(M,"Name","fc_4")
  softmaxLayer("Name","softmax")

  classificationLayer("Name","classoutput")];

% Generate random training data. Create one-hot input vectors and labels.
d = randi([0 M-1],numTrainSymbols,1);
trainSymbols = zeros(numTrainSymbols,M);
trainSymbols(sub2ind([numTrainSymbols, M],...
  (1:numTrainSymbols)',d+1)) = 1;
trainLabels = categorical(d);

% Generate random validation data. Create one-hot input vectors and labels.
d = randi([0 M-1],numValidationSymbols,1);
validationSymbols = zeros(numValidationSymbols,M);
validationSymbols(sub2ind([numValidationSymbols, M],...
  (1:numValidationSymbols)',d+1)) = 1;
validationLabels = categorical(d);

% Set training options
options = trainingOptions('adam', ...
  'InitialLearnRate',trainParams.InitialLearnRate, ...
  'MaxEpochs',trainParams.MaxEpochs, ...
  'MiniBatchSize',trainParams.MiniBatchSize, ...
  'Shuffle','every-epoch', ...
  'ValidationData',{validationSymbols,validationLabels}, ...
  'LearnRateSchedule', trainParams.LearnRateSchedule, ...
  'LearnRateDropPeriod', trainParams.LearnRateDropPeriod, ...
  'LearnRateDropFactor', trainParams.LearnRateDropFactor, ...
  'Plots', trainParams.Plots, ...
  'Verbose', trainParams.Verbose);

% Train the autoencoder network
[trainedNet,info] = trainNetwork(trainSymbols,trainLabels,wirelessAutoEncoder,options);

% Separate the network into encoder and decoder parts. Encoder starts with
% the input layer and ends after the normalization layer.
for idxNorm = 1:length(trainedNet.Layers)
  if isa(trainedNet.Layers(idxNorm), 'helperAEWNormalizationLayer')
    break
  end
end
```

```
lgraph = addLayers(layerGraph(trainedNet.Layers(1:idxNorm)), ...
  regressionLayer('Name', 'txout'));
lgraph = connectLayers(lgraph,'wnorm','txout');
txNet = assembleNetwork(lgraph);

% Decoder starts after the channel layer and ends with the classification
% layer. Add a feature input layer at the beginning.
for idxChan = idxNorm:length(trainedNet.Layers)
  if isa(trainedNet.Layers(idxChan), 'helperAEWAWGNLayer')
    break
  end
end
firstLayerName = trainedNet.Layers(idxChan+1).Name;
n = trainedNet.Layers(idxChan+1).InputSize;
lgraph = addLayers(layerGraph(featureInputLayer(n,'Name','rxin')), ...
  trainedNet.Layers(idxChan+1:end));
lgraph = connectLayers(lgraph,'rxin',firstLayerName);
rxNet = assembleNetwork(lgraph);
```

References

[1] T. O’Shea and J. Hoydis, "An Introduction to Deep Learning for the Physical Layer," in IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563-575, Dec. 2017, doi: 10.1109/TCCN.2017.2758370.

[2] S. Dörner, S. Cammerer, J. Hoydis and S. t. Brink, "Deep Learning Based Communication Over the Air," in *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 132-143, Feb. 2018, doi: 10.1109/JSTSP.2017.2784180.

See Also

`classificationLayer` | `featureInputLayer` | `fullyConnectedLayer` | `reluLayer` | `softmaxLayer`

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Training and Testing a Neural Network for LLR Estimation

This example shows how to generate signals and channel impairments to train a neural network, called LLRNet, to estimate exact log likelihood ratios (LLR).

Most modern communication systems, such as 5G New Radio (NR) and Digital Video Broadcasting for Satellite, Second Generation (DVB-S.2) use forward error correction algorithms that benefit from soft demodulated bit values. These systems calculate soft bit values using the LLR approach. LLR is defined as the log of the ratio of probability of a bit to be 0 to the probability of a bit to be 1 or

$$l_i \triangleq \log\left(\frac{P_r(c_i = 0 \,|\, \hat{s})}{P_r(c_i = 1 \,|\, \hat{s})}\right),\ i = 1, \ldots, k$$

where $\hat{s}$ is an k-bit received symbol, and c_i is the i^{th}bit of the symbol. Assuming an additive white Gaussian noise (AWGN) channel, the exact computation of the LLR expression is

$$l_i \triangleq \log\left(\frac{\sum_{s \in c_i^0} \exp\left(-\frac{\|\hat{s}-s\|_2^2}{\sigma^2}\right)}{\sum_{s \in c_i^1} \exp\left(-\frac{\|\hat{s}-s\|_2^2}{\sigma^2}\right)}\right)$$

where σ^2 is the noise variance. Exponential and logarithmic calculations are very costly especially in embedded systems. Therefore, most practical systems use the max-log approximation. For a given array x, the max-log approximation is

$$\log\left(\sum_j \exp\left(-x_j^2\right)\right) \approx \max_j\left(-x_j^2\right).$$

Substituting this in the exact LLR expression results in the max-log LLR approximation [1] on page 4-0

$$l_i \approx \frac{1}{\sigma^2}\left(\min_{s \in C_i^1} \|\hat{s}-s\|_2^2 - \min_{s \in C_i^0} \|\hat{s}-s\|_2^2\right).$$

LLRNet uses a neural network to estimate the exact LLR values given the baseband complex received symbol for a given SNR value. A shallow network with a small number of hidden layers has the potential to estimate the exact LLR values at a complexity similar to the approximate LLR algorithm [1] on page 4-0 .

Compare Exact LLR, Max-Log Approximate LLR and LLRNet for M-ary QAM

5G NR uses M-ary QAM modulation. This section explores the accuracy of LLRNet in estimating the LLR values for 16-, 64-, and 256-QAM modulation. Assume an M-ary QAM system that operates under AWGN channel conditions. This assumption is valid even when the channel is frequency selective but symbols are equalized. The following shows calculated LLR values for the following three algorithms:

- Exact LLR
- Max-log approximate LLR

- LLRNet

16-QAM LLR Estimation Performance

Calculate exact and approximate LLR values for symbol values that cover the 99.7% ($\pm 3\sigma$) of the possible received symbols. Assuming AWGN, 99.7% ($\pm 3\sigma$) of the received signals will be in the range $\left[\max_{s \in C}(Re(s) + 3\sigma) \min_{s \in C}(Re(s) - 3\sigma)\right] + i\left[\max_{s \in C}(Im(s) + 3\sigma) \min_{s \in C}(Im(s) - 3\sigma)\right]$. Generate uniformly distributed I/Q symbols over this space and use `qamdemod` function to calculate exact LLR and approximate LLR values.

```
M = 16;             % Modulation order
k = log2(M);        % Bits per symbols
SNRValues = -5:5:5; % in dB
numSymbols = 1e4;
numSNRValues = length(SNRValues);
symOrder = llrnetQAMSymbolMapping(M);

const = qammod(0:15,M,symOrder,'UnitAveragePower',1);
maxConstReal = max(real(const));
maxConstImag = max(imag(const));

numBits = numSymbols*k;
exactLLR = zeros(numBits,numSNRValues);
approxLLR = zeros(numBits,numSNRValues);
rxSym = zeros(numSymbols,numSNRValues);
for snrIdx = 1:numSNRValues
    SNR = SNRValues(snrIdx);
    noiseVariance = 10^(-SNR/10);
    sigma = sqrt(noiseVariance);

    maxReal = maxConstReal + 3*sigma;
    minReal = -maxReal;
    maxImag = maxConstImag + 3*sigma;
    minImag = -maxImag;

    r = (rand(numSymbols,1)*(maxReal-minReal)+minReal) + ...
        1i*(rand(numSymbols,1)*(maxImag-minImag)+minImag);
    rxSym(:,snrIdx) = r;

    exactLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','llr','NoiseVariance',noiseVariance);
    approxLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','approxllr','NoiseVariance',noiseVariance);
end
```

Set up and Train Neural Network

Set up a shallow neural network with one input layer, one hidden layer, and one output layer. Input a received symbol to the network and train it to estimate the exact LLR values. Since the network expects real inputs, create a two column vector, where the first column is the real values of the received symbol and the second column is the imaginary values of the received symbol. Also, the output must be a $k \times N$ vector, where k is the number of bits per symbol and N is the number of symbols.

```
nnInput = zeros(numSymbols,2,numSNRValues);
nnOutput = zeros(numSymbols,k,numSNRValues);
```

```
for snrIdx = 1:numSNRValues
    rxTemp = rxSym(:,snrIdx);
    rxTemp = [real(rxTemp) imag(rxTemp)];
    nnInput(:,:,snrIdx) = rxTemp;

    llrTemp = exactLLR(:,snrIdx);
    nnOutput(:,:,snrIdx) = reshape(llrTemp, k, numSymbols)';
end
```

For 16-QAM symbols, the hidden layer has 8 neurons and the output layer has 4 neurons, which corresponds to the number of bits per symbol. The llrnetNeuralNetwork function returns preconfigured neural network. Train the neural network for three different SNR values. Use the exact LLR values calculated using the `qamdemod` function as the expected output values.

```
hiddenLayerSize = 8;
trainedNetworks = cell(1,numSNRValues);
for snrIdx=1:numSNRValues
    fprintf('Training neural network for SNR = %1.1fdB\n', ...
        SNRValues(snrIdx))
    x = nnInput(:,:,snrIdx)';
    y = nnOutput(:,:,snrIdx)';

    MSExactLLR = mean(y(:).^2);
    fprintf('\tMean Square LLR = %1.2f\n', MSExactLLR)

    % Train the Network. Use parallel pool, if available. Train three times
    % and pick the best one.
    mse = inf;
    for p=1:3
        netTemp = llrnetNeuralNetwork(hiddenLayerSize);
        if parallelComputingLicenseExists()
            [netTemp,tr] = train(netTemp,x,y,'useParallel','yes');
        else
            [netTemp,tr] = train(netTemp,x,y);
        end
        % Test the Network
        predictedLLRSNR = netTemp(x);
        mseTemp = perform(netTemp,y,predictedLLRSNR);
        fprintf('\t\tTrial %d: MSE = %1.2e\n', p, mseTemp)
        if mse > mseTemp
            mse = mseTemp;
            net = netTemp;
        end
    end

    % Store the trained network
    trainedNetworks{snrIdx} = net;
    fprintf('\tBest MSE = %1.2e\n', mse)
end
```

```
Training neural network for SNR = -5.0dB

    Mean Square LLR = 4.42

        Trial 1: MSE = 1.95e-06
        Trial 2: MSE = 1.22e-04
        Trial 3: MSE = 4.54e-06

    Best MSE = 1.95e-06
```

```
Training neural network for SNR = 0.0dB

    Mean Square LLR = 15.63

        Trial 1: MSE = 1.90e-03
        Trial 2: MSE = 5.03e-03
        Trial 3: MSE = 8.95e-05

    Best MSE = 8.95e-05

Training neural network for SNR = 5.0dB

    Mean Square LLR = 59.29

        Trial 1: MSE = 2.25e-02
        Trial 2: MSE = 2.23e-02
        Trial 3: MSE = 7.40e-02

    Best MSE = 2.23e-02
```

Performance metric for this network is mean square error (MSE). The final MSE values show that the neural network converges to an MSE value that is at least 40 dB less than the mean square exact LLR values. Note that, as SNR increases so do the LLR values, which results in relatively higher MSE values.

Results for 16-QAM

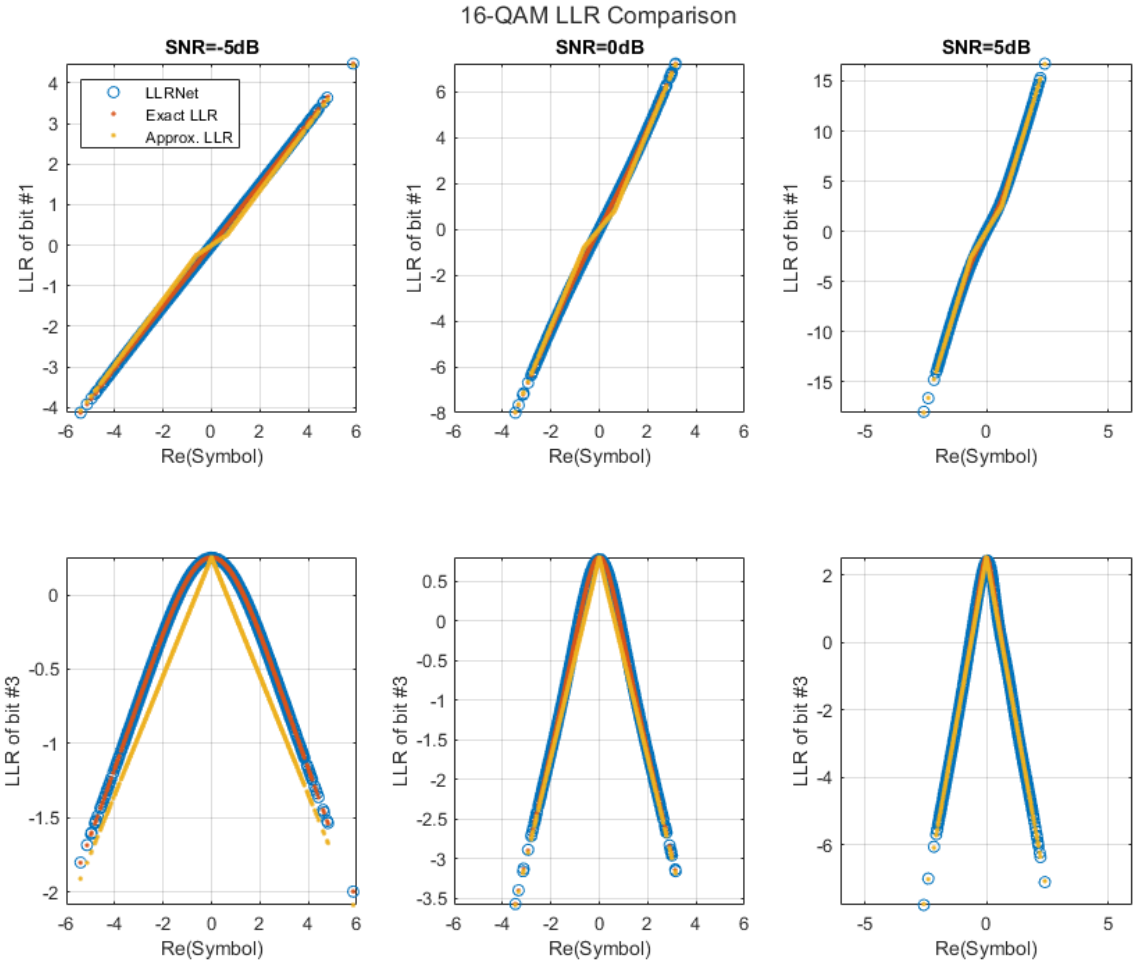
Compare the LLR estimates of LLRNet to that of exact LLR and approximate LLR. Simulate 1e4 16-QAM symbols and calculate LLR values using all three methods. Do not use the symbols that we generated in the previous section so as not to give LLRNet an unfair advantage, since those symbols were used to train the LLRNet.

```
numBits = numSymbols*k;
d = randi([0 1], numBits, 1);

txSym = qammod(d,M,symOrder,'InputType','bit','UnitAveragePower',1);

exactLLR = zeros(numBits,numSNRValues);
approxLLR = zeros(numBits,numSNRValues);
predictedLLR = zeros(numBits,numSNRValues);
rxSym = zeros(length(txSym),numSNRValues);
for snrIdx = 1:numSNRValues
    SNR = SNRValues(snrIdx);
    sigmas = 10^(-SNR/10);
    r = awgn(txSym,SNR);
    rxSym(:,snrIdx) = r;

    exactLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','llr','NoiseVariance',sigmas);
    approxLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','approxllr','NoiseVariance',sigmas);

    net = trainedNetworks{snrIdx};
    x = [real(r) imag(r)]';
    tempLLR = net(x);
    predictedLLR(:,snrIdx) = reshape(tempLLR, numBits, 1);
end

qam16Results.exactLLR = exactLLR;
```

```
qam16Results.approxLLR = approxLLR;
qam16Results.predictedLLR = predictedLLR;
qam16Results.RxSymbols = rxSym;
qam16Results.M = M;
qam16Results.SNRValues = SNRValues;
qam16Results.HiddenLayerSize = hiddenLayerSize;
qam16Results.NumSymbols = numSymbols;
```

The following figure shows exact LLR, max-log approximate LLR, and LLRNet estimate of LLR values versus the real part of the received symbol for odd bits. LLRNet matches the exact LLR values even for low SNR values.

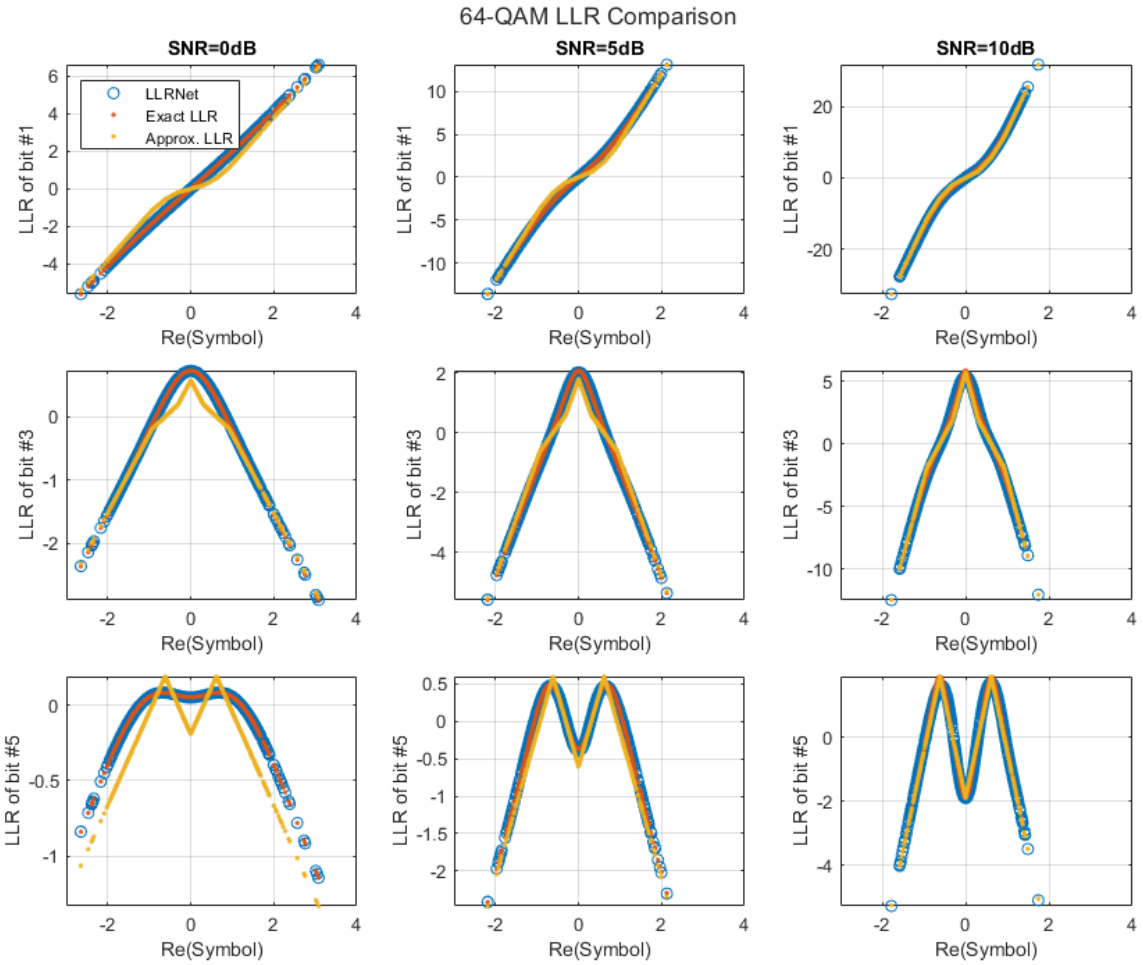
```
llrnetPlotLLR(qam16Results,'16-QAM LLR Comparison')
```

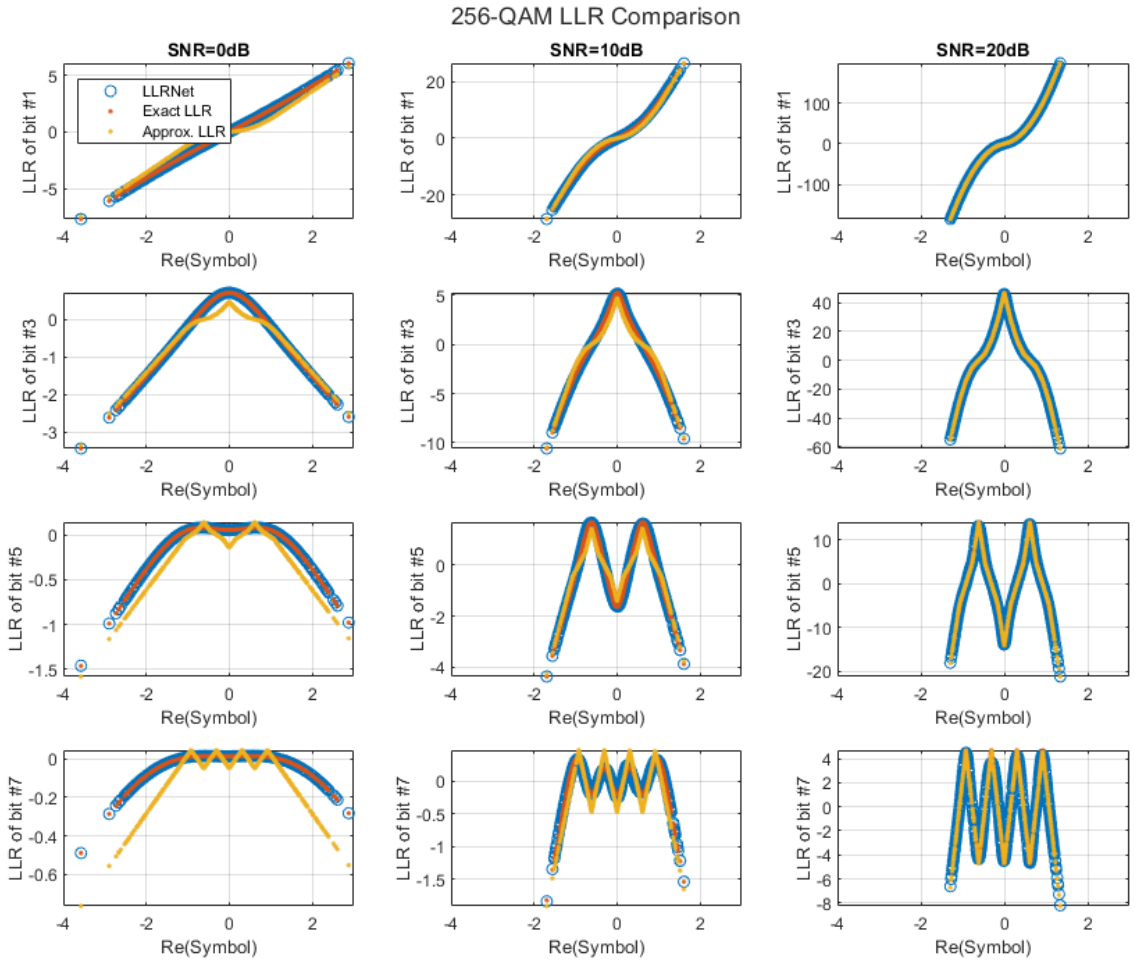


64-QAM and 256-QAM LLR Estimation Performance

Check if the LLRNet can estimate the LLR values for higher order QAM. Repeat the same process you followed for 16-QAM for 64-QAM and 256-QAM using the llrnetQAMLLR helper function. The following figures show exact LLR, max-log approximate LLR, and LLRNet estimate of LLR values versus the real part of the received symbol for odd bits.

```
trainNow = Show saved results;
if trainNow
    % Parameters for 64-QAM
    simParams(1).M = 64; %#ok<UNRCH>
    simParams(1).SNRValues = 0:5:10;
    simParams(1).HiddenLayerSize = 16;
    simParams(1).NumSymbols = 1e4;
    simParams(1).UseReLU = false;

    % Parameters for 256-QAM
    simParams(2).M = 256;
    simParams(2).SNRValues = 0:10:20;
    simParams(2).HiddenLayerSize = 32;
    simParams(2).NumSymbols = 1e4;
    simParams(2).UseReLU = false;

    simResults = llrnetQAMLLR(simParams);
    llrnetPlotLLR(simResults(1),sprintf('%d-QAM LLR Comparison',simResults(1).M))
    llrnetPlotLLR(simResults(2),sprintf('%d-QAM LLR Comparison',simResults(2).M))
else
    load('llrnetQAMPerformanceComparison.mat', 'simResults')
    for p=1:length(simResults)
        llrnetPlotLLR(simResults(p),sprintf('%d-QAM LLR Comparison',simResults(p).M))
    end
end
```


64-QAM LLR Comparison
SNR=0dB
SNR=5dB
SNR=10dB
LLRNet
Exact LLR
Approx. LLR
LLR of bit #1
LLR of bit #3
LLR of bit #5
Re(Symbol)



DVB-S.2 Packet Error Rate

DVB-S.2 system uses a soft demodulator to generate inputs for the LDPC decoder. Simulate the packet error rate (PER) of a DVB-S.2 system with 16-APSK modulation and 2/3 LDPC code using exact LLR, approximate LLR, and LLRNet using llrNetDVBS2PER function. This function uses the `comm.PSKDemodulator` System object and the `dvbsapskdemod` function to calculate exact and approximate LLR values and the `comm.AWGNChannel` System object to simulate the channel.

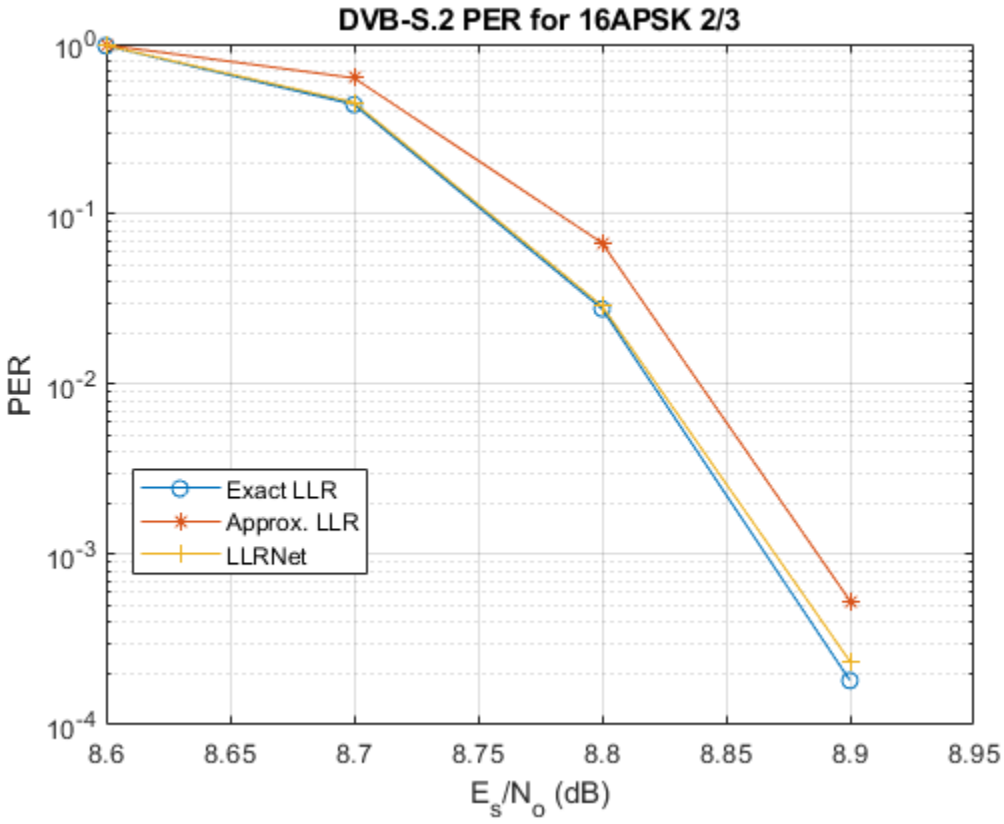
Set `simulateNow` to true (or select "Simulate" in the dropdown) to run the PER simulations for the values of `subsystemType`, `EsNoValues`, and `numSymbols` using the llrnetDVBS2PER function. If Parallel Computing Toolbox™ is installed, this function uses the `parfor` command to run the simulations in parallel. On an Intel® Xeon® W-2133 CPU @ 3.6GHz and running a “Run Code on Parallel Pools” (Parallel Computing Toolbox) of size 6, the simulation takes about 40 minutes. Set `simulateNow` to `false` (or select "Plot saved results" in the dropdown), to load the PER results for the values of `subsystemType='16APSK 2/3'`, `EsNoValues=8.6:0.1:8.9`, and `numSymbols=10000`.

Set `trainNow` to `true` (or select "Train LLRNet" in the dropdown) to train LLR neural networks for each value of `EsNoValues`, for the given `subsystemType` and `numSymbols`. If Parallel Computing

Toolbox™ is installed, the `train` function can be called with the optional name-value pair `'useParallel'` set to `'yes'` to run the simulations in parallel. On an Intel® Xeon® W-2133 CPU @ 3.6GHz and running a “Run Code on Parallel Pools” (Parallel Computing Toolbox) of size 6, the simulation takes about 21 minutes. Set `trainNow` to false (or select "Use saved networks" in the dropdown) to load LLR neural networks trained for `subsystemType='16APSK 2/3'`, `EsNoValues=8.6:0.1:8.9`.

For more information on the DVB-S.2 PER simulation, see the “DVB-S.2 Link, Including LDPC Coding in Simulink” on page 8-372 example. For more information on training the network, refer to the llrnetTrainDVBS2LLRNetwork function and [1] on page 4-0 .

```
simulateNow = Plot saved results;
if simulateNow
    subsystemType = '16APSK 2/3'; %#ok<UNRCH>
    EsNoValues = 8.6:0.1:8.9;     % in dB
    numFrames = 10000;
    numErrors = 200;

    trainNow = Use saved networks;
    if trainNow && (~strcmp(subsystemType,'16APSK 2/3') || ~isequal(EsNoValues,8.6:0.1:9))
        % Train the networks for each EsNo value
        numTrainSymbols = 1e4;
        hiddenLayerSize = 64;
        llrNets = llrnetTrainDVBS2LLRNetwork(subsystemType, EsNoValues, numTrainSymbols, hiddenLa
    else
        load('llrnetDVBS2Networks','llrNets','subsystemType','EsNoValues');
    end

    % Simulate PER with exact LLR, approximate LLR, and LLRNet
    [perLLR,perApproxLLR,perLLRNet] = llrnetDVBS2PER(subsystemType,EsNoValues,llrNets,numFrames,n
    llrnetPlotLLRvsEsNo(perLLR,perApproxLLR,perLLRNet,EsNoValues,subsystemType)
else
    load('llrnetDVBS2PERResults.mat','perApproxLLR','perLLR','perLLRNet',...
        'subsystemType','EsNoValues');
    llrnetPlotLLRvsEsNo(perLLR,perApproxLLR,perLLRNet,EsNoValues,subsystemType)
end
```



The results show that the LLRNet almost matches the performance of exact LLR without using any expensive operations such as logarithm and exponential.

Further Exploration

Try different modulation and coding schemes for the DVB-S.2 system. The full list of modulation types and coding rates are given in the "DVB-S.2 Link, Including LDPC Coding in Simulink" on page 8-372 example. You can also try different sizes for the hidden layer of the network to reduce the number of operations and measure the performance loss as compared to exact LLR.

The example uses these helper functions. Examine these files to learn about details of the implementation.

- llrnetDVBS2PER.m: Simulate DVB-S.2 PER using exact LLR, approximate LLR, and LLRNet LLR
- llrnetTrainDVBS2LLRNetwork.m: Train neural networks for DVB-S.2 LLR estimation
- llrnetQAMLLR.m: Train neural networks for M-ary QAM LLR estimation and calculate exact LLR, approximate LLR, and LLRNet LLR
- llrnetNeuralNetwork.m: Configure a shallow neural network for LLR estimation

References

[1] O. Shental and J. Hoydis, ""Machine LLRning": Learning to Softly Demodulate," 2019 IEEE Globecom Workshops (GC Wkshps), Waikoloa, HI, USA, 2019, pp. 1-7.

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation

This example shows how to design a radio frequency (RF) fingerprinting convolutional neural network (CNN) with simulated data. You train the CNN with simulated wireless local area network (WLAN) beacon frames from known and unknown routers for RF fingerprinting. You then compare the media access control (MAC) address of received signals and the RF fingerprint detected by the CNN to detect WLAN router impersonators.

For more information on how to test the designed neural network with signals captured from real Wi-Fi routers, see the “Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation” on page 4-63 example.

Detect Router Impersonation Using RF Fingerprinting

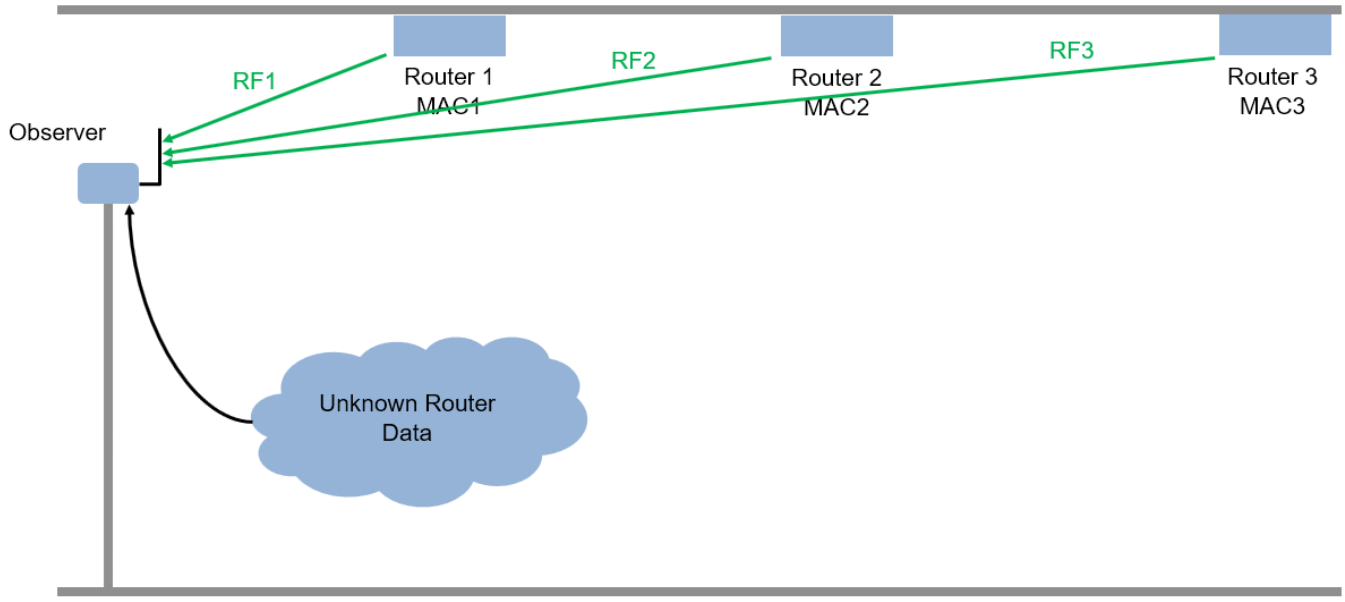
Router impersonation is a form of attack on a WLAN network where a malicious agent tries to impersonate a legitimate router and trick network users to connect to it. Security identification solutions based on simple digital identifiers, such as MAC addresses, IP addresses, and SSID, are not effective in detecting such an attack. These identifiers can be easily spoofed. Therefore, a more secure solution uses other information, such as the RF signature of the radio link, in addition to these simple digital identifiers.

A wireless transmitter-receiver pair creates a unique RF signature at the receiver that is a combination of the channel and RF impairments. *RF Fingerprinting* is the process of distinguishing transmitting radios in a shared spectrum through these signatures. In [1] on page 4-0 , authors designed a deep learning (DL) network that consumes raw baseband in-phase/quadrature (IQ) samples and identifies the transmitting radio. The network can identify the transmitting radios if the RF impairments are dominant or the channel profile stays constant during the operation time. Most WLAN networks have fixed routers that create a static channel profile when the receiver location is also fixed. In such a scenario, the deep learning network can identify router impersonators by comparing the received signal's RF fingerprint and MAC address pair to that of the known routers.

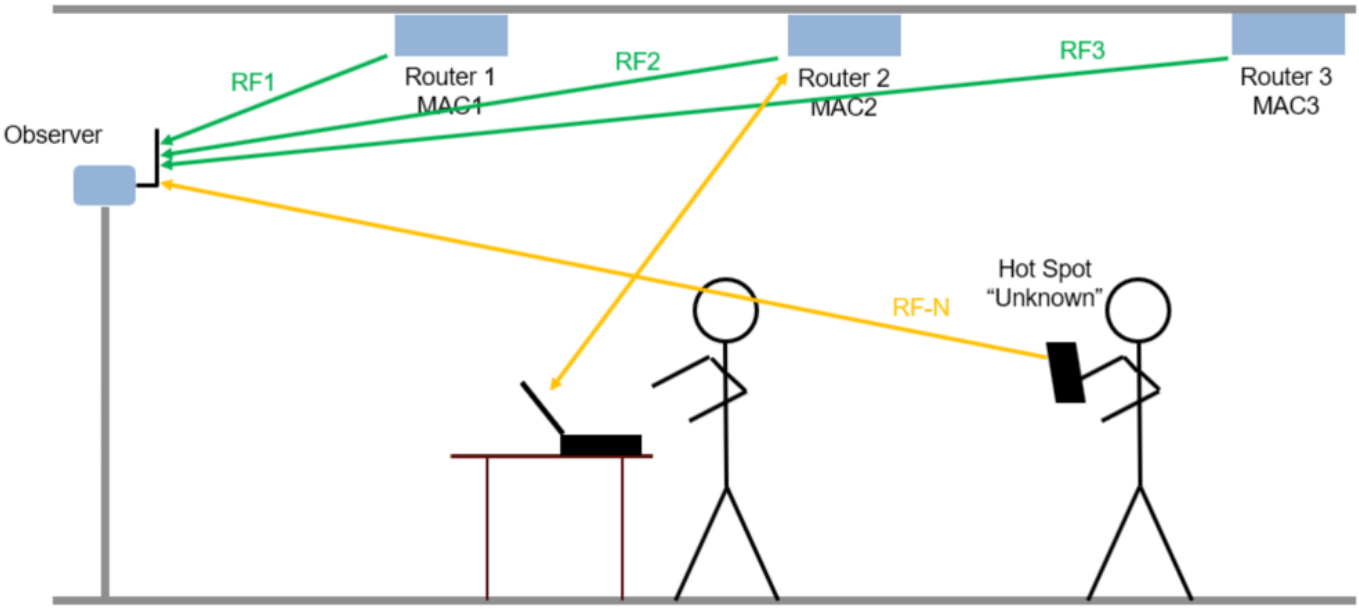
This example simulates a WLAN system with several fixed routers and a fixed observer using the WLAN Toolbox™ and trains a neural network (NN) with the simulated data using Deep Learning Toolbox™.

System Description

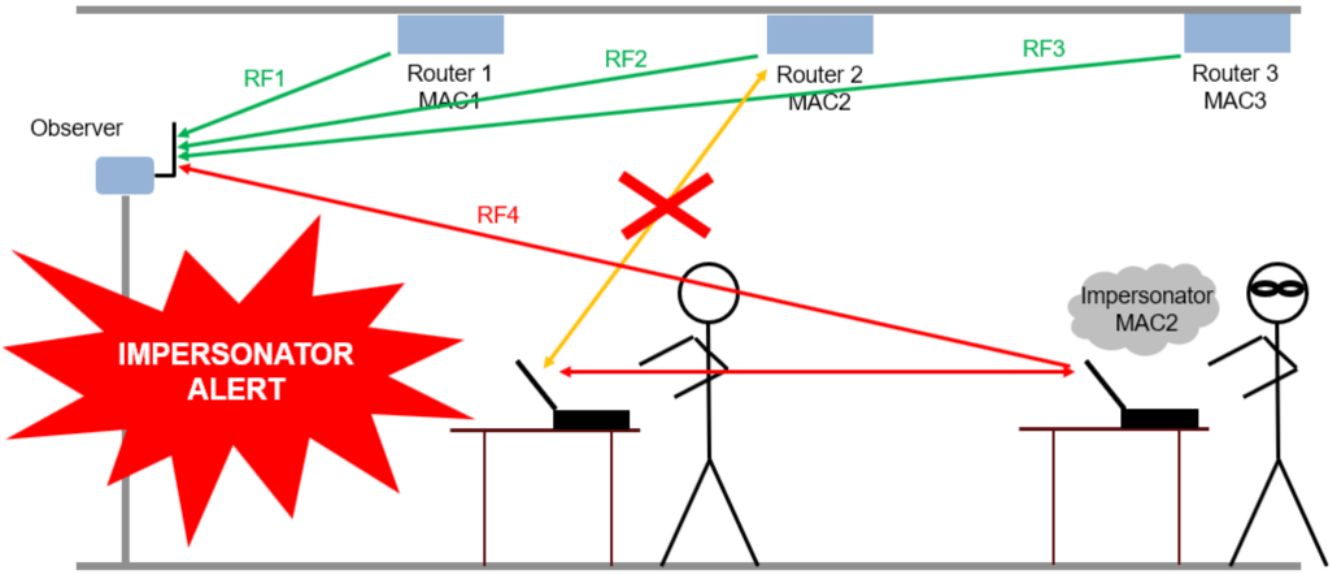
Assume an indoor space with a number of trusted routers with known MAC addresses, which we will refer to as known routers. Also, assume that unknown routers may enter the observation area, some of which may be router impersonators. The class "Unknown" represents any transmitting device that is not contained in the known set. The following figure shows a scenario where there are three known routers. The observer collects non-high throughput (non-HT) beacon signals from these routers and uses the (legacy) long training field (L-LTF) to identify the RF fingerprint. Transmitted L-LTF signals are the same for all routers that enables the algorithm to avoid any data dependency. Since the routers and the observer are fixed, the RF fingerprints (combination of multipath channel profile and RF impairments) RF1, RF2, and RF3 do not vary in time. Unknown router data is a collection of random RF fingerprints, which are different than the known routers.



The following figure shows a user connected to a router and a mobile hot spot. After training, the observer receives beacon frames and decodes the MAC address. Also, the observer extracts the L-LTF signal and uses this signal to classify the RF fingerprint of the source of the beacon frame. If the MAC address and the RF fingerprint matches, as in the case of Router 1, Router 2, and Router3, then the observer declares the source as a "known" router. If the MAC address of the beacon is not in the database and the RF fingerprint does not match any of the known routers, as in the case of a mobile hot spot, then the observer declares the source as an "unknown" router.



The following figure shows a router impersonator in action. A router impersonator (a.k.a. evil twin) can replicate the MAC address of a known router and transmit beacon frames. Then, the hacker can jam the original router and force the user to connect to the evil twin. The observer receives the beacon frames from the evil twin too and decodes the MAC address. The decoded MAC address matches the MAC address of a known router but the RF fingerprint does not match. The observer declares the source as a router impersonator.



Set System Parameters

Generate a dataset of 5,000 Non-HT WLAN beacon frames for each router. Use MAC addresses as labels for the known routers; the remaining are be labeled as "Unknown". A NN is trained to classify the known routers as well as to detect any unknown ones. Split the dataset into training, validation and test, where the splitting ratios are 80%, 10%, and 10%, respectively. Consider an SNR of 20 dB, working on the 5 GHz band. The number of simulated devices is set to 4 but it can be modified by choosing a different value for `numKnownRouters`. Set the number of unknown routers more than the known ones to represent in the dataset the variability in the unknown router RF fingerprints.

```
numKnownRouters = 4;
numUnknownRouters = 10;
numTotalRouters = numKnownRouters+numUnknownRouters;
SNR = 20;                   % dB
channelNumber = 153;        % WLAN channel number
channelBand = 5;            % GHz
frameLength = 160;          % L-LTF sequence length in samples
```

By default, this example downloads training data and trained network from https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingSimulatedData.tar.gz. If you do not have an Internet connection, you can download the file manually on a computer that is connected to the Internet and save to the same directory as the current example files.

To run this example quickly, download the pretrained trained network and generate a small number of frames, for example 10. To train the network on your computer, choose the "Train network now" option (i.e. set `trainNow` to true). Generating 5000 frames of data takes about 50 minutes on an Intel(R) Xeon(R) W-2133 CPU @ 3.6 GHz with 64 MB memory. Training this network takes about 5

minutes with an Nvidia(R) Titan Xp GPU. Training on a CPU may result in a very long training duration.

```
trainNow = Use trained network ;

if trainNow
  numTotalFramesPerRouter = 5000; %#ok<UNRCH>
else
  numTotalFramesPerRouter = 10;
  rfFingerprintingDownloadData('simulated')
end
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingSimulatedData.ta
Download and extracting files done
```

```
  numTrainingFramesPerRouter = numTotalFramesPerRouter*0.8;
  numValidationFramesPerRouter = numTotalFramesPerRouter*0.1;
  numTestFramesPerRouter = numTotalFramesPerRouter*0.1;
```

Generate WLAN Waveforms

Wi-Fi routers that implement 802.11a/g/n/ac protocols transmit beacon frames in the 5 GHz band to broadcast their presence and capabilities using the OFDM non-HT format. The beacon frame consists of two main parts: preamble (SYNC) and payload (DATA). The preamble has two parts: short training and long training. In this example, the payload contains the same bits except the MAC address for each router. The CNN uses the L-LTF part of the preamble as training units. Reusing the L-LTF signal for RF fingerprinting provides an overhead-free fingerprinting solution. Use `wlanMACFrameConfig` (WLAN Toolbox), `wlanMACFrame` (WLAN Toolbox), `wlanNonHTConfig` (WLAN Toolbox), and `wlanWaveformGenerator` (WLAN Toolbox) functions to generate WLAN beacon frames.

```
% Create Beacon frame-body configuration object
frameBodyConfig = wlanMACManagementConfig;

% Create Beacon frame configuration object
beaconFrameConfig = wlanMACFrameConfig('FrameType', 'Beacon', ...
  "ManagementConfig", frameBodyConfig);

% Generate Beacon frame bits
[~, mpduLength] = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

% Create a wlanNONHTConfig object, 20 MHz bandwidth and MCS 1 are used
nonHTConfig = wlanNonHTConfig(...
  'ChannelBandwidth', "CBW20",...
  "MCS", 1,...
  "PSDULength", mpduLength);
```

The rfFingerprintingNonHTFrontEnd object performs front-end processing including extracting the L-LTF signal. The object is configured with a channel bandwidth of 20 MHz to process non-HT signals.

```
rxFrontEnd = rfFingerprintingNonHTFrontEnd('ChannelBandwidth', 'CBW20');

fc = helperWLANChannelFrequency(channelNumber, channelBand);
fs = wlanSampleRate(nonHTConfig);
```

Setup Channel and RF Impairments

Pass each frame through a channel with

- Rayleigh multipath fading
- Radio impairments, such as phase noise, frequency offset and DC offset
- AWGN

Rayleigh Multipath and AWGN

The channel passes the signals through a Rayleigh multipath fading channel using the `comm.RayleighChannel` System object. Assume a delay profile of [0 1.8 3.4] samples with corresponding average path gains of [0 -2 -10] dB. Since the channel is static, set maximum Doppler shift to zero to make sure that the channel does not change for the same radio. Implement the multipath channel with these settings. Add noise using the `awgn` function,

```
multipathChannel = comm.RayleighChannel(...
  'SampleRate', fs, ...
  'PathDelays', [0 1.8 3.4]/fs, ...
  'AveragePathGains', [0 -2 -10], ...
  'MaximumDopplerShift', 0);
```

Radio Impairments

The RF impairments, and their corresponding range of values are:

- Phase noise [0.01, 0.3] rms (degrees)
- Frequency offset [-4, 4] ppm
- DC offset: [-50, -32] dBc

See helperRFImpairments on page 4-0 function for more details on RF impairment simulation. This function uses `comm.PhaseFrequencyOffset` and `comm.PhaseNoise` System objects.

```
phaseNoiseRange = [0.01, 0.3];
freqOffsetRange = [-4, 4];
dcOffsetRange = [-50, -32];

rng(123456)  % Fix random generator

% Assign random impairments to each simulated radio within the previously
% defined ranges
radioImpairments = repmat(...
  struct('PhaseNoise', 0, 'DCOffset', 0, 'FrequencyOffset', 0), ...
  numTotalRouters, 1);
for routerIdx = 1:numTotalRouters
  radioImpairments(routerIdx).PhaseNoise = ...
    rand*(phaseNoiseRange(2)-phaseNoiseRange(1)) + phaseNoiseRange(1);
  radioImpairments(routerIdx).DCOffset = ...
    rand*(dcOffsetRange(2)-dcOffsetRange(1)) + dcOffsetRange(1);
  radioImpairments(routerIdx).FrequencyOffset = ...
    fc/1e6*(rand*(freqOffsetRange(2)-freqOffsetRange(1)) + freqOffsetRange(1));
end
```

Apply Channel Impairments and Generate Data Frames for Training

Apply the RF and channel impairments defined previously. Reset the channel object for each radio to generate an independent channel. Use rfFingerprintingNonHTFrontEnd function to process the

received frames. Finally, extract the L-LTF from every transmitted WLAN frame. Split the received L-LTF signals into training, validation and test sets.

```
% Create variables that will store the training, validation and testing
% datasets
xTrainingFrames = zeros(frameLength, numTrainingFramesPerRouter*numTotalRouters);
xValFrames = zeros(frameLength, numValidationFramesPerRouter*numTotalRouters);
xTestFrames = zeros(frameLength, numTestFramesPerRouter*numTotalRouters);

% Index vectors for train, validation and test data units
trainingIndices = 1:numTrainingFramesPerRouter;
validationIndices = 1:numValidationFramesPerRouter;
testIndices = 1:numTestFramesPerRouter;

tic
generatedMACAddresses = strings(numTotalRouters, 1);
rxLLTF = zeros(frameLength, numTotalFramesPerRouter);     % Received L-LTF sequences
for routerIdx = 1:numTotalRouters

  % Generate a 12-digit random hexadecimal number as a MAC address for
  % known routers. Set the MAC address of all unknown routers to
  % 'AAAAAAAAAAAA'.
  if (routerIdx<=numKnownRouters)
    generatedMACAddresses(routerIdx) = string(dec2hex(bi2de(randi([0 1], 12, 4)))');
  else
    generatedMACAddresses(routerIdx) = 'AAAAAAAAAAAA';
  end

  fprintf('%s - Generating frames for router %d with MAC address %s\n', ...
    datestr(toc/86400,'HH:MM:SS'), routerIdx, generatedMACAddresses(routerIdx))

  % Set MAC address into the wlanFrameConfig object
  beaconFrameConfig.Address2 = generatedMACAddresses(routerIdx);

  % Generate beacon frame bits
  beacon = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

  txWaveform = wlanWaveformGenerator(beacon, nonHTConfig);

  txWaveform = helperNormalizeFramePower(txWaveform);

  % Add zeros to account for channel delays
  txWaveform = [txWaveform; zeros(160,1)]; %#ok<AGROW>

  % Reset multipathChannel object to generate a new static channel
  reset(multipathChannel)

  frameCount= 0;
  while frameCount<numTotalFramesPerRouter

    rxMultipath = multipathChannel(txWaveform);

    rxImpairment = helperRFImpairments(rxMultipath, radioImpairments(routerIdx), fs);

    rxSig = awgn(rxImpairment,SNR,0);

    % Detect the WLAN packet and return the received L-LTF signal using
    % rfFingerprintingNonHTFrontEnd object
```

```
    [valid, ~, ~, ~, ~, LLTF] = rxFrontEnd(rxSig);

    % Save successfully received L-LTF signals
    if valid
      frameCount=frameCount+1;
      rxLLTF(:,frameCount) = LLTF;
    end

    if mod(frameCount,500) == 0
      fprintf('%s - Generated %d/%d frames\n', ...
        datestr(toc/86400,'HH:MM:SS'), frameCount, numTotalFramesPerRouter)
    end
  end

  rxLLTF = rxLLTF(:, randperm(numTotalFramesPerRouter));

  % Split data into training, validation and test
  xTrainingFrames(:, trainingIndices+(routerIdx-1)*numTrainingFramesPerRouter) ...
    = rxLLTF(:, trainingIndices);
  xValFrames(:, validationIndices+(routerIdx-1)*numValidationFramesPerRouter)...
    = rxLLTF(:, validationIndices+ numTrainingFramesPerRouter);
  xTestFrames(:, testIndices+(routerIdx-1)*numTestFramesPerRouter)...
    = rxLLTF(:, testIndices + numTrainingFramesPerRouter+numValidationFramesPerRouter);
end
```

```
00:00:00 - Generating frames for router 1 with MAC address 71153FFD7ACA
00:00:01 - Generating frames for router 2 with MAC address 5F4A8EAD6AD2
00:00:01 - Generating frames for router 3 with MAC address A91A85793DAA
00:00:01 - Generating frames for router 4 with MAC address 841F1BE784B0
00:00:02 - Generating frames for router 5 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 6 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 7 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 8 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 9 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 10 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 11 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 12 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 13 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 14 with MAC address AAAAAAAAAAAA
```

```
% Label received frames. Label the first numKnownRouters with their MAC
% address. Label the rest with "Unknown”.
labels = generatedMACAddresses;
labels(generatedMACAddresses == generatedMACAddresses(numTotalRouters)) = "Unknown";

yTrain = repelem(labels, numTrainingFramesPerRouter);
yVal = repelem(labels, numValidationFramesPerRouter);
yTest = repelem(labels, numTestFramesPerRouter);
```

Create Real-Valued Input Matrices

The Deep Learning model only works on real numbers. Thus, I and Q are split into two separate columns. Then, the data is rearranged into a 2 x frameLength x 1 x numFrames matrix, as required by the Deep Learning Toolbox. Additionally, the training set is shuffled, and the label variables are saved as categorical variables.

```
% Rearrange datasets into a one-column vector
xTrainingFrames = xTrainingFrames(:);
xValFrames = xValFrames(:);
xTestFrames = xTestFrames(:);

% Separate between I and Q
xTrainingFrames = [real(xTrainingFrames), imag(xTrainingFrames)];
xValFrames = [real(xValFrames), imag(xValFrames)];
xTestFrames = [real(xTestFrames), imag(xTestFrames)];

% Reshape training data into a 2 x frameLength x 1 x
% numTrainingFramesPerRouter*numTotalRouters matrix
xTrainingFrames = permute(...
  reshape(xTrainingFrames,[frameLength,numTrainingFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]);

% Shuffle data
vr = randperm(numTotalRouters*numTrainingFramesPerRouter);
xTrainingFrames = xTrainingFrames(:,:,:,vr);

% Create label vector and shuffle
yTrain = categorical(yTrain(vr));

% Reshape validation data into a 2 x frameLength x 1 x
% numValidationFramesPerRouter*numTotalRouters matrix
xValFrames = permute(...
  reshape(xValFrames,[frameLength,numValidationFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]);

% Create label vector
yVal = categorical(yVal);

% Reshape test dataset into a numTestFramesPerRouter*numTotalRouter matrix
xTestFrames = permute(...
  reshape(xTestFrames,[frameLength,numTestFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]); %#ok<NASGU>

% Create label vector
yTest = categorical(yTest); %#ok<NASGU>
```

Train the Neural Network

This example uses a neural network (NN) architecture that consists of two convolutional and three fully connected layers. The intuition behind this design is that the first layer will learn features independently in I and Q. Note that the filter sizes are 1x7. Then, the next layer will use a filter size of 2x7 that will extract features combining I and Q together. Finally, the last three fully connected layers will behave as a classifier using the extracted features in the previous layers [1] on page 4-0 .

```
poolSize = [2 1];
strideSize = [2 1];
layers = [
  imageInputLayer([frameLength 2 1], 'Normalization', 'none', 'Name', 'Input Layer')

  convolution2dLayer([7 1], 50, 'Padding', [1 0], 'Name', 'CNN1')
  batchNormalizationLayer('Name', 'BN1')
  leakyReluLayer('Name', 'LeakyReLu1')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool1')
```

```
  convolution2dLayer([7 2], 50, 'Padding', [1 0], 'Name', 'CNN2')
  batchNormalizationLayer('Name', 'BN2')
  leakyReluLayer('Name', 'LeakyReLu2')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool2')

  fullyConnectedLayer(256, 'Name', 'FC1')
  leakyReluLayer('Name', 'LeakyReLu3')
  dropoutLayer(0.5, 'Name', 'DropOut1')

  fullyConnectedLayer(80, 'Name', 'FC2')
  leakyReluLayer('Name', 'LeakyReLu4')
  dropoutLayer(0.5, 'Name', 'DropOut2')

  fullyConnectedLayer(numKnownRouters+1, 'Name', 'FC3')
  softmaxLayer('Name', 'SoftMax')
  classificationLayer('Name', 'Output')
  ]
```

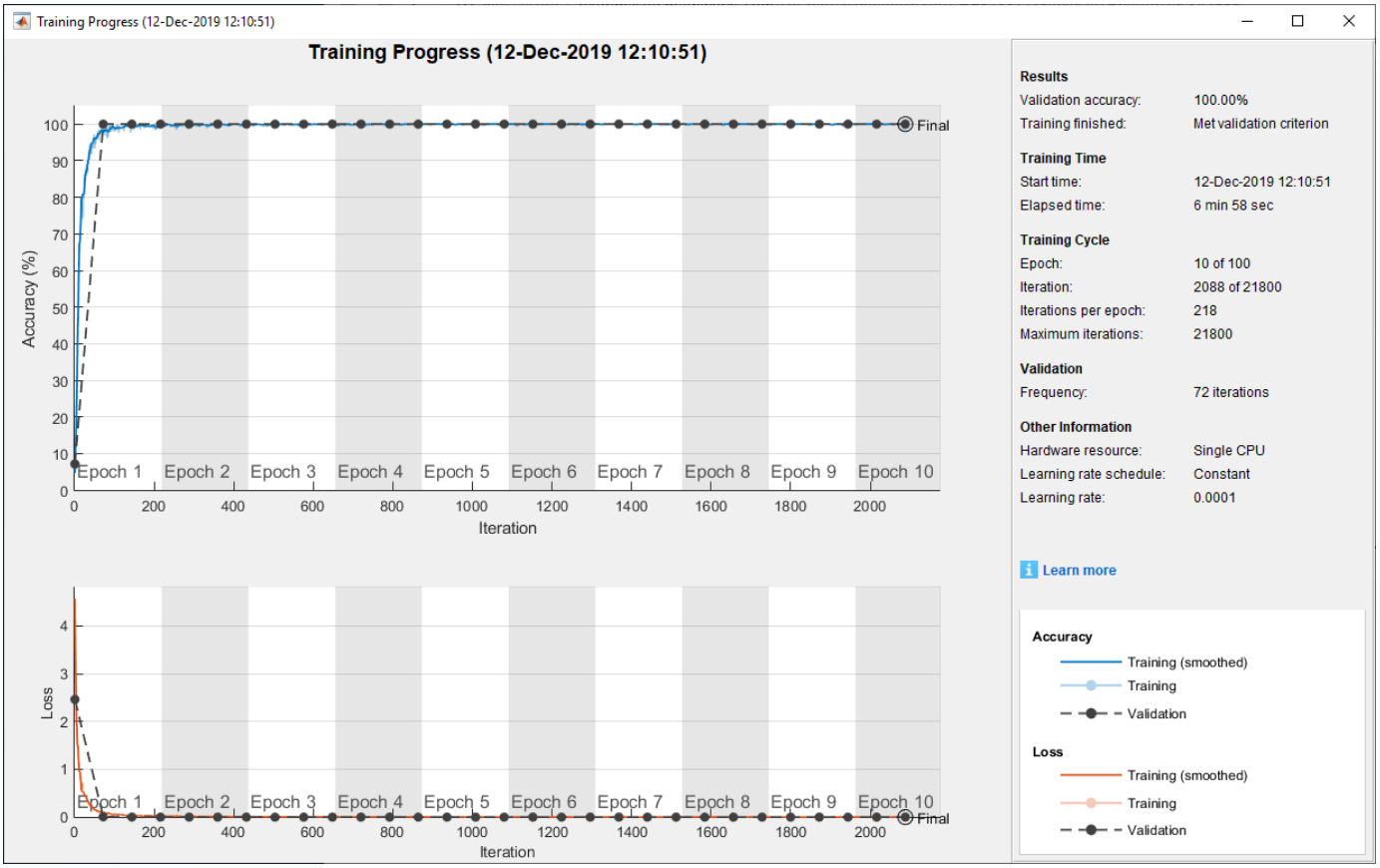
```
layers =
  18×1 Layer array with layers:

     1   'Input Layer'   Image Input             160×2×1 images
     2   'CNN1'          Convolution             50 7×1 convolutions with stride [1  1] and paddi
     3   'BN1'           Batch Normalization     Batch normalization
     4   'LeakyReLu1'    Leaky ReLU              Leaky ReLU with scale 0.01
     5   'MaxPool1'      Max Pooling             2×1 max pooling with stride [2  1] and padding
     6   'CNN2'          Convolution             50 7×2 convolutions with stride [1  1] and paddi
     7   'BN2'           Batch Normalization     Batch normalization
     8   'LeakyReLu2'    Leaky ReLU              Leaky ReLU with scale 0.01
     9   'MaxPool2'      Max Pooling             2×1 max pooling with stride [2  1] and padding
    10   'FC1'           Fully Connected         256 fully connected layer
    11   'LeakyReLu3'    Leaky ReLU              Leaky ReLU with scale 0.01
    12   'DropOut1'      Dropout                 50% dropout
    13   'FC2'           Fully Connected         80 fully connected layer
    14   'LeakyReLu4'    Leaky ReLU              Leaky ReLU with scale 0.01
    15   'DropOut2'      Dropout                 50% dropout
    16   'FC3'           Fully Connected         5 fully connected layer
    17   'SoftMax'       Softmax                 softmax
    18   'Output'        Classification Output   crossentropyex
```

Configure the training options to use the ADAM optimizer with a mini-batch size of 256. By default, '`ExecutionEnvironment`' is set to 'auto', which uses a GPU for training if one is available. Otherwise, `trainNetwork` (Deep Learning Toolbox) uses a CPU for training. To explicitly set the execution environment, set '`ExecutionEnvironment`' to one of '`cpu`', '`gpu`', '`multi-gpu`', or '`parallel`'. Choosing '`cpu`' may result in a very long training duration.

```
if trainNow

  miniBatchSize = 256; %#ok<UNRCH>

  % Training options
  options = trainingOptions('adam', ...
    'MaxEpochs',100, ...
    'ValidationData',{xValFrames, yVal}, ...
    'ValidationFrequency',floor(numTrainingFramesPerRouter*numTotalRouters/miniBatchSize/3), ...
    'Verbose',false, ...
    'L2Regularization', 0.0001, ...
    'InitialLearnRate', 0.0001, ...
```

```
    'MiniBatchSize', miniBatchSize, ...
    'ValidationPatience', 3, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch');

  % Train the network
  simNet = trainNetwork(xTrainingFrames, yTrain, layers, options);
else
  % Load trained network (simNet), testing dataset (xTestFrames and
  % yTest) and the used MACAddresses (generatedMACAddresses)
  load('rfFingerprintingSimulatedDataTrainedNN.mat',...
    'generatedMACAddresses',...
    'simNet',...
    'xTestFrames',...
    'yTest')
end
```

As the plot of the training progress shows, the network converges in about 2 epochs to almost 100% accuracy.



Classify test frames and calculate the final accuracy off the neural network.

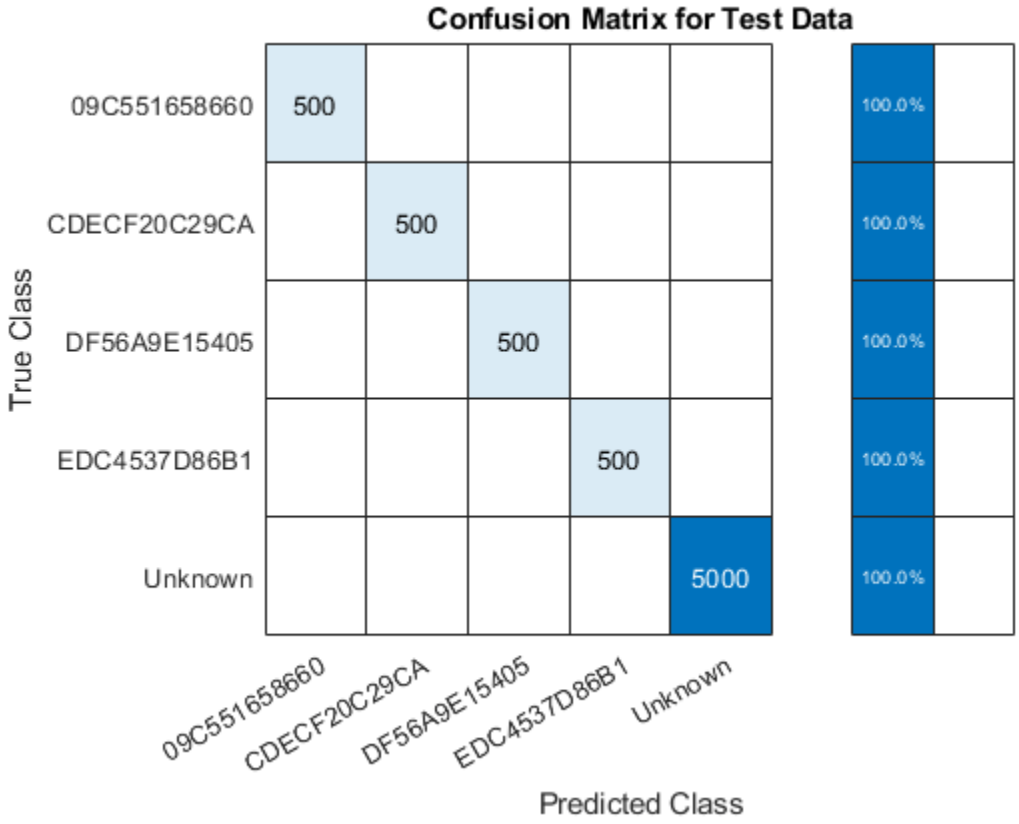
```
% Obtain predicted classes for xTestFrames
yTestPred = classify(simNet,xTestFrames);

% Calculate test accuracy
```

```
testAccuracy = mean(yTest == yTestPred);
disp("Test accuracy: " + testAccuracy*100 + "%")
```

```
Test accuracy: 100%
```

Plot the confusion matrix for the test frames. As mentioned before, perfect classification accuracy is achieved with the synthetic dataset.

```
figure
cm = confusionchart(yTest, yTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
```



Detect Router Impersonator

Generate beacon frames with the known MAC addresses and one unknown MAC address. Generate a new set of RF impairments and multipath channel. Since the impairments are all new, the RF fingerprint for these frames should be classified as "Unknown". The frames with known MAC addresses represent router impersonators while the frames with unknown MAC addresses are simply unknown routers.

```
framesPerRouter = 4;
knownMACAddresses = generatedMACAddresses(1:numKnownRouters);

% Assign random impairments to each simulated radio within the previously
% defined ranges
for routerIdx = 1:numTotalRouters
  radioImpairments(routerIdx).PhaseNoise = rand*( phaseNoiseRange(2)-phaseNoiseRange(1) ) + phase
```

```
    radioImpairments(routerIdx).DCOffset = rand*( dcOffsetRange(2)-dcOffsetRange(1) ) + dcOffsetRa
    radioImpairments(routerIdx).FrequencyOffset = fc/1e6*(rand*( freqOffsetRange(2)-freqOffsetRange
  end
  % Reset multipathChannel object to generate a new static channel
  reset(multipathChannel)

  % Run for all known routers and one unknown
  for macIndex = 1:(numKnownRouters+1)

    beaconFrameConfig.Address2 = generatedMACAddresses(macIndex);

    % Generate Beacon frame bits
    beacon = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

    txWaveform = wlanWaveformGenerator(beacon, nonHTConfig);

    txWaveform = helperNormalizeFramePower(txWaveform);

    % Add zeros to account for channel delays
    txWaveform = [txWaveform; zeros(160,1)]; %#ok<AGROW>

    % Create an unseen multipath channel. In other words, create an unseen
    % RF fingerprint.
    reset(multipathChannel)

    frameCount= 0;
    while frameCount<framesPerRouter

      rxMultipath = multipathChannel(txWaveform);

      rxImpairment = helperRFImpairments(rxMultipath, radioImpairments(routerIdx), fs);

      rxSig = awgn(rxImpairment,SNR,0);

      % Detect the WLAN packet and return the received L-LTF signal using
      % rfFingerprintingNonHTFrontEnd object
      [payloadFull, cfgNonHT, rxNonHTData, chanEst, noiseVar, LLTF] = ...
        rxFrontEnd(rxSig);

      if payloadFull
        frameCount = frameCount+1;
        recBits = wlanNonHTDataRecover(rxNonHTData, chanEst, ...
          noiseVar, cfgNonHT, 'EqualizationMethod', 'ZF');

        % Decode and evaluate recovered bits
        mpduCfg = wlanMPDUDecode(recBits, cfgNonHT);

        % Separate I and Q and reshape for neural network
        LLTF= [real(LLTF), imag(LLTF)];
        LLTF = permute(reshape(LLTF,frameLength ,[] , 2, 1), [1 3 4 2]);

        ypred = classify(simNet, LLTF);

        if sum(contains(knownMACAddresses, mpduCfg.Address2)) ~= 0
          if categorical(convertCharsToStrings(mpduCfg.Address2))~=ypred
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint mismatch, ROUTER
          else
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint match"))
```

```
        end
      else
        disp(strcat("MAC Address ", mpduCfg.Address2," is not recognized, unknown device"))
      end
    end

    % Reset multipathChannel object to generate a new static channel
    reset(multipathChannel)
  end
end
```

```
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
```

Further Exploration

You can test the system under different channel and RF impairments by modifying the

- Multipath profile (`PathDelays` and `AveragePathGains` properties of Rayleigh channel object),
- Channel noise level (SNR input of `awgn` function),
- RF impairments (`phaseNoiseRange`, `freqOffsetRange`, and `dcOffsetRange` variables).

You can also modify the neural network structure by changing

- Convolutional layer parameters (filter size, number of filters, padding),
- Number of fully connected layers,
- Number of convolutional layers.

Appendix: Helper Functions

```
function [impairedSig] = helperRFImpairments(sig, radioImpairments, fs)
% helperRFImpairments Apply RF impairments
%   IMPAIREDSIG = helperRFImpairments(SIG, RADIOIMPAIRMENTS, FS) returns signal
%   SIG after applying the impairments defined by RADIOIMPAIRMENTS
%   structure at the sample rate FS.

% Apply frequency offset
fOff = comm.PhaseFrequencyOffset('FrequencyOffset', radioImpairments.FrequencyOffset,  'SampleRa
```

```
% Apply phase noise
phaseNoise = helperGetPhaseNoise(radioImpairments);
phNoise = comm.PhaseNoise('Level', phaseNoise, 'FrequencyOffset', abs(radioImpairments.FrequencyO

impFOff = fOff(sig);
impPhNoise = phNoise(impFOff);

% Apply DC offset
impairedSig = impPhNoise + 10^(radioImpairments.DCOffset/10);

end

function [phaseNoise] = helperGetPhaseNoise(radioImpairments)
% helperGetPhaseNoise Get phase noise value
load('Mrms.mat','Mrms','MyI','xI');
[~, iRms] = min(abs(radioImpairments.PhaseNoise - Mrms));
[~, iFreqOffset] = min(abs(xI - abs(radioImpairments.FrequencyOffset)));
phaseNoise = -abs(MyI(iRms, iFreqOffset));
end
```

Selected Bibliography

[1] K. Sankhe, M. Belgiovine, F. Zhou, S. Riyaz, S. Ioannidis and K. Chowdhury, "ORACLE: Optimized Radio clAssification through Convolutional neuraL nEtworks," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 370-378.

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation

This example shows how to train a radio frequency (RF) fingerprinting convolutional neural network (CNN) with captured data. You capture wireless local area network (WLAN) beacon frames from real routers using a software defined radio (SDR). You program a second SDR to transmit unknown beacon frames and capture them. You train the CNN using these captured signals. You then program a software-defined radio (SDR) as a router impersonator that transmits beacon signals with the media access control (MAC) address of one of the known routers and use the CNN to identify it as an impersonator.

For more information on router impersonation and validation of the network design with simulated data, see the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” on page 4-49 example.

Train with Captured Data

Collect a dataset of 802.11a/g/n/ac OFDM non-high throughput (non-HT) beacon frames from real WLAN routers. As described in the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” on page 4-49 example, only the legacy long training field (L-LTF) field present in preambles are used as training units in order to avoid any data dependency.

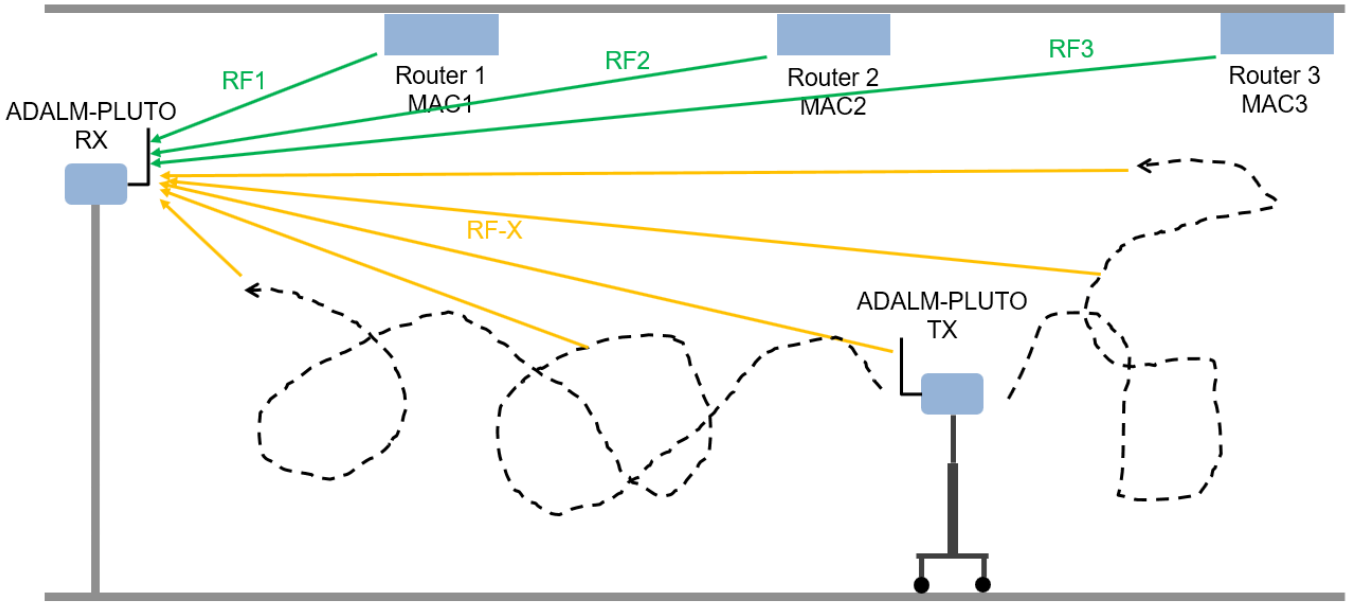
In this example, the data was collected using the scenario depicted in the following figure. The observer is a stationary ADALM-PLUTO radio. Known router data was collected as follows:

1. Set the observer's center frequency based on the WLAN channel used by the routers
2. Receive a beacon frame
3. Extract the L-LTF signal
4. Decode the MAC address to use as the label
5. Save the L-LTF signal together with its label
6. Repeat steps 2-5 to collect `numFramesPerRouter` frames from `numKnownRouters` routers.

Unknown router beacon frames are simulated using a mobile ADALM-PLUTO radio as a transmitter. This radio repeatedly transmits beacon frames with a random MAC address. Unknown router data was collected as follows:

1. Generate beacon frames with a random MAC address
2. Start transmitting the beacon frames repeatedly using the ADALM-PLUTO radio
3. Collect `NUMFRAMES` beacon frames
4. Extract the L-LTF signal
5. Save the L-LTF frames with label "Unknown"
6. Move the radio to another location
7. Repeat steps 3-6 to collect data from `NUMLOC` locations

This combined dataset of known and unknown routers is used to train the same DL model as in the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” on page 4-49 example.



This example downloads training data and trained network from https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData.tar.gz. If you do not have an Internet connection, you can download the file manually on a computer that is connected to the Internet and save to the same directory as the current example files. For privacy reasons, MAC addresses have been anonymized in the downloaded data. To replicate the results of this example, capture your own data as described in Appendix: Known and Unknown Router Data Collection on page 4-0 .

```
rfFingerprintingDownloadData('captured')
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData.tar
Download and extracting files done
```

To run this example quickly, use the downloaded pretrained network. To train the network on your computer, choose the "Train network now" option (i.e. set `trainNow` to true). Training this network takes about 5 minutes with an Nvidia(R) Titan Xp GPU. Training on a CPU may result in a very long training duration.

`trainNow =` Use trained network ; `%#ok<*UNRCH>`

This example uses data from four known routers. The dataset contains 3600 frames per router, where 90% is used as training frames and 10% is used as test frames.

```
numKnownRouters = 4;
numFramesPerRouter = 3600;
numTrainingFramesPerRouter = numFramesPerRouter * 0.9;
numTestFramesPerRouter = numFramesPerRouter * 0.1;
frameLength = 160;
```

Preprocess Known and Unknown Router Data

Separate collected complex baseband data into its in-phase and quadrature components and reshape it into a *2 x frameLength x 1 x numFramesPerRouter*numKnownRouters* matrix. Repeat the same process for the unknown router data. The following code uses previously collected and pre-processed data. To use your own data, first collect data as described in Appendix: Known and Unknown Router Data Collection on page 4-0 . Copy the new data files named `rfFingerprintingCapturedDataUser.mat` and `rfFingerprintingCapturedUnknownFramesUser.mat` to the same directory as this example. Then update the `load` commands to load these files.

```
if trainNow
  % Load known router data
  load('rfFingerprintingCapturedData.mat')

  % Create label vectors
  yTrain = repelem(MACAddresses, numTrainingFramesPerRouter);
  yTest = repelem(MACAddresses, numTestFramesPerRouter);

  % Separate between I and Q
  numTrainingSamples = numTrainingFramesPerRouter*numKnownRouters*frameLength;
  xTrainingFrames = xTrainingFrames(1:numTrainingSamples,1);
  xTrainingFrames = [real(xTrainingFrames), imag(xTrainingFrames)];
  numTestSamples = numTestFramesPerRouter*numKnownRouters*frameLength;
  xTestFrames = xTestFrames(1:numTestSamples,1);
  xTestFrames = [real(xTestFrames), imag(xTestFrames)];

  % Reshape dataset into an 2 x frameLength x 1 x numTrainingFramesPerRouter*numKnownRouters mat
  xTrainingFrames = permute(...
    reshape(xTrainingFrames,[frameLength,numTrainingFramesPerRouter*numKnownRouters, 2, 1]),...
    [1 3 4 2]);

  % Reshape dataset into an 2 x frameLength x 1 x numTestFramesPerRouter*numKnownRouters matrix
  xTestFrames = permute(...
    reshape(xTestFrames,[frameLength,numTestFramesPerRouter*numKnownRouters, 2, 1]),...
    [1 3 4 2]);

  % Load unknown router data
  load('rfFingerprintingCapturedUnknownFrames.mat')

  % Number of training units
  numUnknownFrames = size(unknownFrames, 4);

  % Split data into 90% training and 10% test
  numUnknownTrainingFrames = floor(numUnknownFrames*0.9);
  numUnknownTest = numUnknownFrames - numUnknownTrainingFrames;

  % Add ADALM-PLUTO data into training and test datasets
  xTrainingFrames(:,:,:,(1:numUnknownTrainingFrames) + numTrainingFramesPerRouter*numKnownRouters
    = unknownFrames(:,:,:, 1:numUnknownTrainingFrames);
  xTestFrames(:,:,:,(1:numUnknownTest) + numTestFramesPerRouter*numKnownRouters) ...
    = unknownFrames(:,:,:, (1:numUnknownTest) + numUnknownTrainingFrames);

  % Shuffle data
  vr = randperm(numKnownRouters*numTrainingFramesPerRouter+numUnknownTrainingFrames);
  xTrainingFrames = xTrainingFrames(:,:,:,vr);
```

```
  % Add "unknown" label and shuffle
  yTrain = [yTrain, repmat("Unknown", [1, numUnknownTrainingFrames])];
  yTrain = categorical(yTrain(vr));

  yTest = [yTest, repmat("Unknown", [1, numUnknownTest])];
  yTest = categorical(yTest);
end
```

Train the CNN

Use the same NN architecture and training options as in the training with simulated data example.

```
poolSize = [2 1];
strideSize = [2 1];
% Create network architecture
layers = [
  imageInputLayer([frameLength 2 1], 'Normalization', 'none', 'Name', 'Input Layer')

  convolution2dLayer([7 1], 50, 'Padding', [1 0], 'Name', 'CNN1')
  batchNormalizationLayer('Name', 'BN1')
  leakyReluLayer('Name', 'LeakyReLu1')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool1')

  convolution2dLayer([7 2], 50, 'Padding', [1 0], 'Name', 'CNN2')
  batchNormalizationLayer('Name', 'BN2')
  leakyReluLayer('Name', 'LeakyReLu2')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool2')

  fullyConnectedLayer(256, 'Name', 'FC1')
  leakyReluLayer('Name', 'LeakyReLu3')
  dropoutLayer(0.5, 'Name', 'DropOut1')

  fullyConnectedLayer(80, 'Name', 'FC2')
  leakyReluLayer('Name', 'LeakyReLu4')
  dropoutLayer(0.5, 'Name', 'DropOut2')

  fullyConnectedLayer(numKnownRouters+1, 'Name', 'FC3')
  softmaxLayer('Name', 'SoftMax')
  classificationLayer('Name', 'Output')
  ];
```

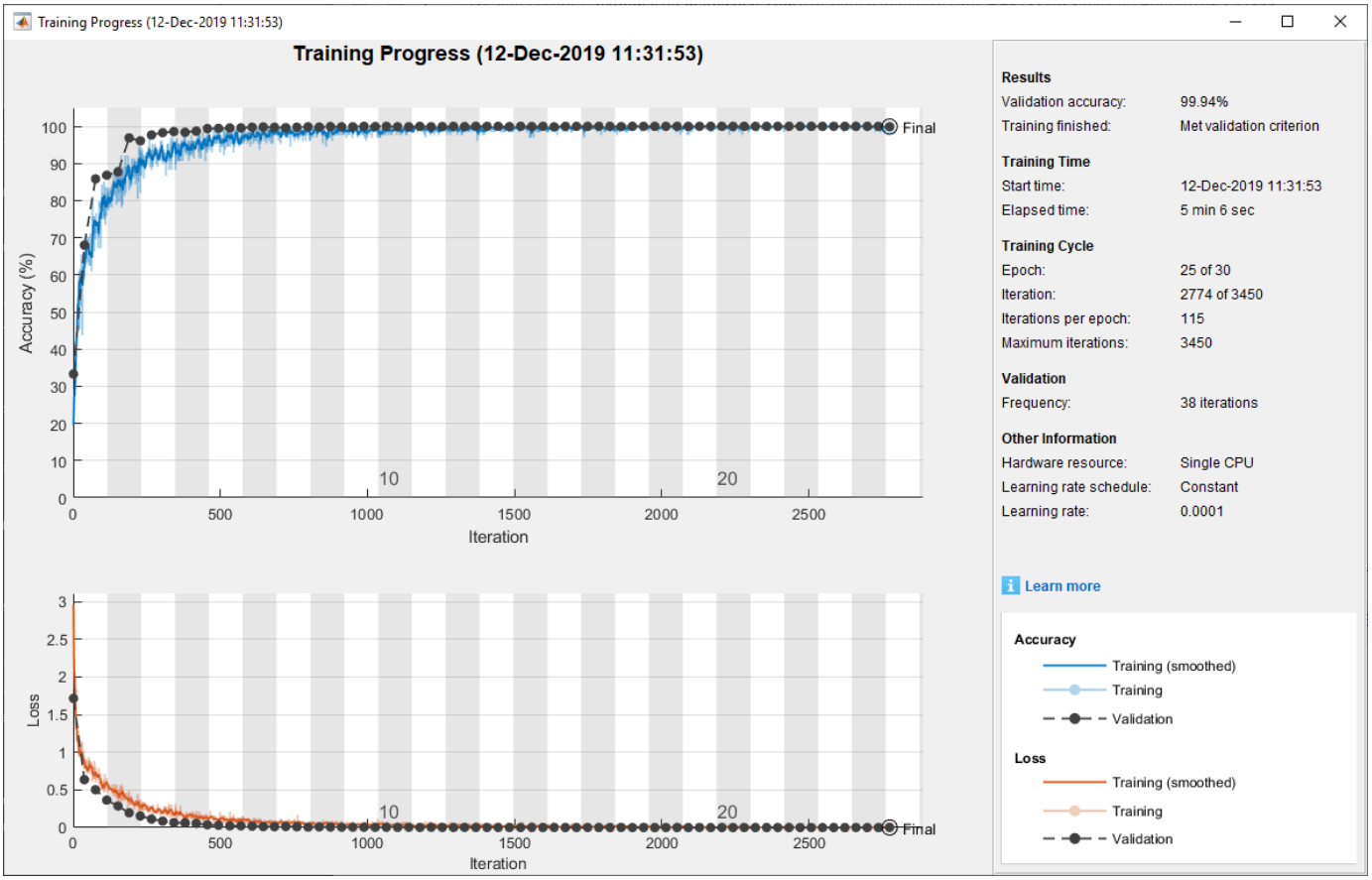
Configure the training options to use ADAM optimizer with a mini-batch size of 128. Use test frames for validation since optimization of hyperparameters were done in [1] on page 4-0 .

By default, `ExecutionEnvironment` is set to `'auto'`, which uses a GPU for training if one is available. Otherwise, `trainNetwork` (Deep Learning Toolbox) uses the CPU for training. To explicitly set the execution environment, set `ExecutionEnvironment` to one of `'cpu'`, `'gpu'`, `'multi-gpu'`, or `'parallel'`. Choosing `'cpu'` may result in a very long training duration.

```
if trainNow
  miniBatchSize = 128;

  % Training options
  options = trainingOptions('adam', ...
    'MaxEpochs',30, ...
    'ValidationData',{xTestFrames, yTest}, ...
    'ValidationFrequency',floor((numTrainingFramesPerRouter*numKnownRouters + numUnknownTrainingF
    'Verbose',false, ...
```

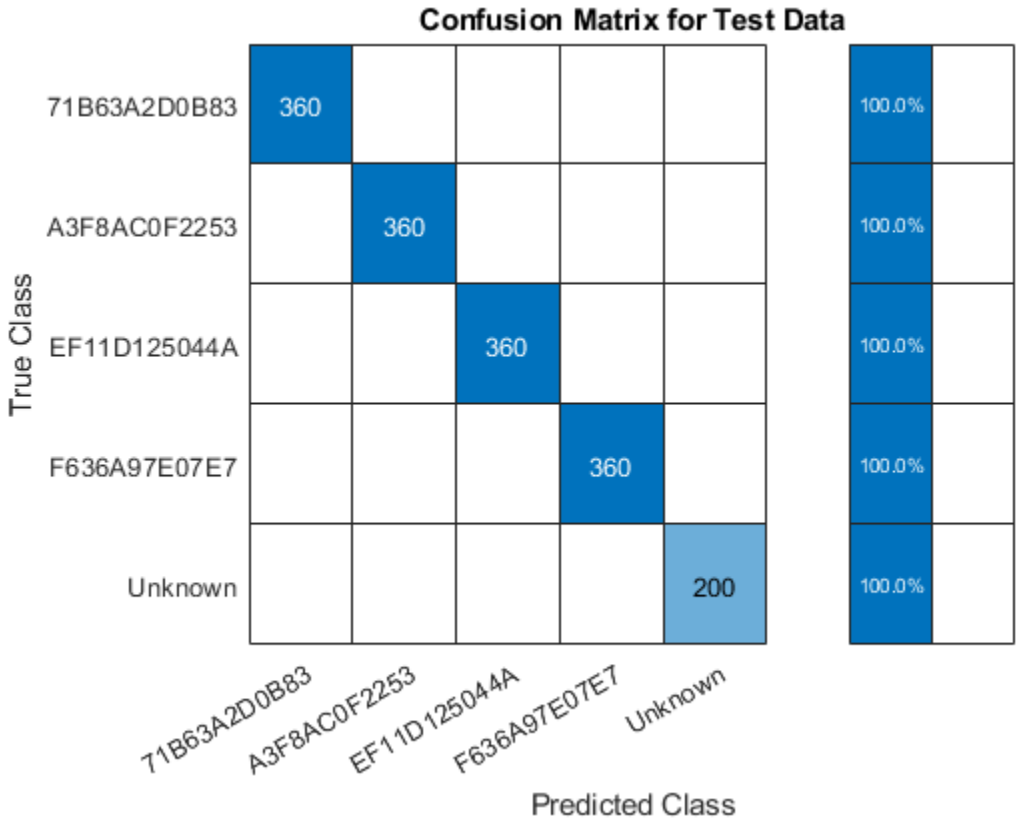
```
    'L2Regularization', 0.0001, ...
    'InitialLearnRate', 0.0001, ...
    'MiniBatchSize', miniBatchSize, ...
    'ValidationPatience', 5, ...
    'Plots','training-progress', ...
    'Shuffle', 'every-epoch');

  % Train the network
  capturedDataNet = trainNetwork(xTrainingFrames, yTrain, layers, options);
else
  load('rfFingerprintingCapturedDataTrainedNN.mat','capturedDataNet','xTestFrames','yTest','MACAd
end
```

The following plot shows the training progress of the network run on a computer with a single Nvidia Titan Xp GPU, where the network converged in about 10 epochs to almost 100% accuracy. The final accuracy of the network is 100%.



Generate the confusion matrix.

```
figure
yTestPred = classify(capturedDataNet,xTestFrames);
cm = confusionchart(yTest, yTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
```



Test with SDR

Test the performance of the trained network on the class "Unknown". Generate beacon frames with MAC addresses of the known routers and one unknown router. Transmit these frames using an ADALM-PLUTO radio and receive using another ADALM-PLUTO radio. Since the channel and RF impairments created between these two radios are different than the ones created between the real routers and the observer, the neural network should classify all of the received signals as "Unknown". If the received MAC address is a known one, then the system declares the source as a router impersonator. If the received MAC address is an unknown one, then the system declares the source as an unknown router. To perform this test, you need two ADALM-PLUTO radios for transmission and reception. Also, you need to install Communication Toolbox Support Package for ADALM-PLUTO Radio.

Waveform Generation

Generate a transmission waveform consisting of beacon frames with different MAC addresses. The transmitter repeatedly transmits these WLAN frames. The receiver captures the WLAN frames and determines if it is a router impersonator using the received MAC address and RF fingerprint detected by the trained NN.

```
chanBW='CBW20';     % Channel Bandwidth
osf = 2;            % Oversampling Factor
frameLength=160;    % Frame Length in samples
% Create Beacon frame-body configuration object
frameBodyConfig = wlanMACManagementConfig;
```

```
% Create Beacon frame configuration object
beaconFrameConfig = wlanMACFrameConfig('FrameType', 'Beacon');
beaconFrameConfig.ManagementConfig = frameBodyConfig;

% Create interpolation and decimation objects
decimator = dsp.FIRDecimator('DecimationFactor',osf);

% Save known MAC addresses
knownMACAddresses = MACAddresses;
MACAddressesToSimulate = [MACAddresses, "ABCDEFABCDEF"];

% Create WLAN waveform with the MAC addresses of known routers and an
% unknown router
txWaveform = zeros(1540,5);
for i = 1:length(MACAddressesToSimulate)

  % Set MAC Address
  beaconFrameConfig.Address2 = MACAddressesToSimulate(i);

  % Generate Beacon frame bits
  [beacon, mpduLength] = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

  nonHTcfg = wlanNonHTConfig(...
    'ChannelBandwidth', chanBW,...
    "MCS", 1,...
    "PSDULength", mpduLength);
  txWaveform(:,i) = [wlanWaveformGenerator(beacon, nonHTcfg); zeros(20,1)];
end

txWaveform = txWaveform(:);

% Get center frequency for channel 153 in 5 GHz band
fc = helperWLANChannelFrequency(153, 5);
fs = wlanSampleRate(nonHTcfg);

txSig  = resample(txWaveform,osf,1);

% Samples per frame in Burst Mode
spf = length(txSig)/length(MACAddressesToSimulate);

runSDRSection = false;
if helperIsPlutoSDRInstalled()
  radios = findPlutoRadio();
  if length(radios) >= 2
    runSDRSection = true;
  else
    disp("Two ADALM-PLUTO radios are needed. Skipping SDR test.")
  end
else
    disp("Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found.
    disp("Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package."
    disp("Skipping SDR test.")
end


if runSDRSection
  % Set up PlutoSDR transmitter
  deviceNameSDR = 'Pluto';
```

```
  txGain = 0;
  txSDR = sdrtx(deviceNameSDR);
  txSDR.RadioID = 'usb:0';
  txSDR.BasebandSampleRate = fs*osf;
  txSDR.CenterFrequency = fc;
  txSDR.Gain = txGain;

  % Set up PlutoSDR Receiver
  rxSDR = sdrrx(deviceNameSDR);
  rxSDR.RadioID = 'usb:1';
  rxSDR.BasebandSampleRate = txSDR.BasebandSampleRate;
  rxSDR.CenterFrequency = txSDR.CenterFrequency;
  rxSDR.GainSource ='Manual';
  rxSDR.Gain = 30;
  rxSDR.OutputDataType = 'double';
  rxSDR.EnableBurstMode=true;
  rxSDR.NumFramesInBurst = 20;
  rxSDR.SamplesPerFrame = osf*spf;
end
```

L-LTF for Classification

The L-LTF sequence present in each beacon frame preamble is used as input units to the NN. rfFingerprintingNonHTFrontEnd System object is used to detect the WLAN packets, perform synchronization tasks and, extract the L-LTF sequences and data. In addition, the MAC address is also decoded. In addition, the data is pre-processed and classified using the trained network.

```
if runSDRSection
  numLLTF = 20;       % Number of L-LTF captured for Testing

  rxFrontEnd = rfFingerprintingNonHTFrontEnd('ChannelBandwidth', 'CBW20');

  disp("The known MAC addresses are:");
  disp(knownMACAddresses)

  % Set PlutoSDR to transmit repeatedly
  disp('Starting transmitter')
  transmitRepeat(txSDR, txSig);

  % Captured Frames counter
  numCapturedFrames = 0;

  disp('Starting receiver')
  % Loop until numLLTF frames are collected
  while numCapturedFrames < numLLTF

    % Receive data using PlutoSDR
    rxSig = rxSDR();
    rxSig = decimator(rxSig);

    % Perform front-end processing and payload buffering
    [payloadFull, cfgNonHT, rxNonHTData, chanEst, noiseVar, LLTF] = ...
      rxFrontEnd(rxSig);

    if payloadFull

      % Recover payload bits
      recBits = wlanNonHTDataRecover(rxNonHTData, chanEst, ...
```

```
        noiseVar, cfgNonHT, 'EqualizationMethod', 'ZF');

      % Decode and evaluate recovered bits
      [mpduCfg, ~, success] = wlanMPDUDecode(recBits, cfgNonHT);

      if success == wlanMACDecodeStatus.Success
        % Update counter
        numCapturedFrames = numCapturedFrames+1;

        % Create real-valued input
        LLTF = [real(LLTF), imag(LLTF)];
        LLTF = permute(reshape(LLTF,frameLength ,[] , 2, 1), [1 3 4 2]);

        ypred = classify(capturedDataNet, LLTF);

        if sum(contains(knownMACAddresses, mpduCfg.Address2)) ~= 0
          if categorical(convertCharsToStrings(mpduCfg.Address2))~=ypred
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint mismatch, ROUTE
          else
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint match"))
          end
        else
          disp(strcat("MAC Address ", mpduCfg.Address2," is not recognized, unknown device"));
        end
      end
    end
  end
  release(txSDR)
end
```

```
The known MAC addresses are:

    "71B63A2D0B83"    "A3F8AC0F2253"    "EF11D125044A"    "F636A97E07E7"

Starting transmitter

## Establishing connection to hardware. This process can take several seconds.
## Waveform transmission has started successfully and will repeat indefinitely.
## Call the release method to stop the transmission.

Starting receiver

## Establishing connection to hardware. This process can take several seconds.

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device
```

```
MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
```

Further Exploration

Capture data from your own routers as explained in Appendix: Known and Unknown Router Data Collection, on page 4-0 train the neural network with this data, and test the performance of the network.

Appendix: Helper Functions

- rfFingerprintingRouterDataCollection
- rfFingerprintingUnknownClassDataCollectionTx
- rfFingerprintingUnknownClassDataCollectionRx
- rfFingerprintingNonHTFrontEnd
- rfFingerprintingNonHTReceiver

Appendix: Known and Unknown Router Data Collection

Use `rfFingerprintingRouterDataCollection` to collect data from known (i.e. trusted) routers. This function extracts L-LTF signals present in 802.11a/g/n/ac OFDM Non-HT beacons frames transmitted from commercial 802.11 hardware. For more information see the “IEEE® 802.11™ WLAN - OFDM Beacon Receiver with USRP® Hardware” (Communications Toolbox Support Package for USRP Radio) example. L-LTF signals and corresponding router MAC addresses are used to train the RF fingerprinting neural network. This method works best if the routers and their antennas are fixed and hard to move unintentionally. For example, in most office environments, routers are mounted on the ceiling. Follow these steps:

1. Connect an ADALM-PLUTO radio to your PC to use as the observer radio.
2. Place the radio in a central location where it can receive signals from as many routers as possible. Fix the radio so that it does not move. If possible, place the observer radio on the ceiling or high on a wall.
3. Determine the channel number of the routers. You can use a Wi-Fi analyzer app on your phone to find out the channel numbers.
4. Start data collection by running "`rfFingerprintingRouterDataCollection(channel)`" where channel is the Wi-Fi channel number
5. Monitor the "max(abs(LLTF))" value. If it is above 1.2 or smaller than 0.01, adjust the gain of the receiver using the GAIN input of `rfFingerprintingRouterDataCollection` function.

Use the helper functions `rfFingerprintingUnknownClassDataCollectionTx` and `rfFingerprintingUnknownClassDataCollectionRx` to collect data from unknown routers. These functions set two ADALM-PLUTO radios to transmit and receive L-LTF signals. The received signals are combined with the known router signals to train the neural network. You need two ADALM-PLUTO radios, preferably connected to two separate PCs. Follow these steps:

1. Connect an ADALM-PLUTO radio to a stationary PC to act as the unknown router
2. Start transmissions by running "`rfFingerprintingUnknownClassDataCollectionTx`"
3. Connect another ADALM-PLUTO radio to a mobile PC to act as the observer
4. Start data collection by running "`rfFingerprintingUnknownClassDataCollectionRx`". This function by default collects 200 frames per location. Each location represents a different unknown router.
5. When the function instructs you to move to a new location, move the observer radio to a new location. By default, this function collects data from 10 locations.
6. If the observer does not receive any beacons or it rarely receives beacons, move the observer closer to the transmitter.
7. Once the data collection is done, call "`release(sdrTransmitter)`" in the transmitting radio's MATLAB session.

Selected Bibliography

[1] K. Sankhe, M. Belgiovine, F. Zhou, S. Riyaz, S. Ioannidis and K. Chowdhury, "ORACLE: Optimized Radio clAssification through Convolutional neuraL nEtworks," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 370-378.

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Modulation Classification with Deep Learning

This example shows how to use a convolutional neural network (CNN) for modulation classification. You generate synthetic, channel-impaired waveforms. Using the generated waveforms as training data, you train a CNN for modulation classification. You then test the CNN with software-defined radio (SDR) hardware and over-the-air signals.

Predict Modulation Type Using CNN

The trained CNN in this example recognizes these eight digital and three analog modulation types:

- Binary phase shift keying (BPSK)
- Quadrature phase shift keying (QPSK)
- 8-ary phase shift keying (8-PSK)
- 16-ary quadrature amplitude modulation (16-QAM)
- 64-ary quadrature amplitude modulation (64-QAM)
- 4-ary pulse amplitude modulation (PAM4)
- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast FM (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

```
modulationTypes = categorical(["BPSK", "QPSK", "8PSK", ...
  "16QAM", "64QAM", "PAM4", "GFSK", "CPFSK", ...
  "B-FM", "DSB-AM", "SSB-AM"]);
```

First, load the trained network. For details on network training, see the Training a CNN on page 4-0 section.

```
load trainedModulationClassificationNetwork
trainedNet
```

```
trainedNet = SeriesNetwork with properties:
         Layers: [28×1 nnet.cnn.layer.Layer]
     InputNames: {'Input Layer'}
    OutputNames: {'Output'}
```

The trained CNN takes 1024 channel-impaired samples and predicts the modulation type of each frame. Generate several PAM4 frames that are impaired with Rician multipath fading, center frequency and sampling time drift, and AWGN. Use following function to generate synthetic signals to test the CNN. Then use the CNN to predict the modulation type of the frames.

- `randi`: Generate random bits
- `pammod` PAM4-modulate the bits
- `rcosdesign`: Design a square-root raised cosine pulse shaping filter
- `filter`: Pulse shape the symbols
- `comm.RicianChannel`: Apply Rician multipath channel

- `comm.PhaseFrequencyOffset`: Apply phase and/or frequency shift due to clock offset
- `interp1`: Apply timing drift due to clock offset
- `awgn`: Add AWGN

```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
rng(123456)
% Random bits
d = randi([0 3], 1024, 1);
% PAM4 modulation
syms = pammod(d,4);
% Square-root raised cosine filter
filterCoeffs = rcosdesign(0.35,4,8);
tx = filter(filterCoeffs,1,upsample(syms,8));

% Channel
SNR = 30;
maxOffset = 5;
fc = 902e6;
fs = 200e3;
multipathChannel = comm.RicianChannel(...
  'SampleRate', fs, ...
  'PathDelays', [0 1.8 3.4] / 200e3, ...
  'AveragePathGains', [0 -2 -10], ...
  'KFactor', 4, ...
  'MaximumDopplerShift', 4);

frequencyShifter = comm.PhaseFrequencyOffset(...
  'SampleRate', fs);

% Apply an independent multipath channel
reset(multipathChannel)
outMultipathChan = multipathChannel(tx);

% Determine clock offset factor
clockOffset = (rand() * 2*maxOffset) - maxOffset;
C = 1 + clockOffset / 1e6;

% Add frequency offset
frequencyShifter.FrequencyOffset = -(C-1)*fc;
outFreqShifter = frequencyShifter(outMultipathChan);

% Add sampling time drift
t = (0:length(tx)-1)' / fs;
newFs = fs * C;
tp = (0:length(tx)-1)' / newFs;
outTimeDrift = interp1(t, outFreqShifter, tp);

% Add noise
rx = awgn(outTimeDrift,SNR,0);

% Frame generation for classification
unknownFrames = helperModClassGetNNFrames(rx);

% Classification
[prediction1,score1] = classify(trainedNet,unknownFrames);
```

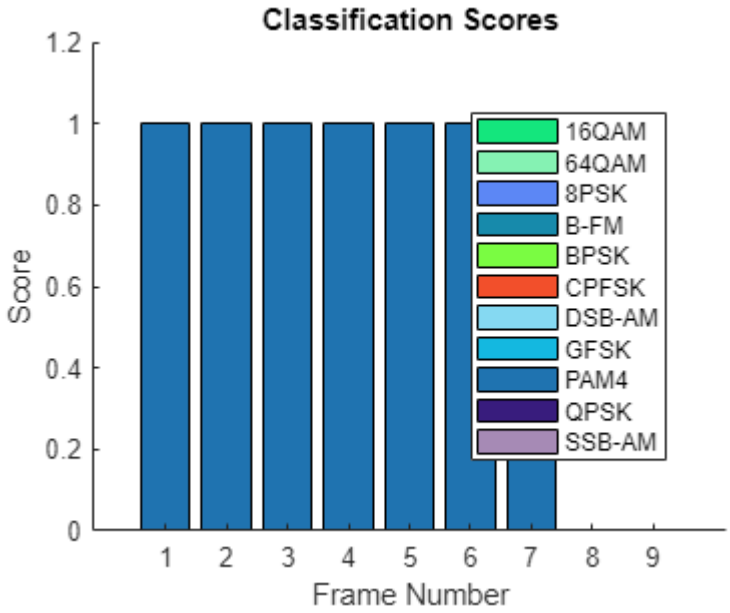
Return the classifier predictions, which are analogous to hard decisions. The network correctly identifies the frames as PAM4 frames. For details on the generation of the modulated signals, see helperModClassGetModulator function.

```
prediction1
```

```
prediction1 = 7×1 categorical
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
```

The classifier also returns a vector of scores for each frame. The score corresponds to the probability that each frame has the predicted modulation type. Plot the scores.

```
helperModClassPlotScores(score1,modulationTypes)
```



Before we can use a CNN for modulation classification, or any other task, we first need to train the network with known (or labeled) data. The first part of this example shows how to use Communications Toolbox features, such as modulators, filters, and channel impairments, to generate synthetic training data. The second part focuses on defining, training, and testing the CNN for the task of modulation classification. The third part tests the network performance with over-the-air signals using software defined radio (SDR) platforms.

Waveform Generation for Training

Generate 10,000 frames for each modulation type, where 80% is used for training, 10% is used for validation and 10% is used for testing. We use training and validation frames during the network training phase. Final classification accuracy is obtained using test frames. Each frame is 1024 samples long and has a sample rate of 200 kHz. For digital modulation types, eight samples represent

a symbol. The network makes each decision based on single frames rather than on multiple consecutive frames (as in video). Assume a center frequency of 902 MHz and 100 MHz for the digital and analog modulation types, respectively.

To run this example quickly, use the trained network and generate a small number of training frames. To train the network on your computer, choose the "Train network now" option (i.e. set trainNow to true).

```
trainNow = Use trained network;
if trainNow == true
  numFramesPerModType = 10000;
else
  numFramesPerModType = 200;
end
percentTrainingSamples = 80;
percentValidationSamples = 10;
percentTestSamples = 10;

sps = 8;                % Samples per symbol
spf = 1024;             % Samples per frame
symbolsPerFrame = spf / sps;
fs = 200e3;             % Sample rate
fc = [902e6 100e6];     % Center frequencies
```

Create Channel Impairments

Pass each frame through a channel with

- AWGN
- Rician multipath fading
- Clock offset, resulting in center frequency offset and sampling time drift

Because the network in this example makes decisions based on single frames, each frame must pass through an independent channel.

AWGN

The channel adds AWGN with an SNR of 30 dB. Implement the channel using `awgn` function.

Rician Multipath

The channel passes the signals through a Rician multipath fading channel using the `comm.RicianChannel` System object. Assume a delay profile of [0 1.8 3.4] samples with corresponding average path gains of [0 -2 -10] dB. The K-factor is 4 and the maximum Doppler shift is 4 Hz, which is equivalent to a walking speed at 902 MHz. Implement the channel with the following settings.

Clock Offset

Clock offset occurs because of the inaccuracies of internal clock sources of transmitters and receivers. Clock offset causes the center frequency, which is used to downconvert the signal to baseband, and the digital-to-analog converter sampling rate to differ from the ideal values. The channel simulator uses the clock offset factor C, expressed as $C = 1 + \frac{\Delta_{\text{clock}}}{10^6}$, where Δ_{clock} is the clock offset. For each frame, the channel generates a random Δ_{clock} value from a uniformly distributed set

of values in the range $[-\max\Delta_{\text{clock}}\ \max\Delta_{\text{clock}}]$, where $\max\Delta_{\text{clock}}$ is the maximum clock offset. Clock offset is measured in parts per million (ppm). For this example, assume a maximum clock offset of 5 ppm.

```
maxDeltaOff = 5;
deltaOff = (rand()*2*maxDeltaOff) - maxDeltaOff;
C = 1 + (deltaOff/1e6);
```

Frequency Offset

Subject each frame to a frequency offset based on clock offset factor C and the center frequency. Implement the channel using `comm.PhaseFrequencyOffset`.

Sampling Rate Offset

Subject each frame to a sampling rate offset based on clock offset factor C. Implement the channel using the `interp1` function to resample the frame at the new rate of $C \times f_s$.

Combined Channel

Use the helperModClassTestChannel object to apply all three channel impairments to the frames.

```
channel = helperModClassTestChannel(...
  'SampleRate', fs, ...
  'SNR', SNR, ...
  'PathDelays', [0 1.8 3.4] / fs, ...
  'AveragePathGains', [0 -2 -10], ...
  'KFactor', 4, ...
  'MaximumDopplerShift', 4, ...
  'MaximumClockOffset', 5, ...
  'CenterFrequency', 902e6)
```

```
channel = helperModClassTestChannel with properties:
                    SNR: 30
        CenterFrequency: 902000000
             SampleRate: 200000
             PathDelays: [0 9.0000e-06 1.7000e-05]
       AveragePathGains: [0 -2 -10]
                KFactor: 4
    MaximumDopplerShift: 4
     MaximumClockOffset: 5
```

You can view basic information about the channel using the info object function.

```
chInfo = info(channel)
```

```
chInfo = struct with fields:
               ChannelDelay: 6
     MaximumFrequencyOffset: 4510
    MaximumSampleRateOffset: 1
```

Waveform Generation

Create a loop that generates channel-impaired frames for each modulation type and stores the frames with their corresponding labels in MAT files. By saving the data into files, you eliminate the need to generate the data every time you run this example. You can also share the data more effectively.

Remove a random number of samples from the beginning of each frame to remove transients and to make sure that the frames have a random starting point with respect to the symbol boundaries.

```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
rng(1235)
tic

numModulationTypes = length(modulationTypes);

channelInfo = info(channel);
transDelay = 50;
dataDirectory = fullfile(tempdir,"ModClassDataFiles");
disp("Data file directory is " + dataDirectory)
```

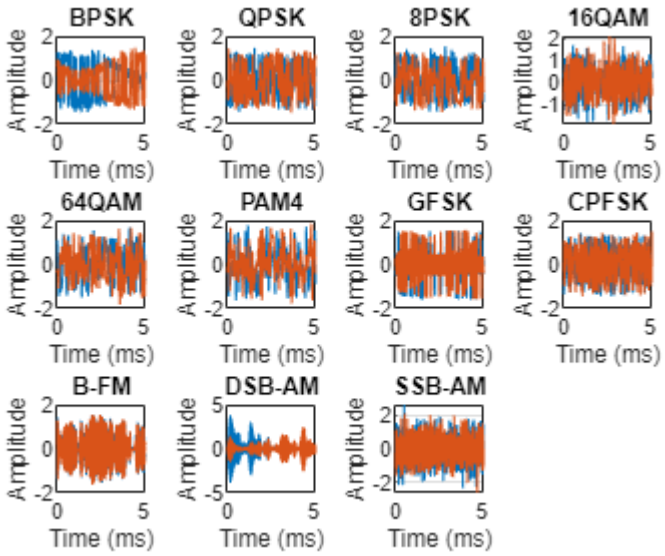
```
Data file directory is C:\TEMP\ModClassDataFiles
```

```
fileNameRoot = "frame";

% Check if data files exist
dataFilesExist = false;
if exist(dataDirectory,'dir')
  files = dir(fullfile(dataDirectory,sprintf("%s*",fileNameRoot)));
  if length(files) == numModulationTypes*numFramesPerModType
    dataFilesExist = true;
  end
end

if ~dataFilesExist
  disp("Generating data and saving in data files...")
  [success,msg,msgID] = mkdir(dataDirectory);
  if ~success
    error(msgID,msg)
  end
  for modType = 1:numModulationTypes
    elapsedTime = seconds(toc);
    elapsedTime.Format = 'hh:mm:ss';
    fprintf('%s - Generating %s frames\n', ...
      elapsedTime, modulationTypes(modType))

    label = modulationTypes(modType);
    numSymbols = (numFramesPerModType / sps);
    dataSrc = helperModClassGetSource(modulationTypes(modType), sps, 2*spf, fs);
    modulator = helperModClassGetModulator(modulationTypes(modType), sps, fs);
    if contains(char(modulationTypes(modType)), {'B-FM','DSB-AM','SSB-AM'})
      % Analog modulation types use a center frequency of 100 MHz
      channel.CenterFrequency = 100e6;
    else
      % Digital modulation types use a center frequency of 902 MHz
      channel.CenterFrequency = 902e6;
    end

    for p=1:numFramesPerModType
      % Generate random data
      x = dataSrc();

      % Modulate
      y = modulator(x);
```

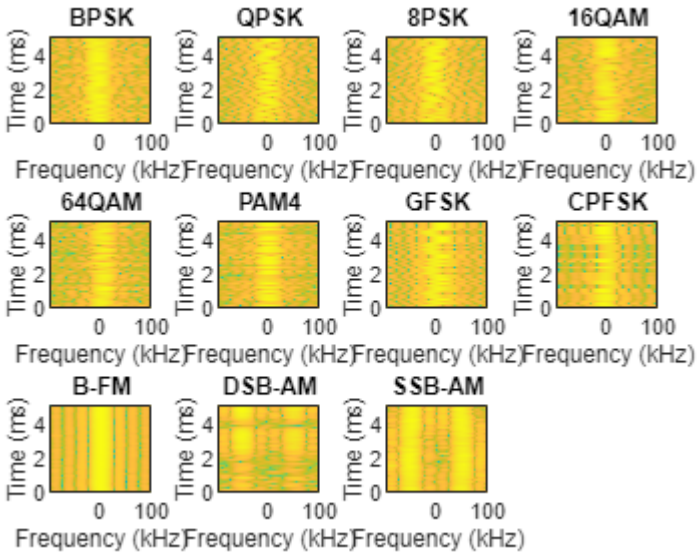
```
        % Pass through independent channels
        rxSamples = channel(y);

        % Remove transients from the beginning, trim to size, and normalize
        frame = helperModClassFrameGenerator(rxSamples, spf, spf, transDelay, sps);

        % Save data file
        fileName = fullfile(dataDirectory,...
          sprintf("%s%s%03d",fileNameRoot,modulationTypes(modType),p));
        save(fileName,"frame","label")
      end
    end
  else
    disp("Data files exist. Skip data generation.")
  end
```

```
Generating data and saving in data files...

00:00:00 - Generating BPSK frames
00:00:01 - Generating QPSK frames
00:00:02 - Generating 8PSK frames
00:00:04 - Generating 16QAM frames
00:00:05 - Generating 64QAM frames
00:00:06 - Generating PAM4 frames
00:00:08 - Generating GFSK frames
00:00:09 - Generating CPFSK frames
00:00:11 - Generating B-FM frames
00:00:12 - Generating DSB-AM frames
00:00:13 - Generating SSB-AM frames
```

```
% Plot the amplitude of the real and imaginary parts of the example frames
% against the sample number
helperModClassPlotTimeDomain(dataDirectory,modulationTypes,fs)
```



```
% Plot the spectrogram of the example frames
helperModClassPlotSpectrogram(dataDirectory,modulationTypes,fs,sps)
```



Create a Datastore

Use a `signalDatastore` object to manage the files that contain the generated complex waveforms. Datastores are especially useful when each individual file fits in memory, but the entire collection does not necessarily fit.

```
frameDS = signalDatastore(dataDirectory,'SignalVariableNames',["frame","label"]);
```

Transform Complex Signals to Real Arrays

The deep learning network in this example expects real inputs while the received signal has complex baseband samples. Transform the complex signals into real valued 4-D arrays. The output frames have size 1-by-spf-by-2-by-N, where the first page (3rd dimension) is in-phase samples and the second page is quadrature samples. When the convolutional filters are of size 1-by-spf, this approach ensures that the information in the I and Q gets mixed even in the convolutional layers and makes better use of the phase information. See helperModClassIQAsPages for details.

```
frameDSTrans = transform(frameDS,@helperModClassIQAsPages);
```

Split into Training, Validation, and Test

Next divide the frames into training, validation, and test data. See helperModClassSplitData for details.

```
splitPercentages = [percentTrainingSamples,percentValidationSamples,percentTestSamples];
[trainDSTrans,validDSTrans,testDSTrans] = helperModClassSplitData(frameDSTrans,splitPercentages);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Import Data Into Memory

Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. If the data fits into the memory of your

computer, importing the data from the files into the memory enables faster training by eliminating this repeated read from file and transform process. Instead, the data is read from the files and transformed once. Training this network using data files on disk takes about 110 minutes while training using in-memory data takes about 50 min.

Import all the data in the files into memory. The files have two variables: `frame` and `label` and each `read` call to the datastore returns a cell array, where the first element is the `frame` and the second element is the `label`. Use the `transform` functions helperModClassReadFrame and helperModClassReadLabel to read frames and labels. Use `readall` with `"UseParallel"` option set to `true` to enable parallel processing of the transform functions, in case you have Parallel Computing Toolbox license. Since `readall` function, by default, concatenates the output of the `read` function over the first dimension, return the frames in a cell array and manually concatenate over the 4th dimension.

```
% Read the training and validation frames into the memory
pctExists = parallelComputingLicenseExists();
trainFrames = transform(trainDSTrans, @helperModClassReadFrame);
rxTrainFrames = readall(trainFrames,"UseParallel",pctExists);
rxTrainFrames = cat(4, rxTrainFrames{:});
validFrames = transform(validDSTrans, @helperModClassReadFrame);
rxValidFrames = readall(validFrames,"UseParallel",pctExists);
rxValidFrames = cat(4, rxValidFrames{:});

% Read the training and validation labels into the memory
trainLabels = transform(trainDSTrans, @helperModClassReadLabel);
rxTrainLabels = readall(trainLabels,"UseParallel",pctExists);
validLabels = transform(validDSTrans, @helperModClassReadLabel);
rxValidLabels = readall(validLabels,"UseParallel",pctExists);
```

Train the CNN

This example uses a CNN that consists of six convolution layers and one fully connected layer. Each convolution layer except the last is followed by a batch normalization layer, rectified linear unit (ReLU) activation layer, and max pooling layer. In the last convolution layer, the max pooling layer is replaced with an average pooling layer. The output layer has softmax activation. For network design guidance, see “Deep Learning Tips and Tricks” (Deep Learning Toolbox).

```
modClassNet = helperModClassCNN(modulationTypes,sps,spf);
```

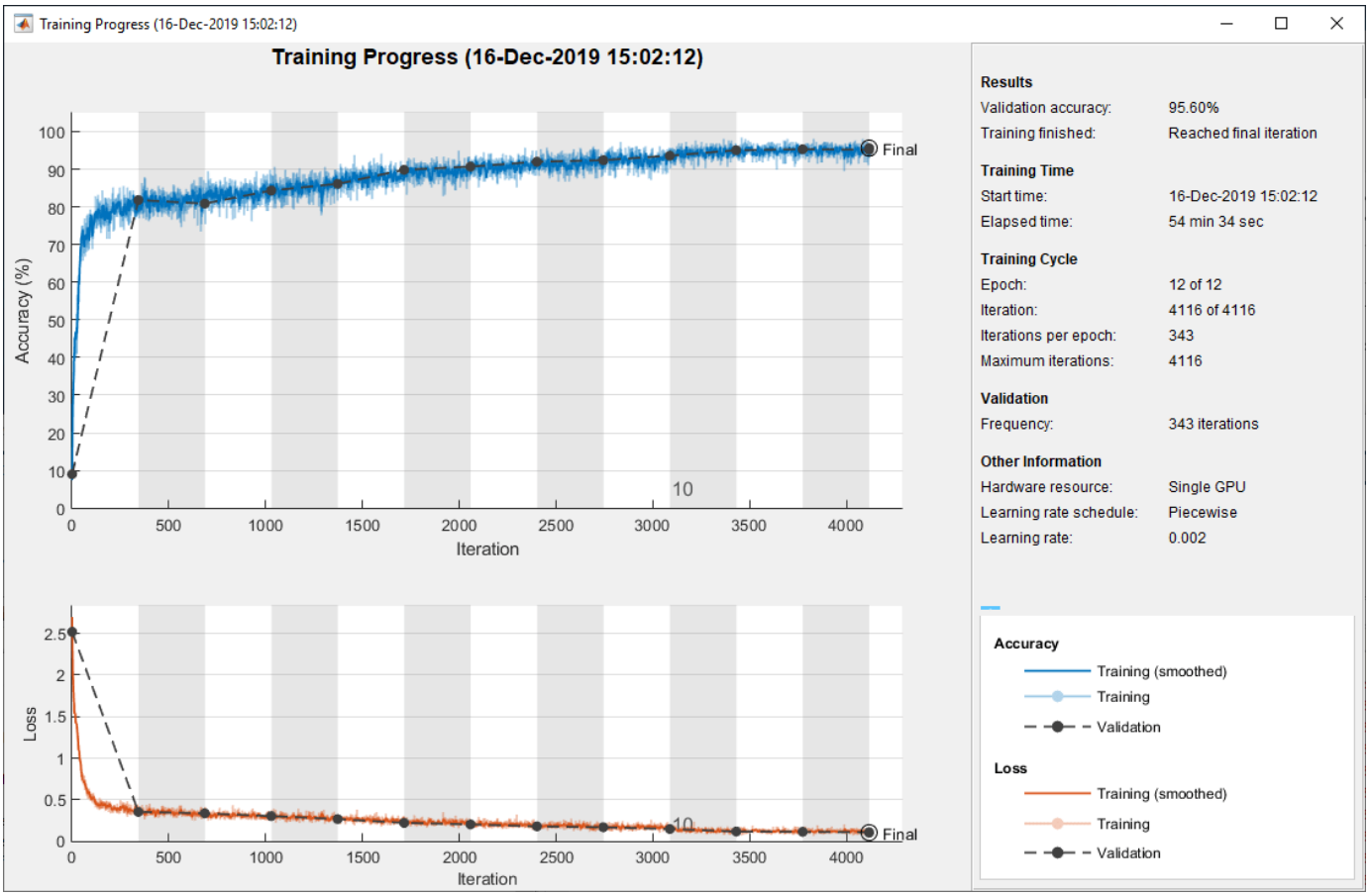
Next configure `TrainingOptionsSGDM` (Deep Learning Toolbox) to use an SGDM solver with a mini-batch size of 256. Set the maximum number of epochs to 12, since a larger number of epochs provides no further training advantage. By default, the `'ExecutionEnvironment'` property is set to `'auto'`, where the `trainNetwork` function uses a GPU if one is available or uses the CPU, if not. To use the GPU, you must have a Parallel Computing Toolbox license. Set the initial learning rate to $2x10^{-2}$. Reduce the learning rate by a factor of 10 every 9 epochs. Set `'Plots'` to `'training-progress'` to plot the training progress. On an NVIDIA Titan Xp GPU, the network takes approximately 25 minutes to train. .

```
maxEpochs = 12;
miniBatchSize = 256;
options = helperModClassTrainingOptions(maxEpochs,miniBatchSize,...
  numel(rxTrainLabels),rxValidFrames,rxValidLabels);
```

Either train the network or use the already trained network. By default, this example uses the trained network.

```
if trainNow == true
  elapsedTime = seconds(toc);
  elapsedTime.Format = 'hh:mm:ss';
  fprintf('%s - Training the network\n', elapsedTime)
  trainedNet = trainNetwork(rxTrainFrames,rxTrainLabels,modClassNet,options);
else
  load trainedModulationClassificationNetwork
end
```

As the plot of the training progress shows, the network converges in about 12 epochs to more than 95% accuracy.



Evaluate the trained network by obtaining the classification accuracy for the test frames. The results show that the network achieves about 94% accuracy for this group of waveforms.

```
elapsedTime = seconds(toc);
elapsedTime.Format = 'hh:mm:ss';
fprintf('%s - Classifying test frames\n', elapsedTime)
```

```
00:01:25 - Classifying test frames
```

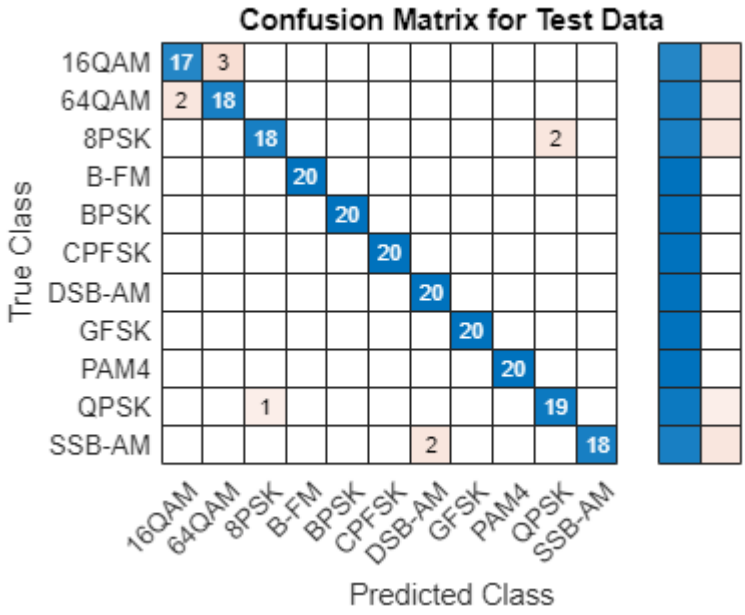
```
% Read the test frames into the memory
testFrames = transform(testDSTrans, @helperModClassReadFrame);
rxTestFrames = readall(testFrames,"UseParallel",pctExists);
rxTestFrames = cat(4, rxTestFrames{:});
```

```
% Read the test labels into the memory
testLabels = transform(testDSTrans, @helperModClassReadLabel);
rxTestLabels = readall(testLabels,"UseParallel",pctExists);

rxTestPred = classify(trainedNet,rxTestFrames);
testAccuracy = mean(rxTestPred == rxTestLabels);
disp("Test accuracy: " + testAccuracy*100 + "%")
```

```
Test accuracy: 95.4545%
```

Plot the confusion matrix for the test frames. As the matrix shows, the network confuses 16-QAM and 64-QAM frames. This problem is expected since each frame carries only 128 symbols and 16-QAM is a subset of 64-QAM. The network also confuses QPSK and 8-PSK frames, since the constellations of these modulation types look similar once phase-rotated due to the fading channel and frequency offset.

```
figure
cm = confusionchart(rxTestLabels, rxTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
cm.Parent.Position = [cm.Parent.Position(1:2) 740 424];
```



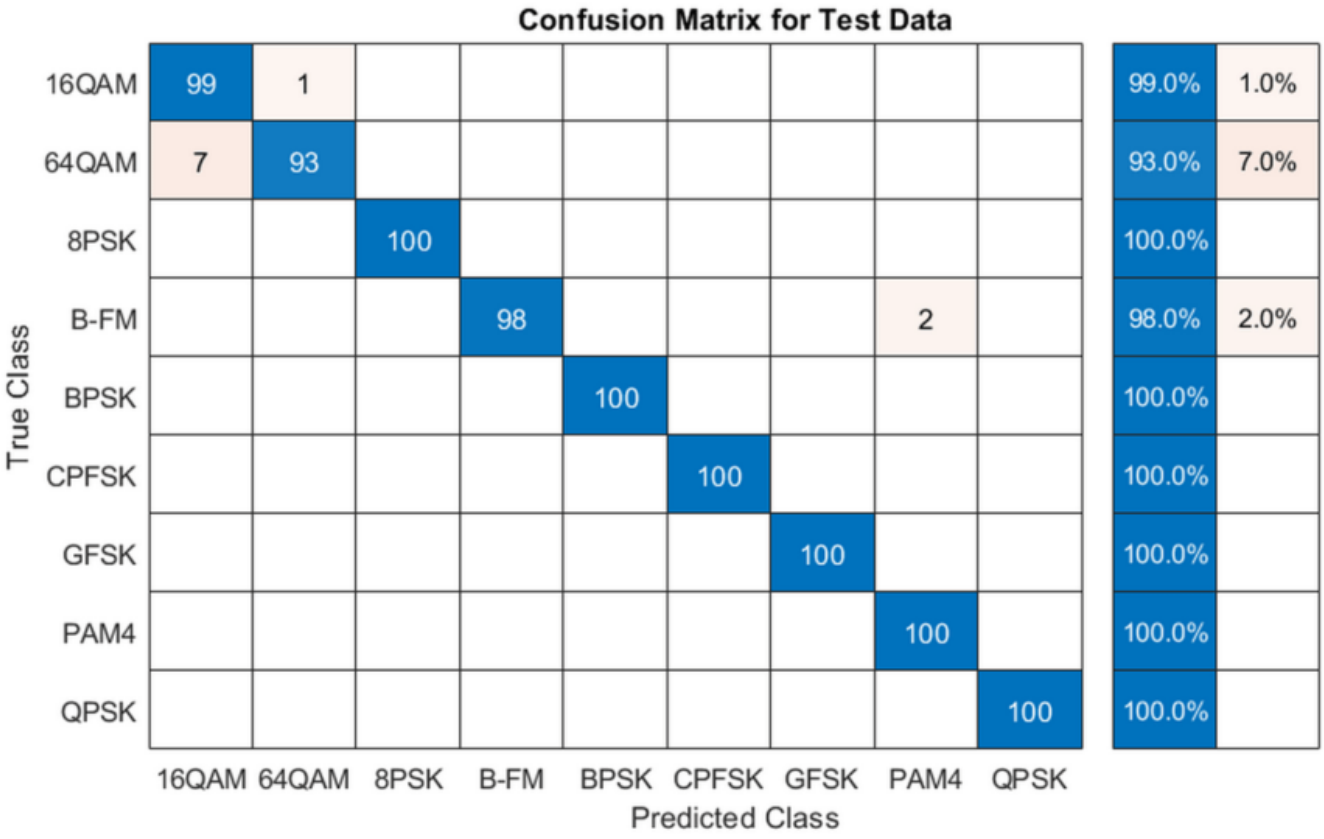
Test with SDR

Test the performance of the trained network with over-the-air signals using the helperModClassSDRTest function. To perform this test, you must have dedicated SDRs for transmission and reception. You can use two ADALM-PLUTO radios, or one ADALM-PLUTO radio for transmission and one USRP® radio for reception. You must install Communications Toolbox Support Package for ADALM-PLUTO Radio. If you are using a USRP® radio, you must also install Communications Toolbox Support Package for USRP® Radio. The `helperModClassSDRTest` function uses the same modulation functions as used for generating the training signals, and then transmits them using an ADALM-PLUTO radio. Instead of simulating the channel, capture the channel-impaired signals using the SDR that is configured for signal reception (ADALM-PLUTO or USRP® radio). Use the trained network with the same `classify` function used previously to predict

the modulation type. Running the next code segment produces a confusion matrix and prints out the test accuracy.

```
radioPlatform = ADALM-PLUTO;

switch radioPlatform
  case "ADALM-PLUTO"
    if helperIsPlutoSDRInstalled() == true
      radios = findPlutoRadio();
      if length(radios) >= 2
        helperModClassSDRTest(radios);
      else
        disp('Selected radios not found. Skipping over-the-air test.')
      end
    end
  case {"USRP B2xx","USRP X3xx","USRP N2xx"}
    if (helperIsUSRPInstalled() == true) && (helperIsPlutoSDRInstalled() == true)
      txRadio = findPlutoRadio();
      rxRadio = findsdru();
      switch radioPlatform
        case "USRP B2xx"
          idx = contains({rxRadio.Platform}, {'B200','B210'});
        case "USRP X3xx"
          idx = contains({rxRadio.Platform}, {'X300','X310'});
        case "USRP N2xx"
          idx = contains({rxRadio.Platform}, 'N200/N210/USRP2');
      end
      rxRadio = rxRadio(idx);
      if (length(txRadio) >= 1) && (length(rxRadio) >= 1)
        helperModClassSDRTest(rxRadio);
      else
        disp('Selected radios not found. Skipping over-the-air test.')
      end
    end
end
```

When using two stationary ADALM-PLUTO radios separated by about 2 feet, the network achieves 99% overall accuracy with the following confusion matrix. Results will vary based on experimental setup.



Further Exploration

It is possible to optimize the hyperparameters parameters, such as number of filters, filter size, or optimize the network structure, such as adding more layers, using different activation layers, etc. to improve the accuracy.

Communication Toolbox provides many more modulation types and channel impairments. For more information see “Modulation” and “Propagation and Channel Models” sections. You can also add standard specific signals with LTE Toolbox, WLAN Toolbox, and 5G Toolbox. You can also add radar signals with Phased Array System Toolbox.

helperModClassGetModulator function provides the MATLAB functions used to generate modulated signals. You can also explore the following functions and System objects for more details:

- helperModClassGetModulator.m
- helperModClassTestChannel.m
- helperModClassGetSource.m
- helperModClassFrameGenerator.m
- helperModClassCNN.m
- helperModClassTrainingOptions.m

References

1. O'Shea, T. J., J. Corgan, and T. C. Clancy. "Convolutional Radio Modulation Recognition Networks." Preprint, submitted June 10, 2016. https://arxiv.org/abs/1602.04105
2. O'Shea, T. J., T. Roy, and T. C. Clancy. "Over-the-Air Deep Learning Based Radio Signal Classification." IEEE Journal of Selected Topics in Signal Processing. Vol. 12, Number 1, 2018, pp. 168–179.
3. Liu, X., D. Yang, and A. E. Gamal. "Deep Neural Network Architectures for Modulation Classification." Preprint, submitted January 5, 2018. https://arxiv.org/abs/1712.00443v3

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Shared phased_comm Examples (comm/phased)

Massive MIMO Hybrid Beamforming

This example shows how hybrid beamforming is employed at the transmit end of a massive MIMO communications system, using techniques for both multi-user and single-user systems. The example employs full channel sounding for determining the channel state information at the transmitter. It partitions the required precoding into digital baseband and analog RF components, using different techniques for multi-user and single-user systems. Simplified all-digital receivers recover the multiple transmitted data streams to highlight the common figures of merit for a communications system, namely, EVM, and BER.

The example employs a scattering-based spatial channel model which accounts for the transmit/receive spatial locations and antenna patterns. A simpler static-flat MIMO channel is also offered for link validation purposes.

The example requires Communications Toolbox™ and Phased Array System Toolbox™.

Introduction

The ever-growing demand for high data rate and more user capacity increases the need to use the available spectrum more efficiently. Multi-user MIMO (MU-MIMO) improves the spectrum efficiency by allowing a base station (BS) transmitter to communicate simultaneously with multiple mobile stations (MS) receivers using the same time-frequency resources. Massive MIMO allows the number of BS antenna elements to be on the order of tens or hundreds, thereby also increasing the number of data streams in a cell to a large value.

The next generation, 5G, wireless systems use millimeter wave (mmWave) bands to take advantage of their wider bandwidth. The 5G systems also deploy large scale antenna arrays to mitigate severe propagation loss in the mmWave band.

Compared to current wireless systems, the wavelength in the mmWave band is much smaller. Although this allows an array to contain more elements within the same physical dimension, it becomes much more expensive to provide one transmit-receive (TR) module, or an RF chain, for each antenna element. Hybrid transceivers are a practical solution as they use a combination of analog beamformers in the RF and digital beamformers in the baseband domains, with fewer RF chains than the number of transmit elements [1].

This example uses a multi-user MIMO-OFDM system to highlight the partitioning of the required precoding into its digital baseband and RF analog components at the transmitter end. Building on the system highlighted in the “MIMO-OFDM Precoding with Phased Arrays” (Phased Array System Toolbox) example, this example shows the formulation of the transmit-end precoding matrices and their application to a MIMO-OFDM system.

```
s = rng(67);                  % Set RNG state for repeatability
```

System Parameters

Define system parameters for the example. Modify these parameters to explore their impact on the system.

```
% Multi-user system with single/multiple streams per user
prm.numUsers = 4;                 % Number of users
prm.numSTSVec = [3 2 1 2];        % Number of independent data streams per user
prm.numSTS = sum(prm.numSTSVec);  % Must be a power of 2
prm.numTx = prm.numSTS*8;         % Number of BS transmit antennas (power of 2)
prm.numRx = prm.numSTSVec*4;      % Number of receive antennas, per user (any >= numSTSVec)
```

```
% Each user has the same modulation
prm.bitsPerSubCarrier = 4;   % 2: QPSK, 4: 16QAM, 6: 64QAM, 8: 256QAM
prm.numDataSymbols = 10;     % Number of OFDM data symbols

% MS positions: assumes BS at origin
%   Angles specified as [azimuth;elevation] degrees
%   az in range [-180 180], el in range [-90 90], e.g. [45;0]
maxRange = 1000;            % all MSs within 1000 meters of BS
prm.mobileRanges = randi([1 maxRange],1,prm.numUsers);
prm.mobileAngles = [rand(1,prm.numUsers)*360-180; ...
                    rand(1,prm.numUsers)*180-90];

prm.fc = 28e9;               % 28 GHz system
prm.chanSRate = 100e6;       % Channel sampling rate, 100 Msps
prm.ChanType = 'Scattering'; % Channel options: 'Scattering', 'MIMO'
prm.NFig = 8;                % Noise figure (increase to worsen, 5-10 dB)
prm.nRays = 500;             % Number of rays for Frf, Fbb partitioning
```

Define OFDM modulation parameters used for the system.

```
prm.FFTLength = 256;
prm.CyclicPrefixLength = 64;
prm.numCarriers = 234;
prm.NullCarrierIndices = [1:7 129 256-5:256]'; % Guards and DC
prm.PilotCarrierIndices = [26 54 90 118 140 168 204 232]';
nonDataIdx = [prm.NullCarrierIndices; prm.PilotCarrierIndices];
prm.CarriersLocations = setdiff((1:prm.FFTLength)', sort(nonDataIdx));

numSTS = prm.numSTS;
numTx = prm.numTx;
numRx = prm.numRx;
numSTSVec = prm.numSTSVec;
codeRate = 1/3;             % same code rate per user
numTails = 6;               % number of termination tail bits
prm.numFrmBits = numSTSVec.*(prm.numDataSymbols*prm.numCarriers* ...
                 prm.bitsPerSubCarrier*codeRate)-numTails;
prm.modMode = 2^prm.bitsPerSubCarrier; % Modulation order
% Account for channel filter delay
numPadSym = 3;          % number of symbols to zeropad
prm.numPadZeros = numPadSym*(prm.FFTLength+prm.CyclicPrefixLength);
```

Define transmit and receive arrays and positional parameters for the system.

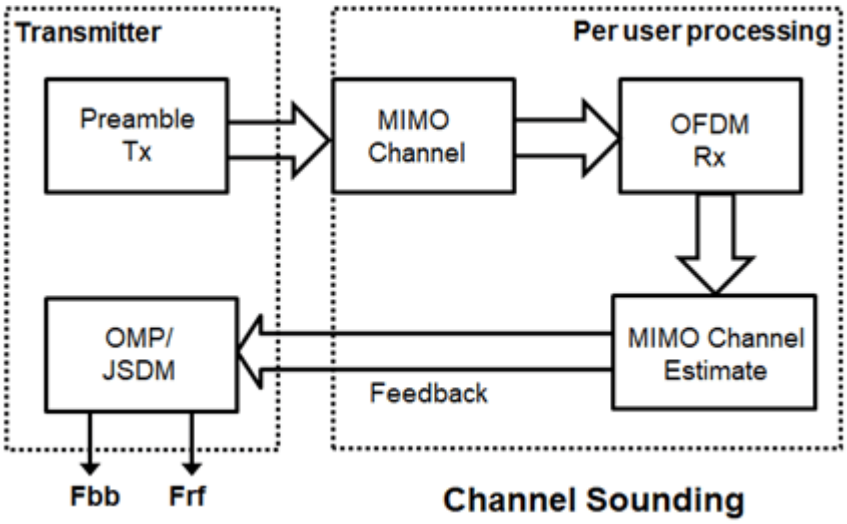
```
prm.cLight = physconst('LightSpeed');
prm.lambda = prm.cLight/prm.fc;

% Get transmit and receive array information
[isTxURA,expFactorTx,isRxURA,expFactorRx] = helperArrayInfo(prm,true);

% Transmit antenna array definition
%   Array locations and angles
prm.posTx = [0;0;0];       % BS/Transmit array position, [x;y;z], meters
if isTxURA
    % Uniform Rectangular array
    txarray = phased.PartitionedArray(...
        'Array',phased.URA([expFactorTx numSTS],0.5*prm.lambda),...
        'SubarraySelection',ones(numSTS,numTx),'SubarraySteering','Custom');
```

```
else
    % Uniform Linear array
    txarray = phased.ULA(numTx, 'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',false));
end
prm.posTxElem = getElementPosition(txarray)/prm.lambda;

spLoss = zeros(prm.numUsers,1);
prm.posRx = zeros(3,prm.numUsers);
for uIdx = 1:prm.numUsers

    % Receive arrays
    if isRxURA(uIdx)
        % Uniform Rectangular array
        rxarray = phased.PartitionedArray(...
            'Array',phased.URA([expFactorRx(uIdx) numSTSVec(uIdx)], ...
            0.5*prm.lambda),'SubarraySelection',ones(numSTSVec(uIdx), ...
            numRx(uIdx)),'SubarraySteering','Custom');
        prm.posRxElem = getElementPosition(rxarray)/prm.lambda;
    else
        if numRx(uIdx)>1
            % Uniform Linear array
            rxarray = phased.ULA(numRx(uIdx), ...
                'ElementSpacing',0.5*prm.lambda, ...
                'Element',phased.IsotropicAntennaElement);
            prm.posRxElem = getElementPosition(rxarray)/prm.lambda;
        else
            rxarray = phased.IsotropicAntennaElement;
            prm.posRxElem = [0; 0; 0]; % LCS
        end
    end

    % Mobile positions
    [xRx,yRx,zRx] = sph2cart(deg2rad(prm.mobileAngles(1,uIdx)), ...
                             deg2rad(prm.mobileAngles(2,uIdx)), ...
                             prm.mobileRanges(uIdx));
    prm.posRx(:,uIdx) = [xRx;yRx;zRx];
    [toRxRange,toRxAng] = rangeangle(prm.posTx,prm.posRx(:,uIdx));
    spLoss(uIdx) = fspl(toRxRange,prm.lambda);
end
```

Channel State Information

For a spatially multiplexed system, availability of channel information at the transmitter allows for precoding to be applied to maximize the signal energy in the direction and channel of interest. Under the assumption of a slowly varying channel, this is facilitated by sounding the channel first. The BS sounds the channel by using a reference transmission, that the MS receiver uses to estimate the channel. The MS transmits the channel estimate information back to the BS for calculation of the precoding needed for the subsequent data transmission.

The following schematic shows the processing for the channel sounding modeled.



For the chosen MIMO system, a preamble signal is sent over all transmitting antenna elements, and processed at the receiver accounting for the channel. The receiver antenna elements perform pre-amplification, OFDM demodulation, and frequency domain channel estimation for all links.

```
% Generate the preamble signal
prm.numSTS = numTx;             % set to numTx to sound out all channels
preambleSig = helperGenPreamble(prm);

% Transmit preamble over channel
prm.numSTS = numSTS;            % keep same array config for channel
[rxPreSig,chanDelay] = helperApplyMUChannel(preambleSig,prm,spLoss);

% Channel state information feedback
hDp = cell(prm.numUsers,1);
prm.numSTS = numTx;             % set to numTx to estimate all links
for uIdx = 1:prm.numUsers

    % Front-end amplifier gain and thermal noise
    rxPreAmp = phased.ReceiverPreamp( ...
        'Gain',spLoss(uIdx), ...    % account for path loss
        'NoiseFigure',prm.NFig,'ReferenceTemperature',290, ...
        'SampleRate',prm.chanSRate);
    rxPreSigAmp = rxPreAmp(rxPreSig{uIdx});
    %   scale power for used sub-carriers
    rxPreSigAmp = rxPreSigAmp * (sqrt(prm.FFTLength - ...
        length(prm.NullCarrierIndices))/prm.FFTLength);

    % OFDM demodulation
    rxOFDM = ofdmdemod(rxPreSigAmp(chanDelay(uIdx)+1: ...
        end-(prm.numPadZeros-chanDelay(uIdx)),:),prm.FFTLength, ...
        prm.CyclicPrefixLength,prm.CyclicPrefixLength, ...
        prm.NullCarrierIndices,prm.PilotCarrierIndices);

    % Channel estimation from preamble
    %       numCarr, numTx, numRx
    hDp{uIdx} = helperMIMOChannelEstimate(rxOFDM(:,1:numTx,:),prm);

end
```

For a multi-user system, the channel estimate is fed back from each MS, and used by the BS to determine the precoding weights. The example assumes perfect feedback with no quantization or implementation delays.

Hybrid Beamforming

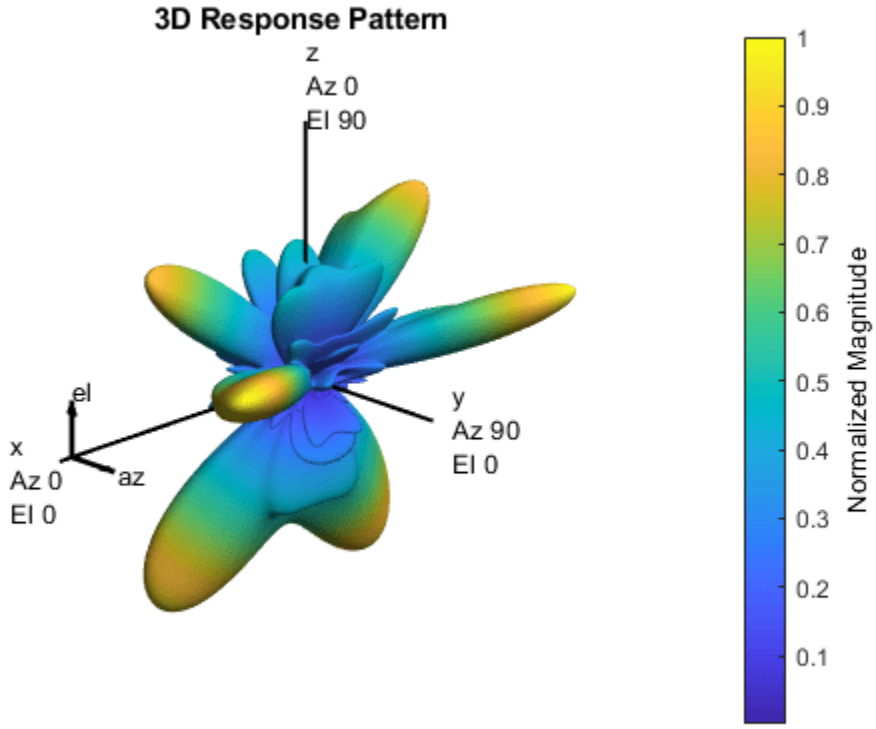
The example uses the orthogonal matching pursuit (OMP) algorithm [3] for a single-user system and the joint spatial division multiplexing (JSDM) technique [2, 4] for a multi-user system, to determine the digital baseband `Fbb` and RF analog `Frf` precoding weights for the selected system configuration.

For a single-user system, the OMP partitioning algorithm is sensitive to the array response vectors `At`. Ideally, these response vectors account for all the scatterers seen by the channel, but these are unknown for an actual system and channel realization, so a random set of rays within a 3-dimensional space to cover as many scatterers as possible is used. The `prm.nRays` parameter specifies the number of rays.

For a multi-user system, JSDM groups users with similar transmit channel covariance together and suppresses the inter-group interference by an analog precoder based on the block diagonalization method [5]. Here each user is assigned to be in its own group, thereby leading to no reduction in the sounding or feedback overhead.

```
% Calculate the hybrid weights on the transmit side
if prm.numUsers==1
    % Single-user OMP
    %   Spread rays in [az;el]=[-180:180;-90:90] 3D space, equal spacing
    %   txang = [-180:360/prm.nRays:180; -90:180/prm.nRays:90];
    txang = [rand(1,prm.nRays)*360-180;rand(1,prm.nRays)*180-90]; % random
    At = steervec(prm.posTxElem,txang);
    AtExp = complex(zeros(prm.numCarriers,size(At,1),size(At,2)));
    for carrIdx = 1:prm.numCarriers
        AtExp(carrIdx,:,:) = At; % same for all sub-carriers
    end

    % Orthogonal matching pursuit hybrid weights
    [Fbb,Frf] = omphybweights(hDp{1},numSTS,numSTS,AtExp);

    v = Fbb;    % set the baseband precoder (Fbb)
    % Frf is same across subcarriers for flat channels
    mFrf = permute(mean(Frf,1),[2 3 1]);

else
    % Multi-user Joint Spatial Division Multiplexing
    [Fbb,mFrf] = helperJSDMTransmitWeights(hDp,prm);

    % Multi-user baseband precoding
    %   Pack the per user CSI into a matrix (block diagonal)
    steeringMatrix = zeros(prm.numCarriers,sum(numSTSVec),sum(numSTSVec));
    for uIdx = 1:prm.numUsers
        stsIdx = sum(numSTSVec(1:uIdx-1))+(1:numSTSVec(uIdx));
        steeringMatrix(:,stsIdx,stsIdx) = Fbb{uIdx};  % Nst-by-Nsts-by-Nsts
    end
    v = permute(steeringMatrix,[1 3 2]);

end

% Transmit array pattern plots
if isTxURA
```

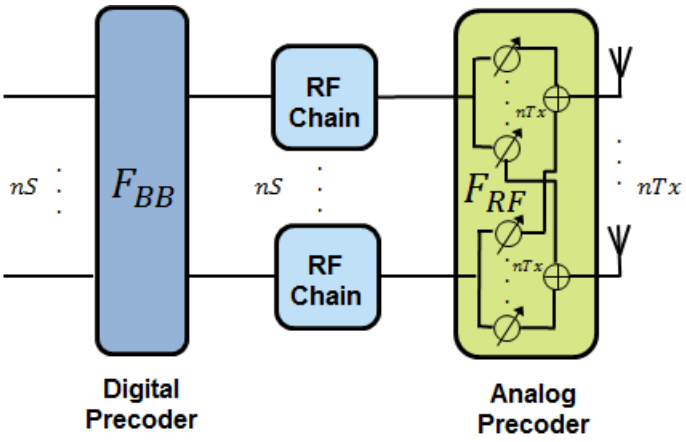
```
    % URA element response for the first subcarrier
    pattern(txarray,prm.fc,-180:180,-90:90,'Type','efield', ...
            'ElementWeights',mFrf.'*squeeze(v(1,:,:)), ...
            'PropagationSpeed',prm.cLight);
else % ULA
    % Array response for first subcarrier
    wts = mFrf.'*squeeze(v(1,:,:));
    pattern(txarray,prm.fc,-180:180,-90:90,'Type','efield', ...
            'Weights',wts(:,1),'PropagationSpeed',prm.cLight);
end
prm.numSTS = numSTS;                % revert back for data transmission
```



For the wideband OFDM system modeled, the analog weights, `mFrf`, are the averaged weights over the multiple subcarriers. The array response pattern shows distinct data streams represented by the stronger lobes. These lobes indicate the spread or separability achieved by beamforming. The “Introduction to Hybrid Beamforming” (Phased Array System Toolbox) example compares the patterns realized by the optimal, fully digital approach, with those realized from the selected hybrid approach, for a single-user system.

Data Transmission

The example models an architecture where each data stream maps to an individual RF chain and each antenna element is connected to each RF chain. This is shown in the following diagram.



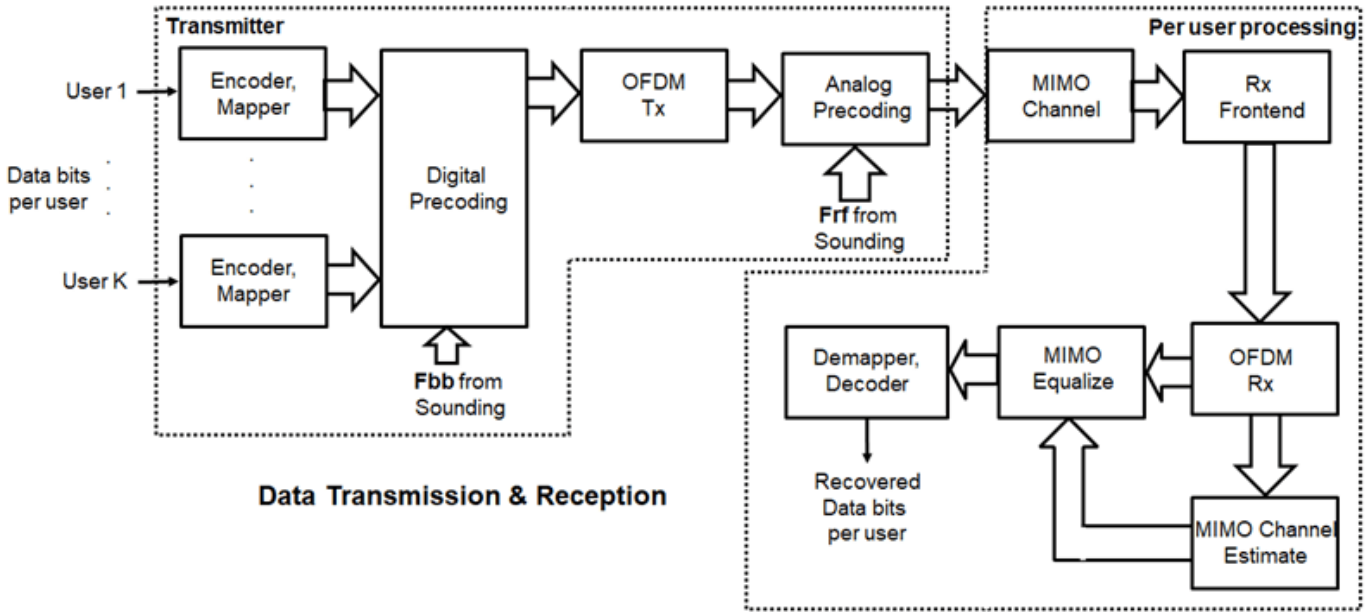
Next, we configure the system's data transmitter. This processing includes channel coding, bit mapping to complex symbols, splitting of the individual data stream to multiple transmit streams, baseband precoding of the transmit streams, OFDM modulation with pilot mapping and RF analog beamforming for all the transmit antennas employed.

```
% Convolutional encoder
encoder = comm.ConvolutionalEncoder( ...
    'TrellisStructure',poly2trellis(7,[133 171 165]), ...
    'TerminationMethod','Terminated');

txDataBits = cell(prm.numUsers, 1);
gridData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for uIdx = 1:prm.numUsers
    % Generate mapped symbols from bits per user
    txDataBits{uIdx} = randi([0,1],prm.numFrmBits(uIdx),1);
    encodedBits = encoder(txDataBits{uIdx});

    % Bits to QAM symbol mapping
    mappedSym = qammod(encodedBits,prm.modMode,'InputType','bit', ...
    'UnitAveragePower',true);

    % Map to layers: per user, per symbol, per data stream
    stsIdx = sum(numSTSVec(1:(uIdx-1)))+(1:numSTSVec(uIdx));
    gridData(:,:,stsIdx) = reshape(mappedSym,prm.numCarriers, ...
        prm.numDataSymbols,numSTSVec(uIdx));
end

% Apply precoding weights to the subcarriers, assuming perfect feedback
preData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for symIdx = 1:prm.numDataSymbols
    for carrIdx = 1:prm.numCarriers
        Q = squeeze(v(carrIdx,:,:));
        normQ = Q * sqrt(numTx)/norm(Q,'fro');
        preData(carrIdx,symIdx,:) = squeeze(gridData(carrIdx,symIdx,:)).' ...
            * normQ;
    end
end

% Multi-antenna pilots
pilots = helperGenPilots(prm.numDataSymbols,numSTS);
```

```
% OFDM modulation of the data
txOFDM = ofdmmod(preData,prm.FFTLength,prm.CyclicPrefixLength,...
                 prm.NullCarrierIndices,prm.PilotCarrierIndices,pilots);
%   scale power for used sub-carriers
txOFDM = txOFDM * (prm.FFTLength/ ...
    sqrt((prm.FFTLength-length(prm.NullCarrierIndices))));

% Generate preamble with the feedback weights and prepend to data
preambleSigD = helperGenPreamble(prm,v);
txSigSTS = [preambleSigD;txOFDM];

% RF beamforming: Apply Frf to the digital signal
%   Each antenna element is connected to each data stream
txSig = txSigSTS*mFrf;
```

For the selected, fully connected RF architecture, each antenna element uses `prm.numSTS` phase shifters, as given by the individual columns of the `mFrf` matrix.

The processing for the data transmission and reception modeled is shown below.



Signal Propagation

The example offers an option for spatial MIMO channel and a simpler static-flat MIMO channel for validation purposes.

The scattering model uses a single-bounce ray tracing approximation with a parametrized number of scatterers. For this example, the number of scatterers is set to 100. The 'Scattering' option models the scatterers placed randomly within a sphere around the receiver, similar to the one-ring model [6].

The channel models allow path-loss modeling and both line-of-sight (LOS) and non-LOS propagation conditions. The example assumes non-LOS propagation and isotropic antenna element patterns with linear or rectangular geometry.

```
% Apply a spatially defined channel to the transmit signal
[rxSig,chanDelay] = helperApplyMUChannel(txSig,prm,spLoss,preambleSig);
```

The same channel is used for both sounding and data transmission. The data transmission has a longer duration and is controlled by the number of data symbols parameter, `prm.numDataSymbols`. The channel evolution between the sounding and transmission stages is modeled by prepending the preamble signal to the data signal. The preamble primes the channel to a valid state for the data transmission, and is ignored from the channel output.

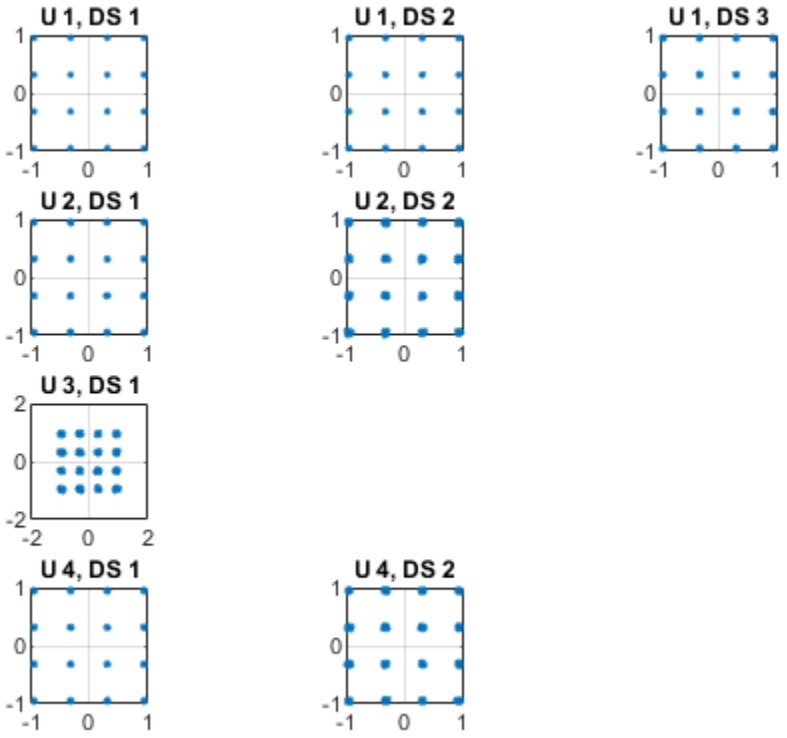
For a multi-user system, independent channels per user are modeled.

Receive Amplification and Signal Recovery

The receiver modeled per user compensates for the path loss by amplification and adds thermal noise. Like the transmitter, the receiver used in a MIMO-OFDM system contains many stages including OFDM demodulation, MIMO equalization, QAM demapping, and channel decoding.

```
hfig = figure('Name','Equalized symbol constellation per stream');
scFact = ((prm.FFTLength-length(prm.NullCarrierIndices))...
         /prm.FFTLength^2)/numTx;
nVar = noisepow(prm.chanSRate,prm.NFig,290)/scFact;
decoder = comm.ViterbiDecoder('InputFormat','Unquantized', ...
    'TrellisStructure',poly2trellis(7, [133 171 165]), ...
    'TerminationMethod','Terminated','OutputDataType','double');

for uIdx = 1:prm.numUsers
    stsU = numSTSVec(uIdx);
    stsIdx = sum(numSTSVec(1:(uIdx-1)))+(1:stsU);

    % Front-end amplifier gain and thermal noise
    rxPreAmp = phased.ReceiverPreamp( ...
        'Gain',spLoss(uIdx), ...        % account for path loss
        'NoiseFigure',prm.NFig,'ReferenceTemperature',290, ...
        'SampleRate',prm.chanSRate);
    rxSigAmp = rxPreAmp(rxSig{uIdx});

    % Scale power for occupied sub-carriers
    rxSigAmp = rxSigAmp*(sqrt(prm.FFTLength-length(prm.NullCarrierIndices)) ...
        /prm.FFTLength);

    % OFDM demodulation
    rxOFDM = ofdmdemod(rxSigAmp(chanDelay(uIdx)+1: ...
        end-(prm.numPadZeros-chanDelay(uIdx)),:),prm.FFTLength, ...
        prm.CyclicPrefixLength,prm.CyclicPrefixLength, ...
        prm.NullCarrierIndices,prm.PilotCarrierIndices);

    % Channel estimation from the mapped preamble
    hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

    % MIMO equalization
    %   Index into streams for the user of interest
    [rxEq,CSI] = helperMIMOEqualize(rxOFDM(:,numSTS+1:end,:),hD(:,stsIdx,:));

    % Soft demodulation
    rxSymbs = rxEq(:)/sqrt(numTx);
    rxLLRBits = qamdemod(rxSymbs,prm.modMode,'UnitAveragePower',true, ...
        'OutputType','approxllr','NoiseVariance',nVar);
```

```
    % Apply CSI prior to decoding
    rxLLRtmp = reshape(rxLLRBits,prm.bitsPerSubCarrier,[], ...
        prm.numDataSymbols,stsU);
    csitmp = reshape(CSI,1,[],1,numSTSVec(uIdx));
    rxScaledLLR = rxLLRtmp.*csitmp;

    % Soft-input channel decoding
    rxDecoded = decoder(rxScaledLLR(:));

    % Decoded received bits
    rxBits = rxDecoded(1:prm.numFrmBits(uIdx));

    % Plot equalized symbols for all streams per user
    scaler = ceil(max(abs([real(rxSymbs(:)); imag(rxSymbs(:))])));
    for i = 1:stsU
        subplot(prm.numUsers, max(numSTSVec), (uIdx-1)*max(numSTSVec)+i);
        plot(reshape(rxEq(:,:,i)/sqrt(numTx), [], 1), '.');
        axis square
        xlim(gca,[-scaler scaler]);
        ylim(gca,[-scaler scaler]);
        title(['U ' num2str(uIdx) ', DS ' num2str(i)]);
        grid on;
    end

    % Compute and display the EVM
    evm = comm.EVM('Normalization','Average constellation power', ...
        'ReferenceSignalSource','Estimated from reference constellation', ...
        'ReferenceConstellation', ...
        qammod((0:prm.modMode-1)',prm.modMode,'UnitAveragePower',1));
    rmsEVM = evm(rxSymbs);
    disp(['User ' num2str(uIdx)]);
    disp(['  RMS EVM (%) = ' num2str(rmsEVM)]);

    % Compute and display bit error rate
    ber = comm.ErrorRate;
    measures = ber(txDataBits{uIdx},rxBits);
    fprintf('  BER = %.5f; No. of Bits = %d; No. of errors = %d\n', ...
        measures(1),measures(3),measures(2));
end

User 1
  RMS EVM (%) = 0.38361
  BER = 0.00000; No. of Bits = 9354; No. of errors = 0
User 2
  RMS EVM (%) = 1.0311
  BER = 0.00000; No. of Bits = 6234; No. of errors = 0
User 3
  RMS EVM (%) = 2.1462
  BER = 0.00000; No. of Bits = 3114; No. of errors = 0
User 4
  RMS EVM (%) = 1.0024
  BER = 0.00000; No. of Bits = 6234; No. of errors = 0
```



For the MIMO system modeled, the displayed receive constellation of the equalized symbols offers a qualitative assessment of the reception. The actual bit error rate offers the quantitative figure by comparing the actual transmitted bits with the received decoded bits per user.

```
rng(s);          % restore RNG state
```

Conclusion and Further Exploration

The example highlights the use of hybrid beamforming for multi-user MIMO-OFDM systems. It allows you to explore different system configurations for a variety of channel models by changing a few system-wide parameters.

The set of configurable parameters includes the number of users, number of data streams per user, number of transmit/receive antenna elements, array locations, and channel models. Adjusting these parameters you can study the parameters' individual or combined effects on the overall system. As examples, vary:

- the number of users, `prm.numUsers`, and their corresponding data streams, `prm.numSTSVec`, to switch between multi-user and single-user systems, or
- the channel type, `prm.ChanType`, or
- the number of rays, `prm.nRays`, used for a single-user system.

Explore the following helper functions used by the example:

- helperApplyMUChannel.m

- helperArrayInfo.m
- helperGenPreamble.m
- helperGenPilots.m
- helperJSDMTransmitWeights.m
- helperMIMOChannelEstimate.m
- helperMIMOEqualize.m

References

1. Molisch, A. F., et al. "Hybrid Beamforming for Massive MIMO: A Survey." IEEE® Communications Magazine, Vol. 55, No. 9, September 2017, pp. 134-141.
2. Li Z., S. Han, and A. F. Molisch. "Hybrid Beamforming Design for Millimeter-Wave Multi-User Massive MIMO Downlink." IEEE ICC 2016, Signal Processing for Communications Symposium.
3. El Ayach, Oma, et al. "Spatially Sparse Precoding in Millimeter Wave MIMO Systems." IEEE Transactions on Wireless Communications, Vol. 13, No. 3, March 2014, pp. 1499-1513.
4. Adhikary A., J. Nam, J-Y Ahn, and G. Caire. "Joint Spatial Division and Multiplexing - The Large-Scale Array Regime." IEEE Transactions on Information Theory, Vol. 59, No. 10, October 2013, pp. 6441-6463.
5. Spencer Q., A. Swindlehurst, M. Haardt, "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.
6. Shui, D. S., G. J. Foschini, M. J. Gans and J. M. Kahn. "Fading Correlation and its Effect on the Capacity of Multielement Antenna Systems." IEEE Transactions on Communications, Vol. 48, No. 3, March 2000, pp. 502-513.

MIMO-OFDM Precoding with Phased Arrays

This example shows how phased arrays are used in a MIMO-OFDM communication system employing beamforming. Using components from Communications Toolbox™ and Phased Array System Toolbox™, it models the radiating elements that comprise a transmitter and the front-end receiver components, for a MIMO-OFDM communication system. With user-specified parameters, you can validate the performance of the system in terms of bit error rate and constellations for different spatial locations and array sizes.

The example uses functions and System objects™ from Communications Toolbox and Phased Array System Toolbox and requires

- WINNER II Channel Model for Communications Toolbox

Introduction

MIMO-OFDM systems are the norm in current wireless systems (e.g. 5G NR, LTE, WLAN) due to their robustness to frequency-selective channels and high data rates enabled. With ever-increasing demands on data rates supported, these systems are getting more complex and larger in configurations with increasing number of antenna elements, and resources (subcarriers) allocated.

With antenna arrays and spatial multiplexing, efficient techniques to realize the transmissions are necessary [6]. Beamforming is one such technique, that is employed to improve the signal to noise ratio (SNR) which ultimately improves the system performance, as measured here in terms of bit error rate (BER) [1].

This example illustrates an asymmetric MIMO-OFDM single-user system where the maximum number of antenna elements on transmit and receive ends can be 1024 and 32 respectively, with up to 16 independent data streams. It models a spatial channel where the array locations and antenna patterns are incorporated into the overall system design. For simplicity, a single point-to-point link (one base station communicating with one mobile user) is modeled. The link uses channel sounding to provide the transmitter with the channel information it needs for beamforming.

The example offers the choice of a few spatially defined channel models, specifically a WINNER II Channel model and a scattering-based model, both of which account for the transmit/receive spatial locations and antenna patterns.

```
s = rng(61);        % Set RNG state for repeatability
```

System Parameters

Define parameters for the system. These parameters can be modified to explore their impact on the system.

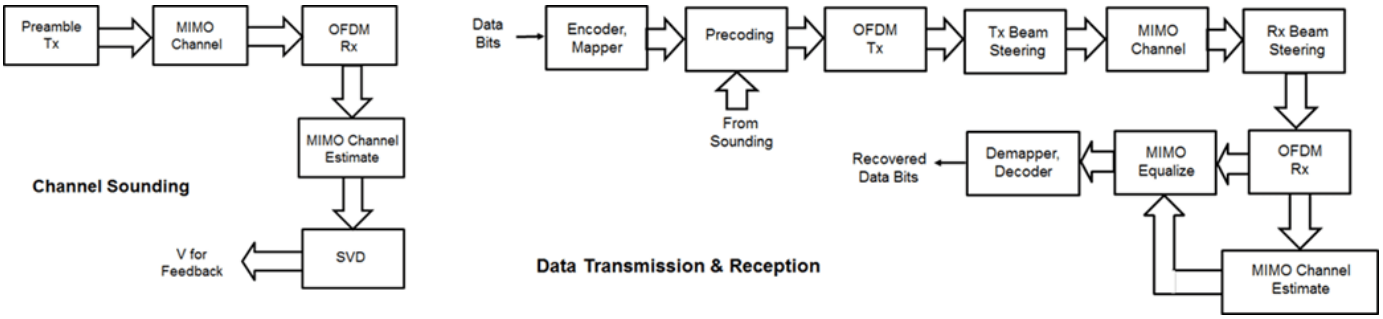
```
% Single-user system with multiple streams
prm.numUsers = 1;            % Number of users
prm.numSTS = 16;             % Number of independent data streams, 4/8/16/32/64
prm.numTx = 32;              % Number of transmit antennas
prm.numRx = 16;              % Number of receive antennas
prm.bitsPerSubCarrier = 6;   % 2: QPSK, 4: 16QAM, 6: 64QAM, 8: 256QAM
prm.numDataSymbols = 10;     % Number of OFDM data symbols

prm.fc = 4e9;                   % 4 GHz system
prm.chanSRate = 100e6;          % Channel sampling rate, 100 Msps
prm.ChanType = 'Scattering';    % Channel options: 'WINNER', 'Scattering',
```

```
                                %           'ScatteringFcn', 'StaticFlat'
prm.NFig = 5;                   % Noise figure, dB

% Array locations and angles
prm.posTx = [0;0;0];            % BS/Transmit array position, [x;y;z], meters
prm.mobileRange = 300;          % meters
% Angles specified as [azimuth;elevation], az=[-90 90], el=[-90 90]
prm.mobileAngle = [33; 0];      % degrees
prm.steeringAngle = [30; -20];  % Transmit steering angle (close to mobileAngle)
prm.enSteering = true;          % Enable/disable steering
```

Parameters to define the OFDM modulation employed for the system are specified below.

```
prm.FFTLength = 256;
prm.CyclicPrefixLength = 64;
prm.numCarriers = 234;
prm.NumGuardBandCarriers = [7 6];
prm.PilotCarrierIndices = [26 54 90 118 140 168 204 232];
nonDataIdx = [(1:prm.NumGuardBandCarriers(1))'; prm.FFTLength/2+1; ...
              (prm.FFTLength-prm.NumGuardBandCarriers(2)+1:prm.FFTLength)'; ...
              prm.PilotCarrierIndices.';];
prm.CarriersLocations = setdiff((1:prm.FFTLength)',sort(nonDataIdx));

numTx = prm.numTx;
numRx = prm.numRx;
numSTS = prm.numSTS;
prm.numFrmBits = numSTS*prm.numDataSymbols*prm.numCarriers* ...
                 prm.bitsPerSubCarrier*1/3-6; % Account for termination bits

prm.modMode = 2^prm.bitsPerSubCarrier; % Modulation order
% Account for channel filter delay
prm.numPadZeros = 3*(prm.FFTLength+prm.CyclicPrefixLength);

% Get transmit and receive array information
prm.numSTSVec = numSTS;
[isTxURA,expFactorTx,isRxURA,expFactorRx] = helperArrayInfo(prm,true);
```

The processing for channel sounding, data transmission and reception modeled in the example are shown in the following block diagrams.



The free space path loss is calculated based on the base station and mobile station positions for the spatially-aware system modeled.

```
prm.cLight = physconst('LightSpeed');
prm.lambda = prm.cLight/prm.fc;
% Mobile position
```

```
[xRx,yRx,zRx] = sph2cart(deg2rad(prm.mobileAngle(1)),...
                         deg2rad(prm.mobileAngle(2)),prm.mobileRange);
prm.posRx = [xRx;yRx;zRx];
[toRxRange,toRxAng] = rangeangle(prm.posTx,prm.posRx);
spLoss = fspl(toRxRange,prm.lambda);
gainFactor = 1;
```

Channel Sounding

For a spatially multiplexed system, availability of channel information at the transmitter allows for precoding to be applied to maximize the signal energy in the direction and channel of interest. Under the assumption of a slowly varying channel, this is facilitated by sounding the channel first, wherein for a reference transmission, the receiver estimates the channel and feeds this information back to the transmitter.

For the chosen system, a preamble signal is sent over all transmitting antenna elements, and processed at the receiver accounting for the channel. The receiver components perform pre-amplification, OFDM demodulation, frequency domain channel estimation, and calculation of the feedback weights based on channel diagonalization using singular value decomposition (SVD) per data subcarrier.

```
% Generate the preamble signal
preambleSigSTS = helperGenPreamble(prm);
%   repeat over numTx
preambleSig = zeros(size(preambleSigSTS,1),numTx);
for i = 1:numSTS
    preambleSig(:,(i-1)*expFactorTx+(1:expFactorTx)) = ...
        repmat(preambleSigSTS(:,i),1,expFactorTx);
end

% Transmit preamble over channel
[rxPreSig,chanDelay] = helperApplyChannel(preambleSig,prm,spLoss);

% Front-end amplifier gain and thermal noise
rxPreAmp = phased.ReceiverPreamp( ...
    'Gain',gainFactor*spLoss, ... % account for path loss
    'NoiseFigure',prm.NFig, ...
    'ReferenceTemperature',290, ...
    'SampleRate',prm.chanSRate);
rxPreSigAmp = rxPreAmp(rxPreSig);
rxPreSigAmp = rxPreSigAmp * ...          % scale power
    (sqrt(prm.FFTLength-sum(prm.NumGuardBandCarriers)-1)/(prm.FFTLength));

% OFDM Demodulation
demodulatorOFDM = comm.OFDMDemodulator( ...
     'FFTLength',prm.FFTLength, ...
     'NumGuardBandCarriers',prm.NumGuardBandCarriers.', ...
     'RemoveDCCarrier',true, ...
     'PilotOutputPort',true, ...
     'PilotCarrierIndices',prm.PilotCarrierIndices.', ...
     'CyclicPrefixLength',prm.CyclicPrefixLength, ...
     'NumSymbols',numSTS, ... % preamble symbols alone
     'NumReceiveAntennas',numRx);

rxOFDM = demodulatorOFDM( ...
    rxPreSigAmp(chanDelay+1:end-(prm.numPadZeros-chanDelay),:));
```

```
% Channel estimation from preamble
%       numCarr, numSTS, numRx
hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

% Calculate the feedback weights
v = diagbfweights(hD);
```

For conciseness in presentation, front-end synchronization including carrier and timing recovery are assumed. The weights computed using `diagbfweights` are hence fed back to the transmitter, for subsequent application for the actual data transmission.

Data Transmission

Next, we configure the system's data transmitter. This processing includes channel coding, bit mapping to complex symbols, splitting of the individual data stream to multiple transmit streams, precoding of the transmit streams, OFDM modulation with pilot mapping and replication for the transmit antennas employed.

```
% Convolutional encoder
encoder = comm.ConvolutionalEncoder( ...
    'TrellisStructure',poly2trellis(7,[133 171 165]), ...
    'TerminationMethod','Terminated');

% Generate mapped symbols from bits
txBits = randi([0, 1],prm.numFrmBits,1);
encodedBits = encoder(txBits);

% Bits to QAM symbol mapping
mappedSym = qammod(encodedBits,prm.modMode,'InputType','Bit', ...
    'UnitAveragePower',true);

% Map to layers: per symbol, per data stream
gridData = reshape(mappedSym,prm.numCarriers,prm.numDataSymbols,numSTS);

% Apply precoding weights to the subcarriers, assuming perfect feedback
preData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for symIdx = 1:prm.numDataSymbols
    for carrIdx = 1:prm.numCarriers
        Q = squeeze(v(carrIdx,:,:));
        normQ = Q * sqrt(numTx)/norm(Q,'fro');
        preData(carrIdx,symIdx,:) = ...
            squeeze(gridData(carrIdx,symIdx,:)).' * normQ;
    end
end

% OFDM modulation of the data
modulatorOFDM = comm.OFDMModulator( ...
    'FFTLength',prm.FFTLength,...
    'NumGuardBandCarriers',prm.NumGuardBandCarriers.',...
    'InsertDCNull',true, ...
    'PilotInputPort',true,...
    'PilotCarrierIndices',prm.PilotCarrierIndices.',...
    'CyclicPrefixLength',prm.CyclicPrefixLength,...
    'NumSymbols',prm.numDataSymbols,...
    'NumTransmitAntennas',numSTS);

% Multi-antenna pilots
pilots = helperGenPilots(prm.numDataSymbols,numSTS);
```

```
txOFDM = modulatorOFDM(preData,pilots);
txOFDM = txOFDM * (prm.FFTLength/ ...
    sqrt(prm.FFTLength-sum(prm.NumGuardBandCarriers)-1)); % scale power

% Generate preamble with the feedback weights and prepend to data
preambleSigD = helperGenPreamble(prm,v);
txSigSTS = [preambleSigD;txOFDM];

% Repeat over numTx
txSig = zeros(size(txSigSTS,1),numTx);
for i = 1:numSTS
    txSig(:,(i-1)*expFactorTx+(1:expFactorTx)) = ...
        repmat(txSigSTS(:,i),1,expFactorTx);
end
```

For precoding, the preamble signal is regenerated to enable channel estimation. It is prepended to the data portion to form the transmission packet which is then replicated over the transmit antennas.

Transmit Beam Steering

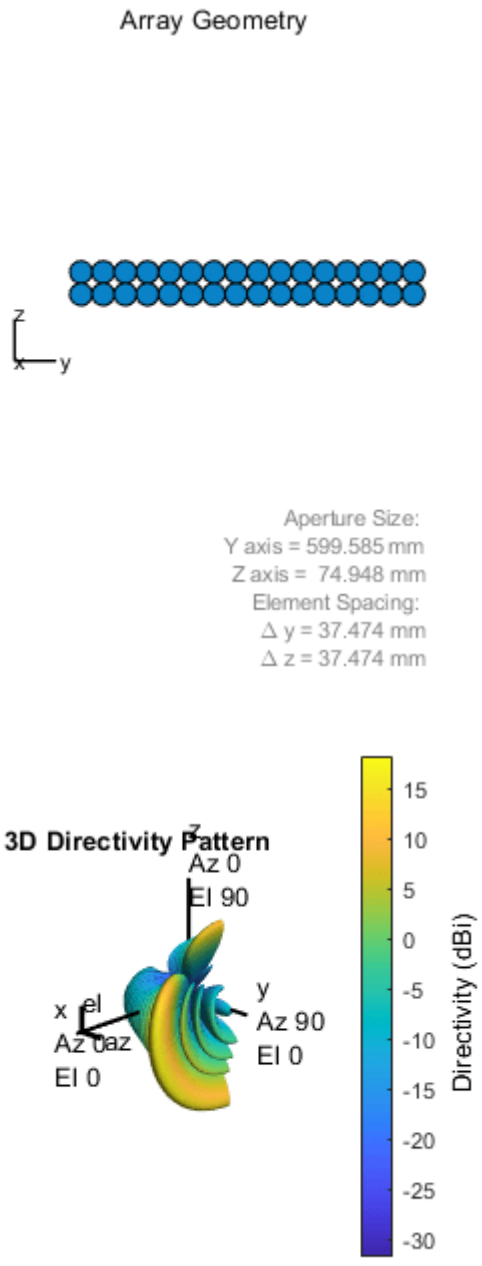
Phased Array System Toolbox offers components appropriate for the design and simulation of phased arrays used in wireless communications systems.

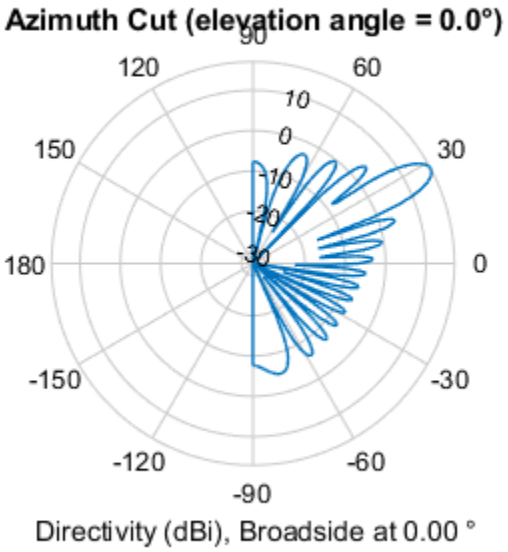
For the spatially aware system, the signal transmitted from the base station is steered towards the direction of the mobile, so as to focus the radiated energy in the desired direction. This is achieved by applying a phase shift to each antenna element to steer the transmission.

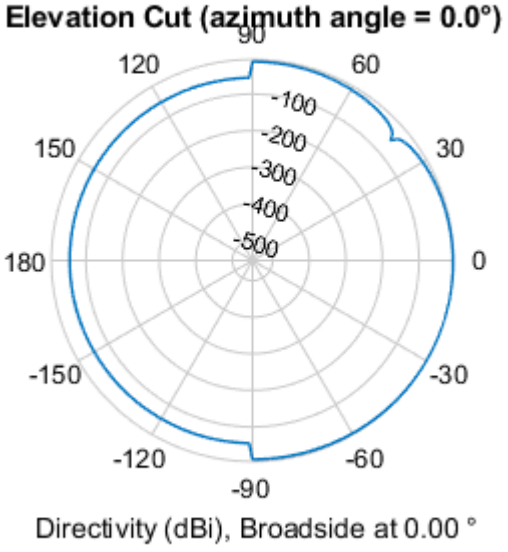
The example uses a linear or rectangular array at the transmitter, depending on the number of data streams and number of transmit antennas selected.

```
% Gain per antenna element
amplifier = phased.Transmitter('PeakPower',1/numTx,'Gain',0);

% Amplify to achieve peak transmit power for each element
for n = 1:numTx
    txSig(:,n) = amplifier(txSig(:,n));
end

% Transmit antenna array definition
if isTxURA
    % Uniform Rectangular array
    arrayTx = phased.URA([expFactorTx,numSTS],[0.5 0.5]*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform Linear array
    arrayTx = phased.ULA(numTx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
end

% For evaluating weights for steering
SteerVecTx = phased.SteeringVector('SensorArray',arrayTx, ...
    'PropagationSpeed',prm.cLight);

% Generate weights for steered direction
wT = SteerVecTx(prm.fc,prm.steeringAngle);
```

```
% Radiate along the steered direction, without signal combining
radiatorTx = phased.Radiator('Sensor',arrayTx, ...
    'WeightsInputPort',true, ...
    'PropagationSpeed',prm.cLight, ...
    'OperatingFrequency',prm.fc, ...
    'CombineRadiatedSignals',false);

if prm.enSteering
    txSteerSig = radiatorTx(txSig,repmat(prm.mobileAngle,1,numTx), ...
        conj(wT));
else
    txSteerSig = txSig;
end

% Visualize the array
h = figure('Position',figposition([10 55 22 35]),'MenuBar','none');
h.Name = 'Transmit Array Geometry';
viewArray(arrayTx);

% Visualize the transmit pattern and steering
h = figure('Position',figposition([32 55 22 30]),'MenuBar','none');
h.Name = 'Transmit Array Response Pattern';
pattern(arrayTx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wT);
h = figure('Position',figposition([54 55 22 35]),'MenuBar','none');
h.Name = 'Transmit Array Azimuth Pattern';
patternAzimuth(arrayTx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wT);
if isTxURA
    h = figure('Position',figposition([76 55 22 35]),'MenuBar','none');
    h.Name = 'Transmit Array Elevation Pattern';
    patternElevation(arrayTx,prm.fc,'PropagationSpeed',prm.cLight, ...
        'Weights',wT);
end
```


Array Geometry
z
x
y
Aperture Size:
Y axis = 599.585 mm
Z axis = 74.948 mm
Element Spacing:
Δ y = 37.474 mm
Δ z = 37.474 mm
3D Directivity Pattern
z
Az 0
El 90
x
el
Az 0
az
El 0
y
Az 90
El 0
15
10
5
0
-5
-10
-15
-20
-25
-30
Directivity (dBi)





The plots indicate the array geometry and the transmit array response in multiple views. The response shows the transmission direction as specified by the steering angle.

The example assumes the steering angle known and close to the mobile angle. In actual systems, this would be estimated from angle-of-arrival estimation at the receiver as a part of the channel sounding or initial beam tracking procedures.

Signal Propagation

The example offers three options for spatial MIMO channels and a simpler static-flat MIMO channel for evaluation purposes.

The WINNER II channel model [5] is a spatially defined MIMO channel that allows you to specify the array geometry and location information. It is configured to use the typical urban microcell indoor scenario with very low mobile speeds.

The two scattering based channels use a single-bounce path through each scatterer where the number of scatterers is user-specified. For this example, the number of scatterers is set to 100. The 'Scattering' option models the scatterers placed randomly within a circle in between the transmitter and receiver, while the 'ScatteringFcn' models their placement completely randomly.

The models allow path loss modeling and both line-of-sight (LOS) and non-LOS propagation conditions. The example assumes non-LOS propagation and isotropic antenna element patterns with linear geometry.

```
% Apply a spatially defined channel to the steered signal
[rxSig,chanDelay] = helperApplyChannel(txSteerSig,prm,spLoss,preambleSig);
```

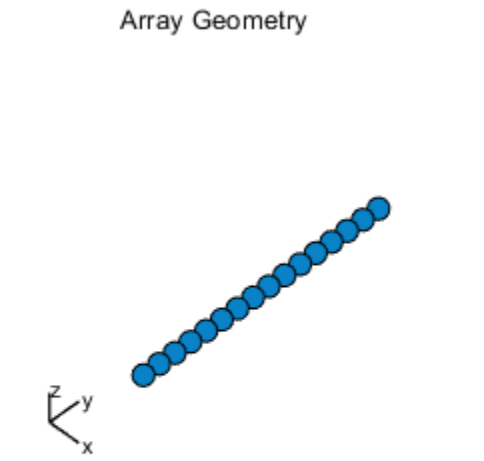
The same channel is used for both sounding and data transmission, with the data transmission having a longer duration controlled by the number of data symbols parameter, `prm.numDataSymbols`.

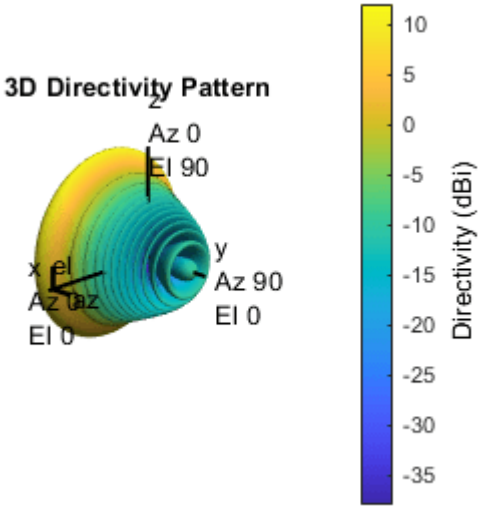
Receive Beam Steering

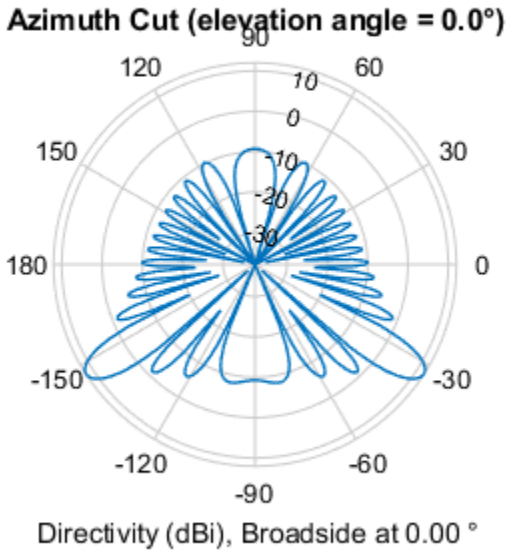
The receiver steers the incident signals to align with the transmit end steering, per receive element. Thermal noise and receiver gain are applied. Uniform linear or rectangular arrays with isotropic responses are modeled to match the channel and transmitter arrays.

```
rxPreAmp = phased.ReceiverPreamp( ...
    'Gain',gainFactor*spLoss, ... % accounts for path loss
    'NoiseFigure',prm.NFig, ...
    'ReferenceTemperature',290, ...
    'SampleRate',prm.chanSRate);

% Front-end amplifier gain and thermal noise
rxSigAmp = rxPreAmp(rxSig);
rxSigAmp = rxSigAmp * ...           % scale power
    (sqrt(prm.FFTLength - sum(prm.NumGuardBandCarriers)-1)/(prm.FFTLength));

% Receive array
if isRxURA
    % Uniform Rectangular array
    arrayRx = phased.URA([expFactorRx,numSTS],0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform Linear array
    arrayRx = phased.ULA(numRx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement);
end
```

```
% For evaluating receive-side steering weights
SteerVecRx = phased.SteeringVector('SensorArray',arrayRx, ...
    'PropagationSpeed',prm.cLight);

% Generate weights for steered direction towards mobile
wR = SteerVecRx(prm.fc,toRxAng);

% Steer along the mobile receive direction
if prm.enSteering
    rxSteerSig = rxSigAmp.*(wR');
else
    rxSteerSig = rxSigAmp;
end

% Visualize the array
h = figure('Position',figposition([10 20 22 35]),'MenuBar','none');
h.Name = 'Receive Array Geometry';
viewArray(arrayRx);

% Visualize the receive pattern and steering
h = figure('Position',figposition([32 20 22 30]));
h.Name = 'Receive Array Response Pattern';
pattern(arrayRx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wR);
h = figure('Position',figposition([54 20 22 35]),'MenuBar','none');
h.Name = 'Receive Array Azimuth Pattern';
patternAzimuth(arrayRx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wR);
if isRxURA
    figure('Position',figposition([76 20 22 35]),'MenuBar','none');
    h.Name = 'Receive Array Elevation Pattern';
    patternElevation(arrayRx,prm.fc,'PropagationSpeed',prm.cLight, ...
        'Weights',wR);
end
```


Array Geometry
z
y
x
Aperture Size:
Y axis = 599.585 mm
Element Spacing:
Δ y = 37.474 mm
Array Axis: Y axis


3D Directivity Pattern
z
Az 0
El 90
y
Az 90
El 0
x
el
Az
az
El 0
10
5
0
-5
-10
-15
-20
-25
-30
-35
Directivity (dBi)



The receive antenna pattern mirrors the transmission steering.

Signal Recovery

The receive antenna array passes the propagated signal to the receiver to recover the original information embedded in the signal. Similar to the transmitter, the receiver used in a MIMO-OFDM system contains many components, including OFDM demodulator, MIMO equalizer, QAM demodulator, and channel decoder.

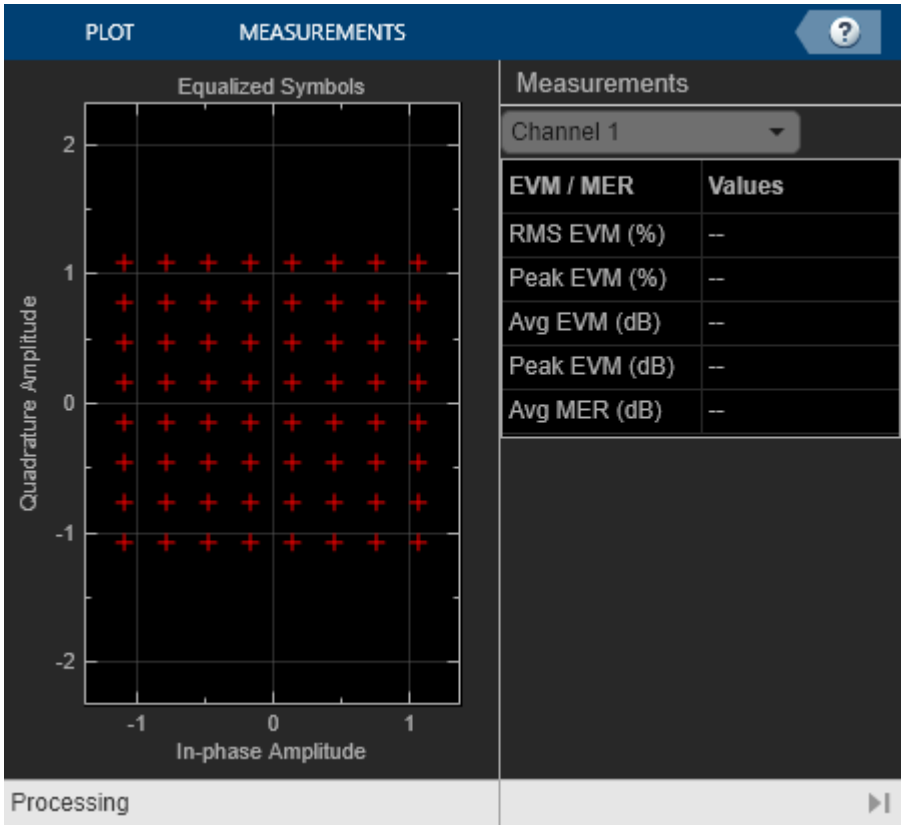
```
demodulatorOFDM = comm.OFDMDemodulator( ...
      'FFTLength',prm.FFTLength, ...
      'NumGuardBandCarriers',prm.NumGuardBandCarriers.', ...
      'RemoveDCCarrier',true, ...
      'PilotOutputPort',true, ...
      'PilotCarrierIndices',prm.PilotCarrierIndices.', ...
      'CyclicPrefixLength',prm.CyclicPrefixLength, ...
      'NumSymbols',numSTS+prm.numDataSymbols, ... % preamble & data
      'NumReceiveAntennas',numRx);

% OFDM Demodulation
rxOFDM = demodulatorOFDM( ...
    rxSteerSig(chanDelay+1:end-(prm.numPadZeros-chanDelay),:));

% Channel estimation from the mapped preamble
hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

% MIMO Equalization
[rxEq,CSI] = helperMIMOEqualize(rxOFDM(:,numSTS+1:end,:),hD);

% Soft demodulation
scFact = ((prm.FFTLength-sum(prm.NumGuardBandCarriers)-1) ...
         /prm.FFTLength^2)/numTx;
```

```
nVar = noisepow(prm.chanSRate,prm.NFig,290)/scFact;
rxSymbs = rxEq(:)/sqrt(numTx);
rxLLRBits = qamdemod(rxSymbs,prm.modMode,'UnitAveragePower',true, ...
    'OutputType','approxllr','NoiseVariance',nVar);

% Apply CSI prior to decoding
rxLLRtmp = reshape(rxLLRBits,prm.bitsPerSubCarrier,[], ...
                   prm.numDataSymbols,numSTS);
csitmp = reshape(CSI,1,[],1,numSTS);
rxScaledLLR = rxLLRtmp.*csitmp;

% Soft-input channel decoding
decoder = comm.ViterbiDecoder(...
     'InputFormat','Unquantized', ...
     'TrellisStructure',poly2trellis(7, [133 171 165]), ...
     'TerminationMethod','Terminated', ...
     'OutputDataType','double');
rxDecoded = decoder(rxScaledLLR(:));

% Decoded received bits
rxBits = rxDecoded(1:prm.numFrmBits);
```

For the MIMO system modeled, the displayed receive constellation of the equalized symbols offers a qualitative assessment of the reception. The actual bit error rate offers the quantitative figure by comparing the actual transmitted bits with the received decoded bits.

```
% Display received constellation
constDiag = comm.ConstellationDiagram( ...
    'SamplesPerSymbol',1, ...
    'ShowReferenceConstellation',true, ...
    'ReferenceConstellation', ...
    qammod((0:prm.modMode-1)',prm.modMode,'UnitAveragePower',true), ...
    'ColorFading',false, ...
    'Position',figposition([20 20 35 40]), ...
    'Title','Equalized Symbols', ...
    'EnableMeasurements',true, ...
    'MeasurementInterval',length(rxSymbs));
constDiag(rxSymbs);

% Compute and display bit error rate
ber = comm.ErrorRate;
measures = ber(txBits,rxBits);
fprintf('BER = %.5f; No. of Bits = %d; No. of errors = %d\n', ...
    measures(1),measures(3),measures(2));

rng(s); % Restore RNG state
```

```
BER = 0.00000; No. of Bits = 74874; No. of errors = 0
```



Conclusion and Further Exploration

The example highlighted the use of phased antenna arrays for a beamformed MIMO-OFDM system. It accounted for the spatial geometry and location of the arrays at the base station and mobile station for a single user system. Using channel sounding, it illustrated how precoding is realized in current wireless systems and how steering of antenna arrays is modeled.

Within the set of configurable parameters, you can vary the number of data streams, transmit/receive antenna elements, station or array locations and geometry, channel models and their configurations to study the parameters' individual or combined effects on the system. E.g. vary just the number of transmit antennas to see the effect on the main lobe of the steered beam and the resulting system performance.

The example also made simplifying assumptions for front-end synchronization, channel feedback, user velocity and path loss models, which need to be further considered for a practical system. Individual systems also have their own procedures which must be folded in to the modeling [2, 3, 4].

Explore the following helper functions used:

- helperApplyChannel.m
- helperArrayInfo.m
- helperGenPilots.m
- helperGenPreamble.m
- helperGetP.m

- helperMIMOChannelEstimate.m
- helperMIMOEqualize.m

Selected Bibliography

1. Perahia, Eldad, and Robert Stacey. Next Generation Wireless LANS: 802.11n and 802.11ac. Cambridge University Press, 2013.
2. IEEE® Std 802.11™-2012 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.
3. 3GPP TS 36.213. "Physical layer procedures." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA). URL: https://www.3gpp.org.
4. 3GPP TS 36.101. "User Equipment (UE) Radio Transmission and Reception." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA). URL: https://www.3gpp.org.
5. Kyosti, Pekka, Juha Meinila, et al. WINNER II Channel Models. D1.1.2, V1.2. IST-4-027756 WINNER II, September 2007.
6. George Tsoulos, Ed., "MIMO System Technology for Wireless Communications", CRC Press, Boca Raton, FL, 2006.

HDL Coder Featured Examples

Airplane Tracking with ADS-B Captured Data

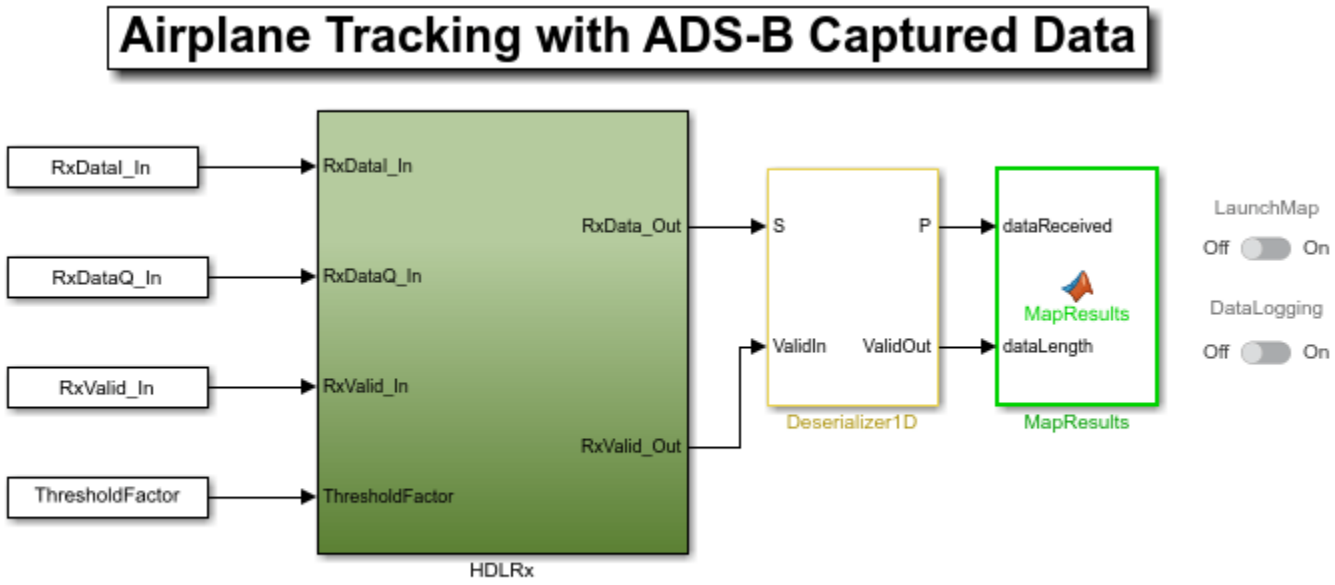
This example shows how to implement the Automatic Dependent Surveillance - Broadcast (ADS-B) receiver for HDL code generation and hardware implementation. This example decodes ADS-B extended squitter messages which can be used to track the airplane. The HDL-optimized model in this example uses Simulink® blocks that support HDL code generation to implement the ADS-B Receiver. This example model is used for real-time processing in “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio), which requires the Communications Toolbox™ Support Package for Xilinx® Zynq®-Based Radio.

Introduction

ADS-B is an air traffic management and control surveillance system. The broadcast messages (approximately once per second) contain the flight information including position and velocity. For introduction on ADS-B technology and modes of transmission, see [1]. The **HDLRx** subsystem is optimized for HDL code generation. The captured received signal is streamed into the receiver (**HDLRx** subsystem) front end. The streaming output of the receiver is buffered and passed to the **MapResults** MATLAB® function to view the output.

Structure of the Example

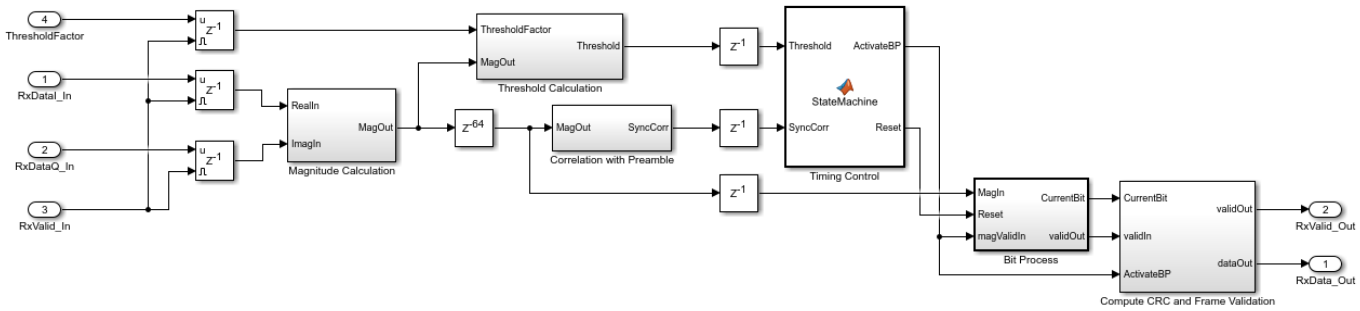
The model supports both Normal and Accelerator modes. The top-level structure of the ADS-B receiver model is shown in the following figure.



The receiver input data is captured using “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) running on the Zynq® platform. The captured data represents the baseband received signal with a sampling rate of 4 MHz. The data contains 8 frames of extended squitter messages. The ADS-B transmitter modulates the 112-bit extended squitter messages using 2-bit pulse-position

modulation, and adds a 16-bit prefix. Then, to generate 4 MHz data, each 240-bit message is zero-padded and upsampled by 2.

This diagram shows the detailed structure of the **HDLRx** subsystem.



The subsystems listed here are described further in the following sections.

1. **Magnitude Calculation** - Finds the complex modulus of the received input signal

2. **Threshold Calculation** - Calculates the threshold value based on received input signal strength

3. **Correlation with Preamble** - Correlates the received signal with reference signal to detect the preamble

4. **Timing Control** - Provides timing synchronization for the receiver

5. **Bit Process** - Decodes symbols using PPM demodulation

6. **Compute CRC and Frame Validation** - Validates the frame by checking for CRC errors

HDL Optimized ADS-B Receiver

1. **Magnitude Calculation**

The inputs to the **Magnitude Calculation** subsystem are the in-phase (real) and quadrature (imaginary) phase samples. This subsystem outputs the modulus of the complex number. The sqrt(I^2+Q^2) can be approximated by the "|L|+0.4*|S| algorithm" described on page 238 of [2].

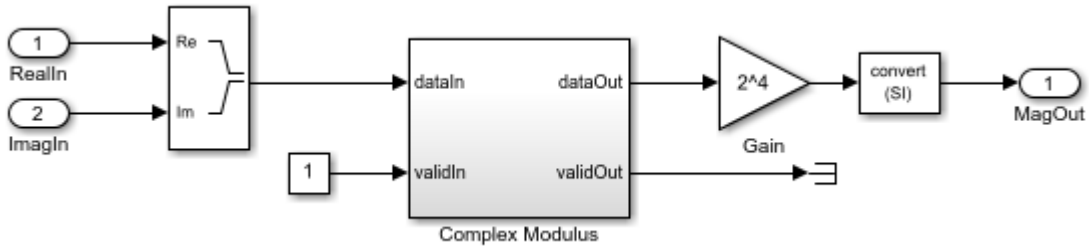
$$\sqrt{I^2 + Q^2} = |L| + 0.4 * |S|$$

where

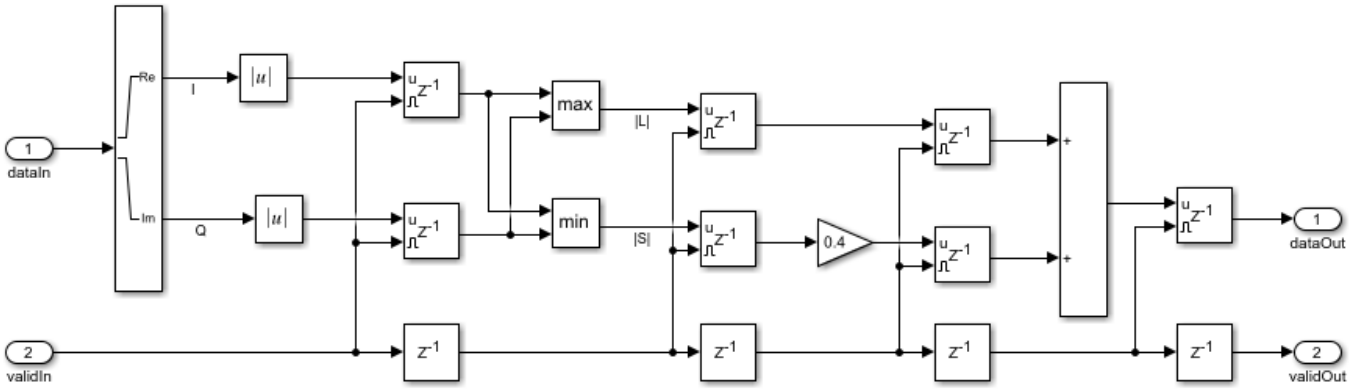
| L | is the larger value of | I | or | Q |

| S | is the smaller value of | I | or | Q |.

The Gain block converts received input from 12-bit to 16-bit word length.

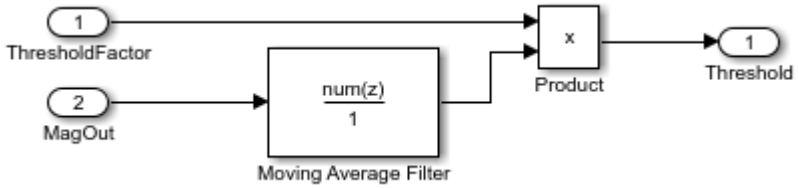


For the implementation of "|L|+0.4|S| algorithm", see the following model.



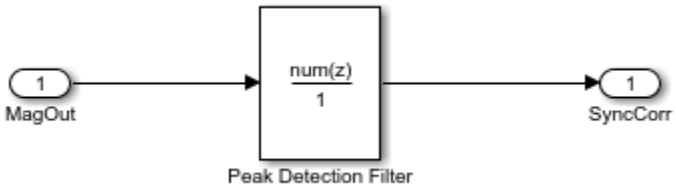
2. **Threshold Calculation**

The **Threshold Calculation** subsystem calculates the signal energy and applies a scaling factor to create a threshold for preamble detection. Moving Average Filter is a serial FIR filter architecture with 32 coefficients that operates on the magnitude values. The coefficients of the FIR filter are selected to find the average energy of the received signal. This example scales the signal energy by 5 to detect valid ADS-B preambles. For details on FIR filter, see Discrete FIR Filter (Simulink).



3. **Correlation with Preamble**

The **Correlation with Preamble** subsystem correlates the received signal with the ADS-B reference/preamble sequence [1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0] using a peak detection filter. The peak detection filter is a serial FIR Filter architecture, configured with coefficients that match the preamble sequence. Preamble correlation identifies potential ADS-B transmissions and aligns our bit detection algorithm with the first message bit. The preamble is detected if the peak amplitude exceeds the scaled threshold value. Once the preamble is detected, the correlation value is passed on as input(SyncCorr) to the **Timing Control** block.

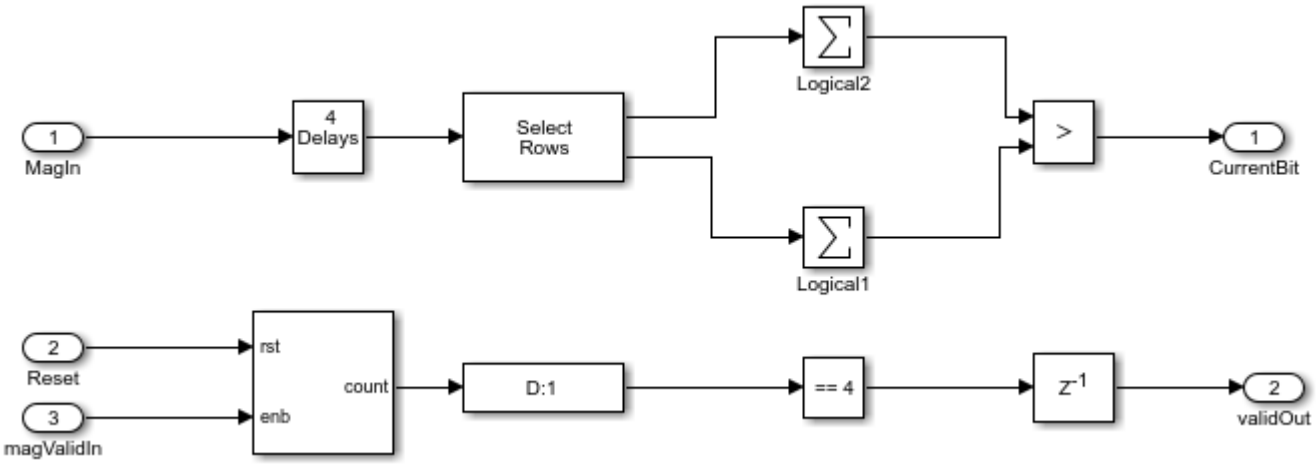


4. **Timing Control**

The **Timing Control** block is a state machine that detects the preamble and generates the control signals ActivateBP and Reset, that indicate the start of frame, end of frame and reset status to the **Bit Process** and **Compute CRC and Frame Validation** blocks.

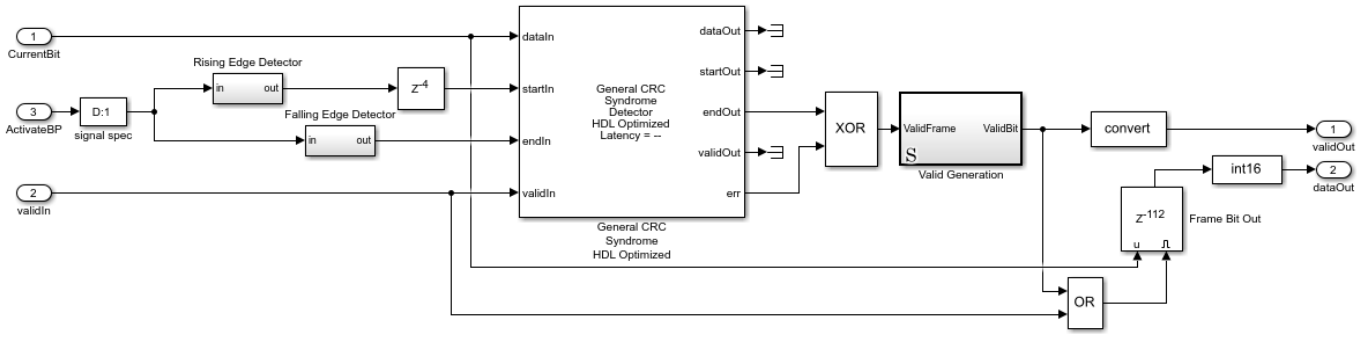
5. **Bit Process**

The **Bit Process** subsystem demodulates and down converts the 4 MHz received signal to a 1 MHz bit sequence. Each data bit is represented by four PPM bits. To demodulate, the block finds the sum of the first two bits and the last two bits of each quadruplet. Then, it compares the sums to determine the original bit value. The output valid signal is asserted every fourth cycle to align with 1 MHz bit sequence.



6. **Compute CRC and Frame Validation**

This subsystem checks for mismatches in the 24-bit checksum of each 88-bit message. The CRC block needs an indication of the frame boundaries to determine which bits are the checksum. The rising edge of the ActivateBP signal generated from the **Timing Control** block indicates the start of frame, and the falling edge indicates the end of the frame. The start signal is delayed to match the demod latency. When the block output err signal is zero, the frame is a valid ADS-B message. The subsystem buffers the message bits until the message is confirmed to have no CRC error.



Launch Map and Log Data

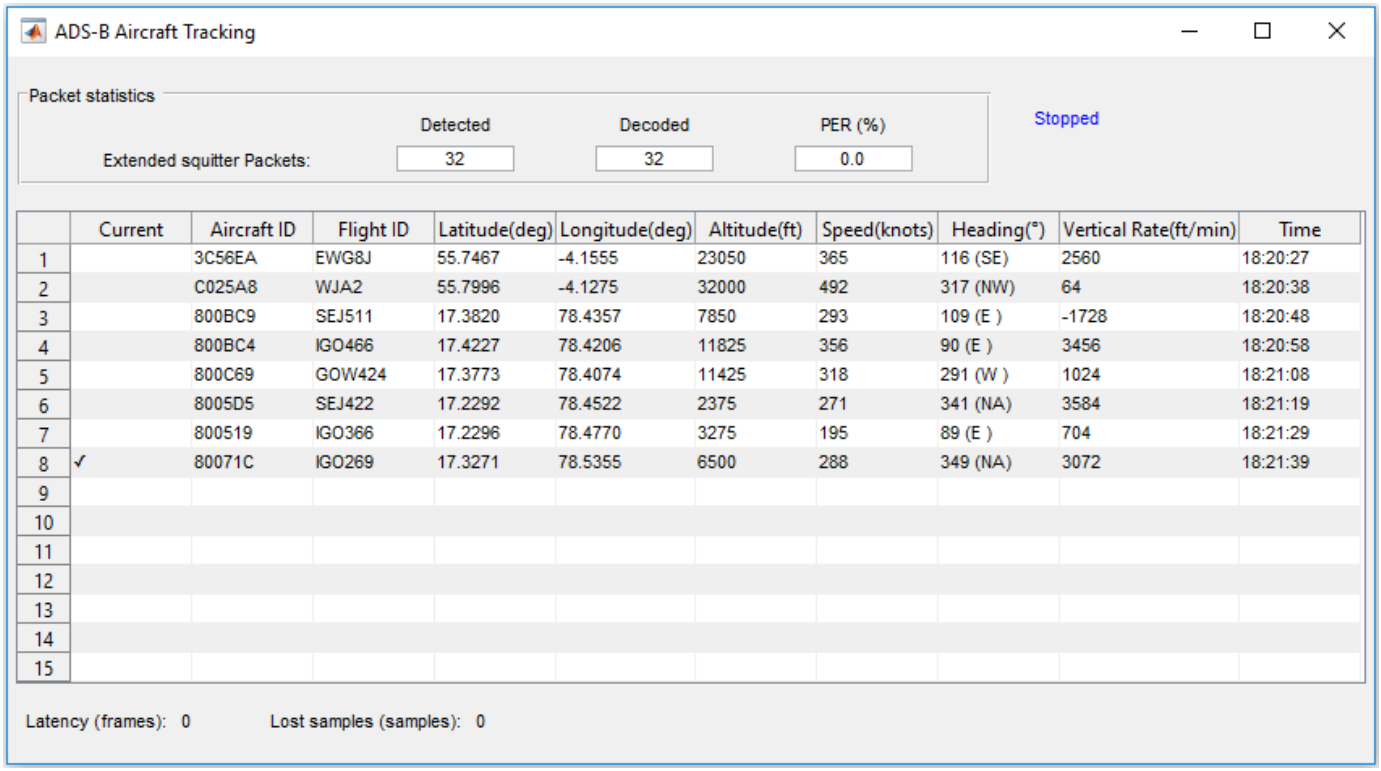
You can launch the map and start text file logging using the two slider switches (Launch Map and Data Logging).

Launch Map - Launch the map where the tracked flights can be viewed. **NOTE:** You must have a Mapping Toolbox™ license to use this feature.

Data Logging - Save the captured data in a TXT file. You can use the saved data for later for post processing.

Results and Displays

The **HDLRx** subsystem demodulates and decodes the ADS-B data and the output is streamed through **Deserializer1D** block and **MapResults** MATLAB function, which produces hexadecimal output information about the aircraft. Each extended squitter Mode S packet contains partial information (any of Aircraft ID, Flight ID, Altitude, Speed, and Location) about the aircraft and the table is built up from multiple messages. The output is obtained as shown in the following diagram. The packet statistics include the number of detected packets, the number of correctly decoded packets, and packet error rate (PER). These aircraft details match the transmitted values from the “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.



HDL Code Generation and Synthesis Results

Pipeline registers have been added to the model to make sure that **HDLRx** subsystem does not have a long critical path. The HDL code generated from the **HDLRx** subsystem was synthesized using Xilinx® Vivado® on a **Zynq** FPGA with the device 7z045ffg900-2, and the design achieves **264.2 MHz** clock frequency, which is sufficient to decode the real-time ADS-B signals. The generated HDL code is tested and verified in the real-time example “HW/SW Co-Design Implementation of ADS-B Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio). To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license. The following table shows the synthesis results of this example.

Synthesis Frequency	264.2 MHz	
Resources	Used	Utilization in %
Slice LUTs	831	0.38
Slice Registers	1689	0.39
DSP48E1	2	0.22
F7 Muxes	24	0.02
F8 Muxes	8	0.01

You can use the commands makehdl and makehdltb to generate HDL code and a test bench for the HDLRx subsystem. To generate the HDL code, use the following command:

```
makehdl('commadsbrxhdl/HDLRx')
```

To generate a test bench, use the following command:

```
makehdltb('commadsbrxhdl/HDLRx')
```

References

1 International Civil Aviation Organization, Annex 10, Volume 4. Surveillance and Collision Avoidance Systems.

2 Marvin E. Frerking, Digital Signal Processing in Communication Systems, Springer Science Business Media, New York,1994.

HDL QAM Transmitter and Receiver

This example shows how to implement a 64-QAM transmitter and receiver for HDL code generation and hardware implementation.

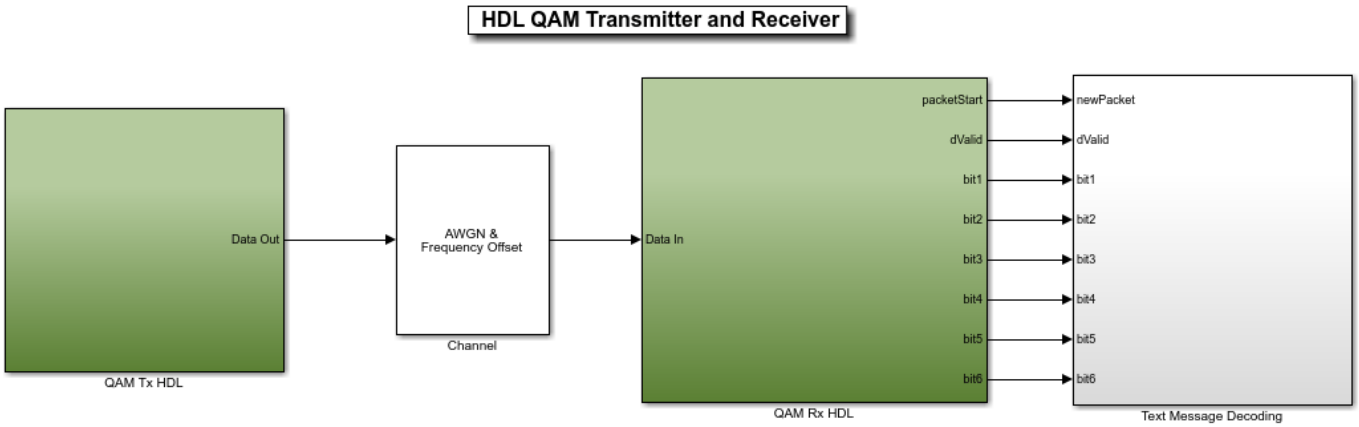
Overview

The **HDL QAM Transmitter and Receiver** example shows how to use Simulink® blocks that support HDL code generation to implement the baseband processing of a digital communications transmitter and receiver.

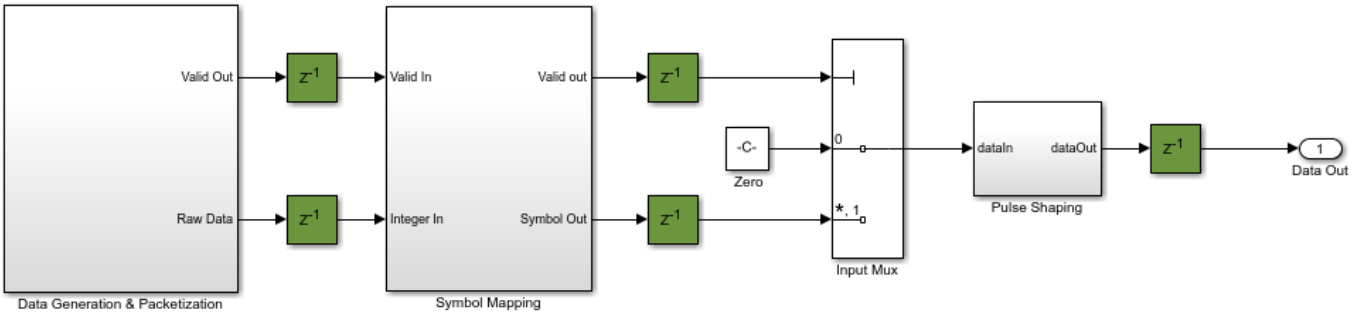
The **HDL QAM Tx** subsystem generates a complex valued, 64-QAM modulated constellation. A floating point channel model, **Channel**, is used to add attenuation, channel noise, carrier frequency offset and fractional delay in order to demonstrate the operation of the receiver subsystem. The **HDL QAM Rx** subsystem implements a practical digital receiver to mitigate the channel impairments using coarse frequency recovery, timing recovery, frame synchronization and magnitude and phase recovery. The received data packets are then decoded and printed to the MATLAB® Command Window by the **Text Message Decoding** subsystem.

Structure of the Example

The top-level structure of the QAM receiver model is shown in the following figure. The **QAM Tx HDL** and **QAM Rx HDL** subsystems are optimized for HDL code generation.



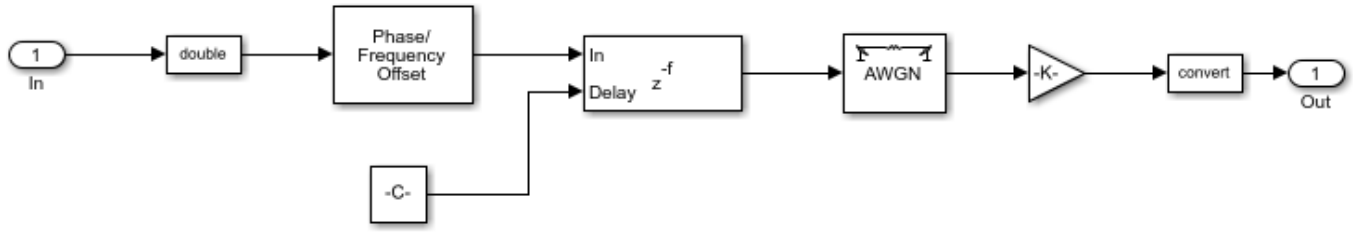
The detailed structure of the **QAM Tx HDL** subsystem can be seen in the figure below.



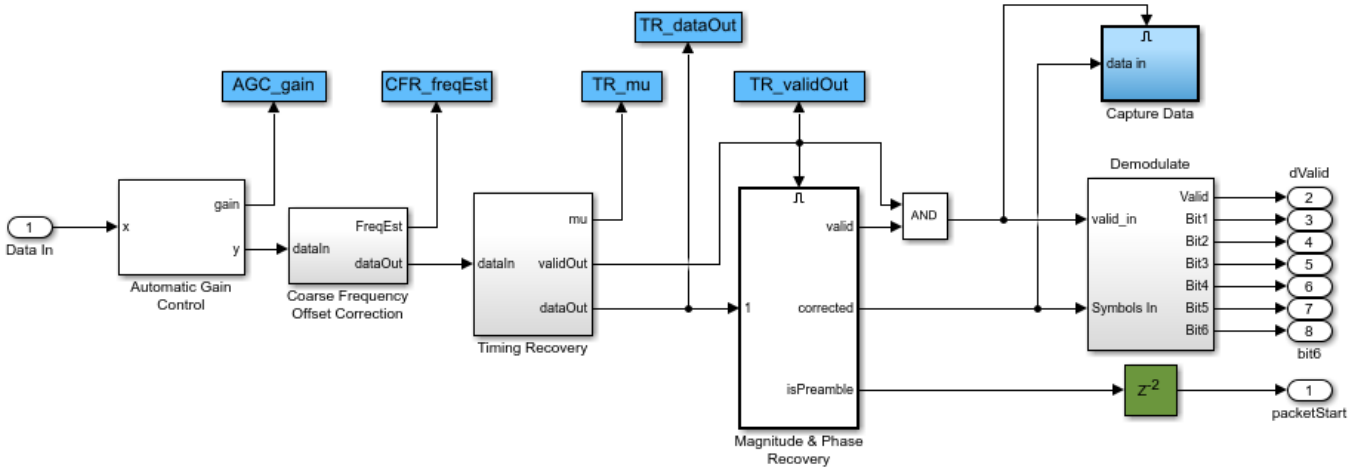
The **QAM Tx HDL** subsystem contains the following components, which are described in more detail in the **HDL QAM Transmitter** section.

- **Data Generation & Packetization** - Generates the packets to be transmitted, grouping the bits for mapping to symbols
- **Symbol Mapping** - Maps the bits output from the **Data Generation & Packetization** subsystem to QAM symbols
- **Pulse Shaping** - Performs pulse shaping and upsampling of the symbols using an interpolating RRC (Root Raised Cosine) filter prior to transmission

The structure of the **Channel** can be seen below. As the **Channel** subsystem is intended to be a rough approximation of a AWGN channel with attenuation and frequency offset it is intended to be run in software. As a result blocks which are not supported for HDL code generation can be used here, such as the **Phase/Frequency Offset** block. The **Phase/Frequency Offset** block does not support fixed point data types, hence the conversion to double at the input of the **Channel** subsystem. The signal is converted back to fixed point before being output from the **Channel** subsystem. A fractional delay and AWGN are applied to the transmitted signal and the **Gain** block attenuates the signal.



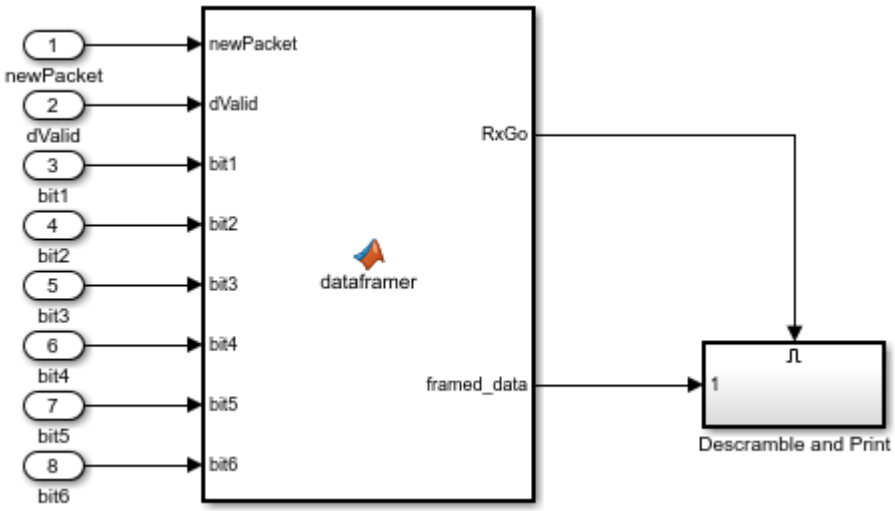
The detailed structure of the **QAM Rx HDL** subsystem can be seen in the figure below.



The **QAM Rx HDL** subsystem contains the following components which are described in more detail in the **HDL QAM Receiver** section.

- **Automatic Gain Control (AGC)** - Normalizes the received signal power
- **Coarse Frequency Offset Correction** - Estimates the approximate frequency offset and corrects. The subsystem also contains the receive RRC filter which downsamples by 2
- **Timing Recovery** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants
- **Magnitude & Phase Recovery** - Performs packet detection, fine grained phase and amplitude correction
- **Demodulate** - Demodulates the signal, de-mapping symbols to bits

The structure of the **Text Message Decoding** subsystem is shown below.



This subsystem is expected to be run in software, therefore, it is preferable to employ frame-based signals to speed up the computation. The **Text Message Decoding** subsystem has eight sample-based Boolean input signals: dValid, packetStart and signals bit1 to bit6. Conversion from sample-based signals to frame-based counterparts is implemented by the **dataframer** MATLAB function block. The demodulated bits are valid only when dValid is set high. The **dataframer** block uses the dValid signal to fill up a delay line with the received bits and the **newPacket** signal to forward the data stored in the delay line to the output and reset the delay line. The **Descramble and Print** subsystem processes the received data only when its enable signal goes high. This occurs when either the delay line accumulates 336 valid demodulated bits or the newPacket signal is high. This will cause the **dataframer** to set the RxGo signal high. While the simulation is running, the **Descramble and Print** subsystem outputs the string "Hello world! ~64QAM test string~ ###" to the MATLAB command window, where '###' is a repeating sequence of '000', '001', '002', ..., '099'. Every 50 packets, the bit error rate of the data in the last 50 successfully received packets is also displayed in the MATLAB Command Window.

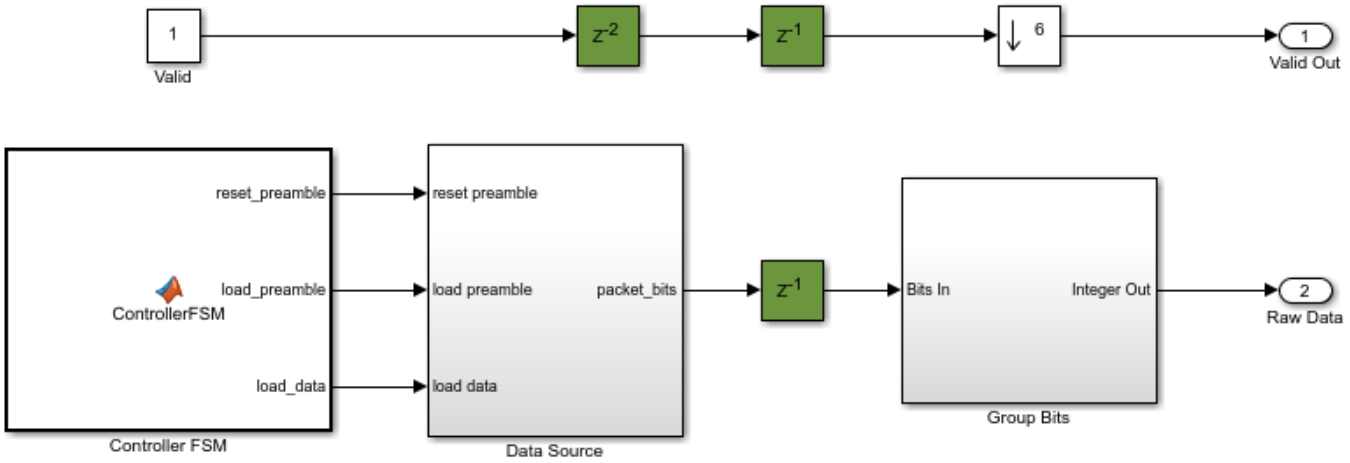
HDL QAM Transmitter (HDL QAM Tx)

The **HDL Transmitter** contains the **Data Generation & Packetization**, **Symbol Mapping**, and **Pulse Shaping** blocks which are described in detail in the following sections.

1 - **Data Generation & Packetization**

The **Controller FSM** (Finite State Machine) and **Data Source** generates the preamble bits, and the data bits, performs scrambling and builds the packets. Each packet consists of an 84-bit Barker code preamble and 252 bits of scrambled data. The **Group Bits** block converts the input data bit stream into a six bit integer at 1/6th of the input sampling rate, as required by the symbol mapper.

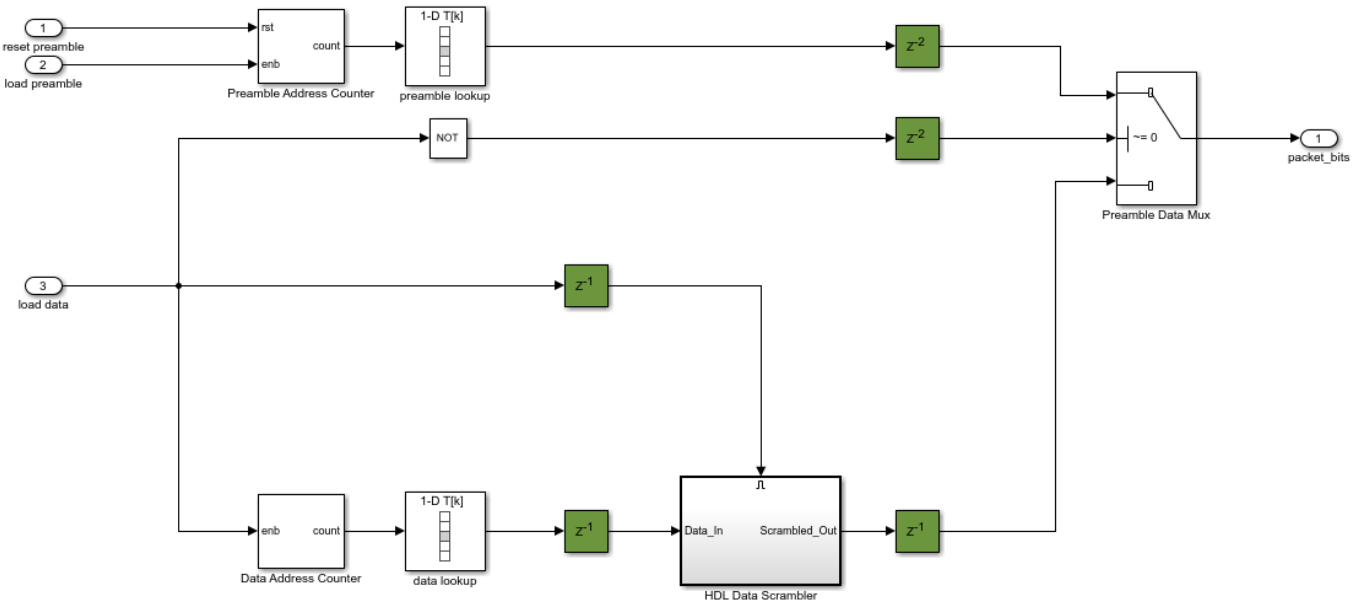
The **Data Source** subsystem has a pipeline delay of 2 samples. In addition there is a pipeline delay between the data source and the bit pairing subsystem. The valid signal is therefore delayed to match the pipeline delay of the data path. The **Group Bits** subsystem reduces the sample rate by a factor of 6. Placing a downsample by 6 in the valid control path ensures that the sample rate matches that of the signal path.



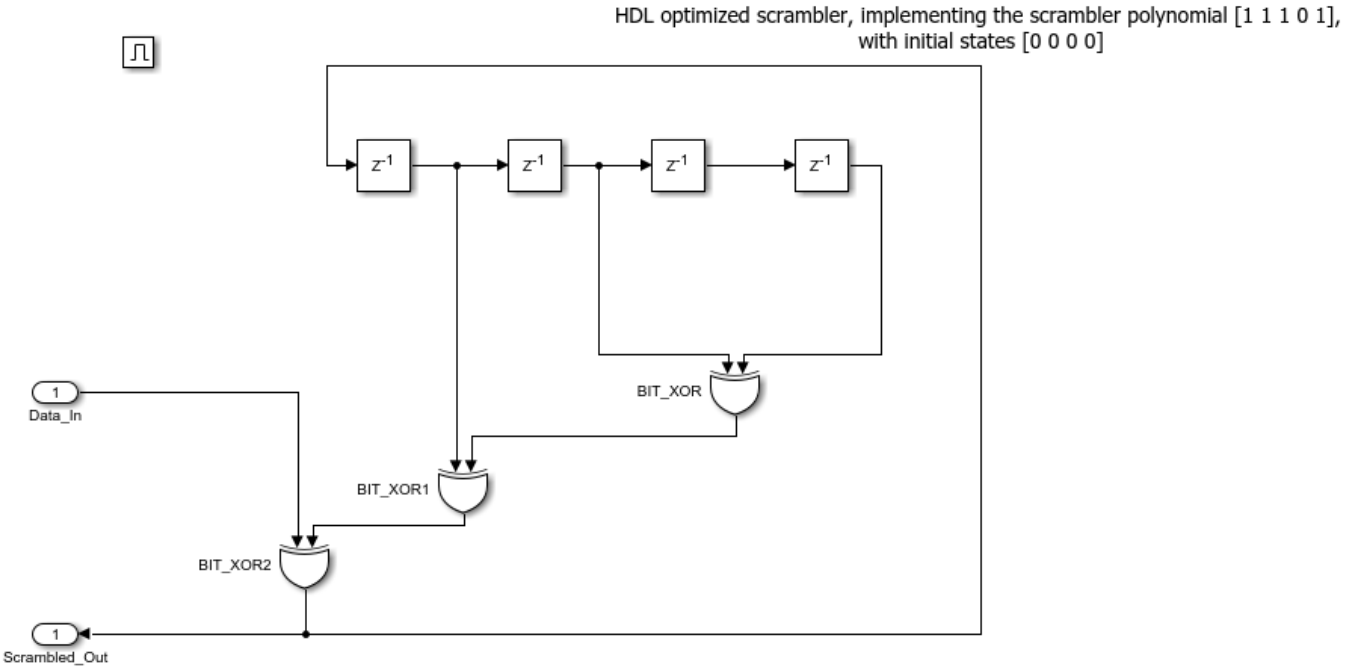
- **Controller FSM** - The **Controller FSM** implements a control state machine using a MATLAB™ function block. The FSM has two states - **Pack_Preamble** and **Append_Data**. The **Pack_Preamble** state asserts the **load_preamble** signal and de-asserts the **reset_preamble** and the **load_data** signals. The FSM will remain in this state for 84 clock cycles. Following this the

FSM moves into the **Append_Data** state, asserting the **load_data** signal and the **reset_preamble** signal while releasing the **load_preamble** signal. The FSM will remain in this state for 252 clock cycles. The **load_preamble** and **reset_preamble** are boolean and are used to control the **Preamble Address Counter** which manages the load of the preamble at the start of each packet. The **load_data** signal is boolean and is used to enable the **Data Address Counter** which controls the loading of data into the packet.

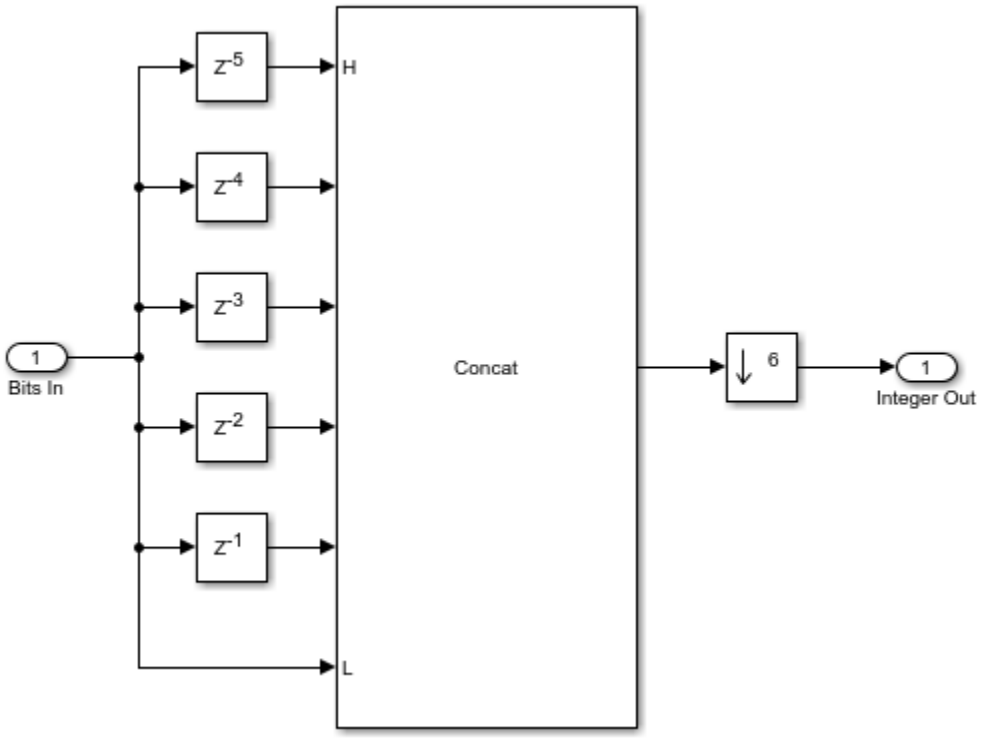
- **Data Source** - The **Data Source** Subsystem contains two LUTs, storing the preamble and data bits. The **preamble lookup** LUT is addressed by the **Preamble Address Counter**, which is controlled by the **reset preamble** and **load preamble** signals generated by the **Controller FSM**. The **data lookup** LUT is addressed by the **Data Address Counter**, which is enabled by the **load_data** signal generated by the **Controller FSM**. The **Preamble Address Counter** has a reset signal, generated by the **Controller FSM**, as the same preamble is inserted at the start of each packet. The **Data Address Counter** does not have a reset signal as the data address sequence is much longer and will vary for each packet as different data bits are placed within each packet. In addition to enabling the counter for the data LUT, the **load data** input is used to control when the **HDL Data Scrambler** component should be enabled, and to control selection of preamble or data bits via the **Preamble Data Mux**.



- **HDL Data Scrambler** - The **HDL Data Scrambler** is shown in the following figure. It is built from first principles using XOR gates (for modulo 2 addition) and registers. An enabled subsystem is used here to ensured that the scrambler is only enabled when there is new input data to be processed.

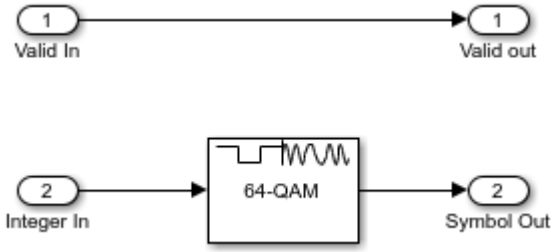


- **Group Bits** - The purpose of the **Group Bits** subsystem is to group six individual bits into a six-bit unsigned integer output - the format expected by the symbol mapping component. A number of delays are used to align 6 bits at the input of the **Bit Concat** block which concatenates into a six-bit unsigned output. This output is then downsampled to select the correct grouping of bits.



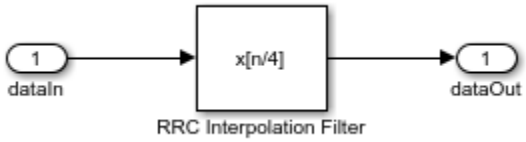
2 - **Symbol Mapping**

The **Symbol Mapping** subsystem uses the **Rectangular QAM Modulator Baseband** block to map the integer input value onto the appropriate 64-QAM complex valued symbol. The block uses a Gray Mapping scheme.



3 - **Pulse Shaping**

The Pulse Shaping subsystem uses an **RRC Interpolation Filter** block with an upsampling factor of 4. A matched filter is implemented in the receiver. The filter is pipelined (see HDL Block Properties).



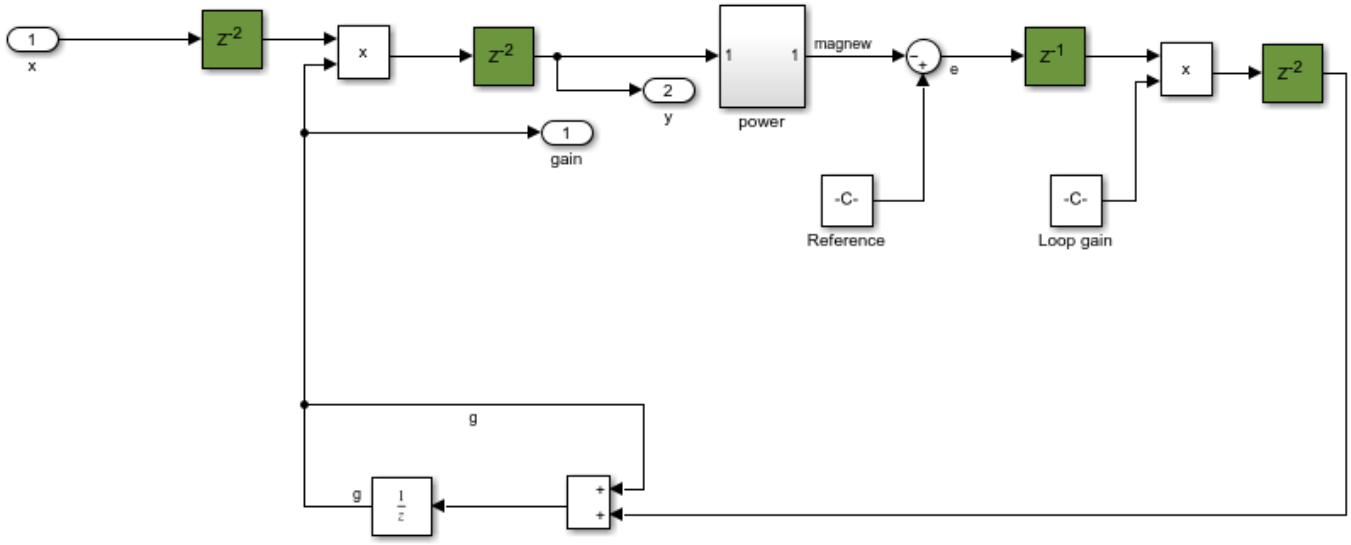
HDL QAM Receiver (HDL QAM Rx)

The **HDL Receiver** contains the **AGC**, **Coarse Frequency Offset Correction**, **Timing Recovery**, **Magnitude & Phase Recovery**, and **Demodulate** blocks, which are described in detail in the following sections.

1 - **AGC**

The **AGC** ensures that the amplitude of the input of the **Coarse Frequency Compensation** is normalized to the range 1 to -1.

The **AGC** structure is shown in the following diagram, with pipeline registers shown in green throughout the model.

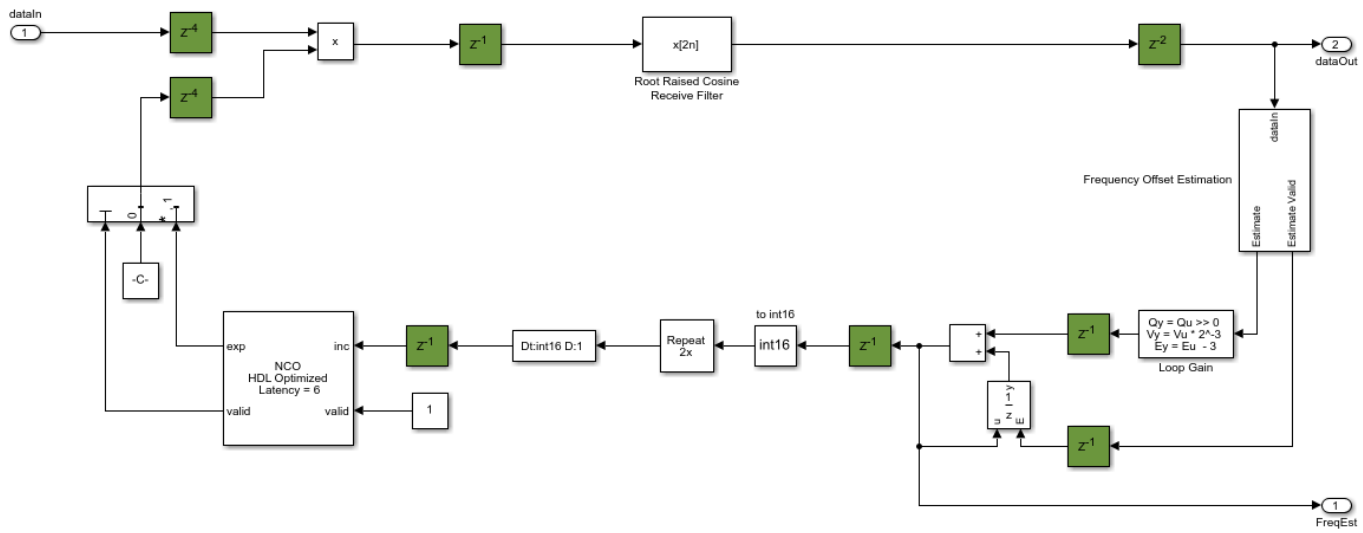


2 - **Coarse Frequency Offset Correction**

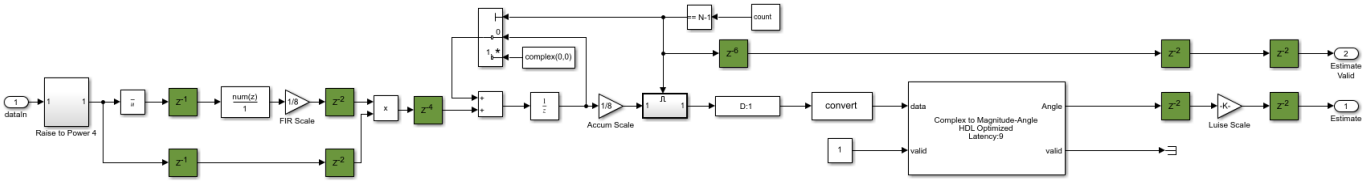
The **Coarse Frequency Offset Correction** subsystem estimates and corrects for the frequency offset using the Luise-Reggiannini algorithm [1]. The **Frequency Offset Estimation** subsystem makes an estimate based on the output of the **Root Raised Cosine Receive Filter**, then frequency offset correction based on this estimate is applied at the input to the **Root Raised Cosine Receive Filter**. This ensures that the desired portion of the received signal bandwidth is better aligned with the receiver filter frequency response, improving the SNR compared to correcting at the output of the **Root Raised Cosine Receive Filter**.

As the estimation and correction algorithm is operating in a closed loop, making iterative updates to the previous estimates of the frequency offset, the system will gradually converge towards a result. A **Loop Gain** is included to implement averaging of the estimates. This architecture is described in [1]. The **Root Raised Cosine Receive Filter** implements a downsampling operation so it is necessary to upsample the feedback signal, using the repeat block, to match the rate at the input to the filter.

Note that there is a residual frequency offset at the output of the **Coarse Frequency Offset Correction** subsystem that varies over time, even if the frequency offset at the input to the subsystem remains the same, as new estimates of the offset are made. Fine grained correction of the residual offset is performed later in the receiver by the **Magnitude and Phase Recovery** subsystem.

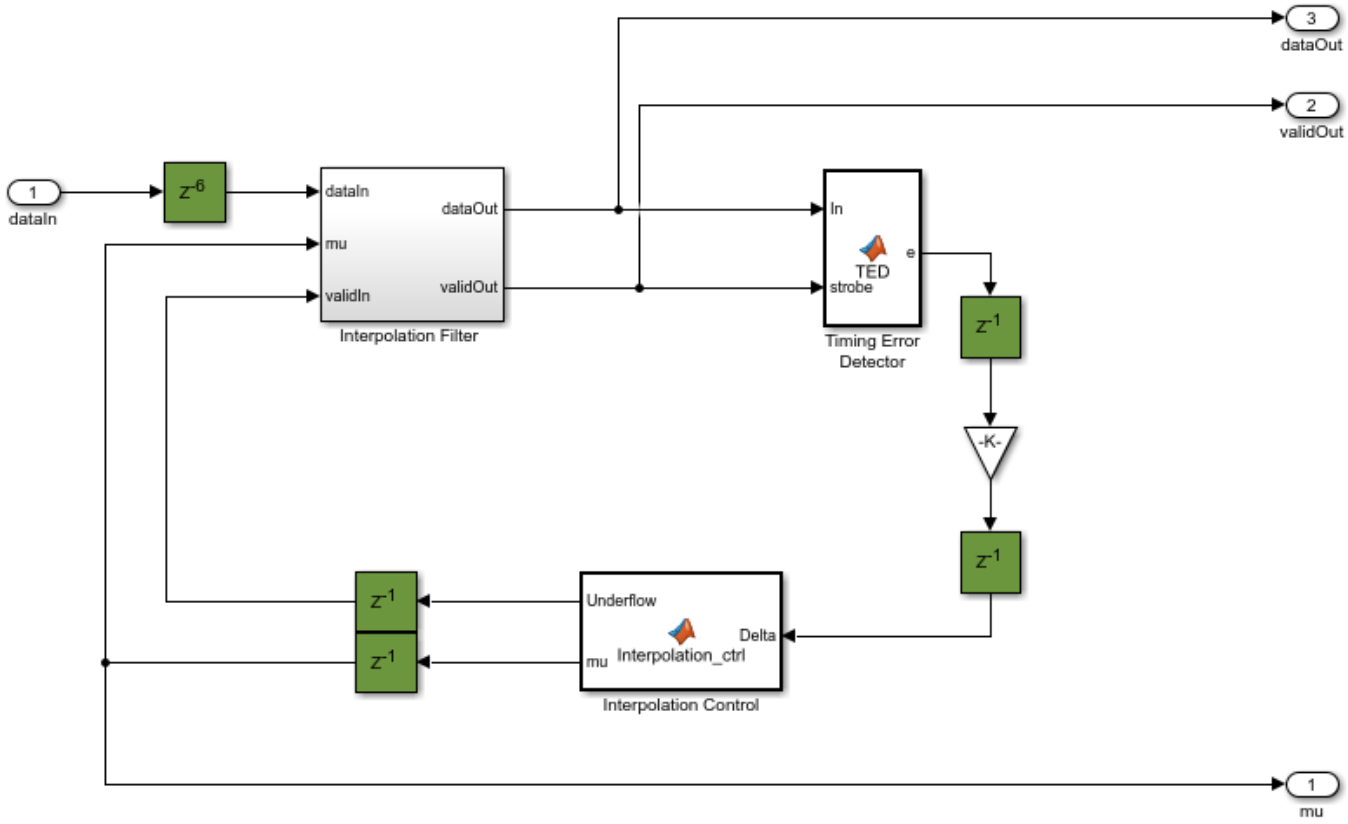


- **Frequency Offset Estimation** : The **Frequency Offset Estimation** subsystem implements the Luise-Regiannini algorithm, described in [1]. The signal is first raised to the power four to implement a 4th power phase estimator as described in [2]. This is implemented by 2 cascaded product blocks, with pipelining added to improve hardware performance. The **Discrete FIR Filter** implements the filter with rectangular weights, made up of all ones, described in [1]. The **FIR Scale** scales the FIR output to account for the filter gain. The **Complex To Magnitude-Angle HDL Optimized** block is used to implement the $angle$ function, as required by the Luise-Reggiannini algorithm. This block computes the phase using the hardware friendly CORDIC algorithm. For more information, see the Complex to Magnitude-Angle HDL Optimized block in DSP System Toolbox™. Before the **Frequency Offset Estimation** subsystem output, the signal is scaled as required by the Luise-Regiannini algorithm and, in addition, is scaled to match the word length of the NCO.



3 - **Timing Recovery**

The **Timing Recovery** subsystem is shown in the following diagram.



The **Timing Recovery** subsystem implements a PLL, described in Chapter 8 of [3], to correct the timing error in the received signal. On average, the **Timing Recovery** subsystem generates one output sample for every two input samples.

The **Interpolation Control** function block implements a decrementing modulo-1 counter, described in Chapter 8.4.3 of [3], to generate the control signal to facilitate the selection of the interpolants of the **Interpolation Filter**. This control signal also enables the **Timing Error Detector (TED)**, so that it calculates the timing errors at the correct timing instants. The **Interpolation Control** subsystem updates the timing difference, **mu**, for the **Interpolation Filter**, generating interpolants at optimum sampling instants.

The **Interpolation Filter** is a Farrow parabolic filter with $\alpha = 0.5$ as described in Chapter 8.4.2 of [3]. The filter uses an α of 0.5 so that all the filter coefficients become 1, -1/2 and 3/2, which significantly simplifies the interpolator structure. Based on the interpolants, timing errors are generated by a zero-crossing **Timing Error Detector** as described in Chapter 8.4.1 of [3].

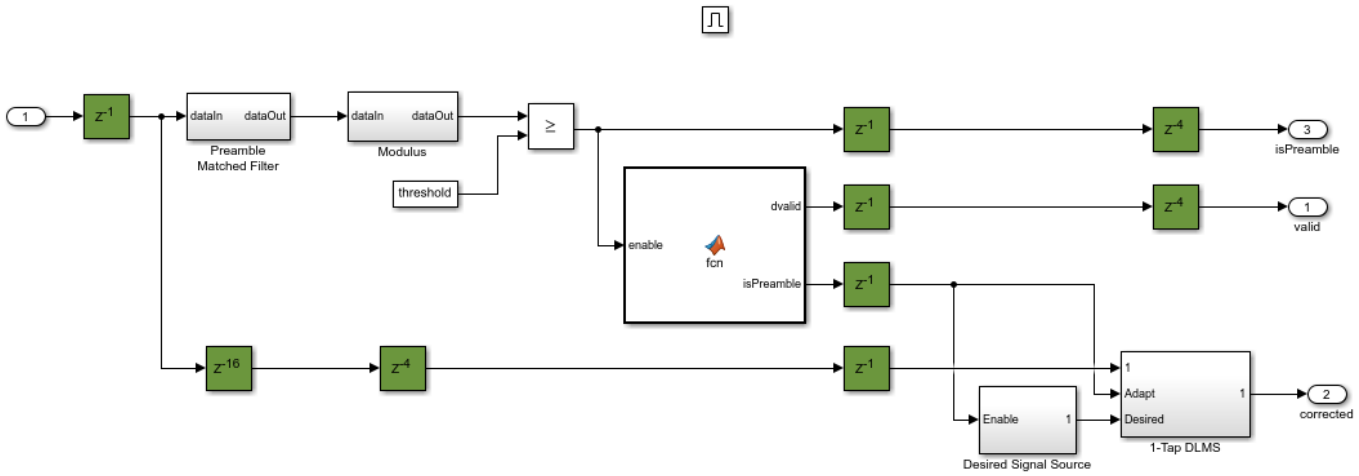
The **Interpolation Filter** introduces a fractional delay to the signal in order to compensate for the timing error. The fractional delay is controlled by the **mu** input signal. When the timing error (delay) reaches symbol boundaries, there is one extra or missing interpolant in the output. The **Timing Error Detector** implements bit stuffing or skipping to handle the extra or missing interpolants.

Refer to Chapter 8.4.4 of [3] for details of bit stuffing and skipping. The timing recovery loop normally generates one output symbol for every two input samples. It also outputs a timing strobe (**validOut** signal) that runs at the input sample rate. Under normal circumstances, the strobe value is simply a sequence of alternating ones and zeros. However, this occurs only when the relative delay

between transmitter and receiver contains some fractional part of one symbol period and the integer part of the delay (in symbols) remains constant. If the integer part of the relative delay changes, the strobe value can have two consecutive zeros or two consecutive ones.

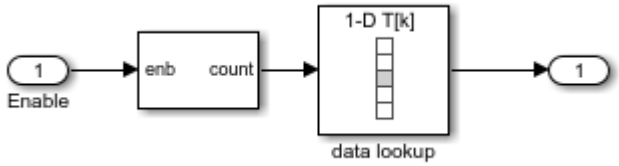
4 - **Magnitude & Phase Recovery**

The **Magnitude & Phase Recovery** subsystem performs packet synchronization, fine grained frequency recovery and fine grained amplitude recovery.

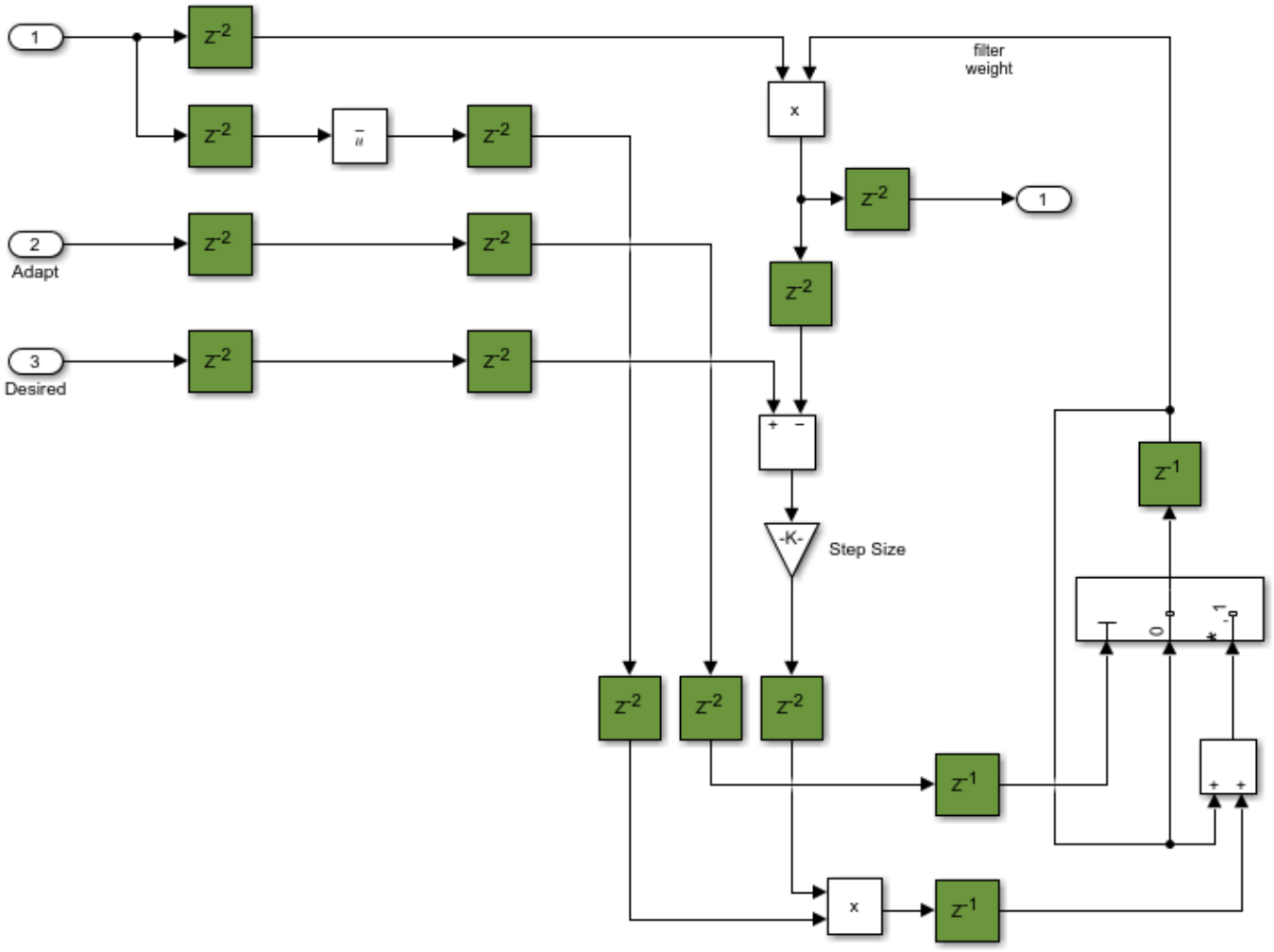


- **Packet Synchronization:** The **Preamble Matched Filter** subsystem uses the time-reversed complex conjugate of the preamble as the filter weights. The modulus of the output of the **Preamble Matched Filter** subsystem is calculated using the **Modulus** subsystem. The output of the **Modulus** subsystem is then compared to a threshold to detect the preamble at the start of a packet. The MATLAB function block generates a signal, **isPreamble**, which is held high for the duration of the preamble of each packet. The MATLAB function block also generates the **dvalid** signal which is set high for the duration of the packet when a preamble has been detected.
- **Fine Grained Magnitude and Phase Recovery :** The **1-Tap DLMS** (Delayed Least Mean Squares) filter subsystem, adapting over the preamble and using the reference signal generated by **Desired Signal Source**, corrects for both phase and magnitude errors. The **isPreamble** signal, generated by the MATLAB function block and set high for the 14 preamble symbols once a packet has been detected, is used to enable the desired signal source and to enable the **Adapt** input of the **1-Tap DLMS**. When the **isPreamble** signal is low, the weight in the **1-Tap DLMS** is held and the **Desired Signal Source** is reset. The Delayed LMS (DLMS) [4] algorithm is used here to allow for more pipelining to be introduced and, therefore, reduce the critical path in the filter and increase the maximum clock rate achievable after being implemented in hardware.

The internal structure of the **Desired Signal Source** subsystem is shown below. The **data lookup** LUT contains the preamble symbols.

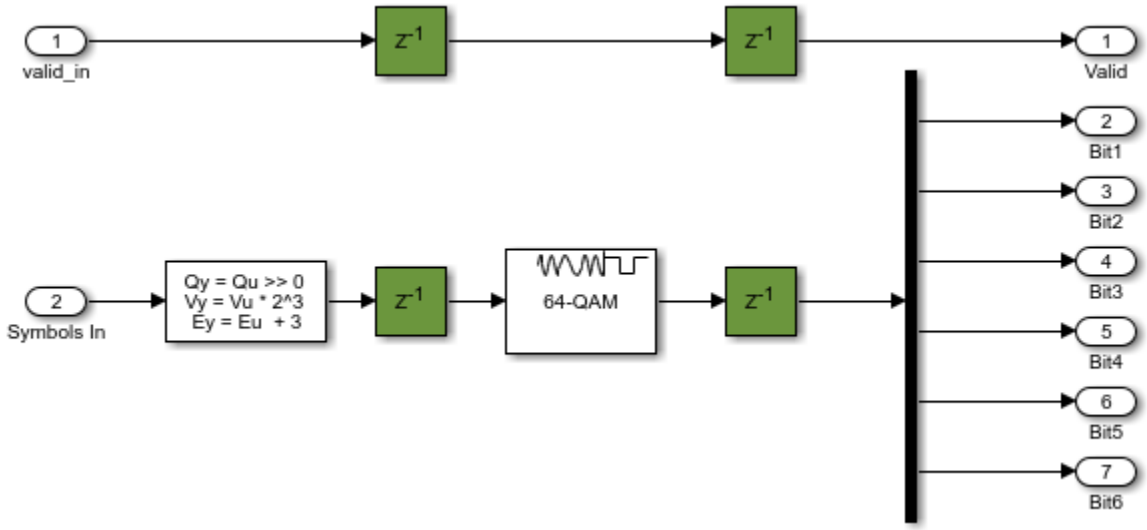


The internal structure of the **1-Tap DLMS** subsystem is shown below.



5 - **Demodulate**

The **Demodulate** subsystem maps each 64-QAM input symbol to bits, outputting 6 bits for each input symbol. To generate HDL for the **Rectangular QAM Demodulator Baseband** block, the minimum distance between symbols must be set to 2. This is 8 times larger than the distance between the symbols generated in the transmitter. As a result, the symbols input to the **Demodulate** subsystem must be scaled up appropriately. This is done using the **Shift Arithmetic** block which shifts the binary point left by 3 bits to achieve the required multiplication by 8.



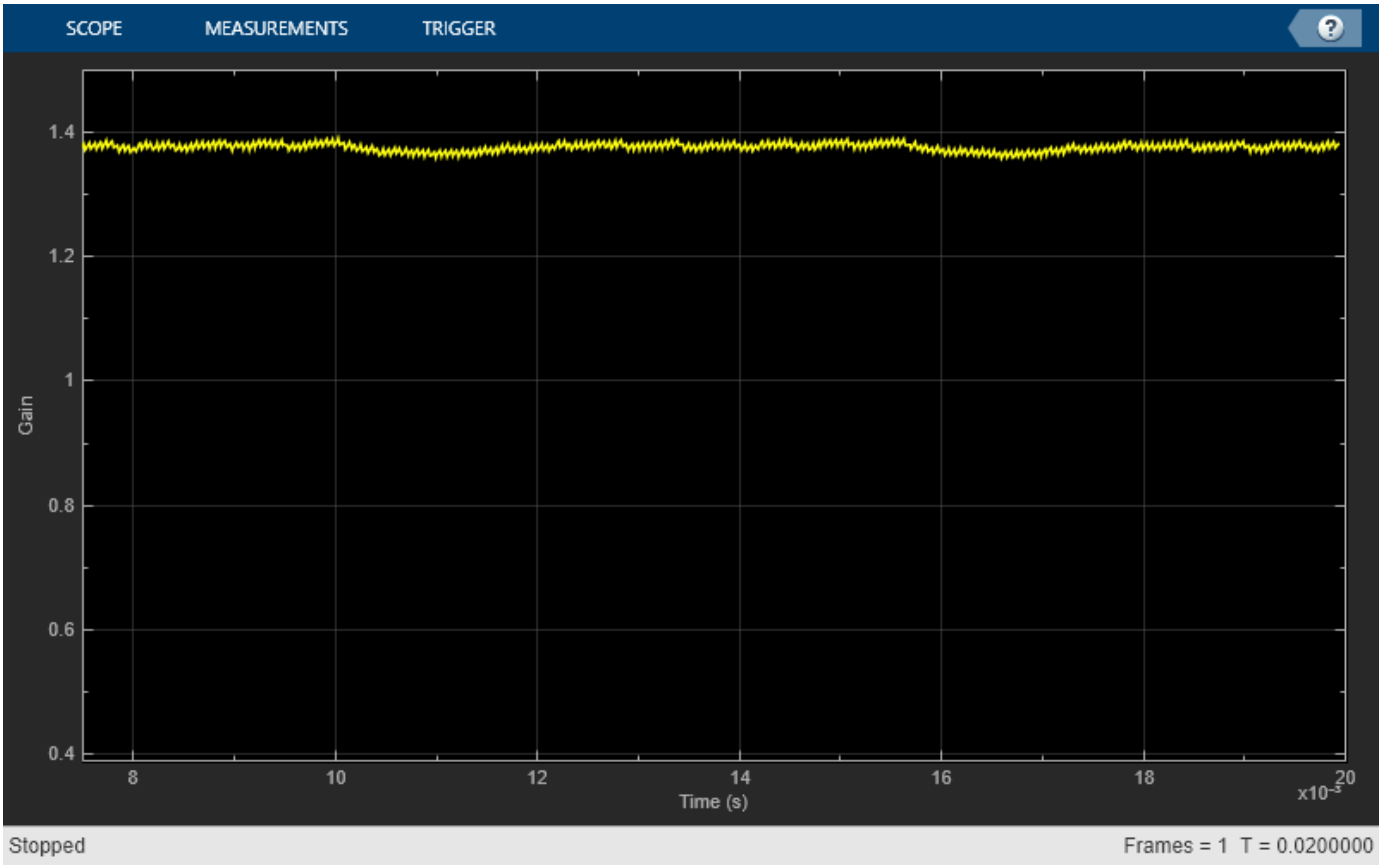
Results and Displays

During the simulation, the model displays successfully received packets in the MATLAB Command Window. At every 50 packets, the bit error rate of the data in the last 50 successfully received packets is also displayed in the MATLAB Command Window.

After running the simulation, the model displays six different figures illustrating different aspects of the receiver performance. These are shown below, along with an explanation of each plot. The first five plots show the adaption, over the simulation duration, of the **Automatic Gain Control**, the **Frequency Offset Estimation**, the **Timing Recovery** position estimate, the real part of the constellation at the output of the **Timing Recovery** subsystem, and at the output of the **Magnitude & Phase Recover** subsystem. The last plot shows the constellation diagram at the output of **Magnitude & Phase Recovery** subsystem after any adaption has taken place.

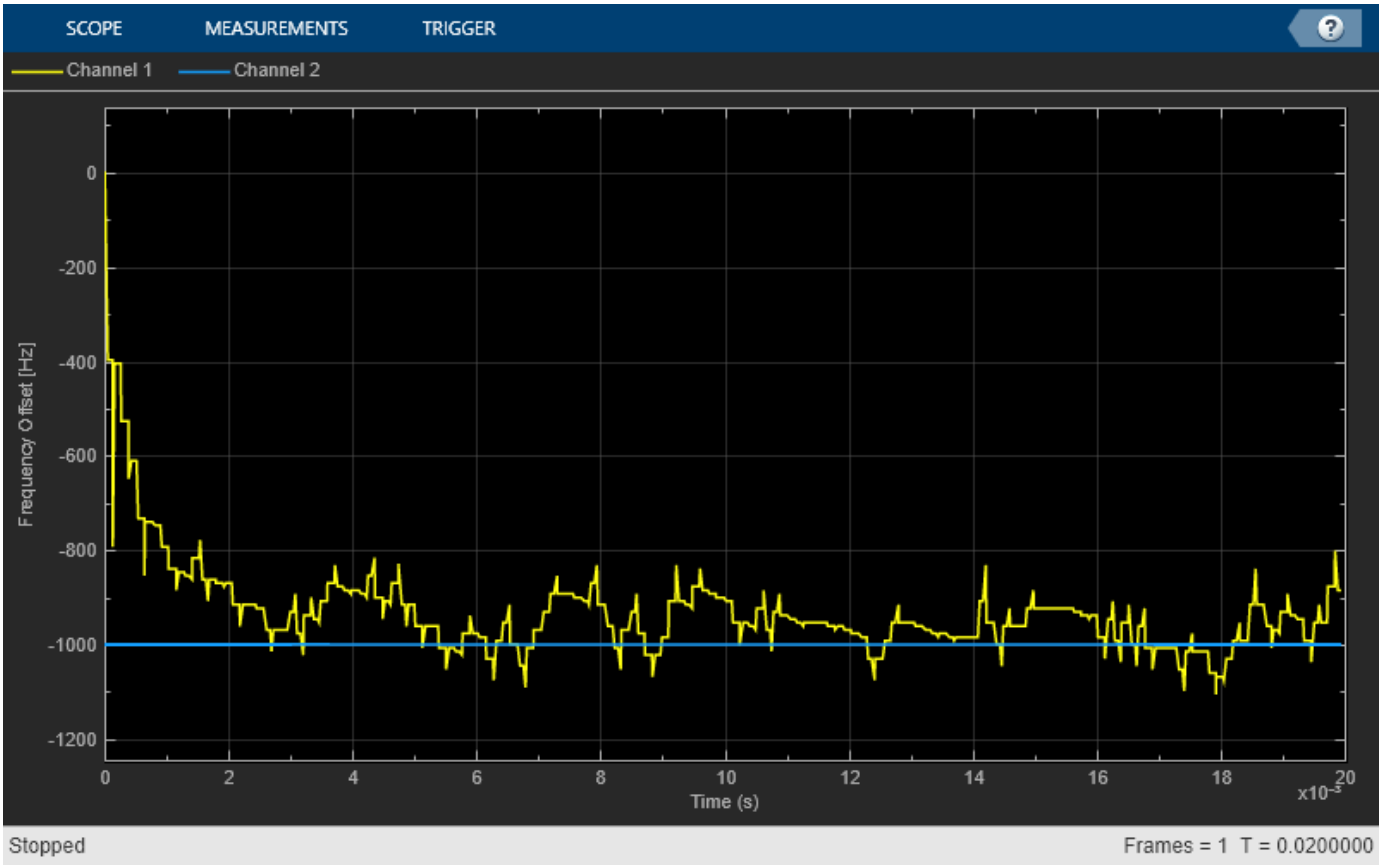
- **AGC Gain Plot**

The following plot illustrates the **Automatic Gain Control** subsystem adapting over time to normalize the output. A balance must be struck between how quickly the AGC adapts and how much ripple there is after the gain has reached a relatively constant level. Using a larger AGC loop gain adapts faster but the amplitude after adaption varies more. Using a smaller loop gain slows the adaption of the AGC, smoothing the level after adaption but taking longer to adapt.



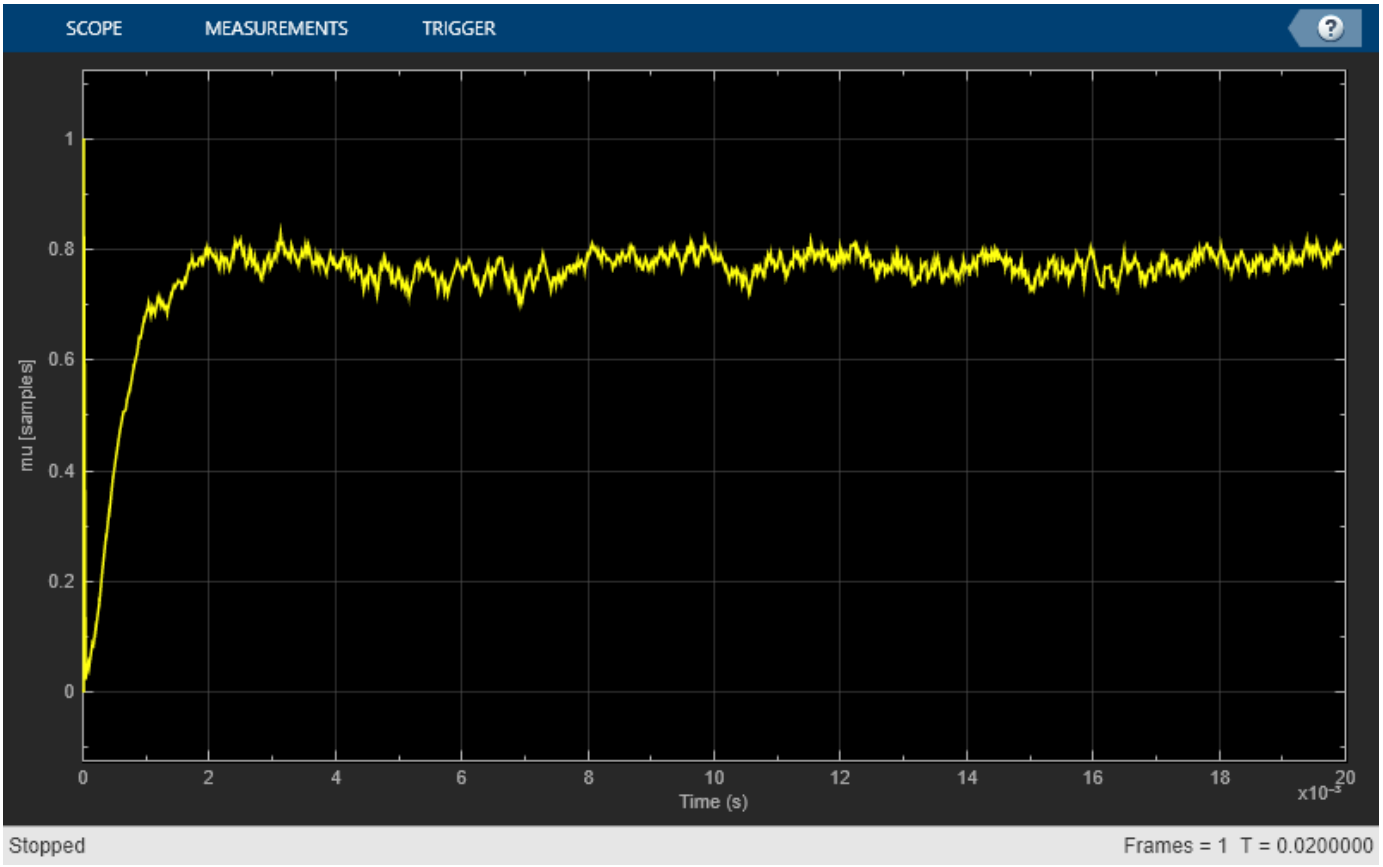
- **Frequency Offset Estimate Plot**

The following plot illustrates how the coarse frequency offset gradually adapts towards the frequency offset introduced by the system (the blue horizontal line). It shows that while the estimate comes close to the actual frequency offset, there is still a residual error that must be addressed later in the system.



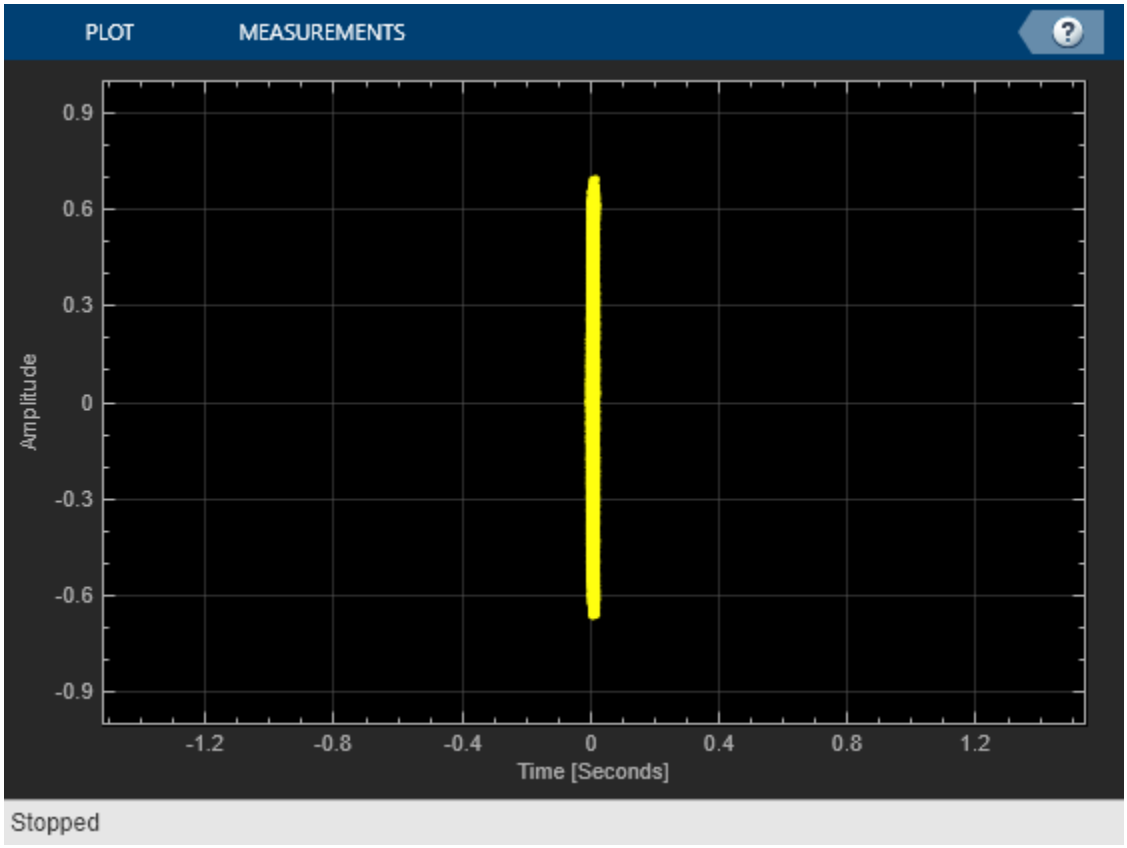
- **Timing Recovery Position Plot**

The following plot shows the **mu** input to the **Interpolation Filter**. Note that **mu** converges to a steady state (with some ripple) over time as the channel delay is not varying during the simulation.



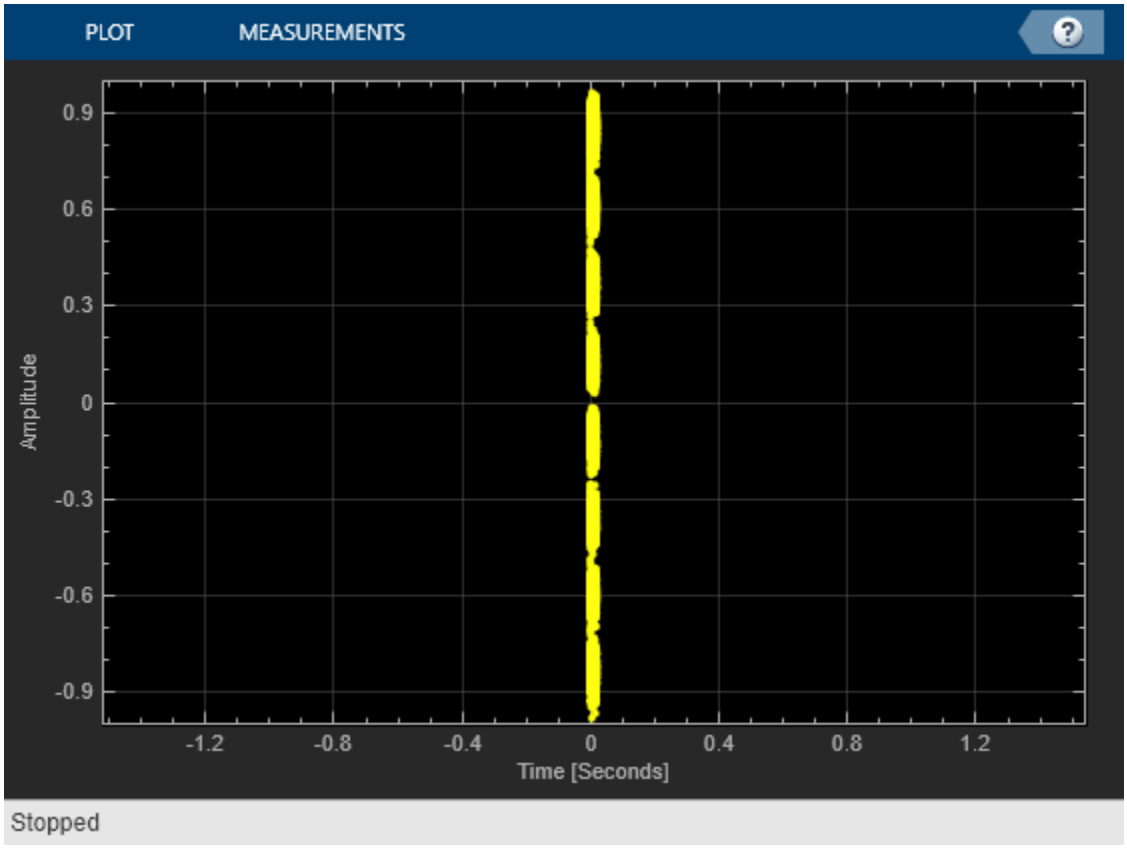
- **Real Part of Timing Recovery Output Plot**

The following plot illustrates how the real part of the **Timing Recovery** subsystem output is beginning to converge towards the eight distinct amplitude levels expected for 64QAM. However, as the residual frequency offset remaining after the coarse frequency recovery has not yet been corrected at this point in the receiver, the quality of the signal varies with the distinct amplitude levels more clearly visible at some points than at others. The constellation still has some rotation at this point in the receiver.



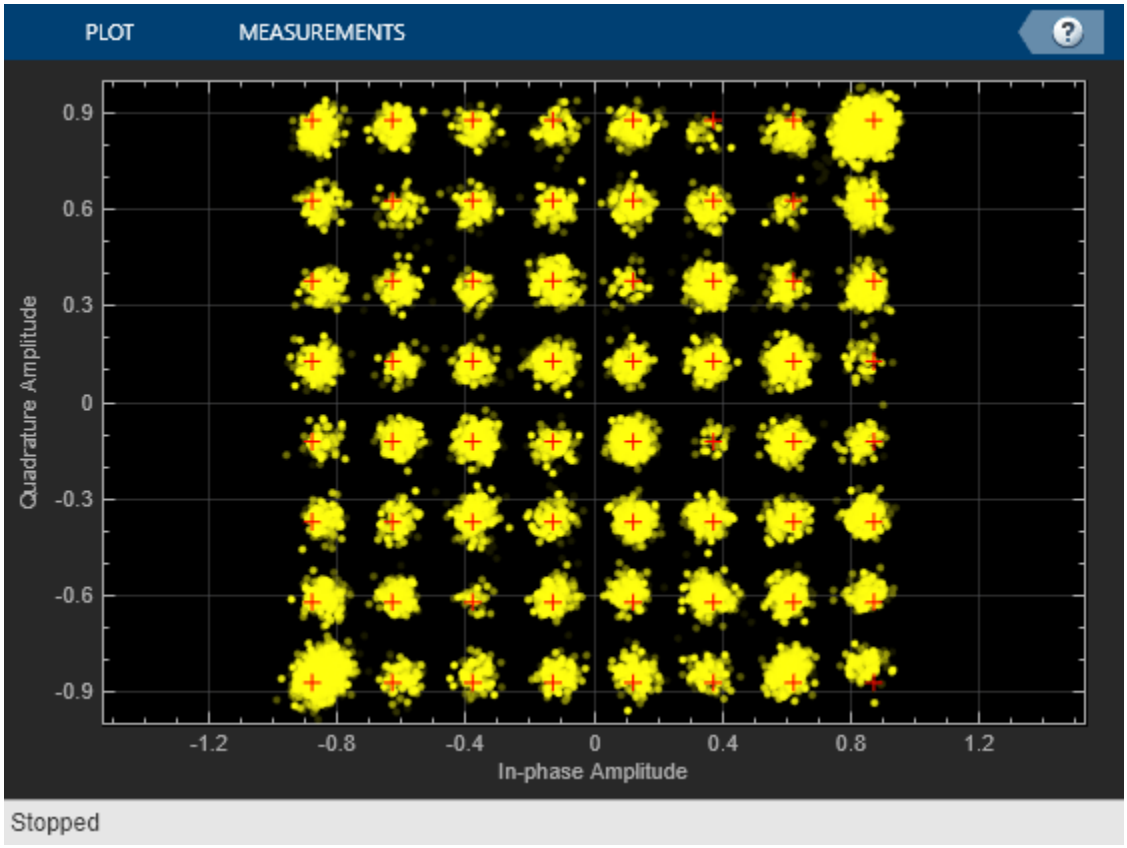
- **Real Part of Symbol Estimates Plot**

The following plot shows how the real part of output of the **Magnitude & Phase Recovery** subsystem adapts over time. Unlike the previous plot, this diagram is generated after the fine frequency recovery, therefore the constellation should not be rotating. There are no samples initially as the output from the block is not valid, and then eight clear amplitude levels should be seen - representing the eight real amplitude levels of the 64-QAM constellation.



- **Recovered Constellation Plot**

The following plot shows the constellation at the output of the **Magnitude & Phase Recovery** subsystem after the system has had time to adapt to the channel. Reducing the channel noise should reduce the size of each of the constellation points; increasing the channel noise begins to merge the distinct constellation points together. If the system has not successfully corrected for the frequency offset, then rotation of the constellation is visible here.



References

1. M. Luise and R. Reggiannini, "Carrier frequency recovery in all-digital modems for burst-mode transmissions," *IEEE Trans. Communications*, pp. 1169-1178, 1995.

2. Moeneclaey, M. and De Jonghe, G. "ML-oriented NDA carrier synchronization for general rotationally symmetric signal constellations", *IEEE Trans. Communications*, pp.2531-2533, 1994.

3. Michael Rice, "Digital Communications - A Discrete-Time Approach", Prentice Hall, April 2008.

4. G. Long , F. Ling and J. G. Proakis "The LMS algorithm with delayed coefficient adaptation", *IEEE Trans. on Acoustics, Speech and Signal Processing*, pp.1397-1405, 1989.

HDL QPSK Transmitter and Receiver

This example shows how to implement a QPSK transmitter and receiver in Simulink® that is optimized for HDL code generation and hardware implementation.

The model shown in this example modulates data based on quadrature phase shift keying (QPSK). The goal of this example is to model an HDL QPSK communication system that can transmit and recover information for a real-time system. The receiver implements symbol timing synchronization and carrier frequency and phase synchronization, which are essential in a single-carrier communication system.

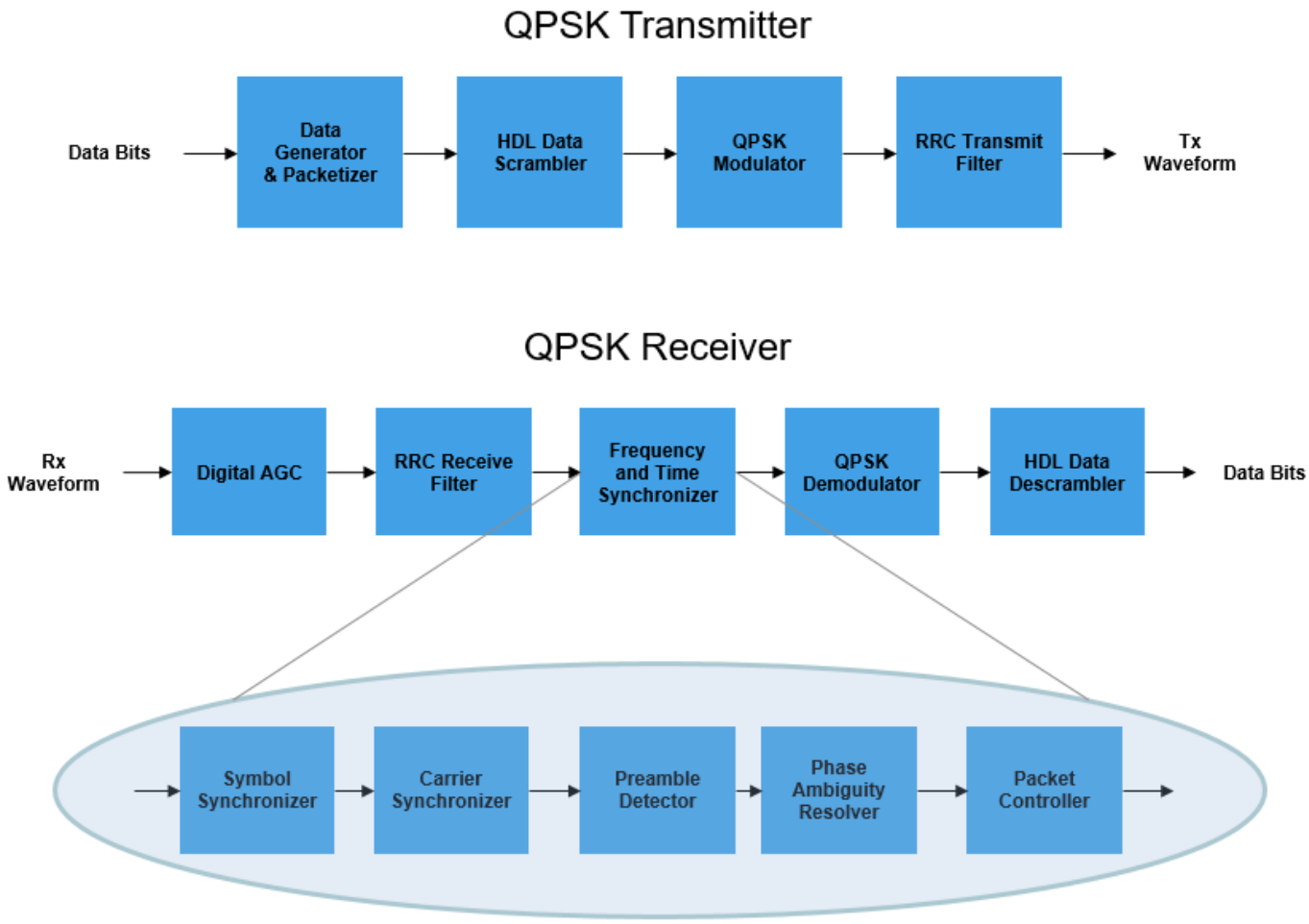
System Specifications

This section explains the specifications of the transmitter and receiver used in this example. The frame format is packet based. Each packet has a preamble of length 26 bits. Each bit of the 13 bit Barker sequence is repeated twice to generate a preamble sequence such that the same bit is modulated in the in phase and quadrature phase by the QPSK Modulator. The preamble sequence is followed by 2240 bits of payload data. The transmitter runs using a root raised cosine (RRC) pulse-shaping filter with a roll-off factor of 0.5, resulting in a bandwidth of 1.5 times the symbol rate and four samples per symbol (sample rate of four times the symbol rate). The RRC impulse response spans over four adjacent symbols. The bit rate is twice the symbol rate. The effective average bit rate is the bit rate times the frame efficiency. The frame efficiency is (2240/(2240+26)) = 0.9885.

The default symbol rate is set to 1.92 Mbaud, which results in a bandwidth of 1.5 times 1.92e6, which equals 2.88 MHz, and a sample rate of 4 times 1.92e6, which equals 7.68 Msps, bit rate of 2 times 1.92e6, which equals 3.84 Mbps. The effective average bit rate supported by this system is 0.9885 times 3.84e6, which equals 3.7959 Mbps. These specifications change with a change in the symbol rate.

Model Architecture

This section explains the high-level architecture of the QPSK transmitter and receiver as in the block diagram. The QPSK transmitter samples the input at a bit rate of twice the symbol rate. The Data Generator & Packetizer collects the data bits, generates the preamble bits, and forms the packet bits. The HDL Data Scrambler scrambles the data bits of each packet to increase bit transitions and avoid long running sequences of the same bit. The QPSK Modulator modulates the packet bits to generate QPSK symbols. The RRC Transmit Filter upsamples and pulse-shapes the QPSK symbols to generate the Tx Waveform at a sample rate of four times that of the symbol rate. The QPSK receiver samples the input at the transmission rate. The Digital AGC performs gain control to the desired amplitude level of the received waveform. The RRC Receive Filter performs matched filtering on the AGC output. The Frequency and Time Synchronizer performs synchronization operations and generates QPSK symbols for each packet. The QPSK Demodulator demodulates the QPSK symbols to generate packet bits. The HDL Data Descrambler descrambles the packet data bits that stream out of the receiver.



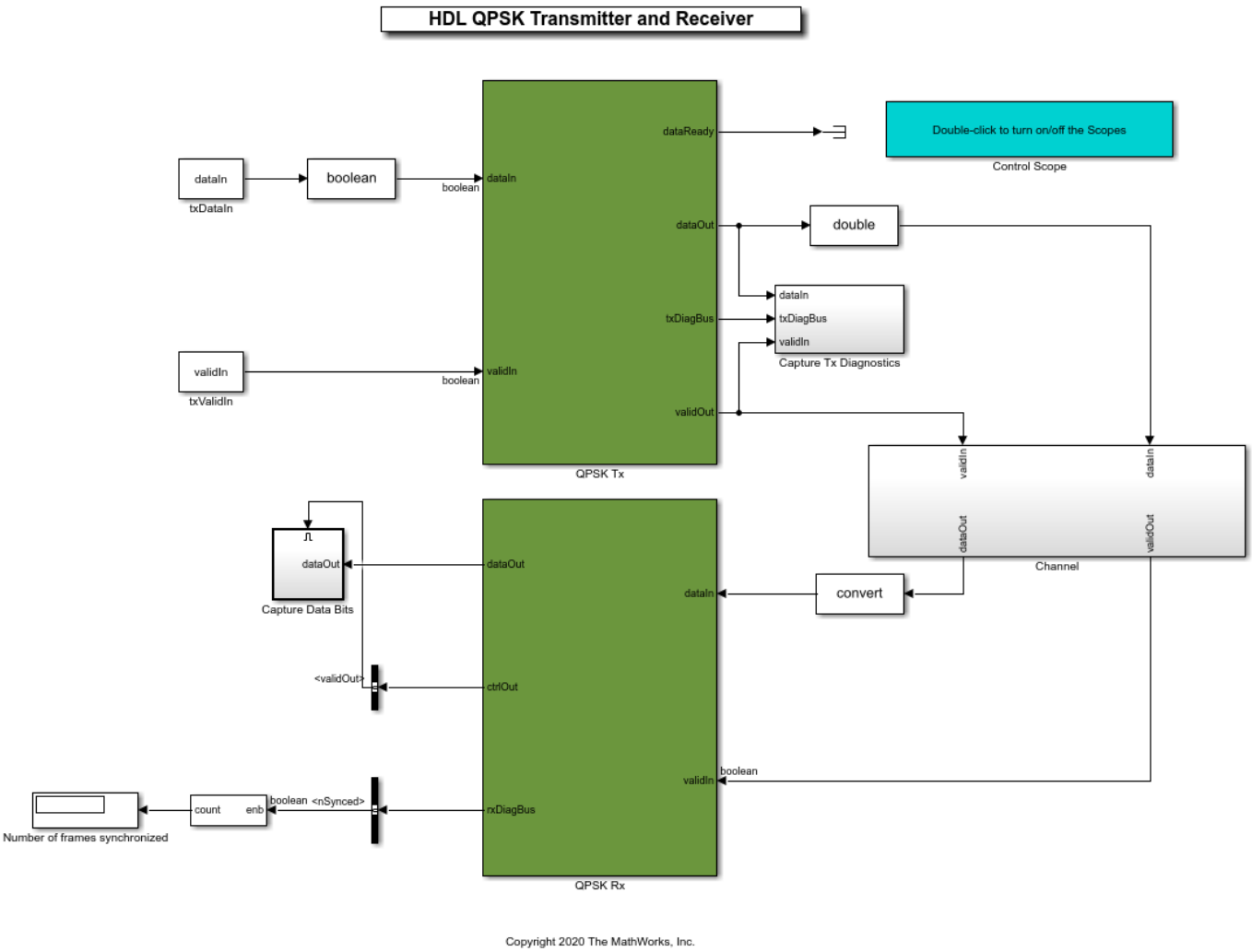
File Structure

One Simulink model and three MATLAB® files construct this example.

- `commhdlQPSKTxRx.slx` — Top-level Simulink model
- `commhdlQPSKTxRxParameters.m` — Generates parameters for QPSK Tx and QPSK Rx required for initialization
- `commhdlQPSKTxRxModelInit.m` — Initializes the model `commhdlQPSKTxRx.slx`
- `generateHelloworldMsgBits.m` — Generates "Hello world *xxx* " message bits. *xxx* refers to any value from 000 to 100

System Interface

This figure shows the top-level model of the QPSK transmitter and receiver system.



Transmitter Inputs

- **dataIn** — Input data, specified as a Boolean scalar.
- **validIn** — Control signal to validate the **dataIn** , specified as a Boolean scalar.

Transmitter Outputs

- **dataOut** — Output transmitted waveform, returned as 16-bit complex data at a sample rate four times that of the symbol rate.
- **validOut** — Control signal to validate the **dataOut** , returned as a Boolean scalar.
- **txDiagBus** — Status signal with diagnostic outputs, returned as a Bus signal.
- **dataReady** — Signal to indicate a ready for the input signals, returned as a Boolean scalar.

The transmitter enables the **dataReady** signal to indicate that it is ready to accept input bits. The transmitter constructs a packet after it accepts all the data bits corresponding to that packet. If all the data bits corresponding to that packet are not received, the transmitter generates dummy packets. For a dummy packet, the Barker sequence is not used for the preamble and the data bits are generated randomly internally. As long as the input bit rate is less than or equal to the effective bit

rate, the **dataReady** signal remains high so that the input does not get any back pressure from **dataReady**.

Receiver Inputs

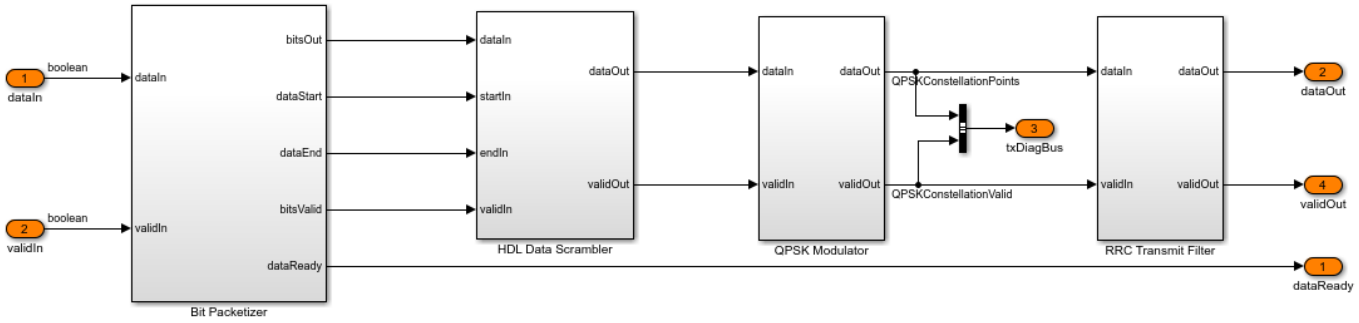
- **dataIn** — Input data, specified as a 16-bit complex data with sample rate as the transmitter output.
- **validIn** — Control signal to validate the **dataIn**, specified as a Boolean scalar.

Receiver Outputs

- **dataOut** — Decoded output data bits, returned as a Boolean scalar.
- **ctrlOut** — Bus signal with start, end, and valid signals, returned as a bus signal.
- **rxDiagBus** — Status signal with diagnostic outputs, returned as a bus signal.

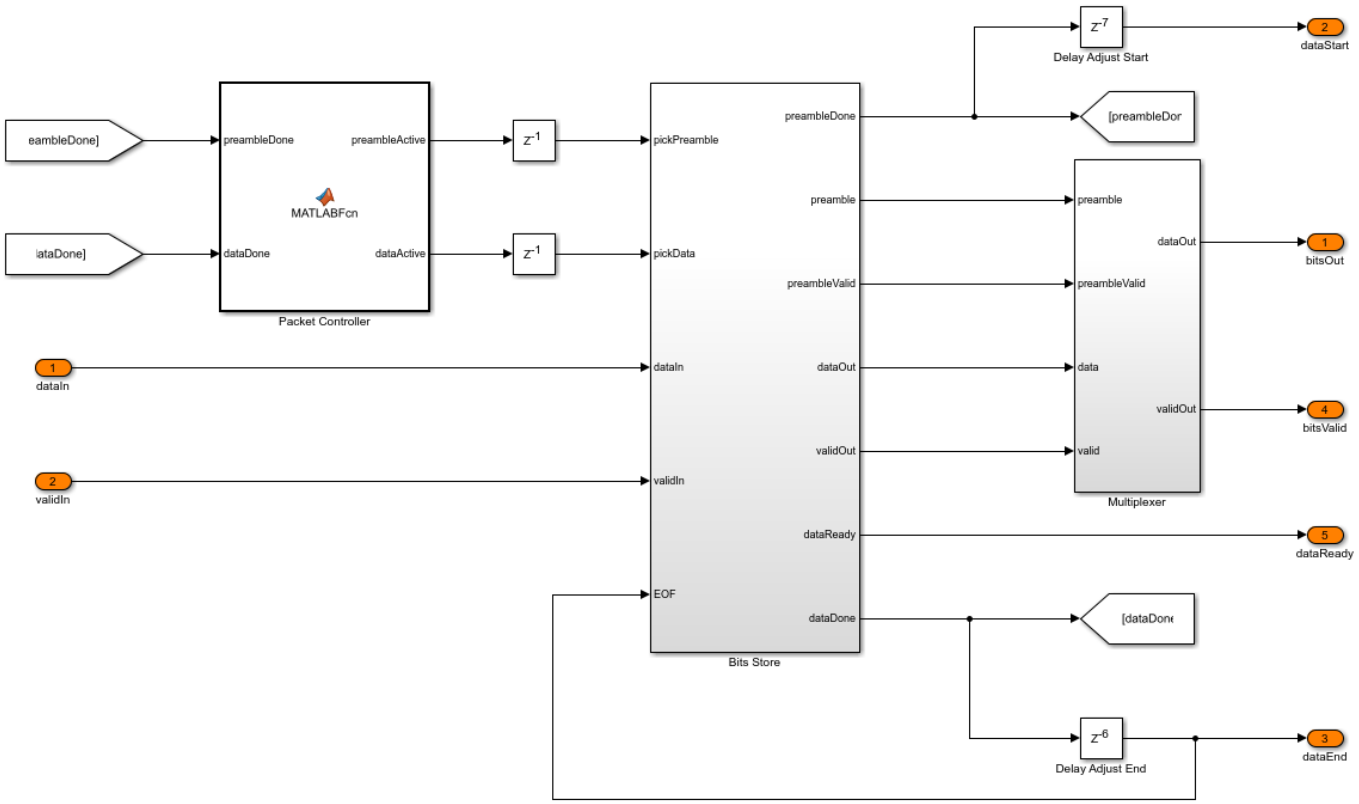
Transmitter Structure

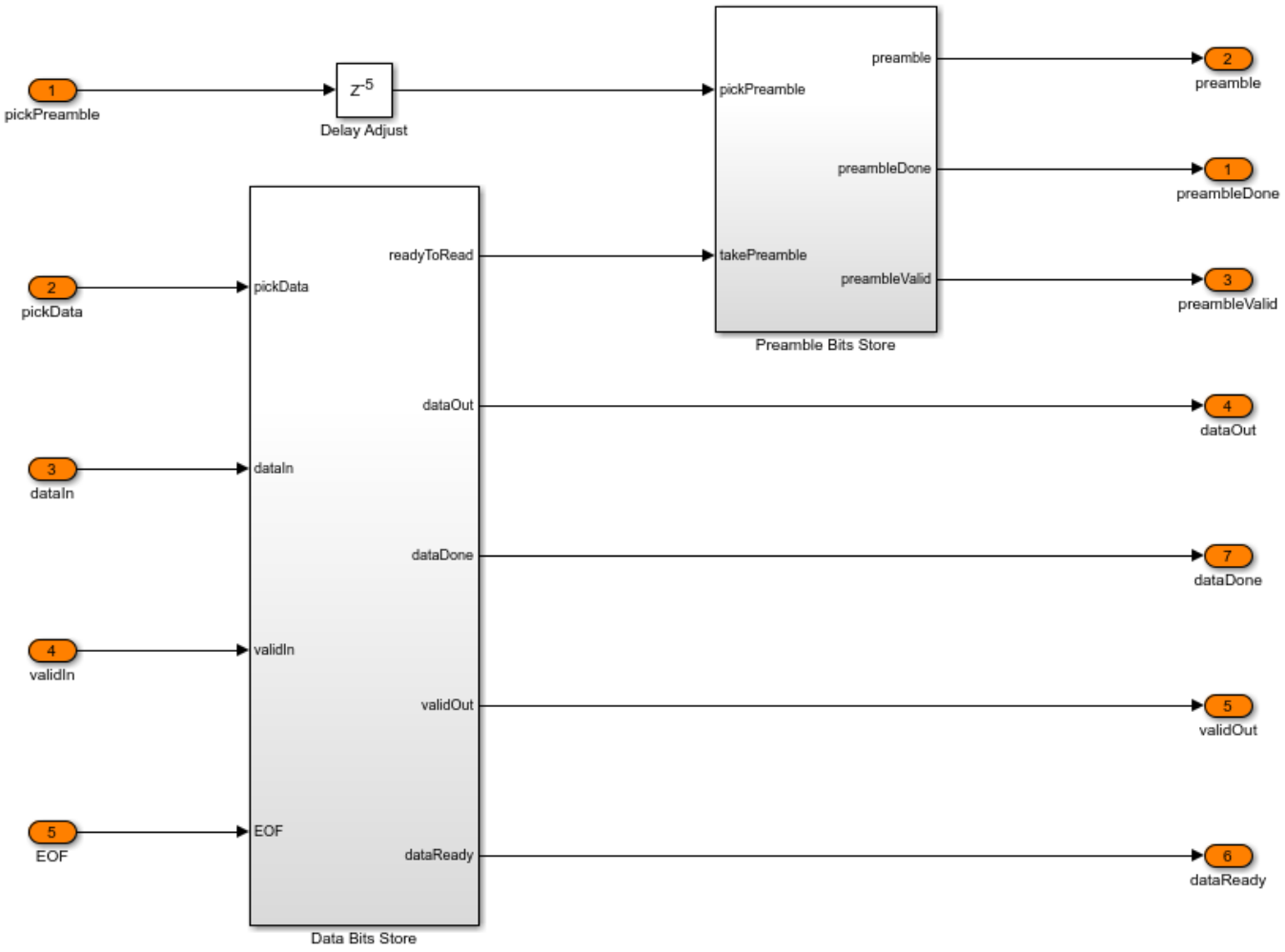
This figure shows the top-level model of the QPSK Tx subsystem.



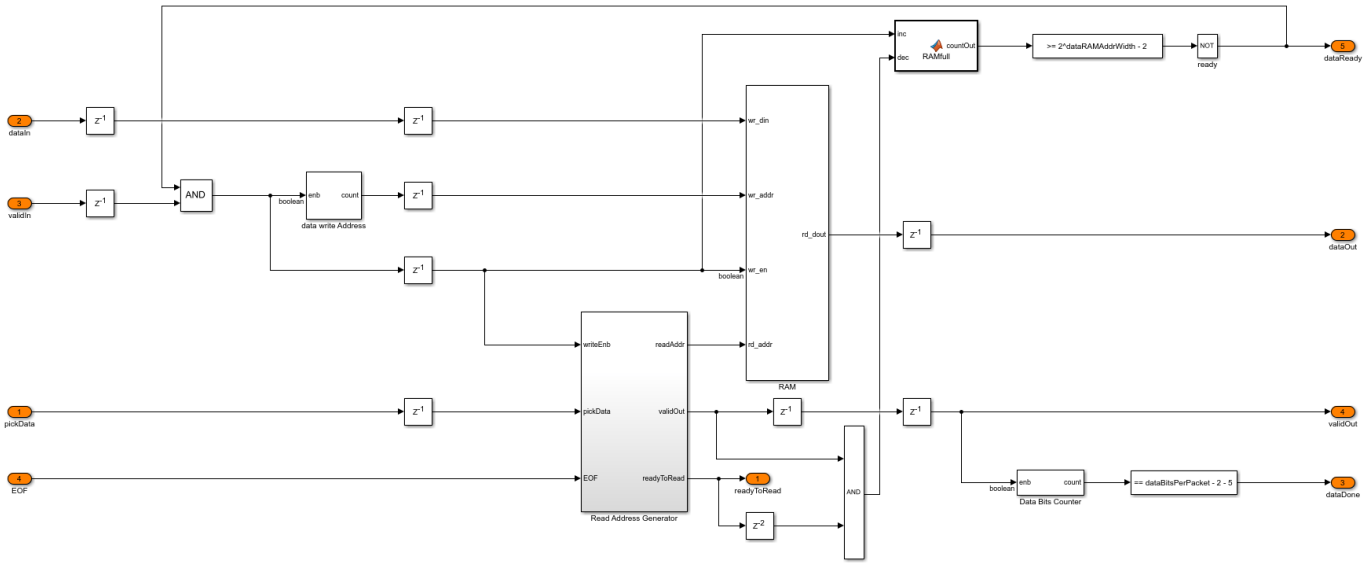
Bit Packetizer

The Bit Packetizer subsystem consists of a Packet Controller MATLAB function, a Bits Store, and a Multiplexer subsystem. The preamble sequence is stored in an look up table (LUT) inside the Preamble Bits Store subsystem. The data bits stream into the Bits Store subsystem and are stored in a RAM inside the Data Bits Store subsystem. The Packet Controller MATLAB function reads the preamble sequence followed by the data bits stored in the RAM for each packet. The Multiplexer subsystem streamlines the preamble bits and the data bits.


eambleDone]
ataDone]
preambleDone
preambleActive
dataDone
dataActive
MATLABFcn
Packet Controller
z-1
z-1
1
dataIn
2
validIn
pickPreamble
pickData
dataIn
validIn
EOF
preambleDone
preamble
preambleValid
dataOut
validOut
dataReady
dataDone
Bits Store
z-7
Delay Adjust Start
2
dataStart
[preambleDor
preamble
preambleValid
data
valid
dataOut
validOut
Multiplexer
1
bitsOut
4
bitsValid
5
dataReady
[dataDone
z-6
Delay Adjust End
3
dataEnd

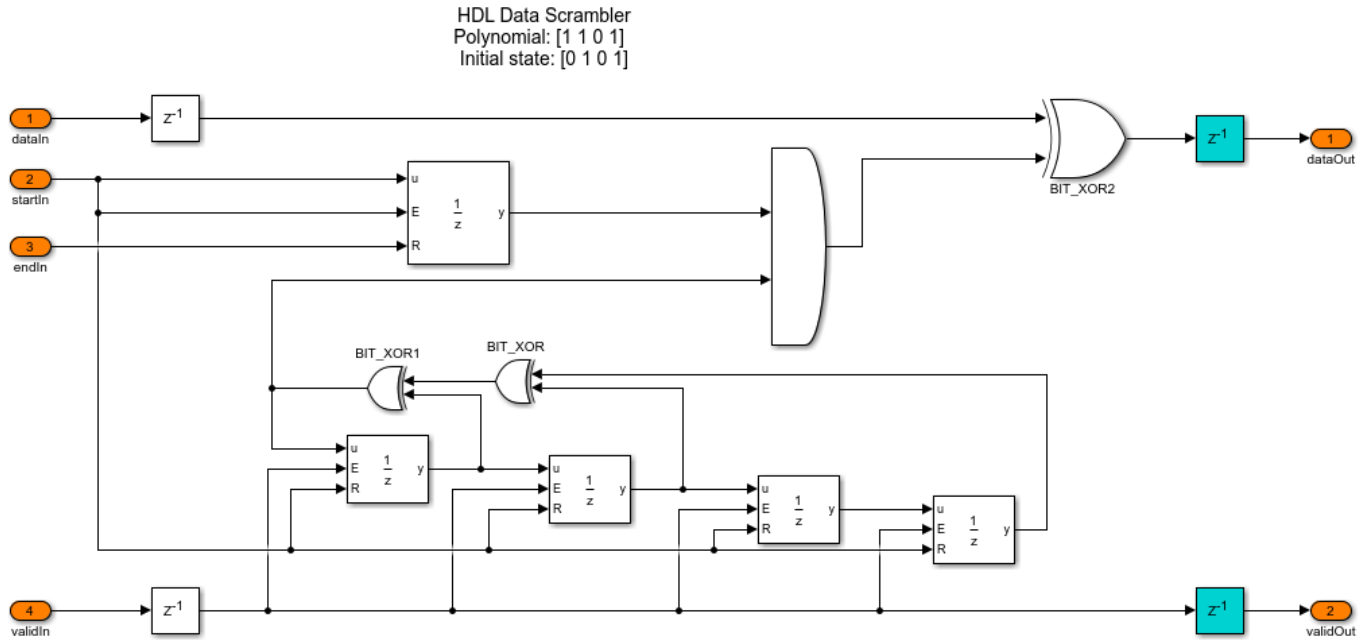


The Data Bits Store subsystem consists of a RAM that can store two packets. This RAM provides the flexibility to operate the transmitter with a discrete valid input. The Packet Controller MATLAB function reads data from the RAM only if the RAM contains a minimum of one packet of the data bits. The read and write logic is designed in such a way that the RAM does not overflows. When the RAM does not contain a minimum of one packet of the data bits, the transmitter generates a dummy packet. The Preamble does not use the Barker sequence for a dummy packet so that preamble detection does not detect it.



HDL Data Scrambler

The HDL Data Scrambler subsystem scrambles the data bits in each packet by using the control signals generated by the Bits Generator subsystem.

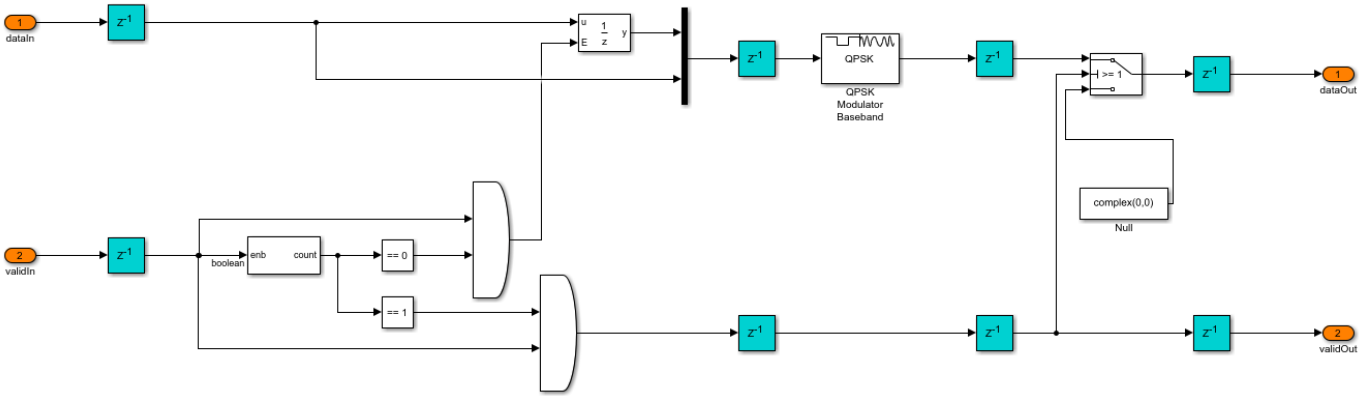


QPSK Modulator

The QPSK Modulator subsystem uses the QPSK Modulator Baseband block to modulate the preamble and data bits to generate QPSK symbols. It uses a gray mapping as described in this table.

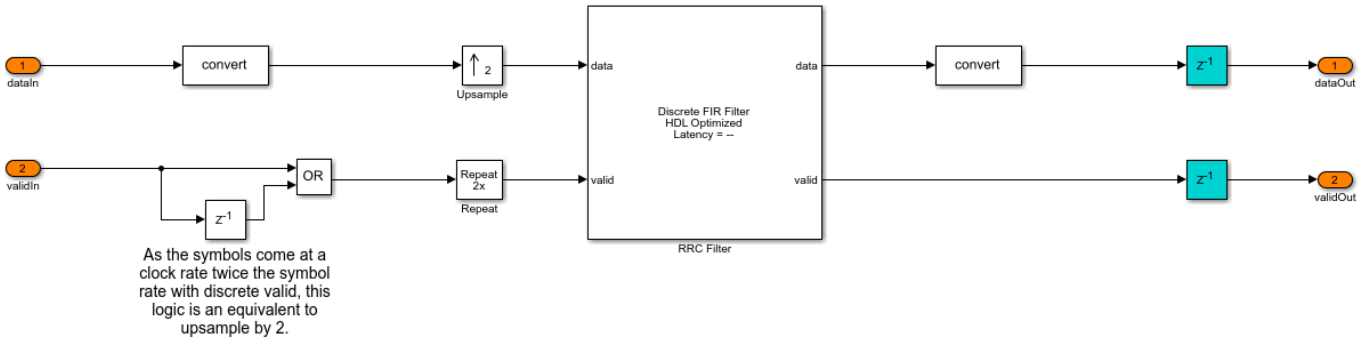
Bits	Mapping

```
00     0.70711+0.70711i
01     -0.70711+0.70711i
11     -0.70711-0.70711i
10     0.70711-0.70711i
```



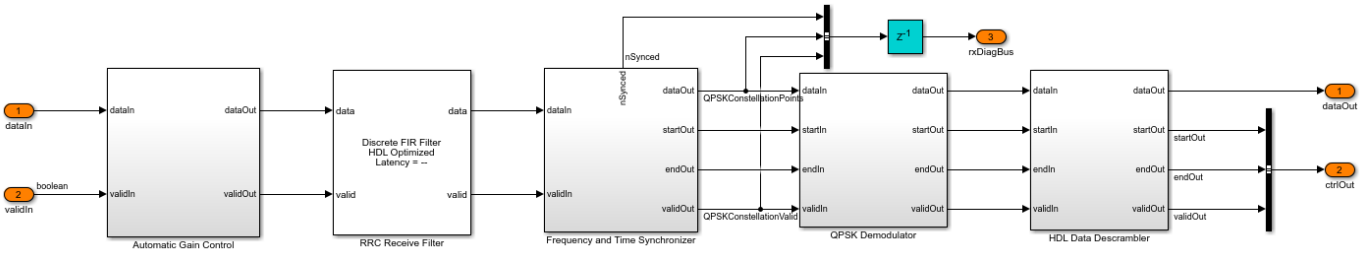
RRC Transmit Filter

The RRC Transmit Filter subsystem upsamples the input by a factor of four and uses the Discrete FIR Filter HDL Optimized block with an RRC impulse response to pulse-shape the transmitter waveform. The receive filter in the QPSK Receiver forms a matched filter to this transmit filter.



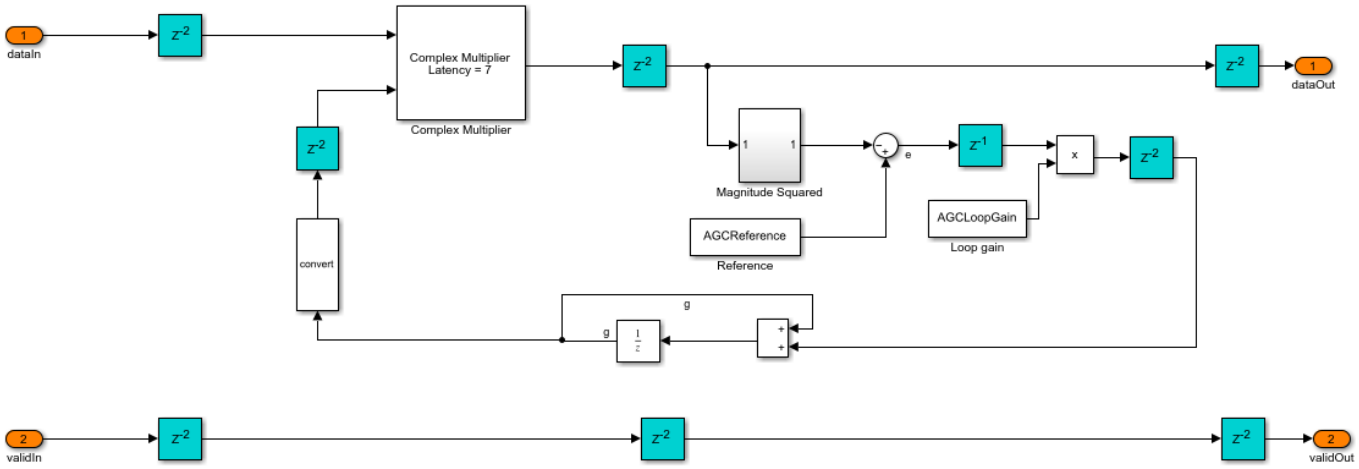
Receiver Structure

This figure shows the top-level model of the QPSK Rx subsystem.



Automatic Gain Control

As the input signal amplitude affects the symbol and carrier synchronizer PLL performance, the Automatic Gain Control subsystem is placed ahead of them. The magnitude squared output is compared with the AGC reference to generate an amplitude error. This error is multiplied with the loop gain and passed through an integrator to calculate the required gain. The resulted gain is multiplied with the AGC input to generate the AGC output. For more information, see Chapter 9.5 of [1].

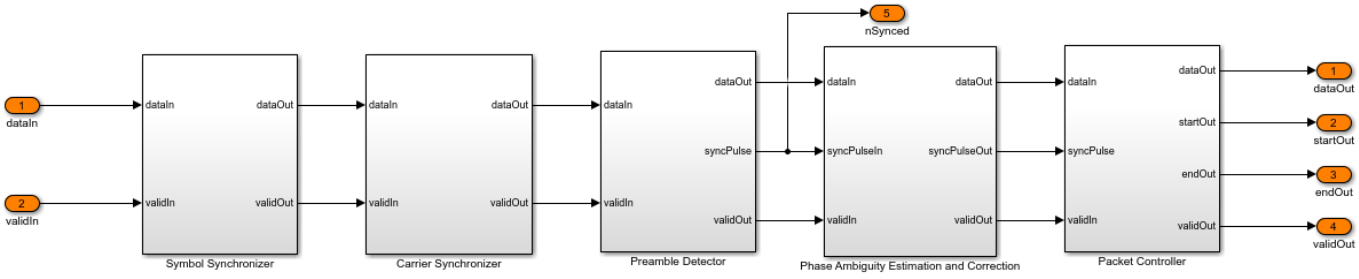


RRC Receive Filter

The RRC Receive Filter is a Discrete FIR Filter HDL Optimized block with matched filter coefficients of the filter used for pulse-shaping in the transmitter. The RRC matched filtering generates an RC pulse-shaped waveform, which has zero ISI characteristics at maximum eye opening in the eye diagram of the waveform. Also, the matched filtering process maximizes the signal to noise power ratio (SNR) of the filter output.

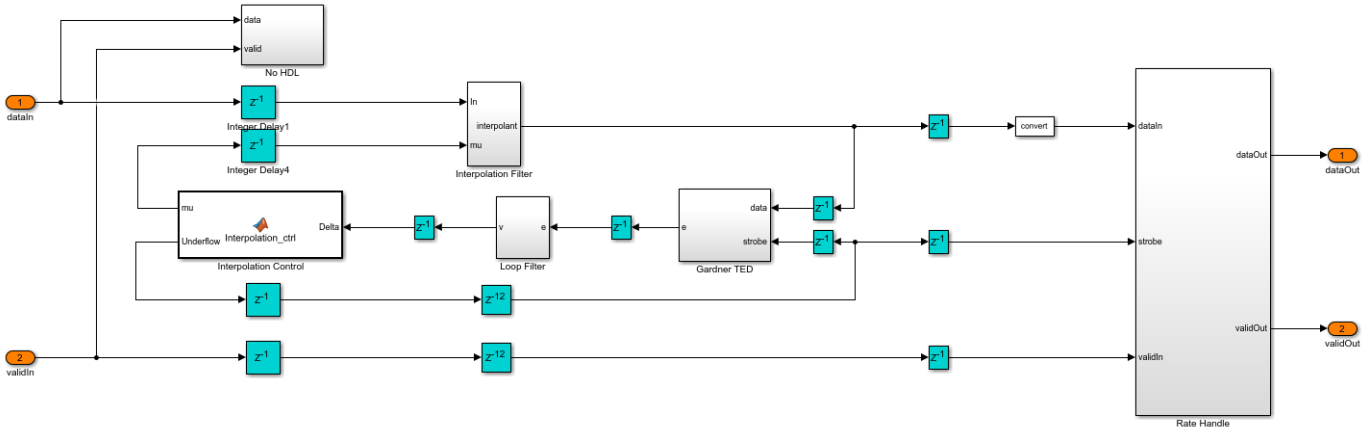
Frequency and Time Synchronizer

The Frequency and Time Synchronizer subsystem performs symbol synchronization, carrier synchronization, and preamble detection for packet synchronization. It also estimates and resolves the phase ambiguity that is left uncorrected in carrier synchronization.

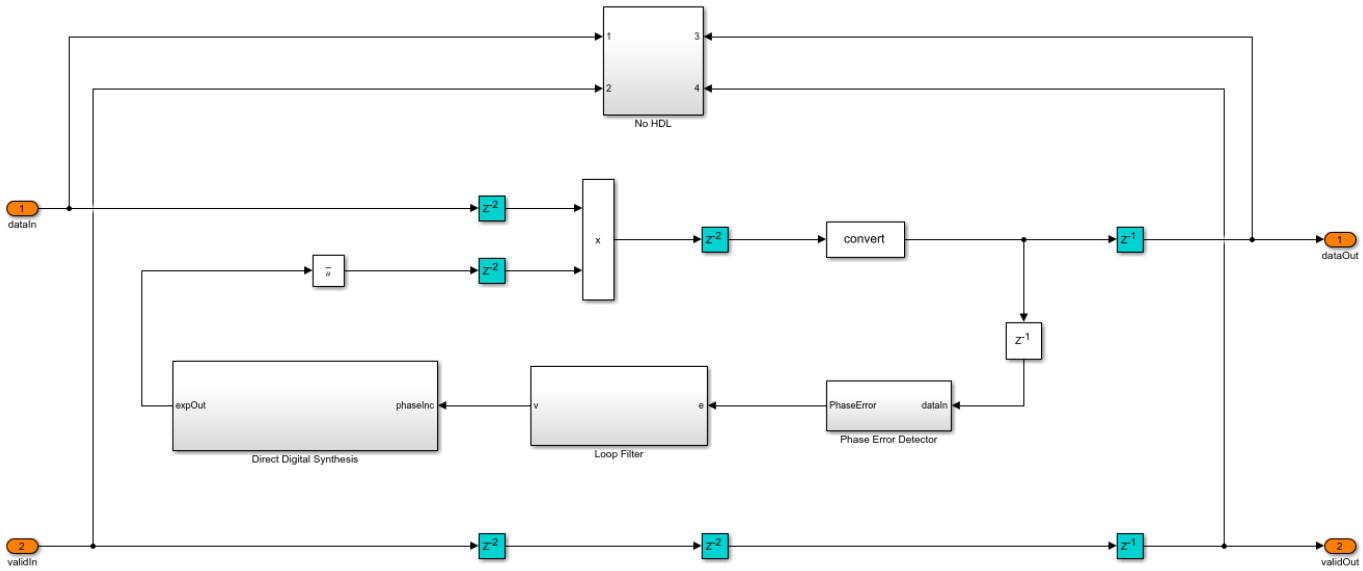


The Symbol Synchronizer subsystem is a PLL-based implementation. It generates samples at the optimum time instant (maximum eye opening instant) as described in Chapter 8.5 of [1]. The subsystem generates one output sample for every four input samples. The Interpolation Filter subsystem implements a piecewise parabolic interpolator with a hardware resource efficient farrow structure as described in Chapter 8.4.2, and the farrow coefficients are tabulated in Table 8.4.1 (the

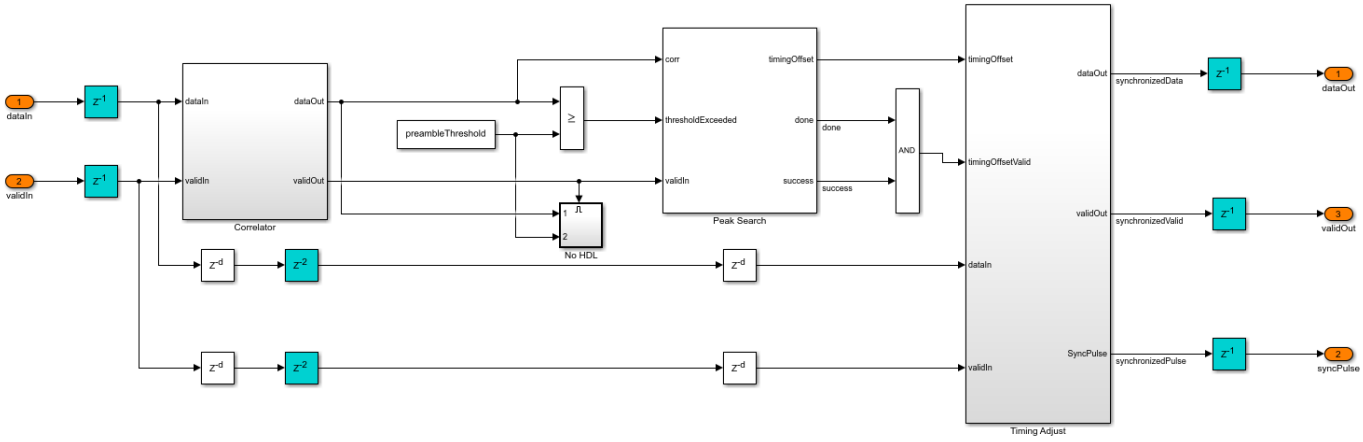
free parameter α of the coefficients is taken as 0.5) of [1]. This filter introduces fractional delays in the input waveform. The Gardner TED subsystem implements a Gardner timing error detector. The timing error detector is described in Chapter 8.4.1 of [1]. The loop filter filters the timing error and the timing error is passed on to the Interpolation Control MATLAB function block. This block implements a mod-1 decrementing counter to calculate fractional delays based on the loop filtered timing error as described in Chapter 8.4.3 of [1] to generate interpolants at optimum sampling instants. The Rate Handle subsystem selects the required interpolant indicated by the strobe. This sample corresponds to the maximum eye opening of the eye diagram before symbol synchronization.



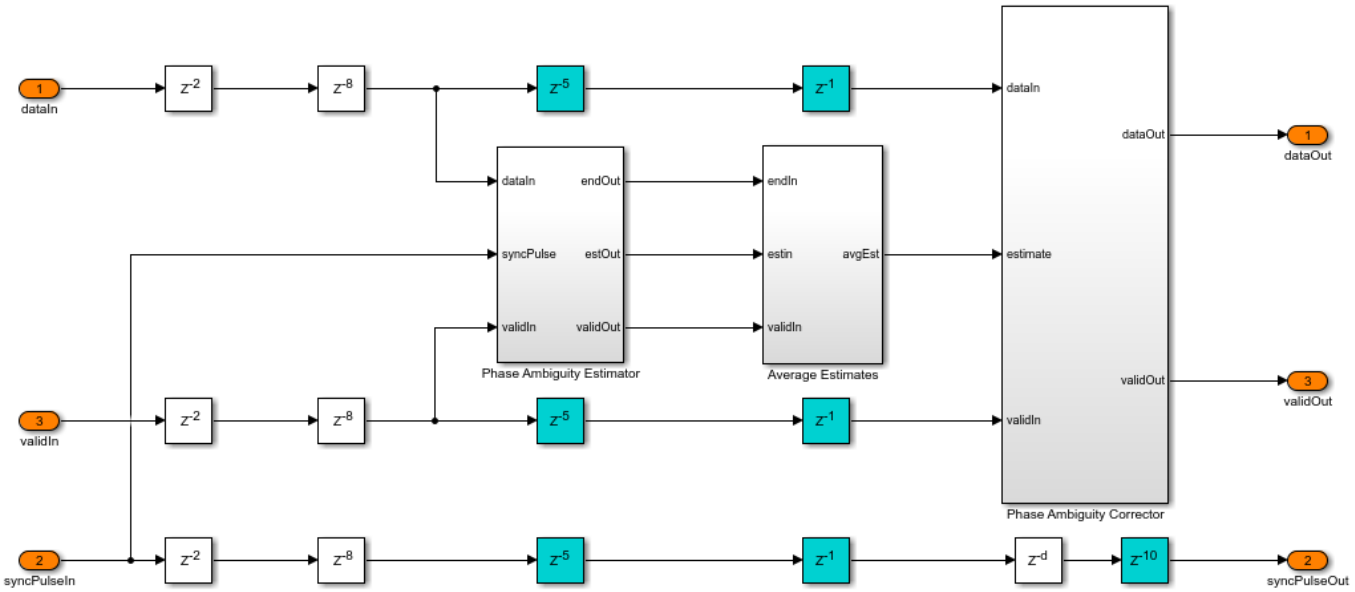
The Carrier Synchronizer subsystem is a TYPE II PLL with a sinusoidal phase error detector, which operates at a 45 degrees operating point. The phase error detector is described in Chapter 7.2.2, and the design equations are described in the Appendix C of [1]. A detailed analysis of TYPE II PLL with a zero operating point sinusoidal phase detector is described in Chapter 4 of [2]. The sign function of the phase detector in the real and imaginary parts converts all of the angles in the 4 quadrants into a first-quadrant angle (0 to 90 degrees), which creates an ambiguity of 90,180,270 degrees for second (90 to 180 degrees), third (-180 to -90 degrees) and fourth (-90 to 0 degrees) quadrant angles, respectively. The phase error is calculated as a deviation from the operating point (45 degrees) of the phase detector. The proportional plus integrator filter in the Loop Filter subsystem filters the phase error. The loop filter sets the normalized loop bandwidth (normalized by the sample rate) and the loop damping factor. The default normalized loop bandwidth is set to 0.005, and the default damping factor is set to 0.7071. The filtered error is given as a phase increment source to the Direct Digital Synthesis subsystem, which uses the NCO HDL Optimized block for complex exponential phase generation. The complex exponential phase is used to correct the frequency and phase of the input. A detailed analysis of direct digital synthesis is described in Chapter 9.2.2 of [1].



The Preamble Detector subsystem performs continuous correlation for the input with the Barker sequence. The correlation is implemented as convolution with the reversed Barker sequence as coefficients for the Discrete FIR Filter HDL Optimized block, and the magnitude of the correlated output is found using the Complex to Magnitude-Angle HDL Optimized block inside the Correlator subsystem. The magnitude of the correlation is compared with a threshold. The Peak Search subsystem begins searching for the maximum correlation peak that exceeded the threshold for every one frame duration and records the timing offset. The Timing Adjust subsystem synchronizes packet timing based on the timing offset to generate **syncPulse** signal, which indicates a packet synchronized sample to the subsequent subsystem.



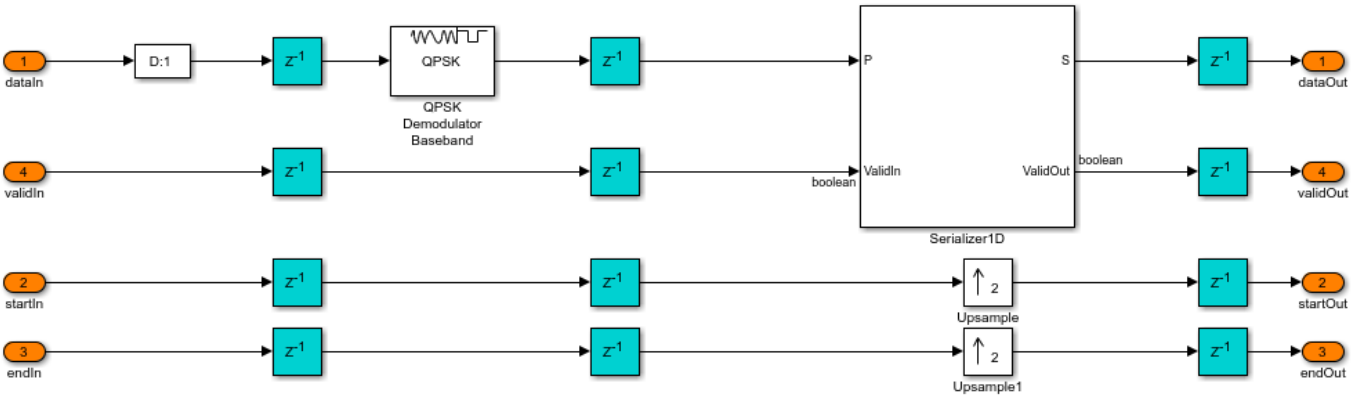
The Phase Ambiguity Estimation and Correction subsystem works based on the unique word method for phase ambiguity resolution described in Chapter 7.7.1 of [1]. This method uses the preamble sequence as the reference sequence. The reference sequence is conjugated and multiplied with the preamble sequence in the input, and the residual phase is extracted as the phase ambiguity estimate. This estimate is used to correct the ambiguity by rotating the constellation in the opposite direction of ambiguity.



The Packet Controller subsystem generates control signals for the packet boundaries.

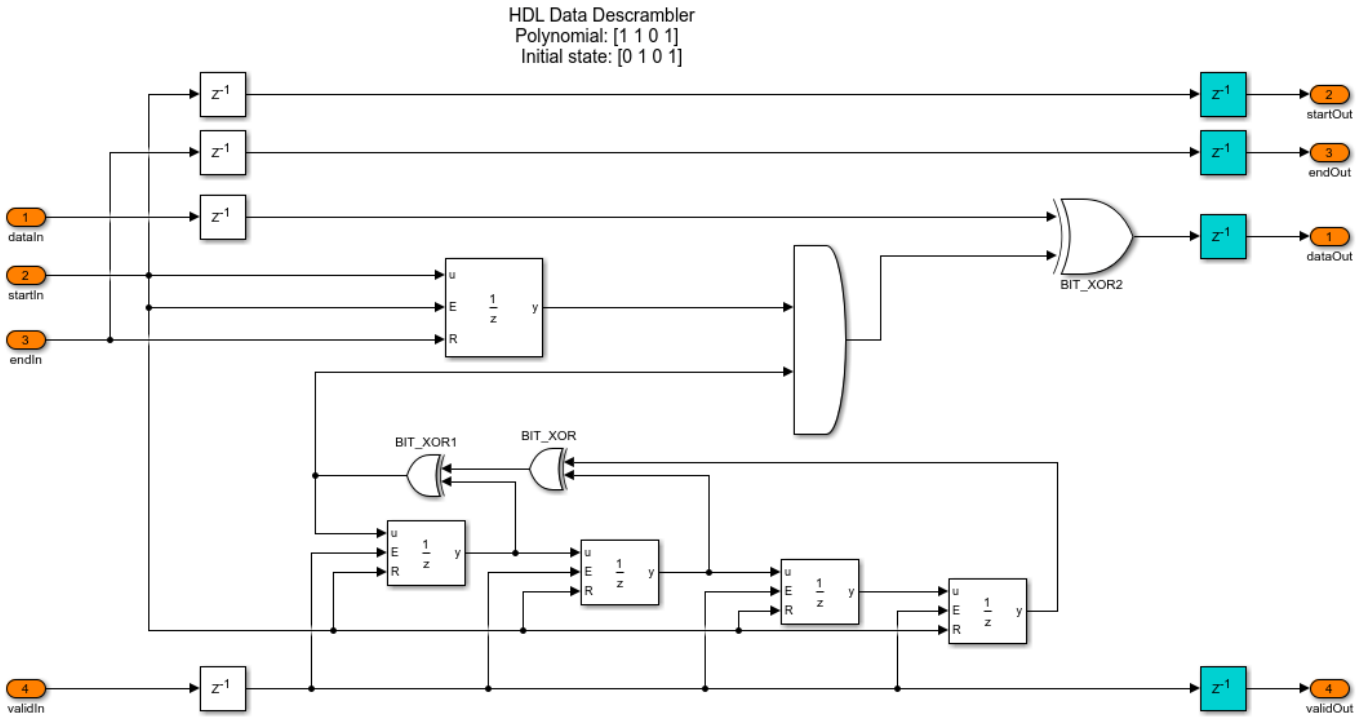
QPSK Demodulator

The QPSK Demodulator subsystem uses the QPSK Demodulator Baseband block to demodulate the packet synchronized symbols and generate bits.



HDL Data Descrambler

The HDL Data Descrambler subsystem descrambles the demodulated bits to generate the user bits.



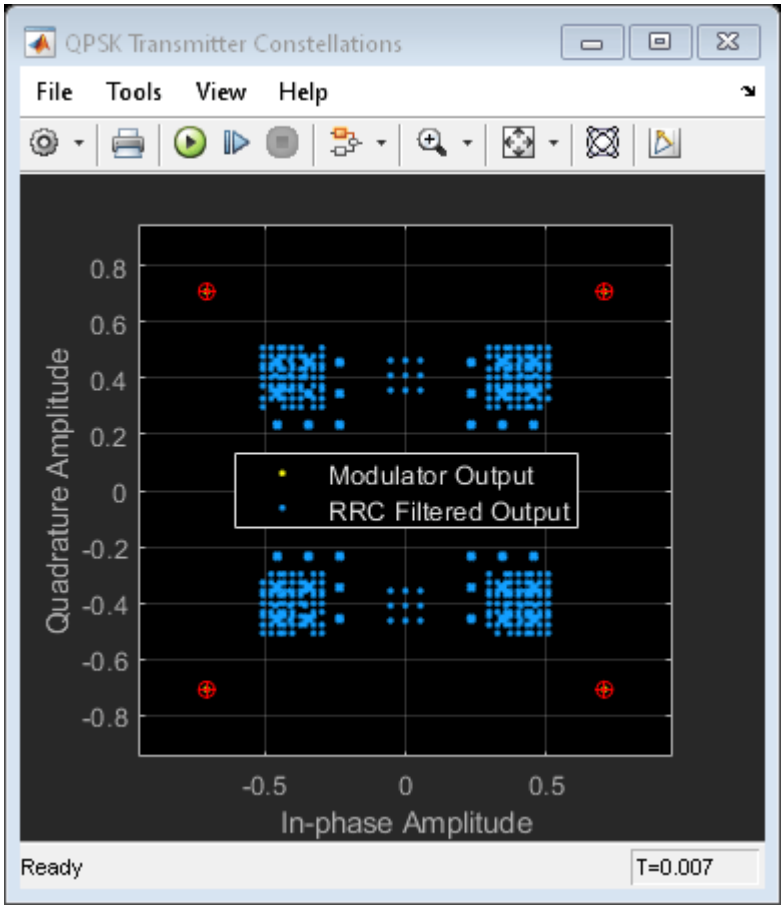
Run the Model

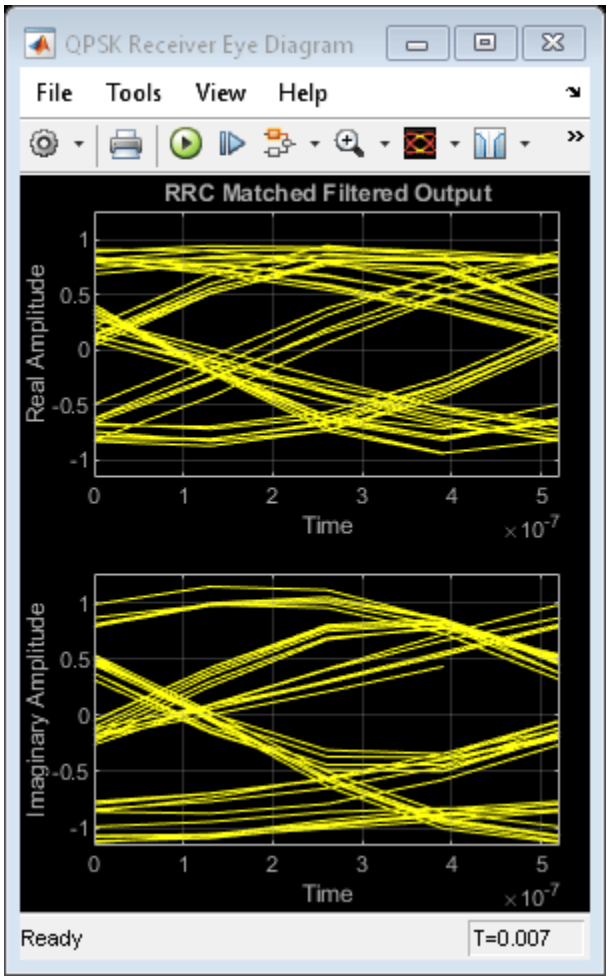
The `QPSKTxRxVerification.m` script describes a procedure to initialize, generate inputs, run, and verify the `commhdlQPSKTxRx.slx` model by using the `commhdlQPSKTxRxModelInit.m` initialization script. You can assign custom data to the variables from the Custom Frame Configuration section in this script and run the script to run the model. This verification script generates a reference waveform within the script, compares the reference waveform with the transmitter output, and compares the transmitted bits with the decoded user bits.

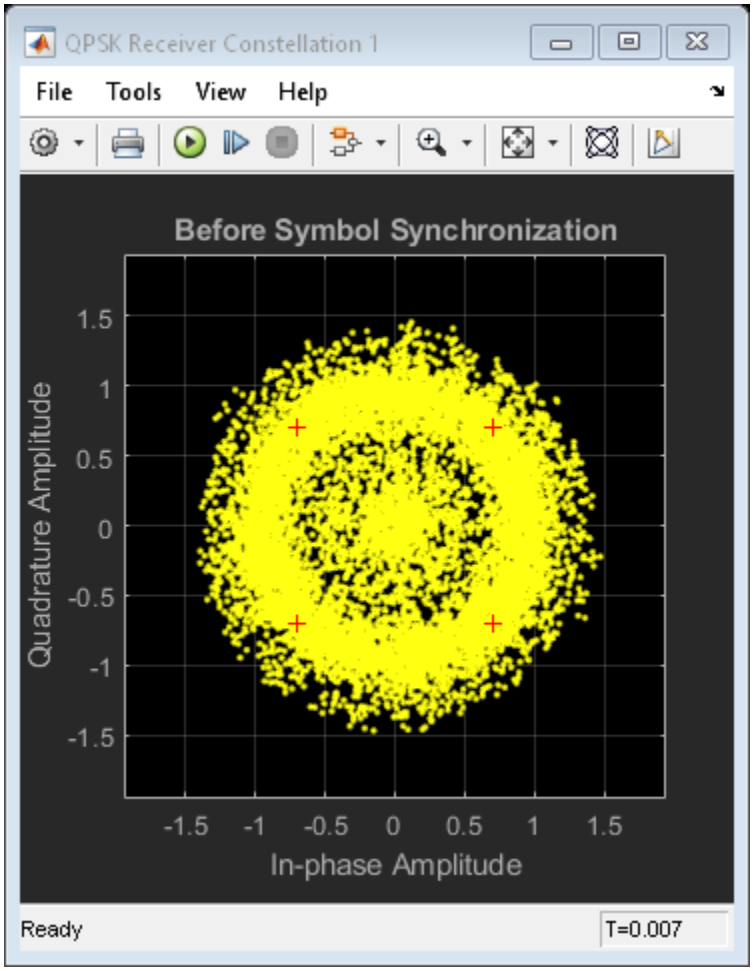
Verification and Results

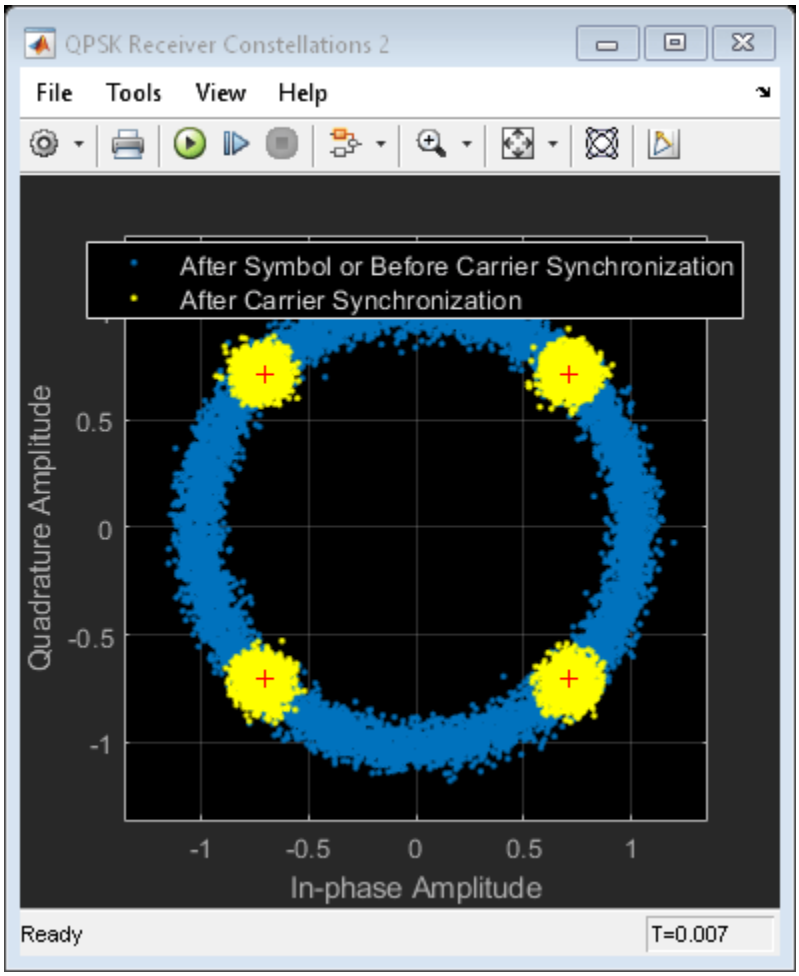
Run `QPSKTxRxVerification.m` to run the model.

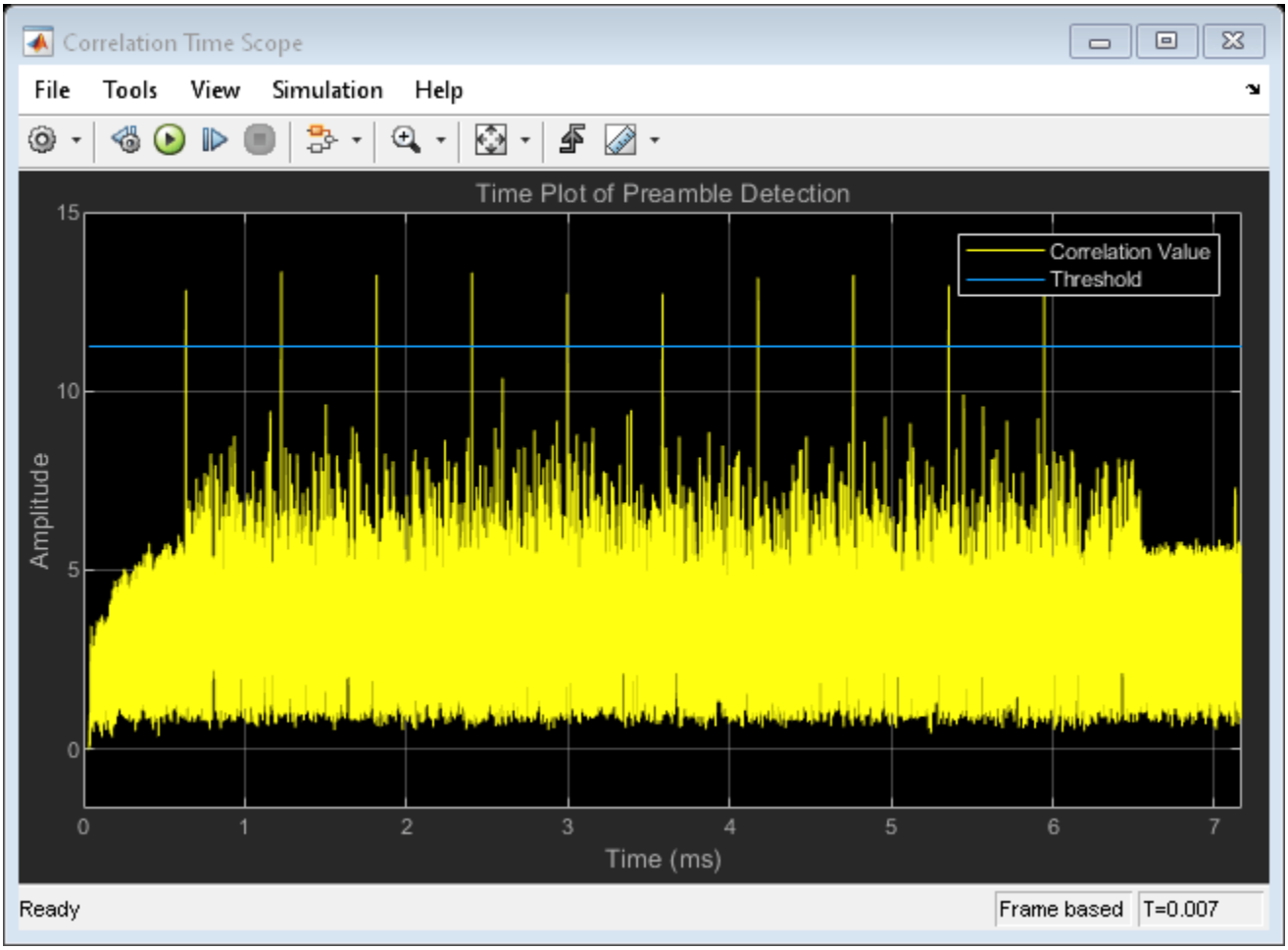
```
>> QPSKTxRxVerification


Tx:
Maximum QPSK Tx symbol error: Real:1.4496e-05 Imaginary:1.4496e-05

Maximum QPSK Tx waveform error: Real:7.8708e-05 Imaginary:7.8708e-05

Rx:
Number of packets missed = 0 out of 10

Number of packets false detected = 0 out of 10

Number of bits errored = 0 out of 20160
```


QPSK Transmitter Constellations
File
Tools
View
Help
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
Quadrature Amplitude
Modulator Output
RRC Filtered Output
-0.5
0
0.5
In-phase Amplitude
Ready
T=0.007


QPSK Receiver Eye Diagram
File
Tools
View
Help
RRC Matched Filtered Output
Real Amplitude
1
0.5
0
-0.5
-1
0
1
2
3
4
5
Time
×10^-7
Imaginary Amplitude
1
0.5
0
-0.5
-1
0
1
2
3
4
5
Time
×10^-7
Ready
T=0.007


QPSK Receiver Constellation 1
File
Tools
View
Help
Before Symbol Synchronization
Quadrature Amplitude
In-phase Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
Ready
T=0.007


QPSK Receiver Constellations 2
File
Tools
View
Help
After Symbol or Before Carrier Synchronization
After Carrier Synchronization
Quadrature Amplitude
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=0.007



HDL Code Generation

Pipeline registers (shown in cyan) are added throughout the model to make sure the transmitter and receiver subsystems do not have a long critical path.

To check and generate the HDL code referenced in this example, you must have the HDL Coder™ product.

To generate the HDL code for transmitter and receiver subsystems, update the models and use the following command:

```
makehdl('commhdlQPSKTxRx/QPSK Tx')  and  makehdl('commhdlQPSKTxRx/QPSK Rx')
```

To generate test bench, use the following command:

```
makehdltb('commhdlQPSKTxRx/QPSK Tx')  and  makehdltb('commhdlQPSKTxRx/QPSK Rx')
```

Test bench generation time depends on the simulation time.

The resulting HDL code is synthesized for the Xilinx® Zynq®-7000 ZC706 evaluation board. The post place and route resource utilization is shown in this table. The maximum frequency of operation is 280 MHz for the transmitter and 215 MHz for the receiver.

Resources	Tx Usage	Rx Usage
Slice Registers	318	6231
Slice LUT	137	4506
RAMB36	0	8
RAMB18	1	0
DSP48	18	88

Further Exploration

You can modify the channel conditions by tuning the variables listed in this table in the `QPSKTxRxVerification.m` script and then running the script. The script applies the channel conditions and runs the model.

Variable Name	Description
dataBits	Data bits to the transmitter
Rsym	Symbol rate specified in symbols per second
fractionalTimingOffset	Normalized timing phase offset specified in the range >= 0 and < 1
timingFrequencyOffset	Timing frequency offset specified in PPM
EbN0dB	Energy per information bit to single sided noise power spectral den
CFO	Carrier frequency offset specified in Hz
CPO	Carrier phase offset specified in degrees

References

1. Michael Rice, *Digital Communications - A Discrete-Time Approach*, Prentice Hall, April 2008.

2. Floyd M.Gardner, *Phaselock Techniques*, Third Edition, John Wiley & Sons, Inc., 2005

See Also

Blocks

QPSK Modulator Baseband | QPSK Demodulator Baseband

Communications Toolbox Library for ZigBee and UWB - Featured Examples

UWB Ranging Using IEEE 802.15.4z

This example shows how to estimate distance between two devices as per the IEEE® 802.15.4z™ standard [2] on page 7-0 by using features in the Communications Toolbox™ Library for ZigBee® and UWB add-on.

Overview

The IEEE 802.15.4z amendment [2] on page 7-0 of the IEEE® 802.15.4 standard [1] on page 7-0 specifies the MAC and PHY layers, and associated ranging and localization using ultra wideband (UWB) communication. The very short pulse durations of UWB allow a finer granularity in the time domain and therefore more accurate estimates in the spatial domain.

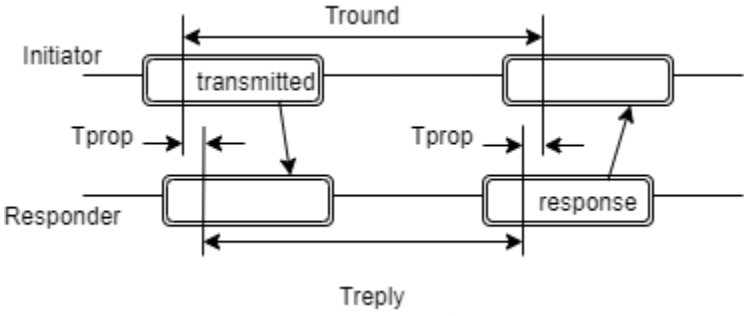
The key ranging and localization functionality of the 802.15.4z amendment includes three MAC-level techniques:

- Single-Sided Two-Way Ranging (SS-TWR) - One device estimates the distance between two devices by using frame transmission in both directions of a wireless 802.15.4z link.
- Double-Sided Two-Way Ranging (DS-TWR) - Both devices estimate the distance between the two devices by using frame transmission in both directions of a wireless 802.15.4z link.
- One-Way Ranging / Time-Difference of Arrival (OWR/TDOA) - Network-assisted localization whereby one device communicates with a set of synchronized nodes to estimate the position of the device. This technique is demonstrated in the “UWB Localization Using IEEE 802.15.4z” on page 7-9 example.

This example demonstrates the SS-TWR technique by using PHY frames that are compatible with the IEEE 802.15.4 standard [1] on page 7-0 and the IEEE 802.15.4z amendment [2] on page 7-0 . For more information on generating PHY-level IEEE 802.15.4z waveforms, see the “HRP UWB IEEE 802.15.4a/z Waveform Generation” on page 7-23 example.

Single-Sided Two-Way Ranging (SS-TWR)

Two-way ranging involves frame transmission in both directions of a wireless 802.15.4z link. Single-sided ranging means that only one of the two devices estimates the distance between them.



Each frame is timed at its ranging marker (RMARKER), which is the time of the first symbol following the start-of-frame delimiter (SFD). For more information on the fields in the transmitted frame, see the “HRP UWB IEEE 802.15.4a/z Waveform Generation” on page 7-23 example. The ranging responder device, transmits the response frame after a certain reply time (Treply). The ranging initiator device computes the round-trip time (Tround) as the time-distance between the RMARKERs of the transmitted and the response frames. Treply is communicated from the ranging responder

device to the ranging initiator device, so that the latter estimates the propagation time (Tprop) as Tprop = (Tround - Treply)/2.

The IEEE 802.15.4z amendment [2] on page 7-0 specifies multiple possibilities for sharing Treply:

- Communication of Treply from the responder to the initiator is deferred, and performed with another message following the response frame.
- Embed Treply in the response frame.
- Set Treply to a fixed value known between the initiator and the responder.

This example considers the fixed reply time scenario between the two devices.

IEEE 802.15.4 [1] on page 7-0 specifies that the exchanged frames must be a Data frame and its acknowledgement. The IEEE 802.15.4z amendment [2] on page 7-0 relaxes this specification and allows the ranging measurement to be performed over any pair of transmitted and response frames. However, for the fixed reply time scenario, the 802.15.4z amendment specifies exchange of scrambled timestamp sequence packet configuration option three (SP3) frames. SP3 frames contain a scrambled timestamp sequence (STS) and no PHY header (PHR) or payload.

This example focuses on the basic ranging exchange without demonstrating the preceding set-up and following finish-up activities associated with the ranging procedure.

Setup

Confirm installation of the Communications Toolbox™ Library for ZigBee® and UWB add-on.

```
% Ensure that the ZigBee/UWB add-on is installed:
commSupportPackageCheck('ZIGBEE');
```

Determine the actual distance and Tprop, and initialize visualizations. Configure a `timescope` object to plot the initiator and responder signals.

```
c = physconst('LightSpeed');    % Speed of light (m/s)
actualDistance = 5;             % In meters
actualTprop = actualDistance/c; % In seconds
SNR = 30;                       % Signal-to-Noise ratio
symbolrate = 499.2e6;           % Symbol rate for HRP PHY
sps = 10;                       % Samples per symbol
ts = timescope( ...
    SampleRate=sps*symbolrate, ...
    ChannelNames={'Initiator','Responder'}, ...
    LayoutDimensions=[2 1], ...
    Name='SS-TWR');
ts.YLimits = [-0.25 0.25];
ts.ActiveDisplay = 2;
ts.YLimits = [-0.25 0.25];
```

Transmitted Frame

Transmission from Initiator

Generate the waveform containing SP3 PHY frames (with no MAC frame/PSDU) to be transmitted between the devices. Register the transmitted frame on the timeline of the initiator.

```
sp3Config = lrwpanHRPConfig( ...
    Mode='HPRF', ...
```

```
    STSPacketConfiguration=3, ...
    PSDULength=0, ...
    Ranging=true);
sp3Wave = lrwpanWaveformGenerator([],sp3Config);
[transmitFrame,responseFrame] = deal(sp3Wave);

% start initiator time at the start of transmission
initiatorView = transmitFrame;
```

Wireless Channel

Filter the transmission frame through an AWGN channel and add propagation delay. Then, update timeline for both link endpoints.

```
samplesToDelay = actualTprop*sp3Config.SampleRate;
receivedTransmitted = lclDelayWithNoise( ...
    transmitFrame,samplesToDelay,SNR);

initiatorView = [initiatorView; zeros(ceil(samplesToDelay),1)];
responderView = receivedTransmitted;
```

Reception at Responder

At the responder side, detect the preamble of the 802.15.4z PHY frame, and then process the transmitted frame. Preamble detection consists of determining the first instance of the preamble out of Nsync = PreambleDuration. Plot the initiator and responder views on a `timescope`.

```
ind = lrwpanHRPFieldIndices(sp3Config); % length (start/end) of each field

sp3Preamble = sp3Wave(1:ind.SYNC(end)/sp3Config.PreambleDuration);
preamPos = helperFindFirstHRPPreamble( ...
    receivedTransmitted,sp3Preamble,sp3Config);

ts(initiatorView,responderView);
```

Response Frame

Transmission from Responder

Set the Treply time to the length of three SP3 frames to specify when to transmit the response frame. Set the first and last RMARKER sample indices on the responder side to be the beginning of first post-SFD symbol and Treply samples later. After Treply samples, transmit the response frame from the responder device.

```
Treply = 3*length(sp3Wave); % in samples

% Find RMARKERs at responder side
frameStart = 1+preamPos-ind.SYNC(end)/sp3Config.PreambleDuration;
sfdEnd = frameStart + ind.SYNC(end) + diff(ind.SFD);
RMARKER_R1 = sfdEnd+1;
RMARKER_R2 = RMARKER_R1 + Treply;

% Transmit after Treply. Find how long the responder needs to remain idle.
idleResponderTime = Treply - diff(ind.STS)-1 - diff(ind.SHR)-1;
responderView = [responderView; zeros(idleResponderTime,1); responseFrame; zeros(ceil(samplesToDe

initiatorView = [initiatorView; zeros(idleResponderTime, 1)];
```

Wireless Channel

Filter the transmission frame through an AWGN channel and add propagation delay. Then, update timeline for both link endpoints.

```
receivedResponse = lclDelayWithNoise( ...
    responseFrame,samplesToDelay,SNR);
initiatorView = [initiatorView; receivedResponse];
```

Reception at Initiator

Back at the initiator side, detect the preamble of the 802.15.4z PHY frame, and then process the transmitted frame.

```
txFrameEnd = ind.STS(end);
preamPos = helperFindFirstHRPPreamble( ...
    initiatorView(txFrameEnd+1:end),sp3Preamble,sp3Config);
```

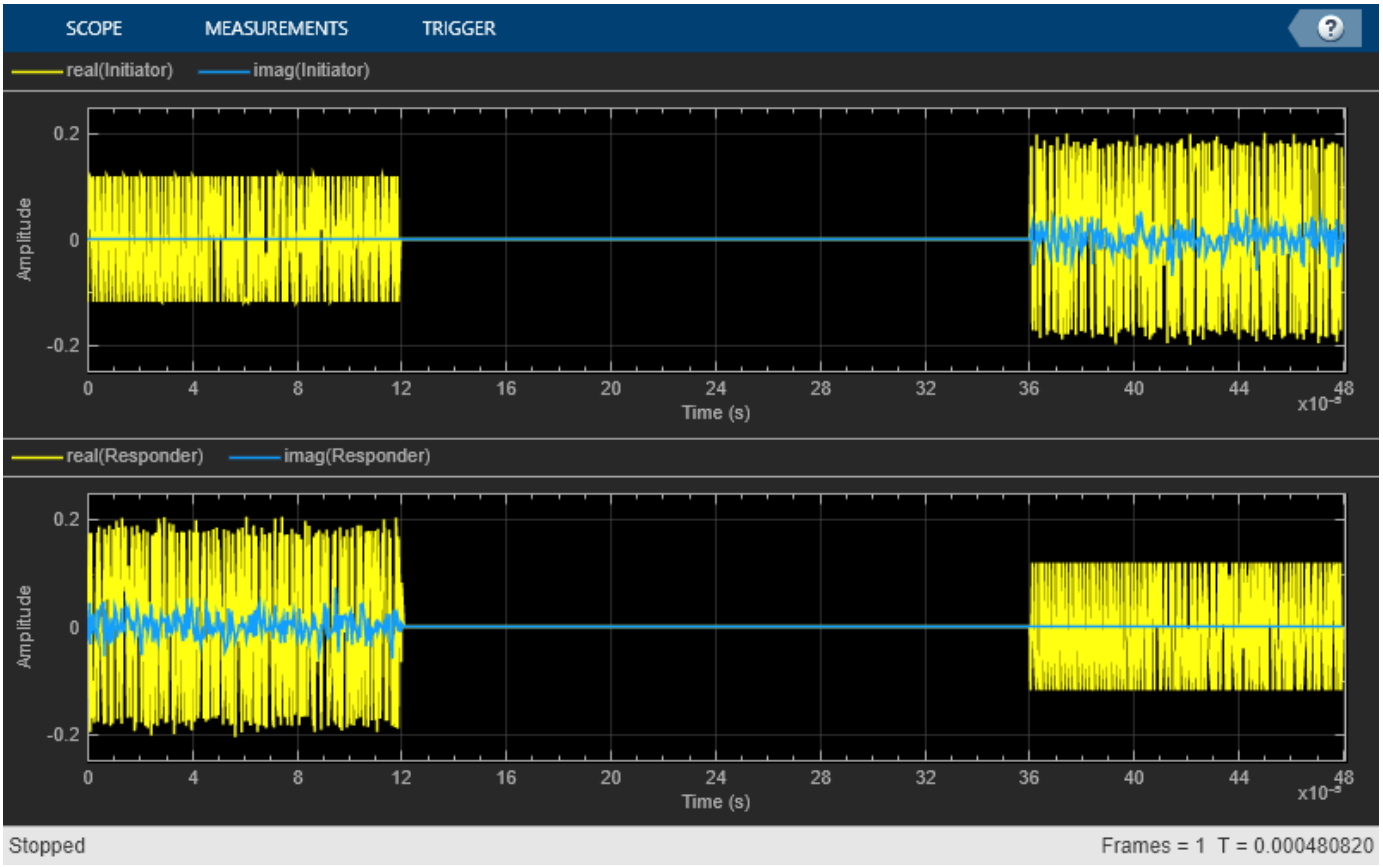
Range Estimation

Estimate the propagation delay and the distance between two devices. Set the first and last RMARKER sample indices on the initiator side to be the start of transmission (which is known at t=0) and the beginning of first post-SFD symbol. Use the RMARKERs, Tround, and Tprop to estimate the distance between initiator and responder.

```
RMARKER_I1 = 1+ind.SFD(end);
frameStart = 1+preamPos-ind.SYNC(end)/sp3Config.PreambleDuration;
sfdEnd = txFrameEnd + frameStart + ind.SYNC(end) + diff(ind.SFD);
RMARKER_I2 = sfdEnd+1;

Tround = RMARKER_I2 - RMARKER_I1;                 % In samples
Tprop = (Tround-Treply)/(2*sp3Config.SampleRate); % In seconds
estimatedDistance = c*Tprop;                      % In meters
```

This `timescope` illustrates the frame exchange as in Fig. 6-47a in [2] on page 7-0 with X-axis limit zoomed in to see the propagation delay between the transmitted and response frames.

```
reset(ts);
ts([initiatorView; zeros(ceil(samplesToDelay),1)],responderView);
release(ts);
```



The estimated distance is a few centimeters different than the actual distance.

```
fprintf(['Actual distance = %d m.' ...
    '\nEstimated Distance = %0.2f m' ...
    '\nError = %0.3f m (%0.2f%%)\n'], ...
    actualDistance,estimatedDistance, ...
    estimatedDistance-actualDistance, ...
    100*(estimatedDistance-actualDistance)/actualDistance)
```

```
Actual distance = 5 m.
Estimated Distance = 5.01 m
Error = 0.015 m (0.29%)
```

For ranging methods that rely on estimating the time of flight (TOF), errors in the distance estimate are primarily caused when the propagation time (Tprop) is not an integer multiple of the sample time. The largest distance error for such ranging methods occurs when Tprop lasts half a sample time more than an integer multiple of sample time. The smallest distance error occurs when Tprop is an integer multiple of sample time. For the higher pulse repetition frequency (HRPF) mode of the high rate pulse repetition frequency (HRP) PHY used in this example, the symbol rate is 499.2 MHz and the number of samples per symbol is 10, which results in a maximum error in Tprop estimation of $0.5 \times c/(499.2 \times 10)$. So, the default ranging error lies between 0 and 3 cm.

In general, the larger channel bandwidth in UWB corresponds to shorter symbol duration and smaller ranging error as compared to narrowband communication. For the narrowband communication as specified in IEEE 802.11az, the channel bandwidth ranges from 20 MHz to 160 MHz. Considering the maximum Tprop error for narrowband communication, estimates for the ranging error lie between 0

and 10 cm for 160 MHz and between 0 and 75 cm for 20 MHz. For more information regarding ranging with IEEE 802.11az, see the “802.11az Positioning Using Super-Resolution Time of Arrival Estimation” (WLAN Toolbox) example.

Further Exploration

This example uses these objects and functions from the Communications Toolbox™ Library for ZigBee® and UWB add-on.

- lrwpanHRPConfig**:** HRP waveform configuration
- lrwpanWaveformGenerator**:** Create an IEEE 802.15.4a/z HRP UWB waveform

These utilities are undocumented and their API or functionality may change in the future.

```
function received = lclDelayWithNoise(transmitted, samplesToDelay, SNR)
% lclDelayWithNoise Operations of wireless channel (propagation delay, AWGN)

  vfd = dsp.VariableFractionalDelay;
  % zero pad @ end, to get entire frame out of VFD
  delayedTransmitted = vfd( ...
      [transmitted; zeros(ceil(samplesToDelay), 1)],samplesToDelay);
  % add white gaussian noise:
  received = awgn(delayedTransmitted,SNR);
end
```

Selected Bibliography

1 - "IEEE Standard for Low-Rate Wireless Networks," in IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp.1-800, 23 July 2020, doi: 10.1109/IEEESTD.2020.9144691.

2 - "IEEE Standard for Low-Rate Wireless Networks--Amendment 1: Enhanced Ultra Wideband (UWB) Physical Layers (PHYs) and Associated Ranging Techniques," in IEEE Std 802.15.4z-2020 (Amendment to IEEE Std 802.15.4-2020), pp.1-174, 25 Aug. 2020, doi: 10.1109/IEEESTD.2020.9179124.

UWB Localization Using IEEE 802.15.4z

This example shows how to estimate the location of a single device as per the IEEE® 802.15.4z™ standard [2] on page 7-0 , using the Communications Toolbox™ Library for ZigBee® and UWB add-on.

Overview

The IEEE 802.15.4z amendment [2] on page 7-0 of the IEEE® 802.15.4 standard [1] on page 7-0 is a MAC and PHY specification designed for ranging and localization using ultra wideband (UWB) communication. The very short pulse durations of UWB allow a finer granularity in the time domain and therefore more accurate estimates in the spatial domain.

The key ranging and localization functionality of the 802.15.4z amendment includes 3 MAC-level techniques:

- Single-Sided Two-Way Ranging (SS-TWR) - One device estimates the distance between two devices by using frame transmission in both directions of a wireless 802.15.4z link. This technique is demonstrated in the “UWB Ranging Using IEEE 802.15.4z” on page 7-2 example.
- Double-Sided Two-Way Ranging (DS-TWR) - Both devices estimate the distance between the two devices by using frame transmission in both directions of a wireless 802.15.4z link.
- One-Way Ranging / Time-Difference of Arrival (OWR/TDOA) - Network-assisted localization whereby one device communicates with a set of synchronized nodes to estimate the position of the device.

This example demonstrates the OWR/TDOA technique for uplink transmissions, by using MAC and PHY frames are compatible with the IEEE 802.15.4 standard [1] on page 7-0 and the IEEE 802.15.4z amendment [2] on page 7-0 . For more information on generating PHY-level IEEE 802.15.4z waveforms, see the “HRP UWB IEEE 802.15.4a/z Waveform Generation” on page 7-23 example. For more information on generating IEEE 802.15.4 MAC frames, see the “IEEE 802.15.4 - MAC Frame Generation and Decoding” on page 7-50 example.

One-Way Ranging / Time-Difference of Arrival (OWR/TDOA)

One-way ranging (OWR) involves frame transmission either in the uplink or in the downlink direction. In the uplink case, the device to be localized periodically broadcasts short messages referred to as *blinks*. The IEEE 802.15.4z amendment [2] on page 7-0 does not stipulate a specific frame format for the blinks, however it states that blinks should be as short as possible. These blink messages are received by a set of infrastructure nodes that are synchronized either through a wired backbone or via an UWB wireless communications link. In the downlink case, the synchronized nodes periodically transmit broadcast messages with a known time offset.

The time-difference of arrival (TDOA) between the periodic messages places the device in one hyperbolic surface for each pair of synchronized nodes [3] on page 7-0 . The intersection of all hyperbolic surfaces (for every pair of synchronized nodes) gives the location estimate for the device.

This example demonstrates the uplink OWR case.

Setup

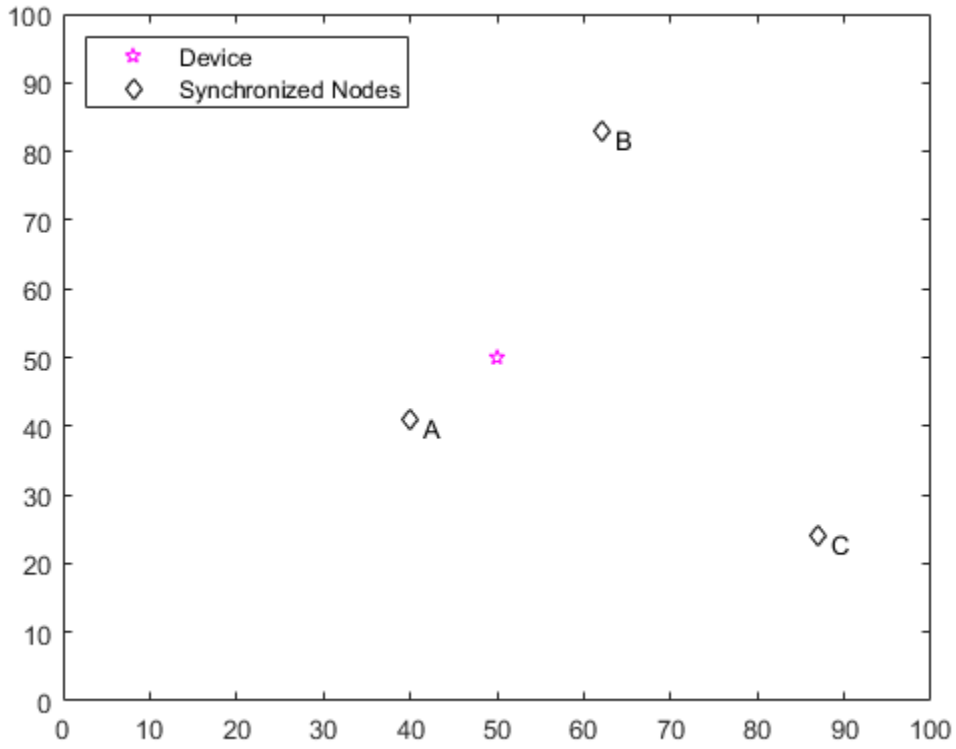
Confirm installation of the Communications Toolbox™ Library for ZigBee® and UWB add-on.

```
% Check if the 'Communications Toolbox Library for ZigBee and UWB' support package is installed:
commSupportPackageCheck('ZIGBEE');
```

Configure Network

Set up a network with 3 synchronized nodes and 1 device, in a 100x100 plane:

```
numNodes = 3;
deviceLoc = [50 50];  % place device at the center
nodeLoc = [40,41;
           62,83;
           87,24];
TDOA = nan(numNodes);
helperShowLocations(deviceLoc,nodeLoc);
```



Calculate the actual distance and time of flight (TOF) between nodes and the device.

```
actualDistances = sqrt(sum((nodeLoc - deviceLoc).^2, 2));
c = physconst('LightSpeed'); % speed of light (m/s)
actualTOF = actualDistances/c;

SNR = 30;
```

Configure Blinks

Use a short (IEEE 802.15.4 MAC) data frame as a blink.

```
numBlinks = 1;

% MAC layer:
payload = '00';
```

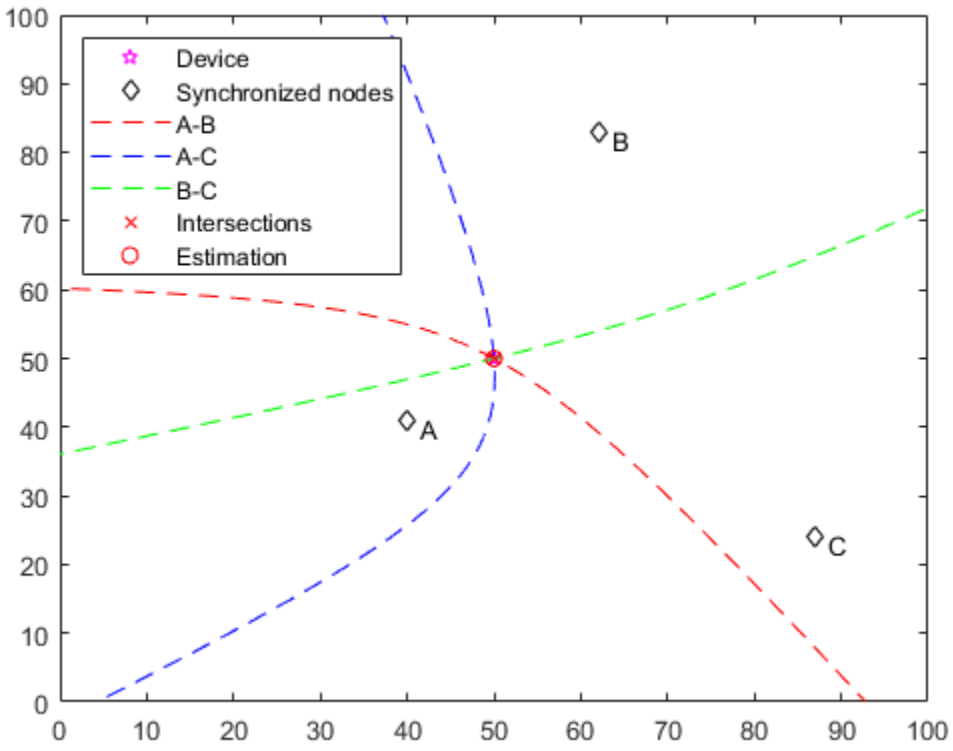
```
cfg = lrwpan.MACFrameConfig( ...
    FrameType='Data', ...
    SourceAddressing='Short address', ...
  SourcePANIdentifier='AB12', ...
  SourceAddress='CD77');
blinkMAC = lrwpan.MACFrameGenerator(cfg,payload);

% PHY layer:
% Ensure the Ranging field is enabled.
% Also set the proper PSDU length.
blinkPHYConfig = lrwpanHRPConfig( ...
    Mode='HPRF', ...
    STSPacketConfiguration=1, ...
    PSDULength=length(blinkMAC), ...
    Ranging=true);
blinkPHY = lrwpanWaveformGenerator( ...
    blinkMAC, ...
    blinkPHYConfig);

% Cache preamble, to use in preamble detection.
% Get the 1st instance out of the Nsync=PreambleDuration repetitions.
ind = lrwpanHRPFieldIndices(blinkPHYConfig); % length (start/end) of each field
blinkPreamble = blinkPHY( ...
    1:ind.SYNC(end)/blinkPHYConfig.PreambleDuration); % 1 of the Nsync repetitions
```

Run Simulation

In the simulation loop, a blink propagates to each node with a propagation delay that is determined by their distinct distance. Next, each pair of nodes calculates the difference of their blink arrival times. As a result, the position of the device is estimated within a hyperbolic surface for each pair of nodes. The intersection of all surfaces gives the position estimate for the device. Here, a plot of 2D curves shows the intersection point to indicate the position estimate for the device.

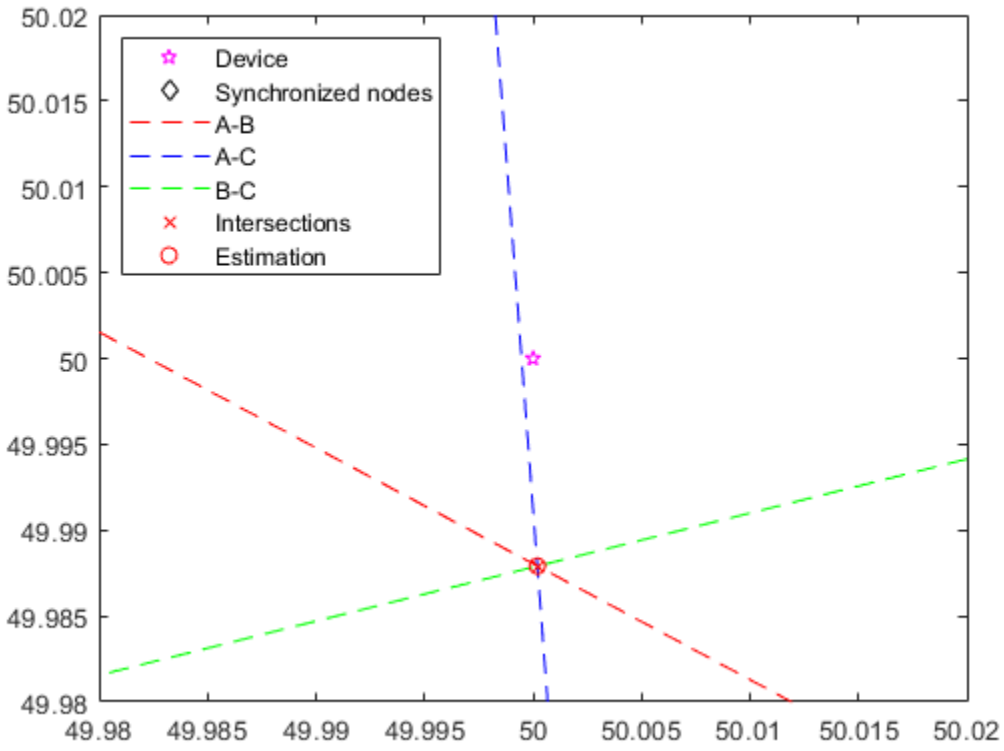
```
vfd = dsp.VariableFractionalDelay;
arrivalTime = zeros(1,numNodes);

plotStr = {'r--','b--','g--'};
[x, y] = deal(cell(1, 3));

for idx = 1:numBlinks
  for node = 1:numNodes
    % Transmission and reception of blink
    % Each node receives a specifically delayed version of the blink
    tof = actualTOF(node);
    samplesToDelay = tof * blinkPHYConfig.SampleRate;
    reset(vfd);
    release(vfd);
    vfd.MaximumDelay = ceil(1.1*samplesToDelay);
    delayedBlink = vfd( ...
        [blinkPHY; zeros(ceil(samplesToDelay), 1)], ...
        samplesToDelay);

    % Add white Gaussian noise
    receivedBlink = awgn(delayedBlink,SNR);

    % Node receiver detection of preamble
    preamPos = helperFindFirstHRPPreamble( ...
        receivedBlink,blinkPreamble,blinkPHYConfig);
```

```
        % Transmit each blink at t=0 of each period
        arrivalTime(node) = ( ...
            preamPos - ind.SYNC(end) / ...
            blinkPHYConfig.PreambleDuration)/blinkPHYConfig.SampleRate;
    end

    % Localization: Estimate position at synchronized backbone for each pair
    % of nodes
    pairCnt = 1;
    for node1 = 1:numNodes
        for node2 = (node1+1):numNodes
            % Calculate Time Difference of Arrival (TDOA)
            TDOA(node1, node2) = arrivalTime(node1)-arrivalTime(node2);

            % Get hyperbolic surface for the TDOA between node1 and node2
            [x{pairCnt}, y{pairCnt}] = helperGetHyperbolicSurface( ...
                nodeLoc(node1,:), ...
                nodeLoc(node2,:), ...
                TDOA(node1,node2));

            plot(x{pairCnt},y{pairCnt},plotStr{pairCnt});
            pairCnt = pairCnt + 1;
        end
    end
    % Find intersection points between hyperbolic surfaces
    [xC,yC] = helperFindHyperbolicIntersection(x,y);
    plot(xC,yC,'rx')
    % Estimate location as the center of intersection triangle
    x0 = mean(xC);
    y0 = mean(yC);
    plot(x0,y0,'ro')
end
legend( ...
    'Device', ...
    'Synchronized nodes', ...
    'A-B', ...
    'A-C', ...
    'B-C', ...
    'Intersections', ...
    'Estimation', ...
    'location', ...
    'northwest')
```



Zoom in to estimation area:

```
axis([deviceLoc(1) + 2e-2*[-1 1],deviceLoc(2) + 2e-2*[-1 1]])
```



Calculate the localization error.

```
locError = sqrt(sum([x0 y0]-deviceLoc).^2);
fprintf('Localization error = %0.3f m.\n',locError);
```

```
Localization error = 0.012 m.
```

For localization methods that rely on estimating the time of arrival, errors in the distance estimate are primarily caused when the arrival time is not an integer multiple of the sample time. The largest distance error for such localization methods occurs when the arrival time lasts half a sample time more than an integer multiple of sample time. The smallest distance error occurs when the arrival time is an integer multiple of sample time. For the higher pulse repetition frequency (HRPF) mode of the high rate pulse repetition frequency (HRP) PHY used in this example, the symbol rate is 499.2 MHz and the number of samples per symbol is 10. The maximum distance estimation error is $0.5 \times c/(499.2 \times 10)$, which is approximately 3 cm.

In general, the larger channel bandwidth in UWB corresponds to shorter symbol duration and smaller ranging error as compared to narrowband communication. For the narrowband communication as specified in IEEE 802.11az, the channel bandwidth ranges from 20 MHz to 160 MHz. Considering the maximum distance error for narrowband communication, estimates for the localization error lie between 0 and 10 cm for 160 MHz and between 0 and 75 cm for 20 MHz. For more information regarding positioning with IEEE 802.11az, see the “802.11az Positioning Using Super-Resolution Time of Arrival Estimation” (WLAN Toolbox) example.

Further Exploration

This example uses these objects and functions from the Communications Toolbox™ Library for ZigBee® and UWB add-on.

- lrwpan.MACFrameConfig: Create configuration for 802.15.4 MAC frames
- lrwpan.MACFrameGenerator: Generate 802.15.4 MAC frames
- lrwpanHRPConfig: HRP waveform configuration
- lrwpanWaveformGenerator: Create an IEEE 802.15.4a/z HRP UWB waveform

These utilities are undocumented and their API or functionality may change in the future.

Selected Bibliography

1 - "IEEE Standard for Low-Rate Wireless Networks," in IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp.1-800, 23 July 2020, doi: 10.1109/IEEESTD.2020.9144691.

2 - "IEEE Standard for Low-Rate Wireless Networks--Amendment 1: Enhanced Ultra Wideband (UWB) Physical Layers (PHYs) and Associated Ranging Techniques," in IEEE Std 802.15.4z-2020 (Amendment to IEEE Std 802.15.4-2020), pp.1-174, 25 Aug. 2020, doi: 10.1109/IEEESTD.2020.9179124.

3 - Wong, S.; Zargani, R. Jassemi; Brookes, D. & Kim, B. "Passive target localization using a geometric approach to the time-difference-of-arrival method", Defence Research and Development Canada Scientific Report, DRDC-RDDC-2017-R079, June 2017, pp. 1-77

End-to-End Simulation of HRP UWB IEEE 802.15.4a/z PHY

This example performs end-to-end simulation over an additive white Gaussian noise (AWGN) channel for the high rate pulse repetition frequency (HRP) ultra wideband (UWB) PHY of the IEEE® 802.15.4a/z™ standard ([1], [2]), using the Communications Toolbox™ Library for ZigBee® and UWB add-on.

Background

The **IEEE 802.15.4** standard specifies the **PHY** and **MAC** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 PHY and MAC layers are used by other higher-layer standards, such as **ZigBee®**, WirelessHart®, 6LoWPAN, and MiWi.

These PHY schemes are specified in different amendments of the IEEE 802.15.4 standard:

- IEEE 802.15.4a introduced a high rate pulse repetition frequency (**HRP**) UWB PHY used for ranging and localization [1].
- IEEE 802.15.4f introduced a low rate pulse repetition frequency (**LRP**) UWB PHY used for RFID, ranging, and reduced energy consumption [1].
- IEEE 802.15.4z introduced new enhanced modes for both the HRP and LRP UWB IEEE 802.15.4a/f PHYs [2].

The HRP UWB PHY has a channel bandwidth of 0.5-1.3 GHz and a pulse duration of 2 ns. The extra short pulse duration makes UWB PHYs suitable for ranging applications, because several ranging techniques rely on calculating the time duration of packet transmission. A finer granularity in the time domain translates to smaller errors in distance estimation.

This example performs end-to-end simulation, computes bit error rate (BER) curves, and demonstrates a tradeoff between reliability and bit rate for these HRP IEEE 802.15.4a/z PHY modes:

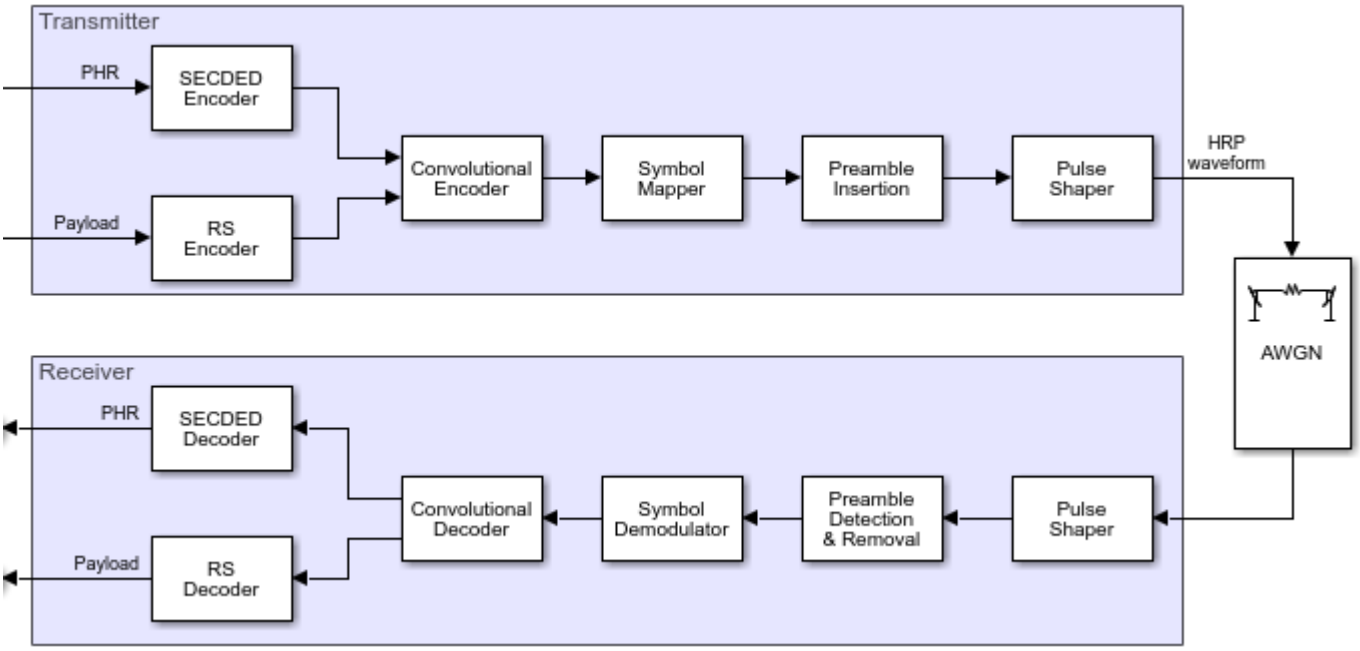
1. Higher Pulse Repetition Frequency (HPRF) mode, which was introduced in IEEE 802.15.4z [2].
2. Base Pulse Repetition Frequency (BPRF) mode, which was introduced in IEEE 802.15.4z [2] (but can reduce to IEEE 802.15.4a [1]).
3. IEEE 802.15.4a [1], which has a lower mean pulse repetition frequency (PRF) and lower data rate than the HPRF and BPRF modes.

This table shows the different modulation schemes, data rates, and number of chips per payload symbol used by these operational modes.

	HPRF, 249 MHz	HPRF, 124 MHz	BPRF	802.15.4a
Data rate (Mbps)	27.24	6.81	6.81	0.11, 0.85, 1.7, 6.81, 27.24
Number of chips per payload symbol	8	16	8	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 4096

Common Processing Steps

As shown in this PHY chain, the various HRP modes share certain common components.



The IEEE 802.15.4 and IEEE 802.15.4z standards only specify the transmitter operation ([1], [2]). The receiver performs the inverse operations of the transmitter. The receiver implementation does not perform frequency or timing recovery.

SECDED Coding: The PHY header (PHR) is encoded with a single-error-correction, double-error detection (SECDED) Hamming block code. The BER calculation in this example does not use the PHR bits.

RS Coding: The payload is encoded/decoded with a (63, 55) Reed-Solomon code.

Convolutional coding: The payload and the PHR are encoded/decoded with a rate 1/2 convolutional code and a constraint length of 3. An optional rate 1/2 convolutional code with a constraint length 7 is offered for the HPRF mode, but it is not used in this example.

Preamble insertion/removal: A selected code sequence is spread and repeated. The SYNC field consists of this preamble with a start-of-frame delimiter (SFD) appended at the end. The receiver expects the input waveform to begin with the preamble, without any delay that would necessitate preamble detection.

Pulse shaper: The output of symbol mapping and preamble insertion are ternary symbols (-1,0,1). The IEEE 802.15.4a/z standard allows multiple pulse shapes to represent the symbol sequence in the analog domain. Section 15.4 in [1], [2] specify RF conformance specifications. In this example, the ternary symbol sequence is passed to a Butterworth filter to create Butterworth pulses. On the receiver side, an integrate-and-dump operation converts the pulses back to ternary symbols.

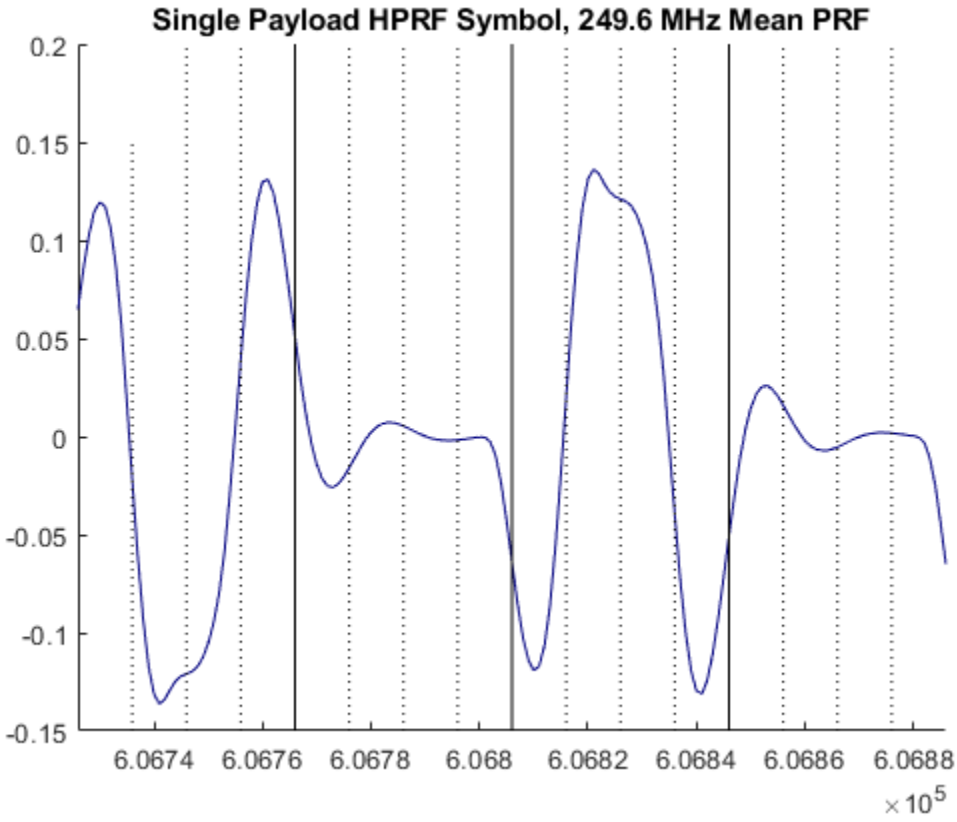
The main difference between the 3 different modes lies in the **Symbol Mapper** component (and the respective demodulator). Other differences lie in the PHR format, as well as the length and value of the preamble code sequence and SFD.

HPRF mode

In the HPRF mode, the mean pulse repetition frequency (PRF) is either 249.6 MHz or 124.8 MHz, with data rates of 27.24 Mbps or 6.81 Mbps, respectively. In both cases, each symbol duration

consists of alternating segments of transmitted chip sequences and guardbands. This code segment generates a plot to show a payload symbol for the 249.6 MHz HPRF mode.

```
% Ensure ZigBee/UWB support package is installed:
commSupportPackageCheck('ZIGBEE');

msg = randi([0 1], 1000, 1);
cfgHPRF = lrwpanHRPConfig(Mode='HPRF',PSDULength=length(msg));
waveHPRF = lrwpanWaveformGenerator(msg,cfgHPRF);
fig = lrwpanPlotFrame(waveHPRF,cfgHPRF);
hZoomTo1stHPRFPayloadSymbol(fig,cfgHPRF)
```

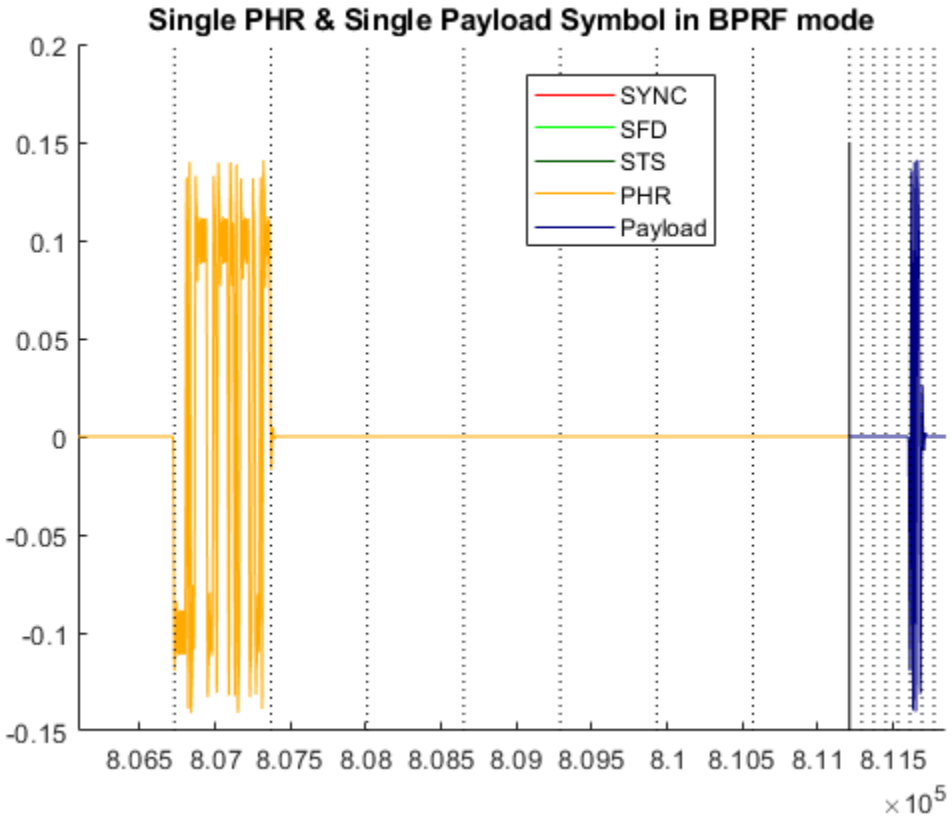


Each convolutional codeword is 2 bits long (one parity bit for each systematic). These 2 bits map to 8 payload and 16 PHR bits for the 249.6 MHz mean PRF (see 8 pulses in above plot), and map to 16 payload and 32 PHR bits for the 124.8 MHz mean PRF.

BPRF mode

The BPRF mode uses burst-position BPSK modulation (BPM-BPSK). The symbol duration is split into a set of candidate burst positions. Each burst contains a specified number of chips per burst (*Ncpb*). One candidate burst contains a pattern. All other candidate bursts transmit zeros.

```
cfgBPRF = lrwpanHRPConfig(Mode='BPRF',CodeIndex=9);
waveBPRF = lrwpanWaveformGenerator(repmat([0; 1],508,1),cfgBPRF);
fig = lrwpanPlotFrame(waveBPRF,cfgBPRF);
hZoomToBPMBPSKSymbols(fig,cfgBPRF);
```



In the BPRF mode, the mean PRF is 62.4 MHz and the payload data rate is 6.81 Mbps. The *Ncpb* for each burst within a symbol is 8 chips, and the number candidate burst positions (*Nhop*) is 4. The systematic bit reduces the set of candidate positions by 50%, and the active burst is selected among the remaining 2 *Nhop* based on a spreading (burst-hopping) sequence.

IEEE 802.15.4a

The IEEE 802.15.4a HRP PHY also uses BPM-BPSK modulation, similar to the BPRF mode. The only difference is that more values are allowed for the mean PRF and data rate combination.

Specifically, mean PRF can be 3.9, 15.6 or 62.4 MHz, while data rate can be 0.11, 0.85, 1.7, 6.81, or 27.24 Mbps. *Ncpb* can be 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, or 4096. *Nhop* is 2, 8 or 32. The range of *Ncpb* values enables transmissions that can be either more aggressive or more conservative than the BPRF mode.

BER Curve Calculation

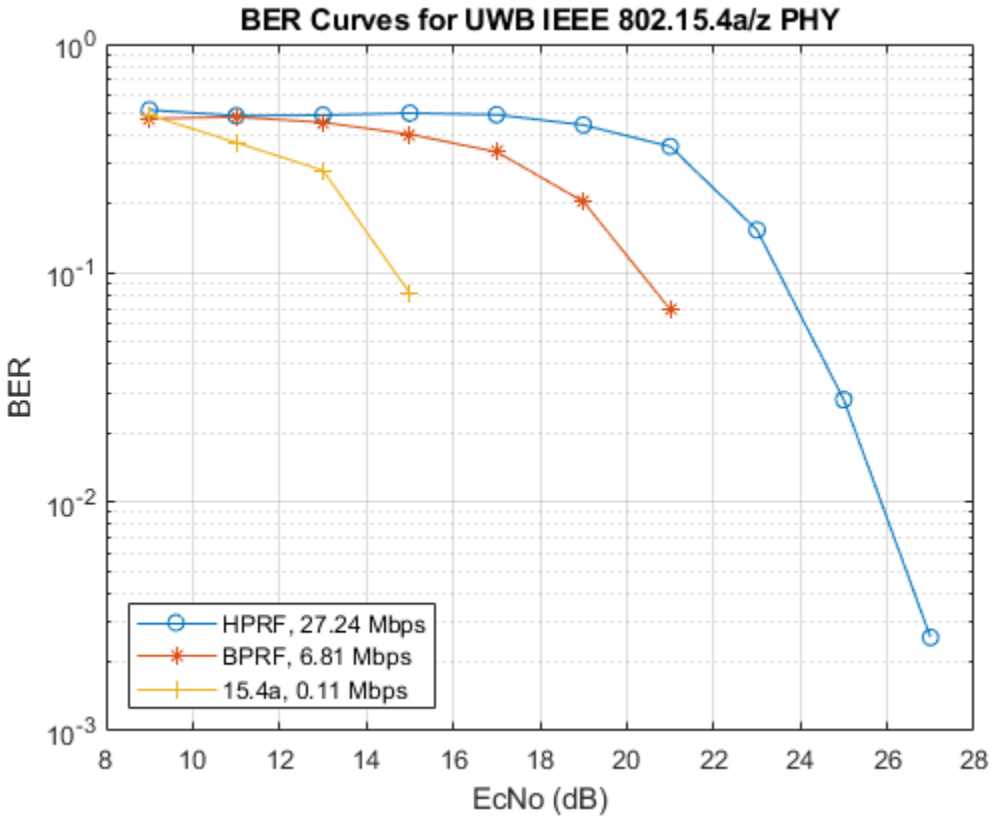
For each of the three HRP PHY modes, the example calculates BER over the EcNo range [6,32] in dB (mapped to equivalent SNR values) using end-to-end simulations over an AWGN channel.

For IEEE 802.15.4a, a 15.6 MHz mean PRF is used with a 0.11 Mbps data rate. This combination enables *Ncpb* equal to 128 and *Nhop* equal to 8.

```
msgLen = 2^10 - 8;
msg = randi([0 1],msgLen,1);
EcNo = 9:2:35;
numSNR = length(EcNo);
```

```
[berHPRF,berBPRF,ber4a] = deal(zeros(1,numSNR));

% Construct fixed configurations and waveforms for each mode:
cfgHPRF = lrwpanHRPConfig(Mode='HPRF',PSDULength=msgLen);
waveHPRF = lrwpanWaveformGenerator(msg,cfgHPRF);

cfgBPRF = lrwpanHRPConfig(Mode='BPRF',CodeIndex=9,PSDULength=msgLen);
waveBPRF = lrwpanWaveformGenerator(msg,cfgBPRF);

cfg4a = lrwpanHRPConfig( ...
    Mode='802.15.4a', ...
    MeanPRF='15.6MHz', ...
    DataRate='0.11Mbps', ...
    CodeIndex=1, ...
    PSDULength=msgLen);
wave4a = lrwpanWaveformGenerator(msg,cfg4a);


% Compute BER curve until required number of errors have been found or
% maximum number of bits have been simulated.
MAXBITS = msgLen*5;
MINERRORS = 10;
for idx = 1:numSNR
  fprintf('Calculating BER for EcNo=%d dB\n',EcNo(idx));
  errCnt = 0;
  bitCnt = 0;
  errHPRF = 0;
  errBPRF = 0;
  err4a = 0;
  while errCnt < MINERRORS && bitCnt < MAXBITS
    % HPRF mode
    SNR = EcNo(idx) - 10*log10(cfgHPRF.SamplesPerPulse);
    noisyHPRF = awgn(waveHPRF,SNR);
    psduHPRF = lrwpanWaveformDecoder(noisyHPRF,cfgHPRF);
    errHPRF = biterr(msg, psduHPRF)+errHPRF;

    % BPRF mode
    SNR = EcNo(idx) - 10*log10(cfgBPRF.SamplesPerPulse);
    noisyBPRF = awgn(waveBPRF,SNR);
    psduBPRF = lrwpanWaveformDecoder(noisyBPRF,cfgBPRF);
    errBPRF = biterr(msg,psduBPRF)+errBPRF;

    % Legacy 802.15.4a
    SNR = EcNo(idx) - 10*log10(cfg4a.SamplesPerPulse);
    noisy4a = awgn(wave4a,SNR);
    psdu4a = lrwpanWaveformDecoder(noisy4a,cfg4a);
    err4a = biterr(msg,psdu4a)+err4a;

    bitCnt = bitCnt + msgLen;
    errCnt = min([errHPRF errBPRF err4a]);
  end
  berHPRF(idx) = errHPRF/bitCnt;
  berBPRF(idx) = errBPRF/bitCnt;
  ber4a(idx) = err4a/bitCnt;
end

% Plot BER curve
figure
```

```
semilogy(EcNo,berHPRF,'-o',EcNo,berBPRF,'-*',EcNo,ber4a,'-+')
legend('HPRF, 27.24 Mbps','BPRF, 6.81 Mbps', ...
    '15.4a, 0.11 Mbps','Location','southwest')
title('BER Curves for UWB IEEE 802.15.4a/z PHY')
xlabel('EcNo (dB)')
ylabel('BER')
grid on

Calculating BER for EcNo=9 dB
Calculating BER for EcNo=11 dB
Calculating BER for EcNo=13 dB
Calculating BER for EcNo=15 dB
Calculating BER for EcNo=17 dB
Calculating BER for EcNo=19 dB
Calculating BER for EcNo=21 dB
Calculating BER for EcNo=23 dB
Calculating BER for EcNo=25 dB
Calculating BER for EcNo=27 dB
Calculating BER for EcNo=29 dB
Calculating BER for EcNo=31 dB
Calculating BER for EcNo=33 dB
Calculating BER for EcNo=35 dB
```



The BER curve results demonstrate higher bit error rate for more aggressive modulation schemes and lower BER for more conservative modulation schemes. Lower data rates use more chips for each transmitted convolutional codeword. A higher number of transmitted chips provide more opportunity for error correction, which is similar in concept to using more parity bits in channel coding.

Further Exploration

The Communications Toolbox Library for ZigBee and UWB add-on contains the following object and functions:

- **lrwpanHRPConfig:** HRP waveform configuration
- **lrwpanWaveformGenerator:** Create an IEEE 802.15.4a/z HRP UWB waveform
- **lrwpanWaveformDecoder:** Decode HRP IEEE 802.15.4a/z UWB waveform

These utilities are undocumented and their API or functionality may change in the future.

Selected Bibliography

1 "IEEE Standard for Low-Rate Wireless Networks," in IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp.1-800, 23 July 2020, doi: 10.1109/IEEESTD.2020.9144691.

2 "IEEE Standard for Low-Rate Wireless Networks--Amendment 1: Enhanced Ultra Wideband (UWB) Physical Layers (PHYs) and Associated Ranging Techniques," in IEEE Std 802.15.4z-2020 (Amendment to IEEE Std 802.15.4-2020), pp.1-174, 25 Aug. 2020, doi: 10.1109/IEEESTD.2020.9179124.

HRP UWB IEEE 802.15.4a/z Waveform Generation

This example shows how to generate standard-compliant high rate pulse repetition frequency (HRP) ultra wideband (UWB) waveforms of the IEEE® 802.15.4a/z™ standard ([1], [2]), using the Communications Toolbox™ Library for ZigBee® and UWB add-on.

Background

The **IEEE 802.15.4** standard specifies the **PHY** and **MAC** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 PHY and MAC layers are used by higher-layer standards, such as **ZigBee®**, WirelessHart®, 6LoWPAN and MiWi.

Multiple PHY schemes are specified in different amendments of the IEEE 802.15.4 standard:

- IEEE 802.15.4a introduced a high rate pulse repetition frequency (**HRP**) UWB PHY used for ranging (i.e., localization) [1].
- IEEE 802.15.4f introduced a low rate pulse repetition frequency (**LRP**) UWB PHY used for RFID, ranging, and reduced energy consumption [1].
- IEEE 802.15.4z introduced new enhanced modes for both the HRP and LRP UWB IEEE 802.15.4a/f PHYs [2].

The HRP UWB PHY specifies a channel bandwidth of 0.5-1.3 GHz and a pulse duration of 2 ns. Since the calculations used for ranging techniques rely on the time duration of packet transmission, the extra short pulse duration makes UWB PHYs suitable for ranging applications. A finer granularity in the time domain translates to smaller errors in distance estimation.

This example generates standard-compliant HRP UWB 802.15.4a/z waveforms for three pulse repetition frequency (PRF) transmission modes (802.15.4a, and 802.15.4z BPRF and HPRF). For **IEEE 802.15.4a**, the valid mean PRF values are 3.9, 15.6 or 62.4 MHz. The **IEEE 802.15.4z** amendment defines these two PRF modes:

- Base pulse repetition frequency (**BPRF**), where the mean PRF is 62.4 MHz and the payload data rate is 6.81 Mbps
- Higher pulse repetition frequency (**HPRF**), where the mean PRF is either 124.8 or 249.6 MHz.

The scrambled timestamp sequence (**STS**) field is another key feature introduced by 802.15.4z to enhance data integrity. Transmission of the STS field is optional for the BRPF and HPRF modes.

Configuration for HRP Waveform Generation

The helper `lrwpanHRPConfig` object configures the waveform of each transmission mode. This table lists the properties, conditions under which they apply and valid settings.

Property	Application	Allowed Values	Default
Channel	Always	[0:3, 5, 6, 8:10, 12:14]	0
Mode	Always	'HPRF' 'BPRF' '802.15.4a'	'HPRF'
MeanPRF	Mode='HPRF' Mode='802.15.4a'	{'249.6MHz', '124.8MHz'} when Mode='HPRF' '62.4MHz' when Mode='BPRF' {'3.9MHz', '15.6MHz', '62.4MHz'} when Mode='802.15.4a'	'249.6MHz'
DataRate	Mode='802.15.4a'	'6.81Mbps' when Mode='BPRF' {'0.11Mbps', '0.85Mbps', '6.81Mbps', '27.24Mbps'} when Mode='802.15.4a' && MeanPRF={'15.6MHz', '62.4MHz'} {'0.11Mbps', '0.85Mbps', '1.7Mbps', '6.81Mbps'} when Mode='802.15.4a' && MeanPRF='3.9MHz'	'0.85Mbps'
PHRDataRate	Mode='BPRF'	{'0.85Mbps', '6.81Mbps'}	'0.85Mbps'
SamplesPerPulse	Always	Positive integers	10
STSPacketConfiguration	Mode='HPRF' Mode='BPRF'	[0, 1, 2, 3]	1
NumSTSSegments	Mode='HPRF'	[0, 1, 2, 3]	1
ActiveSTSLength	Mode='HPRF'	[16, 32, 64, 128, 256]	64
ExtraSTSGapLength	Mode='HPRF' && STSPacketConfiguration=2	0:127	0
ExtraSTSGapIndex	Mode='HPRF' && STSPacketConfiguration=2	[0, 1, 2, 3]	0
CodeIndex	Always	1:2 when MeanPRF='{'3.9MHz', '15.6MHz'}, Channel=[0,1,8,12] 3:4 when MeanPRF='{'3.9MHz', '15.6MHz'}, Channel=[2,5,9,13] 5:6 when MeanPRF='{'3.9MHz', '15.6MHz'}, Channel=[3,6,10,14] [9:16, 21:24] when MeanPRF='{'62.4MHz'} 25:32 when Mode={'HPRF', 'BPRF'}	25
PreambleMeanPRF	Mode='802.15.4a' && MeanPRF={'3.9MHz', '15.6MHz'}	{'4.03MHz', '16.1MHz'}	'16.1MHz'
PreambleDuration	Always	[16, 24, 32, 48, 64, 96, 128, 256], when Mode='HPRF' [16, 64, 1024, 4096], when Mode='BPRF' || (Mode='802.15.4a' && PreambleMeanPRF~='4.03MHz') [16, 64, 1024], when Mode='802.15.4a' && PreambleMeanPRF='4.03MHz'	64
SFDNumber	Mode='HPRF' Mode='BPRF'	[0, 1, 2, 3, 4] when Mode='HPRF' [0, 2] when Mode='BPRF'	0
Ranging	Always	true/false	false
ConstraintLength	Mode='HPRF'	[3, 7], when Mode='HPRF' 3 otherwise	3
PSDULength	Always	Multiple of 8 in [0 32760] when Mode='HPRF' && (STSPacketConfiguration~=2 || (ExtraSTSGapLength=0 && ExtraSTSGapIndex=0)) Multiple of 8 in [0 8184] when Mode='HPRF' && (STSPacketConfiguration==2 && (ExtraSTSGapLength>0 || ExtraSTSGapIndex>0)) Multiple of 8 in [0 1016] when Mode='BPRF', '802.15.4a'	1016

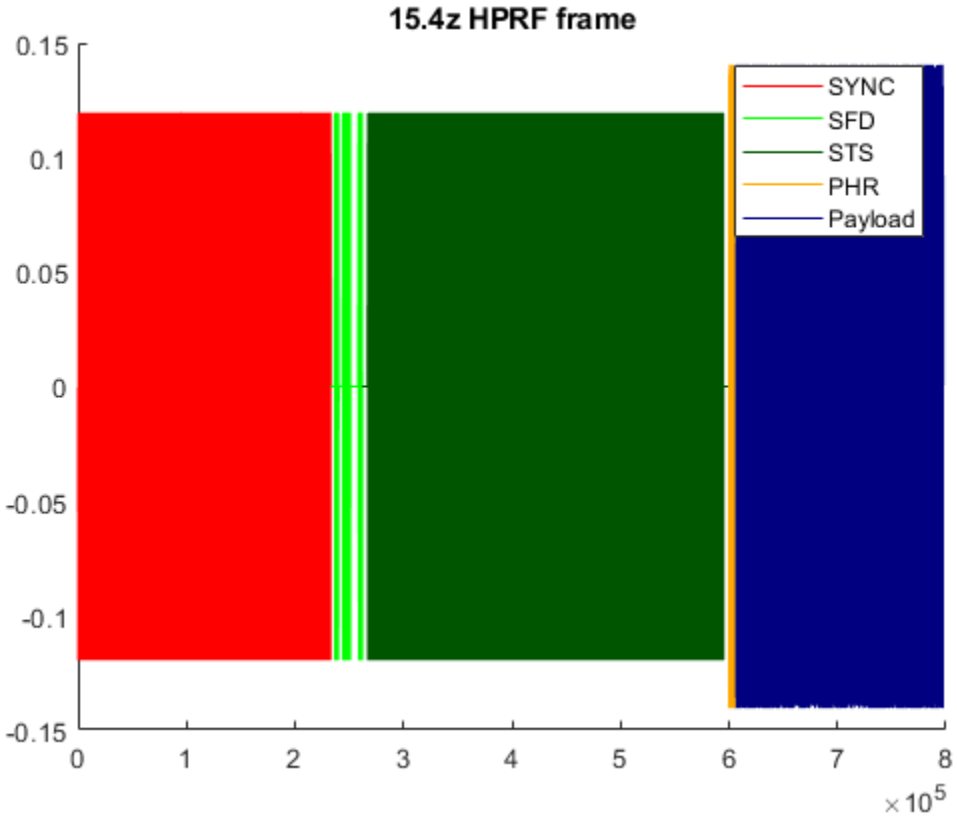
The `lrwpanWaveformGenerator` helper function generates HRP UWB IEEE 802.15.4a/z waveforms using `lrwpanHRPConfig` objects and the PHY service data unit (PSDU) as inputs.

HPRF mode in IEEE 802.15.4z

In the higher pulse repetition frequency (**HPRF**) mode of IEEE 802.15.4z, the mean PRF is either 124.8 or 249.6 MHz. Since HPRF mode uses higher PRFs than BPRF or IEEE 802.15.4a, the HPRF mode can estimate range more accurately. The default mean PRF of the lrwpanHRPConfig object is 249.6 MHz.

```
% This code confirms the Communications Toolbox(TM) Library for ZigBee(R)
% and UWB add-on is installed.
commSupportPackageCheck('ZIGBEE');

msg = randi([0 1], 1000, 1);
cfgHPRF = lrwpanHRPConfig(Mode='HPRF', PSDULength=length(msg)) %#ok<NOPTS>
waveHPRF = lrwpanWaveformGenerator(msg, cfgHPRF);

lrwpanPlotFrame(waveHPRF, cfgHPRF);
```

```
cfgHPRF =

  lrwpanHRPConfig with properties:

                  Channel: 0
                     Mode: 'HPRF'
                  MeanPRF: '249.6MHz'
           SamplesPerPulse: 10
    STSPacketConfiguration: 1
            NumSTSSegments: 1
          ActiveSTSLength: 64
                CodeIndex: 25
          PreambleDuration: 64
                 SFDNumber: 0
                   Ranging: 0
          ConstraintLength: 3
                PSDULength: 1000

   Read-only properties:
                  PeakPRF: '499.2MHz'
            ChipsPerSymbol: [16 8]
       ConvolutionalCoding: 1
        PreambleCodeLength: 91
  PreambleSpreadingFactor: 4
                SampleRate: 4.9920e+09
```



The HPRF frame consists of the following fields:

Synchronization (SYNC) field: The SYNC field contains the specified number of repetitions (*Nsync*) of a 91-symbol long code spread according to the `PreambleSpreadingFactor` property. The `CodeIndex` property determines which code is used. (*Nsync*) is specified by the `PreambleDuration` property.

Start-of-frame delimiter (SFD) field: The SFD field is a 4-, 8-, 16- or 32-symbol sequence spread with the SYNC code corresponding to the `CodeIndex` property. The length of the starting SFD sequence is determined by the `SFDNumber` property.

Scrambled timestamp sequence (STS) field: The STS field is explained in the next section.

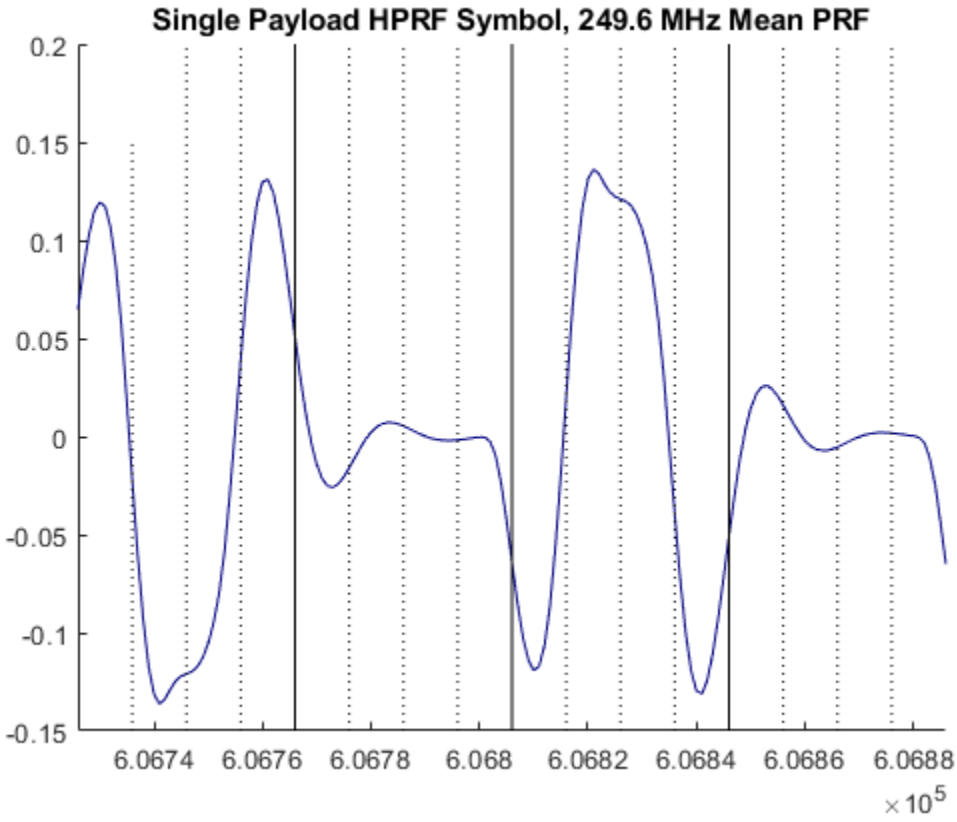
PHY Header (PHR) field: The PHR field is a 19 bit sequence that contains 6 parity bits generated by a single error correction, double error detection (SECDED) Hamming block code. The `Ranging` property determines one of the 13 systematic PHR bits. Subsequently, the PHR is convolutionally encoded with a rate 1/2 convolutional code. The `ConstraintLength` property (3 or 7) chooses between two rate 1/2 convolutional encoders.

For the HPRF modulation scheme (Sec. 15.3.4 in [2]), each PHR convolutional codeword is mapped to a sequence of 16 or 32 pulses (for mean PRF 249.6 and 124.8 MHz, respectively). Pulse sequences are separated by guard intervals. The first element of the `ChipsPerSymbol` property conveys the number of pulses in each PHR symbol.

Payload: The PSDU is encoded with a (63, 55) Reed-Solomon code. Subsequently, it is convolutionally encoded (together with the PHR) with a rate 1/2 convolutional code. The `ConstraintLength` property (3 or 7) chooses between two rate 1/2 convolutional encoders.

For the HPRF modulation scheme (Sec. 15.3.4 in [2]), each convolutional codeword of the payload is mapped to a sequence of 8 or 16 pulses (for mean PRF 249.6 and 124.8 MHz, respectively). Pulse sequences are separated by guard intervals. The last element of the `ChipsPerSymbol` property conveys the number of pulses in each payload symbol. This figure illustrates a single payload symbol at a 249.6 MHz Mean PRF.

```
fig = lrwpanPlotFrame(waveHPRF, cfgHPRF);
hZoomTo1stHPRFPayloadSymbol(fig, cfgHPRF)
```



The second and the fourth quarter of the symbol are guard intervals. The first and third quarter contain 4 chip transmissions each.

Scrambled Timestamp Sequence (STS)

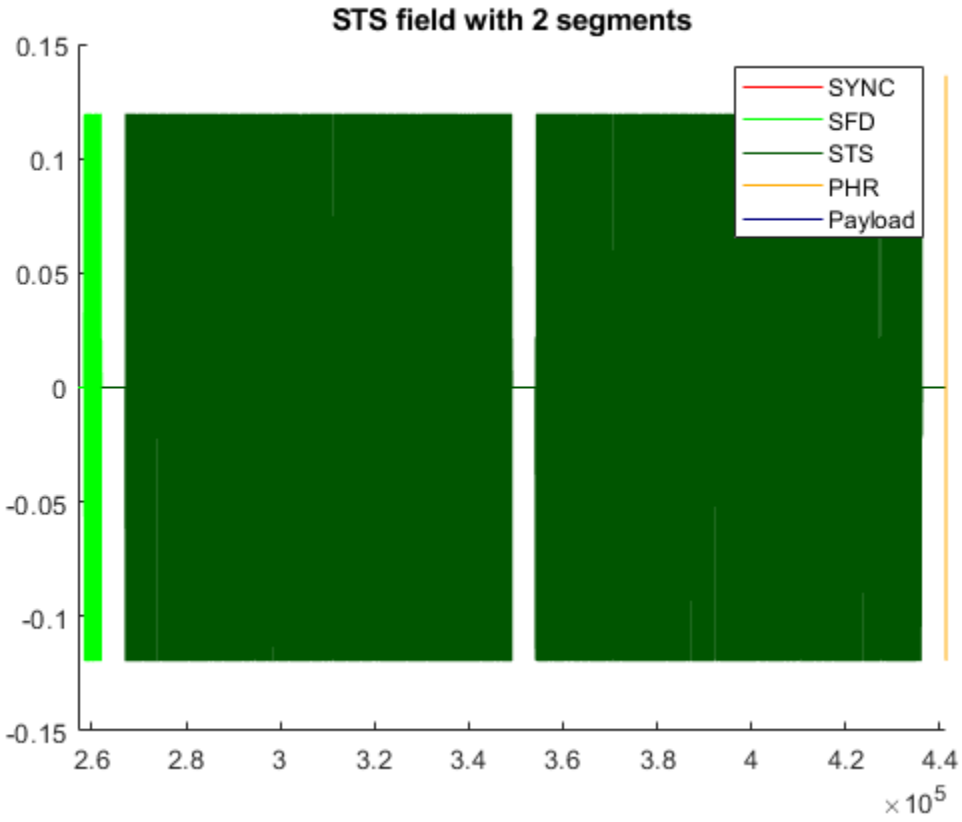
The **STS** field can be used to ensure the authenticity of the ranging estimates. This field is optional for the HPRF and BPRF modes. The `STSPacketConfiguration` property specifies the initial configuration of the STS field. To omit the STS field, specify `0` for the `STSPacketConfiguration` property. The other values determine the STS and PHR/payload placement within the PHY frame.

The STS field consists of multiple segments separated by a gap. The `NumSTSSegments` property determines the number of segments (1 to 4) and the `ActiveSTSLength` property determines the length of each segment (16, 32, 64, 128 or 256 in multiples of 512 chips).

This code configures, generates, and visualizes a waveform containing 2 STS segments with gaps before and after each segment. A portion of the preceding SFD field is included.

```
msg = randi([0 1], 2000, 1);
cfgSTS = lrwpanHRPConfig( ...
    Mode='HPRF', ...
    NumSTSSegments=2, ...
    ActiveSTSLength=16, ...
    PSDULength=length(msg));
waveSTS = lrwpanWaveformGenerator(msg,cfgSTS);

lrwpanPlotFrame(waveSTS, cfgSTS);
ind = lrwpanHRPFieldIndices(cfgSTS);
```

```
set(gca,'XLim',ind.STS - [5e3 0]) % portion of preceding field (SFD)
title('STS field with 2 segments')
```



The STS generation in this example creates the STS structure (including number of segments, gaps, segment length, STS spreading, and pulse polarity), but does not perform AES-128 encryption. Random bits are used in place of the AES-128 output. To implement AES-128, incorporate the `aes128Placeholder` subfunction of this file. The `aes128Placeholder` subfunction includes the counter and the 128-bit V value.

BPRF mode in IEEE 802.15.4a/z

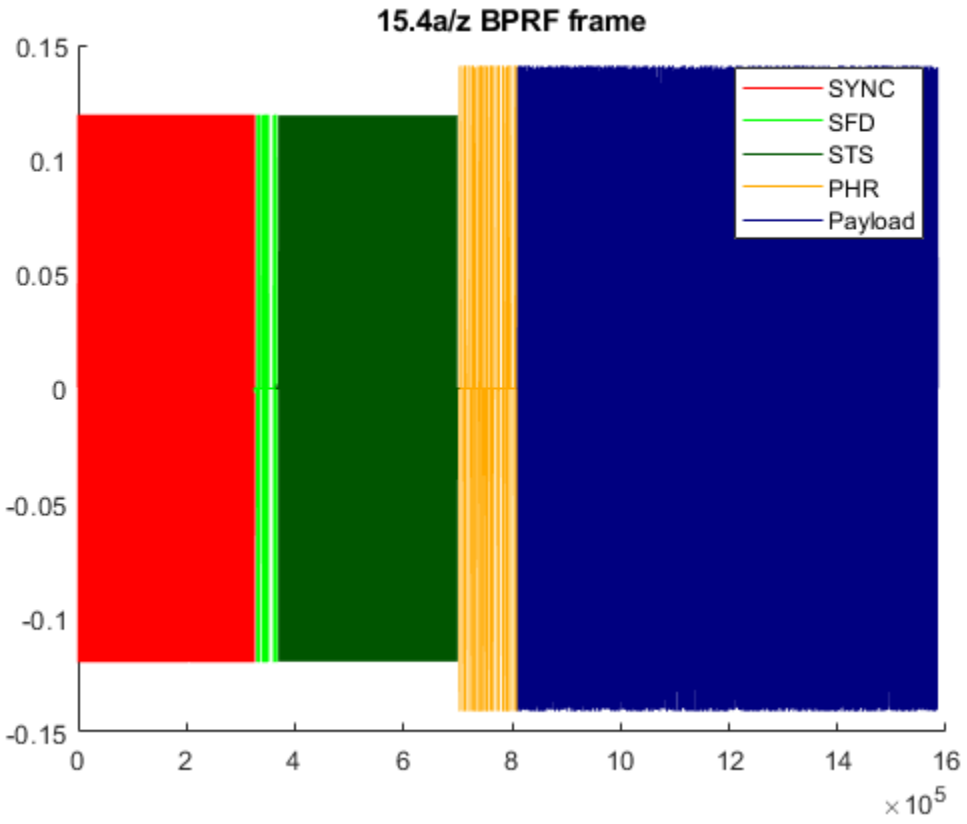
In the base pulse repetition frequency (**BPRF**) mode, mean PRF is 62.4 MHz and data rate is 6.81 Mbps.

The key difference between the BPRF and the HPRF mode is that in BPRF the PHR and the payload are modulated with the burst position modulation (BPM) BPSK technique.

```
msg = randi([0 1],1016,1);
cfgBPRF = lrwpanHRPConfig(Mode='BPRF',CodeIndex=9) %#ok<NOPTS>
waveBPRF = lrwpanWaveformGenerator(msg,cfgBPRF);
lrwpanPlotFrame(waveBPRF,cfgBPRF);
```

```
cfgBPRF =

  lrwpanHRPConfig with properties:
```

```
                Channel: 0
                   Mode: 'BPRF'
            PHRDataRate: '0.85Mbps'
         SamplesPerPulse: 10
  STSPacketConfiguration: 1
              CodeIndex: 9
       PreambleDuration: 64
              SFDNumber: 0
                Ranging: 0
             PSDULength: 1016

Read-only properties:
                PeakPRF: '499.2MHz'
          BurstsPerSymbol: 8
             NumHopBursts: 2
            ChipsPerBurst: [64 8]
           ChipsPerSymbol: [512 64]
      ConvolutionalCoding: 1
       PreambleCodeLength: 127
  PreambleSpreadingFactor: 4
              SampleRate: 4.9920e+09
```



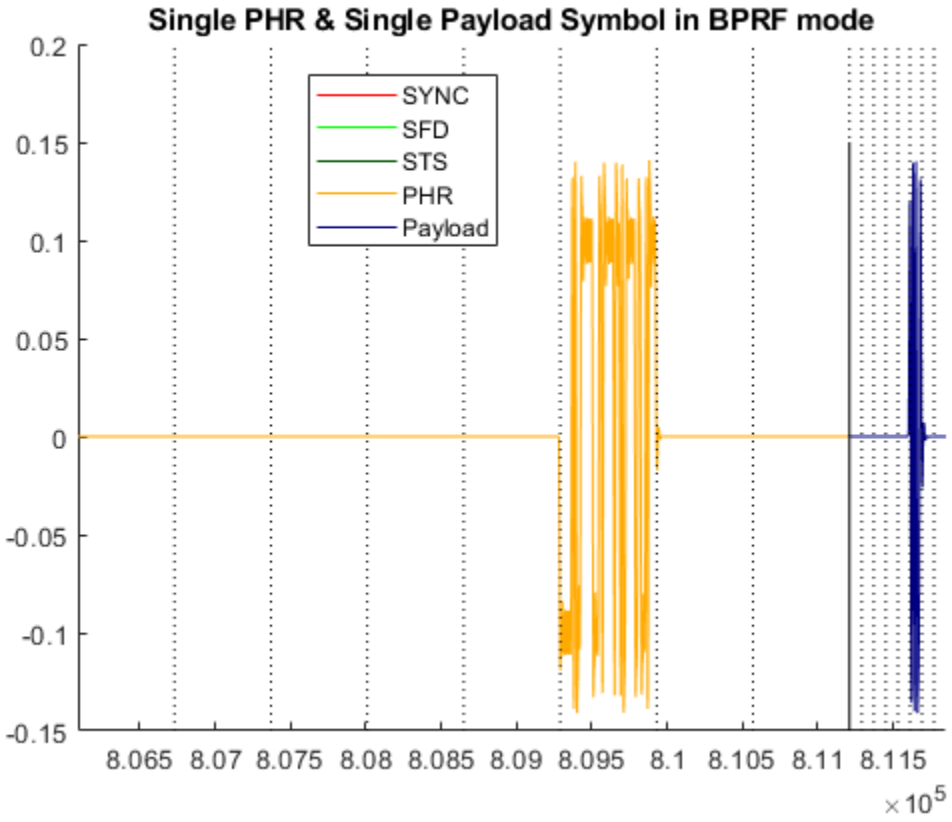
BPRF frames include the SYNC and SFD fields and is BPM-BPSK modulated.

*The SYNC field is constructed similar to HPRF mode, but the selected code can be 127 symbols long, so the `CodeIndex` property setting can be as low as 9.

*The SFD field is always 8 symbols long.

*Sec. 15.3 of [1] specifies the BPM-BPSK modulation scheme. As shown by this code, a single PHR and a single payload symbol appear together under BPM-BPSK modulation for the BPRF mode.

```
fig = lrwpanPlotFrame(waveBPRF,cfgBPRF);
hZoomToBPMBPSKSymbols(fig,cfgBPRF);
```



In the plot, solid black vertical line separates the PHR and payload symbol durations, and dashed lines separate different candidate burst positions. In BPM-BPSK modulation, each symbol duration is divided in 4 quarters, and transmission can occur either in the 1st or the 3rd quarter. The systematic bit of the convolutional codeword determines when transmissions occur. Each quarter is divided into 2, 8, or 32 candidate bursts as specified by the `NumHopBursts` property and determined by the mean PRF. When the mean PRF is 62.4 MHz (BPRF), the number of candidate active bursts in a quarter symbol is 2, which corresponds to a total of 8 burst durations per symbol. In the plotted PHR and payload symbols, the active transmissions occupy 1/8 of their symbol durations. A PN sequence determines the location of the single burst transmissions in the quarter symbol determined by the systematic bit. Specifically, where the burst hopping occurs over time. Within the selected burst position, *Ncpb* chips are transmitted, as specified by the `ChipsPerBurst` property. The first element contains the PHR and the last element contains the payload. The number of chips per burst is determined by the mean PRF and data rate combination.

PHR: The PHR data rate is either 850 kbps or 6.81 Mbps, as determined by the `PHRDataRate` property. A PHR data rate of 850 kbps corresponds to 64 chips per burst and 512 chips per symbol. A PHR data rate of 6.81 Mbps corresponds to 8 chips per burst and 64 chips per symbol. The PHR field has the same length (19 bits) and encoding (SECDED and convolutional) as the HPRF mode.

Payload: As shown in Table 15-9a of [2], for BPRF mode the payload data rate is 6.81 Mbps, which corresponds to 8 chips per burst and 64 chips per symbol duration. Similar to HPRF mode, the payload field uses rate 1/2 convolutional encoding, but for BPRF the constraint length can only be 3.

IEEE 802.15.4a

Similar to BPRF mode, **IEEE 802.15.4a** uses the BPM-BPSK modulation scheme. These are the key differences between legacy 15.4a and the BPRF mode.

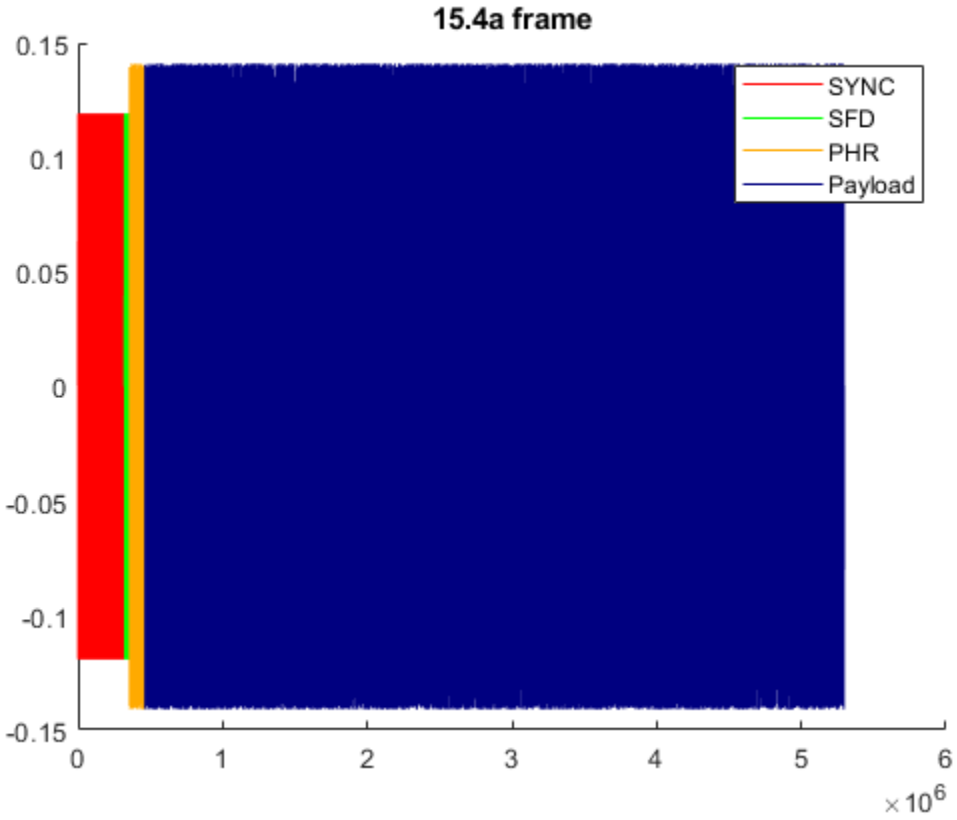
- IEEE 802.15.4a has no STS field.
- The mean PRF of the payload can be 3.9, 15.6, or 62.4 MHz. For mean PRF values of 3.9 or 15.6 MHz, the spreading factor of the SYNC codes (`PreambleSpreadingFactor`) is configurable by the `PreambleMeanPRF` property.
- The payload data rate is dependent on the mean PRF and is not limited 6.81 Mbps. The data rate and mean PRF values can enable different values for the number of hop bursts (`NumHopBursts` can be 2, 8, or 32) and chips per burst (`ChipsPerBurst` can be 1, 2, 4, 8, 16, 32, 64, 128, or 512).
- The data rate of the PHR equals the minimum of 850 kbps and the data rate of the payload. The data rate can be either 110 or 850 kbps.
- Convolutional coding is disabled for the highest data rates (6.81 and 27.24 Mbps) of the 3.9 and 15.6 MHz mean PRF, respectively.

In an 802.15.4a configuration, you can set the mean PRF to 62.4 MHz only when the data rate is not 6.81 Mbps. A data rate of 6.82 MHz corresponds to BPRF mode. When the mean PRF is 3.9 or 15.6 MHz, the code index must be in the range [1, 6]. When the mean PRF is 62.4 MHz, the code index must be in the range [9, 16] or [21, 24].

```
msg = randi([0 1],800,1);
cfg4a = lrwpanHRPConfig( ...
    Mode='802.15.4a', ...
    MeanPRF='15.6MHz', ...
    Channel=3, ...
    CodeIndex=6, ...
    PSDULength=length(msg)) %#ok<NOPTS>
wave4a = lrwpanWaveformGenerator(msg,cfg4a);

lrwpanPlotFrame(wave4a,cfg4a);
```

```
cfg4a =

  lrwpanHRPConfig with properties:

                    Channel: 3
                       Mode: '802.15.4a'
                    MeanPRF: '15.6MHz'
                   DataRate: '0.85Mbps'
             SamplesPerPulse: 10
                   CodeIndex: 6
              PreambleMeanPRF: '16.1MHz'
            PreambleDuration: 64
                     Ranging: 0
                  PSDULength: 800

   Read-only properties:
                    PeakPRF: '499.2MHz'
```

```
        BurstsPerSymbol: 32
           NumHopBursts: 8
          ChipsPerBurst: 16
         ChipsPerSymbol: 512
    ConvolutionalCoding: 1
     PreambleCodeLength: 31
PreambleSpreadingFactor: 16
             SampleRate: 4.9920e+09
```



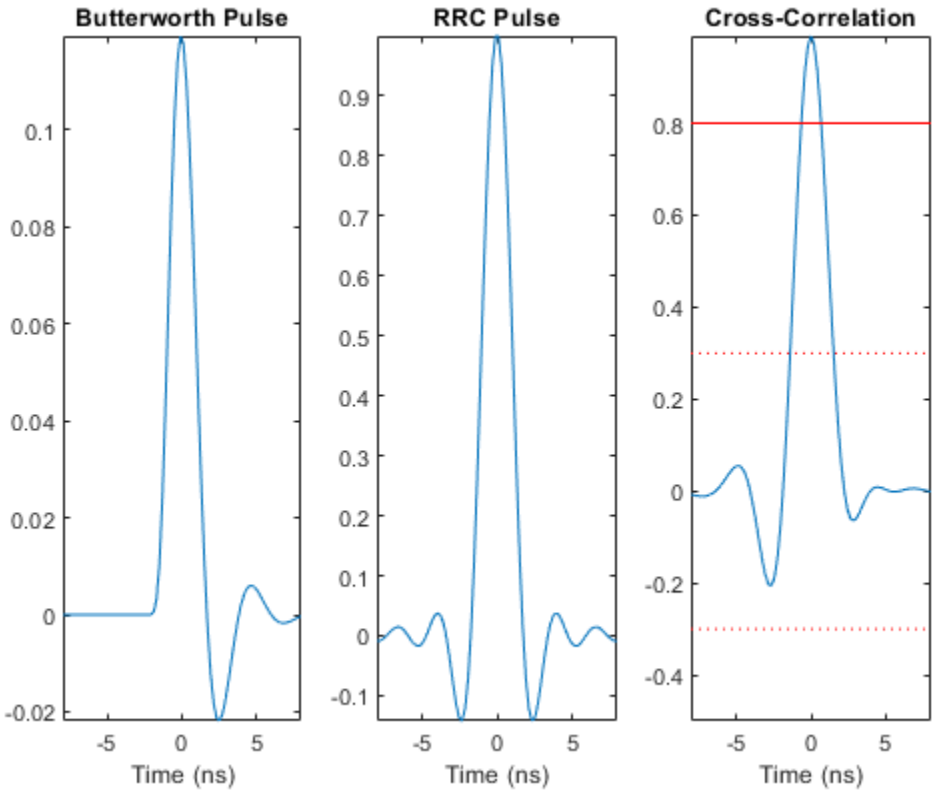
RF Conformance

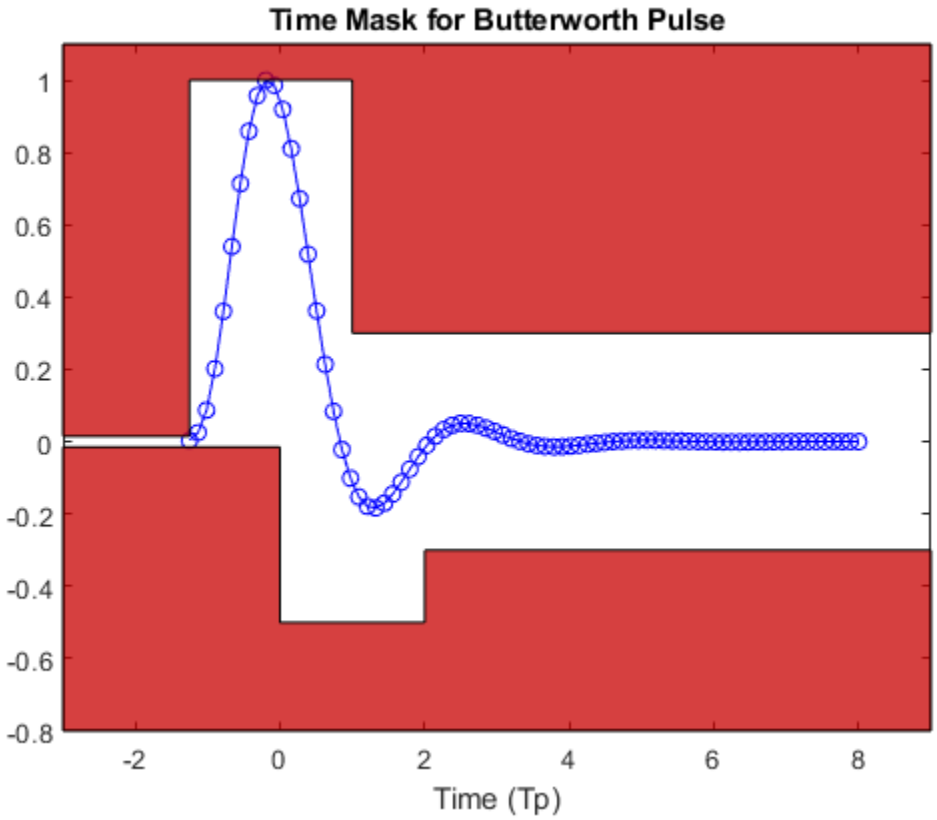
All IEEE 802.15.4a/z waveforms generated in this example are repetitions of Butterworth pulses. Such pulses are obtained by passing a sequence of ternary symbols (-1, 0 or 1) to a Butterworth filter.

The IEEE 802.15.4a/z HRP standard specifies a compliance check for HRP pulses (see Sec. 15.4.4 in [1]). Specifically, the **cross-correlation** between the used pulse and a root raised cosine pulse with a rolloff factor of 0.5, must be higher than 0.8 for 0.5 ns in the main (central) lobe, and all other side lobes must have cross-correlation lower than 0.3. The cross-correlation result is shown in the figure on the left.

The IEEE 802.15.4z amendment specifies that transmitted pulses conform to the **time-domain mask** shown in Fig. 15-13a of [2]. As shown in the figure on the right, the Butterworth pulses used in this example comply with the transmit mask recommendation.

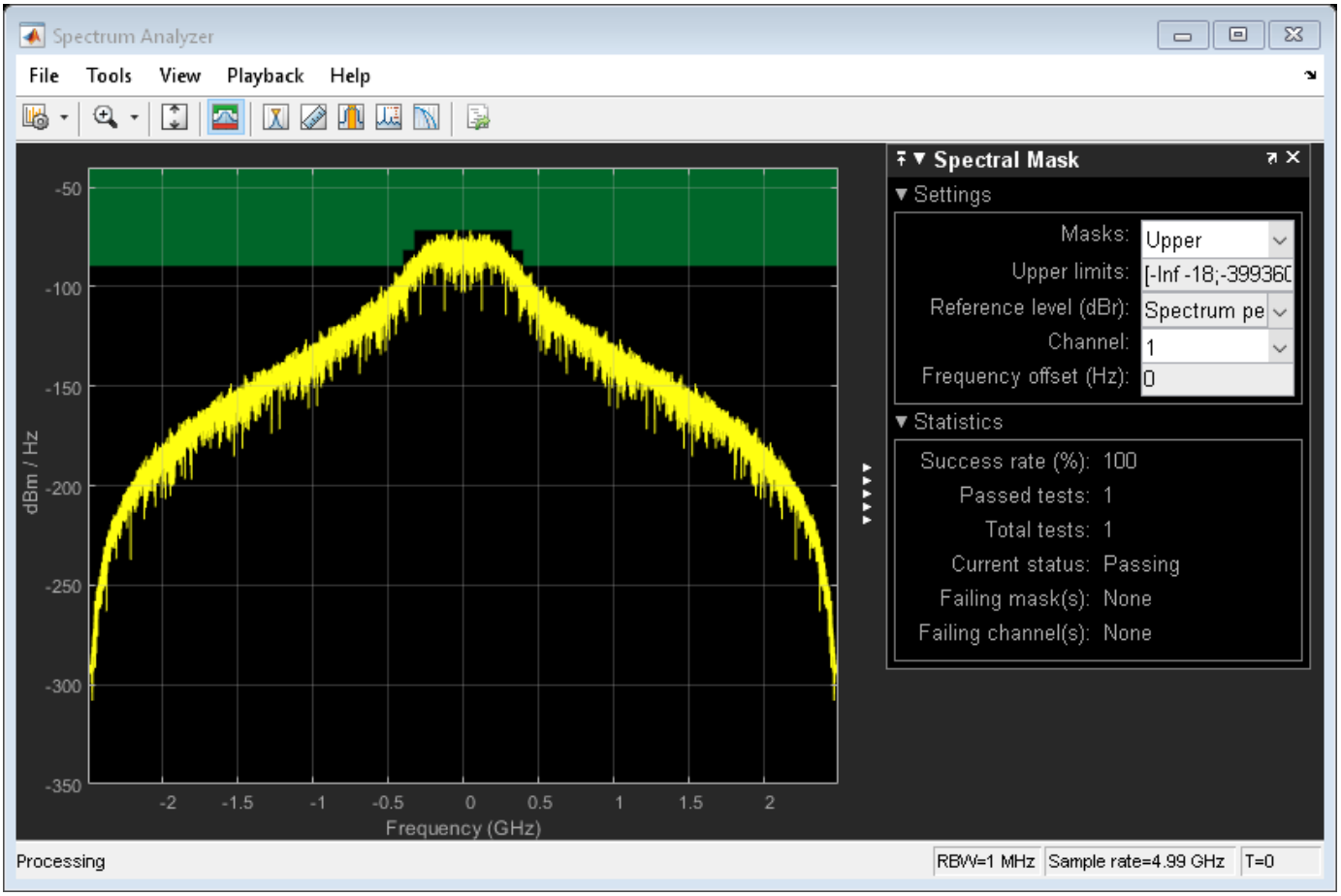
```
lrwpanHRPPulseConformance(cfgHPRF);
```


Butterworth Pulse
RRC Pulse
Cross-Correlation
Time (ns)
Time (ns)
Time (ns)



The IEEE 802.15.4 standard specifies a **mask** for the transmit power **spectral** density PSD (see Sec. 15.4.5 in [1]). The `lrwpanHRPTxPSDMask` helper function displays the spectral density of the generated waveform and examines the conformance to the spectral mask.

```
lrwpanHRPTxPSDMask(waveHPRF, cfgHPRF)
```



Similar results can be attained for the other generated waveforms, using these commands.

```
% lrwpanHRPTxPSDMask(waveBPRF,cfgBPRF)
% lrwpanHRPTxPSDMask(wave4a,cfg4a)
```

Further Exploration

The Communications Toolbox Library for ZigBee and UWB add-on contains the following object and functions:

- **lrwpanHRPConfig:** HRP waveform configuration
- **lrwpanWaveformGenerator:** Create an IEEE 802.15.4a/z HRP UWB waveform
- **lrwpanHRPFieldIndices:** Find starting and ending index for each field of PHY frame
- **lrwpanPlotFrame:** Visualize HRP UWB IEEE 802.15.4a/z waveform

These utilities are undocumented and their API or functionality may change in the future.

Selected Bibliography

1 "IEEE Standard for Low-Rate Wireless Networks," in IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), pp.1-800, 23 July 2020, doi: 10.1109/IEEESTD.2020.9144691.

2 "IEEE Standard for Low-Rate Wireless Networks--Amendment 1: Enhanced Ultra Wideband (UWB) Physical Layers (PHYs) and Associated Ranging Techniques," in IEEE Std 802.15.4z-2020

(Amendment to IEEE Std 802.15.4-2020), pp.1-174, 25 Aug. 2020, doi: 10.1109/IEEESTD.2020.9179124.

EVM Measurements for a 802.15.4 (ZigBee®) System

This example shows how to use the `comm.EVM` System object™ to measure the error vector magnitude (EVM) of a simulated IEEE® 802.15.4 [1] transmitter. IEEE 802.15.4 is the basis for the ZigBee specifications.

Error Vector Magnitude (EVM)

The error vector magnitude (EVM) is a measure of the difference between a reference waveform, which is the error-free modulated signal, and the actual transmitted waveform. EVM is used to quantify the modulation accuracy of a transmitter. [1] requires that a 802.15.4 transmitter shall not have an RMS EVM value worse than 35%.

System Parameters

An 802.15.4 system for 868 MHz band employs direct sequence spread spectrum (DSSS) with binary phase-shift keying (BPSK) used for chip modulation and differential encoding used for data symbol encoding.

```
dataRate = 20e3;   % Bit rate in Hz
M = 2;             % Modulation order (BPSK)
chipValues = [1;1;1;1;0;1;0;1;1;0;0;1;0;0;0];
                   % Chip values for bit 0.
                   % Chip values for 1 is the opposite.
```

Section 6.7.3 of [1] specifies that the measurements are performed over 1000 samples of I and Q baseband outputs. To account for filter delays, we include 1 more bit in the simulation of the transmitted symbols. We chose to oversample the transmitted signal by four. We assume an SNR of 60 dB to account for transmitter and test hardware imperfections.

```
numSymbols = 1000;          % Number of symbols required for one EVM value
numFrames = 100;            % Number of frames
nSamps = 4;                 % Number of samples that represents a symbol
filtSpan = 8;               % Filter span in symbols
gain = length(chipValues);  % Spreading gain (number of chips per symbol)
chipRate = gain*dataRate;   % Chip rate
sampleRate = nSamps*chipRate;    % Final sampling rate
numBits = ceil((numSymbols)/gain)+1;
                            % Number of bits required for one EVM value
SNR = 60;                   % Simulated signal-to-noise ratio in dB
```

Initialization

We can obtain BPSK modulated symbols with a simple mapping of 0 to +1 and 1 to -1. If we also map the chip values, then we can modulate before bit-to-chip conversion and use matrix math to write efficient MATLAB® code. ZigBee specifications also define the pulse shaping filter as having a raised cosine pulse shape with rolloff factor of 1.

```
% Map chip values
chipValues = 1 - 2*chipValues;

% Design a raised cosine filter with rolloff factor 1
rctFilt = comm.RaisedCosineTransmitFilter('RolloffFactor', 1, ...
  'OutputSamplesPerSymbol', nSamps, ...
  'FilterSpanInSymbols', filtSpan);
rcrFilt = comm.RaisedCosineReceiveFilter('RolloffFactor', 1, ...
```

```
  'InputSamplesPerSymbol', nSamps, ...
  'FilterSpanInSymbols', filtSpan, ...
  'DecimationFactor', nSamps);
```

EVM Measurements

The Communications Toolbox™ provides comm.EVM to calculate RMS EVM, Maximum EVM, and Xth percentile EVM values. Section 6.7.3 of [1] defines the EVM calculation method, where the average error of measured I and Q samples are normalized by the power of a symbol. For a BPSK system, the power of both constellation symbols is the same. Therefore, we can use the 'Peak constellation power' normalization option. Other available normalization options, which can be used with other communications system standards, are average constellation power and average reference signal power.

```
evm = comm.EVM('Normalization', 'Peak constellation power')
```

```
evm =

  comm.EVM with properties:

                Normalization: 'Peak constellation power'
      PeakConstellationPower: 1
       ReferenceSignalSource: 'Input port'
    MeasurementIntervalSource: 'Input length'
          AveragingDimensions: 1
        MaximumEVMOutputPort: false
     XPercentileEVMOutputPort: false
```

Simulation

We first generate random data bits, differentially encode these bits using a `comm.DifferentialEncoder` System object, and modulate using BPSK. We spread the modulated symbols by employing a matrix multiplication with the mapped chip values. The spread symbols are then passed through a pulse shaping filter. The EVM object assumes that received symbols, rd, and reference symbols, c, are synchronized, and sampled at the same rate. We downsample the received signal, r, and synchronize with the reference signal, c.

[1] requires that 1000 symbols be used in one RMS EVM calculation. To ensure we have enough averaging, we simulate 100 frames of 1000 symbols and use the maximum of these 100 RMS EVM measurements as the measurement result. We see that the simulated transmitter meets the criteria mentioned in Error Vector Magnitude section above.

```
% Tx and Rx filter delays are identical and equal to half the filter span.
% Total delay is equal to the sum of two filter delays, which is the filter
% span of one filter.
refSigDelay = rctFilt.FilterSpanInSymbols;

% Simulated number of symbols in a frame
simNumSymbols = numBits*gain;

% Initialize peak RMS EVM
peakRMSEVM = -inf;

% Create a comm.DifferentialEncoder object to differentially encode data
diffenc = comm.DifferentialEncoder;
```

```
% Create an comm.AWGNChannel System object and set its NoiseMethod property
% to 'Signal to noise ratio (SNR)'
chan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', SNR);

% Loop over bursts
for p=1:numFrames
    % Generate random data
    b = randi([0 M-1], numBits, 1);
    % Differentially encode
    d = diffenc(b);
    % Modulate
    x = 1-2*d;
    % Convert symbols to chips (spread)
    c = reshape(chipValues*x', simNumSymbols, 1);
    % Pulse shape
    cUp = rctFilt(c);
    % Calculate and set the 'SignalPower' property of the channel object
    chan.SignalPower = sum(cUp.^2)/length(cUp);
    % Add noise
    r = chan(cUp);
    % Downsample received signal.  Account for the filter delay.
    rd = rcrFilt(r);
    % Measure using the EVM System object
    rmsEVM = evm(complex(rd(refSigDelay+(1:numSymbols))), ...
      complex(c(1:numSymbols)));
    % Update peak RMS EVM calculation
    if (peakRMSEVM < rmsEVM)
        peakRMSEVM = rmsEVM;
    end
end

% Display results
fprintf(' Worst case RMS EVM (%%): %1.2f\n', peakRMSEVM)
```

```
 Worst case RMS EVM (%): 0.19
```

Comments

We showed how to utilize comm.EVM to test if a ZigBee transmitter complies with the standard specified EVM values. We used a crude model that only introduces additive white Gaussian noise and showed that the measured EVM is less than the standard specified 35%.

Selected Bibliography

1 IEEE Standard 802.15.4, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks, 2003.

End-to-End IEEE 802.15.4 PHY Simulation

This example shows how to generate waveforms, decode waveforms and compute BER curves for different PHY specifications of the IEEE® 802.15.4™ standard [1], using the Communications Toolbox™ Library for ZigBee and UWB.

Background

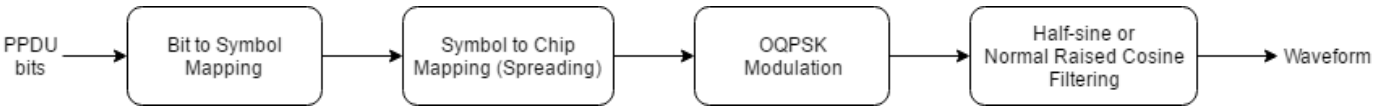
The **IEEE 802.15.4** standard specifies the **PHY** and **MAC** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 PHY and MAC layers provide the basis of other higher-layer standards, such as **ZigBee**, WirelessHart®, 6LoWPAN and MiWi. Such standards find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

Physical Layer Implementations of IEEE 802.15.4

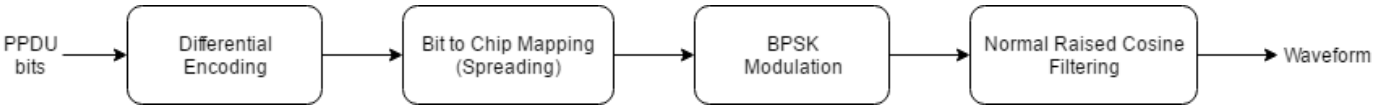
The original IEEE 802.15.4 standard and its amendments specify multiple PHY layers, which use different modulation schemes and support different data rates. These physical layers were devised for specific frequency bands and, to a certain extent, for specific countries. This example provides functions that generate and decode waveforms for the physical layers proposed in the original IEEE 802.15.4 specification (OQPSK in 2.4 GHz, BPSK in 868/915 MHz), IEEE 802.15.4b (OQPSK and ASK in 868/915 MHz), IEEE 802.15.4c (OQPSK in 780 MHz) and IEEE 802.15.4d (GFSK and BPSK in 950 MHz).

These physical layers specify a format for the PHY protocol data unit (PPDU) that includes a preamble, a start-of-frame delimiter (SFD), and the length and contents of the MAC protocol data unit (MPDU). The preamble and SFD are used for frame-level synchronization. In the following description, the term symbol denotes the integer index of a chip sequence (as per the IEEE 802.15.4 standard), not a modulation symbol (i.e., a complex number).

- **OQPSK PHY**: All OQPSK PHYs map every 4 PPDU bits to one symbol. The 2.4 GHz OQPSK PHY spreads each symbol to a 32-chip sequence, while the other OQPSK PHYs spread it to a 16-chip sequence. Then, the chip sequences are OQPSK modulated and passed to a half-sine pulse shaping filter (or a normal raised cosine filter, in the 780 MHz band). For a detailed description, see Clause 10 in [1].

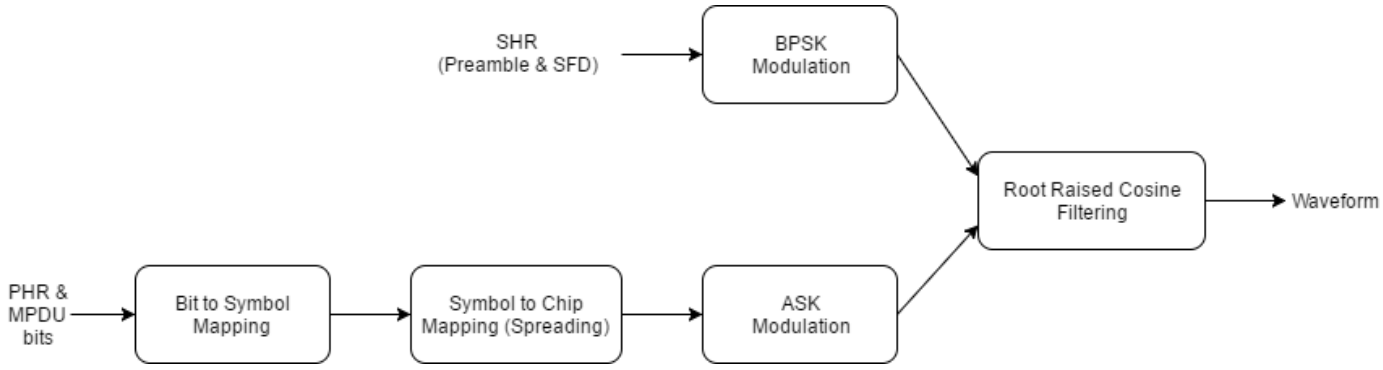


- **BPSK PHY**: The BPSK PHY differentially encodes the PPDU bits. Each resulting bit is spread to a 15-chip sequence. Then, the chip sequences are BPSK modulated and passed to a normal raised cosine filter. For a detailed description, see Clause 11 in [1].

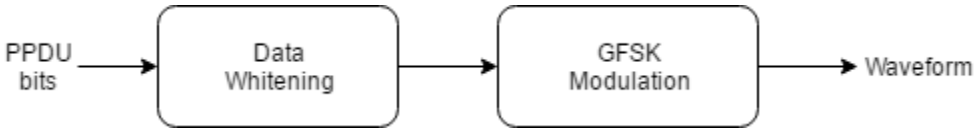


- **ASK PHY**: The ASK PHY uses BPSK modulation for the preamble and the SFD only. The remaining PPDU bits, i.e., the PHY header (PHR) and the MPDU, are first mapped to 20-bit symbols in the 868 MHz band and to 5-bit symbols in the 915 MHz band. Each symbol is spread to a 32-chip sequence using a technique known as Parallel Sequence Spread Spectrum (PSSS) or Orthogonal

Code Division Multiplexing (OCDM). The chip sequence is then ASK modulated and passed to a root raised cosine filter. For a detailed description, see Clause 12 in [1].



- **GFSK PHY**: The GFSK PHY first whitens the PPDU bits using modulo-2 addition with a PN9 sequence. The whitened bits are then GFSK modulated. For a detailed description, see Clause 15 in [1].

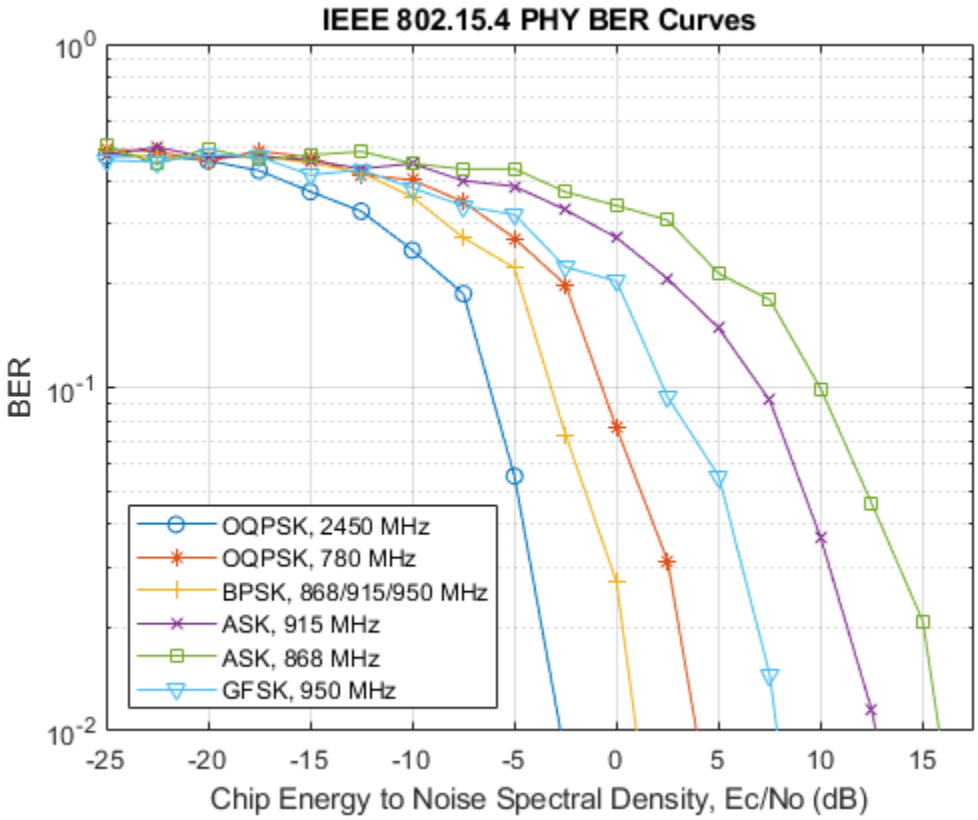


Waveform Generation, Decoding and BER Curve Calculation

This code illustrates how to use the waveform generation and decoding functions for different frequency bands and compares the corresponding BER curves.

```
EcNo = -25:2.5:17.5;                % Ec/No range of BER curves
spc = 4;                            % samples per chip
msgLen = 8*120;                     % length in bits
message = randi([0 1], msgLen, 1);  % transmitted message

% Preallocate vectors to store BER results:
[berOQPSK2450, berOQPSK780, berBPSK, berASK915, ...
 berASK868, berGFSK] = deal(zeros(1, length(EcNo)));

for idx = 1:length(EcNo) % loop over the EcNo range

  % O-QPSK PHY, 2450 MHz
  waveform = lrwpan.PHYGeneratorOQPSK(message, spc, '2450 MHz');
  K = 2;      % information bits per symbol
  SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
  received = awgn(waveform, SNR);
  bits     = lrwpan.PHYDecoderOQPSKNoSync(received,  spc, '2450 MHz');
  [~, berOQPSK2450(idx)] = biterr(message, bits);

  % O-QPSK PHY, 780MHz
  waveform = lrwpan.PHYGeneratorOQPSK(message, spc, '780 MHz'); % or '868 MHz'/'915 MHz'
  SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
  received = awgn(waveform, SNR);
  bits     = lrwpan.PHYDecoderOQPSKNoSync(received,  spc, '780 MHz'); % or '868 MHz'/'915 MHz'
  [~, berOQPSK780(idx)] = biterr(message, bits);
```

```
    % BPSK PHY, 868/915/950 MHz
    waveform = lrwpan.PHYGeneratorBPSK(message, spc);
    K = 1;      % information bits per symbol
    SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
    received = awgn(waveform, SNR);
    bits     = lrwpan.PHYDecoderBPSK(received, spc);
    [~, berBPSK(idx)] = biterr(message, bits);

    % ASK PHY, 915 MHz
    waveform = lrwpan.PHYGeneratorASK(message, spc, '915 MHz');
    K = 1;      % information bits per symbol
    SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
    received = awgn(waveform, SNR);
    bits     = lrwpan.PHYDecoderASK(received,  spc, '915 MHz');
    [~, berASK915(idx)] = biterr(message, bits(1:msgLen));

    % ASK PHY, 868 MHz
    waveform = lrwpan.PHYGeneratorASK(message, spc, '868 MHz');
    K = 1;      % information bits per symbol
    SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
    received = awgn(waveform, SNR);
    bits     = lrwpan.PHYDecoderASK(received,  spc, '868 MHz');
    [~, berASK868(idx)] = biterr(message, bits(1:msgLen));

    % GFSK PHY, 950 MHz
    waveform = lrwpan.PHYGeneratorGFSK(message, spc);
    K = 1;      % information bits per symbol
    SNR = EcNo(idx) - 10*log10(spc) + 10*log10(K);
    received = awgn(waveform, SNR);
    bits     = lrwpan.PHYDecoderGFSK(received, spc);
    [~, berGFSK(idx)] = biterr(message, bits);
  end

  % plot BER curve
  figure
  semilogy(EcNo, berOQPSK2450, '-o', EcNo, berOQPSK780, '-*', EcNo, berBPSK, '-+', ...
           EcNo, berASK915,    '-x', EcNo, berASK868,    '-s', EcNo, berGFSK, '-v')
  legend('OQPSK, 2450 MHz', 'OQPSK, 780 MHz', 'BPSK, 868/915/950 MHz', 'ASK, 915 MHz', ...
         'ASK, 868 MHz', 'GFSK, 950 MHz', 'Location', 'southwest')
  title('IEEE 802.15.4 PHY BER Curves')
  xlabel('Chip Energy to Noise Spectral Density, Ec/No (dB)')
  ylabel('BER')
  axis([min(EcNo) max(EcNo) 10^-2 1])
  grid on
```



Further Exploration

You can further explore the following generator and decoding functions:

- lrwpan.PHYGeneratorOQPSK, lrwpan.PHYDecoderOQPSKNoSync and lrwpan.PHYDecoderOQPSK
- lrwpan.PHYGeneratorBPSK and lrwpan.PHYDecoderBPSK
- lrwpan.PHYGeneratorASK and lrwpan.PHYDecoderASK
- lrwpan.PHYGeneratorGFSK and lrwpan.PHYDecoderGFSK

Selected Bibliography

1 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

Recovery of IEEE 802.15.4 OQPSK Signals

This example shows how to implement a **practical** IEEE® 802.15.4™ PHY receiver decoding OQPSK waveforms that may have been received from wireless radios, using the Communications Toolbox™ Library for for ZigBee and UWB. This practical receiver has decoded standard-compliant waveforms received from commercial ZigBee radios enabling home automation in the 2.4 GHz band, using a USRP® B200-mini radio and the Communications Toolbox Support Package for USRP® radio.

Background

The **IEEE 802.15.4** standard specifies the **MAC** and **PHY** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 MAC and PHY layers provide the basis of other higher-layer standards, such as **ZigBee**, WirelessHart®, 6LoWPAN and MiWi. Such standards find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

Receiver Architecture

Overall, the receiver performs the following operations:

- Matched filtering
- Coarse frequency compensation
- Fine frequency compensation
- Timing Recovery
- Preamble detection
- Phase ambiguity resolution
- Despreading

Between these steps, the signal is visualized to illustrate the signal impairments and the corrections.

Matched Filtering

```
load lrwpanPHYCaptures % load OQPSK signals captured in the 2.4 GHz band
spc = 12;  % 12 samples per chip; the frame was captured at 12 x chiprate = 12 MHz
```

A matched filter improves the SNR of the signal. The 2.4 GHz OQPSK PHY uses half-sine pulses, therefore the following matched filtering operation is needed.

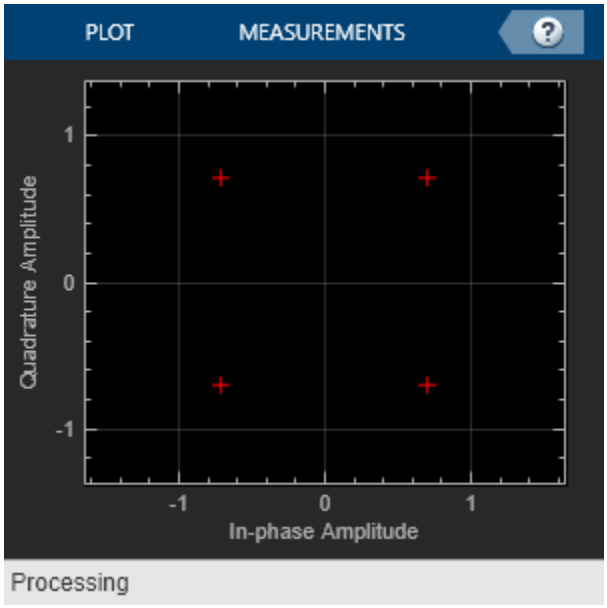
```
% Matched filter for captured OQPSK signal:
halfSinePulse = sin(0:pi/spc:(spc)*pi/spc);
decimationFactor = 3; % reduce spc to 4, for faster processing
matchedFilter = dsp.FIRDecimator(decimationFactor, halfSinePulse);
filteredOQPSK = matchedFilter(capturedFrame1); % matched filter output
```

Frequency Offsets

Decoding a signal under the presence of frequency offsets is a challenge for any wireless receiver. Frequency offsets up to 30 kHz were measured for signals transmitted from commercial ZigBee radios and captured using a USRP® B200-mini radio.

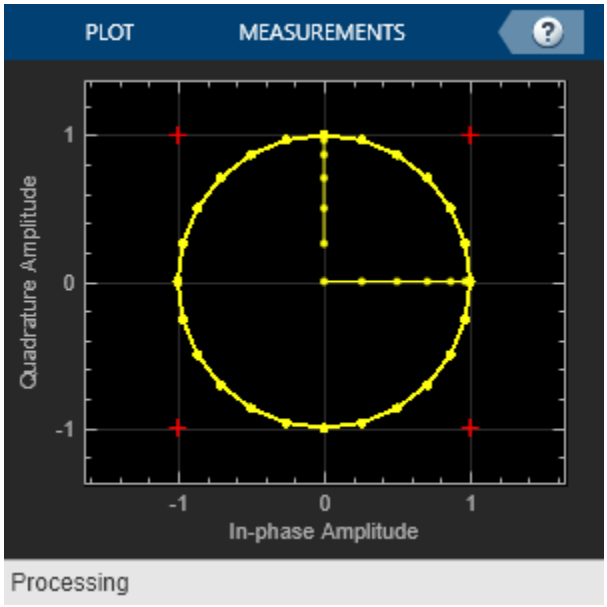
Constellation diagrams can illustrate the quality of the received signal, but it is first important to note that the trajectory of an ideal OQPSK signal follows a circle.

```
% Plot constellation of ideal OQPSK signal
msgLen = 8*120;                     % length in bits
message = randi([0 1], msgLen, 1);  % transmitted message
idealOQPSK = lrwpan.PHYGeneratorOQPSK(message, spc, '2450 MHz');
constellation = comm.ConstellationDiagram('Name', 'Ideal OQPSK Signal', 'ShowTrajectory', true);
constellation.Position = [constellation.Position(1:2) 300 300];
constellation(idealOQPSK);
```



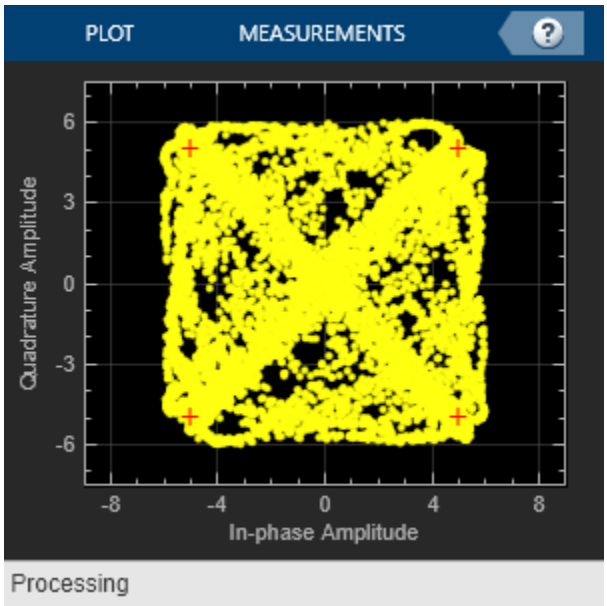
The above constellation also contains one radius corresponding to the frame start, and one radius corresponding to the frame end. At the same time, frequency offsets circularly rotate constellations, resulting in ring-shaped constellations as well. Therefore, it is more meaningful to observe the constellation of a QPSK-equivalent signal that is obtained by delaying the in-phase component of the OQPSK signal by half a symbol. When half-sine pulse filtering is used, and the oversampling factor is greater than one, the ideal QPSK constellation resembles an 'X'-shaped region connecting the four QPSK symbols (red crosses) with the origin.

```
% Plot constellation of ideal QPSK-equivalent signal
idealQPSK = complex(real(idealOQPSK(1:end-spc/2)), imag(idealOQPSK(spc/2+1:end))); % align I and
release(constellation);
constellation.Name = 'Ideal QPSK-Equivalent Signal';
constellation.ReferenceConstellation = [1+1i 1-1i 1i-1 -1i-1];
constellation(idealQPSK);
```



However, the samples of the captured frame are dislocated from this 'X'-shaped region due to frequency offsets:

```
% Plot constellation of QPSK-equivalent (impaired) received signal
filteredQPSK = complex(real(filteredOQPSK(1:end-spc/(2*decimationFactor))), imag(filteredOQPSK(sp
constellation = comm.ConstellationDiagram('XLimits', [-7.5 7.5], 'YLimits', [-7.5 7.5], ...
                                          'ReferenceConstellation', 5*qammod(0:3, 4), 'Name', 'Re
constellation.Position = [constellation.Position(1:2) 300 300];
constellation(filteredQPSK);
```

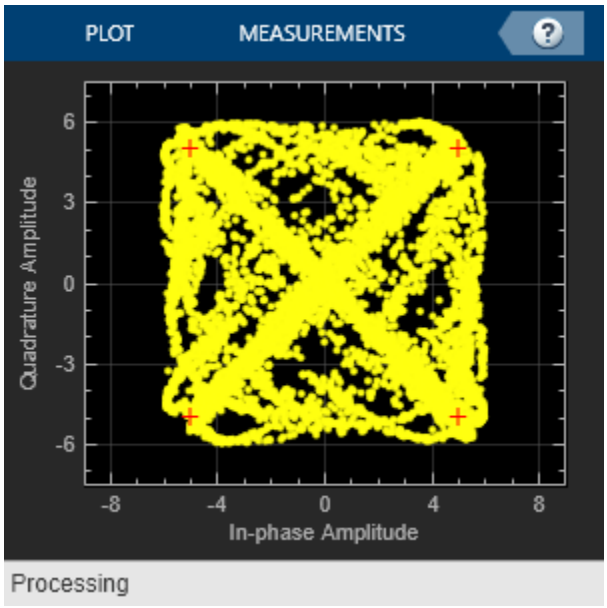


Coarse Frequency Compensation

Such frequency offsets are first coarsely corrected using an FFT-based method [2] that squares the OQPSK signal and reveals two spectral peaks. The coarse frequency offset is obtained by averaging and halving the frequencies of the two spectral peaks.

```
% Coarse frequency compensation of OQPSK signal
coarseFrequencyCompensator = comm.CoarseFrequencyCompensator('Modulation', 'OQPSK', ...
      'SampleRate', spc*1e6/decimationFactor, 'FrequencyResolution', 1e3);
[coarseCompensatedOQPSK, coarseFrequencyOffset] = coarseFrequencyCompensator(filteredOQPSK);
fprintf('Estimated frequency offset = %.3f kHz\n', coarseFrequencyOffset/1000);

% Plot QPSK-equivalent coarsely compensated signal
coarseCompensatedQPSK = complex(real(coarseCompensatedOQPSK(1:end-spc/(2*decimationFactor))), ima
release(constellation);
constellation.Name = 'Coarse frequency compensation (QPSK-Equivalent)';
constellation(coarseCompensatedQPSK);
```

```
Estimated frequency offset = 26.367 kHz
```



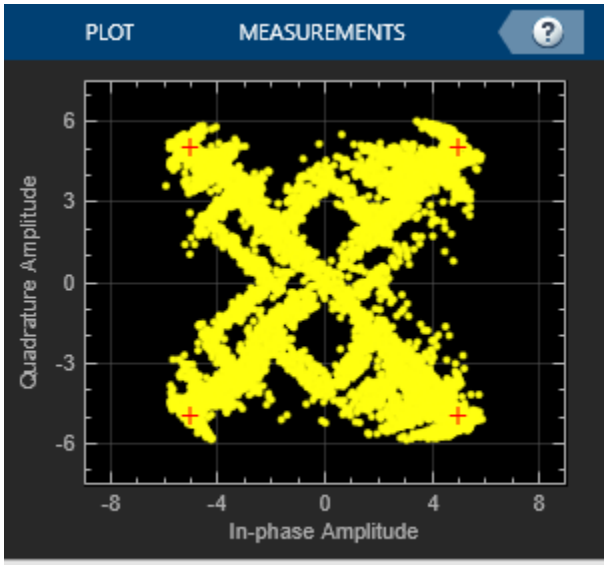
Some samples still lie outside the 'X'-shaped region connecting the origin with the QPSK symbols (red crosses), as fine frequency compensation is also needed.

Fine Frequency Compensation

Fine frequency compensation follows the **OQPSK carrier-recovery algorithm** described in [3]. This algorithm is behaviorally different than its QPSK counterpart, which does not apply to OQPSK signals even if their in-phase signal component is delayed by half a symbol.

```
% Fine frequency compensation of OQPSK signal
fineFrequencyCompensator = comm.CarrierSynchronizer('Modulation', 'OQPSK', 'SamplesPerSymbol', sp
fineCompensatedOQPSK = fineFrequencyCompensator(coarseCompensatedOQPSK);

% Plot QPSK-equivalent finely compensated signal
fineCompensatedQPSK = complex(real(fineCompensatedOQPSK(1:end-spc/(2*decimationFactor))), imag(fi
release(constellation);
```

```
constellation.Name = 'Fine frequency compensation (QPSK-Equivalent)';
constellation(fineCompensatedQPSK);
```

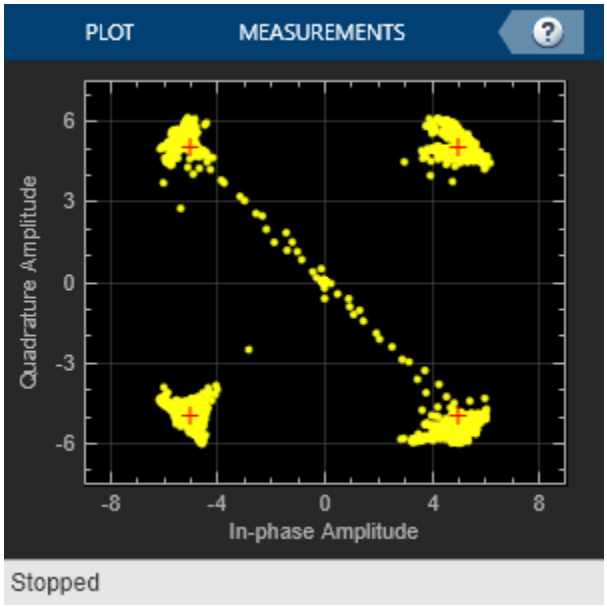


The constellation is now closer to its ideal form, but still timing recovery is needed.

Timing Recovery

Symbol synchronization occurs according to the OQPSK timing-recovery algorithm described in [3]. In contrast to carrier recovery, the OQPSK timing recovery algorithm is equivalent to its QPSK counterpart for QPSK-equivalent signals that are obtained by delaying the in-phase component of the OQPSK signal by half a symbol.

```
% Timing recovery of OQPSK signal, via its QPSK-equivalent version
symbolSynchronizer = comm.SymbolSynchronizer('Modulation', 'OQPSK', 'SamplesPerSymbol', spc/decir
syncedQPSK = symbolSynchronizer(fineCompensatedOQPSK);

% Plot QPSK symbols (1 sample per chip)
release(constellation);
constellation.Name = 'Timing Recovery (QPSK-Equivalent)';
constellation(syncedQPSK);
```



Note that the output of the Symbol Synchronizer contains one sample per symbol. At this stage, the constellation truly resembles a QPSK signal. The few symbols that gradually move away from the origin correspond to the frame start and end.

Preamble Detection, Despreading and Phase Ambiguity Resolution:

Once the signal has been synchronized, the next step is preamble detection, which is more successful if the signal has been despreaded. It is worth noting that fine frequency compensation results in a $\pi/2$-phase ambiguity, indicating the true constellation may have been rotated by 0, $\pi/2$, π, or $3\pi/2$ radians. Preamble detection resolves the phase ambiguity by considering all four possible constellation rotations. The next function operates on the synchronized OQPSK signal, performs joint despreading, resolution of phase ambiguity and preamble detection, and then outputs the MAC protocol data unit (MPDU).

```
MPDU = lrwpan.PHYDecoderOQPSKAfterSync(syncedQPSK);
```

```
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- lrwpan.PHYDecoderOQPSKAfterSync and lrwpan.PHYDecoderOQPSK

Selected Bibliography

1 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

2 "Designing an OQPSK demodulator", Jonathan Olds.

3 Rice, Michael. *Digital Communications - A Discrete-Time Approach*. 1st ed. New York, NY: Prentice Hall, 2008.

IEEE 802.15.4 - MAC Frame Generation and Decoding

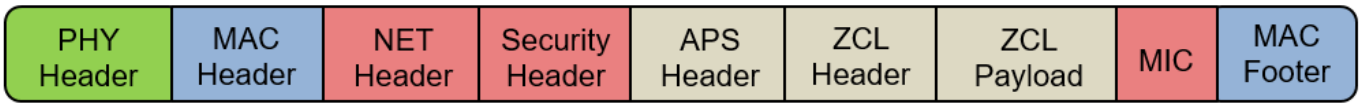
This example shows how to generate and decode MAC frames of the IEEE® 802.15.4™ standard [1] using the Communications Toolbox™ Library for ZigBee and UWB.

Background

The **IEEE 802.15.4** standard specifies the **MAC** and **PHY** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 MAC and PHY layers provide the basis of other higher-layer standards, such as **ZigBee**, WirelessHart®, 6LoWPAN and MiWi. Such standards find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

Architecture

The IEEE 802.15.4 MAC layer inserts a MAC header and a MAC footer before and after a network-layer frame, respectively. The MAC footer contains a CRC check.



Format of IEEE 802.15.4 / ZigBee Frame

A lrwpan.MACFrameConfig configuration object is used both in generating and decoding IEEE 802.15.4 MAC frames. Such objects describe a MAC frame and specify its frame type and all applicable properties. The lrwpan.MACFrameGenerator function accepts a lrwpan.MACFrameConfig object describing the frame, and optionally a MAC-layer payload (NET-layer frame) in bytes (two-characters), and outputs the MAC frame in bits. The lrwpan.MACFrameDecoder function accepts a MAC Protocol Data Unit (MPDU) in bits and outputs a lrwpan.MACFrameConfig object describing the frame and possibly a NET-layer frame in bytes. Clause 5 in [1] describes the MAC frame formats.

Decoding MAC Frames of Home Automation ZigBee Radios

This section decodes MAC frames transmitted from commercial ZigBee radios enabling home automation, and captured using a USRP® B200-mini radio and the Communications Toolbox Support Package for USRP® radio. The PHY layer of the captured waveforms has been decoded according to the methodology described in the “Recovery of IEEE 802.15.4 OQPSK Signals” on page 7-44 example. The resulting MPDUs are stored into a MAT file:

```
load lrwpanMACCaptures
```

First, a data frame is decoded:

```
[dataFrameMACConfig, netFrame] = lrwpan.MACFrameDecoder(MPDU_data);
if ~isempty(dataFrameMACConfig)
  fprintf('CRC check passed for the MAC frame.\n');
  dataFrameMACConfig
end
```

```
CRC check passed for the MAC frame.

dataFrameMACConfig =

  MACFrameConfig with properties:
```

```
                    FrameType: 'Data'

   General MAC properties:
                 SequenceNumber: 244
         AcknowledgmentRequest: 1
         DestinationAddressing: 'Short address'
      DestinationPANIdentifier: '1E16'
            DestinationAddress: '35EA'
              SourceAddressing: 'Short address'
                 SourceAddress: '0000'
    PANIdentificationCompression: 1
                  FramePending: 0
                  FrameVersion: '2003'
                      Security: 0

   Security properties:
    No properties.

   Beacon properties:
    No properties.

   "MAC Command" properties:
    No properties.
```

Next, an acknowledgment frame is decoded:

```
ackFrameMACConfig = lrwpan.MACFrameDecoder(MPDU_ack)
```

```
ackFrameMACConfig =

  MACFrameConfig with properties:

               FrameType: 'Acknowledgment'

   General MAC properties:
         SequenceNumber: 165
    DestinationAddressing: 'Not present'
         SourceAddressing: 'Not present'
             FramePending: 0
             FrameVersion: '2003'
                 Security: 0

   Security properties:
    No properties.

   Beacon properties:
    No properties.

   "MAC Command" properties:
    No properties.
```

Generating MAC Frames

The lrwpan.MACFrameGenerator function can generate all MAC frame types from the IEEE 802.15.4 standard [1], i.e., 'Beacon', 'Data', 'Acknowledgment', and 'MAC Command' frame types. The MAC

Command frame types can be further specified as: 'Association request', 'Association response', 'Disassociation notification', 'Data request', 'PAN ID conflict notification', 'Orphan notification', 'Beacon request', and 'GTS request'.

This code illustrates how to generate frames for all frame types:

```
% Beacon
beaconConfig = lrwpan.MACFrameConfig('FrameType', 'Beacon');
beaconMACFrame = lrwpan.MACFrameGenerator(beaconConfig);

% Data
dataConfig = lrwpan.MACFrameConfig('FrameType', 'Data');
numOctets = 50;
payload = dec2hex(randi([0 2^8-1], numOctets, 1), 2);
dataMACFrame = lrwpan.MACFrameGenerator(dataConfig, payload);

% Acknowledgment
ackConfig = lrwpan.MACFrameConfig('FrameType', 'Acknowledgment');
ackFrame = lrwpan.MACFrameGenerator(ackConfig);

% MAC Command
commandConfig = lrwpan.MACFrameConfig('FrameType', 'MAC Command');
commandConfig.MACCommand = 'Association request';
% Valid settings for MACCommand also include: 'Association response',
% 'Disassociation notification', 'Data request', 'PAN ID conflict
% notification', 'Orphan notification', 'Beacon request', and 'GTS request'.
commandFrame = lrwpan.MACFrameGenerator(commandConfig);
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- lrwpan.MACFrameGenerator
- lrwpan.MACFrameDecoder
- lrwpan.MACFrameConfig

Selected Bibliography

1 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

IEEE 802.15.4 - Asynchronous CSMA MAC

This example shows how to simulate the IEEE® 802.15.4™ asynchronous CSMA MAC [1] using the Communications Toolbox™ Library for ZigBee and UWB.

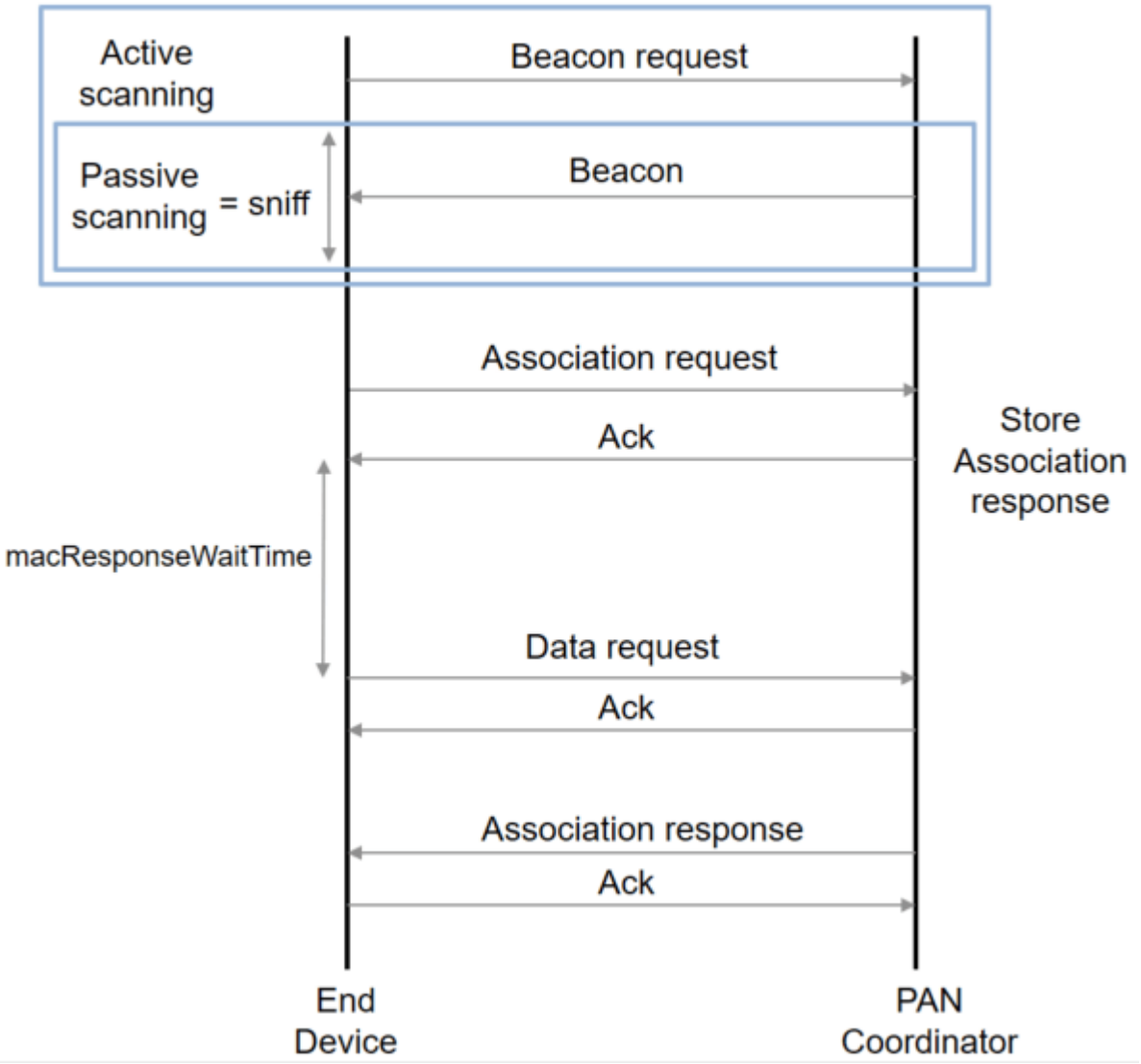
Background

The **IEEE 802.15.4** standard specifies the **MAC** and **PHY** layers of Low-Rate Wireless Personal Area Networks (**LR-WPANs**) [1]. The IEEE 802.15.4 MAC and PHY layers provide the basis of other higher-layer standards, such as **ZigBee**, WirelessHart®, 6LoWPAN and MiWi. Such standards find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

The IEEE 802.15.4 MAC [1] specifies two-basic MAC modes: *(i)* non-beacon-enabled, and *(ii)* beacon-enabled MAC. The non-beacon enabled MAC is an asynchronous CSMA (Carrier-sense Multiple Access) MAC, which is very similar to the IEEE 802.11 MAC. The beacon-enabled MAC allows two different MAC periods: *(i)* a synchronized-CSMA MAC period, and *(ii)* a time-slotted, contention-free MAC period. This example provides an extensive simulation of the non-beacon-enabled, asynchronous, CSMA-based IEEE 802.15.4 MAC.

Network Setup

An IEEE 802.15.4 PAN (personal area network) is set up by a standard process between end devices and PAN coordinators. First, devices that would like to join a network perform either active or passive **scanning**. Active scanning means that a device first transmits a **Beacon Request** and later on it performs passive scanning. Passive scanning means that the device sniffs to collect **beacon frames** from PAN coordinators (who may have received their Beacon Request in the case of active scanning). Upon the collection of beacons during passive scanning, the end device chooses the PAN with which it would like to associate. Then it transmits an **Association Request** to the coordinator of this PAN and the coordinator acknowledges it.



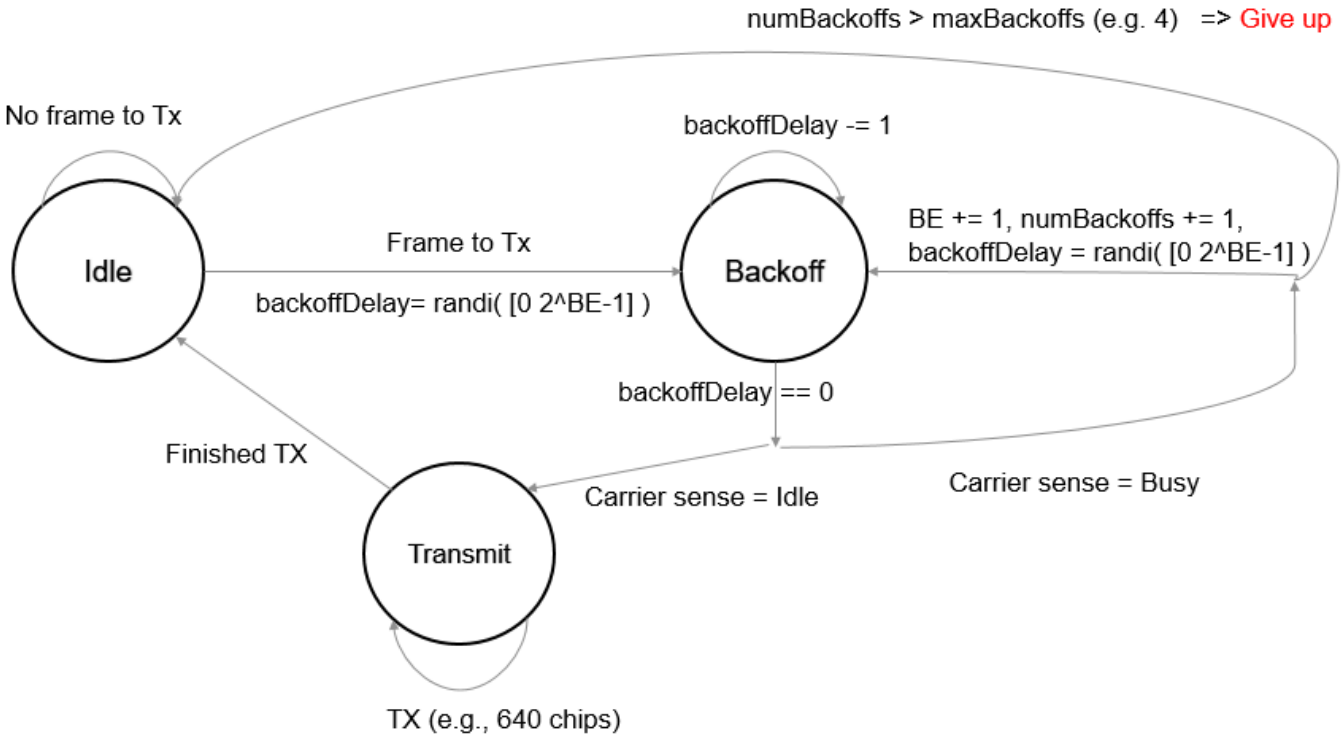
In contrast to IEEE 802.11, the coordinator does not follow the **acknowledgment** of an Association Request with an immediate transmission of an **Association Response**. Instead, the IEEE 802.15.4 coordinator first stores the Association Response locally; it is only transmitted when the end device sends a **Data Request** and the coordinator acknowledges it. The IEEE 802.15.4 standard uses the term **indirect transmission** to refer to this mechanism for transmitting frames. In general, this mechanism is very useful for battery-powered devices of low-traffic networks (e.g., sensor networks). Such devices may periodically activate their radios to check whether any frames are pending for them, instead of continuously using their radios to receive a frame immediately.

Once the Association response is received and acknowledged, the end device is associated with the PAN. At that time, **data frames** can be exchanged between the coordinator and the end device in any direction. The data frames may be acknowledged, depending on their 'Acknowledgment Request' indication.

Asynchronous Medium-Access Control (MAC)

The asynchronous CSMA IEEE 802.15.4 MAC is similar to the generic CSMA operation and the IEEE 802.11 MAC. In this MAC scheme, acknowledgment frames are transmitted immediately, without using the CSMA method. All other frames are transmitted using CSMA.

Specifically, once a device has a frame to transmit, it randomly chooses a **backoff** delay (number of backoff periods) from the range [0 2^BE-1], where BE is the backoff exponent. The duration of each backoff period is 20 symbols. For the OQPSK PHY in 2.4 GHz, this duration corresponds to 640 chips and 0.32 ms. Once the device has waited for the chosen number of backoff periods, it performs **carrier sensing**. If the medium is idle, the device begins transmission of its frame, until it is entirely transmitted.



If the medium is busy during carrier sense, then the backoff exponent increments by 1 and a new number of backoff periods is selected from the new [0 2^BE-1] range. When the backoff counter expires again, carrier sensing is performed. If the maximum number of backoff countdowns is reached without the medium being idle during any carrier sensing instance, then the device terminates its attempts to transmit the frame.

Network Simulation Capabilities

This example offers an implementation for the described network setup process and the CSMA method via the lrwpan.MACFullFunctionDevice and the lrwpan.MACReducedFunctionDevice classes. Specifically, the following capabilities are enabled:

- Active and passive scanning
- Association Request and Association Response exchange

- Indirect transmissions using Data Requests
- Frame acknowledgments and frame retransmissions if acknowledgments are not timely received
- Short and long interframe spacing (SIFS and LIFS)
- Binary exponential backoff
- Carrier sensing

Network Simulation

In this section, we create an IEEE 802.15.4 network of 3 nodes: one PAN coordinator and two end devices. The network simulator is configured to process all devices at increments of a single backoff duration (20 symbols, 0.32 ms).

First, the following code illustrates the association of the first device with the network.

```
symbolsPerStep = 20;
chipsPerSymbol = 32;
samplesPerChip = 4;
symbolRate = 65.5e3; % symbols/sec
time = 0;
stopTime = 5; % sec

% Create PAN Coordinator
panCoordinator = lrwpan.MACFullFunctionDevice('PANCoordinator', true, 'SamplesPerChip', 4, ....
  'PANIdentifier', '7777', 'ExtendedAddress', [repmat('0', 1, 8) repmat('7', 1, 8)], ...
  'ShortAddress', '1234');

% Create first end-device:
endDevice1 = lrwpan.MACReducedFunctionDevice('SamplesPerChip', 4, ...
  'ShortAddress', '0001', 'ExtendedAddress', [repmat('0', 1, 8) repmat('3', 1, 8)]);

% Initialize device inputs
received1 = zeros(samplesPerChip * chipsPerSymbol * symbolsPerStep/2, 1);
received2 = zeros(samplesPerChip * chipsPerSymbol * symbolsPerStep/2, 1);

while time < stopTime
  % Pass the received signals to the nodes for processing. Also, fetch what
  % they have to transmit:
  transmitted1 = panCoordinator(received1);
  transmitted2 = endDevice1(received2);

  % Ideal wireless channel, where both nodes are within range:
  received1 = transmitted2; % half-duplex radios, none receiving while transmitting
  received2 = transmitted1;

  time = time + symbolsPerStep/symbolRate; % update clock
end
```

```
0001: ********* Adding Beacon Request frame to the queue
0001: Passive scanning for 1584 steps
0001: Processing next frame from the queue
0001: Initializing transmission; backoff delay = 1 steps
0001: Backoff delay = 1 steps -> 0 steps
0001: Carrier sensing: Medium is idle.
0001: Clear to transmit
0001: Transmitting Beacon Request
0001: IFS offset = 0 samples
```

```
0001: Transmitting 1-1280 of 2050
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
0001: IFS offset = 0 samples
0001: Transmitting 1281-2050 of 2050
0001: Finished transmission
0001: Need to wait for SIFS (12) symbols. Offset = 12, next IFS = 4
0001: Entering passive scanning
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: *********** Received frame type = MAC command
1234: *********** Received MAC Command type = Beacon request
1234: Need to wait for SIFS (12) symbols. Offset = 12, next IFS = 4
1234: ********* Adding Beacon frame to the queue
1234: next IFS = 4
1234: Processing next frame from the queue
1234: Initializing transmission; backoff delay = 7 steps
1234: Backoff delay = 7 steps -> 6 steps
1234: Backoff delay = 6 steps -> 5 steps
1234: Backoff delay = 5 steps -> 4 steps
1234: Backoff delay = 4 steps -> 3 steps
1234: Backoff delay = 3 steps -> 2 steps
1234: Backoff delay = 2 steps -> 1 steps
1234: Backoff delay = 1 steps -> 0 steps
1234: Carrier sensing: Medium is idle.
1234: Clear to transmit
1234: IFS offset = 256 samples
1234: Transmitting 1-1024 of 2562
1234: IFS offset = 0 samples
1234: Transmitting 1025-2304 of 2562
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
1234: IFS offset = 0 samples
1234: Transmitting 2305-2562 of 2562
1234: Finished transmission
1234: Need to wait for LIFS (40) symbols. Offset = 4, next IFS = 24
1234: Decreased wait time by 20 symbols to 4
0001: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
0001: *********** Received frame type = Beacon
0001: Need to wait for SIFS (12) symbols. Offset = 4, next IFS = -4
0001: next IFS = 0
0001: Scanning finished
0001: ********* Adding Association request frame to the queue
0001: Processing next frame from the queue
0001: Initializing transmission; backoff delay = 0 steps
0001: Carrier sensing: Medium is idle.
0001: Clear to transmit
0001: IFS offset = 0 samples
0001: Transmitting 1-1280 of 3458
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
0001: IFS offset = 0 samples
0001: Transmitting 1281-2560 of 3458
0001: IFS offset = 0 samples
0001: Transmitting 2561-3458 of 3458
0001: Finished transmission
0001: will wait for ack for 54 symbols additional to IFS = 0
```

```
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: ********** Received frame type = MAC command
1234: ********** Received MAC Command type = Association request
1234: Need to wait for LIFS (40) symbols. Offset = 14, next IFS = 34
1234: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
1234: ********* Adding Data response frame to the PENDING queue
1234: next IFS = 34
0001: Decreasing ack wait durations by 20 symbols to 34
1234: IFS offset = 896 samples
1234: Transmitting 1-384 of 1410
0001: Decreasing ack wait durations by 20 symbols to 14
1234: IFS offset = 0 samples
1234: Transmitting 385-1410 of 1410
1234: Finished transmission
1234: Need to wait for SIFS (12) symbols. Offset = 16, next IFS = 8
0001: Decreasing ack wait durations by 20 symbols to -6
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
0001: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
0001: ********** Received frame type = Acknowledgment
0001: Need to wait for SIFS (12) symbols. Offset = 16, next IFS = 8
0001: ********** Adding Data request frame to the queue
0001: next IFS = 1920
0001: Decreased wait time by 20 symbols to 1900
0001: Decreased wait time by 20 symbols to 1880
0001: Decreased wait time by 20 symbols to 1860
0001: Decreased wait time by 20 symbols to 1840
0001: Decreased wait time by 20 symbols to 1820
0001: Decreased wait time by 20 symbols to 1800
0001: Decreased wait time by 20 symbols to 1780
0001: Decreased wait time by 20 symbols to 1760
0001: Decreased wait time by 20 symbols to 1740
0001: Decreased wait time by 20 symbols to 1720
0001: Decreased wait time by 20 symbols to 1700
0001: Decreased wait time by 20 symbols to 1680
0001: Decreased wait time by 20 symbols to 1660
0001: Decreased wait time by 20 symbols to 1640
0001: Decreased wait time by 20 symbols to 1620
0001: Decreased wait time by 20 symbols to 1600
0001: Decreased wait time by 20 symbols to 1580
0001: Decreased wait time by 20 symbols to 1560
0001: Decreased wait time by 20 symbols to 1540
0001: Decreased wait time by 20 symbols to 1520
0001: Decreased wait time by 20 symbols to 1500
0001: Decreased wait time by 20 symbols to 1480
0001: Decreased wait time by 20 symbols to 1460
0001: Decreased wait time by 20 symbols to 1440
0001: Decreased wait time by 20 symbols to 1420
0001: Decreased wait time by 20 symbols to 1400
0001: Decreased wait time by 20 symbols to 1380
0001: Decreased wait time by 20 symbols to 1360
0001: Decreased wait time by 20 symbols to 1340
0001: Decreased wait time by 20 symbols to 1320
0001: Decreased wait time by 20 symbols to 1300
0001: Decreased wait time by 20 symbols to 1280
0001: Decreased wait time by 20 symbols to 1260
```

```
0001: Decreased wait time by 20 symbols to 1240
0001: Decreased wait time by 20 symbols to 1220
0001: Decreased wait time by 20 symbols to 1200
0001: Decreased wait time by 20 symbols to 1180
0001: Decreased wait time by 20 symbols to 1160
0001: Decreased wait time by 20 symbols to 1140
0001: Decreased wait time by 20 symbols to 1120
0001: Decreased wait time by 20 symbols to 1100
0001: Decreased wait time by 20 symbols to 1080
0001: Decreased wait time by 20 symbols to 1060
0001: Decreased wait time by 20 symbols to 1040
0001: Decreased wait time by 20 symbols to 1020
0001: Decreased wait time by 20 symbols to 1000
0001: Decreased wait time by 20 symbols to 980
0001: Decreased wait time by 20 symbols to 960
0001: Decreased wait time by 20 symbols to 940
0001: Decreased wait time by 20 symbols to 920
0001: Decreased wait time by 20 symbols to 900
0001: Decreased wait time by 20 symbols to 880
0001: Decreased wait time by 20 symbols to 860
0001: Decreased wait time by 20 symbols to 840
0001: Decreased wait time by 20 symbols to 820
0001: Decreased wait time by 20 symbols to 800
0001: Decreased wait time by 20 symbols to 780
0001: Decreased wait time by 20 symbols to 760
0001: Decreased wait time by 20 symbols to 740
0001: Decreased wait time by 20 symbols to 720
0001: Decreased wait time by 20 symbols to 700
0001: Decreased wait time by 20 symbols to 680
0001: Decreased wait time by 20 symbols to 660
0001: Decreased wait time by 20 symbols to 640
0001: Decreased wait time by 20 symbols to 620
0001: Decreased wait time by 20 symbols to 600
0001: Decreased wait time by 20 symbols to 580
0001: Decreased wait time by 20 symbols to 560
0001: Decreased wait time by 20 symbols to 540
0001: Decreased wait time by 20 symbols to 520
0001: Decreased wait time by 20 symbols to 500
0001: Decreased wait time by 20 symbols to 480
0001: Decreased wait time by 20 symbols to 460
0001: Decreased wait time by 20 symbols to 440
0001: Decreased wait time by 20 symbols to 420
0001: Decreased wait time by 20 symbols to 400
0001: Decreased wait time by 20 symbols to 380
0001: Decreased wait time by 20 symbols to 360
0001: Decreased wait time by 20 symbols to 340
0001: Decreased wait time by 20 symbols to 320
0001: Decreased wait time by 20 symbols to 300
0001: Decreased wait time by 20 symbols to 280
0001: Decreased wait time by 20 symbols to 260
0001: Decreased wait time by 20 symbols to 240
0001: Decreased wait time by 20 symbols to 220
0001: Decreased wait time by 20 symbols to 200
0001: Decreased wait time by 20 symbols to 180
0001: Decreased wait time by 20 symbols to 160
0001: Decreased wait time by 20 symbols to 140
0001: Decreased wait time by 20 symbols to 120
0001: Decreased wait time by 20 symbols to 100
```

```
0001: Decreased wait time by 20 symbols to 80
0001: Decreased wait time by 20 symbols to 60
0001: Decreased wait time by 20 symbols to 40
0001: Decreased wait time by 20 symbols to 20
0001: Decreased wait time by 20 symbols to 0
0001: Processing next frame from the queue
0001: Initializing transmission; backoff delay = 2 steps
0001: Backoff delay = 2 steps -> 1 steps
0001: Backoff delay = 1 steps -> 0 steps
0001: Carrier sensing: Medium is idle.
0001: Clear to transmit
0001: IFS offset = 0 samples
0001: Transmitting 1-1280 of 3074
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
0001: IFS offset = 0 samples
0001: Transmitting 1281-2560 of 3074
0001: IFS offset = 0 samples
0001: Transmitting 2561-3074 of 3074
0001: Finished transmission
0001: will wait for ack for 54 symbols additional to IFS = 0
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: ********** Received frame type = MAC command
1234: ********** Received MAC Command type = Data request
1234: Need to wait for SIFS (12) symbols. Offset = 8, next IFS = 0
1234: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
1234: Moving frame for 0000000033333333 from pending queue to the transmission queue
1234: next IFS = 0
1234: IFS offset = 0 samples
1234: Transmitting 1-1280 of 1410
0001: Decreasing ack wait durations by 20 symbols to 34
1234: IFS offset = 0 samples
1234: Transmitting 1281-1410 of 1410
1234: Finished transmission
1234: Need to wait for SIFS (12) symbols. Offset = 2, next IFS = -6
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
0001: Decreasing ack wait durations by 20 symbols to 14
1234: Processing next frame from the queue
1234: Initializing transmission; backoff delay = 5 steps
1234: Backoff delay = 5 steps -> 4 steps
0001: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
0001: ********** Received frame type = Acknowledgment
0001: Need to wait for SIFS (12) symbols. Offset = 2, next IFS = -6
0001: next IFS = 0
1234: Backoff delay = 4 steps -> 3 steps
1234: Backoff delay = 3 steps -> 2 steps
1234: Backoff delay = 2 steps -> 1 steps
1234: Backoff delay = 1 steps -> 0 steps
1234: Carrier sensing: Medium is idle.
1234: Clear to transmit
1234: IFS offset = 0 samples
1234: Transmitting 1-1280 of 4226
1234: IFS offset = 0 samples
1234: Transmitting 1281-2560 of 4226
Found preamble of OQPSK PHY.
```

```
Found start-of-frame delimiter (SFD) of OQPSK PHY.
1234: IFS offset = 0 samples
1234: Transmitting 2561-3840 of 4226
1234: IFS offset = 0 samples
1234: Transmitting 3841-4226 of 4226
1234: Finished transmission
1234: will wait for ack for 54 symbols additional to IFS = 0
1234: Decreasing ack wait durations by 20 symbols to 34
0001: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
0001: ********** Received frame type = MAC command
0001: ********** Received MAC Command type = Association response
0001: Need to wait for LIFS (40) symbols. Offset = 6, next IFS = 26
0001: ********** Association successful, changing short address to = 8CEC
8CEC: ********** Association successful, associated to PAN = 7777
8CEC: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
8CEC: next IFS = 26
1234: Decreasing ack wait durations by 20 symbols to 14
8CEC: IFS offset = 384 samples
8CEC: Transmitting 1-896 of 1410
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
1234: Decreasing ack wait durations by 20 symbols to -6
8CEC: IFS offset = 0 samples
8CEC: Transmitting 897-1410 of 1410
8CEC: Finished transmission
8CEC: Need to wait for SIFS (12) symbols. Offset = 8, next IFS = 0
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: ********** Received frame type = Acknowledgment
1234: Need to wait for SIFS (12) symbols. Offset = 8, next IFS = 0
1234: next IFS = 0
8CEC: ********* (t=4.079360) Injecting data frame to the queue. From: 8CEC -> To: 1234
8CEC: Processing next frame from the queue
8CEC: Initializing transmission; backoff delay = 6 steps
8CEC: Backoff delay = 6 steps -> 5 steps
8CEC: Backoff delay = 5 steps -> 4 steps
8CEC: Backoff delay = 4 steps -> 3 steps
8CEC: Backoff delay = 3 steps -> 2 steps
8CEC: Backoff delay = 2 steps -> 1 steps
8CEC: Backoff delay = 1 steps -> 0 steps
8CEC: Carrier sensing: Medium is idle.
8CEC: Clear to transmit
8CEC: IFS offset = 0 samples
8CEC: Transmitting 1-1280 of 8578
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
8CEC: IFS offset = 0 samples
8CEC: Transmitting 1281-2560 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 2561-3840 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 3841-5120 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 5121-6400 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 6401-7680 of 8578
8CEC: IFS offset = 0 samples
```

```
8CEC: Transmitting 7681-8578 of 8578
8CEC: Finished transmission
8CEC: will wait for ack for 54 symbols additional to IFS = 0
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: ************ Received frame type = Data
1234: Need to wait for SIFS (12) symbols. Offset = 14, next IFS = 6
1234: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
1234: next IFS = 6
1234: IFS offset = 384 samples
1234: Transmitting 1-896 of 1410
8CEC: Decreasing ack wait durations by 20 symbols to 34
1234: IFS offset = 0 samples
1234: Transmitting 897-1410 of 1410
1234: Finished transmission
1234: Need to wait for SIFS (12) symbols. Offset = 8, next IFS = 0
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
8CEC: Decreasing ack wait durations by 20 symbols to 14
8CEC: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
8CEC: ************ Received frame type = Acknowledgment
8CEC: Need to wait for LIFS (40) symbols. Offset = 8, next IFS = 28
8CEC: next IFS = 28
8CEC: Decreased wait time by 20 symbols to 8
8CEC: ********* (t=4.795200) Injecting data frame to the queue. From: 8CEC -> To: 1234
8CEC: Processing next frame from the queue
8CEC: Initializing transmission; backoff delay = 6 steps
8CEC: Backoff delay = 6 steps -> 5 steps
8CEC: Backoff delay = 5 steps -> 4 steps
8CEC: Backoff delay = 4 steps -> 3 steps
8CEC: Backoff delay = 3 steps -> 2 steps
8CEC: Backoff delay = 2 steps -> 1 steps
8CEC: Backoff delay = 1 steps -> 0 steps
8CEC: Carrier sensing: Medium is idle.
8CEC: Clear to transmit
8CEC: IFS offset = 512 samples
8CEC: Transmitting 1-768 of 8578
Found preamble of OQPSK PHY.
8CEC: IFS offset = 0 samples
8CEC: Transmitting 769-2048 of 8578
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
8CEC: IFS offset = 0 samples
8CEC: Transmitting 2049-3328 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 3329-4608 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 4609-5888 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 5889-7168 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 7169-8448 of 8578
8CEC: IFS offset = 0 samples
8CEC: Transmitting 8449-8578 of 8578
8CEC: Finished transmission
8CEC: will wait for ack for 54 symbols additional to IFS = 0
1234: PHY decoded IEEE 802.15.4 frame
```

```
CRC check passed for the MAC frame.
1234: *********** Received frame type = Data
1234: Need to wait for SIFS (12) symbols. Offset = 2, next IFS = -6
1234: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
1234: next IFS = 0
1234: IFS offset = 0 samples
1234: Transmitting 1-1280 of 1410
8CEC: Decreasing ack wait durations by 20 symbols to 34
1234: IFS offset = 0 samples
1234: Transmitting 1281-1410 of 1410
1234: Finished transmission
1234: Need to wait for SIFS (12) symbols. Offset = 2, next IFS = -6
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
8CEC: Decreasing ack wait durations by 20 symbols to 14
8CEC: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
8CEC: *********** Received frame type = Acknowledgment
8CEC: Need to wait for LIFS (40) symbols. Offset = 2, next IFS = 22
8CEC: next IFS = 22
8CEC: Decreased wait time by 20 symbols to 2
1234: ********* (t=5.001280) Injecting data frame to the queue. From: 1234 -> To: 8CEC
1234: Processing next frame from the queue
1234: Initializing transmission; backoff delay = 1 steps
1234: Backoff delay = 1 steps -> 0 steps
1234: Carrier sensing: Medium is idle.
1234: Clear to transmit
1234: IFS offset = 0 samples
1234: Transmitting 1-1280 of 8578
1234: IFS offset = 0 samples
1234: Transmitting 1281-2560 of 8578
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
1234: IFS offset = 0 samples
1234: Transmitting 2561-3840 of 8578
1234: IFS offset = 0 samples
1234: Transmitting 3841-5120 of 8578
1234: IFS offset = 0 samples
1234: Transmitting 5121-6400 of 8578
1234: IFS offset = 0 samples
1234: Transmitting 6401-7680 of 8578
1234: IFS offset = 0 samples
1234: Transmitting 7681-8578 of 8578
1234: Finished transmission
1234: will wait for ack for 54 symbols additional to IFS = 0
1234: Decreasing ack wait durations by 20 symbols to 34
8CEC: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
8CEC: *********** Received frame type = Data
8CEC: Need to wait for SIFS (12) symbols. Offset = 14, next IFS = 6
8CEC: ********** Directly transmitting acknowledgement frame (no CSMA/CA)
8CEC: next IFS = 6
8CEC: IFS offset = 384 samples
8CEC: Transmitting 1-896 of 1410
Found preamble of OQPSK PHY.
Found start-of-frame delimiter (SFD) of OQPSK PHY.
1234: Decreasing ack wait durations by 20 symbols to 14
8CEC: IFS offset = 0 samples
```

```
8CEC: Transmitting 897-1410 of 1410
8CEC: Finished transmission
8CEC: Need to wait for SIFS (12) symbols. Offset = 8, next IFS = 0
1234: PHY decoded IEEE 802.15.4 frame
CRC check passed for the MAC frame.
1234: ********** Received frame type = Acknowledgment
1234: Need to wait for LIFS (40) symbols. Offset = 8, next IFS = 28
1234: next IFS = 28
1234: Decreased wait time by 20 symbols to 8
```

Once the 1st end device has been associated, data frames are randomly injected into the link between the end device and the PAN Coordinator.

Next, a third device joins the PAN and data frames are subsequently exchanged between the coordinator and both end devices, in a star topology fashion (end devices must only transmit frames to coordinators). In this case, the output is suppressed.

```
% Create second end-device:
endDevice2 = lrwpan.MACReducedFunctionDevice('SamplesPerChip', 4, ...
  'ShortAddress', '0002', 'ExtendedAddress', [repmat('0', 1, 8) repmat('4', 1, 8)], 'Verbosity',
% Suppress detailed output:
endDevice1.Verbosity = false;
panCoordinator.Verbosity = false;

% Initialize input
received3 = zeros(samplesPerChip * chipsPerSymbol * symbolsPerStep/2, 1);

stopTime = 10; % sec
while time < stopTime
  % Pass the received signals to the nodes for processing. Also, fetch what
  % they have to transmit:
  transmitted1 = panCoordinator(received1);
  transmitted2 = endDevice1(received2);
  transmitted3 = endDevice2(received3);

  % Ideal wireless channel, where all nodes are within range:
  received1 = transmitted2 + transmitted3; % half-duplex radios, none receiving while transmitti
  received2 = transmitted1 + transmitted3;
  received3 = transmitted1 + transmitted2;

  time = time + symbolsPerStep/symbolRate; % update clock
end
```

More nodes can be added to the network, as long as the channel relationship is established accordingly (i.e., the received signals as a function of the transmitted signals).

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- lrwpan.MACFullFunctionDevice
- lrwpan.MACReducedFunctionDevice
- lrwpan.MACDevice

Selected Bibliography

1 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

ZigBee NET Frame Generation and Decoding

This example shows how to use the Communications Toolbox™ Library for ZigBee and UWB to generate and decode NET frames of the ZigBee specification [1].

Background

The ZigBee standard specifies the network (NET or NWK) and application (APP or APL) layers for low-rate wireless personal area networks. These NET- and APP-layer specifications build upon the PHY and MAC specifications of IEEE® 802.15.4™ [2]. ZigBee devices find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

Architecture

A zigbee.NETFrameConfig configuration object is used both in generating and decoding ZigBee NET frames. Such objects describe a NET-layer frame and specify its frame type and all applicable properties. The zigbee.NETFrameGenerator function accepts a zigbee.NETFrameConfig object describing the frame, and optionally a NET-layer payload (APP-layer frame) in bytes (two-characters), and outputs the NET frame in bytes. The zigbee.NETFrameDecoder function accepts a NET Protocol Data Unit (NPDU) in bytes and outputs a zigbee.NETFrameConfig object describing the frame and possibly a NET-layer frame in bytes. Clause 3.3 in [1] describes the NET frame formats.

Decoding NET Frames of Home Automation ZigBee Radios

This section decodes NET frames transmitted from a commercial ZigBee radio enabling home automation, and captured using a USRP® B200-mini radio and the Communications Toolbox Support Package for USRP® radio.

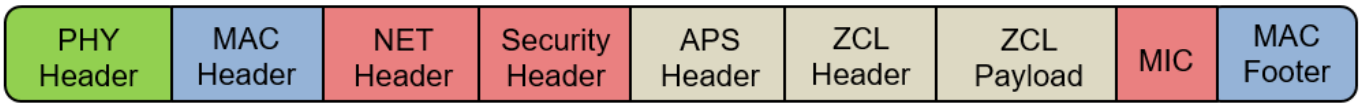
The zigbee.NETFrameDecoder function can decode NET-layer ZigBee data frames and the header of net-command frame types.

```
load zigbeeNETCaptures % netFrame

[netConfig, netPayload] = zigbee.NETFrameDecoder(netFrame);
netConfig
```

```
netConfig =

  NETFrameConfig with properties:

               FrameType: 'Data'
         ProtocolVersion: 'ZigBee 2007'
          SequenceNumber: 212

   Addressing:
           SourceAddress: '0000'
      DestinationAddress: '35EA'
          IEEEAddressing: 'None'

   Security:
                Security: 1
          DataEncryption: 0
               MICLength: 0
           KeyIdentifier: 'Network'
           ExtendedNonce: 1
```

```
          FrameCounter: 193458
 SecuritySourceAddress: '24FD5B00000014B6'
           KeySequence: 0

Routing:
                Radius: 30
         DiscoverRoute: 1
         SourceRouting: 1
            RelayIndex: 0
             RelayList: [0x4 char]

Multicast:
             Multicast: 0
```

Note that NET-layer decoding indicates that the NET-layer payload is encrypted (Security = true). Security can be used either in the network or the application layer; this frame uses network-layer security. On the one hand, the DataEncryption field is false in the frame and the Message Integrity Code (MIC) length is zero, which indicate that security level #0 is used and that the payload is not encrypted. However, according to the ZigBee standard (Clause 4.4.1.2 in [1]), these two fields are **overwritten** with values locally stored during network setup. In this case, this frame was secured with security level #5, which means that the NET-payload is encrypted and that the MIC length is 32 bits.



Format of IEEE 802.15.4 / ZigBee Frame

Generating NET Frames

The zigbee.NETFrameGenerator function can generate unsecure NET-layer ZigBee data frames. The configuration object can be further customized.

```
netConfig = zigbee.NETFrameConfig('SequenceNumber', 123, 'DestinationAddress', 'E568');
numOctets = 50;
payload = dec2hex(randi([0 2^8-1], numOctets, 1), 2);
netFrame = zigbee.NETFrameGenerator(netConfig, payload);
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- zigbee.NETFrameGenerator
- zigbee.NETFrameDecoder
- zigbee.NETFrameConfig

Selected Bibliography

1 ZigBee Alliance, ZigBee Specification Document 053474r17, 2007

2 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

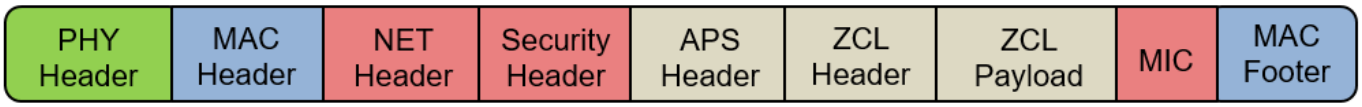
ZigBee Home Automation Frame Generation and Decoding

This example shows how to generate and decode application-layer frames for the Home Automation application profile [1] of the ZigBee® specification [2] using the Communications Toolbox™ Library for ZigBee and UWB.

Background

The ZigBee standard [2] specifies the network (NET or NWK) and application (APP or APL) layers for low-rate wireless personal area networks. These NET- and APP-layer specifications build upon the PHY and MAC specifications of IEEE® 802.15.4™ [3]. ZigBee devices find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

The ZigBee application layer consists of multiple sub-layers: *(i)* the Application Support Sublayer (APS), and *(ii)* the ZigBee Cluster Library (ZCL). The APS sublayer follows a format that is common for all application profiles and ZigBee clusters (see Clause 2.2.5 in [2]). The ZCL **header** follows a format that is common for all clusters (see Clause 2.4 in [4]). The ZCL **payload** is used only by some clusters and it follows a cluster-specific format.



Format of IEEE 802.15.4 / ZigBee Frame

Clusters and Frame Captures

Out of all the clusters used in the Home Automation application profile, this example decodes and generates frames for: *(i)* the On/Off cluster (used by light devices), and *(ii)* the Intruder Alarm System (IAS) Zone cluster (used by motion sensors) [4]. The On/Off cluster does not make use of a ZCL payload, but the IAS Zone cluster does.

Frames of these clusters have been captured from commercial ZigBee radios enabling home automation, using a USRP® B200-mini radio and the Communications Toolbox Support Package for USRP® radio. ZigBee can employ security either at the network or the application layer. The captured frames employed security at the network layer and were later on decrypted. This example decodes the application layer of the decrypted NET-layer payloads.

```
load zigbeeAPPCaptures
```

Decoding APS Frames of Home Automation ZigBee Radios

A zigbee.APSFrameConfig configuration object is used both in generating and decoding ZigBee APS frames. Such objects describe a APS-layer frame and specify its frame type and all applicable properties. The zigbee.APSFrameDecoder function accepts a APS Protocol Data Unit (APDU) in bytes and outputs a zigbee.APSFrameConfig object describing the frame and possibly a ZCL frame in bytes. Clause 2.2.5.1 in [2] describes the APS frame formats.

Next, the APS sublayer of a captured IAS Zone frame is decoded:

```
[apsConfig, apsPayload] = zigbee.APSFrameDecoder(motionDetectedFrame);
apsConfig
```

```
apsConfig =
```

```
  APSFrameConfig with properties:

               FrameType: 'Data'
              APSCounter: 230
    AcknowledgmentRequest: 1

   Addressing:
             DeliveryMode: 'Unicast'
      DestinationEndpoint: '01'
                ClusterID: '0500'
                ProfileID: '0104'
           SourceEndpoint: '01'

   Extended header:
           ExtendedHeader: 0

   Security:
                 Security: 0
```

Decoding ZCL Header of Home Automation ZigBee Radios

A zigbee.ZCLFrameConfig configuration object is used both in generating and decoding ZigBee ZCL headers. Such objects describe a ZCL-layer frame and specify its frame type and all applicable properties.

The zigbee.ZCLFrameDecoder function accepts a ZCL frame in bytes and outputs a zigbee.ZCLFrameConfig object describing the header and possibly a ZCL payload in bytes. Clause 2.4.1 in [4] describes the ZCL header frame formats. Note that the ZCL header may either specify a 'Library-wide' or a 'Cluster-specific' command type. In the latter case, the zigbee.ZCLFrameDecoder also needs the cluster ID, which is present in the APS header, in order to decode the cluster-specific command ID into a command type. For example, the next command decodes the ZCL header of a captured IAS Zone frame.

```
[zclConfig, zclPayload] = zigbee.ZCLFrameDecoder(apsPayload, apsConfig.ClusterID);
zclConfig
```

```
zclConfig =

  ZCLFrameConfig with properties:

                FrameType: 'Cluster-specific'
              CommandType: 'Zone Status Change Notification'
           SequenceNumber: 9
       ManufacturerCommand: 0
                Direction: 'Downlink'
    DisableDefaultResponse: 0
```

Decoding ZCL Payload of IAS Zone Frame from ZigBee Radio

In contrast to the On/Off cluster, the IAS Zone Cluster specifies a ZCL payload in addition to the ZCL header. A zigbee.IASZoneFrameConfig configuration object is used both in generating and decoding IAS Zone ZCL payloads. Such objects describe an IAS Zone payload and all applicable properties. The zigbee.IASZoneFrameDecoder function accepts an IAS Zone payload in bytes and outputs a zigbee.IASZoneFrameConfig object describing the IAZ Zone payload.

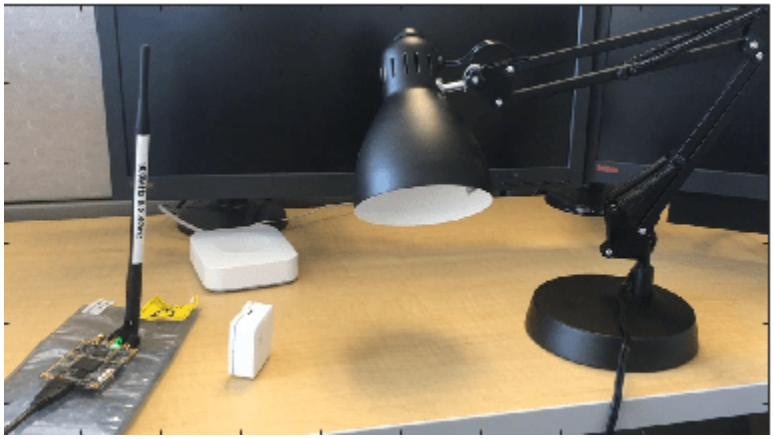
```
iasZoneConfig = zigbee.IASZoneFrameDecoder(zclPayload)
```

```
iasZoneConfig =

  IASZoneFrameConfig with properties:

       CommandType: 'Zone Status Change Notification'
            ZoneID: 0
            Alarm1: 'Not alarmed'
            Alarm2: 'Alarmed'
          Tampered: 0
       LowBattery: 0
    PeriodicReports: 0
     RestoreReports: 1
            Trouble: 0
           ACFault: 0
     BatteryDefect: 0
          TestMode: 0
             Delay: 0
```

Decoding Motion-Triggered Lighting Automation of ZigBee Radios

A lighting automation has been established for the commercial home-automation ZigBee radios whose frames have been captured and decoded. Specifically, once a motion sensor detects motion, it sends a signal to the ZigBee hub, which in turn sends a signal to a light bulb so that it turns on. When the motion sensor detects that the motion has stopped (e.g., after 10 seconds without motion) it sends a signal to the ZigBee hub, which in turn wirelessly triggers the light bulb to turn off. The following video illustrates the lighting automation.

```
helperPlaybackVideo('LightingAutomation.mp4', 2/5);
```



The following code decodes the actual frames transmitted between the ZigBee radios. These were captured with a USRP® device (also shown in the video).

```
apsFrames = {motionDetectedFrame; turnOnFrame; motionStoppedFrame; turnOffFrame};
for idx = 1:length(apsFrames)
  % APS decoding:
  [apsConfig, apsPayload] = zigbee.APSFrameDecoder(apsFrames{idx});
  % ZCL header decoding:
  [zclConfig, zclPayload] = zigbee.ZCLFrameDecoder(apsPayload, apsConfig.ClusterID);
```

```
  zclConfig

  % On-off cluster (does not have ZCL payload)
  onOffClusterID = '0006';
  if strcmp(apsConfig.ClusterID, onOffClusterID)
    fprintf(['Turn light bulb ' lower(zclConfig.CommandType) '.\n']);
  end

  % Intruder Alarm System (IAS) Zone cluster has ZCL payload:
  iasZoneClusterID = '0500';
  if ~isempty(zclPayload) && strcmp(apsConfig.ClusterID, iasZoneClusterID)
    iasConfig = zigbee.IASZoneFrameDecoder(zclPayload)

    if any(strcmp('Alarmed', {iasConfig.Alarm1, iasConfig.Alarm2}))
      fprintf('Motion detected.\n');
    else
      fprintf('Motion stopped.\n');
    end
  end
end


zclConfig =

  ZCLFrameConfig with properties:

                FrameType: 'Cluster-specific'
              CommandType: 'Zone Status Change Notification'
           SequenceNumber: 9
       ManufacturerCommand: 0
                Direction: 'Downlink'
    DisableDefaultResponse: 0


iasConfig =

  IASZoneFrameConfig with properties:

         CommandType: 'Zone Status Change Notification'
              ZoneID: 0
              Alarm1: 'Not alarmed'
              Alarm2: 'Alarmed'
            Tampered: 0
         LowBattery: 0
    PeriodicReports: 0
     RestoreReports: 1
             Trouble: 0
             ACFault: 0
      BatteryDefect: 0
           TestMode: 0
              Delay: 0

Motion detected.

zclConfig =

  ZCLFrameConfig with properties:
```

```
                FrameType: 'Cluster-specific'
              CommandType: 'On'
           SequenceNumber: 64
       ManufacturerCommand: 0
                Direction: 'Uplink'
    DisableDefaultResponse: 0

Turn light bulb on.

zclConfig =

  ZCLFrameConfig with properties:

                FrameType: 'Cluster-specific'
              CommandType: 'Zone Status Change Notification'
           SequenceNumber: 10
       ManufacturerCommand: 0
                Direction: 'Downlink'
    DisableDefaultResponse: 0


iasConfig =

  IASZoneFrameConfig with properties:

        CommandType: 'Zone Status Change Notification'
             ZoneID: 0
             Alarm1: 'Not alarmed'
             Alarm2: 'Not alarmed'
           Tampered: 0
         LowBattery: 0
    PeriodicReports: 0
     RestoreReports: 1
            Trouble: 0
            ACFault: 0
      BatteryDefect: 0
           TestMode: 0
              Delay: 0

Motion stopped.

zclConfig =

  ZCLFrameConfig with properties:

                FrameType: 'Cluster-specific'
              CommandType: 'Off'
           SequenceNumber: 70
       ManufacturerCommand: 0
                Direction: 'Uplink'
    DisableDefaultResponse: 0

Turn light bulb off.
```

Generating IAS Zone ZCL Payloads

The zigbee.IASZoneFrameGenerator function accepts a zigbee.IASZoneFrameConfig object describing the IAS Zone payload and outputs the payload in bytes. The following code creates two ZCL payloads for this cluster indicating that intrusion has or has not been detected.

```
iasConfigIntrusion = zigbee.IASZoneFrameConfig('Alarm2', 'Alarmed');
zclPayloadIntrusion = zigbee.IASZoneFrameGenerator(iasConfigIntrusion);

iasConfigNoIntrusion = zigbee.IASZoneFrameConfig('Alarm2', 'Not alarmed');
zclPayloadNoIntrusion = zigbee.IASZoneFrameGenerator(iasConfigNoIntrusion);
```

Generating ZCL Frames

The zigbee.ZCLFrameGenerator function accepts a zigbee.ZCLFrameConfig object describing the frame, and optionally a ZCL payload in bytes (two-characters), and outputs the ZCL frame in bytes. The following code generates ZCL frames for the On/Off cluster (no payload) and the IAS Zone cluster (payload needed).

```
% IAS Zone Cluster
zclConfigIntrusion = zigbee.ZCLFrameConfig('FrameType', 'Cluster-specific', ...
                                           'CommandType', 'Zone Status Change Notification', ...
                                           'SequenceNumber', 1, 'Direction', 'Downlink');
zclFrameIntrusion = zigbee.ZCLFrameGenerator(zclConfigIntrusion, zclPayloadIntrusion);

% On/Off Cluster
zclConfigOn = zigbee.ZCLFrameConfig('FrameType', 'Cluster-specific', ...
                                    'CommandType', 'On', ...
                                    'SequenceNumber', 2, 'Direction', 'Uplink');
zclFrameOn = zigbee.ZCLFrameGenerator(zclConfigOn);
```

Generating APS Frames

The zigbee.APSFrameGenerator function accepts a zigbee.APSFrameConfig object describing the frame, and optionally a APS payload (ZCL-layer frame) in bytes (two-characters), and outputs the APS frame in bytes. The following code illustrates how to generate APS frames for the ZCL frames created in the previous section.

```
% IAS Zone Cluster
apsConfigIntrusion = zigbee.APSFrameConfig('FrameType', 'Data', ...
                                           'ClusterID', iasZoneClusterID, ...
                                           'ProfileID', zigbee.profileID('Home Automation'), ...
                                           'APSCounter', 1, ...
                                           'AcknowledgmentRequest', true);
apsFrameIntrusion = zigbee.APSFrameGenerator(apsConfigIntrusion, zclFrameIntrusion);

% On/Off cluster
apsConfigOn = zigbee.APSFrameConfig('FrameType', 'Data', ...
                                    'ClusterID', onOffClusterID, ...
                                    'ProfileID', zigbee.profileID('Home Automation'), ...
                                    'APSCounter', 2, ...
                                    'AcknowledgmentRequest', true);
apsFrameOn = zigbee.APSFrameGenerator(apsConfigOn, zclFrameOn);
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- zigbee.APSFrameConfig, zigbee.APSFrameGenerator, zigbee.APSFrameDecoder
- zigbee.ZCLFrameConfig, zigbee.ZCLFrameGenerator, zigbee.ZCLFrameDecoder
- zigbee.IASZoneFrameConfig, zigbee.IASZoneFrameGenerator, zigbee.IASZoneFrameDecoder

Selected Bibliography

1 ZigBee Alliance, ZigBee Home Automation Public Application Profile, revision 29, v. 1.2, Jun. 2013.

2 ZigBee Alliance, ZigBee Specification Document 053474r17, 2007

3 IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

4 ZigBee Alliance, ZigBee Cluster Library Specification, Revision 6, Jan. 2016.

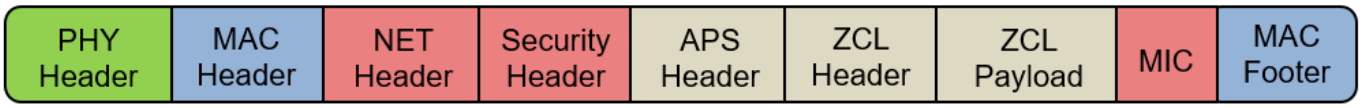
ZigBee Light Link Frame Generation and Decoding

This example shows how to generate and decode frames of the ZigBee® Light Link application profile [1] using the Communications Toolbox™ Library for ZigBee and UWB.

Background

The ZigBee standard [2] specifies network (NET or NWK) and application (APP or APL) layers of low-rate wireless personal area networks (LR-WPANs). These NET- and APP-layer specifications build upon the PHY and MAC specifications of IEEE® 802.15.4™ [3]. ZigBee devices find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

The ZigBee application layer consists of multiple sub-layers: *(i)* the Application Support Sublayer (APS), and *(ii)* the ZigBee Cluster Library (ZCL).



Format of IEEE 802.15.4 / ZigBee Frame

The APS and ZCL **headers** follow a format that is common for all application profiles and ZigBee clusters (see Clauses 2.2.5 in [2] and 2.4 in [4], respectively). The ZCL **payload** is used only by some clusters and it follows a cluster-specific format. The generic APS and ZCL header generation and decoding is illustrated in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example. This example illustrates the cluster-specific generation and decoding of **ZigBee Light Link ZCL payloads**.

Clusters and Commands

Out of the 7 clusters specified in the Light Link application profile [1], this example generates and decodes frames for the following clusters:

1. **Identify cluster**: This cluster sets a device into identification mode (e.g., flashing a light). This example illustrates frame generation and decoding for the **Identify command** (described in Clause 3.5 of [4]).
2. **Color Control cluster**: This cluster changes the color of a lighting device. This example illustrates frame generation and decoding for the **Move to Color command** (described in Clause 5.2 of [4]).
3. **Level Control** cluster: This cluster modifies the level of a device, e.g., the intensity of a light bulb, how closed a door is, or the intensity of a heater. This example illustrates frame generation and decoding for the **Move to Level command** (described in Clause 3.10 of [4]).
4. **Scenes cluster**: The scenes cluster sets up and recalls scenes (i.e., sets of stored attribute values for other clusters in the same device). This example illustrates frame generation and decoding for the **View Scene command** (described in Clause 3.7 of [4]).
5. **Group cluster**: This cluster manages groups of devices, e.g., by creating or removing a group, or by discovering group membership. This example illustrates frame generation and decoding for the **Add group command** (described in Clause 3.6 of [4]).

In addition to the illustrated commands, this example provides an implementation for generating and decoding frames for all commands of the five mentioned clusters (see Further Exploration for a complete list).

Generating and Decoding ZCL Payload of Identify Cluster

A zigbee.IdentifyFrameConfig configuration object is used both in generating and decoding ZCL payloads of the Identify cluster. Such objects describe an Identify cluster payload and all applicable properties. The zigbee.IdentifyFrameGenerator function accepts a zigbee.IdentifyFrameConfig object describing the Identify cluster payload and outputs the generated payload in bytes. The following code creates a ZCL payload for a command asking a device to identify for 4 seconds.

```
% Creation of configuration object for Identify cluster
identifyConfigTx = zigbee.IdentifyFrameConfig('CommandType', 'Identify', ...
                                              'IdentifyTime', 4);

% Frame generation (ZCL payload) for Identify cluster
identifyPayload = zigbee.IdentifyFrameGenerator(identifyConfigTx);
```

The zigbee.IdentifyFrameDecoder function accepts the command name and a Identify cluster payload in bytes and outputs a zigbee.IdentifyFrameConfig object describing the Identify cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
identifyConfigRx = zigbee.IdentifyFrameDecoder('Identify', identifyPayload)
```

```
identifyConfigRx =

  IdentifyFrameConfig with properties:

     CommandType: 'Identify'
    IdentifyTime: 4
```

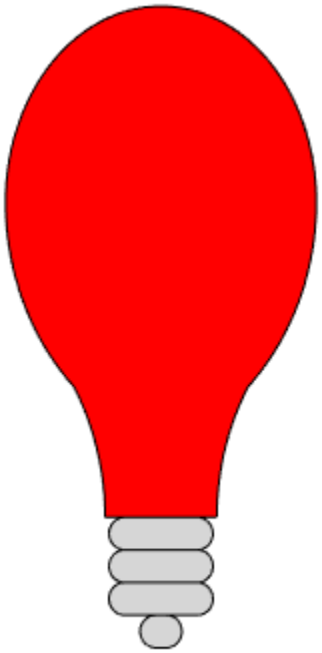
The following code visualizes a "software bulb" that illustrates the identification effect specified in the received frame.

```
bulb = plotBulb('white');
zigbeeIdentifyBulb(bulb, identifyConfigRx.IdentifyTime);
close(bulb);
```

Generating and Decoding ZCL Payload of Color Control Cluster

A zigbee.ColorControlFrameConfig configuration object is used both in generating and decoding ZCL payloads of the Color Control cluster. Such objects describe a Color Control cluster payload and all applicable properties. The zigbee.ColorControlFrameGenerator function accepts a zigbee.ColorControlFrameConfig object describing the Color Control cluster payload and outputs the generated payload in bytes. The following code generates a Color Control cluster payload that instructs a lighting device to progressively change its current color (red) to a different value (green) within 50 deciseconds (i.e., 5 seconds). Color is described in terms of *x*, *y* values according to the CIE 1931 color space established by the International Commission on Illumination (CIE) [5].

```
bulb = plotBulb('red');
```



```
% Creation of configuration object for Color Control cluster
colorCtrlConfigTx = zigbee.ColorControlFrameConfig('CommandType', 'Move to Color', ...
                                                   'ColorX', 16384, 'ColorY', 39322, 'Time', 50);

% Frame generation (ZCL payload) for Color Control cluster
colorControlPayload = zigbee.ColorControlFrameGenerator(colorCtrlConfigTx);
```

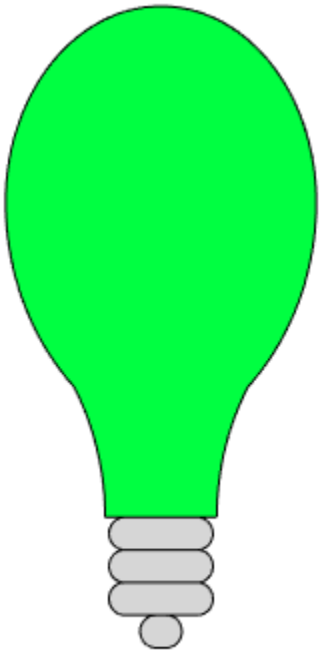
The zigbee.ColorControlFrameDecoder function accepts the command name and a Color Control cluster payload in bytes and outputs a zigbee.ColorControlFrameConfig object describing the Color Control cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
colorCtrlConfigRx = zigbee.ColorControlFrameDecoder('Move to Color', colorControlPayload)
```

```
colorCtrlConfigRx =

  ColorControlFrameConfig with properties:

    CommandType: 'Move to Color'
         ColorX: 16384
         ColorY: 39322
           Time: 50
```

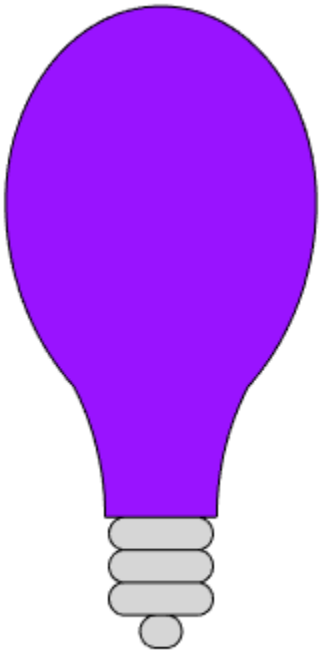
The following command uses a "software bulb" to visualize the Color Control effect specified in the received frame. Specifically, the color of a bulb progressively changes from red to green within 5 seconds.

```
zigbeeMoveBulbColor(bulb, colorCtrlConfigRx.ColorX, colorCtrlConfigRx.ColorY, colorCtrlConfigRx.T
```



Next, the same effect occurs on a different color trajectory (from green to violet).

```
colorCtrlConfigTx2 = zigbee.ColorControlFrameConfig('CommandType', 'Move to Color', ...
                                                    'ColorX',  19661, 'ColorY', 6554, 'Time', 50)
colorControlPayload2 = zigbee.ColorControlFrameGenerator(colorCtrlConfigTx2);
colorCtrlConfigRx2 = zigbee.ColorControlFrameDecoder('Move to Color', colorControlPayload2);
zigbeeMoveBulbColor(bulb, colorCtrlConfigRx2.ColorX, colorCtrlConfigRx2.ColorY, colorCtrlConfigRx

pause(1.5);
```



Generating and Decoding ZCL Payload of Level Control Cluster

A zigbee.LevelControlFrameConfig configuration object is used both in generating and decoding Level Control cluster ZCL payloads. Such objects describe a Level Control cluster payload and all applicable properties. The zigbee.LevelControlFrameGenerator function accepts a zigbee.LevelControlFrameConfig object describing the Level Control cluster payload and outputs the generated payload in bytes. The following code creates a Level Control cluster payload that instructs a device to change its current level to the specified value.

```
% Creation of Level Control cluster configuration object
levelCtrlConfigTx = zigbee.LevelControlFrameConfig('CommandType', 'Move to Level', ...
                                                   'Level', 20, 'TransitionTime', 1);

% Level Control cluster frame generation (ZCL payload)
levelControlPayload = zigbee.LevelControlFrameGenerator(levelCtrlConfigTx);
```

The zigbee.LevelControlFrameDecoder function accepts the command name and a Level Control cluster payload in bytes and outputs a zigbee.LevelControlFrameConfig object describing the Level Control cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

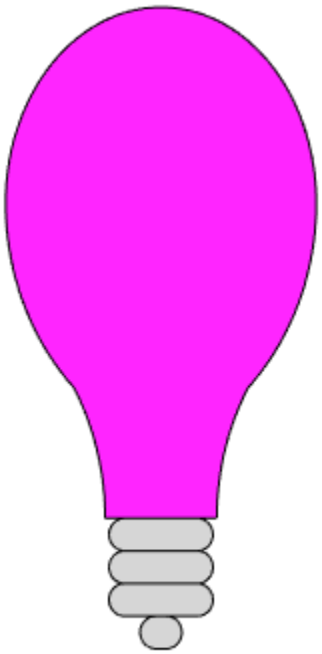
```
levelCtrlConfigRx = zigbee.LevelControlFrameDecoder('Move to Level', levelControlPayload)
```

```
levelCtrlConfigRx =

  LevelControlFrameConfig with properties:
```

```
       CommandType: 'Move to Level'
             Level: 20
    TransitionTime: 1
```

While the Level Control cluster can be used to regulate the intensity of a light, the Color Control cluster leaves it to the Level Control cluster to control the **luminance** of a lighting device's color. The following example uses the received Level Control frame to increase the luminance level of a light bulb.

```
zigbeeMoveBulbColor(bulb, colorCtrlConfigRx2.ColorX, colorCtrlConfigRx2.ColorY, 1, levelCtrlConfi
```



Generating and Decoding ZCL Payload of Scenes Cluster

A zigbee.SceneFrameConfig configuration object is used both in generating and decoding Scenes cluster ZCL payloads. Such objects describe a Scenes cluster payload and all applicable properties. The zigbee.ScenesFrameGenerator function accepts a zigbee.ScenesFrameConfig object describing the Scenes cluster payload and outputs the generated payload in bytes. The following code generates a Scenes cluster payload that requests a device to transmit a different frame (View Scene Response) describing a scene.

```
% Creation of Scenes cluster configuration object
scenesConfigTx = zigbee.ScenesFrameConfig('CommandType', 'View Scene', ...
                                          'GroupID', '1234', 'SceneID', '56');

% Scenes cluster frame generation (ZCL payload)
scenesPayload = zigbee.ScenesFrameGenerator(scenesConfigTx);
```

The zigbee.SceneFrameDecoder function accepts the command name and a Scenes cluster payload in bytes and outputs a zigbee.SceneFrameConfig object describing the Scenes cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
scenesConfigRx = zigbee.ScenesFrameDecoder('View Scene', scenesPayload)
```

```
scenesConfigRx =

  ScenesFrameConfig with properties:

    CommandType: 'View Scene'
        GroupID: '1234'
        SceneID: '56'
```

Generating and Decoding ZCL Payload of Groups Cluster

A zigbee.GroupFrameConfig configuration object is used both in generating and decoding Groups cluster ZCL payloads. Such objects describe a Groups cluster payload and all applicable properties. The zigbee.GroupsFrameGenerator function accepts a zigbee.GroupsFrameConfig object describing the Groups cluster payload and outputs the generated payload in bytes. The following code creates a Groups cluster payload that instructs a device to add the specified group to its Group table.

```
% Creation of Groups cluster configuration object
groupsConfigTx = zigbee.GroupsFrameConfig('CommandType', 'Add group', ...
                        'GroupName', 'Dining Hall', 'GroupID', '1234');

% Groups cluster frame generation (ZCL payload)
groupsPayload = zigbee.GroupsFrameGenerator(groupsConfigTx);
```

The zigbee.GroupFrameDecoder function accepts the command name and a Groups cluster payload in bytes and outputs a zigbee.GroupFrameConfig object describing the Groups cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
groupsConfigRx = zigbee.GroupsFrameDecoder('Add group', groupsPayload)
```

```
groupsConfigRx =

  GroupsFrameConfig with properties:

    CommandType: 'Add group'
        GroupID: '1234'
      GroupName: 'Dining Hall'
```

Wireshark Decoding

The generated frames can be converted to a PCAP format, which can be analyzed and visualized with Wireshark [6]. This process can serve as an additional verification step advocating that the Communications Toolbox Library for the ZigBee Protocol generates and decodes frames in a standard-compliant manner.

The PCAP file needs the ZCL payloads to be enclosed with headers from all other layers and sublayers (MAC, NET, APS, ZCL). The following commands generate a PCAP file, for the ZCL payloads generated in this example, that can be loaded with Wireshark.

```
% ZLL profile ID
zllProfileID = zigbee.profileID('Light Link');

payloadsWithInfo(1) = struct('Payload',   identifyPayload,       'ProfileID',   zllProfileID, ..
                             'ClusterSpecific', true,            'ClusterID',   zigbee.clusterID
payloadsWithInfo(2) = struct('Payload',   colorControlPayload,   'ProfileID',   zllProfileID, ..
                             'ClusterSpecific', true,            'ClusterID',   zigbee.clusterID
payloadsWithInfo(3) = struct('Payload',   levelControlPayload,   'ProfileID',   zllProfileID, ..
                             'ClusterSpecific', true,            'ClusterID',   zigbee.clusterID
payloadsWithInfo(4) = struct('Payload',   scenesPayload,         'ProfileID',   zllProfileID, ..
                             'ClusterSpecific', true,            'ClusterID',   zigbee.clusterID
payloadsWithInfo(5) = struct('Payload',   groupsPayload,         'ProfileID',   zllProfileID, ..
                             'ClusterSpecific', true,            'ClusterID',   zigbee.clusterID

% Add headers from other layers/sublayers:
MPDUs = zigbeeAddProtocolHeaders(payloadsWithInfo);

% Export MPDUs to a PCAP format
zigbeeExportToPcap(MPDUs, 'zigbeeLightLink.pcap');

% Open PCAP file with Wireshark
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- zigbee.APSFrameConfig, zigbee.APSFrameGenerator, zigbee.APSFrameDecoder
- zigbee.ZCLFrameConfig, zigbee.ZCLFrameGenerator, zigbee.ZCLFrameDecoder
- zigbee.IdentifyFrameConfig, zigbee.IdentifyFrameGenerator, zigbee.IdentifyFrameDecoder
- zigbee.ColorControlFrameConfig, zigbee.ColorControlFrameGenerator, zigbee.ColorControlFrameDecoder
- zigbee.LevelControlFrameConfig, zigbee.LevelControlFrameGenerator, zigbee.LevelControlFrameDecoder
- zigbee.ScenesFrameConfig, zigbee.ScenesFrameGenerator, zigbee.ScenesFrameDecoder
- zigbee.GroupsFrameConfig, zigbee.GroupsFrameGenerator, zigbee.GroupsFrameDecoder

In addition to the commands illustrated in this example, the offered implementation also supports the commands listed in the following table. The commands listed in the middle column can be exported to a PCAP file that can be analyzed with Wireshark.

Cluster	Implemented commands validated with Wireshark	Implemented commands not supported by Wireshark 2.2.5
Identify	Identify, Identify Query, Identify Query Response	Trigger effect
Color Control	Move to Hue, Move Hue, Step Hue, Move to Saturation, Move Saturation, Step Saturation, Move to Hue and Saturation, Move to Color, Move Color, Step Color, Move to Color Temperature	Enhanced Move to Hue, Enhanced Move Hue, Enhanced Step Hue, Enhanced Move to Hue and Saturation, Color Loop Set, Stop Move Step, Move Color Temperature, Step Color Temperature
Level Control	Move to Level, Move, Step, Stop, Move to Level (with On/Off), Move (with On/Off), Step (with On/Off), Stop (with On/Off)	-
Scenes	Add Scene, View Scene, Remove Scene, Remove All Scenes, Store Scene, Recall Scene, Get Scene Membership, View Scene Response, Remove Scene Response, Remove All Scenes Response, Store Scene Response, Get Scene Membership Response	Enhanced Add Scene, Enhanced View Scene, Copy Scene, Add Scene ResponseEnhanced Add Scene Response, Enhanced View Scene Response, Copy Scene Response
Groups	Add group, View group, Get group membership, Remove group, Remove all groups, Add group if identifying, Add group response, View group response, Remove group response, Get group membership response	-

Selected Bibliography

1. ZigBee Alliance, ZigBee Light Link Standard, v. 1.0, April 5th, 2012.
2. ZigBee Alliance, ZigBee Specification Document 053474r17, 2007
3. IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)
4. ZigBee Alliance, ZigBee Cluster Library Specification, Revision 6, Jan. 2016.
5. CIE 1931 Color Space. Commission Internationale de l'Eclairage Proceedings. Cambridge University Press, Cambridge
6. Wireshark software: https://www.wireshark.org/

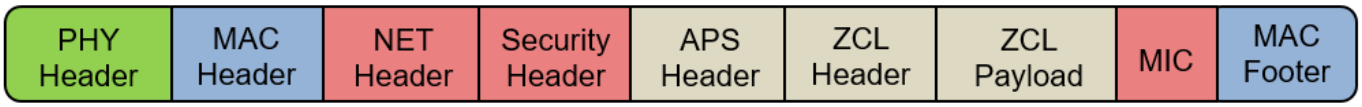
ZigBee Frame Generation and Decoding for General Commands

This example shows how to generate and decode General Command frames of the ZigBee® specification [1] using the Communications Toolbox™ Library for ZigBee and UWB.

Background

The ZigBee standard [1] specifies network (NET or NWK) and application (APP) layers of low-rate wireless personal area networks. These NET- and APP-layer specifications build upon the PHY and MAC specifications of IEEE® 802.15.4™ [2]. ZigBee devices find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

The ZigBee application layer consists of multiple sub-layers: *(i)* the Application Support Sublayer (APS), and *(ii)* the ZigBee Cluster Library (ZCL).



Format of IEEE 802.15.4 / ZigBee Frame

The APS and ZCL **headers** follow a format that is common for all application profiles and ZigBee clusters/commands (see Clauses 2.2.5 in [1] and 2.4 in [3], respectively). The APS header declares the cluster of the frame and the ZCL header declares the command of the frame. The ZCL payload is present only for some clusters/commands and follows a command-specific format.

Some commands only apply for a specific cluster, while some other (general) commands can be used for all clusters. General command frames are used for manipulating attributes and other general tasks that are not specific to an individual cluster (see Clause 2.5 in [3]). This example illustrates how to generate and decode ZCL payloads for such general, library-wide ZigBee commands. The generic APS and ZCL header generation and decoding is illustrated in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

Commands

This examples illustrates frame generation and decoding for the following general commands:

1. **Read Attributes:** This command inquires an attribute value at a different device.
2. **Read Attributes Response:** This command responds with an attribute value.
3. **Write Attributes:** This command modifies an attribute value at a different device.
4. **Write Attributes Response:** This command responds with the result of a Write Attributes command.

In addition, this example provides an implementation for the following commands (which are not illustrated):

1. **Write Attributes Undivided:** This command is the same with "Write Attributes" with the only exception that an attribute is updated only if all other specified attributes can also be updated.
2. **Write Attributes No Response:** This command is the same with "Write Attributes" with the only exception that a response frame is not required.
3. **Report Attributes:** This command reports all attributes and their values.

4 **Default Response:** This command generates response frames of generic format.

A zigbee.GeneralFrameConfig configuration object is used both in generating and decoding ZCL payloads of General Commands. Such objects describe a General Commands payload and all applicable properties.

Generating ZCL Payloads of General Commands

The zigbee.GeneralFrameGenerator function accepts a zigbee.GeneralFrameConfig object describing the payload of the general command and generates the payload in bytes. The following code creates the payload of the Read/Write Attribute commands and their responses.

```
% Read Attributes command:
readConfigTx = zigbee.GeneralFrameConfig('CommandType', 'Read Attributes', 'AttributeID', '0000')
readPayload = zigbee.GeneralFrameGenerator(readConfigTx);

% Read Attributes Response command:
readResponseConfigTx = zigbee.GeneralFrameConfig('CommandType', 'Read Attributes Response', ...
                           'AttributeID', '0000', 'Status', 'Success', 'AttributeType', 'boolean
readResponsePayload = zigbee.GeneralFrameGenerator(readResponseConfigTx);

% Write Attributes command:
writeConfigTx = zigbee.GeneralFrameConfig('CommandType', 'Write Attributes', 'AttributeID', '0000
writePayload = zigbee.GeneralFrameGenerator(writeConfigTx);

% % Write Attributes Response command:
writeResponseConfigTx = zigbee.GeneralFrameConfig('CommandType', 'Write Attributes Response', 'St
writeResponsePayload = zigbee.GeneralFrameGenerator(writeResponseConfigTx);
```

```
readConfigTx =

  GeneralFrameConfig with properties:

    CommandType: 'Read Attributes'
    AttributeID: '0000'


readResponseConfigTx =

  GeneralFrameConfig with properties:

       CommandType: 'Read Attributes Response'
       AttributeID: '0000'
            Status: 'Success'
     AttributeType: 'Boolean'
    AttributeValue: 0


writeConfigTx =

  GeneralFrameConfig with properties:

       CommandType: 'Write Attributes'
       AttributeID: '0000'
     AttributeType: 'Boolean'
    AttributeValue: 1
```

```
writeResponseConfigTx =

  GeneralFrameConfig with properties:

    CommandType: 'Write Attributes Response'
         Status: 'Success'
```

Decoding ZCL Payloads of General Commands Captured from ZigBee Radios

This section decodes ZCL payloads of general commands captured from commercial Home-Automation ZigBee radios> with a USRP® B200-mini radio and the Communications Toolbox Support Package for USRP® radio. For more information, see section 'Clusters and Frame Captures' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
% load captured payloads
load zigbeeGeneralCommandCaptures
```

The zigbee.GeneralFrameDecoder function accepts a general command name and its payload in bytes and outputs a zigbee.GeneralFrameConfig object describing the payload of the general command. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
% Read Attributes :
readConfigRx = zigbee.GeneralFrameDecoder('Read Attributes', capturedReadPayload)

% Read Attributes Response:
readResponseRx = zigbee.GeneralFrameDecoder('Read Attributes Response', capturedReadResponsePaylo

% Default Response
defaultResponseRx = zigbee.GeneralFrameDecoder('Default Response', capturedDefaultResponsePayloa
```

```
readConfigRx =

  GeneralFrameConfig with properties:

    CommandType: 'Read Attributes'
    AttributeID: '0000'


readResponseRx =

  GeneralFrameConfig with properties:

       CommandType: 'Read Attributes Response'
       AttributeID: '0000'
            Status: 'Success'
     AttributeType: 'Boolean'
    AttributeValue: 1


defaultResponseRx =

  GeneralFrameConfig with properties:
```

```
         CommandType: 'Default Response'
              Status: 'Success'
    CommandToRespond: '01'
```

Decoding Generated ZCL Payloads of General Commands

This section illustrates the decoding of the remaining generated general commands (i.e., 'Write Attributes', 'Write Attributes Response').

```
% Write Attributes :
writeConfigRx = zigbee.GeneralFrameDecoder('Write Attributes', writePayload)

% Write Attributes Response:
writeResponseRx = zigbee.GeneralFrameDecoder('Write Attributes Response', writeResponsePayload)
```

```
writeConfigRx =

  GeneralFrameConfig with properties:

       CommandType: 'Write Attributes'
       AttributeID: '0000'
     AttributeType: 'Boolean'
    AttributeValue: 1


writeResponseRx =

  GeneralFrameConfig with properties:

    CommandType: 'Write Attributes Response'
         Status: 'Success'
```

Wireshark Decoding

The generated frames can be converted to a PCAP format, which can be analyzed and visualized with Wireshark [4]. This process can serve as an additional verification step advocating that the Communications Toolbox Library for the ZigBee Protocol generates and decodes frames in a standard-compliant manner.

The PCAP file needs the ZCL payloads to be enclosed with headers from all other layers and sublayers (MAC, NET, APS, ZCL). The following commands generate a PCAP file, for the ZCL payloads generated in this example, that can be loaded with Wireshark.

```
% Profile ID
profileID = zigbee.profileID('Home Automation');
onOffID      = zigbee.clusterID('On/Off');

payloadsWithInfo(1) = struct('Payload',   readPayload,           'ProfileID',  profileID, ...
                             'ClusterSpecific', false,           'ClusterID',  onOffID,      'Cor
payloadsWithInfo(2) = struct('Payload',   readResponsePayload,   'ProfileID',  profileID, ...
                             'ClusterSpecific', false,           'ClusterID',  onOffID,      'Cor
payloadsWithInfo(3) = struct('Payload',   writePayload,          'ProfileID',  profileID, ...
                             'ClusterSpecific', false,           'ClusterID',  onOffID,      'Cor
payloadsWithInfo(4) = struct('Payload',   writeResponsePayload,  'ProfileID',  profileID, ...
                             'ClusterSpecific', false,           'ClusterID',  onOffID,      'Cor
```

```
% Add headers from other layers/sublayers:
MPDUs = zigbeeAddProtocolHeaders(payloadsWithInfo);

% Export MPDUs to a PCAP format
zigbeeExportToPcap(MPDUs, 'zigbeeGeneralCommands.pcap');

% Open PCAP file with Wireshark
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- zigbee.GeneralFrameConfig, zigbee.GeneralFrameGenerator, zigbee.GeneralFrameDecoder
- zigbee.ZCLFrameConfig, zigbee.ZCLFrameGenerator, zigbee.ZCLFrameDecoder
- zigbee.APSFrameConfig, zigbee.APSFrameGenerator, zigbee.APSFrameDecoder

Selected Bibliography

1. ZigBee Alliance, ZigBee Specification Document 053474r17, 2007
2. IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)
3. ZigBee Alliance, ZigBee Cluster Library Specification, Revision 6, Jan. 2016.
4. Wireshark software: https://www.wireshark.org/

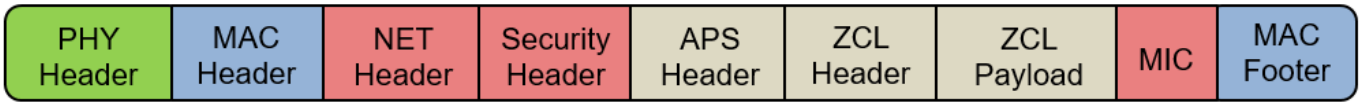
ZigBee Smart Energy Frame Generation and Decoding

This example shows how to generate and decode ZigBee® Smart Energy frames using the Communications Toolbox™ Library for ZigBee and UWB.

Background

The ZigBee standard [2] specifies network (NET or NWK) and application (APP or APL) layers of low-rate wireless personal area networks (LR-WPANs). These NET- and APP-layer specifications build upon the PHY and MAC specifications of IEEE® 802.15.4™ [3]. ZigBee devices find application in home automation and sensor networking and are highly relevant to the Internet of Things (IoT) trend.

The ZigBee application layer consists of multiple sub-layers: *(i)* the Application Support Sublayer (APS), and *(ii)* the ZigBee Cluster Library (ZCL).



Format of IEEE 802.15.4 / ZigBee Frame

The APS and ZCL **headers** follow a format that is common for all application profiles and ZigBee clusters (see Clauses 2.2.5 in [2] and 2.4 in [4], respectively). The ZCL **payload** is used only by some clusters and it follows a cluster-specific format. The generic APS and ZCL header generation and decoding is illustrated in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example. This example illustrates the cluster-specific generation and decoding of **ZigBee Smart Energy ZCL payloads**.

Clusters and Commands

Out of the 7 clusters used in the Smart Energy application profile, this example generates and decodes frames for the following clusters:

1. **Demand Response and Load Control (DRLC) cluster**: This cluster advertises changes to energy demand and consumption. This example illustrates frame generation and decoding for the **Load Control Event command** (described in Clause 10.3.2.3.1 of [4]).
2. **Price cluster**: This cluster communicates Energy, Gas or Water pricing information. This example illustrates frame generation and decoding for the **Get Current Price** and **Publish Price** commands (described in Clause 10.2.2.3.1 of [4]).
3. **Messaging cluster**: This cluster exchanges text messages between ZigBee devices. This example illustrates frame generation and decoding for the **Display Message command** (described in Clause 10.5.2.3.1 of [4]).

In addition to the illustrated commands, the implementation offered in this example also generates and decodes frames of the following commands:

Cluster	Implemented commands validated with Wireshark	Implemented commands not supported by Wireshark 2.2.5
DRLC	-	Load Control Event, Cancel Load Control Event, Cancel All Load Control Events, Report Event Status
Price	-	Get Current Price, Price Acknowledgement, Publish Price
Messaging	Display Message, Cancel Message, Get Last Message, Message Confirmation	-

Generating and Decoding ZCL Payload of DRLC Cluster

A zigbee.DRLCFrameConfig configuration object is used both in generating and decoding ZCL payloads for the Demand Response and Load Control (DRLC) cluster. Such objects describe a DRLC cluster payload and all applicable properties. The zigbee.DRLCFrameGenerator function accepts a zigbee.DRLCFrameConfig object describing the DRLC cluster payload and outputs the payload in bytes. The following code creates a ZCL payload for a command that sets the set point of heating devices to 23.5 C.

```
% Creation of DRLC cluster configuration object
drlcConfigTx = zigbee.DRLCFrameConfig('CommandType', 'Load Control Event', ...
                                      'EventID', '00000001', 'DeviceClass', 'Strip Heaters/Baseb
                                      'HeatingSetPoint', 23.5);

% DRLC cluster frame generation (ZCL payload)
drlcPayload = zigbee.DRLCFrameGenerator(drlcConfigTx);
```

The zigbee.DRLCFrameDecoder function accepts the command name and a DRLC cluster payload in bytes and outputs a zigbee.DRLCFrameConfig object describing the DRLC cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
drlcConfigRx = zigbee.DRLCFrameDecoder('Load Control Event', drlcPayload)
```

```
drlcConfigRx =

  DRLCFrameConfig with properties:

         CommandType: 'Load Control Event'
             EventID: '00000001'
         DeviceClass: 'Strip Heaters/Baseboard Heaters'
         DeviceGroup: '00'
                Time: 0
            Duration: 0
    CriticalityLevel: 'Green'
     HeatingSetPoint: 23.5000
         RandomStart: 1
           RandomEnd: 1
```

Generating and Decoding ZCL Payload of Price Cluster

A zigbee.PriceFrameConfig configuration object is used both in generating and decoding ZCL payloads for the Price cluster. Such objects describe a Price cluster payload and all applicable

properties. The zigbee.PriceFrameGenerator function accepts a zigbee.PriceFrameConfig object describing the Price cluster payload and outputs the payload in bytes. The following code creates a ZCL payload for a command that requests the current price of a commodity.

```
% Creation of Price cluster configuration object
priceConfigTx = zigbee.PriceFrameConfig('CommandType', 'Get Current Price');

% Price cluster frame generation (ZCL payload)
pricePayload = zigbee.PriceFrameGenerator(priceConfigTx);
```

The zigbee.PriceFrameDecoder function accepts the command name and a Price cluster payload in bytes and outputs a zigbee.PriceFrameConfig object describing the Price cluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
priceConfigRx = zigbee.PriceFrameDecoder('Get Current Price', pricePayload)
```

```
priceConfigRx =

  PriceFrameConfig with properties:

      CommandType: 'Get Current Price'
    IdleReceiving: 0
```

Upon receiving a 'Get Current Price' command, a server replies with a 'Publish Price' command.

```
priceConfigTx = zigbee.PriceFrameConfig('CommandType', 'Publish Price', 'Price', 0.4899, 'Duratio
pricePayload = zigbee.PriceFrameGenerator(priceConfigTx);
```

The client device can then decode the published price:

```
priceConfigRx = zigbee.PriceFrameDecoder('Publish Price', pricePayload)
```

```
priceConfigRx =

  PriceFrameConfig with properties:

       CommandType: 'Publish Price'
        ProviderID: 0
         RateLabel: ''
           EventID: 0
    GenerationTime: 0
              Unit: 'kW'
        UnitFormat: 'Binary'
          Currency: 840
         PriceTier: 1
      RegisterTier: 1
     NumPriceTiers: 0
         StartTime: 0
          Duration: 14400
             Price: 0.4899
```

Generating and Decoding ZCL Payload of Messaging Cluster

A zigbee.MessagingFrameConfig configuration object is used both in generating and decoding ZCL payloads for the Messaging cluster. Such objects describe a Messaging cluster payload and all applicable properties. The zigbee.MessagingFrameGenerator function accepts a zigbee.MessagingFrameConfig object describing the Messaging cluster payload and outputs the payload in bytes. The following code creates a ZCL payload for a command that displays a message.

```
% Creation of messaging cluster configuration object
messageID = 1234;
messagingConfigTx = zigbee.MessagingFrameConfig('CommandType', 'Display Message', ...
                                        'MessageID', messageID, 'Message', 'This is a custom messag

% Messaging cluster frame generation (ZCL payload)
displayMessagePayload = zigbee.MessagingFrameGenerator(messagingConfigTx);
```

The zigbee.MessagingFrameDecoder function accepts the command name and a Messaging cluster payload in bytes and outputs a zigbee.MessagingFrameConfig object describing the Messagingcluster payload. The command name is retrieved from the decoding of the ZCL header. See section 'Decoding ZCL Header of Home Automation ZigBee Radios' in the “ZigBee Home Automation Frame Generation and Decoding” on page 7-68 example.

```
messagingConfigRx = zigbee.MessagingFrameDecoder('Display Message', displayMessagePayload)
```

```
messagingConfigRx =

  MessagingFrameConfig with properties:

           CommandType: 'Display Message'
             MessageID: 1234
      TransmissionType: 'Normal Transmission Only'
              Priority: 'Low'
    MessageConfirmation: 0
              Duration: 90
               Message: 'This is a custom message'
```

A server that displays a message also has the ability to cancel the message using the "Cancel Message" command:

```
cancelMsgConfig = zigbee.MessagingFrameConfig('CommandType', 'Cancel Message', ...
                                              'MessageID', messageID);
cancelMessagePayload = zigbee.MessagingFrameGenerator(messagingConfigTx);
```

Clients can then decode the Cancel Message command:

```
messagingConfigRx = zigbee.MessagingFrameDecoder('Cancel Message', cancelMessagePayload)
```

```
messagingConfigRx =

  MessagingFrameConfig with properties:

           CommandType: 'Cancel Message'
             MessageID: 1234
      TransmissionType: 'Normal Transmission Only'
              Priority: 'Low'
```

```
    MessageConfirmation: 0
```

Wireshark Decoding

The generated Messaging frames can be converted to a PCAP-formatted file that can be analyzed and visualized with Wireshark [5]. This process can serve as an additional verification step advocating that the Communications Toolbox Library for the ZigBee Protocol generates and decodes frames in a standard-compliant manner.

The PCAP file needs the ZCL payloads to be enclosed with headers from all other layers and sublayers (MAC, NET, APS, ZCL). This task is performed by the following commands.

```
zllProfileID = zigbee.profileID('Smart Energy'); % ZLL profile ID
msgClusterID = zigbee.clusterID('Messaging'); % Messaging cluster ID

payloadsWithInfo(1) = struct('Payload', displayMessagePayload,  'ProfileID',   zllProfileID, ...
                             'ClusterSpecific', true,           'ClusterID',   msgClusterID, 'Cor
payloadsWithInfo(2) = struct('Payload', cancelMessagePayload,   'ProfileID',   zllProfileID, ...
                             'ClusterSpecific', true,           'ClusterID',   msgClusterID, 'Cor

% Add headers from other layers/sublayers:
MPDUs = zigbeeAddProtocolHeaders(payloadsWithInfo);

% Export MPDUs to a PCAP format
zigbeeExportToPcap(MPDUs, 'zigbeeSmartEnergy.pcap');

% Open PCAP file with Wireshark
```

Further Exploration

You can further explore the following generator and decoding functions, as well as the configuration object:

- zigbee.DRLCFrameConfig, zigbee.DRLCFrameGenerator, zigbee.DRLCFrameDecoder
- zigbee.PriceFrameConfig, zigbee.PriceFrameGenerator, zigbee.PriceFrameDecoder
- zigbee.MessagingFrameConfig, zigbee.MessagingFrameGenerator, zigbee.MessagingFrameDecoder
- zigbee.APSFrameConfig, zigbee.APSFrameGenerator, zigbee.APSFrameDecoder
- zigbee.ZCLFrameConfig, zigbee.ZCLFrameGenerator, zigbee.ZCLFrameDecoder

Selected Bibliography

1. ZigBee Alliance, ZigBee Smart Energy Standard, Revision 19, Version 1.2a, December 3, 2014.
2. ZigBee Alliance, ZigBee Specification Document 053474r17, 2007
3. IEEE 802.15.4-2011 - IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)
4. ZigBee Alliance, ZigBee Cluster Library Specification, Revision 6, Jan. 2016.
5. Wireshark software: https://www.wireshark.org/

Communications Toolbox Featured Examples

Simulate and Verify Power Amplifier Backoff

This example shows how to use backoff to scale a signal prior to inputting it to a table-based power amplifier. It also shows how to examine the power distribution of the signal input to the amplifier, and to verify that the actual behavior of the amplifier matches the specification. The Appendix on page 8-0 lists helper functions used in the example.

System Setup

```
totalTime = 0;
M = 16; % Modulation order
fs = 1e6; % Sample rate, in Hz (also used for measurement bandwidth)
sigDuration = 0.01; % sec
msgLen = round(sigDuration*fs); % samples
```

Specify the amplifier as a table-based object. Using measured amplifier data stored in an Excel spreadsheet, read the output power vs. input power and phase change vs. input power. The powers are given in dBm, and the phase change in degrees. The reference impedance is used to convert the signal's voltage values to power values.

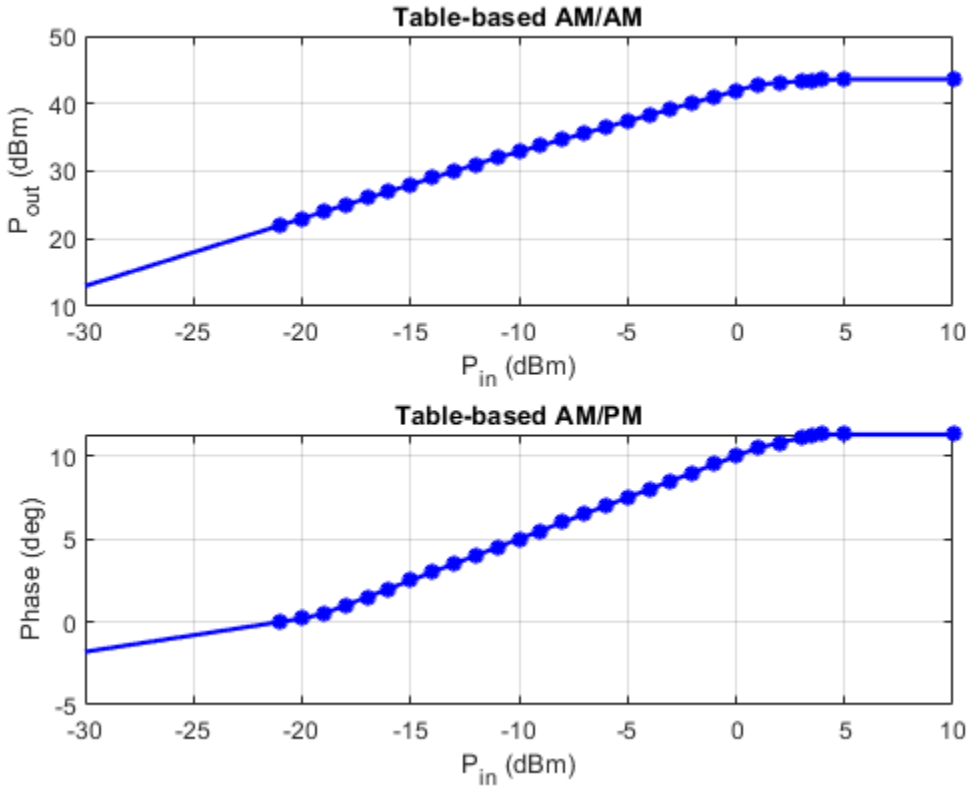
```
table = table2array(readtable("PACharacteristic.xlsx", ...
  "PreserveVariableNames",true));
mnl = comm.MemorylessNonlinearity("Method","Lookup table", ...
  "Table",table,"ReferenceImpedance",1);
```

Determine the input power that results in the peak output power. That input power is the point from which the signal will be backed off. Use the input backoff to determine the required signal power at the input to the amplifier.

```
[pkOpPwr, idxPk] = max(mnl.Table(:,2));  % dBm
ipPwrAtPkOut = mnl.Table(idxPk,1);  % dBm
IBO = 6;  % input backoff set point, dB
rqdIpPwr = ipPwrAtPkOut - IBO;  % dBm
```

Plot AM/AM and AM/PM amplifier characteristics. The plotted values match those in the spreadsheet.

```
plot(mnl);
```



System Simulation and Verification

Create a raised cosine transmit filter System object™ for pulse shaping.

```
txFilt = comm.RaisedCosineTransmitFilter(...
  'Shape','Square root', ...
  'RolloffFactor',0.2, ...
  'FilterSpanInSymbols',10, ...
  'OutputSamplesPerSymbol',4);
```

Create a power meter System object to measure power at multiple points in the processing chain. Set the measurement window of the power meter to 10 ms.

```
pm = powermeter(...
  "Measurement","Average power", ...
  "WindowLength",round(sigDuration*fs), ...
  "ReferenceLoad",mnl.ReferenceImpedance, ...
  "PowerUnits","dBm");
```

Generate a modulated signal, filter it, scale it to -10 dBm, and measure powers. The filtered signal is roughly constant amplitude throughout its duration, so the power measurement window can extend over the entire duration.

```
filtTransient = txFilt.FilterSpanInSymbols*txFilt.OutputSamplesPerSymbol;
msg = randi([0 M-1],msgLen+filtTransient,1);
modOut = qammod(msg,M,'UnitAveragePower',true);  % 0 dBW (30 dBm)
filtOut = txFilt(modOut);
filtOut = filtOut(1+filtTransient:end);  % Truncate beginning transient
```

```
PFiltOutdBm = pm(filtOut);
Pdesired = -10;  % dBm
scaleFactor = 10.^((Pdesired - PFiltOutdBm(end))/20);
filtOut = scaleFactor * filtOut;
reset(pm);
PFiltOutdBm = pm(filtOut);
fprintf('The filtered, scaled signal power is %4.2f dBm.\n',PFiltOutdBm(end))
```

```
The filtered, scaled signal power is -10.00 dBm.
```

```
PFiltOutdBW = PFiltOutdBm(end) - 30;
```

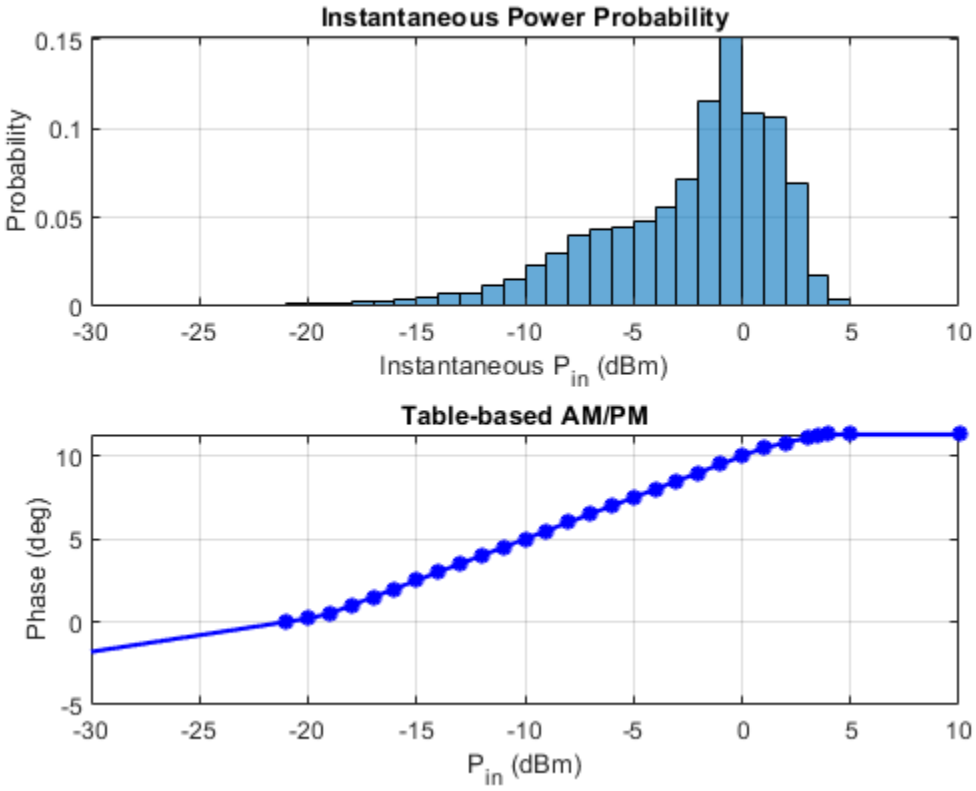
Scale the amplifier input power to the desired backoff. The measured power of the backed off signal must be equal to the input power at peak output (5 dBm) less the input backoff (6 dB). The power meter verifies that the signal has been properly backed off.

```
gain = helperBackoffGain(ipPwrAtPkOut,PFiltOutdBm(end),IBO);
ampIn = gain * filtOut;
reset(pm);
PAmpIndBm = pm(ampIn);
fprintf('The backed off signal power is %4.2f dBm.\n',PAmpIndBm(end))
```

```
The backed off signal power is -1.00 dBm.
```

Plot a histogram of instantaneous input power into the amplifier. The following figure shows that a significant percentage of the amplifier input samples have a power that should cause gain compression at the amplifier output. Many signal samples have powers above 0 dBm, where the amplifier behaves nonlinearly.

```
PAmpInInst = abs(ampIn).^2 / mnl.ReferenceImpedance;
PAmpInInstdBm = 10*log10(PAmpInInst) + 30;
edges = -29:9;
histogram(PAmpInInstdBm,edges,"Normalization","probability")
title("Instantaneous Power Probability");
xlabel("Instantaneous P_i_n (dBm)");
ylabel("Probability");
xlim([-30 10]);
grid on;
```



Pass the signal through the amplifier. The measured average power at the amplifier output closely corresponds to the expected instantaneous power illustrated by the previous figure.

```
ampOut = mnl(ampIn);
PAmpOutdBm = pm(ampOut);
fprintf('The amplifier output power is %4.2f dBm.\n',PAmpOutdBm(end))
```

```
The amplifier output power is 40.63 dBm.
```

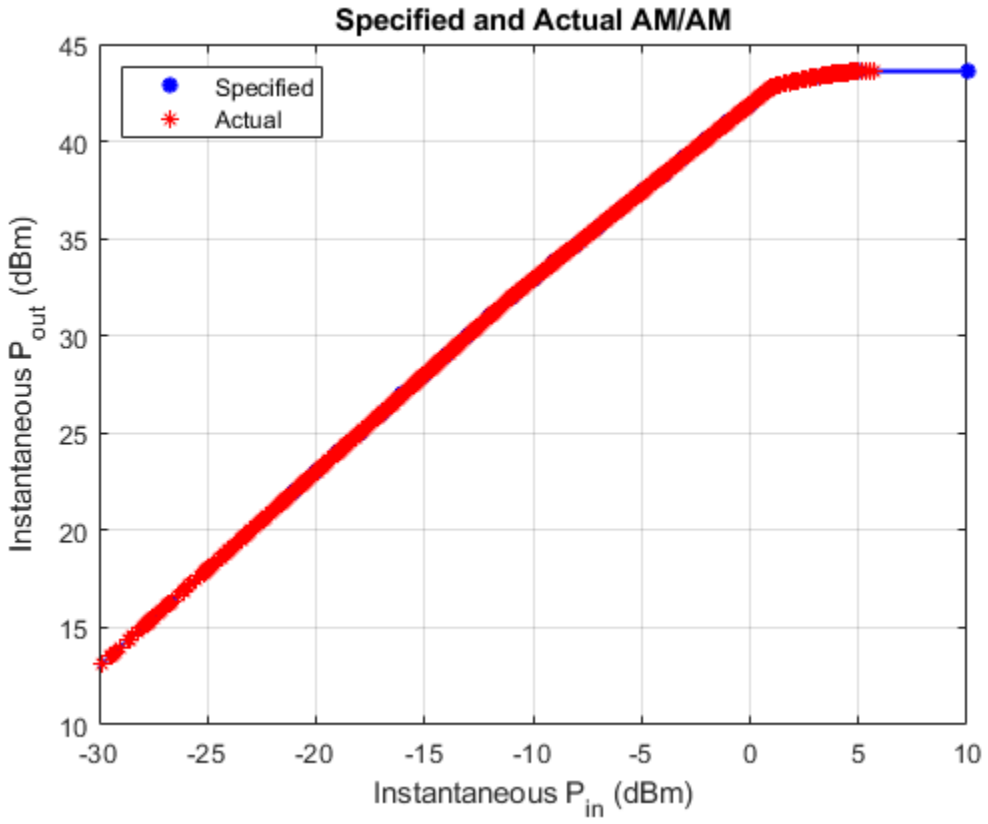
Calculate average amplifier gain.

```
ampGaindB = PAmpOutdBm(end) - PAmpIndBm(end);
fprintf('The amplifier gain is %4.2f dB.\n',ampGaindB)
```

```
The amplifier gain is 41.63 dB.
```

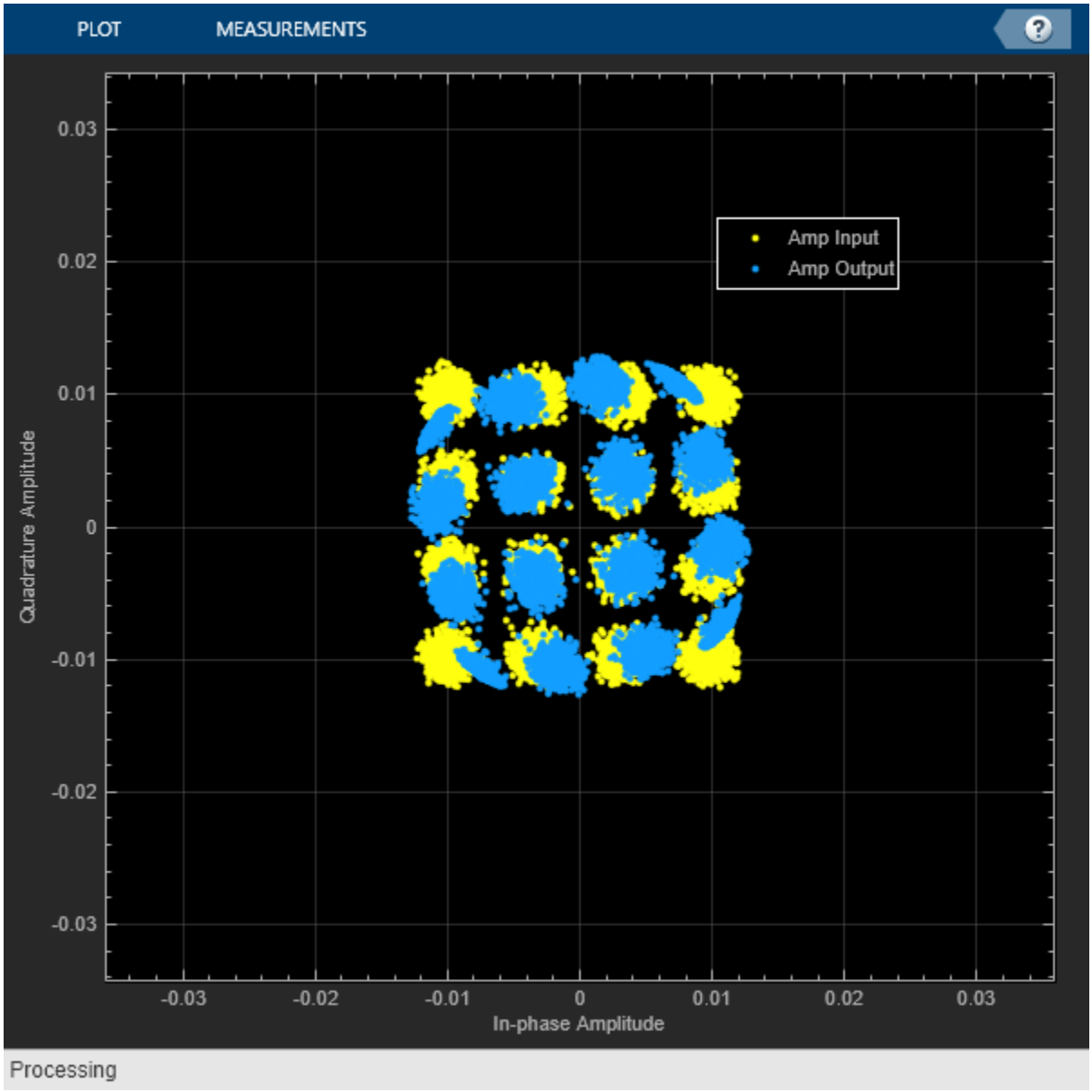
Plot the specified and actual instantaneous P_{out} vs. P_{in} to show that the actual behavior of the amplifier matches the behavior specified by the table-based object.

```
figure;
hFig = helperPlotAMAM(mnl); % Specified Pout vs. Pin
hold on;
pAmpOutInst = abs(ampOut).^2 / mnl.ReferenceImpedance;
pAmpOutInstdBm = 10*log10(pAmpOutInst) + 30;  % Actual Pout vs Pin
plot(PAmpInInstdBm,pAmpOutInstdBm,'r*');
grid on;
lines = hFig.Children.Children;
legend(lines([2 1]),"Specified","Actual","Location","Northwest");
```



Create a constellation diagram to illustrate the amplifier input and output signals. The constellation diagram of the 16QAM constellation shows the amplifier output has been slightly rotated (AM/PM distortion), and the corner points have incurred some gain compression (AM/AM distortion).

```
constDiag = comm.ConstellationDiagram(...
  'ShowReferenceConstellation',false, ...
  'SamplesPerSymbol',txFilt.OutputSamplesPerSymbol, ...
  'ShowLegend',true, ...
  'ChannelNames',{'Amp Input','Amp Output'});

% Set plot limits
maxLim = 2 * max(real(filtOut));
constDiag.XLimits = [-maxLim maxLim];
constDiag.YLimits = [-maxLim maxLim];

magFiltOut = sqrt(mean(abs(filtOut).^2));
magAmpOut = sqrt(mean(abs(ampOut).^2));
gain = magAmpOut / magFiltOut;
constDiag([filtOut,ampOut/gain]);  % Scale amp output for plotting ease
```



Exploring the Example

You can experiment with the example by trying different backoff levels or modulated signals (e.g. 64QAM or OFDM). You can load a spreadsheet with your own table-based P_{out} vs. P_{in}characteristics to apply this backoff technique to your PA characterization.

Summary

This example demonstrated how to apply backoff to the input signal of a nonlinear amplifier. The technique was verified by comparing P_{out} vs. P_{in} behavior of the specified and actual data.

Appendix

These helper files are used in the example:

- helperBackoffGain.m
- helperPlotAMAM.m

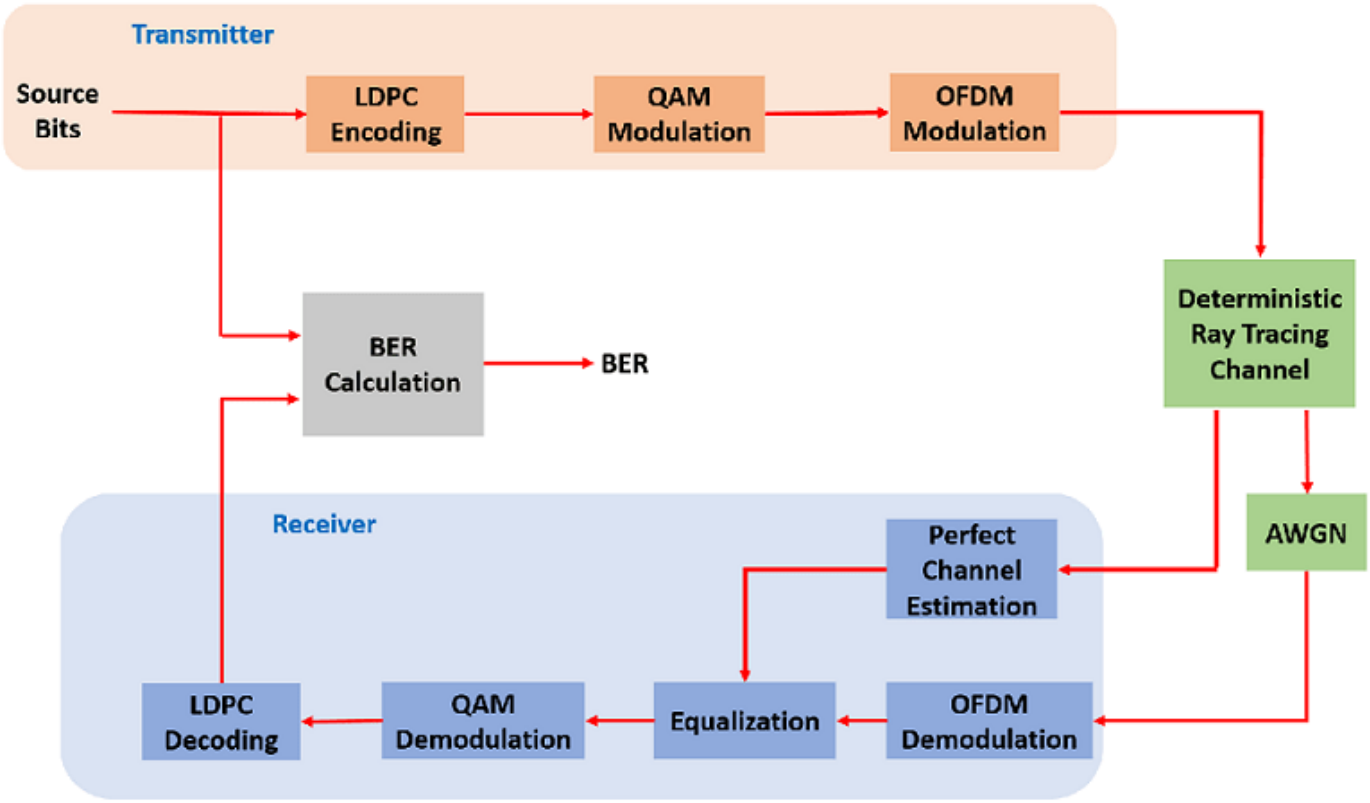
Indoor MIMO-OFDM Communication Link Using Ray Tracing

This example shows how to perform ray tracing in an indoor environment and use the results to build a channel model for a link level simulation with the MIMO-OFDM technique.

Introduction

Ray tracing [1] on page 8-0 has become a popular technique for radio frequency (RF) analysis, site planning, channel modelling, and link level analysis due to the trend for modern communications systems to operate at RF frequencies in the tens of GHz range. Unlike stochastic models, the ray tracing method is 3-D environment and transceiver sites specific and can have high sensitivity in the surrounding environment. Without a simple formula to calculate distance-based path losses, the ray tracing method relies on numeric simulations, and is typically less costly than field measurements. Results from ray tracing can be used to build multipath channel models for communication systems. For example, a ray tracing based channel model has been specified in Section 8 of TR 38.901 [2] on page 8-0 for 5G and in IEEE 802.11ay for WLAN [3] on page 8-0 .

This example starts with ray tracing analysis between one transmitter site and one receiver site in a 3-D conference room. Computed rays are used to construct a deterministic channel model which is specific for the two sites. The channel model is used in the simulation of a MIMO-OFDM communication link. This diagram characterizes the communication link.



The ray tracing is performed in an indoor environment. The same ray tracing methods can be applied to build channel models for indoor or outdoor environments. For ray tracing analysis in an outdoor urban setting, refer to the “Urban Link and Coverage Analysis Using Ray Tracing” on page 2-21 example.

3-D Indoor Scenario

Specify the indoor 3-D map in STL format for a small conference room with one table and four chairs. The STL format is one of the most common 3-D map formats and can often be converted from other 3-D map formats in a variety of 3-D software.

```
mapFileName = "conferenceroom.stl";
```

Define carrier frequency at 5.8 GHz and calculate wavelength

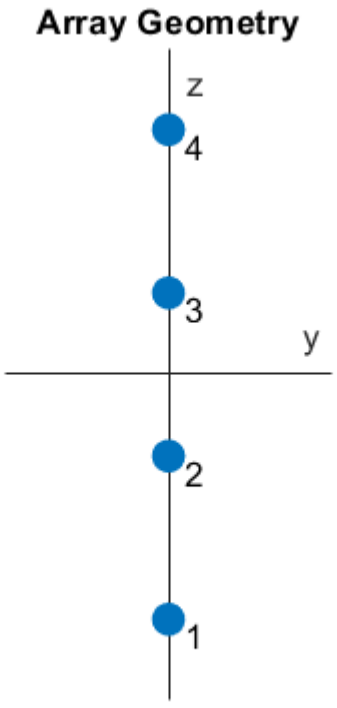
```
fc = 5.8e9;
lambda = physconst('lightspeed')/fc;
```

The transmit antenna is a 4-element uniform linear array (ULA) which has twice of the wavelength between the elements. The receive antenna is a 4x4 uniform rectangular array (URA) which has one wavelength between the elements. Both antennas are specified by an `arrayConfig` object.

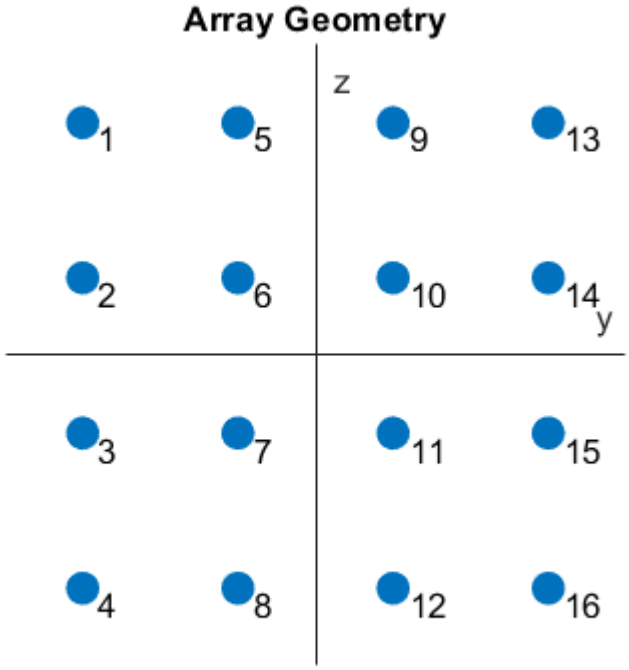
```
txArray = arrayConfig("Size",[4 1],'ElementSpacing',2*lambda);
rxArray = arrayConfig("Size",[4 4],'ElementSpacing',lambda);
```

Use the helperViewArray function to visualize the ULA and URA geometries where antenna elements are numbered for input/output streams.

```
helperViewArray(txArray);
```




```
helperViewArray(rxArray);
```



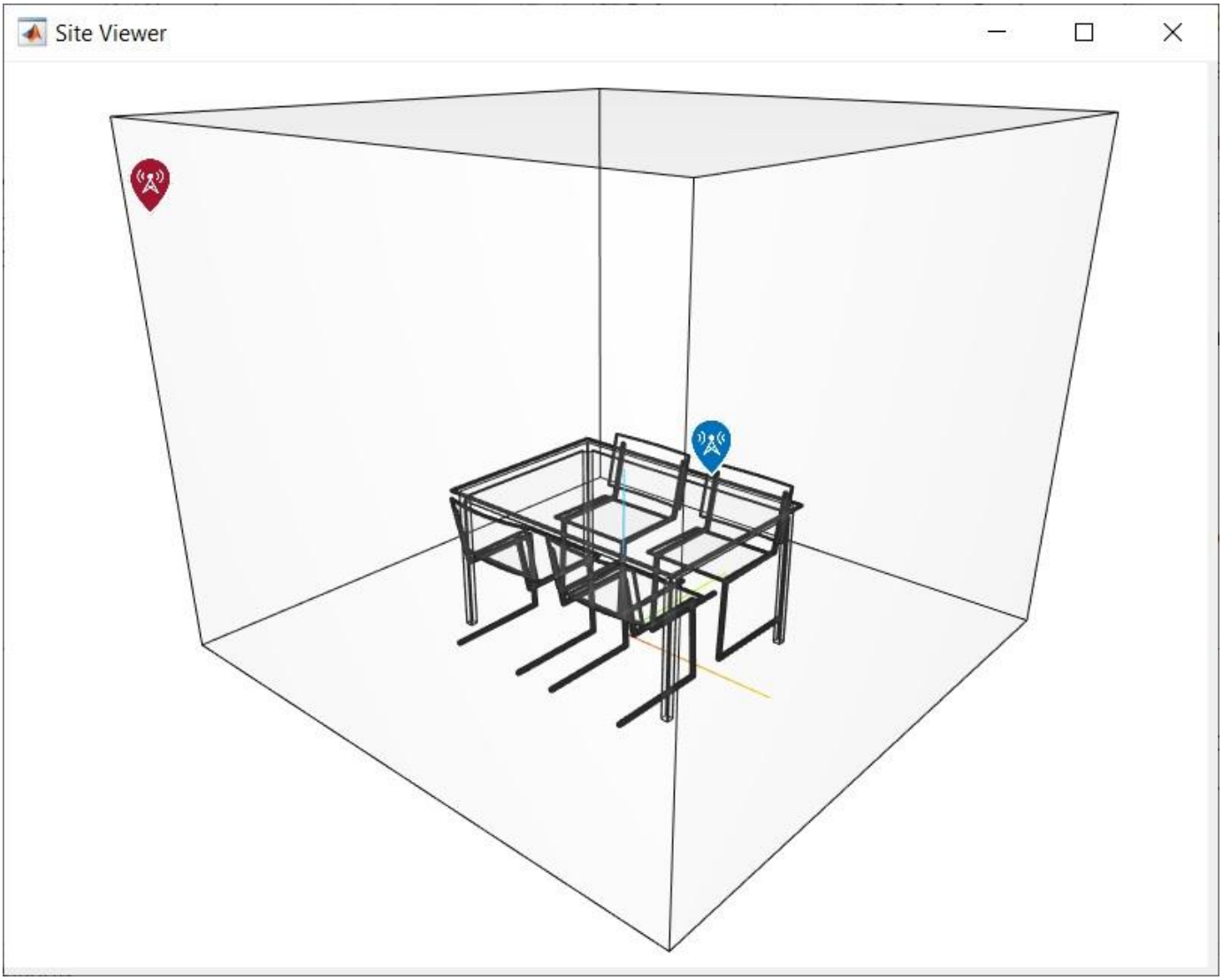
Specify a transmitter site close to the upper corner of the room, which can be a Wi-Fi access point. Specify a receiver site slightly above the table and in front of a chair to represent a laptop or mobile device.

```
tx = txsite("cartesian", ...
    "Antenna",txArray, ...
    "AntennaPosition",[-1.46; -1.42; 2.1], ...
    'TransmitterFrequency',5.8e9);

rx = rxsite("cartesian", ...
    "Antenna",rxArray, ...
    "AntennaPosition",[.3; .3; .85], ...
    "AntennaAngle",[0;90]);
```

Use the `siteviewer` function with the map file specified to view the scene in 3-D in Site Viewer. Use the `show` function to visualize the transmitters and receivers.

```
siteviewer("SceneModel",mapFileName);
show(tx,'ShowAntennaHeight', false)
show(rx,'ShowAntennaHeight', false)
```

Pan by left-clicking, zoom by right-clicking or by using the scroll wheel, and rotate the visualization by clicking the middle button and dragging or by pressing Ctrl and left-clicking and dragging.



Ray Tracing

Perform ray tracing analysis between the transmitter and receiver sites and return the `comm.Ray` objects, using the shooting and bouncing rays (SBR) method. Specify the surface material of the scene as wood and search for rays with up to 2 reflections. The SBR method supports up to 10 order of reflections.

```
pm = propagationModel("raytracing", ...
    "CoordinateSystem","cartesian", ...
    "Method","sbr", ...
    "AngularSeparation","low", ...
    "MaxNumReflections",2, ...
    "SurfaceMaterial","wood");

rays = raytrace(tx,rx,pm);
```

Extract the computed rays from the cell array return.

```
rays = rays{1,1};
```

Examine the ray tracing results by looking at the number of reflections, propagation distance and path loss value of each ray. There are 25 rays found (one line-of-sight ray, 6 rays with one reflection, and 18 rays with two reflections).

```
[rays.NumInteractions]
```

```
ans = 1×25

     0     1     1     1     1     1     1     2     2     2     2     2     2     2     2     2
```

```
[rays.PropagationDistance]
```

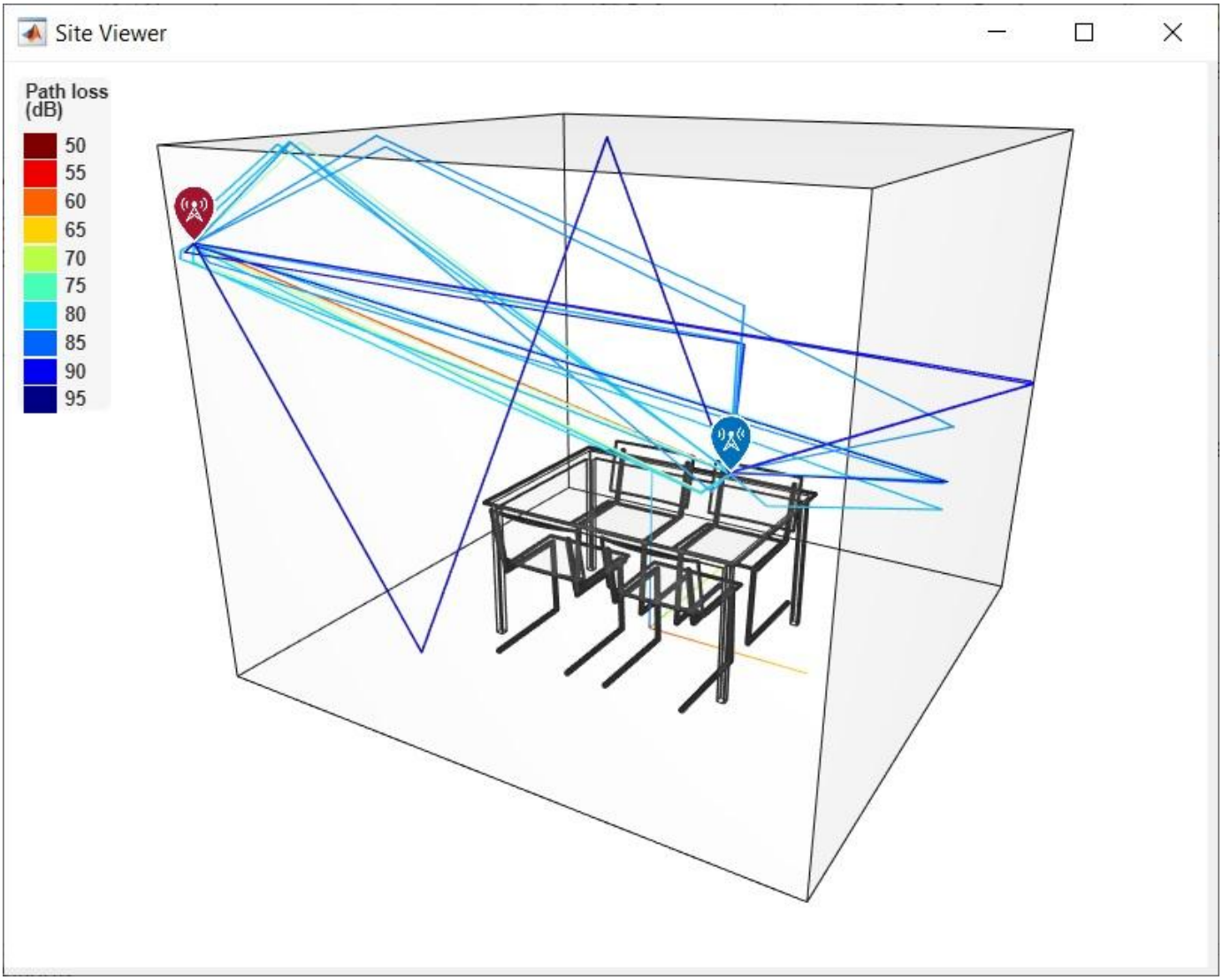
```
ans = 1×25

    2.7602    2.8118    2.8487    2.8626    3.2029    4.6513    4.6719    2.8988    2.9125    2.9
```

```
[rays.PathLoss]
```

```
ans = 1×25

   56.5350   72.1633   70.0647   72.3180   73.3102   76.4133   76.4508   81.5418   83.8254   81.5
```

Use the `plot` function to plot the rays in the 3-D scene in Site Viewer. Each ray is colored based on its path loss value. Click on a ray to view information about that ray.

```
plot(rays,'Colormap',jet,'ColorLimits',[50, 95])
```



Deterministic Channel Model from Ray Tracing

Create a deterministic multipath channel model using the above ray tracing results. Specify the instantaneous velocity of the receiver to reflect typical low mobility of a device in an indoor environment.

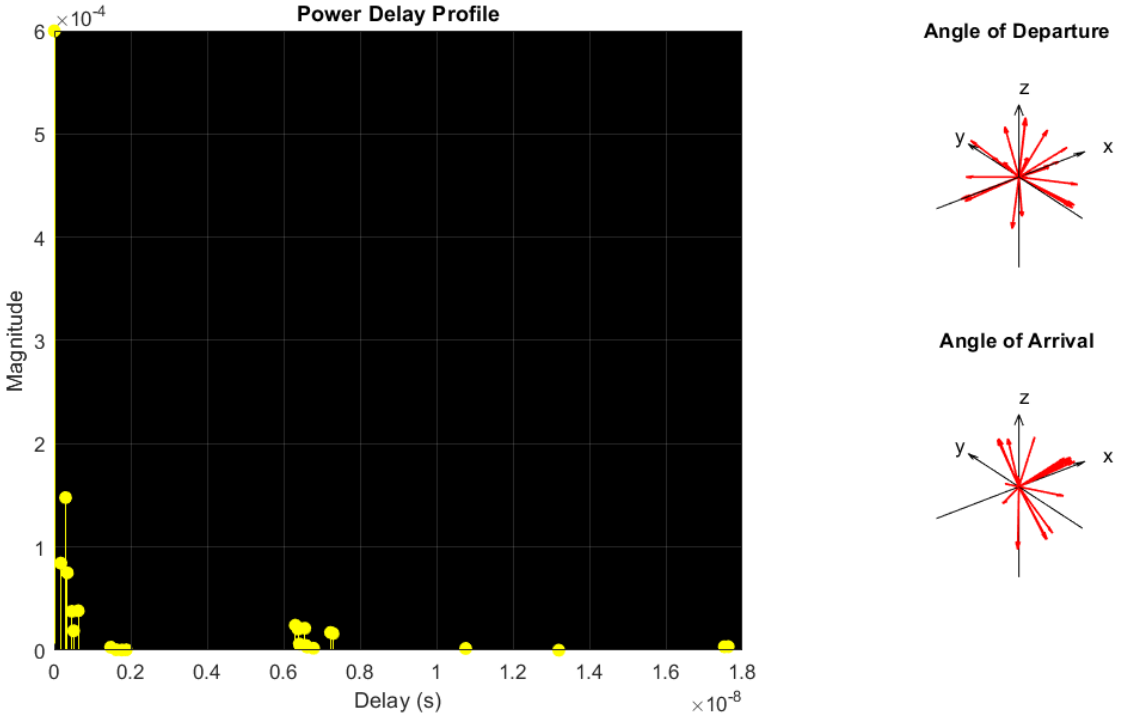
```
rtChan = comm.RayTracingChannel(rays,tx,rx);
rtChan.SampleRate = 300e6;
rtChan.ReceiverVirtualVelocity = [0.1; 0.1; 0]
```

```
rtChan =
  comm.RayTracingChannel with properties:

                    SampleRate: 300000000
               PropagationRays: [1×25 comm.Ray]
      MinimizePropagationDelay: true
                  TransmitArray: [1×1 arrayConfig]
  TransmitArrayOrientationAxes: [3×3 double]
                  ReceiveArray: [1×1 arrayConfig]
```

```
    ReceiveArrayOrientationAxes: [3×3 double]
        ReceiverVirtualVelocity: [3×1 double]
      NormalizeImpulseResponses: true
        NormalizeChannelOutputs: true
                ChannelFiltering: true
```

Use the `showProfile` object function to visualize the power delay profile (PDP), angle of departure (AoD) and angle of arrival (AoA) of the rays in the channel. In the visualization, the PDP has taken into account the transmit and receive array pattern gains in addition to the path loss for each ray.

```
showProfile(rtChan);
```



Use the `info` object function to obtain the number of transmit and receive elements.

```
rtChanInfo = info(rtChan)
```

```
rtChanInfo = struct with fields:
             CarrierFrequency: 5.8000e+09
             CoordinateSystem: 'Cartesian'
        TransmitArrayLocation: [3×1 double]
         ReceiveArrayLocation: [3×1 double]
           NumTransmitElements: 4
            NumReceiveElements: 16
            ChannelFilterDelay: 7
    ChannelFilterCoefficients: [25×21 double]
          NumSamplesProcessed: 0
                LastFrameTime: 0
```

```
numTx = rtChanInfo.NumTransmitElements;
numRx = rtChanInfo.NumReceiveElements;
```

System Parameters

Configure a communications link that uses LDPC coding, 64-QAM and OFDM with 256 subcarriers. Specify 4 LDPC codewords per frame, which results in 50 OFDM symbols per frame.

```
% Create LDPC encoder and decoder configuration objects
cfgLDPCEnc = ldpcEncoderConfig(dvbs2ldpc(1/2));
cfgLDPCDec = ldpcDecoderConfig(cfgLDPCEnc);
numCodewordsPerFrame = 4;
codewordLen = cfgLDPCEnc.BlockLength;

% Parameters for QAM modulation per subcarrier
bitsPerCarrier = 6;
modOrder = 2^bitsPerCarrier;
codeRate = cfgLDPCEnc.CodeRate;

% Create OFDM modulator and demodulator objects
fftLen = 256;
cpLen = fftLen/4;
numGuardBandCarriers = [9; 8];
pilotCarrierIdx = [19:10:119, 139:10:239]';
numDataCarriers = ...
    fftLen - sum(numGuardBandCarriers) - length(pilotCarrierIdx) - 1;
numOFDMSymbols = ...
    numCodewordsPerFrame * codewordLen / ...
    bitsPerCarrier / numDataCarriers / numTx;
ofdmMod = comm.OFDMModulator( ...
    "FFTLength",fftLen, ....
    "NumGuardBandCarriers",numGuardBandCarriers, ...
    "InsertDCNull",true, ...
    "PilotInputPort",true, ...
    "PilotCarrierIndices",pilotCarrierIdx, ...
    "CyclicPrefixLength",cpLen, ...
    "NumSymbols",numOFDMSymbols, ...
    "NumTransmitAntennas",numTx);
ofdmDemod = comm.OFDMDemodulator(ofdmMod);
ofdmDemod.NumReceiveAntennas = numRx;
```

Create an error rate calculation object to compute bit error rate (BER).

```
errRate = comm.ErrorRate;
```

Assign Eb/No value and derive SNR value from it for AWGN.

```
EbNo = 30;                                  % in dB
bitsPerSymbol = bitsPerCarrier*codeRate;
snr = 10^(EbNo/10) * bitsPerSymbol;      % Linear
```

Link Simulation

The helperIndoorRayTracingWaveformGen function generates a waveform consisting of one frame at the transmitter site by performing these following steps:

1 Encode randomly generated bits by LDPC

2. Modulate encoded bits by 64-QAM
3. Apply OFDM modulation to convert signals from frequency domain to time domain

```
rng(100); % Set RNG for repeatability
[txWave,srcBits] = ...
    helperIndoorRayTracingWaveformGen( ...
    numCodewordsPerFrame,cfgLDPCEnc,modOrder,ofdmMod);
```

Pass the waveform through the ray tracing channel model and add white noise. To account for channel filtering delay, append an additional null OFDM symbol to the end of the waveform.

```
chanIn = [txWave; zeros(fftLen + cpLen,numTx)];
[chanOut,CIR] = rtChan(chanIn);
rxWave = awgn(chanOut,snr,numTx/numRx,'linear');
```

The helperIndoorRayTracingRxProcessing function decodes the channel-impaired waveform at the receiver site by performing these following steps:

1. Perfect channel estimation using the channel impulse response (CIR) output and the channel filter coefficients from the channel object's `info` method.
2. OFDM demodulation to bring the signals back into frequency domain
3. Symbol equalization on each subcarrier
4. Soft 64-QAM demodulation to get LLR
5. LDPC decoding

```
[decBits, eqSym] = ...
    helperIndoorRayTracingRxProcessing(rxWave,CIR, ...
    rtChanInfo,cfgLDPCDec,modOrder,ofdmDemod,snr);
```

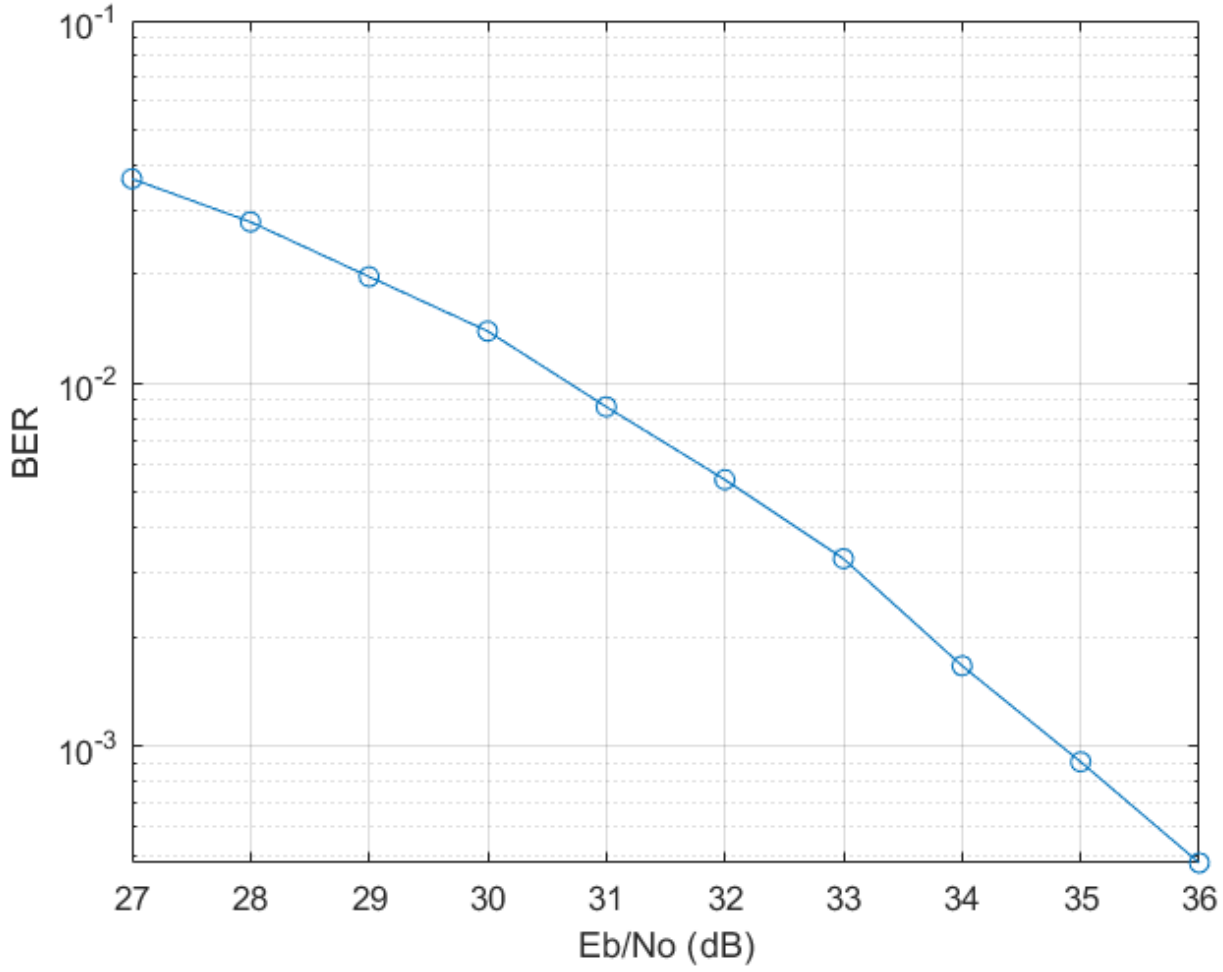
Calculate BER:

```
ber = errRate(srcBits,double(decBits));
disp(ber(1));
```

```
    0.0140
```

To plot a BER curve against a range of EbNo values, use the helperIndoorRayTracingSimulationLoop function to repeat the above single frame processing for up to 300 frames at each EbNo value.

```
EbNoRange = 27:36;
helperIndoorRayTracingSimulationLoop( ...
    cfgLDPCEnc,cfgLDPCDec,ofdmMod,ofdmDemod,rtChan,errRate, ...
    modOrder,numCodewordsPerFrame,EbNoRange);
```



Conclusion and Further Exploration

This example shows how to build a deterministic channel model using ray tracing results in an indoor conference room. Link-level simulations using LDPC and MIMO-OFDM techniques were performed for the channel model and BER results were plotted.

Further exploration includes but not limits to:

- Different 3-D maps and/or surface materials
- Different transmitter and/or receiver site positions
- Different transmit and/or receive antenna array specifications
- Different transmit and/or receive antenna array orientations
- Higher number of reflections for the SBR ray tracing method
- Transmit and/or receive beamforming

Appendix

This example uses the following helper functions:

- helperEqualize.m
- helperPerfectChannelEstimate.m
- helperIndoorRayTracingRxProcessing.m
- helperIndoorRayTracingSimulationLoop.m
- helperIndoorRayTracingWaveformGen.m
- helperViewArray.m

Selected Bibliography

[1] Z. Yun, and M. F. Iskander, "*Ray tracing for radio propagation modeling: Principles and applications*," *IEEE Access*, vol. 3, pp. 1089-1100, Jul. 2015.

[2] 3GPP TR 38.901. *Study on channel model for frequencies from 0.5 to 100 GHz.* 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.

[3] Maltsev, A., *et al. Channel Models for 802.11ay*. IEEE 802.11-15/1150r9, March 2017.

See Also

Functions

`propagationModel` | `raytrace`

Objects

`arrayConfig` | `siteviewer` | `comm.RayTracingChannel` | `ldpcEncoderConfig`

Related Examples

- "Ray Tracing for Wireless Communications" on page 30-12

Effect of a High-Power Interferer on ADC Performance

This example shows the effect of a high-power in-band or out-of-band interferer on the performance of a communications system with an analog-to-digital converter (ADC).

Introduction

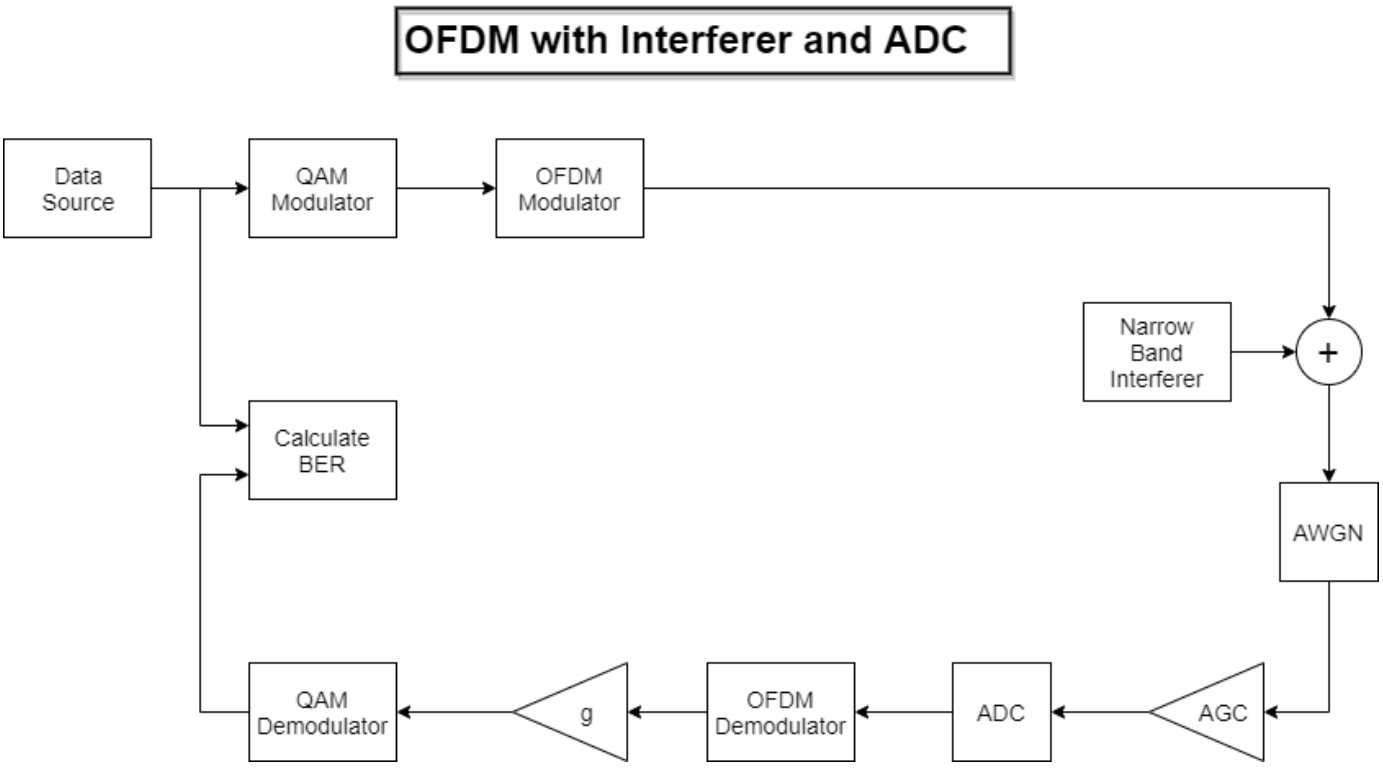
Ideal multiuser communication systems, that use orthogonal frequency division multiplexed (OFDM) signals and forward error correction (FEC), are essentially immune to high-power narrowband interference because the narrowband interference affects only one or two subcarriers. For in-band interference, FEC can recover the bit errors caused by these jammed subcarriers. For out-of-band interferers, bandpass filtering can remove the adjacent channel interference in these ideal multiuser systems.

In practical systems, an ADC digitizes signals received at the antenna. Since the ADC has a fixed full-scale voltage V_{fs}, the input signal is first scaled to the $[-V_{fs}, V_{fs})$ range. If the ADC has N bits of resolution, then the maximum quantization error is given by $2V_{fs}/2^{N+1}$. In a system with sufficient bits of resolution (for example N=16) and no interfering signal, this quantization error is negligible as compared to other noise sources in the system and can be ignored.

In the presence of a high-power interferer, the automatic gain controller (AGC) scales the whole signal to fit in the full-scale range of the ADC. The scaling effectively reduces the number of bits used to represent the desired signal. Since the quantization error does not change, the effective signal to noise ratio decreases. Depending on the power of the interfering signal and the number of ADC bits, the system performance can be adversely affected.

Simulating Effect of Narrowband Interferer on OFDM Signals

Generate an OFDM signal with 128 subcarriers. Assign a 64-QAM modulated signal to each subcarrier. To exaggerate the quantization error effects, set the number of ADC bits to 7. Assume an AWGN channel with 30 dB SNR for simplicity.




```
M = 64;           % Modulation order per subcarrier
numSC = 128;      % Number of OFDM subcarriers
SNR = 30;         % Signal-to-noise ratio in dB
numADCBits = 7;   % Number of ADC bits
```

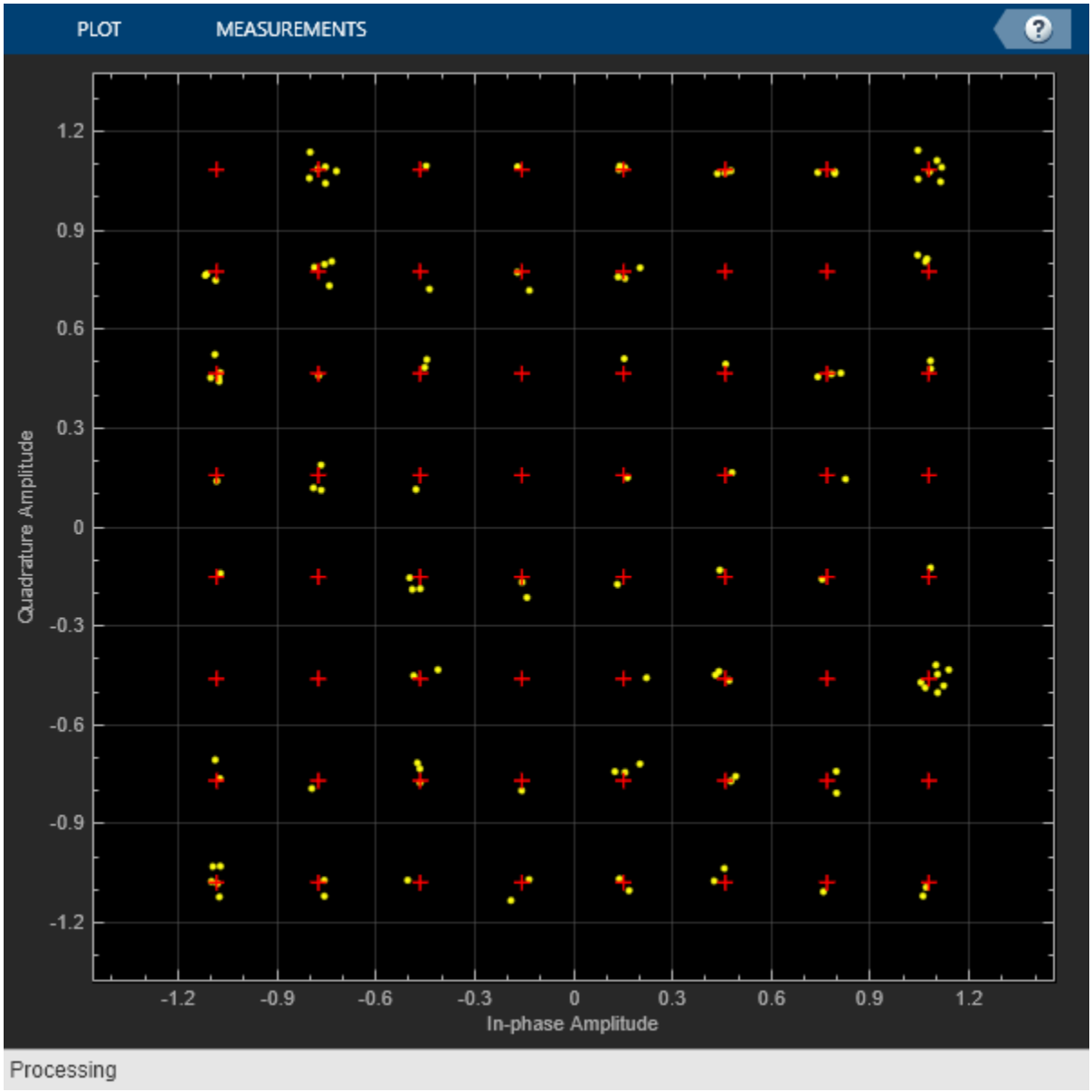
OFDM with ADC Over AWGN Channel

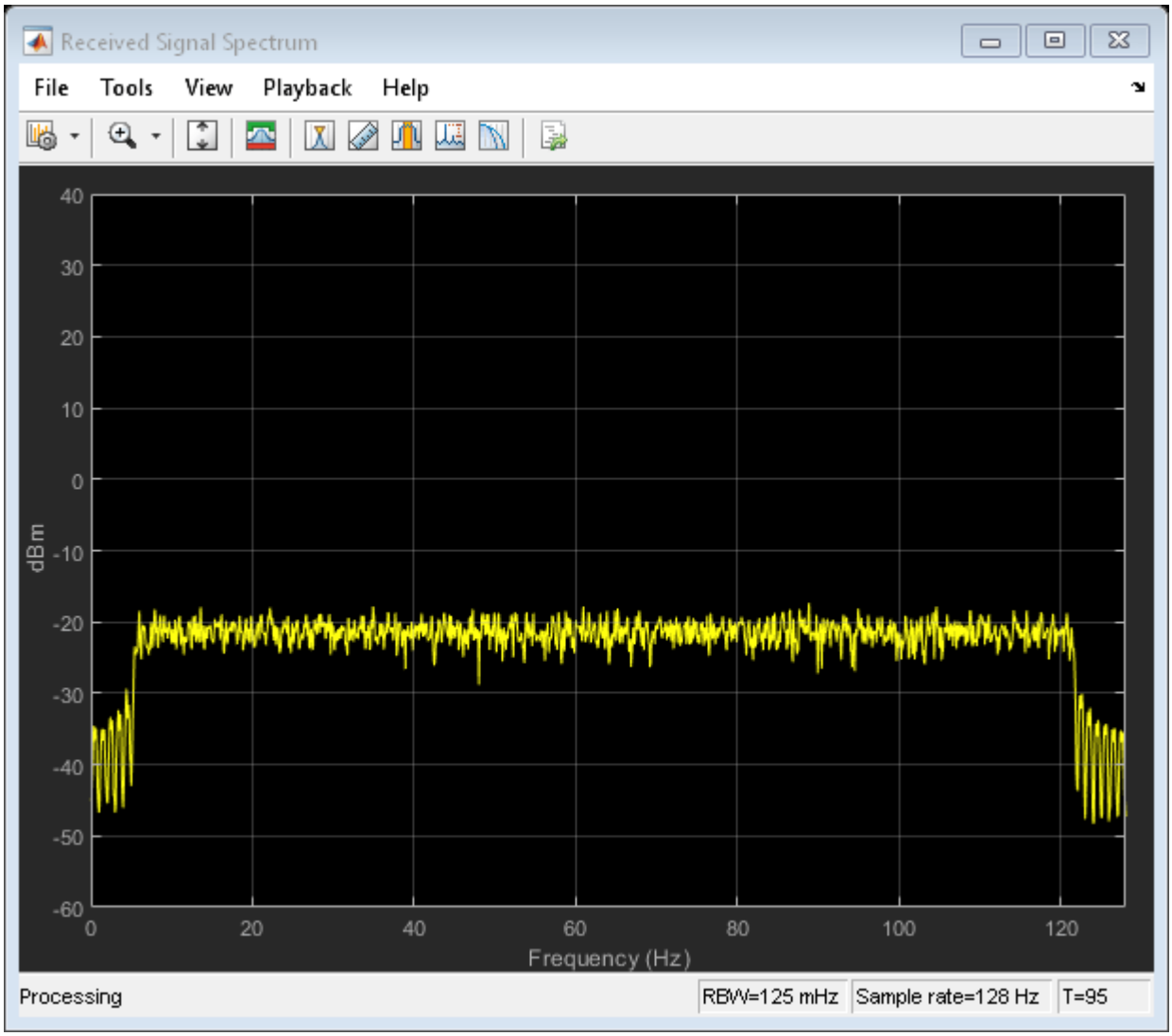
Pass the generated OFDM signal through an AWGN channel. The AGC scales the received signal to [-1 1] range. Pass the scaled signal through the bipolar ADC. Rescale the signal before applying OFDM and QAM demodulation. The `narrowbandInterfererAndOFDM` function simulates this system.

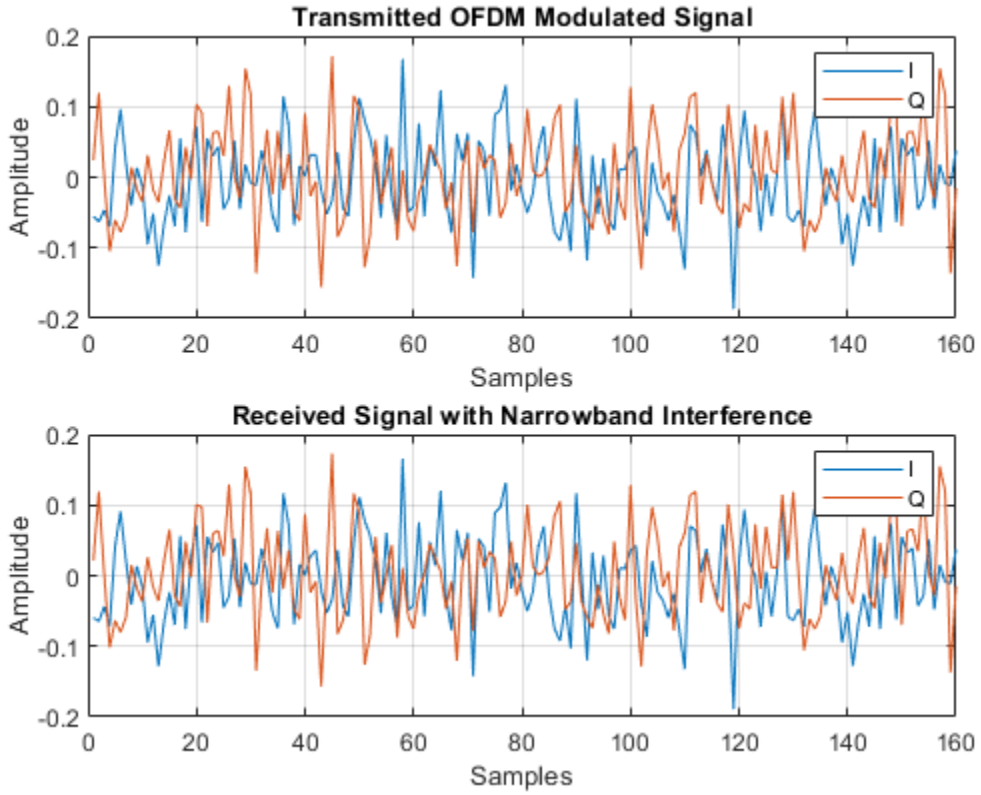
Run the simulation without interference. All the bits can be received without errors.

```
interfererAmp = 0;
ber = narrowbandInterfererAndOFDM(M,numSC,interfererAmp,numADCBits,SNR);
disp('BER:')
disp(ber)
```

```
BER:
     0
```


PLOT
MEASUREMENTS
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Quadrature Amplitude
-1.2
-0.9
-0.6
-0.3
0
0.3
0.6
0.9
1.2
In-phase Amplitude
Processing


Received Signal Spectrum
File
Tools
View
Playback
Help
40
30
20
10
0
-10
-20
-30
-40
-50
-60
dBm
0
20
40
60
80
100
120
Frequency (Hz)
Processing
RBW=125 mHz
Sample rate=128 Hz
T=95

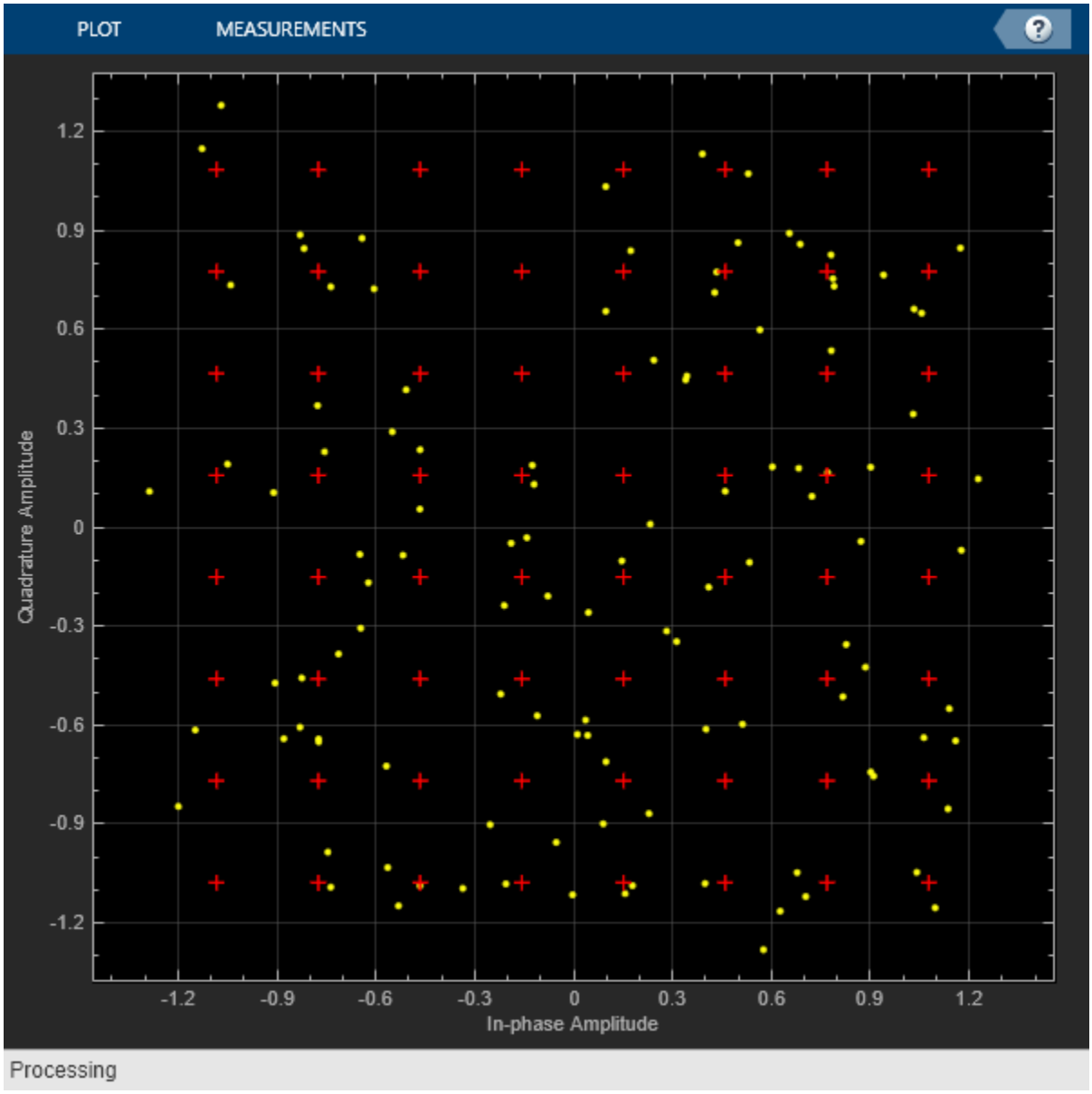


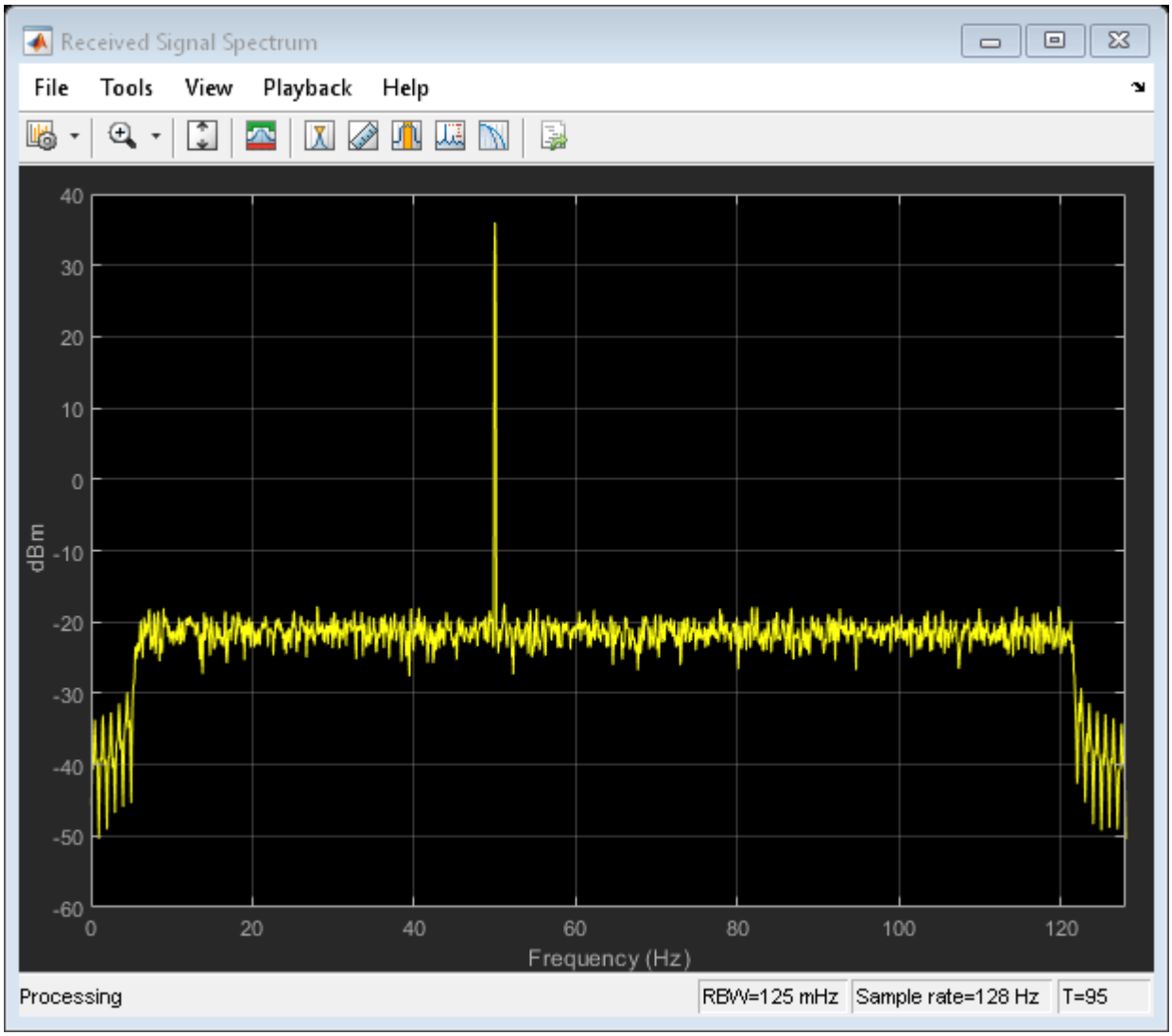
OFDM with ADC Over AWGN Channel with High-Power Interferer

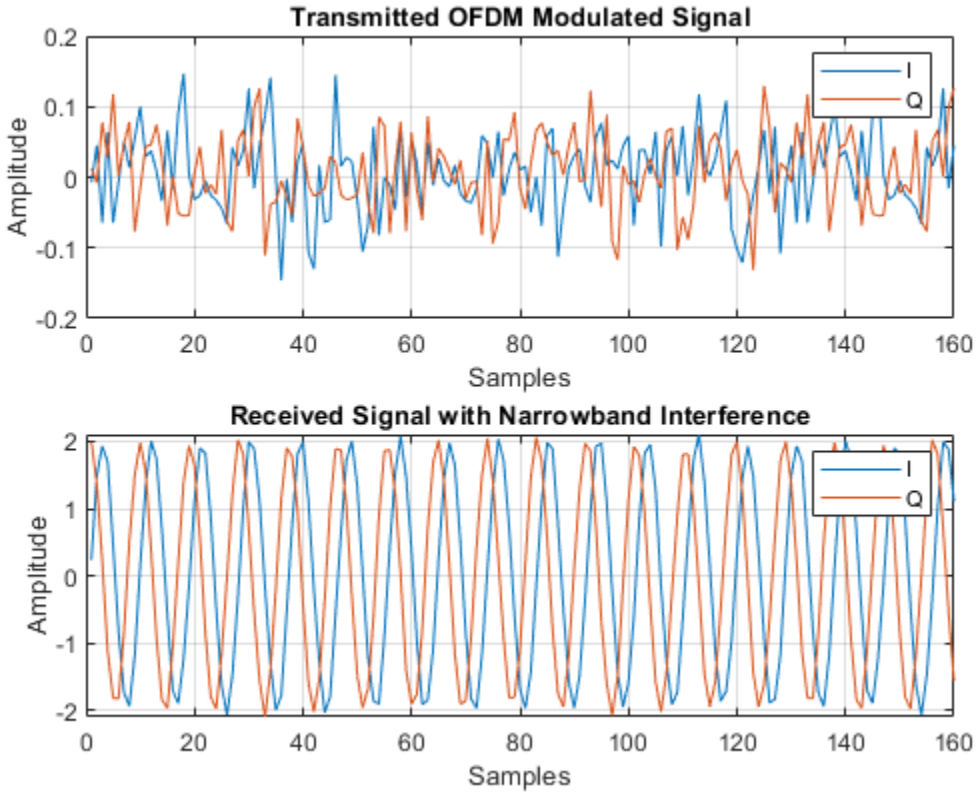
Use a tone to interfere with the 50th subcarrier of the OFDM signal. Set the amplitude of the interferer to 2 corresponding to an SIR value of about -28 dB. The high amplitude of the interfering signal forces the AGC to reduce its gain to avoid saturation. This scaling decreases the number of bits assigned to the desired signal and reduces the effective power of the desired signal. Quantization noise is a function of the fixed full-scale voltage and the number of bits properties of the ADC. As a result, the effective signal to noise ratio (SNR) decreases and the system starts to introduce bit errors.

```
interfererAmp = 2;
ber = narrowbandInterfererAndOFDM(M,numSC,interfererAmp,numADCBits,SNR);
disp('BER:')
disp(ber)
```

```
BER:
    0.0531
```

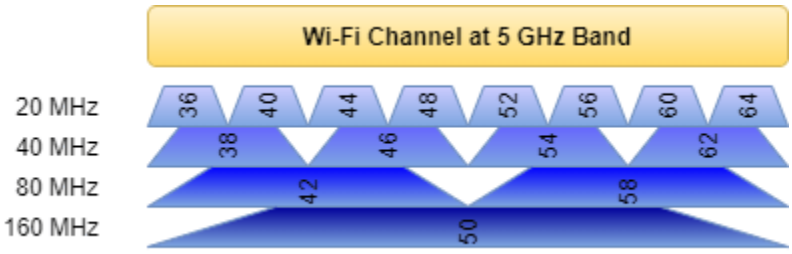

PLOT
MEASUREMENTS
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Quadrature Amplitude
-1.2
-0.9
-0.6
-0.3
0
0.3
0.6
0.9
1.2
In-phase Amplitude
Processing


Received Signal Spectrum
File
Tools
View
Playback
Help
40
30
20
10
0
-10
-20
-30
-40
-50
-60
dBm
0
20
40
60
80
100
120
Frequency (Hz)
Processing
RBW=125 mHz
Sample rate=128 Hz
T=95



Effect of Adjacent Channel Users on a Multiuser System

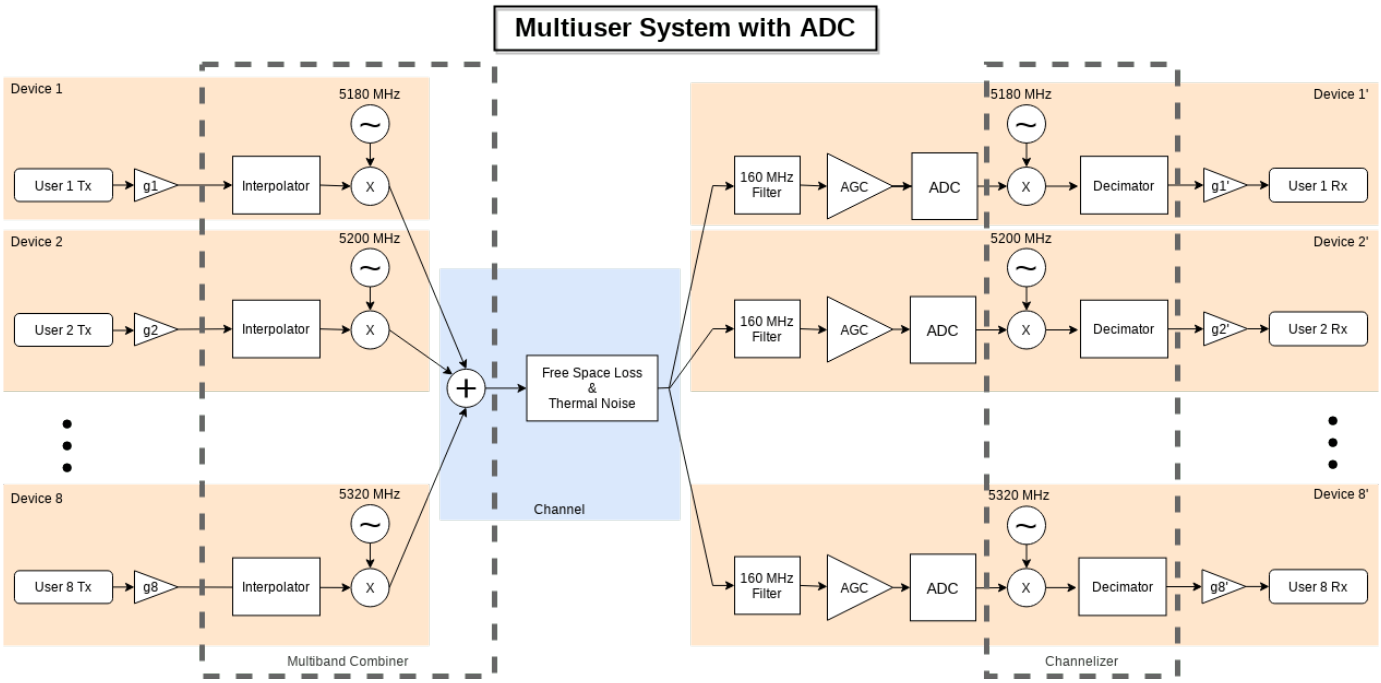
Modern communication systems define multiple signal bandwidths to provide flexibility in choosing between highly reliable connections or high throughput. For example, 802.11 WLAN standard defines channel bandwidths that range from 20 MHz to 160 MHz. This figure shows the available WLAN channel bandwidths.



Typically, such systems are designed with fixed high bandwidth analog RF filters followed by programmable digital filters. An AGC and ADC combo is used to digitize the analog signal. If one of the users (i.e. channels) has much higher power than the rest of the users, the ADC quantization may cause a low SNR value for the low power users. The following demonstrates such a scenario.

Consider a Wi-Fi like system where there are eight independent transmitters (Device 1-8) and eight independent receivers (Device 1'-8'). Each transmitter-receiver pair is assigned one of the available 20 MHZ bands. 64-QAM modulated signals are OFDM modulated with 56 subcarriers in a bandwidth of 20 MHz. As shown in this figure, eight possible users are carried over channels 36, 40, 44, 48, 52,

56, 60, and 54, with corresponding carrier frequencies (5180:20:5320) MHz. The receivers employ analog filters that pass through the whole available 160 MHz band then use channelizer filters to select the desired user. To simplify the simulation, assume same path loss and thermal noise for each device pair. Also, the simulator uses the multiband combiner to combine signals from the eight users in the channel and channelizer to separate them in an efficient way. The dotted lines show the multiband combiner and channelizer.




```
M = 64;           % Modulation order per subcarrier
noiseFigure = 7;  % Noise figure in dB
numADCBits = 7;   % Number of ADC bits
```

Multiuser System with ADC Over AWGN Channel

Generate OFDM modulated signals for all the active users and combine them using a `comm.MultibandCombiner` System object. Apply a path loss equivalent to a nominal distance of 10 meters. Pass the signal through an RF front-end with a noise figure of 7 dB to mimic an AWGN channel. The AGC scales the received signal to a [-1 1] range. Pass the scaled signal through the bipolar ADC. Rescale the signal after passing through the channelizer filter, which separates the user signals. Then apply OFDM and QAM demodulation. All the bits can be received without errors. The `multiuserInterferenceAndADC` function simulates this system.

Set all users as active with all users 0 dB relative gain. Run the simulation. All users operate without errors.

```
activeUsers = [1 1 1 1 1 1 1 1];
userGaindB = [0 0 0 0 0 0 0 0];

ber = multiuserInterferenceAndADC(M,noiseFigure,numADCBits,activeUsers,userGaindB);

disp('BER for each user:')
disp(ber)
```

```
BER for each user:
     0     0     0     0     0     0     0     0
```

Multiuser System with ADC Over AWGN Channel with High Power User

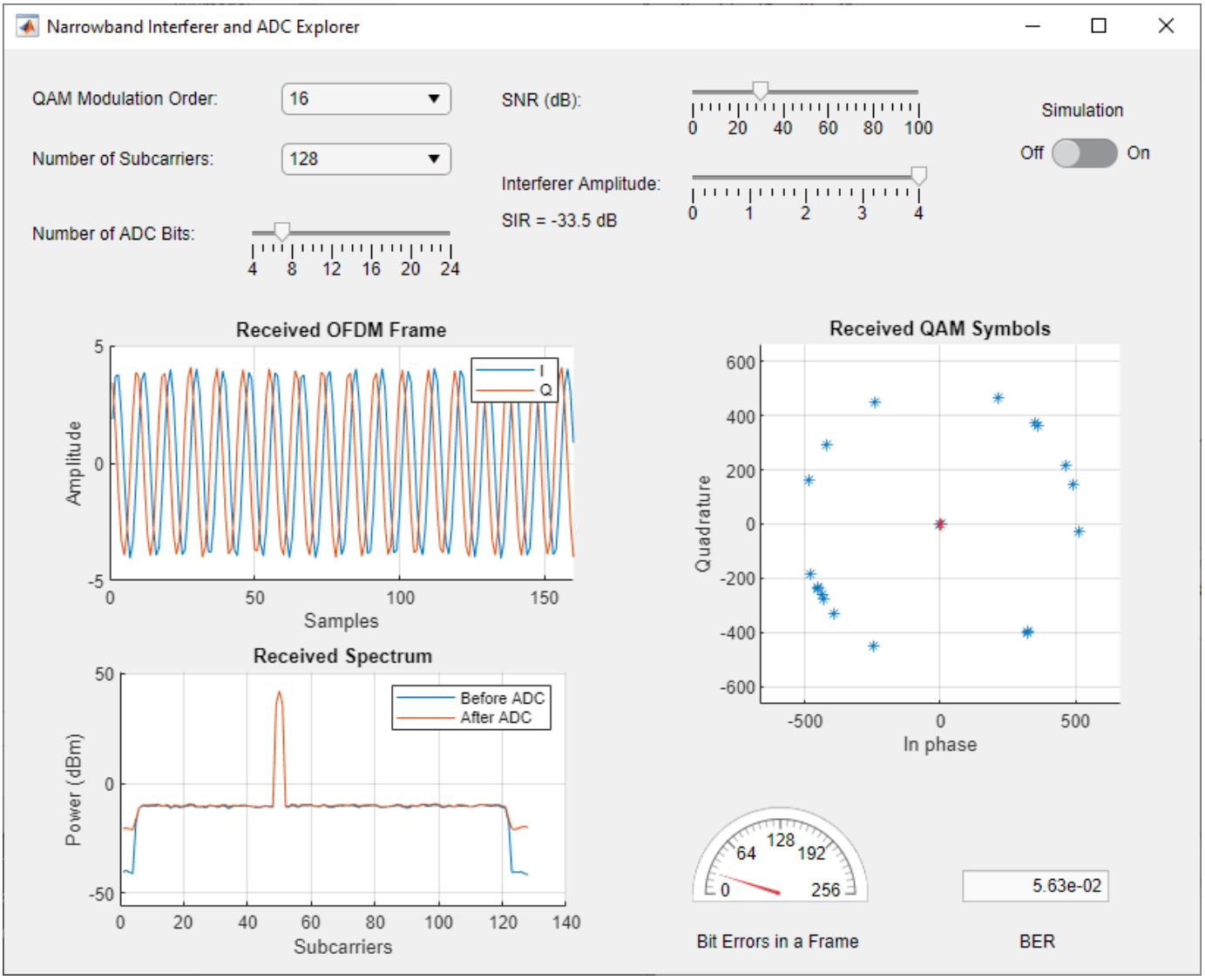
Repeat the same experiment with a high-power user. Set the relative gain of the third user to 30 dB. Due to the decrease in the effective signal power as compared to the quantization noise (except the high-power user), the low power users experience bit errors and the BER performance degrades.

```
userGaindB = [0 0 30 0 0 0 0 0];
ber = multiuserInterferenceAndADC(M,noiseFigure,numADCBits,activeUsers,userGaindB);

disp('BER for each user:')
disp(ber)
```

```
BER for each user:
  Columns 1 through 7

    0.0369    0.0404         0    0.0408    0.0364    0.0383    0.0392

  Column 8

    0.0382
```

Further Exploration

The Narrowband Interferer and ADC Explorer app helps you quickly try different system settings to explore the effect of a high-power narrowband interferer on the system performance due to the fixed full-scale voltage and the quantization noise introduced by the ADC. Run the `Narrowband Interferer and ADC Explorer app`.



- Click "Simulation" switch to start the simulations.
- Change "QAM Modulation order" to 16
- Increase the interferer amplitude to 4. Subcarrier 50 experiences interference by the narrowband interferer. "Bit errors in a Frame" gauge shows bit error between 0 and 4 bits since a single subcarrier is affected
- Reduce the "Number of ADC Bits" in and observe the received spectrum and bit errors in a frame. Around 7 bits, the ADC quantization errors start to degrade the system performance noticeably.

Experiment with different SNR and modulation order values and find out the limits of the system to handle a high-power narrowband interferer.

The Multiuser Interference and ADC Explorer app helps you quickly try different system settings to explore the effect of multiuser interference on the system performance due to the fixed full-scale voltage and the quantization noise introduced by the ADC. Run the `Multiuser Interference and ADC Explorer app`.

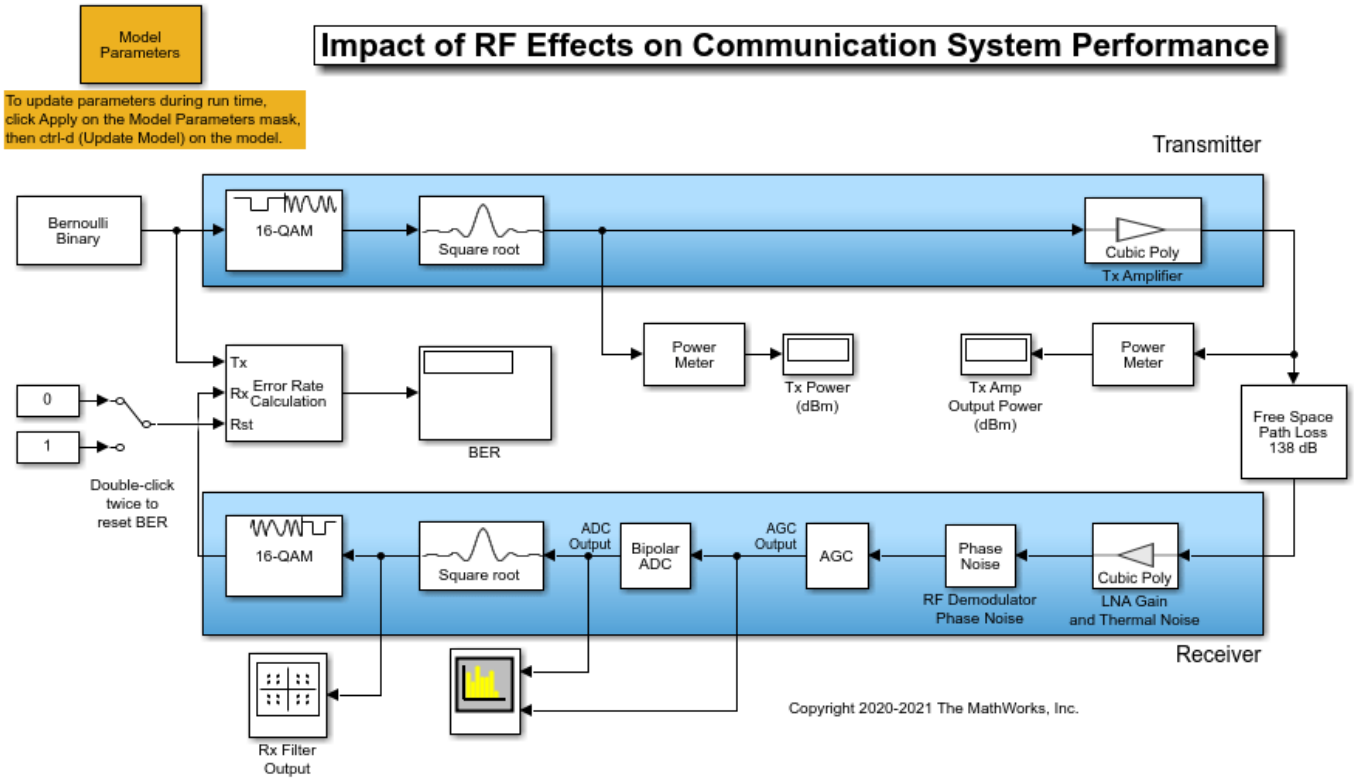
- Click "Simulation" switch to start the simulations.
- Change "QAM Modulation order" to 64.
- Increase the gain of the 1st user to 40 dB.
- Decrease the number of ADC bits in small steps. The noise floor in the received spectrum starts increasing. Around 10 bits, the low power users start to experience bit errors.
- Reducing the number of ADC bits to 5, raises the noise floor above the signal level.

Impact of RF Effects on Communication System Performance

This example shows how to use Communications Toolbox™ blocks to model thermal noise, phase noise, and nonlinearity impairments of an RF transceiver. The model measures the effects of the impairments on the bit error rate (BER) of a communications system.

Overview

The model, shown in the following figure, includes blocks to simulate a transmitter, a channel, a receiver, and to measure and visualize communications link performance.



The transmitter models:

- A 16QAM-modulated waveform of random bits
- A square root raised cosine (RRC) pulse-shaping filter to limit spectral leakage and minimize interference (ISI)
- A memoryless power amplifier (PA) with an ideal (infinite) third order intercept (IIP3). The IIP3 value can be changed to model a more realistic PA. The transmitter PA models the third order nonlinearity because it is the major source of degradation at that end of the link.

The channel models 138 dB of free space path loss.

The RF receiver front end models the analog portion of the receiver, prior to analog-to-digital conversion. It includes:

- A low noise amplifier (LNA) with an ideal noise figure (NF) of 0 dB and a power gain of 20 dB. The NF can be changed to model a more realistic LNA. At this end of the link, noise is a much more significant source of degradation than nonlinearity.
- An RF demodulator (RFD) with minimal phase noise. This value can also be changed to model a more realistic RFD. The phase noise can be a significant source of degradation for a 16QAM link.
- An automatic gain control (AGC) to properly scale the signal prior to quantizing.

The remainder of the receiver models:

- An idealized analog-to-digital converter (ADC) with 12 bits of quantization
- An RRC filter for noise reduction and ISI minimization
- A hard decision 16QAM demodulator

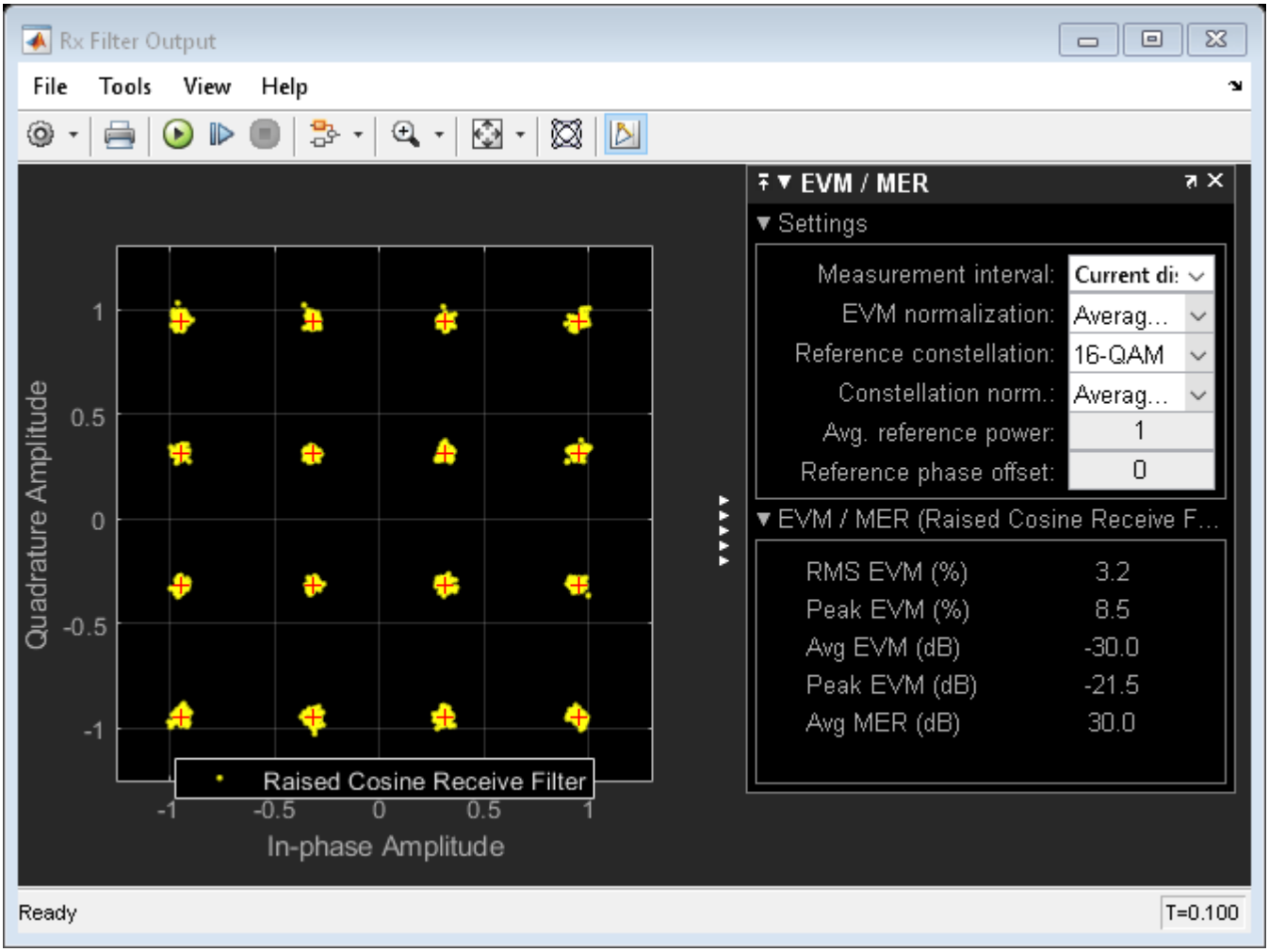
The model testbench includes:

- Power meters before and after the transmitter PA
- Power spectrum scopes before and after the ADC, to illustrate the spectral effects of nonlinear amplification, noise addition, phase noise, and quantization
- A constellation diagram after the receive filter, with error vector magnitude (EVM) calculation turned on
- Resettable BER calculation

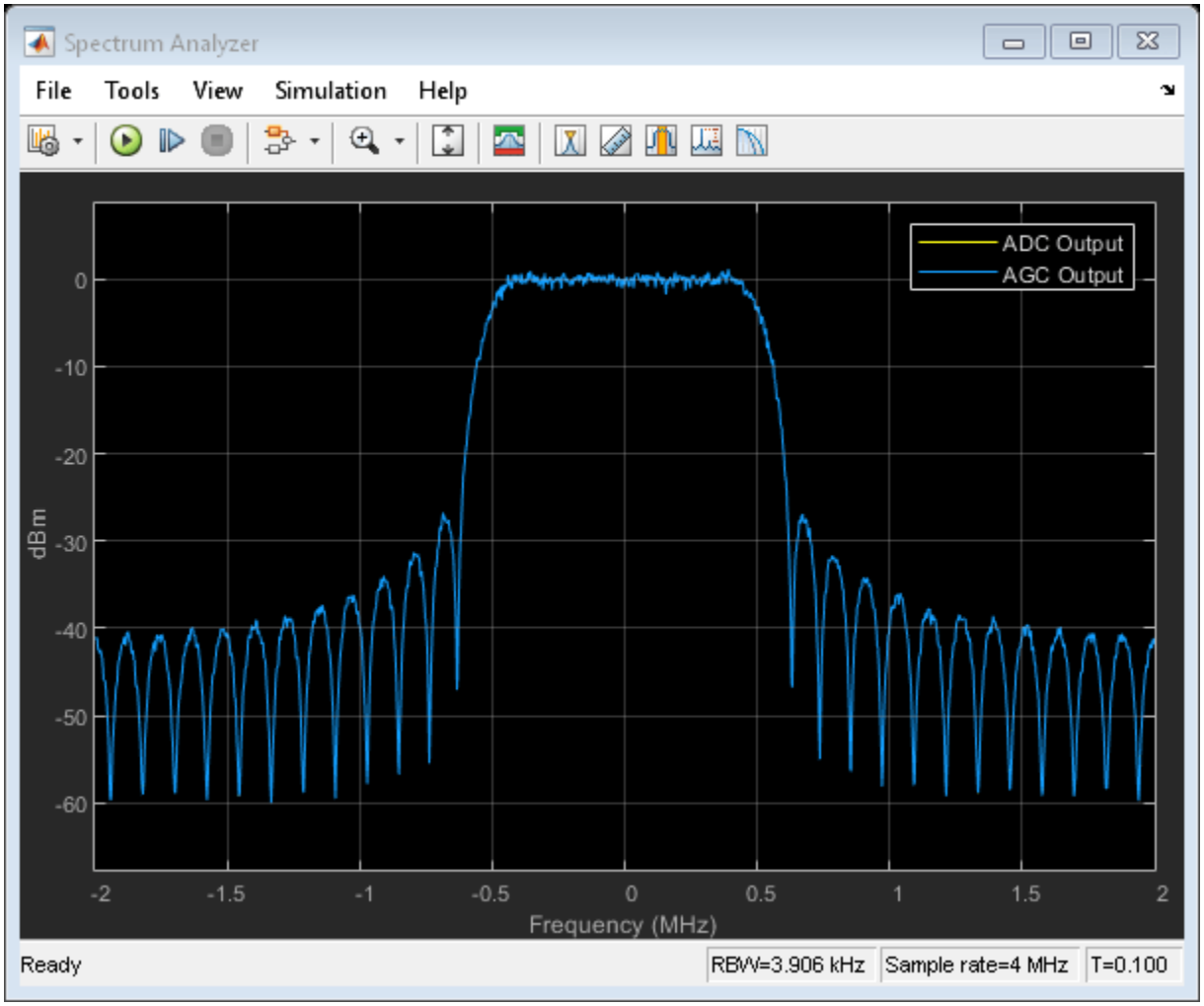
The model sets some parameter values by creating base workspace variables in its preload function. It sets additional values by creating additional base workspace variables through the initialization of the `Model Parameters` block.

Run the Simulation

The default model configuration has nonzero EVM and shows distortion of the signal in the constellation diagram below, due to the finite lengths of the transmit and receive FIR filters.



In this same default configuration, the received power spectrum below is noiseless and has no nonlinear distortions. The sidelobes of the spectrum are from the transmit and receive filter responses.



The `Error Rate Calculation` (ERC) block computes the system BER. In the default configuration, with the ERC block discarding transient effects at the beginning of the simulation, the BER is 0.

Exploring the Example

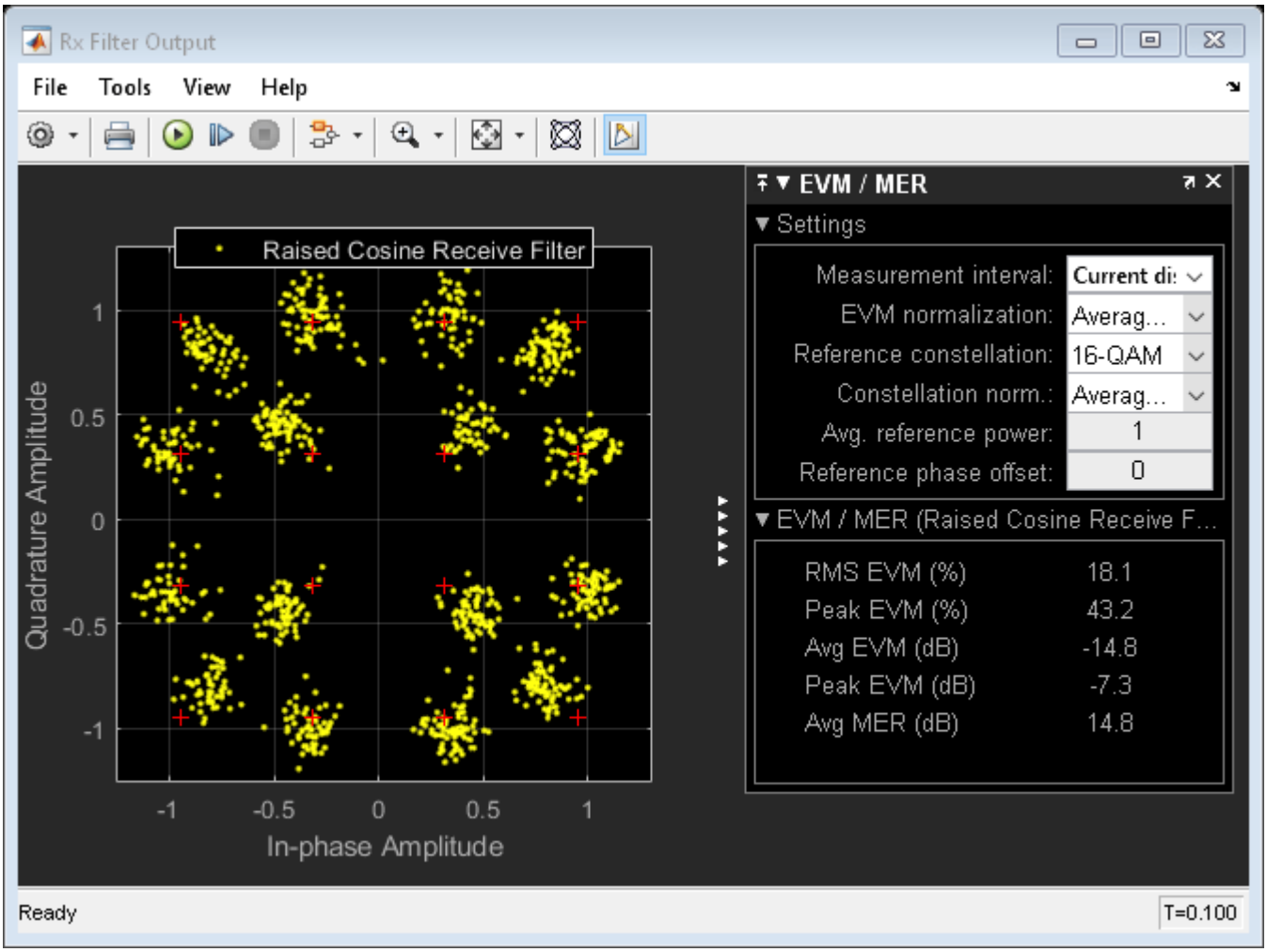
You can investigate multiple RF effects by using the `Model Parameters` block. By default the `Model Parameters` block mask default settings applies distortionless values for transmitter IIP3, LNA noise figure, RF Demodulator phase noise, and the ADC number of bits. Typical degraded value levels are shown after the '%' for each of these parameters in the block mask. If you run the simulation with any one of these degraded values set, you will see effects in the constellation, spectrum, and/or BER.

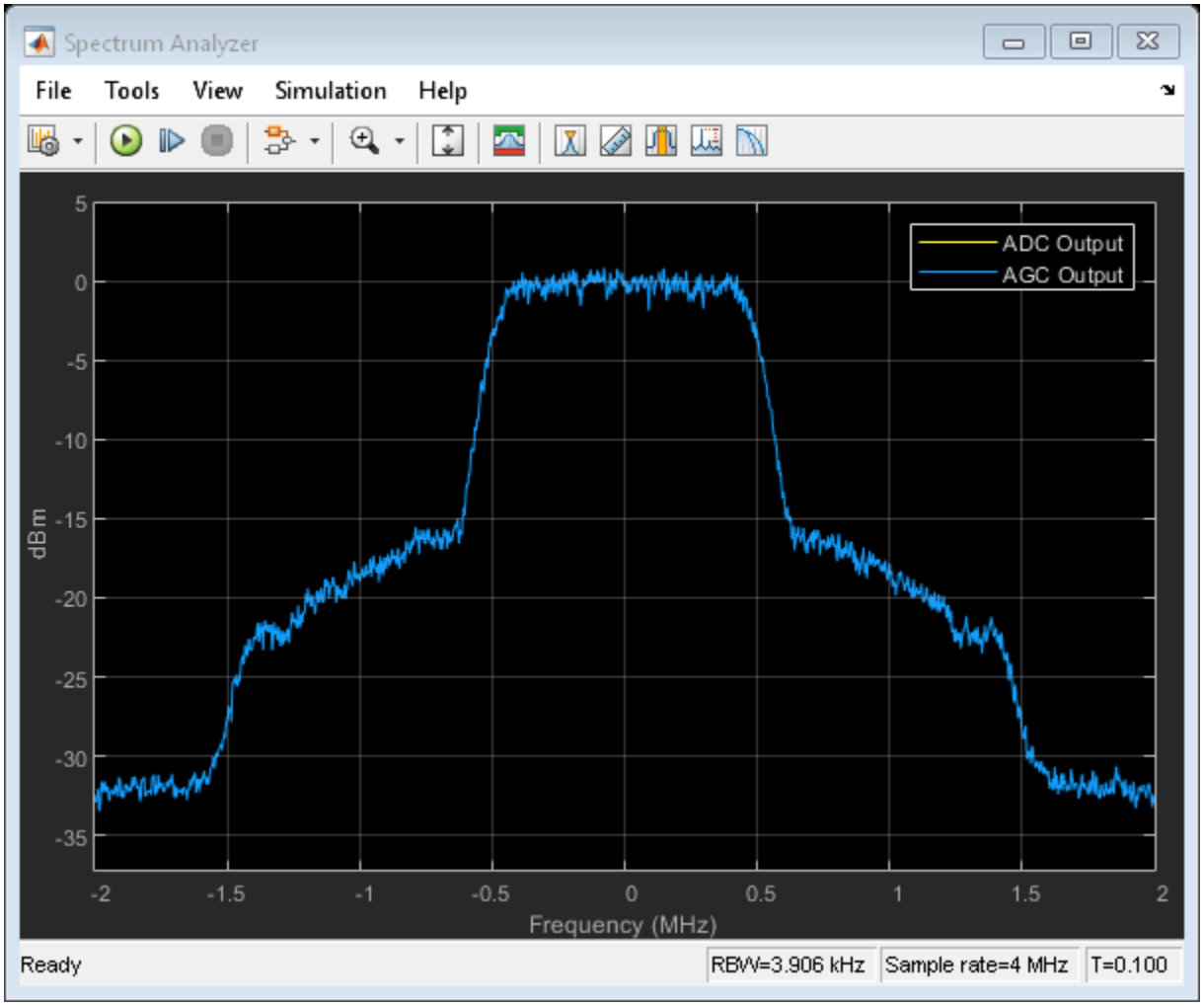
You can reset the following parameters in the `Model Parameters` block while the simulation is running:

- Transmitter IIP3
- LNA noise figure
- ADC number of bits
- ADC full scale voltage

To specify new phase noise values, stop the model first.

For example, if the transmitter IIP3 is set to 15 dBm, the signal spectrum and constellation diagram show a degraded signal, and the BER degrades to approximately 2.8e-3.





You can reset the BER counter while the simulation is running, by double-clicking on the manual switch twice. This is useful to examine the BER effect when you change a parameter value during simulation.

Summary

This example showed how various RF front end impairments, such as amplifier nonlinearities and phase noise, can impact the spectrum, EVM, and BER of a communications system.

See Also

Blocks

AGC Block | Memoryless Nonlinearity | Phase Noise | Power Meter

Related Examples

- “Impact of Thermal Noise on Communication System Performance” (RF Blockset)
- “Idealized Baseband Amplifier with Nonlinearity and Noise” on page 1-2

Interference Modeling

This example shows interference modeling in a bent pipe satellite communications link using Communications Toolbox™.

Introduction

Signal interference is the addition of unwanted signals to a desired signal and is a common problem in many communications systems. Some examples of interference are:

1. The coexistence of 5G and LTE waveforms in the same or similar frequency bands results in one waveform interfering with another waveform
2. Signals from a secondary base station interfering with the signal from the primary base station at a mobile device
3. Downlink adjacent satellite interference occurs when the ground receiving antenna receives significant signal levels from beams of adjacent satellites
4. Interference occurs when a satellite receives and re-broadcasts a strong uplink signal from secondary ground station

Modeling such interference scenarios allows you to analyze their impact on system performance and to design mitigation strategies.

System Setup

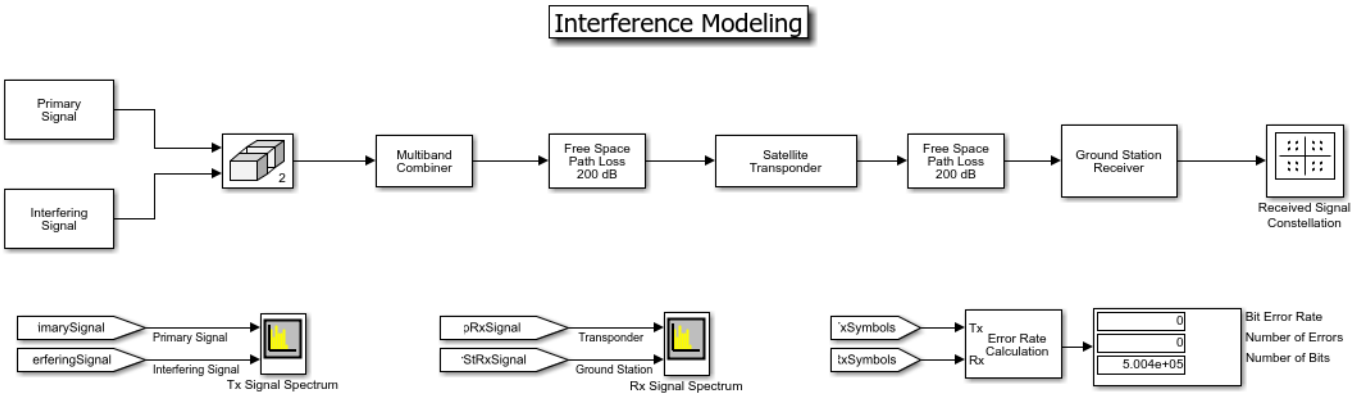
This example models a bent pipe satellite communication link and illustrates how to model an uplink interference scenario. A bent pipe link consists of an uplink from a ground station to a satellite, which acts as a repeater, and downlinks to another ground station without performing any bit-level processing. The satellite transponder receives a primary signal and an interfering signal from a secondary ground station. The combined signal is re-broadcast by the satellite, received and processed at the ground station.

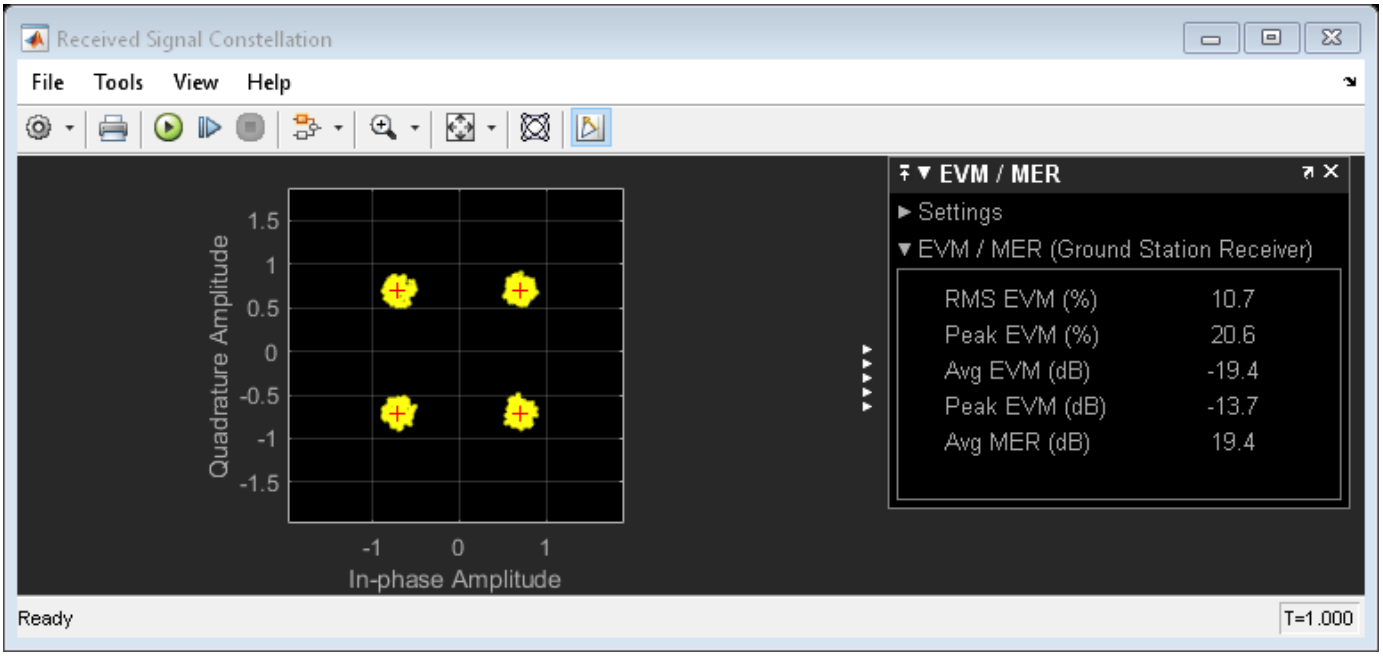
A Multiband Combiner block provides an efficient approach to combine the primary and the interfering signals at baseband. The Multiband Combiner block interpolates the two signals so that the resulting sample rate of the signals guarantees no aliasing when the signals are frequency shifted to model the interference scenario. Then it applies the specified frequency shifts to the signals and combines them into one signal. The block allows modeling of various amounts of spectral overlap to simulate varying severity of interference. For more information, see the Multiband Combiner block reference page.

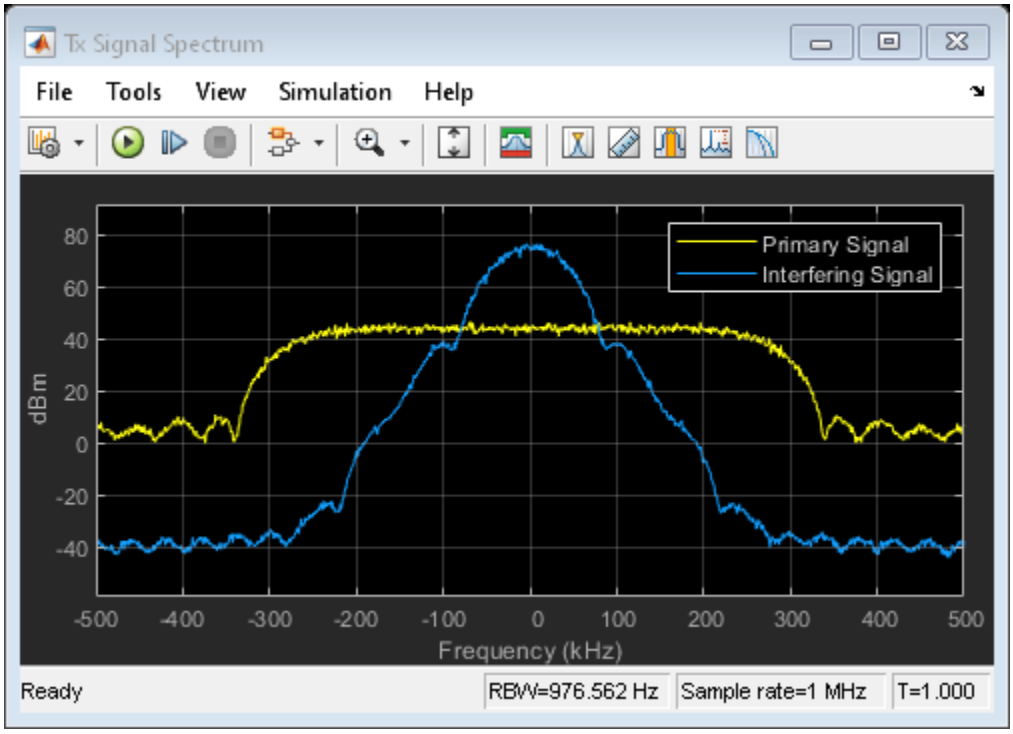
System Simulation

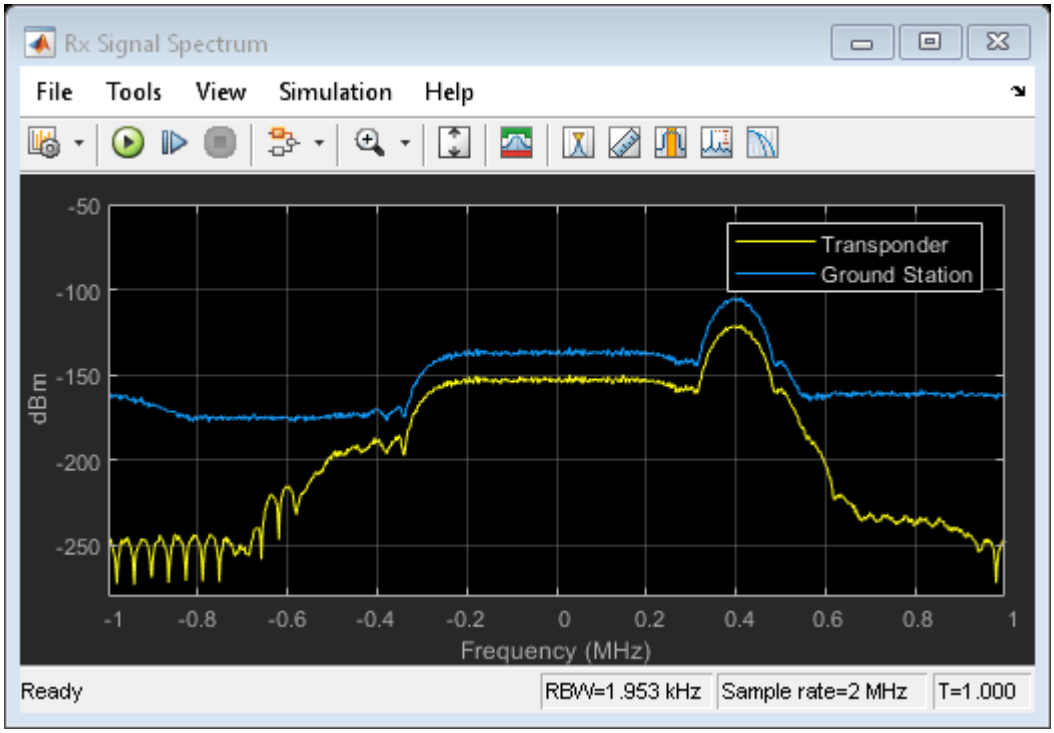
Each of the two baseband signals has a bandwidth of 500 kHz as seen in the Tx Signal Spectrum scope. The **Frequency offsets** parameter of the Multiband Combiner block is set up to model spectral overlap of 100 kHz. This spectral overlap is seen in the Rx Signal Spectrum that shows the spectra of the signals received at the satellite transponder and ground station receiver.

A bit error rate of 0 shows that the system performance is not degraded by this amount of interference. Also, the Received Signal Constellation at the ground station receiver is well clustered around the reference QPSK constellation of the primary signal with a low RMS EVM.

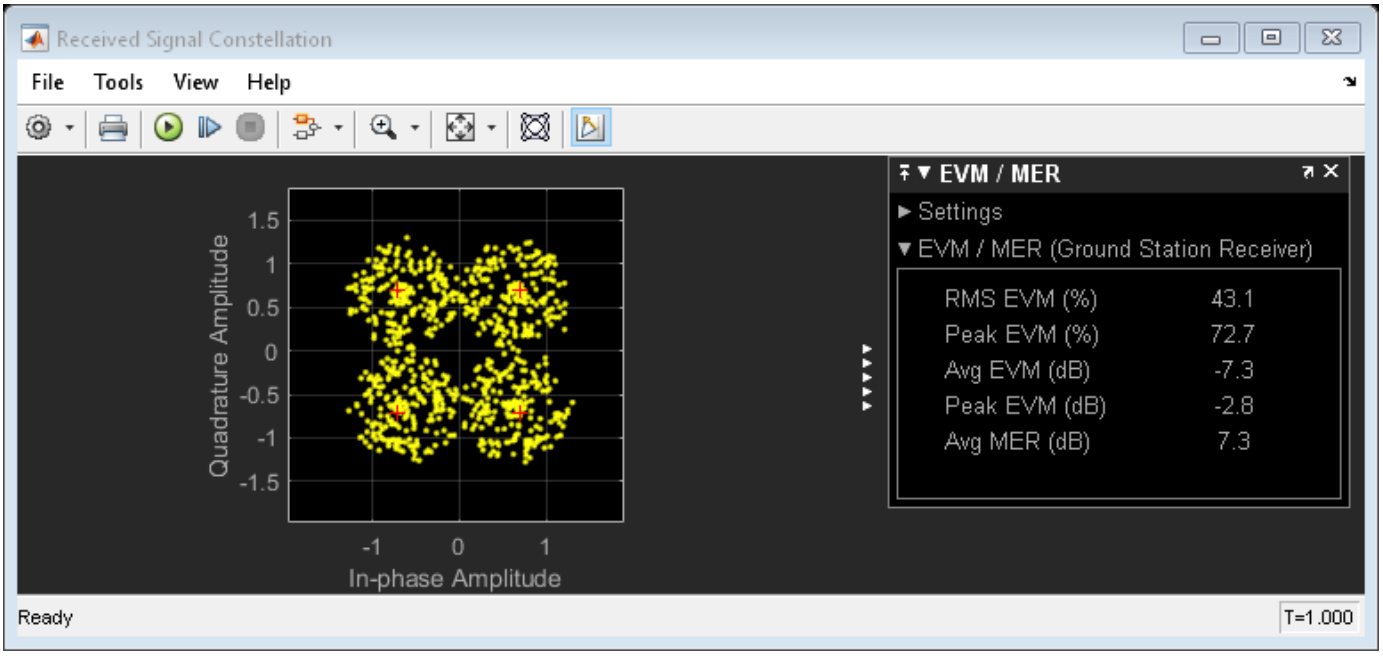


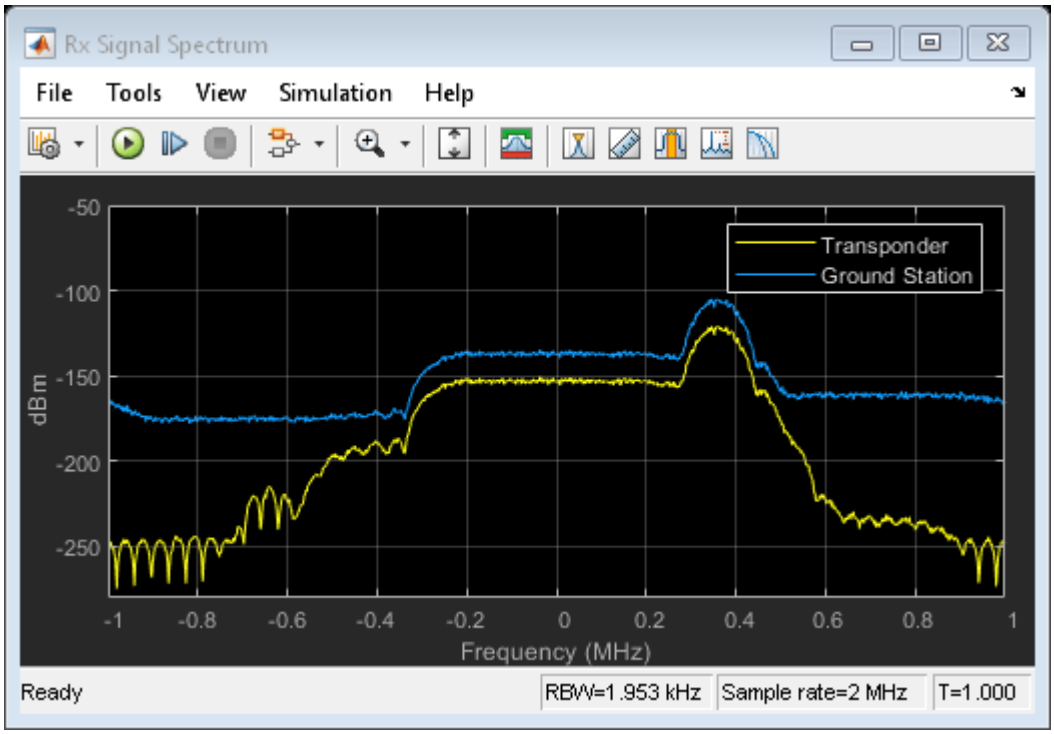






Increase the interference effect by increasing the spectral overlap between the two signals. The increased interference degrades the system performance, as seen from the nonzero bit error rate and a more spread out received signal constellation with higher RMS EVM.





Summary and Further Exploration

This example illustrates a technique to model signal interference that is common in many wireless communications systems. The Multiband Combiner block encompasses the necessary processing of interpolation, frequency shift and signal combining required to simulate various interference scenarios. Other ways to explore interference with this model include:

1. Using baseband signals with different bandwidths
2. Activating and deactivating interference using the switch in 'Interfering Signal' subsystem
3. Modeling more than two baseband signals and more than one interfering signal
4. Modeling various amounts of interference by setting parameters of Signal Aggregator block appropriately
5. Modeling various approaches to minimize the impact of interference at the satellite transponder and ground station receiver

Experiment with the Multiband Combiner block and possibly alter the processing necessary for the particular interference scenario. When the **Output sample rate options** parameter is set to 'Auto', the Multiband Combiner block interpolates the input signals such that the frequency content of the original signals is not distorted after they are frequency shifted. You can also interpolate the baseband input signals to the rate you desire before using the Multiband Combiner block and set the **Output sample rate options** parameter to 'Specify via property', set **Output sample rate** to the same value as 'Input sample rate' which will turn off the builtin interpolation. This example uses two signals, but the block can process any number of input signals once they are concatenated into a matrix.

“Multiband Signal Generation” on page 8-43 example illustrates a `comm.MultibandCombiner` System object™ to perform similar processing as the Multiband Combiner block in MATLAB®.

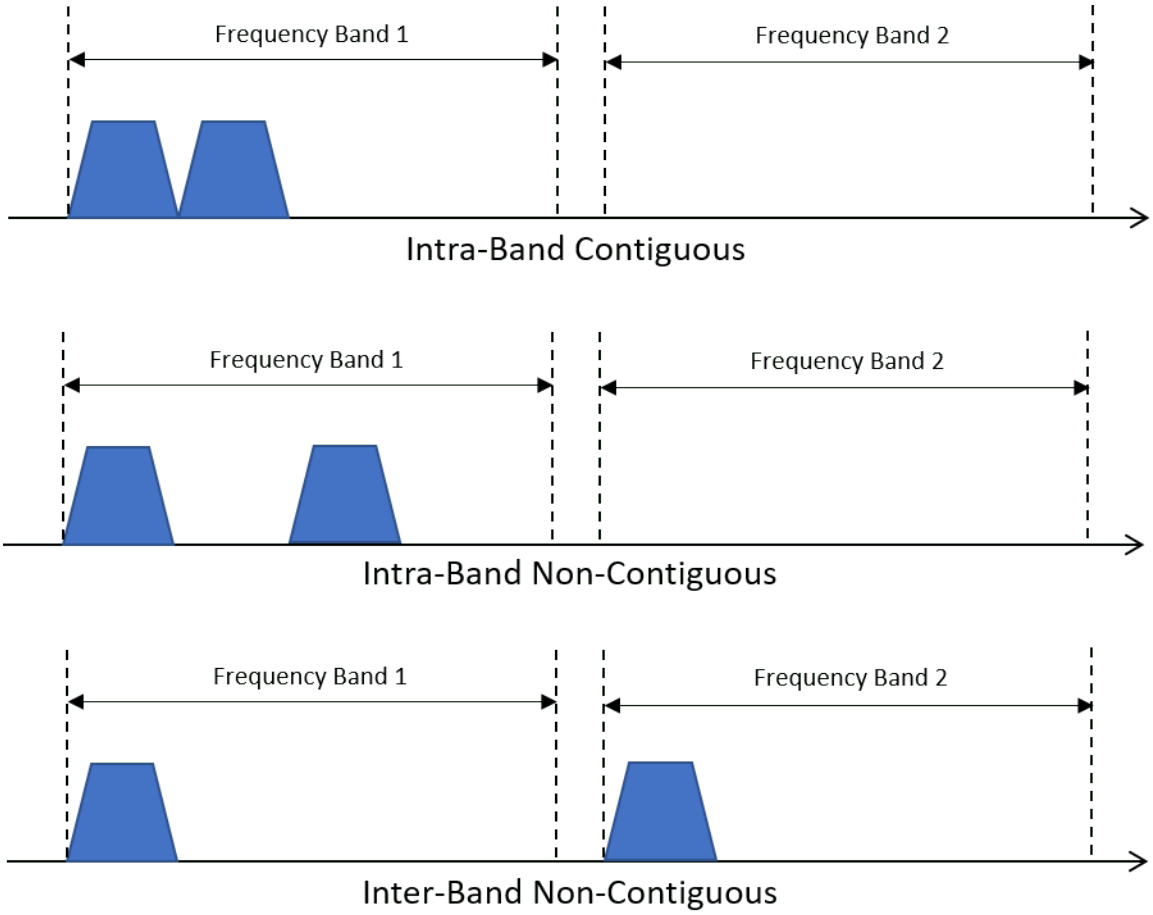
See “Adjacent and Co-Channel Interference” on page 8-179 example to model the effects of adjacent and co-channel interference on a signal.

Multiband Signal Generation

This example shows how to generate a multiband signal efficiently using the Communications Toolbox™.

Introduction

The explosive growth of consumer demand for higher data rates in mobile applications leads to higher transmission rates. Most modern wireless standards include a technique to enhance the data capacity by combining two or more carriers into one data channel. This technique is called carrier aggregation in 5G and LTE terminology, and channel bonding in Wi-Fi® terminology. This figure illustrates three different types of carrier aggregation.



System Setup

This example demonstrates one approach to model carrier aggregation in a baseband simulation. Two baseband signals are generated - one is a QPSK modulated signal and the other is a GMSK modulated signal. Each signal occupies 60 kHz of bandwidth.

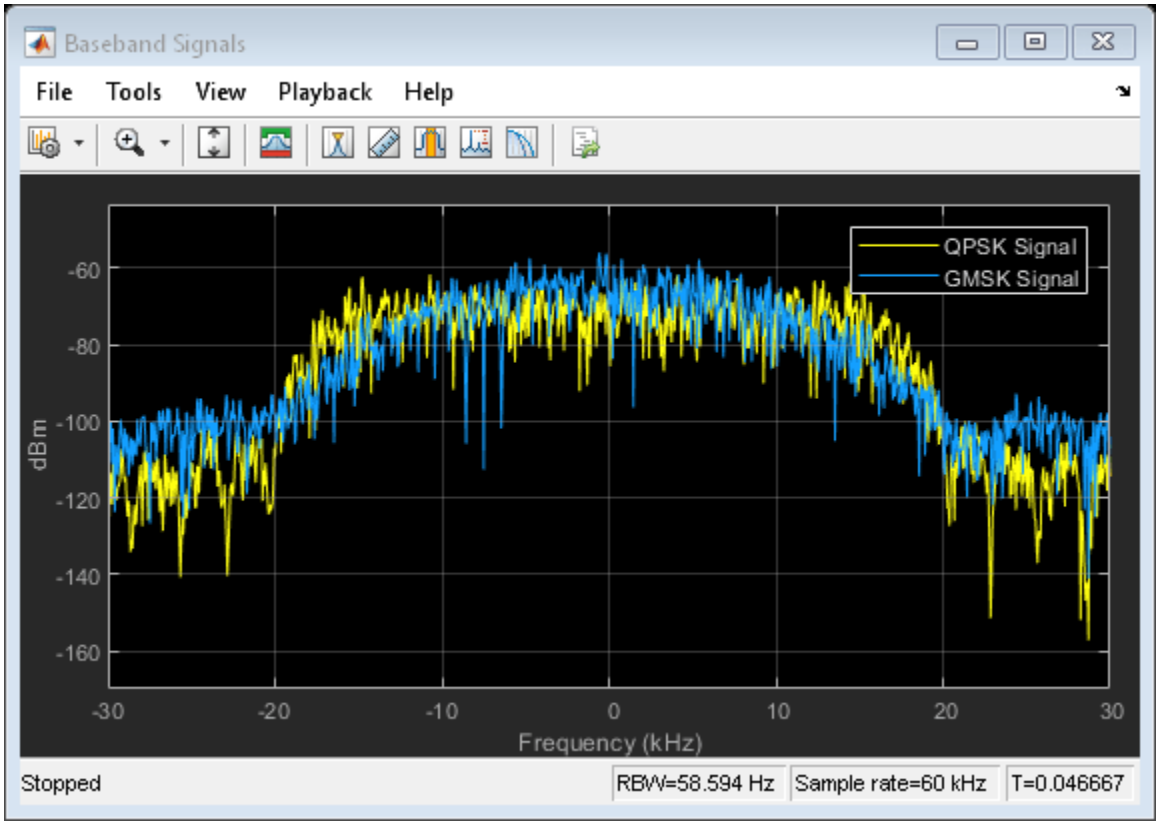
A `MultibandCombiner` System object™ performs the tasks necessary for carrier aggregation. If the sample rate of the input signals is not high enough, the frequency content will be distorted when the original signals are frequency shifted to produce the desired carrier aggregation. Setting the

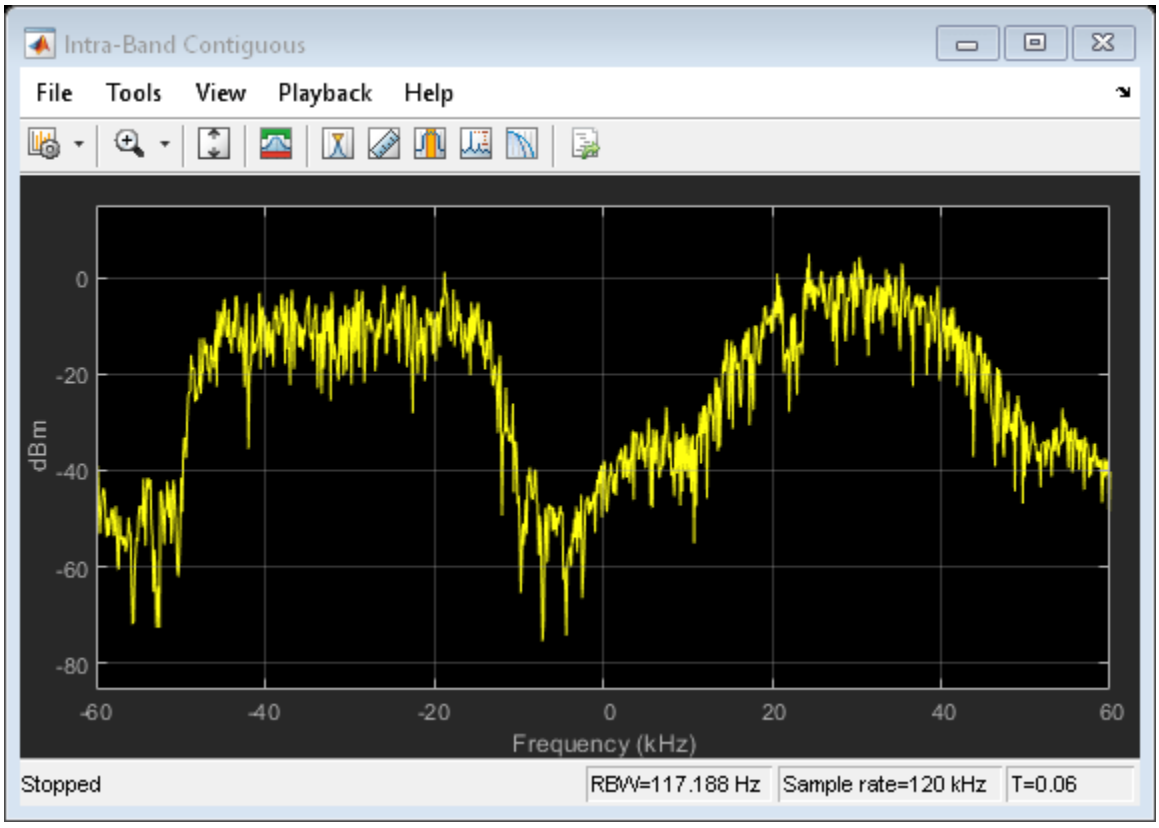
`OutputSampleRateSource` property to `'Auto'` configures the object to automatically compute the output sample rate and interpolate the two signals if necessary to ensure that the resulting signal sample rate is high enough to avoid aliasing. The `info` method of the System object shows the sample rate of the output signal. After the interpolation, the object applies the specified frequency shifts to the signals and combines them into one signal. For more information about the algorithm processing, see the `comm.MultibandCombiner` reference page.

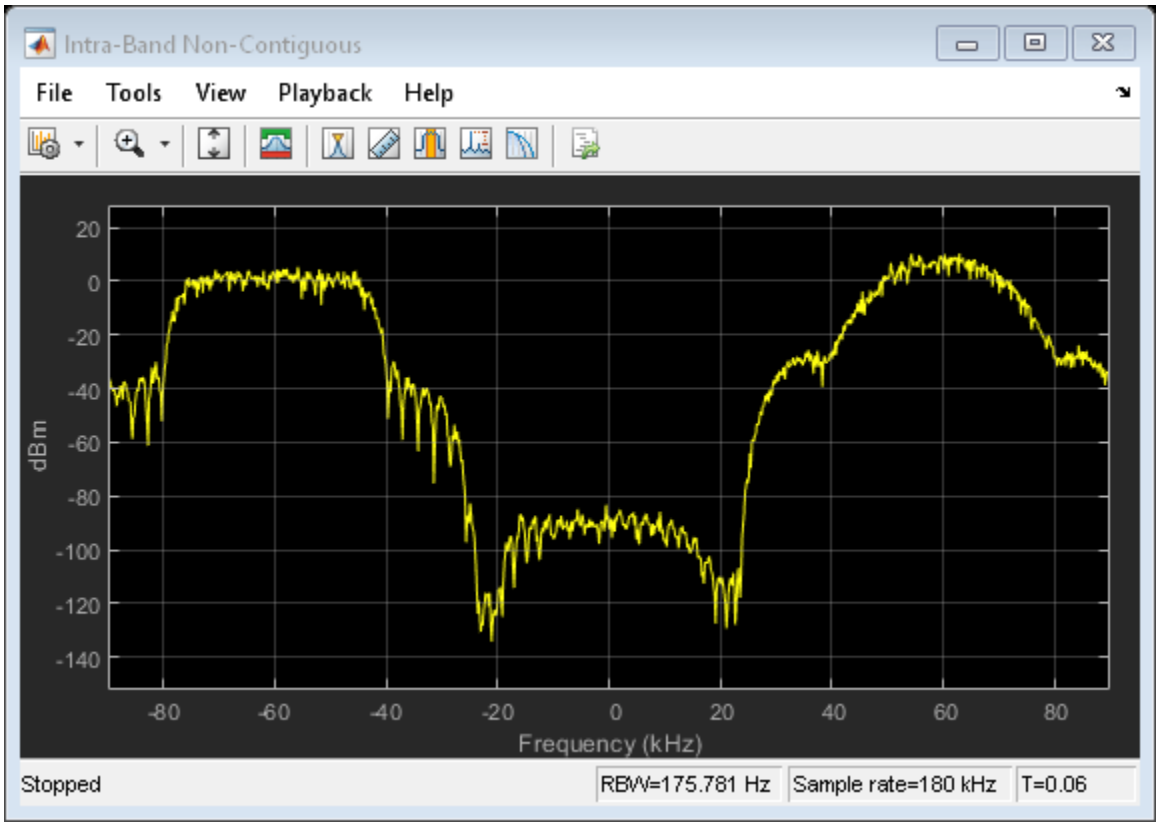
System Simulation

```
nFrames = 10; % Number of data frames
M = 4;        % Modulation order (QPSK modulation)
Fs1 = 60e3;   % Input sample rate

qpskTxFilter = comm.RaisedCosineTransmitFilter(RolloffFactor = 0.3, ...
    OutputSamplesPerSymbol = 2);

gmskMod = comm.GMSKModulator(BitInput = true, SamplesPerSymbol = 2);

% Create multiband combiner object with specified frequency offsets for the
% intra-band contiguous aggregation
sigCombinerCB = comm.MultibandCombiner(InputSampleRate = Fs1, ...
    FrequencyOffsets = [-30e3, 30e3], OutputSampleRateSource = 'Auto');
Fs2 = info(sigCombinerCB).OutputSampleRate;

% Create multiband combiner object with specified frequency offsets for the
% intra-band noncontiguous aggregation
sigCombinerNCB = comm.MultibandCombiner(InputSampleRate = Fs1, ...
    FrequencyOffsets = [-60e3, 60e3], OutputSampleRateSource = 'Auto');
Fs3 = info(sigCombinerNCB).OutputSampleRate;

scopeSF = 0.7; % Scale factor for scope position
spectrumBB = dsp.SpectrumAnalyzer(Name = 'Baseband Signals', ...
    NumInputPorts = 2, SampleRate = 60e3, ...
    Method = 'Filter bank', AveragingMethod = 'Exponential', ...
    ShowLegend = true, ChannelNames = {'QPSK Signal', 'GMSK Signal'});
spectrumBB.Position = scopeSF * spectrumBB.Position;
spectrumBB.Position(1) = spectrumBB.Position(1) - ...
    spectrumBB.Position(3);

spectrumCB = dsp.SpectrumAnalyzer(Name = 'Intra-Band Contiguous', ...
    NumInputPorts = 1, SampleRate = Fs2, ...
    Method = 'Filter bank', AveragingMethod = 'Exponential');
spectrumCB.Position = scopeSF * spectrumCB.Position;

spectrumNCB = dsp.SpectrumAnalyzer(Name = 'Intra-Band Non-Contiguous', ...
    NumInputPorts = 1, SampleRate = Fs3, ...
    Method = 'Filter bank', AveragingMethod = 'Exponential');
spectrumNCB.Position = scopeSF * spectrumNCB.Position;
spectrumNCB.Position(1) = spectrumNCB.Position(1) + ...
    spectrumNCB.Position(3);

for k = 1:nFrames

    % Generate QPSK signal
    data = randi([0, M-1], 200, 1);
    modSig = pskmod(data, M, pi/4, 'gray');
    qpskSignal = qpskTxFilter(modSig);
```

```
    % Generate GMSK signal
    data = randi([0, 1], 200, 1);
    gmskSignal = gmskMod(data);

    % Visualize the two signals
    spectrumBB(qpskSignal, gmskSignal)

    % Upsample, frequency shift and combine the two signals to model
    % intra-band contiguous carrier aggregation
    combinedSignal = sigCombinerCB([qpskSignal, gmskSignal]);

    % Visualize the resulting signal
    spectrumCB(combinedSignal)

    % Upsample, frequency shift and combine the two signals to model
    % intra-band non contiguous or inter-band non contiguous carrier
    % aggregation
    combinedSignal = sigCombinerNCB([qpskSignal, gmskSignal]);

    % Visualize the resulting signal
    spectrumNCB(combinedSignal)
end

release(spectrumBB)
release(spectrumCB)
release(spectrumNCB)
```




Intra-Band Contiguous
File
Tools
View
Playback
Help
0
-20
-40
-60
-80
dBm
-60
-40
-20
0
20
40
60
Frequency (kHz)
Stopped
RBW=117.188 Hz
Sample rate=120 kHz
T=0.06



Visualization

Intra-band contiguous aggregation results in a signal that has two original signals, each 60 kHz wide, occupying two contiguous bands of 60 kHz each. In intra-band non-contiguous aggregation, the two signals occupy non-contiguous bands as shown by the gap between the signal spectra in the Intra-Band Non-Contiguous Spectrum Analyzer. Inter-band non-contiguous aggregation can be similarly achieved by appropriate frequency shifts of the signals.

Summary and Further Exploration

This example illustrates a technique to model the carrier aggregation that is used by most modern wireless communications standards to increase data rates. A System object is used to encapsulate the necessary processing of interpolation, frequency shift and signal combining. You can explore further in various ways:

1. Use baseband signals with different bandwidths. As `MultibandCombiner` System object requires all input signals to have the same sample rate, resample one or more signals to bring all baseband signals to the same rate before using `MultibandCombiner` System object.
2. Aggregate more than two baseband signals,
3. Use different aggregation bands and carriers to model inter-band non-contiguous aggregation.

Also, explore the `MultibandCombiner` System object to study and possibly alter the processing necessary for carrier aggregation. Besides configuring the object to automatically compute the output sample rate by setting the `OutputSampleRateSource` to `'Auto'`, you can also interpolate the baseband input signals to the rate you desire before using the `MultibandCombiner` object, then set

the `OutputSampleRateSource` to `'Property'` and set the `'OutputSampleRate'` equal to `'InputSampleRate'` which configures the System object to not perform any interpolation.

Ship Tracking Using AIS Signals

This example shows you how to track ships by processing Automatic Identification System (AIS) signals using MATLAB® and Communications Toolbox™. You can either use captured signals or receive signals in real time using the RTL-SDR Radio. The example can show the tracked ships on a map, if you have the Mapping Toolbox™.

Required Hardware and Software

To run this example using captured signals, you need the Communications Toolbox™.

To receive signals in real time, you also need an RTL-SDR radio and the corresponding Communications Toolbox Support Package for RTL-SDR Radio support package Add-On.

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

Background

In marine transportation, vessel traffic services use AIS as a component of the overall marine traffic monitoring system. AIS performs the following functions:

- Transmit vessel identifier, position, course, and speed.
- Receive and process specified interrogating calls.
- Operate continuously while under way or at anchor.

Specifications of AIS:

- Transmit Frequency Range: 156.025 MHz-162.025 MHz
- Modulation Scheme: Gaussian frequency shift keying
- Bit Rate: 9600 bits/sec
- Transmit Bandwidth Time Product: 0.4
- Receive Bandwidth Time Product: 0.5
- Modulation Index: 0.5

AIS transmission packets contain these fields:

- Training Sequence: 24-bit sequence of alternating zeros and ones (0101...).
- Start Flag: 8-bit sequence, 01111110.
- Data: The data portion is 168 bits long in the default transmission packet.
- Frame Check Sequence(FCS): Uses the cyclic redundancy check (CRC) 16-bit polynomial to calculate the checksum.
- End Flag: Identical to the start flag.
- Buffer: The buffer is normally 24 bits long to account for bit stuffing (maximum 4 bits), distance delay (14 bits) and synchronization jitter (6 bits).

This figure shows the AIS packet format

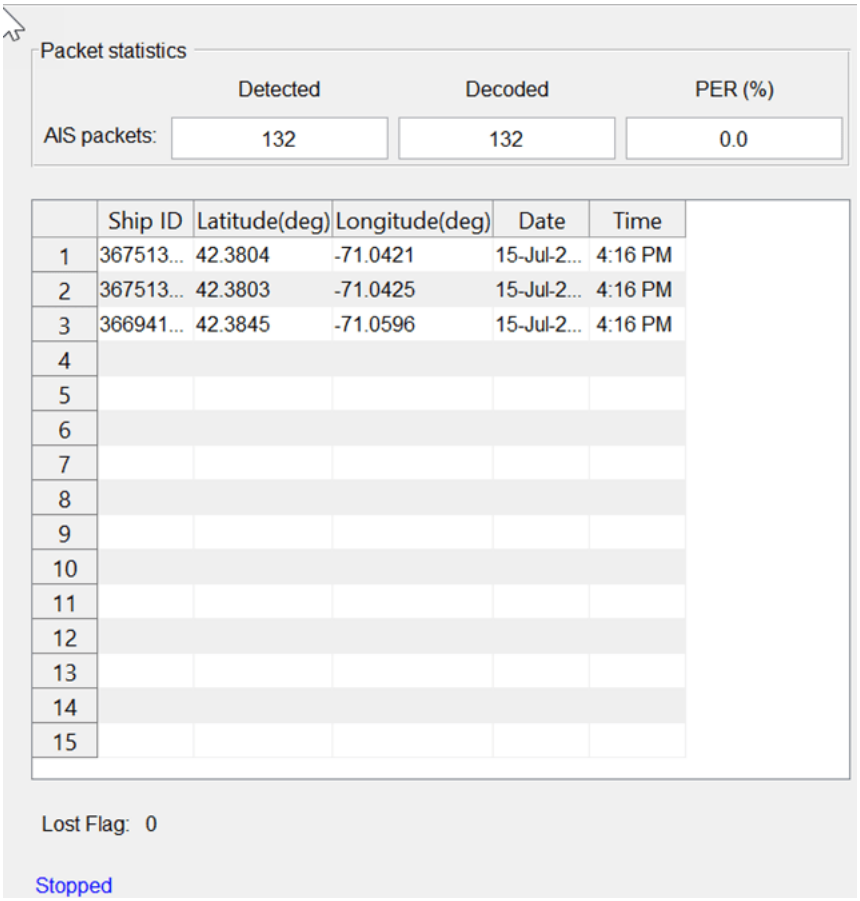
Training Sequence	Start Flag	Data	FCS	End Flag	Buffer

Run the Example

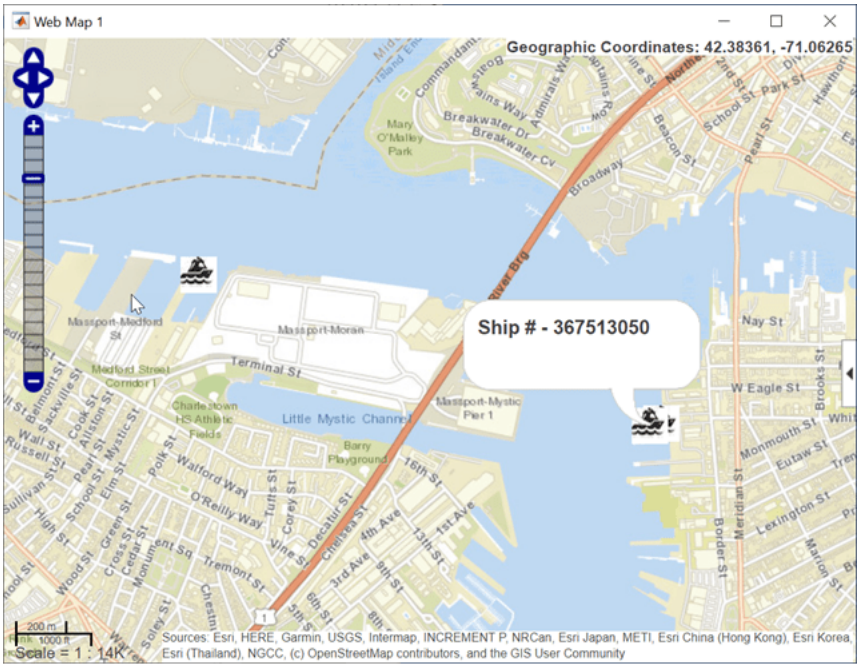
You can open the example by selecting the **Open script** button. The default configuration runs for a duration of 10 seconds, uses signal data from a captured data file, and outputs to a text file. To provide input values from the command line, you must change `cmdlineInput` to 1, then you will be prompted to enter the following information when you run the example:

1. Reception duration in seconds,
2. Signal source (Captured data file or RTL-SDR radio),
3. Optional output methods (map, text file, or both).

The example shows the information on the detected ships in a tabular form as shown in the following figure.

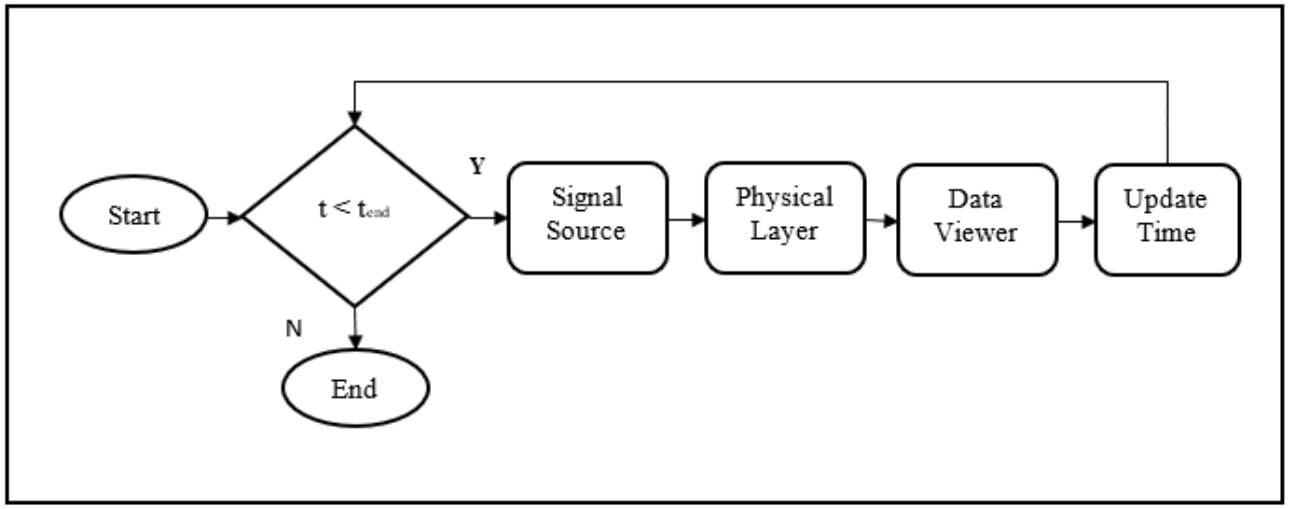


If you have the Mapping Toolbox licensed, you can also observe AIS tracking of ships on a map.



Receiver Structure

The following block diagram summarizes the receiver code structure. The processing has three main parts: Signal Source, Physical Layer and Data Viewer.



Signal Source

Specify the signal source as "File" or "RTL-SDR".

1 ''File'': Uses the `comm.BasebandFileReader` to read a file that contains a previously signal captured over-the-air.

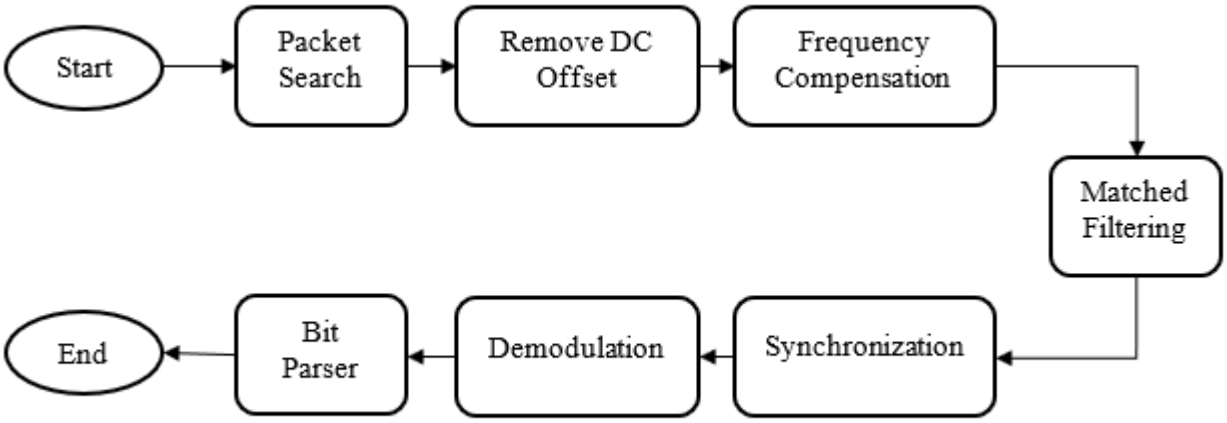
2 ''RTL-SDR'': Uses the RTL-SDR radio to receive a live signal.

The code uses a signal symbol rate of 9600 Hz and 24 samples per symbol.

If you assign ''RTL-SDR'' as the signal source, the example searches your computer for the RTL-SDR radio at radio address '0' and uses it as the signal source.

Physical Layer

The baseband samples received from the signal source are processed by the physical layer (PHY) to produce packets that contain the ship position information and raw message bytes. This figure shows the PHY processing components.



- **Packet Search**: Searches for the strongest burst in the received signal by dividing into multiple windows.
- **DC Offset Removal**: Removes DC offset from the detected signal.
- **Frequency Compensation**: Estimate and compensates for the carrier frequency offset.
- **Matched Filtering**: Performs filtering with Gaussian pulse generated as per AIS specifications.
- **Synchronization and Demodulation**: Performs timing synchronization by correlating the received signal with known preamble and demodulates to produce bits.
- **AIS Bit Parser**: Detects the Start Flag and End Flags, then performs CRC detection. If CRC is successful, then the ship information is decoded.

There are 64 specific message types in AIS. Ship position information is included in 11 of the message types. This example decodes all 11 of the message types that contain position information.

As seen in the earlier figure, ship ID, latitude, longitude, date, and time are displayed by this example. Messages contain additional information that can be decoded as described in [1].

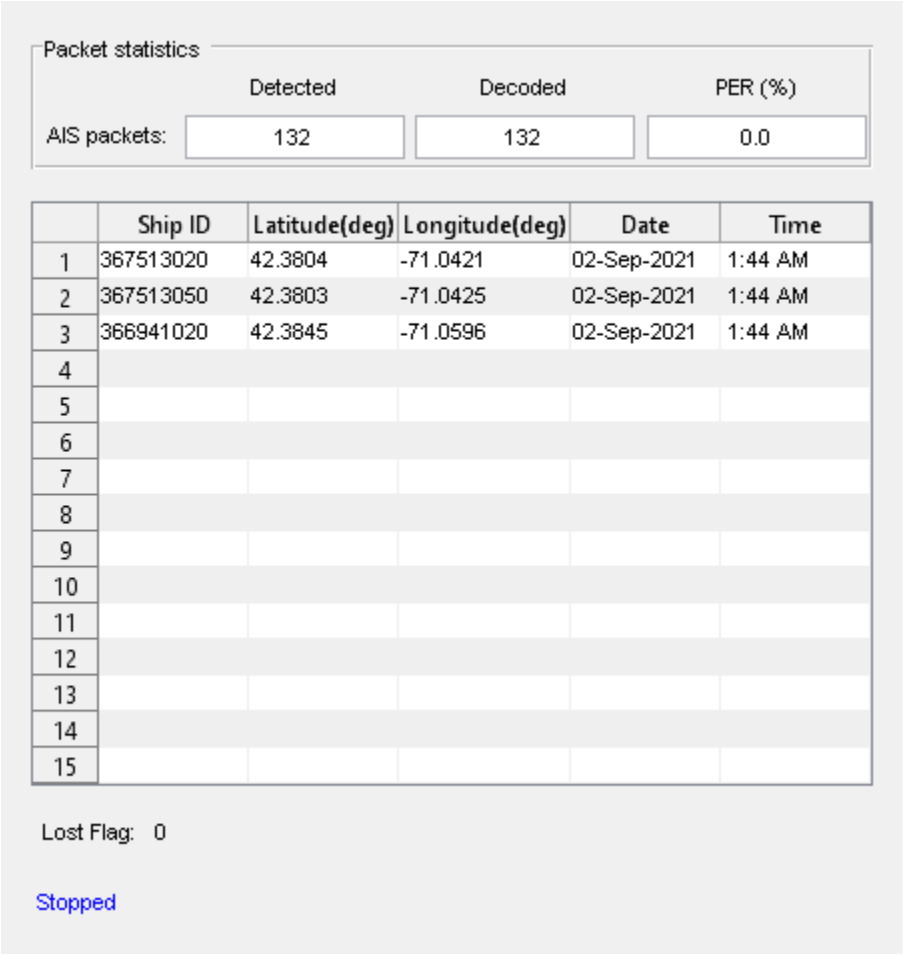
Data Viewer

The data viewer shows the received messages on a graphical user interface (GUI). As data is captured, the application lists information decoded from these messages in a tabular form.

Example Code

The example steps are described below. To see the detailed operations look at the code run in the helper functions called by the example. For the option to change default settings, set `cmdlineInput` to 1.

```
cmdlineInput = 0;
if cmdlineInput
    % Request user input from the command-line for application parameters
    userInput = helperAISUserInput;
else
    load('defaultInputs.mat');
end
% Calculate AIS parameters based on the user input
[aisParam,sigSrc] = helperAISConfig(userInput);

% Create the data viewer object and configure based on user input
viewer = helperAISViewer('LogFileName',userInput.LogFilename, ...
                         'SignalSourceType',userInput.SignalSourceType);

% Launch map based on user input
if userInput.LaunchMap
    startMapUpdate(viewer);
end

% Log data based on user input
if userInput.LogData
    startDataLog(viewer);
end

% Start the viewer and initialize radio time
start(viewer)
radioTime = 0;

% Main loop for capturing and decoding the AIS samples
while radioTime < userInput.Duration
    if aisParam.isSourceRadio       % For RTL-SDR
        [rcv,~,lost,~] = sigSrc();
        lostFlag = logical(lost);
    else                            % For baseband file
        rcv = sigSrc();
        lostFlag = uint32(0);
    end

    % Recover the information by decoding AIS samples
    [info, pkt] = helperAISRxPhy(rcv,aisParam);

    % View decoded information on viewer
    update(viewer, info, pkt, lostFlag);

    % Update radio time
    radioTime = radioTime + aisParam.FrameDuration;
end

% Stop the viewer and release the signal source
stop(viewer)
release(sigSrc)
```



Further Exploration

You can also type AISExampleApp in the MATLAB Command Window or click the link to use the AISExampleApp user interface to explore AIS signals. The app interface allows you to select the signal source and change the duration.

You can explore following functions and System objects for details of the physical layer implementation:

- helperAISRxPhy.m
- helperAISRxPhyPacketSearch.m
- helperAISRxPhyFreqComp.m
- helperAISRxPhySyncDemod.m
- helperAISRxPhyBitParser.m

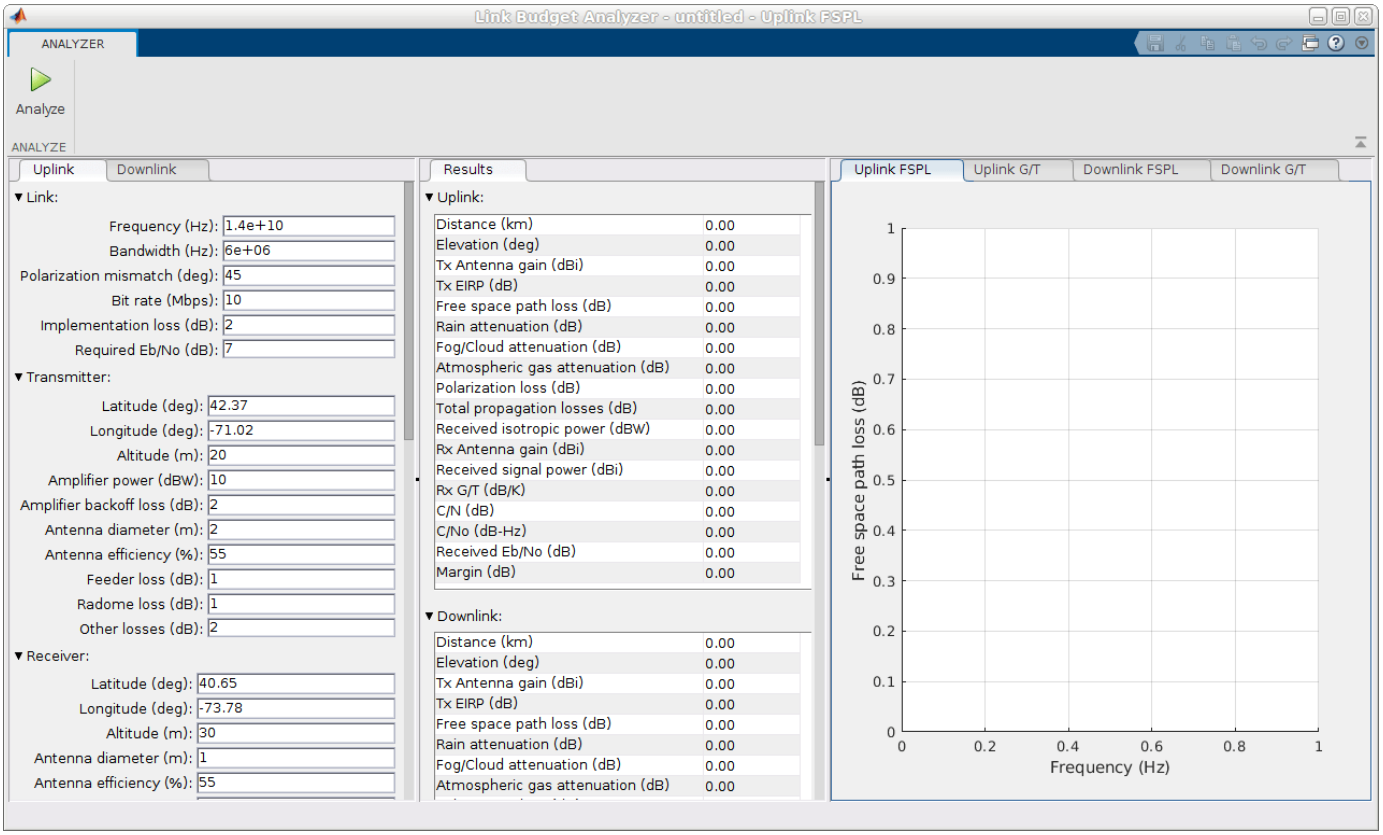
Selected Bibliography

1 Recommendation ITU-R M.1371-5, Technical characteristics for an automatic identification system using time division multiple access in the VHF maritime mobile frequency band.

Link Budget Analysis

In the design of wireless communications links between two sites, issues of range, throughput, and received signal quality are of critical importance to the system engineer. Link budget analysis accounts for all gains and losses in the communication link. Some factors and design choices, such as propagation path length, signal polarization, and antenna feed cable, degrade signal quality, while others, such as the power amplifier and antenna size, can increase transmitted signal strength.

This example uses linkBudgetAnalyzer app to tabulate system parameters and compute gains and losses that impact system performance. Separate tabs specify settings for **Uplink** and **Downlink**. After specifying the uplink and downlink settings, select **Analyze** to update the gains and losses reported in the **Results** tab and the tabs with plots of free space path loss (FSPL) and G/T for uplink and downlink.



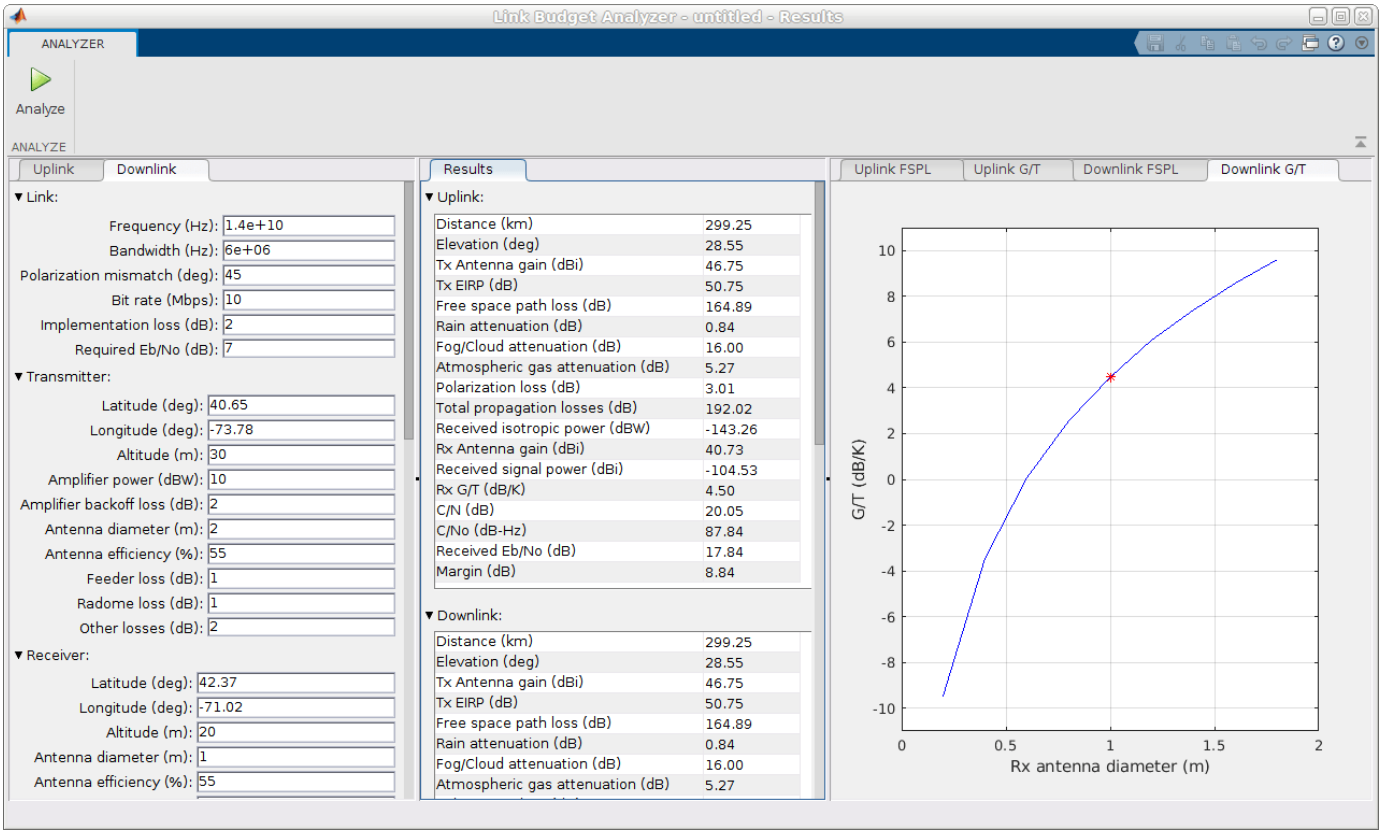
Settings

The **Uplink** and **Downlink** tabs contain these collapsible input parameter sections:

- **Link** - Contains link level parameters, such as frequency, bandwidth, required Eb/N0
- **Transmitter** - Contains transmitter specific parameters
- **Receiver** - Contains the receiver specific parameters
- **Propagation** - Contains parameters to specify various atmospheric elements that are used to compute losses in the signal propagation path.

Results

The **Results** tab contains **Uplink** and **Downlink** collapsible sections that provide the link budget results for uplink and downlink, respectively. The **Appendix** contains a list of functions used to compute the results.



- **Distance** and **Elevation** are computed from the **Latitude**, **Longitude** and **Altitude** input parameters for the transmitter and receiver.
- **Tx Antenna gain** and **Rx Antenna gain** are functions of the corresponding **Antenna diameter**, **Antenna efficiency**, and **Frequency** input parameters.
- **Tx EIRP** is a function of the **Amplifier power**, **Amplifier backoff loss**, **Feeder loss**, **Radome loss**, **Other losses**, and **Tx Antenna gain** input parameters. **Tx EIRP**, which represents transmitted equivalent isotropically radiated power (EIRP), is the amount of power that would have to be radiated by an isotropic antenna to produce the equivalent power density observed from the actual antenna in a specified direction. Typically, EIRP is quoted for antenna boresight, which is defined as the axis of maximum radiation.
- The transmitted signal power is diminished by geometric spreading of the wavefront. This loss is represented by **Free space path loss** which is computed using the `fspl` function, **Distance**, and **Frequency**.
- **Rain attenuation** is computed by the `rainpl` function using **Distance**, **Frequency**, **Rain rate**, **Elevation** and **Polarization tilt**. The `rainpl` function applies the International Telecommunication Union (ITU) rainfall attenuation model which applies only for frequencies at 1-1000 GHz [1].

- The `fogpl` function computes **Fog/Cloud attenuation** using **Distance**, **Frequency**, **Fog/Cloud temperature** and **Fog/Cloud water density**. The `fogpl` function applies the ITU cloud and fog attenuation model which is valid only for frequencies at 10-1000 GHz [2].
- **Atmospheric gas attenuation** is a function of **Distance**, **Frequency**, **Temperature**, **Atmospheric pressure** and **Water vapor density** and is calculated using the `gaspl` function which applies ITU atmospheric gas attenuation model that is valid for frequencies at 1-1000 GHz [3].
- **Polarization loss** is derived from **Polarization mismatch** angle.
- **Total propagation losses** consists of all the above-mentioned losses.
- **Tx EIRP** is diminished by **Total propagation losses** and receiver **Radome loss** to provide **Received isotropic power** at the receiver.
- At the receiver, antenna amplifies the **Received isotropic power** by **Rx Antenna gain** while **Feeder loss** and **Other losses** degrade the signal. **Received signal power** shows the net result.
- **Rx G/T** provides information on the performance of the receiver and is computed from **Rx Antenna gain** and **System temperature**. The receiver performance improves as G/T increases.
- **C/N** represents SNR (Signal-to-Noise Ratio) and is a function of **Received signal power**, **System temperature**, **Bandwidth** and Boltzmann's constant.
- **C/No** is computed from **C/N** and **Bandwidth**.
- **Received Eb/No** indicates energy per bit and is a function of **C/No** and **Bit rate**.
- **Margin** is computed from **Received Eb/No**, **Required Eb/No**, and **Implementation loss**. One goal when performing a link budget analysis is to have a satisfactory margin for the chosen data rate, bandwidth, EIRP and receiver figure of merit. Often some adjustment is needed to get the desired link margin.

Visualization

For path loss and receiver performance plots, see the uplink and downlink FSPL and G/T tabs. Free space path loss constitutes the largest component of propagation losses. It is proportional to distance and frequency. Receiver figure of merit increases with antenna gain, which is proportional to antenna diameter. The specified **Frequency** and receiver **Antenna diameter** are shown by the red * marker in the plots.

Appendix

Following functions are used to compute the various parameters and losses mentioned in this example:

- comm.internal.linkBudgetApp.computeAntennaGain.m
- comm.internal.linkBudgetApp.computeAtmGasAtt.m
- comm.internal.linkBudgetApp.computeCbyN0.m
- comm.internal.linkBudgetApp.computeCbyN.m
- comm.internal.linkBudgetApp.computeDistance.m
- comm.internal.linkBudgetApp.computeEbN0.m
- comm.internal.linkBudgetApp.computeEIRP.m
- comm.internal.linkBudgetApp.computeFigureOfMerit.m
- comm.internal.linkBudgetApp.computeFogAtt.m

- comm.internal.linkBudgetApp.computeFSPL.m
- comm.internal.linkBudgetApp.computeMargin.m
- comm.internal.linkBudgetApp.computePolarizationLoss.m
- comm.internal.linkBudgetApp.computeRainAtt.m
- comm.internal.linkBudgetApp.computeWavelength.m

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.840-6: Attenuation due to clouds and fog. 2013.

3 Radiocommunication Sector of International Telecommunication Union. Recommendation ITU-R P.676-10: Attenuation by atmospheric gases 2013.

Parallel Concatenated Convolutional Coding: Turbo Codes

This example characterizes the performance of turbo codes over a noisy channel. It shows the basic structure of turbo codes at the transmitter and receiver. We chose the Long Term Evolution (LTE) specifications [4] for the constituent component parameters.

The invention of turbo codes [1], along with the development of iterative decoding principles with near Shannon limit performance, has led to their absorption in a wide variety of applications some of which include deep space communications, third generation wireless standards, and digital video broadcasting [3].

Available Example Implementations

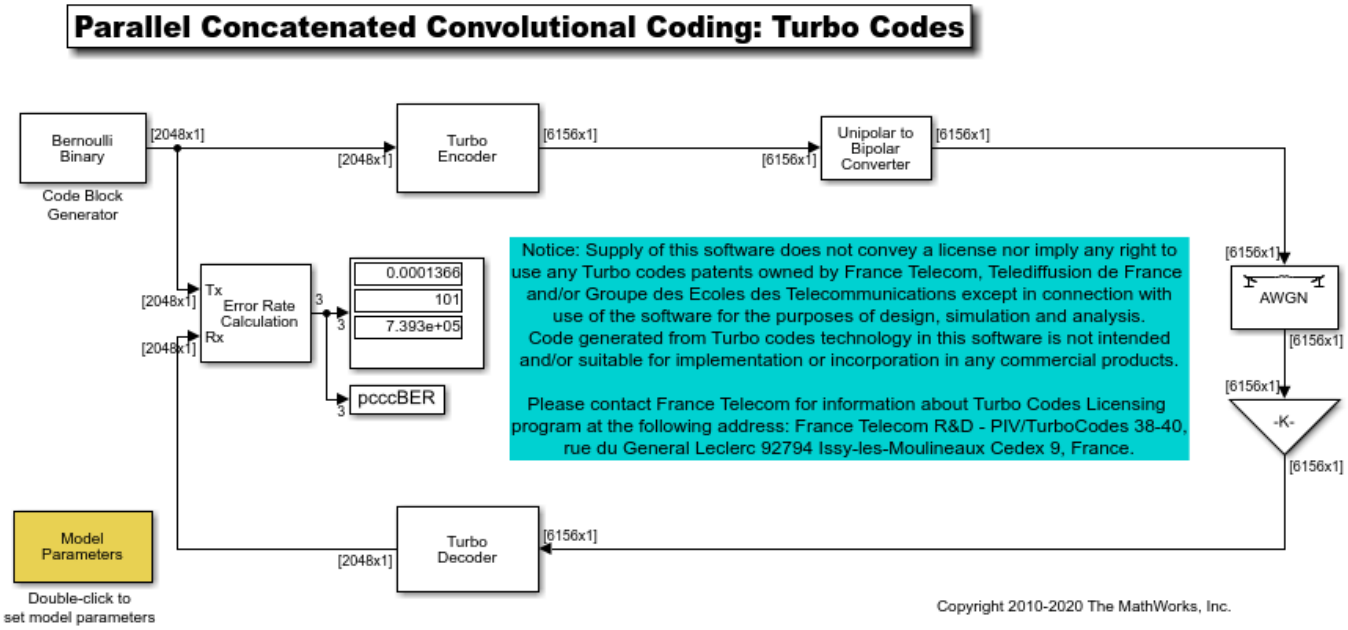
This example includes both MATLAB® and Simulink® implementations:

MATLAB script using System objects: commTurboCoding.m

Simulink model using System blocks: commpccc.slx

Simulink model using variable-sized code-blocks: commpcccvs.slx

Both the MATLAB and Simulink implementations of the system are set up so you can simulate the system over a range of Eb/No values for user-specified system parameters like code block length and number of decoding iterations. The following sections use the fixed-size code-block Simulink implementation to describe the details of the coding scheme.



Turbo Encoder

A `comm.TurboEncoder` is a parallel concatenation scheme with multiple constituent Convolutional encoders. The first encoder operates directly on the input bit sequence, while any others operate on interleaved input sequences, obtained by interleaving the input bits over a block length.

The System block based Turbo Encoder block uses two identical 8-state recursive systematic convolutional encoders. The `comm.ConvolutionalEncoder` System object™ uses the "Terminated" setting for the `TerminationMethod` property. This restores the encoders to the starting all-zeros state for each frame of data the block processes. The internal block interleaver uses pre-computed permutation indices, based on the user-specified **Code block length** parameter (see the `Model Parameters block`). The bit reordering subsystem removes the extra set of systematic bits from the second encoder output and realizes the trellis termination as per [4].

Iterative Decoding

For iterative decoding of the parallel concatenated encoding scheme, the `comm.TurboDecoder` uses the a posteriori probability (APP) decoder [2] as the constituent decoder component.

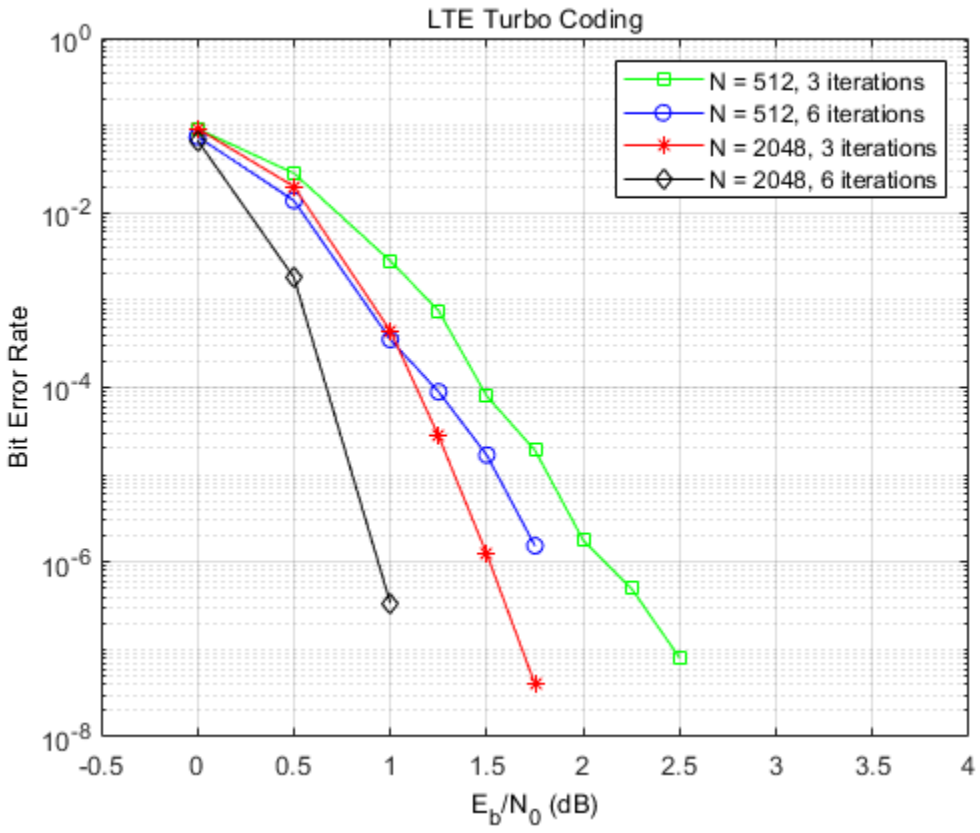
Each `comm.APPDecoder` System object corresponds to a constituent encoder which provides an updated sequence of log-likelihood values for the uncoded bits from the received sequence of log-likelihoods for the channel (coded) bits. For each set of received channel sequences, the decoder iteratively updates the log-likelihoods for the uncoded bits until a stopping criterion is met. This example uses a fixed number of decoding iterations, as specified by the **Number of decoding iterations** parameter in the model's `Model Parameters` block. The default number of iterations is six.

The `TerminationMethod` property for the APP Decoder System object is set to be "Terminated" to match the encoders. The decoder does not assume knowledge of the tail bits and as a result, these are excluded from the multiple iterations.

The internal interleaver of the decoder is identical to the one the encoder uses. It reorders the sequences so that they are properly aligned at the two decoders.

BER Performance

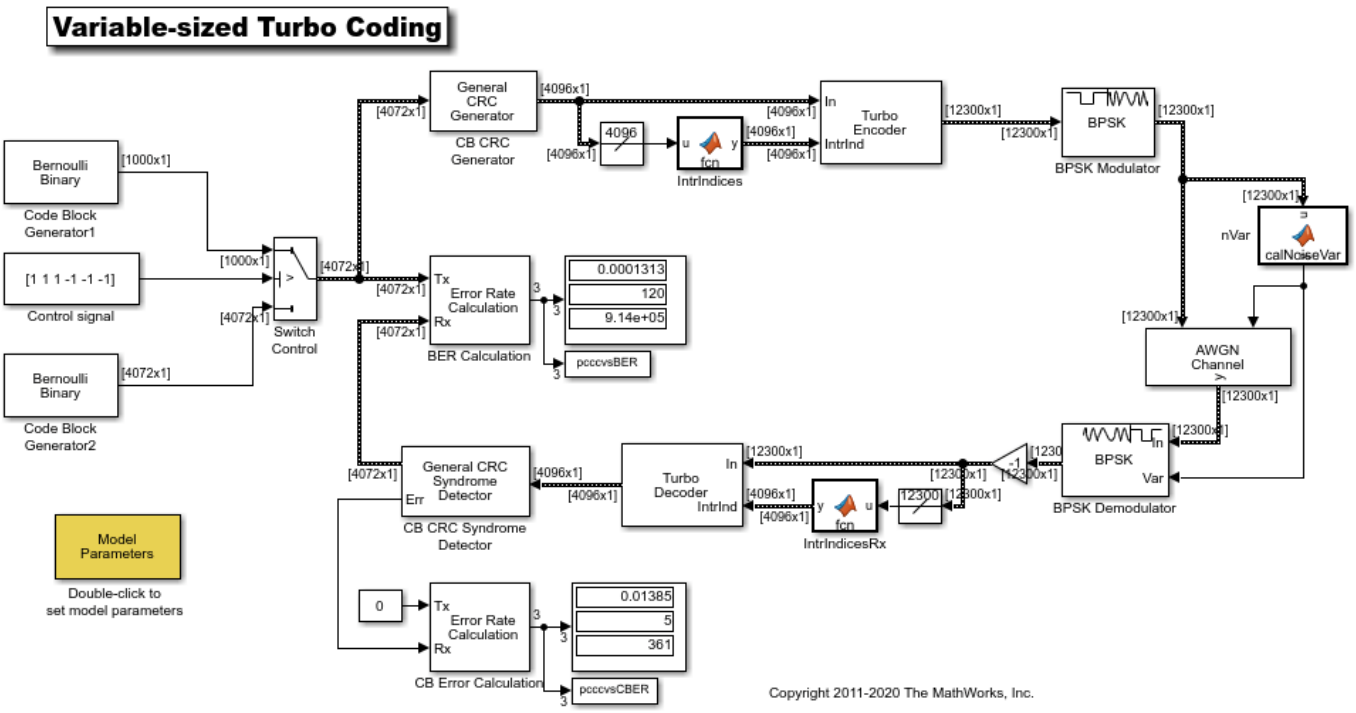
The following figure shows the bit error rate performance of the parallel concatenated coding scheme in an AWGN channel over a range of Eb/No values for two sets of code block lengths and number of decoding iterations.



As the figure shows, the iterative decoding performance improves with an increase in the number of decoding iterations (at the expense of computational complexity) and larger block lengths (at the expense of decoding latency).

Variable-sized Turbo Coding

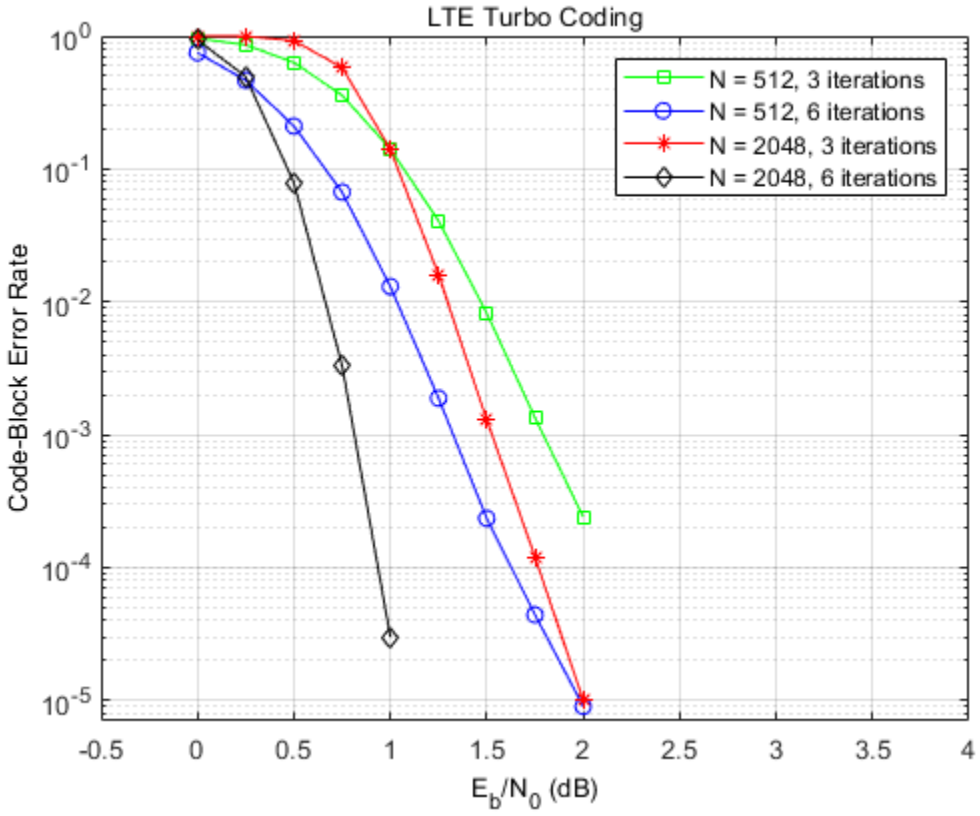
The companion model commpcccvs.slx highlights turbo coding using variable-sized code-blocks within a simulation run.



The model is set up to run two user specified code-block lengths, which vary as per the selected control signal. The interleaver indices per block length and the noise variance are calculated per time step. Using the CRC syndrome detector, the model displays the code-block error rate in addition to the bit error rate, as the former is the more relevant performance metric with variable-sized code blocks.

CBER Performance

The following figure shows the code-block error rate performance of the parallel concatenated coding scheme in an AWGN channel over a range of Eb/No values for a similar set up as used for BER.



We observe similar improvements as before in performance with increase in the number of decoding iterations and/or block lengths.

Further Exploration

The example allows you to explore the effects of different block lengths and number of decoding iterations on the system performance. It supports all of the 188 code block sizes specified in [4] for a user-specified fixed number of decoding iterations.

Selected References

1. C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error correcting coding and decoding: turbo codes," Proc. IEEE® Int. Conf. on Communications, Geneva, Switzerland, May 1993, pp. 1064-1070.
2. Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara, "A Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes," JPL TDA Progress Report, Vol. 42-127, Nov. 1996.
3. Schlegel, Christian B. and Lance C. Perez, "Trellis and Turbo Coding", IEEE Press, 2004.
4. 3GPP TS 36.212 v10.8.0, "3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (Release 10)", 2013-06.

Tail-Biting Convolutional Coding

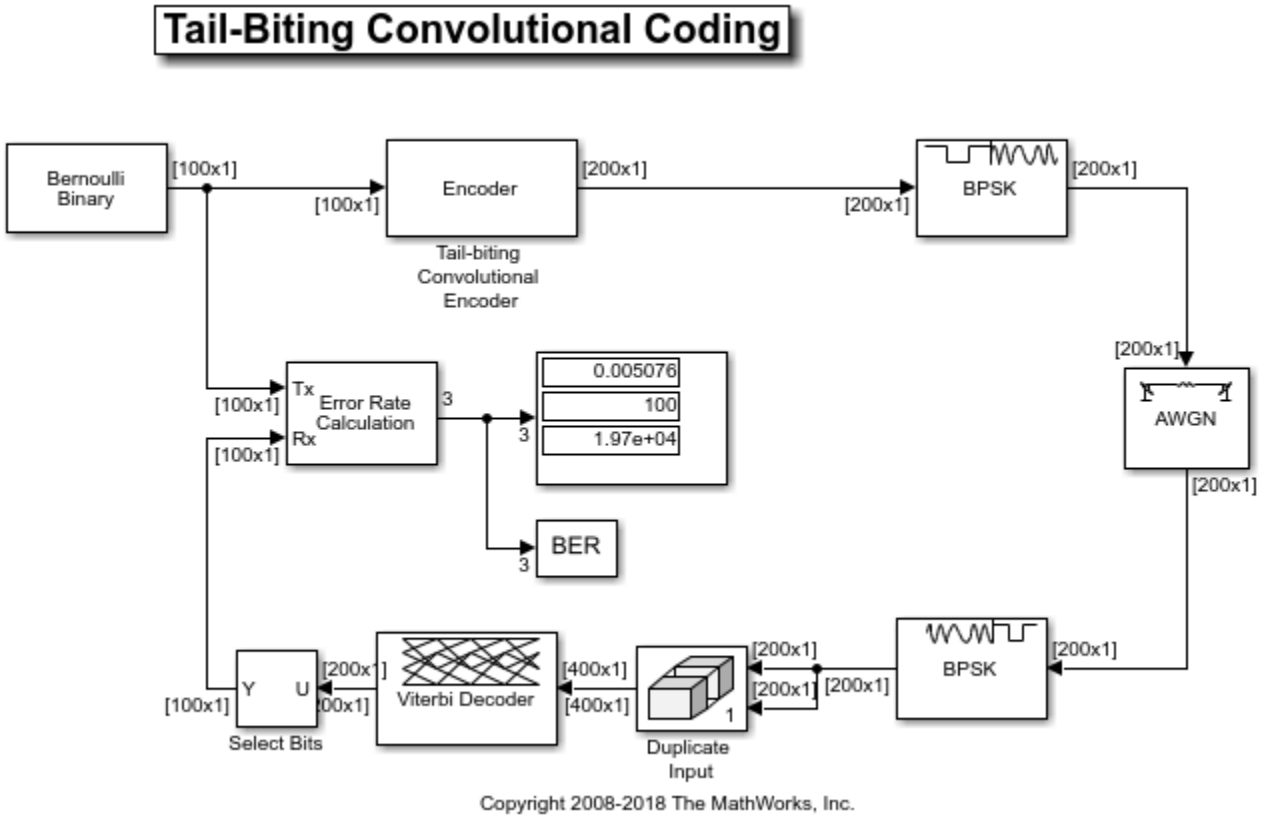
This model shows how to use the `Convolutional Encoder` and `Viterbi Decoder` blocks to simulate a tail-biting convolutional code. Terminating the trellis of a convolutional code is a key parameter in the code's performance for packet-based communications. Tail-biting convolutional coding is a technique of trellis termination which avoids the rate loss incurred by zero-tail termination at the expense of a more complex decoder [1].

The example uses an ad-hoc suboptimal decoding method for tail-biting decoding and shows how the encoding is achieved for a feed-forward encoder. Bit-Error-Rate performance comparisons are made with the zero-tailed case for a standard convolutional code.

Tail-Biting Encoding

Tail-biting encoding ensures that the starting state of the encoder is the same as it's ending state (and that this state value does not necessarily have to be the all-zero state). For a rate `1/n` feed-forward encoder, this is achieved by initializing the `m` memory elements of the encoder with the last `m` information bits of a block of data of length `L`, and ignoring the output. All of the `L` bits are then input to the encoder and the resultant `L*n` output bits are used as the codeword.

This is modeled by the `Tail-biting Convolutional Encoder` subsystem in the following model, commtailbiting.slx:



For a block length of 100 bits, the encoder subsystem outputs 200 bits for a rate 1/2 feed-forward encoder with 6 memory elements. The `Display` block in the subsystem indicates the initial and final states are identical for each block of processed data.

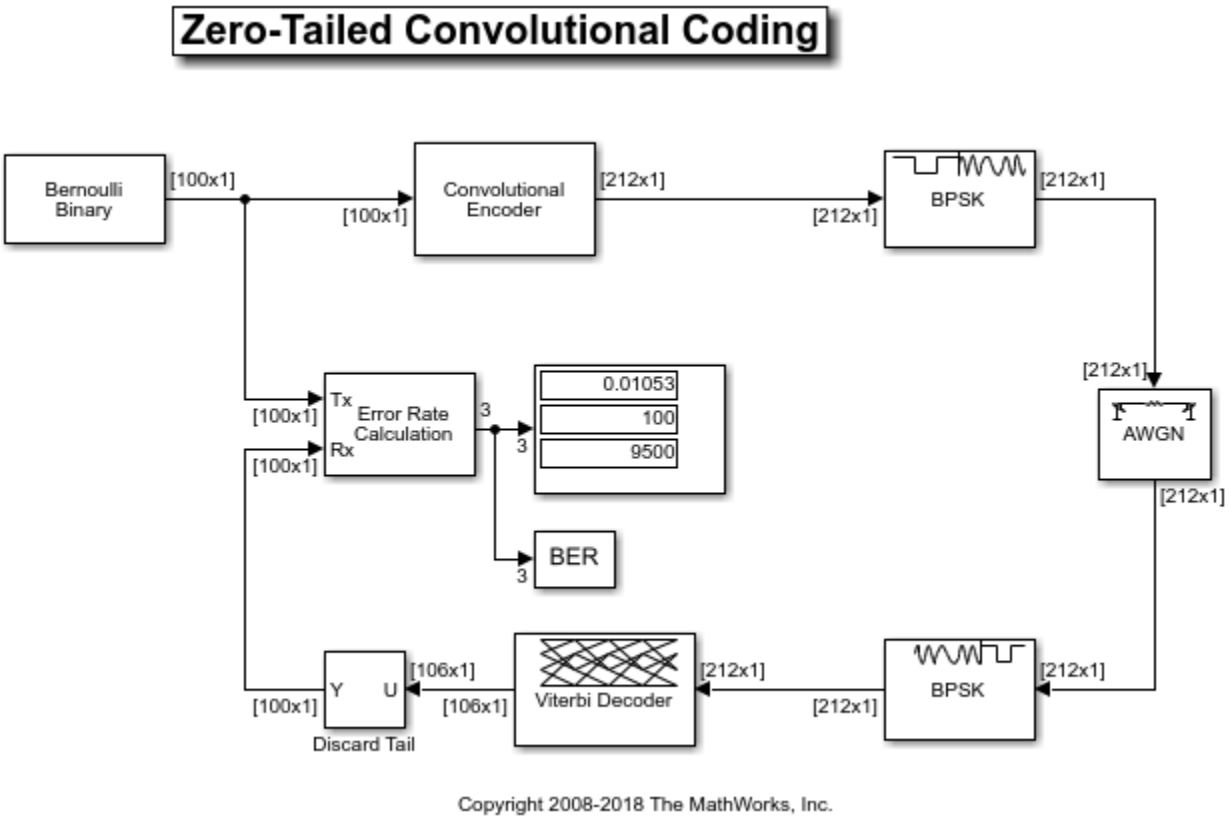
The Convolutional Encoder blocks use the "Truncated (reset every frame)" setting for the Operation mode parameter to indicate the block-wise processing.

Refer to the “Tailbiting Encoding Using Feedback Encoders” on page 16-53 as per [2] on how to achieve tail-biting encoding for a feedback encoder.

Zero-Tailed Encoding

In comparison, the zero-tail termination method appends `m` zeros to a block of data to ensure the feed-forward encoder starts from and ends in the all-zero state for each block. This incurs a rate loss due to the extra tail bits (i.e. non-informational bits) that are transmitted.

Referring to the following model, commterminatedcnv.slx,



observe that for the same block length of 100 bits, the encoder output now includes the zero-tail bits resulting in an actual code rate of 100/212 which is less than that achieved by the tail-biting encoder.

The `Convolutional Encoder` block uses the "Terminate trellis by appending bits" setting for the Operation mode parameter for this case, which works for feedback encoders as well.

Tail-Biting Decoding

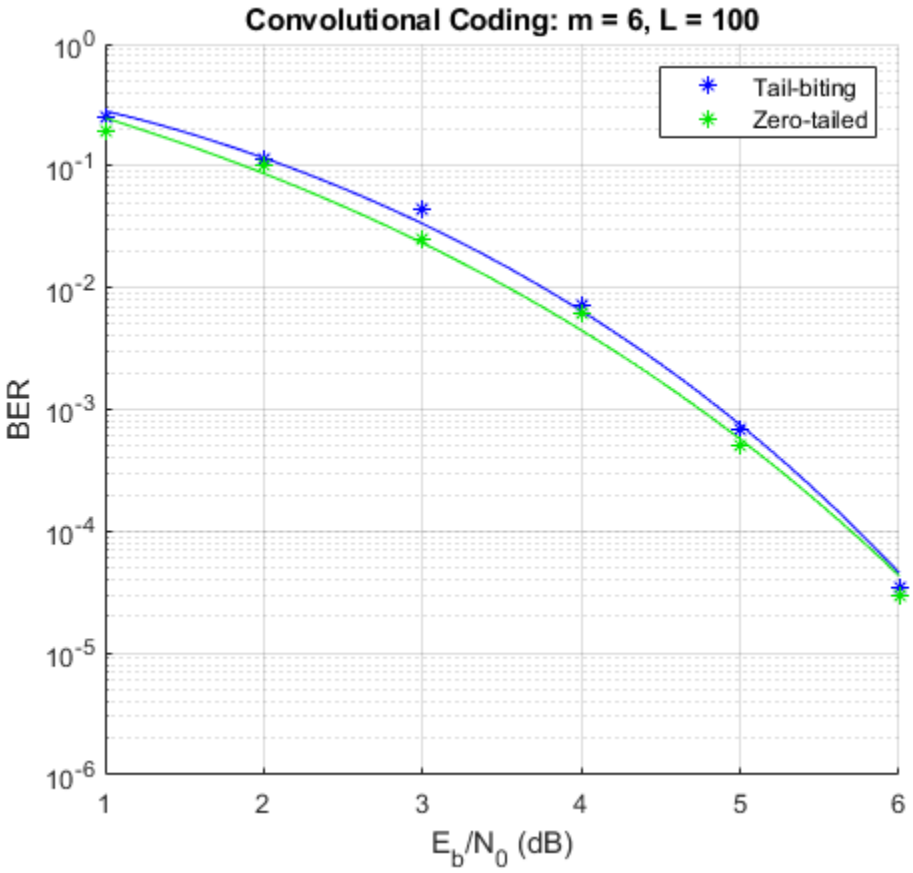
The maximum likelihood tail-biting decoder involves determining the best path in the trellis under the constraint that it starts and ends in the same state. A way to implement this is to run `M` parallel Viterbi algorithms where `M` is the number of states in the trellis, and select the decoded bits based on the Viterbi algorithm that gives the best metric. However this makes the decoding `M` times more complex than that for zero-tailed encoding.

This example uses an ad-hoc suboptimum scheme as per [3], which is much simpler than the maximum likelihood approach and yet performs comparably. The scheme is based on the premise that the tail-biting trellis can be considered circular as it starts and ends in the same state. This allows the Viterbi algorithm to be continued past the end of a block by repeating the received codeword circularly. As a result, the model repeats the received codeword from the demodulator and runs this data set through the Viterbi decoder, performing the traceback from the best state at the end of the repeated data set. Only a portion of the decoded bits from the middle are selected as the decoded message bits.

The Operation mode parameter for the Viterbi Decoder block is set to be "Truncated" for the tail-biting case while it is set to "Terminated" for the zero-tailed case.

BER Performance

The example compares the Bit-error-rate performance of the two termination methods for hard-decision decoding in an AWGN channel over a range of Eb/No values. Note that the two models are set such that they can be simulated over a range of Eb/No values using BERTool.



As the figure shows the ad-hoc tail-biting decoding scheme performs comparatively close to the lower bounded performance of the zero-tailed convolutional code for the chosen parameters.

Further Exploration

Upon loading, the models initialize a set of variables that control the simulation. These include the block length, Eb/No and the maximum number of errors and bits simulated. You are encouraged to play with the values of these variables to see their effects on the link performance.

Note that the ad-hoc decoding scheme's performance is sensitive to the block length used. Also the performance of the code is dependent on the traceback decoding length used for the Viterbi algorithm.

Selected Bibliography

1 H. Ma and J. Wolf, "On Tail Biting convolutional codes," *IEEE Transactions on Communications*, Vol. COM-34, No. 2, Feb. 1986, pp. 104-11.

2 C. Weiss, C. Bettstetter, S. Riedel, "Code Construction and Decoding of Parallel Concatenated Tail-Biting Codes," *IEEE Transactions on Information Theory*, vol. 47, No. 1, Jan. 2001, pp. 366-386.

3 Y. E. Wang and R. Ramesh, "To Bite or not to Bite ? A study of Tail Bits vs. Tail-Biting," Personal, Indoor and Mobile Radio Communications, 1996. PIMRC'96, Seventh IEEE® International Symposium, Volume 2, Oct. 15-18, 1996, Page(s):317 - 321.

Log-Likelihood Ratio (LLR) Demodulation

This example shows the BER performance improvement for QPSK modulation when using log-likelihood ratio (LLR) instead of hard-decision demodulation in a convolutionally coded communication link. With LLR demodulation, one can use the Viterbi decoder either in the unquantized decoding mode or the soft-decision decoding mode. Unquantized decoding, where the decoder inputs are real values, though better in terms of BER, is not practically viable. In the more practical soft-decision decoding, the demodulator output is quantized before being fed to the decoder. It is generally observed that this does not incur a significant cost in BER while significantly reducing the decoder complexity. We validate this experimentally through this example.

For a Simulink™ version of this example, see “LLR vs. Hard Decision Demodulation in Simulink” on page 8-105.

Initialization

Initialize simulation parameters.

```
M = 4;                % Modulation order
k = log2(M);          % Bits per symbol
bitsPerIter = 1.2e4; % Number of bits to simulate
EbNo = 3;             % Information bit Eb/No in dB
```

Initialize coding properties for a rate 1/2, constraint length 7 code.

```
codeRate = 1/2;          % Code rate of convolutional encoder
constLen = 7;            % Constraint length of encoder
codeGenPoly = [171 133]; % Code generator polynomial of encoder
tblen = 32;              % Traceback depth of Viterbi decoder
trellis = poly2trellis(constLen,codeGenPoly);
```

Create a `comm.ConvolutionalEncoder` System object™ by using `trellis` as an input.

```
enc = comm.ConvolutionalEncoder(trellis);
```

Modulator and Channel

Create a `comm.QPSKModulator` and two `comm.QPSKDemodulator` System objects. Configure the first demodulator to output hard-decision bits. Configure the second to output LLR values.

```
qpskMod = comm.QPSKModulator('BitInput',true);
demodHard = comm.QPSKDemodulator('BitOutput',true,...
    'DecisionMethod','Hard decision');
demodLLR = comm.QPSKDemodulator('BitOutput',true,...
    'DecisionMethod','Log-likelihood ratio');
```

Create an `comm.AWGNChannel` object. The signal going into the AWGN channel is the modulated encoded signal. To achieve the required noise level, adjust the Eb/No for coded bits and multi-bit symbols. Set this as the `EbNo` of the channel object.

```
chan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'BitsPerSymbol',k);
EbNoCoded = EbNo + 10*log10(codeRate);
chan.EbNo = EbNoCoded;
```

Viterbi Decoding

Create `comm.ViterbiDecoder` objects to act as the hard-decision, unquantized, and soft-decision decoders. For all three decoders, set the traceback depth to `tblen`.

```
decHard = comm.ViterbiDecoder(trellis,'InputFormat','Hard', ...
    'TracebackDepth',tblen);

decUnquant = comm.ViterbiDecoder(trellis,'InputFormat','Unquantized', ...
    'TracebackDepth',tblen);

decSoft = comm.ViterbiDecoder(trellis,'InputFormat','Soft', ...
    'SoftInputWordLength',3,'TracebackDepth',tblen);
```

Quantization for soft-decoding

Before using a `comm.ViterbiDecoder` object in the soft-decision mode, the output of the demodulator needs to be quantized. This example uses a `comm.ViterbiDecoder` object with a `SoftInputWordLength` of 3. This value is a good compromise between short word lengths and a small BER penalty. Define parition points for 3-bit quantization.

```
snrdB = EbNoCoded + 10*log10(k);
NoiseVariance = 10.^(-snrdB/10);
demodLLR.Variance = NoiseVariance;
paritionPoints = (-1.5:0.5:1.5)/NoiseVariance;
```

Calculating the Error Rate

Create `comm.ErrorRate` objects to compare the decoded bits to the original transmitted bits. The Viterbi decoder creates a delay in the decoded bit stream output equal to the traceback length. To account for this delay, set the `ReceiveDelay` property of the `comm.ErrorRate` objects to `tblen`.

```
errHard = comm.ErrorRate('ReceiveDelay',tblen);
errUnquant = comm.ErrorRate('ReceiveDelay',tblen);
errSoft = comm.ErrorRate('ReceiveDelay',tblen);
```

System Simulation

Generate `bitsPerIter` message bits. Then convolutionally encode and modulate the data.

```
txData = randi([0 1],bitsPerIter,1);
encData = enc(txData);
modData = qpskMod(encData);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = chan(modData);
```

Demodulate the received signal and output hard-decision bits.

```
hardData = demodHard(rxSig);
```

Demodulate the received signal and output LLR values.

```
LLRData = demodLLR(rxSig);
```

Hard-decision decoding

Pass the demodulated data through the Viterbi decoder. Compute the error statistics.

```
rxDataHard = decHard(hardData);
berHard = errHard(txData,rxDataHard);
```

Unquantized decoding

Pass the demodulated data through the Viterbi decoder. Compute the error statistics.

```
rxDataUnquant = decUnquant(LLRData);
berUnquant = errUnquant(txData,rxDataUnquant);
```

Soft-decision decoding

Pass the demodulated data to the `quantiz` function. This data must be multiplied by `-1` before being passed to the quantizer, because, in soft-decision mode, the Viterbi decoder assumes that positive numbers correspond to 1s and negative numbers to 0s. Pass the quantizer output to the Viterbi decoder. Compute the error statistics.

```
quantizedValue = quantiz(-LLRData,paritionPoints);
rxDataSoft = decSoft(double(quantizedValue));
berSoft = errSoft(txData,rxDataSoft);
```

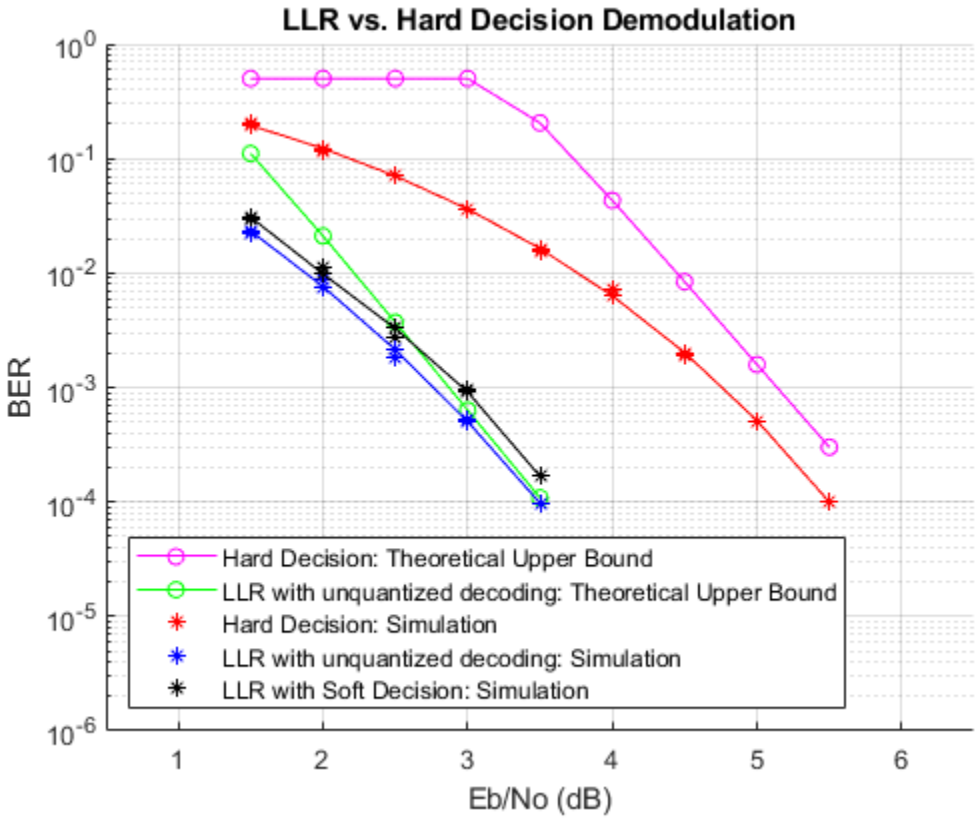
Running Simulation Example

Simulate the previously described communications system over a range of Eb/No values by executing the simulation file `simLLRvsHD`. It plots BER results as they are generated. BER results for hard-decision demodulation and LLR demodulation with unquantized and soft-decision decoding are plotted in red, blue, and black, respectively. A comparison of simulation results with theoretical results is also shown. Observe that the BER is only slightly degraded by using soft-decision decoding instead of unquantized decoding. The gap between the BER curves for soft-decision decoding and the theoretical bound can be narrowed by increasing the number of quantizer levels.

This example may take some time to compute BER results. If you have the Parallel Computing Toolbox™ (PCT) installed, you can set `usePCT` to `true` to run the simulation in parallel. In this case, the file `LLRvsHDwithPCT` is run.

To obtain results over a larger range of Eb/No values, modify the appropriate supporting files. Note that you can obtain more statistically reliable results by collecting more errors.

```
usePCT = false;
if usePCT && license('checkout','Distrib_Computing_Toolbox') ...
        && ~isempty(ver('parallel'))
    LLRvsHDwithPCT(1.5:0.5:5.5,5);
else
    simLLRvsHD(1.5:0.5:5.5,5);
end
```



Appendix

The following functions are used in this example:

- simLLRvsHD.m — Simulates system without PCT.
- LLRvsHDwithPCT.m — Simulates system with PCT.
- simLLRvsHDPCT.m — Helper function called by LLRvsHDwithPCT.

FBMC vs. OFDM Modulation

This example compares Filter Bank Multi-Carrier (FBMC) with Orthogonal Frequency Division Multiplexing (OFDM) and highlights the merits of the candidate modulation scheme for Fifth Generation (5G) communication systems.

FBMC was considered as an alternate waveform to OFDM in the 3GPP RAN study phase I during 3GPP Release 14.

Introduction

This example compares Filter Bank Multi-Carrier (FBMC) modulation with generic OFDM modulation. FBMC offers ways to overcome the known limitations of OFDM of reduced spectral efficiency and strict synchronization requirements. These advantages have led it to being considered as one of the modulation techniques for 5G communication systems [2 on page 8-0 , 4 on page 8-0].

This example models Filter Bank Multi-Carrier modulation with configurable parameters and highlights the basic transmit and receive processing.

```
s = rng(211);            % Set RNG state for repeatability
```

System Parameters

Define system parameters for the example. You can modify these parameters to explore their impact on the system.

```
numFFT = 1024;           % Number of FFT points
numGuards = 212;         % Guard bands on both sides
K = 4;                   % Overlapping symbols, one of 2, 3, or 4
numSymbols = 100;        % Simulation length in symbols
bitsPerSubCarrier = 2;   % 2: 4QAM, 4: 16QAM, 6: 64QAM, 8: 256QAM
snrdB = 12;              % SNR in dB
```

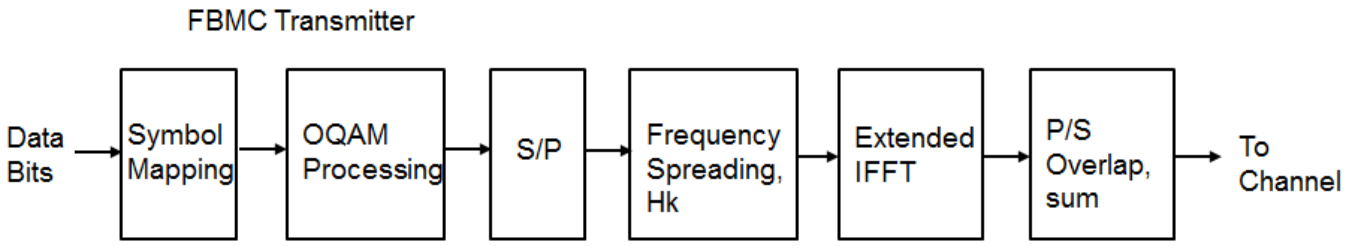
Filter Bank Multi-Carrier Modulation

FBMC filters each subcarrier modulated signal in a multicarrier system. The prototype filter is the one used for the zero frequency carrier and is the basis for the other subcarrier filters. The filters are characterized by the overlapping factor, K which is the number of multicarrier symbols that overlap in the time domain. The prototype filter order can be chosen as $2*K-1$ where K = 2, 3, or 4 and is selected as per the PHYDYAS project [1 on page 8-0].

The current FBMC implementation uses frequency spreading. It uses an $N*K$ length IFFT with symbols overlapped with a delay of $N/2$, where N is the number of subcarriers. This design choice makes it easy to analyze FBMC and compare with other modulation methods.

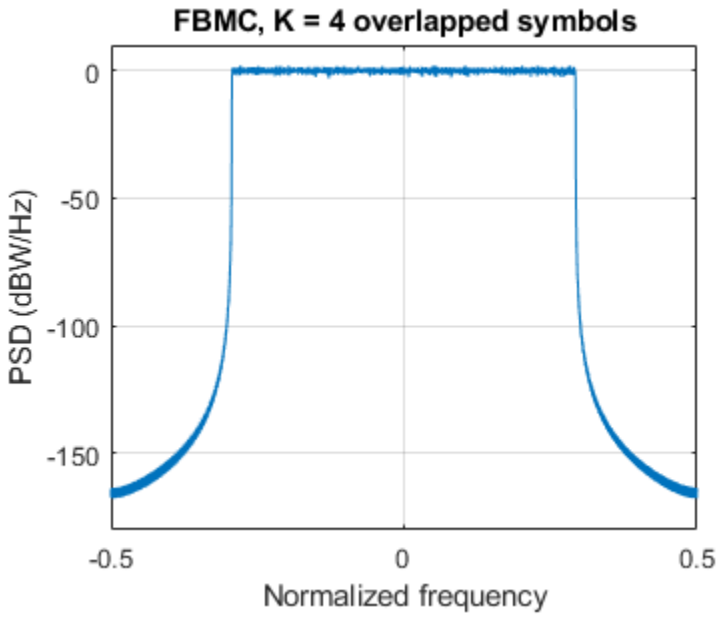
To achieve full capacity, offset quadrature amplitude modulation (OQAM) processing is employed. The real and imaginary parts of a complex data symbol are not transmitted simultaneously, as the imaginary part is delayed by half the symbol duration.

The transmit-end processing is shown in the following diagram.




```
% Prototype filter
switch K
    case 2
        HkOneSided = sqrt(2)/2;
    case 3
        HkOneSided = [0.911438 0.411438];
    case 4
        HkOneSided = [0.971960 sqrt(2)/2 0.235147];
    otherwise
        return
end
% Build symmetric filter
Hk = [fliplr(HkOneSided) 1 HkOneSided];

% Transmit-end processing
%   Initialize arrays
L = numFFT-2*numGuards;  % Number of complex symbols per OFDM symbol
KF = K*numFFT;
KL = K*L;
dataSubCar = zeros(L, 1);
dataSubCarUp = zeros(KL, 1);

sumFBMCSpec = zeros(KF*2, 1);
sumOFDMSpec = zeros(numFFT*2, 1);

numBits = bitsPerSubCarrier*L/2;    % account for oversampling by 2
inpData = zeros(numBits, numSymbols);
rxBits = zeros(numBits, numSymbols);
txSigAll = complex(zeros(KF, numSymbols));
symBuf = complex(zeros(2*KF, 1));

% Loop over symbols
for symIdx = 1:numSymbols

    % Generate mapped symbol data
    inpData(:, symIdx) = randi([0 1], numBits, 1);
    modData = qammod(inpData(:, symIdx), 2^bitsPerSubCarrier, ...
        'InputType', 'Bit', 'UnitAveragePower', true);

    % OQAM Modulator: alternate real and imaginary parts
    if rem(symIdx,2)==1     % Odd symbols
        dataSubCar(1:2:L) = real(modData);
        dataSubCar(2:2:L) = 1i*imag(modData);
    else                    % Even symbols
        dataSubCar(1:2:L) = 1i*imag(modData);
        dataSubCar(2:2:L) = real(modData);
    end
```

```
    % Upsample by K, pad with guards, and filter with the prototype filter
    dataSubCarUp(1:K:end) = dataSubCar;
    dataBitsUpPad = [zeros(numGuards*K,1); dataSubCarUp; zeros(numGuards*K,1)];
    X1 = filter(Hk, 1, dataBitsUpPad);
    % Remove 1/2 filter length delay
    X = [X1(K:end); zeros(K-1,1)];

    % Compute IFFT of length KF for the transmitted symbol
    txSymb = fftshift(ifft(X));

    % Transmitted signal is a sum of the delayed real, imag symbols
    symBuf = [symBuf(numFFT/2+1:end); complex(zeros(numFFT/2,1))];
    symBuf(KF+(1:KF)) = symBuf(KF+(1:KF)) + txSymb;

    % Compute power spectral density (PSD)
    currSym = complex(symBuf(1:KF));
    [specFBMC, fFBMC] = periodogram(currSym, hann(KF, 'periodic'), KF*2, 1);
    sumFBMCSpec = sumFBMCSpec + specFBMC;

    % Store transmitted signals for all symbols
    txSigAll(:,symIdx) = currSym;
end

% Plot power spectral density
sumFBMCSpec = sumFBMCSpec/mean(sumFBMCSpec(1+K+2*numGuards*K:end-2*numGuards*K-K));
plot(fFBMC-0.5,10*log10(sumFBMCSpec));
grid on
axis([-0.5 0.5 -180 10]);
xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['FBMC, K = ' num2str(K) ' overlapped symbols'])
set(gcf, 'Position', figposition([15 50 30 30]));
```

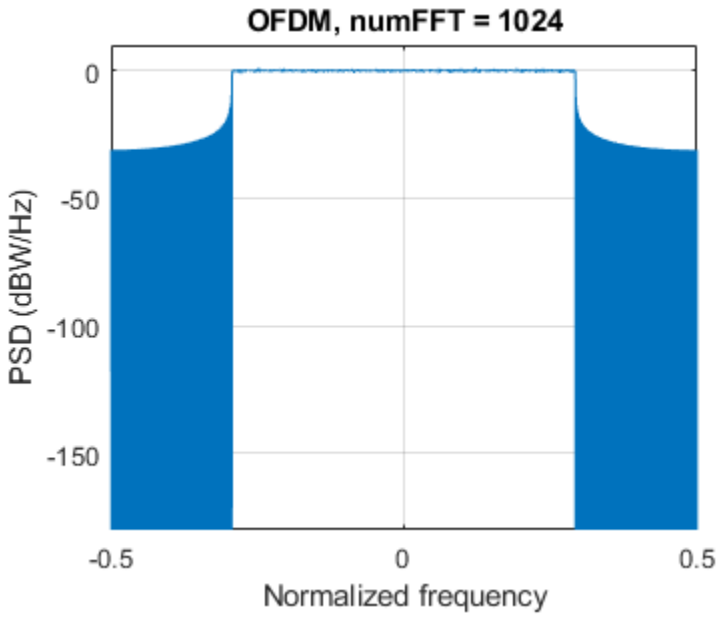


The power spectral density of the FBMC transmit signal is plotted to highlight the low out-of-band leakage.

OFDM Modulation with Corresponding Parameters

For comparison, we review the existing OFDM modulation technique, using the full occupied band, however, without a cyclic prefix.

```
for symIdx = 1:numSymbols

    inpData2 = randi([0 1], bitsPerSubCarrier*L, 1);
    modData = qammod(inpData2, 2^bitsPerSubCarrier, ...
        'InputType', 'Bit', 'UnitAveragePower', true);

    symOFDM = [zeros(numGuards,1); modData; zeros(numGuards,1)];
    ifftOut = sqrt(numFFT).*ifft(ifftshift(symOFDM));

    [specOFDM,fOFDM] = periodogram(ifftOut, rectwin(length(ifftOut)), ...
        numFFT*2, 1, 'centered');
    sumOFDMSpec = sumOFDMSpec + specOFDM;
end

% Plot power spectral density (PSD) over all subcarriers
sumOFDMSpec = sumOFDMSpec/mean(sumOFDMSpec(1+2*numGuards:end-2*numGuards));
figure;
plot(fOFDM,10*log10(sumOFDMSpec));
grid on
axis([-0.5 0.5 -180 10]);
xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['OFDM, numFFT = ' num2str(numFFT)])
set(gcf, 'Position', figposition([46 50 30 30]));
```

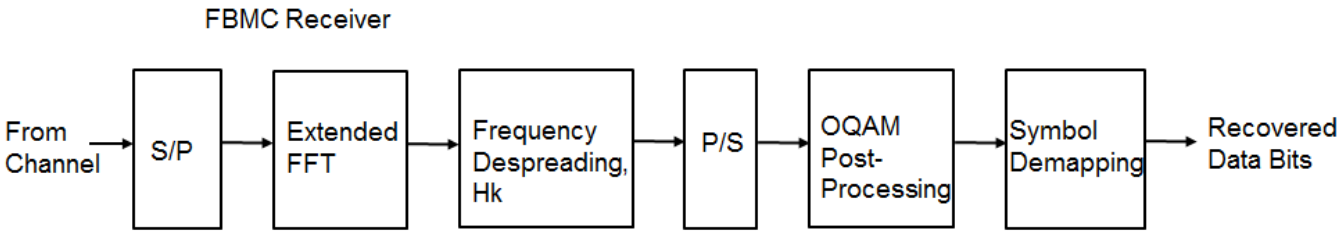


Comparing the plots of the spectral densities for OFDM and FBMC schemes, FBMC has lower side lobes. This allows a higher utilization of the allocated spectrum, leading to increased spectral efficiency.

FBMC Receiver with No Channel

The example implements a basic FBMC demodulator and measures the BER for the chosen configuration in the absence of a channel. The processing includes matched filtering followed by OQAM separation to form the received data symbols. These are de-mapped to bits and the resultant bit error rate is determined. In the presence of a channel, linear multi-tap equalizers may be used to mitigate the effects of frequency-selective fading.

The receive-end processing is shown in the following diagram.




```
BER = comm.ErrorRate;

% Process symbol-wise
for symIdx = 1:numSymbols
    rxSig = txSigAll(:, symIdx);

    % Add WGN
    rxNsig = awgn(rxSig, snrdB, 'measured');

    % Perform FFT
    rxf = fft(fftshift(rxNsig));

    % Matched filtering with prototype filter
    rxfmf = filter(Hk, 1, rxf);
    % Remove K-1 delay elements
    rxfmf = [rxfmf(K:end); zeros(K-1,1)];
    % Remove guards
    rxfmfg = rxfmf(numGuards*K+1:end-numGuards*K);

    % OQAM post-processing
    %  Downsample by 2K, extract real and imaginary parts
    if rem(symIdx, 2)
        % Imaginary part is K samples after real one
        r1 = real(rxfmfg(1:2*K:end));
        r2 = imag(rxfmfg(K+1:2*K:end));
        rcomb = complex(r1, r2);
    else
        % Real part is K samples after imaginary one
        r1 = imag(rxfmfg(1:2*K:end));
        r2 = real(rxfmfg(K+1:2*K:end));
        rcomb = complex(r2, r1);
    end
    %  Normalize by the upsampling factor
    rcomb = (1/K)*rcomb;
```

```
    % De-mapper: Perform hard decision
    rxBits(:, symIdx) = qamdemod(rcomb, 2^bitsPerSubCarrier, ...
        'OutputType', 'bit', 'UnitAveragePower', true);
end

% Measure BER with appropriate delay
BER.ReceiveDelay = bitsPerSubCarrier*KL;
ber = BER(inpData(:), rxBits(:));

% Display Bit error
disp(['FBMC Reception for K = ' num2str(K) ', BER = ' num2str(ber(1)) ...
    ' at SNR = ' num2str(snrdB) ' dB'])
```

```
FBMC Reception for K = 4, BER = 0 at SNR = 12 dB
```

```
% Restore RNG state
rng(s);
```

Conclusion and Further Exploration

The example presents the basic transmit and receive characteristics of the FBMC modulation scheme. Explore this example by changing the number of overlapping symbols, FFT lengths, guard band lengths, and SNR values.

Refer to “UFMC vs. OFDM Modulation” on page 8-87 for an example that describes the Universal Filtered Multi-Carrier (UFMC) modulation scheme.

FBMC is considered advantageous in comparison to OFDM by offering higher spectral efficiency. Due to the per subcarrier filtering, it incurs a larger filter delay (in comparison to UFMC) and also requires OQAM processing, which requires modifications for MIMO processing.

Further explorations should include modifications for MIMO processing with more complete link-level processing including channel estimation and equalization [2 on page 8-0].

Selected Bibliography

1. "FBMC physical layer: a primer", PHYDYAS EU FP7 Project 2010. http://www.ict-phydyas.org
2. Schellman, M., Zhao, Z., Lin, H., Siohan, P., Rajatheva, N., Luecken, V., Ishaque, A., "FBMC-based air interface for 5G mobile: Challenges and proposed solutions", CROWNCOM 2014, pp 102-107.
3. Farhang-Boroujeny, B., "OFDM versus filter bank multicarrier", IEEE® Signal Proc. Mag., vol. 28, pp. 92-112, May 2011.
4. Wunder, G., Kasparick, M., Wild, T., Schaich, F., Yejian Chen, Dryjanski, M., Buczkowski, M., Pietrzyk, S., Michailow, N., Matthe, M., Gaspar, I., Mendes, L., Festag, A., Fettweis, G., Dore, J.-B., Cassiau, N., Ktenas, D., Berg, V., Eged, B., Vago, P., "5GNOW: Intermediate frame structure and transceiver concepts", Globecom workshops, pp. 565-570, 2014.

F-OFDM vs. OFDM Modulation

This example compares Orthogonal Frequency Division Multiplexing (OFDM) with Filtered-OFDM (F-OFDM) and highlights the merits of the candidate modulation scheme for Fifth Generation (5G) communication systems.

Introduction

This example compares Filtered-OFDM modulation with generic Cyclic Prefix OFDM (CP-OFDM) modulation. For F-OFDM, a well-designed filter is applied to the time domain OFDM symbol to improve the out-of-band radiation of the sub-band signal, while maintaining the complex-domain orthogonality of OFDM symbols.

This example models Filtered-OFDM modulation with configurable parameters. It highlights the filter design technique and the basic transmit/receive processing.

```
s = rng(211);       % Set RNG state for repeatability
```

System Parameters

Define system parameters for the example. These parameters can be modified to explore their impact on the system.

```
numFFT = 1024;           % Number of FFT points
numRBs = 50;             % Number of resource blocks
rbSize = 12;             % Number of subcarriers per resource block
cpLen = 72;              % Cyclic prefix length in samples

bitsPerSubCarrier = 6;   % 2: QPSK, 4: 16QAM, 6: 64QAM, 8: 256QAM
snrdB = 18;              % SNR in dB

toneOffset = 2.5;        % Tone offset or excess bandwidth (in subcarriers)
L = 513;                 % Filter length (=filterOrder+1), odd
```

Filtered-OFDM Filter Design

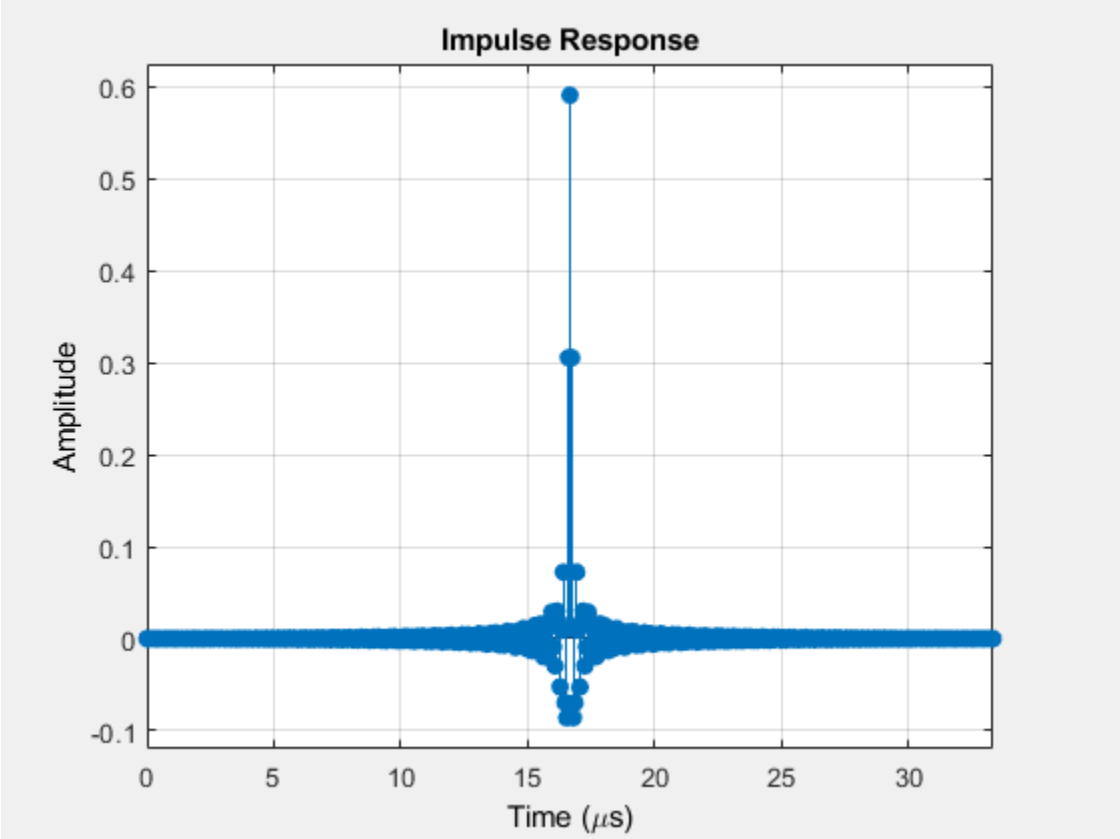
Appropriate filtering for F-OFDM satisfies the following criteria:

- Should have a flat passband over the subcarriers in the sub-band
- Should have a sharp transition band to minimize guard-bands
- Should have sufficient stop-band attenuation

A filter with a rectangular frequency response, i.e. a sinc impulse response, meets these criteria. To make this causal, the low-pass filter is realized using a window, which, effectively truncates the impulse response and offers smooth transitions to zero on both ends [3 on page 8-0].

```
numDataCarriers = numRBs*rbSize;    % number of data subcarriers in sub-band
halfFilt = floor(L/2);
n = -halfFilt:halfFilt;

% Sinc function prototype filter
pb = sinc((numDataCarriers+2*toneOffset).*n./numFFT);

% Sinc truncation window
w = (0.5*(1+cos(2*pi.*n/(L-1)))).^0.6;
```

```
% Normalized lowpass filter coefficients
fnum = (pb.*w)/sum(pb.*w);

% Filter impulse response
h = fvtool(fnum, 'Analysis', 'impulse', ...
    'NormalizedFrequency', 'off', 'Fs', 15.36e6);
h.CurrentAxes.XLabel.String = 'Time (\mus)';
h.FigureToolbar = 'off';
```

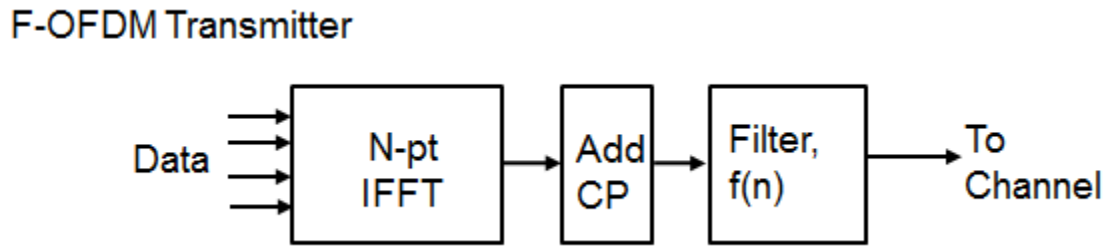



```
% Use dsp filter objects for filtering
filtTx = dsp.FIRFilter('Structure', 'Direct form symmetric', ...
    'Numerator', fnum);
filtRx = clone(filtTx); % Matched filter for the Rx
```

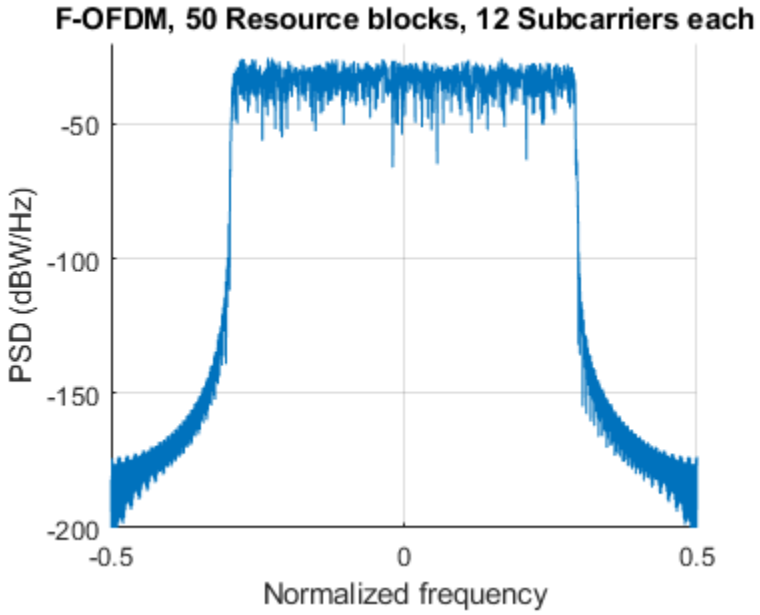
F-OFDM Transmit Processing

In F-OFDM, the sub-band CP-OFDM signal is passed through the designed filter. As the filter's passband corresponds to the signal's bandwidth, only the few subcarriers close to the edge are affected. A key consideration is that the filter length can be allowed to exceed the cyclic prefix length for F-OFDM [1 on page 8-0]. The inter-symbol interference incurred is minimized due to the filter design using windowing (with soft truncation).

Transmit-end processing operations are shown in the following F-OFDM transmitter diagram.




```
% Set up a figure for spectrum plot
hFig = figure('Position', figposition([46 50 30 30]), 'MenuBar', 'none');
axis([-0.5 0.5 -200 -20]);
hold on;
grid on
xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['F-OFDM, ' num2str(numRBs) ' Resource blocks, '  ...
    num2str(rbSize) ' Subcarriers each'])

% Generate data symbols
bitsIn = randi([0 1], bitsPerSubCarrier*numDataCarriers, 1);

% QAM Symbol mapper
symbolsIn = qammod(bitsIn, 2^bitsPerSubCarrier, 'InputType', 'bit', ...
    'UnitAveragePower', true);

% Pack data into an OFDM symbol
offset = (numFFT-numDataCarriers)/2; % for band center
symbolsInOFDM = [zeros(offset,1); symbolsIn; ...
    zeros(numFFT-offset-numDataCarriers,1)];
ifftOut = ifft(ifftshift(symbolsInOFDM));

% Prepend cyclic prefix
txSigOFDM = [ifftOut(end-cpLen+1:end); ifftOut];

% Filter, with zero-padding to flush tail. Get the transmit signal
txSigFOFDM = filtTx([txSigOFDM; zeros(L-1,1)]);

% Plot power spectral density (PSD)
[psd,f] = periodogram(txSigFOFDM, rectwin(length(txSigFOFDM)), ...
    numFFT*2, 1, 'centered');
plot(f,10*log10(psd));
```



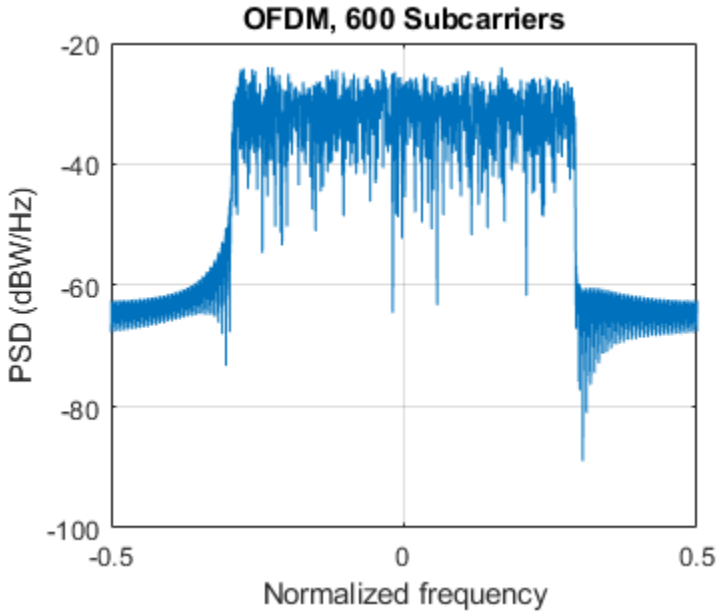

```
% Compute peak-to-average-power ratio (PAPR)
PAPR = comm.CCDF('PAPROutputPort', true, 'PowerUnits', 'dBW');
[~,~,paprFOFDM] = PAPR(txSigFOFDM);
disp(['Peak-to-Average-Power-Ratio for F-OFDM = ' num2str(paprFOFDM) ' dB']);
```

```
Peak-to-Average-Power-Ratio for F-OFDM = 11.371 dB
```

OFDM Modulation with Corresponding Parameters

For comparison, we review the existing OFDM modulation technique, using the full occupied band, with the same length cyclic prefix.

```
% Plot power spectral density (PSD) for OFDM signal
[psd,f] = periodogram(txSigOFDM, rectwin(length(txSigOFDM)), numFFT*2, ...
    1, 'centered');
hFig1 = figure('Position', figposition([46 15 30 30]));
plot(f,10*log10(psd));
grid on
axis([-0.5 0.5 -100 -20]);
xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['OFDM, ' num2str(numRBs*rbSize) ' Subcarriers'])
```




```
% Compute peak-to-average-power ratio (PAPR)
PAPR2 = comm.CCDF('PAPROutputPort', true, 'PowerUnits', 'dBW');
[~,~,paprOFDM] = PAPR2(txSigOFDM);
disp(['Peak-to-Average-Power-Ratio for OFDM = ' num2str(paprOFDM) ' dB']);
```

```
Peak-to-Average-Power-Ratio for OFDM = 9.721 dB
```

Comparing the plots of the spectral densities for CP-OFDM and F-OFDM schemes, F-OFDM has lower side lobes. This allows a higher utilization of the allocated spectrum, leading to increased spectral efficiency.

Refer to the `comm.OFDMModulator` System object™ which can also be used to implement the CP-OFDM modulation.

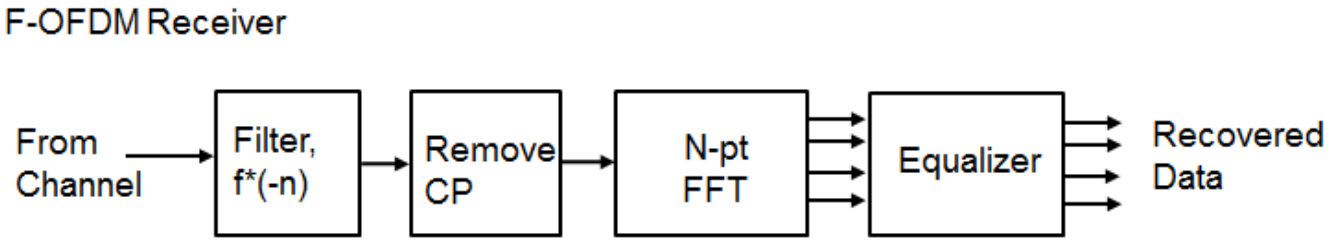
F-OFDM Receiver with No Channel

The example next highlights the basic receive processing for F-OFDM for a single OFDM symbol. The received signal is passed through a matched filter, followed by the normal CP-OFDM receiver. It accounts for both the filtering ramp-up and latency prior to the FFT operation.

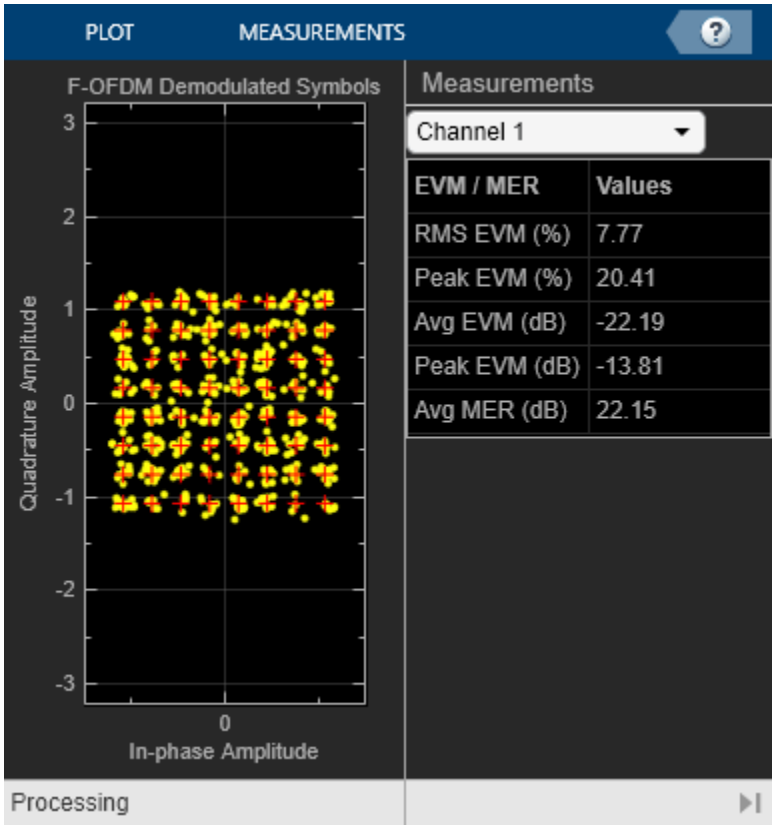
No fading channel is considered in this example but noise is added to the received signal to achieve the desired SNR.

```
% Add WGN
rxSig = awgn(txSigFOFDM, snrdB, 'measured');
```

Receive processing operations are shown in the following F-OFDM receiver diagram.




```
% Receive matched filter
rxSigFilt = filtRx(rxSig);

% Account for filter delay
rxSigFiltSync = rxSigFilt(L:end);

% Remove cyclic prefix
rxSymbol = rxSigFiltSync(cpLen+1:end);

% Perform FFT
RxSymbols = fftshift(fft(rxSymbol));

% Select data subcarriers
dataRxSymbols = RxSymbols(offset+(1:numDataCarriers));

% Plot received symbols constellation
switch bitsPerSubCarrier
    case 2  % QPSK
        refConst = qammod((0:3).', 4, 'UnitAveragePower', true);
    case 4  % 16QAM
        refConst = qammod((0:15).', 16,'UnitAveragePower', true);
    case 6  % 64QAM
        refConst = qammod((0:63).', 64,'UnitAveragePower', true);
    case 8  % 256QAM
        refConst = qammod((0:255).', 256,'UnitAveragePower', true);
end
constDiagRx = comm.ConstellationDiagram( ...
    'ShowReferenceConstellation', true, ...
    'ReferenceConstellation', refConst, ...
    'Position', figposition([20 15 30 40]), ...
    'EnableMeasurements', true, ...
    'MeasurementInterval', length(dataRxSymbols), ...
    'Title', 'F-OFDM Demodulated Symbols', ...
    'Name', 'F-OFDM Reception', ...
    'XLimits', [-1.5 1.5], 'YLimits', [-1.5 1.5]);
constDiagRx(dataRxSymbols);
```




```
% Channel equalization is not necessary here as no channel is modeled

% BER computation
BER = comm.ErrorRate;

% Perform hard decision and measure errors
rxBits = qamdemod(dataRxSymbols, 2^bitsPerSubCarrier, 'OutputType', 'bit', ...
    'UnitAveragePower', true);
ber = BER(bitsIn, rxBits);

disp(['F-OFDM Reception, BER = ' num2str(ber(1)) ' at SNR = ' ...
    num2str(snrdB) ' dB']);
```

```
F-OFDM Reception, BER = 0.00083333 at SNR = 18 dB
```

```
% Restore RNG state
rng(s);
```

As highlighted, F-OFDM adds a filtering stage to the existing CP-OFDM processing at both the transmit and receive ends. The example models the full-band allocation for a user, but the same approach can be applied for multiple bands (one per user) for an uplink asynchronous operation.

Refer to the `comm.OFDMDemodulator` System object™ which can be used to implement the CP-OFDM demodulation after receive matched filtering.

Conclusion and Further Exploration

The example presents the basic characteristics of the F-OFDM modulation scheme at both transmit and receive ends of a communication system. Explore different system parameter values for the number of resource blocks, number of subcarriers per blocks, filter length, tone offset and SNR.

Universal Filtered Multi-Carrier (UFMC) modulation scheme is another approach to sub-band filtered OFDM. For more information, see the “UFMC vs. OFDM Modulation” on page 8-87 example. This F-OFDM example uses a single sub-band while the UFMC example uses multiple sub-bands.

F-OFDM and UFMC both use time-domain filtering with subtle differences in the way the filter is designed and applied. For UFMC, the length of filter is constrained to be equal to the cyclic-prefix length, while for F-OFDM, it can exceed the CP length.

For F-OFDM, the filter design leads to a slight loss in orthogonality (strictly speaking) which affects only the edge subcarriers.

Selected Bibliography

1. Abdoli J., Jia M. and Ma J., "Filtered OFDM: A New Waveform for Future Wireless Systems," 2015 IEEE® 16th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), Stockholm, 2015, pp. 66-70.
2. R1-162152. "OFDM based flexible waveform for 5G." 3GPP TSG RAN WG1 meeting 84bis. Huawei; HiSilicon. April 2016.
3. R1-165425. "F-OFDM scheme and filter design." 3GPP TSG RAN WG1 meeting 85. Huawei; HiSilicon. May 2016.

UFMC vs. OFDM Modulation

This example compares Universal Filtered Multi-Carrier (UFMC) with Orthogonal Frequency Division Multiplexing (OFDM) and highlights the merits of the candidate modulation scheme for Fifth Generation (5G) communication systems.

UFMC was considered as an alternate waveform to OFDM in the 3GPP RAN study phase I during 3GPP Release 14.

Introduction

OFDM, as a multi-carrier modulation technique, has been widely adopted by 4G communication systems, such as LTE and Wi-Fi®. It has many advantages: robustness to channel delays, single-tap frequency domain equalization, and efficient implementation. What is often not highlighted are its costs such as the loss in spectral efficiency due to higher side-lobes and the strict synchronization requirements. New modulation techniques are, thus, being considered for 5G communication systems to overcome some of these factors.

As an example, an LTE system at 20 MHz channel bandwidth uses 100 resource blocks of 12 subcarriers each, at an individual subcarrier spacing of 15 kHz. This utilizes only 18 MHz of the allocated spectrum, leading to a 10 percent loss. Additionally, the cyclic prefix of 144 or 160 samples per OFDM symbol leads to another ~7 percent efficiency loss, for an overall 17 percent loss in possible spectral efficiency.

With the now defined ITU requirements for 5G systems, applications require higher data rates, lower latency and more efficient spectrum usage. This example focuses on the new modulation technique known as Universal Filtered Multi-Carrier (UFMC) and compares it with OFDM within a generic framework.

```
s = rng(211);       % Set RNG state for repeatability
```

System Parameters

Define system parameters for the example. These parameters can be modified to explore their impact on the system.

```
numFFT = 512;        % number of FFT points
subbandSize = 20;    % must be > 1
numSubbands = 10;    % numSubbands*subbandSize <= numFFT
subbandOffset = 156; % numFFT/2-subbandSize*numSubbands/2 for band center

% Dolph-Chebyshev window design parameters
filterLen = 43;      % similar to cyclic prefix length
slobeAtten = 40;     % side-lobe attenuation, dB

bitsPerSubCarrier = 4;   % 2: 4QAM, 4: 16QAM, 6: 64QAM, 8: 256QAM
snrdB = 15;              % SNR in dB
```

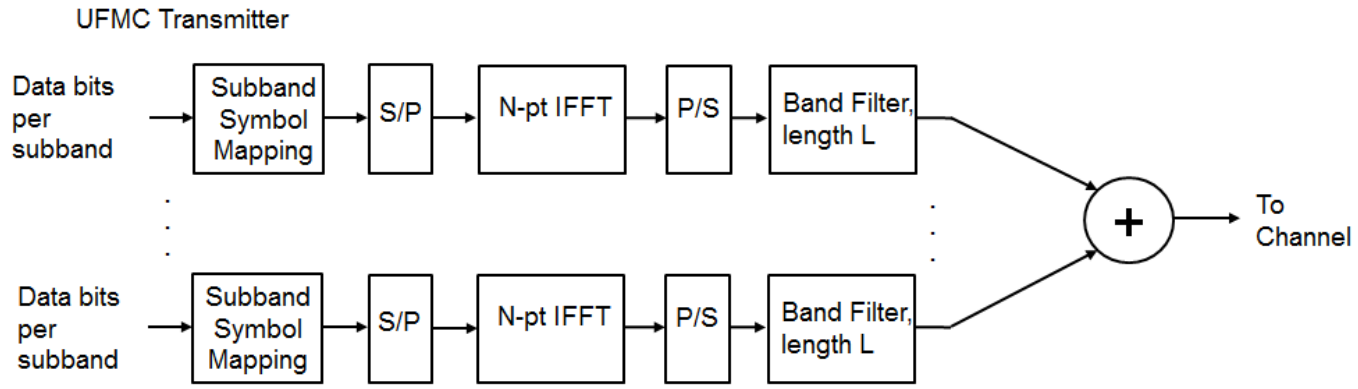
Universal Filtered Multi-Carrier Modulation

UFMC is seen as a generalization of Filtered OFDM and FBMC (Filter Bank Multi-carrier) modulations. The entire band is filtered in filtered OFDM and individual subcarriers are filtered in FBMC, while groups of subcarriers (sub-bands) are filtered in UFMC.

This subcarrier grouping allows one to reduce the filter length (when compared with FBMC). Also, UFMC can still use QAM as it retains the complex orthogonality (when compared with FBMC), which works with existing MIMO schemes.

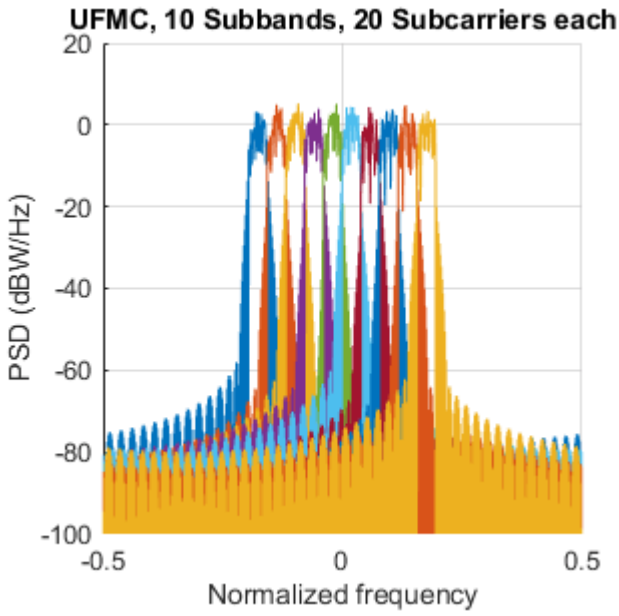
The full band of subcarriers (*N*) is divided into sub-bands. Each subband has a fixed number of subcarriers and not all sub-bands need to be employed for a given transmission. An *N*-pt IFFT for each subband is computed, inserting zeros for the unallocated carriers. Each subband is filtered by a filter of length *L*, and the responses from the different sub-bands are summed. The filtering is done to reduce the out-of-band spectral emissions. Different filters per subband can be applied, however, in this example, the same filter is used for each subband. A Chebyshev window with parameterized side-lobe attenuation is employed to filter the IFFT output per subband [1 on page 8-0].

The transmit-end processing is shown in the following diagram.




```
% Design window with specified attenuation
prototypeFilter = chebwin(filterLen, slobeAtten);

% Transmit-end processing
%  Initialize arrays
inpData = zeros(bitsPerSubCarrier*subbandSize, numSubbands);
txSig = complex(zeros(numFFT+filterLen-1, 1));

hFig = figure;
axis([-0.5 0.5 -100 20]);
hold on;
grid on

xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['UFMC, ' num2str(numSubbands) ' Subbands, '  ...
    num2str(subbandSize) ' Subcarriers each'])

%  Loop over each subband
for bandIdx = 1:numSubbands

    bitsIn = randi([0 1], bitsPerSubCarrier*subbandSize, 1);
    % QAM Symbol mapper
    symbolsIn = qammod(bitsIn, 2^bitsPerSubCarrier, 'InputType', 'bit', ...
    'UnitAveragePower', true);
    inpData(:,bandIdx) = bitsIn; % log bits for comparison

    % Pack subband data into an OFDM symbol
```

```
    offset = subbandOffset+(bandIdx-1)*subbandSize;
    symbolsInOFDM = [zeros(offset,1); symbolsIn; ...
                     zeros(numFFT-offset-subbandSize, 1)];
    ifftOut = ifft(ifftshift(symbolsInOFDM));

    % Filter for each subband is shifted in frequency
    bandFilter = prototypeFilter.*exp( 1i*2*pi*(0:filterLen-1)'/numFFT* ...
                 ((bandIdx-1/2)*subbandSize+0.5+subbandOffset+numFFT/2) );
    filterOut = conv(bandFilter,ifftOut);

    % Plot power spectral density (PSD) per subband
    [psd,f] = periodogram(filterOut, rectwin(length(filterOut)), ...
                          numFFT*2, 1, 'centered');
    plot(f,10*log10(psd));

    % Sum the filtered subband responses to form the aggregate transmit
    % signal
    txSig = txSig + filterOut;
end
set(hFig, 'Position', figposition([20 50 25 30]));
hold off;
```



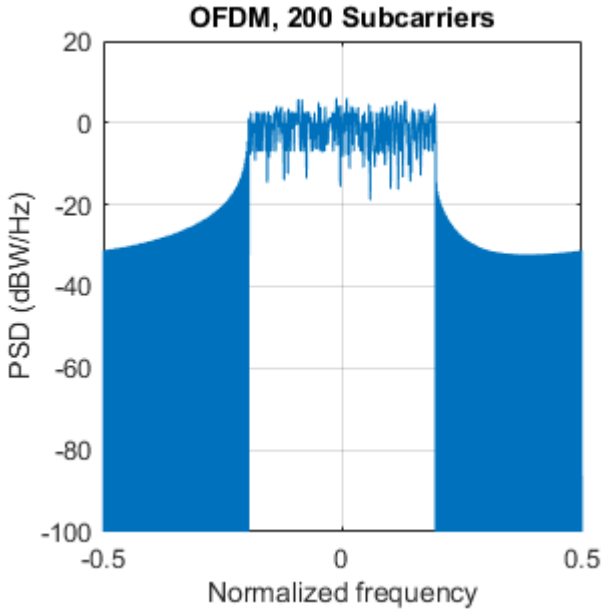

```
% Compute peak-to-average-power ratio (PAPR)
PAPR = comm.CCDF('PAPROutputPort', true, 'PowerUnits', 'dBW');
[~,~,paprUFMC] = PAPR(txSig);
disp(['Peak-to-Average-Power-Ratio (PAPR) for UFMC = ' num2str(paprUFMC) ' dB']);
```

```
Peak-to-Average-Power-Ratio (PAPR) for UFMC = 8.2379 dB
```

OFDM Modulation with Corresponding Parameters

For comparison, we review the existing OFDM modulation technique, using the full occupied band, however, without a cyclic prefix.

```
symbolsIn = qammod(inpData(:), 2^bitsPerSubCarrier, 'InputType', 'bit', ...
    'UnitAveragePower', true);

% Process all sub-bands together
offset = subbandOffset;
symbolsInOFDM = [zeros(offset, 1); symbolsIn; ...
                 zeros(numFFT-offset-subbandSize*numSubbands, 1)];
ifftOut = sqrt(numFFT).*ifft(ifftshift(symbolsInOFDM));

% Plot power spectral density (PSD) over all subcarriers
[psd,f] = periodogram(ifftOut, rectwin(length(ifftOut)), numFFT*2, ...
                      1, 'centered');
hFig1 = figure;
plot(f,10*log10(psd));
grid on
axis([-0.5 0.5 -100 20]);
xlabel('Normalized frequency');
ylabel('PSD (dBW/Hz)')
title(['OFDM, ' num2str(numSubbands*subbandSize) ' Subcarriers'])
set(hFig1, 'Position', figposition([46 50 25 30]));
```




```
% Compute peak-to-average-power ratio (PAPR)
PAPR2 = comm.CCDF('PAPROutputPort', true, 'PowerUnits', 'dBW');
[~,~,paprOFDM] = PAPR2(ifftOut);
disp(['Peak-to-Average-Power-Ratio (PAPR) for OFDM = ' num2str(paprOFDM) ' dB']);
```

```
Peak-to-Average-Power-Ratio (PAPR) for OFDM = 8.8843 dB
```

Comparing the plots of the spectral densities for OFDM and UFMC schemes, UFMC has lower side-lobes. This allows a higher utilization of the allocated spectrum, leading to increased spectral efficiency. UFMC also shows a slightly better PAPR.

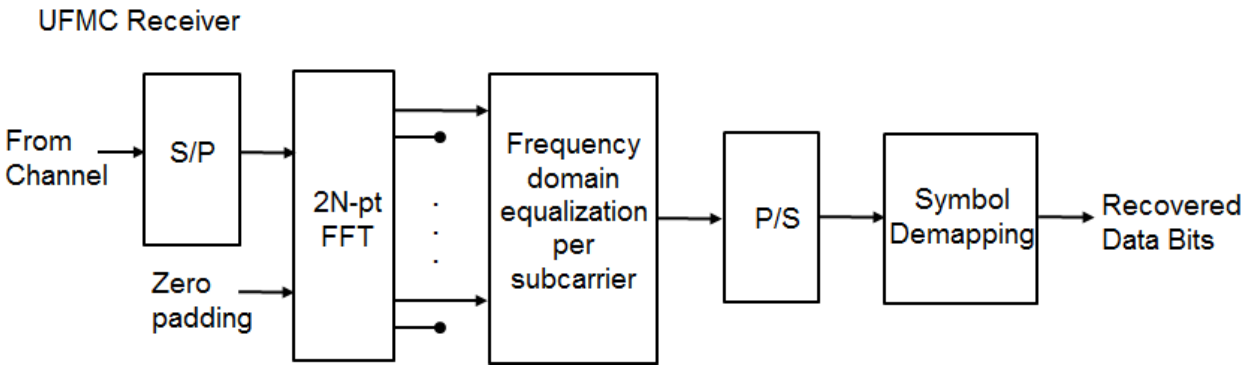
UFMC Receiver with No Channel

The example next highlights the basic UFMC receive processing, which, like OFDM, is FFT-based. The subband filtering extends the receive time window to the next power-of-two length for the FFT operation. Every alternate frequency value corresponds to a subcarrier main lobe. In typical scenarios, per-subcarrier equalization is used for equalizing the joint effect of the channel and the subband filtering.

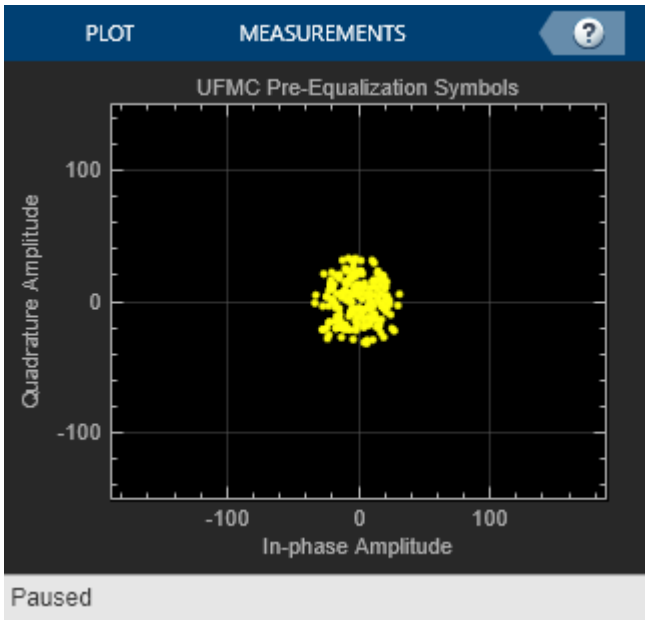
In this example, only the subband filter is equalized because no channel effects are modeled. Noise is added to the received signal to achieve the desired SNR.

```
% Add WGN
rxSig = awgn(txSig, snrdB, 'measured');
```

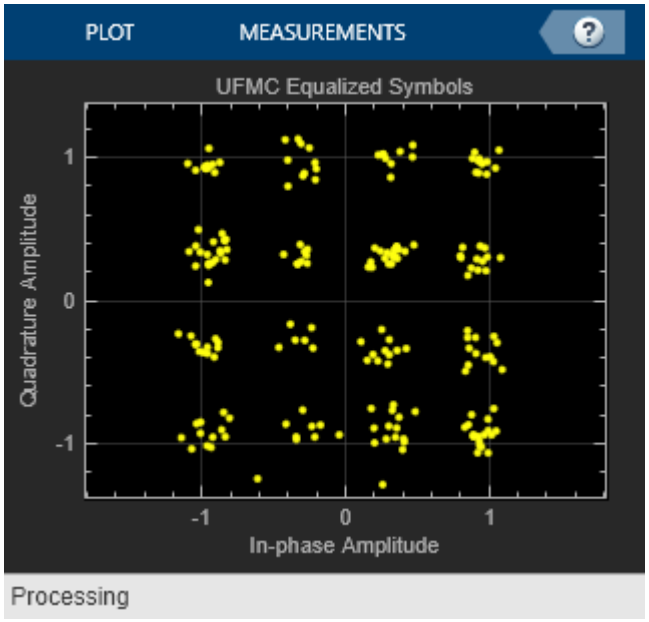
The receive-end processing is shown in the following diagram.




```
% Pad receive vector to twice the FFT Length (note use of txSig as input)
%   No windowing or additional filtering adopted
yRxPadded = [rxSig; zeros(2*numFFT-numel(txSig),1)];

% Perform FFT and downsample by 2
RxSymbols2x = fftshift(fft(yRxPadded));
RxSymbols = RxSymbols2x(1:2:end);

% Select data subcarriers
dataRxSymbols = RxSymbols(subbandOffset+(1:numSubbands*subbandSize));

% Plot received symbols constellation
constDiagRx = comm.ConstellationDiagram('ShowReferenceConstellation', ...
    false, 'Position', figposition([20 15 25 30]), ...
    'Title', 'UFMC Pre-Equalization Symbols', ...
    'Name', 'UFMC Reception', ...
    'XLimits', [-150 150], 'YLimits', [-150 150]);
constDiagRx(dataRxSymbols);
```




```
% Use zero-forcing equalizer after OFDM demodulation
rxf = [prototypeFilter.*exp(1i*2*pi*0.5*(0:filterLen-1)'/numFFT); ...
       zeros(numFFT-filterLen,1)];
prototypeFilterFreq = fftshift(fft(rxf));
prototypeFilterInv = 1./prototypeFilterFreq(numFFT/2-subbandSize/2+(1:subbandSize));

% Equalize per subband - undo the filter distortion
dataRxSymbolsMat = reshape(dataRxSymbols,subbandSize,numSubbands);
EqualizedRxSymbolsMat = bsxfun(@times,dataRxSymbolsMat,prototypeFilterInv);
EqualizedRxSymbols = EqualizedRxSymbolsMat(:);

% Plot equalized symbols constellation
constDiagEq = comm.ConstellationDiagram('ShowReferenceConstellation', ...
    false, 'Position', figposition([46 15 25 30]), ...
    'Title', 'UFMC Equalized Symbols', ...
    'Name', 'UFMC Equalization');
constDiagEq(EqualizedRxSymbols);
```




```
% BER computation
BER = comm.ErrorRate;

% Perform hard decision and measure errors
rxBits = qamdemod(EqualizedRxSymbols, 2^bitsPerSubCarrier, 'OutputType', 'bit', ...
    'UnitAveragePower', true);
ber = BER(inpData(:), rxBits);

disp(['UFMC Reception, BER = ' num2str(ber(1)) ' at SNR = ' ...
    num2str(snrdB) ' dB']);
```

```
UFMC Reception, BER = 0 at SNR = 15 dB
```

```
% Restore RNG state
rng(s);
```

Conclusion and Further Exploration

The example presents the basic characteristics of the UFMC modulation scheme at both transmit and receive ends of a communication system. Explore different system parameter values for the number of sub-bands, number of subcarriers per subband, filter length, side-lobe attenuation, and SNR.

Refer to “FBMC vs. OFDM Modulation” on page 8-73 for an example that describes the Filter Bank Multi-Carrier (FBMC) modulation scheme. The “F-OFDM vs. OFDM Modulation” on page 8-79 example describes the Filtered-OFDM modulation scheme.

UFMC is considered advantageous in comparison to OFDM by offering higher spectral efficiency. Subband filtering has the benefit of reducing the guards between sub-bands and also reducing the filter length, which makes this scheme attractive for short bursts. The latter property also makes it attractive in comparison to FBMC, which suffers from much longer filter length.

Selected Bibliography

1 Schaich, F., Wild, T., Chen, Y., "Waveform Contenders for 5G - Suitability for Short Packet and Low Latency Transmissions", Vehicular Technology Conference, pp. 1-5, 2014.

2 Wild, T., Schaich, F., Chen Y., "5G air interface design based on Universal Filtered (UF-)OFDM ", Proc. of 19th International Conf. on Digital Signal Processing, pp. 699-704, 2014.

P25 Spectrum Sensing with Synthesized and Captured Data

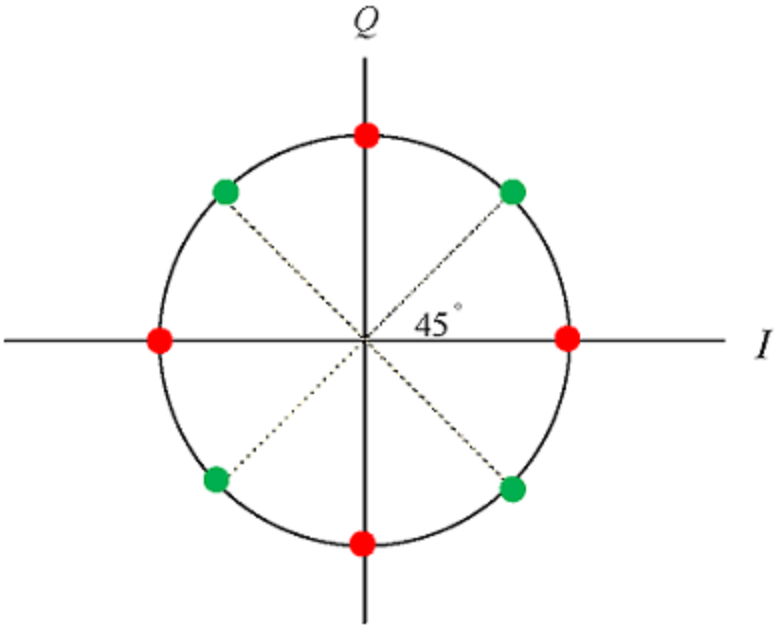
This example shows how to use cyclostationary feature detection to distinguish signals with different modulation schemes, including P25 signals [1]. It defines four cases of signals: noise only, C4FM, CQPSK, and one arbitrary type. The example applies the detection algorithm to signals with different SNR values, and to a captured real-world P25 signal, and then classifies the signals as one of the four types. Graphical results show that the detection algorithm succeeds in all the cases.

Project 25 (P25)

Project 25 (P25 or APCO-25) is a suite of standards for digital radio communications for use by federal, state, province and local public safety agencies in North America. When emergencies arise, this protocol suite enables communication among government agencies and mutual aid response teams. In this regard, P25 fills the same role as the European Terrestrial Trunked Radio (TETRA) [2] protocol, although the two standards are not interoperable with each other. In North America, P25 is widely used in public safety, security, public service, and commercial applications [1].

Project 25 is deployed in two phases. In Phase 1, P25 uses C4FM, an acronym for compatible 4 level frequency modulation. In its simplest form, it is a special type of 4FSK modulation, which uses four different frequencies to represent symbols. Phase 1 uses this modulation scheme to transmit digital information over a 12.5 kHz channel.

Phase 2 transmits digital information over a 6.25 kHz channel using the compatible quadrature phase shift keying (CQPSK) modulation format. CQPSK modulation is essentially pi/4 differential quadrature phase shift keying (pi/4 DQPSK), where encoding is symmetric, using phase change values of -135 degrees, -45 degrees, +45 degrees and +135 degrees, as shown in the following figure.



In this figure, the next state of the red dots can only be green dots, and vice versa. Although the data rate and bits per symbol are identical, the main difference between the two modulation schemes is that C4FM uses a frequency shift to depict a symbol, which provides a fixed amplitude signal. In contrast, CQPSK, uses a phase shift to depict a symbol, which imparts an amplitude component to the signal.

Cyclostationary Feature Detection

Modulation recognition and signal classification has been a subject of considerable research for over two decades. Classification schemes can generally be separated into one of two broad categories: likelihood-based (LB) approaches and feature-based (FB) approaches [3]. Cyclostationary feature detection is an FB technique based on the fact that communications signals are not accurately described as stationary, but rather more appropriately modeled as cyclostationary [4].

A cyclostationary process is a signal having statistical properties that vary cyclically with time [5]. These periodicities occur for signals in well defined manners due to processes such as sampling, scanning, modulating, multiplexing, and coding. This resulting periodic nature of signals can be exploited to determine the modulation scheme of the unknown signal [4].

Cyclostationary feature detection is a robust spectrum sensing technique because modulated information is a cyclostationary process, while noise is not. As a result, cyclic detectors can successfully operate even in low SNR environments.

Noise-only Case

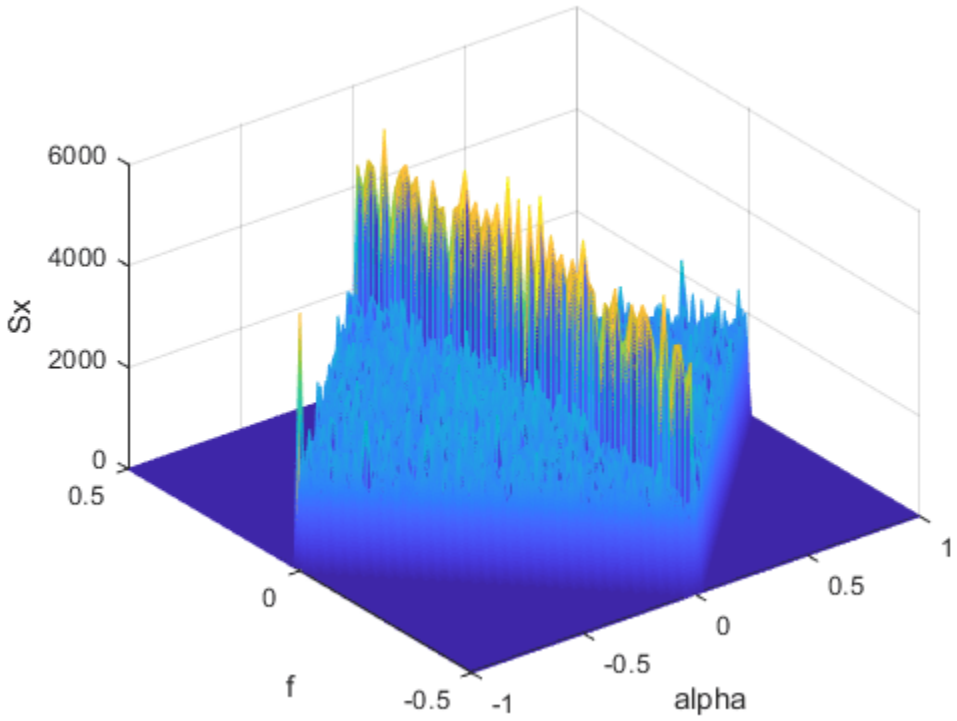
For the noise-only case, generate a (4*N)-by-1 vector of white Gaussian noise with a power of 1 dBW. 1/(4*N) is the cyclic resolution used to calculate the spectral autocorrelation function (SAF) in commP25ssca.m.

```
N = 4096;
input = wgn(4*N,1,1);
```

Use the time domain spectral autocorrelation function to analyze the cyclostationary features of the signal x(t). Run the spectral autocorrelation function commP25ssca.m on the input signal. This function estimates the ideal spectral autocorrelation function using the strip spectrum correlation algorithm (SSCA) [3] temporal smoothing method. It is an FFT based time smoothing algorithm. Refer to [6] for more information about the implementation of this algorithm.

Run the plot function commP25plot.m. This step illustrates the spectral autocorrelation function, which is a three-dimensional figure. Its x-axis represents the cyclic frequency (alpha) from -1 to 1. Its y-axis represents the spectral frequency (f) from -0.5 to 0.5, and its z-axis (Sx) represents the corresponding magnitude of the spectral autocorrelation function for each (alpha , f) pair. Cyclic resolution dalpha = 1/T, where T is the observation time of the data. Spectral resolution df = 1/Tw, where Tw is the window time to calculate the complex demodulate [7]. Since T > Tw, dalpha < df. Note that when alpha does not equal zero, the SAF values are approximately zero.

```
% 64 represents the window time Tw, 4*N represents the observation time T
[Sx,alphao,fo] = commP25ssca(input,1,1/64,1/(4*N));
fig1 = figure('Position',figposition([5 40 40 40]));
commP25plot(Sx,alphao,fo);
```



commP25decision_noise.m determines if the input signal contains only noise. commP25decision_c4fm.m determines if the input signal is a C4FM signal. And commP25decision_cqpsk.m determines if the input signal is a CQPSK signal. These decisions are based upon the location of the peaks in the SAF. In this example, the code correctly concludes that there is no P25 signal present.

```
[c,d] = size(Sx);
[Ades,Index] = sort(Sx(:),'descend');   % sort Sx by its element and store in Ades
[Ridx,Cidx]  = ind2sub(size(Sx),Index); % corresponding row index and column index
leng = length(Ades);

noise_decision = commP25decision_noise(Ades,Ridx,Cidx,leng,c,d);
if noise_decision == 0
    c4fm_decision = commP25decision_c4fm(Ades,Ridx,leng,c);
    if c4fm_decision == 0
        commP25decision_cqpsk(Ades,Ridx,Cidx,leng,c,d);
    end
end
```

```
There is no P25 signal.
```

C4FM Case with Synthesized Data

According to [8], the following modulation structure generates a C4FM output signal.

A normal raised cosine filter, which satisfies the Nyquist pulse shaping criterion, minimizes intersymbol interference. The parameters of the raised cosine filter are chosen per the filter's specifications in [8]. Specifically, this raised cosine filter has an upsampling factor of 4, and a roll-off factor of 0.2. The C4FM standard also calls for an inverse sinc filter after the raised cosine filter, to compensate for the sinc response of a P25 receiver integrate and dump filter. The FM modulator has a deviation of 600 Hz.

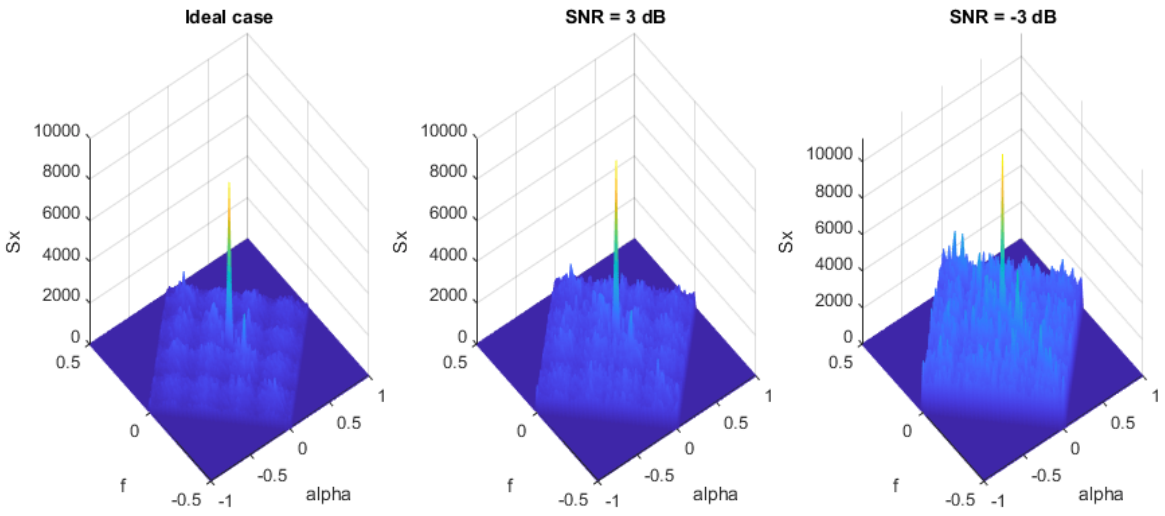
To observe the effects of noise on the design decisions, run the detection at SNR values of -3 dB, 3 dB and infinity dB.

```
% The length of input bits is N. The length of the output bits must also be
% N
x = randi([0,3],N,1);
sym = 2*x-3;                    % integer input

% Raised Cosine Filter
sampsPerSym = 4;                % Upsampling factor
% Design raised cosine filter with given order in symbols. Apply gain to
% the unit energy filter to obtain max amplitude of 1.
rctFilt = comm.RaisedCosineTransmitFilter(...
    'Shape', 'Normal', ...
    'RolloffFactor', 0.2, ...
    'OutputSamplesPerSymbol', sampsPerSym, ...
    'FilterSpanInSymbols', 60, ...
    'Gain', 1.9493);
c4fm_init = rctFilt(sym);

shape2 = 'Inverse-sinc Lowpass';
d2 = fdesign.interpolator(2, shape2);
intrpltr = design(d2, 'SystemObject', true);
c4fm_init = intrpltr(c4fm_init);

% Baseband Frequency Modulator
Fs = 4800;
freqdev = 600;
int_x = cumsum(c4fm_init)/Fs;
c4fm_output = exp(1i*2*pi*freqdev*int_x);
y = c4fm_output(1:N); % Ideal case, SNR = infinity
y1 = awgn(y,3);       % SNR = 3 dB
y2 = awgn(y,-3);      % SNR = -3 dB
```

The corresponding spectral autocorrelation functions are calculated and plotted. Note that the SAF peaks become more indistinct as the SNR decreases.

```
[Sx0,alphao0,fo0] = commP25ssca(y,1,1/64,1/(4*N));
[Sx1,alphao1,fo1] = commP25ssca(y1,1,1/64,1/(4*N));
[Sx2,alphao2,fo2] = commP25ssca(y2,1,1/64,1/(4*N));
fig2 = figure('Position',figposition([5 40 80 40]));
subplot(131);
commP25plot(Sx0,alphao0,fo0);
```

```
title('Ideal case');
subplot(132);
commP25plot(Sx1,alphao1,fo1);
title('SNR = 3 dB');
subplot(133);
commP25plot(Sx2,alphao2,fo2);
title('SNR = -3 dB');
```



This section follows the same procedures as in the previous one and obtains the classification results for each SNR value. The function commP25decision.m performs spectrum sensing classification for all possible input signal types.

```
commP25decision(Sx0); % Ideal case
```

```
There is signal present. Checking for presence of C4FM.
This is C4FM.
```

```
commP25decision(Sx1); % SNR = 3 dB
```

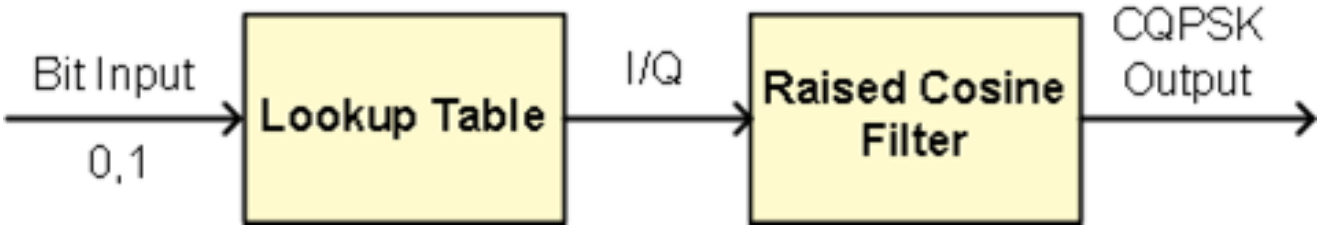
```
There is signal present. Checking for presence of C4FM.
This is C4FM.
```

```
commP25decision(Sx2); % SNR = -3 dB
```

```
There is signal present. Checking for presence of C4FM.
This is C4FM.
```

CQPSK Case with Synthesized Data

According to [8], the following modulation structure generates a CQPSK output signal.

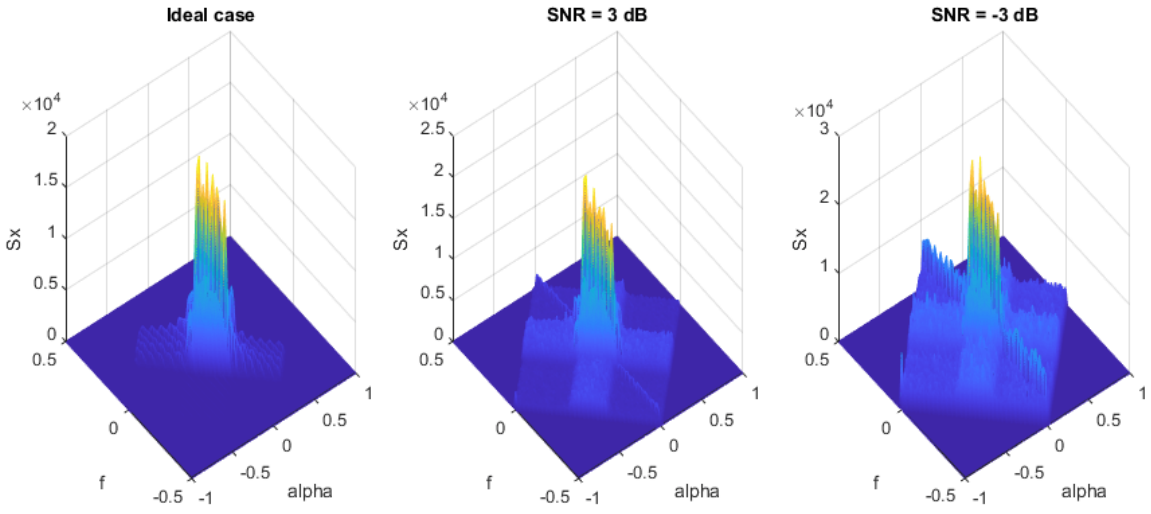


The CQPSK modulator consists of In Phase and Quadrature (I and Q) parts. The input bits are processed by the lookup table [8] to yield a 5-level I/Q signal. Because the specification of the lookup table is equivalent to pi/4 DQPSK, the example uses the DQPSK modulator System object™ to implement this lookup table. The I/Q signals are then filtered with the raised cosine filter described in the previous case.

```
% The size of input bits is 2*N, the size of output is 4*N
x = randi([0,1],2*N,1);

% Create a DQPSK modulator System object(TM) with bits as inputs, phase
% rotation of pi/4 and Gray-coded constellation
dqpskMod = comm.DQPSKModulator(pi/4,'BitInput',true);
% Modulate and filter
modout = dqpskMod(x);
release(rctFilt);
cqpsk_output = rctFilt(modout);
y = cqpsk_output;   % Ideal case, SNR = infinity
y1 = awgn(y,3);     % SNR = 3 dB
y2 = awgn(y,-3);    % SNR = -3 dB
```

Calculate and plot the corresponding spectral autocorrelation functions.

```
[Sx0,alphao0,fo0] = commP25ssca(y,1,1/64,1/(4*N));
[Sx1,alphao1,fo1] = commP25ssca(y1,1,1/64,1/(4*N));
[Sx2,alphao2,fo2] = commP25ssca(y2,1,1/64,1/(4*N));
fig3 = figure('Position',figposition([5 40 80 40]));
subplot(131);
commP25plot(Sx0,alphao0,fo0);
title('Ideal case');
subplot(132);
commP25plot(Sx1,alphao1,fo1);
title('SNR = 3 dB');
subplot(133);
commP25plot(Sx2,alphao2,fo2);
title('SNR = -3 dB');
```



The code outputs below show the results of CQPSK detection for three different SNR values.

```
commP25decision(Sx0); % Ideal case

There is signal present. Checking for presence of C4FM.
This is NOT C4FM. Checking for presence of CQPSK.
This is CQPSK.

commP25decision(Sx1); % SNR = 3 dB

There is signal present. Checking for presence of C4FM.
This is NOT C4FM. Checking for presence of CQPSK.
This is CQPSK.

commP25decision(Sx2); % SNR = -3 dB

There is signal present. Checking for presence of C4FM.
This is NOT C4FM. Checking for presence of CQPSK.
This is CQPSK.
```

Non-P25 Signal Case with Synthesized Data

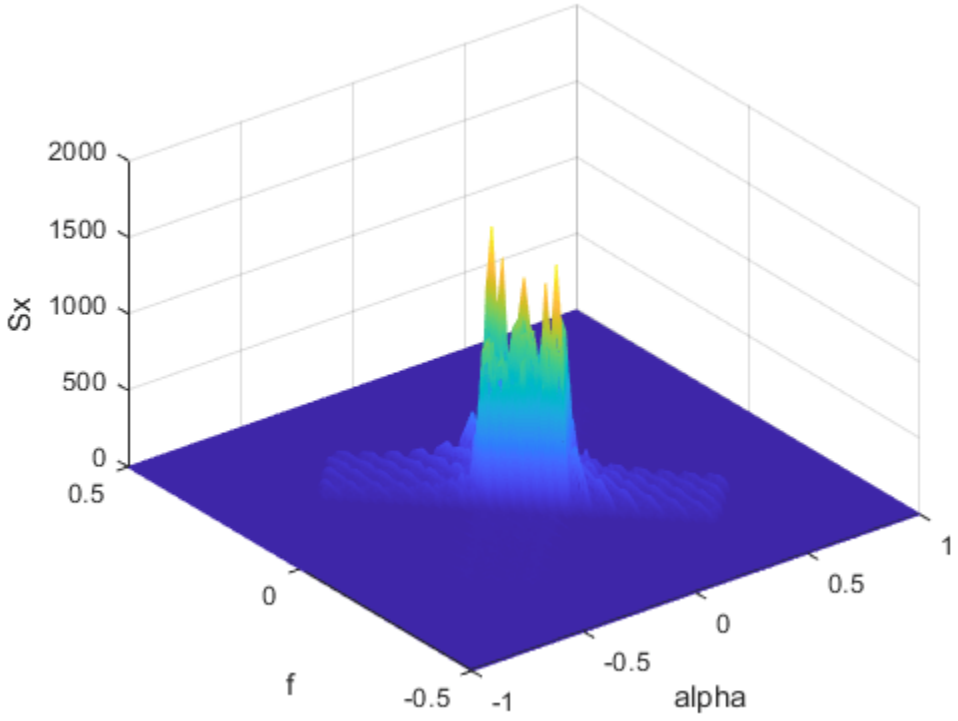
This case defines one arbitrary signal type, processes it with the P25 cyclostationary detector, and determines if it is a P25 signal.

Design an FIR equiripple lowpass filter, and apply it to a random input. Do not add any noise to the signal in this case. Try additional signal types, and let the cyclostationary feature detector classify them.

```
bcoeffs = firpm(200,[0 0.2 0.22 1],[1 1 0 0]); % Set N to achieve 40 dB rejection
input = randn(N,1);
y = filter(bcoeffs,1,input);
```

Then we calculate and plot the spectral autocorrelation function. Note that the different modulation characteristics of each signal yield significantly different SAFs.

```
[Sx,alphao,fo] = commP25ssca(y,1,1/64,1/(4*N));
fig4 = figure('Position',figposition([5 40 40 40]));
commP25plot(Sx,alphao,fo);
```



Follow the same procedures and obtain the classification result.

```
commP25decision(Sx);
```

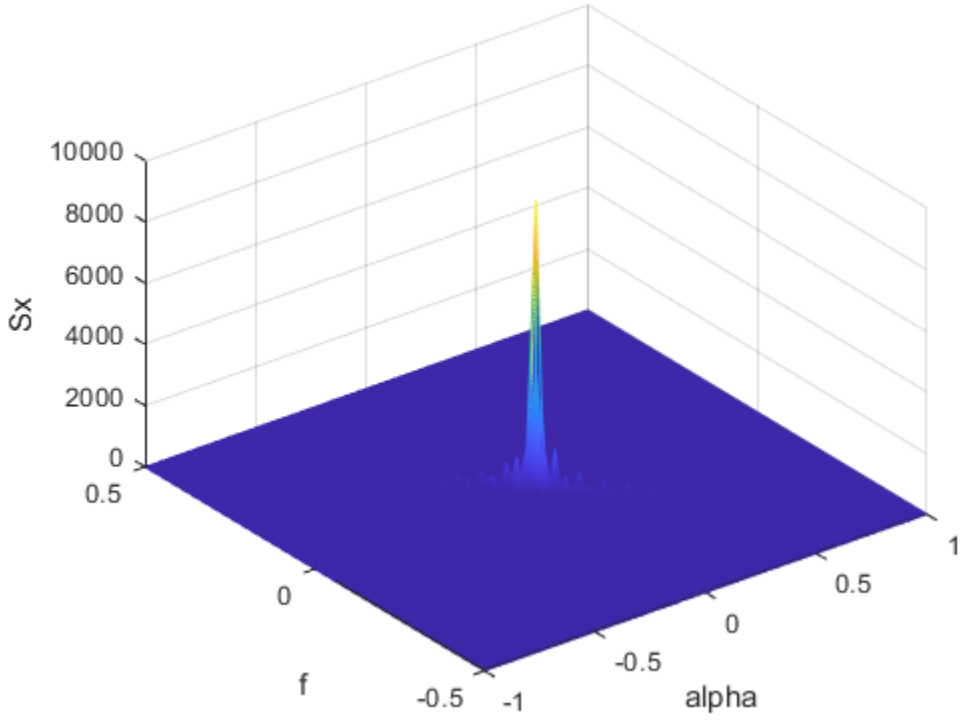
```
There is signal present. Checking for presence of C4FM.
This is NOT C4FM. Checking for presence of CQPSK.
This is NOT CQPSK either, so it is not a P25 signal.
```

C4FM Case with Captured Data

This case applies the detection algorithm to a captured real-world C4FM signal. The signal was transmitted by a P25 radio at 446 MHz, received by a USRP™ radio, and then saved by MATLAB® in capturedc4fm.mat. Follow the same procedures and obtain the classification result.

```
load capturedc4fm.mat;
y = y(1:4*N);
agc = comm.AGC;
y = 0.1*agc(y);
[Sx,alphao,fo] = commP25ssca(y,1,1/64,1/(4*N));
fig5 = figure('Position',figposition([5 40 40 40]));
commP25plot(Sx,alphao,fo);
commP25decision(Sx);
```

```
There is signal present. Checking for presence of C4FM.
This is C4FM.
```



Conclusion

This example shows how to use cyclostationary feature detection to distinguish signals of different modulation schemes. The algorithm classifies the signals based on the location of the peaks in spectral autocorrelation function. Cyclostationary feature detection has advantages over some detectors, like the energy detector, due to its resilience to noise.

Appendix

This example uses the following scripts and helper functions:

- commP25ssca.m
- commP25plot.m
- commP25decision.m
- commP25decision_noise.m
- commP25decision_c4fm.m
- commP25decision_cqpsk.m

Selected Bibliography

1 P25 Technology Interest Group: http://www.project25.org/

2 TETRA <https://en.wikipedia.org/wiki/Terrestrial_Trunked_Radio>

3 E. C. Like, "Non-Cooperative Modulation Recognition Via Exploitation of Cyclic Statistics". MS Thesis. 2007

4 E. C. Like, V. D. Chakravarthy, P. Ratazzi, and Z. Wu, "Signal Classification in Fading Channels Using Cyclic Spectral Analysis", EURASIP Journal on Wireless Communications and Networking, Volume 2009, 2009.

5 W. A. Gardner, A. Napolitano and L. Paura, "Cyclostationarity: Half a century of research", Signal Processing, Vol. 86, No. 4, pp. 639-697, 2006.

6 E. L. Da Costa, "Detection and Identification of Cyclostationary Signals". MS Thesis. 1996.

7 Antonio F. Lima, Jr., "Analysis of Low Probability of Intercept (LPI) Radio Signals using Cyclostationary Processing". MS Thesis. 2002.

8 TIA Standard Project 25: <https://tiaonline.org/what-we-do/standards/>

LLR vs. Hard Decision Demodulation in Simulink

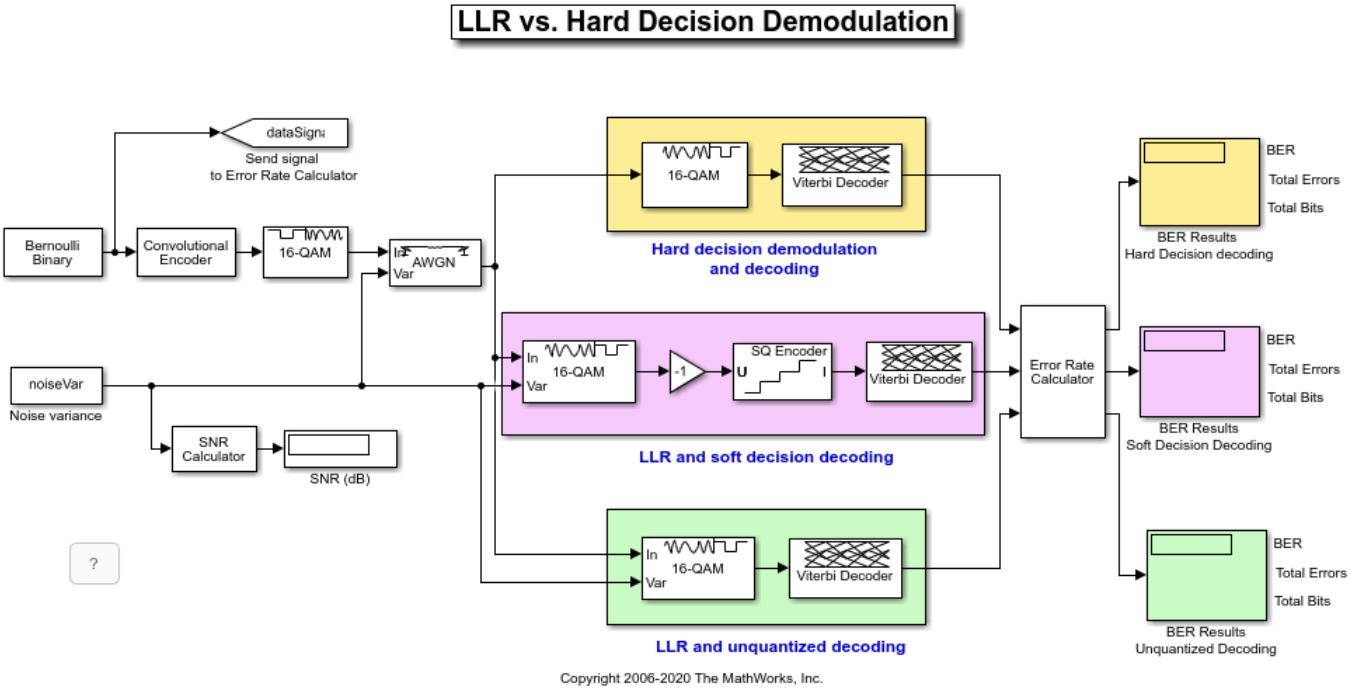
This model shows the improvement in BER performance when using log-likelihood ratio (LLR) instead of hard decision demodulation in a convolutionally coded communication link.

For a MATLAB™ version of this example, see “Log-Likelihood Ratio (LLR) Demodulation” on page 8-69.

System Setup

This example model simulates a convolutionally coded communication system having one transmitter, an AWGN channel and three receivers. The convolutional encoder has a code rate of 1/2. The system employs a 16-QAM modulation. The modulated signal passes through an additive white Gaussian noise channel. The top receiver performs hard decision demodulation in conjunction with a Viterbi decoder that is set up to perform hard decision decoding. The second receiver has the demodulator configured to compute log-likelihood ratios (LLRs) that are then quantized using a 3-bit quantizer. It is well known that the quantization levels are dependent on noise variance for optimum performance [2]. The exact boundaries of the quantizer are empirically determined here. A Viterbi decoder that is set up for soft decision decoding processes these quantized values. The LLR values computed by the demodulator are multiplied by -1 to map them to the right quantizer index for use with Viterbi Decoder. To compute the LLR, the demodulator must be given the variance of noise as seen at its input. The third receiver includes a demodulator that computes LLRs which are processed by a Viterbi decoder that is set up in unquantized mode. The BER performance of each receiver is computed and displayed.

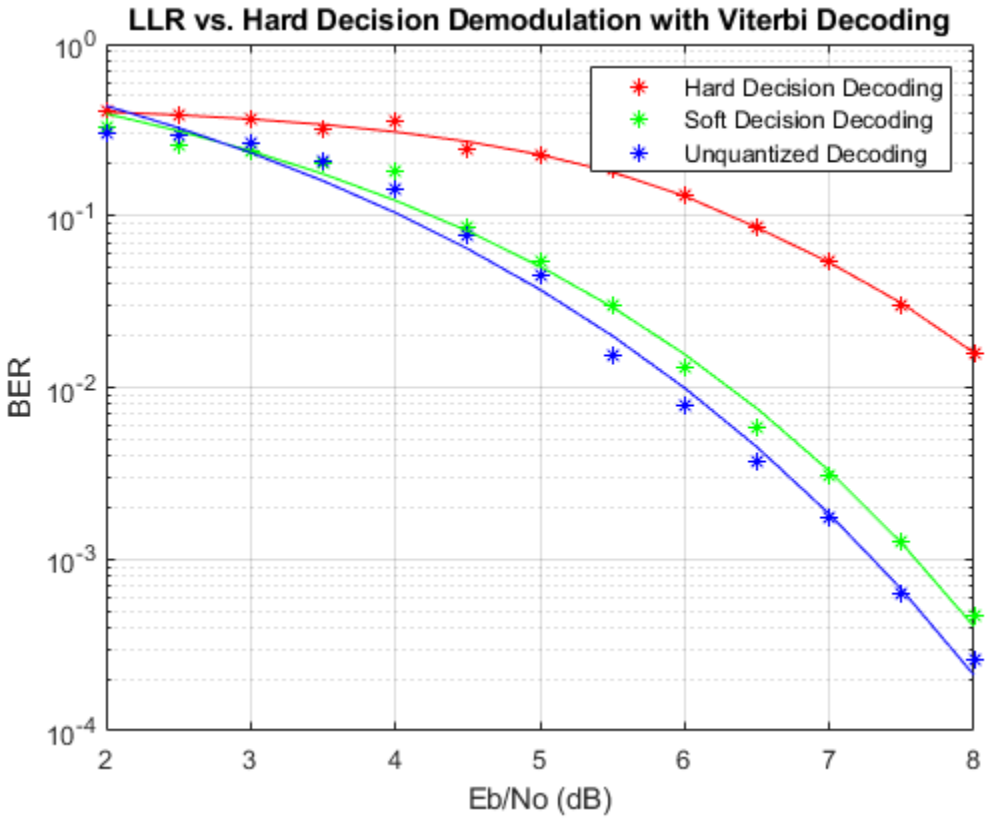
```
modelName = 'commLLRvsHD';
open_system(modelName);
```



System Simulation and Visualization

Simulate this system over a range of information bit Eb/No values. Adjust these Eb/No values for coded bits and multi-bit symbols to get noise variance values required for the AWGN block and Rectangular QAM Baseband Demodulator block. Collect BER results for each Eb/No value and visualize the results.

```
EbNo     = 2:0.5:8; % information rate Eb/No in dB
codeRate = 1/2;     % code rate of convolutional encoder
nBits    = 4;       % number of bits in a 16-QAM symbol
Pavg     = 10;      % average signal power of a 16-QAM modulated signal
snr      = EbNo - 10*log10(1/codeRate) + 10*log10(nBits); % SNR in dB
noiseVarVector = Pavg ./ (10.^(snr./10)); % noise variance

% Initialize variables for storing the BER results
ber_HD  = zeros(1,length(EbNo));
ber_SD  = zeros(1,length(EbNo));
ber_LLR = zeros(1, length(EbNo));

% Loop over all noiseVarVector values
for idx=1:length(noiseVarVector)
    noiseVar = noiseVarVector(idx); %#ok<NASGU>
    sim(modelName);
    % Collect BER results
    ber_HD(idx)  = BER_HD(1);
    ber_SD(idx)  = BER_SD(1);
    ber_LLR(idx) = BER_LLR(1);
end

% Perform curve fitting and plot the results
fitBER_HD  = real(berfit(EbNo,ber_HD));
fitBER_SD  = real(berfit(EbNo,ber_SD));
fitBER_LLR = real(berfit(EbNo,ber_LLR));
semilogy(EbNo,ber_HD,'r*', ...
    EbNo,ber_SD,'g*', ...
    EbNo,ber_LLR,'b*', ...
    EbNo,fitBER_HD,'r', ...
    EbNo,fitBER_SD,'g', ...
    EbNo,fitBER_LLR,'b');
legend('Hard Decision Decoding', ...
    'Soft Decision Decoding','Unquantized Decoding');
xlabel('Eb/No (dB)');
ylabel('BER');
title('LLR vs. Hard Decision Demodulation with Viterbi Decoding');
grid on;
```



To experiment with this system further, try different modulation types. This system uses a binary mapped modulation scheme for faster error collection but it is well known that Gray mapped signal constellation provides better BER performance. Experiment with various constellation ordering options in the modulator and demodulator blocks. Configure the demodulator block to compute approximate LLR to see the difference in the BER performance compared to hard decision demodulation and LLR. Try out a different range of Eb/No values. Finally, investigate different quantizer boundaries for your modulation scheme and Eb/No values.

Using Dataflow in Simulink

You can configure this example to use data-driven execution by setting the Domain parameter to dataflow for Dataflow Subsystem. With dataflow, blocks inside the domain, execute based on the availability of data as rather than the sample timing in Simulink. Simulink automatically partitions the system into concurrent threads. This autopartitioning accelerates simulation and increases data throughput. To learn more about dataflow and how to run this example using multiple threads, see “Multicore Simulation of Comparing Demodulation Types” on page 8-556.

```
% Cleanup
close_system(modelName,0);
clear modelName EbNo codeRate nBits Pavg snr noiseVarVector ...
    ber_HD ber_SD ber_LLR idx noiseVar fitBER_HD fitBER_SD fitBER_LLR;
```

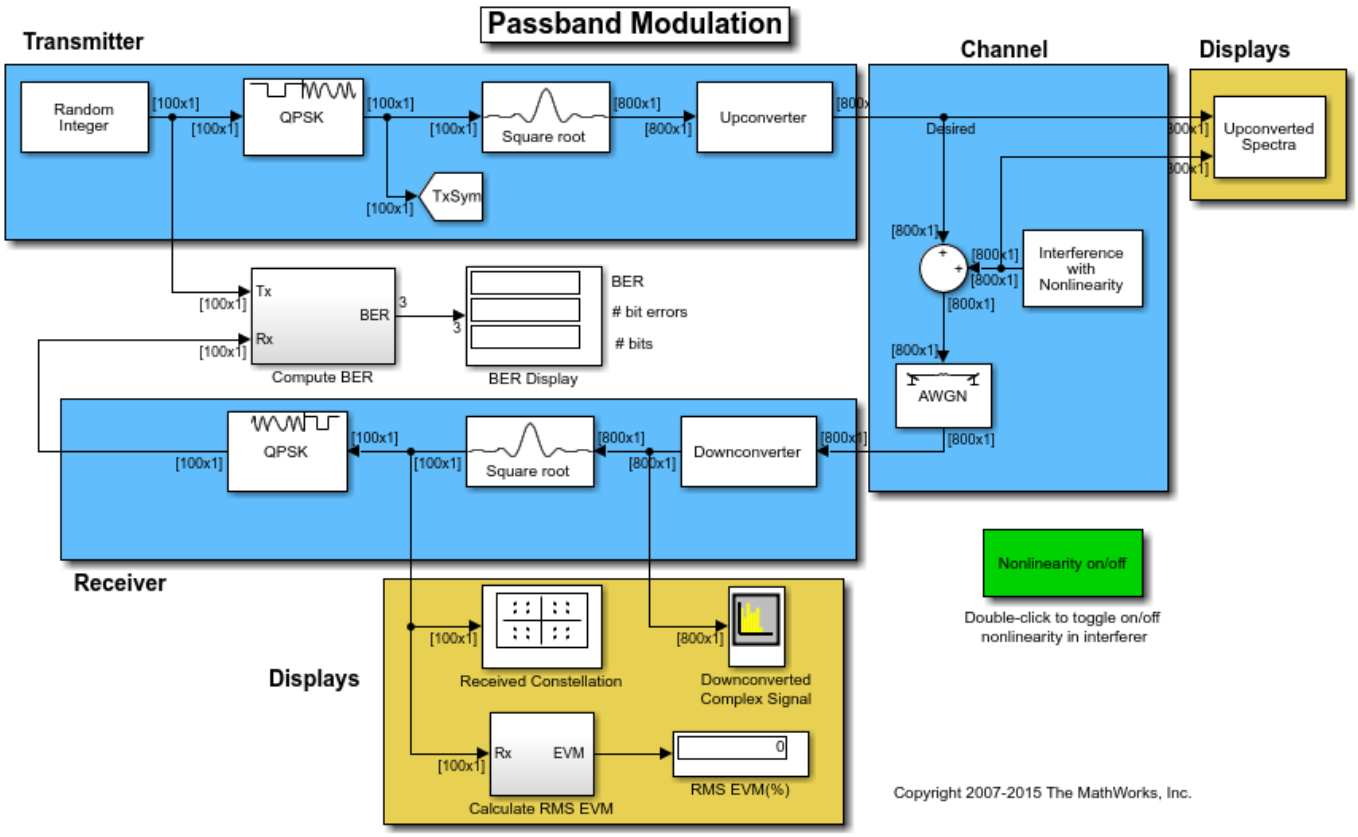
Selected Bibliography

[1] J. L. Massey, "Coding and Modulation in Digital Communications", Proc. Int. Zurich Seminar on Digital Communications, 1974

[2] J. A. Heller, I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication", IEEE® Trans. Comm. Tech. vol COM-19, October 1971

Passband Modulation

This model shows a straightforward way to perform passband modulation, by multiplying a modulated complex signal with a complex sine wave to perform frequency upconversion. In general, it is preferable to model a system at complex baseband. However, there are some circumstances where it is necessary to model the system at real passband. An example of this is when an adjacent band signal is processed with a nonlinearity, and causes interference in the desired band. This model also illustrates the effect of such interference.



Structure of the Example

The communications link in this model includes these components:

- A `Random Integer Generator` block, used as source of random data
- A modulator and a pulse shaping filter that perform QPSK modulation and root raised cosine pulse shaping.
- An `Upconverter` block that multiplies the modulated signal by a carrier frequency.
- A source of tone interference. The interference has a cubic nonlinearity which may be toggled on or off. When the nonlinearity is off, the interference falls completely out of band, but when on, the third harmonic of the tone is introduced into the desired band, causing co-channel interference.
- An `AWGN Channel` block, set to Eb/No mode. It specifies two bits per symbol because the modulation format is QPSK. The signal power is 1/(2*8) watts. This is because the original signal power at the modulator is 1 watt. The root-raised cosine filter upsamples the signal by a factor of

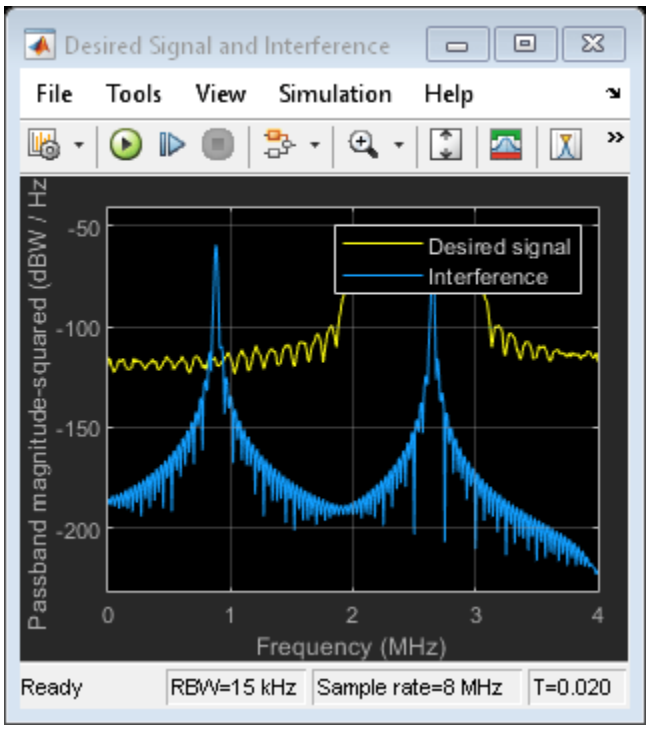
8, which reduces the power by that factor. The frequency upconversion block output takes only the real part of the signal, thereby reducing the power again, this time by a factor of 2. Finally, the symbol period is 1e-6 seconds, to match the original sample time on the `Random Integer Generator` source.

- A `Downconverter` block that converts the signal from real passband to complex baseband.
- A root raised cosine pulse shaping filter that decimates back to one sample per symbol, and a QPSK demodulator block.
- BER and RMS EVM metric calculation blocks.

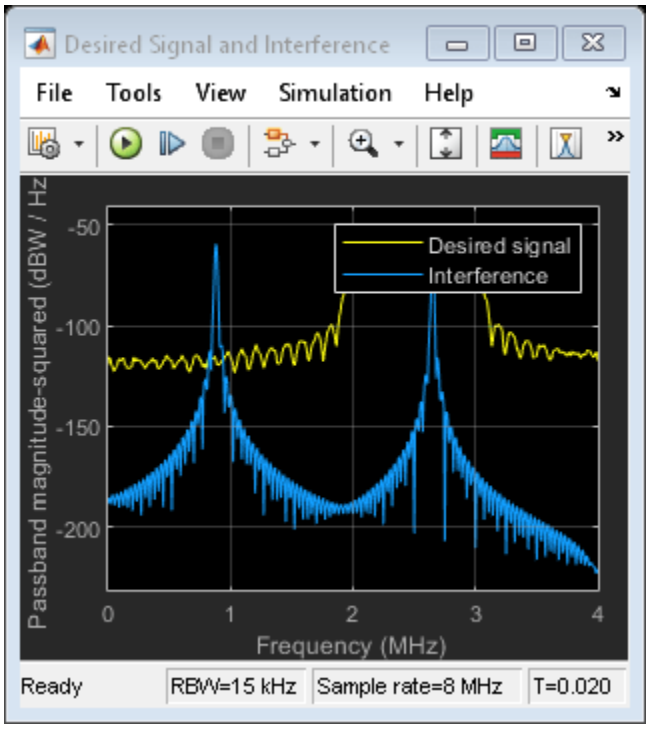
Results and Displays

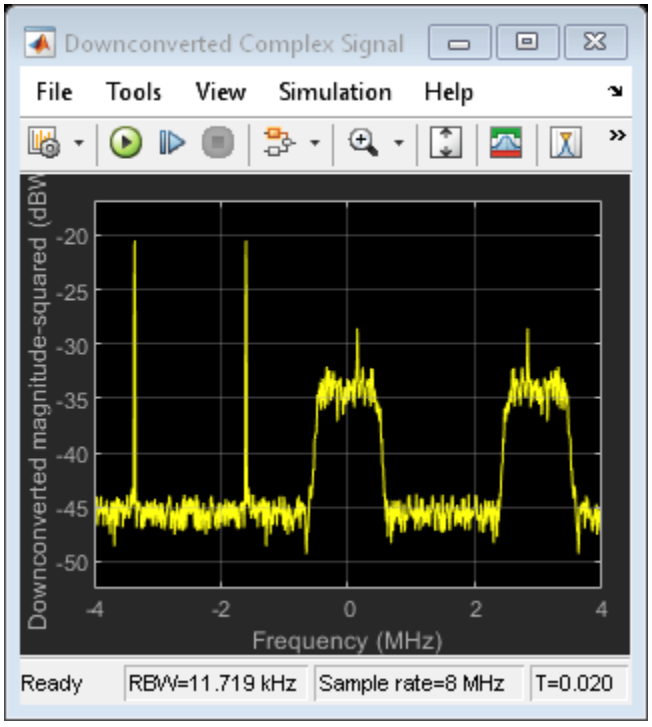
When the simulation runs, two spectrum analyzers and one scatter plot open.

The first spectrum analyzer shows the signal and the interference signal at passband. With the nonlinearity turned off, the spectrum of the tone interferer falls outside the bandwidth of the desired signal. With the cubic nonlinearity on, the third harmonic of the interference falls into the band of the desired signal.

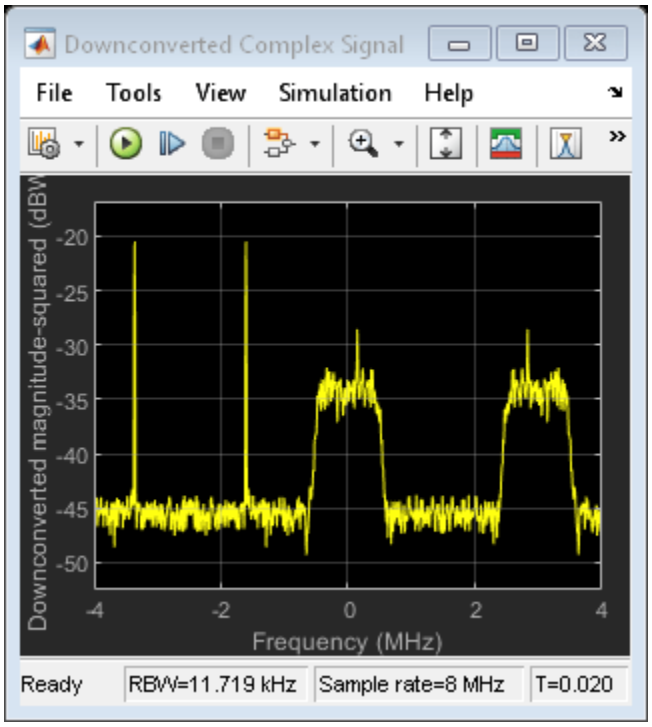


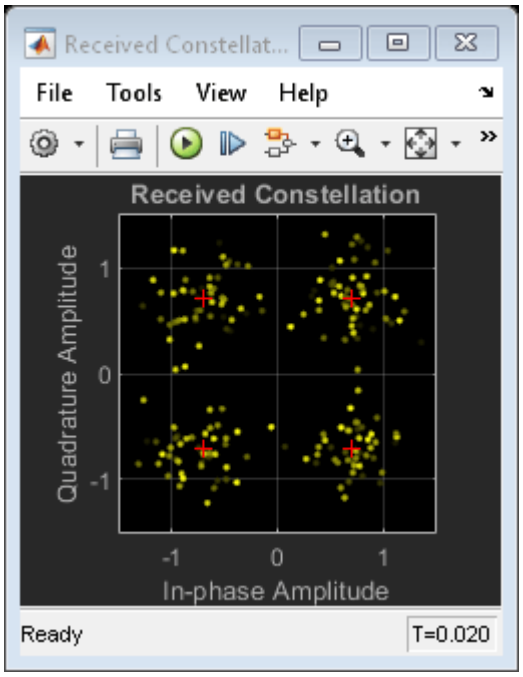
The second scope illustrates the signal after it has been downconverted back to baseband at the receiver, prior to the root raised cosine filtering. Note that with the nonlinearity on, you can see the interfering tone present with the baseband signal.

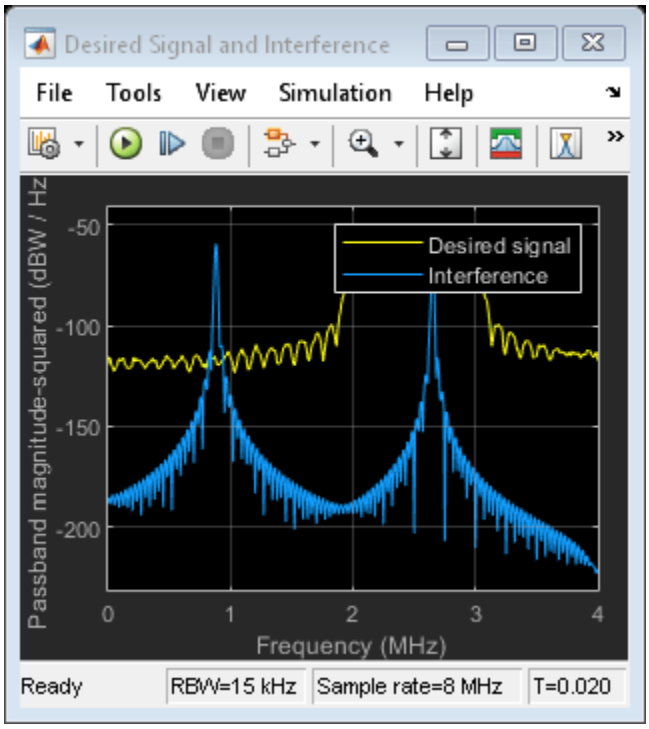




The third scope shows the scatter plot of the received signal, and by toggling the nonlinearity on and off, you can view the effect the interference has on the scatter plot. With the nonlinearity on, the signal constellation is more diffuse than when the nonlinearity is not present.


Downconverted Complex Signal
File Tools View Simulation Help
Downconverted magnitude-squared (dBW
-20
-25
-30
-35
-40
-45
-50
-4
-2
0
2
4
Frequency (MHz)
Ready
RBW=11.719 kHz
Sample rate=8 MHz
T=0.020


Received Constellat...
File Tools View Help
Received Constellation
Quadrature Amplitude
1
0
-1
-1
0
1
In-phase Amplitude
Ready
T=0.020



The model also contains two numerical displays. The first one displays the BER of the link. The BER calculation resets each time the nonlinearity is toggled on or off.

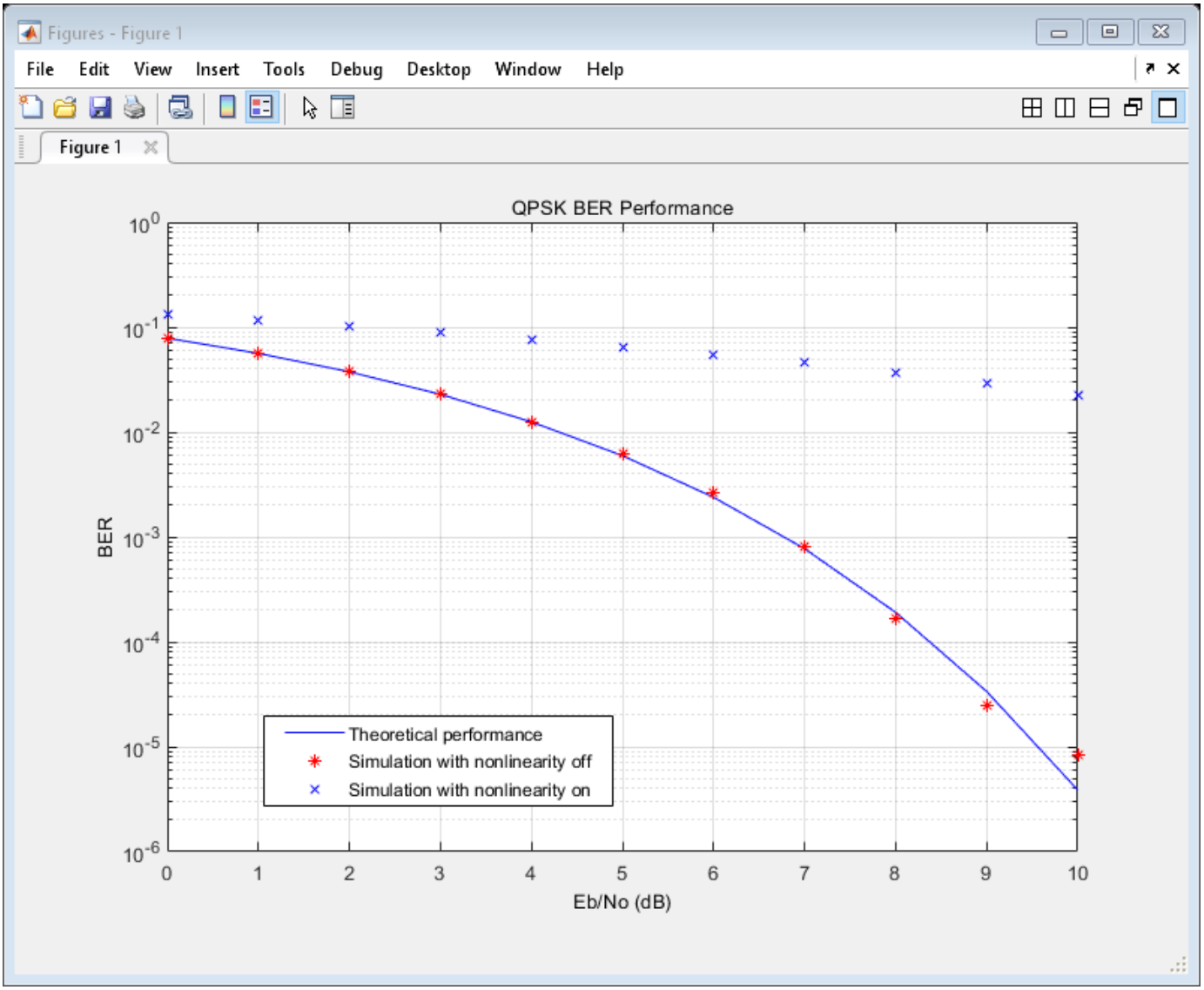
The second numerical display is the RMS Error Vector Magnitude (EVM) measured with the EVM Measurement block.

Experimenting with the Example

Double-click on the `Nonlinearity on/off` block to toggle the nonlinearity on the interference signal. Observe the changes this has on the received spectrum, constellation, BER and EVM.

By varying the Eb/No parameter, you can produce BER curves, and compare the results of the model with theoretical results. Note that the model achieves expected theoretical results[1] for QPSK with the nonlinearity off. Furthermore, you can see the effects the nonlinearity has on overall BER.

For further experimentation, try changing the value of the Eb/No parameter on the AWGN channel block, or changing the power of the interference signal. To change the power of the interference signal, open the `Interference with Nonlinearity` subsystem, and modify the gain value.



See Also

The `Downconverter` block uses a simple complex multiplication method to perform downconversion. You can find an example showing more efficient downconversion using IF subsampling at: “IF Subsampling with Complex Multirate Filters”.

Bibliography

1 Proakis, John G., *Digital Communications*, Fourth Ed., sec. 5.2.7, New York, McGraw-Hill, 2001.

256-Channel ADSL

This model shows part of the asymmetric digital subscriber line (ADSL) technology for transmitting data and multimedia information over telephone lines. It illustrates a downstream path from the central office to the end user. It incorporates the discrete multitone (DMT) signaling modulation technique.

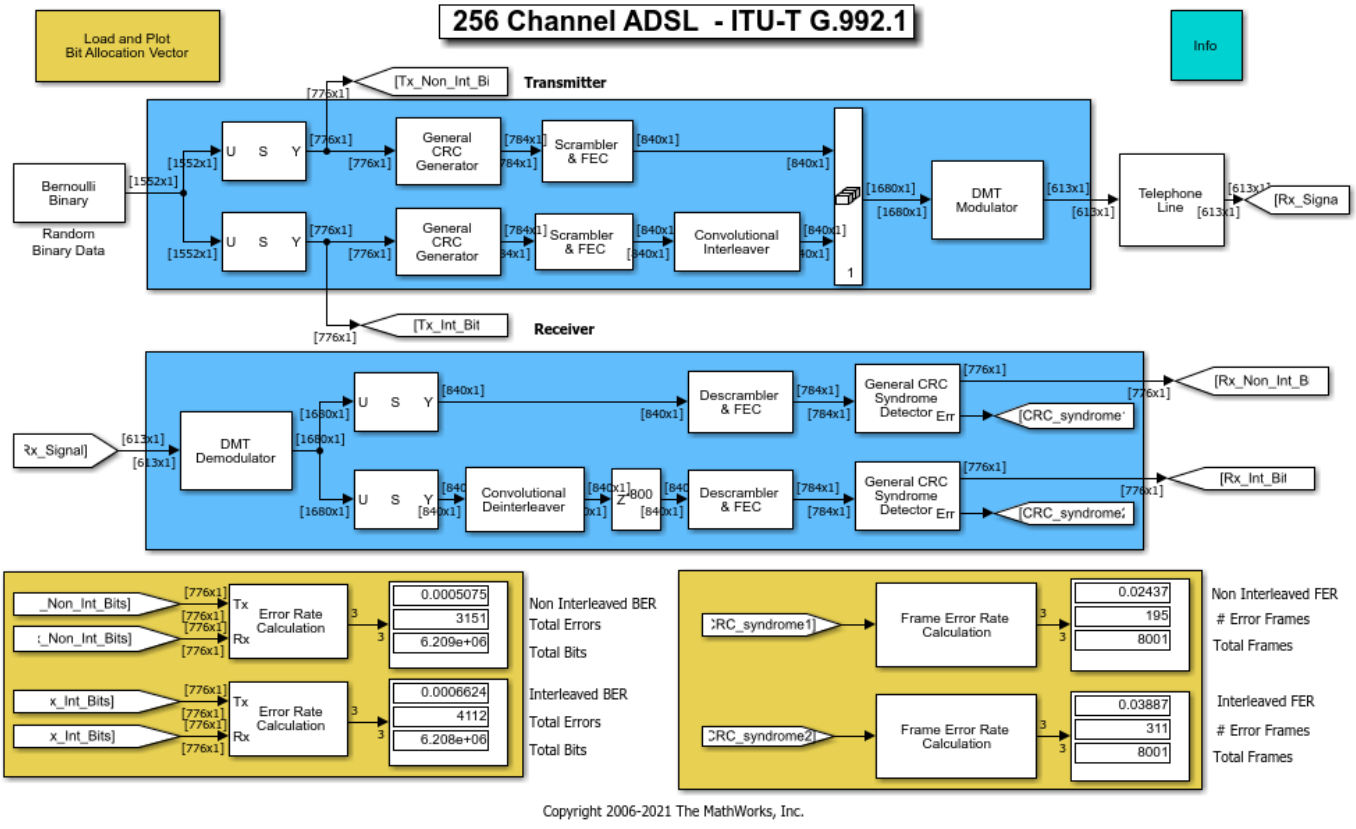
The DMT modulator and demodulator subsystems in the model have been updated to allow code reuse when generating code. These subsystems now generate only 10 unique reusable functions compared to the 256 chunks of code for each modulator/demodulator block generated earlier. This leads to shorter compile times and smaller executable sizes.

Structure of the Example

When the simulation is run, the model:

- Generates random binary data frames,
- Transmits the binary data frames according to the ADSL specification,
- Simulates a channel, specifically the telephone line, using an FIR filter of length 101 and the AWGN Channel block,
- Attempts to recover the transmitted information from the received data,
- Computes error statistics.

The model uses frame-based processing, thereby processing many bits in each time step. For more information, see “Sample- and Frame-Based Concepts”.



Transmitting Data

The transmitter portion of the model, shaded in blue at the top of the model, contains two parallel paths. One path (the fast buffer) processes the first 776 bits of each 1552-bit data frame, while the other path (the interleaved buffer) processes the last 776 bits of each data frame. Each path appends eight cyclic redundancy check (CRC) bits to its 776-bit frame, scrambles the bits, and encodes them using a shortened Reed-Solomon code. The scrambling and encoding operations interpret the bits as integers between 0 and 127. In the second path but not the first, a Convolutional Interleaver block interleaves the encoded data. This interleaving operation increases the second path's resistance to burst errors but also adds latency. Finally, the data from the two routes is concatenated and modulated. Data from the fast buffer is modulated to the low frequency subcarriers, while data from the interleaved buffer is modulated to the high frequency subcarriers, according to the bit allocation vector b. This example assumes that the bit allocation vector is known and uses the vector to calculate the channel. Click `commadsl;get_param('commadsl','ModelWorkspace');commandwindow` to see in the MATLAB® Command Window the calculations involved.

Processing Received Data

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

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

Results and Displays

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

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

Selected Bibliography

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

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

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

Simultaneous Simulation of Multiple Fading Channels with WINNER II Channel Model

This example shows how to set up a system with multiple base stations (BS), multiple mobile stations (MS) and multiple MIMO downlinks from one BS sector to one MS. You must download and install the WINNER II Channel Model for Communications Toolbox™ Add-On to run this example. Each link is assigned with a propagation scenario and condition. Fading channel coefficients for all links are generated simultaneously. An impulse signal is passed through the fading channel for each link. The received impulse and frequency responses are plotted for selected links.

Check for Support Package Installation

Check if the 'WINNER II Channel Model for Communications Toolbox' support package is installed.

```
commSupportPackageCheck('CST_WINNER2');
```

Antenna Array Inventory

In the WINNER II channel model, each BS is composed of one or more sectors, and each BS sector and MS is assigned with an antenna array. We need to first establish a set of arrays that are available for BS sectors and MS to employ, which we call the antenna array inventory.

In this example, all available antenna arrays are uniform circular array (UCA). There are four different UCAs in the inventory:

- 16 elements with a radius of 30cm
- 12 elements with a radius of 30cm
- 8 elements with a radius of 30cm
- 4 elements with a radius of 5cm

Each antenna element in the UCAs is omnidirectional.

```
s = rng(21); % For repeatability

AA(1) = winner2.AntennaArray('UCA',16,0.3);
AA(2) = winner2.AntennaArray('UCA',12,0.3);
AA(3) = winner2.AntennaArray('UCA',8,0.3);
AA(4) = winner2.AntennaArray('UCA',4,0.05);
```

Configure System Layout

On a 300-by-300 (meters) map, we will set up 3 BS, 5 MS, and 6 links. The first BS has one sector which is equipped with a 16-element UCA. The second BS also has one sector that is equipped with a 12-element UCA. The third BS has three sectors which are equipped with a 8-element UCA each. Each MS is assigned with a 4-element UCA.

```
BSIdx    = {1; 2; [3 3 3]}; % Index in antenna array inventory vector
MSIdx    = [4 4 4 4 4];     % Index in antenna array inventory vector
numLinks = 6;               % Number of links
range    = 300;             % Layout range (meters)
cfgLayout = winner2.layoutparset(MSIdx,BSIdx,numLinks,AA,range);
```

Six links are modeled in the system. The first BS connects to the first and second MSs. The second BS connects to the third MS. For the third BS, its first sector connects to the third and fourth MSs, its

second sector connects to the fifth MS, and its third sector does not connect to any MS. From MS perspective, each of them connects to one BS except for the third one, which connects to both the second and third BSs. Each link is assigned with one propagation scenario, chosen from B4 (outdoor to indoor), C2 (Urban macro-cell) and C4 (Urban macro outdoor to indoor). Non-line-of-sight (NLOS) is modeled for each link.

```
cfgLayout.Pairing = [1 1 2 3 3 4; 6 7 8 8 9 10];  % Index in cfgLayout.Stations
cfgLayout.ScenarioVector = [6 6 13 13 11 11];     % 6 for B4, 11 for C2 and 13 for C4
cfgLayout.PropagConditionVector = [0 0 0 0 0 0];  % 0 for NLOS
```

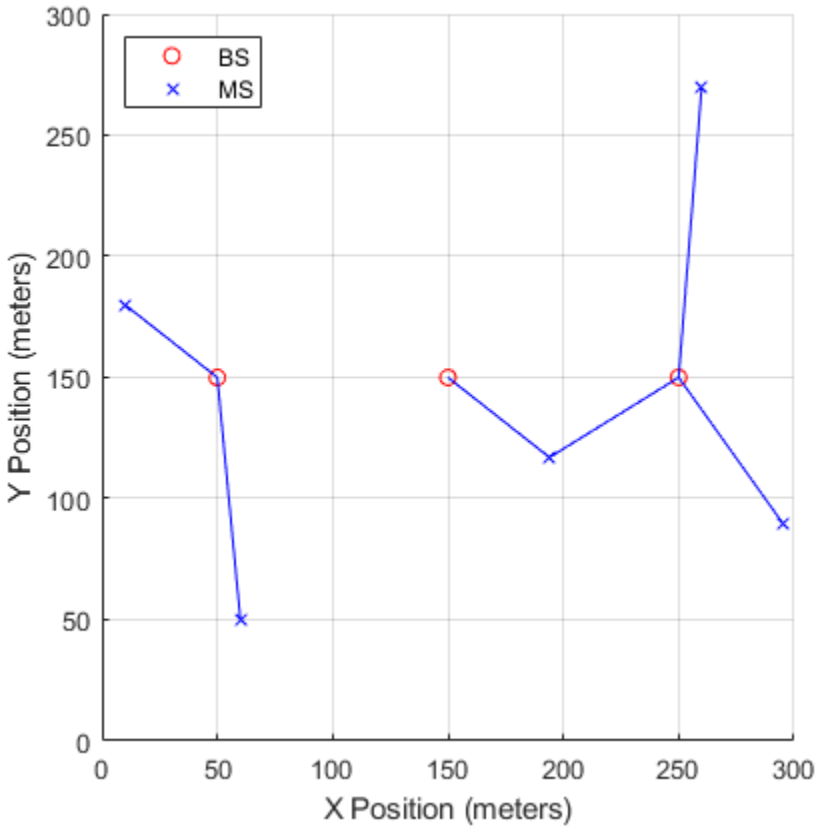
The three BSs are uniformly spaced between 0 and 300 on the x-axis and have the same position on the y-axis. MS positions are assigned to ensure that their distances to the connected BSs are in the valid path loss ranges for the corresponding scenarios. Specifically, the ranges for the B4, C2 and C4 scenarios are [3, 1000], [50, 5000] and [50, 5000] meters, respectively. By default, each BS sector is 32 meters high and MS is 1.5 meters high. Each MS is randomly assigned with a velocity which does not exceed 0.5 m/s in any of the X, Y and Z directions.

```
% Number of BS sectors and MSs in the system
numBSSect = sum(cfgLayout.NofSect);
numMS = length(MSIdx);

% Set up positions for BS sectors. Same position for the third, fourth and
% fifth sectors as they belong to one BS.
cfgLayout.Stations(1).Pos(1:2) = [50;  150];
cfgLayout.Stations(2).Pos(1:2) = [150; 150];
cfgLayout.Stations(3).Pos(1:2) = [250; 150];
cfgLayout.Stations(4).Pos(1:2) = [250; 150];
cfgLayout.Stations(5).Pos(1:2) = [250; 150];

% Set up MS positions
cfgLayout.Stations(6).Pos(1:2)  = [10;  180];  % 50m from 1st BS
cfgLayout.Stations(7).Pos(1:2)  = [60;  50];   % 111.8m from 1st BS
cfgLayout.Stations(8).Pos(1:2)  = [194; 117];  % 55m and 65m from 2nd and 3rd BSs respectively
cfgLayout.Stations(9).Pos(1:2)  = [260; 270];  % 120.4m from 3rd BS
cfgLayout.Stations(10).Pos(1:2) = [295; 90];   % 75m from 3rd BS

% Randomly draw MS velocity
for i = numBSSect + (1:numMS)
    cfgLayout.Stations(i).Velocity = rand(3,1) - 0.5;
end
```

To illustrate the system setup, we plot the BSs, the MSs, and the links between them on a 2-D map. In the plot, each BS sector is represented by a circle, each MS is represented by a cross, and each link is represented by a straight line between the corresponding BS and MS. As the third BS has three sectors, only three circles are shown on the map.

```
% Get all BS sector and MS positions
BSPos = cell2mat({cfgLayout.Stations(1:numBSSect).Pos});
MSPos = cell2mat({cfgLayout.Stations(numBSSect+1:end).Pos});

scrsz = get(groot,'ScreenSize');
figSize = min(scrsz([3,4]))/2.3;
figure('Position',[scrsz(3)*.5-figSize/2,scrsz(4)*.7-figSize/2,figSize,figSize]);
hold on; grid on;
hBS = plot(BSPos(1,:),BSPos(2,:),'or');   % Plot BS
hMS = plot(MSPos(1,:),MSPos(2,:),'xb');   % Plot MS
for linkIdx = 1:numLinks  % Plot links
```

```
    pairStn = cfgLayout.Pairing(:,linkIdx);
    pairPos = cell2mat({cfgLayout.Stations(pairStn).Pos});
    plot(pairPos(1,:),pairPos(2,:),'-b');
end
xlim([0 300]); ylim([0 300]);
xlabel('X Position (meters)'); ylabel('Y Position (meters)')
legend([hBS, hMS],'BS','MS','location','northwest');
```



Configure Model Parameters

There are multiple model parameters that can be adjusted in the structure created by the `winner2.wimparset` function. In this example, the center frequency is 5.25 GHz. Path loss and shadowing fading are modeled for each link. To support bandwidth up to 100 MHz, the two strongest clusters of each link are divided into 3 subclusters each which are 5 ns apart. All links are sampled at different rates which depend on the velocity of the MSs. Because the third and fourth links connect to the same MS, they share the same sample rate.

```
frameLen = 1600; % Number of samples to be generated

cfgWim = winner2.wimparset;
cfgWim.NumTimeSamples      = frameLen;
cfgWim.IntraClusterDsUsed  = 'yes';
cfgWim.CenterFrequency     = 5.25e9;
cfgWim.UniformTimeSampling = 'no';
cfgWim.ShadowingModelUsed  = 'yes';
```

```
cfgWim.PathLossModelUsed   = 'yes';
cfgWim.RandomSeed          = 31415926;  % For repeatability
```

Create WINNER II Channel System Object™

We are now able to use the model and layout configurations to create a WINNER II channel System object. Once the object is created, you can call its `info` method to view some derived system parameters. For example, in the `info` method return, the `NumBSElements`, `NumMSElements` and `NumPaths` fields indicate the number of array elements at BS sectors, the number of array elements at MSs and the number of paths for each link. The `SampleRate` field also shows the sample rate for each link.

```
WINNERChan = comm.WINNER2Channel(cfgWim,cfgLayout);
chanInfo = info(WINNERChan)
```

```
chanInfo = struct with fields:
               NumLinks: 6
          NumBSElements: [16 16 12 8 8 8]
          NumMSElements: [4 4 4 4 4 4]
               NumPaths: [16 16 16 16 24 24]
             SampleRate: [3.0636e+07 3.5303e+07 2.7559e+07 2.7559e+07 ... ]
     ChannelFilterDelay: [7 7 7 7 7 7]
    NumSamplesProcessed: 0
```

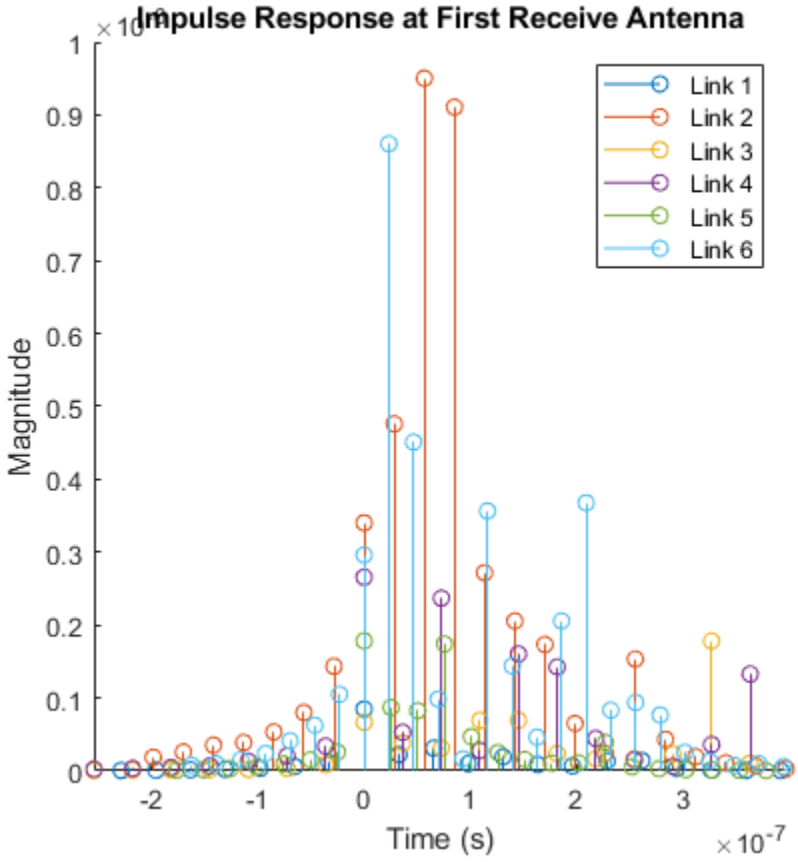
Process Impulse Signal for Each Link

We pass an impulse signal through each link and observe the impulse and frequency responses at the MS. To do so, we need to create the impulse signal for each link and aggregate them into a cell array. This is achieved by using the `NumBSElements` field of the `info` method return and the `cellfun` function. The impulse signal cell array is to be processed by the channel object.

```
txSig = cellfun(@(x) [ones(1,x);zeros(frameLen-1,x)], ...
    num2cell(chanInfo.NumBSElements)','UniformOutput',false);

rxSig = WINNERChan(txSig); % Pass impulse signal through each link
```

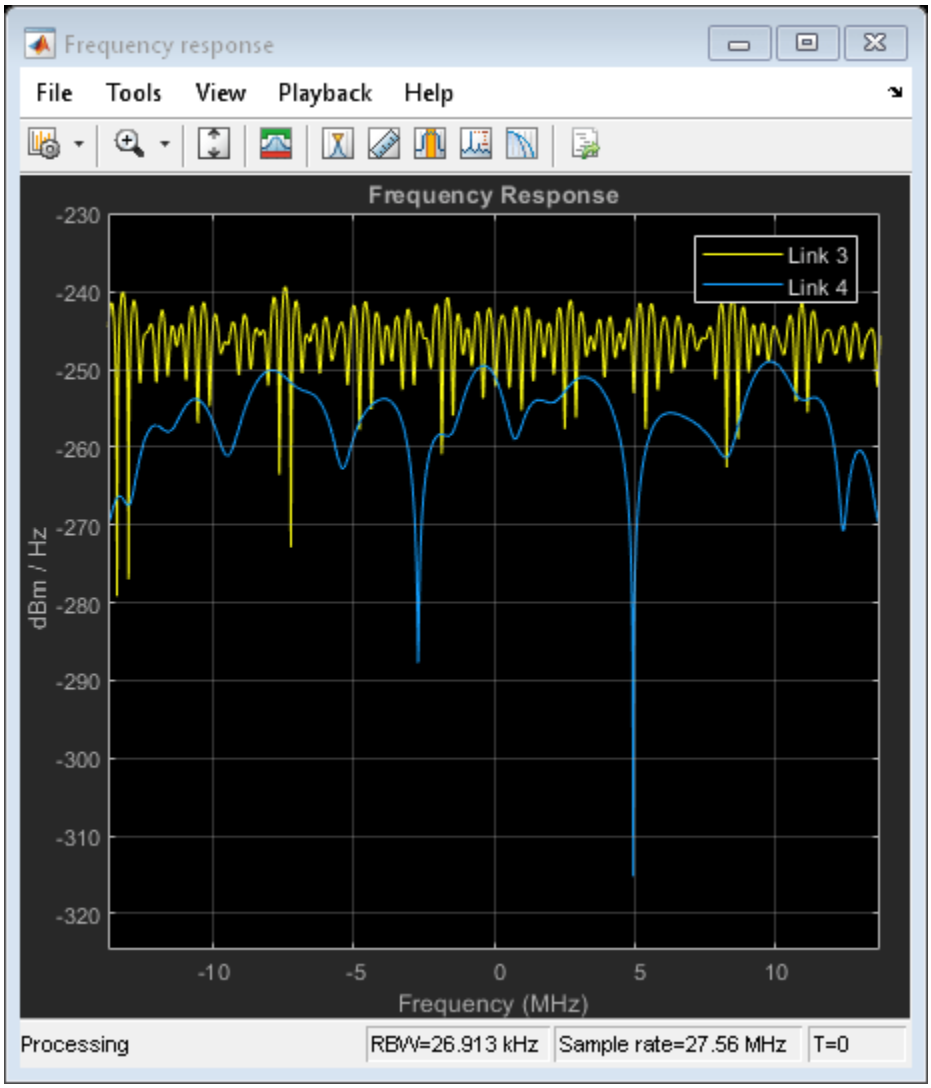
Plotting the received signal at MSs gives an idea about how the fading channel's impulse and frequency responses look for each link. Out of the 4 antennas at each MS, only the received signal at the first antenna is plotted. The fact that the links are sampled at different rates is captured in the impulse response plot. For each link, the first few samples from a channel filter delay are plotted in the negative time axis, if any.

```
figure('Position',[scrsz(3)*.3-figSize/2,scrsz(4)*.25-figSize/2,figSize,figSize]);
hold on;
for linkIdx = 1:numLinks
    delay = chanInfo.ChannelFilterDelay(linkIdx);
    stem(((0:(frameLen-1))-delay)/chanInfo.SampleRate(linkIdx), ...
        abs(rxSig{linkIdx}(:,1)));
end
maxX = max((cell2mat(cellfun(@(x) find(abs(x) < 1e-8,1,'first'), ...
    rxSig.','UniformOutput',false)) - chanInfo.ChannelFilterDelay)./ ...
    chanInfo.SampleRate);
minX = -max(chanInfo.ChannelFilterDelay./chanInfo.SampleRate);
xlim([minX, maxX]);
xlabel('Time (s)'); ylabel('Magnitude');
legend('Link 1','Link 2','Link 3','Link 4','Link 5','Link 6');
title('Impulse Response at First Receive Antenna');
```



As the third and fourth links connect to the same MS and hence have the same sample rate, we plot them together using the Spectrum Analyzer System object. The two links have 16 paths each and demonstrate significant frequency selectivity.

```
SA = dsp.SpectrumAnalyzer( ...
    'Name',          'Frequency response', ...
    'SpectrumType', 'Power density', ...
    'SampleRate',    chanInfo.SampleRate(3), ...
    'Position',      [scrsz(3)*.7-figSize/2,scrsz(4)*.25-figSize/2,figSize,figSize], ...
    'Title',         'Frequency Response', ...
    'ShowLegend',    true, ...
    'ChannelNames', {'Link 3','Link 4'});

SA(cell2mat(cellfun(@(x) x(:,1),rxSig(3:4,1)','UniformOutput',false)));
```




```
rng(s); % Restore RNG
```

Further Exploration

The example shows how to configure a WINNER II fading channel System object to model a system with multiple MIMO links from BSs to MSs. Further exploration includes modifications to the fields of the `cfgLayout` and `cfgWim` to model different antenna arrays like uniform linear arrays (ULA), BS/MS locations and pairings, propagation scenarios and conditions, and so on.

Because the third and fourth links are connecting to the same MS, you can combine the received signals from both links, by offsetting the samples appropriately to account for the channel filter delays on the two links.

Selected Bibliography

1 IST WINNER II, "WINNER II Channel Models", D1.1.2, Sep. 2007.

802.11ac Multiuser MIMO Precoding with WINNER II Channel Model

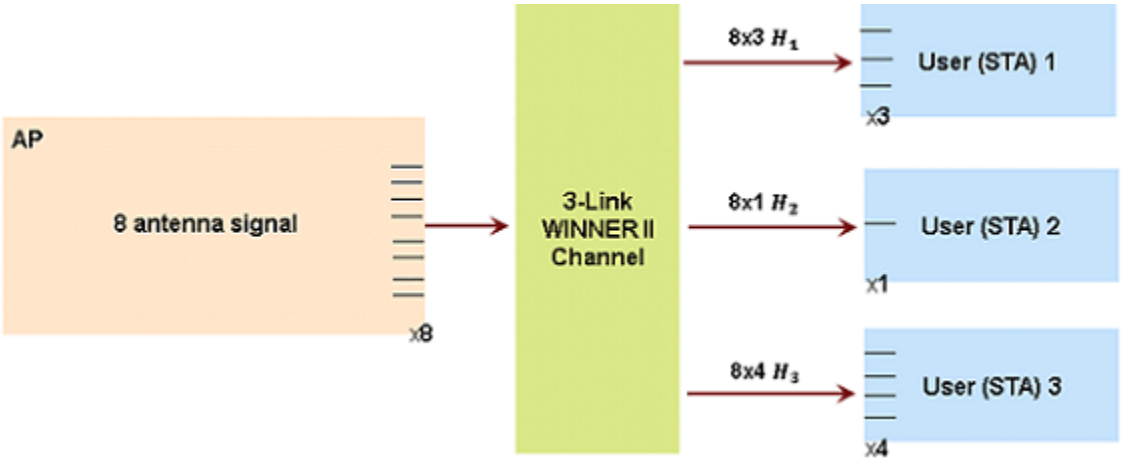
This example shows the transmit and receive processing for a 802.11ac™ multiuser downlink transmission over a WINNER II fading channel. You must download and install the WINNER II Channel Model for Communications Toolbox™ Add-On to run this example. Only one WINNER II channel System object™ is needed to set up the channels from one access point to all users.

Introduction

802.11ac supports downlink (access-point to station) multiuser transmissions for up to four users and up to eight transmit antennas to increase the aggregate throughput of the link [1]. Based on a scheduled transmission time for a user, the scheduler looks for other smaller packets ready for transmission to other users. If available, it schedules these users over the same interval, which reduces the overall time taken for multiple transmissions.

This simultaneous transmission comes at a higher complexity because successful reception of the individual user's payloads requires precoding, also known as transmit-end beamforming. Precoding assumes that channel state information (CSI) is known at the transmitter. A sounding packet, as described in the “802.11ac Transmit Beamforming” (WLAN Toolbox) example, is used to determine the CSI for each user in a multiuser transmission. Each of the users feed back their individual CSI to the beamformer. The beamformer uses the CSI from all users to set the precoding (spatial mapping) matrix for subsequent data transmission.

This example uses a channel inversion technique for a three-user transmission with a different number of spatial streams allocated per user and different rate parameters per user. The system can be characterized by the figure below.



The example generates the multiuser transmit waveform, passes it through a multiuser WINNER II channel and decodes the received signal for each user to calculate the bits in error. Prior to the data transmission, the example uses a null-data packet (NDP) transmission to sound the different channel for each user and determines the precoding matrix under the assumption of perfect feedback.

Check for Support Package Installation

Check if the 'WINNER II Channel Model for Communications Toolbox' support package is installed.

```
commSupportPackageCheck('CST_WINNER2');
```

Simulation Parameters and Configuration

For 802.11ac, a maximum of eight spatial streams is allowed. An 8x8 MIMO configuration for three users is used in this example, where the first user has three streams, second has one, and the third has four streams allocated to it. Different rate parameters and payload sizes for each user are specified as vector parameters for the transmission configuration.

```
s = rng(10);                            % Set RNG seed for repeatability

% Transmission parameters
chanBW      = 'CBW80';                  % Channel bandwidth
numUsers    = 3;                        % Number of users
numSTSVec   = [3 1 4];                  % Number of streams per user
userPos     = [0 1 2];                  % User positions
mcsVec      = [4 6 8];                  % MCS per user: 16QAM, 64QAM, 256QAM
apepVec     = [520 192 856];            % Payload per user, in bytes
chCodingVec = {'BCC','LDPC','LDPC'}; % Channel coding per user

% Precoding and equalization parameters
precodingType = 'ZF';                   % Precoding type; ZF or MMSE
snr           = 47;                     % SNR in dB
eqMethod      = 'ZF';                   % Equalization method

% Create the multiuser VHT format configuration object
numTx = sum(numSTSVec);
cfgVHTMU = wlanVHTConfig('ChannelBandwidth',chanBW, ...
    'NumUsers',numUsers, ...
    'NumTransmitAntennas',numTx, ...
    'GroupID',2, ...
    'NumSpaceTimeStreams',numSTSVec,...
    'UserPositions',userPos, ...
    'MCS',mcsVec, ...
    'APEPLength',apepVec, ...
    'ChannelCoding',chCodingVec);
```

The number of transmit antennas is set to be the sum total of all the used space-time streams. This implies no space-time block coding (STBC) or spatial expansion is employed for the transmission.

Sounding (NDP) Configuration

For precoding, channel sounding is first used to determine the channel experienced by the users (receivers). This channel state information is sent back to the transmitter, for it to be used for subsequent data transmission. It is assumed that the channel varies slowly over the two transmissions. For multiuser transmissions, the same NDP (Null Data Packet) is transmitted to each of the scheduled users [2].

```
% VHT sounding (NDP) configuration, for same number of streams
cfgVHTNDP = wlanVHTConfig('ChannelBandwidth',chanBW, ...
    'NumUsers',1, ...
    'NumTransmitAntennas',numTx, ...
    'GroupID',0, ...
    'NumSpaceTimeStreams',sum(numSTSVec),...
    'MCS',0, ...
    'APEPLength',0);
```

The number of streams specified is the sum total of all space-time streams used. This allows the complete channel to be sounded.

```
% Generate the null data packet, with no data
txNDPSig = wlanWaveformGenerator([],cfgVHTNDP);
NPDSigLen = size(txNDPSig, 1);
```

WINNER II Channel for Indoor Office (A1) Scenario

In this example, one `comm.WINNER2Channel` System object in the WINNER II Channel Model for Communications Toolbox™ is set up to simulate the three channels to different users. The indoor office (A1) non-line-of-sight (NLOS) scenario is configured for each user. With a fixed power delay profile, each user experiences a 16-path fading channel with the largest delay of 175 us. Each user is also assigned a low mobility as appropriate for 802.11ac.

The access point employs a uniform circular array (UCA) with a radius of 20cm. Each user employs a uniform linear array (ULA) with 5cm spacing between elements. It is also assumed that each user's number of receive antennas is equal to the number of space-time streams allocated to them.

```
% Set up layout parameters for WINNER II channel
AA = winner2.AntennaArray('UCA',numTx,0.2);
for i = 1:numUsers
    AA(i+1) = winner2.AntennaArray('ULA',numSTSVec(i),0.05);
end
STAIdx   = 2:(numUsers+1);
APIdx    = {1};
rndSeed = 12;
cfgLayout = winner2.layoutparset(STAIdx,APIdx,numUsers,AA,[],rndSeed);
cfgLayout.Pairing = [ones(1,numUsers);2:(numUsers+1)]; % One access point to all users
cfgLayout.ScenarioVector = ones(1,numUsers);           % A1 scenario for all links
cfgLayout.PropagConditionVector = zeros(1,numUsers);  % NLOS
for i = 1:numUsers % Randomly set velocity for each user
    v = rand(3,1) - 0.5;
    cfgLayout.Stations(i+1).Velocity = v/norm(v,'fro');
end

% Set up model parameters for WINNER II channel
cfgModel = winner2.wimparset;
cfgModel.FixedPdpUsed       = 'yes';
cfgModel.FixedAnglesUsed    = 'yes';
cfgModel.IntraClusterDsUsed = 'no';
cfgModel.RandomSeed         = 111;    % Repeatability

% The maximum velocity for the 3 users is 1m/s. Set up the SampleDensity
% field to ensure that the sample rate matches the channel bandwidth.
maxMSVelocity = max(cell2mat(cellfun(@(x) norm(x,'fro'), ...
    {cfgLayout.Stations.Velocity},'UniformOutput',false)));
cfgModel.UniformTimeSampling = 'yes';
cfgModel.SampleDensity = round(physconst('LightSpeed')/ ...
    cfgModel.CenterFrequency/2/(maxMSVelocity/wlanSampleRate(cfgVHTMU)));

% Create the WINNER II channel System object
WINNERChan = comm.WINNER2Channel(cfgModel,cfgLayout);

% Call the info method to check some derived channel parameters
chanInfo = info(WINNERChan)
```

```
chanInfo =

  struct with fields:
```

```
             NumLinks: 3
        NumBSElements: [8 8 8]
        NumMSElements: [3 1 4]
             NumPaths: [16 16 16]
           SampleRate: [8.0000e+07 8.0000e+07 8.0000e+07]
    ChannelFilterDelay: [7 7 7]
    NumSamplesProcessed: 0
```

The channel filtering delay for each user is stored to account for its compensation at the receiver. In practice, symbol timing estimation would be used. At transmitter, an extra ten all-zero samples are appended to account for channel filter delay.

```
chanDelay   = chanInfo.ChannelFilterDelay;
numPadZeros = 10;

% Set ModelConfig.NumTimeSamples to match the length of the input signal to
% avoid warning
WINNERChan.ModelConfig.NumTimeSamples = NPDSigLen + numPadZeros;

% Sound the WINNER II channel for all users
chanOutNDP = WINNERChan([txNDPSig;zeros(numPadZeros,numTx)]);

% Add AWGN
rxNDPSig = cellfun(@awgn,chanOutNDP, ...
    num2cell(snr*ones(numUsers,1)),'UniformOutput',false);
```

Channel State Information Feedback

Each user estimates its own channel using the received NDP signal and computes the channel state information that it can send back to the transmitter. This example uses the singular value decomposition of the channel seen by each user to compute the CSI feedback.

```
mat = cell(numUsers,1);
for uIdx = 1:numUsers
    % Compute the feedback matrix based on received signal per user
    mat{uIdx} = vhtCSIFeedback(rxNDPSig{uIdx}(chanDelay(uIdx)+1:end,:), ...
        cfgVHTNDP,uIdx,numSTSVec);
end
```

Assuming perfect feedback, with no compression or quantization loss of the CSI, the transmitter computes the steering matrix for the data transmission using either Zero-Forcing or Minimum-Mean-Square-Error (MMSE) based precoding techniques. Both methods attempt to cancel out the intra-stream interference for the user of interest and interference due to other users. The MMSE-based approach avoids the noise enhancement inherent in the zero-forcing technique. As a result, it performs better at low SNRs.

```
% Pack the per user CSI into a matrix
numST = length(mat{1});         % Number of subcarriers
steeringMatrix = zeros(numST,sum(numSTSVec),sum(numSTSVec));
%   Nst-by-Nt-by-Nsts
for uIdx = 1:numUsers
    stsIdx = sum(numSTSVec(1:uIdx-1))+(1:numSTSVec(uIdx));
    steeringMatrix(:,:,stsIdx) = mat{uIdx};     % Nst-by-Nt-by-Nsts
end
```

```
% Zero-forcing or MMSE precoding solution
if strcmp(precodingType, 'ZF')
    delta = 0; % Zero-forcing
else
    delta = (numTx/(10^(snr/10))) * eye(numTx); % MMSE
end
for i = 1:numST
    % Channel inversion precoding
    h = squeeze(steeringMatrix(i,:,:));
    steeringMatrix(i,:,:) = h/(h'*h + delta);
end

% Set the spatial mapping based on the steering matrix
cfgVHTMU.SpatialMapping = 'Custom';
cfgVHTMU.SpatialMappingMatrix = permute(steeringMatrix,[1 3 2]);
```

Data Transmission

Random bits are used as the payload for the individual users. A cell array is used to hold the data bits for each user, `txDataBits`. For a multiuser transmission the individual user payloads are padded such that the transmission duration is the same for all users. This padding process is described in Section 9.12.6 of [1]. In this example for simplicity the payload is padded with zeros to create a PSDU for each user.

```
% Create data sequences, one for each user
txDataBits = cell(numUsers,1);
psduDataBits = cell(numUsers,1);
for uIdx = 1:numUsers
    % Generate payload for each user
    txDataBits{uIdx} = randi([0 1],cfgVHTMU.APEPLength(uIdx)*8,1,'int8');

    % Pad payload with zeros to form a PSDU
    psduDataBits{uIdx} = [txDataBits{uIdx}; ...
        zeros((cfgVHTMU.PSDULength(uIdx)-cfgVHTMU.APEPLength(uIdx))*8,1,'int8')];
end
```

Using the format configuration, `cfgVHTMU`, with the steering matrix, to generate the multiuser VHT waveform.

```
txSig = wlanWaveformGenerator(psduDataBits,cfgVHTMU);
```

The WINNER II channel object does not allow the input signal size to change once locked, so we have to call the release method before passing the waveform through it. In addition, as we restart the channel, we want it to re-process the NDP before the waveform so as to accurately mimic the channel continuity. Only the waveform portion of the channel's output is extracted for the subsequent processing of each user.

```
release(WINNERChan);

% Set ModelConfig.NumTimeSamples to match the total length of NDP plus
% waveform and padded zeros
WINNERChan.ModelConfig.NumTimeSamples = ...
    WINNERChan.ModelConfig.NumTimeSamples + length(txSig) + numPadZeros;

% Transmit through the WINNER II channel for all users, with 10 all-zero
% samples appended to account for channel filter delay
chanOut = WINNERChan([txNDPSig; zeros(numPadZeros,numTx); ...
```

```
    txSig; zeros(numPadZeros,numTx)]);

% Extract the waveform output for each user
chanOut = cellfun(@(x) x(NPDSigLen+numPadZeros+1:end,:),chanOut,'UniformOutput',false);

% Add AWGN
rxSig = cellfun(@awgn,chanOut, ...
    num2cell(snr*ones(numUsers,1)),'UniformOutput',false);
```

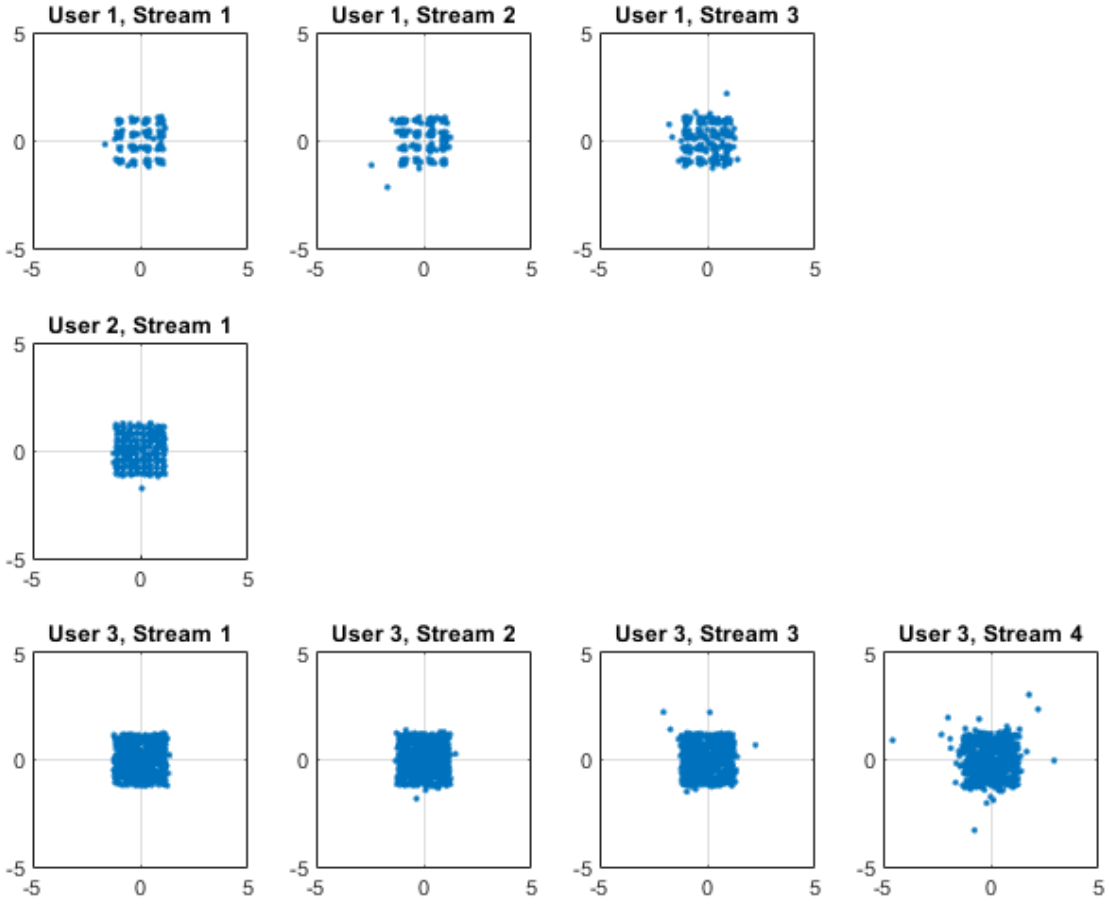
Data Recovery Per User

The receive signals for each user are processed individually. The example assumes that there are no front-end impairments and that the transmit configuration is known by the receiver for simplicity.

A user number specifies the user of interest being decoded for the transmission. This is also used to index into the vector properties of the configuration object that are user-specific.

```
% Get field indices from configuration, assumed known at receiver
ind = wlanFieldIndices(cfgVHTMU);

% Single-user receivers recover payload bits
rxDataBits = cell(numUsers,1);
scaler = zeros(numUsers,1);
spAxes = gobjects(sum(numSTSVec),1);
hfig = figure('Name','Per-stream equalized symbol constellation');
for uIdx = 1:numUsers
    rxNSig = rxSig{uIdx}(chanDelay(uIdx)+1:end, :);

    % User space-time streams
    stsU = numSTSVec(uIdx);

    % Estimate noise power in VHT fields
    lltf = rxNSig(ind.LLTF(1):ind.LLTF(2),:);
    demodLLTF = wlanLLTFDemodulate(lltf,chanBW);
    nVar = helperNoiseEstimate(demodLLTF,chanBW,sum(numSTSVec));

    % Perform channel estimation based on VHT-LTF
    rxVHTLTF  = rxNSig(ind.VHTLTF(1):ind.VHTLTF(2),:);
    demodVHTLTF = wlanVHTLTFDemodulate(rxVHTLTF,chanBW,numSTSVec);
    chanEst = wlanVHTLTFChannelEstimate(demodVHTLTF,chanBW,numSTSVec);

    % Recover information bits in VHT Data field
    rxVHTData = rxNSig(ind.VHTData(1):ind.VHTData(2),:);
    [rxDataBits{uIdx},~,eqsym] = wlanVHTDataRecover(rxVHTData, ...
        chanEst,nVar,cfgVHTMU,uIdx, ...
        'EqualizationMethod',eqMethod,'PilotPhaseTracking','None', ...
        'LDPCDecodingMethod','layered-bp','MaximumLDPCIterationCount',6);

    % Plot equalized symbols for all streams per user
    scaler(uIdx) = ceil(max(abs([real(eqsym(:)); imag(eqsym(:))])));
    for i = 1:stsU
        subplot(numUsers,max(numSTSVec),(uIdx-1)*max(numSTSVec)+i);
        plot(reshape(eqsym(:,:,i),[],1),'.');
        axis square
        spAxes(sum([0 numSTSVec(1:(uIdx-1))])+i) = gca; % Store axes handle
        title(['User ' num2str(uIdx) ', Stream ' num2str(i)]);
        grid on;
    end
```

```
end

% Scale axes for all subplots and scale figure
for i = 1:numel(spAxes)
    xlim(spAxes(i),[-max(scaler) max(scaler)]);
    ylim(spAxes(i),[-max(scaler) max(scaler)]);
end
pos = get(hfig,'Position');
set(hfig,'Position',[pos(1)*0.7 pos(2)*0.7 1.3*pos(3) 1.3*pos(4)]);
```



Per-stream equalized symbol constellation plots validate the simulation parameters and convey the effectiveness of the technique. Note the discernible 16QAM, 64QAM and QPSK constellations per user as specified on the transmit end. Also observe the EVM degradation over the different streams for an individual user. This is a representative characteristic of the channel inversion technique.

The recovered data bits are compared with the transmitted payload bits to determine the bit error rate.

```
% Compare recovered bits against per-user APEPLength information bits
ber = inf(1, numUsers);
for uIdx = 1:numUsers
```

```
    idx = (1:cfgVHTMU.APEPLength(uIdx)*8).';
    [~,ber(uIdx)] = biterr(txDataBits{uIdx}(idx),rxDataBits{uIdx}(idx));
    disp(['Bit Error Rate for User ' num2str(uIdx) ': ' num2str(ber(uIdx))]);
end

rng(s); % Restore RNG state
```

```
Bit Error Rate for User 1: 0
Bit Error Rate for User 2: 0
Bit Error Rate for User 3: 0.00014603
```

The small number of bit errors, within noise variance, indicate successful data decoding for all streams for each user, despite the variation in EVMs seen in individual streams.

Conclusion and Further Exploration

The example shows how to use the WINNER II fading channel System object to model a multiuser VHT transmission in 802.11ac. Further exploration includes modifications to the transmission parameters, antenna arrays, channel scenarios, LOS vs. NLOS propagations, path loss modeling and shadowing modeling.

There is another version of this example in the WLAN Toolbox, which uses three independent TGac fading channels for three users: “802.11ac Multi-User MIMO Precoding” (WLAN Toolbox).

Appendix

This example uses the following helper functions from WLAN Toolbox™:

- helperNoiseEstimate.m
- vhtCSIFeedback.m

Selected Bibliography

1. IEEE® Std 802.11ac™-2013 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz.
2. Perahia, E., R. Stacey, "Next Generation Wireless LANS: 802.11n and 802.11ac", Cambridge University Press, 2013.
3. IEEE Std 802.11™-2012 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.
4. IST WINNER II, "WINNER II Channel Models", D1.1.2, Sep. 2007.
5. Breit, G., H. Sampath, S. Vermani, et al., "TGac Channel Model Addendum", Version 12. IEEE 802.11-09/0308r12, March 2010.

End-to-End QAM Simulation with RF Impairments and Corrections

This example provides visualization capabilities to see the effects of RF impairments and corrections in a satellite downlink. The link employs 16-QAM modulation in the presence of AWGN and uses a High Power Amplifier (HPA) to overcome the losses associated with satellite communications. The HPA introduces nonlinear behavior that, when combined with other RF impairments, requires the use of mitigation techniques.

This example includes:

- A MATLAB® GUI, QAMwithRFImpairmentsExample.
- A MATLAB-based simulator function, QAMwithRFImpairmentsSim.m, which receives its input parameters from the GUI.

Keywords: QAM, RF impairments, I/Q imbalance, nonlinearity, RF correction.

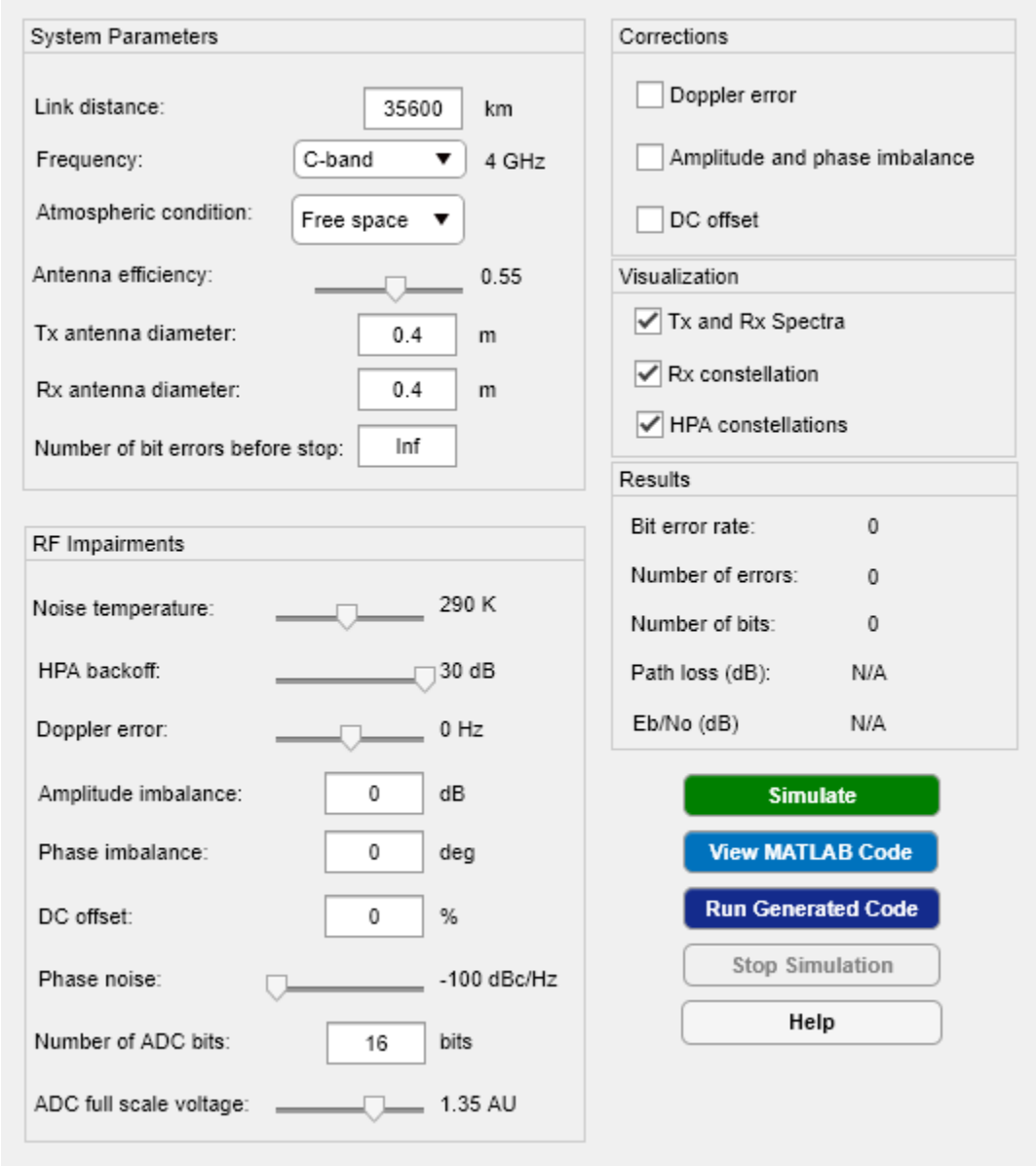
Introduction

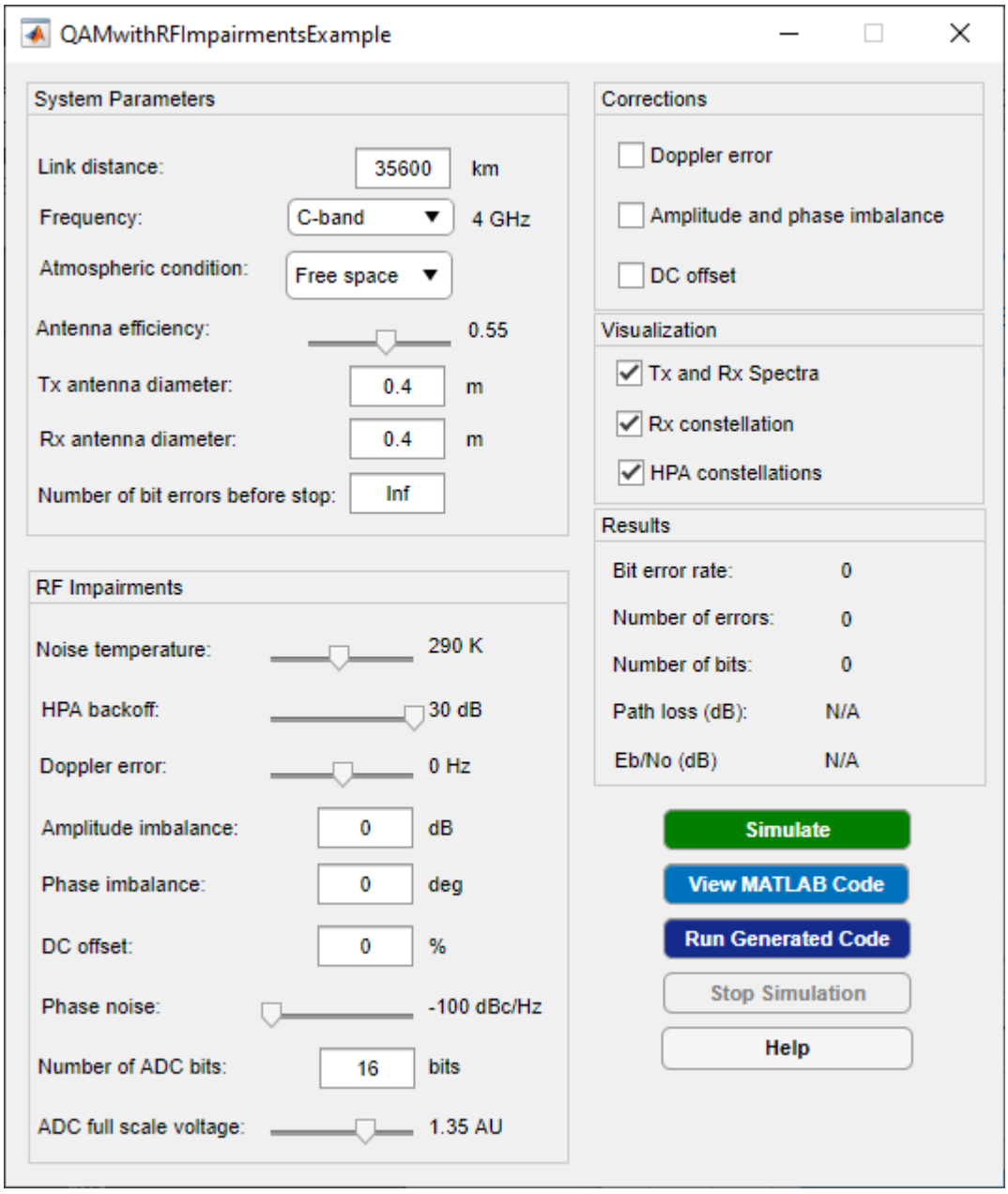
The simulation allows you to configure the parameters shown in the GUI.

Open the GUI to:

- Modify the parameters
- Run the simulation with MATLAB
- Visualize signal constellations and spectra
- View the underlying MATLAB code
- Generate C code and run the simulation (with a valid MATLAB Coder™ license)

```
QAMwithRFImpairmentsExample
```


System Parameters
Link distance:
35600
km
Frequency:
C-band
4 GHz
Atmospheric condition:
Free space
Antenna efficiency:
0.55
Tx antenna diameter:
0.4
m
Rx antenna diameter:
0.4
m
Number of bit errors before stop:
Inf
RF Impairments
Noise temperature:
290 K
HPA backoff:
30 dB
Doppler error:
0 Hz
Amplitude imbalance:
0
dB
Phase imbalance:
0
deg
DC offset:
0
%
Phase noise:
-100 dBc/Hz
Number of ADC bits:
16
bits
ADC full scale voltage:
1.35 AU
Corrections
Doppler error
Amplitude and phase imbalance
DC offset
Visualization
Tx and Rx Spectra
Rx constellation
HPA constellations
Results
Bit error rate:
0
Number of errors:
0
Number of bits:
0
Path loss (dB):
N/A
Eb/No (dB)
N/A
Simulate
View MATLAB Code
Run Generated Code
Stop Simulation
Help



The **Simulate** button simulates the configured link using interpreted MATLAB code. While the simulation is running, you can modify some simulation parameters using the GUI. The impact of parameter setting updates is immediately observable on the Results panel or on the plots. Parameters that are nontunable while the simulation is running are grayed out. To modify nontunable parameters, the simulation must be stopped.

The **View MATLAB Code** button opens the simulator code in the editor allowing for visual inspection and further exploration of the underlying functions used in the simulation.

The **Run Generated Code** button compiles the MATLAB function into an executable MEX-file and runs the simulation once the compiling process is complete. The MEX version of the simulation runs

much faster though there is a time penalty from the compiling process itself. You can modify the same parameters when running from either interpreted mode or from the MEX-file.

The **Stop Simulation** button stops the simulation during execution. This works for both interpreted MATLAB and the MEX-file. The button is active only when a simulation is running.

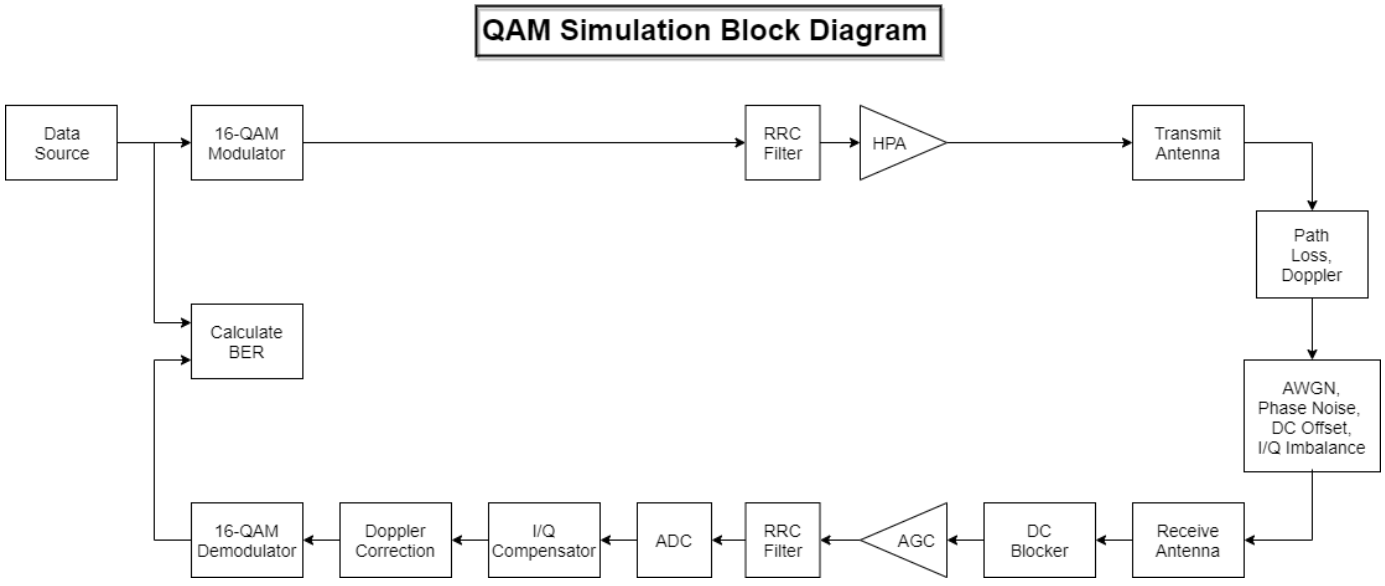
The **Help** button brings up this HTML page.

Simulation Overview

The simulation executes the following steps:

- Generate random integers
- Modulate with 16-QAM
- Root raised cosine (RRC) transmit filter
- Pass through an HPA
- Apply transmit antenna gain
- Apply path loss based on atmospheric condition
- Pass the signal through an AWGN channel with RF impairments
- Apply receive antenna gain
- Remove DC offset
- Apply automatic gain control
- RRC receive filter
- Apply ADC effects
- Compensate for I/Q amplitude and phase imbalance
- Correct for the Doppler shift
- Demodulate 16-QAM
- Calculate the bit error rate

The following block diagram shows the architecture of the system.



You can specify the following signal impairments:

- Receiver noise temperature in the range [0, 600] K
- Doppler error in the range [-3, 3] Hz
- DC offset, expressed as a percentage of the maximum signal voltage, in the range [0, 20]
- Phase noise in the range [-100, -48] dBc/Hz
- I/Q amplitude imbalance in the range [-5, 5] dB
- I/Q phase imbalance in the range [-30, 30] degrees
- HPA backoff level in the range [1, 30] dB
- Quantization error by changing the number of ADC bits in the range of [2 16] bits
- Saturation due to ADC full scale voltage in the range of [0.1 2] amplitude units (AUs)

An HPA backoff of 30 dB corresponds to negligible distortion because the amplifier is operating in its linear region, while 1 dB corresponds to severe distortion. A Saleh model is used to simulate the behavior of the HPA. Further information is available on the `comm.MemorylessNonlinearity` page.

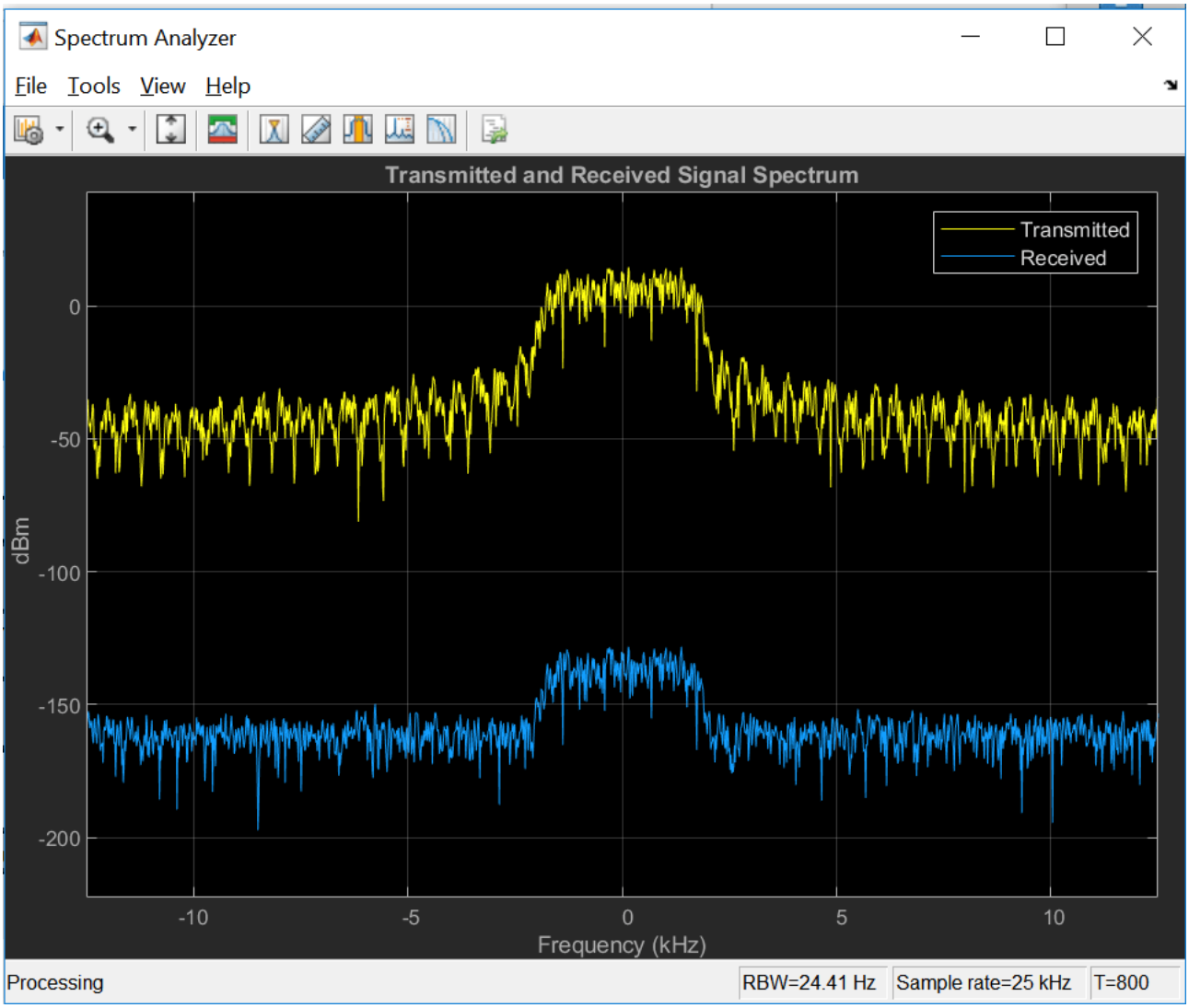
The GUI provides the ability to enable or disable corrections for Doppler error, I/Q imbalance, and DC offset. These corrections are provided by three System objects. The `comm.CarrierSynchronizer` compensates for the frequency offset due to Doppler, the `comm.IQImbalanceCompensator` corrects the amplitude and phase imbalance, and the `dsp.DCBlocker` compensates for the DC offset.

Results and Displays

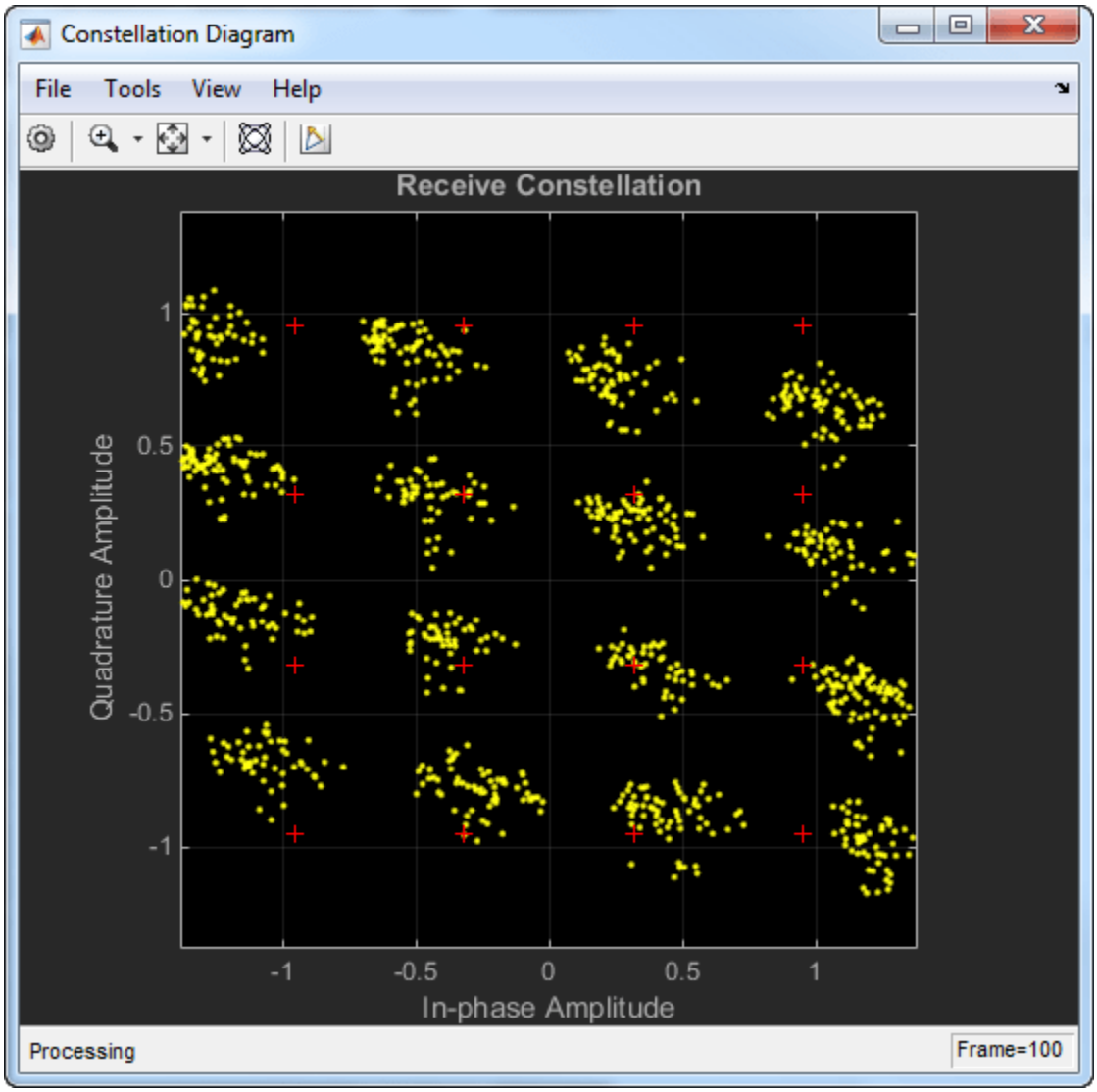
You can use GUI controls to display:

- The spectrum of the transmitted signal measured at the output of the transmit RRC filter.
- The spectrum of the received signal measured at the input of the receive RRC filter.
- The constellation diagram of the received signal.
- The constellation diagrams of the HPA input signal.
- The constellation diagrams of the HPA output signal

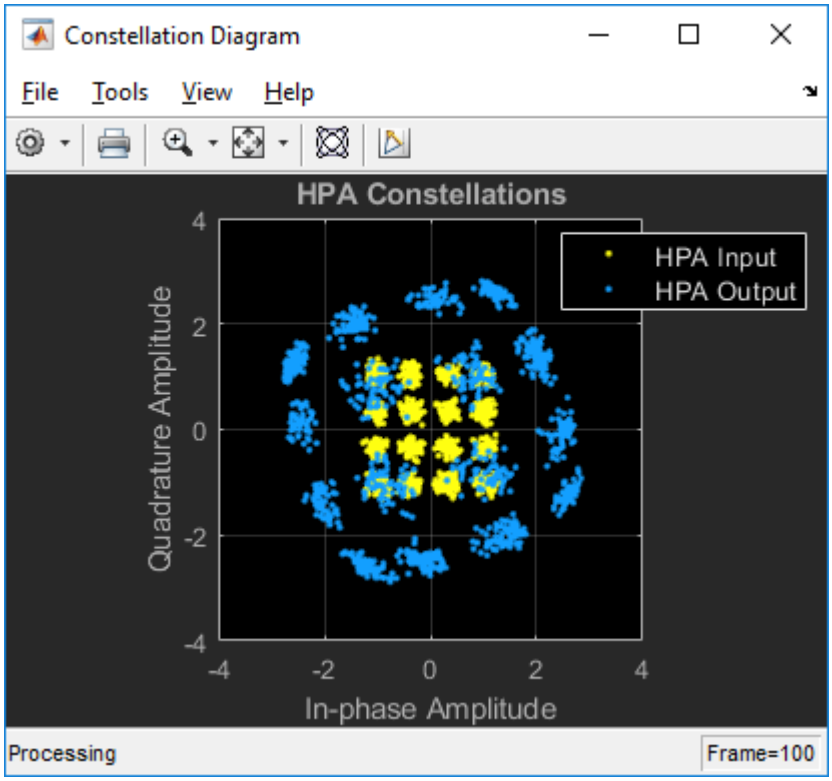
A typical spectrum plot, using the default parameters, is shown. The effects of AWGN are most easily seen in the out-of-band signal spectrum, where the noise floor of the received signal is 20 dB higher than the transmitted signal spectrum. The received signal spectrum also shows the effect of propagation loss through the channel.



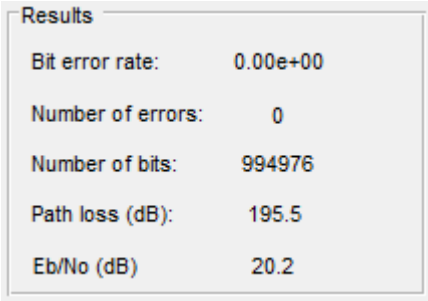
A plot of the constellation diagram is shown for the case in which the I/Q imbalance correction is disabled. The red + symbols denote the 16-QAM reference constellation. The constellation is scaled and rotated by the uncorrected imbalance.



The effects of nonlinear HPA behavior are shown as **HPA Input** and **HPA Output** using the same constellation diagram plot. The diagrams show the effects of AM/AM and AM/PM distortion when the amplifier operates 7 dB below saturation. AM/AM distortion causes the 'rounded' appearance of the HPA output signal constellation, while AM/PM causes the constellation to rotate.



The bit error rate, number of errors, total number of transmitted symbols, path loss, and the Eb/No are displayed directly on the results panel of the GUI.



Further Exploration

Use the GUI to change the parameters listed below.

- **Link gains and losses:** Vary the noise temperature between 0 to 290 K (typical) to view the effects on the received signal spectrum analyzer plot. Likewise, change the link distance, atmospheric condition and carrier frequency to view the impact on the received signal spectrum. Changes in the link margin are also reflected in the calculated path loss and Eb/No.
- **HPA AM-to-AM and AM-to-PM conversion:** Vary the *HPA Backoff* between 30 dB (negligible nonlinearity) to 1 dB (severe nonlinearity). A value of 7 dB corresponds to moderate nonlinearity. View the effects on the spectrum plot, the HPA output constellation, the received signal constellation diagram, and on the bit error rate. Increasing nonlinearity increases spectral regrowth and causes the HPA output constellation to become 'rounder' and rotate. The *HPA Backoff* parameter can be adjusted while the simulation is executing.

- **Phase noise:** Set the *Phase Noise* to -48 dBc/Hz (high) and observe the increased variance in the tangential direction in the received signal constellation diagram. This level of phase noise is sufficient to cause errors in an otherwise error-free channel. Set the *Phase Noise* to -55 dBc/Hz (low) and observe that the variance in the tangential direction has decreased. This level of phase noise does not significantly increase the error rate. Now, set the *HPA Backoff* level parameter to 7 dB (moderate nonlinearity). Note that even though the moderate HPA nonlinearity and the moderate phase noise do not cause many bit errors when applied individually, they do cause significantly more bit errors when applied together. The *Phase Noise* parameter can be adjusted only when the simulation is stopped.
- **DC offset and DC offset correction:** Set the *DC offset* to `10` and disable the DC offset correction by unchecking the *DC Offset* checkbox. The constellation diagram changes significantly. Re-enable the *DC Offset* correction and view the received signal constellation diagram and signal spectrum to verify that the DC offset is removed. Both the DC offset and the DC offset correction parameters can be modified during simulation execution.
- **I/Q imbalance:** Disable the *Amplitude and phase imbalance* box to view the effects of an I/Q imbalance on the received constellation diagram. Modify the amplitude and phase imbalance fields to observe the effects of different values on the received signal constellation diagram. Re-enable the I/Q Imbalance correction to verify that the receive constellation aligns with its reference points. These parameters can be modified during execution.
- **Doppler and Doppler compensation:** Set *Doppler error* to 0.7 Hz and disable the *Doppler error* correction to show the effect of uncorrected Doppler on the received signal. Note that the BER is close to 0.5. Re-enable the *Doppler error* correction to correct for the Doppler error. Verify that the BER decreases. These parameters are available only when the simulation is stopped.
- **ADC Effects:** Decrease the number of ADC bits to view the effect of increasing quantization errors on the received signal. Decrease the ADC full scale voltage to impose saturation on the received signal and view its effect on the system performance.
- **Code Generation:** Run the simulation by clicking the *Run Generated Code* button. The first time this is done, the simulation compiles before executing, which makes the process take longer than it does when simulating with interpreted MATLAB. Change the *HPA backoff* level and rerun the simulation. Note that the results panel updates very quickly. Now, change the *Phase noise* and click the *Run Generated Code* button. The code is recompiled because the phase noise is a nontunable parameter. Enable the *Rx constellation* option and rerun the simulation. You can see that when the scope is activated, the bit error results accumulate more slowly but the scope updates much faster than it does when running with interpolated MATLAB.
- **BER estimation:** By default, the *Number of bit errors* parameter is set to `Inf` so that the effects of the impairments and corrections can be easily visualized on the scopes. For BER estimation, it is typically sufficient to collect 50 to 200 errors; consequently, disable the scopes and change the *Number of bit errors* parameter from `Inf` to 100. It is important to leave the modifiable parameters unchanged when the simulation is running to obtain a valid BER estimate.

Selected Bibliography

[1] Saleh, Adel A.M., "Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers," IEEE® Transactions on Communications, Vol. COM-29, No. 11, November 1981.

[2] Kasdin, N.J., "Discrete Simulation of Colored Noise and Stochastic Processes and 1/(f^alpha); Power Law Noise Generation," The Proceedings of the IEEE, Vol. 83, No. 5, May, 1995.

[3] Kasdin, N. Jeremy, and Todd Walter, "Discrete Simulation of Power Law Noise," 1992 IEEE Frequency Control Symposium.

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

HF Ionospheric Channel Models

This example shows how to simulate High-Frequency (HF) ionospheric channels, based on the models described in Recommendation ITU-R F.1487. In particular, it shows how to simulate the general Watterson channel model, and other simplified channel models used in the quantitative testing of HF modems. It makes use of the `comm.RayleighChannel` System object™ and `stdchan` function along with the Gaussian and bi-Gaussian `doppler` structures from Communications Toolbox™.

ITU-R HF Channel Models: Overview

In HF ionospheric radio communications, the transmitted signal can bounce off several times from the E and F layers of the ionosphere, which results in several propagation paths, also called modes [1 on page 8-0]. Typically, the multipath delay spreads are large, as compared to mobile radio. Also, the signal can suffer from Doppler spread due to the turbulence of the ionosphere. However, the fading rate is usually smaller than for mobile radio.

Recommendation ITU-R F.1487 [1 on page 8-0] proposes a general Gaussian scatter model for the simulation of HF ionospheric channels. This model is based on Watterson's channel model [2 on page 8-0]. Simpler models are also proposed in [1 on page 8-0] for use in HF modem tests, with specified parameters.

Initialization of Simulation-Specific Parameters

The simulation sampling rate `Rs` is specified to 9.6K Hz, and kept the same for the remainder of the example. We use a QPSK modulation scheme with zero phase offset.

```
Rs = 9.6e3;                      % Channel sampling rate
M = 4;                           % Modulation order
qpskMod = comm.QPSKModulator(0); % QPSK modulator object
```

Watterson Channel Model

The Watterson channel model consists of a tapped delay line, where each tap corresponds to a resolvable propagation path. On each tap, two magneto-ionic components are present: each one is modeled as a complex Gaussian random process with a given gain and frequency shift, and whose Doppler spectrum is Gaussian with a given standard deviation [2 on page 8-0]. Hence, each tap is characterized by a bi-Gaussian Doppler spectrum, which consists of two Gaussian functions in the frequency domain, each one with its own set of parameters (power gain, frequency shift, and standard deviation).

In this example, we follow the Watterson simulation model specified in [1 on page 8-0], in which the complex fading process on each tap is obtained by adding two independent frequency-shifted complex Gaussian random processes (with Gaussian Doppler spectra) corresponding to the two magneto-ionic components. This simulation model leads to a complex fading process whose envelope is in general *not* Rayleigh distributed. Hence, to be faithful to the simulation model, we cannot simply generate a Rayleigh channel with a bi-Gaussian Doppler spectrum. Instead, we generate two independent Rayleigh channels, each with a frequency-shifted Gaussian Doppler spectrum, gain-scale them, and add them together to obtain the Watterson channel model with a bi-Gaussian Doppler spectrum. For simplicity, we simulate a Watterson channel with only one tap.

A frequency-shifted Gaussian Doppler spectrum can be seen as a bi-Gaussian Doppler spectrum in which only one Gaussian function is present (the second one having a zero power gain). Hence, to emulate the frequency-shifted Gaussian Doppler spectrum of each magneto-ionic component, we construct a bi-Gaussian Doppler structure such that one of the two Gaussian functions has the specified frequency shift and standard deviation, while the other has a zero power gain.

The first magneto-ionic component has a Gaussian Doppler spectrum with standard deviation `sGauss1`, frequency shift `fGauss1`, and power gain `gGauss1`. A bi-Gaussian Doppler structure `dopplerComp1` is constructed such that the second Gaussian function has a zero power gain (its standard deviation and center frequency are hence irrelevant, and take on default values), while the first Gaussian function has a normalized standard deviation `sGauss1/fd` and a normalized frequency shift `fGauss1/fd`, where the normalization factor `fd` is the maximum Doppler shift of the corresponding channel. In this example, since the gain of the second Gaussian function is zero, the value assigned to the gain of the first Gaussian function is irrelevant (we leave it to its default value of 0.5), because the associated channel System object created later normalizes the Doppler spectrum to have a total power of 1.

For more information on how to construct a bi-Gaussian Doppler structure, see `doppler`.

```
fd = 10; % Chosen maximum Doppler shift for simulation
sGauss1 = 2.0;
fGauss1 = -5.0;
dopplerComp1 = doppler('BiGaussian', ...
    'NormalizedStandardDeviations', [sGauss1/fd 1/sqrt(2)], ...
    'NormalizedCenterFrequencies',  [fGauss1/fd 0], ...
    'PowerGains',                   [0.5        0])
```

```
dopplerComp1 = struct with fields:
                    SpectrumType: 'BiGaussian'
    NormalizedStandardDeviations: [0.2000 0.7071]
     NormalizedCenterFrequencies: [-0.5000 0]
                     PowerGains: [0.5000 0]
```

To simulate the first magneto-ionic component, we construct a single-path Rayleigh channel System object `chanComp1` with a frequency-shifted Gaussian Doppler spectrum specified by the Doppler structure `dopplerComp1`. The average path power gain of the channel is 1 (0 dB).

```
chanComp1 = comm.RayleighChannel( ...
    'SampleRate',          Rs, ...
    'MaximumDopplerShift', fd, ...
    'DopplerSpectrum',     dopplerComp1, ...
    'RandomStream',        'mt19937ar with seed', ...
    'Seed',                99, ...
    'PathGainsOutputPort', true)
```

```
chanComp1 =
  comm.RayleighChannel with properties:

             SampleRate: 9600
             PathDelays: 0
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 10
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: true

  Show all properties
```

Similarly, the second magneto-ionic component has a Gaussian Doppler spectrum with standard deviation `sGauss2`, frequency shift `fGauss2`, and power gain `gGauss2`. A bi-Gaussian Doppler

structure `dopplerComp2` is constructed such that the second Gaussian function has a zero power gain (its standard deviation and center frequency are hence irrelevant, and take on default values), while the first Gaussian function has a normalized standard deviation `sGauss2/fd` and a normalized frequency shift `fGauss2/fd` (again its power gain is irrelevant).

```
sGauss2 = 1.0;
fGauss2 = 4.0;
dopplerComp2 = doppler('BiGaussian', ...
    'NormalizedStandardDeviations', [sGauss2/fd 1/sqrt(2)], ...
    'NormalizedCenterFrequencies',  [fGauss2/fd 0], ...
    'PowerGains',                   [0.5        0])
```

```
dopplerComp2 = struct with fields:
                    SpectrumType: 'BiGaussian'
    NormalizedStandardDeviations: [0.1000 0.7071]
     NormalizedCenterFrequencies: [0.4000 0]
                     PowerGains: [0.5000 0]
```

To simulate the second magneto-ionic component, we construct a single-path Rayleigh channel System object `chanComp2` with a frequency-shifted Gaussian Doppler spectrum specified by the Doppler structure `dopplerComp2`.

```
chanComp2 = comm.RayleighChannel( ...
    'SampleRate',          Rs, ...
    'MaximumDopplerShift', fd, ...
    'DopplerSpectrum',     dopplerComp2, ...
    'RandomStream',        'mt19937ar with seed', ...
    'Seed',                999, ...
    'PathGainsOutputPort', true)
```

```
chanComp2 =
  comm.RayleighChannel with properties:

             SampleRate: 9600
             PathDelays: 0
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 10
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: true

  Show all properties
```

We compute in the loop below the output to the Watterson channel in response to an input signal, and store it in `y`. In obtaining `y`, the function call on `chanComp1` emulates the effect of the first magneto-ionic component, while the function call on `chanComp2` emulates the effect of the second component.

To obtain the desired power gains, `gGauss1` and `gGauss2`, of each magneto-ionic component, we need to scale the output signal for each magneto-ionic component by their corresponding amplitude gains, `sqrt(gGauss1)` and `sqrt(gGauss2)`.

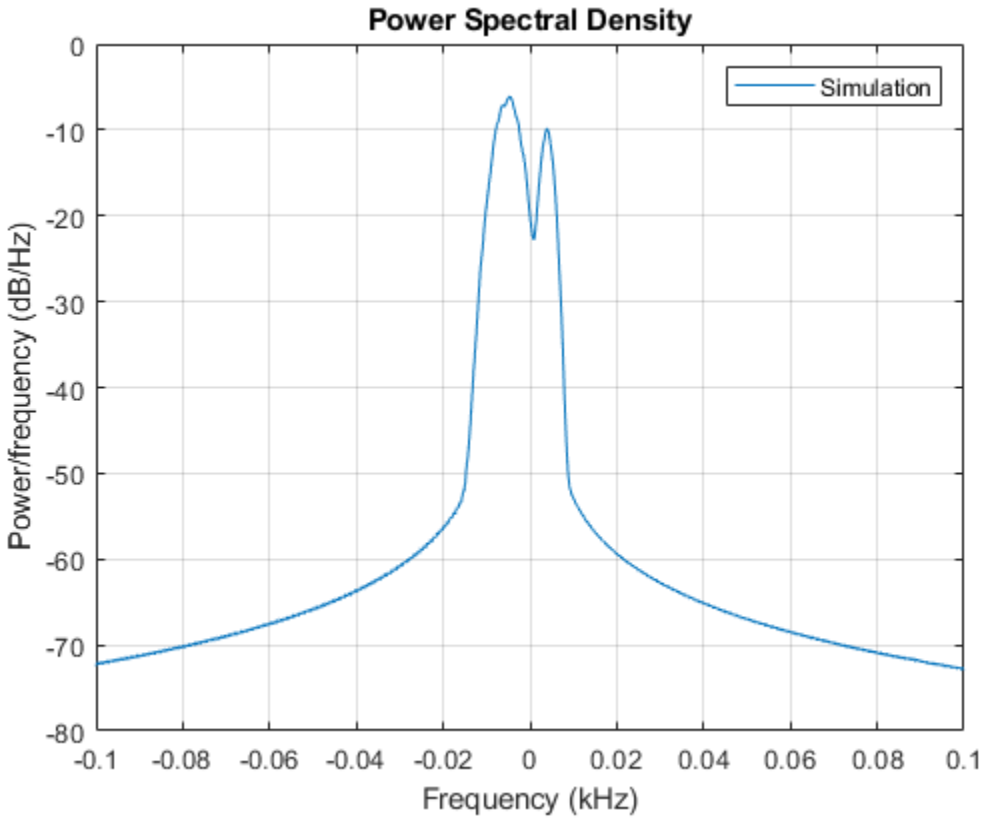
Due to the low Doppler shifts found in HF environments and the fact that the bi-Gaussian Doppler spectrum is combined from two objects, obtaining measurements for the Doppler spectrum using the built-in visualization of the System objects is not appropriate. Instead, we store the channel's complex

path gains and later compute the Doppler spectrum for each path at the command line. In the loop below, the channel's complex path gains are obtained by summing (after scaling by the corresponding amplitude gains) the complex path gains associated with each magneto-ionic component, and then stored in g.

```
gGauss1 = 1.2;       % Power gain of first component
gGauss2 = 0.25;      % Power gain of second component

Ns     = 2e6;        % Total number of channel samples
frmLen = 1e3;        % Number of samples per frame
numFrm = Ns/frmLen;  % Number of frames

[y, g] = deal(zeros(Ns, 1));
for frmIdx = 1:numFrm
    x = qpskMod(randi([0 M-1], frmLen, 1));
    [y1, g1] = chanComp1(x);
    [y2, g2] = chanComp2(x);
    y(frmLen*(frmIdx-1)+(1:frmLen)) = sqrt(gGauss1) * y1 ...
        + sqrt(gGauss2) * y2;
    g(frmLen*(frmIdx-1)+(1:frmLen)) = sqrt(gGauss1) * g1 ...
        + sqrt(gGauss2) * g2;
end
```

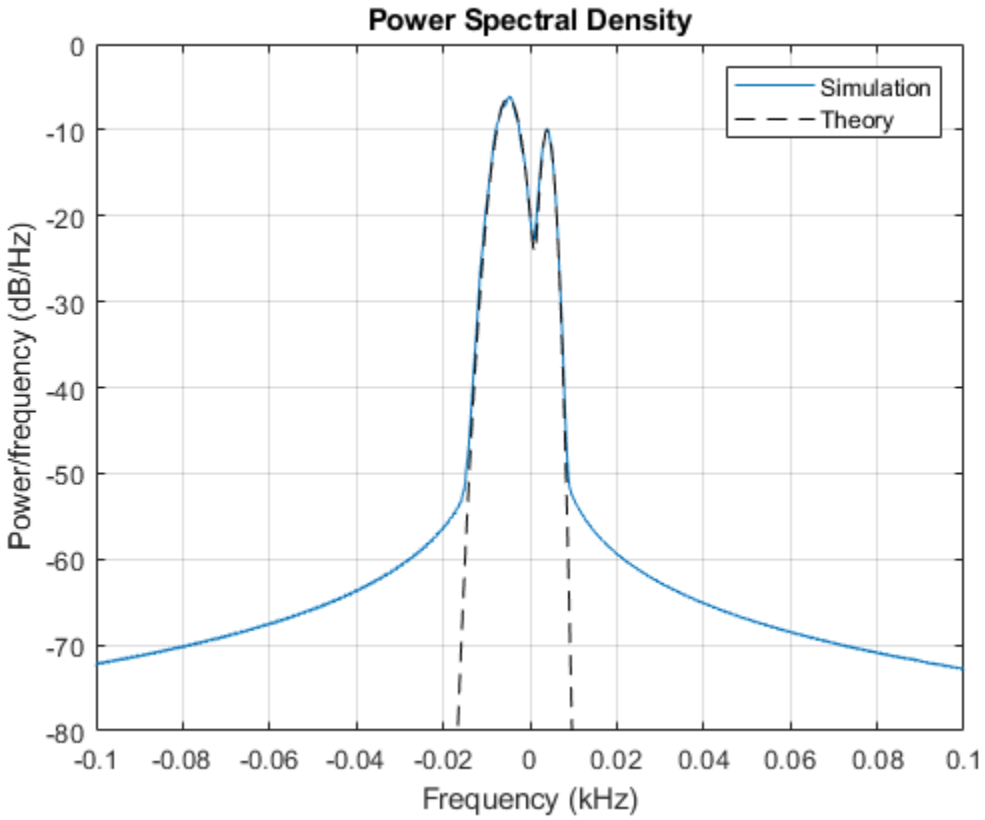
The Doppler spectrum is estimated from the complex path gains and plotted.

```
hFig = figure;
pwelch(g, hamming(Ns/100), [], [], Rs, 'centered');
axis([-0.1 0.1 -80 0]);
legend('Simulation');
```



The theoretical bi-Gaussian Doppler spectrum is overlaid to the estimated Doppler spectrum. We observe a good fit between both.

```
f = -(Rs/2):0.1:(Rs/2);
Sd = gGauss1 * 1/sqrt(2*pi*sGauss1^2) * exp(-(f-fGauss1).^2/(2*sGauss1^2)) ...
    + gGauss2 * 1/sqrt(2*pi*sGauss2^2) * exp(-(f-fGauss2).^2/(2*sGauss2^2));

hold on;
plot(f(Sd>0)/1e3, 10*log10(Sd(Sd>0)), 'k--');
legend('Simulation', 'Theory');
```



ITU-R F.1487 Low Latitudes, Moderate Conditions (LM) Channel Model

Recommendation ITU-R F.1487 specifies simplified channel models used in the quantitative testing of HF modems. These models consist of two independently fading paths with equal power. On each path, the two magneto-ionic components are assumed to have zero frequency shift and equal variance: hence the bi-Gaussian Doppler spectrum on each tap reduces to a single Gaussian Doppler spectrum, and the envelope of the complex fading process is Rayleigh-distributed.

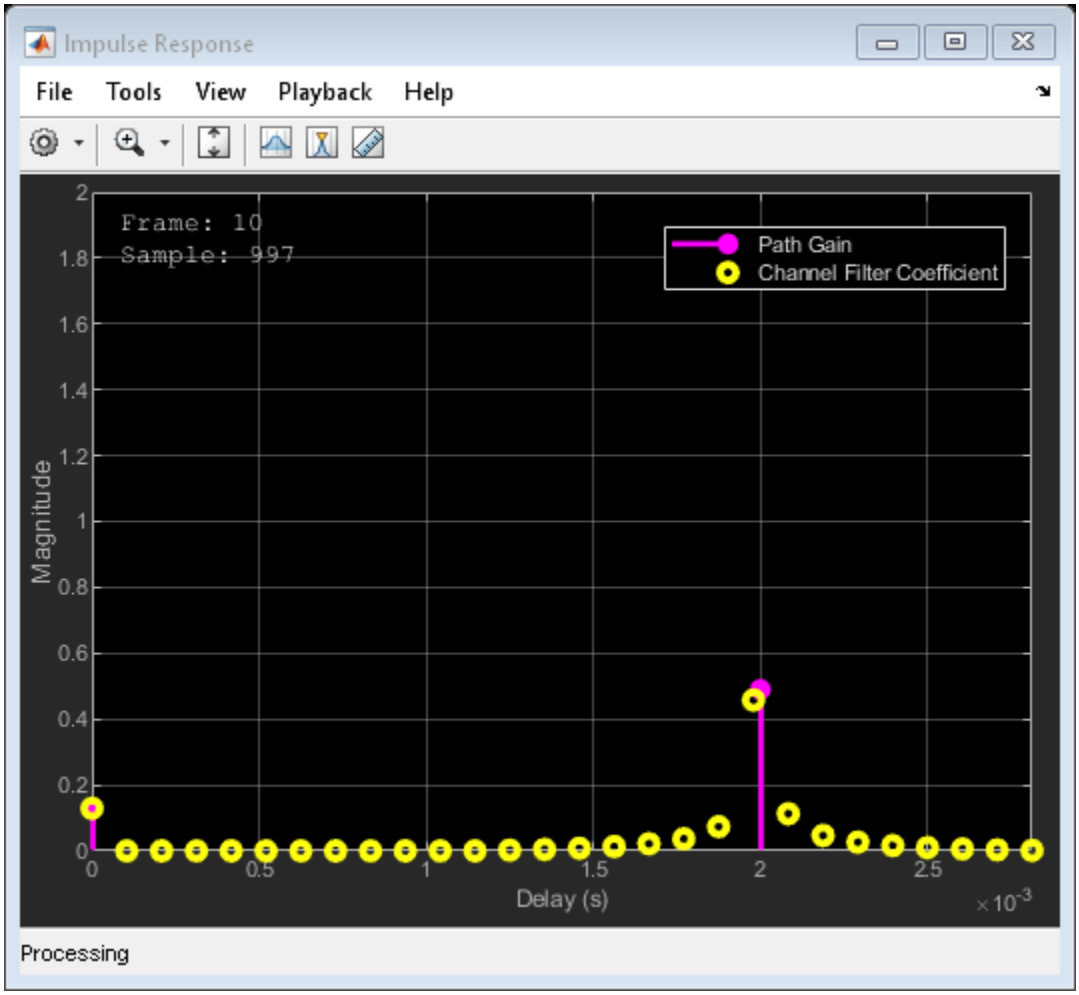
Below, we construct a channel object according to the Low Latitudes, Moderate Conditions (LM) channel model specified in Annex 3 of ITU-R F.1487, using the `stdchan` function. The path delays are 0 and 2 ms. The frequency spread, defined as twice the standard deviation of the Gaussian Doppler spectrum, is 1.5 Hz. The Gaussian Doppler spectrum structure is hence constructed with a normalized standard deviation of (1.5/2)/ `fd`, where `fd` is 1 Hz (type `help doppler` for more information). When using `stdchan` to construct ITU-R HF channel models, the maximum Doppler shift must be set to 1 Hz: this ensures that the Gaussian Doppler spectrum of the constructed channel has the correct standard deviation.

```
close(hFig);

fd = 1;
chanLM = stdchan('iturHFLM', Rs, fd);
chanLM.RandomStream = 'mt19937ar with seed';
chanLM.Seed = 9999;
chanLM.PathGainsOutputPort = true;
chanLM.Visualization = 'Impulse response'
```

```
chanLM =
  comm.RayleighChannel with properties:

            SampleRate: 9600
            PathDelays: [0 0.0020]
      AveragePathGains: [0 0]
     NormalizePathGains: true
    MaximumDopplerShift: 1
        DopplerSpectrum: [1x1 struct]
      ChannelFiltering: true
    PathGainsOutputPort: true

  Show all properties
```

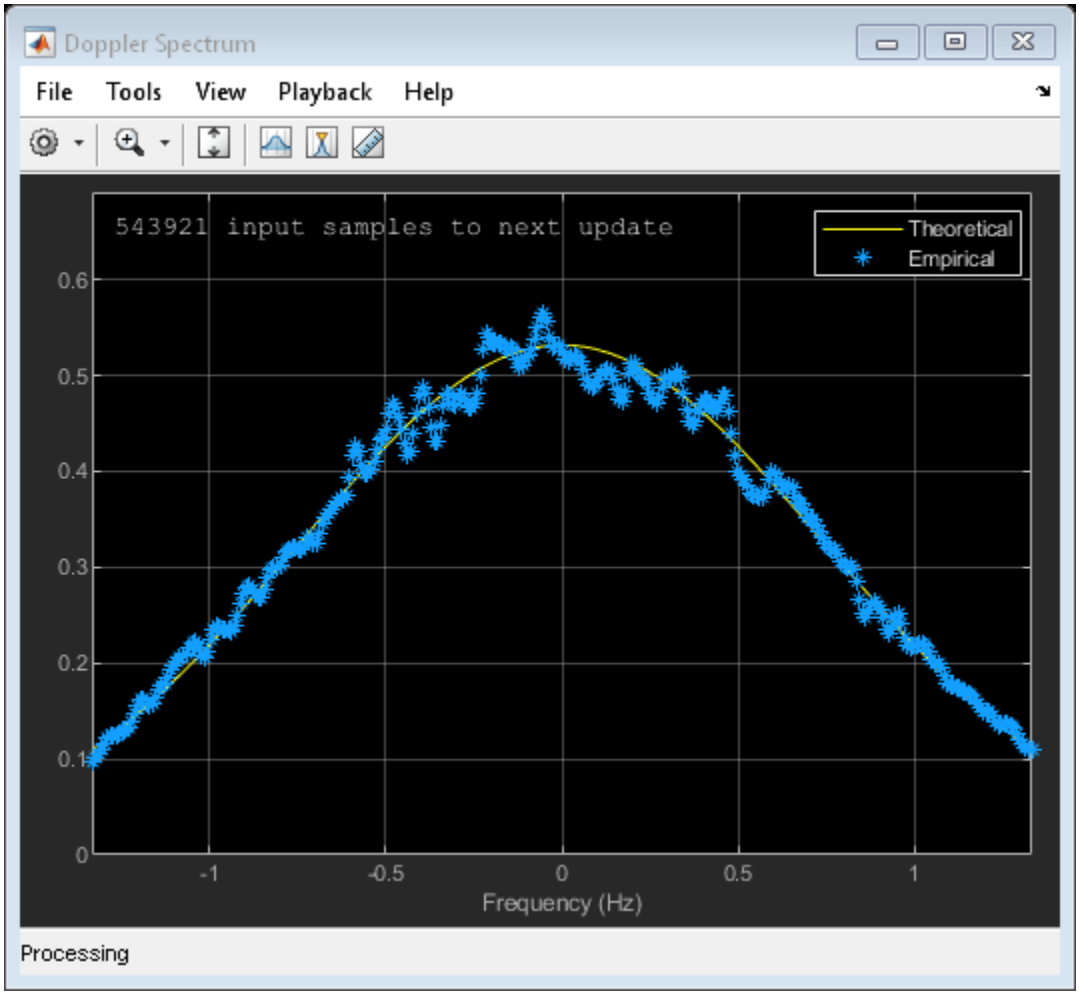
We have turned on the impulse response visualization in the Rayleigh channel System object. The code below simulates the LM channel and visualizes its bandlimited impulse response. By default, the channel responses for one of every four samples are visualized for faster simulation. In other words, for a frame of length 1000, the responses for the 1st, 5th, 9th, ..., 997th samples are shown. To observe the response for every sample, set the `SamplesToDisplay` property of `chanLM` to `'100%'`.

```
numFrm = 10;       % Number of frames
for frmIdx = 1:numFrm
    x = qpskMod(randi([0 M-1], frmLen, 1));
    chanLM(x);
end
```



We now turn on the Doppler spectrum visualization for the channel object to observe the theoretical and empirical Gaussian Doppler spectra for the first discrete path. Due to the very low Doppler shift, it may take a while to have the empirical spectrum converge to the theoretical spectrum.

```
release(chanLM);
chanLM.Visualization = 'Doppler spectrum';

frmLen = 2e6;      % Number of samples per frame
numFrm = 80;       % Number of frames
for frmIdx = 1:numFrm
    x = qpskMod(randi([0 M-1], frmLen, 1));
    chanLM(x);
end
```



Selected Bibliography

1 - Recommendation ITU-R F.1487, "Testing of HF modems with bandwidths of up to about 12 kHz using ionospheric channel simulators," 2000.

2 - C. C. Watterson, J. R. Juroshek, and W. D. Bensema, "Experimental confirmation of an HF channel model," IEEE® Trans. Commun. Technol., vol. COM-18, no. 6, Dec. 1970.

GSM, CDMA and WiMAX Channel Models

This example shows how to simulate multipath fading channels defined for GSM/EDGE [1 2], CDMA [3], and WiMAX [4] wireless standards. The example uses the Rayleigh and MIMO fading channel System objects™ from Communications Toolbox™ to simulate and visualize the channels.

GSM Channel Model

GSM (Global System for Mobile Communications) is the global standard for 2G mobile communications. The multipath fading channel for GSM was defined in [1 2] for different communication scenarios including rural area (RAx), hilly terrain (HTx), urban area (TUx). Each scenario was assigned a specific power delay profile (PDP) and Doppler spectrum. In this example, we simulate the hilly terrain scenario (HTx) with 12 taps. We pass GMSK modulated signals through the fading channel and observe its impulse response.

```
% Set random number generator for repeatability
rng('default');
```

Create a GMSK modulator using the `comm.GMSKModulator` object and use it to modulate randomly generated bits. This object is to illustrate that the GMSK modulation is used in the GSM system.

```
gmskMod = comm.GMSKModulator( ...
    'BitInput', true, ...
    'SamplesPerSymbol', 8);

% Modulate random bits using the GMSK object
x = gmskMod(randi([0 1], 1e4, 1));
```

Assume mobile speed at 120 km/h. Calculate the Doppler shift at the carrier frequency of 1.8 GHz.

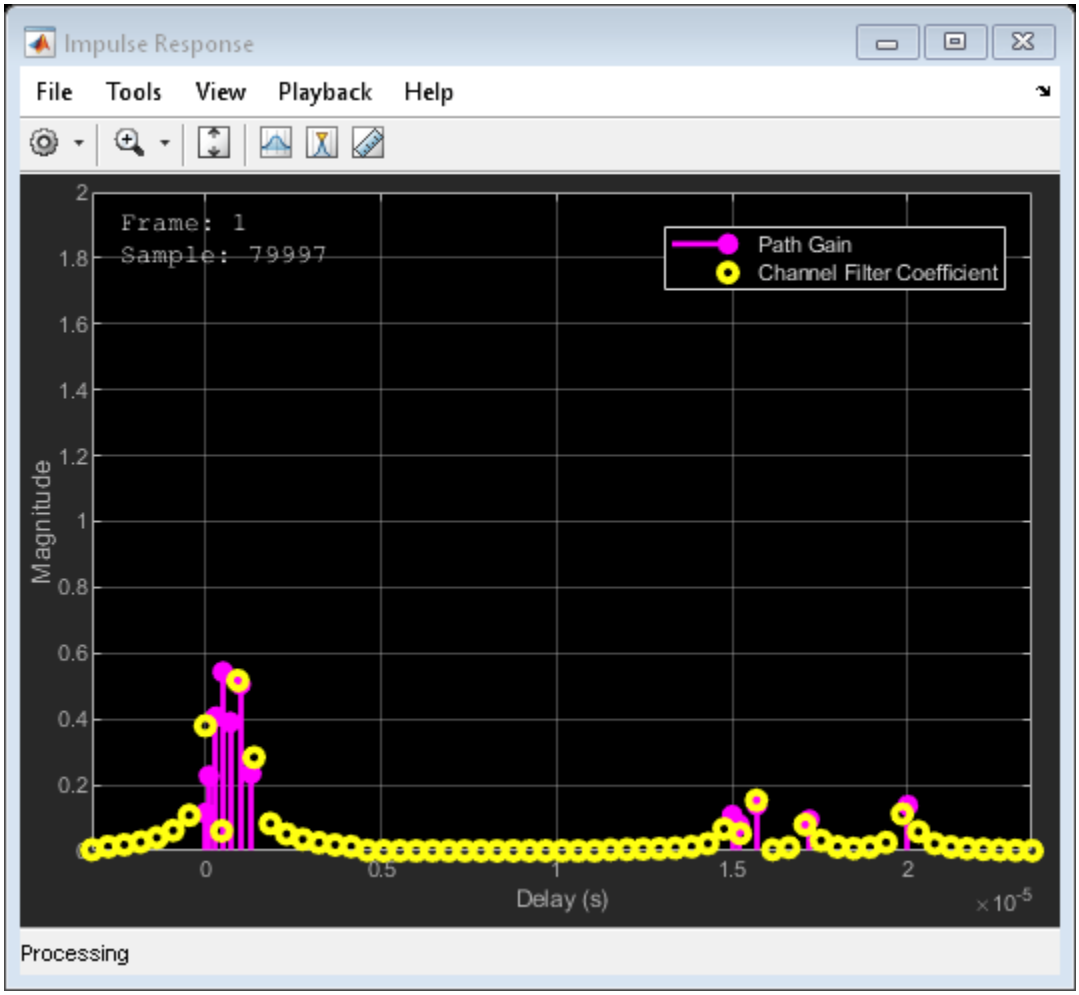
```
v = 120*1e3/3600;                   % Mobile speed (m/s)
fc = 1.8e9;                         % Carrier frequency
fd = v*fc/physconst('lightspeed');  % Maximum Doppler shift
```

To simulate the fading channel for HTx, we can configure a `comm.RayleighChannel` object following the PDP specification in [1 2]. Alternatively, we can use the `stdchan` function to create the desired `comm.RayleighChannel` object, given the scenario input 'gsmHTx12c1'. So we do not have to refer to [1 2] for PDP and Doppler spectrum specifications.

```
Rsym = 270.833e3; % GSM symbol rate
Rsamp = gmskMod.SamplesPerSymbol * Rsym; % GSM sample rate
gsmChan = stdchan('gsmHTx12c1', Rsamp, fd);
```

We turn on the impulse response visualization for the channel object and send the GMSK modulated data through it. You can observe that the path (tap) delays last over 5 samples. The first 7 and last 5 taps can be grouped into two different clusters. In that sense, the channel characterizes two dominant paths from the transmitter to the receiver with scattering. You can also observe that the impulse response changes reasonably fast at this mobile speed of 120 km/h.

```
gsmChan.Visualization = 'Impulse response';
gsmChan(x);
```



CDMA Channel Model

CDMA (Code-Division Multiple Access) is the standard for 3G mobile communications. Like GSM, the multipath fading channel for CDMA was defined in [3] for different communication scenarios with different PDPs and Doppler spectra. In this example, we simulate the typical urban scenario (TUx) with a low mobile speed and visualize the channel's frequency response. The `cdma2000ForwardReferenceChannels` and `cdma2000ForwardWaveformGenerator` functions are used to configure and simulate a CDMA 2000 waveform, which is subsequently transmitted through the fading channel.

```
% Configure a CDMA waveform and change the packet length
config = cdma2000ForwardReferenceChannels('ALL-RC3');
config.NumChips = 1e4;

% Generate a waveform
waveform = cdma2000ForwardWaveformGenerator(config);
```

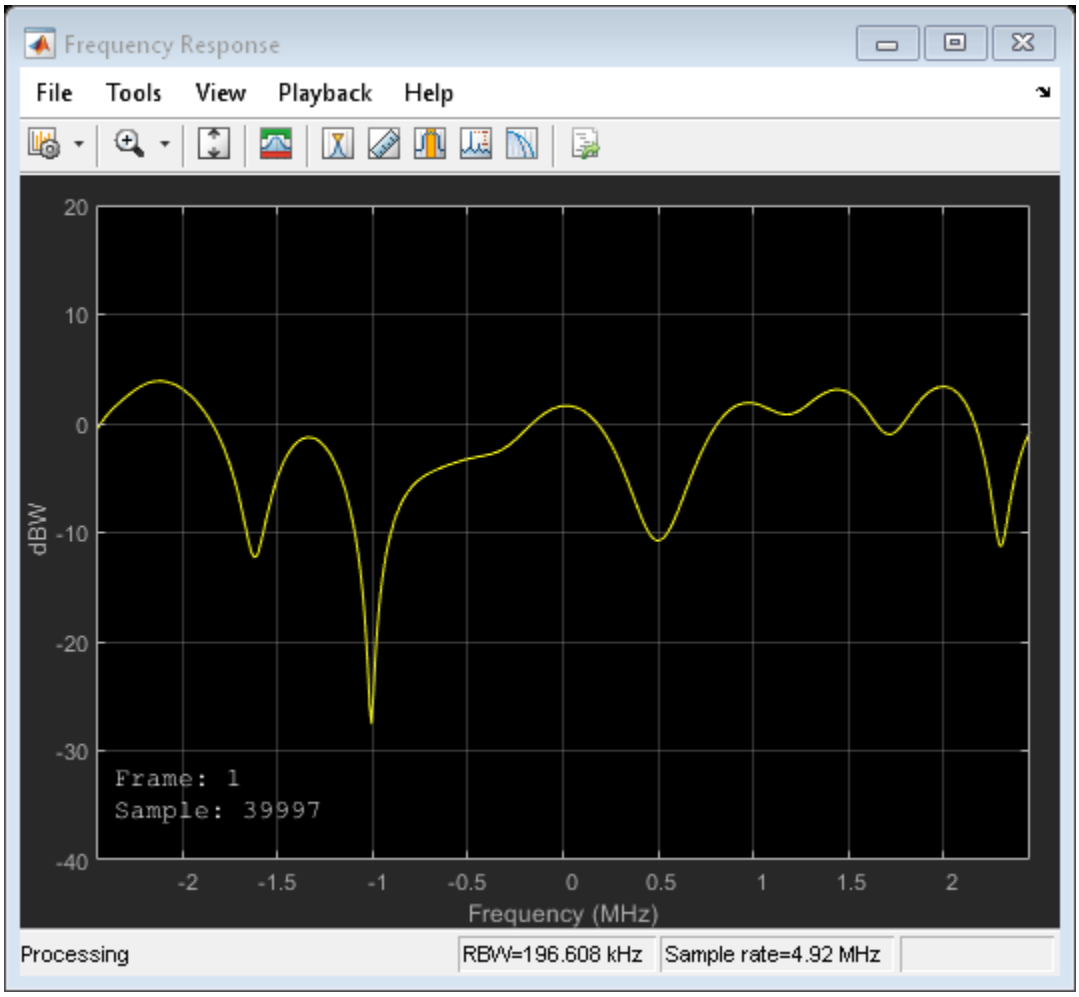
Derive channel sample rate from the waveform configuration. If the SpreadingRate field is 'SR1', it corresponds to a 1.2288 Mcps waveform. If it is 'SR3', it corresponds to a 3.6864 Mcps waveform.

```
Rsprd = str2double(config.SpreadingRate(3)) * 1.2288e6;
Rsamp = Rsprd * config.OversamplingRatio;
```

```
% Assume a human walking speed which is about 5 km/h. Calculate the Doppler
% shift at the carrier frequency of 1.9 GHz.
v = 5*1e3/3600;                     % Mobile speed (m/s)
fc = 1.9e9;                         % Carrier frequency
fd = v*fc/physconst('lightspeed');  % Maximum Doppler shift
```

Again, configure a CDMA channel for TUx using the `stdchan` function. Turn on the channel's frequency response visualization and pass the waveform through it. You can observe the obvious frequency-selectivity of the channel. The frequency response varies slowly at this low mobile speed of 5 km/h.

```
cdmaChan = stdchan('cdmatux', Rsamp, fd);
cdmaChan.Visualization = 'Frequency response';
y = cdmaChan(waveform);
```



WiMAX Channel Model

The WiMAX (IEEE® 802.16) channel models [4] for fixed wireless applications are proposed for scenarios where the cell radius is less than 10 km, the directional antennas at the receiver are installed under-the-eaves/windows or on the rooftop, and the base station (BS) antennas are 15 to 40 m in height. The channel models comprise a set of path loss models including shadowing (suburban,

urban) and a multipath fading model, which describes the multipath delay profile, the K-factor distribution, and the Doppler spectrum. The antenna gain reduction factor, due to the use of directional antennas, is also characterized.

This example uses a MIMO multipath fading channel System object™ `comm.MIMOChannel` with two transmit antennas, one receive antenna, and a rounded Doppler spectrum structure. The modified Stanford University Interim (SUI) channel models consist of a set of 6 typical channels used to simulate the IEEE 802.16 channel models (more specifically the 2004 version of the standard for fixed wireless applications). They are proposed for a scenario where: the cell size is 7 km, the BS antenna height is 30 m, the receive antenna height is 6 m, the BS antenna beamwidth is 120 degrees, the receive antenna is either omnidirectional or directional (30 degrees), and only vertical polarization is used.

Each modified SUI channel model has three taps. Each tap is characterized by a relative delay (with respect to the first path delay), a relative power, a Rician K-factor, and a maximum Doppler shift. Two sets of relative powers are specified for each channel model: one for an omnidirectional antenna, and one for a 30 degrees directional antenna. Furthermore, for each set of relative powers, two different K-factors are specified, a K-factor for 90% cell coverage, and a K-factor for 75% cell coverage. Hence, each of the 6 modified SUI channel models comprises parameters for four distinct scenarios. Each modified SUI channel model is further assigned an antenna correlation, defined as the envelope correlation coefficient between signals received at different antenna elements.

The code below constructs a MIMO fading channel System object according to the modified SUI-1 channel model, for an omnidirectional antenna and 90% cell coverage.

The channel model has 3 paths: the first path is Rician while the remaining two are Rayleigh. Each path has a rounded Doppler spectrum for its diffuse component: the parameters are as specified in the doppler('Rounded') structure. While different maximum Doppler shifts are specified for each path in [4], we use the maximum value of the Doppler shifts for all paths.

We use 2 transmit antennas and 1 receive antenna. Similar to Appendix B of [4], the correlation coefficient between the two links on each path is taken equal to the antenna spatial correlation. The correlation coefficient is 0.7.

The sample rate for a WiMAX system is 1.429, 2.857, 5.714, 11.429 or 22.857 MHz. At such rates with a small Doppler shift, we need many samples and long simulation time to sufficiently exhibit the channel statistical characteristics. To avoid that, we arbitrarily choose a smaller sample rate of 0.1 MHz. You can increase the sample rate, `Rsamp`, and number of samples, `Ns`, at the same time to see the similar statistical results.

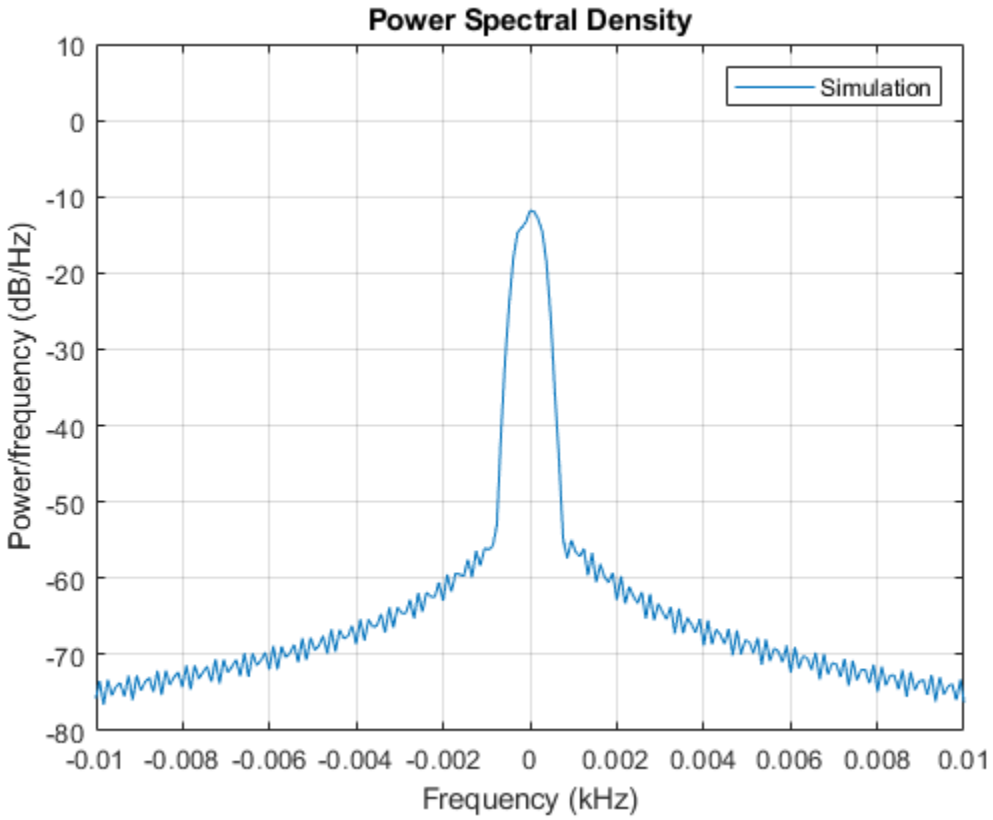
```
Rsamp = 0.1e6;
Ns = 3e6;

wimaxChan = comm.MIMOChannel( ...
    'SampleRate', Rsamp, ...
    'PathDelays', [0 0.4 0.9]*1e-6, ...
    'AveragePathGains', [0 -15 -20], ...
    'FadingDistribution', 'Rician', ...
    'KFactor', 4, ...
    'MaximumDopplerShift', .5, ...
    'DopplerSpectrum', doppler('Rounded'), ...
    'TransmitCorrelationMatrix', [1 0.7; 0.7 1], ...
    'ReceiveCorrelationMatrix', 1, ...
    'PathGainsOutputPort', true);
```

The code below simulates the modified SUI-1 channel model with a long QPSK modulated frame input.

```
Nt = size(wimaxChan.TransmitCorrelationMatrix, 1);
x = pskmod(randi([0 3], Ns, Nt), 4);
[~, g] = wimaxChan(x);
```

The Doppler spectrum of the 1st link of the second path is estimated from the complex path gains and plotted.

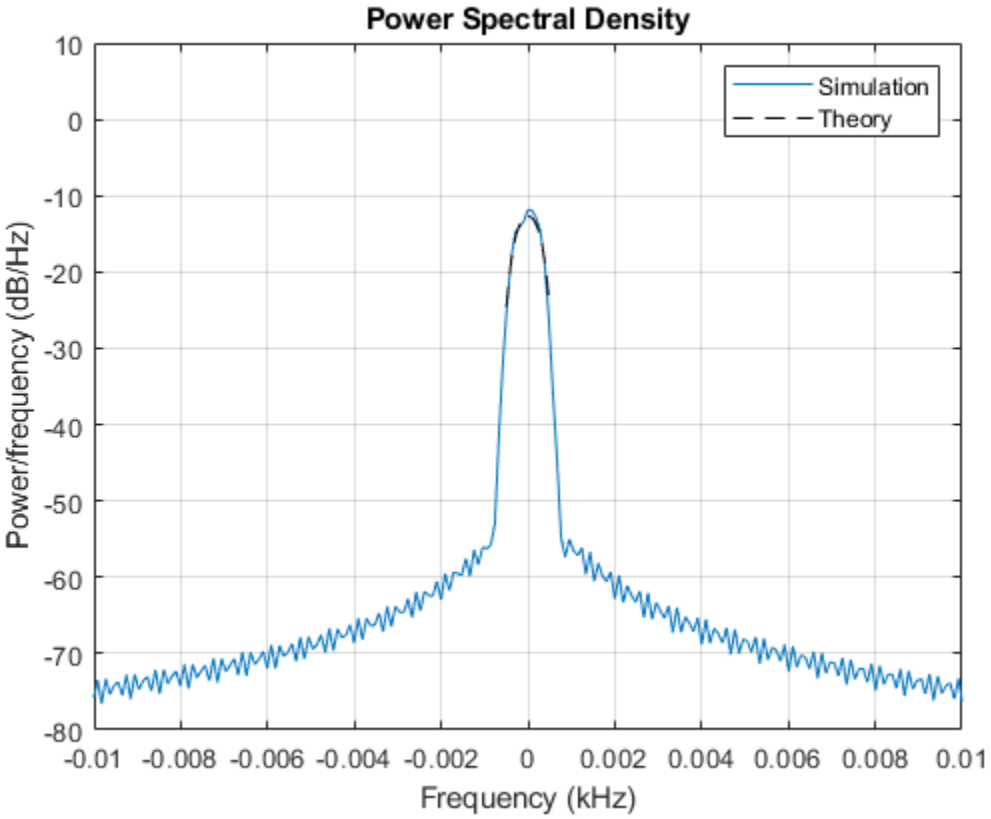
```
figure;
win = hamming(Ns/5);
Noverlap = Ns/10;
pwelch(g(:,2,1),win,Noverlap,[],Rsamp,'centered')
axis([-0.1/10 0.1/10 -80 10]);
legend('Simulation');
```



The theoretical rounded Doppler spectrum is overlaid on the estimated Doppler spectrum. We observe a good fit between them.

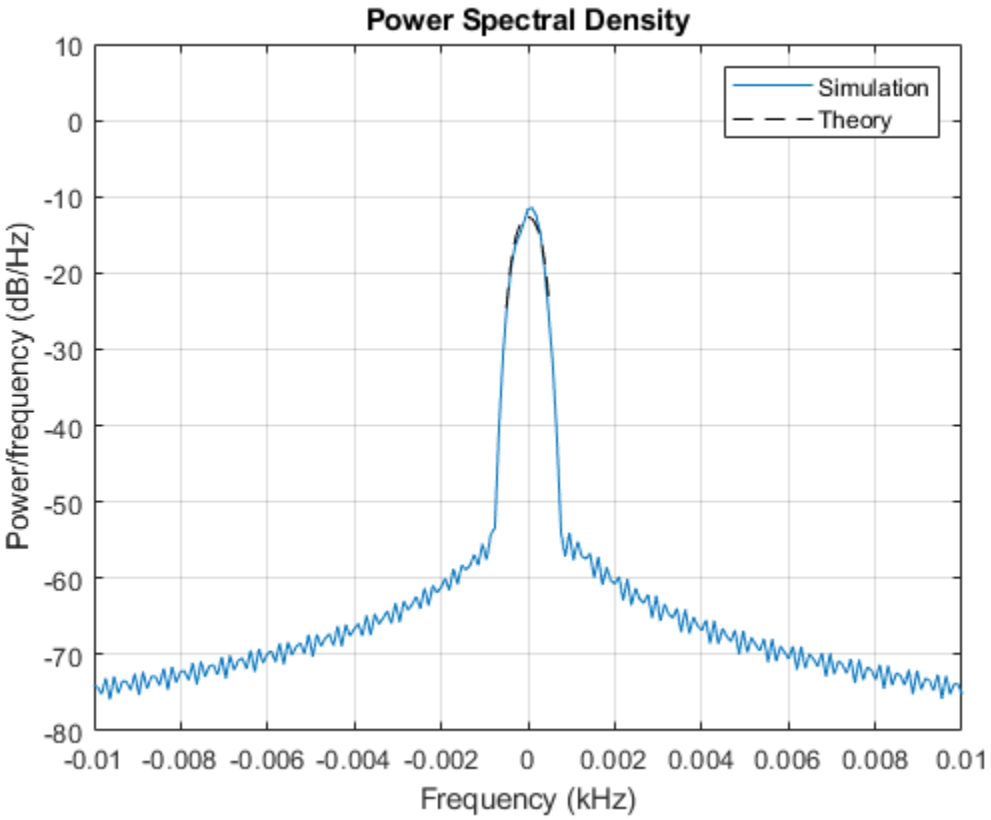
```
fd = wimaxChan.MaximumDopplerShift;
f  = -fd:0.01:fd;
a  = wimaxChan.DopplerSpectrum.Polynomial;      % Parameters of the rounded Doppler spectrum
Sd = 1/(2*fd*(a(1)+a(2)/3+a(3)/5))*(a(1)+a(2)*(f/fd).^2+a(3)*(f/fd).^4);
Sd = Sd*10^(wimaxChan.AveragePathGains(2)/10);  % Scaling by average path power

hold on;
```

```
plot(f(Sd>0)/1e3,10*log10(Sd(Sd>0)),'k--');
legend('Simulation','Theory');
```



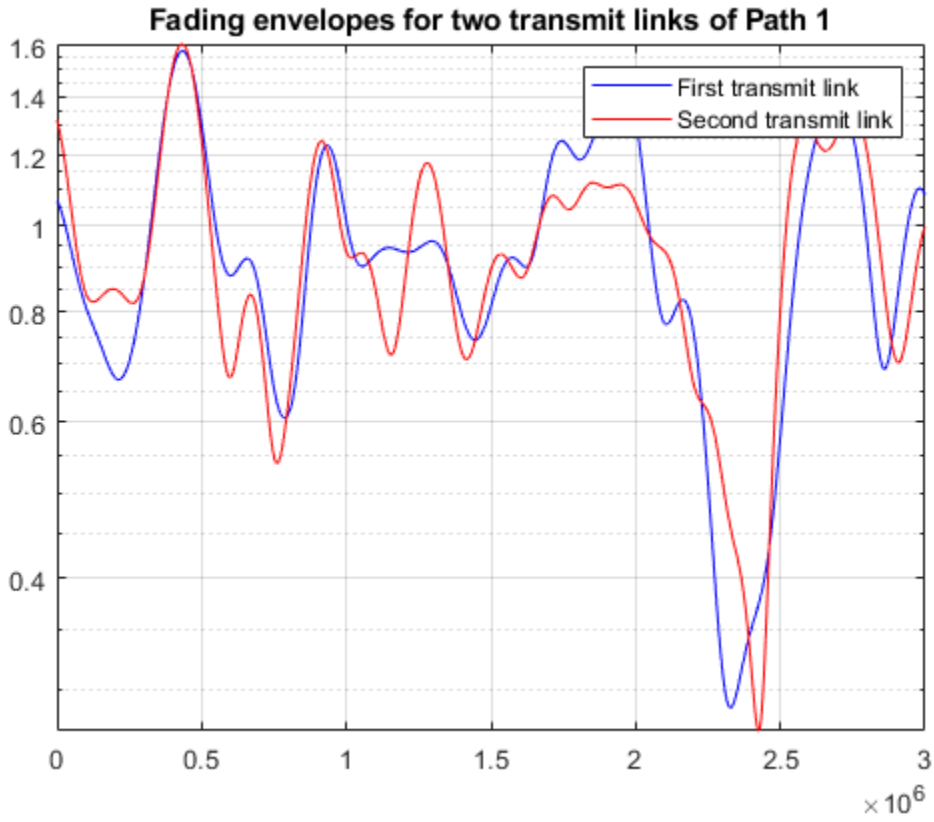
The Doppler spectrum for the 2nd link of the 2nd path is also estimated and compared to the theoretical spectrum. We also observe a good fit between them.

```
figure;
pwelch(g(:,2,2),win,Noverlap,[],Rsamp,'centered')
axis([-0.1/10 0.1/10 -80 10]);
legend('Simulation');
hold on;
plot(f(Sd>0)/1e3,10*log10(Sd(Sd>0)),'k--');
legend('Simulation','Theory');
```

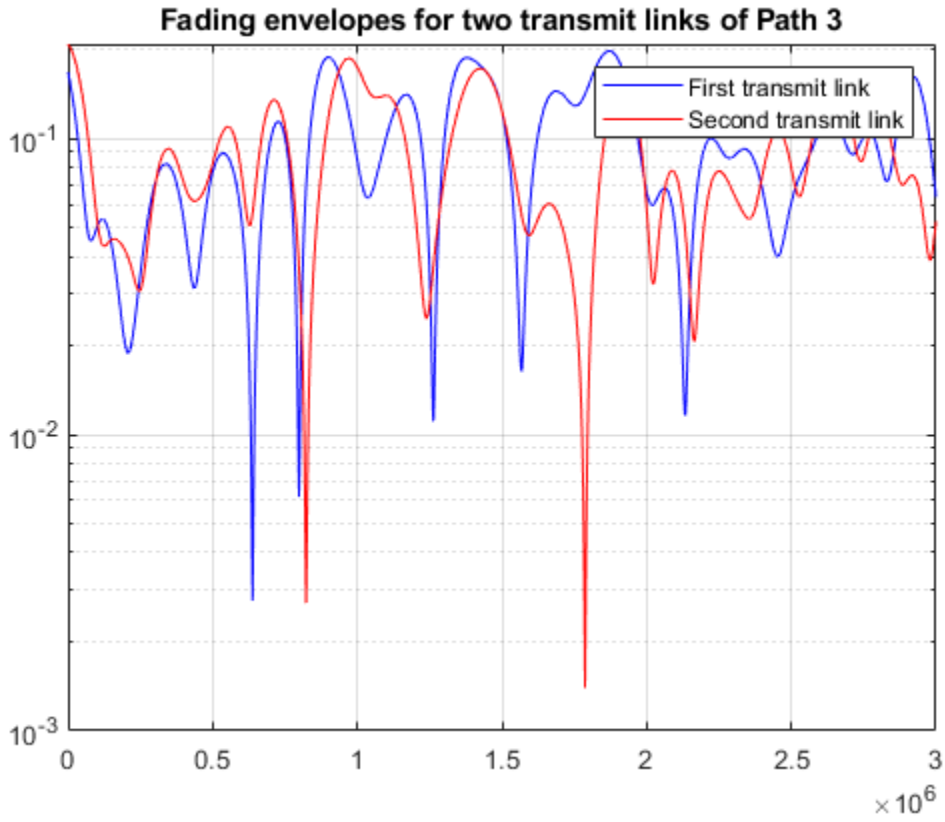


For each path, we plot the fading envelope waveforms of both transmit links. We can observe a correlation between the fading envelopes.

```
figure;
semilogy(abs(g(:,1,1)),'b');
hold on;
grid on;
semilogy(abs(g(:,1,2)),'r');
legend('First transmit link','Second transmit link');
title('Fading envelopes for two transmit links of Path 1');

figure;
semilogy(abs(g(:,2,1)),'b');
hold on;
grid on;
semilogy(abs(g(:,2,2)),'r');
legend('First transmit link','Second transmit link');
title('Fading envelopes for two transmit links of Path 2');

figure;
semilogy(abs(g(:,3,1)),'b');
hold on;
grid on;
semilogy(abs(g(:,3,2)),'r');
legend('First transmit link','Second transmit link');
title('Fading envelopes for two transmit links of Path 3');
```


Fading envelopes for two transmit links of Path 1
First transmit link
Second transmit link
1.6
1.4
1.2
1
0.8
0.6
0.4
0
0.5
1
1.5
2
2.5
3
×10^6


Fading envelopes for two transmit links of Path 2
First transmit link
Second transmit link
10^0
10^-1
10^-2
10^-3
0
0.5
1
1.5
2
2.5
3
×10^6



We compute the spatial correlation matrices for each path. We observe that they show a match with the theoretical values Rt. Note that `corrcoef` function estimate can be improved if `Ns` is increased.

```
TxCorrMatrixPath1 = corrcoef(g(:,1,1),g(:,1,2)).'
TxCorrMatrixPath2 = corrcoef(g(:,2,1),g(:,2,2)).'
TxCorrMatrixPath3 = corrcoef(g(:,3,1),g(:,3,2)).'
```

```
TxCorrMatrixPath1 =

   1.0000 + 0.0000i   0.7537 + 0.0388i
   0.7537 - 0.0388i   1.0000 + 0.0000i


TxCorrMatrixPath2 =

   1.0000 + 0.0000i   0.7605 + 0.2331i
   0.7605 - 0.2331i   1.0000 + 0.0000i


TxCorrMatrixPath3 =

   1.0000 + 0.0000i   0.7113 + 0.1282i
   0.7113 - 0.1282i   1.0000 + 0.0000i
```

Selected Bibliography

1. 3GPP TS 05.05 V8.20.0 (2005-11): 3rd Generation Partnership Project; Technical Specification Group GSM/EDGE Radio Access™ Network; Radio transmission and reception (Release 1999).
2. 3GPP TS 45.005 V7.9.0 (2007-2): 3rd Generation Partnership Project; Technical Specification Group GSM/EDGE Radio Access Network; Radio transmission and reception (Release 7).
3. 3GPP TR 25.943 V6.0.0 (2004-12): 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Deployment aspects (Release 6).
4. IEEE 802.16 Broadband Wireless Access Working Group, "Channel models for fixed wireless applications", IEEE 802.16a-03/01, 2003-06-27.

GSM Multiframe Generation in Simulink

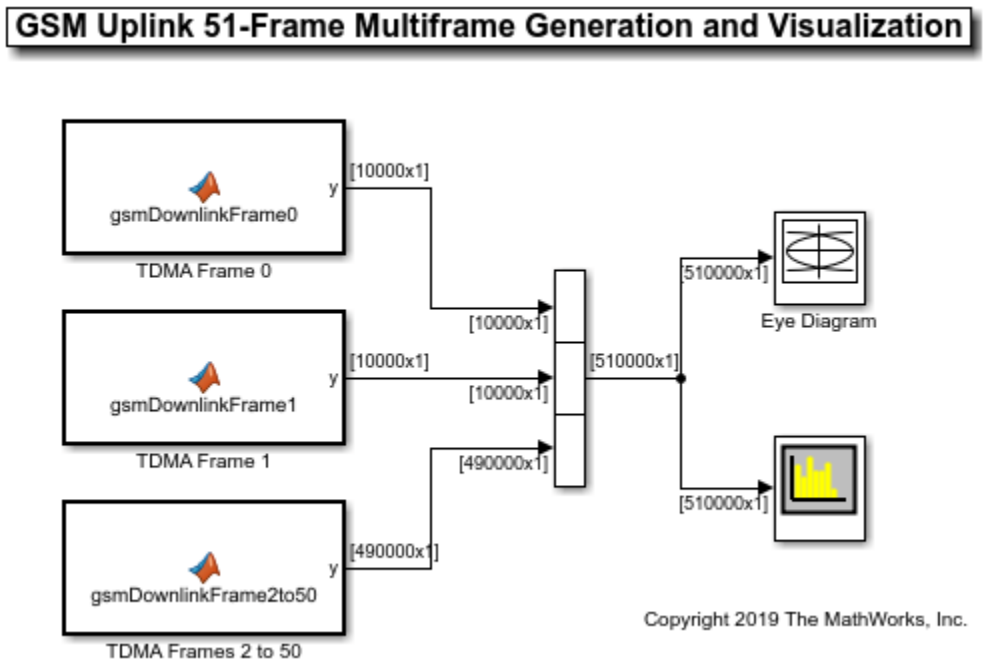
This example shows how to model a GSM® waveform generator to generate a 51-frame multiframe in Simulink®. For more information see GSM TDMA Frame Parameterization for Waveform Generation Example.

Introduction

This model generates a 51-frame GSM downlink multiframe with the following configuration. Downlink frames can carry normal burst (NB), frequency correction burst (FB), synchronization burst (SB) and dummy burst. The first frame is [FB NB NB NB NB Dummy NB NB], the second frame is [SB NB NB NB NB Dummy NB NB], and the next 49 frames are [NB NB NB NB NB Dummy NB NB]. Repeat this structure 3 times.

```
cfg1 =

  gsmDownlinkConfig with properties:

          BurstType: [FB    NB    NB    NB    NB    Dummy    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

   Read-only properties:
    No properties.


cfg2 =

  gsmDownlinkConfig with properties:

          BurstType: [SB    NB    NB    NB    NB    Dummy    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

   Read-only properties:
    No properties.


cfg3 =

  gsmDownlinkConfig with properties:

          BurstType: [NB    NB    NB    NB    NB    Dummy    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

  Read-only properties:
    No properties.
```



GSM 51-frame Multiframe Generation

Double click the `TDMA Frame 0` block. The `gsmDownlinkFrame0` function uses the `gsmDownlinkConfig` function to configure the GSM downlink TDMA frame for the first frame. The `gsmFrame` function generates the samples of the frame. Double click the `TDMA Frame 2 to 50` block. This block generates 49 frames at once using the `y = gsmFrame(cfg,49)` function call.

Setup Model

The GSM standard [1] specifies the symbol rate as R = 1625e3/6 symbols per second. Set the `gsmDownlinkWaveform` blocks' sample time to match the GSM specifications. Use the `gsmInfo` function to get information on the generated waveform based on the configuration object, cfg.

```
wfInfo =

  struct with fields:

              SymbolRate: 2.7083e+05
              SampleRate: 2.1667e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
```

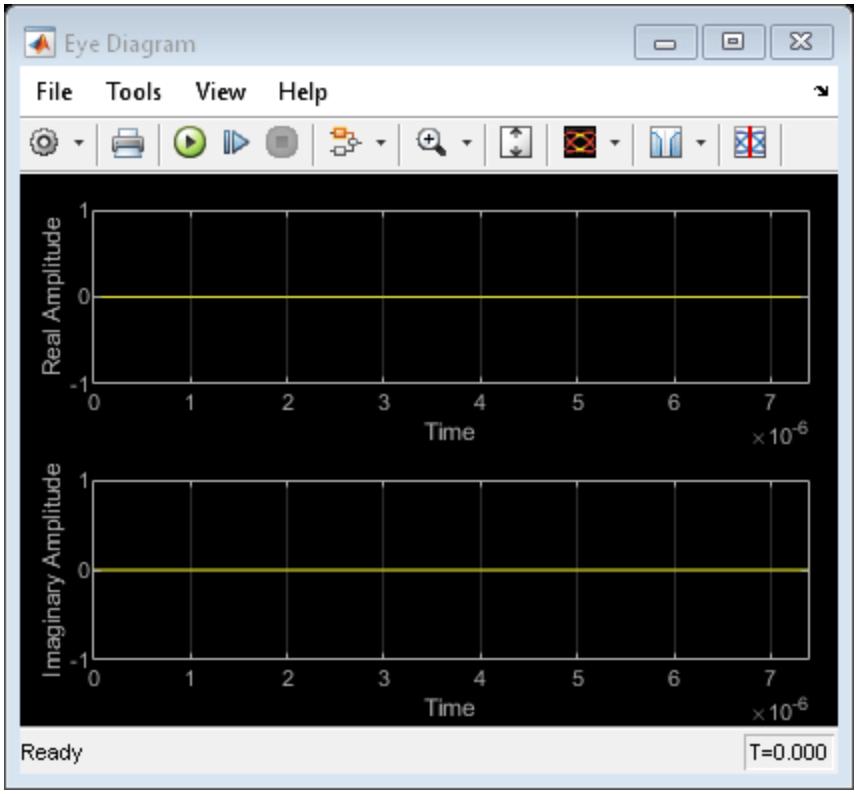
```
BurstLengthInSamples: 1250
FrameLengthInSamples: 10000
```

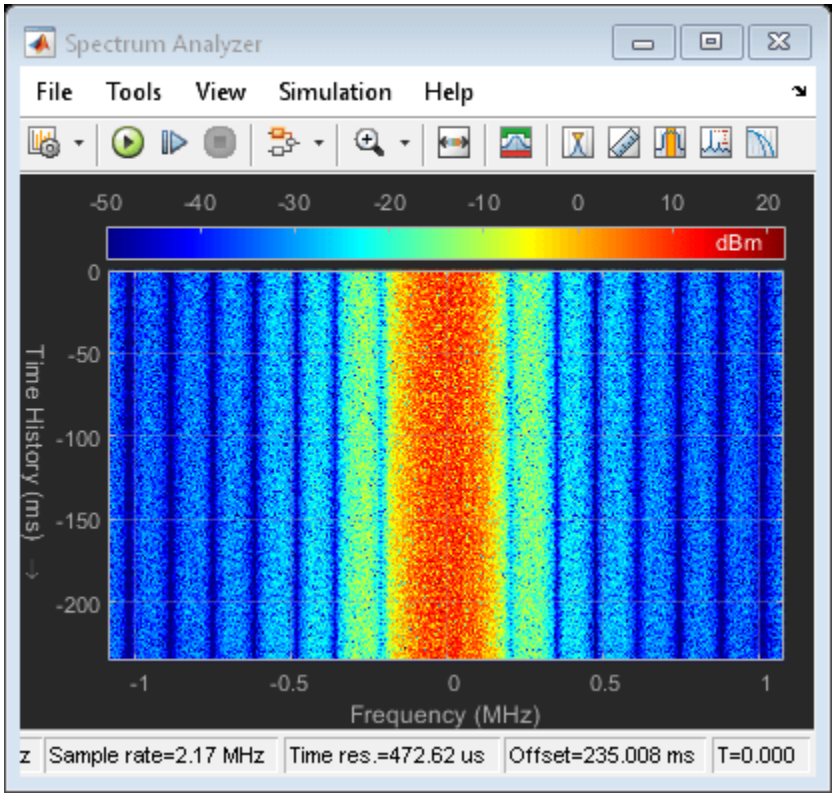
Setup MATLAB Function Block

Select each `MATLAB Function` block and open the Property Inspector. In the Modeling tab, expand the Design group and click on the Property Inspector under the General category. In the Properties tab, make sure that Update method is set to Discrete and Sample Time is set to 51*wfInfo.FrameLengthInSamples/Rs. Close the Property Inspector.

Results

Running the simulation displays the time domain signal and the spectrogram.





Selected Bibliography

1 3GPP TS 45.001, Radio Access Network; Physical layer on the radio path; General description (Release 8)

2 3GPP TS 45.002, Radio Access Network; Multiplexing and multiple access on the radio path (Release 8)

Multipath Fading Channel

This example shows how to use Rayleigh and Rician multipath fading channel System objects and their built-in visualization to model a fading channel. Rayleigh and Rician fading channels are useful models of real-world phenomena in wireless communication. These phenomena include multipath scattering effects, time dispersion, and Doppler shifts that arise from relative motion between the transmitter and receiver.

Processing a signal using a fading channel involves the following steps:

1. Create a channel System object™ that describes the channel that you want to use. A channel object is a type of MATLAB® variable that contains information about the channel, such as the maximum Doppler shift.
2. Adjust properties of the System object, if necessary, to tailor it to your needs. For example, you can change the path delays or average path gains.
3. Apply the channel System object to your signal using the step method, which generates random discrete path gains and filters the input signal.

The characteristics of a channel can be shown with the built-in visualization support of the System object.

Initialization

The following variables control both the Rayleigh and Rician channel objects. By default, the channel is modeled as four fading paths, each representing a cluster of multipath components received at around the same delay.

```
sampleRate500kHz = 500e3; % Sample rate of 500K Hz
sampleRate20kHz  = 20e3; % Sample rate of 20K Hz
maxDopplerShift  = 200; % Maximum Doppler shift of diffuse components (Hz)
delayVector = (0:5:15)*1e-6; % Discrete delays of four-path channel (s)
gainVector  = [0 -3 -6 -9]; % Average path gains (dB)
```

The maximum Doppler shift is computed as v*f/c, where v is the mobile speed, f is the carrier frequency, and c is the speed of light. For example, a maximum Doppler shift of 200 Hz (as above) corresponds to a mobile speed of 65 mph (30 m/s) and a carrier frequency of 2 GHz.

By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 m difference in path length. In some outdoor multipath environments, reflected paths can be up to several kilometers longer than the shortest path. With the path delays specified above, the last path is 4.5 km longer than the shortest path, and thus arrives 15 microseconds later.

Together, the path delays and path gains specify the average delay profile of the channel. Typically, the average path gains decay exponentially with delay (i.e., the dB values decay linearly), but the specific delay profile depends on the propagation environment. In the delay profile specified above, we assume a 3 dB decrease in average power for every 5 microseconds of path delay.

The following variables control the Rician channel System object. The Doppler shift of the specular component is typically smaller than the maximum Doppler shift (above) and depends on the direction of travel of the mobile relative to the direction of the specular component. The K-factor specifies the linear ratio of average received power from the specular component relative to that of the associated diffuse components.

```
KFactor = 10; % Linear ratio of specular power to diffuse power
specDopplerShift = 100; % Doppler shift of specular component (Hz)
```

Creating Channel System Objects

With the parameters specified above, we can now create the `comm.RayleighChannel` and `comm.RicianChannel` System objects. We configure the objects to use their self-contained random stream with a specified seed for path gain generation.

```
% Configure a Rayleigh channel object
rayChan = comm.RayleighChannel( ...
    'SampleRate',sampleRate500kHz, ...
    'PathDelays',delayVector, ...
    'AveragePathGains',gainVector, ...
    'MaximumDopplerShift',maxDopplerShift, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',10, ...
    'PathGainsOutputPort',true);

% Configure a Rician channel object
ricChan = comm.RicianChannel( ...
    'SampleRate',sampleRate500kHz, ...
    'PathDelays',delayVector, ...
    'AveragePathGains',gainVector, ...
    'KFactor',KFactor, ...
    'DirectPathDopplerShift',specDopplerShift, ...
    'MaximumDopplerShift',maxDopplerShift, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',100, ...
    'PathGainsOutputPort',true);
```

Modulation and Channel Filtering

Create a `comm.QPSKModulator` System object to modulate the channel data, which has been generated using the `randi` function. In the code here a 'frame' refers to a vector of information bits. A phase offset of pi/4 is used for this example.

```
qpskMod = comm.QPSKModulator('BitInput',true,'PhaseOffset',pi/4);

% Number of bits transmitted per frame is set to be 1000. For QPSK
% modulation, this corresponds to 500 symbols per frame.
bitsPerFrame = 1000;
msg = randi([0 1],bitsPerFrame,1);

% Modulate data for transmission over channel
modSignal = qpskMod(msg);

% Apply Rayleigh or Rician channel object on the modulated data
rayChan(modSignal);
ricChan(modSignal);
```

Visualization

The fading channel System objects have built-in visualization to show the channel impulse response, frequency response, or Doppler spectrum when the object runs. To invoke it, set the `Visualization` property to the desired value before calling the object. Release the Rayleigh and Rician channel System objects now so that to change their property values.

```
release(rayChan);
release(ricChan);
```

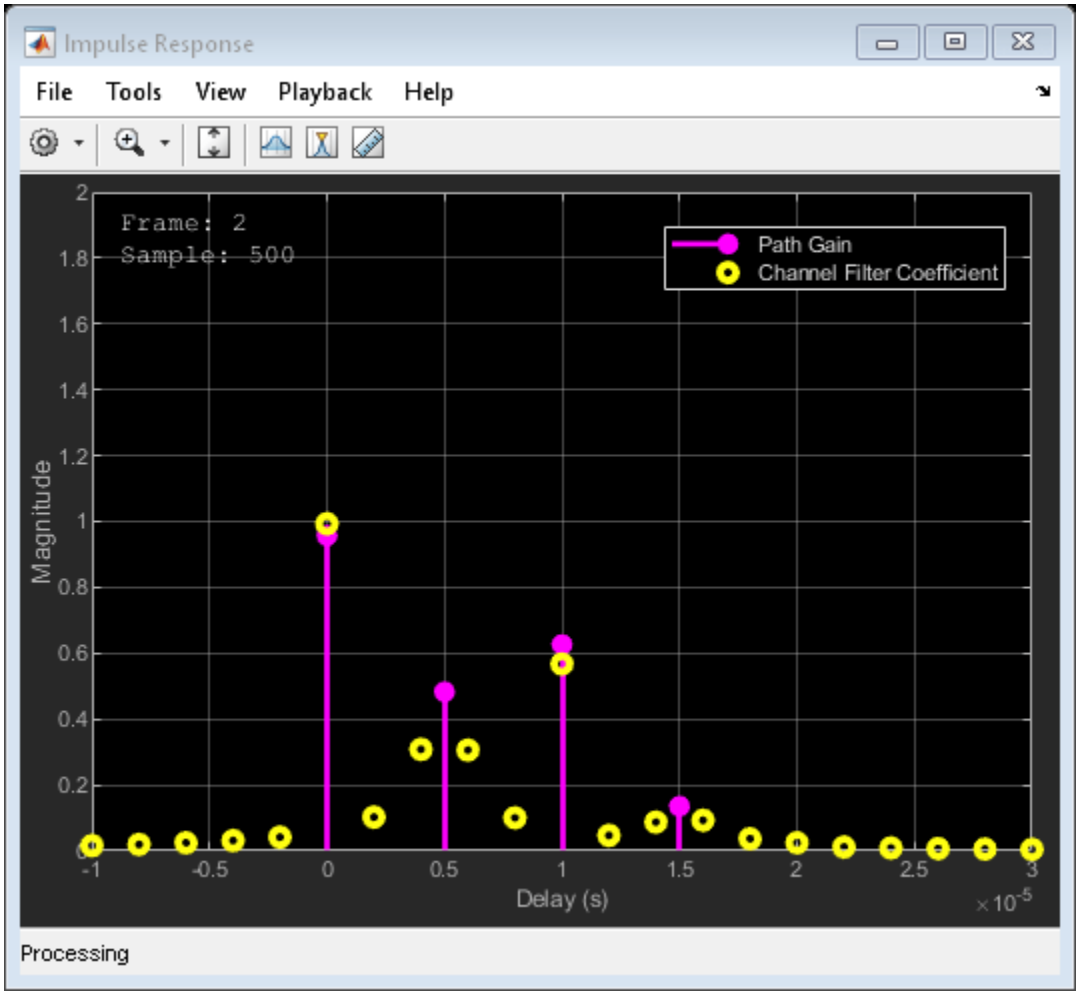
Wideband or Frequency-Selective Fading

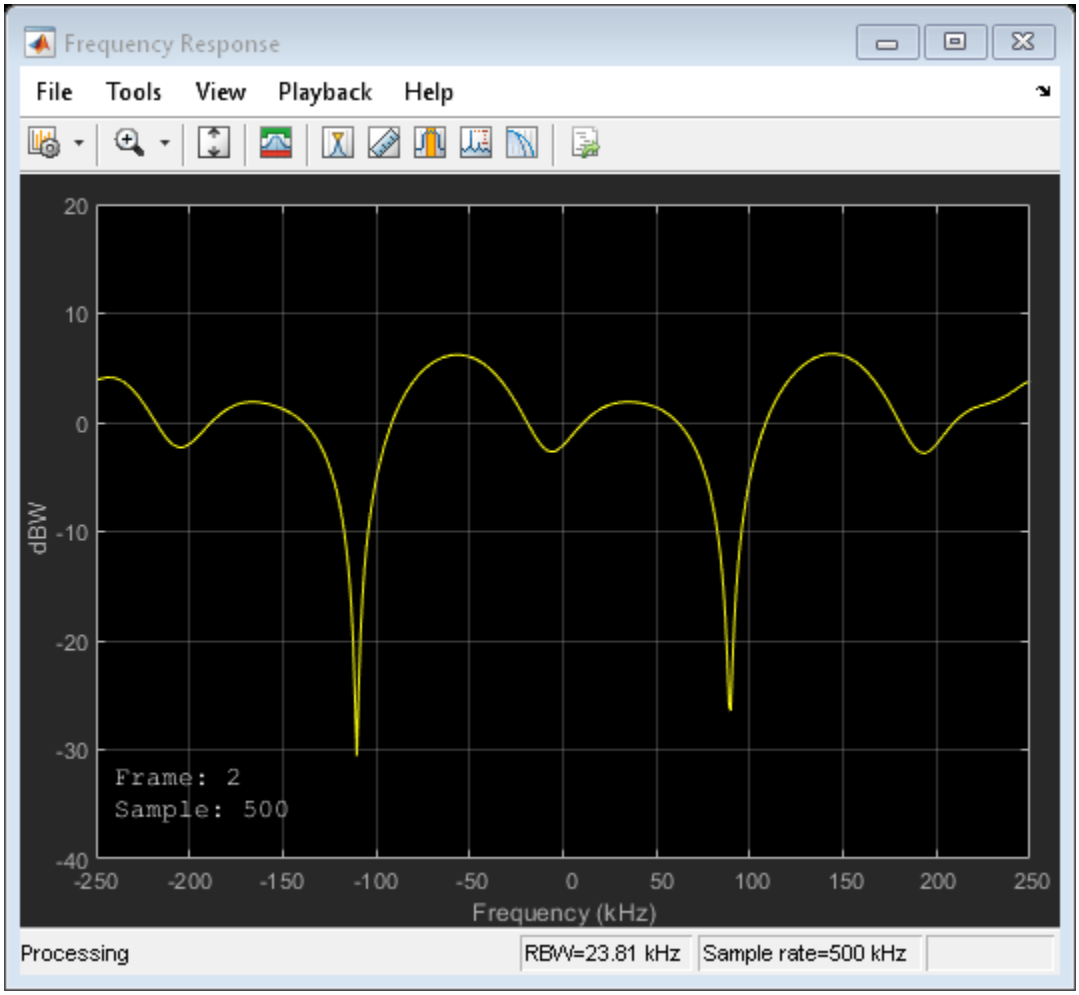
Setting the `Visualization` property to `'Impulse response'` shows the bandlimited impulse response (yellow circles). The visualization also shows the delays and magnitudes of the underlying fading path gains (pink stembars) clustered around the peak of the impulse response. Note that the path gains do not equal the `AveragePathGains` property value because the Doppler effect causes the gains to fluctuate over time.

Similarly, setting the `Visualization` property to `'Frequency response'` shows the frequency response (DFT transformation) of the impulses. You can also set `Visualization` to `'Impulse and frequency responses'` to display both impulse and frequency responses side by side.

You can control the percentage of the input samples to be visualized by changing the `SamplesToDisplay` property. In general, the smaller the percentage, the faster the simulation runs. Once the visualization figure opens up, click the `Playback` button and turn off the "Reduce Updates to Improve Performance" or "Reduce Plot Rate to Improve Performance" option to further improve display accuracy. The option is on by default for faster simulation. To see the channel response for every input sample, uncheck this option and set `SamplesToDisplay` to `'100%'`.

```
rayChan.Visualization = 'Impulse and frequency responses';
rayChan.SamplesToDisplay = '100%';

% Display impulse and frequency responses for 2 frames
numFrames = 2;
for i = 1:numFrames
    % Create random data
    msg = randi([0 1],bitsPerFrame,1);
    % Modulate data
    modSignal = qpskMod(msg);
    % Filter data through channel and show channel responses
    rayChan(modSignal);
end
```

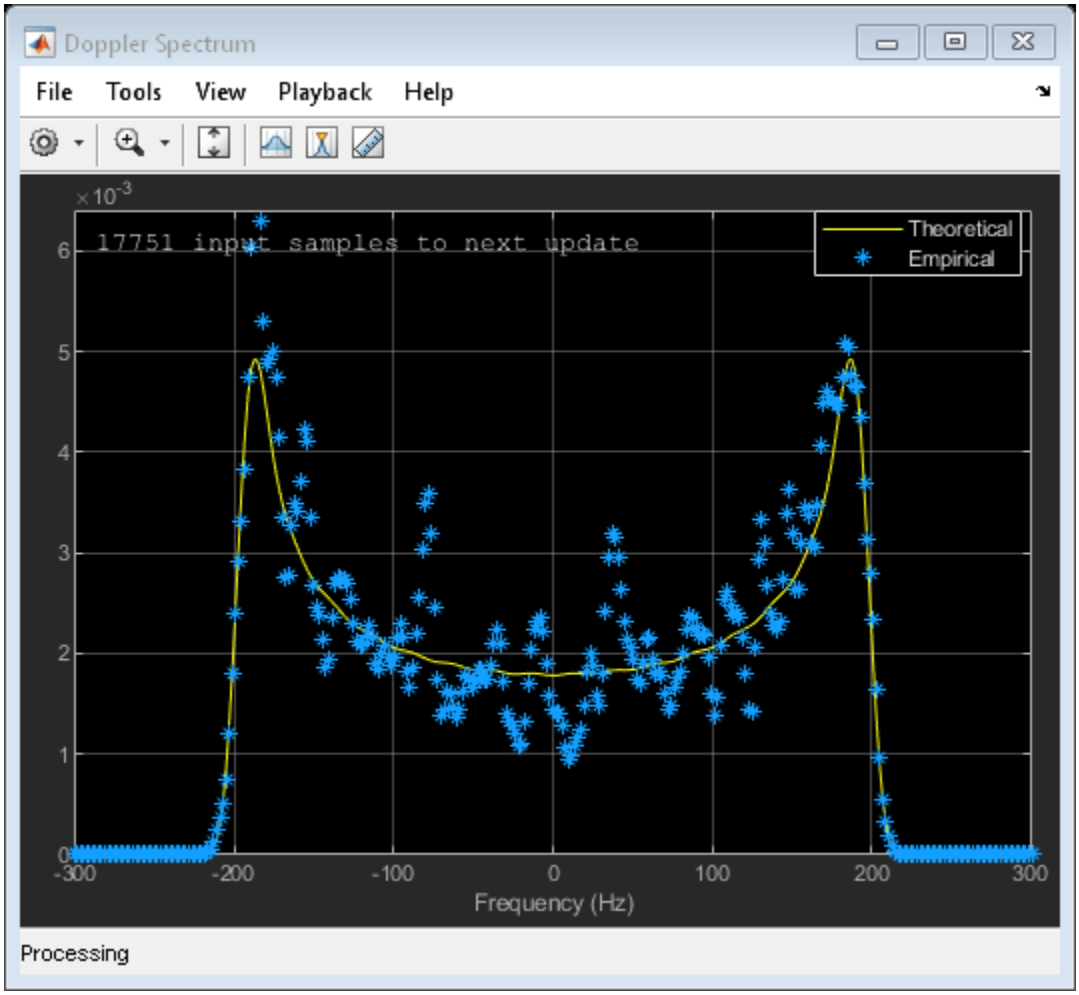

Impulse Response
File
Tools
View
Playback
Help
Frame: 2
Sample: 500
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-5
Processing



As you can see, the channel frequency response is not flat and may have deep fades over the 500K Hz bandwidth. Because the power level varies over the bandwidth of the signal, it is referred to as frequency-selective fading.

For the same channel specification, we now display the Doppler spectrum for its first discrete path, which is a statistical characterization of the fading process. The System object makes periodic measurements of the Doppler spectrum (blue stars). Over time with more samples processed by the System object, the average of this measurement better approximates the theoretical Doppler spectrum (yellow curve).

```
release(rayChan);
rayChan.Visualization = 'Doppler spectrum';

% Display Doppler spectrum from 5000 frame transmission
numFrames = 5000;
for i = 1:numFrames
    msg = randi([0 1],bitsPerFrame,1);
    modSignal = qpskMod(msg);
    rayChan(modSignal);
end
```



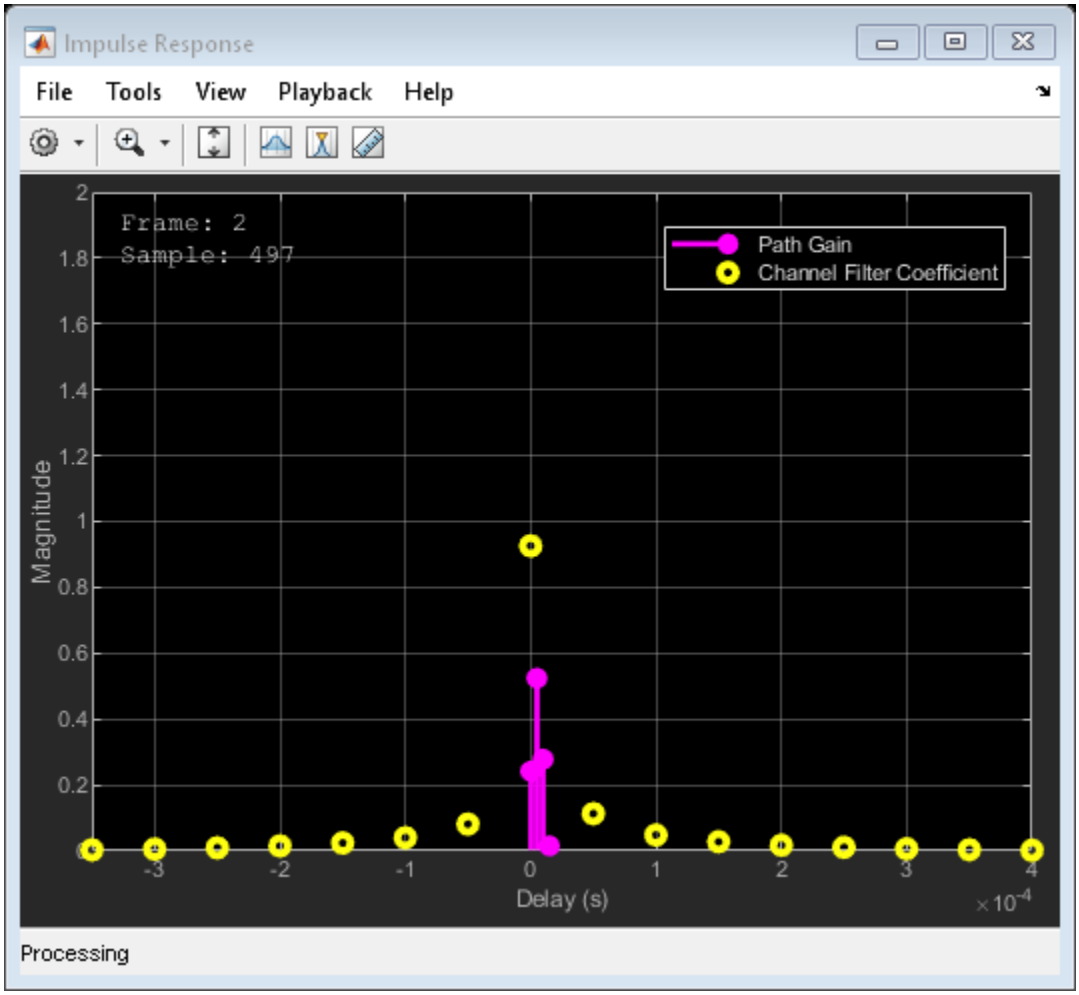
Narrowband or Frequency-Flat Fading

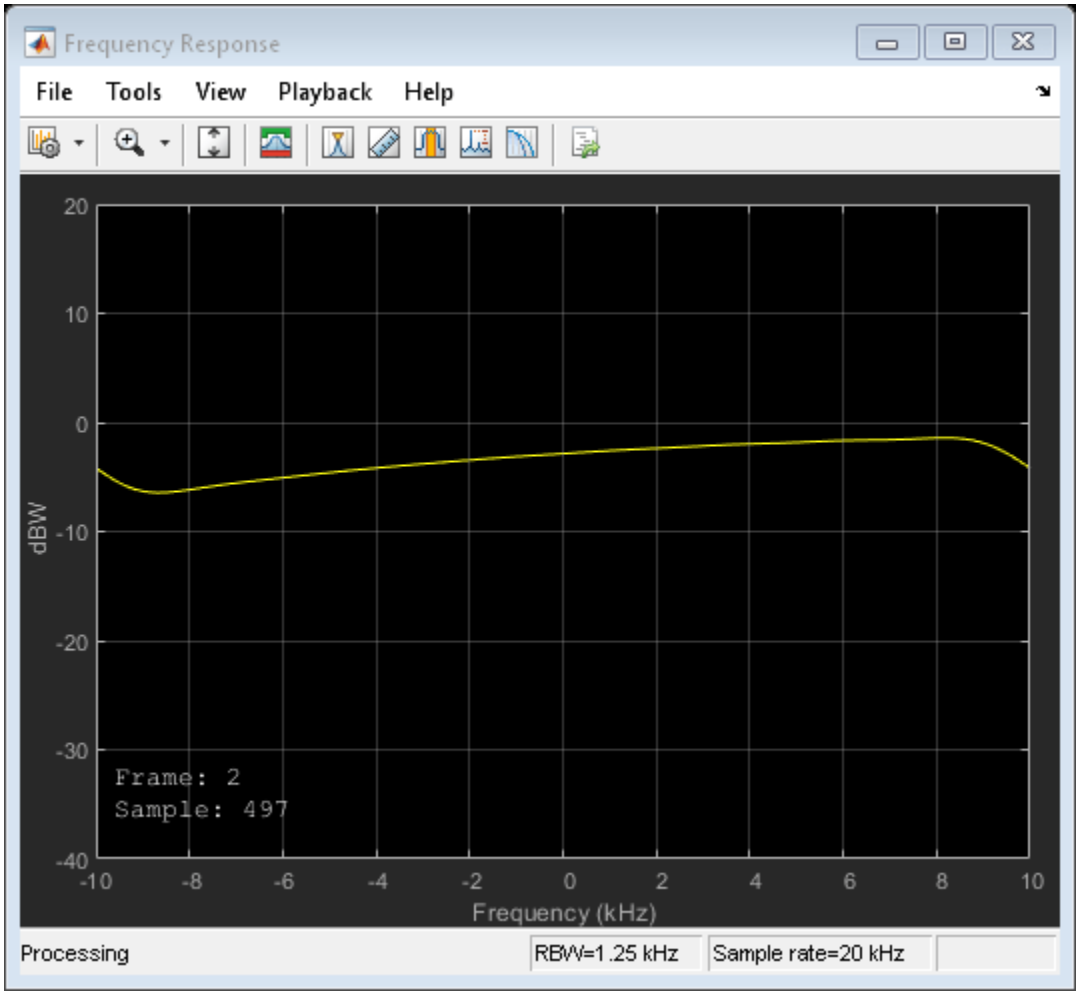
When the bandwidth is too small for the signal to resolve the individual components, the frequency response is approximately flat because of the minimal time dispersion caused by the multipath channel. This kind of multipath fading is often referred to as narrowband fading, or frequency-flat fading.

We now reduce the signal bandwidth from 500 kb/s (250 ksym/s) to 20 kb/s (10 ksym/s), so the delay span (15 microseconds) of the channel is much smaller than the QPSK symbol period (100 microseconds). The resultant impulse response has very small intersymbol interference (ISI) and the frequency response is approximately flat.

```
release(rayChan);
rayChan.Visualization = 'Impulse and frequency responses';
rayChan.SampleRate = sampleRate20kHz;
rayChan.SamplesToDisplay = '25%';  % Display one of every four samples

% Display impulse and frequency responses for 2 frames
numFrames = 2;
for i = 1:numFrames
    msg = randi([0 1],bitsPerFrame,1);
    modSignal = qpskMod(msg);
```

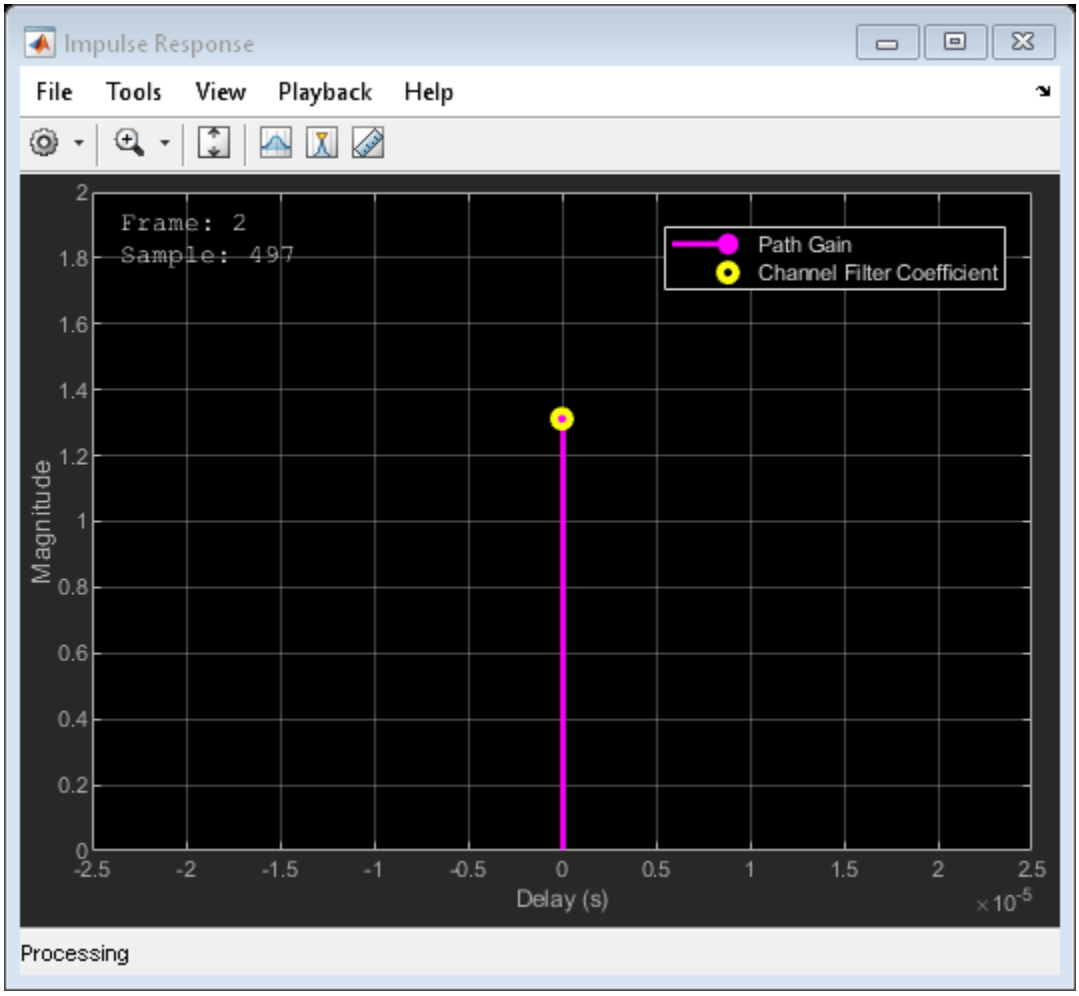
```
    rayChan(modSignal);
end
```

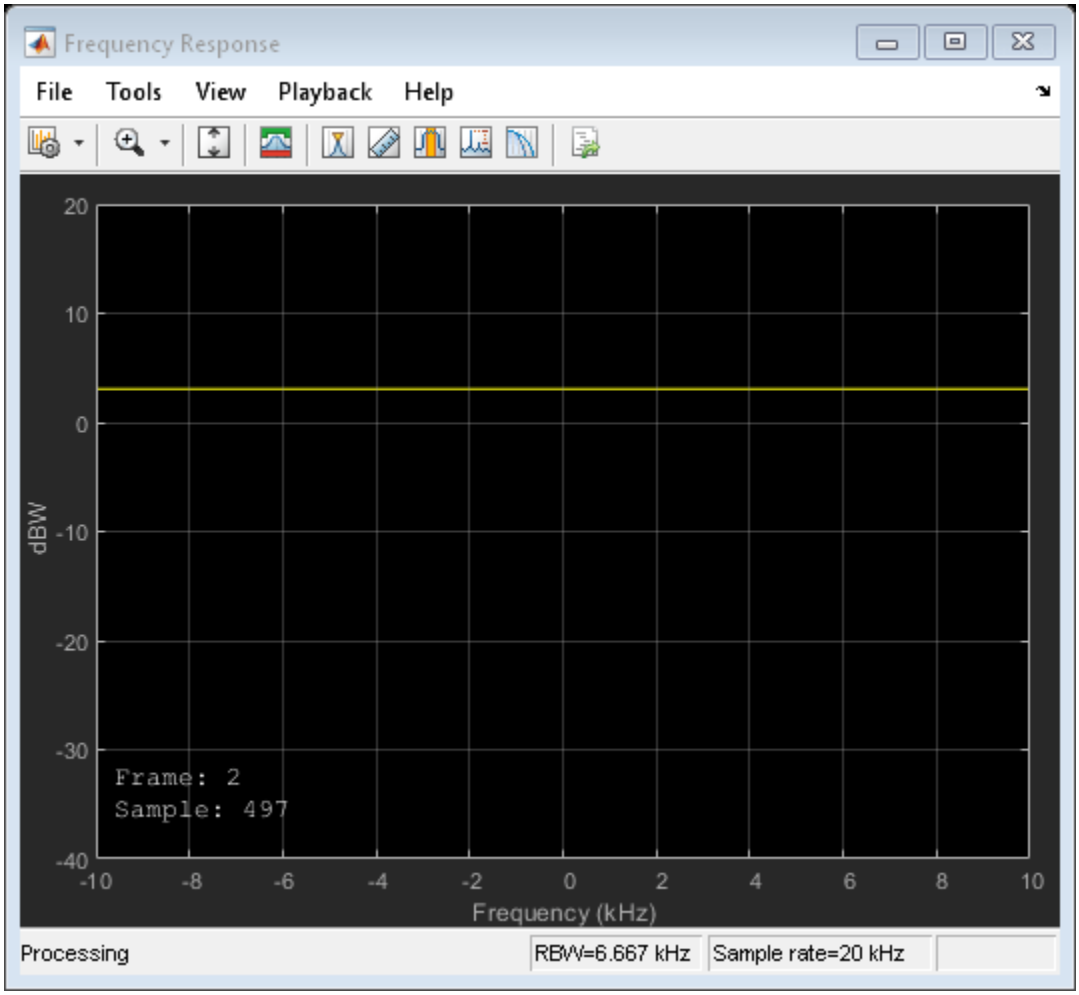




To simplify and speed up modeling, narrowband fading channels are typically modeled as a single-path fading channel. That is, a multipath fading model overspecifies a narrowband fading channel. The following settings correspond to a narrowband fading channel. Notice that the shape of the bandlimited impulse response is flat.

```
release(rayChan);
rayChan.PathDelays = 0;        % Single fading path with zero delay
rayChan.AveragePathGains = 0;  % Average path gain of 1 (0 dB)

for i = 1:numFrames % Display impulse and frequency responses for 2 frames
    msg = randi([0 1],bitsPerFrame,1);
    modSignal = qpskMod(msg);
    rayChan(modSignal);
end
```

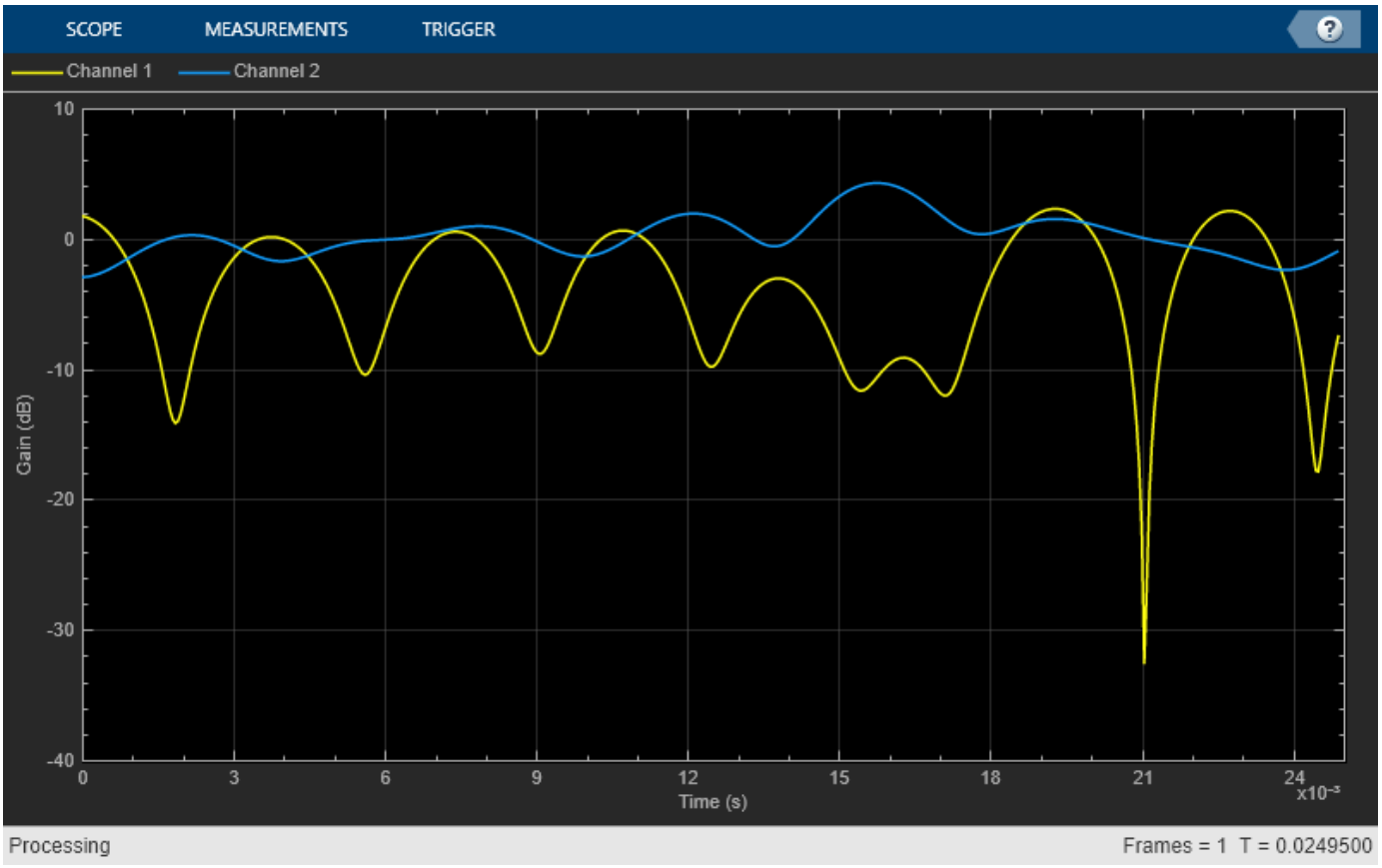

Impulse Response
File
Tools
View
Playback
Help
Frame: 2
Sample: 497
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-5
Processing



The Rician fading channel System object models line-of-sight propagation in addition to diffuse multipath scattering. This results in a smaller variation in the magnitude of path gains. To compare the variation between Rayleigh and Rician channels, we make use of a `timescope` object to view their path gains over time. Note that the magnitude fluctuates over approximately a 10 dB range for the Rician fading channel (blue curve), compared with 30-40 dB for the Rayleigh fading channel (yellow curve). For the Rician fading channel, this variation would be further reduced by increasing the K-factor (currently set to 10).

```
release(rayChan);
rayChan.Visualization = 'Off'; % Turn off Rayliegh object visualization
ricChan.Visualization = 'Off'; % Turn off Rician object visualization

% Same sample rate and delay profile for the Rayleigh and Rician objects
ricChan.SampleRate = rayChan.SampleRate;
ricChan.PathDelays = rayChan.PathDelays;
ricChan.AveragePathGains = rayChan.AveragePathGains;

% Configure a Time Scope System object to show path gain magnitude
gainScope = timescope( ...
    'SampleRate',rayChan.SampleRate, ...
    'TimeSpanSource','Property',...
    'TimeSpan',bitsPerFrame/2/rayChan.SampleRate, ... % One frame span
```

```
    'Name','Multipath Gain', ...
    'ShowGrid',true, ...
    'YLimits',[-40 10], ...
    'YLabel','Gain (dB)');

% Compare the path gain outputs from both objects for one frame
msg = randi([0 1],bitsPerFrame,1);
modSignal = qpskMod(msg);
[~,rayPathGain] = rayChan(modSignal);
[~,ricPathGain] = ricChan(modSignal);
% Form the path gains as a two-channel input to the time scope
gainScope(10*log10(abs([rayPathGain,ricPathGain]).^2));
```

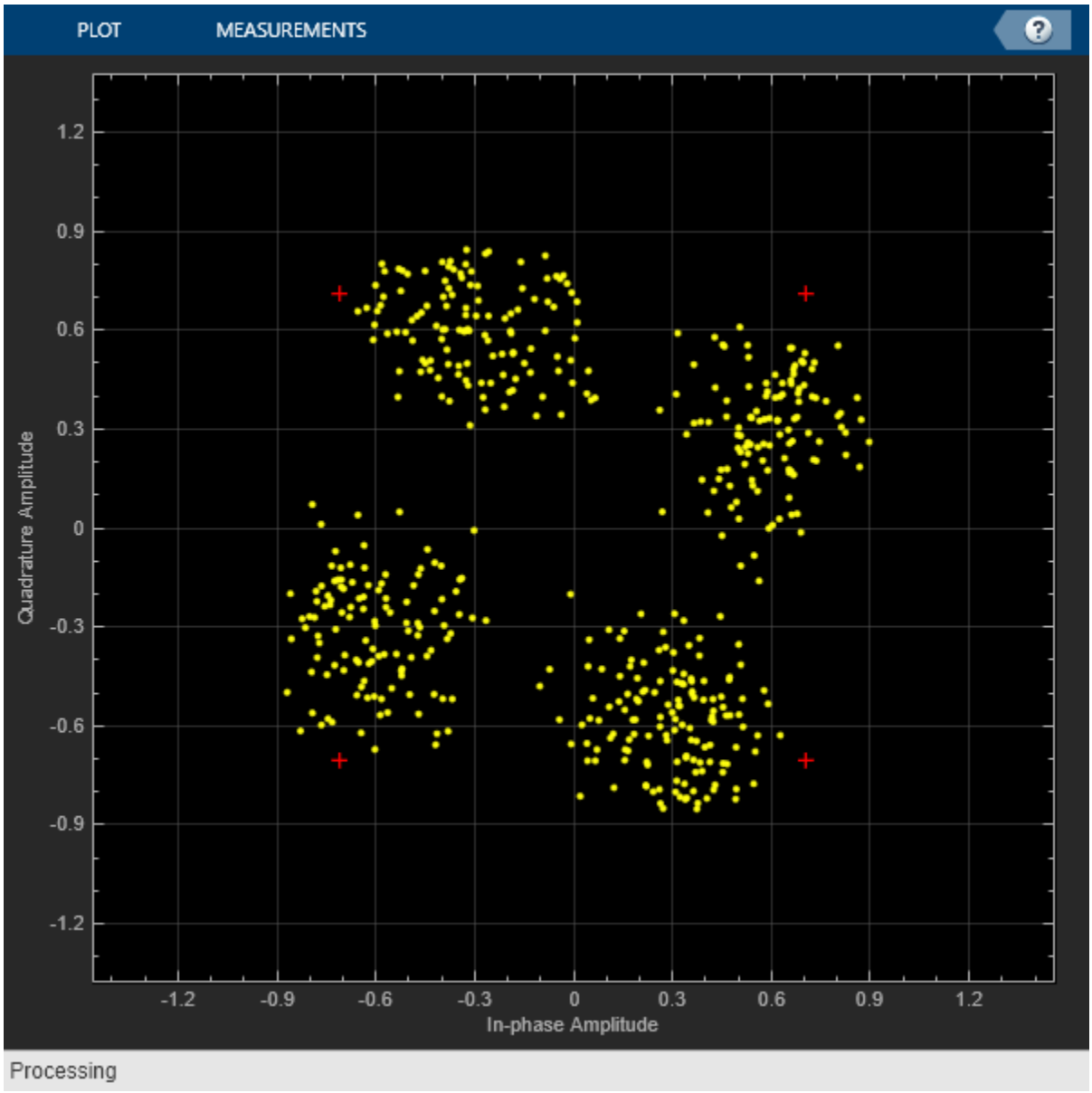


Fading Channel Impact on Signal Constellation

We now return to our original four-path Rayleigh fading channel. We use a `comm.ConstellationDiagram` System object to show the impact of narrowband fading on the signal constellation. To slow down the channel dynamics for visualization purposes, we reduce the maximum Doppler shift to 5 Hz. Compared with the QPSK channel input signal, you can observe signal attenuation and rotation at the channel output, as well as some signal distortion due to the small amount of ISI in the received signal.

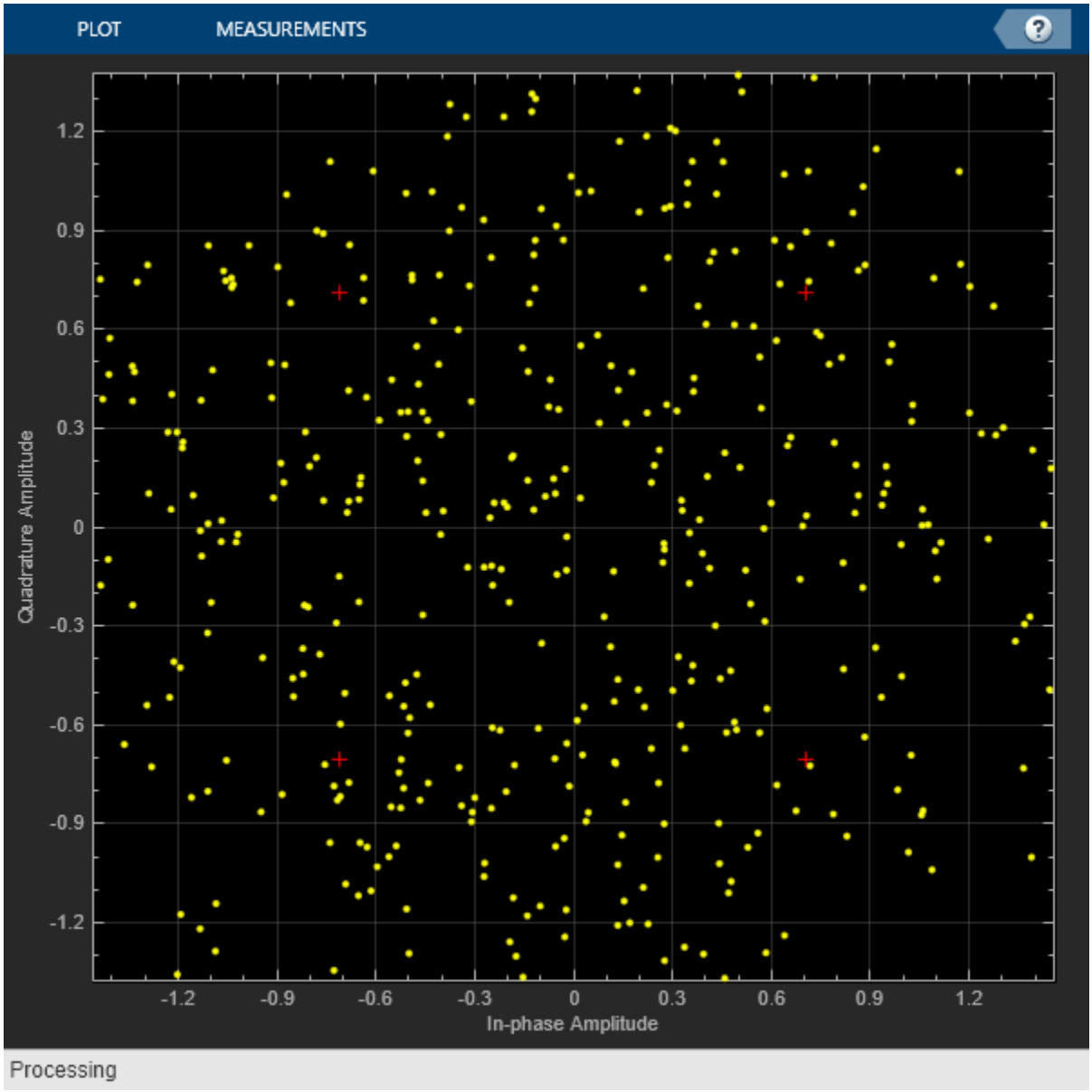
```
clear hRicChan hMultipathGain;
release(rayChan);

rayChan.PathDelays = delayVector;
rayChan.AveragePathGains = gainVector;
```

```
rayChan.MaximumDopplerShift = 5;

% Configure a Constellation Diagram System object to show received signal
constDiag = comm.ConstellationDiagram( ...
    'Name','Received Signal After Rayleigh Fading');

numFrames = 16;

for n = 1:numFrames
    msg = randi([0 1],bitsPerFrame,1);
    modSignal = qpskMod(msg);
    rayChanOut = rayChan(modSignal);
    % Display constellation diagram for Rayleigh channel output
    constDiag(rayChanOut);
end
```



When we increase the signal bandwidth to 500 kb/s (250 ksym/s), we see much greater distortion in the signal constellation. This distortion is the ISI that comes from time dispersion of the wideband signal. The delay span (15 microseconds) of the channel is now larger than the QPSK symbol period (4 microseconds), so the resultant bandlimited impulse response is no longer approximately flat.

```
release(rayChan);
release(constDiag);
rayChan.SampleRate = sampleRate500kHz;

for n = 1:numFrames
    msg = randi([0 1],bitsPerFrame,1);
    modSignal = qpskMod(msg);
    rayChanOut = rayChan(modSignal);
    constDiag(rayChanOut);
end
```


PLOT
MEASUREMENTS
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Quadrature Amplitude
-1.2
-0.9
-0.6
-0.3
0
0.3
0.6
0.9
1.2
In-phase Amplitude
Processing

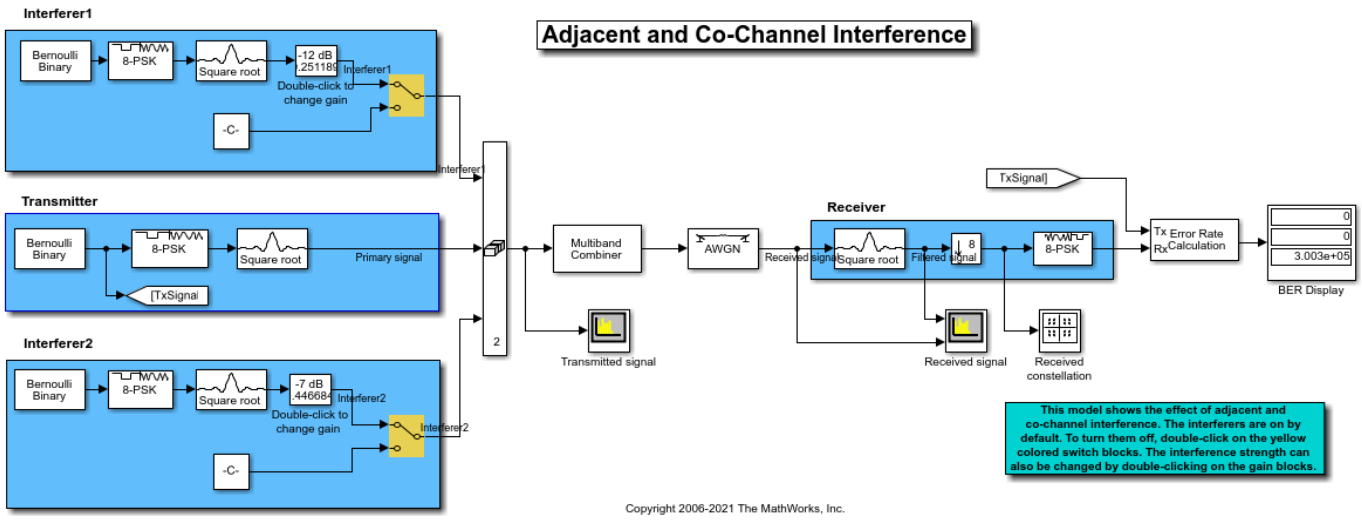
Adjacent and Co-Channel Interference

This model uses PSK-modulated signals to show the effects of adjacent and co-channel interference on a transmitted signal. You can view the effect of adjacent channel interferer and co-channel interferer together or individually.

Exploring the Example

The communication system in this example includes these components:

- Transmitter - Creates a PSK-modulated signal and applies a square root raised cosine filter. The interference is added to this **Primary signal**.
- Interferer1 - Creates a PSK-modulated interference signal.
- Interferer2 - Creates a PSK-modulated interference signal.
- Multiband Combiner - Combines all the signals without introducing signal distortion and enables interference modeling. The Multiband Combiner block interpolates input signals, frequency shifts the signals by the value specified in the "Frequency offsets" parameter, and then combines the signals into one output signal.
- AWGN Channel - Adds noise to the transmitted signals.
- Receiver - Filters, downsamples, and demodulates the received signal.
- Error Rate Calculation - Computes the bit error rate.

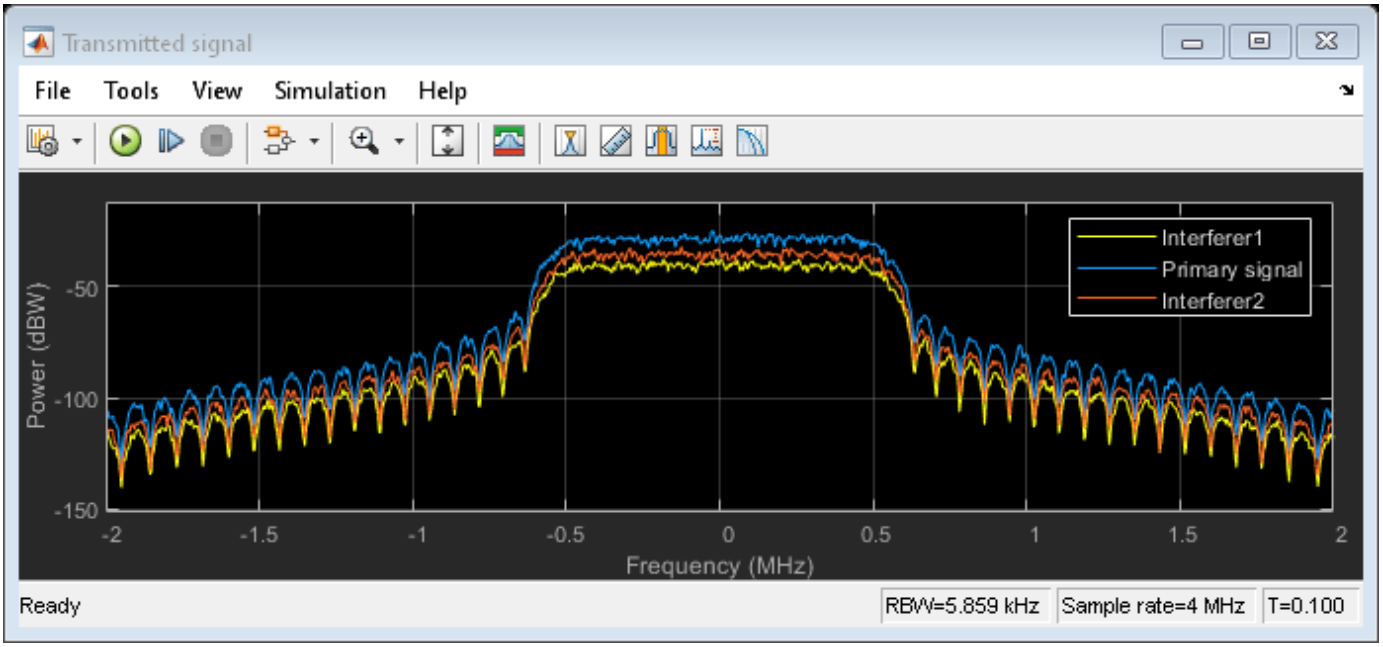


Results and Displays

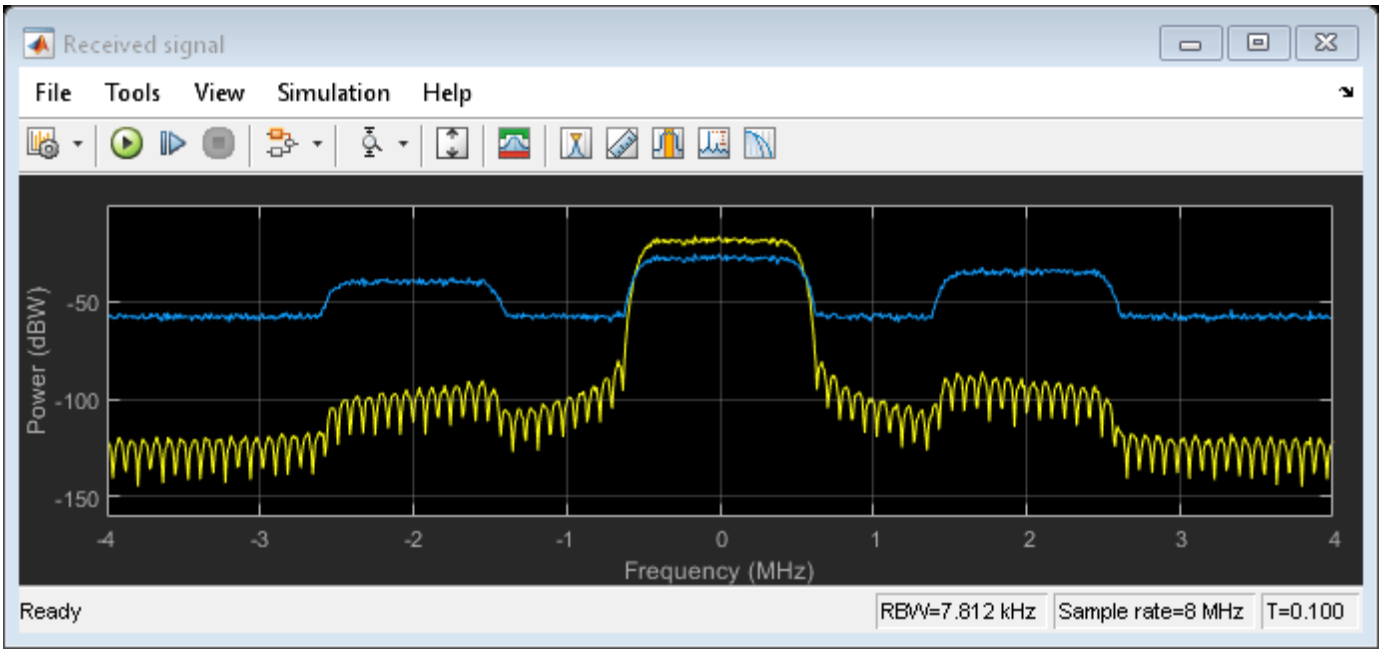
When you run the simulation, the block labeled **BER Display** shows the bit error rate for the original signal. The BER Display block shows a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

Scope blocks in the model display the spectra of the primary and interfering signals, spectrum of the received combined signal, and constellation diagram of the received signal.

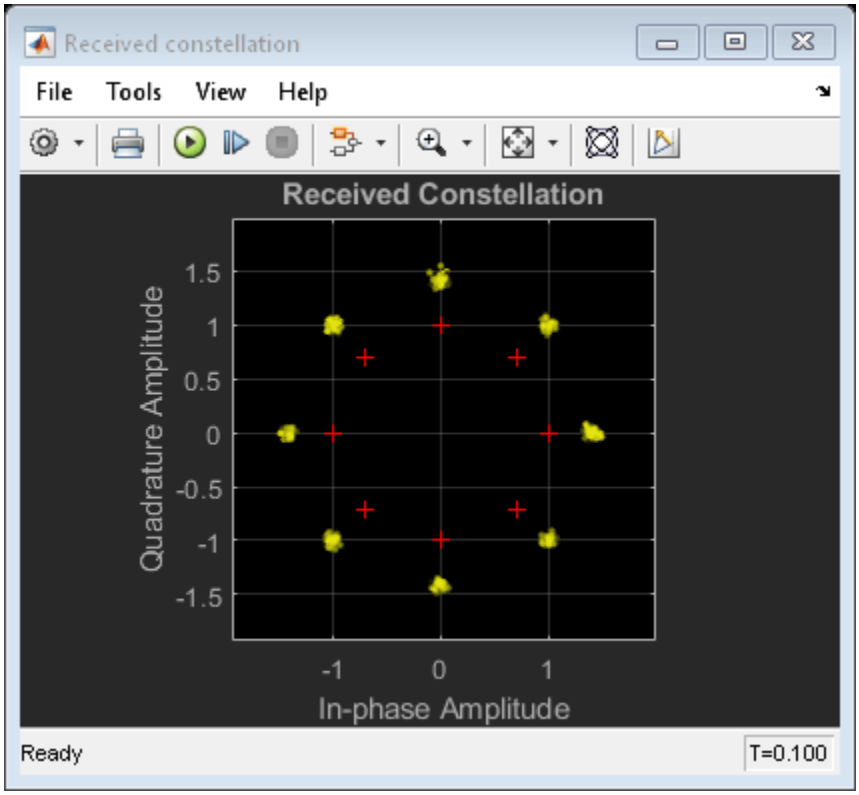
- The spectra of the primary and interfering signals



- The spectrum of the received signal



- A scatter plot of the received signal constellation



Experimenting with the Example

To deactivate an interferer, double-click the switch block that corresponds to that interferer. In the "Received signal" spectrum analyzer, notice the effect of omitting the interfering signal.

To change the spectral overlap between primary signal and interfering signals, set the "Frequency offsets" parameter of Multiband Combiner block. As you decrease the offset, the "Received signal" spectrum analyzer shows the interfering signal slowly moving from the adjacent channel into the frequency band of the original signal and eventually causing co-channel interference. The values reported in the `BER Display` block slowly deteriorate as the offset decreases, because the 8-PSK constellation points become difficult to demodulate correctly.

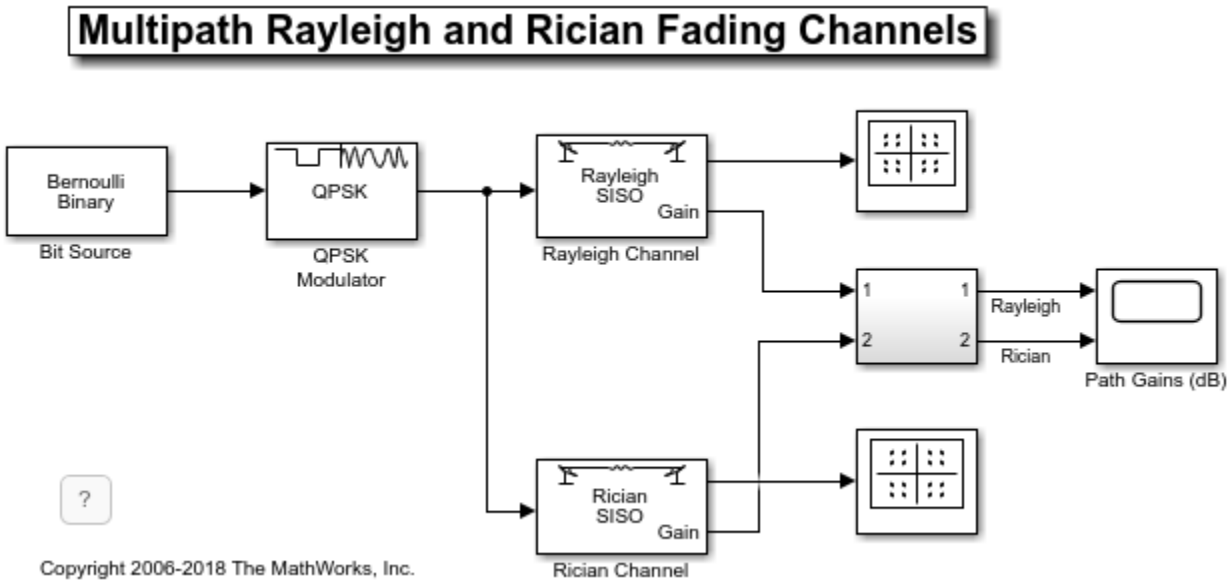
To change the power gain of an interfering signal, double-click the dB Gain block and change the Gain parameter. Observe the effect on the "Transmitted signal" and the "Received signal" spectrum analyzers. If you decrease the negative dB gain, the BER worsens, especially in the presence of co-channel interference.

Multipath Fading Channel in Simulink

This model shows how to use the SISO Fading Channel block from the Communications Toolbox™ to simulate multipath Rayleigh and Rician fading channels, which are useful models of real-world phenomena in wireless communications. These phenomena include multipath scattering effects, time dispersion, and Doppler shifts that arise from relative motion between the transmitter and receiver. The model also shows how to visualize channel characteristics such as the impulse and frequency responses, Doppler spectrum and component gains.

Model and Parameters

The example model simulates QPSK transmission over a multipath Rayleigh fading channel and a multipath Rician fading channel. Both the channel blocks are configured from the SISO Fading Channel library block. You can control transmission and channel parameters via workspace variables.



The following variables control the "Bit Source" block. By default, the bit rate is 10M b/s (5M sym/s) and each transmitted frame is 2000 bits long (1000 symbols).

```
bitRate =

    10000000


bitsPerFrame =

        2000
```

The following variables control both the Rayleigh and Rician fading channel blocks. By default, the channels are modeled as four fading paths, each representing a cluster of multipath components received at around the same delay.

```
delayVector =

   1.0e-06 *

         0    0.2000    0.4000    0.8000


gainVector =

     0    -3    -6    -9
```

By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 m difference in path length. In some outdoor multipath environments, reflected paths can be up to several kilometers longer than the shortest path. With the path delays specified above, the last path is 240 m longer than the shortest path, and thus arrives 0.8 microseconds later.

Together, the path delays and average path gains specify the delay profile of the channel. Typically, the average path gains decay exponentially with delay (i.e., the dB values decay linearly), but the specific delay profile depends on the propagation environment. On each channel block, we have also turned on the option to normalize the average path gains so that their average gain is 0 dB over time.

The following variable controls the maximum Doppler shift which is computed as v*f/c, where v is the mobile speed, f is the carrier frequency, and c is the speed of light. The default maximum Doppler shift in the model is 200 Hz which corresponds to a mobile speed of 65 mph (30 m/s) and a carrier frequency of 2 GHz.

```
maxDopplerShift =

   200
```

The following variables apply to the Rician fading channel block. The Doppler shift of the line-of-sight component is typically smaller than the maximum Doppler shift, `maxDopplerShift`, and depends on the direction of travel of the mobile relative to the direction of the line-of-sight path. The K-factor specifies the ratio of average received power from the line-of-sight path relative to that of the associated diffuse components.

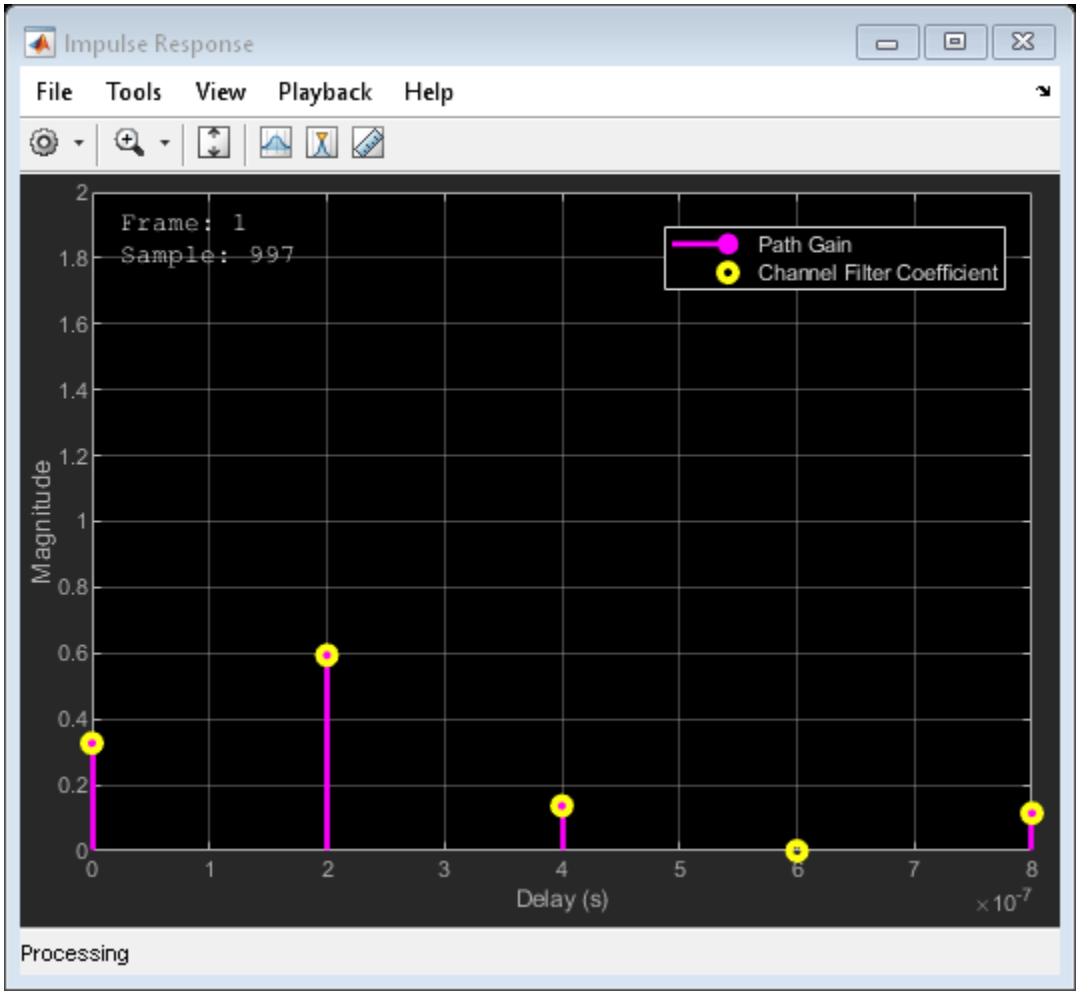
```
LOSDopplerShift =

   100


KFactor =

    10
```

The SISO Fading Channel block can visualize channel impulse response, frequency response, and Doppler spectrum while the model is running. To invoke it, set the `Channel visualization` parameter to the desired channel characteristic(s) before running the model. Note that turning on channel visualization may slow down your simulation.

Wideband or Frequency-Selective Fading

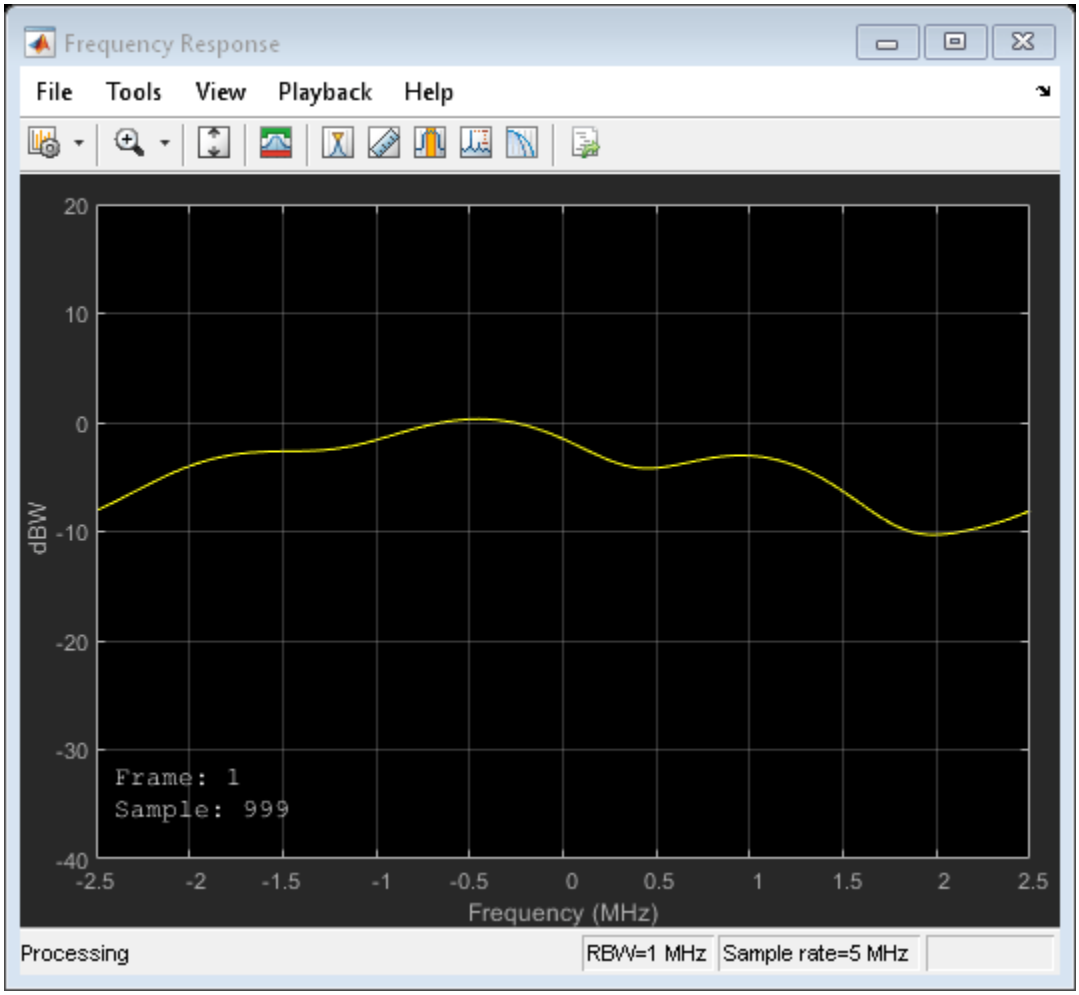
By default, the delay span (0.8 microseconds) of the channel is larger than the input QPSK symbol period (0.2 microseconds), and causes considerable intersymbol interference (ISI). So the resultant channel frequency response is not flat and may have deep fades over the 10M Hz signal bandwidth. Because the power level varies over the bandwidth, it is referred to as frequency-selective fading.

Setting the `Channel visualization` parameter of the channel block to 'Impulse response' shows the bandlimited impulse response (yellow circles). The visualization also shows the delays and magnitudes of the underlying fading path gains (pink stems) clustered around the peak of the impulse response. Note that the path gains do not equal the `Average path gains (dB)` parameter value because the Doppler effect causes the gains to fluctuate over time.



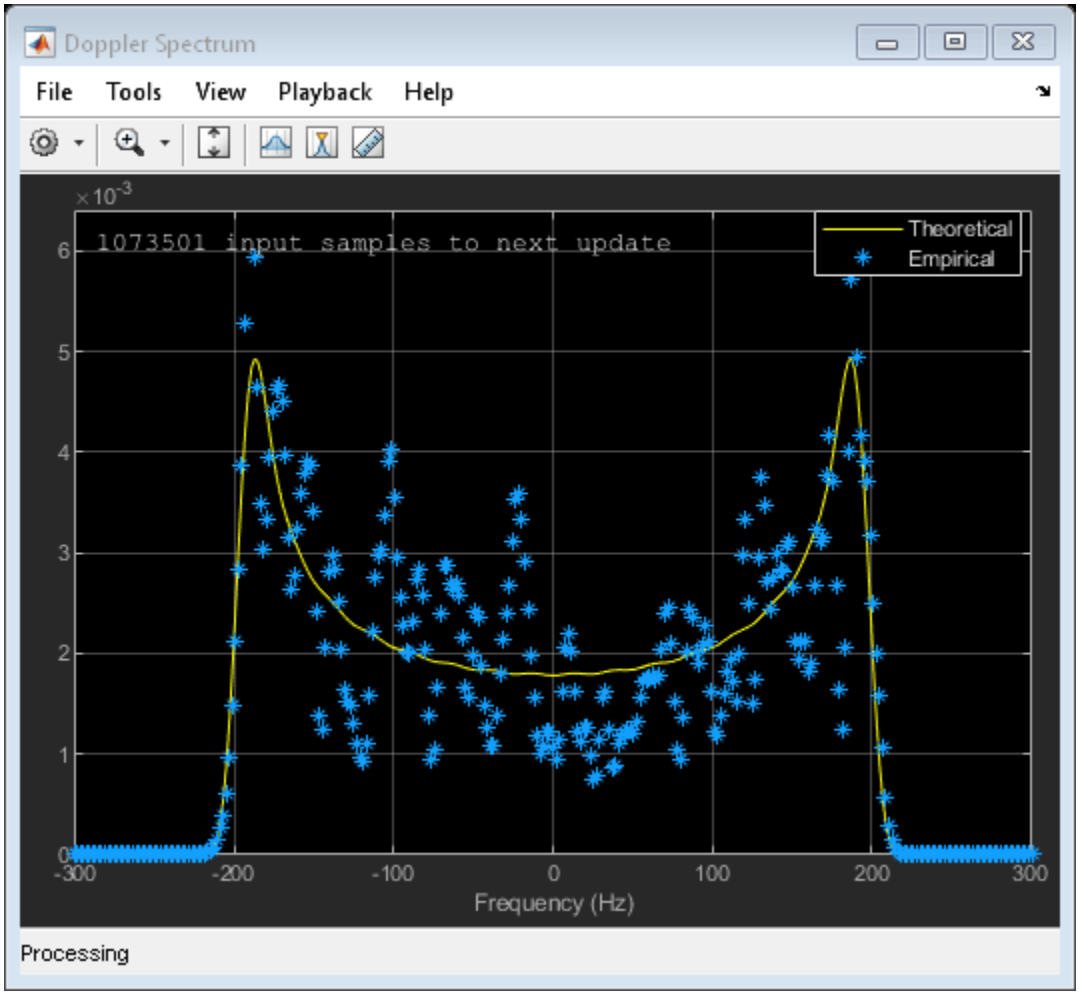
As displayed, the channel impulse response coincides with the path gains for this delay profile because the discrete path delays are all integer multiples of the input symbol period. In this case, there is also no channel filter delay.

Similarly, setting the `Channel visualization` parameter to 'Frequency response' shows the frequency response of the channel. You can also set `Channel visualization` to 'Impulse and frequency responses' to display both impulse and frequency responses side by side. You can see that the power level of the channel varies across the whole bandwidth.

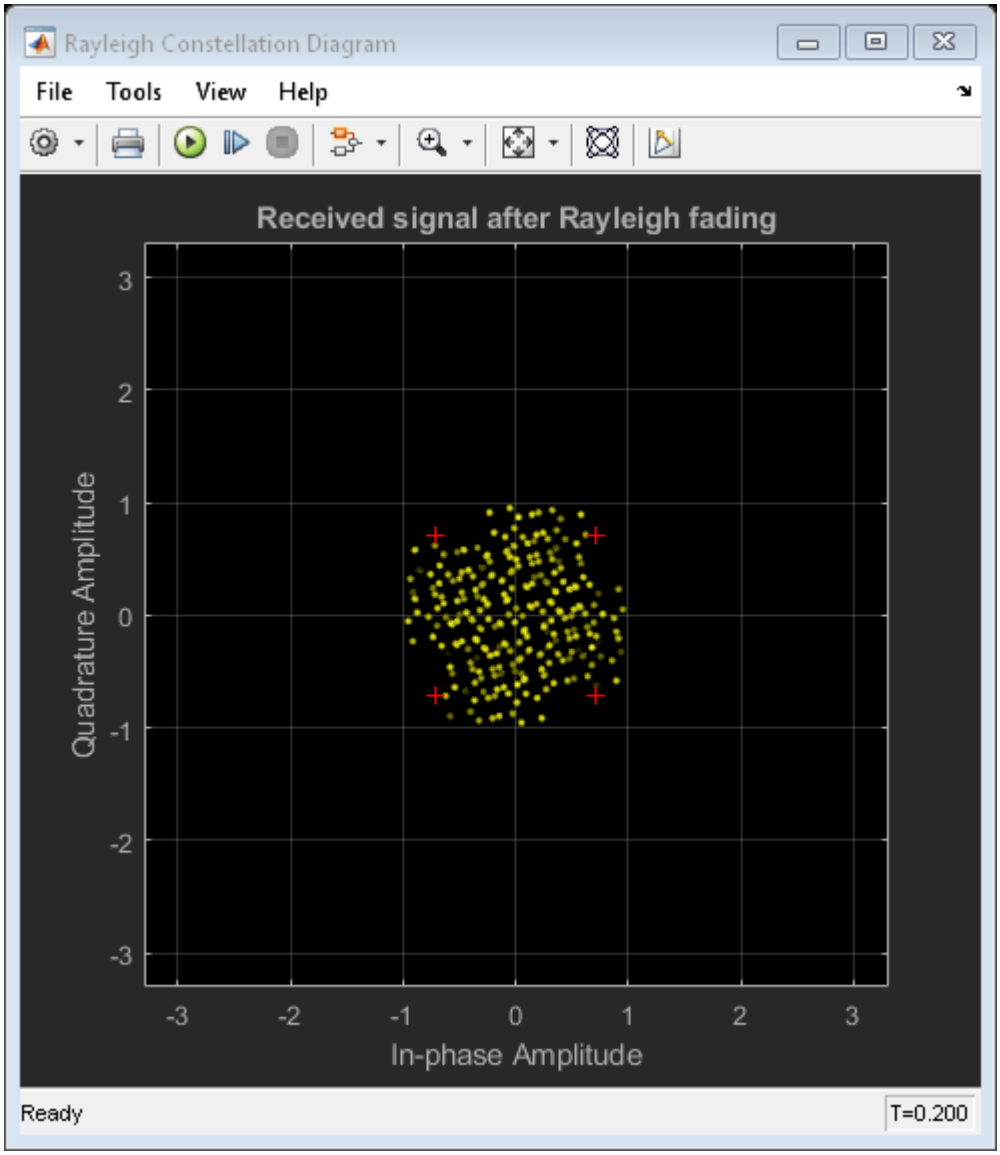


As shown in the channel visualization plots, you can also control the percentage of the input samples to be visualized by changing the `Percentage of samples to display` parameter of the channel block. In general, the smaller the percentage, the faster the model runs. Once the visualization figure opens, click the `Playback` button and turn off the `Reduce Updates to Improve Performance` or `Reduce Plot Rate to Improve Performance` option to further improve display accuracy. The option is on by default for faster simulation. To see the channel response for every input sample, uncheck this option and set `Percentage of samples to display` to '100%'.

For the same channel specification, we now display the Doppler spectrum for its first discrete path, which is a statistical characterization of the fading process. The channel block makes periodic measurements of the Doppler spectrum (blue stars). Over time with more samples processed by the block, the average of this measurement better approximates the theoretical Doppler spectrum (yellow curve).



By opening the constellation diagram following the Rayleigh channel block, you can see the impact of wideband fading on the signal constellation. To slow down the channel dynamics for visualization purposes, we reduce the maximum Doppler shift to 5 Hz. Compared with the QPSK channel input signal, you can observe obvious distortion in the channel output signal, due to the ISI from the time dispersion of the wideband signal.



Narrowband or Frequency-Flat Fading

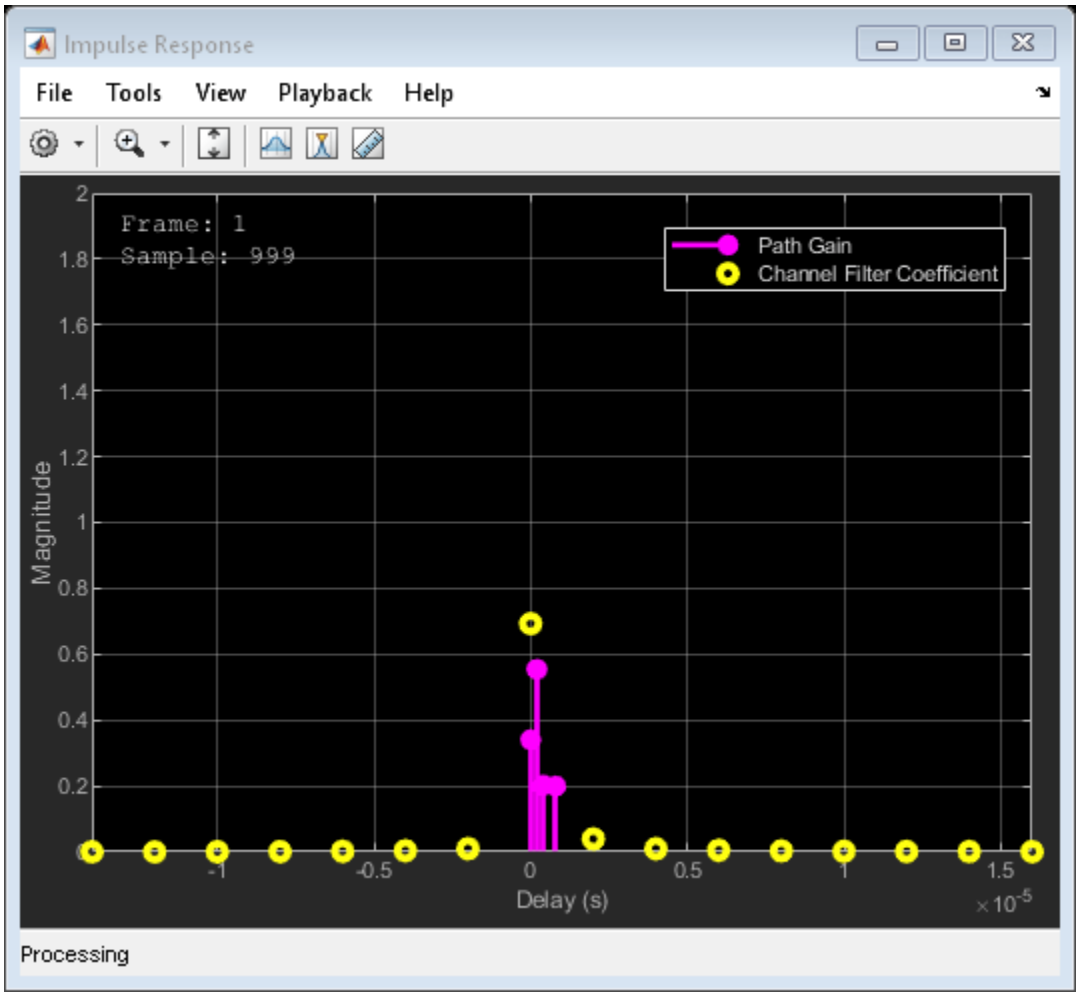
When the bandwidth is too small for the signal to resolve the individual components, the frequency response is approximately flat because of the minimal time dispersion and very small ISI from the impulse response. This kind of multipath fading is often referred to as narrowband fading, or frequency-flat fading.

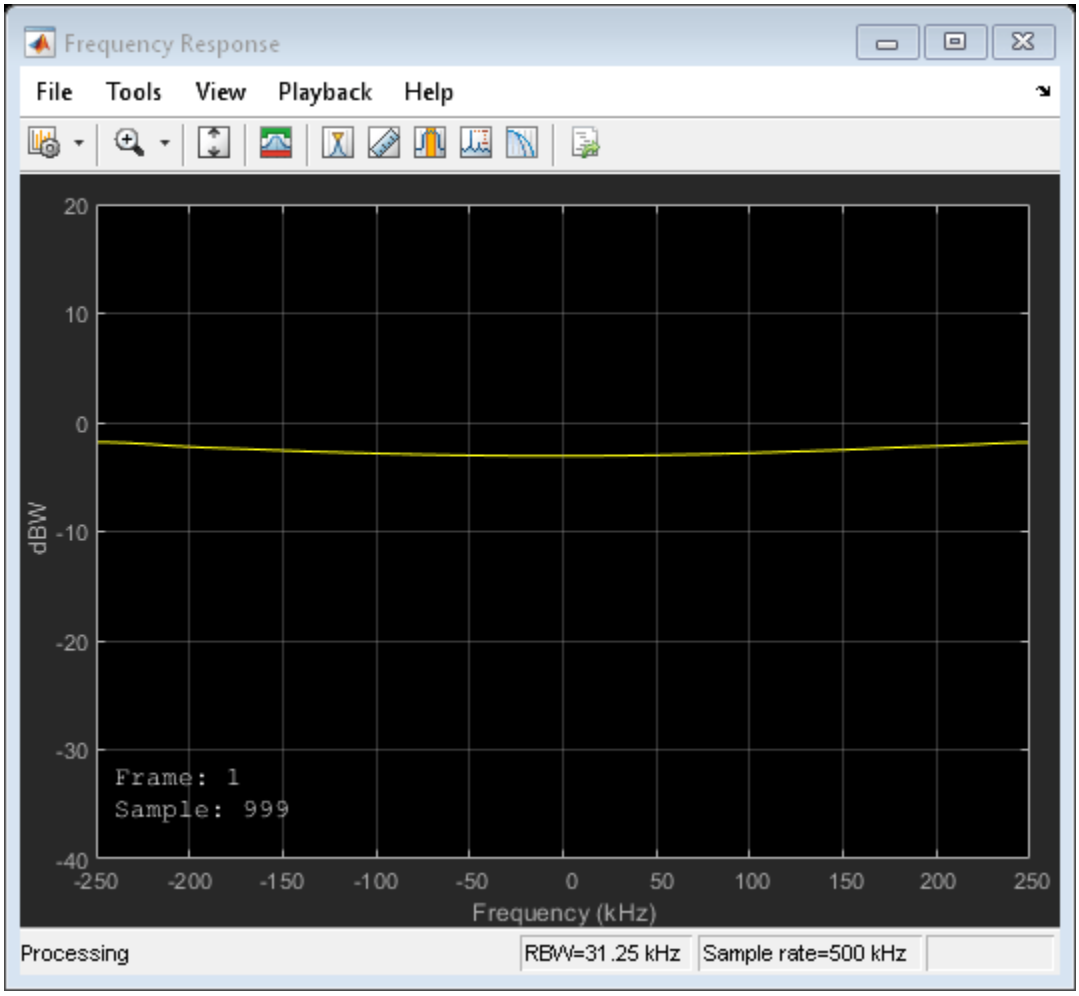
To observe the effect, we now reduce the signal bandwidth from 10M b/s (5M sym/s) to 1M b/s (500K sym/s), so the delay span (0.8 microseconds) of the channel is much smaller than the QPSK symbol period (2 microseconds). Effectively, all delayed components combine at a single delay (in this case, at zero).

```
bitRate =
```

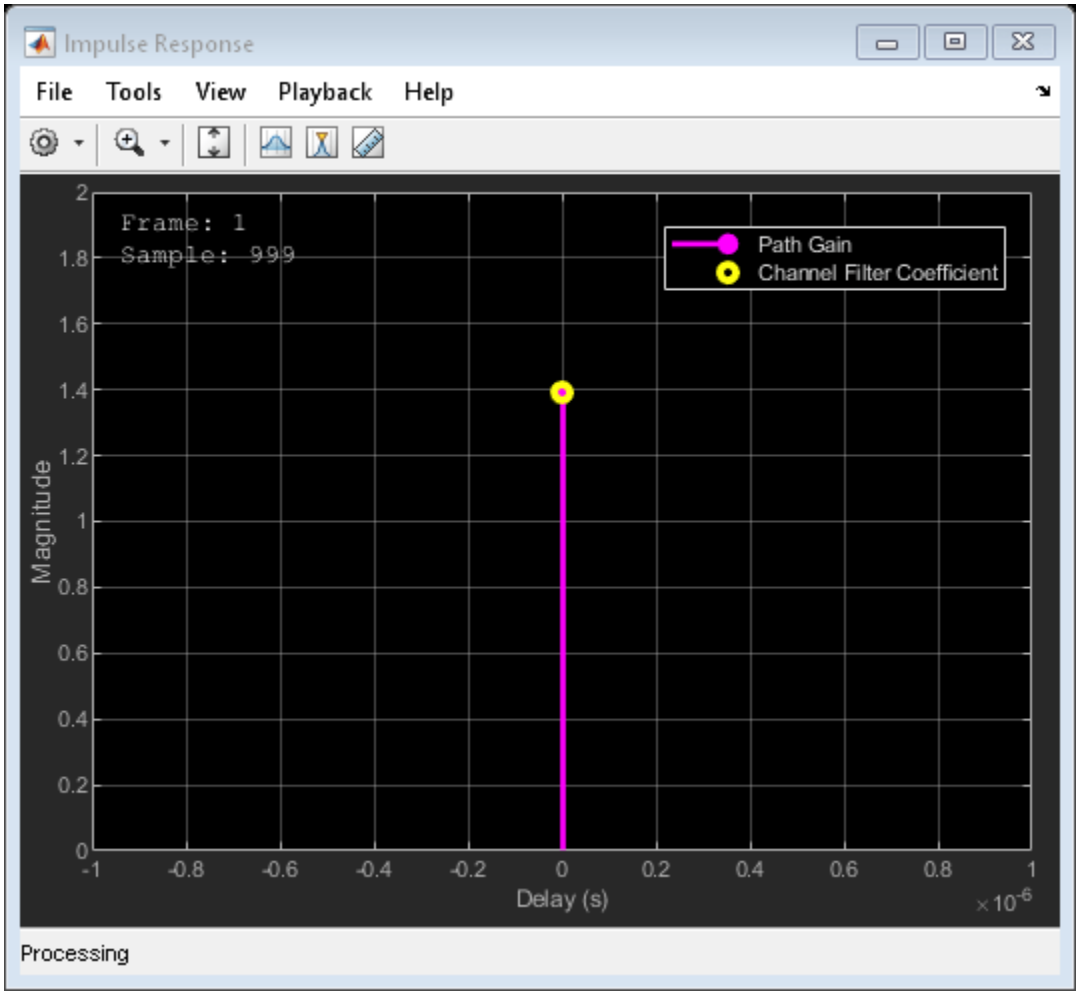
```
1000000
```

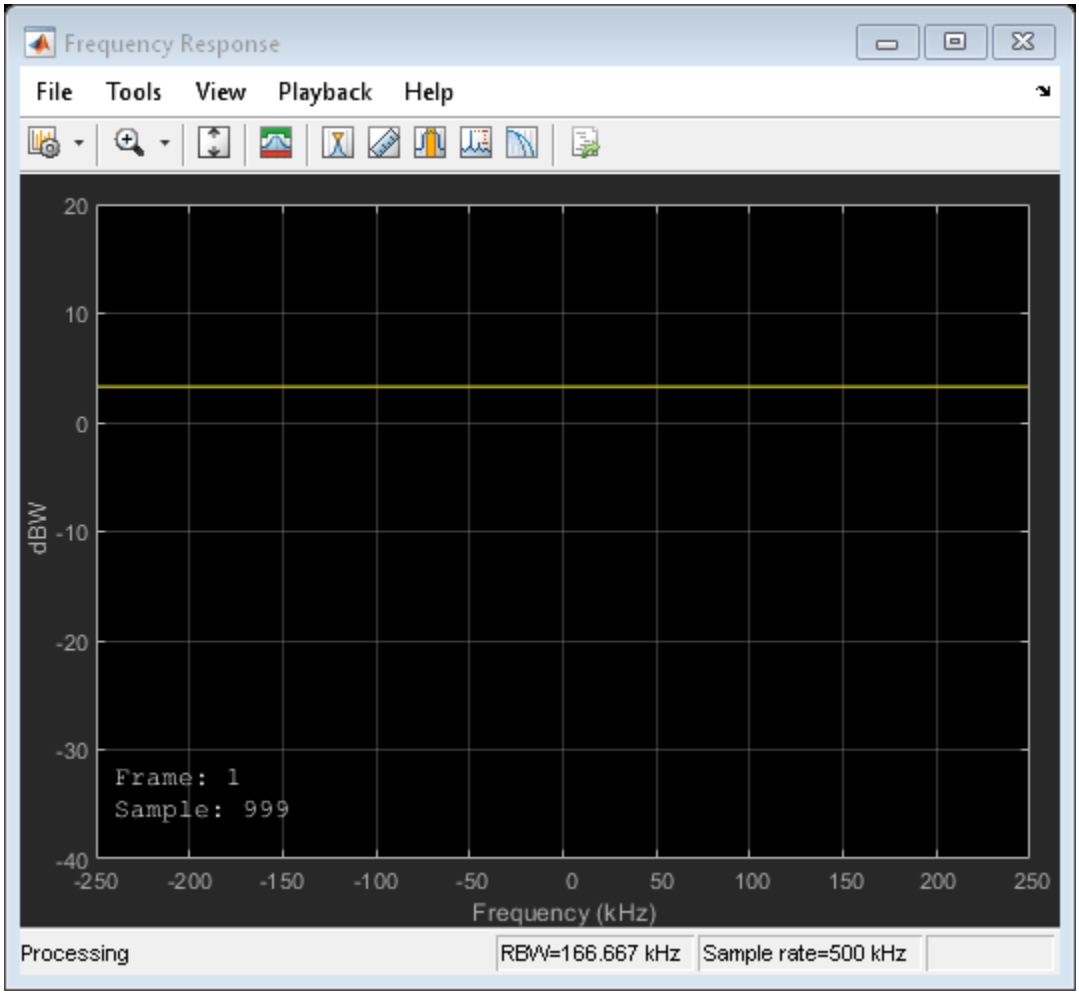
We can visually validate this narrowband fading behavior by setting the `Channel visualization` parameter to 'Impulse and frequency responses' for the Rayleigh channel block and then running the model.





To simplify and speed up simulation, narrowband fading channels are often modeled as a single-path fading channel. That is, a multiple-path fading model overspecifies a narrowband fading channel. The following settings correspond to a narrowband fading channel with a completely flat frequency response.

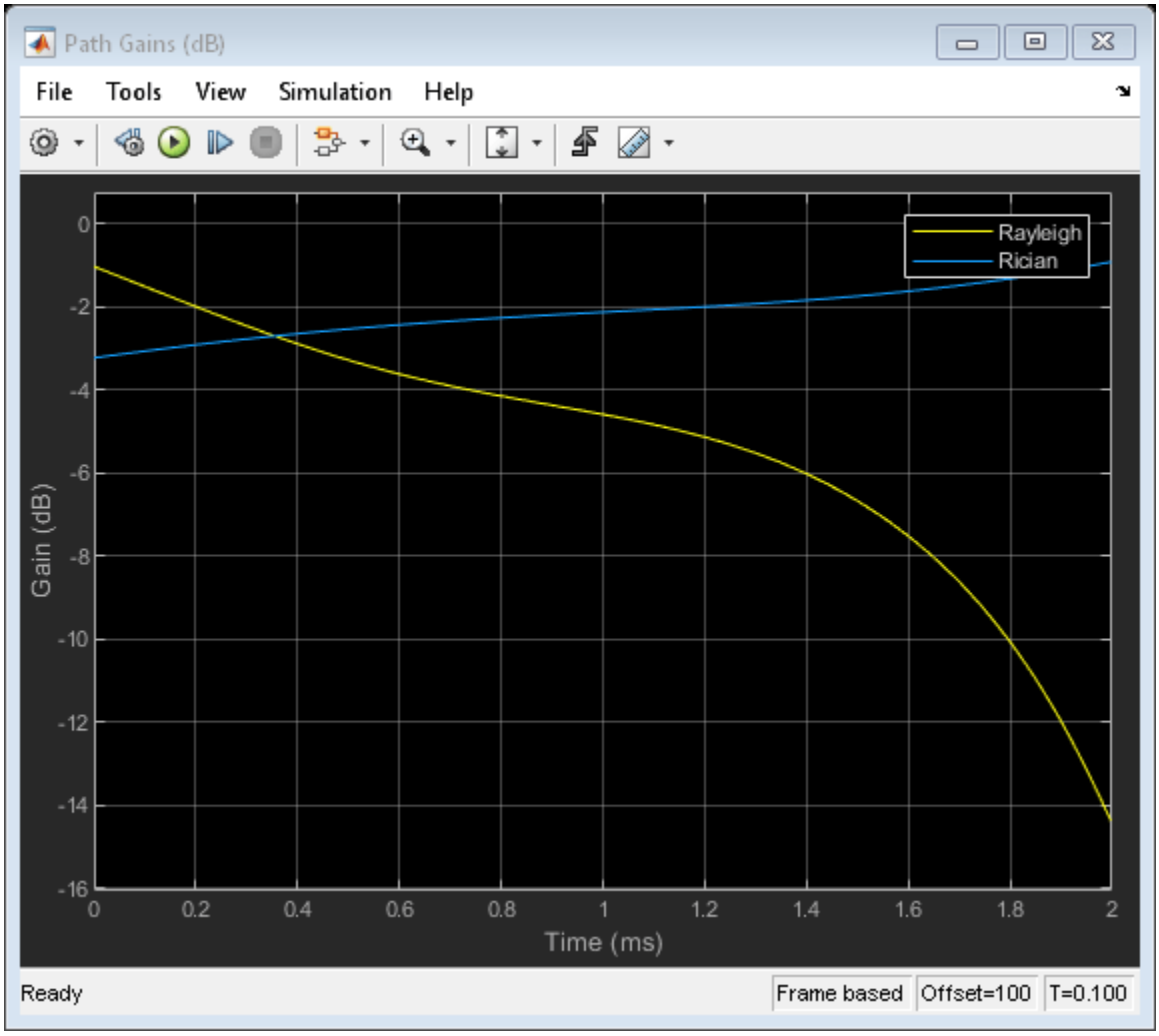

Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 999
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-6
Processing



We now return to our original four-path fading channel and observe how narrowband fading causes signal attenuation and phase rotation, by opening the constellation diagram following the Rayleigh channel block. In addition to attenuation and rotation, you can see some signal distortion because of the small amount of ISI in the channel output signal. The distortion is far less than that seen above for a wideband channel.

Rician Fading

The Rician fading channel block models line-of-sight propagation in addition to diffuse multipath scattering. This results in a smaller variation in the magnitude of path gains. To compare the variation between Rayleigh and Rician channels, we re-configure the channel blocks to model a single-path delay and make use of a Time Scope block to view their path gains over time. Note that the magnitude fluctuates over approximately a 5 dB range for the Rician fading channel, compared with approximate 15 dB for the Rayleigh fading channel. For the Rician fading channel, this variation would be further reduced by increasing the K-factor (currently set to 10).


Path Gains (dB)
File
Tools
View
Simulation
Help
Rayleigh
Rician
Gain (dB)
0
-2
-4
-6
-8
-10
-12
-14
-16
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time (ms)
Ready
Frame based
Offset=100
T=0.100

RF Satellite Link

This model shows a satellite link, using the blocks from the Communications Toolbox™ to simulate the following impairments:

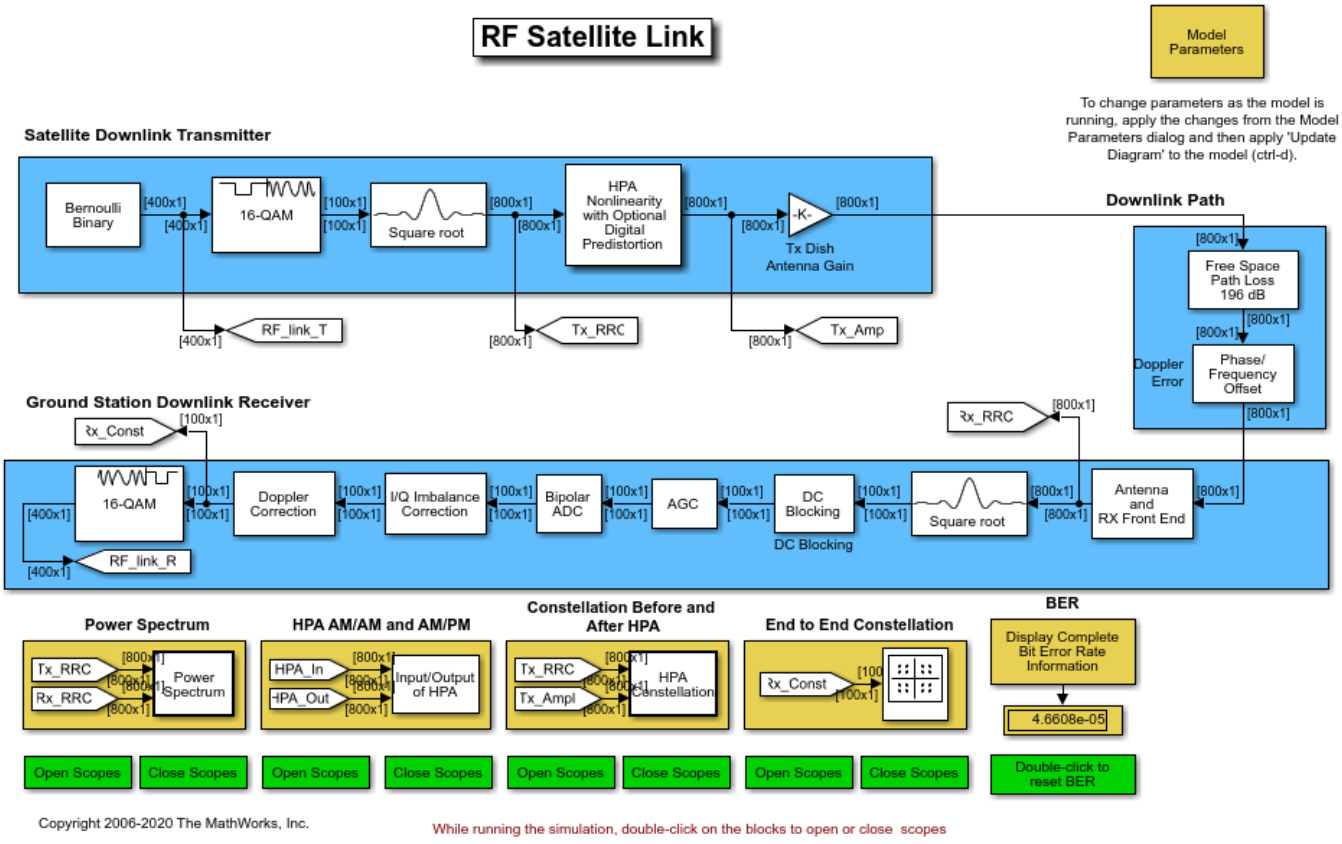
- Memoryless nonlinearity
- Free space path loss
- Doppler error
- Receiver thermal noise
- Phase noise
- In-phase and quadrature imbalances
- DC offsets

The model optionally corrects most of these impairments.

By modeling the gains and losses on the link, this model implements link budget calculations that determine whether a downlink can be closed with a given bit error rate (BER). The gain and loss blocks, including the Free Space Path Loss block and the Receiver Thermal Noise block, determine the data rate that can be supported on the link in an additive white Gaussian noise channel.

Structure of the Example

The example highlights both the satellite link model and its signal scopes. The model consists of a Satellite Downlink Transmitter, Downlink Path, and Ground Station Downlink Receiver.



The blocks that correspond to each of these sections are

Satellite Downlink Transmitter

- Bernoulli Binary Generator - Creates a random binary data stream.
- Rectangular QAM Modulator Baseband - Maps the data stream to 16-QAM constellation.
- Raised Cosine Transmit Filter - Upsamples and shapes the modulated signal using the square root raised cosine pulse shape.
- `HPA Nonlinearity with Optional Digital Predistortion` (High Power Amplifier) - Models a traveling wave tube amplifier (TWTA) using the Saleh model option of the Memoryless Nonlinearity and optionally corrects the AM/AM and AM/PM with a Digital Predistortion block.
- `Gain` (Tx Dish Antenna Gain) - Applies gain of the transmitter parabolic dish antenna.

Downlink Path

- Free Space Path Loss (Downlink Path) - Attenuates the signal by the free space path loss.
- Phase/Frequency Offset (Doppler Error) - Rotates the signal to model Doppler error on the link.

Ground Station Downlink Receiver

- `Gain` (Rx Dish Antenna Gain) - Applies gain of the receiver parabolic dish antenna.
- Receiver Thermal Noise (Satellite Receiver System Temp) - Adds white Gaussian noise that represents the effective system temperature of the receiver.

- Phase Noise - Introduces random phase perturbations that result from 1/f or phase flicker noise.
- I/Q Imbalance - Introduces DC offset, amplitude imbalance, or phase imbalance to the signal.
- `LNA` (Low Noise Amplifier)- Applies low noise amplifier gain.
- Raised Cosine Receive Filter - Applies a matched filter to the modulated signal using the square root raised cosine pulse shape.
- DC Blocker - Compensates for the DC offset in the I/Q Imbalance block.
- AGC - Sets the signal power to a desired level.
- I/Q Imbalance Compensator - Estimates and removes I/Q imbalance from the signal by a blind adaptive algorithm.
- `Doppler Correction` - Uses the Carrier Synchronizer block to compensate for the carrier frequency offset due to Doppler.
- Rectangular QAM Demodulator Baseband - Demaps the data stream from the 16-QAM constellation space.

Exploring the Example

Double-click the block labeled **Model Parameters** to view the parameter settings for the model. All these parameters are tunable. To make changes to the parameters as the model is running, apply them in the dialog, then update the model via ctrl+d. The parameters are:

Satellite altitude (km) - Distance between the satellite and the ground station. Changing this parameter updates the Free Space Path Loss block. The default setting is 35600.

Frequency (MHz) - Carrier frequency of the link. Changing this parameter updates the Free Space Path Loss block. The default setting is 4000.

Transmit and receive antenna diameters (m) - The first element in the vector represents the transmit antenna diameter and is used to calculate the gain in the Tx Dish Antenna Gain block. The second element represents the receive antenna diameter and is used to calculate the gain in the Rx Dish Antenna Gain block. The default setting is [.4 .4].

Noise temperature (K) - Allows you to select from four effective receiver system noise temperatures. The selected noise temperature changes the **Noise Temperature** of the Receiver Thermal Noise block. The default setting is 20 K. The choices are

- `0 (no noise)` - Use this setting to view the other RF impairments without the perturbing effects of noise.
- `20 (very low noise level)` - Use this setting to view how easily a low level of noise can, when combined with other RF impairments, degrade the performance of the link.
- `290 (typical noise level)` - Use this setting to view how a typical quiet satellite receiver operates.
- `500 (high noise level)` - Use this setting to view the receiver behavior when the system noise figure is 2.4 dB and the antenna noise temperature is 290K.

HPA backoff level - Allows you to select from three backoff levels. This parameter is used to determine how close the satellite high power amplifier is driven to saturation. The selected backoff is used to set the input and output gain of the Memoryless Nonlinearity block. The default setting is 30 dB (negligible nonlinearity). The choices are

- `30 dB (negligible nonlinearity)` - Sets the average input power to 30 decibels below the input power that causes amplifier saturation (that is, the point at which the gain curve becomes

flat). This causes negligible AM-to-AM and AM-to-PM conversion. AM-to-AM conversion is an indication of how the amplitude nonlinearity varies with the signal magnitude. AM-to-PM conversion is a measure of how the phase nonlinearity varies with signal magnitude.

- `7 dB (moderate nonlinearity)` - Sets the average input power to 7 decibels below the input power that causes amplifier saturation. This causes moderate AM-to-AM and AM-to-PM conversion, which is correctable with digital predistortion.
- `1 dB (severe nonlinearity)` - Sets the average input power to 1 decibel below the input power that causes amplifier saturation. This causes severe AM-to-AM and AM-to-PM conversion, and is not correctable with digital predistortion.

Doppler error - Allows you to select one of two values of Doppler. The selection updates the Phase/Frequency Offset (Doppler Error) block. The default setting is `0 Hz`. The choices are

- `0 Hz` - No Doppler on the link.
- `3 Hz` - Adds 3 Hz carrier frequency offset.

Phase noise - Allows you to select from three values of phase noise at the receiver. The selection updates the Phase Noise block. The default setting is `Negligible (-100 dBc/Hz @ 100 Hz)`. The choices are

- `Negligible (-100 dBc/Hz @ 100 Hz)` - Almost no phase noise.
- `Low (-55 dBc/Hz @ 100 Hz)` - Enough phase noise to be visible in both the spectral and I/Q domains, and cause bit errors when combined with thermal noise or other RF impairments.
- `High (-48 dBc/Hz @ 100 Hz)` - Enough phase noise to cause errors without the addition of thermal noise or other RF impairments.

I/Q imbalance and DC offset - Allows you to select from five types of in-phase and quadrature imbalances at the receiver. The selection updates the I/Q Imbalance block. The default setting is `None`. The choices are

- `None` - No imbalances.
- `Amplitude imbalance (3 dB)` - Applies a 1.5 dB gain to the in-phase signal and a -1.5 dB gain to the quadrature signal.
- `Phase imbalance (20 deg)` - Rotates the in-phase signal by 10 degrees and the quadrature signal by -10 degrees.
- `In-phase DC offset (1e-8)` - Adds a DC offset of 1e-8 to the in-phase signal amplitude. This offset changes the received signal constellation diagram, but does not cause errors on the link unless combined with thermal noise or other RF impairments.
- `Quadrature DC offset (5e-8)` - Adds a DC offset of 5e-8 to the quadrature signal amplitude. This offset causes errors on the link even when not combined with thermal noise or another RF impairment. This offset also causes a DC spike in the received signal spectrum.

Digital predistortion - Allows you to enable or disable the Digital Predistortion subsystem. The default setting is `Disabled`.

DC offset correction - Allows you to enable or disable the DC Blocking subsystem. The default setting is `Disabled`.

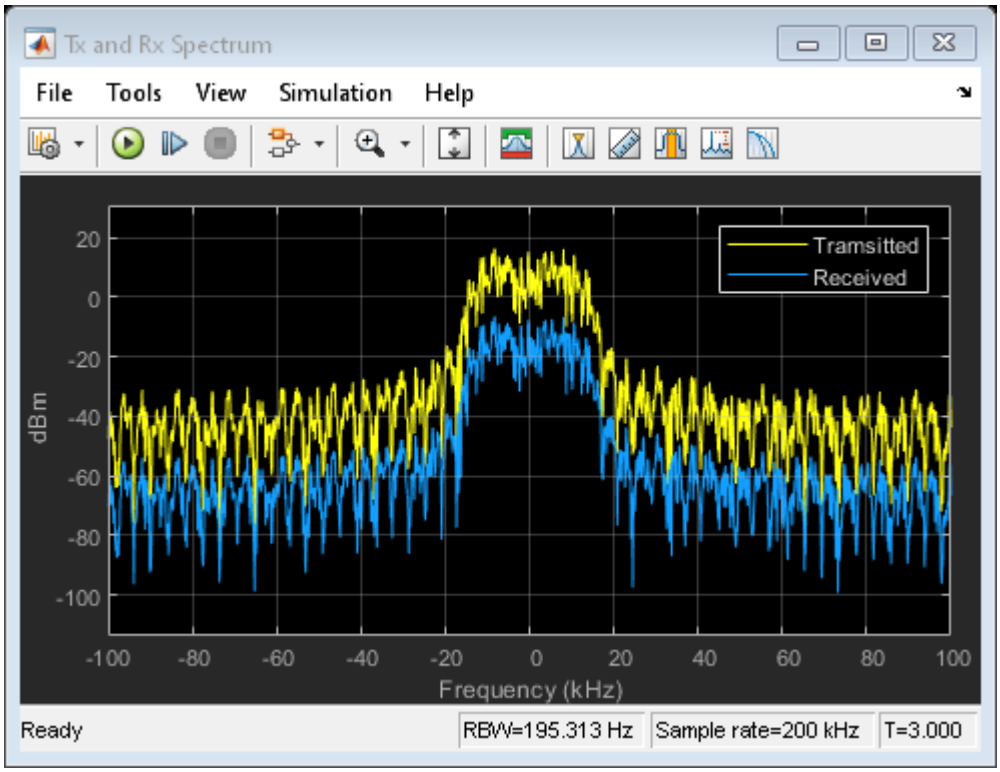
Doppler correction - Allows you to enable or disable the Doppler Correction subsystem. The default setting is `Disabled`.

I/Q imbalance correction - Allows you to enable or disable the I/Q Imbalance Correction subsystem. The default setting is `Disabled`.

Results and Displays

When you run this model, the following displays are active:

Power Spectrum - Double-clicking this Open Scopes block enables you to view the spectrum of the modulated/filtered signal (yellow) and the received signal before demodulation (blue).

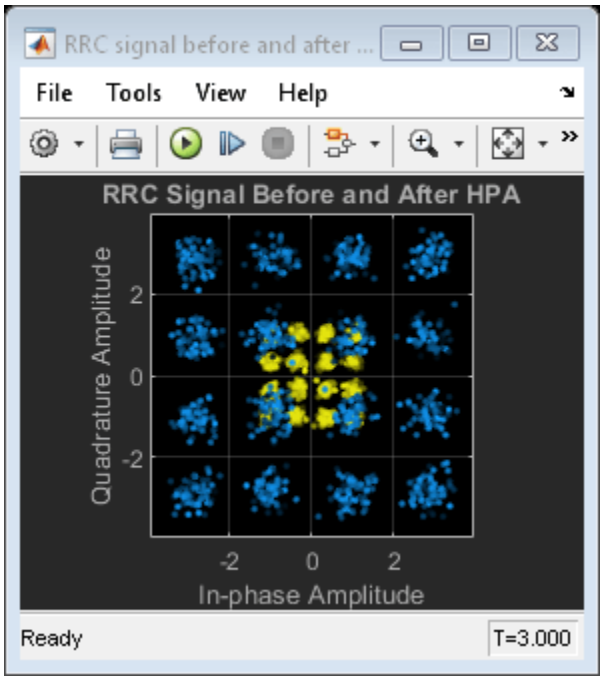


Comparing the two spectra allows you to view the effect of the following RF impairments:

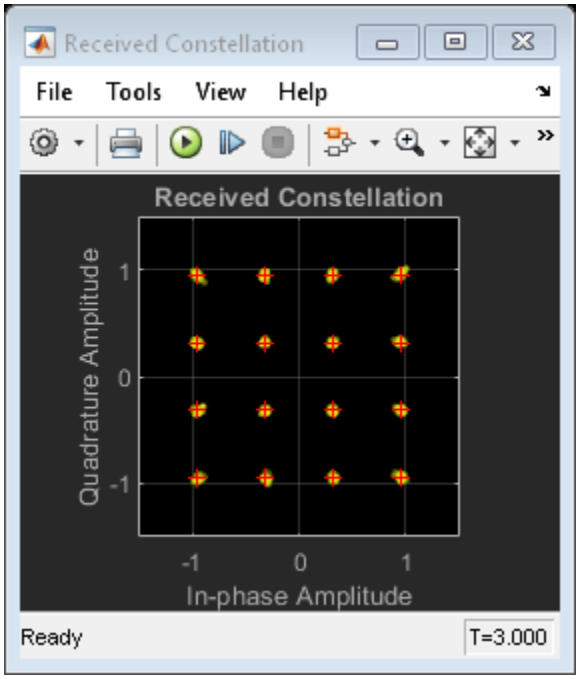
- Spectral regrowth due to HPA nonlinearities caused by the Memoryless Nonlinearity block
- Thermal noise caused by the Receiver Thermal Noise block
- Phase flicker (that is, 1/f noise) caused by the Phase Noise block

HPA AM/AM and AM/PM - Double-clicking this Open Scopes block enables you to view the AM/AM and AM/PM conversion after the HPA. These plots enable you to view the impact that the Digital Predistortion block and HPA have on the linearity of the signal.

Constellation Before and After HPA - Double-clicking this Open Scopes block enables you to compare the constellation of the transmitted signal before (yellow) and after (blue) the HPA. The amplifier gain causes the HPA Output signal to be larger than the HPA Input signal. This plot enables you to view the combined effect of both the HPA nonlinearity and digital predistortion.



End to End Constellation - Double-clicking this Open Scopes block enables you to compare the reference 16-QAM constellation (red) with the received QAM constellation before demodulation (yellow). Comparing these constellation diagrams allows you to view the impact of all the RF impairments on the received signal and the effectiveness of the compensations.



Bit error rate (BER) display - In the lower right corner of the model is a display of the BER of the model. The BER computation can be reset manually by double-clicking the green "Double-click to

reset BER" button. This allows you to view the impact of the parameter changes as the model is running.

Experimenting with the Example

This section describes some ways that you can change the model parameters to experiment with the effects of the blocks from the RF Impairments library and other blocks in the model. You can double-click the block labeled "Model Parameters" in the model and try some of the following scenarios:

Link gains and losses - Change **Noise temperature** to `290 (typical noise level)`, `0 (no noise)` or `500 (high noise level)`. Change the value of the **Satellite altitude (km)** or **Satellite frequency (MHz)** parameters to change the free space path loss. In addition, increase or decrease the **Transmit and receive antenna diameters (m)** parameter to increase or decrease the received signal power. You can view the changes in the received constellation in the received signal constellation diagram scope and the changes in received power in the spectrum analyzer.

Raised cosine pulse shaping - Make sure **Noise temperature** is set to `0 (no noise)`. Turn on the Constellation Before and After HPA scopes. Observe that the square-root raised cosine filtering results in intersymbol interference (ISI). This results in the points being scattered loosely around ideal constellation points, which you can see in the After HPA constellation diagram. The square-root raised cosine filter in the receiver, in conjunction with the transmit filter, controls the ISI, which you can see in the received signal constellation diagram.

HPA AM-to-AM conversion and AM-to-PM conversion - Change the **HPA backoff level** parameter to `7 dB (moderate nonlinearity)` and observe the AM-to-AM and AM-to-PM conversions by comparing the Transmit RRC filtered signal constellation diagram with the RRC signal after HPA constellation diagram. Note how the AM-to-AM conversion varies according to the different signal amplitudes. You can also view the effect of this conversion on the received signal in the received signal constellation diagram. In addition, you can observe the spectral regrowth in the received signal spectrum analyzer. You can also view the phase change in the received signal in the received signal constellation diagram scope.

Digital predistortion With the Digital predistortion checkbox checked, change the **HPA backoff level** parameter to `30 dB (negligible nonlinearity)`, `7 dB (moderate nonlinearity)`, and `1 dB (severe nonlinearity)` to view the effect of digital predistortion on the HPA nonlinearity.

Phase noise plus AM-to-AM conversion - Set the **Phase Noise** parameter to `High` and observe the increased variance in the tangential direction in the received signal constellation diagram. Also note that this level of phase noise is sufficient to cause errors in an otherwise error-free channel.

DC offset and DC offset compensation - Set the **I/Q imbalance and DC offset** parameter to `In-phase DC offset (1e-8)` and view the shift of the constellation in the received signal constellation diagram. Set **DC offset correction** to `Enabled` and view the received signal constellation diagram to view how the DC offset block estimates the DC offset value and removes it from the signal. Set **DC offset compensation** to `Disabled` and change **I/Q imbalance** to `Quadrature DC offset (5e-8)`. View the changes in the received signal constellation diagram for a large DC offset and the DC spike in the received signal spectrum. Note that the LNA amplifies the small DC offsets so that they are visible on the constellation diagram with much larger axis limits. Set **DC offset compensation** to `Enabled` and view the received signal constellation diagram and spectrum analyzer to see how the DC component is removed.

Amplitude imbalance - With the **I/Q imbalance correction** disabled, set the **I/Q Imbalance and DC offset** parameter to `Amplitude imbalance (3 dB)` to view the effect of unbalanced I and Q

gains in the received signal constellation diagram. Enable the **I/Q imbalance correction** to compensate for the amplitude imbalance.

Doppler and Doppler compensation - Disable Doppler correction by unchecking the **Doppler correction** check box. Set **Doppler error** to `3 Hz` to show the effect of uncorrected Doppler on the received signal constellation diagram. Enable **Doppler correction** to show that the carrier synchronizer restores the received constellation. Repeat the exercise with different I/Q imbalance and DC offsets.

Selected Bibliography

[1] Saleh, Adel A.M., "Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers," IEEE® Transactions on Communications, Vol. COM-29, No. 11, November 1981.

[2] Kasdin, N.J., "Discrete Simulation of Colored Noise and Stochastic Processes and 1/(f^alpha); Power Law Noise Generation," The Proceedings of the IEEE, Vol. 83, No. 5, May, 1995.

[3] Kasdin, N. Jeremy, and Todd Walter, "Discrete Simulation of Power Law Noise," 1992 IEEE Frequency Control Symposium.

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

Introduction to MIMO Systems

This example shows Multiple-Input-Multiple-Output (MIMO) systems, which use multiple antennas at the transmitter and receiver ends of a wireless communication system. MIMO systems are increasingly being adopted in communication systems for the potential gains in capacity they realize when using multiple antennas. Multiple antennas use the spatial dimension in addition to the time and frequency ones, without changing the bandwidth requirements of the system.

For a generic communications link, this example focuses on transmit diversity in lieu of traditional receive diversity. Using the flat-fading Rayleigh channel, it illustrates the concept of Orthogonal Space-Time Block Coding, which is employable when multiple transmitter antennas are used. It is assumed here that the channel undergoes independent fading between the multiple transmit-receive antenna pairs.

For a chosen system, it also provides a measure of the performance degradation when the channel is imperfectly estimated at the receiver, compared to the case of perfect channel knowledge at the receiver.

PART 1: Transmit Diversity vs. Receive Diversity

Using diversity reception is a well-known technique to mitigate the effects of fading over a communications link. However, it has mostly been relegated to the receiver end. In [1], Alamouti proposes a transmit diversity scheme that offers similar diversity gains, using multiple antennas at the transmitter. This was conceived to be more practical as, for example, it would only require multiple antennas at the base station in comparison to multiple antennas for every mobile in a cellular communications system.

This section highlights this comparison of transmit vs. receive diversity by simulating coherent binary phase-shift keying (BPSK) modulation over flat-fading Rayleigh channels. For transmit diversity, we use two transmit antennas and one receive antenna (2x1 notationally), while for receive diversity we employ one transmit antenna and two receive antennas (1x2 notationally).

The simulation covers an end-to-end system showing the encoded and/or transmitted signal, channel model, and reception and demodulation of the received signal. It also provides the no-diversity link (single transmit- receive antenna case) and theoretical performance of second-order diversity link for comparison. It is assumed here that the channel is known perfectly at the receiver for all systems. We run the simulation over a range of Eb/No points to generate BER results that allow us to compare the different systems.

We start by defining some common simulation parameters

```
frmLen = 100;       % frame length
numPackets = 1000;  % number of packets
EbNo = 0:2:20;      % Eb/No varying to 20 dB
N = 2;              % maximum number of Tx antennas
M = 2;              % maximum number of Rx antennas
```

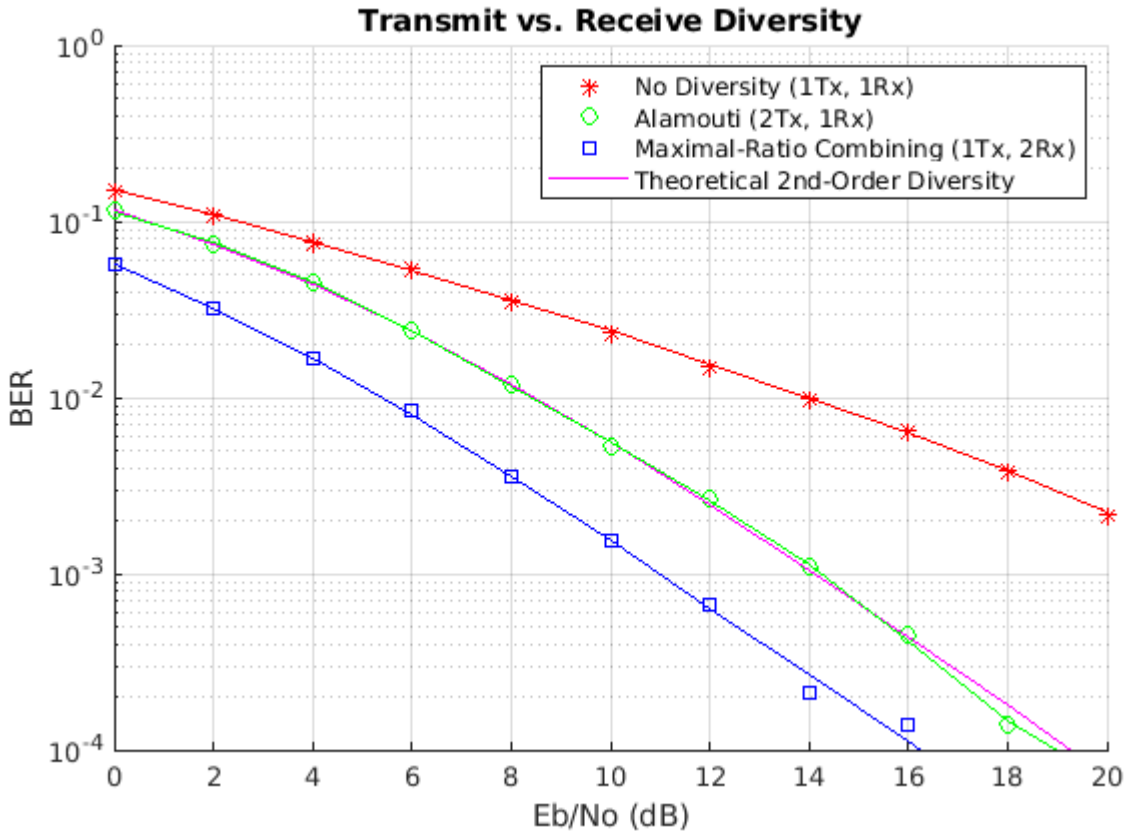
and set up the simulation.

```
% Create comm.BPSKModulator and comm.BPSKDemodulator System objects(TM)
P = 2;                % modulation order
bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('OutputDataType','double');

% Create comm.OSTBCEncoder and comm.OSTBCCombiner System objects
```

```
ostbcEnc = comm.OSTBCEncoder;
ostbcComb = comm.OSTBCCombiner;

% Create two comm.AWGNChannel System objects for one and two receive
% antennas respectively. Set the NoiseMethod property of the channel to
% 'Signal to noise ratio (Eb/No)' to specify the noise level using the
% energy per bit to noise power spectral density ratio (Eb/No). The output
% of the BPSK modulator generates unit power signals; set the SignalPower
% property to 1 Watt.
awgn1Rx = comm.AWGNChannel(...
    'NoiseMethod', 'Signal to noise ratio (Eb/No)', ...
    'SignalPower', 1);
awgn2Rx = clone(awgn1Rx);

% Create comm.ErrorRate calculator System objects to evaluate BER.
errorCalc1 = comm.ErrorRate;
errorCalc2 = comm.ErrorRate;
errorCalc3 = comm.ErrorRate;

% Since the comm.AWGNChannel System objects as well as the RANDI function
% use the default random stream, the following commands are executed so
% that the results will be repeatable, i.e., same results will be obtained
% for every run of the example. The default stream will be restored at the
% end of the example.
s = rng(55408);

% Pre-allocate variables for speed
H = zeros(frmLen, N, M);
ber_noDiver  = zeros(3,length(EbNo));
ber_Alamouti = zeros(3,length(EbNo));
ber_MaxRatio = zeros(3,length(EbNo));
ber_thy2     = zeros(1,length(EbNo));

% Set up a figure for visualizing BER results
fig = figure;
grid on;
ax = fig.CurrentAxes;
hold(ax,'on');

ax.YScale = 'log';
xlim(ax,[EbNo(1), EbNo(end)]);
ylim(ax,[1e-4 1]);
xlabel(ax,'Eb/No (dB)');
ylabel(ax,'BER');
fig.NumberTitle = 'off';
fig.Renderer = 'zbuffer';
fig.Name = 'Transmit vs. Receive Diversity';
title(ax,'Transmit vs. Receive Diversity');
set(fig, 'DefaultLegendAutoUpdate', 'off');
fig.Position = figposition([15 50 25 30]);

% Loop over several EbNo points
for idx = 1:length(EbNo)
    reset(errorCalc1);
    reset(errorCalc2);
    reset(errorCalc3);
    % Set the EbNo property of the AWGNChannel System objects
    awgn1Rx.EbNo = EbNo(idx);
```

```
awgn2Rx.EbNo = EbNo(idx);
% Loop over the number of packets
for packetIdx = 1:numPackets
    % Generate data vector per frame
    data = randi([0 P-1], frmLen, 1);

    % Modulate data
    modData = bpskMod(data);

    % Alamouti Space-Time Block Encoder
    encData = ostbcEnc(modData);

    % Create the Rayleigh distributed channel response matrix
    %   for two transmit and two receive antennas
    H(1:N:end, :, :) = (randn(frmLen/2, N, M) + ...
                        1i*randn(frmLen/2, N, M))/sqrt(2);
    %   assume held constant for 2 symbol periods
    H(2:N:end, :, :) = H(1:N:end, :, :);

    % Extract part of H to represent the 1x1, 2x1 and 1x2 channels
    H11 = H(:,1,1);
    H21 = H(:,:,1)/sqrt(2);
    H12 = squeeze(H(:,1,:));

    % Pass through the channels
    chanOut11 = H11 .* modData;
    chanOut21 = sum(H21.* encData, 2);
    chanOut12 = H12 .* repmat(modData, 1, 2);

    % Add AWGN
    rxSig11 = awgn1Rx(chanOut11);
    rxSig21 = awgn1Rx(chanOut21);
    rxSig12 = awgn2Rx(chanOut12);

    % Alamouti Space-Time Block Combiner
    decData = ostbcComb(rxSig21, H21);

    % ML Detector (minimum Euclidean distance)
    demod11 = bpskDemod(rxSig11.*conj(H11));
    demod21 = bpskDemod(decData);
    demod12 = bpskDemod(sum(rxSig12.*conj(H12), 2));

    % Calculate and update BER for current EbNo value
    %   for uncoded 1x1 system
    ber_noDiver(:,idx)  = errorCalc1(data, demod11);
    %   for Alamouti coded 2x1 system
    ber_Alamouti(:,idx) = errorCalc2(data, demod21);
    %   for Maximal-ratio combined 1x2 system
    ber_MaxRatio(:,idx) = errorCalc3(data, demod12);

end % end of FOR loop for numPackets

% Calculate theoretical second-order diversity BER for current EbNo
ber_thy2(idx) = berfading(EbNo(idx), 'psk', 2, 2);

% Plot results
semilogy(ax,EbNo(1:idx), ber_noDiver(1,1:idx), 'r*', ...
         EbNo(1:idx), ber_Alamouti(1,1:idx), 'go', ...
```

```
            EbNo(1:idx), ber_MaxRatio(1,1:idx), 'bs', ...
            EbNo(1:idx), ber_thy2(1:idx), 'm');
    legend(ax,'No Diversity (1Tx, 1Rx)', 'Alamouti (2Tx, 1Rx)',...
           'Maximal-Ratio Combining (1Tx, 2Rx)', ...
           'Theoretical 2nd-Order Diversity');

    drawnow;
end  % end of for loop for EbNo

% Perform curve fitting and replot the results
fitBER11 = berfit(EbNo, ber_noDiver(1,:));
fitBER21 = berfit(EbNo, ber_Alamouti(1,:));
fitBER12 = berfit(EbNo, ber_MaxRatio(1,:));
semilogy(ax,EbNo, fitBER11, 'r', EbNo, fitBER21, 'g', EbNo, fitBER12, 'b');
hold(ax,'off');

% Restore default stream
rng(s);
```



The transmit diversity system has a computation complexity very similar to that of the receive diversity system.

The resulting simulation results show that using two transmit antennas and one receive antenna provides the same diversity order as the maximal-ratio combined (MRC) system of one transmit antenna and two receive antennas.

Also observe that transmit diversity has a 3 dB disadvantage when compared to MRC receive diversity. This is because we modeled the total transmitted power to be the same in both cases. If we calibrate the transmitted power such that the received power for these two cases is the same, then the performance would be identical. The theoretical performance of second-order diversity link matches the transmit diversity system as it normalizes the total power across all the diversity branches.

The accompanying functional scripts, mrc1m.m and ostbc2m.m aid further exploration for the interested users.

PART 2: Space-Time Block Coding with Channel Estimation

Building on the theory of orthogonal designs, Tarokh et al. [2] generalized Alamouti's transmit diversity scheme to an arbitrary number of transmitter antennas, leading to the concept of Space-Time Block Codes. For complex signal constellations, they showed that Alamouti's scheme is the only full-rate scheme for two transmit antennas.

In this section, we study the performance of such a scheme with two receive antennas (i.e., a 2x2 system) with and without channel estimation. In the realistic scenario where the channel state information is not known at the receiver, this has to be extracted from the received signal. We assume that the channel estimator performs this using orthogonal pilot signals that are prepended to every packet [3]. It is assumed that the channel remains unchanged for the length of the packet (i.e., it undergoes slow fading).

A simulation similar to the one described in the previous section is employed here, which leads us to estimate the BER performance for a space-time block coded system using two transmit and two receive antennas.

Again we start by defining the common simulation parameters

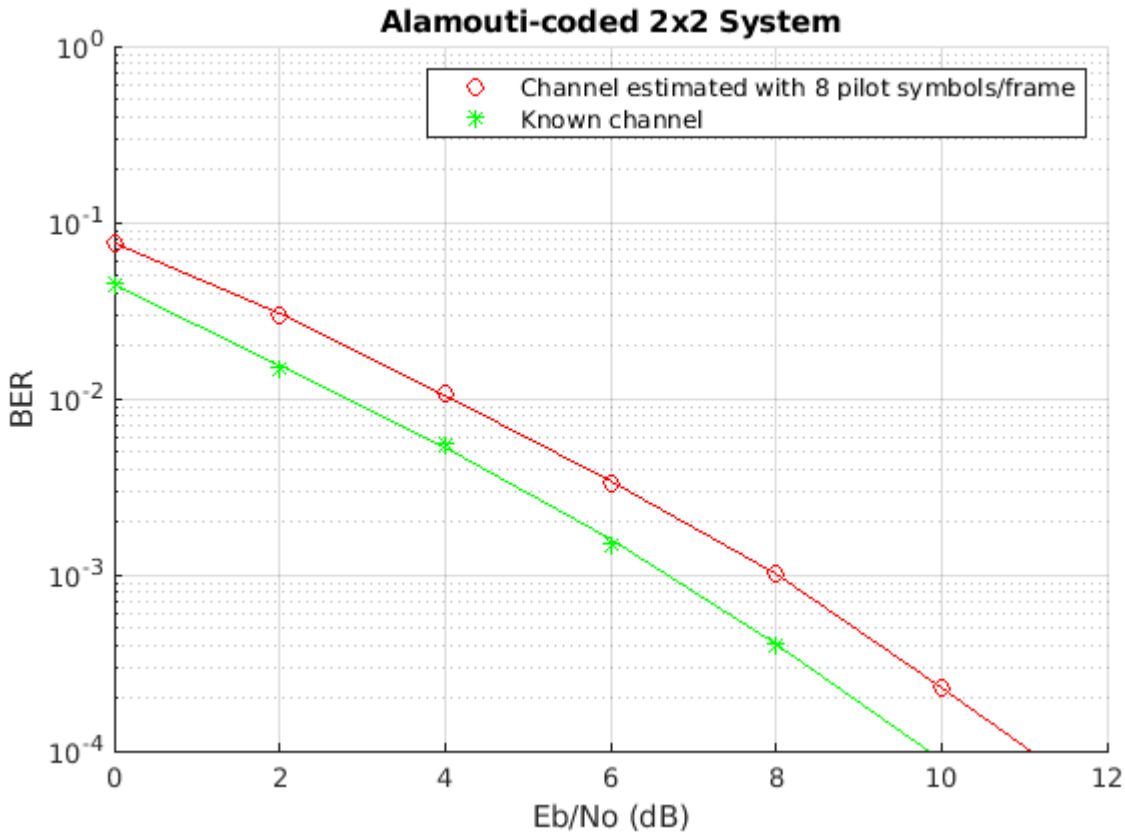
```
frmLen = 100;           % frame length
maxNumErrs = 300;       % maximum number of errors
maxNumPackets = 3000;   % maximum number of packets
EbNo = 0:2:12;          % Eb/No varying to 12 dB
N = 2;                  % number of Tx antennas
M = 2;                  % number of Rx antennas
pLen = 8;               % number of pilot symbols per frame
W = hadamard(pLen);
pilots = W(:, 1:N);     % orthogonal set per transmit antenna
```

and set up the simulation.

```
% Create a comm.MIMOChannel System object to simulate the 2x2 spatially
% independent flat-fading Rayleigh channel
chan = comm.MIMOChannel( ...
    'MaximumDopplerShift', 0, ...
    'SpatialCorrelationSpecification', 'None', ...
    'NumTransmitAntennas', N, ...
    'NumReceiveAntennas', M, ...
    'PathGainsOutputPort', true);

% Change the NumReceiveAntennas property value of the hAlamoutiDec System
% object to M that is 2
release(ostbcComb);
ostbcComb.NumReceiveAntennas = M;
```

```
% Release the hAWGN2Rx System object
release(awgn2Rx);

% Set the global random stream for repeatability
s = rng(55408);

% Pre-allocate variables for speed
HEst = zeros(frmLen, N, M);
ber_Estimate = zeros(3,length(EbNo));
ber_Known    = zeros(3,length(EbNo));

% Set up a figure for visualizing BER results
fig = figure;
grid on;
ax = fig.CurrentAxes;
hold(ax,'on');

ax.YScale = 'log';
xlim(ax,[EbNo(1), EbNo(end)]);
ylim(ax,[1e-4 1]);
xlabel(ax,'Eb/No (dB)');
ylabel(ax,'BER');
fig.NumberTitle = 'off';
fig.Name = 'Orthogonal Space-Time Block Coding';
fig.Renderer = 'zbuffer';
title(ax,'Alamouti-coded 2x2 System');
set(fig,'DefaultLegendAutoUpdate','off');
fig.Position = figposition([41 50 25 30]);

% Loop over several EbNo points
for idx = 1:length(EbNo)
    reset(errorCalc1);
    reset(errorCalc2);
    awgn2Rx.EbNo = EbNo(idx);

    % Loop till the number of errors exceed 'maxNumErrs'
    % or the maximum number of packets have been simulated
    while (ber_Estimate(2,idx) < maxNumErrs) && ...
          (ber_Known(2,idx) < maxNumErrs) && ...
          (ber_Estimate(3,idx)/frmLen < maxNumPackets)
        % Generate data vector per frame
        data = randi([0 P-1], frmLen, 1);

        % Modulate data
        modData = bpskMod(data);

        % Alamouti Space-Time Block Encoder
        encData = ostbcEnc(modData);

        % Prepend pilot symbols for each frame
        txSig = [pilots; encData];

        % Pass through the 2x2 channel
        reset(chan);
        [chanOut, H] = chan(txSig);

        % Add AWGN
        rxSig = awgn2Rx(chanOut);
```

```
        % Channel Estimation
        %   For each link => N*M estimates
        HEst(1,:,:) = pilots(:,:).' * rxSig(1:pLen, :) / pLen;
        %   assume held constant for the whole frame
        HEst = HEst(ones(frmLen, 1), :, :);

        % Combiner using estimated channel
        decDataEst = ostbcComb(rxSig(pLen+1:end,:), HEst);

        % Combiner using known channel
        decDataKnown = ostbcComb(rxSig(pLen+1:end,:), ...
                            squeeze(H(pLen+1:end,:,:,:)));

        % ML Detector (minimum Euclidean distance)
        demodEst   = bpskDemod(decDataEst);      % estimated
        demodKnown = bpskDemod(decDataKnown);    % known

        % Calculate and update BER for current EbNo value
        %   for estimated channel
        ber_Estimate(:,idx) = errorCalc1(data, demodEst);
        %   for known channel
        ber_Known(:,idx)    = errorCalc2(data, demodKnown);

    end % end of FOR loop for numPackets

    % Plot results
    semilogy(ax,EbNo(1:idx), ber_Estimate(1,1:idx), 'ro');
    semilogy(ax,EbNo(1:idx), ber_Known(1,1:idx), 'g*');
    legend(ax,['Channel estimated with ' num2str(pLen) ' pilot symbols/frame'],...
           'Known channel');
    drawnow;
end  % end of for loop for EbNo

% Perform curve fitting and replot the results
fitBEREst   = berfit(EbNo, ber_Estimate(1,:));
fitBERKnown = berfit(EbNo, ber_Known(1,:));
semilogy(ax,EbNo, fitBEREst, 'r', EbNo, fitBERKnown, 'g');
hold(ax,'off');

% Restore default stream
rng(s)
```



For the 2x2 simulated system, the diversity order is different than that seen for either 1x2 or 2x1 systems in the previous section.

Note that with 8 pilot symbols for each 100 symbols of data, channel estimation causes about a 1 dB degradation in performance for the selected Eb/No range. This improves with an increase in the number of pilot symbols per frame but adds to the overhead of the link. In this comparison, we keep the transmitted SNR per symbol to be the same in both cases.

The accompanying functional script, ostbc2m_e.m aids further experimentation for the interested users.

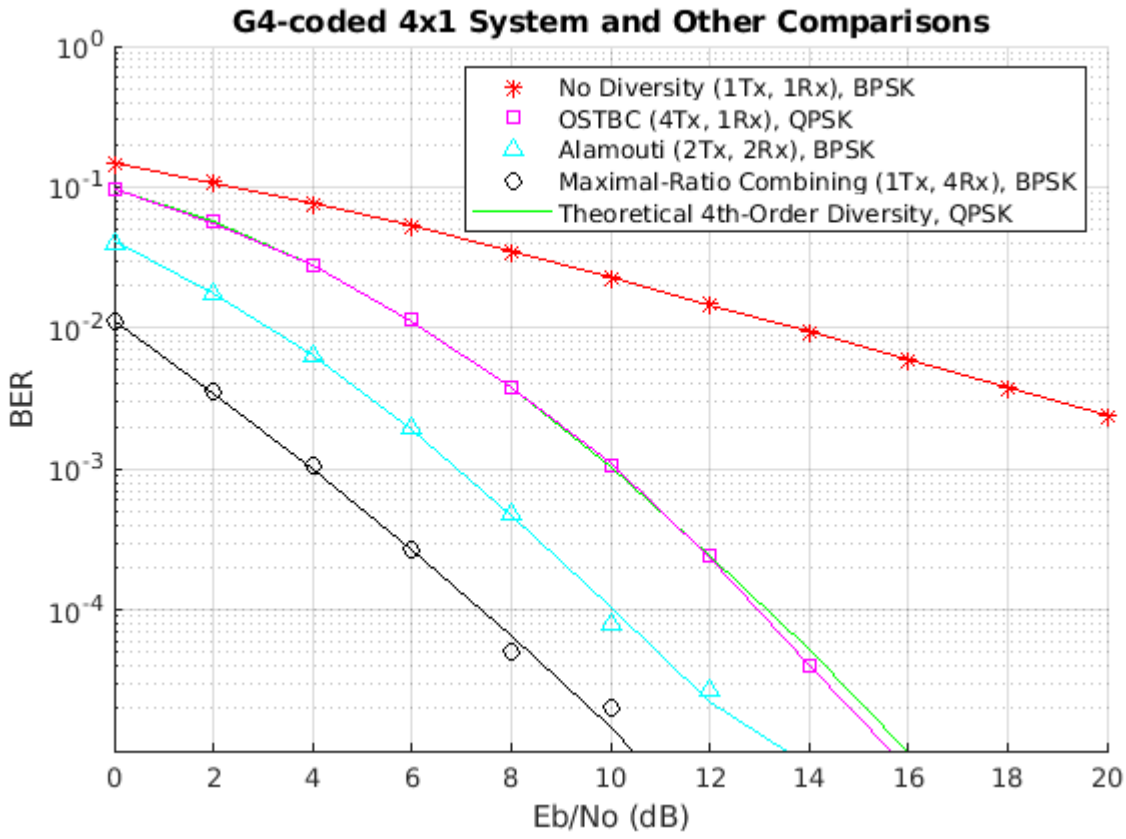
PART 3: Orthogonal Space-Time Block Coding and Further Explorations

In this final section, we present some performance results for orthogonal space-time block coding using four transmit antennas (4x1 system) using a half-rate code, G4, as per [4].

We expect the system to offer a diversity order of 4 and will compare it with 1x4 and 2x2 systems, which have the same diversity order also. To allow for a fair comparison, we use quaternary PSK with the half-rate G4 code to achieve the same transmission rate of 1 bit/sec/Hz.

These results take some time to generate on a single core. If you do not have Parallel Computing Toolbox™ (PCT) installed, we load the results from a prior simulation. The functional script ostbc4m.m is included, which, along with mrc1m.m and ostbc2m.m, was used to generate these results. If PCT is installed, these simulations are performed in parallel. In this case the functional scripts ostbc4m_pct.m, mrc1m_pct.m and ostbc2m_pct.m are used. The user is urged to use these scripts as a starting point to study other codes and systems.

```
[licensePCT,~] = license( 'checkout' , 'Distrib_Computing_Toolbox');
if (licensePCT && ~isempty(ver('parallel')))
    EbNo = 0:2:20;
    [ber11, ber14, ber22, ber41] = mimoOSTBCWithPCT(100,4e3,EbNo);
else
    load ostbcRes.mat;
end

% Set up a figure for visualizing BER results
fig = figure;
grid on;
ax = fig.CurrentAxes;
hold(ax,'on');
fig.Renderer  = 'zbuffer';
ax.YScale = 'log';
xlim(ax,[EbNo(1), EbNo(end)]);
ylim(ax,[1e-5 1]);
xlabel(ax,'Eb/No (dB)');
ylabel(ax,'BER');
fig.NumberTitle = 'off';
fig.Name = 'Orthogonal Space-Time Block Coding(2)';
title(ax,'G4-coded 4x1 System and Other Comparisons');
set(fig,'DefaultLegendAutoUpdate','off');
fig.Position = figposition([30 15 25 30]);

% Theoretical performance of fourth-order diversity for QPSK
BERthy4 = berfading(EbNo, 'psk', 4, 4);

% Plot results
semilogy(ax,EbNo, ber11, 'r*', EbNo, ber41, 'ms', EbNo, ber22, 'c^', ...
         EbNo, ber14, 'ko', EbNo, BERthy4, 'g');
legend(ax,'No Diversity (1Tx, 1Rx), BPSK', 'OSTBC (4Tx, 1Rx), QPSK', ...
       'Alamouti (2Tx, 2Rx), BPSK', 'Maximal-Ratio Combining (1Tx, 4Rx), BPSK', ...
       'Theoretical 4th-Order Diversity, QPSK');

% Perform curve fitting
fitBER11 = berfit(EbNo, ber11);
fitBER41 = berfit(EbNo(1:9), ber41(1:9));
fitBER22 = berfit(EbNo(1:8), ber22(1:8));
fitBER14 = berfit(EbNo(1:7), ber14(1:7));
semilogy(ax,EbNo, fitBER11, 'r', EbNo(1:9), fitBER41, 'm', ...
         EbNo(1:8), fitBER22, 'c', EbNo(1:7), fitBER14, 'k');
hold(ax,'off');

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```



As expected, the similar slopes of the BER curves for the 4x1, 2x2 and 1x4 systems indicate an identical diversity order for each system.

Also observe the 3 dB penalty for the 4x1 system that can be attributed to the same total transmitted power assumption made for each of the three systems. If we calibrate the transmitted power such that the received power for each of these systems is the same, then the three systems would perform identically. Again, the theoretical performance matches the simulation performance of the 4x1 system as the total power is normalized across the diversity branches.

Appendix

This example uses the following helper functions:

- mrc1m.m
- ostbc2m.m
- ostbc4m.m
- mimoOSTBCWithPCT.m
- mrc1m_pct.m
- ostbc2m_pct.m
- ostbc4m_pct.m

References

1. S. M. Alamouti, "A simple transmit diversity technique for wireless communications", IEEE® Journal on Selected Areas in Communications, Vol. 16, No. 8, Oct. 1998, pp. 1451-1458.
2. V. Tarokh, H. Jafarkhami, and A.R. Calderbank, "Space-time block codes from orthogonal designs", IEEE Transactions on Information Theory, Vol. 45, No. 5, Jul. 1999, pp. 1456-1467.
3. A.F. Naguib, V. Tarokh, N. Seshadri, and A.R. Calderbank, "Space-time codes for high data rate wireless communication: Mismatch analysis", Proceedings of IEEE International Conf. on Communications, pp. 309-313, June 1997.
4. V. Tarokh, H. Jafarkhami, and A.R. Calderbank, "Space-time block codes for wireless communications: Performance results", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 3, Mar. 1999, pp. 451-460.

Spatial Multiplexing

This example shows spatial multiplexing schemes wherein the data stream is subdivided into independent sub-streams, one for each transmit antenna employed. As a consequence, these schemes provide a multiplexing gain and do not require explicit orthogonalization as needed for space-time block coding.

Spatial multiplexing requires powerful decoding techniques at the receiver though. Of the many proposed [1], this example highlights two ordered Successive Interference Cancellation (SIC) detection schemes. These schemes are similar to the original Bell Labs Layered Space-Time (BLAST) techniques as per [2], [3].

For expositional benefits the example uses the basic 2x2 MIMO system employing two transmit and two receive antennas. For an uncoded QPSK modulated system it employs flat Rayleigh fading over independent transmit-receive links. At the receiver end, we assume perfect channel knowledge with no feedback to the transmitter, i.e., an open-loop spatial multiplexing system.

The example shows two nonlinear interference cancellation methods - Zero-Forcing (ZF) and Minimum-Mean-Square-Error (MMSE) - with symbol cancellation and compares their performance with the Maximum-Likelihood (ML) optimum receiver.

Simulation

We start by defining some common simulation parameters

```
N = 2;                  % Number of transmit antennas
M = 2;                  % Number of receive antennas
EbNoVec = 2:3:8;        % Eb/No in dB
modOrd = 2;             % constellation size = 2^modOrd
```

and set up the simulation.

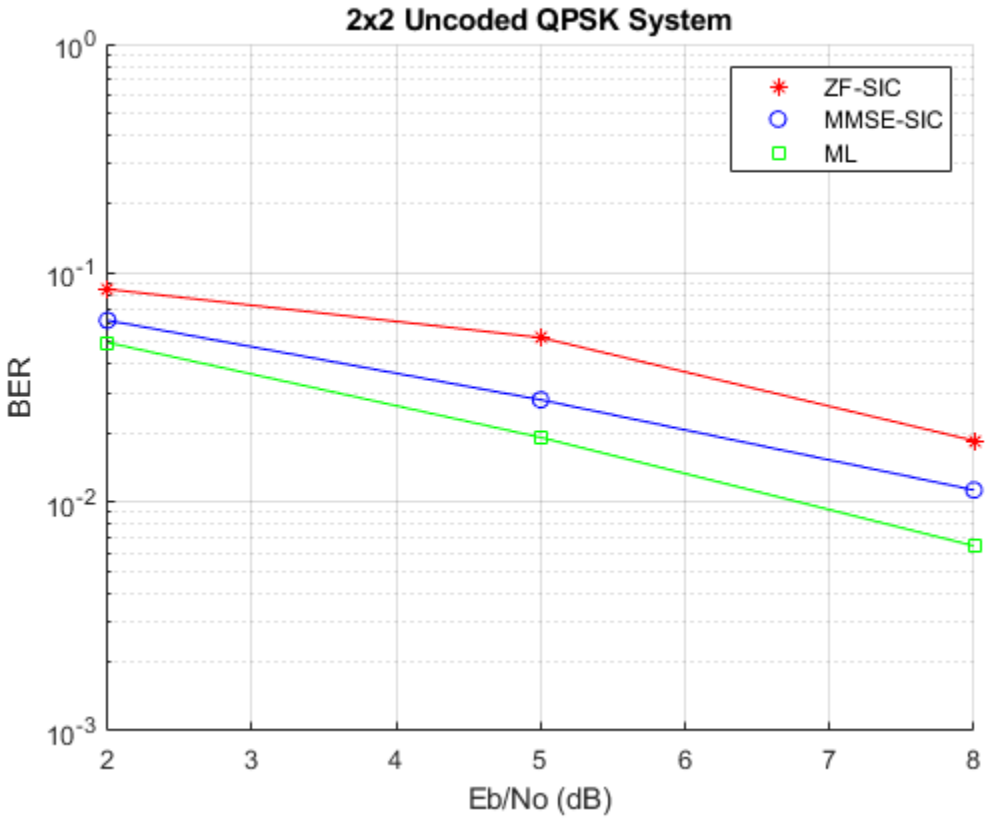
```
% Create a local random stream to be used by random number generators for
% repeatability.
stream = RandStream('mt19937ar');

% Create PSK modulator and demodulator System objects
pskModulator   = comm.PSKModulator(...
            'ModulationOrder',  2^modOrd, ...
            'PhaseOffset',      0, ...
            'BitInput',         true);
pskDemodulator = comm.PSKDemodulator( ...
            'ModulationOrder',  2^modOrd, ...
            'PhaseOffset',      0, ...
            'BitOutput',        true);

% Create error rate calculation System objects for 3 different receivers
zfBERCalc = comm.ErrorRate;
mmseBERCalc = comm.ErrorRate;
mlBERCalc = comm.ErrorRate;

% Get all bit and symbol combinations for ML receiver
allBits = int2bit(0:2^(modOrd*N)-1, modOrd*N);
allTxSig = reshape(pskModulator(allBits(:)), N, 2^(modOrd*N));

% Pre-allocate variables to store BER results for speed
[BER_ZF, BER_MMSE, BER_ML] = deal(zeros(length(EbNoVec), 3));
```

The simulation loop below simultaneously evaluates the BER performance of the three receiver schemes for each Eb/No value using the same data and channel realization. A short range of Eb/No values are used for simulation purposes. Results for a larger range, using the same code, are presented later.

```
% Set up a figure for visualizing BER results
fig = figure;
grid on;
hold on;
ax = fig.CurrentAxes;
ax.YScale = 'log';
xlim([EbNoVec(1)-0.01 EbNoVec(end)]);
ylim([1e-3 1]);
xlabel('Eb/No (dB)');
ylabel('BER');
fig.NumberTitle = 'off';
fig.Renderer = 'zbuffer';
fig.Name = 'Spatial Multiplexing';
title('2x2 Uncoded QPSK System');
set(fig,'DefaultLegendAutoUpdate','off');

% Loop over selected EbNo points
for idx = 1:length(EbNoVec)
    % Reset error rate calculation System objects
    reset(zfBERCalc);
    reset(mmseBERCalc);
    reset(mlBERCalc);

    % Calculate SNR from EbNo for each independent transmission link
    snrIndB = EbNoVec(idx) + 10*log10(modOrd);
    snrLinear = 10^(0.1*snrIndB);

    while (BER_ZF(idx, 3) < 1e5) && ((BER_MMSE(idx, 2) < 100) || ...
          (BER_ZF(idx, 2) < 100) ||  (BER_ML(idx, 2)   < 100))
        % Create random bit vector to modulate
        msg = randi(stream, [0 1], [N*modOrd, 1]);

        % Modulate data
        txSig = pskModulator(msg);

        % Flat Rayleigh fading channel with independent links
        rayleighChan = (randn(stream, M, N) +  1i*randn(stream, M, N))/sqrt(2);

        % Add noise to faded data
        rxSig = awgn(rayleighChan*txSig, snrIndB, 0, stream);

        % ZF-SIC receiver
        r = rxSig;
        H = rayleighChan; % Assume perfect channel estimation
        % Initialization
        estZF = zeros(N*modOrd, 1);
        orderVec = 1:N;
        k = N+1;
        % Start ZF nulling loop
        for n = 1:N
            % Shrink H to remove the effect of the last decoded symbol
            H = H(:, [1:k-1,k+1:end]);
```

```
            % Shrink order vector correspondingly
            orderVec = orderVec(1, [1:k-1,k+1:end]);
            % Select the next symbol to be decoded
            G = (H'*H) \ eye(N-n+1); % Same as inv(H'*H), but faster
            [~, k] = min(diag(G));
            symNum = orderVec(k);

            % Hard decode the selected symbol
            decBits = pskDemodulator(G(k,:) * H' * r);
            estZF(modOrd * (symNum-1) + (1:modOrd)) = decBits;

            % Subtract the effect of the last decoded symbol from r
            if n < N
                r = r - H(:, k) * pskModulator(decBits);
            end
        end

        % MMSE-SIC receiver
        r = rxSig;
        H = rayleighChan;
        % Initialization
        estMMSE = zeros(N*modOrd, 1);
        orderVec = 1:N;
        k = N+1;
        % Start MMSE nulling loop
        for n = 1:N
            H = H(:, [1:k-1,k+1:end]);
            orderVec = orderVec(1, [1:k-1,k+1:end]);
            % Order algorithm (matrix G calculation) is the only difference
            % with the ZF-SIC receiver
            G = (H'*H + ((N-n+1)/snrLinear)*eye(N-n+1)) \ eye(N-n+1);
            [~, k] = min(diag(G));
            symNum = orderVec(k);

            decBits = pskDemodulator(G(k,:) * H' * r);
            estMMSE(modOrd * (symNum-1) + (1:modOrd)) = decBits;

            if n < N
                r = r - H(:, k) * pskModulator(decBits);
            end
        end

        % ML receiver
        r = rxSig;
        H = rayleighChan;
        [~, k] = min(sum(abs(repmat(r,[1,2^(modOrd*N)]) - H*allTxSig).^2));
        estML = allBits(:,k);

        % Update BER
        BER_ZF(  idx, :) = zfBERCalc(msg, estZF);
        BER_MMSE(idx, :) = mmseBERCalc(msg, estMMSE);
        BER_ML(  idx, :) = mlBERCalc(msg, estML);
    end

    % Plot results
    semilogy(EbNoVec(1:idx), BER_ZF(  1:idx, 1), 'r*', ...
             EbNoVec(1:idx), BER_MMSE(1:idx, 1), 'bo', ...
             EbNoVec(1:idx), BER_ML(  1:idx, 1), 'gs');
```

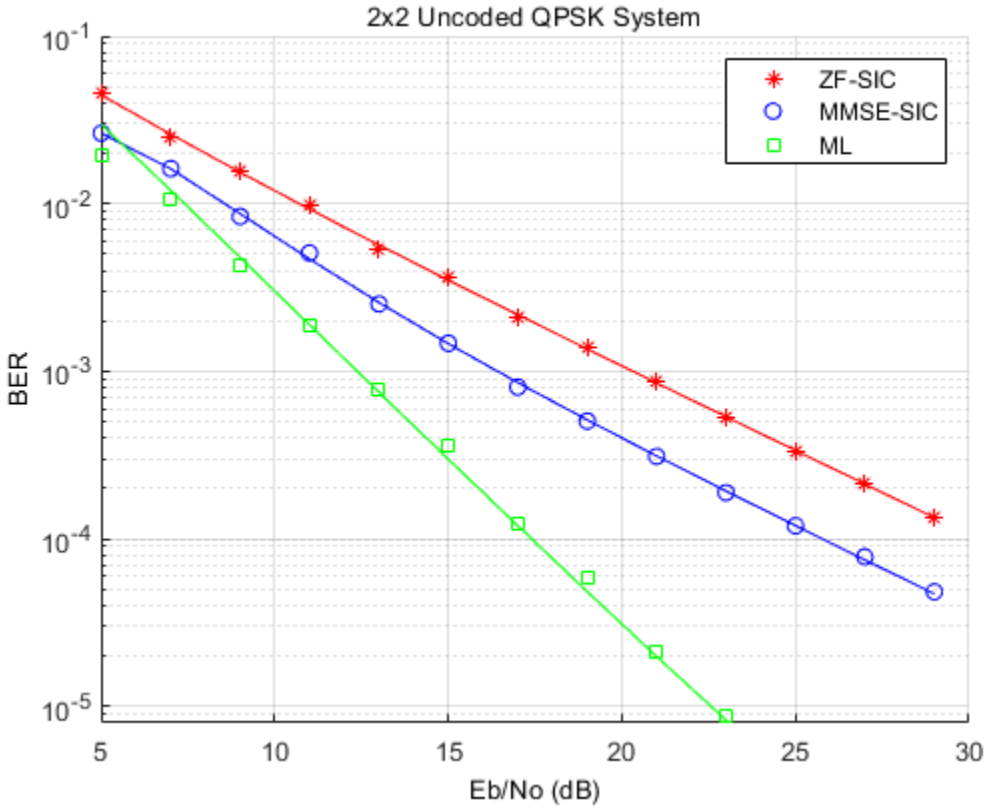
```
    legend('ZF-SIC', 'MMSE-SIC', 'ML');
    drawnow;
end

% Draw the lines
semilogy(EbNoVec, BER_ZF(  :, 1), 'r-', ...
         EbNoVec, BER_MMSE(:, 1), 'b-', ...
         EbNoVec, BER_ML(  :, 1), 'g-');
hold off;
```



We observe that the ML receiver is the best in performance followed by the MMSE-SIC and ZF-SIC receivers, as also seen in [4]. In terms of receiver complexity, ML grows exponentially with the number of transmit antennas while the ZF-SIC and MMSE-SIC are linear receivers combined with successive interference cancellation. Optimized ZF-SIC and MMSE-SIC algorithms for reduced complexity can be found in [5].

Simulation results comparing the three schemes for a larger range of Eb/No values are displayed next. These curves allow you to gauge the diversity order attained from the slope of the BER curve.

```
openfig('spatMuxResults.fig');
```



Some areas of further exploration would be to try these methods for a larger number of antennas, with and without channel estimation.

Selected References

1. George Tsoulos, Ed., "MIMO System Technology for Wireless Communications", CRC Press, Boca Raton, FL, 2006.
2. G. J. Foschini, "Layered space-time architecture for wireless communication in a fading environment when using multiple antennas," The Bell Sys. Tech. Journal, 1996, No. 1, pp. 41-59.
3. P. W. Wolniansky, G. J. Foschini, G. D. Golden, R. A. Valenzuela, "V-BLAST: An Architecture for realizing very high data rates over the rich scattering wireless channel," 1998 URSI International Symposium on Signals, Systems, and Electronics, 29 Sep.-2 Oct. 1998, pp. 295-300.
4. X. Li, H. C. Huang, A. Lozano, G. J. Foschini, "Reduced-complexity detection algorithms for systems using multi-element arrays", IEEE® Global Telecommunications Conference, 2000. Volume 2, 27 Nov.-1 Dec. 2000, pp. 1072-76.
5. Y. Shang and X.-G. Xia, "On fast recursive algorithms for V-BLAST with optimal ordered SIC detection," IEEE Trans. Wireless Communications, vol. 8, no. 6, pp. 2860-2865, Jun. 2009.

OSTBC Transmission with Antenna Coupling

This example shows how the antenna mutual coupling affects the performance of an orthogonal space-time block code (OSTBC) transmission over a multiple-input multiple-output (MIMO) channel. The transmitter and receiver have two dipole antenna elements each. The BER vs. SNR curves are plotted under different correlation and coupling scenarios. To run this example, you need Antenna Toolbox™.

System Parameters

A QPSK modulated Alamouti OSTBC is simulated over a 2x2 quasi-static frequency-flat Rayleigh channel [1 on page 8-0]. The system operates at 2.4 GHz. The SNR range to be simulated is 0 to 10 dB.

```
fc = 2.4e9;         % Center frequency
Nt = 2;             % Number of Tx antennas
Nr = 2;             % Number of Rx antennas
blkLen = 2;         % Alamouti code block length
snr = 0:10;         % SNR range
maxNumErrs = 3e2;   % Maximum number of errors
maxNumBits = 5e4;   % Maximum number of bits
```

Create objects to perform QPSK modulation and demodulation, Alamouti encoding and combining, AWGN channel as well as BER calculation.

```
qpskMod = comm.QPSKModulator;
qpskDemod = comm.QPSKDemodulator;
alamoutiEnc = comm.OSTBCEncoder( ...
    'NumTransmitAntennas', Nt);
alamoutiDec = comm.OSTBCCombiner( ...
    'NumTransmitAntennas', Nt, ...
    'NumReceiveAntennas',  Nr);
awgnChanNC = comm.AWGNChannel( ... % For no coupling case
    'NoiseMethod', 'Signal to noise ratio (SNR)',...
    'SignalPower', 1);
berCalcNC = comm.ErrorRate;       % For no coupling case

% Clone objects for mutual coupling case
awgnChanMC = clone(awgnChanNC);
berCalcMC  = clone(berCalcNC);
```

Antenna Arrays and Coupling Matrices

A two-element resonant dipole array is used at both transmit (Tx) and receive (Rx) side. At Tx, the dipoles are spaced a half-wavelength apart. At Rx, the spacing is a tenth of a wavelength.

```
txSpacing = 0.5;
rxSpacing = 0.1;
lambda = physconst('lightspeed')/fc;
antElement = dipole( ...
    'Length', lambda/2, ...
    'Width',  lambda/100);
txArray = linearArray( ...
    'Element',        antElement,...
    'NumElements',    Nt,...
    'ElementSpacing', txSpacing*lambda);
rxArray = linearArray( ...
```

```
    'Element',        antElement,...
    'NumElements',    Nr,...
    'ElementSpacing', rxSpacing*lambda);
```

The coupling matrix is calculated based on a circuit model of the array as per [2 on page 8-0]. The s-parameter calculation is performed for the transmit and receive arrays and from this the impedance matrix representation of the array is derived.

```
txMCMtx = helperCalculateCouplingMatrix(txArray, fc, [1 Nt]);
rxMCMtx = helperCalculateCouplingMatrix(rxArray, fc, [1 Nr]);
```

Spatial Correlation Matrices

The transmit and receive spatial correlation matrices capture the propagation environment of the channel. Without coupling, it is assumed that the two elements at Tx are uncorrelated and the two elements at Rx have high correlation. The combined/overall correlation matrix for the whole channel is their Kronecker product.

```
txCorrMtx = eye(2);
rxCorrMtx = [1 0.9; 0.9 1];
combCorrMtx = kron(txCorrMtx, rxCorrMtx);
```

With coupling, we use the approach in [3 on page 8-0] to modify the Tx and Rx correlation matrices by pre and post-multiplying them by the corresponding coupling matrices. This is valid under the assumption that the correlation and coupling can be modeled independently.

```
txMCCorrMtx = txMCMtx * txCorrMtx * txMCMtx';
rxMCCorrMtx = rxMCMtx * rxCorrMtx * rxMCMtx';
```

The combined spatial correlation with coupling is `kron(txMCCorr, rxMCCorr)`. Alternatively, we can treat the Tx/Rx coupling matrix as being "absorbed" into the Tx/Rx correlation matrix and derive the combined correlation matrix as follows:

```
txSqrtCorrMtx = txMCMtx * sqrtm(txCorrMtx);
rxSqrtCorrMtx = rxMCMtx * sqrtm(rxCorrMtx);
combMCCorrMtx = kron(txSqrtCorrMtx, rxSqrtCorrMtx);
combMCCorrMtx = combMCCorrMtx * combMCCorrMtx';
```

MIMO Channel Modeling

Create two `comm.MIMOChannel` objects to simulate the 2x2 MIMO channels with and without coupling. The combined spatial correlation matrix is assigned in each case. The `MaximumDopplerShift` property of the objects is set to 0 to model a quasi-static channel.

```
mimoChanNC = comm.MIMOChannel( ...  % For no coupling case
    'MaximumDopplerShift',             0, ...
    'SpatialCorrelationSpecification', 'Combined', ...
    'SpatialCorrelationMatrix',        combCorrMtx,...
    'PathGainsOutputPort',             true);

% Clone objects for mutual coupling case
mimoChanMC = clone(mimoChanNC);
mimoChanMC.SpatialCorrelationMatrix = combMCCorrMtx;
```

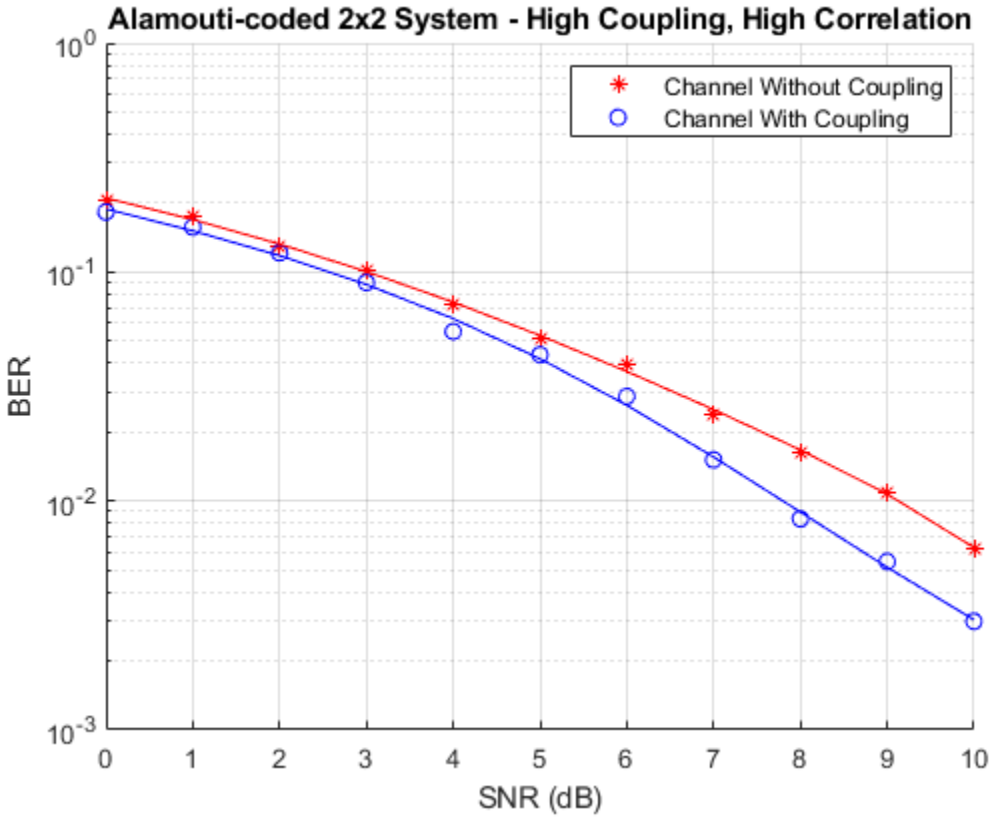
Simulations

Simulate the QPSK modulated Alamouti code for each SNR value with and without antenna coupling. One Alamouti code is simulated through the MIMO channel in each iteration. To model a quasi-static

channel, we reset the `comm.MIMOChannel` object to obtain a new set of channel gains for each code transmission (iteration).

```
% Set up a figure to visualize BER results
h1 = figure; grid on; hold on;
ax = gca;
ax.YScale = 'log';
xlim([snr(1), snr(end)]); ylim([1e-3 1]);
xlabel('SNR (dB)'); ylabel('BER');
h1.NumberTitle = 'off';
h1.Name = 'Orthogonal Space-Time Block Coding';
h1.Renderer = 'zbuffer';
title('Alamouti-coded 2x2 System - High Coupling, High Correlation');

s = rng(108);  % For repeatability
[berNC, berMC] = deal(zeros(3,length(snr)));

% Loop over SNR values
for idx = 1:length(snr)
    awgnChanNC.SNR = snr(idx);
    awgnChanMC.SNR = snr(idx);
    reset(berCalcNC);
    reset(berCalcMC);

    while min(berNC(2,idx),berMC(2,idx)) <= maxNumErrs && (berNC(3,idx) <= maxNumBits)
        % Generate random data
        txData = randi([0 3], blkLen, 1);

        % Perform QPSK modulation and Alamouti encoding
        txSig = alamoutiEnc(qpskMod(txData));

        % Pass through MIMO channel
        reset(mimoChanNC); reset(mimoChanMC);
        [chanOutNC, estChanNC] = mimoChanNC(txSig);
        [chanOutMC, estChanMC] = mimoChanMC(txSig);

        % Add AWGN
        rxSigNC = awgnChanNC(chanOutNC);
        rxSigMC = awgnChanMC(chanOutMC);

        % Perform Alamouti decoding with known channel state information
        decSigNC = alamoutiDec(rxSigNC, squeeze(estChanNC));
        decSigMC = alamoutiDec(rxSigMC, squeeze(estChanMC));

        % Perform QPSK demodulation
        rxDataNC = qpskDemod(decSigNC);
        rxDataMC = qpskDemod(decSigMC);

        % Update BER
        berNC(:, idx) = berCalcNC(txData, rxDataNC);
        berMC(:, idx) = berCalcMC(txData, rxDataMC);
    end

    % Plot results
    semilogy(snr(1:idx), berNC(1,1:idx), 'r*');
    semilogy(snr(1:idx), berMC(1,1:idx), 'bo');
    legend({'Channel Without Coupling', 'Channel With Coupling'});
    drawnow;
```

```
end

% Perform curve fitting
fitBERNC = berfit(snr, berNC(1,:));
fitBERMC = berfit(snr, berMC(1,:));
semilogy(snr, fitBERNC, 'r', snr, fitBERMC, 'b');
legend({'Channel Without Coupling', 'Channel With Coupling'});
```

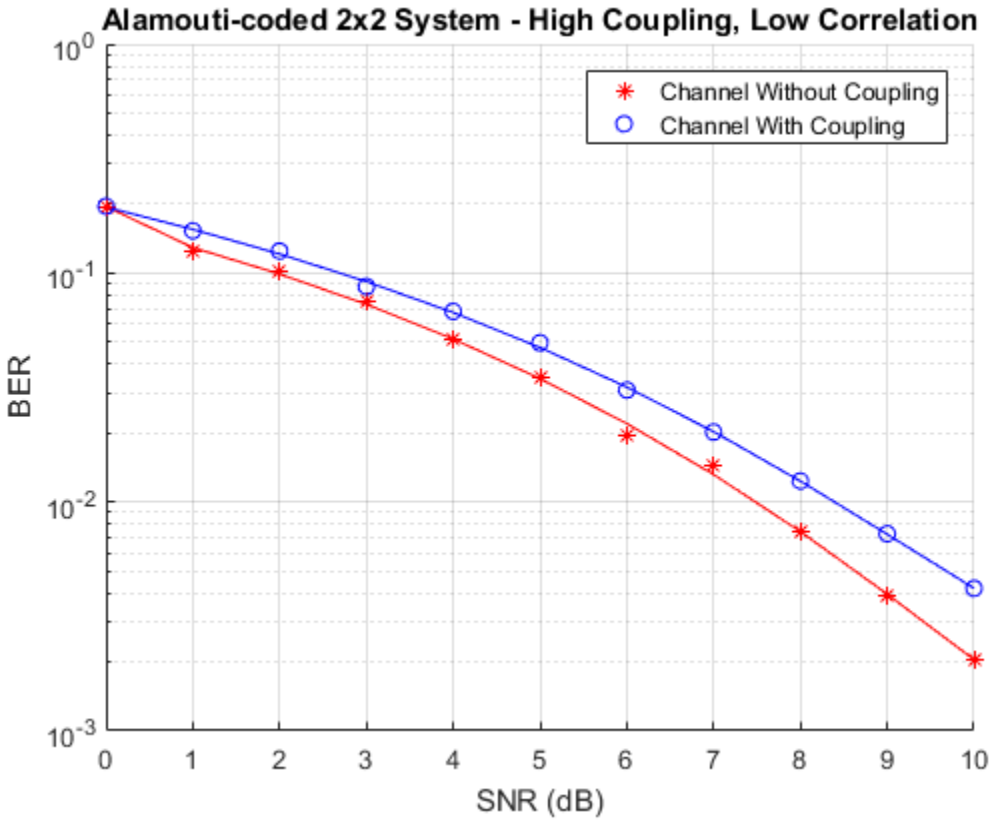



```
rng(s); % Restore RNG
```

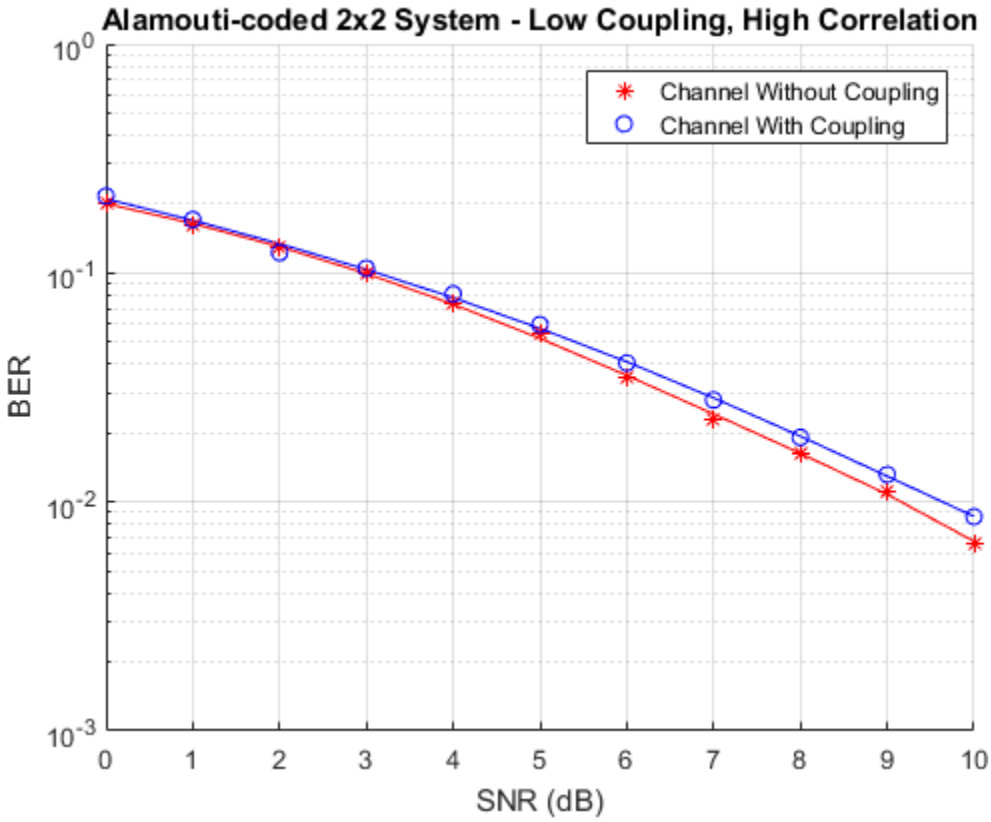
Further Exploration

The effect of correlation and mutual coupling on the BER performance can be further studied by modifying the correlation coefficient and/or by changing the spacing between the elements. The smaller the spacing is, the higher the coupling is. Similar to what has been done above for high correlation (0.9) and high coupling (spacing = 0.1λ) at Rx, we now show the BER vs. SNR results for low correlation (0.1) and/or low coupling (spacing = 0.5λ).

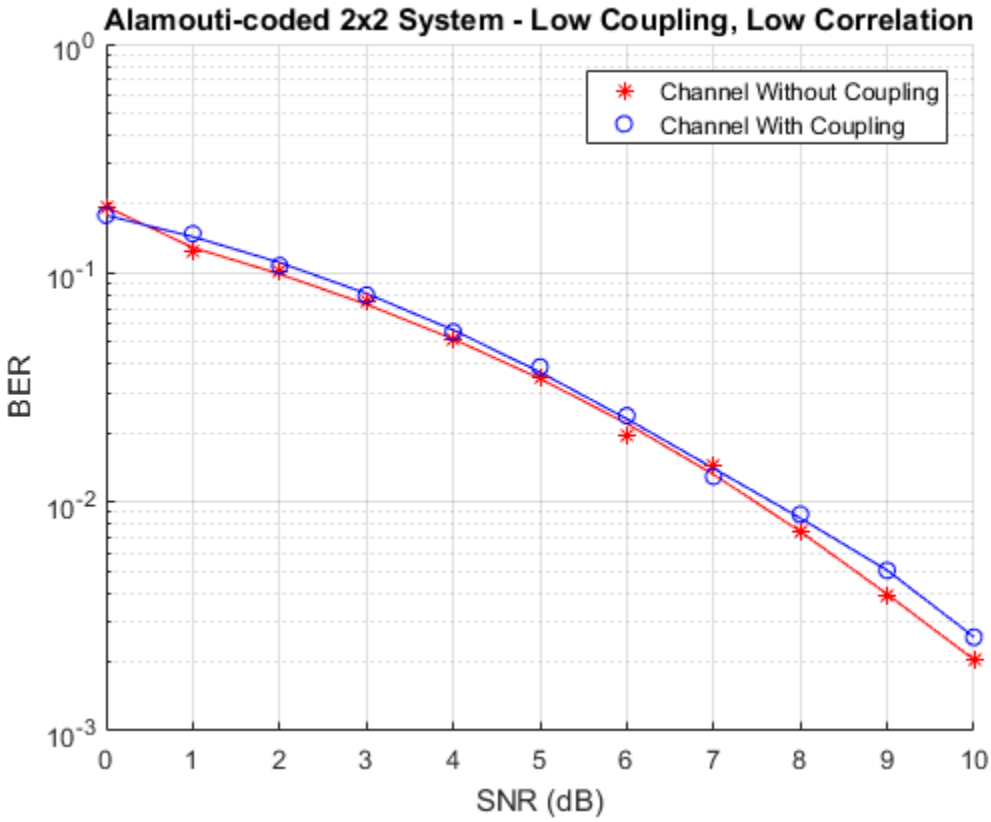
- High Coupling (spacing = 0.1λ), Low Correlation (0.1)



- Low Coupling (spacing = 0.5λ), High Correlation (0.9)



- Low Coupling (spacing = 0.5λ), Low Correlation (0.1)



Conclusion

The simulation results are similar to those reported in [1 on page 8-0]. A spacing of 0.5λ has a negligible impact on BER under both high and low correlation conditions. For the case with high coupling, i.e., 0.1λ element spacing, the results indicate that depending on the correlation conditions, the BER could be either higher or lower than if coupling were not considered.

Appendix

This example uses the following helper functions:

- helperCalculateCouplingMatrix.m

References

1 - A. A. Abouda, H. M. El-Sallabi, and S. G. Haggman, "Effect of Mutual Coupling on BER Performance of Alamouti Scheme," *IEEE International Symposium on Antennas and Propagation*, July 2006.

2 - I. J. Gupta and A. A. Ksienski, "Effect of mutual coupling on the performance of adaptive arrays," *IEEE Trans. on Antennas and Propagation*, vol. 31, no. 5, pp. 785-791, 1989.

3 - Y. Wu, J. P. Linnartz, J. W. M. Bergmans, and S. Attallah, "Effects of Antenna Mutual Coupling on the Performance of MIMO Systems," *Proc. 29th Symposium on Information Theory in the Benelux*, May 2008.

Concatenated OSTBC with TCM

This example shows an orthogonal space-time block code (OSTBC) concatenated with trellis-coded modulation (TCM) for information transmission over a multiple-input multiple-output (MIMO) channel with 2 transmit antennas and 1 receive antenna. The example uses communications System objects™ to simulate this system.

Introduction

OSTBCs [1], [2] are an attractive technique for MIMO wireless communications. They exploit full spatial diversity order and enjoy symbol-wise maximum likelihood (ML) decoding. However, they offer no coding gain. The combiner for OSTBC at the receiver side provides soft information of the transmitted symbols, which can be utilized for decoding or demodulation of an outer code.

TCM [3] is a bandwidth efficient scheme that integrates coding and modulation to provide a large coding gain. Concatenating TCM with an inner code will usually offer an improved performance.

This example illustrates the advantages of an OSTBC and TCM concatenation scheme: the spatial diversity gain offered by OSTBC and the coding gain offered by TCM. For comparison, two reference systems containing only TCM or OSTBC are also provided. The diversity and coding gains of the concatenation scheme over the reference models can be clearly observed from the simulation results. More discussions about concatenating OSTBC and TCM can be found in, for example, [4], [5] and references therein.

The configureTCMOSTBCDemo.m script creates System objects used to simulate the concatenated OSTBC system. It also initializes some simulation parameters.

```
% Trellis structure of the TCM modulator
trellis = poly2trellis([2, 3], [1, 2, 0; 4, 1, 2]);

% Create System objects of the concatenated OSTBC system and set simulation
% parameters such as SNR and frame length.
configureTCMOSTBCDemo
```

PSK TCM Modulator and Demodulator

The PSK TCM modulator System object modulates the random message data to a PSK constellation that has unit average energy. The TrellisStructure property accepts a MATLAB® structure to specify the trellis of the TCM. The ModulationOrder property specifies the size of the PSK constellation. This example uses the Ungerboeck TCM scheme for 8-PSK constellation with 8 trellis states [3], and sets the corresponding TrellisStructure property to the result of poly2trellis([2 3], [1 2 0; 4 1 2]). This object has an output length of 50, as every two input bits produce one symbol.

The PSK TCM demodulator System object uses the Viterbi algorithm for TCM to decode the signals from the OSTBC combiner. The example sets the TerminationMethod property to 'Truncated'; therefore treats each frame independently. The example also sets the TracebackDepth property to 30, which compared to the constraint length of the TCM, is long enough to ensure an almost lossless performance.

```
psktcmMod = comm.PSKTCMModulator(trellis, ...
                'TerminationMethod', 'Truncated');

psktcmDemod = comm.PSKTCMDemodulator(trellis, ...
                'TerminationMethod', 'Truncated', ...
                'TracebackDepth', 30, ...
                'OutputDataType', 'logical');
```

Orthogonal Space-Time Block Codes (OSTBC)

The OSTBC encoder System object encodes the information symbols from the TCM Encoder by using the Alamouti code [1] for 2 transmit antennas. The output of this object is a 50x2 matrix, where entries on each column correspond to the data transmitted from one antenna.

The OSTBC combiner System object uses a single antenna and decodes the received signal utilizing the channel state information (CSI). The output of the step method of this object represents the estimates of the transmitted symbols, which are then fed into the PSK TCM demodulator. In this example, the CSI is assumed perfectly known at the receiver side.

```
ostbcEnc = comm.OSTBCEncoder;
ostbcComb = comm.OSTBCCombiner;
```

2x1 MIMO Fading Channel

The 2x1 MIMO fading channel System object simulates the spatially independent flat Rayleigh fading channel from the 2 transmit antennas to the 1 receive antenna.

The example sets the maximumDopplerShift property of the channel object to 30. The reason for using this value is to make the MIMO channel behave like a quasi-static fading channel, i.e., it keeps constant during one frame transmission and varies along multiple frames. The example sets the PathGainsOutputPort property to true to use the channel path gain values as perfect estimates of CSI. The example also sets the RandomStream property to 'mt19937ar with seed' so that the object uses a self-contained random number generator to generate repeatable channel coefficients. The 2x1 MIMO channel has normalized path gains.

```
mimoChan = comm.MIMOChannel(...
                'SampleRate', 1/Tsamp, ...
                'MaximumDopplerShift', maxDopp, ...
                'SpatialCorrelationSpecification', 'None', ...
                'NumReceiveAntennas', 1, ...
                'RandomStream', 'mt19937ar with seed', ...
                'PathGainsOutputPort', true);
```

Concatenated OSTBC with TCM

This section of the code calls the processing loop for a concatenated OSTBC system. The main loop processes the data frame-by-frame, where the transmitter modulates the random data using an 8-PSK TCM modulator and then applies Alamouti coding. The two transmitted signals from the OSTBC encoder pass through the 2x1 MIMO Rayleigh fading channel and are also impaired by AWGN. The OSTBC combiner uses one receive antenna and provides soft inputs to the 8-PSK TCM demodulator. The example compares the output of the demodulator with the generated random data to obtain frame error rate (FER).

Stream Processing

```
fer = zeros(3,1);
while (fer(3) < maxNumFrms) && (fer(2) < maxNumErrs)
  data      = logical(randi([0 1], frameLen, 1)); % Generate data
  modData   = psktcmMod(data);                    % Modulate
  txSignal  = ostbcEnc(modData);                  % Apply Alamouti coding
  [chanOut, chanEst] = mimoChan(txSignal);        % 2x1 fading channel
  rxSignal  = awgnChan(chanOut);                  % Add receiver noise
  modDataRx = ostbcComb(rxSignal, ...
                  squeeze(chanEst));              % Decode
```

```
  dataRx    = psktcmDemod(modDataRx);             % Demodulate
  frameErr  = any(dataRx - data);                 % Check frame error
  fer       = FERData(false, frameErr);           % Update frame error rate
end
```

The step method of the error rate measurement System object, FERData, outputs a 3-by-1 vector containing updates of the measured FER value, the number of errors, and the total number of frame transmissions. Display FER values.

```
frameErrorRate = fer(1)
```

```
frameErrorRate =

    0.1481
```

TCM over Flat Fading Channel

This section of the example simulates the TCM in the previous concatenation scheme over a single-input single-output (SISO) flat Rayleigh fading channel, without space-time coding. The fading channel has the same specification as one subchannel of the 2x1 MIMO fading channel in the previous system. So this section of the example sets the NumTransmitAntennas property of the fading channel System object to 1 after releasing it. This section of the example also sets the SignalPower property of the AWGN channel System object to 1, as there is only one symbol transmitted per symbol period.

Initialize the processing loop

```
release(mimoChan);
mimoChan.NumTransmitAntennas = 1;
awgnChan.SignalPower = 1;
reset(FERData)
fer = zeros(3,1);
```

Stream Processing Loop

```
while (fer(3) < maxNumFrms) && (fer(2) < maxNumErrs)
  data      = logical(randi([0 1], frameLen, 1)); % Generate data
  modData   = psktcmMod(data);                    % Modulate
  [chanOut, chanEst] = mimoChan(modData);         % SISO fading channel
  rxSignal  = awgnChan(chanOut);                  % Add receiver noise
  modDataRx = (rxSignal.*conj(chanEst)) / ...
              (chanEst'*chanEst);                 % Equalize
  dataRx    = psktcmDemod(modDataRx);             % Demodulate
  frameErr  = any(dataRx - data);                 % Check frame error
  fer       = FERData(false, frameErr);           % Update frame error rate
end
```

OSTBC over 2x1 Flat Rayleigh Fading Channel

This section of the example replaces the TCM in the previous concatenation scheme by a QPSK modulation so that both systems have the same symbol (frame) rate. It uses the same 2x1 flat Rayleigh fading channel as in the TCM-OSTBC concatenation model. The QPSK modulator System object, qpskMod, maps the information bits to a QPSK constellation and the QPSK demodulator System object, QPSKDemod, demodulates the signals from the OSTBC Combiner.

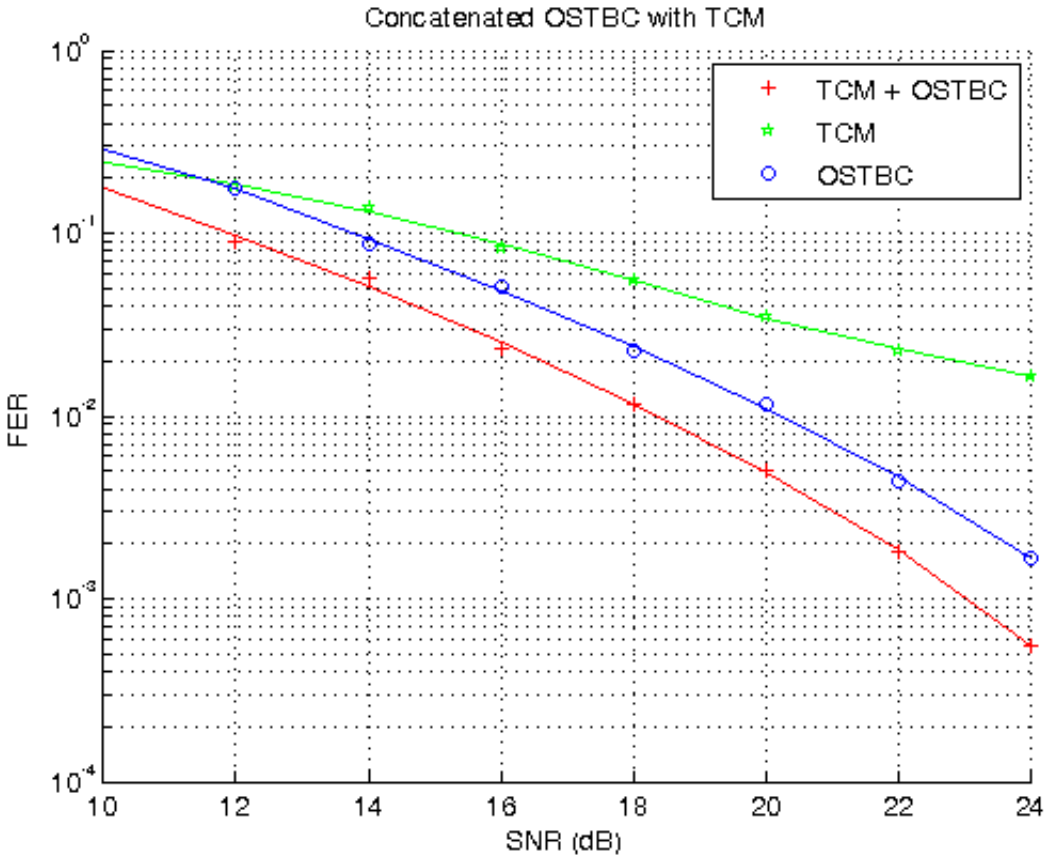
Initialize the processing loop

```
release(mimoChan);
mimoChan.NumTransmitAntennas = 2;
awgnChan.SignalPower = 2;
reset(FERData)
fer = zeros(3,1);
```

Stream Processing Loop

```
while (fer(3) < maxNumFrms) && (fer(2) < maxNumErrs)
  data      = logical(randi([0 1], frameLen, 1)); % Generate data
  modData   = qpskMod(data);                      % Modulate
  txSignal  = ostbcEnc(modData);                  % Apply Alamouti coding
  [chanOut, chanEst] = mimoChan(txSignal);        % 2x1 fading channel
  rxSignal  = awgnChan(chanOut);                  % Add receiver noise
  modDataRx = ostbcComb(rxSignal, ...
                  squeeze(chanEst));              % Decode
  dataRx    = qpskDemod(modDataRx);               % Demodulate
  frameErr  = any(dataRx - data);                 % Check frame error
  fer       = FERData(false, frameErr);           % Update frame error rate
end
```

You can add a for-loop around the previous processing loops to run simulations for a set of SNR values. Simulations were run offline for SNR values of (10:2:24) dB, target number of errors equal to 1000, and maximum number of transmissions equal to 5e6. The following figure shows the results from this simulation.



Summary

This example utilized several System objects to simulate a concatenated OSTBC with TCM over a 2x1 flat Rayleigh fading channel. This base system was modified to model a TCM system over a SISO flat fading channel and an OSTBC system over the same 2x1 flat Rayleigh fading channel. System performance was measured using the FER curves obtained with the error rate measurement System object. This example showed that the concatenation scheme provides a significant diversity gain over the TCM scheme and about 2dB coding gain over the Alamouti code.

Appendix

This example uses the following script and helper function:

- configureTCMOSTBCDemo.m

Selected Bibliography

1 S. M. Alamouti, "A simple transmit diversity technique for wireless communications," *IEEE® Journal on Selected Areas in Communications*, vol. 16, no. 8, pp. 1451-1458, Oct. 1998.

2 V. Tarokh, H. Jafarkhami, and A. R. Calderbank, "Space-time block codes from orthogonal designs," *IEEE Transactions on Information Theory*, vol. 45, no. 5, pp. 1456-1467, Jul. 1999.

3 G. Ungerboeck, "Channel coding with multilevel/phase signals," *IEEE Transactions on Information Theory*, vol. IT-28, no. 1, pp. 55?67, Jan. 1982.

4 S. M. Alamouti, V. Tarokh, and P. Poon, "Trellis-coded modulation and transmit diversity: Design criteria and performance evaluation," in *Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC'98)*, Florence, Italy, vol. 1, Oct. 5-9, 1998, pp. 703-707.

5 Y. Gong and K. B. Letaief, "Concatenated space-time block coding with trellis coded modulation in fading channels," *IEEE Transactions on Wireless Communications*, vol. 1, no. 4, pp. 580-590, Oct. 2002.

Concatenated OSTBC with TCM in Simulink

This model shows an orthogonal space-time block code (OSTBC) concatenated with trellis-coded modulation (TCM) for information transmission over a multiple-input multiple-output (MIMO) channel with 2 transmit antennas and 1 receive antenna.

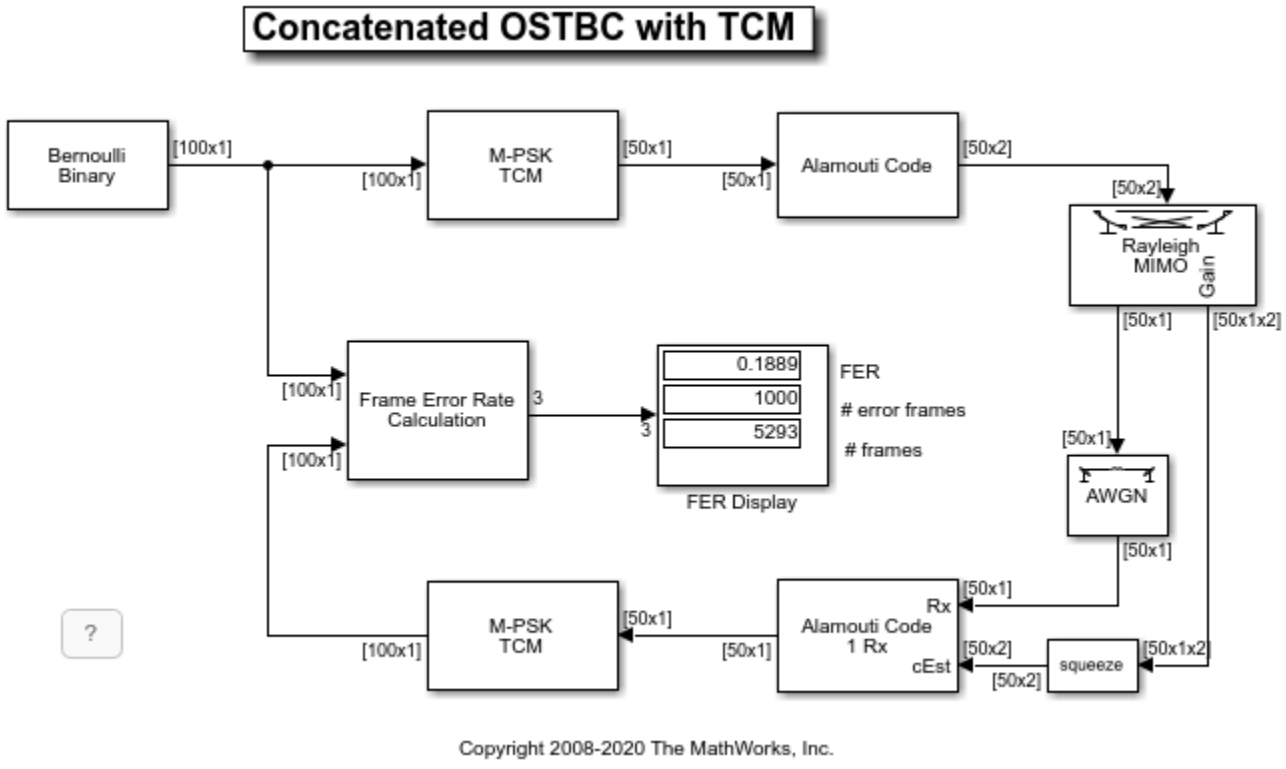
Introduction

OSTBCs [1], [2] are an attractive technique for MIMO wireless communications. They exploit full spatial diversity order and enjoy symbol-wise maximum likelihood (ML) decoding. However, they offer no coding gain. The combiner for OSTBC at the receiver side provides soft information of the transmitted symbols, which can be utilized for decoding or demodulation of an outer code.

TCM [3] is a bandwidth efficient scheme that integrates coding and modulation to provide a large coding gain. Concatenating TCM with an inner code will usually offer an improved performance.

This example illustrates the advantages of an OSTBC and TCM concatenation scheme: the spatial diversity gain offered by OSTBC and the coding gain offered by TCM. For comparison, two reference models containing only TCM or OSTBC are also provided. The diversity and coding gains of the concatenation scheme over the reference models can be clearly observed from the simulation results. More discussions about concatenating OSTBC and TCM can be found in, for example, [4], [5] and references therein.

Structure of the Example



The individual tasks performed by the model include:

Random Data Generation

The Bernoulli Binary Generator block produces the information source for this simulation. The block generates a frame of 100 random bits. The `Samples per frame` parameter determines the length of the output frame (100 in this case).

Trellis-Coded Modulation (TCM)

The M-PSK TCM Encoder block modulates the message data from the Bernoulli Binary Generator to a PSK constellation that has unit average energy. The `Trellis structure` parameter accepts a MATLAB® structure to specify the trellis of the TCM. The `M-ary number` parameter specifies the size of the PSK constellation. In this example, we use the Ungerboeck TCM scheme for 8-PSK constellation with 8 trellis states [3]. Correspondingly, the `Trellis structure` parameter is set to `poly2trellis([2 3], [1 2 0; 4 1 2])`. This block has an output frame length of 50 as every two input bits produce one symbol.

The M-PSK TCM Decoder block uses the Viterbi algorithm for TCM to decode the signals from the OSTBC Combiner. The `Operation mode` parameter is set to `Truncated` to treat each frame independently. The `Traceback depth` parameter is set to `30` that, compared with the constraint length of the TCM, is long enough to ensure an almost lossless performance.

Orthogonal Space-Time Block Codes (OSTBC)

The OSTBC Encoder block encodes the information symbols from the TCM Encoder by using the Alamouti code [1] for 2 transmit antennas. The output of this block is a 50x2 matrix whose entries on each column correspond to the data transmitted over one antenna.

The OSTBC Combiner block combines the received signals from the receive antenna with the channel state information (CSI) to output the estimates of the transmitted symbols, which are then fed into the M-PSK TCM Decoder. In this example, the CSI is assumed perfectly known at the receiver side.

2x1 MIMO Channel

The MIMO Fading Channel block simulates a 2x1 frequency-flat Rayleigh fading channel. The `Sample rate (Hz)` parameter is set to 500000 that is calculated based on the input signal length and model sample time. The `Maximum Doppler shift (Hz)` parameter is set to `30`. The reason for using this value is to make the MIMO channel behave like a quasi-static fading channel, i.e., it keeps constant during one frame transmission and varies along multiple frames.

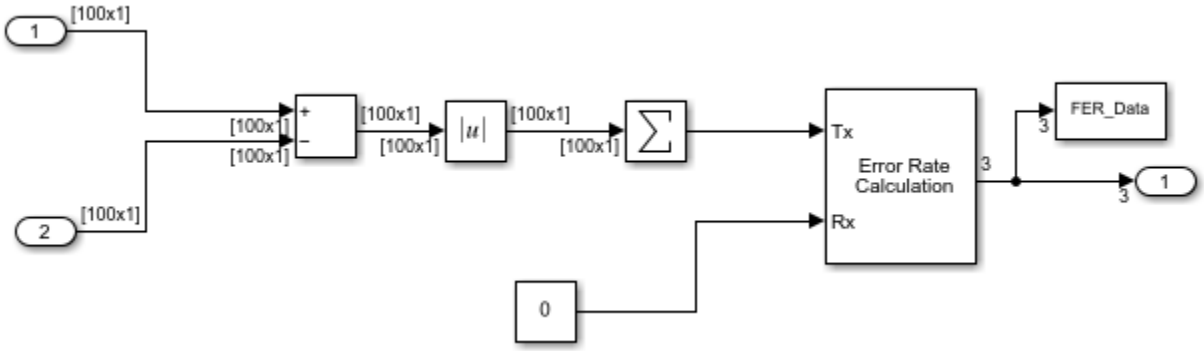
Receiver Noise

The AWGN Channel block adds white Gaussian noises at the receiver side. The `Mode` parameter is set to `Signal to noise ratio (SNR)` mode and the `Input signal power, referenced to 1 ohm (watts)` parameter is set to `2` because the PSK constellation for TCM has unit average energy and the path gains of the MIMO channel are normalized.

Frame Error Rate (FER) Calculation

The Frame Error Rate (FER) Calculation subsystem compares the decoded bits with the original source bits per frame to detect errors and dynamically updates the FER along the simulation. The output of this subsystem is a three-element vector containing the FER, the number of error frames observed and the number of frames processed. This vector is from the Error Rate Calculation block and also saved as a MATLAB® workspace variable `FER_Data` to ease the simulation for multiple SNR values described below.

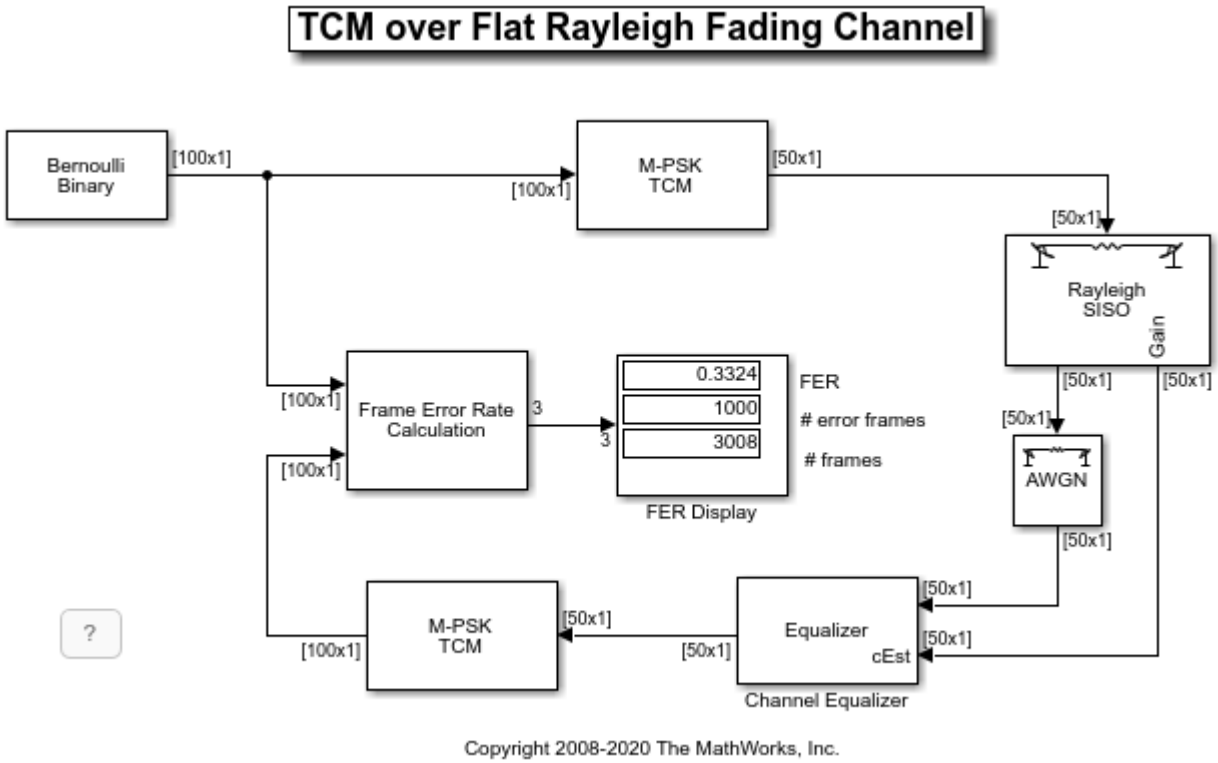
The `Stop simulation` parameter is checked to control the duration of the simulation. The simulation stops upon detecting a target number of error frames (specified by the `Target number of errors` parameter) or a maximum number of frames (specified by the `Maximum number of symbols` parameter), whichever comes first.



We now briefly describe the two reference models used for comparison.

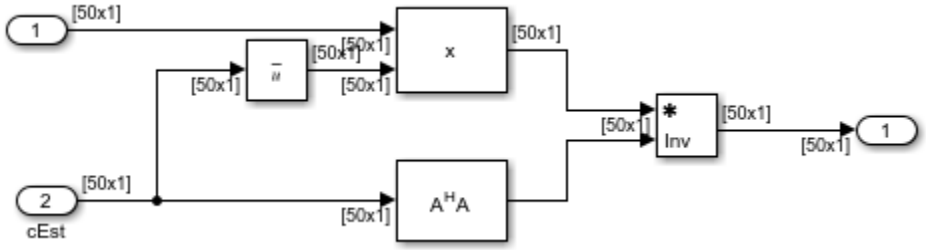
TCM over Flat Rayleigh Fading Channel

The model commtcm.slx simulates the TCM in the above concatenation scheme over a single-input single-output (SISO) flat Rayleigh fading channel. No space-time coding is used. The SISO Fading Channel block has the same specification as one subchannel of the 2x1 MIMO channel in the above model. The `Input signal power, referenced to 1 ohm (watts)` parameter of the AWGN Channel block is set to `1` as there is only one symbol transmitted per symbol period.



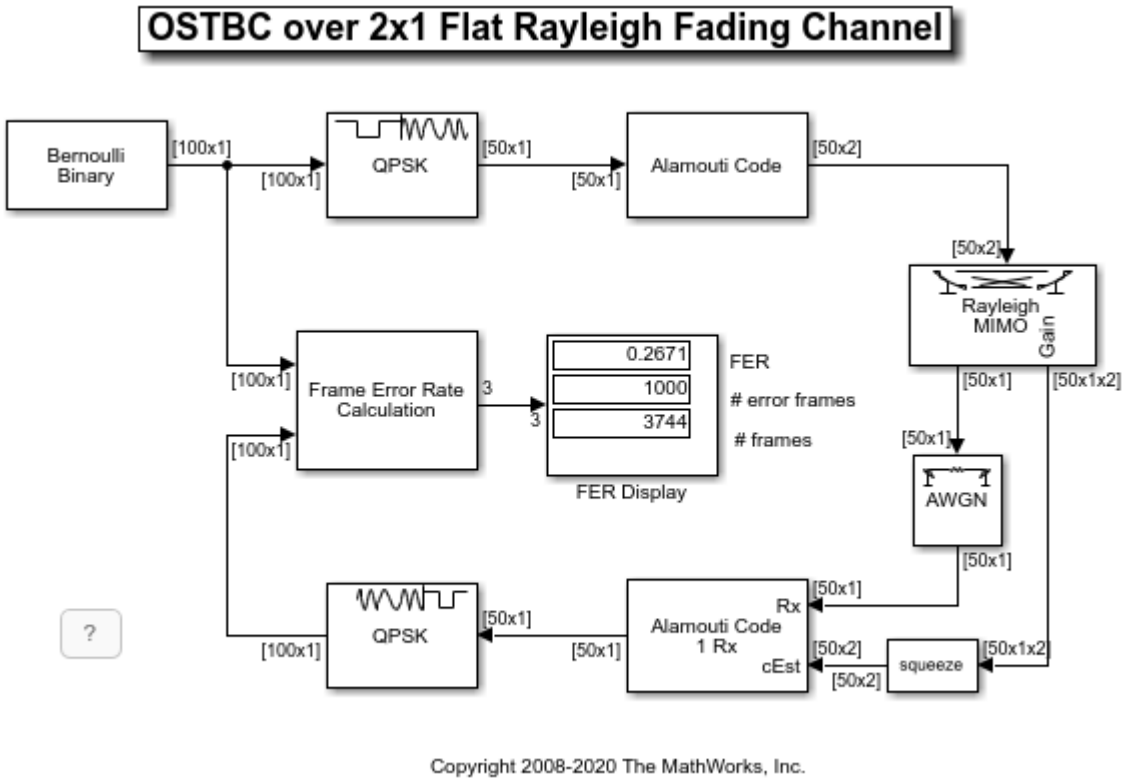
Channel Equalizer

The Channel Equalizer subsystem compensates the fading channel effect at the receiver side and its output is fed into the M-PSK TCM Decoder block for decoding. Note that the channel is flat Rayleigh fading in this model.



OSTBC over 2x1 Flat Rayleigh Fading Channel

The model commostbc.slx replaces the TCM in the above concatenation scheme by a QPSK modulation so that both the models have the same symbol (frame) rate. It uses the same 2x1 MIMO Fading Channel block as in the TCM-OSTBC concatenation model. The QPSK Modulator Baseband block maps the information bits to a QPSK constellation and the QPSK Demodulator Baseband block demodulates the signals from the OSTBC Combiner.



Performance Results

Creating a FER vs. SNR performance curve requires simulations for multiple SNR values, which can be performed by using the `sim` command. We start by defining some simulation parameters

```
SNRRange = 10:2:24;
maxNumErrs = 1e3;  % Number of frame errors
maxNumFrms = 5e6;  % Number of frames processed
```

and then initialize a figure in order to visualize the performance results.

```
fig = figure;
grid on;
hold on;
ax = fig.CurrentAxes;
ax.YScale = 'log';
xlim([SNRRange(1), SNRRange(end)]);
ylim([1e-4 1]);

xlabel('SNR (dB)');
ylabel('FER');
fig.NumberTitle = 'off';
fig.Rrenderer = 'zbuffer';
fig.Name = 'Concatenated OSTBC with TCM';
title('Concatenated OSTBC with TCM');
```

To simulate the OSTBC-TCM concatenated model, we execute the following commands that run the simulation multiple times and plot the results.

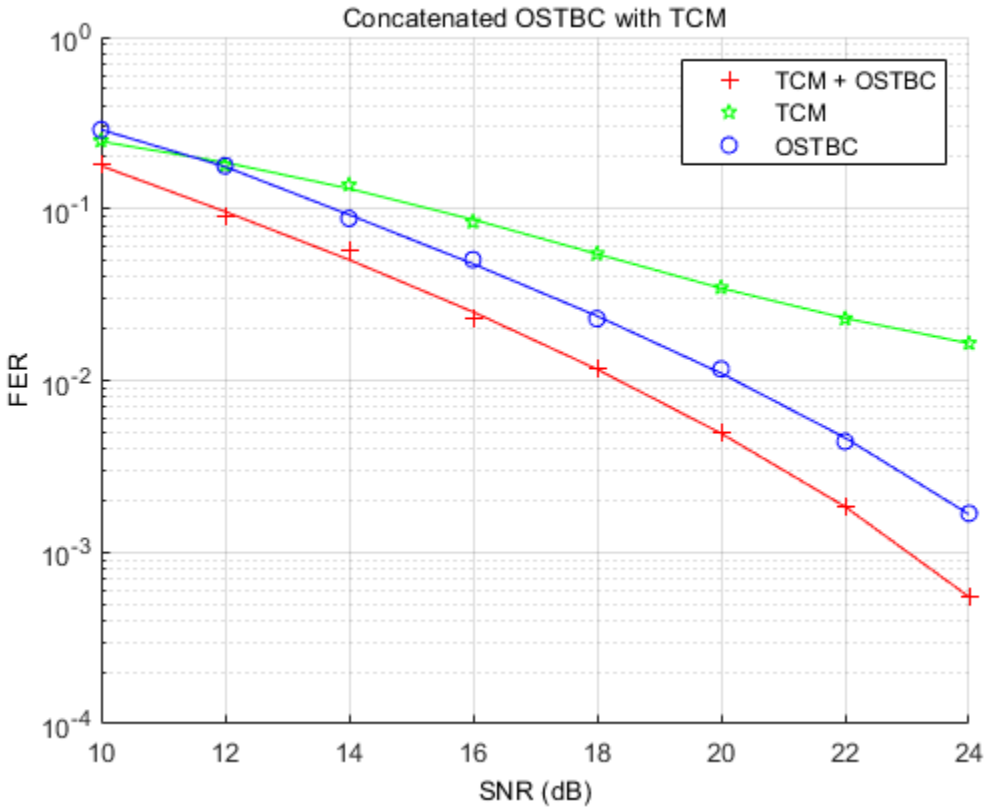
```
FERTCMOSTBC = zeros(length(SNRRange), 3);
for idx = 1:length(SNRRange)
    SNR = SNRRange(idx);
    sim('commtcmostbc');
    FERTCMOSTBC(idx, :) = FER_Data;
    h1 = semilogy(SNRRange(1:idx), FERTCMOSTBC(1:idx, 1), 'r+');
end
fitFERTCMOSTBC = berfit(SNRRange, FERTCMOSTBC(:, 1)');
semilogy(SNRRange, fitFERTCMOSTBC, 'r');
```

Similarly, we can simulate the two reference models via executing

```
FERTCM = zeros(length(SNRRange), 3);
for idx = 1:length(SNRRange)
    SNR = SNRRange(idx);
    sim('commtcm');
    FERTCM(idx, :) = FER_Data;
    h2 = semilogy(SNRRange(1:idx), FERTCM(1:idx, 1), 'gp');
end
fitFERTCM = berfit(SNRRange, FERTCM(:, 1)');
semilogy(SNRRange, fitFERTCM, 'g');

FEROSTBC = zeros(length(SNRRange), 3);
for idx = 1:length(SNRRange)
    SNR = SNRRange(idx);
    sim('commostbc');
    FEROSTBC(idx, :) = FER_Data;
    h3 = semilogy(SNRRange(1:idx), FEROSTBC(1:idx, 1), 'bo');
end
```

```
fitFEROSTBC = berfit(SNRRange, FEROSTBC(:, 1)');
semilogy(SNRRange, fitFEROSTBC, 'b');

legend([h1, h2, h3], 'TCM + OSTBC', 'TCM', 'OSTBC');
```

The FER vs. SNR performance result is presented in the following figure.



As expected, the concatenation scheme provides a significant diversity gain over the TCM scheme and about 2dB coding gain over the Alamouti code.

Further Exploration

Upon loading the simulation models, variables are created in the MATLAB® workspace which can be modified to explore the effects of different parameter settings such as `Samples per frame` (variable `frameLen`), `Trellis structure` (variable `trellis`) or `Maximum Doppler shift (Hz)` (variable `maxDopp`) on the system performance.

Selected Bibliography

1 S. M. Alamouti, "A simple transmit diversity technique for wireless communications," *IEEE® Journal on Selected Areas in Communications*, vol. 16, no. 8, pp. 1451-1458, Oct. 1998.

2 V. Tarokh, H. Jafarkhami, and A. R. Calderbank, "Space-time block codes from orthogonal designs," *IEEE Transactions on Information Theory*, vol. 45, no. 5, pp. 1456-1467, Jul. 1999.

3 G. Ungerboeck, "Channel coding with multilevel/phase signals," *IEEE Transactions on Information Theory*, vol. IT-28, no. 1, pp. 55-67, Jan. 1982.

4 S. M. Alamouti, V. Tarokh, and P. Poon, "Trellis-coded modulation and transmit diversity: Design criteria and performance evaluation," in *Proc. IEEE International Conference on Universal Personal Communications (ICUPC'98)*, Florence, Italy, Oct. 1998, pp. 703-707.

5 Y. Gong and K. B. Letaief, "Concatenated space-time block coding with trellis coded modulation in fading channels," *IEEE Transactions on Wireless Communications*, vol. 1, no. 4, pp. 580-590, Oct. 2002.

BER Performance of Different Equalizers

This example shows the BER performance of several types of equalizers in a static channel with a null in the passband. The example constructs and implements a linear equalizer object and a decision feedback equalizer (DFE) object. It also initializes and invokes a maximum likelihood sequence estimation (MLSE) equalizer. The MLSE equalizer is first invoked with perfect channel knowledge, then with a straightforward but imperfect channel estimation technique.

As the simulation progresses, it updates a BER plot for comparative analysis between the equalization methods. It also shows the signal spectra of the linearly equalized and DFE equalized signals. It also shows the relative burstiness of the errors, indicating that at low BERs, both the MLSE algorithm and the DFE algorithm suffer from error bursts. In particular, the DFE error performance is burstier with detected bits fed back than with correct bits fed back. Finally, during the "imperfect" MLSE portion of the simulation, it shows and dynamically updates the estimated channel response.

To experiment with this example, you can change such parameters as the channel impulse response, the number of equalizer tap weights, the recursive least squares (RLS) forgetting factor, the least mean square (LMS) step size, the MLSE traceback length, the error in estimated channel length, and the maximum number of errors collected at each Eb/No value.

Code Structure

This example relies on these helper scripts and functions to perform link simulations over a range of Eb/No values.

eqber_adaptive.m - a script that runs link simulations for linear and DFE equalizers

eqber_mlse.m - a script that runs link simulations for ideal and imperfect MLSE equalizers

eqber_siggen.m - a script that generates a binary phase shift keying (BPSK) signal with no pulse shaping, then processes it through the channel and adds noise

eqber_graphics.m - a function that generates and updates plots showing the performance of the linear, DFE, and MLSE equalizers.

The scripts eqber_adaptive and eqber_mlse illustrate how to use adaptive and MLSE equalizers across multiple blocks of data such that state information is retained between data blocks.

Signal and Channel Parameters

Set parameters related to the signal and channel. Use BPSK without any pulse shaping, and a 5-tap real-valued symmetric channel impulse response. (See section 10.2.3 of Digital Communications by J. Proakis, 4th Ed., for more details on the channel.) Set initial states of data and noise generators. Set the Eb/No range.

```
% System simulation parameters
Fs = 1;           % sampling frequency (notional)
nBits = 2048;     % number of BPSK symbols per vector
maxErrs = 200;    % target number of errors at each Eb/No
maxBits = 1e6;    % maximum number of symbols at each Eb/No

% Modulated signal parameters
M = 2;                    % order of modulation
Rs = Fs;                  % symbol rate
nSamp = Fs/Rs;            % samples per symbol
```

```
Rb = Rs*log2(M);           % bit rate

% Channel parameters
chnl = [0.227 0.460 0.688 0.460 0.227]';  % channel impulse response
chnlLen = length(chnl);                   % channel length, in samples
EbNo = 0:14;                              % in dB
BER = zeros(size(EbNo));                  % initialize values

% Create BPSK modulator
bpskMod = comm.BPSKModulator;

% Specify a seed for the random number generators to ensure repeatability.
rng(12345)
```

Adaptive Equalizer Parameters

Set parameter values for the linear and DFE equalizers. Use a 31-tap linear equalizer, and a DFE with 15 feedforward and feedback taps. Use the recursive least squares (RLS) algorithm for the first block of data to ensure rapid tap convergence. Use the least mean square (LMS) algorithm thereafter to ensure rapid execution speed.

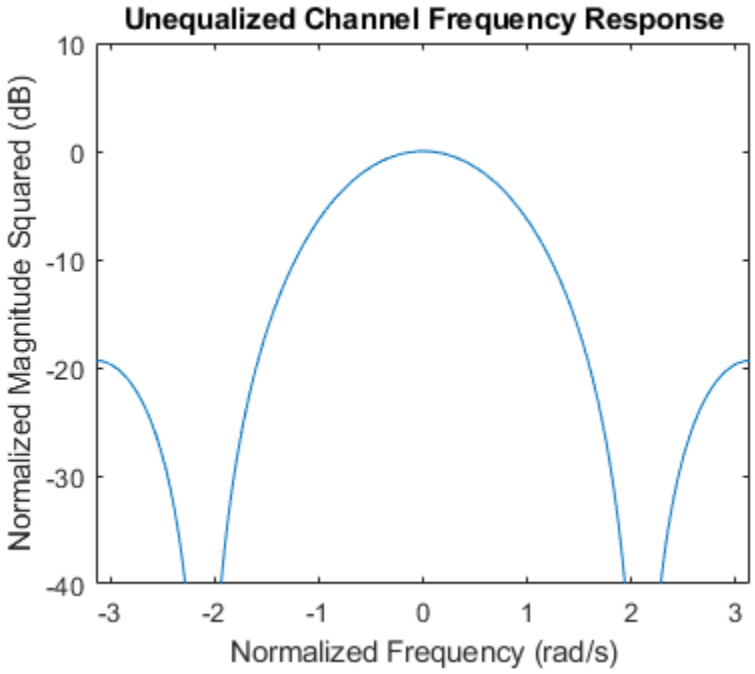
```
% Linear equalizer parameters
nWts = 31;               % number of weights
algType = 'RLS';         % RLS algorithm
forgetFactor = 0.999999; % parameter of RLS algorithm

% DFE parameters - use same update algorithms as linear equalizer
nFwdWts = 15;            % number of feedforward weights
nFbkWts = 15;            % number of feedback weights
```

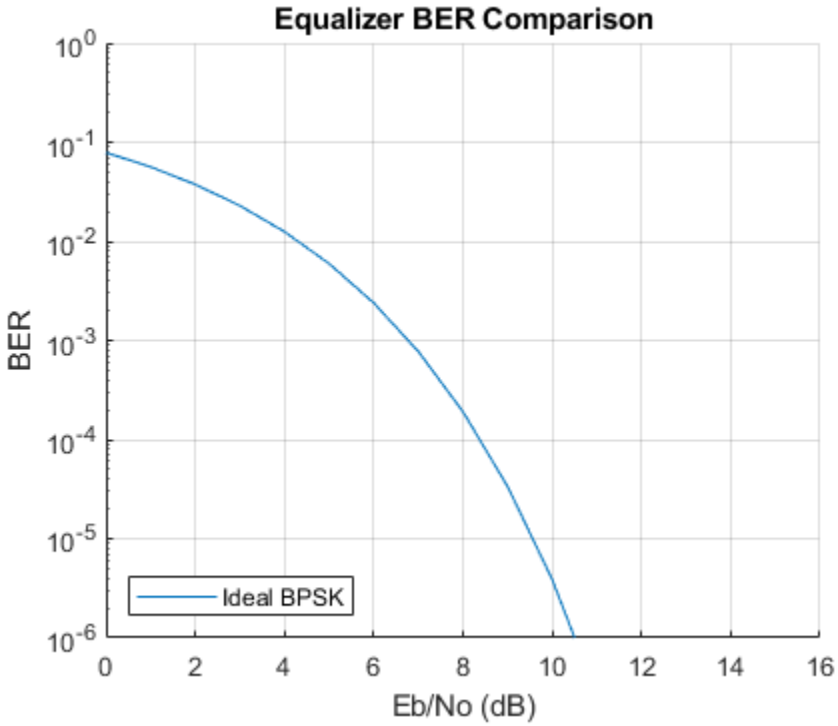
MLSE Equalizer & Channel Estimation Parameters and Initial Visualization

Set the parameters of the MLSE equalizer. Use a traceback length of six times the length of the channel impulse response. Initialize the equalizer states. Set the equalization mode to "continuous", to enable seamless equalization over multiple blocks of data. Use a cyclic prefix in the channel estimation technique, and set the length of the prefix. Assume that the estimated length of the channel impulse response is one sample longer than the actual length.

```
% MLSE equalizer parameters
tbLen = 30;                         % MLSE equalizer traceback length
numStates = M^(chnlLen-1);          % number of trellis states
[mlseMetric,mlseStates,mlseInputs] = deal([]);
const = constellation(bpskMod);     % signal constellation
mlseType = 'ideal';                 % perfect channel estimates at first
mlseMode = 'cont';                  % no MLSE resets

% Channel estimation parameters
chnlEst = chnl;          % perfect estimation initially
prefixLen = 2*chnlLen;   % cyclic prefix length
excessEst = 1;           % length of estimated channel impulse response
% beyond the true length

% Initialize the graphics for the simulation. Plot the unequalized channel
% frequency response, and the BER of an ideal BPSK system.
idealBER = berawgn(EbNo,'psk',M,'nondiff');

[hBER,hLegend,legendString,hLinSpec,hDfeSpec,hErrs,hText1,hText2, ...
```

```
hFit,hEstPlot,hFig,hLinFig,hDfeFig] = eqber_graphics('init', ...
chnl,EbNo,idealBER,nBits);
```





Construct RLS and LMS Linear and DFE Equalizer Objects

The RLS update algorithm is used to adapt the equalizer tap weights and reference tap is set to center tap.

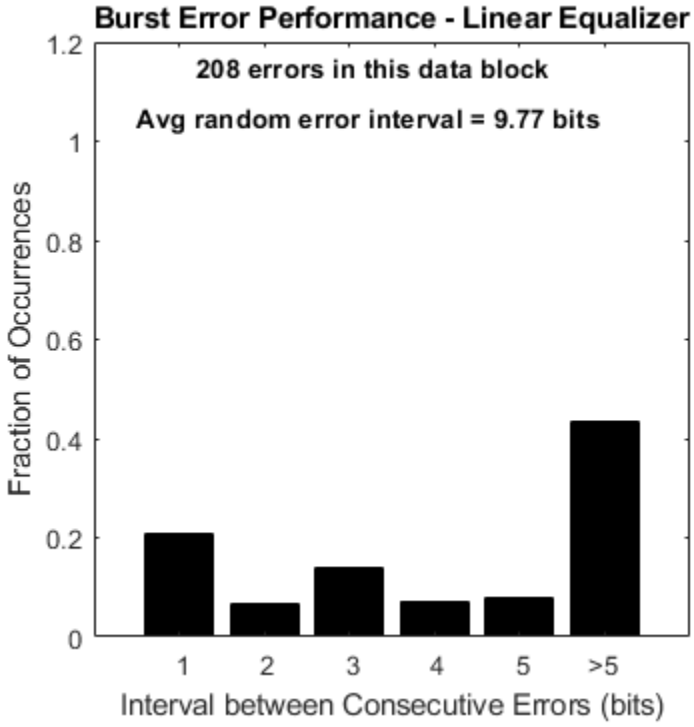
```
linEq = comm.LinearEqualizer('Algorithm',algType, ...
    'ForgettingFactor',forgetFactor, ...
    'NumTaps',nWts, ...
    'Constellation',const, ...
    'ReferenceTap',round(nWts/2), ...
    'TrainingFlagInputPort',true);

dfeEq = comm.DecisionFeedbackEqualizer('Algorithm',algType, ...
    'ForgettingFactor',forgetFactor, ...
    'NumForwardTaps',nFwdWts, ...
    'NumFeedbackTaps',nFbkWts, ...
    'Constellation',const, ...
    'ReferenceTap',round(nFwdWts/2), ...
    'TrainingFlagInputPort',true);
```

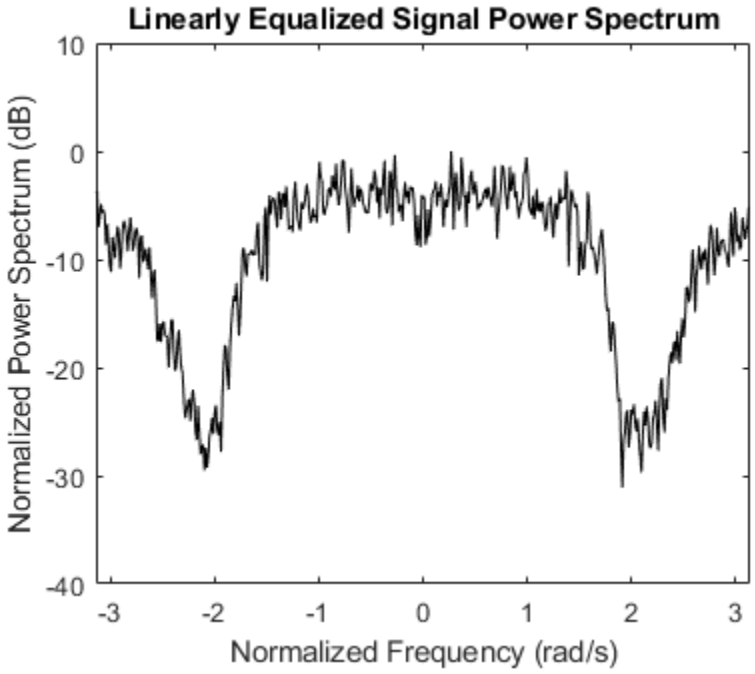
Linear Equalizer

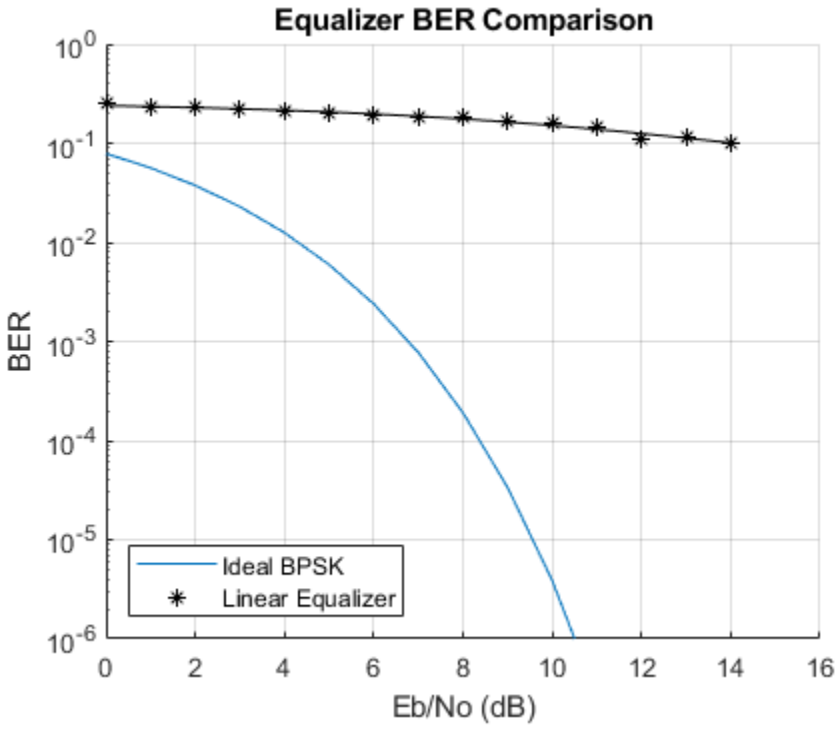
Run the linear equalizer, and plot the equalized signal spectrum, the BER, and the burst error performance for each data block. Note that as the Eb/No increases, the linearly equalized signal spectrum has a progressively deeper null. This highlights the fact that a linear equalizer must have many more taps to adequately equalize a channel with a deep null. Note also that the errors occur with small inter-error intervals, which is to be expected at such a high error rate.

See eqber_adaptive.m for a listing of the simulation code for the adaptive equalizers.

```
firstRun = true;  % flag to ensure known initial states for noise and data
eqType = 'linear';
eqber_adaptive;
```


Burst Error Performance - Linear Equalizer
208 errors in this data block
Avg random error interval = 9.77 bits
Fraction of Occurrences
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
>5
Interval between Consecutive Errors (bits)


Linearly Equalized Signal Power Spectrum
Normalized Power Spectrum (dB)
10
0
-10
-20
-30
-40
-3
-2
-1
0
1
2
3
Normalized Frequency (rad/s)



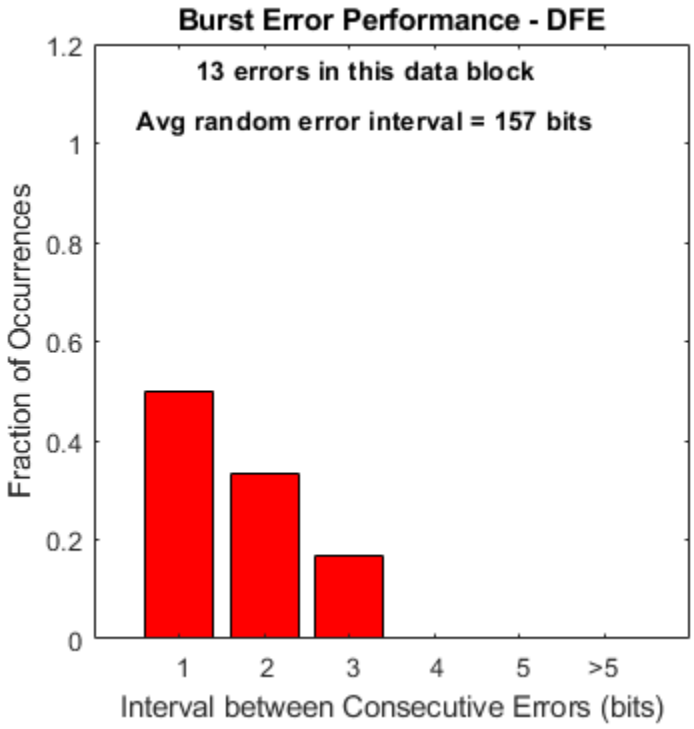
Decision Feedback Equalizer

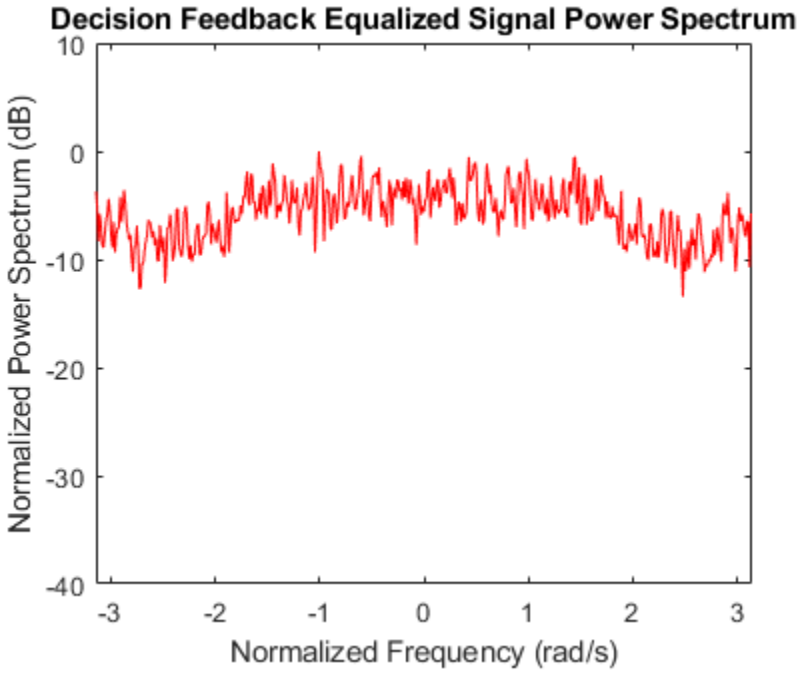
Run the DFE, and plot the equalized signal spectrum, the BER, and the burst error performance for each data block. Note that the DFE is much better able to mitigate the channel null than the linear equalizer, as shown in the spectral plot and the BER plot. The plotted BER points at a given Eb/No value are updated every data block, so they move up or down depending on the number of errors collected in that block. Note also that the DFE errors are somewhat bursty, due to the error propagation caused by feeding back detected bits instead of correct bits. The burst error plot shows that as the BER decreases, a significant number of errors occurs with an inter-error arrival of five bits or less. (If the DFE equalizer were run in training mode at all times, the errors would be far less bursty.)

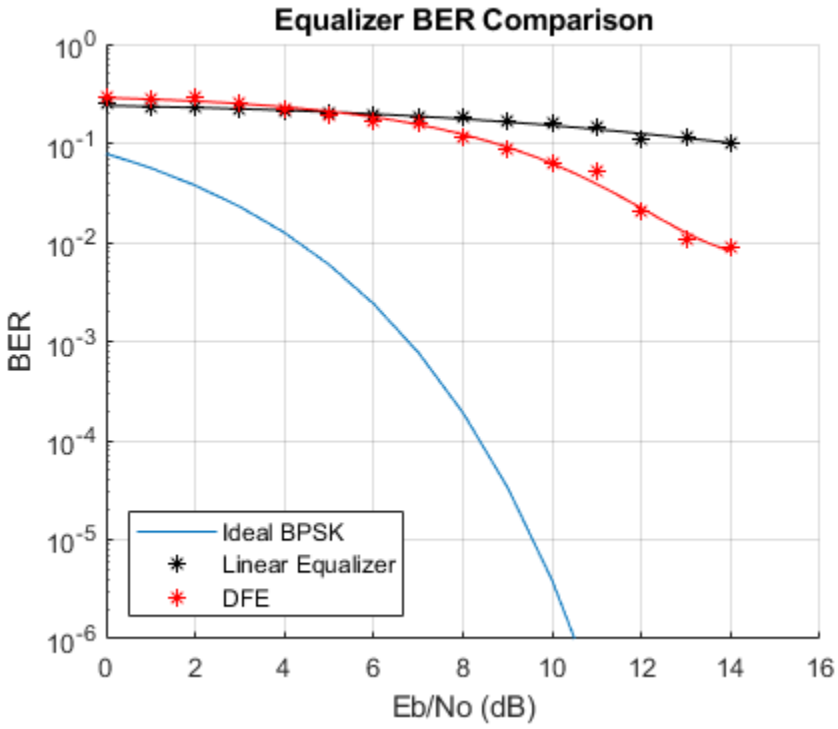
For every data block, the plot also indicates the average inter-error interval if those errors were randomly occurring.

See eqber_adaptive.m for a listing of the simulation code for the adaptive equalizers.

```
close(hFig(ishghandle(hFig)));

eqType = 'dfe';
eqber_adaptive;
```


Burst Error Performance - DFE
13 errors in this data block
Avg random error interval = 157 bits
Fraction of Occurrences
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
>5
Interval between Consecutive Errors (bits)


Decision Feedback Equalized Signal Power Spectrum
Normalized Power Spectrum (dB)
10
0
-10
-20
-30
-40
-3
-2
-1
0
1
2
3
Normalized Frequency (rad/s)

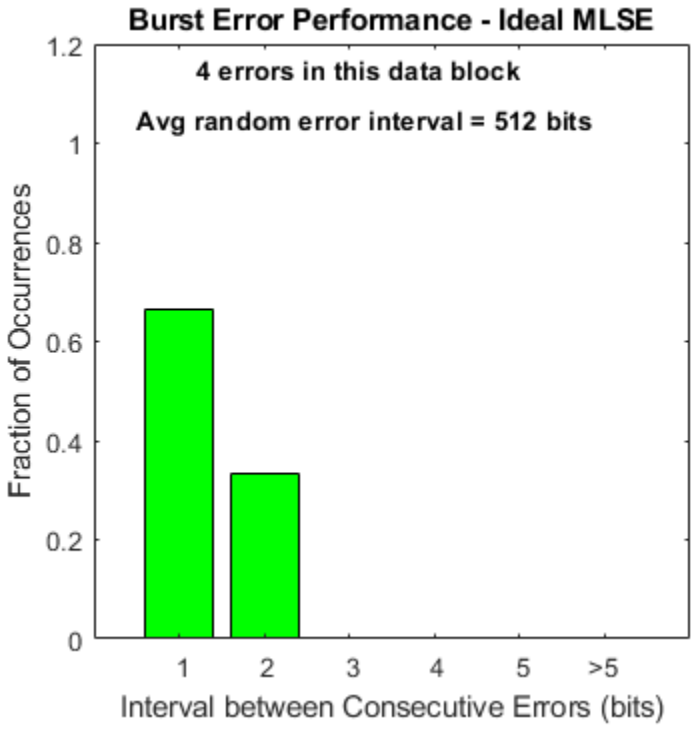


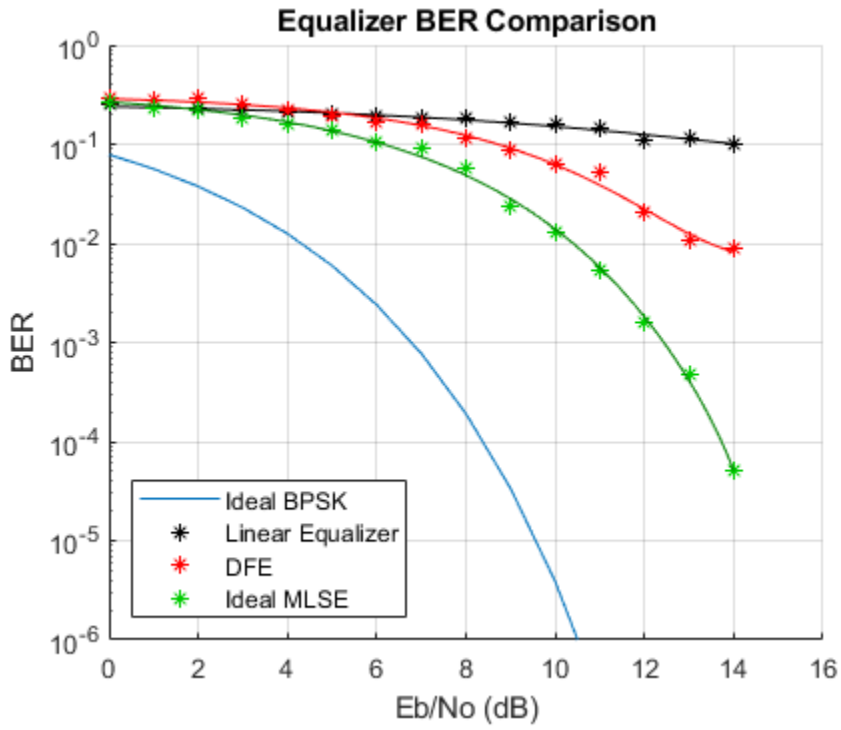
Ideal MLSE Equalizer, with Perfect Channel Knowledge

Run the MLSE equalizer with a perfect channel estimate, and plot the BER and the burst error performance for each data block. Note that the errors occur in an extremely bursty fashion. Observe, particularly at low BERs, that the overwhelming percentage of errors occur with an inter-error interval of one or two bits.

See eqber_mlse.m for a listing of the simulation code for the MLSE equalizers.

```
close(hLinFig(ishghandle(hLinFig)),hDfeFig(ishghandle(hDfeFig)));

eqType = 'mlse';
mlseType = 'ideal';
eqber_mlse;
```





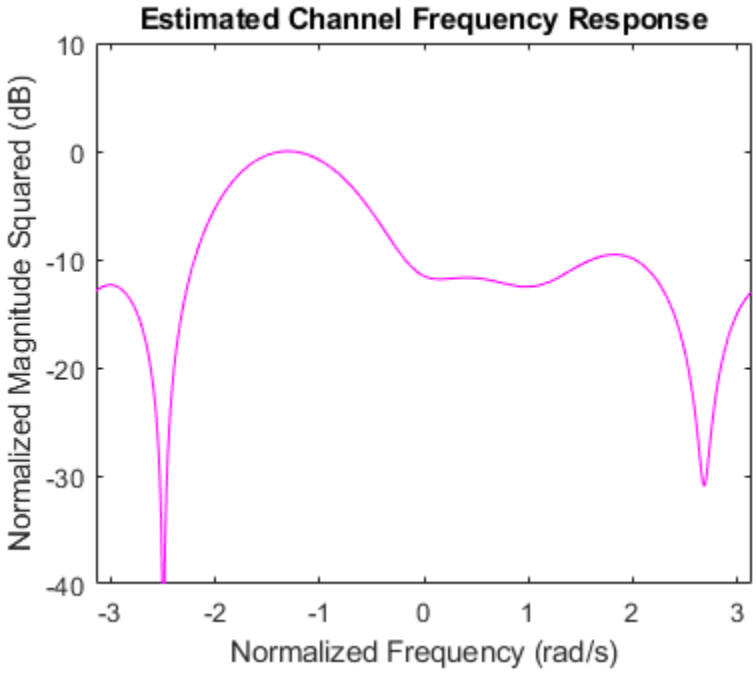
MLSE Equalizer with an Imperfect Channel Estimate

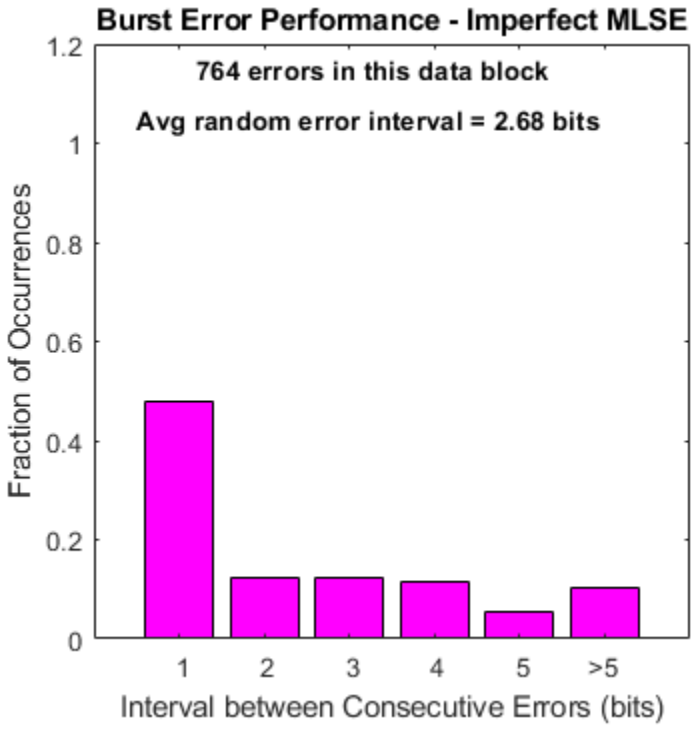
Run the MLSE equalizer with an imperfect channel estimate, and plot the BER and the burst error performance for each data block. These results align fairly closely with the ideal MLSE results. (The

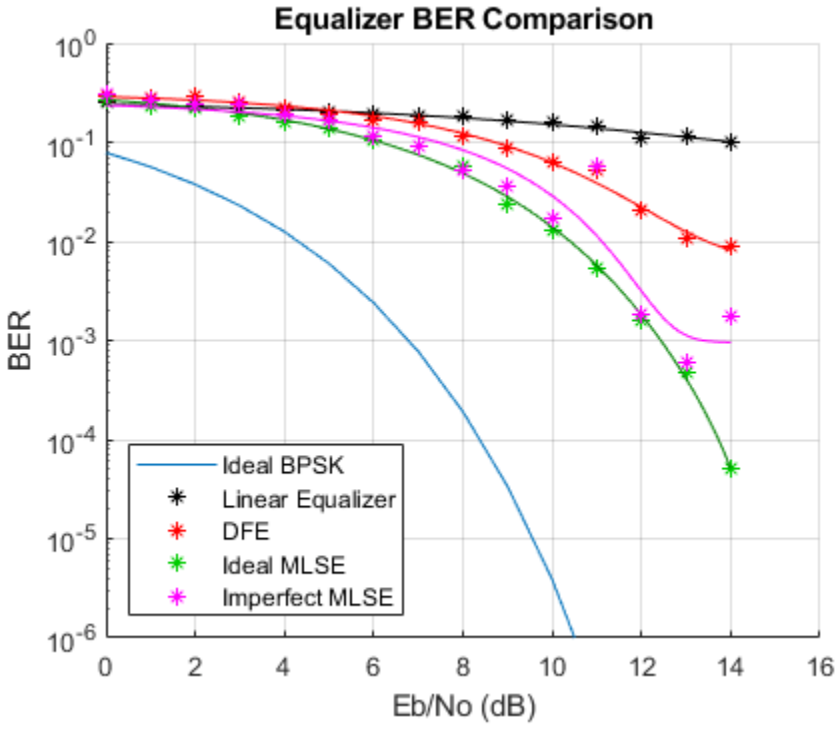
channel estimation algorithm is highly dependent on the data, such that an FFT of a transmitted data block has no nulls.) Note how the estimated channel plots compare with the actual channel spectrum plot.

See eqber_mlse.m for a listing of the simulation code for the MLSE equalizers.

```
mlseType = 'imperfect';
eqber_mlse;
```




Burst Error Performance - Imperfect MLSE
764 errors in this data block
Avg random error interval = 2.68 bits
Fraction of Occurrences
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
>5
Interval between Consecutive Errors (bits)


Equalizer BER Comparison
BER
Ideal BPSK
Linear Equalizer
DFE
Ideal MLSE
Imperfect MLSE
Eb/No (dB)

OFDM Synchronization

This example shows a method for digital communication with OFDM synchronization based upon the IEEE® 802.11a™ standard. System objects™ from the Communications Toolbox are utilized to provide OFDM modulation and demodulation and help synchronization functionality. In particular, this example illustrates methods to address real-world wireless communication issues like carrier frequency recovery, timing recovery, and frequency domain equalization.

Implementations

This example describes a MATLAB® implementation of OFDM synchronization, based upon the IEEE 802.11a standard [3].

Introduction

The IEEE 802.11a standard describes the transmission of an OFDM modulated signal for information exchange between systems in local and metropolitan area networks. This example utilizes the physical layer outlined by that standard, specifically the preamble symbols and the OFDM grid structure.

The purpose of this example is:

- To model a general OFDM wireless communication system that is able to successfully recover a message, which was corrupted by various simulated channel impairments.
- To illustrate the use of key Communications Toolbox™ tools for OFDM system design and OFDM symbol synchronization
- To illustrate the performance benefits of MATLAB Coder™

Initialization

Adjustable transmitter parameters including the payload message in each frame that consists of several OFDM symbols and the number of transmitted frames.

```
message = 'Live long and prosper, from the Communications Toolbox Team at MathWorks!';
numFrames = 1e2;

% Adjustable channel parameters
EbN0dB = 12; % Channel noise level (dB)
frequencyOffset = 1e4; % Frequency offset (Hz)
phaseOffset = 15; % Phase offset (Degrees)
delay = 80; % Initial sample offset for entire data stream (samples)

% Display recovered messages
displayRecoveredMsg = false;

% Enable scope visualizations
useScopes = true;

% Check for MATLAB Coder license
useCodegen = checkCodegenLicense;
if useCodegen
  fprintf(['--MATLAB Coder license found. ',...
    'Transmitter and receiver functions will be compiled for ',...
    'additional simulation acceleration.--\n']);
```

```
end

% By default the transmitter and receiver functions will be recompiled
% between every run, which is not always necessary. To disable receiver
% compilation, change "compileIt" to false.
compileIt = useCodegen;
```

```
--MATLAB Coder license found. Transmitter and receiver functions will be compiled for additional
```

Code Architecture for the System

This example models a digital communication system based upon the IEEE 802.11a standard [3]. The system is broken down into four functions: generateOFDMSignal, applyOFDMChannel, receiveOFDMSignal, and calculateOFDMBER.

1) generateOFDMSignal: set up and step an OFDMTransmitter System object. The object converts the payload message into a bit stream which is first PSK modulated, then OFDM modulated, and finally prepended by preamble OFDM symbols to form an individual frame. The transmitter repeats this frame `numFrames` times.

2) applyOFDMChannel: models the channel with carrier offset, timing offset, and additive white Gaussian noise (AWGN).

3) receiveOFDMSignal: set up and step an OFDMReceiver System object. The object models a series of components at the receiver, including timing recovery, carrier frequency recovery, channel equalization, and demodulation. The object can also be configured to show multiple scopes to visualize the receiver processing. The output of the OFDMReceiver object's step method is the decoded bit stream from those detected frames.

4) calculateOFDMBER: calculate the system frame error rate (FER) and bit error rate (BER) based on the original payload message in each frame and the bit output from the OFDMReceiver System object.

Description of the Individual Components and Algorithms

Transmitter

The OFDMTransmitter System object generates an OFDM signal based upon the IEEE 802.11a standard with a supplied ASCII payload. Each transmission frame is made up of several OFDM symbols, including preamble and data symbols. Identical frames are repeated by the transmitter based on the value supplied. Frames are padded to fill the OFDM grid when necessary.

Channel

This component simulates the effects of over-the-air transmission. It degrades the transmitted signal with both phase and frequency offset, a delay to mimic channel delay between transmitter and receiver, and AWGN. The noise level of the AWGN is given in dB.

Receiver

This OFDMReceiver System object recovers the original transmitted payload message. It is divided into four primary operations in this order:

1) Timing Recovery: This component is responsible for determining the sample location of the start of a given frame. More specifically, it utilizes a known preamble sequence in the received frame found through a cross-correlation. The cross-correlated data will contain a specific peak arrangement/

spacing which allows for identification. The preamble itself is designed to produce this specific shape in the time domain. This identification method is based upon [1]. The locatePreamble method of the object, which is responsible for this operation, uses a normalized minimum peak height, and a minimum number of required peaks to provide a possible preamble match.

2) Carrier Frequency Recovery: Frequency estimation is accomplished by calculating the phase difference in the time domain between halves of the long portion of the 802.11a preamble. This phase difference Phi is then converted to a frequency offset. This is a common technique originally published by Schmidl and Cox [2]. This implementation of the phase measurement assumes that the true offset is within pi, or one frequency bin of the FFT. In the case of 802.11a a bin is 312.5kHz wide.

3) Frequency Domain Equalization: Since the frequency estimate can be inaccurate, additional phase rotation will exist at the subcarrier level of the OFDM symbol. As well as phase rotations, channel fading will also affect the received signal. Both of these impairments are corrected by a frequency domain equalizer. The equalizer has two stages, utilizing both preamble and pilot data. First, the received payload is equalized through the use of taps generated from the received long preamble samples. Then the pilot subcarriers are extracted, and interpolated in frequency to provide a full channel estimate. The payload is next equalized using these pilot estimates.

4) Data Decoder: Finally the OFDM subcarriers are demodulated and then, PSK demodulated into bits, from which the original payload message can be recovered.

BER Calculation

This component calculates the system FER and BER based on the original payload message and the decoded bit stream from the detected frames at the receiver. The undetected frames are not counted in the calculation.

Display of Recovered Message

The recovered message at the receiver is displayed for each detected frame. Since the original message length is not sent to the receiver, the padded bits in each frame are also recovered into characters and displayed. So you may see up to 7 meaningless characters at the end of each recovered message.

Scopes

- constellation diagrams showing the received signal before and after frequency domain equalization
- vector plot of the equalizer taps used for a given frame
- a spectrum analyzer displaying detected frames of data
- a time plot displaying the start of detected frames
- a time plot displaying the frequency estimate of the transmitter's carrier offset for detected frames

OFDM Synchronization Test Overview

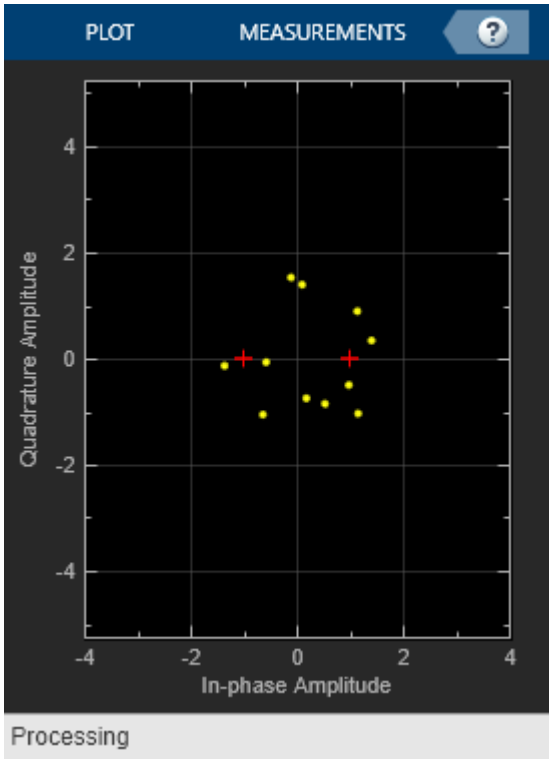
A large data vector is regenerated for a given EbN0 value by the generateOFDMSignal function. This data is then passed through the applyOFDMChannel function which introduces several common channel impairments. Finally the data is passed to the receiver for recovery. The receiveOFDMSignal function operates by processing data on a frame-by-frame basis. This processing mechanism is self-

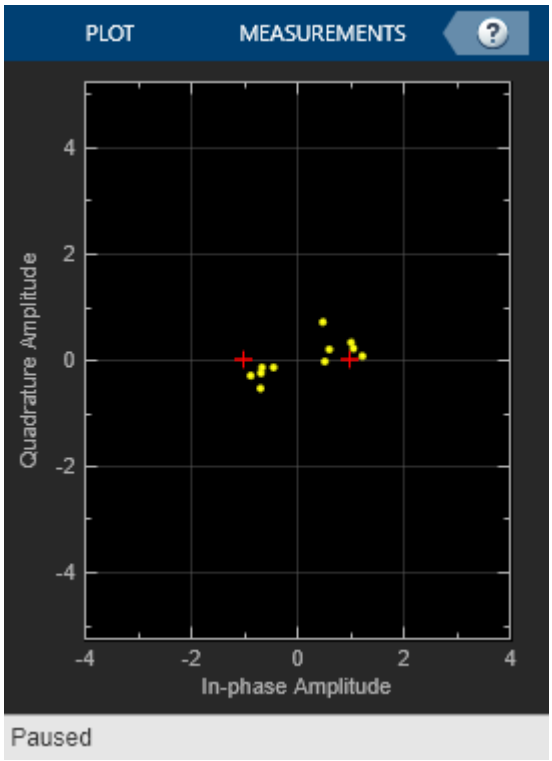
contained for performance benefits when using code generation and for code simplicity. This script by default generates code for the transmitter and receiver functions; this is accomplished by using the **codegen** command provided by the MATLAB Coder™ product. The **codegen** command translates MATLAB® functions to a C++ static or dynamic library, executable, or to a MEX file, producing a code for accelerated execution. The generated C code runs several times faster than the original MATLAB code.

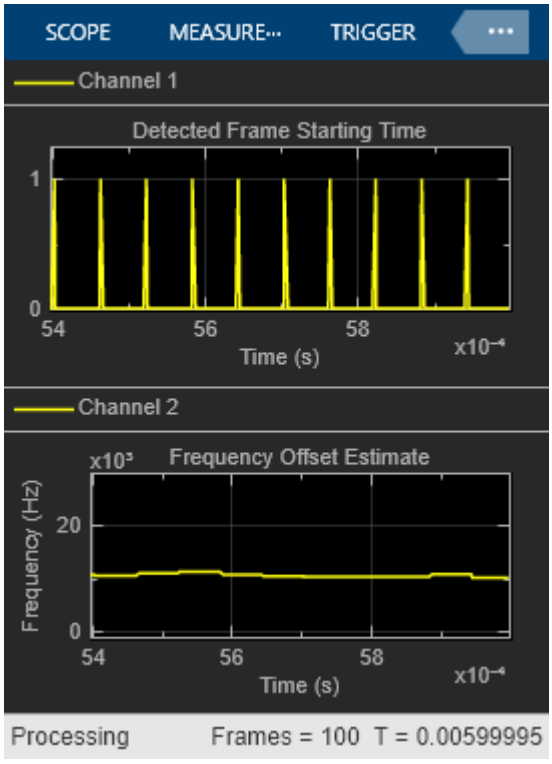
During operation, the receiver will display a series of plots illustrating certain synchronization results and effects on the signal.

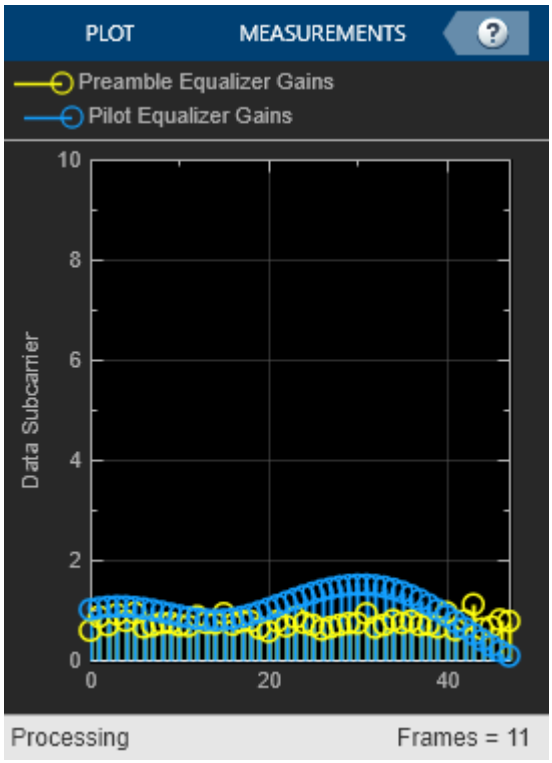
```
% Compile transmitter with MATLAB Coder
if compileIt
    codegen generateOFDMSignal -args {coder.Constant(message), coder.Constant(numFrames)}
end

% Generate transmission signal
if useCodegen
    [txSig, frameLen] = generateOFDMSignal_mex(message, numFrames);
else
    [txSig, frameLen] = generateOFDMSignal(message, numFrames);
end

% Pass signal through channel
rxSig = applyOFDMChannel(txSig, EbN0dB, delay, frequencyOffset, phaseOffset);

% Compile receiver with MATLAB Coder
if compileIt
   codegen  receiveOFDMSignal -args {rxSig, coder.Constant(frameLen), coder.Constant(displayReco
end

% Recover signal
if useCodegen
    [decMsgInBits, numFramesDetected] = receiveOFDMSignal_mex(rxSig, frameLen, displayRecoveredMs
else
    [decMsgInBits, numFramesDetected] = receiveOFDMSignal(rxSig, frameLen, displayRecoveredMsg, u
end

% Calculate average BER
[FER, BER] = calculateOFDMBER(message, decMsgInBits, numFramesDetected);
fprintf('\nAt EbNo = %5.2fdB, %d frames detected among the %d transmitted frames with FER = %f an
    EbN0dB, numFramesDetected, numFrames, FER, BER);
```

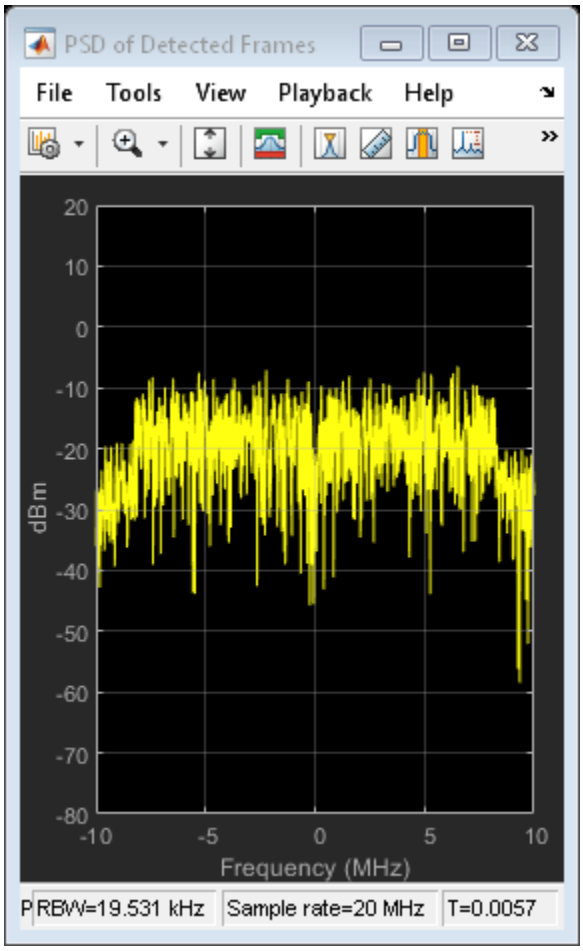
```
Code generation successful.

Code generation successful.


At EbNo = 12.00dB, 100 frames detected among the 100 transmitted frames with FER = 0.010000 and B
```


PLOT
MEASUREMENTS
Quadrature Amplitude
In-phase Amplitude
4
2
0
-2
-4
-4
-2
0
2
4
Processing


PLOT
MEASUREMENTS
Quadrature Amplitude
In-phase Amplitude
4
2
0
-2
-4
-4
-2
0
2
4
Paused


SCOPE
MEASURE
TRIGGER
Channel 1
Detected Frame Starting Time
1
0
54
56
58
Time (s)
x10⁻⁴
Channel 2
x10³
Frequency Offset Estimate
Frequency (Hz)
20
0
54
56
58
Time (s)
x10⁻⁴
Processing
Frames = 100 T = 0.00599995


PLOT
MEASUREMENTS
Preamble Equalizer Gains
Pilot Equalizer Gains
Data Subcarrier
10
8
6
4
2
0
0
20
40
Processing
Frames = 11



Summary

This example utilizes several MATLAB System objects to simulate digital communication with OFDM over an AWGN channel. It shows how to model several parts of the OFDM system such as modulation, frequency estimation, timing recovery, and equalization. The simulation also displays information about the operation of the synchronization algorithms through a series of plots. This example also utilizes code generation, allowing the simulation to run several times faster than the original MATLAB code.

Appendix

The following System objects are used in this example:

- OFDMTransmitter.m
- OFDMReceiver.m
- OFDMScopes.m

The following helper functions are used in this example:

- generateOFDMSignal.m
- applyOFDMChannel.m

- receiveOFDMSignal.m
- processOFDMScopes.m
- calculateOFDMBER.m
- getOFDMPreambleAndPilot.m

References

1 Minn, H.; Zeng, M.; Bhargava, V.K., "On timing offset estimation for OFDM systems," Communications Letters, IEEE , vol.4, no.7, pp.242,244, July 2000

2 Schmidl, T.M.; Cox, D.C., "Robust frequency and timing synchronization for OFDM," Communications, IEEE Transactions on , vol.45, no.12, pp.1613,1621, Dec 1997

3 IEEE Std 802.11a, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," 1999.

QPSK Transmitter and Receiver

This example shows the implementation of a QPSK transmitter and receiver with MATLAB®. In particular, this example illustrates methods to address real-world wireless communications issues like carrier frequency and phase offset, timing recovery and frame synchronization. For the Simulink® implementation of the same system, refer to the “QPSK Transmitter and Receiver in Simulink” on page 8-262 example.

Introduction

The transmitted QPSK data undergoes impairments that simulate the effects of wireless transmission such as addition of Additive White Gaussian Noise (AWGN), introduction of carrier frequency and phase offset, and timing drift. To cope with these impairments, this example provides a reference design of a practical digital receiver. The receiver includes correlation-based coarse frequency compensation, PLL-based fine frequency compensation, PLL-based symbol timing recovery, frame synchronization, and phase ambiguity resolution.

This example serves two main purposes:

- To model a general wireless communication system that is able to successfully recover a message, which was corrupted by various simulated channel impairments.
- To illustrate the use of key Communications Toolbox™ synchronization components including coarse and fine carrier frequency compensation, closed-loop timing recovery with bit stuffing and stripping, frame synchronization and carrier phase ambiguity resolution.

Initialization

The commqpsktxrx_init.m script initializes simulation parameters and generates the structure prmQPSKTxRx.

```
prmQPSKTxRx = commqpsktxrx_init %#ok<*NOPTS> % QPSK system parameters

useScopes = true;   % true if scopes are to be used
printReceivedData = false; %true if the received data is to be printed
compileIt = false;  % true if code is to be compiled
useCodegen = false; % true to run the generated mex file
```

```
prmQPSKTxRx =

  struct with fields:

               ModulationOrder: 4
                 Interpolation: 2
                    Decimation: 1
                          Rsym: 50000
                          Tsym: 2.0000e-05
                            Fs: 100000
                    TotalFrame: 1000
                    BarkerCode: [1 1 1 1 1 -1 -1 1 1 -1 1 -1 1]
                  BarkerLength: 13
                  HeaderLength: 26
                       Message: 'Hello world'
                 MessageLength: 16
               NumberOfMessage: 20
```

```
              PayloadLength: 2240
                  FrameSize: 1133
                  FrameTime: 0.0227
              RolloffFactor: 0.5000
              ScramblerBase: 2
         ScramblerPolynomial: [1 1 1 0 1]
 ScramblerInitialConditions: [0 0 0 0]
     RaisedCosineFilterSpan: 10
                PhaseOffset: 47
                       EbNo: 13
            FrequencyOffset: 5000
                  DelayType: 'Triangle'
               DesiredPower: 2
            AveragingLength: 50
               MaxPowerGain: 20
     MaximumFrequencyOffset: 6000
 PhaseRecoveryLoopBandwidth: 0.0100
 PhaseRecoveryDampingFactor: 1
TimingRecoveryLoopBandwidth: 0.0100
TimingRecoveryDampingFactor: 1
    TimingErrorDetectorGain: 5.4000
  PreambleDetectorThreshold: 20
                MessageBits: [11200x1 double]
                    BerMask: [1540x1 double]
```

Code Architecture for the System Under Test

This example models a digital communication system using QPSK modulation. The function runQPSKSystemUnderTest.m models this communication environment. The QPSK transceiver model in this script is divided into the following four main components.

1) QPSKTransmitter.m: generates the bit stream and then encodes, modulates and filters it.

2) QPSKChannel.m: models the channel with carrier offset, timing offset, and AWGN.

3) QPSKReceiver.m: models the receiver, including components for phase recovery, timing recovery, decoding, demodulation, etc.

4) QPSKScopes.m: optionally visualizes the signal using time scopes, frequency scopes, and constellation diagrams.

Each component is modeled using a System object. To see the construction of the four main System object components, refer to runQPSKSystemUnderTest.m.

Description of the Individual Components

Transmitter

This component generates a message using ASCII characters, converts the characters to bits, and prepends a Barker code for receiver frame synchronization. This data is then modulated using QPSK and filtered with a square root raised cosine filter.

Channel

This component simulates the effects of over-the-air transmission. It degrades the transmitted signal with both phase and frequency offset, a time-varying delay to mimic clock skew between transmitter and receiver, and AWGN.

Receiver

This component regenerates the original transmitted message. It is divided into seven subcomponents.

1) Automatic Gain Control: Sets its output power to a level ensuring that the equivalent gains of the phase and timing error detectors keep constant over time. The AGC is placed before the **Raised Cosine Receive Filter** so that the signal amplitude can be measured with an oversampling factor of two. This process improves the accuracy of the estimate.

2) Coarse frequency compensation: Uses a correlation-based algorithm to roughly estimate the frequency offset and then compensate for it. The estimated coarse frequency offset is averaged so that fine frequency compensation is allowed to lock/converge. Hence, the coarse frequency offset is estimated using a **comm.CoarseFrequencyCompensator** System object and an averaging formula; the compensation is performed using a **comm.PhaseFrequencyOffset** System object.

3) Timing recovery: Performs timing recovery with closed-loop scalar processing to overcome the effects of delay introduced by the channel, using a **comm.SymbolSynchronizer** System object. The object implements a PLL to correct the symbol timing error in the received signal. The rotationally-invariant Gardner timing error detector is chosen for the object in this example; thus, timing recovery can precede fine frequency compensation. The input to the object is a fixed-length frame of samples. The output of the object is a frame of symbols whose length can vary due to bit stuffing and stripping, depending on actual channel delays.

4) Fine frequency compensation: Performs closed-loop scalar processing and compensates for the frequency offset accurately, using a **comm.CarrierSynchronizer** System object. The object implements a phase-locked loop (PLL) to track the residual frequency offset and the phase offset in the input signal.

5) Preamble Detection: Detects the location of the known Barker code in the input using a **comm.PreambleDetector** System object. The object implements a cross-correlation based algorithm to detect a known sequence of symbols in the input.

6) Frame Synchronization: Performs frame synchronization and, also, converts the variable-length symbol inputs into fixed-length outputs, using a **FrameSynchronizer** System object. The object has a secondary output that is a boolean scalar indicating if the first frame output is valid.

7) Data decoder: Performs phase ambiguity resolution and demodulation. Also, the data decoder compares the regenerated message with the transmitted one and calculates the BER.

Scopes

This component provides optional visualization to plot:

- A spectrum scope depicting the received signal before and after square root raised cosine filtering,
- Constellation diagrams showing the received signal after receiver filtering, after timing recovery and then after fine frequency compensation.

For more information about the system components, refer to the “QPSK Transmitter and Receiver in Simulink” on page 8-262 Simulink example.

System Under Test

The main loop in the system under test script processes the data frame-by-frame. Set the MATLAB variable `compileIt` to true in order to generate code. This can be accomplished by using the **codegen** command provided by the MATLAB Coder™ product. The **codegen** command translates MATLAB® functions to a MEX file, producing code for accelerated execution. The generated C code runs several times faster than the original MATLAB code. For this example, set `useCodegen` to true to use the code generated by **codegen** instead of the MATLAB code.

The inner loop of runQPSKSystemUnderTest.m uses the four System objects previously mentioned. In this file, there is a for-loop around the system under test to process one frame at a time.

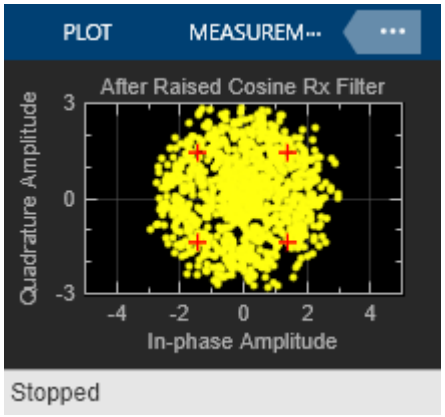
```
for count = 1:prmQPSKTxRx.FrameCount
    transmittedSignal = qpskTx(); rcvdSignal =
    qpskChan(transmittedSignal, count); [RCRxSignal, timingRecSignal,
    freqRecSignal, BER] = qpskRx(rcvdSignal); % Receiver if useScopes
      runQPSKScopes(qpskScopes, rcvdSignal, RCRxSignal, timingRecSignal,
      freqRecSignal); % Plots all the scopes
    end
end
```

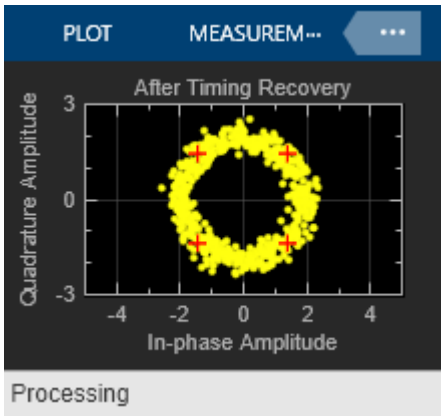
Execution and Results

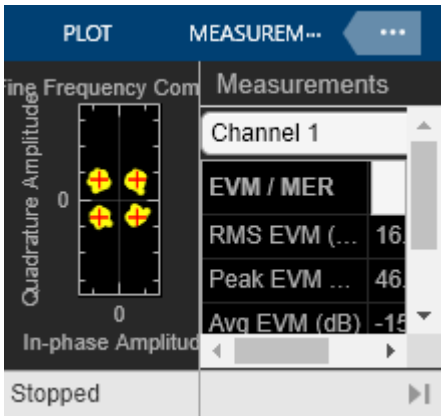
To run the System Under Test script and obtain BER values for the simulated QPSK communication, the following code is executed. When you run the simulations, it displays the bit error rate data, and some graphical results. The displayed scopes are constellation diagrams of the **Raised Cosine Receive Filter** output, the **Symbol Synchronizer** output, and the **Fine Frequency Compensation** output, and the power spectrum of the **Raised Cosine Receive Filter** output.

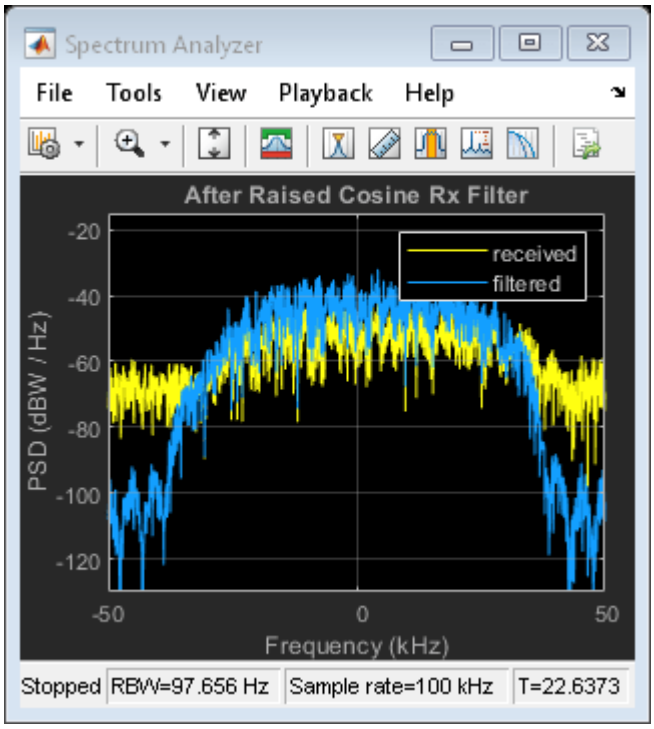
```
if compileIt
    codegen -report runQPSKSystemUnderTest.m -args {coder.Constant(prmQPSKTxRx),coder.Constant(us
end
if useCodegen
    BER = runQPSKSystemUnderTest_mex(prmQPSKTxRx,useScopes,printReceivedData);
else
    BER = runQPSKSystemUnderTest(prmQPSKTxRx,useScopes,printReceivedData);
end
fprintf('Error rate = %f.\n',BER(1));
fprintf('Number of detected errors = %d.\n',BER(2));
fprintf('Total number of compared samples = %d.\n',BER(3));
```

```
Error rate = 0.000238.
Number of detected errors = 366.
Total number of compared samples = 1536920.
```


PLOT
MEASUREM...
After Raised Cosine Rx Filter
Quadrature Amplitude
3
0
-3
-4
-2
0
2
4
In-phase Amplitude
Stopped


PLOT
MEASUREM...
After Timing Recovery
Quadrature Amplitude
3
0
-3
-4
-2
0
2
4
In-phase Amplitude
Processing


PLOT
MEASUREM...
Frequency Com
Measurements
Channel 1
EVM / MER
RMS EVM (...
Peak EVM ...
Avg EVM (dB)
Quadrature Amplitude
0
0
In-phase Amplitud
Stopped



Alternate Execution Options

As already mentioned in the section **System Under Test**, by using the variables at the beginning of the example, it is possible to interact with the code to explore different aspects of System objects and coding options.

By default, the variables `useScopes` and `printReceivedData` are set to true and false, respectively. The `useScopes` variable enables MATLAB scopes to be opened during the example execution. Using the scopes, you can see how the simulated subcomponents behave and also obtain a better understanding of how the system functions in simulation time. When you set this variable to false, the scopes will not open during the example execution. When you set `printReceivedData` to true, you can also see the decoded received packets printed in the command window. The other two variables, compileIt and `useCodegen`, are related to speed performance and can be used to analyze design tradeoffs.

When you set compileIt to true, this example script will use MATLAB Coder™ capabilities to compile the script runQPSKSystemUnderText for accelerated execution. This command will create a MEX file (runQPSKSystemUnderTest_mex) and save it in the current folder. Once you set useCodegen to true to run the mex file, the example is able to run the system implemented in MATLAB much faster. This feature is essential for implementation of real-time systems and is an important simulation tool. To maximize simulation speed, set `useScopes` to false and `useCodegen` to true to run the mex file.

For other exploration options, refer to the “QPSK Transmitter and Receiver in Simulink” on page 8-262 example.

Summary

This example simulates digital communication over an AWGN channel. It shows how to model several parts of the QPSK system such as modulation, frequency and phase recovery, timing recovery, and

frame synchronization. It measures the system performance by calculating BER. It also shows that the generated C code runs several times faster than the original MATLAB code.

Appendix

This example uses the following script and helper functions:

- runQPSKSystemUnderTest.m
- QPSKTransmitter.m
- QPSKChannel.m
- QPSKReceiver.m
- QPSKScopes.m
- QPSKBitsGenerator.m
- QPSKDataDecoder.m
- FrameSynchronizer.m

References

1. Rice, Michael. *Digital Communications - A Discrete-Time Approach*. 1st ed. New York, NY: Prentice Hall, 2008.

QPSK Transmitter and Receiver in Simulink

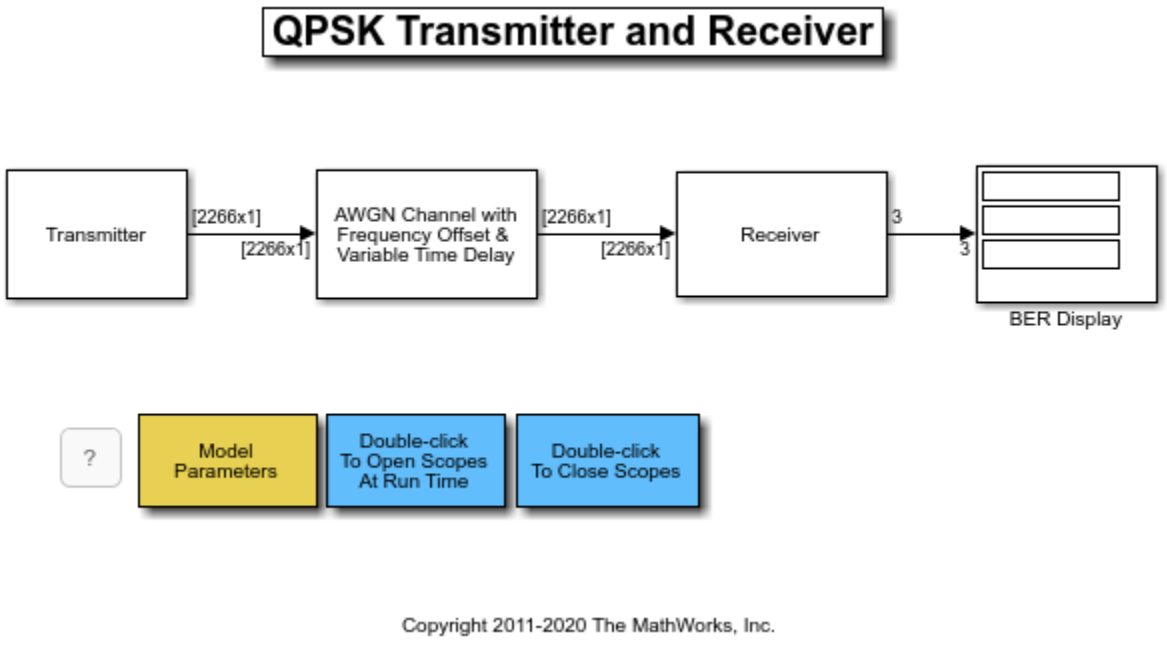
This model shows the implementation of a QPSK transmitter and receiver with Simulink®. The receiver addresses practical issues in wireless communications, such as carrier frequency and phase offset, timing drift and frame synchronization. The receiver demodulates the received symbols and outputs a simple message to the Diagnostic Viewer. For the MATLAB® implementation of the same system, refer to the “QPSK Transmitter and Receiver” on page 8-255.

Overview

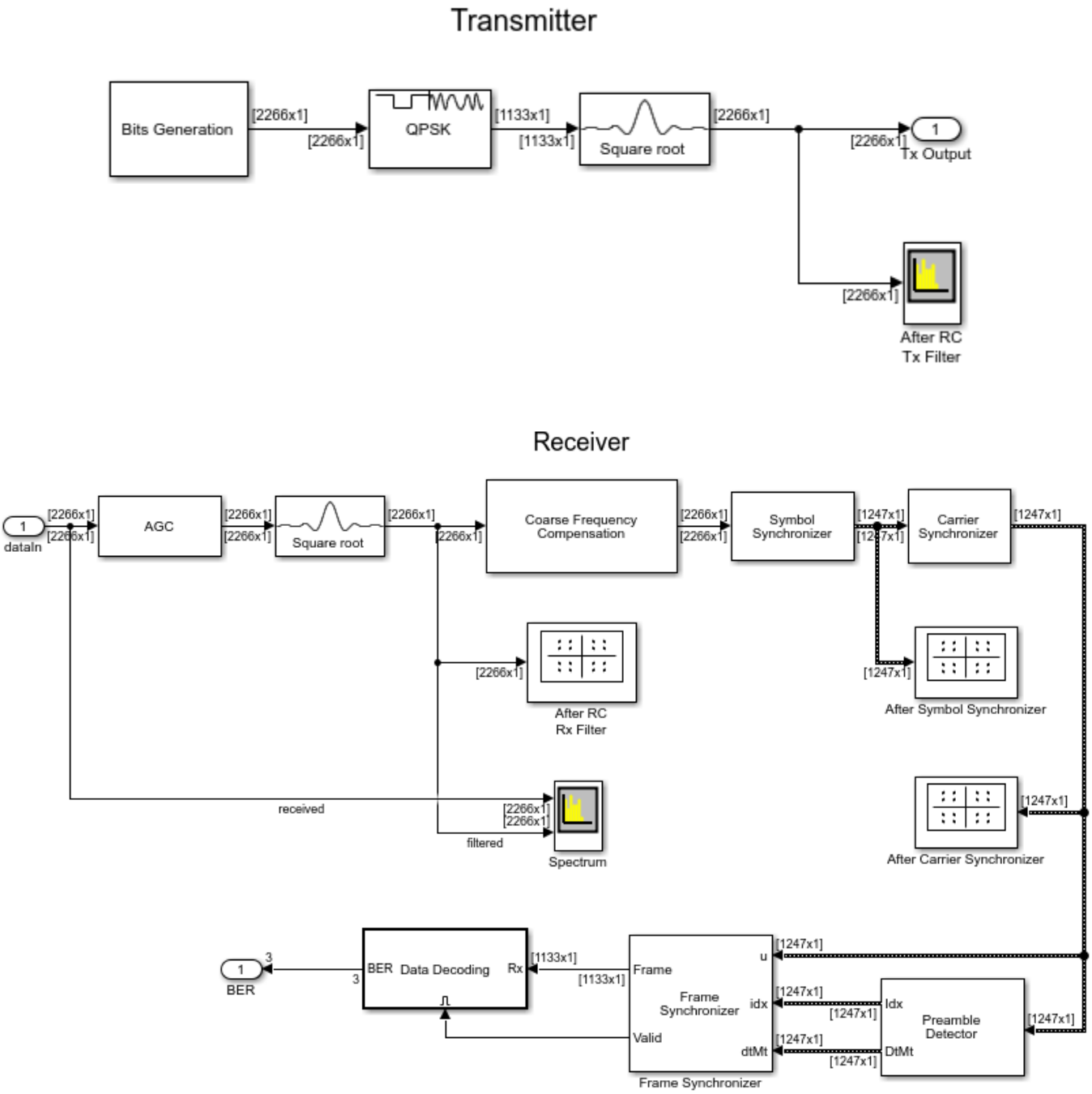
This example model performs all processing at complex baseband to handle a static frequency offset, a timing drift, and Gaussian noise. To cope with the above-mentioned impairments, this example provides a reference design of a practical digital receiver, which includes correlation-based coarse frequency compensation, PLL-based fine frequency compensation, PLL-based symbol timing recovery, frame synchronization, and phase ambiguity resolution. The example showcases a few library blocks in Communications Toolbox™ that implement synchronization algorithms in the receiver processing.

Structure of the Example

The top-level structure of the model is shown in the following figure, which includes the **Transmitter** subsystem, the channel subsystem, and the **Receiver** subsystem.



The detailed structures of the **Transmitter** subsystem and the **Receiver** subsystem are illustrated in the following figures.



The components are further described in the following sections.

Transmitter

- **Bit Generation** - Generates the bits for each frame
- **QPSK Modulator** - Modulates the bits into QPSK symbols
- **Raised Cosine Transmit Filter** - Uses a rolloff factor of 0.5, and upsamples the QPSK symbols by two

Channel

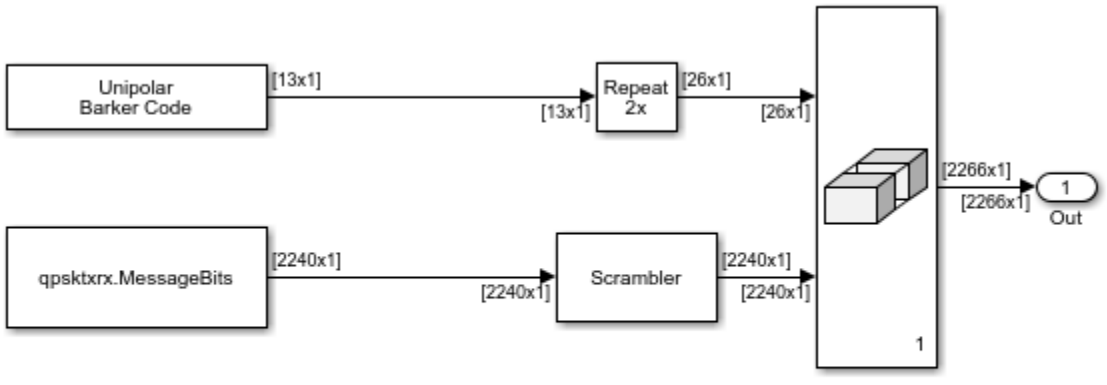
- **AWGN Channel with Frequency Offset and Variable Time Delay** - Applies the frequency offset, a timing drift, and additive white Gaussian noise to the signal

Receiver

- **Raised Cosine Receive Filter** - Uses a rolloff factor of 0.5
- **Coarse Frequency Compensation** - Estimates an approximate frequency offset of the received signal and corrects it
- **Symbol Synchronizer** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants
- **Carrier Synchronizer** - Compensates for the residual frequency offset and the phase offset
- **Preamble Detector** - Detect location of the frame header
- **Frame Synchronizer** - Aligns the frame boundaries at the known frame header
- **Data Decoding** - Resolves the phase ambiguity caused by the **Carrier Synchronizer**, demodulates the signal, and decodes the text message

Transmitter

The transmitter includes the **Bit Generation** subsystem, the **QPSK Modulator** block, and the **Raised Cosine Transmit Filter** block. The **Bit Generation** subsystem uses a MATLAB workspace variable as the payload of a frame, as shown in the figure below. Each frame contains 20 'Hello world ###' messages and a header. The first 26 bits are header bits, a 13-bit Barker code that has been oversampled by two. The Barker code is oversampled by two in order to generate precisely 13 QPSK symbols for later use in the **Data Decoding** subsystem of the receiver model. The remaining bits are the payload. The payload correspond to the ASCII representation of 'Hello world ###', where '###' is a repeating sequence of '000', '001', '002', ..., '099'. The payload is scrambled to guarantee a balanced distribution of zeros and ones for the timing recovery operation in the receiver model. The scrambled bits are modulated by the **QPSK Modulator** (with Gray mapping). The modulated symbols are upsampled by two by the **Raised Cosine Transmit Filter** with a roll-off factor 0.5. The symbol rate of the transmitter system is 50k symbols per second, and the sample rate after the **Raised Cosine Transmit Filter** is 100k samples per second.

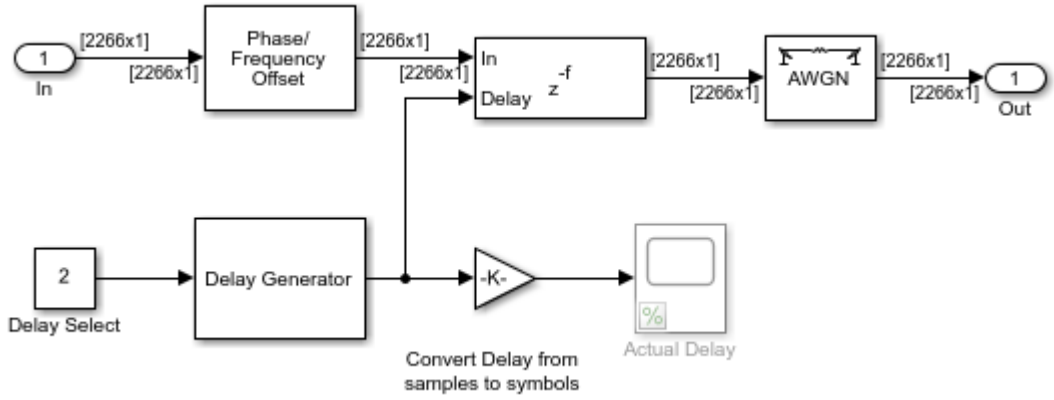


AWGN Channel with Frequency Offset and Variable Delay

The **AWGN Channel with Frequency Offset and Variable Delay** subsystem first applies the frequency offset and a preset phase offset to the transmit signal. Then it adds a variable delay with a choice of the following two types of delay to the signal:

- **Ramp delay** - This type of delay is initialized at *DelayStart* samples, and increases linearly at a rate of *DelayStep* samples in each frame. When the actual delay reaches one frame, the delay buffer is full, and it maintains a delay of one frame.
- **Triangle delay** - This type of delay linearly changes back and forth between *MinDelay* samples and *MaxDelay* samples at a rate of *DelayStep* samples in each frame

The use of multiple delay characteristics allows you to investigate their effects on receiver performance, particularly on the **Symbol Synchronizer** block. The delayed signal is processed through an **AWGN Channel**. The diagram of the **AWGN Channel with Frequency Offset and Variable Delay** subsystem is as shown in the following.



Receiver

Raised Cosine Receive Filter

The **Raised Cosine Receive Filter** provides matched filtering for the transmitted waveform with a rolloff factor of 0.5.

AGC

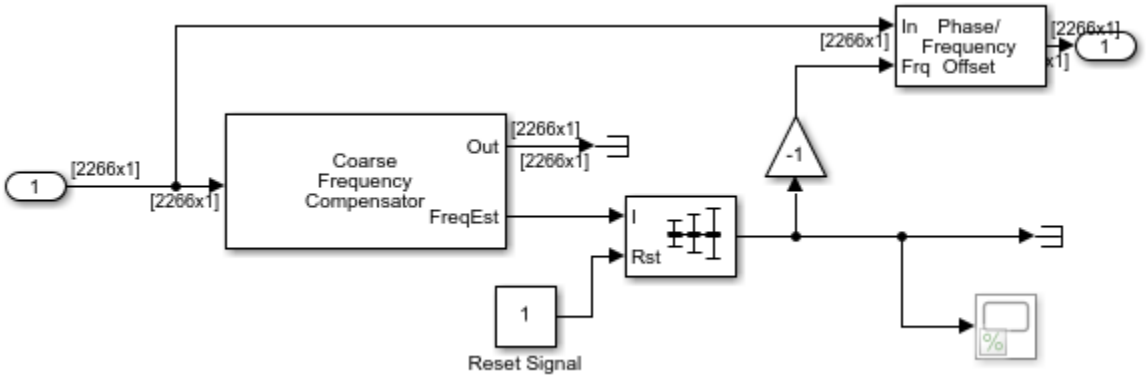
The received signal amplitude affects the accuracy of the carrier and symbol synchronizer. Therefore the signal amplitude should be stabilized to ensure an optimum loop design. The AGC output power is set to a value ensuring that the equivalent gains of the phase and timing error detectors keep constant over time. The AGC is placed before the **Raised Cosine Receive Filter** so that the signal amplitude can be measured with an oversampling factor of two, thus improving the accuracy of the estimate. You can refer to Chapter 7.2.2 and Chapter 8.4.1 of [1] for details on how to design the phase detector gain.

Coarse Frequency Compensation

The **Coarse Frequency Compensation** subsystem corrects the input signal with a rough estimate of the frequency offset. The following diagram shows the subsystem, in which the frequency offset is estimated by averaging the output of the correlation-based algorithm of the **Coarse Frequency**

Compensator block. The compensation is performed by the Phase/Frequency Offset block. There is usually a residual frequency offset even after the coarse frequency compensation, which would cause a slow rotation of the constellation. The **Carrier Synchronizer** block compensates for this residual frequency.

The accuracy of the **Coarse Frequency Compensator** decreases with its maximum frequency offset value. Ideally, this value should be set just above the expected frequency offset range. For example, this model introduces a 5 kHz frequency offset and the **Coarse Frequency Compensator** is configured with a 6 kHz maximum frequency offset.



Symbol Synchronizer

The timing recovery is performed by a **Symbol Synchronizer** library block, which implements a PLL, described in Chapter 8 of [1], to correct the timing error in the received signal. The timing error detector is estimated using the Gardner algorithm, which is rotationally invariant. In other words, this algorithm can be used before or after frequency offset compensation. The input to the block is oversampled by two. On average, the block generates one output symbol for every two input samples. However, when the channel timing error (delay) reaches symbol boundaries, there will be one extra or missing symbol in the output frame. In that case, the block implements bit stuffing/skipping thus the output of this block is a variable-size signal.

The *Damping factor*, *Normalized loop bandwidth*, and *Detector gain* parameters of the block are tunable. Their default values are set to 1 (critical damping), 0.01 and 5.4 respectively, so that the PLL quickly locks to the correct timing while introducing little timing jitter.

Carrier Synchronizer

The fine frequency compensation is performed by a **Carrier Synchronizer** library block, which implements a phase-locked loop (PLL), described in Chapter 7 of [1], to track the residual frequency offset and the phase offset in the input signal. The PLL uses a Direct Digital Synthesizer (DDS) to generate the compensating phase that offsets the residual frequency and phase offsets. The phase offset estimate from DDS is the integral of the phase error output of a Loop Filter.

The *Damping factor* and *Normalized loop bandwidth* parameters of the block are tunable. Their default values are set to 1 (critical damping) and 0.01 respectively, so that the PLL quickly locks to the intended phase while introducing little phase noise.

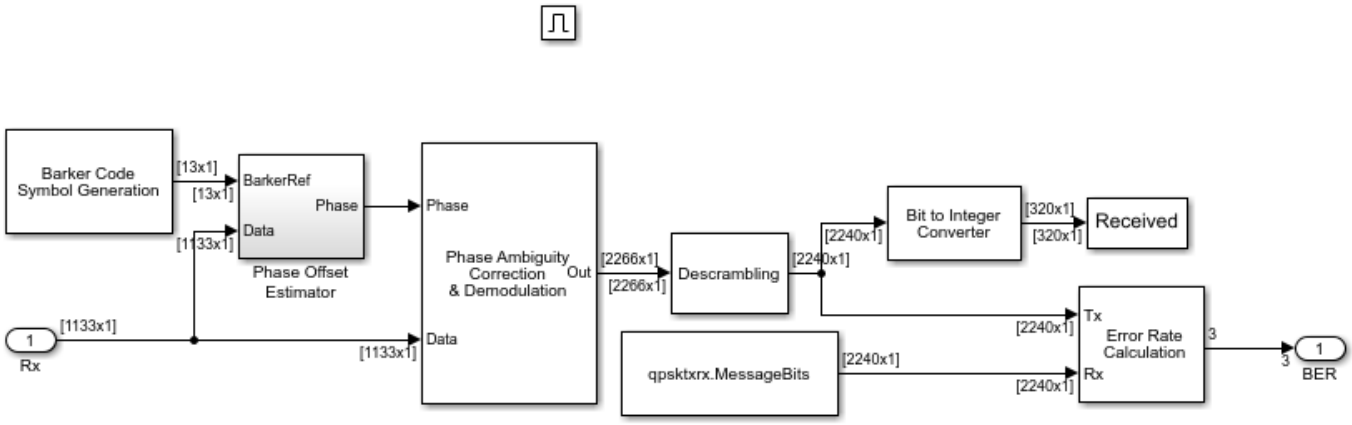
Preamble Detector and Frame Synchronizer

The location of the known frame header is detected by a **Preamble Detector** library block and the frame synchronization is performed by a MATLAB System block using a **FrameSynchronizer** System

object™. The Preamble Detector block uses the known frame header (QPSK-modulated Barker code) to correlate against the received QPSK symbols in order to find the location of the frame header. The Frame Synchronizer block uses this location information to align the frame boundaries. It also transforms the variable-size output of the **Symbol Synchronizer** block into a fixed-size frame, which is necessary for the downstream processing. The second output of the block is a boolean scalar indicating if the first output is a valid frame with the desired header and if so, enables the **Data Decoding** subsystem to run.

Data Decoding

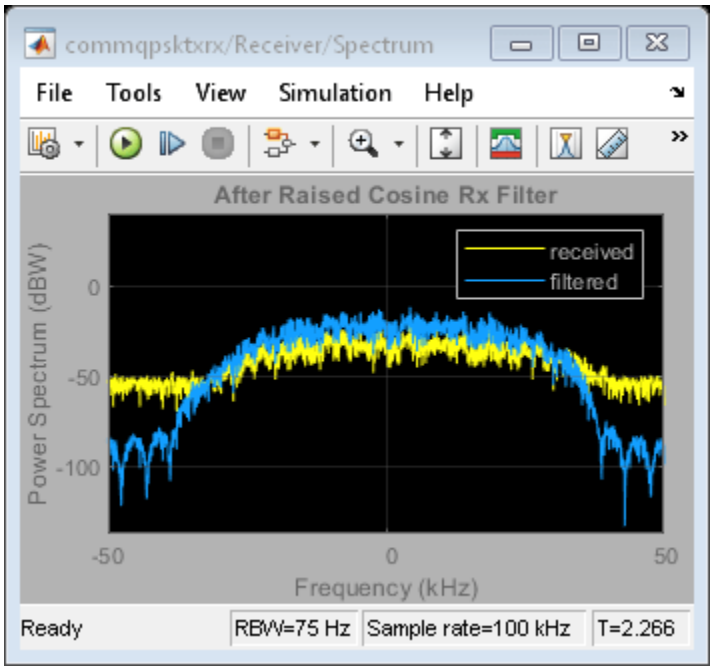
The **Data Decoding** enabled subsystem performs phase ambiguity resolution, demodulation and text message decoding. The **Carrier Synchronizer** block may lock to the unmodulated carrier with a phase shift of 0, 90, 180, or 270 degrees, which can cause a phase ambiguity. For details of phase ambiguity and its resolution, please refer to Chapter 7.2.2 and 7.7 in [1]. The **Phase Offset Estimator** subsystem determines this phase shift. The **Phase Ambiguity Correction & Demodulation** subsystem rotates the input signal by the estimated phase offset and demodulates the corrected data. The payload bits are descrambled, and printed out to the Simulink Diagnostic Viewer at the end of the simulation.

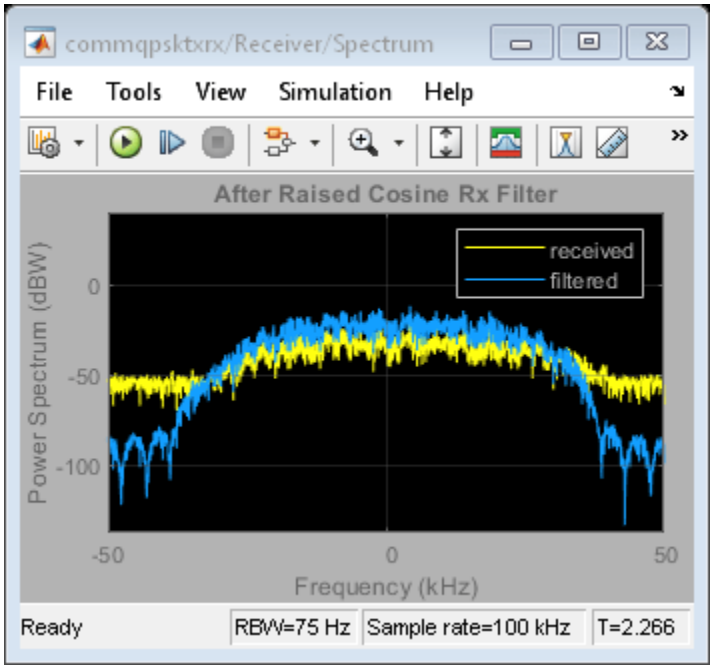


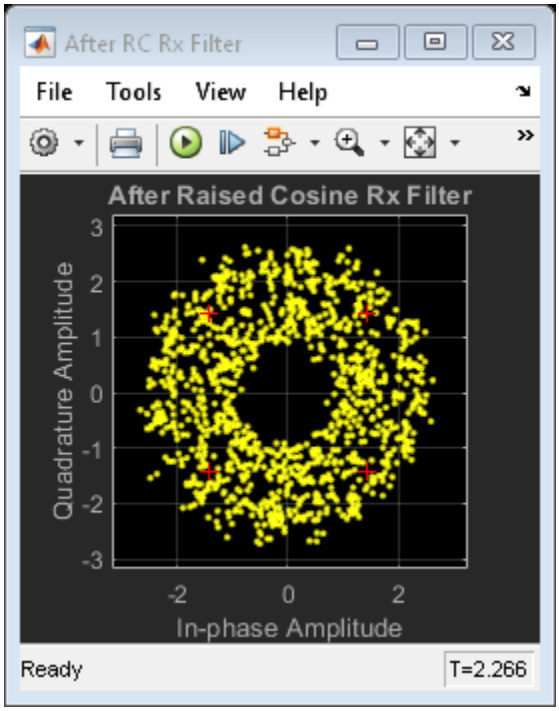
Results and Displays

When you run the simulation, it displays bit error rate and numerous graphical results.

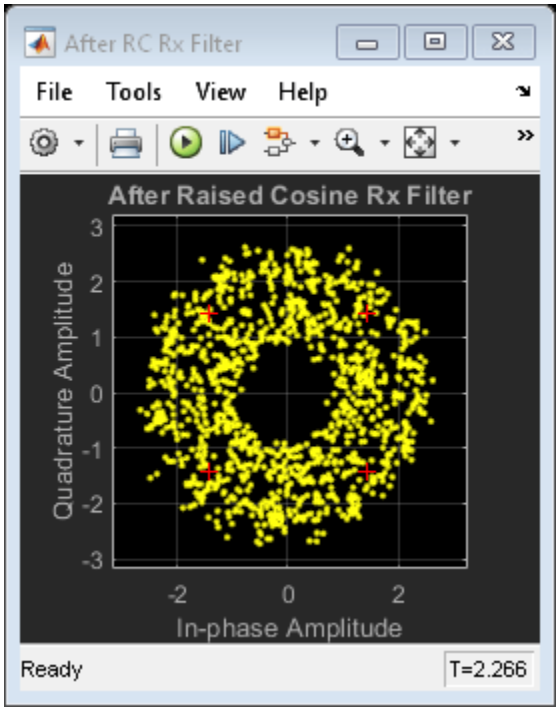
These following scopes illustrate the spectrum of the received signal before and after filtering, as well as the signal constellation after filtering, after timing recovery and after fine frequency compensation.

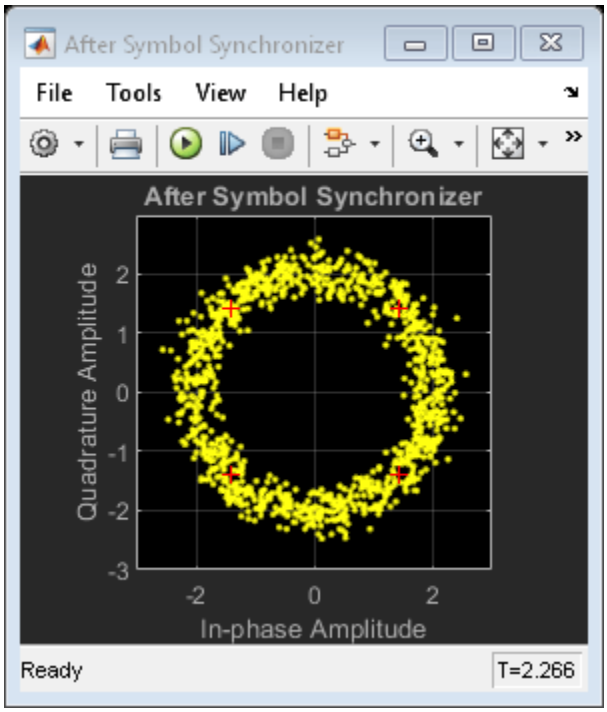

commqpsktxrx/Receiver/Spectrum
File Tools View Simulation Help
After Raised Cosine Rx Filter
received
filtered
Power Spectrum (dBW)
0
-50
-100
-50
0
50
Frequency (kHz)
Ready
RBW=75 Hz
Sample rate=100 kHz
T=2.266

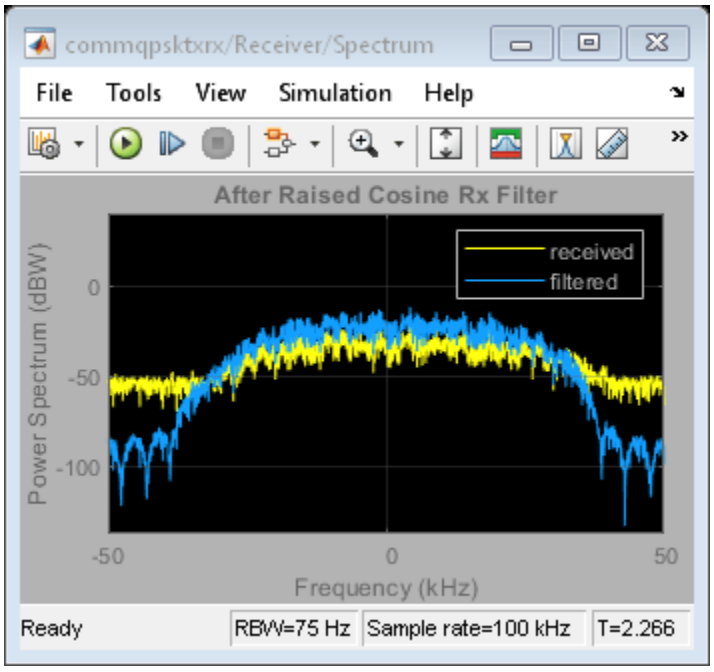

commqpsktxrx/Receiver/Spectrum
File Tools View Simulation Help
After Raised Cosine Rx Filter
received
filtered
Power Spectrum (dBW)
0
-50
-100
-50
0
50
Frequency (kHz)
Ready
RBW=75 Hz
Sample rate=100 kHz
T=2.266

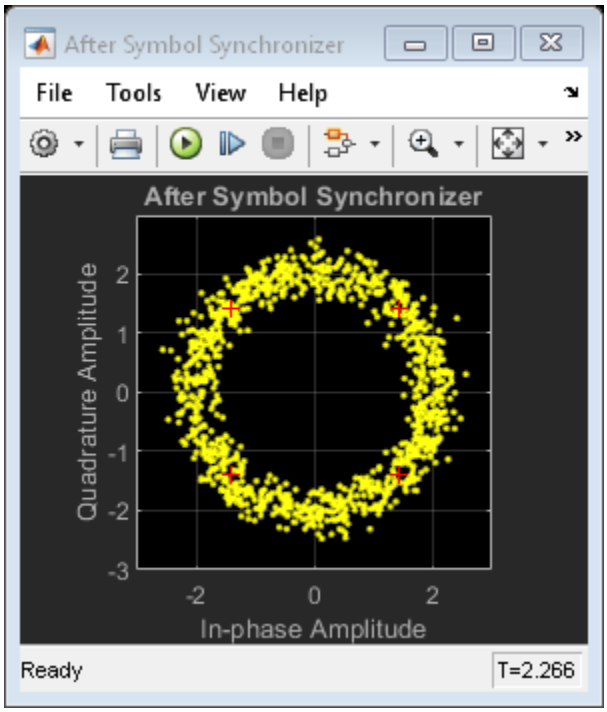


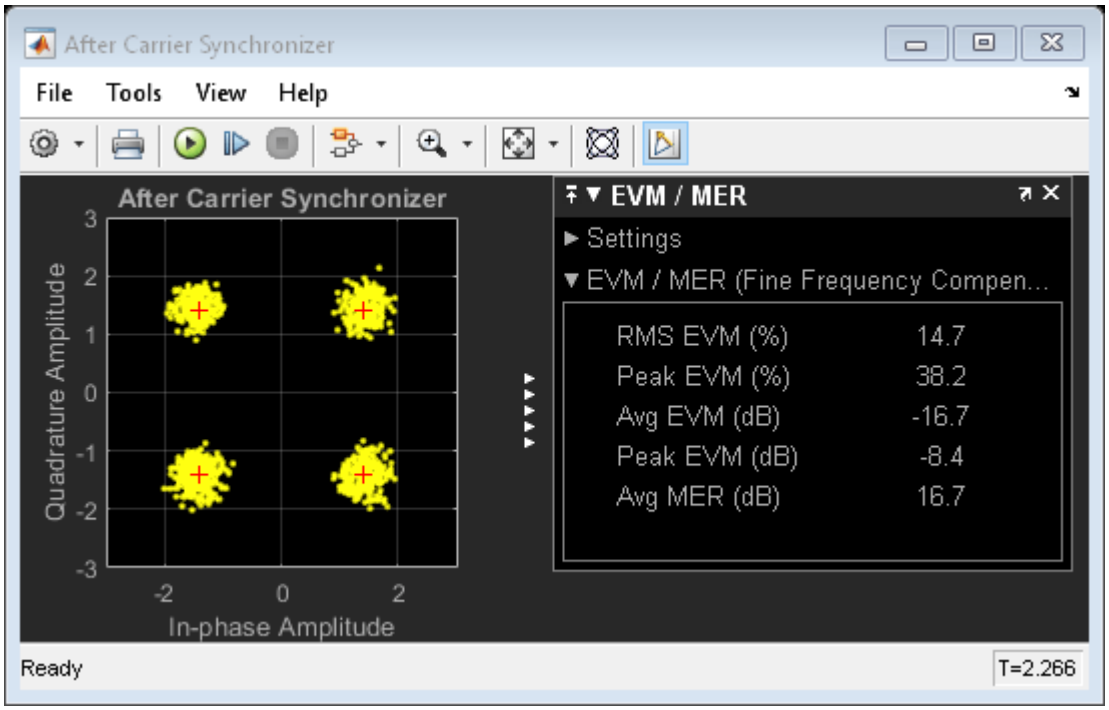
In the following are the constellation diagrams at the output of the **Symbol Synchronizer** and **Carrier Synchronizer** blocks respectively.

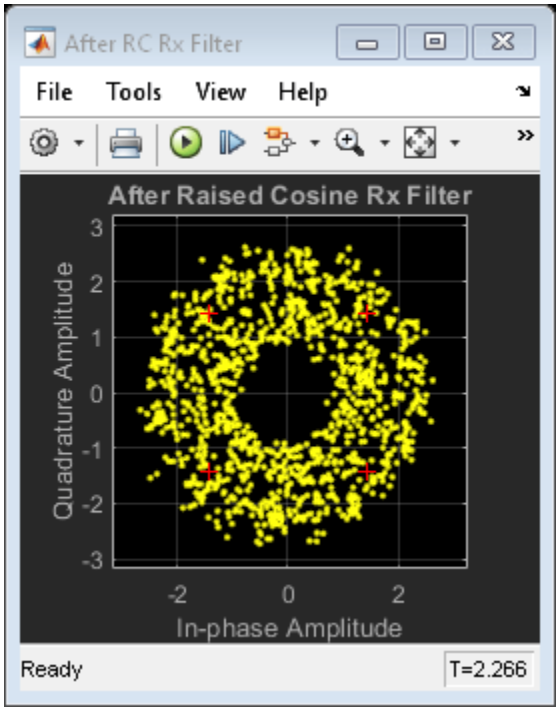


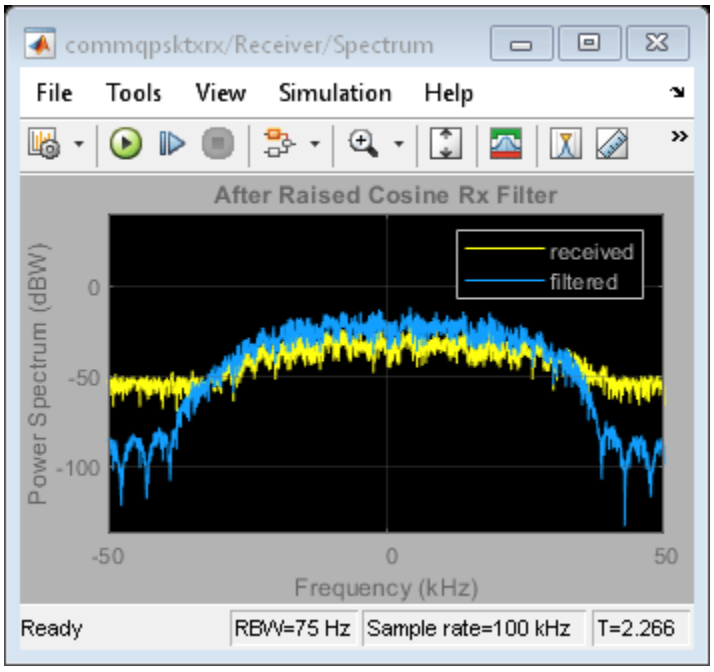

After Symbol Synchronizer
File Tools View Help
After Symbol Synchronizer
Quadrature Amplitude
2
1
0
-1
-2
-3
-2
0
2
In-phase Amplitude
Ready
T=2.266


commqpsktxrx/Receiver/Spectrum
File Tools View Simulation Help
After Raised Cosine Rx Filter
received
filtered
Power Spectrum (dBW)
0
-50
-100
-50
0
50
Frequency (kHz)
Ready
RBW=75 Hz
Sample rate=100 kHz
T=2.266


After Symbol Synchronizer
File
Tools
View
Help
After Symbol Synchronizer
Quadrature Amplitude
2
1
0
-1
-2
-3
-2
0
2
In-phase Amplitude
Ready
T=2.266


After Carrier Synchronizer
File
Tools
View
Help
After Carrier Synchronizer
Quadrature Amplitude
3
2
1
0
-1
-2
-3
-2
0
2
In-phase Amplitude
EVM / MER
Settings
EVM / MER (Fine Frequency Compen...
RMS EVM (%) 14.7
Peak EVM (%) 38.2
Avg EVM (dB) -16.7
Peak EVM (dB) -8.4
Avg MER (dB) 16.7
Ready
T=2.266





Exploring the Example

The example allows you to experiment with multiple system capabilities to examine their effect on bit error rate performance. For example, you can view the effect of changing the frequency offset, delay type and E_b/N_0 on the various displays.

This example models a static frequency offset. In practice, the frequency offset might vary over time. This model can still track a time-varying frequency drift via the **Coarse Frequency Compensation** subsystem. If the actual frequency offset exceeds the maximum frequency offset that can be tracked by the current coarse frequency compensation subsystem, you can increase its tracking range by increasing the oversampling factor. Alternatively, you can change the algorithm from correlation-based to FFT-based, in the **Model Parameters** block. The FFT-based algorithm performs better than the correlation-based algorithm at low Eb/No.

You can also tune the *Normalized loop bandwidth* and *Damping factor* parameters of the **Symbol Synchronizer** and **Carrier Synchronizer** blocks, to assess their convergence time and estimation accuracy. In addition, you can assess the pull-in range of the **Carrier Synchronizer** block. With a large *Normalized loop bandwidth* and *Damping factor*, the PLL can acquire over a greater frequency offset range. However a large *Normalized loop bandwidth* allows more noise, which leads to a large mean squared error in the phase estimation. "Underdamped systems (with Damping Factor less than one) have a fast settling time, but exhibit overshoot and oscillation; overdamped systems (with Damping Factor greater than one) have a slow settling time but no oscillations." [1]. For more detail on the design of these PLL parameters, you can refer to Appendix C in [1].

References

1. Michael Rice, "Digital Communications - A Discrete-Time Approach", Prentice Hall, April 2008.

Raised Cosine Filtering

This example shows the intersymbol interference (ISI) rejection capability of the raised cosine filter, and how to split the raised cosine filtering between transmitter and receiver, using raised cosine transmit and receive filter System objects (comm.RaisedCosineTransmitFilter and comm.RaisedCosineReceiveFilter, respectively).

Raised Cosine Filter Specifications

The main parameter of a raised cosine filter is its roll-off factor, which indirectly specifies the bandwidth of the filter. Ideal raised cosine filters have an infinite number of taps. Therefore, practical raised cosine filters are windowed. The window length is controlled using the `FilterSpanInSymbols` property. In this example, we specify the window length as six symbol durations, i.e., the filter spans six symbol durations. Such a filter also has a group delay of three symbol durations. Raised cosine filters are used for pulse shaping, where the signal is upsampled. Therefore, we also need to specify the upsampling factor. The following is a list of parameters used to design the raised cosine filter for this example.

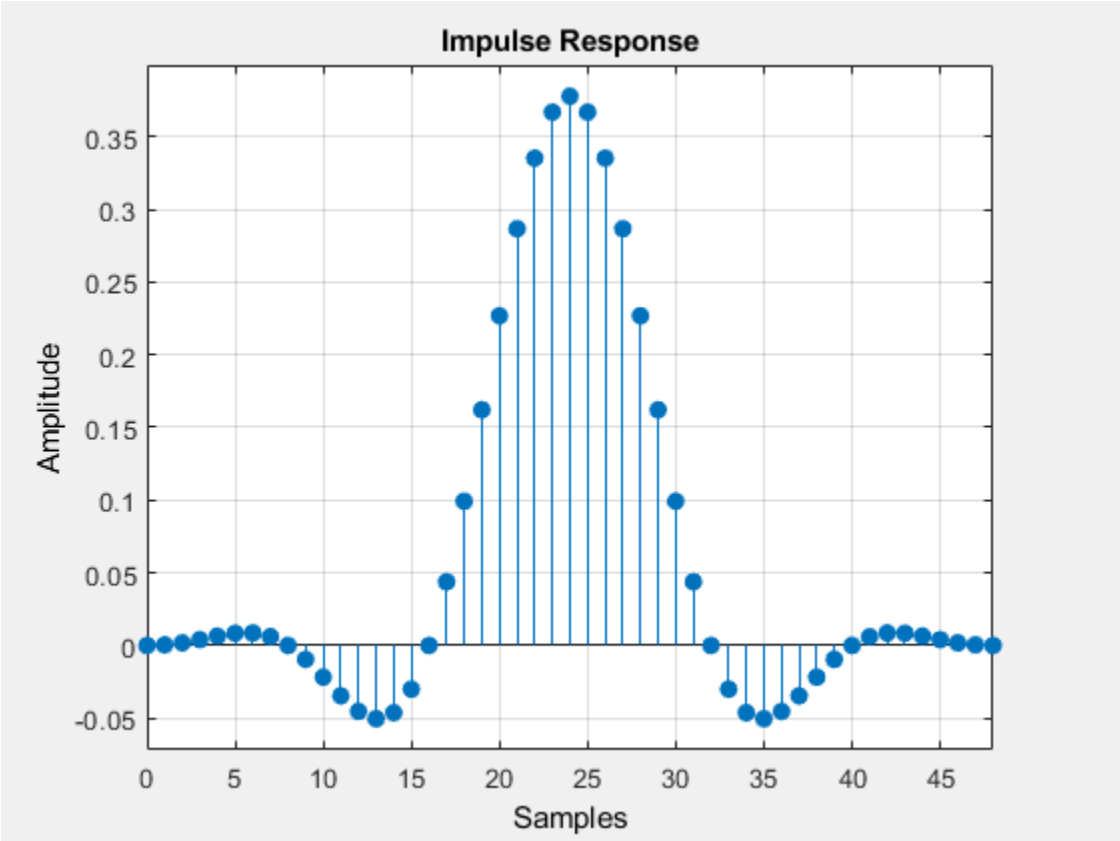
```
Nsym = 6;           % Filter span in symbol durations
beta = 0.5;         % Roll-off factor
sampsPerSym = 8;    % Upsampling factor
```

We use a raised cosine transmit filter System object and set its properties to obtain the desired filter characteristics. We also use `fvtool` to visualize filter characteristics.

```
rctFilt = comm.RaisedCosineTransmitFilter(...
  'Shape','Normal', ...
  'RolloffFactor',beta, ...
  'FilterSpanInSymbols',Nsym, ...
  'OutputSamplesPerSymbol',sampsPerSym)
```

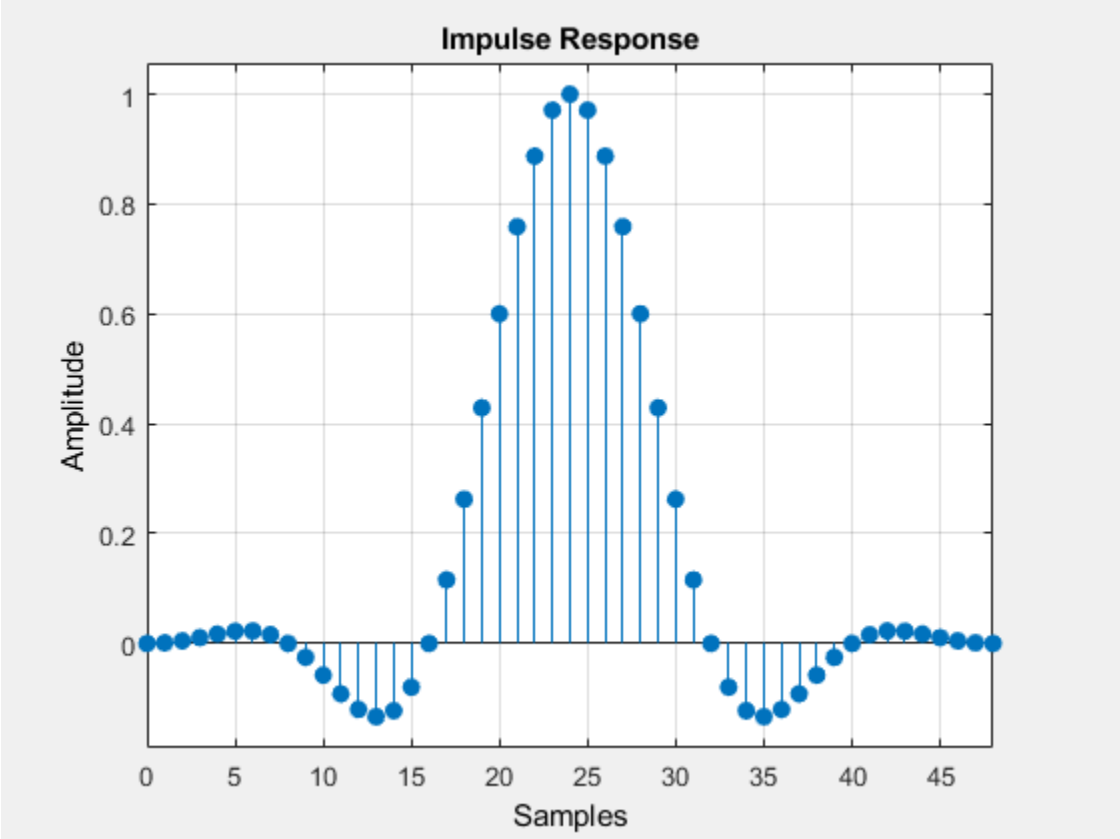
```
rctFilt =
  comm.RaisedCosineTransmitFilter with properties:

                     Shape: 'Normal'
              RolloffFactor: 0.5000
        FilterSpanInSymbols: 6
      OutputSamplesPerSymbol: 8
                      Gain: 1
```

```
% Visualize the impulse response
fvtool(rctFilt,'Analysis','impulse')
```



This object designs a direct-form polyphase FIR filter with unit energy. The filter has an order of Nsym*sampsPerSym, or Nsym*sampsPerSym+1 taps. You can utilize the Gain property to normalize the filter coefficients so that the filtered and unfiltered data matches when overlayed.

```
% Normalize to obtain maximum filter tap value of 1
b = coeffs(rctFilt);
rctFilt.Gain = 1/max(b.Numerator);

% Visualize the impulse response
fvtool(rctFilt,'Analysis','impulse')
```



Pulse Shaping with Raised Cosine Filters

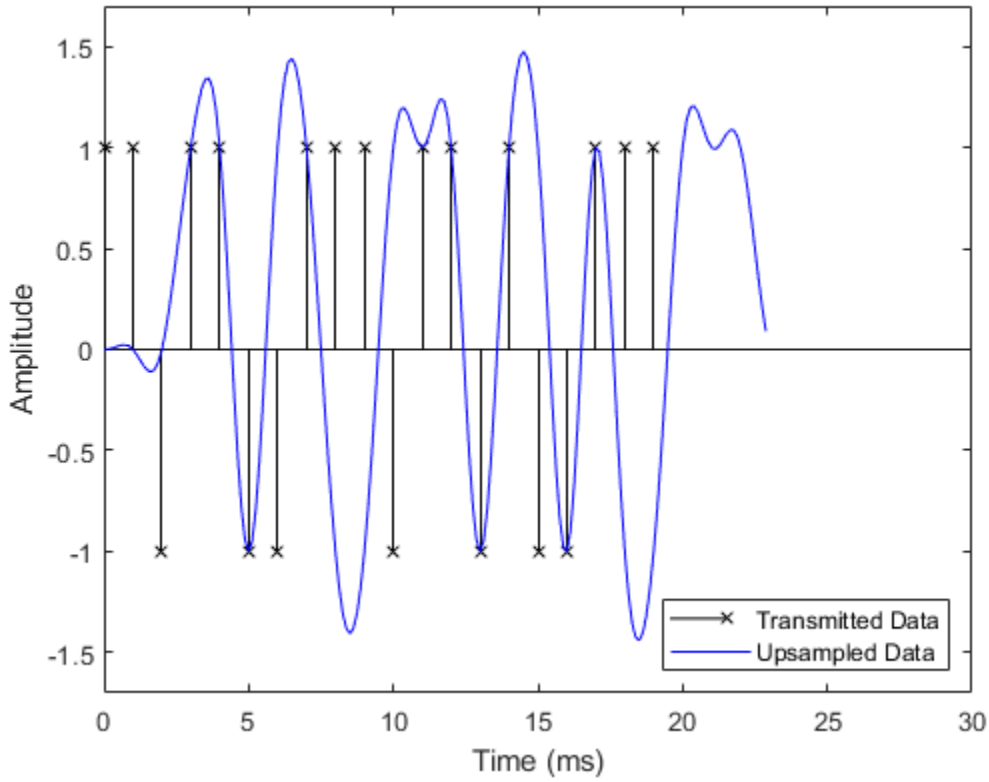
We generate a bipolar data sequence. We use the raised cosine filter to shape the waveform without introducing ISI.

```
% Parameters
DataL = 20;             % Data length in symbols
R = 1000;               % Data rate
Fs = R * sampsPerSym;   % Sampling frequency

% Create a local random stream to be used by random number generators for
% repeatability
hStr = RandStream('mt19937ar','Seed',0);

% Generate random data
x = 2*randi(hStr,[0 1],DataL,1)-1;
% Time vector sampled at symbol rate in milliseconds
tx = 1000 * (0: DataL - 1) / R;
```

The plot compares the digital data and the interpolated signal. It is difficult to compare the two signals because the peak response of the filter is delayed by the group delay of the filter (Nsym/(2*R)). Note that, we append Nsym/2 zeros at the end of input `x` to flush all the useful samples out of the filter.

```
% Filter
yo = rctFilt([x; zeros(Nsym/2,1)]);
% Time vector sampled at sampling frequency in milliseconds
```

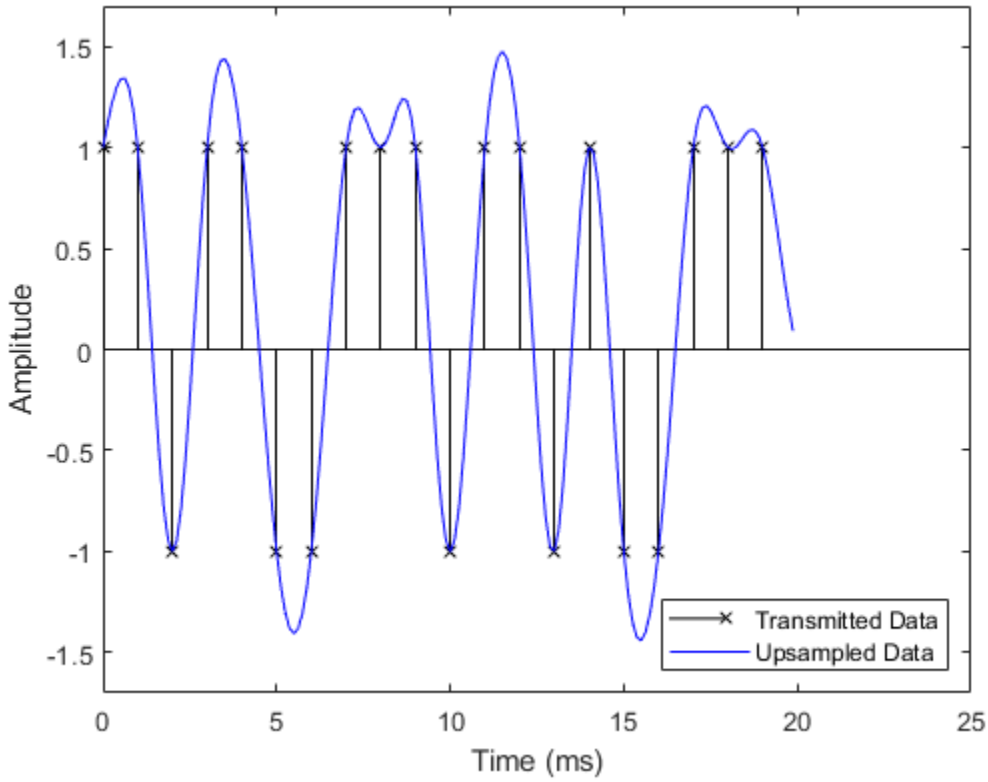
```
to = 1000 * (0: (DataL+Nsym/2)*sampsPerSym - 1) / Fs;
% Plot data
fig1 = figure;
stem(tx, x, 'kx'); hold on;
% Plot filtered data
plot(to, yo, 'b-'); hold off;
% Set axes and labels
axis([0 30 -1.7 1.7]);  xlabel('Time (ms)'); ylabel('Amplitude');
legend('Transmitted Data','Upsampled Data','Location','southeast')
```



This step compensates for the raised cosine filter group delay by delaying the input signal. Now it is easy to see how the raised cosine filter upsamples and filters the signal. The filtered signal is identical to the delayed input signal at the input sample times. This shows the raised cosine filter capability to band-limit the signal while avoiding ISI.

```
% Filter group delay, since raised cosine filter is linear phase and
% symmetric.
fltDelay = Nsym / (2*R);
% Correct for propagation delay by removing filter transients
yo = yo(fltDelay*Fs+1:end);
to = 1000 * (0: DataL*sampsPerSym - 1) / Fs;
% Plot data.
stem(tx, x, 'kx'); hold on;
% Plot filtered data.
plot(to, yo, 'b-'); hold off;
% Set axes and labels.
```

```
axis([0 25 -1.7 1.7]);  xlabel('Time (ms)'); ylabel('Amplitude');
legend('Transmitted Data','Upsampled Data','Location','southeast')
```

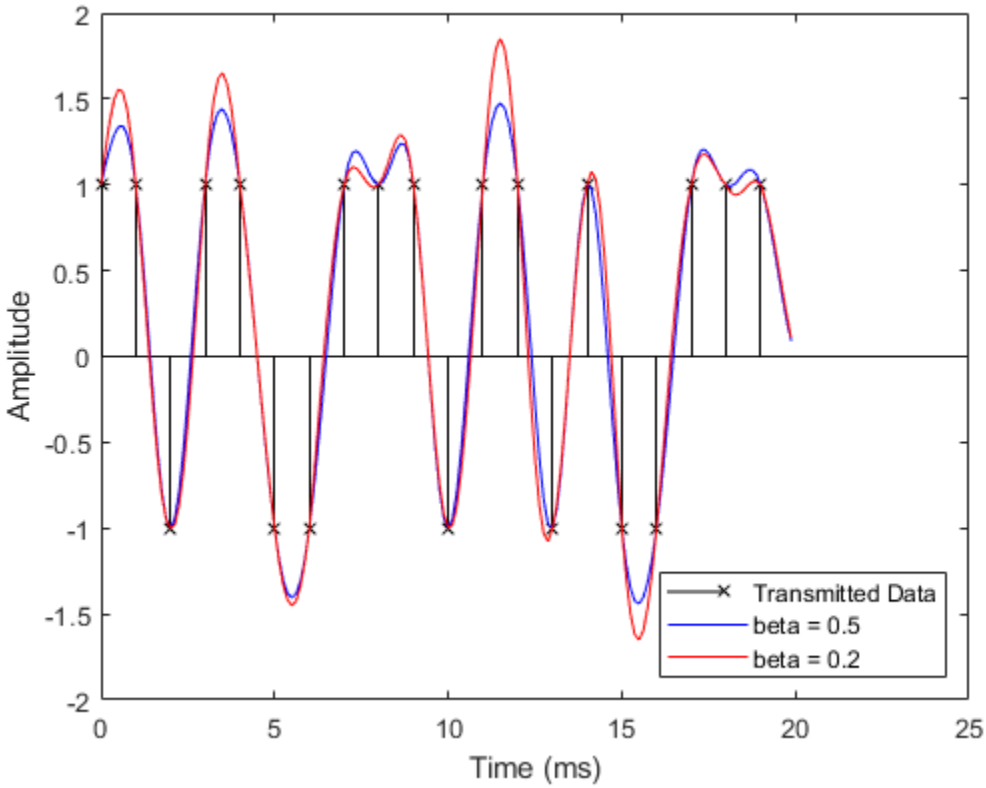


Roll-off Factor

This step shows the effect that changing the roll-off factor from .5 (blue curve) to .2 (red curve) has on the resulting filtered output. The lower value for roll-off causes the filter to have a narrower transition band causing the filtered signal overshoot to be greater for the red curve than for the blue curve.

```
% Set roll-off factor to 0.2
rctFilt2 = comm.RaisedCosineTransmitFilter(...
  'Shape',                  'Normal', ...
  'RolloffFactor',          0.2, ...
  'FilterSpanInSymbols',    Nsym, ...
  'OutputSamplesPerSymbol', sampsPerSym);
% Normalize filter
b = coeffs(rctFilt2);
rctFilt2.Gain = 1/max(b.Numerator);
% Filter
yo1 = rctFilt2([x; zeros(Nsym/2,1)]);
% Correct for propagation delay by removing filter transients
yo1 = yo1(fltDelay*Fs+1:end);
% Plot data
stem(tx, x, 'kx'); hold on;
% Plot filtered data
plot(to, yo, 'b-',to, yo1, 'r-'); hold off;
```

```
% Set axes and labels
axis([0 25 -2 2]);  xlabel('Time (ms)'); ylabel('Amplitude');
legend('Transmitted Data','beta = 0.5','beta = 0.2',...
    'Location','southeast')
```



Square-Root Raised Cosine Filters

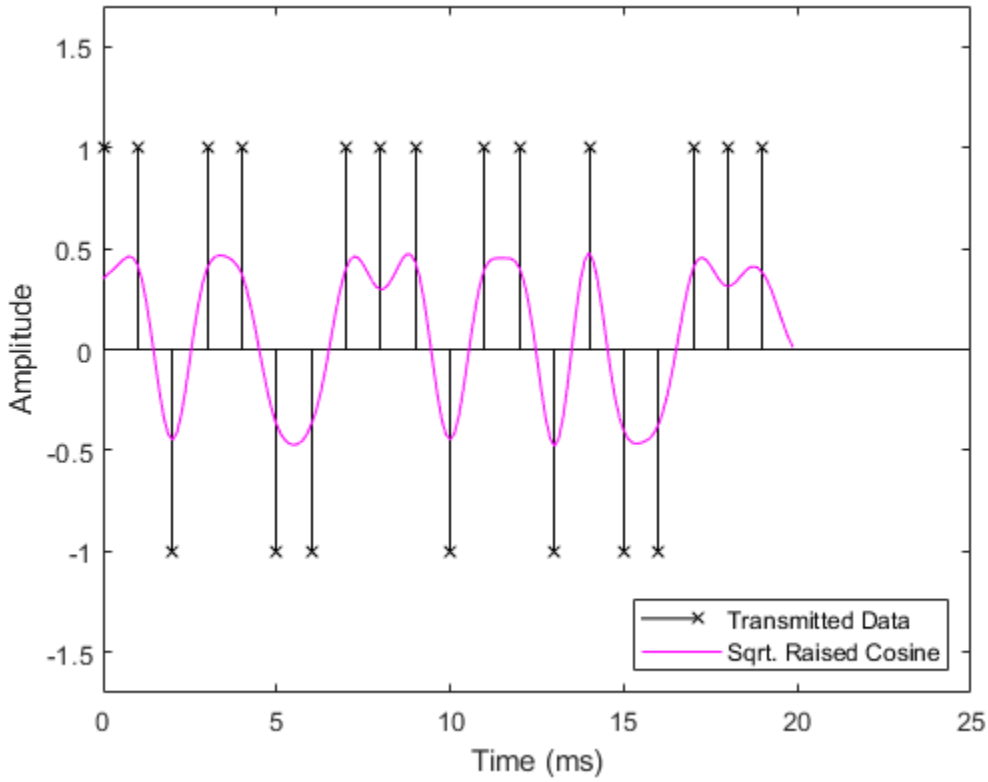
A typical use of raised cosine filtering is to split the filtering between transmitter and receiver. Both transmitter and receiver employ square-root raised cosine filters. The combination of transmitter and receiver filters is a raised cosine filter, which results in minimum ISI. We specify a square-root raised cosine filter by setting the shape as 'Square root'.

```
% Design raised cosine filter with given order in symbols
rctFilt3 = comm.RaisedCosineTransmitFilter(...
  'Shape',                 'Square root', ...
  'RolloffFactor',         beta, ...
  'FilterSpanInSymbols',   Nsym, ...
  'OutputSamplesPerSymbol', sampsPerSym);
```

The data stream is upsampled and filtered at the transmitter using the designed filter. This plot shows the transmitted signal when filtered using the square-root raised cosine filter.

```
% Upsample and filter.
yc = rctFilt3([x; zeros(Nsym/2,1)]);
% Correct for propagation delay by removing filter transients
yc = yc(fltDelay*Fs+1:end);
% Plot data.
```

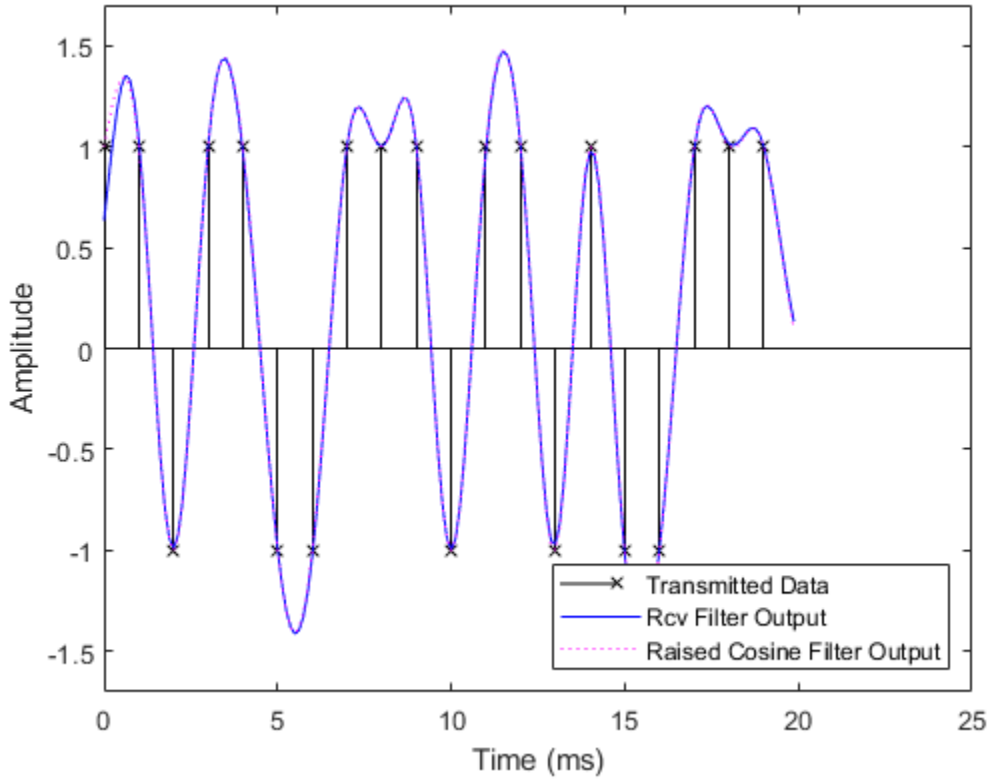
```
stem(tx, x, 'kx'); hold on;
% Plot filtered data.
plot(to, yc, 'm-'); hold off;
% Set axes and labels.
axis([0 25 -1.7 1.7]);  xlabel('Time (ms)'); ylabel('Amplitude');
legend('Transmitted Data','Sqrt. Raised Cosine','Location','southeast')
```



The transmitted signal (magenta curve) is then filtered at the receiver. We did not decimate the filter output to show the full waveform. The default unit energy normalization ensures that the gain of the combination of the transmit and receive filters is the same as the gain of a normalized raised cosine filter. The filtered received signal, which is virtually identical to the signal filtered using a single raised cosine filter, is depicted by the blue curve at the receiver.

```
% Design and normalize filter.
rcrFilt = comm.RaisedCosineReceiveFilter(...
  'Shape',                 'Square root', ...
  'RolloffFactor',         beta, ...
  'FilterSpanInSymbols',   Nsym, ...
  'InputSamplesPerSymbol', sampsPerSym, ...
  'DecimationFactor',      1);
% Filter at the receiver.
yr = rcrFilt([yc; zeros(Nsym*sampsPerSym/2, 1)]);
% Correct for propagation delay by removing filter transients
yr = yr(fltDelay*Fs+1:end);
% Plot data.
stem(tx, x, 'kx'); hold on;
% Plot filtered data.
```

```
plot(to, yr, 'b-',to, yo, 'm:'); hold off;
% Set axes and labels.
axis([0 25 -1.7 1.7]);  xlabel('Time (ms)'); ylabel('Amplitude');
legend('Transmitted Data','Rcv Filter Output', ...
    'Raised Cosine Filter Output','Location','southeast')
```



Computational Cost

In the following table, we compare the computational cost of a polyphase FIR interpolation filter and polyphase FIR decimation filter.

```
C1 = cost(rctFilt3);
C2 = cost(rcrFilt);
```

```
-------------------------------------------------------------------------
                    Implementation Cost Comparison
-------------------------------------------------------------------------
                        Multipliers  Adders  Mult/Symbol  Add/Symbol
Multirate Interpolator      49         41          49         41
Multirate Decimator         49         48           6.125      6
```

CORDIC-Based QPSK Carrier Synchronization

This model shows the use of a CORDIC (COordinate Rotation DIgital Computer) rotation algorithm in a digital PLL (Phase Locked Loop) implementation for QPSK carrier synchronization. Fixed-Point Designer™ is needed to run this model.

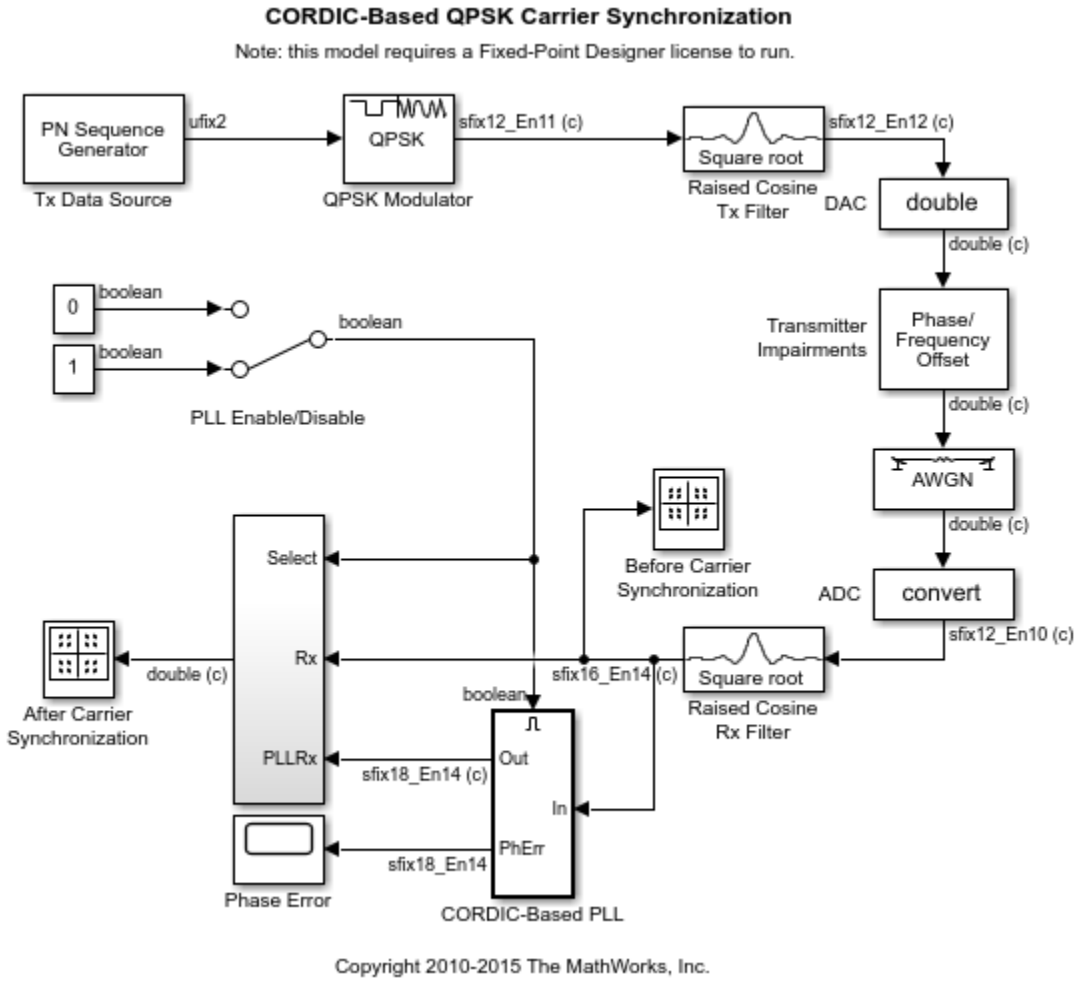
Introduction

The structure of a digital PLL is essentially equivalent to that of a continuous-time PLL. A PLL has the following components: a phase error detector (PED), a loop filter, and a controlled oscillator.

In the case of QPSK carrier (phase and frequency) synchronization, implementing the loop filter as a digital `P+I` (proportional-plus-integrator) filter produces a second order PLL. The controlled oscillator (`Phase Accumulator`) adjusts the angle of the received QPSK signal via a complex rotation.

You can implement the complex rotation using a variety of approaches, including direct complex multiplication by `exp(j*theta)`. However, such an implementation can be relatively expensive in terms of hardware (e.g., FPGA or ASIC) resources. An alternative approach uses a CORDIC-based rotation algorithm to implement the complex multiplication. This example uses this approach, via the Fixed-Point Designer™ `CORDICROTATE` function. This results in a multiplier-less complex rotation approximation, where the trade-off is in terms of speed. A small number of CORDIC iterations may often be enough to achieve a good digital PLL response, without the full hardware resource cost of a true complex multiplication.

Structure of the Example



Tx Data Source

The `PN Sequence Generator` library block from the Communications Toolbox™ is the Tx Data Source, generating unsigned 2-bit integer symbols.

QPSK Modulator

The `QPSK Modulator Baseband` library block from the Communications Toolbox uses a `pi/4` phase offset and binary ordering to compute signed 12-bit fixed-point modulator output values.

Raised Cosine Tx Filter

The `Raised Cosine Transmit Filter` library block from the Communications Toolbox performs square root FIR filtering with an upsampling factor of 8.

Transmitter Impairments

The `Phase/Frequency Offset` library block from the Communications Toolbox simulates the associated transmitter impairments. You can tune the `Phase offset` and `Frequency offset`

parameter values to see the effect on the PLL `Phase Error` time scope and the receive signal scatter plot displays.

AWGN Channel

The `AWGN Channel` library block from the Communications Toolbox simulates a noisy channel. You can tune the block `Eb/No` parameter to see the effect on the PLL `Phase Error` time scope and the receive signal scatter plot displays.

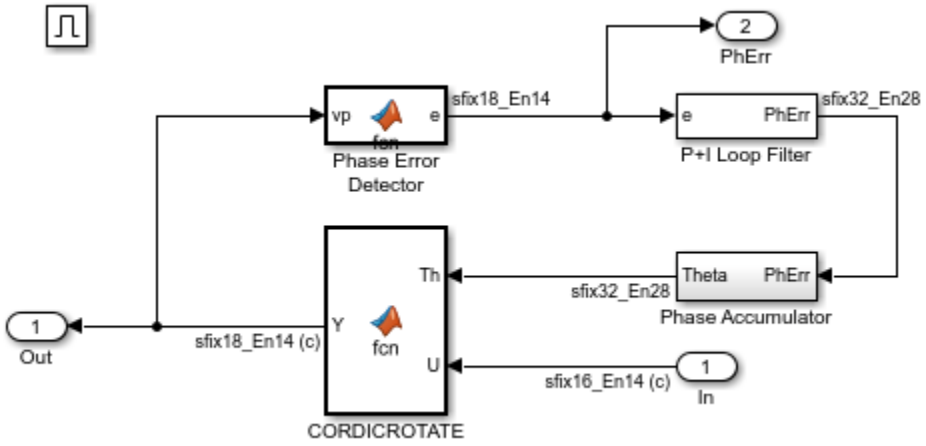
Raised Cosine Rx Filter

The `Raised Cosine Receive Filter` library block from the Communications Toolbox performs square root FIR filtering with a downsampling factor of 8.

CORDIC-Based PLL Subsystem

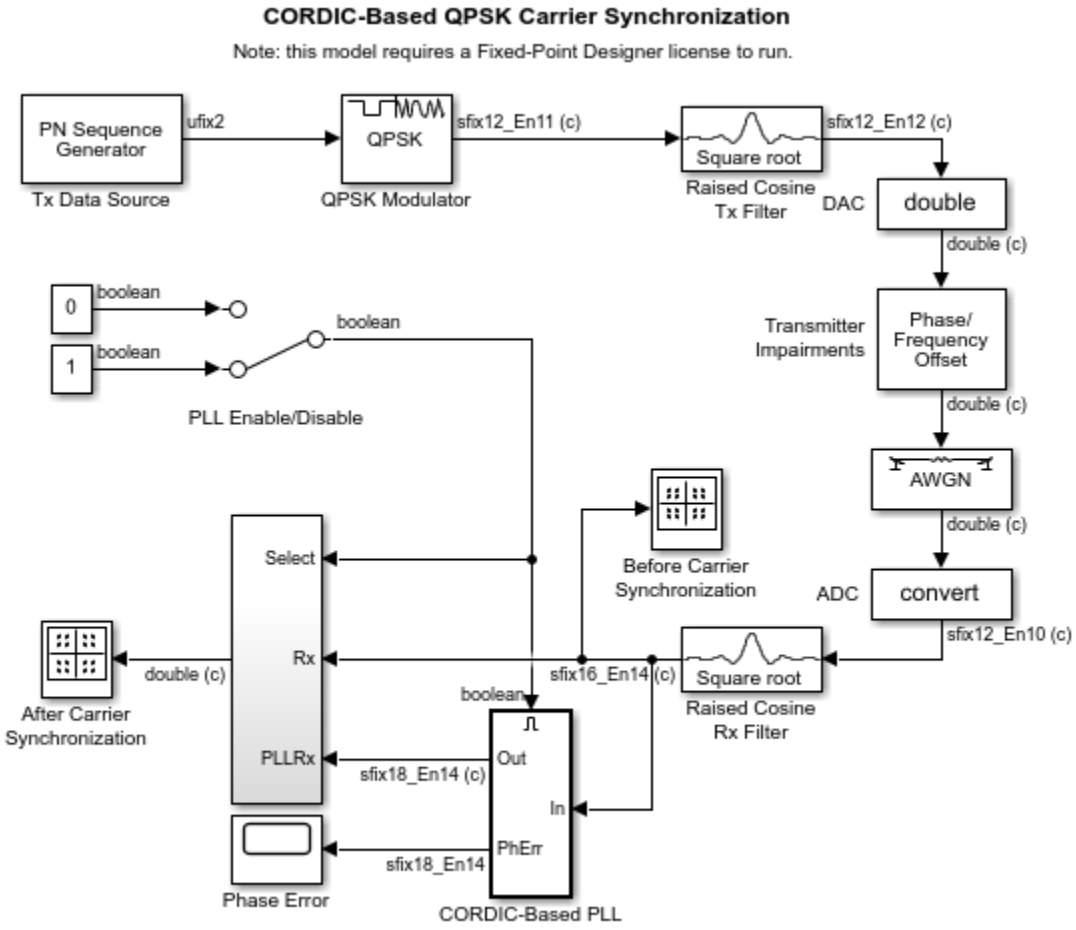
The `CORDIC-Based PLL` subsystem consists of a `Phase Error Detector` (PED), `P+I Loop Filter`, `Phase Accumulator`, and `CORDICROTATE` to form the corrected complex signal output values.

CORDIC-Based PLL



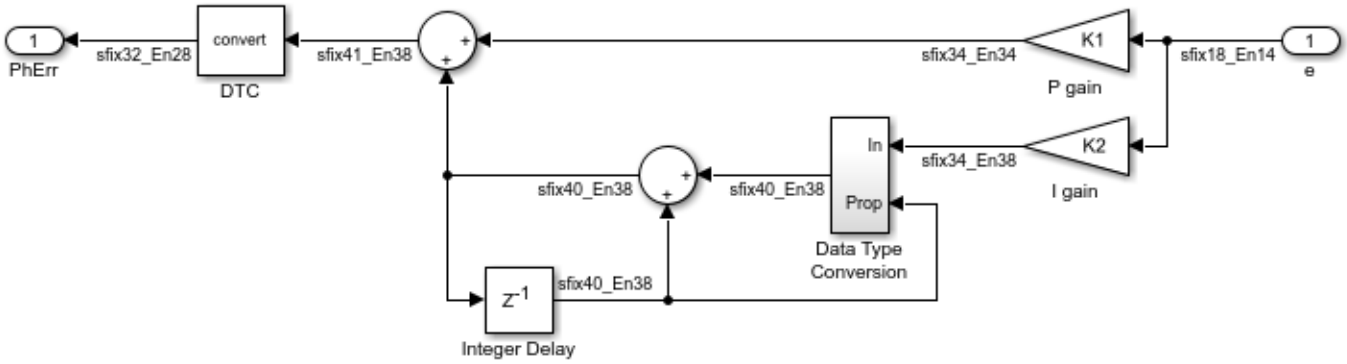
Phase Error Detector

The `Phase Error Detector` is implemented using a MATLAB® function.



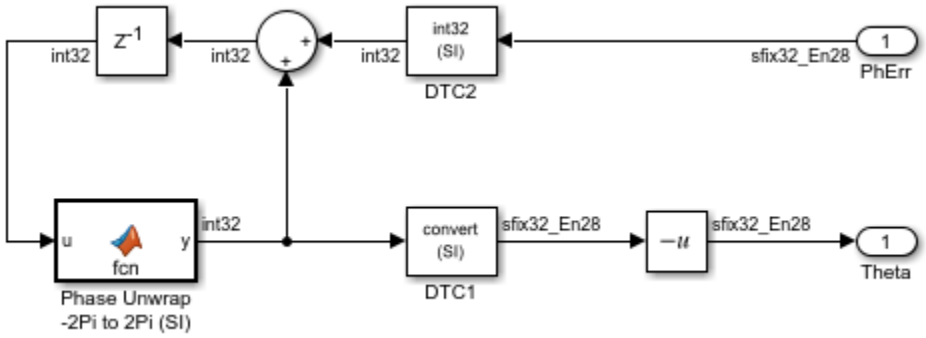
P+I Loop Filter

A `P+I Loop Filter` implements a second order PLL. The loop constants `K1` (P gain) and `K2` (I gain) are derived from the `Normalized loop bandwidth` and `Damping factor` parameters of the masked `CORDIC-Based PLL` subsystem.



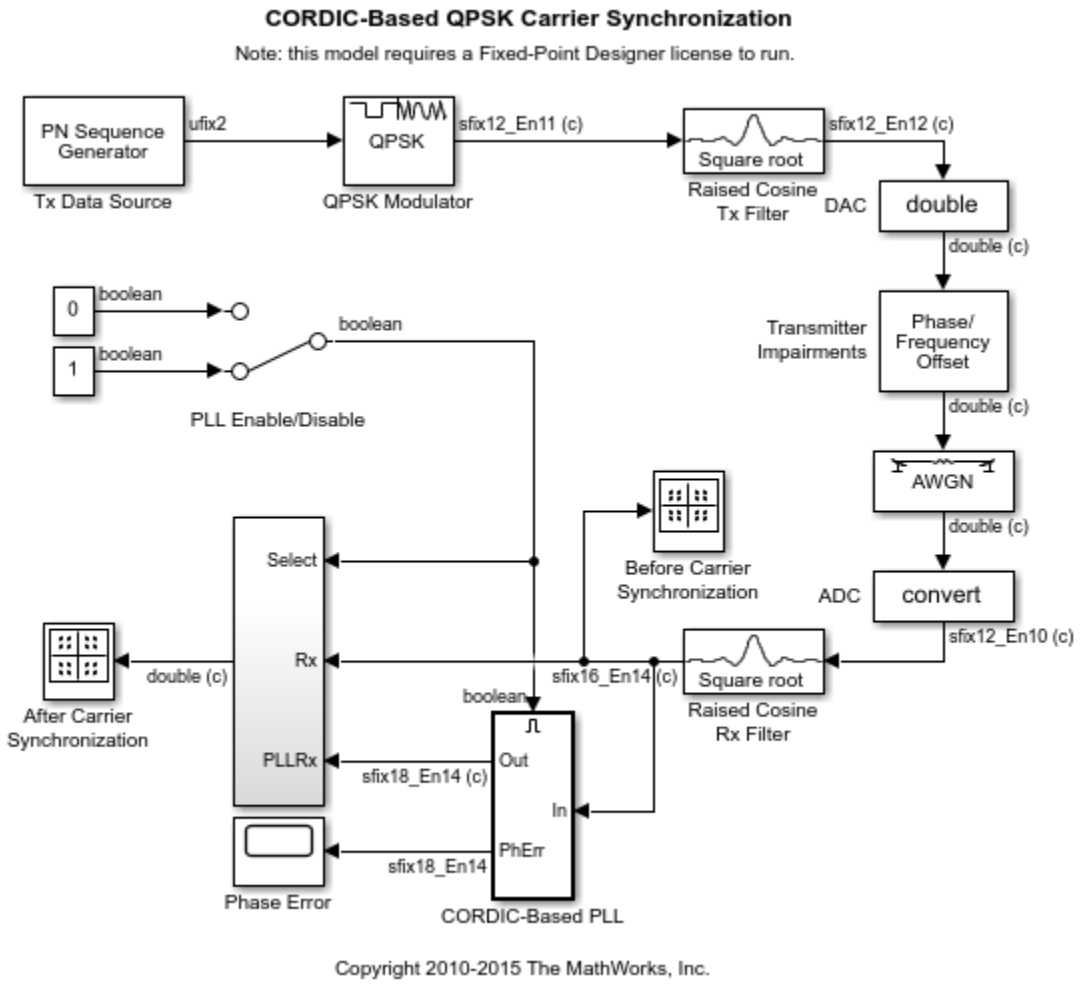
Phase Accumulator

The `Phase Accumulator` computes the angle `Theta`.



CORDICROTATE

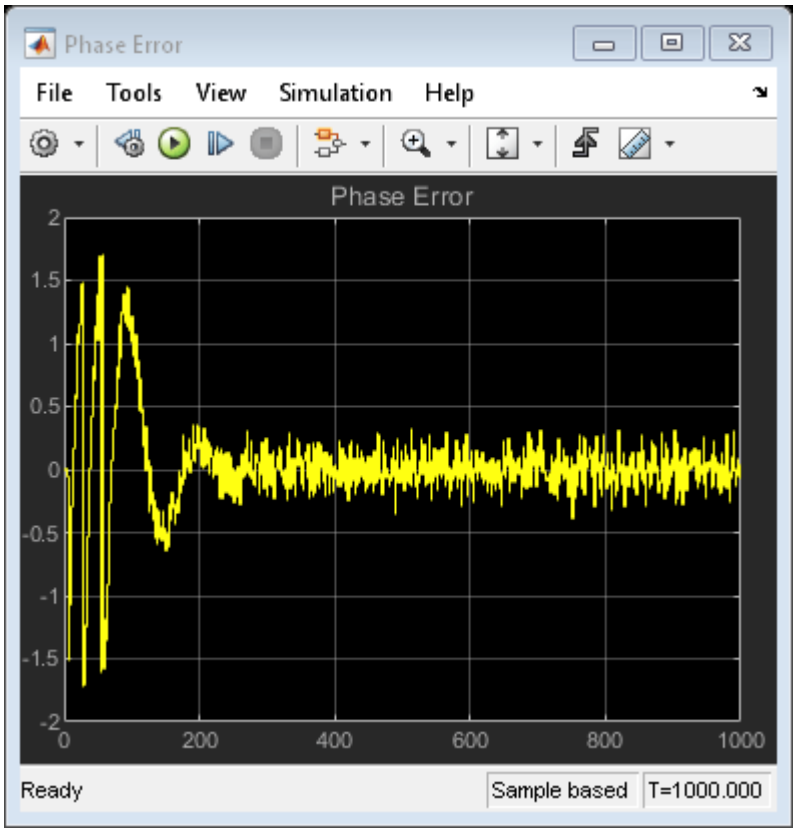
The MATLAB function `CORDICROTATE` rotates the complex received signal by `Theta` using an iterative, multiplier-less, CORDIC-based algorithm.



Results and Displays

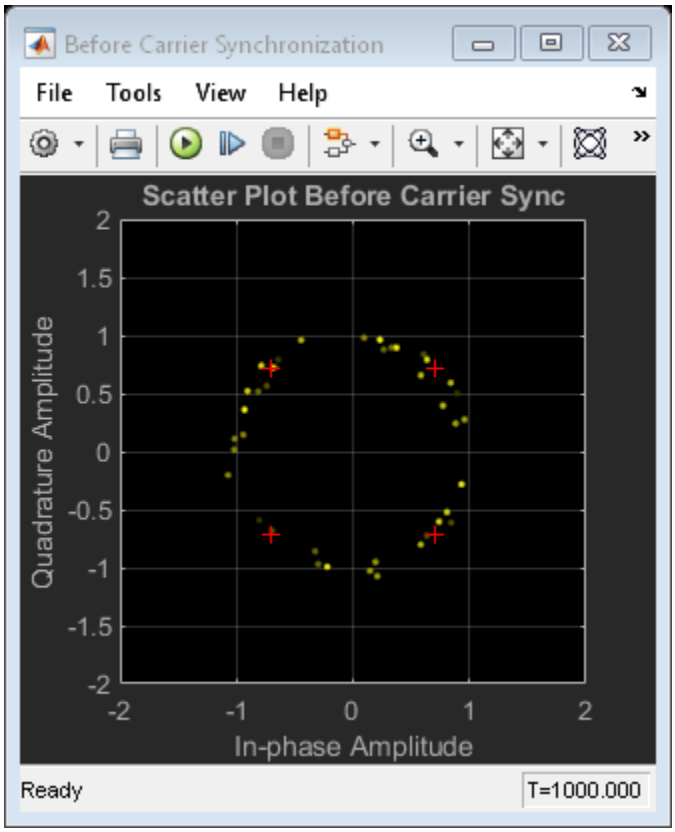
Phase Error

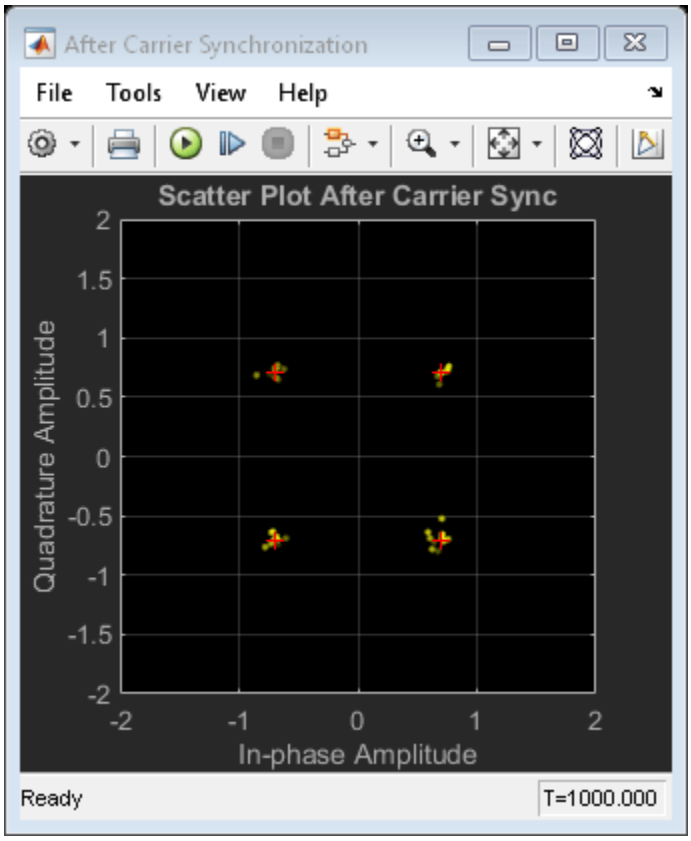
Use the `Phase Error` time scope block to view the time-varying PLL `Phase Error Detector` output values.



Scatter Plots

Use the `Before Carrier Synchronization` and `After Carrier Synchronization` scope blocks to observe the effects of tuning the `Transmitter Impairments` and `AWGN Channel` parameters.


Before Carrier Synchronization
File
Tools
View
Help
Scatter Plot Before Carrier Sync
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1
0
1
2
Ready
T=1000.000



Experimenting with the Example

Transmitter Impairments

To see the effects of transmitter phase and frequency offset impairments, change the `Phase offset` and `Frequency offset` parameter values while the model is running. Set the model `StopTime` to `inf` and use the `PLL Enable/Disable` switch to observe changes in the transient response.

AWGN Channel

To see the effects of a noisy channel, change the `Eb/No` parameter value while the model is running. Set the model `StopTime` to `inf` and use the `PLL Enable/Disable` switch to observe changes in the transient response.

CORDIC-Based PLL

Vary the PLL `Normalized loop bandwidth` and `Damping factor` parameters to tune the underlying `P+I Loop Filter` behavior while the model is running. Set the model `StopTime` to `inf` and use the `PLL Enable/Disable` switch to observe changes in the transient response.

Note that the phase-locked QPSK receive signal output contains phase ambiguity. For further analysis (e.g., symbol error rate computations), this phase ambiguity may be resolved using one of a number of well known methods, including known training (preamble) signals, varying demodulator phase offsets, constellation re-ordering, etc.

Selected Bibliography

Rice, Michael, "Discrete-Time Phase Locked Loops", **Digital Communications: A Discrete-Time Approach**, Appendix C, Sec. C.3, Pearson Prentice Hall, 2008.

Andraka, Ray, "A survey of CORDIC algorithm for FPGA based computers", **Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays**, 191 - 200, Feb. 22-24, 1998.

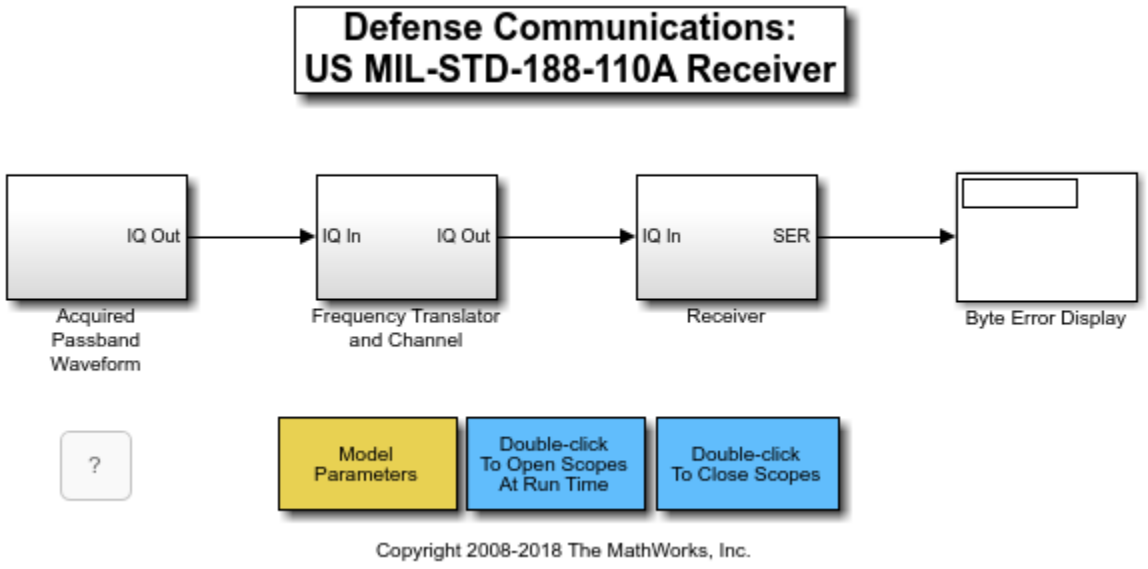
Volder, Jack E., "The CORDIC Trigonometric Computing Technique", **IRE Transactions on Electronic Computers**, Volume EC-8, 330 - 334, September 1959.

Defense Communications: US MIL-STD-188-110A Receiver

This model shows a communications system compliant with the U. S. MIL-STD-188-110A military standard. In particular, the model implements a full receiver that demodulates and outputs a text message, which was modulated by a reference transmitter and captured with data acquisition equipment. This model supports a 1200 bps data rate. It also implements an interleaver length of 0.6 s.

The system described in this standard is intended for long-haul and tactical communications over HF (high frequency) channels. The system is compatible with the NATO standard STANAG 4539.

Structure of the Example



This example consists of the following pieces, further described in the sections below:

- **Acquired Passband Waveform** - Outputs a bandpass MIL-STD-188-110A waveform centered at 1800 Hz
- **Frequency Translator and Channel** - Downconverts the signal to complex baseband and processes it with a choice of channels
- **Receiver** - Performs synchronization and baseband processing, and outputs a text message

Acquired Passband Waveform

The **Acquired Passband Waveform** subsystem uses a MATLAB® workspace variable to stream as an output. This variable represents data that has been generated by a standard-compliant transmitter and captured with data acquisition equipment. The nominal sample rate of the A/D is 9600 sps, but the actual A/D sampling rate is somewhat offset from that value, resulting in a symbol timing frequency offset.

Frequency Translator and Channel

This subsystem performs ideal downconversion to complex baseband, then processes the input signal with a choice of four successively degraded channels:

- a noiseless channel
- an AWGN channel
- a static frequency selective channel plus AWGN
- a fading frequency selective channel plus AWGN

The fading frequency selective channel is implemented by the SISO Fading Channel library block.

The use of multiple channels allows you to investigate their effects on receiver performance, especially that of the symbol synchronization blocks. The noiseless channel most effectively isolates the operation of the receiver, and the AWGN-only and static frequency selective channels show a graceful degradation in performance. The fading frequency selective channel models the moderate Watterson channel described in [2].

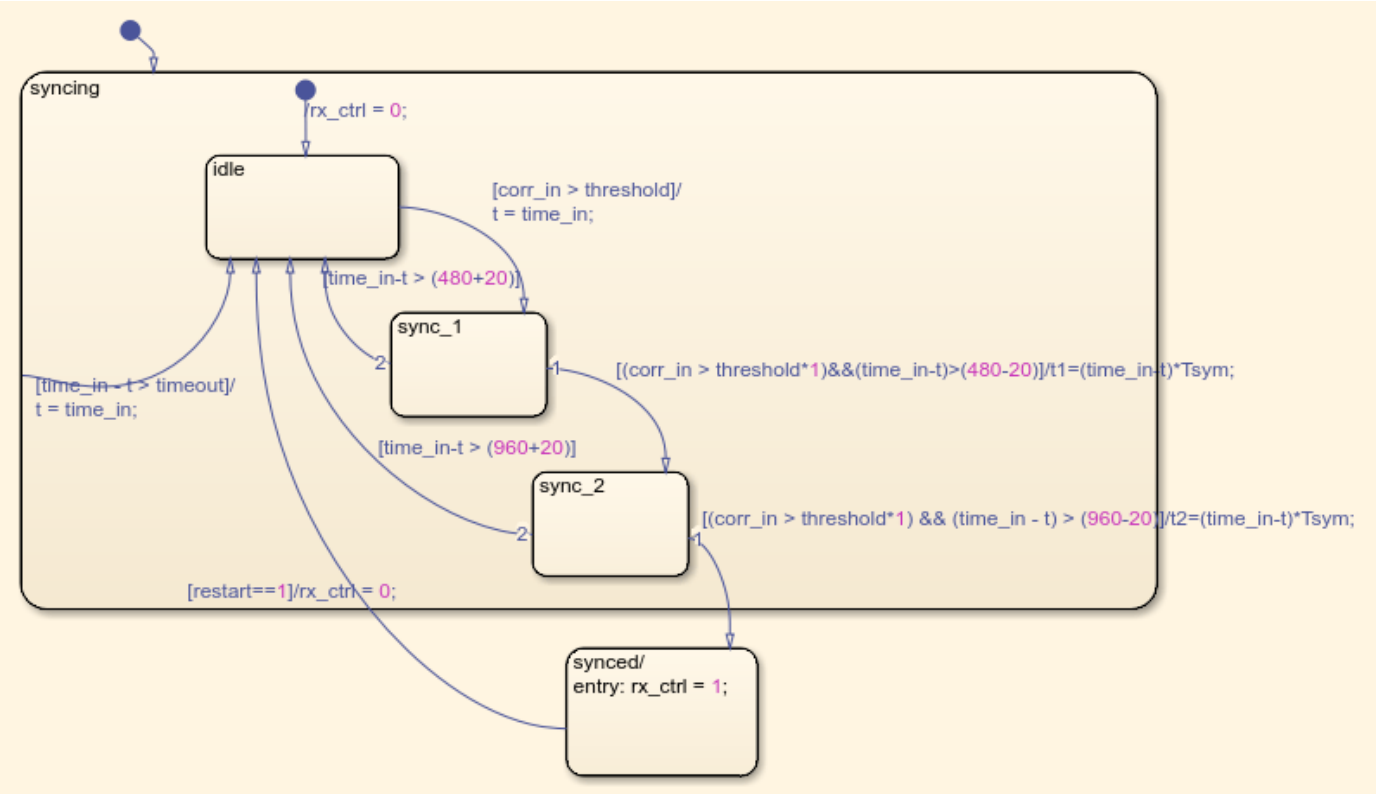
Receiver

The MIL-STD-188-110A receiver consists of four subsystems:

- **RRC Filter and AGC**
- **Preamble Detect to Enable Downstream Processing**
- **Carrier Recovery, Timing Recovery, and Equalization**
- **Demodulation and Error Correction**

The **RRC Filter and AGC** subsystem performs square root raised cosine filtering on the received signal, providing matched filtering for the transmitted waveform. The AGC ensures that the average signal power into the equalizer is 1 watt. This operation ensures that the constellation of the equalizer input signal is most closely matched to the ideal constellation against which it makes symbol decisions.

The **Preamble Detect to Enable Downstream Processing** subsystem performs a correlation on the known 0.6 sec synchronization preamble, which consists of three virtually identical 0.2 sec data segments. It detects three consecutive correlation peaks at 0.2, 0.4, and 0.6 sec in order to declare preamble detection. Once the preamble is detected, the subsystem sends a control signal to turn on the downstream processing, including: carrier recovery, timing recovery, equalization, demodulation, and error correction. The three consecutive peaks are detected with the Stateflow® state machine shown below. The block diagram shows the state machine in context with the preamble correlator, and the state machine is below the block diagram.



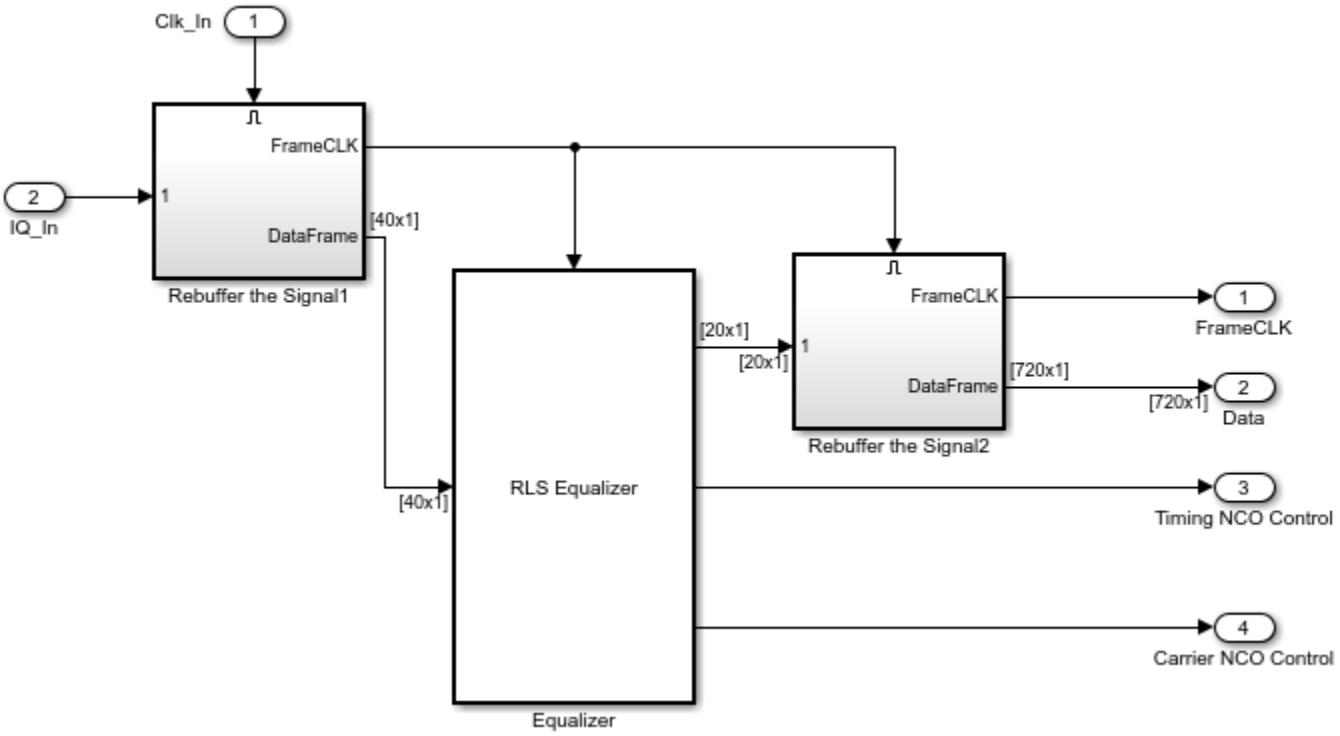
The **Carrier Recovery, Timing Recovery, and Equalization** subsystem uses a switchable NCO to generate a compensating sinusoid to remove the relatively constant carrier frequency offset. The NCO control signal is generated by estimating the phase error between the output of the RLS equalizer and its input. The RLS equalizer is implemented by the Decision Feedback Equalizer library block. The estimation is performed by computing the cross-spectrum between the equalizer input and its output, and performing a linear least squares fit on the resulting phase characteristic. The phase error is then filtered by a proportional-integral (PI) controller and fed to the carrier recovery NCO.

To compensate for the timing frequency error inherent in the acquired waveform, the **Carrier Recovery, Timing Recovery, and Equalization** subsystem uses a switchable timing control unit to generate a fractional delay value and a symbol clock. The fractional delay value is used to drive a variable delay block that uses a Farrow filter structure to interpolate its input. The variable delay is implemented by the Variable Fractional Delay library block.

The symbol clock, which runs at 9600 sps, is used to downsample the input signal, which is oversampled by four, down to the symbol rate of 2400 sym/sec. The clock typically goes high every four samples, but because of the timing frequency offset, it periodically goes high every five samples. The clock drives a rebuffering operation that creates symbol-spaced data in frames 40 samples long. These frames are ideally suited for processing by the RLS equalizer, since it has 40 taps. The rebuffering occurs in the **Carrier Recovery, Timing Recovery, and Equalization -> Equalize and Re-Buffer** subsystem. This subsystem also generates a frame clock that enables the RLS equalizer. This frame clock **also** runs at the oversampled rate of 9600 sps, but goes high nominally every 160 samples. Because of the timing frequency offset, it periodically goes high every 161 samples.

The pattern of using a high rate clock to drive a lower rate processing system can be used liberally in communications receiver designs. This pattern is shown in a more fundamental form in the DSP System Toolbox™ example “WWV Digital Receiver - Synchronization and Detection”. The **Carrier**

Recovery, Timing Recovery, and Equalization -> Equalize and Re-Buffer subsystem is shown below:



The time delay incurred by the RLS equalizer is estimated once again by a cross-spectral technique, and is used to drive the NCO of the timing control unit. A linear least squares fit is made to the phase characteristic of the cross spectrum between the equalizer input and its output. The slope of this phase estimates the delay induced by the equalizer.

The Decision Feedback Equalizer block is configured to use the RLS algorithm' and has 20 feedforward and 20 feedback taps. A DFE structure is necessary because of the deep spectral nulls induced by the Watterson channel. The quickly converging RLS weight update algorithm is needed to combat the rapid fading of the Watterson channel. Half the data that the equalizer processes is training data. This large percentage of training data is necessary because of the rapidly fluctuating HF channel. Once the training data is discarded, the equalizer output rate is nominally 1200 sps. Also, the equalizer subsystem performs descrambling to undo the scrambling performed by the transmitter.

The **Equalize and Re-Buffer** subsystem also generates a frame clock to enable the downstream processing performed in the **Demodulation and Error Correction** subsystem. The data into that downstream subsystem is packaged in frames of 720 samples long, which corresponds to a time duration of 0.6 sec. This second frame clock, as with the first one, **also** runs at the oversampled rate of 9600 sps, but goes high nominally every 5760 samples. However, due to the previous downsampling by four to derive symbol-rate data, and the effective downsampling by two from discarding equalizer training data, the clock triggers roughly every 5760 / 8 = 720 samples. However, because of the timing frequency offset, the clock actually goes high either every 5762 or every 5763 samples.

The **Demodulation and Error Correction** subsystem performs the following functions:

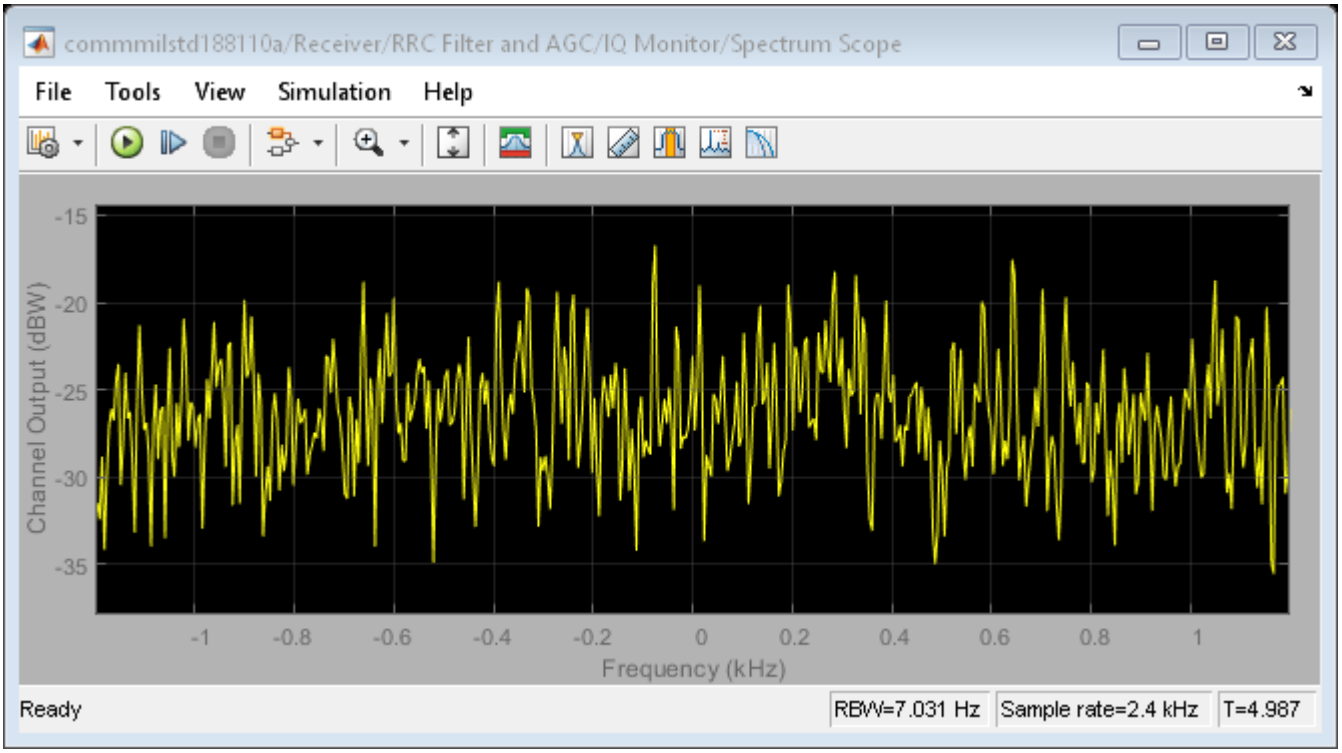
- Symbol extraction via QPSK demodulation
- Modified Gray encoding
- Block deinterleaving
- Viterbi decoding of the rate 1/2, constraint length 7 convolutional code
- Byte error rate calculations
- End-of-message detection
- Printing of the text message that drove the transmitter

Results and Displays

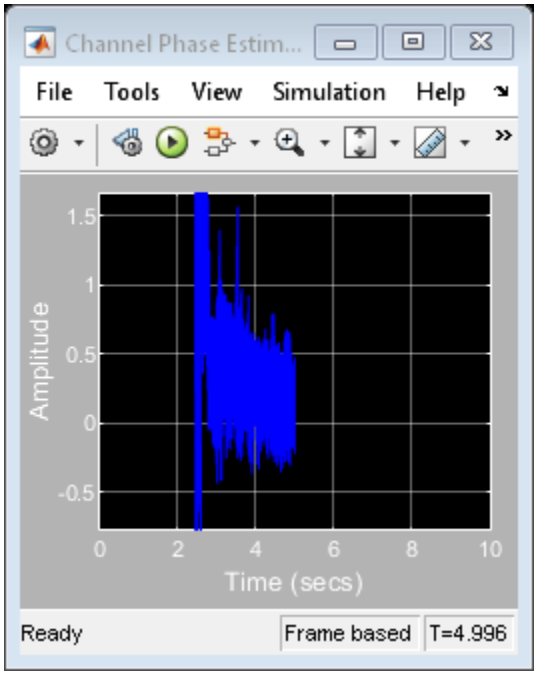
When you run the simulation, it displays these numerical or graphical results:

- The byte error rate
- The power spectrum of the channel output
- The estimate of the cross-spectral phase between the equalizer input and its output
- The control signal used to drive the Farrow fractional delay
- A scatter plot of the equalizer input
- A scatter plot of the equalizer output
- A scatter plot of the descrambler output
- A window showing the demodulated, decoded text message

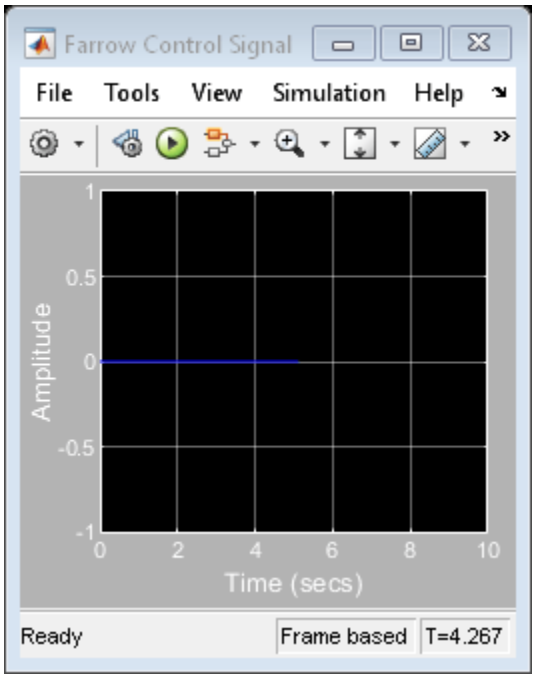
These plots are shown below, starting with the channel output power spectrum.



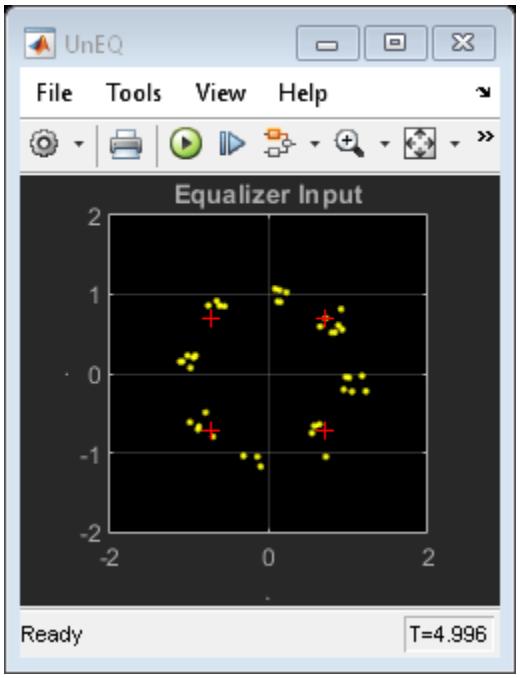
Below is the estimate of the cross-spectral phase between the equalizer input and its output.



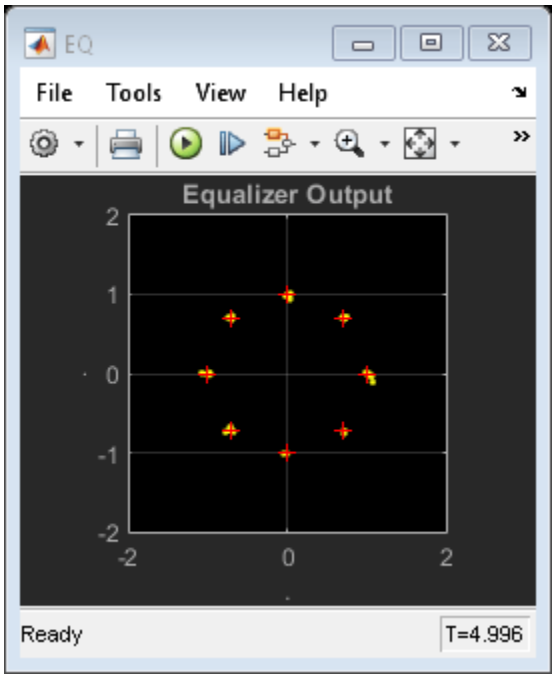
Below is the control signal used to drive the Farrow fractional delay.



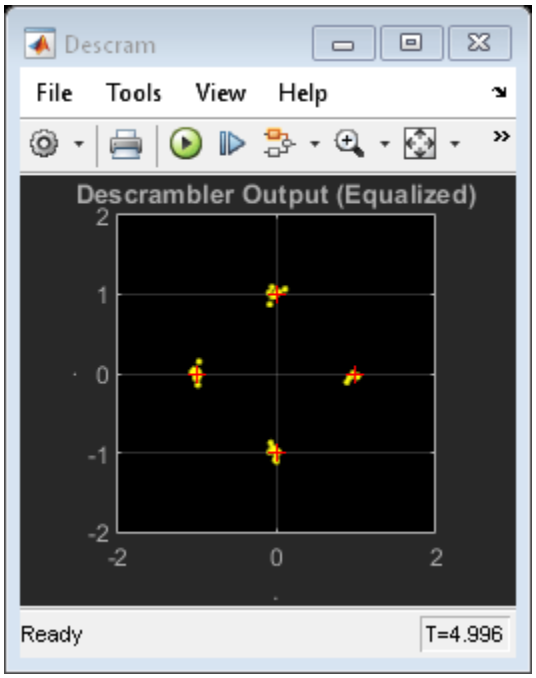
Below is the scatter plot of the equalizer input.



Below is the scatter plot of the equalizer output.



Below is the scatter plot of the descrambler output. Note that the 8PSK constellation has been collapsed to a QPSK constellation, per the MIL-STD-188-110A spec for this data rate.



Below is an excerpt from the demodulated message, which is taken from the MIL-STD-188-110a standard [1].

```
5.3.2 Serial (single-tone) mode.

5.3.2.1 General.

This mode shall employ M-ary phase-shift keying (PSK) on a

single carrier frequency as the modulation technique for

data transmission. Serial binary information accepted at

the line-side input is converted into a single 8-ary

PSK-modulated output carrier. The modulation of this output
```

Exploring the Example

The example allows you to experiment with multiple system capabilities to examine their effect on byte error rate performance. For instance, you can view the effect of changing the channel model on the various displays. In particular, when you select the fading frequency selective channel, the

channel phase estimate, the Farrow control signal, and the scatter plot displays are all noticeably degraded.

You can also enable or disable the timing control unit and the Farrow fractional delay. When the timing control unit is disabled, the demodulation operates properly for a time, but eventually the symbol timing frequency offset exceeds the length of the equalizer, which can no longer compensate for the delay. At that point, the demodulation process breaks down completely. When the Farrow fractional delay is disabled, and the timing control unit is enabled, the effect is more nuanced. However, in that case you can see the scatter plots flicker when the symbol timing crosses a symbol boundary. This is most easily seen in the noiseless case.

Take note of the quality of the demodulated message in the MATLAB figure window. For successively degraded channel and/or receiver configurations, the demodulated message becomes progressively more unreadable.

To generate executable code for this model, you will need to disable the display of the text message, via the Model Parameters subsystem. The block that performs the text printing is implemented with the Interpreted MATLAB Function block, which does not generate code.

Selected Bibliography

[1] MIL-STD-188-110B: Interoperability and Performance Standards for Data Modems, U. S. Department of Defense, 2000. (A superset of the MIL-STD-188-110A standard)

[2] ITU-R Recommendation 520-2: Use of High Frequency Ionospheric Channel Simulators, 1978/1982/1992.

See Also

The “Defense Communications: US MIL-STD-188-110B Baseband End-to-End Link” on page 8-405 example shows both a MIL-STD-188-110B transmitter and receiver, without the synchronization operations. It also enables a flexible choice of data rates, whereas this example has a fixed data rate of 1200 bps.

cdma2000 Waveform Generation

This example shows how to generate standard-compliant forward (downlink) and reverse (uplink) cdma2000® waveforms using the Communications Toolbox™.

Introduction

The Communications Toolbox can be used to generate preset or customized standard-compliant forward and reverse cdma2000 waveforms. Specifically, the following channels are supported:

Forward cdma2000:

- *Forward Pilot Channel (F-PICH)*
- *Forward Auxiliary Pilot Channel (F-APICH)*
- *Forward Transmit Diversity Pilot Channel (F-TDPICH)*
- *Forward Auxiliary Transmit Diversity Pilot Channel (F-ATDPICH)*
- *Forward Sync Channel (F-SYNC)*
- *Forward Paging Channel (F-PCH)*
- *Forward Quick Paging Channel (F-QPCH)*
- *Forward Broadcast Control Channel (F-BCCH)*
- *Forward Common Control Channel (F-CCCH)*
- *Forward Dedicated Control Channel (F-DCCH)*
- *Forward Common Power Control Channel (F-CPCCH)*
- *Forward Fundamental Traffic Channel (F-FCH), including Power Control Subchannel*
- *Forward Supplemental Code Channel (F-SCCH)*
- *Forward Supplemental Channel (F-SCH)*
- *Forward Packet Data Common Control Channel (F-PDCCH)*
- *Forward Orthogonal Channel Noise (F-OCNS)*

Reverse cdma2000:

- *Reverse Pilot Channel (R-PICH), including Power Control Subchannel*
- *Reverse Access Channel (R-ACH)*
- *Reverse Enhanced Access Channel (R-EACH)*
- *Reverse Common Control Channel (R-CCCH)*
- *Reverse Dedicated Control Channel (R-DCCH)*
- *Reverse Fundamental Traffic Channel (R-FCH)*
- *Reverse Supplemental Code Channel (R-SCCH)*
- *Reverse Supplemental Channel (R-SCH)*

The generated waveforms can be used for the following applications:

- *Golden reference for transmitter implementations*
- *Receiver testing and algorithm development*
- *Testing RF hardware and software*

- *Interference testing*

Waveform Generation Techniques

- Waveforms can be generated using the `cdma2000ForwardWaveformGenerator` and `cdma2000ReverseWaveformGenerator` functions. The input of these functions is a structure containing top-level waveform parameters as well as substructures containing channel-specific parameters. This example will illustrate how such structures can be constructed from scratch.
- Preset structure configurations can be created using the `cdma2000ForwardReferenceChannels` and `cdma2000ReverseReferenceChannels` functions. Such preset configurations can represent common Test and Measurement scenarios or provide a good starting point (wizard) for customizing a waveform configuration.

Generation of Preset-driven Forward and Reverse cdma2000 Waveforms

The preset structure configurations can then be passed to the waveform generation functions. For example, the following commands generate all forward and reverse channels allowable for Radio Configuration 4:

```
forwardPresetConfig     = cdma2000ForwardReferenceChannels('ALL-RC4');
forwardPresetWaveform   = cdma2000ForwardWaveformGenerator(forwardPresetConfig);

reversePresetConfig     = cdma2000ReverseReferenceChannels('ALL-RC4');
reversePresetWaveform   = cdma2000ReverseWaveformGenerator(reversePresetConfig);
```

Generation of a Forward cdma2000 Waveform Using Full Parameter List

Next, we illustrate the creation of equivalent configuration structures from scratch (for forward cdma2000). This is also useful for customizing the preset configurations.

```
fManualConfig.SpreadingRate          = 'SR1';           % Spreading Rate 1 or 3
fManualConfig.Diversity              = 'NTD';           % No Transmit Diversity (other options ar
fManualConfig.QOF                    = 'QOF1';          % Quasi-orthogonal function 1, 2 or 3
fManualConfig.PNOffset               = 0;               % PN offset of Base station
fManualConfig.LongCodeState          = 0;               % Initial long code state
fManualConfig.PowerNormalization     = 'Off';           % Power normalization: 'Off', 'Normalize
fManualConfig.OversamplingRatio      = 4;               % Upsampling factor
fManualConfig.FilterType             = 'cdma2000Long';  % Filter coefficients: 'cdma2000Long', 'c
fManualConfig.InvertQ                = 'Off';           % Negate the imaginary part of the wavefo
fManualConfig.EnableModulation       = 'Off';           % Enable carrier modulation
fManualConfig.ModulationFrequency    = 0;               % Modulation frequency (Hz)
fManualConfig.NumChips               = 1000;            % Number of chips in the waveform

fpich.Enable                = 'On';                     % Enable the F-PICH channel
fpich.Power                 = 0;                        % Relative channel power (dBW)
fManualConfig.FPICH         = fpich;                    % Add the channel to the waveform configu

fapich.Enable               = 'On';                     % Enable the F-APICH channel
fapich.Power                = 0;                        % Relative channel power (dBW)
fapich.WalshCode            = 10;                       % Unique Walsh code number
fapich.WalshLength          = 64;                       % Walsh code length
fManualConfig.FAPICH        = fapich;                   % Add the channel to the waveform configu

ftdpich.Enable              = 'On';                     % Enable the F-TDPICH channel
ftdpich.Power               = 0;                        % Relative channel power (dBW)
fManualConfig.FTDPICH       = ftdpich;                  % Add the channel to the waveform configu
```

```
fatdpich.Enable             = 'On';                 % Enable the F-ATDPICH channel
fatdpich.Power              = 0;                    % Relative channel power (dBW)
fatdpich.WalshCode          = 11;                   % Unique Walsh code number
fatdpich.WalshLength        = 64;                   % Walsh code length
fManualConfig.FATDPICH      = fatdpich;             % Add the channel to the waveform configu

fpch.Enable                 = 'On';                 % Enable the F-PCH channel
fpch.Power                  = 0;                    % Relative channel power (dBW)
fpch.LongCodeMask           = 0;                    % Long code mask
fpch.DataRate               = 4800;                 % Data rate (bps)
fpch.EnableCoding           = 'On';                 % Enable channel coding
fpch.DataSource             = {'PN9', 1};           % Input message: {'PNX', Seed} or numeric
fpch.WalshCode              = 1;                    % Unique Walsh code number
fManualConfig.FPCH          = fpch;                 % Add the channel to the waveform configu

fsync.Enable                = 'On';                 % Enable the F-SYNC channel
fsync.Power                 = 0;                    % Relative channel power (dBW)
fsync.EnableCoding          = 'On';                 % Enable channel coding
fsync.DataSource            = {'PN9', 1};           % Input message: {'PNX', Seed}, numerical
fManualConfig.FSYNC         = fsync;                % Add the channel to the waveform configu

fbcch.Enable                = 'On';                 % Enable the F-BCCH channel
fbcch.Power                 = 0;                    % Relative channel power (dBW)
fbcch.LongCodeMask          = 0;                    % Long code mask
fbcch.DataRate              = 4800;                 % Data rate (bps)
fbcch.FrameLength           = 160;                  % Frame length (ms)
fbcch.EnableCoding          = 'On';                 % Enable channel coding
fbcch.DataSource            = {'PN9', 1};           % Input message: {'PNX', Seed} or numeric
fbcch.WalshCode             = 2;                    % Unique Walsh code number
fbcch.CodingType            = 'conv';               % Coding type: 'conv' or 'turbo'
fManualConfig.FBCCH         = fbcch;                % Add the channel to the waveform configu

fcach.Enable                = 'On';                 % Enable the F-CACH channel
fcach.Power                 = 0;                    % Relative channel power (dBW)
fcach.LongCodeMask          = 0;                    % Long code mask
fcach.EnableCoding          = 'On';                 % Enable channel coding
fcach.DataSource            = {'PN9', 1};           % Input message: {'PNX', Seed} or numeric
fcach.WalshCode             = 3;                    % Unique Walsh code number
fcach.CodingType            = 'conv';               % Coding type: 'conv' or 'turbo'
fManualConfig.FCACH         = fcach;                % Add the channel to the waveform configu

fccch.Enable                = 'On';                 % Enable the F-CCCH channel
fccch.Power                 = 0;                    % Relative channel power (dBW)
fccch.LongCodeMask          = 0;                    % Long code mask
fccch.DataRate              = 9600;                 % Data rate (bps)
fccch.FrameLength           = 20;                   % Frame length (ms)
fccch.EnableCoding          = 'On';                 % Enable channel coding
fccch.DataSource            = {'PN9', 1};           % Input message: {'PNX', Seed} or numeric
fccch.WalshCode             = 4;                    % Unique Walsh code number
fccch.CodingType            = 'conv';               % Coding type: 'conv' or 'turbo'
fManualConfig.FCCCH         = fccch;                % Add the channel to the waveform configu

fcpcch.Enable               = 'On';                 % Enable the F-CPCCH channel
fcpcch.Power                = 0;                    % Relative channel power (dBW)
fcpcch.LongCodeMask         = 0;                    % Long code mask
fcpcch.EnableCoding         = 'On';                 % Enable channel coding
fcpcch.DataSource           = {'PN9', 1};           % Input message: {'PNX', Seed} or numeric
fcpcch.WalshCode            = 5;                    % Unique Walsh code number
```

```
fManualConfig.FCPCCH        = fcpcch;                   % Add the channel to the waveform configu

fqpch.Enable                = 'On';                     % Enable the F-QPCH channel
fqpch.Power                 = 0;                        % Relative channel power (dBW)
fqpch.LongCodeMask          = 0;                        % Long code mask
fqpch.DataRate              = 2400;                     % Data rate (bps)
fqpch.EnableCoding          = 'On';                     % Enable channel coding
fqpch.DataSource            = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
fqpch.WalshCode             = 6;                        % Unique Walsh code number
fManualConfig.FQPCH         = fqpch;                    % Add the channel to the waveform configu

ffch.Enable                 = 'On';                     % Enable the F-FCH channel
ffch.Power                  = 0;                        % Relative channel power (dBW)
ffch.RadioConfiguration     = 'RC4';                    % Radio Configuration: 1-9
ffch.DataRate               = 9600;                     % Data rate (bps)
ffch.FrameLength            = 20;                       % Frame length (ms)
ffch.LongCodeMask           = 0;                        % Long code mask
ffch.EnableCoding           = 'On';                     % Enable channel coding
ffch.DataSource             = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
ffch.WalshCode              = 7;                        % Unique Walsh code number
ffch.EnableQOF              = 'Off';                    % Enable QOF spreading
ffch.PowerControlEnable     = 'Off';                    % Enable the Power Control Subchannel
fManualConfig.FFCH          = ffch;                     % Add the channel to the waveform configu

focns.Enable                = 'On';                     % Enable the F-OCNS channel
focns.Power                 = -30;                      % Relative channel power (dBW)
focns.WalshCode              = 12;                      % Unique Walsh code number
focns.WalshLength           = 128;                      % Walsh code length
fManualConfig.FOCNS         = focns;                    % Add the channel to the waveform configu

fdcch.Enable                = 'On';                     % Enable the F-DCCH channel
fdcch.Power                 = 0;                        % Relative channel power (dBW)
fdcch.RadioConfiguration    = 'RC4';                    % Radio Configuration: 1-9
fdcch.LongCodeMask          = 0;                        % Long code mask
fdcch.DataRate              = 9600;                     % Data rate (bps)
fdcch.FrameLength           = 5;                        % Frame length (ms)
fdcch.EnableCoding          = 'On';                     % Enable channel coding
fdcch.DataSource            = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
fdcch.WalshCode             = 8;                        % Unique Walsh code number
fdcch.EnableQOF             = 'off';                    % Enable QOF spreading
fManualConfig.FDCCH         = fdcch;                    % Add the channel to the waveform configu

fsch.Enable                 = 'On';                     % Enable the F-SCH channel
fsch.Power                  = 0;                        % Relative channel power (dBW)
fsch.RadioConfiguration     = 'RC4';                    % Radio Configuration: 1-9
fsch.DataRate               = 9600;                     % Data rate (bps)
fsch.FrameLength            = 20;                       % Frame length (ms)
fsch.LongCodeMask           = 0;                        % Long code mask
fsch.EnableCoding           = 'On';                     % Enable channel coding
fsch.DataSource             = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
fsch.WalshCode              = 9;                        % Unique Walsh code number
fsch.EnableQOF              = 'Off';                    % Enable QOF spreading
fsch.CodingType             = 'conv';                   % Coding type: 'conv' or 'turbo'
fManualConfig.FSCH          = fsch;                     % Add the channel to the waveform configu

forwardManualWaveform    = cdma2000ForwardWaveformGenerator(fManualConfig);

% Demonstrate that the above two parameterization approaches are equivalent:
```

```
if(isequal(forwardPresetConfig, fManualConfig))
    disp([  'Configuration structures generated with and without the ' ...
            'cdma2000ForwardReferenceChannels function are the same.']);
end
```

```
Configuration structures generated with and without the cdma2000ForwardReferenceChannels functio
```

Generation of a Reverse cdma2000 Waveform Using Full Parameter List

```
rManualConfig.RadioConfiguration    = 'RC4';            % Radio Configuration: 1-6
rManualConfig.PowerNormalization    = 'Off';            % Power normalization: 'Off', 'Normalize
rManualConfig.OversamplingRatio     = 4;                % Upsampling factor
rManualConfig.FilterType            = 'cdma2000Long';   % Filter coefficients: 'cdma2000Long', '
rManualConfig.InvertQ               = 'Off';            % Negate the imaginary part of the wavef
rManualConfig.EnableModulation      = 'Off';            % Enable carrier modulation
rManualConfig.ModulationFrequency   = 0;                % Modulation frequency (Hz)
rManualConfig.NumChips              = 1000;             % Number of chips in the waveform

rfch.Enable                 = 'On';                     % Enable the R-FCH channel
rfch.Power                  = 0;                        % Relative channel power (dBW)
rfch.LongCodeMask           = 0;                        % Long code mask
rfch.EnableCoding           = 'On';                     % Enable channel coding
rfch.DataSource             = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
rfch.DataRate               = 14400;                    % Data rate (bps)
rfch.FrameLength            = 20;                       % Frame length (ms)
rfch.WalshCode              = 1;                        % Unique Walsh code number
rManualConfig.RFCH          = rfch;                     % Add the channel to the waveform configu

rpich.Enable                = 'On';                     % Enable the R-PICH channel
rpich.Power                 = 0;                        % Relative channel power (dBW)
rpich.LongCodeMask          = 0;                        % Long code mask
rpich.PowerControlEnable    = 'Off';                    % Enable the Power Control Subchannel
rManualConfig.RPICH         = rpich;                    % Add the channel to the waveform configu

reach.Enable                = 'On';                     % Enable the R-EACH channel
reach.Power                 = 0;                        % Relative channel power (dBW)
reach.LongCodeMask          = 0;                        % Long code mask
reach.EnableCoding          = 'On';                     % Enable channel coding
reach.DataSource            = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
reach.DataRate              = 9600;                     % Data rate (bps)
reach.FrameLength           = 20;                       % Frame length (ms)
reach.WalshCode             = 2;                        % Unique Walsh code number
rManualConfig.REACH         = reach;                    % Add the channel to the waveform configu

rcch.Enable                 = 'On';                     % Enable the R-CCH channel
rcch.Power                  = 0;                        % Relative channel power (dBW)
rcch.LongCodeMask           = 0;                        % Long code mask
rcch.EnableCoding           = 'On';                     % Enable channel coding
rcch.DataSource             = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
rcch.DataRate               = 9600;                     % Data rate (bps)
rcch.FrameLength            = 20;                       % Frame length (ms)
rcch.WalshCode              = 3;                        % Unique Walsh code number
rManualConfig.RCCCH         = rcch;                     % Add the channel to the waveform configu

rdcch.Enable                = 'On';                     % Enable the R-DCCH channel
rdcch.Power                 = 0;                        % Relative channel power (dBW)
rdcch.LongCodeMask          = 0;                        % Long code mask
rdcch.EnableCoding          = 'On';                     % Enable channel coding
```

```
rdcch.DataSource            = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
rdcch.DataRate              = 14400;                    % Data rate (bps)
rdcch.FrameLength           = 20;                       % Frame length (ms)
rdcch.WalshCode             = 4;                        % Unique Walsh code number
rManualConfig.RDCCH         = rdcch;                    % Add the channel to the waveform configu

rsch1.Enable                = 'On';                     % Enable the R-SCH1 channel
rsch1.Power                 = 0;                        % Relative channel power (dBW)
rsch1.LongCodeMask          = 0;                        % Long code mask
rsch1.EnableCoding          = 'On';                     % Enable channel coding
rsch1.DataSource            = {'PN9', 1};               % Input message: {'PNX', Seed} or numeric
rsch1.DataRate              = 14400;                    % Data rate (bps)
rsch1.FrameLength           = 20;                       % Frame length (ms)
rsch1.WalshLength           = 8;                        % Walsh code length
rsch1.WalshCode             = 5;                        % Unique Walsh code number
rManualConfig.RSCH1         = rsch1;                    % Add the channel to the waveform configu

rsch2                       = rsch1;                    % Apply the same settings with R-SCH1
rsch2.WalshCode             = 6;                        % Except for the unique Walsh code numbe
rManualConfig.RSCH2         = rsch2;                    % Add the channel to the waveform configu

reverseManualWaveform   = cdma2000ReverseWaveformGenerator(rManualConfig);

% Demonstrate that the above two parameterization approaches are equivalent:
if(isequal(reversePresetConfig, rManualConfig))
    disp([  'Configuration structures generated with and without the ' ...
            'cdma2000ForwardReferenceChannels function are the same.']);
end
```

```
Configuration structures generated with and without the cdma2000ForwardReferenceChannels functio
```

Waveform Comparison

Compare the waveforms generated using both approaches described above and see that the generated waveforms are identical

```
if(isequal(forwardPresetWaveform, forwardManualWaveform))
    disp([  'Forward waveforms generated with and without the ' ...
            'cdma2000ForwardReferenceChannels function are the same.']);
end

if(isequal(reversePresetWaveform, reverseManualWaveform))
    disp([  'Reverse waveforms generated with and without the ' ...
            'cdma2000ReverseReferenceChannels function are the same.']);
end
```

```
Forward waveforms generated with and without the cdma2000ForwardReferenceChannels function are t
Reverse waveforms generated with and without the cdma2000ReverseReferenceChannels function are t
```

Customization of Configuration

The configuration structures can be customized in order to create a waveform that better suits your objective. You can also customize the preset waveforms in order to exploit additional capabilities, such as:

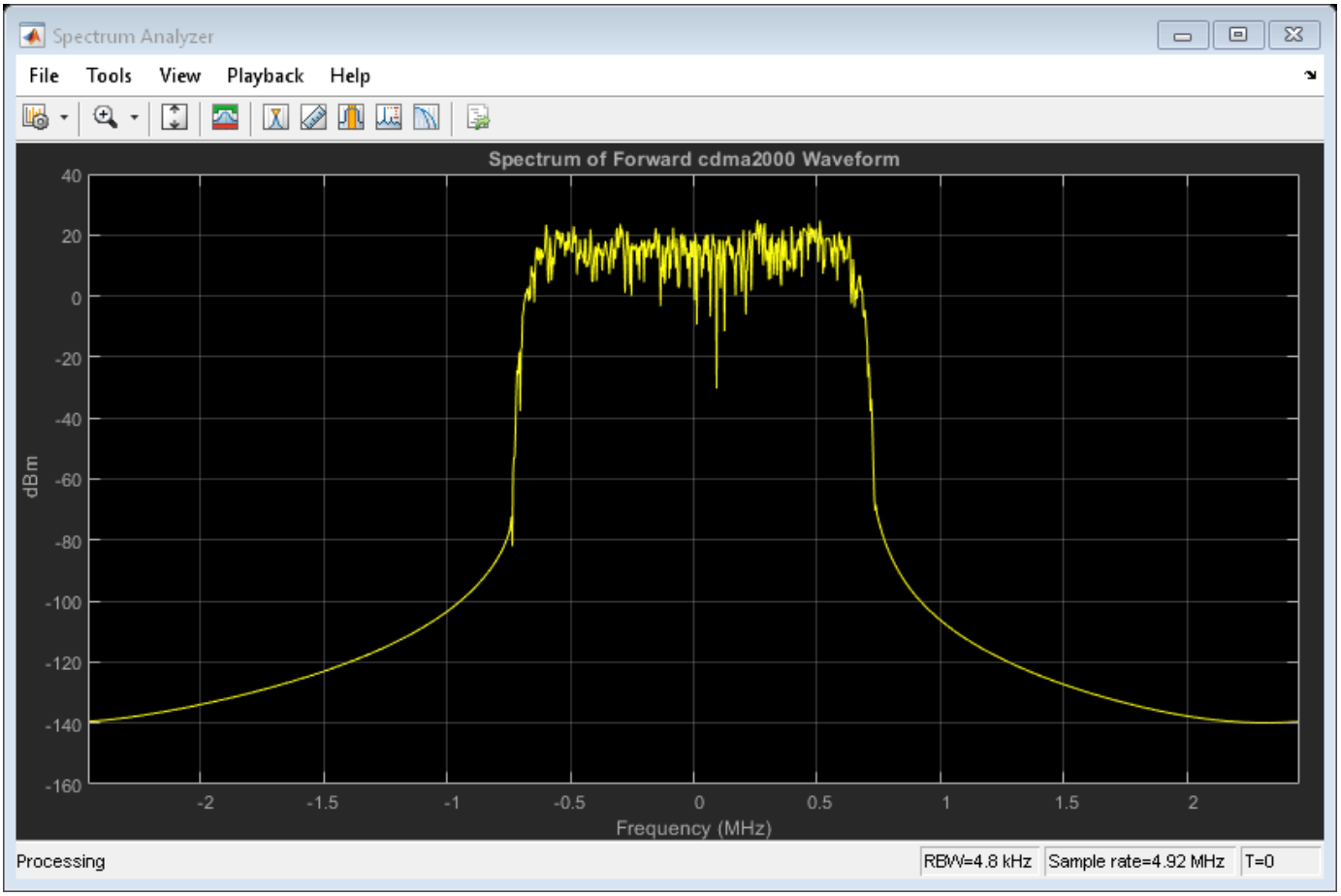
```
% 1. Specifying the message of the Sync channel:
fManualConfig2              = fManualConfig;
fsync.Enable                = 'On';                     % Enable the F-SYNC channel
```

```
fsync.Power                = 0;                          % Relative channel power (dBW)
fsync.EnableCoding         = 'On';                       % Enable channel coding
fsync.DataSource           = 'SyncMessage';              % Input message: {'PNX', Seed}, numerica
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 Savings 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
fsync.SyncMessage          = sm;                         % Sync channel message substructure, use
fManualConfig2.FSYNC        = fsync;                      % Add the channel to the waveform confi

% 2. Enabling the Power Control Subchannel of the Forward Fundamental Channel:
ffch.Enable                = 'On';                       % Enable the F-FCH channel
ffch.Power                 = 0;                          % Relative channel power (dBW)
ffch.RadioConfiguration    = 'RC4';                      % Radio Configuration: 1-9
ffch.DataRate              = 9600;                       % Data rate (bps)
ffch.FrameLength           = 20;                         % Frame length (ms)
ffch.LongCodeMask          = 0;                          % Long code mask
ffch.EnableCoding          = 'On';                       % Enable channel coding
ffch.DataSource            = {'PN9', 1};                 % Input message: {'PNX', Seed} or numeri
ffch.WalshCode             = 7;                          % Unique Walsh code number
ffch.EnableQOF             = 'Off';                      % Enable QOF spreading
ffch.PowerControlEnable    = 'On';                       % Enable the Power Control Subchannel
ffch.PowerControlPower     = 0;                          % Power control subchannel power (relati
ffch.PowerControlDataSource = {'PN9',1};                 % Power control subchannel data source
fManualConfig2.FFCH         = ffch;                       % Add the channel to the waveform confi

forwardManualWaveform2   = cdma2000ForwardWaveformGenerator(fManualConfig2);
```

Plot Spectrum of Forward cdma2000 Waveform

Plot the spectrum of the time domain signal `forwardManualWaveform`.

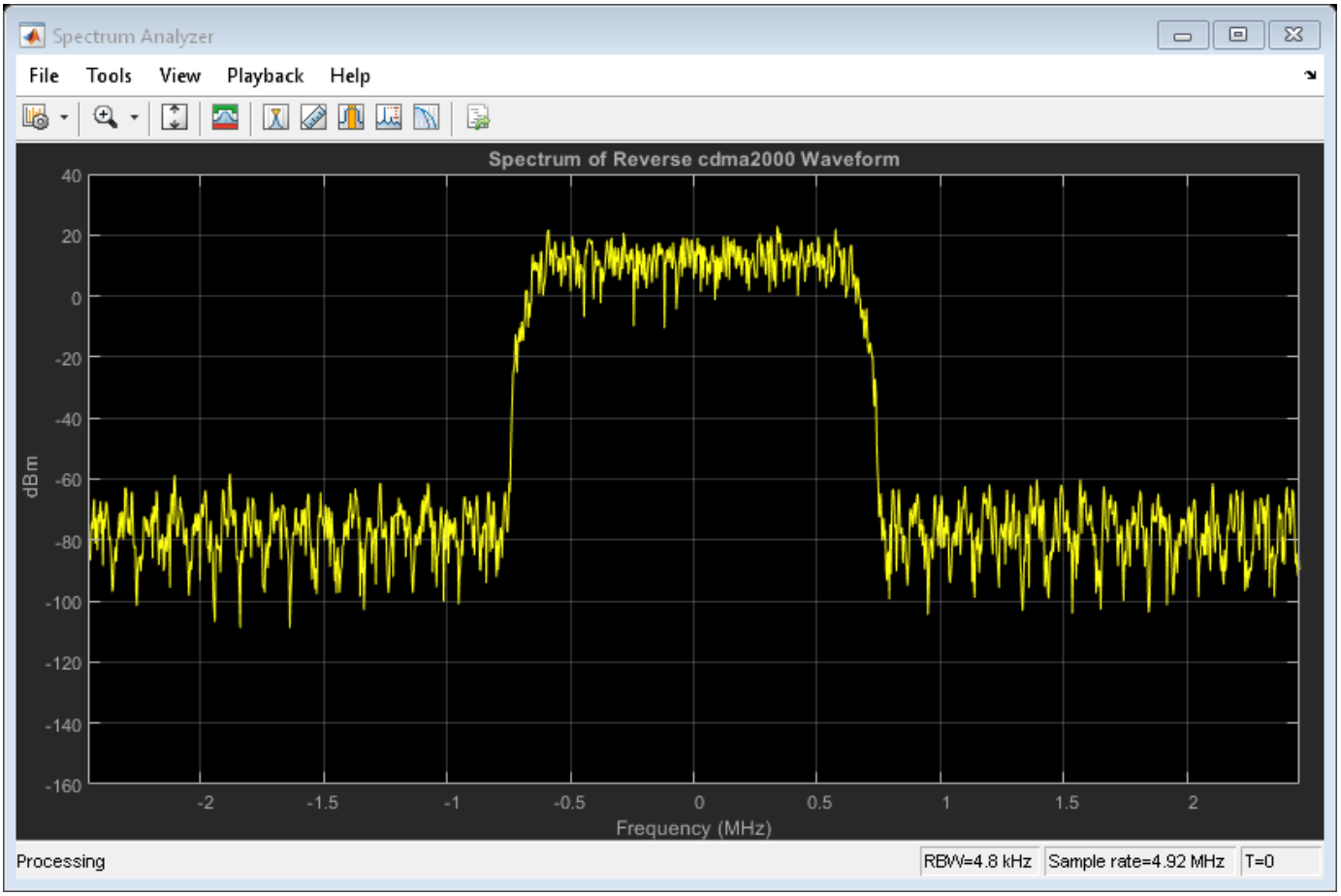
```
chiprate                = 1.2288e6;   % Chip rate of the baseband waveform (SR1)
fSpectrumPlot           = dsp.SpectrumAnalyzer('SampleRate', chiprate*fManualConfig.Oversampling
fSpectrumPlot.Title     = 'Spectrum of Forward cdma2000 Waveform';
fSpectrumPlot.YLimits   = [-160,40];
fSpectrumPlot(forwardManualWaveform);
```



Plot Spectrum of Reverse cdma2000 Waveform

Plot the spectrum of the time domain signal `reverseManualWaveform`.

```
chiprate                = 1.2288e6;   % Chip rate of the baseband waveform (SR1)
rSpectrumPlot           = dsp.SpectrumAnalyzer('SampleRate', chiprate*rManualConfig.Oversampling
rSpectrumPlot.Title     = 'Spectrum of Reverse cdma2000 Waveform';
rSpectrumPlot.YLimits   = [-160,40];
rSpectrumPlot(reverseManualWaveform);
```



Selected Bibliography

1 C.S0002-F v2.0: Physical Layer Standard for cdma2000 Spread Spectrum Systems.

1xEV-DO Waveform Generation

This example shows how to generate standard-compliant forward (downlink) and reverse (uplink) 1xEV-DO waveforms using the Communications Toolbox™.

Introduction

The Communications Toolbox can be used to generate preset or customized standard-compliant forward and reverse, Release 0 and Revision A 1xEV-DO waveforms.

The generated waveforms can be used for the following applications:

- *Golden reference for transmitter implementations*
- *Receiver testing and algorithm development*
- *Testing RF hardware and software*
- *Interference testing*

Waveform Generation Techniques

- Waveforms can be generated using the `evdoForwardWaveformGenerator` and `evdoReverseWaveformGenerator` functions. The input of these functions is a structure containing top-level waveform parameters as well as substructures containing channel- or packet-specific parameters. This example will illustrate how such structures can be constructed from scratch.
- Preset structure configurations can be created using the `evdoForwardReferenceChannels` and `evdoReverseReferenceChannels` functions. Such preset configurations can represent common Test and Measurement scenarios or provide a good starting point (wizard) for customizing a waveform configuration.

Generation of Preset-driven Forward and Reverse 1xEV-DO Waveforms

The preset structure configurations can then be passed to the waveform generation functions. For example, the following commands generate Revision A and Release 0 forward and reverse waveforms, respectively.

```
forwardPresetConfig = evdoForwardReferenceChannels('RevA-5120-2-64', 10);
forwardPresetWaveform = evdoForwardWaveformGenerator(forwardPresetConfig);

reversePresetConfig = evdoReverseReferenceChannels('Rel0-38400', 10);
reversePresetWaveform = evdoReverseWaveformGenerator(reversePresetConfig);
```

Generation of a Forward 1xEV-DO Waveform Using Full Parameter List

Next, we illustrate the creation of equivalent configuration structures from scratch. This is also useful for customizing the preset configurations.

```
% Create top-level waveform parameters:
fManualConfig.Release = 'RevisionA';            % 'Release0' or 'RevisionA'
fManualConfig.PNOffset = 0;                     % PN Offset of the Base station
fManualConfig.IdleSlotsWithControl = 'Off';
fManualConfig.EnableControl = 'On';
fManualConfig.OversamplingRatio = 4;            % Upsampling factor
fManualConfig.FilterType = 'cdma2000Long';      % Filter coefficients: 'cdma2000Long', 'cdma2000
fManualConfig.InvertQ = 'Off';                  % Negate the imaginary output
```

```
fManualConfig.EnableModulation = 'Off';         % Enable modulation
fManualConfig.ModulationFrequency = 0;          % Modulation frequency (Hz)
fManualConfig.NumChips = 41600;                 % Number of chips in the waveform

% Create a input message source for the packets:
pds.MACIndex = 0;                               % MAC index associated with data
pds.DataSource = {'PN9', 1};                    % Input message: {'PNX', Seed} or numerical vecto
pds.EnableCoding = 'On';                        % Enable channel coding
fManualConfig.PacketDataSources = pds;          % Add the data source specification to the wavefo

% Create a single packet:
fPacket.MACIndex = 0;                           % MAC index associated with this packet
fPacket.PacketSize = 5120;                      % The packet size: 128, 256, 512, 1024, 2048 4096
fPacket.NumSlots = 2;                           % The number of slots: 1, 2, 4, 8 or 16
fPacket.PreambleLength = 64;                    % The preamble length: 64, 128, 256, 512 or 1024
% Create a sequence of 10 packets:
fManualConfig.PacketSequence = repmat(fPacket, 1, 10);

% Generate waveform:
forwardManualWaveform = evdoForwardWaveformGenerator(fManualConfig);

% Demonstrate that the above two parameterization approaches are equivalent:
if(isequal(forwardPresetConfig, fManualConfig))
    disp([  'Configuration structures generated with and without the ' ...
            'evdoForwardReferenceChannels function are the same.']);
end
```

```
Configuration structures generated with and without the evdoForwardReferenceChannels function are
```

Generation of a Reverse 1xEV-DO Waveform Using Full Parameter List

```
% Create top-level waveform parameters:
rManualConfig.Release = 'Release0';             % 'Release0' or 'RevisionA'
rManualConfig.LongCodeMaskI = 0;                % Initial long code mask for I channel
rManualConfig.LongCodeMaskQ = 0;                % Initial long code mask for Q channel
rManualConfig.OversamplingRatio = 4;            % Upsampling factor
rManualConfig.FilterType = 'cdma2000Long';      % Filter coefficients: 'cdma2000Long', 'cdma2000S
rManualConfig.InvertQ = 'Off';                  % Negate the imaginary output
rManualConfig.EnableModulation = 'Off';         % Enable modulation
rManualConfig.ModulationFrequency = 0;          % Modulation frequency (Hz)
rManualConfig.NumChips = 327680;                % Number of chips in the waveform

% Create a single packet:
rPacket.Power = 0;                              % Relative channel power (dBW)
rPacket.DataSource = {'PN9', 1};                % Input message: {'PNX', Seed} or numerical vecto
rPacket.EnableCoding = 'On';                    % Enable channel coding
rPacket.DataRate = 38400;                       % Data rate (bps)
% Create a sequence of 10 packets:
rManualConfig.PacketSequence = repmat(rPacket, 1, 10);

% Add a Pilot Channel:
pich.Enable = 'On';                             % Enable the pilot channel
pich.Power = 0;                                 % Relative channel power (dBW)
pich.DataSource = {'PN9', 1};                   % Input message: {'PNX', Seed} or numerical vecto
pich.EnableCoding = 'On';                       % Enable channel coding
rManualConfig.PilotChannel = pich;              % Add the channel to the waveform configuration

% Add an ACK Channel, but do not enable it:
```

```
ach.Enable = 'Off';                                 % Do not enable the ack channel
ach.Power = 0;                                      % Relative channel power (dBW)
ach.DataSource = {'PN9', 1};                        % Input message: {'PNX', Seed} or numerical vecto
rManualConfig.ACKChannel = ach;                     % Add the disabled channel specification to the w

% Generate waveform:
reverseManualWaveform   = evdoReverseWaveformGenerator(rManualConfig);

% Demonstrate that the above two parameterization approaches are equivalent:
if(isequal(reversePresetConfig, rManualConfig))
    disp([  'Configuration structures generated with and without the ' ...
            'evdoForwardReferenceChannels function are the same.']);
end
```

```
Configuration structures generated with and without the evdoForwardReferenceChannels function are
```

Waveform Comparison

Compare the waveforms generated using both approaches described above and see that the generated waveforms are identical

```
if(isequal(forwardPresetWaveform, forwardManualWaveform))
    disp([  'Forward waveforms generated with and without the ' ...
            'evdoForwardReferenceChannels function are the same.']);
end
```

```
Forward waveforms generated with and without the evdoForwardReferenceChannels function are the sa
```

```
if(isequal(reversePresetWaveform, reverseManualWaveform))
    disp([  'Reverse waveforms generated with and without the ' ...
            'evdoReverseReferenceChannels function are the same.']);
end
```

```
Reverse waveforms generated with and without the evdoReverseReferenceChannels function are the sa
```

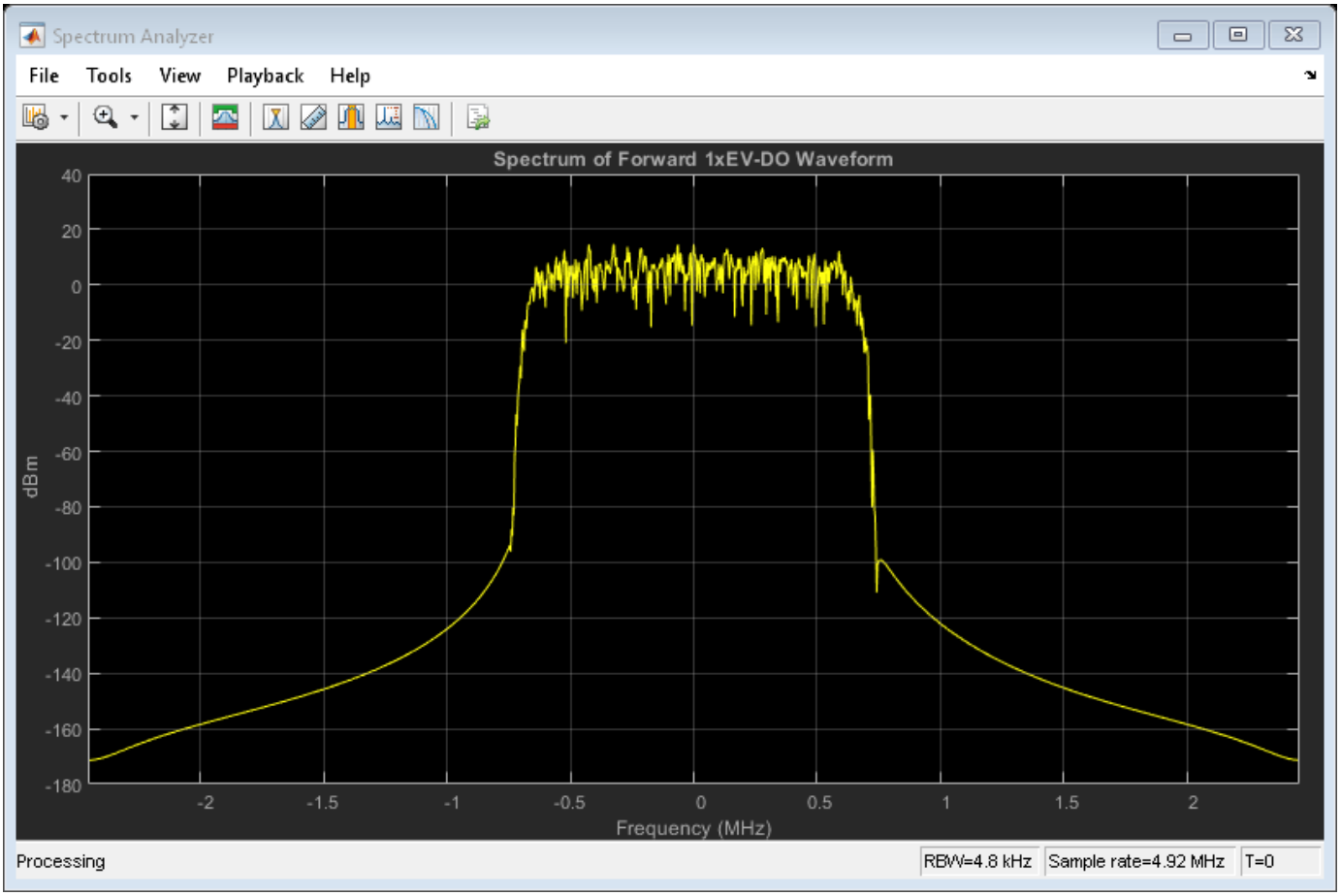
Customizing Configurations

The configuration structures can be customized in order to create a waveform that better suits your objective. For example:

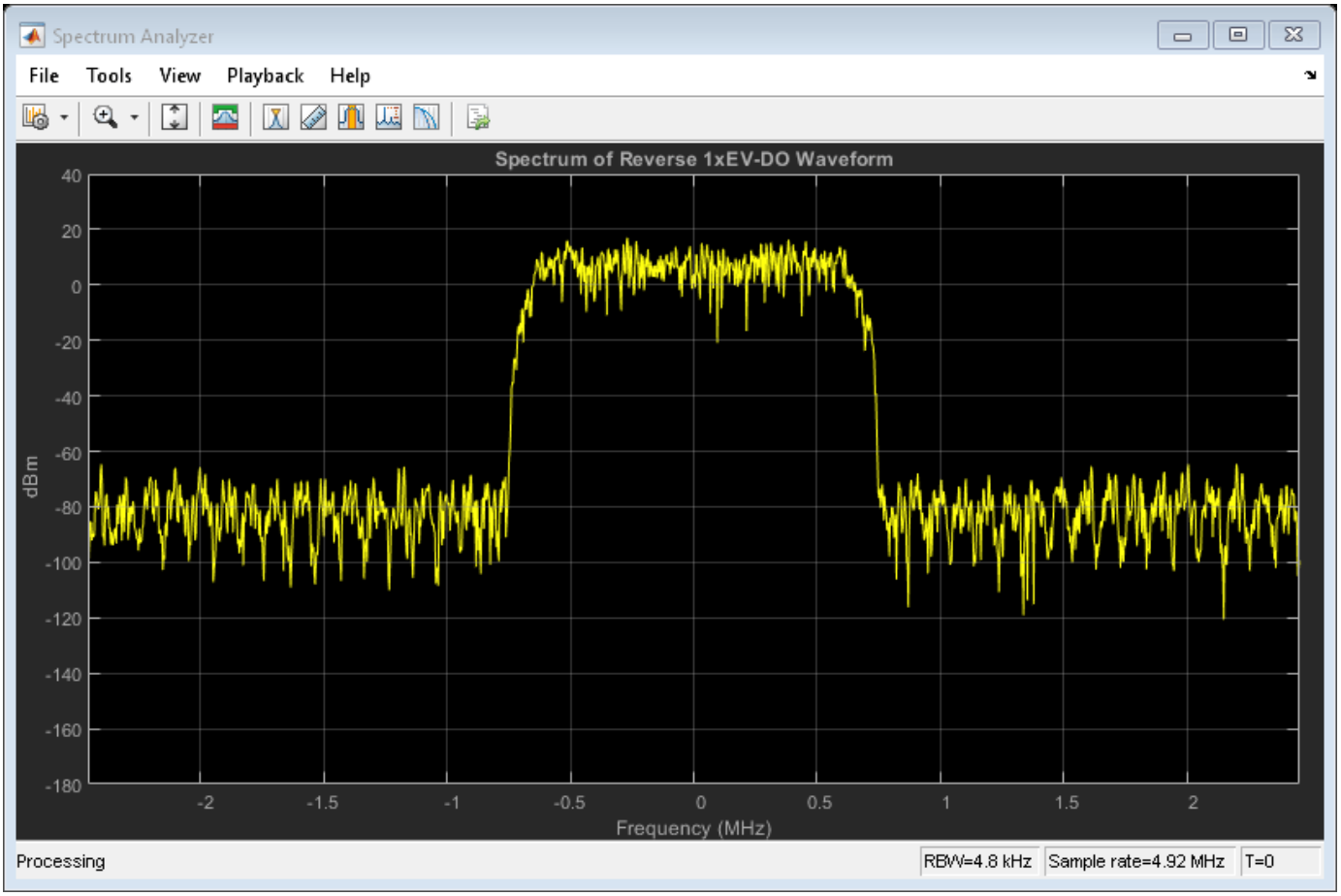
```
rManualConfig2 = rManualConfig;
rPacket.Power = -10;                                % Relative channel power (dBW)
rPacket.DataSource = {'PN23', 1};                   % Input message: {'PNX', Seed} or numerical vecto
rPacket.EnableCoding = 'Off';                       % Enable channel coding
rPacket.DataRate = 38400;                           % Data rate (bps)
rManualConfig2.PacketSequence = repmat(rPacket, 1, 10);

% Regenerate the waveform accounting for the customizations:
reverseManualWaveform2   = evdoReverseWaveformGenerator(rManualConfig2);
```

Plot Spectrum of Generated 1xEV-DO Waveforms

```
chiprate = 1.2288e6;   % Chip rate of the baseband waveform (SR1)
spectrumPlot = dsp.SpectrumAnalyzer('SampleRate', chiprate*fManualConfig.OversamplingRatio);
spectrumPlot.Title = 'Spectrum of Forward 1xEV-DO Waveform';
spectrumPlot.YLimits = [-180,40];
spectrumPlot(forwardManualWaveform);
```




```
spectrumPlot2 = dsp.SpectrumAnalyzer('SampleRate', chiprate*rManualConfig.OversamplingRatio);
spectrumPlot2.Title = 'Spectrum of Reverse 1xEV-DO Waveform';
spectrumPlot2.YLimits = [-180,40];
spectrumPlot2(reverseManualWaveform2);
```



Selected Bibliography

1 C.S0024-A v3.0: cdma2000 High Rate Packet Data Air Interface Specification.

cdma2000 Physical Layer in Simulink

This example demonstrates how the Communications Toolbox™ can be used for: (i) working with standard-compliant cdma2000® waveforms in Simulink® and (ii) building standard-compliant decoder subsystems. Specifically, the model mainly covers the Forward Fundamental Channel (F-FCH) of the downlink channel between a base station and a mobile station for radio configuration 3 and spreading rate 1.

Introduction

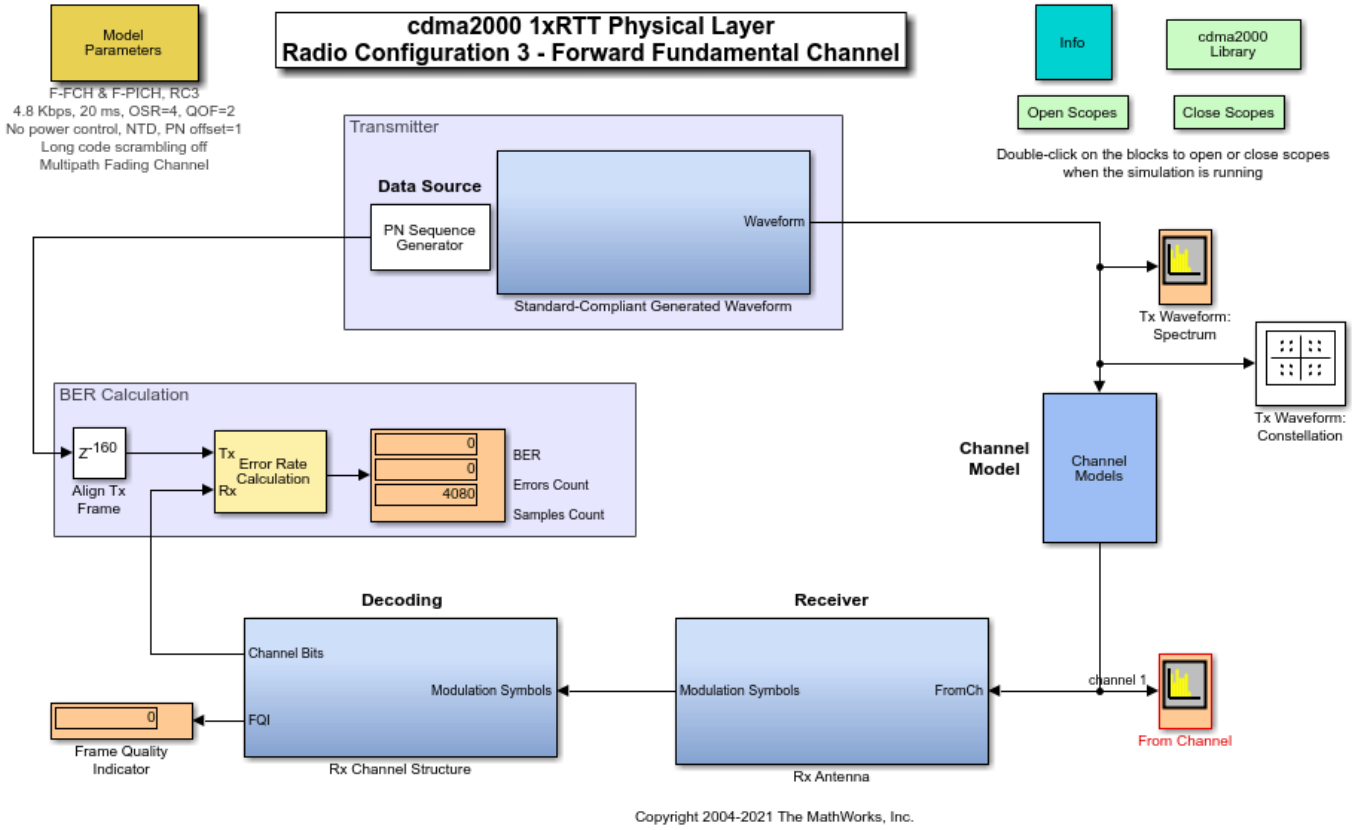
cdma2000 is a terrestrial radio interface for the third generation of wireless communications developed within the framework of the International Mobile Telecommunications (IMT)-2000 standard, as defined by the International Telecommunication Union (ITU). The specifications of the cdma2000 system are being developed by the Third Generation Partnership Project 2 (3GPP2).

The cdma2000 air interface is a direct spread technology. This means that it spreads encoded user data at a relatively low rate over a much wider bandwidth (1.23 MHz for the 1x case), using a sequence of pseudorandom units called chips at a much higher rate (1.2288 Mcps). By assigning a unique code to each user, the receiver, which has knowledge of the code of the intended user, can successfully separate the desired signal from the received waveform.

Structure of the Example

The key components of the forward cdma2000 physical layer are the transmitting base station, the channel, and the mobile station (receiver).

- The base station is represented by a standard-compliant forward waveform, which is generated with the `cdma2000ForwardWaveformGenerator` function and is imported from the MATLAB® Workspace.
- The channel can function as a SISO Fading Channel, an AWGN Channel, or as an empty subsystem.
- The mobile receiver includes the decoder and receiver subsystem, which perform all operations needed for decoding the standard-compliant waveform.



Parameters in the Model

A configuration block labeled "Model Parameters" enables you to configure the generated waveform, as well as the channel model.

Base Station

For waveform generation, you can customize the data rate, the oversampling ratio, the QOF index and the Walsh code. Every customization regenerates a standard-compliant waveform in the MATLAB Workspace. The waveform generation performs the following steps:

- Generating a PN9 bit sequence
- Inserting frame quality indicator bits (CRC)
- Appending tail bits before coding
- Convolutional encoding
- Repetition
- Puncturing
- Block interleaving
- Mapping and scaling
- Spreading by a Walsh code
- Spreading by a QOF (quasiorthogonal function) mask

- Walsh code rotation
- Quadrature scrambling by a PN (pseudonoise) sequence
- Transmit filtering by an oversampled square root raised cosine filter

Channel

The default channel model includes the effects of both multipath Rayleigh fading and additive white Gaussian noise. Alternatively, you can use the channel as a source of Gaussian noise only, or as an empty subsystem. You can configure the channel characteristics using the Model Parameters block in the top left corner of the model.

Receiver

The most important parts of the receiver subsystem are the Rake receiver and the channel estimator.

Besides these two components,the other receiver operations are straightforward inverses of some waveform-generation operations.

Decoder

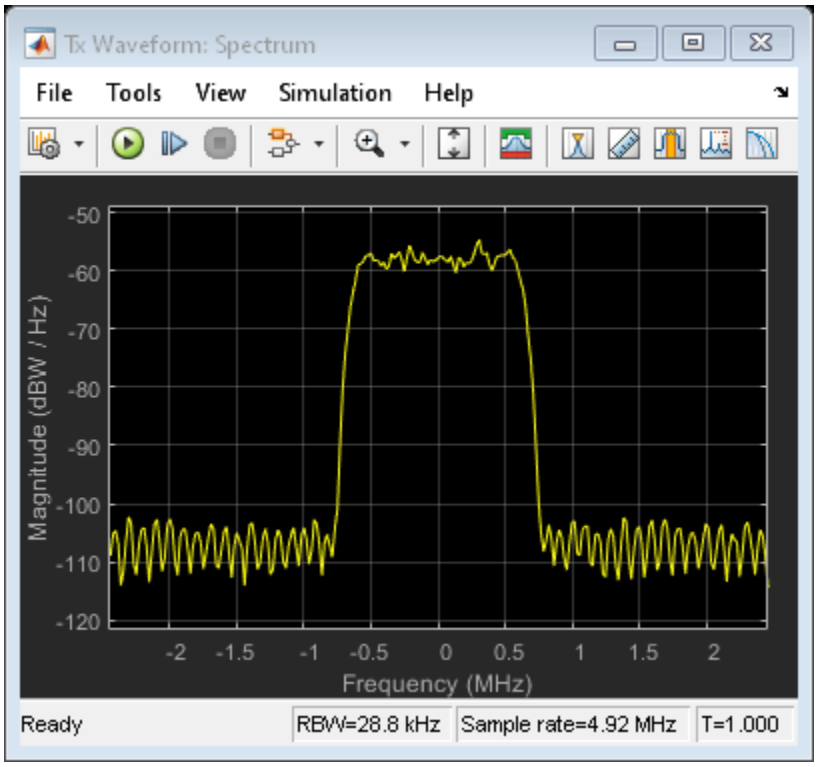
The decoder subsystem conducts the inverse operations of the remaining waveform-generation operations.

Results and Displays

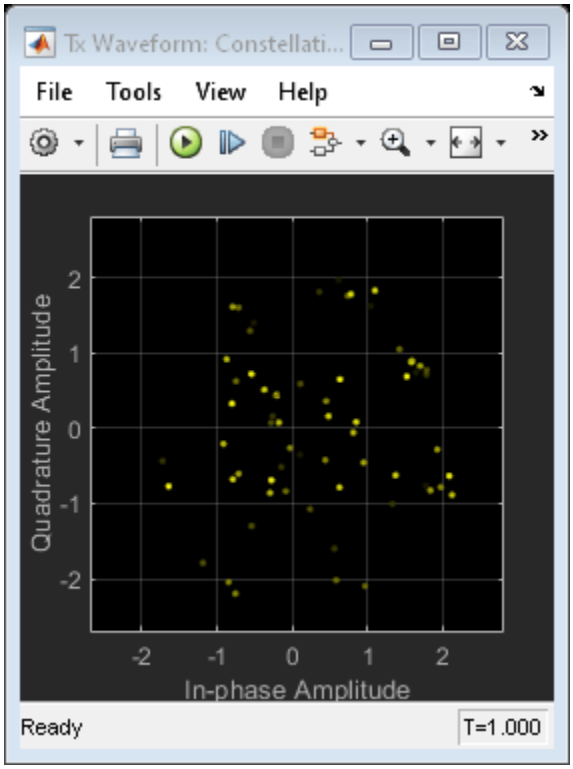
The BER Calculation component compares the decoded signal with the signal of the data source. BER equals zero under all possible configurations for waveform generation, assuming no changes have been made to the model. Notice that signal delay is properly handled and frames are aligned.

To view data graphically, open the scopes by double-clicking the Open Scopes icon. The scopes show the following information:

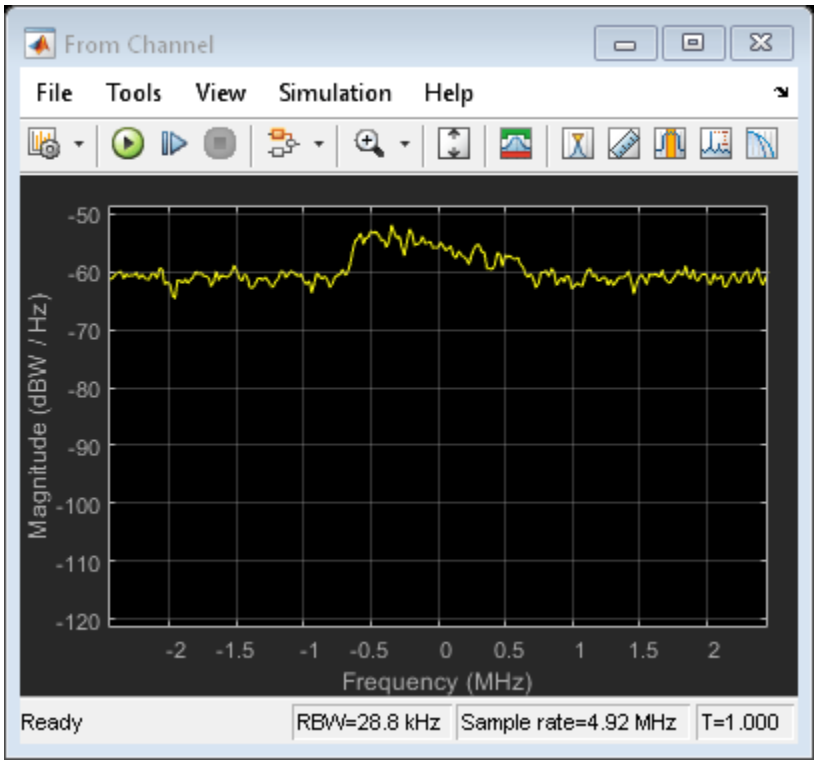
- The 'Tx Waveform: Spectrum' scope shows the power spectrum of the generated waveform.



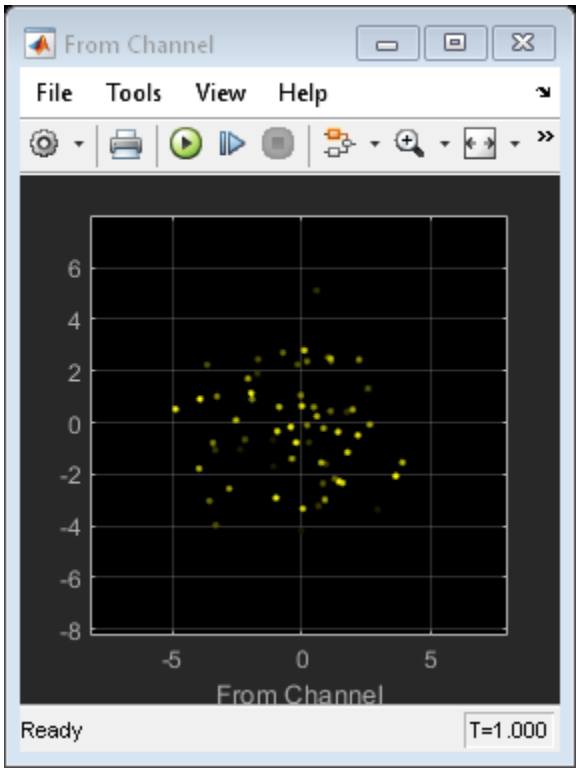
- The 'Tx Waveform: Constellation' scope shows the generated waveform in the I-Q plane. The transmitted signal seems scattered, as a result of spreading.



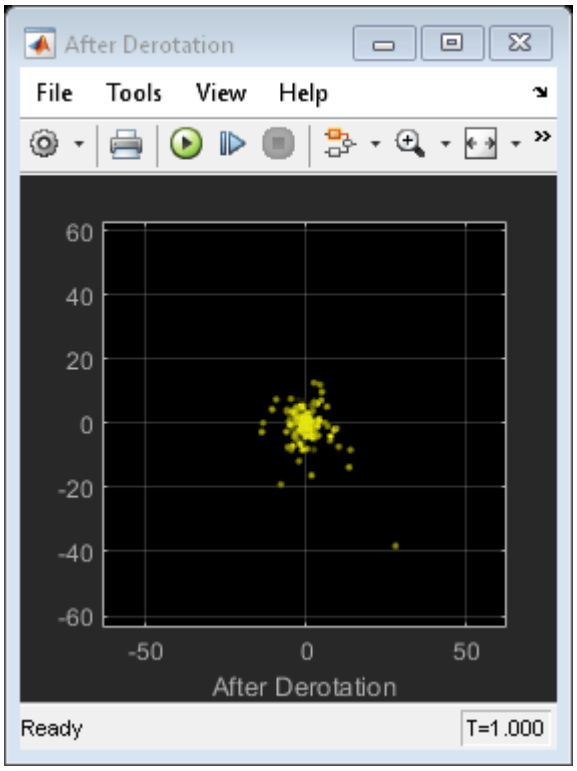
- The 'From Channel' spectrum scope noticeably illustrates the effects of the channel on the received signal.



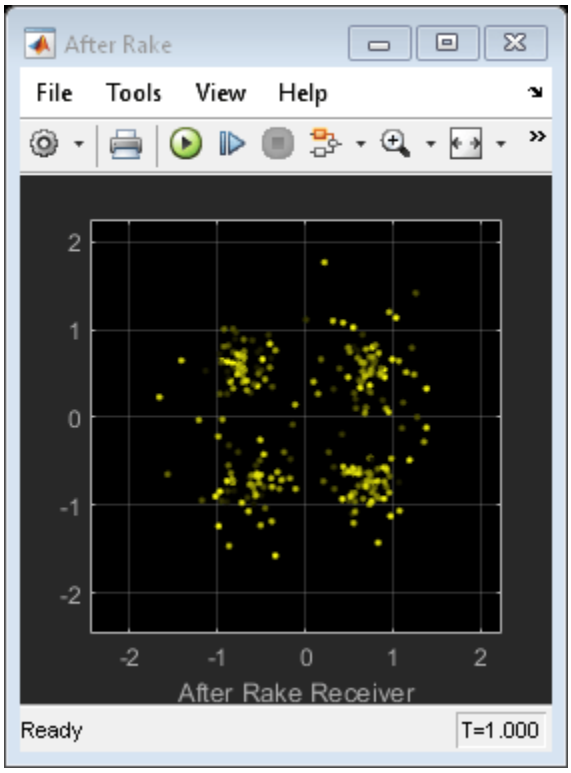
- The 'From Channel' constellation scope shows the I-Q output of the channel. The signal is still spreaded.



- The 'After Derotation' constellation scope shows the data after the receiver subsystem has despreaded the signal and compensated for the phase rotation caused by the channel. The signal still suffers from some effects of the multipath fading channel.



- The 'After Rake' constellation scope shows the output of the rake receiver after the rake receiver has compensated for the attenuation caused by the channel. Even though some bit errors may exist at this stage, these are later on corrected by a powerful decoding operation.



Simulink Techniques Illustrated in the Example

In addition to illustrating a cdma2000 application, this example also illustrates several techniques for modeling in Simulink. In particular, this example shows how you can:

1. Use the Communications Toolbox extensively to implement wireless systems.

2. Represent the architecture of the design using subsystems.

3. Import signals from the MATLAB Workspace.

4. Reuse and share custom-built blocks using a library. To view the library for this example, double-click the cdma2000 Library icon in the top right corner of model.

5. Control the parameters of the simulation using a configuration dialog box.

6. Handle end-to-end delays and perform frame alignment.

Selected Bibliography

[1] C.S0002-F v2.0: Physical Layer Standard for cdma2000 Spread Spectrum Systems.

[2] E.G. Tiedemann, "cdma2000 1X: New Capabilities for CDMA Networks," *IEEE Vehicular Technology Society News*, vol. 48, no. 4, pp. 4-12, Nov. 2001.

[3] TIA/EIA/IS-2000.2-A, Physical Layer Standard for cdma2000 Spread Spectrum Systems, Telecommunications Industry Association, Arlington, VA, March 2000.

Near Field Communication (NFC)

This example shows you how to model communication between two Near Field Communication (NFC) devices.

Introduction

Near Field Communication (NFC) is a standards-based short-range wireless connectivity technology, designed for intuitive and simple communication between two electronic devices. NFC operates at 13.56 MHz center frequency (Fc), at rates ranging from 106 kbps to 424 kbps, and its typical operating range is 10 cm or less. NFC always involves an Initiator and a Target - the Initiator actively generates an electromagnetic field that can power a passive Target.

ISO®/IEC 18092 (Telecommunications and Information Exchange Between Systems - Near Field Communication - Interface and Protocol), also referred to as NFCIP-1 (Near Field Communication - Interface and Protocol Specification), is the governing international Standard for NFC. It is based on ISO/IEC 14443. ISO/IEC 18092 includes two communication modes:

- Passive: The Initiator device generates a carrier field and the Target device answers by modulating the existing field. In this mode, the Target device draws its operating power from the Initiator-provided electromagnetic field.
- Active: Both Initiator and Target device communicate by alternately generating their own fields. A device deactivates its RF field while it is waiting for data. In this mode, both devices typically have power supplies.

Within the two modes of communication there are three modes of operation defined in ISO/IEC 18092:

- Read/Write: In this mode, the NFC device can read data from or write data to any of the supported NFC tags (contactless cards) in a standard NFC data format. The applications include reading information stored in inexpensive NFC tags embedded in labels or smart posters.
- Card Emulation: The NFC device can also act as an NFC tag for other readers. This enables NFC-enabled devices like smart phones to act like smart cards to perform transactions such as payments or ticketing.
- Peer-to-Peer: Two NFC devices can exchange data. The applications include sharing a WiFi or Bluetooth® link, or exchanging data in the form of virtual business cards and photos.

System Setup

This example illustrates the NFC protocol and commands required to transmit data from an Initiator to a Target. The passive communication mode is used here whereby the Initiator provides the electromagnetic field and the Target sends the information back by modulating this field. The Initiator is operating as a writer and the Target as card emulator or tag. The Initiator and Target use the Type A air interface defined in ISO/IEC 14443-2 (Identification cards - Contactless integrated circuit cards - Proximity cards - Part 2: Radio frequency power and signal interface) and are operating at 106 kbps. The Initiator uses Modified Miller coding with 100% ASK, as shown in the Time scope below. The Target generates a subcarrier with frequency 847.5 kHz (Fs), via load modulation, using the Initiator's field and then modulates the data onto the Initiator's carrier frequency using this subcarrier. The Spectrum Analyzer illustrates the load modulation below. To highlight the subcarrier at 847.5 kHz, select Tools->Measurements->Peak Finder in the spectrum analyzer window. The Target uses Manchester coding with 10% ASK as shown in the Time scope below. Note that the time domain signals shown in the two Time scopes are baseband signals i.e the 13.56 MHz carrier signal is stripped out.

The nfcInitiator object represents the Initiator. The UserData property holds the data to be transmitted to the Target. The nfcTarget object represents the Target and ReceivedUserData holds the data received from the Initiator. Due to short range of NFC devices, the system SNR is very high.

```
initiator = nfcInitiator
```

```
initiator =
                    Fc: 13560000
       SamplesPerSymbol: 64
                    t1: 32
              AppLayer: []
              UserData: 'Hello, from MathWorks.'
    EnableVisualization: 1
```

```
target = nfcTarget
```

```
target =
                    Fc: 13560000
                    Fs: 847500
       SamplesPerSymbol: 64
                   UID: '11aa22bb'
              AppLayer: []
       ReceivedUserData: ''
    EnableVisualization: 1
```

```
% Signal to noise ratio, in dB
snrdB = 50;
% Reset the RNG for reproducible results
s = rng(0);
```

Initialization and Anticollision

The Initiator and Target follow initialization and anticollision sequences to establish a communication link. Figure 9 (Initialization and anticollision flowchart for PCD) and Figure 10 (Anticollision loop, flowchart for PCD) in ISO/IEC 14443-3 (Identification cards - Contactless integrated circuit cards - Proximity cards, Part 3: Initialization and anticollision) illustrate the corresponding flowcharts. Section 6 (Type A - Initialization and anticollision) of ISO/IEC 14443-3 describes the commands and protocol in detail. Functions nfcInitialization() and nfcAnticollisionLoop() implement the corresponding sequence of commands and protocol. The example prints the status and actions of Initiator and Target devices, along with important information that is exchanged, to indicate the flow of commands.

Transport Protocol

As described in ISO/IEC 18092, Transport protocol has three parts -

- Activation of protocol: Various protocol parameters, like bit rates, are negotiated and selected during this phase. Section 12.5 (Activation of the protocol) of ISO/IEC 18092 describes this phase in details. The function nfcProtocolActivation() implements the sequence of commands required during this phase.
- Data Exchange Protocol: The information is exchanged during this phase using a half-duplex protocol that supports block oriented data transmission with error handling. See section 12.6 (Data Exchange Protocol) of ISO/IEC 18092 for details. Function nfcDataExchangeProtocol() shows how to implement the exchange of data as prescribed by the ISO/IEC 18092.

- Deactivation of Protocol: After completing data exchange, the Initiator deactivates the protocol and connection with the Target. Function nfcProtocolDeactivation() implements the sequence described in section 12.7 (Deactivation of the protocol) of ISO/IEC 18092.

```
nfcPrint.Message('The message to transmit from Initiator to Target:');
```

```
The message to transmit from Initiator to Target:
```

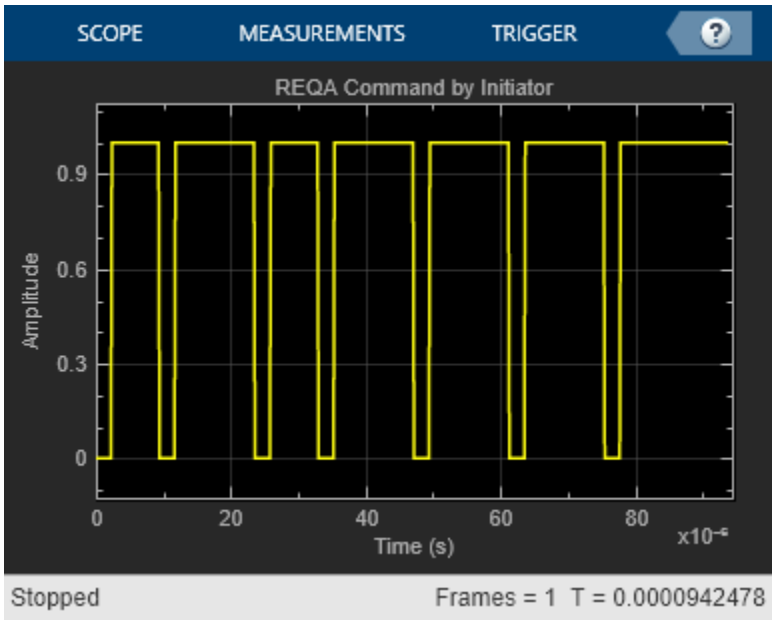
```
nfcPrint.Message(initiator.UserData);
```

```
Hello, from MathWorks.
```

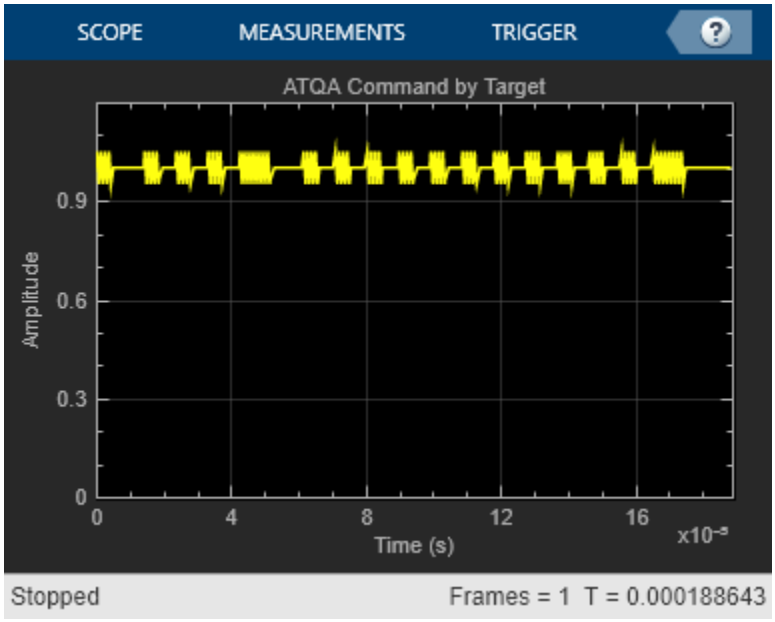
```
nfcPrint.Start();
```

```
Start of NFC Communication between Initiator and Target
```

```
nfcInitialization(initiator, target, snrdB);
```



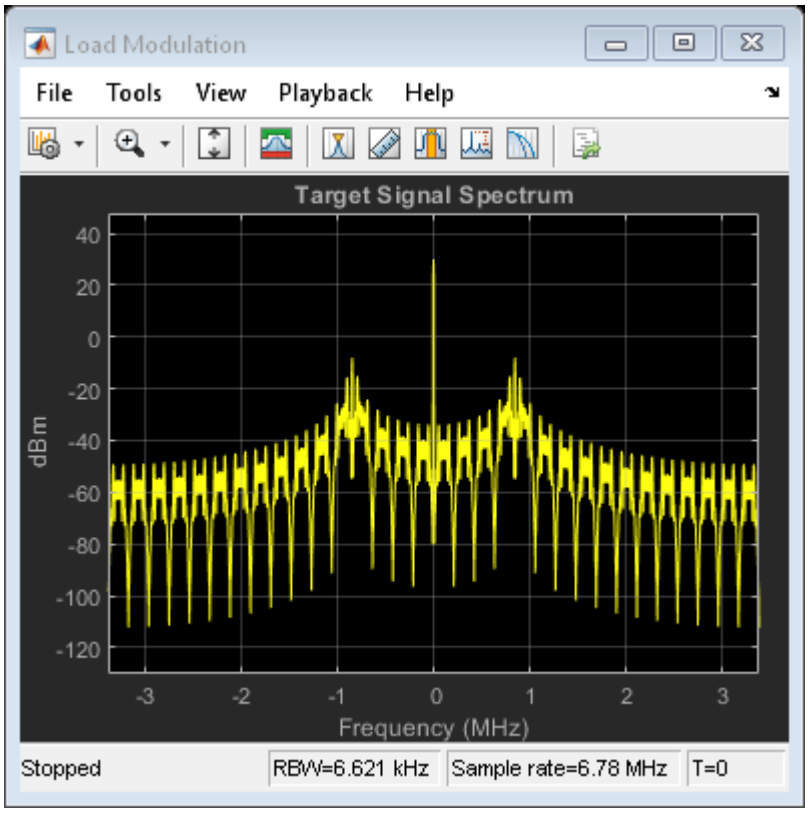

```
Initiator transmitted REQA
Target received REQA
```




```
        Target transmitted ATQA in response to REQA
    Initiator received ATQA
        Target supports bit frame anticollision
        Target's UID size: single
```

```
nfcAnticollisionLoop(initiator, target, snrdB);
```

```
    Start of Anticollision loop
        Cascade Level-1
            Initiator transmitted ANTICOLLISION command
            Target received Cascade Level-1 SEL code
```




```
            Target transmitted full UID
        Initiator received CL1 UID without collision
            Complete UID received: 0x11aa22bb
            Initiator transmitted SELECT command
        Target received Cascade Level-1 SEL code
            Target selection confirmed
            Target transmitted SAK with UID complete flag
        Initiator received SAK
            UID complete. Exit Anticollision loop.
    End of Anticollision loop

    Target compliant with NFCIP-1. Continue with Transport Protocol Activation
```

```
nfcProtocolActivation(initiator, target, snrdB);
```

```
    Start of Transport Protocol Activation
        Initiator transmitted ATR_REQ
        Target received ATR_REQ
            Target transmitted ATR_RES in response to ATR_REQ
        Initiator received ATR_RES
            Initiator transmitted PSL_REQ in response to ATR_REQ
            Selected send rate: 106 Kbps
            Selected receive rate: 106 Kbps
        Target received PSL_REQ
            Target transmitted PSL_RES in response to PSL_REQ
        Initiator received PSL_RES
            PSL_RES validated. All selected rates confirmed
    End of Transport Protocol Activation
```

```
nfcDataExchangeProtocol(initiator, target, snrdB);

    Start of Data Exchange Protocol (DEP)
        Initiator transmitted an Information PDU in DEP_REQ
            Initiator PNI: 0
        Target received an Information PDU in DEP_REQ
            MI chaining not activated in received information PDU
            Received Initiator PNI: 0
            Target PNI: 0
            Target transmitted an Information PDU in DEP_RES in response to DEP_REQ
        Initiator received an Information PDU in DEP_RES
            Received Target PNI: 0
        All data transmitted from Initiator to Target. Exit DEP.
    End of Data Exchange Protocol (DEP)


nfcProtocolDeactivation(initiator, target, snrdB)

    Start of Transport Protocol Deactivation
        Initiator transmitted RLS_REQ
        Target received RLS_REQ
            Target transmitted RLS_RES in response to RLS_REQ
        Initiator received RLS_RES
            Target released
    End of Transport Protocol Deactivation


nfcPrint.End();

End of NFC Communication between Initiator and Target


nfcPrint.Message('The message received by Target from Initiator:');

The message received by Target from Initiator:

nfcPrint.Message(target.ReceivedUserData);

Hello, from MathWorks.

nfcPrint.NewLine;

% Restore RNG state
rng(s);

function nfcInitialization(initiator, target, snrdB)
    % Initialization and anticollision
    % Reference: ISO/IEC 14443-3, section 6

    txREQA = transmitREQA(initiator);
    rxREQA = awgn(txREQA, snrdB, 'measured');

    txATQA = receiveREQA(target, rxREQA);
    rxATQA = awgn(txATQA, snrdB, 'measured');

    [isATQAValid, isCollisionDetected, isTargetCompliant] = ...
        receiveATQA(initiator, rxATQA);

    coder.internal.errorIf(~isATQAValid, 'comm:NFC:InvalidATQA');
```

```
    coder.internal.errorIf(isCollisionDetected, 'comm:NFC:CollisionATQA');
    coder.internal.errorIf(~isTargetCompliant, 'comm:NFC:TargetNotCompliant');
end

function nfcAnticollisionLoop(initiator, target, snrdB)
    % Anticollision Loop
    % Reference: ISO/IEC 14443-3, section 6

    nfcPrint.NewLine;
    nfcPrint.Heading1('Start of Anticollision loop');

    % Start anticollision loop
    cascadeLevel = 1;
    targetRxAC = [];
    nfcPrint.CascadeLevel(cascadeLevel);
    [initiatorTxAC, newCascadeLevel, uidComplete, isoCompliantTarget] = ...
        antiCollisionLoop(initiator, targetRxAC, cascadeLevel);

    while (newCascadeLevel <= 3) && ~uidComplete

        nfcPrint.CascadeLevel(newCascadeLevel, cascadeLevel);
        cascadeLevel = newCascadeLevel;

        targetRxAC = awgn(initiatorTxAC, snrdB, 'measured');
        % Target's anticollision loop
        targetTxAC = antiCollisionLoop(target, targetRxAC);
        initiatorRxAC = awgn(targetTxAC, snrdB, 'measured');
        % Initiator's anticollision loop
        [initiatorTxAC, newCascadeLevel, uidComplete, isoCompliantTarget] = ...
            antiCollisionLoop(initiator, initiatorRxAC, cascadeLevel);
    end

    coder.internal.errorIf(~uidComplete, 'comm:NFC:IncompleteUID');
    coder.internal.errorIf(~isoCompliantTarget, ...
        'comm:NFC:TargetNotCompliantWithNFCIP1');

    nfcPrint.Heading1('End of Anticollision loop');
    nfcPrint.NewLine;
    nfcPrint.Heading1(['Target compliant with NFCIP-1. '...
        'Continue with Transport Protocol Activation']);
end

function nfcProtocolActivation(initiator, target, snrdB)
    % NFCIP-1 Transport Protocol Activation
    % Reference: ISO/IEC 18092, section 12.5

    nfcPrint.NewLine;
    nfcPrint.Heading1('Start of Transport Protocol Activation');

    txATR_REQ = transmitATR_REQ(initiator);
    rxATR_REQ = awgn(txATR_REQ, snrdB, 'measured');

    txATR_RES = receiveATR_REQ(target, rxATR_REQ);
    rxATR_RES = awgn(txATR_RES, snrdB, 'measured');

    txPSL_REQ = receiveATR_RES(initiator, rxATR_RES);
    rxPSL_REQ = awgn(txPSL_REQ, snrdB, 'measured');
    txPSL_RES = receivePSL_REQ(target, rxPSL_REQ);
```

```
    status = receivePSL_RES(initiator, txPSL_RES);
    coder.internal.errorIf(~status, 'comm:NFC:TPActivationFailed');

    nfcPrint.Heading1('End of Transport Protocol Activation');
end

function nfcDataExchangeProtocol(initiator, target, snrdB)
    % Data Exchange Protocol
    % Reference: ISO/IEC 18092, section 12.6

    nfcPrint.NewLine;
    nfcPrint.Heading1('Start of Data Exchange Protocol (DEP)');

    status = nfcDEP(initiator, target, snrdB);
    coder.internal.errorIf(~status, 'nfc:NFC:DEPFailed');

    nfcPrint.Heading1('End of Data Exchange Protocol (DEP)');
    nfcPrint.NewLine;
end

function nfcProtocolDeactivation(initiator, target, snrdB)
    % Transport Protocol Deactivation
    % Reference: ISO/IEC 18092, section 12.7

    nfcPrint.NewLine;
    nfcPrint.Heading1('Start of Transport Protocol Deactivation');

    txRLS_REQ = transmitRLS_REQ(initiator);
    rxRLS_REQ = awgn(txRLS_REQ, snrdB, 'measured');

    txRLS_RES = receiveRLS_REQ(target, rxRLS_REQ);
    rxRLS_RES = awgn(txRLS_RES, snrdB, 'measured');

    status = receiveRLS_RES(initiator, rxRLS_RES);
    coder.internal.errorIf(~status, 'comm:NFC:TPDeactivationFailed');

    nfcPrint.Heading1('End of Transport Protocol Deactivation');
end
```

Exploration

Explore various methods of nfcInitiator and nfcTarget objects to understand various commands and protocols described by NFC standards. Experiment with various system parameters like SNR, UID type (Single or Double), UID value, SamplesPerSymbol to see how they impact the system.

References

1. https://nfc-forum.org/
2. ISO/IEC 14443-2 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 2: Radio frequency power and signal interface
3. ISO/IEC 14443-3 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 3: Initialization and anticollision
4. ISO/IEC 18092 Information technology - Telecommunications and information exchange between systems - Near Field Communication - Interface and Protocol (NFCIP-1)

NFC Application Layer

This example demonstrates exchange of data between application layers of two Near Field Communication (NFC) devices.

Introduction

The example demonstrates a Near Field Communication (NFC) smart poster, which is an NFC feature used by retailers, advertising agencies, transportation authorities, health care providers, and various industries who want to interact with consumers. Since NFC requires the user to take action, the opt-in nature of the technology creates an engaged user leading to a much more meaningful interaction with a brand.

An NFC smart poster is a magazine advertisement, flier, billboard, or other physical medium that includes embedded or affixed NFC tags. When an NFC device is placed within a few centimeters of the NFC tag, information is transferred to the device or a task is launched on the device.

Depending on the smart poster and environment, the consumer can receive targeted information about their current location. Examples of NFC smart poster use include:

- A poster that contains an NFC tag that instructs the receiving NFC device to launch a map application with directions to help a lost tourist find a nearby landmark.
- In a retail setting, a poster that offers coupons, information about a product, or loyalty points. The consumer's phone acts as the loyalty card and stores the information, thus eliminating the need to keep track of multiple cards.

The data to transfer is encoded in the NFC Tag in the NDEF (NFC Data Exchange Format) format and is stored into the Tag data structure. The NDEF is a data format for NFC Devices and Tags to construct a valid NDEF message. The NDEF file consists of NDEF Messages, which consist of NDEF Records. The NDEF format is used to store and exchange information like URI (Uniform Resource Identifier), plain text, encrypted data, image data like GIF and JPEG files, etc.

System Setup

The application layer of the NFC Tag stores the data in the NDEF file. This example illustrates the NFC protocols and commands required to transfer data between the application layers of an NFC Device and an NFC Tag for an application like an NFC smart poster.

NDEF is a lightweight, binary message format that can be used to encapsulate one or more application-defined payloads of arbitrary type and size into a single message construct. Each payload is described by a type, a length, and an optional identifier. Type identifiers may be URIs, MIME media types, or NFC-specific types. NDEF payloads may include nested NDEF messages or chains of linked data chunks of length unknown at the time the data is generated [5 on page 8-0].

```
txURL = 'mathworks.com';
targetRecord = ndefRecord('Type', 'U', 'URIID', '01', ...
    'ActualText', txURL)
```

```
targetRecord =
  ndefRecord with properties:

       TypeNameFormat: 'NFC Forum well-known type'
    IsIDLengthPresent: '0'
        IsShortRecord: '1'
              IsChunk: '0'
```

```
       IsMessageBegin: '1'
         IsMessageEnd: '1'
           TypeLength: ''
        PayloadLength: ''
             IDLength: '00'
                 Type: 'U'
                   ID: ''
              Payload: ''
          StatusBytes: '02'
         LanguageCode: 'en'
                URIID: '01'
           ActualText: 'mathworks.com'
```

The NFC Tag containing the URI in the NDEF file is a Type 4 Tag in this example, as specified in the Type 4 Tag Operation Specification [6 on page 8-0]. An nfcTarget object models this tag. It contains an nfcApp object that models the application layer and contains the data to be exchanged.

```
targetAppLayer = nfcApp();
targetAppLayer.AppData = create(targetRecord)
```

```
targetAppLayer =
  nfcApp with properties:

       AppName: 'D2760000850101'
     CCFileID: 'E103'
   NDEFFileID: 'E104'
          CLA: '00'
          INS: 'A4'
           P1: '04'
           P2: '00'
           Lc: ''
           Le: ''
          SW1: '90'
          SW2: '00'
      AppData: 'D1010E55016D617468776F726B732E636F6D'
```

```
target = nfcTarget('AppLayer', targetAppLayer, 'EnableVisualization', false)
```

```
target =
                    Fc: 13560000
                    Fs: 847500
       SamplesPerSymbol: 64
                   UID: '11aa22bb'
              AppLayer: [1x1 nfcApp]
       ReceivedUserData: ''
   EnableVisualization: 0
```

The consumer device is modeled by an nfcInitiator object, which also contains an nfcApp object to model its application layer.

```
initiator = nfcInitiator('AppLayer', nfcApp(), 'UserData', '', ...
    'EnableVisualization', false)
```

```
initiator =
                    Fc: 13560000
       SamplesPerSymbol: 64
```

```
                    t1: 32
                AppLayer: [1x1 nfcApp]
                UserData: ''
    EnableVisualization: 0
```

```
% Signal to noise ratio, in dB
snrdB = 50;
% Reset the RNG for reproducible results
s = rng(0);
```

The Initialization and Anticollision sequences are as described in “Near Field Communication (NFC)” on page 8-323 example. The application data is exchanged using half-duplex block transmission protocol, as described in ISO®/IEC 14443-4 [4 on page 8-0]. The example prints the status and actions of Initiator and Target devices, along with important information that is exchanged, to indicate the flow of commands.

```
nfcPrint.Message('URL to transmit:');
```

```
URL to transmit:
```

```
nfcPrint.Message(sprintf('%s%s', 'http://www.', txURL));
```

```
http://www.mathworks.com
```

```
nfcPrint.Start();
```

```
Start of NFC Communication between Initiator and Target
```

```
nfcInitialization(initiator, target, snrdB);
```

```
    Initiator transmitted REQA
    Target received REQA
        Target transmitted ATQA in response to REQA
    Initiator received ATQA
        Target supports bit frame anticollision
        Target's UID size: single
```

```
nfcAnticollisionLoop(initiator, target, snrdB);
```

```
    Start of Anticollision loop
        Cascade Level-1
            Initiator transmitted ANTICOLLISION command
            Target received Cascade Level-1 SEL code
                Target transmitted full UID
            Initiator received CL1 UID without collision
                Complete UID received: 0x11aa22bb
                Initiator transmitted SELECT command
            Target received Cascade Level-1 SEL code
                Target selection confirmed
                Target transmitted SAK with UID complete flag
            Initiator received SAK
                UID complete. Exit Anticollision loop.
    End of Anticollision loop

    Target compliant with NFCIP-1. Continue with Transport Protocol Activation
```

```
nfcProtocolActivation(initiator, target, snrdB);
```

```
    Start of Protocol Activation
        Initiator transmitted RATS
        Target received RATS
            Target transmitted ATS in response to RATS
        Initiator received ATS
    End of Protocol Activation
```

```
nfcBlockTransmissionProtocol(initiator, target, snrdB);
```

```
    Start of Half-Duplex Block Transmission Protocol
        Initiator transmitted SELECT command for APP NAME
        Target received SELECT command for APP NAME
            Target transmitted STATUS Bytes for APP NAME
        Initiator received valid STATUS Bytes for APP NAME
            Initiator transmitted SELECT command for CC File
        Target received SELECT command for CC File
            Target transmitted STATUS Bytes for CC File
        Initiator received valid STATUS Bytes for CC File
            Initiator transmitted READ command for CC File
        Target received READ command for CC File
            Target transmitted CCFILE CONTENT and STATUS Bytes for CC File
        Initiator received valid CCFILE CONTENT and STATUS Bytes for CC File
            Initiator transmitted SELECT command for NDEF File
        Target received SELECT command for NDEF File
            Target transmitted STATUS Bytes for NDEF File
        Initiator received valid STATUS Bytes for NDEF File
            Initiator transmitted READ command for NDEF NLEN
        Target received READ command for NDEF NLEN
            Target transmitted NLEN and STATUS Bytes for NDEF
        Initiator received NLEN and valid STATUS Bytes
            Initiator transmitted READ command for NDEF File
        Target received READ command for NDEF File
            Target transmitted NDEF CONTENT and STATUS Bytes for NDEF File
        Initiator received NDEF File content and valid STATUS Bytes
    End of Half-Duplex Block Transmission Protocol
```

```
nfcProtocolDeactivation(initiator, target, snrdB)
```

```
    Start of Protocol Deactivation
        Initiator transmitted DESELECT
        Target received DESELECT
            Target transmitted DESELECT response
        Initiator received DESELECT response
            Target released
    End of Protocol Deactivation
```

```
nfcPrint.End();
```

```
End of NFC Communication between Initiator and Target
```

```
recDataType = getReceivedNDEFType(initiator.AppLayer);
recData = getReceivedNDEFData(initiator.AppLayer);
```

```
if strcmpi(recDataType, 'U')
    nfcPrint.Message('Received URL:');
else
    nfcPrint.Message('Received telephone number:');
end

Received URL:

nfcPrint.Message(recData);

http://www.mathworks.com

nfcPrint.NewLine;

% Restore RNG state
rng(s);

function nfcInitialization(initiator, target, snrdB)
    % Initialization and anticollision
    % Reference: ISO/IEC 14443-3, section 6

    txREQA = transmitREQA(initiator);
    rxREQA = awgn(txREQA, snrdB, 'measured');

    txATQA = receiveREQA(target, rxREQA);
    rxATQA = awgn(txATQA, snrdB, 'measured');

    [isATQAValid, isCollisionDetected, isTargetCompliant] = ...
        receiveATQA(initiator, rxATQA);

    coder.internal.errorIf(~isATQAValid, 'comm:NFC:InvalidATQA');
    coder.internal.errorIf(isCollisionDetected, 'comm:NFC:CollisionATQA');
    coder.internal.errorIf(~isTargetCompliant, 'comm:NFC:TargetNotCompliant');
end

function nfcAnticollisionLoop(initiator, target, snrdB)
    % Anticollision Loop
    % Reference: ISO/IEC 14443-3, section 6

    nfcPrint.NewLine;
    nfcPrint.Heading1('Start of Anticollision loop');

    % Start anticollision loop
    cascadeLevel = 1;
    targetRxAC = [];
    nfcPrint.CascadeLevel(cascadeLevel);
    [initiatorTxAC, newCascadeLevel, uidComplete, isoCompliantTarget] = ...
        antiCollisionLoop(initiator, targetRxAC, cascadeLevel);

    while (newCascadeLevel <= 3) && ~uidComplete

        nfcPrint.CascadeLevel(newCascadeLevel, cascadeLevel);
        cascadeLevel = newCascadeLevel;

        targetRxAC = awgn(initiatorTxAC, snrdB, 'measured');
        % Target's anticollision loop
        targetTxAC = antiCollisionLoop(target, targetRxAC);
        initiatorRxAC = awgn(targetTxAC, snrdB, 'measured');
        % Initiator's anticollision loop
```

```
            [initiatorTxAC, newCascadeLevel, uidComplete, isoCompliantTarget] = ...
                antiCollisionLoop(initiator, initiatorRxAC, cascadeLevel);
        end

        coder.internal.errorIf(~uidComplete, 'comm:NFC:IncompleteUID');
        coder.internal.errorIf(~isoCompliantTarget, ...
            'comm:NFC:TargetNotCompliantWithNFCIP1');

        nfcPrint.Heading1('End of Anticollision loop');
        nfcPrint.NewLine;
        nfcPrint.Heading1(['Target compliant with NFCIP-1. '...
            'Continue with Transport Protocol Activation']);
    end

    function nfcProtocolActivation(initiator, target, snrdB)
        % ISO/IEC 14443-4 Protocol Activation
        % Reference: ISO/IEC 14443-4, section 5

        nfcPrint.NewLine;
        nfcPrint.Heading1('Start of Protocol Activation');

        txRATS = transmitRATS(initiator);
        rxRATS = awgn(txRATS, snrdB, 'measured');

        txATS = receiveRATS(target, rxRATS);
        rxATS = awgn(txATS, snrdB, 'measured');

        status = receiveATS(initiator, rxATS);
        coder.internal.errorIf(~status, 'comm:NFC:ProtocolActivationFailed');

        nfcPrint.Heading1('End of Protocol Activation');
    end

    function nfcBlockTransmissionProtocol(initiator, target, snrdB)
        % Half-duplex Block Transmission Protocol
        % Reference: ISO/IEC 14443-4, section 7

        nfcPrint.NewLine;
        nfcPrint.Heading1('Start of Half-Duplex Block Transmission Protocol');

        nfcTransmissionProtocol(initiator, target, snrdB);

        nfcPrint.Heading1('End of Half-Duplex Block Transmission Protocol');
        nfcPrint.NewLine;
    end

    function nfcProtocolDeactivation(initiator, target, snrdB)
        % Protocol Deactivation
        % Reference: ISO/IEC 14443-4, section 8

        nfcPrint.NewLine;
        nfcPrint.Heading1('Start of Protocol Deactivation');

        txDESLECT = transmitDESELECT(initiator);
        rxDESELECT = awgn(txDESLECT, snrdB, 'measured');

        txDESELECT_RES = receiveDESELECT(target, rxDESELECT);
        rxDESELECT_RES = awgn(txDESELECT_RES, snrdB, 'measured');
```

```
    status = receiveDESELECT_RES(initiator, rxDESELECT_RES);
    coder.internal.errorIf(~status, 'comm:NFC:ProtocolDeactivationFailed');

    nfcPrint.Heading1('End of Protocol Deactivation');
end
```

Exploration

Explore various methods of nfcInitiator, nfcTarget, nfcApp, and ndefRecord objects, and various functions used in this example to understand various commands and protocols described by NFC standards. Experiment with various system parameters like SNR, NDEF messages to see how they impact the system.

References

1. https://nfc-forum.org/

2. ISO/IEC 14443-2 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 2: Radio frequency power and signal interface

3. ISO/IEC 14443-3 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 3: Initialization and anticollision

4. ISO/IEC 14443-4 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 4: Transmission protocol

5. NFC Data Exchange Format (NDEF) Technical Specification 1.0

6. Type 4 Tag Operation Specification Technical Specification 2.0

Bluetooth Full Duplex Voice and Data Transmission

This model shows the full duplex communication between two Bluetooth® devices. Both data packets and voice packets can be transmitted between the two devices:

- Supported voice packet types: HV1, HV2, HV3 and SCORT
- Supported data packet types: DM1

A system parameters block configures the packet type, slot pair, and channel type. Stateflow® is used to implement the acknowledgement scheme for the data packets and the SCORT receiver state machine.

Structure of the Example

A Bluetooth core system consists of an RF transceiver, baseband, and protocol stack. The system offers services that enable the connection of devices and the exchange of a variety of classes of data between these devices. This example is focused on the simulation of a piconet consisting of a master, a slave, and a transmission channel.

This model includes CVSD speech coding, HEC, payload CRC for DM1, FEC, framing, GFSK Modulation, frequency hopping, hop sequence generation, an 802.11b interferer, wave file I/O, BER meters, spectrum, timing, and spectrogram plot.

You can set the system parameters by double-clicking the `Model Parameters` block in the top left. You can toggle instrumentation (spectrum, spectrogram, and timing diagram) by double-clicking the switch. The ARQN display for data transmission can be turned on or off.

Transmitter

The transmitter consists of:

- The controller block (based on BT spec Part B 7.6 ARQ Scheme)
- The payload and FEC block (based on BT spec Part B 7)
- The framing block (based on BT spec Part B 6.1 6.4 and 7.3)
- The radio block (based on BT spec Part A 3.1 Basic Rate)

Receiver

The receiver consists of:

- The radio block (based on BT spec Part A 4.1 Basic Rate)
- The deframing block (based on BT spec Part B 7)
- The controller block(based on BT spec Part B 7)

Channel

The following subsystems are constructed in the Bluetooth Full Duplex library:

- AWGN Channel
- AWGN Channel and 80211b interference
- None (direct connection)

Blocks Used

This model shows the use of the following blocks:

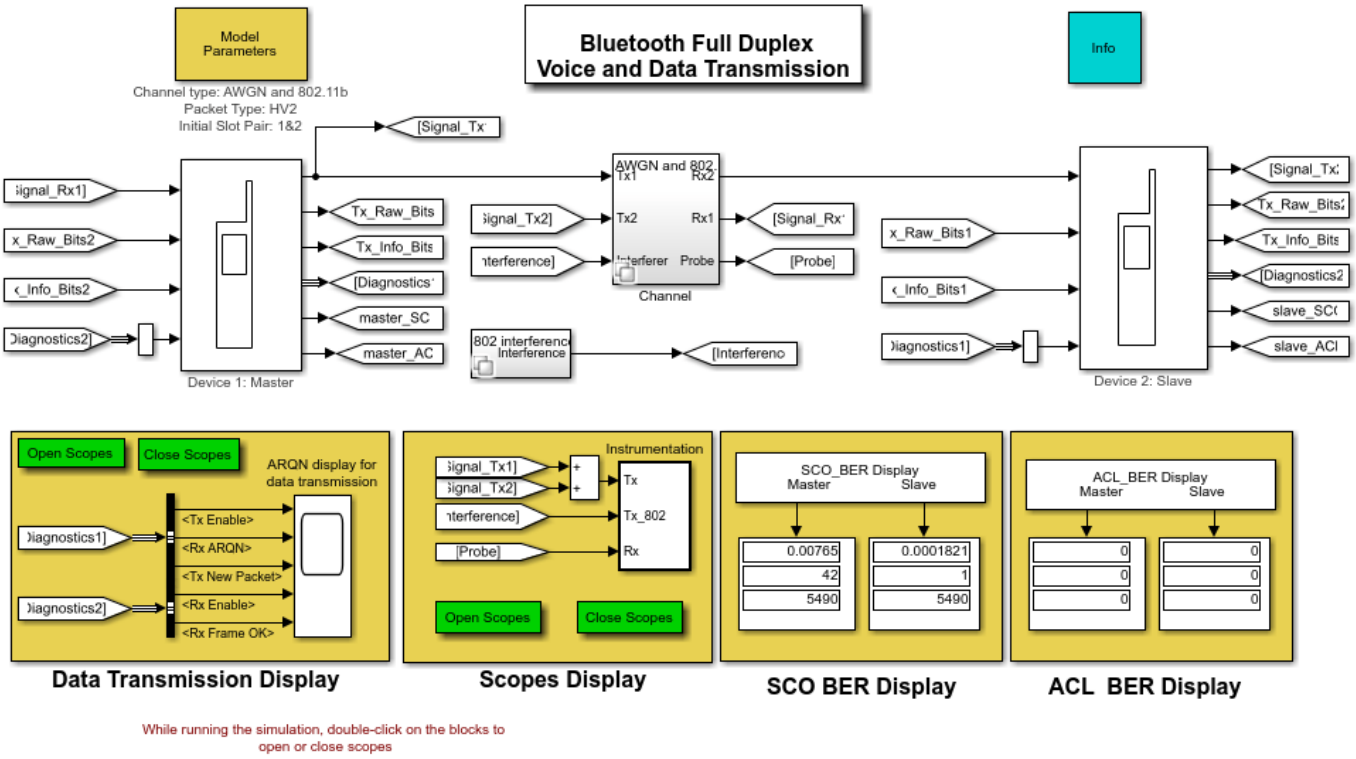
- The `CPM Modulator Baseband` block is used to implement the GFSK (Gaussian frequency shift keying). The Bluetooth radio module uses GFSK, where a binary one is represented by a positive frequency deviation and a binary zero by a negative frequency deviation.
- The `M-FSK Modulator Baseband` block is used to implement the frequency hopping in Bluetooth Radio. The Bluetooth radio accomplishes spectrum spreading by using 79 frequency hops, each displaced by 1 MHz, starting at 2.402GHz and finishing at 2.480GHz.
- The `Free Space Path Loss` block, together with the `AWGN` block and the 802.11b interference subsystem, shows the construction of a transmission channel.
- The `General CRC Generator` block is used for transmitted data CRC calculation.
- The use of the `M-FSK Demodulator` block, the `General CRC Syndrome Detector` block, and the implementation of rate 1/3 and rate 2/3 payload FEC are also included.

The model also uses Stateflow charts to implement:

- The Transmitter Controller
- The Receiver Controller, which decides on the successful reception of a packet by looking at the status of the access code, HEC and CRC

Signals Between the Two Devices

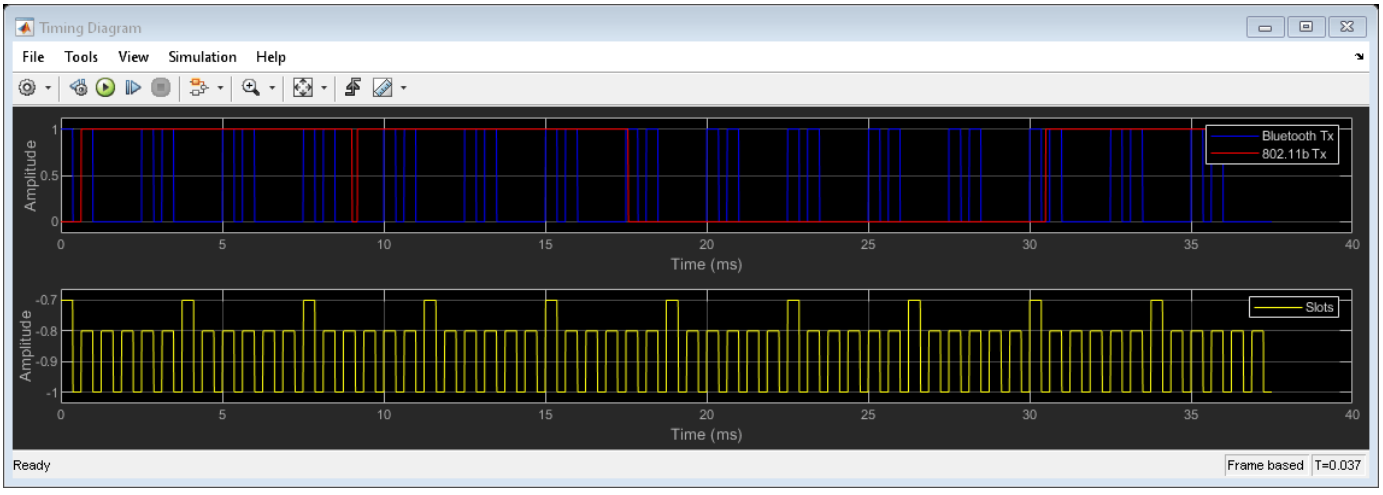
- **Tx_Raw_Bits1**: The master device generates information data randomly, does CRC and FEC payload, and packs them according to the Bluetooth defined format (similarly, **Tx_Raw_Bits2** is for the slave device).
- **Signal_Tx1**: The master device takes Tx_Raw_Bits1 and modulates according to the Bluetooth standard. Signal_Tx1 will be transmitted through the channel (similarly, **Signal_Tx2** is for the slave device).
- **Signal_Rx1**: The raw received signal after AWGN and interference. Signal_Rx1 is fed to the master device for demodulation and detection (similarly, **Signal_Rx2** is for the slave device).
- **Tx_Info_Bits1**: The information data generated by the master with CRC payload but no FEC. Tx_Info_Bits1 is used for SCO BER check on the slave side (similarly, **Tx_Info_Bits2** is for the master device).
- **Diagnostics2**: A collection of frame and packet information for the ACL BER check on the master side (similarly, **Diagnostics1** is for the slave device).
- **master_SCO**: SCO BER information from the master device for display (similarly, **slave_SCO** is for the slave device).
- **master_ACL**: ACL BER information from the master device for display (similarly, **slave_ACL** is for the slave device).
- **Interference**: The interference signal generated from a 802.11b channel.



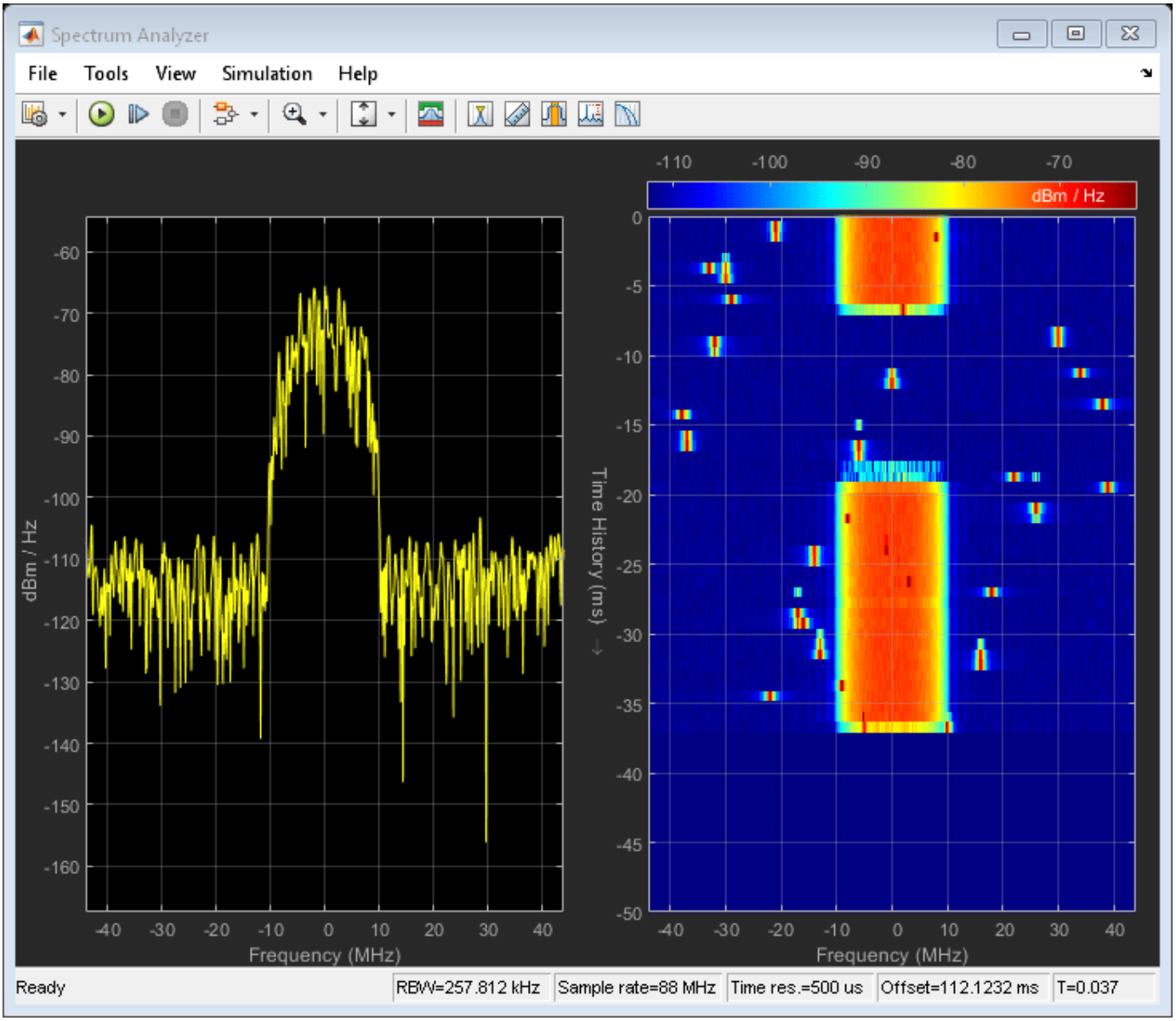
Results and Displays

The scope display includes:

- The timing diagram of the received signal



- The received signal spectrum and the spectrogram of the channel



The Master/Slave BER meters calculate:

- The data BER
- The data throughput

A successful system is decided by:

- The ACL(Asynchronous connection-oriented) BER being zero.
- The SCO (Synchronous connection-oriented) BER (which includes Raw BER, Residual BER, and FER) being within the specifications.

References

Standards can be found at: https://www.bluetooth.com/

See Also

More About

- “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51

DOCSIS Upstream TDMA Link Simulation

This example shows how to implement the physical layer (PHY) of Data Over Cable Service Interface Specification (DOCSIS®) in the upstream TDMA operating mode [1] on page 8-0 [2] on page 8-0 .

Introduction

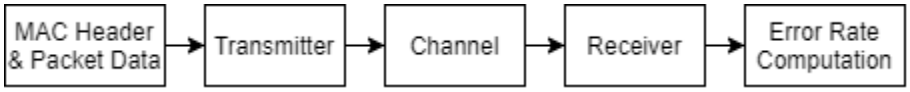
DOCSIS defines the international standards for high-speed data-over-cable systems and specifies a variety of operating modes. This example focuses on the upstream Time Division Multiple Access (TDMA) mode, where Single Carrier Quadrature Amplitude Modulation (SC-QAM) is used. This access mode is compatible with all versions of DOCSIS, including 4.0. The example implements a flexible PHY signal processing chain by incorporating a configuration object that specifies numerous configurable parameters. It also includes the medium access control layer (MAC) header format and simulates data packets compliant with the MAC configuration parameters.

Using features available with Communications Toolbox™ and Signal Processing Toolbox™, the example:

- Models the baseband PHY of a DOCSIS communications system
- Includes helper functions to configure objects and uses these objects to specify, validate, and organize configuration parameters
- Generates statistics to compare the error rate performance of the model to theoretical results.

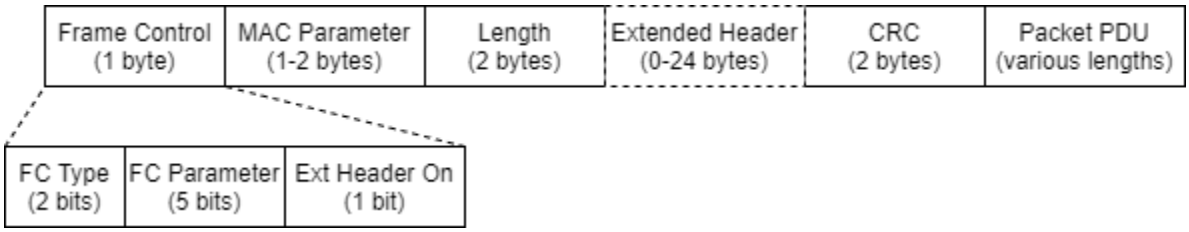
System Model

The high level simulation flow is shown in this image. The individual blocks will be explained in more detail in the following paragraphs.



MAC Frame Structure

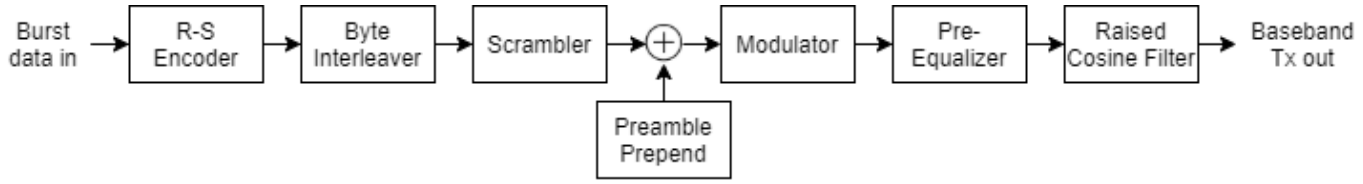
The MAC header format shown in this image complies with DOCSIS [2] on page 8-0 . The Extended Header and Packet Protocol Data Unit (Packet PDU) fields of the frame structure use random bits.



If the Extended Header On field is 1, then MAC Parameter specifies the length of Extended Header in bytes. Otherwise MAC Parameter can be reserved for other usage.

Transmitter Signal Processing

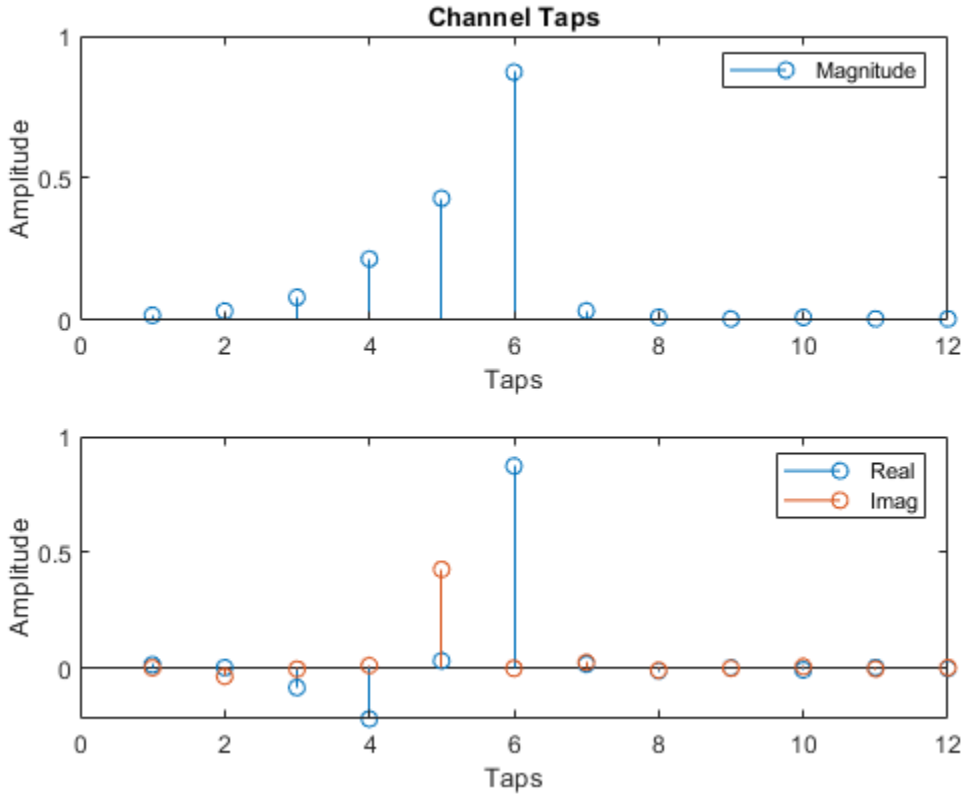
This image shows the transmitter signal processing chain. The input data bits undergo Reed-Solomon encoding, interleaving, scrambling, preamble prepending, SC-QAM, pre-equalization (for more information see Effect of Transmit Pre-Equalizer on page 8-0), and transmit filtering.



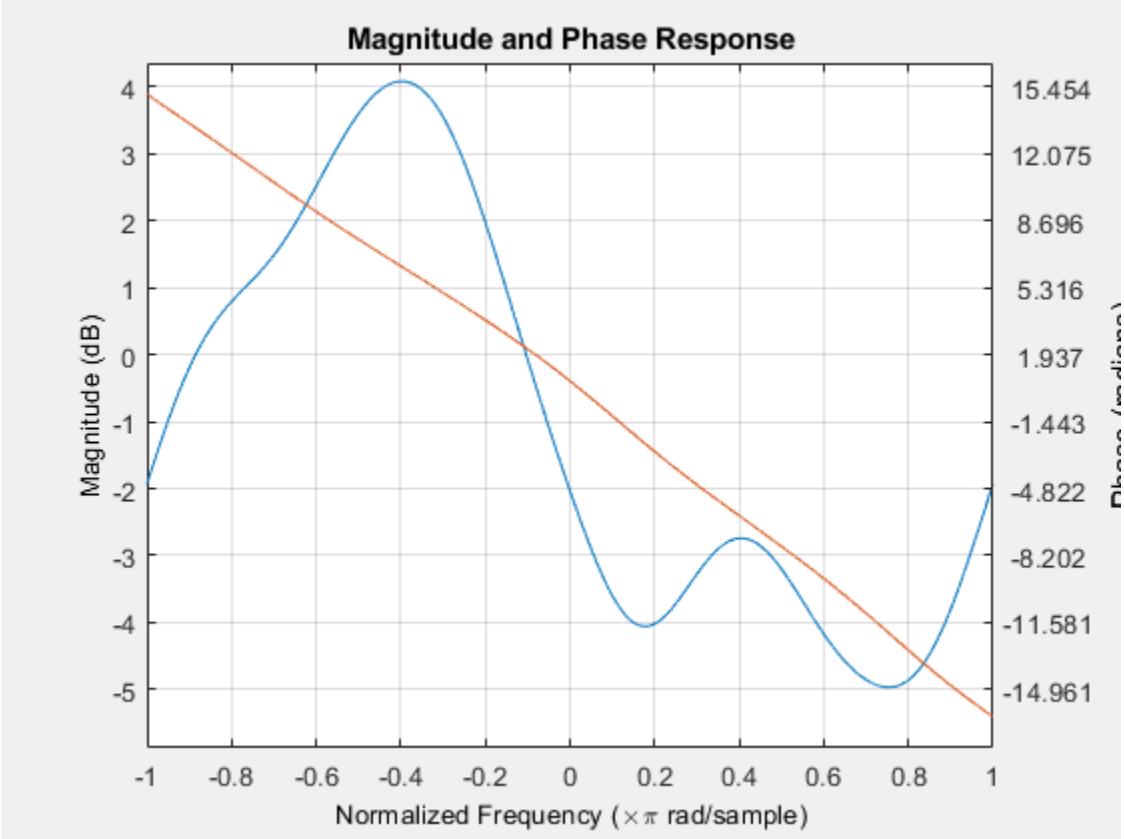
Channel Model

The helper function `helperDocsisChannel` models a multipath channel with a static channel response and stochastic additive white Gaussian noise (AWGN) to reproduce the practical cable channel shown in Figures 40-42 of [3] on page 8-0 . This code filters a unit impulse through the modeled channel and plots the channel taps and frequency response. The magnitude response matches the one shown in Figure 40.

```
% Probe channel with a unit impulse. Pad zeros at the beginning and end to
% account for channel delay.
probeSignal = [zeros(1,12),1,zeros(1,12)];
sampsPerSymbol = 1;
chanTaps = helperDocsisChannel(probeSignal,sampsPerSymbol);
% Remove zero values
chanTaps = nonzeros(chanTaps);
% Time domain tap values
figure
subplot(2,1,1)
stem(abs(chanTaps))
title('Channel Taps')
xlabel('Taps')
ylabel('Amplitude')
legend('Magnitude')
subplot(2,1,2)
stem(real(chanTaps))
hold on
stem(imag(chanTaps))
legend('Real','Imag')
xlabel('Taps')
ylabel('Amplitude')
```

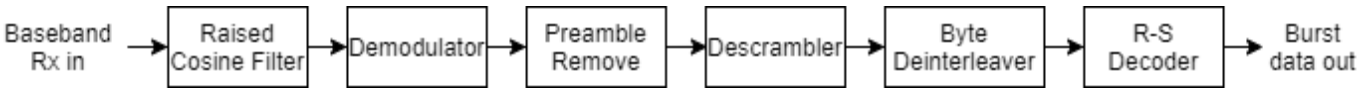



```
% Frequency domain response
fvtool(chanTaps,'Analysis','freq')
title('Magnitude and Phase Response')
```



Receiver Signal Processing

This image shows an ideal receiver signal processing chain which assumes perfect synchronization. The input received baseband symbols undergo processing that reverses the transmitter operations to recover the transmitted data bits and compute the bit error rate (BER).



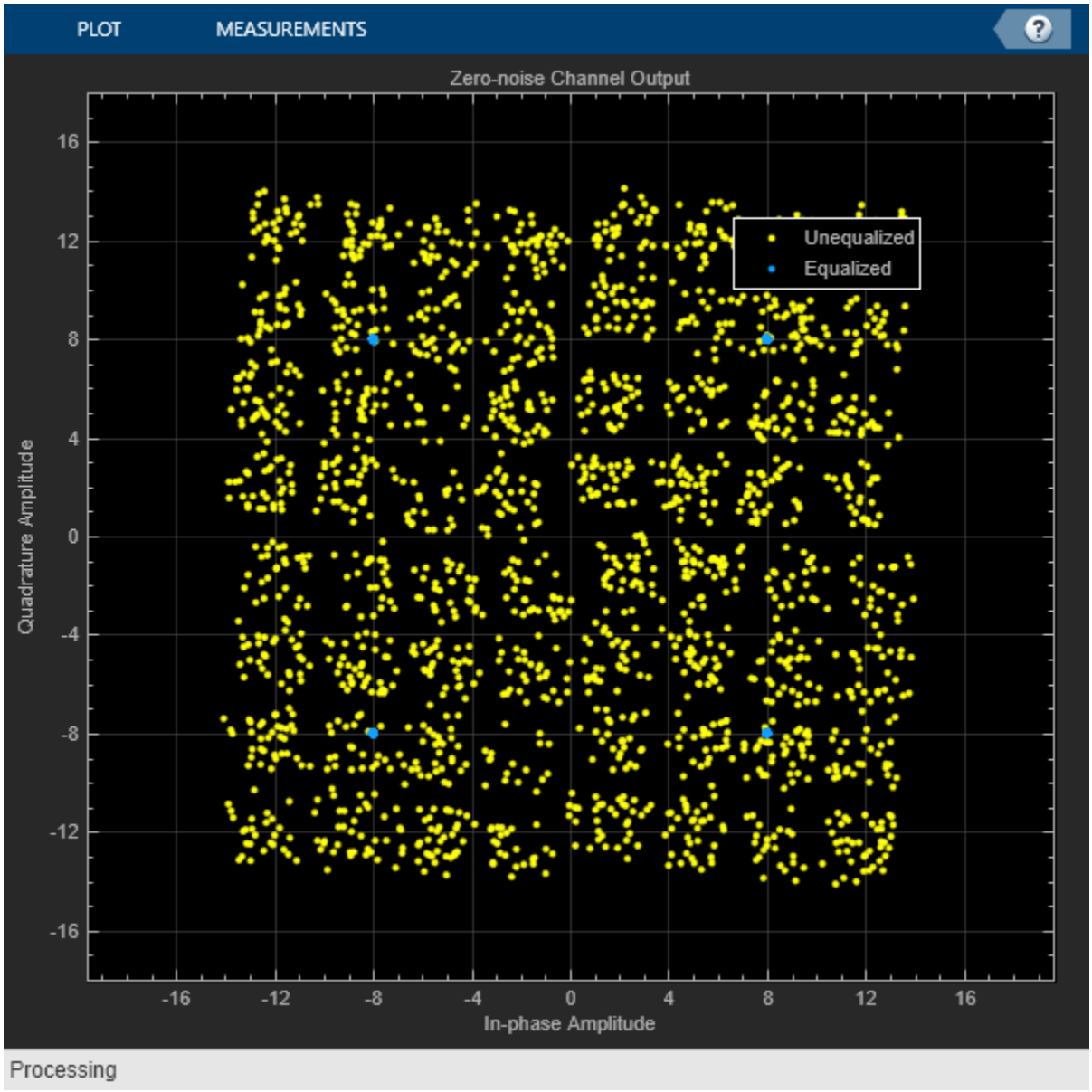
Effect of Transmit Pre-Equalizer

The DOCSIS standard specifies pre-equalization of the transmitter symbols to counter the intersymbol interference (ISI) introduced by the multipath channel. Since a static channel frequency response is used, the pre-equalizer taps at the transmitter are fixed for the duration of the simulation.

This code shows the transmission of a QPSK modulated signal with and without pre-equalization. Both signals are filtered through using the `helperDocsisChannel` function with no AWGN added. The constellation diagram of symbols without pre-equalization applied shows ISI distortion after the channel filtering. The constellation diagram of symbols with pre-equalization applied shows no distortion after the channel filtering.

In fact, since there is no noise in the channel, the equalized symbols align with the reference constellation so well that they can be difficult to see. Toggle the visibility of the two sets of symbols on the constellation diagram by clicking their respective labels in the legend for a better view.

```
% Create a DOCSIS configuration object with the specified parameters. Do
% not use Reed-Solomon encoding or append any preamble bits. 500 bytes of
% data are transmitted in total.
docsisCfg = docsisConfig( ...
    'NumBytes',500, ...
    'RSEnabled',false, ...
    'PreambleLength',0, ...
    'SamplesPerSymbol',1);
% Validate configuration parameters after they're all set
validateConfig(docsisCfg);

% Generate random data bits manually
srcData = randi([0 1],docsisCfg.NumBytes*8,1);

% Get the output from the modulator and pre-equalizer
[~,~,modOut,eqOut] = helperDocsisTx(srcData,[],docsisCfg);

% Pass both signals through the example channel
uneqChanOut = helperDocsisChannel(modOut,docsisCfg.SamplesPerSymbol);
eqChanOut = helperDocsisChannel(eqOut,docsisCfg.SamplesPerSymbol);

% Show received symbols in constellation diagram
constDiagram0 = comm.ConstellationDiagram( ...
    'NumInputPorts',2, ...
    'Title','Zero-noise Channel Output', ...
    'ChannelNames',{'Unequalized','Equalized'}, ...
    'ShowLegend',true, ...
    'XLimits',[-18 18], ...
    'YLimits',[-18 18], ...
    'ShowReferenceConstellation',false);
constDiagram0(uneqChanOut,eqChanOut)
```



End-to-end Link Simulation

The simulated end-to-end communications link complies with transmissions specified by DOCSIS. These helper functions and objects are used:

- `docsisConfig`: configuration object that captures all the parameters affecting waveform generation
- `docsisMACFrameConfig`: configuration object that is a sub-component of `docsisConfig` and specifies the MAC frame structure
- `helperDocsisConstellation`: returns the modulation order (total number of constellation points) and symbol mapping given the modulation name
- `helperDocsisGenerateSourceData`: generates a random burst of data bits, including MAC frame configuration, according to the parameters specified by the configuration object

- `helperDocsisTx`: implements the transmitter signal processing chain; accepts burst data and preamble bits as input, and returns both the baseband transmitter samples and other intermediate block outputs (refer to transmitter block diagram)
- `helperDocsisChannel`: applies the example cable channel with static tap values defined earlier
- `helperDocsisRx`: implements the receiver signal processing chain; takes channel output as input, and returns the decoded data bits as well as other intermediate block outputs (refer to receiver block diagram)

Run the link simulation with a range of Eb/No values. For each Eb/No, generate random source data, pass it through the transmitter and channel, and retrieve the bits at the receiver. These bits are then compared with the source data to check for bit errors. Move on to the next Eb/No value when the bit errors collected or the total bits sent exceeds a specified threshold.

Create configuration object and list all the parameters:

```
docsisCfg = docsisConfig( ...
    'PayloadModulation',16-QAM , ...
    'RSMessageLength',251, ...
    'RSCodewordLength',255)
```

```
docsisCfg =
  docsisConfig with properties:

                        MACFrame: [1x1 docsisMACFrameConfig]
                        NumBytes: 2000
                  ModulationRate: 1280000
                       RSEnabled: 1
                   RSMessageLength: 251
                 RSCodewordLength: 255
               InterleaverNumRows: 4
                   ScramblerSeed: [1 1 0 1 1 1 1 1 0 0 1 1 0 0 1]
                  PreambleLength: 1536
               PreambleModulation: 'QPSK0'
                PayloadModulation: '16-QAM'
                 PreEqualizerTaps: [24x1 double]
                 RaisedCosineSpan: 10
                 SamplesPerSymbol: 2
                       SampleRate: 2560000

   Read-only properties:
                PreEqualizerDelay: 7
          PreambleModulationOrder: 4
    PreambleModulationBitsPerSymbol: 2
        PreambleModulationSymbolMap: [-8.0000 - 8.0000i ... ]
             PayloadModulationOrder: 16
     PayloadModulationBitsPerSymbol: 4
         PayloadModulationSymbolMap: [-4.0000 - 4.0000i ... ]
               SignalPowerPerSample: 80
```

```
% Validate configuration parameters after they're all set
validateConfig(docsisCfg);
```

The example preamble sequence reproduces the sequence specified in Appendix I of [4] on page 8-0 .

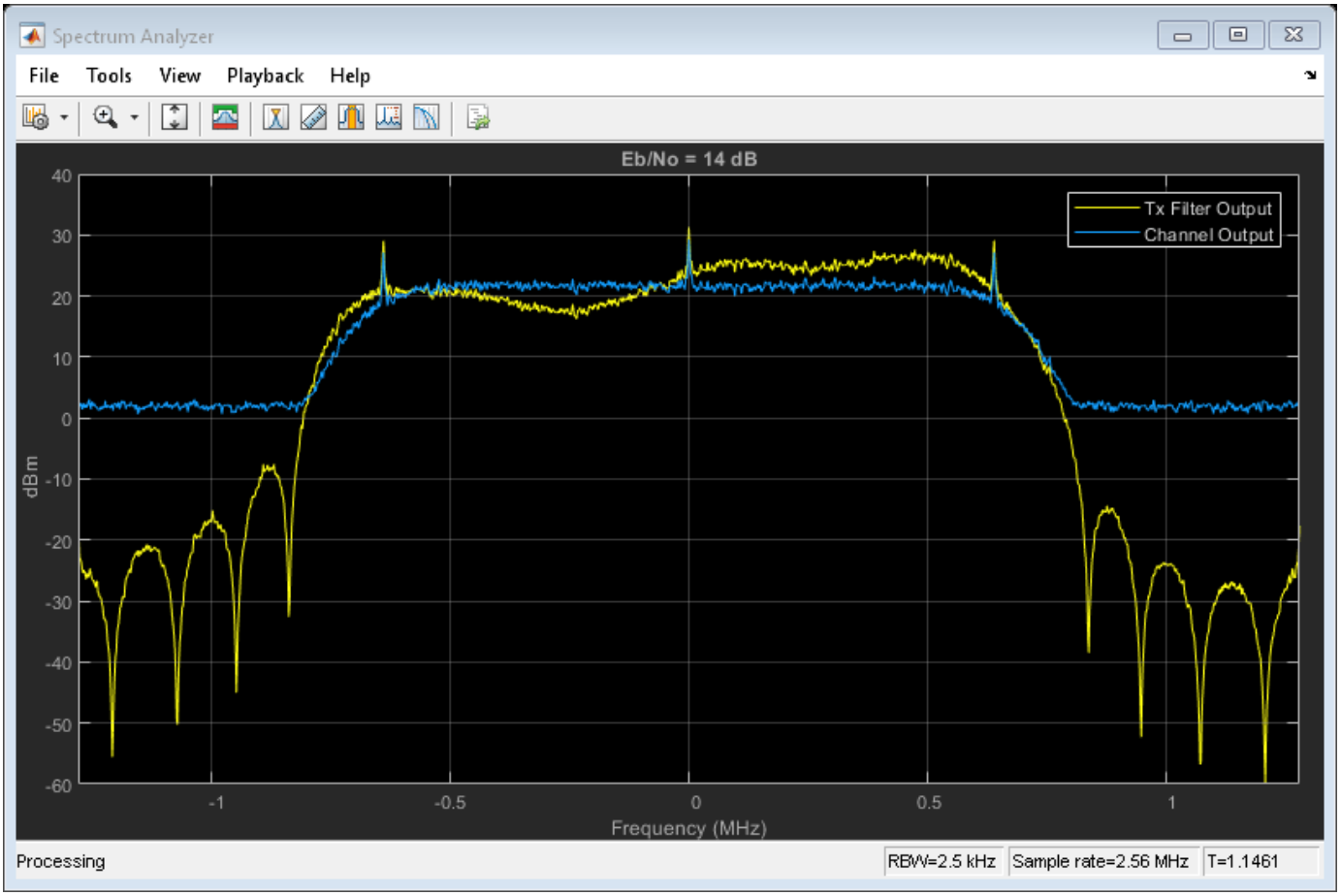
```
load('docsisExamplePreamble.mat')
prmbBits = examplePreamble(end-docsisCfg.PreambleLength+1:end);
```

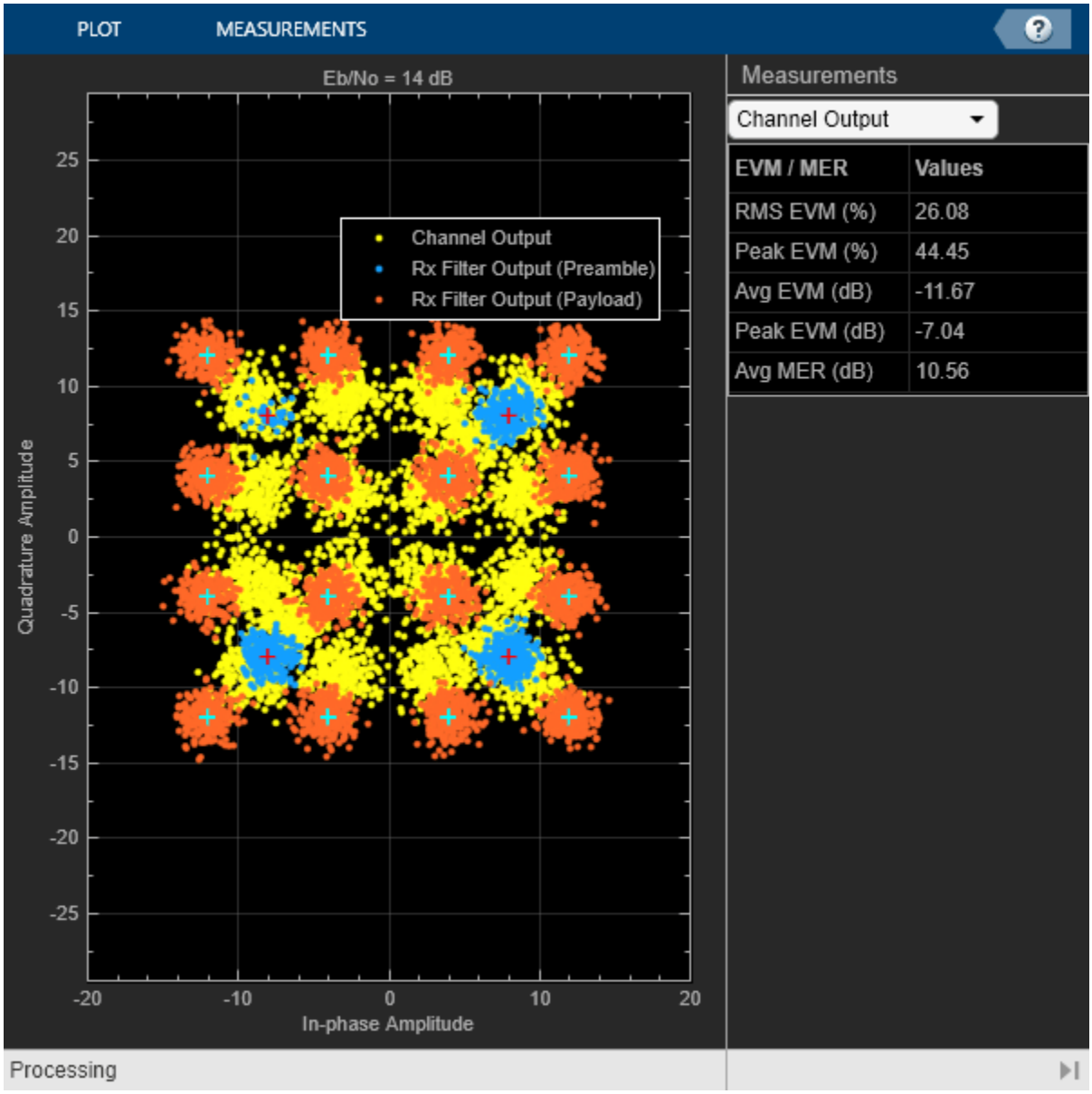
Initialize other relevant variables and visualization scopes:

```
% Max number of bit errors to collect and max number of bits to send
maxErr = 1e3;
maxBits = 1e6;
% Use an Eb/No range that results in meaningful BERs
EbNoRange = helperDocsisEbNoRange(docsisCfg.PayloadModulationOrder);
ber = zeros(size(EbNoRange));
berUncoded = zeros(size(EbNoRange));
% Initialize spectrum analyzer scope and constellation diagram scope
[specAnalyzer,constDiagram] = helperInitializeScopes(docsisCfg);
% Initialize AWGN channel
awgnChan = comm.AWGNChannel( ...
    'BitsPerSymbol',docsisCfg.PayloadModulationBitsPerSymbol, ...
    'SignalPower',docsisCfg.SignalPowerPerSample, ...
    'SamplesPerSymbol',docsisCfg.SamplesPerSymbol);
```

Run the main loop:

```
for i = 1:length(EbNoRange)
    awgnChan.EbNo = EbNoRange(i);
    totalErr = 0; totalBits = 0;
    totalErrUncoded = 0; totalBitsUncoded = 0;
    while totalErr < maxErr && totalBits < maxBits
        % Generate source data bits
        srcData = helperDocsisGenerateSourceData(docsisCfg);
        % Transmitter signal processing
        [txrcOut,modIndexOut] = helperDocsisTx(srcData,prmbBits,docsisCfg);
        % Apply example cable channel and add Gaussian noise
        chanOut = helperDocsisChannel(txrcOut,docsisCfg.SamplesPerSymbol);
        awgnOut = awgnChan(chanOut);
        % Receiver signal processing
        [decoderOut,rxrcOut,demodOut] = helperDocsisRx(awgnOut,docsisCfg);

        % Helper function to show visualization on the scopes
        helperShowScopes(specAnalyzer,constDiagram,txrcOut,rxrcOut,awgnOut, ...
            EbNoRange(i),docsisCfg)

        % Tally bit errors and total bits sent
        [nErr,nBits,nErrUncoded,nBitsUncoded] = helperBitErrors( ...
            srcData,decoderOut,modIndexOut,demodOut,docsisCfg);
        totalErr = totalErr + nErr;
        totalBits = totalBits + nBits;
        totalErrUncoded = totalErrUncoded + nErrUncoded;
        totalBitsUncoded = totalBitsUncoded + nBitsUncoded;
    end
    % Compute BER
    ber(i) = totalErr / totalBits;
    berUncoded(i) = totalErrUncoded / totalBitsUncoded;
end
```


Spectrum Analyzer
File
Tools
View
Playback
Help
Eb/No = 14 dB
Tx Filter Output
Channel Output
dBm
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-1
-0.5
0
0.5
1
Frequency (MHz)
Processing
RBW=2.5 kHz
Sample rate=2.56 MHz
T=1.1461



The spectrum analyzer shows the power spectral density of the signals at the transmit filter output and at the cable channel output. The transmit filter output signal has been pre-equalized and thus its spectrum has the reciprocal shape of the channel response (see Channel Model on page 8-0) in the main lobe. The side lobes and notches are due to raised cosine filtering. After channel filtering, the channel output signal has a flat spectrum in its bandwidth, and the out of band signal power is increased due to AWGN.

The constellation diagram shows three sets of symbols: channel output, receive filter output preamble symbols, and receive filter output payload symbols. The channel output symbols do not align with the reference constellations; after receiver raised cosine filtering, they separate into clusters centered at the reference constellation points. The preamble symbols are always modulated with QPSK, and they may be different from the modulation of the payload symbols. Note that the example preamble bits from DOCSIS are not independently and uniformly distributed -- they result in fewer constellation

symbols in the upper left cluster than the other three clusters. The payload bits, however, are mostly randomly generated, so they result in evenly distributed clusters of points.

Plot BER against Eb/No

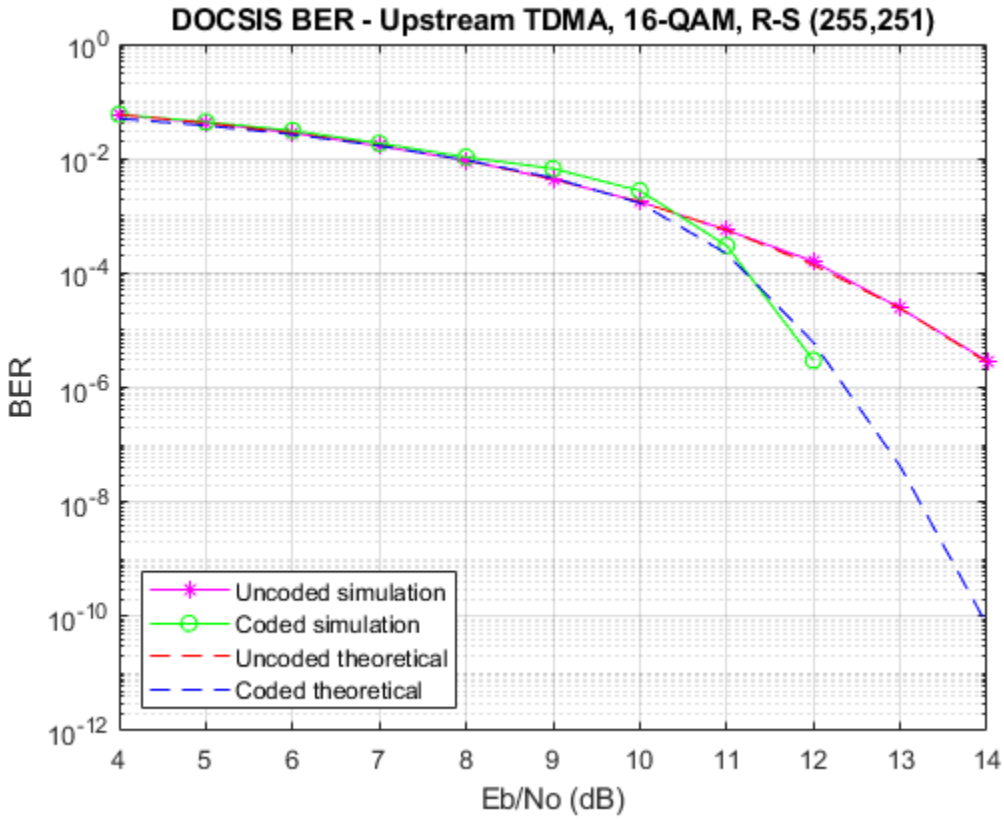
Plot the empirically found BER against the Eb/No values, and compare them with theoretical results. The figure omits the theoretical curves for modulation orders of odd powers of 2 because the DOCSIS standard uses different symbol constellations than the ones assumed in the `bercoding` and `berawgn` functions. For even powers of 2, the functions assume the same constellations as the simulations, and thus the simulation is comparable with theory.

The BER curves show that both the coded and uncoded error rates match the theory reasonably well. For some combinations of (n,k) in Reed-Solomon codes, the coding gain may only appear in the higher Eb/No range, and sometimes the coded BER may even be higher than the uncoded BER at low Eb/No. This is expected behavior of R-S codes.

To get a more accurate simulated BER at higher Eb/No where the errors are very rare, increase the values of `maxErr` and `maxBits` in the previous section, and rerun the simulation. This allows the system to collect more bit errors for each Eb/No. If no errors occur at an Eb/No value, the BER curve will omit that data point.

```
% Find theoretical uncoded and coded BER
berUncodedTheoretical = berawgn(EbNoRange, ...
    'qam',docsisCfg.PayloadModulationOrder);
% Theoretical BER with R-S coding is only available when the codeword
% length is of the form 2^m-1.
if mod(log2(docsisCfg.RSCodewordLength+1),1) == 0
    berTheoretical = bercoding(EbNoRange,'RS','hard', ...
        docsisCfg.RSCodewordLength,docsisCfg.RSMessageLength, ...
        'qam',docsisCfg.PayloadModulationOrder);
else
    berTheoretical = [];
end

% Plot the curves
figure
semilogy(EbNoRange,berUncoded,'m*-')
hold on
semilogy(EbNoRange,ber,'go-')
legendText = {'Uncoded simulation','Coded simulation'};
if mod(docsisCfg.PayloadModulationBitsPerSymbol,2) == 0
    semilogy(EbNoRange,berUncodedTheoretical,'r--')
    legendText{end+1} = 'Uncoded theoretical';
    if ~isempty(berTheoretical)
        semilogy(EbNoRange,berTheoretical,'b--')
        legendText{end+1} = 'Coded theoretical';
    end
end
grid on

if docsisCfg.RSEnabled
    title(sprintf('DOCSIS BER - Upstream TDMA, %s, R-S (%d,%d)', ...
        docsisCfg.PayloadModulation, ...
        docsisCfg.RSCodewordLength,docsisCfg.RSMessageLength))
else
    title(sprintf('DOCSIS BER - Upstream TDMA, %s, uncoded', ...
        docsisCfg.PayloadModulation))
```

```
end
xlabel('Eb/No (dB)')
ylabel('BER')
legend(legendText,'Location','southwest')
```



Further Exploration

Alter the parameters of `docsisCfg` to see how they affect the output. For instance, alter the modulation and coding rate and rerun the simulations to see what effect they have on the system BER performance. Alter the raised cosine filter span, samples per symbol, and sample rate to see how they affect the visualization.

References

[1] CM-SP-PHYv4.0-I02-200429: Data-Over-Cable Service Interface Specifications DOCSIS® 4.0; Physical Layer Specification. Cable Television Laboratories, Inc., 2019-2020.

[2] CM-SP-MULPIv4.0-I02-200429: Data-Over-Cable Service Interface Specifications DOCSIS® 4.0; MAC and Upper Layer Protocols Interface Specification. Cable Television Laboratories, Inc., 2019-2020.

[3] CM-GL-PNMP-V03-160725: DOCSIS® Best Practices and Guidelines; PNM Best Practices: HFC Networks (DOCSIS 3.0). Cable Television Laboratories, Inc., 2010-2016.

[4] CM-SP-PHYv3.0-C01-171207: Data-Over-Cable Service Interface Specifications DOCSIS® 3.0; Physical Layer Specification. Cable Television Laboratories, Inc., 2006-2017.

ATSC Digital Television

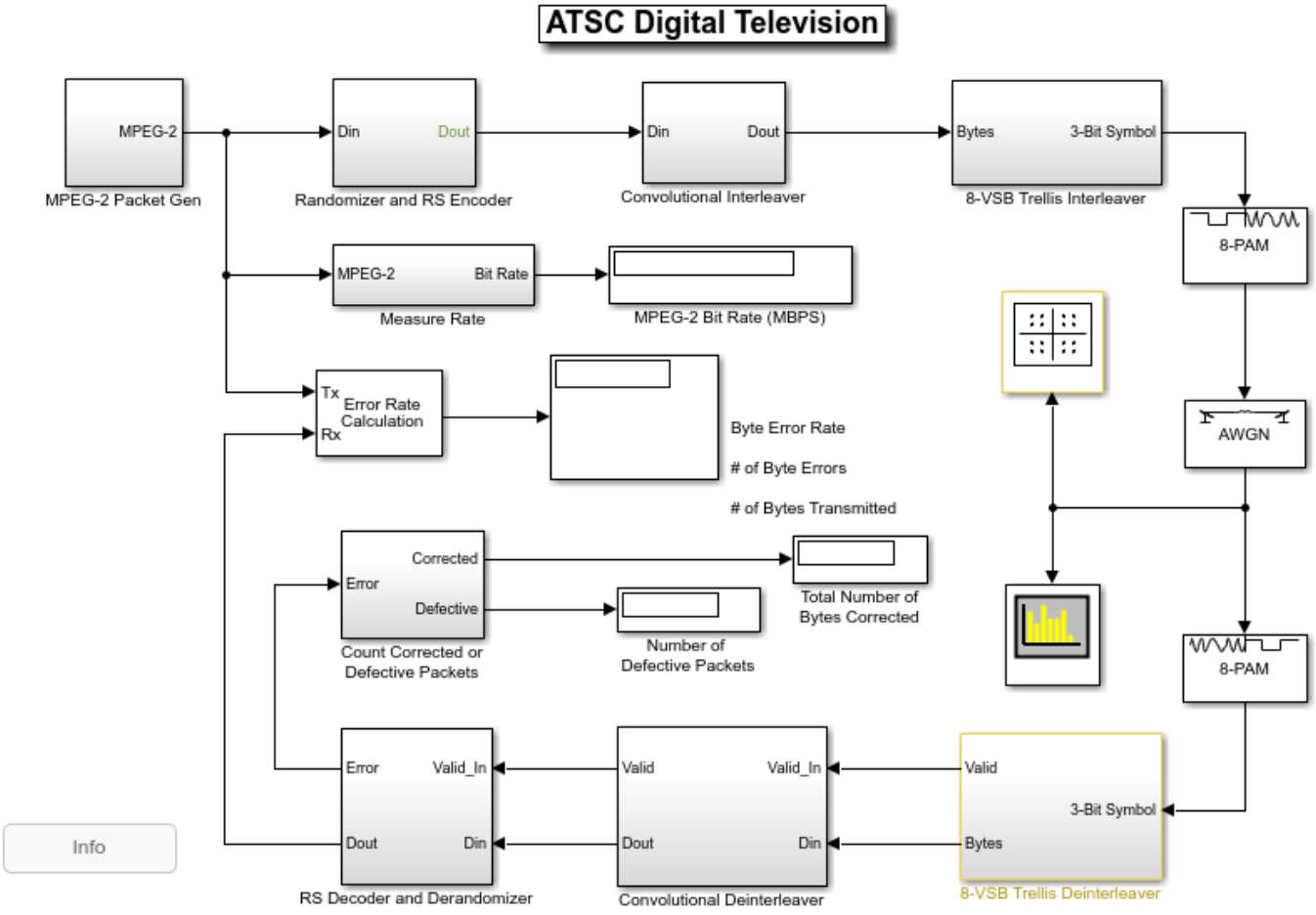
This model shows the vestigial sideband modulation with 8 discrete amplitude levels (8-VSB) transmission subsystem of the Advanced Television Systems Committee (ATSC) digital television standard [1]. The standard describes the characteristics of the U.S. advanced television system that is designed to transmit high-quality video, audio, and ancillary data within a single 6 MHz terrestrial television broadcast channel.

The purpose of this example is to:

- Model the primary portions of a Main Service 8-VSB transmitter with MPEG-2 Transport Packet data as inputs
- Model the primary portions of a possible Main Service 8-VSB receiver design
- Generate error statistics including number of corrected bytes, number of defective packets and byte error rate

Structure of the Example

The model consists of MPEG-2 Transport Packet generation, transmitter baseband processing, AWGN Channel, receiver baseband processing, and error rate calculation. The following sections describe each subcomponent in detail.



MATLAB® Workspace Variable Definitions

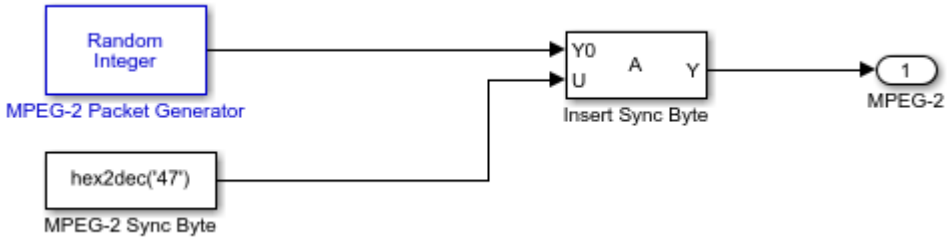
When the model is first loaded, it creates a MATLAB workspace variable `prmATSC`. This structure variable contains fields that specify the block parameters in the model. This variable is cleared when the model is closed.

```
prmATSC =

  struct with fields:

         MPEG2PacketLen: 188
          RSCodewordLen: 207
            BitsPerByte: 8
          BitsPerNibble: 2
         NibblesPerByte: 4
       NibblesPerGroup: 48
     NibblesPerSegment: 828
      SegmentsPerField: 313
       RSPrimitivePoly: [1 0 0 0 1 1 1 0 1]
       RSGeneratorPoly: [1 152 185 240 5 111 99 6 220 112 150 69 36 ... ]
```

```
      IntlvrNumShiftRegs: 52
      IntlvrShiftRegStep: 4
     DeintlvrAlignDelay: 156
       DeintlvrPktDelay: 52
       NumTrellisCoders: 12
         TraceBackDepth: 8
  TrellisDecAlignDelay: 159
    TrellisDecPktDelay: 2
            SymbolRate: 1.0762e+07
               MPEG2BPS: 1.9393e+07
          MPEG2PktRate: 1.2894e+04
      ChannelSampleTime: 9.3666e-08
            PAMSigPower: 4.5826
                   EsNo: 10
```

MPEG-2 Data Source

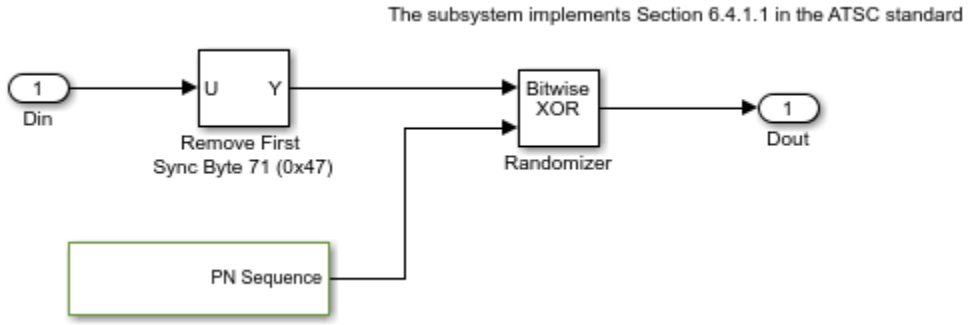
The MPEG-2 Transport Packet is a randomly generated 188-byte vector with the first byte replaced by the sync byte 0x47 (Hexadecimal).



Transmitter Baseband Processing

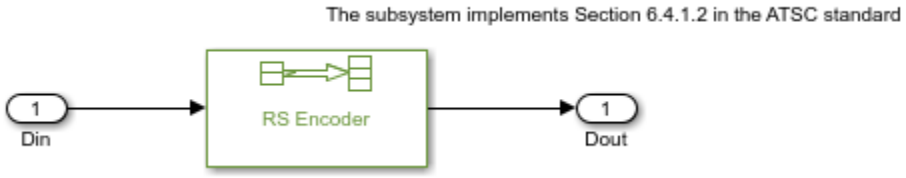
- Randomizer

This subsystem corresponds to Section 6.4.1.1 in [1]. The MPEG-2 sync byte should not be randomized and encoded, and hence is thrown away before the XOR operation. The pseudo random byte sequence that scrambles input data bytes is re-initialized at the beginning of each Data Field. In this model, each Data Field consists of 312 Data Segments because the Data Field Sync segment is not modeled.



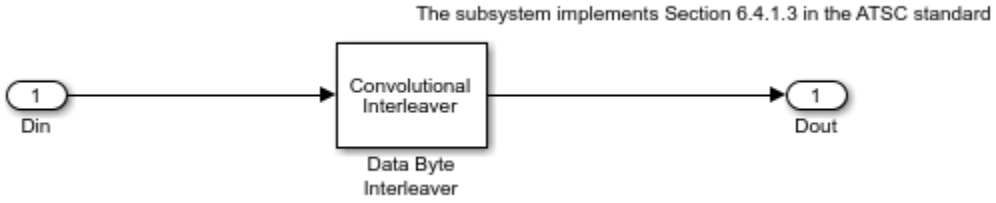
- Reed-Solomon Encoder

This subsystem corresponds to Section 6.4.1.2 in [1]. The (207, 187) Integer-Input RS Encoder block adds 20 parity bytes to the input packet and produces an output of 207 bytes per frame. This allows up to 10 erroneous bytes per transport packet to be corrected by the corresponding Integer-Output RS Decoder block at the receiver.



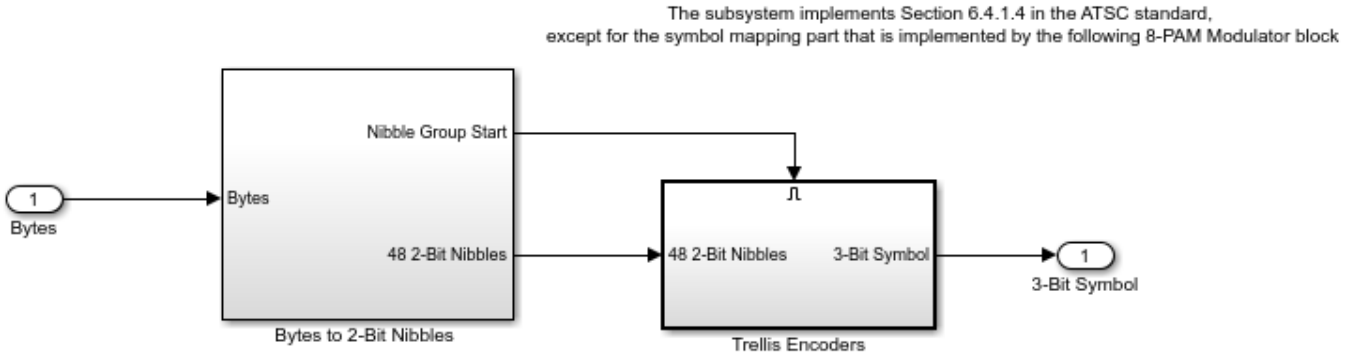
- Convolutional Interleaver

This subsystem corresponds to Section 6.4.1.3 in [1]. The Convolutional Interleaver block interleaves the bytes from 52 Data Segments (intersegment), which is one-sixth (1/6) of a Data Field. The transmitter synchronizes the interleaver to the first data byte of each Data Field.



- Trellis Interleaver

This subsystem, together with the subsequent M-PAM Modulator Baseband block, corresponds to Section 6.4.1.4 in [1]. It creates serial 3-bit outputs from parallel bytes by feeding every two bits of each data byte through one of 12 two-thirds (2/3) rate Convolutional Encoder blocks. Each byte produces four 3-bit outputs and the implementation processes every 12 bytes as a group. A block controls which Convolutional Encoder processes which two bits in a group. A complete conversion of parallel bytes to serial bits needs four Data Segments, i.e., 828 data bytes, to produce 3312 3-bit outputs from the 12 encoders, and each encoder processes 69 data bytes. Each Data Field needs 312/4 = 78 conversion operations.



- 8-PAM Constellation Mapping

The M-PAM Modulator Baseband block corresponds to the symbol mapper portion of the Figure 6.8 in [1]. It maps 3-bit integer inputs to symbols on an 8-level one-dimensional real constellation with values [-7 -5 -3 -1 1 3 5 7].

AWGN Channel

The AWGN Channel block uses the `Signal to noise ratio (Es/No)` mode. Signal power and symbol period have been calculated and stored in the workspace variable `prmATSC`. The Es/No value is set to 10 dB, which produces a byte error rate of approximately 0.0039.

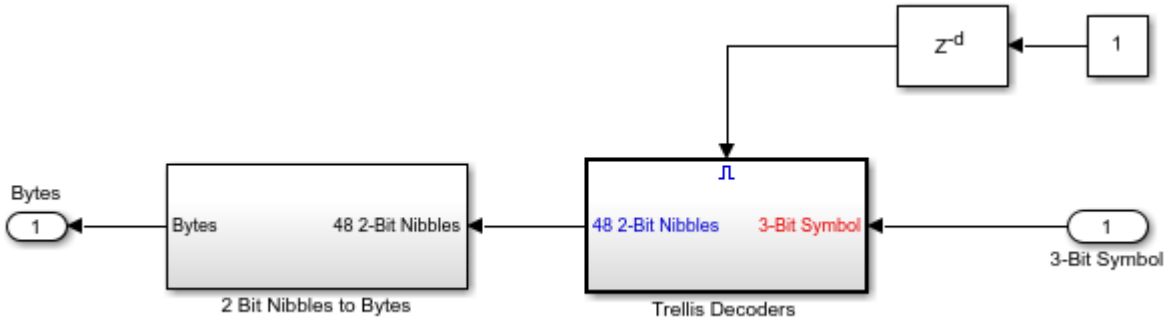
Receiver Baseband Processing

- 8-PAM Demodulator

The M-PAM Demodulator Baseband block converts the received baseband 8-PAM constellation symbols to 3-bit integer outputs. The block has the same constellation settings as the upstream M-PAM Modulator Baseband block.

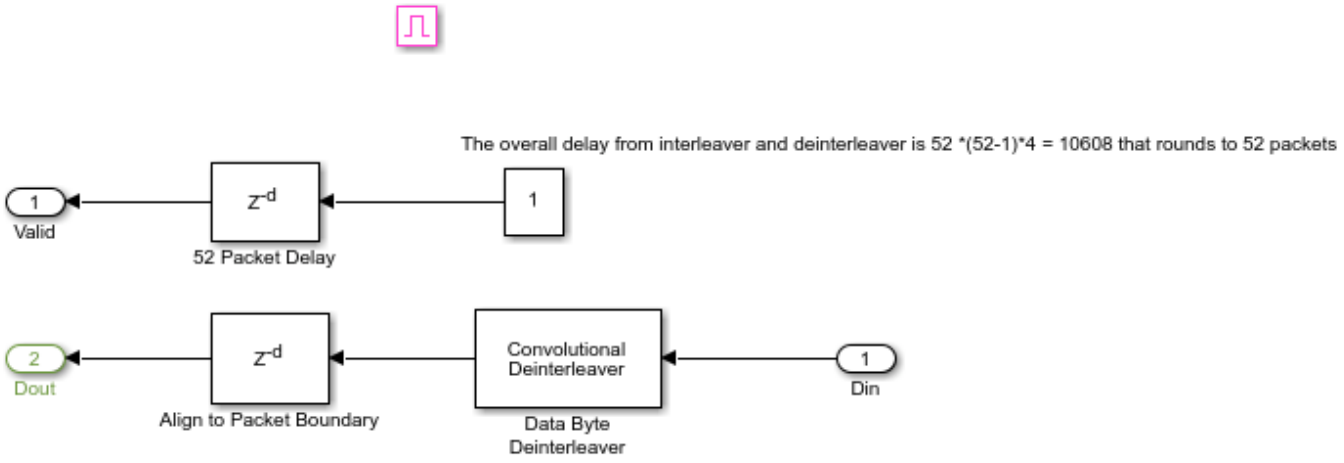
- Trellis Deinterleaver

This subsystem converts serial 3-bit inputs to parallel bytes by feeding each input through one of 12 two-thirds (2/3) rate Viterbi Decoder blocks. Then, the subsystem concatenates the decoded bits into bytes. The deinterleaver processes every 48 inputs corresponding to 12 bytes as a group, and introduces one group (48 inputs) of delay before performing Viterbi decoding. The same control block as in the `Trellis Interleaver` subsystem is used to select which Viterbi Decoder block processes which input in a group. Note that the `Trellis Interleaver` and `Trellis Deinterleaver` subsystems together introduce 207 + 48 = 255 bytes of delay into the system (from Buffer blocks). So, the `Trellis Deinterleaver` subsystem output is delayed by 159 bytes for frame alignment, and the first two frames received by the downstream subsystem should be ignored. To notify the subsequent subsystem of this frame delay, the `Trellis Deinterleaver` subsystem creates a frame valid flag and passes it downstream.



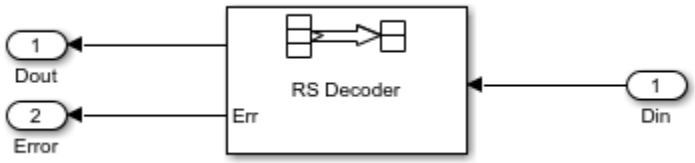
- Convolutional Deinterleaver

The Convolutional Deinterleaver block corresponds to the Convolutional Interleaver block at the transmitter and both blocks have the same configuration. Note that the Convolutional Interleaver and Convolutional Deinterleaver blocks together introduce 10608 bytes of delay into the system. As a result, the subsystem delays Convolutional Deinterleaver block output by 156 bytes for packet alignment, and the first 52 packets received by the downstream subsystem should be ignored. To notify the subsequent subsystem of this packet delay, the `Convolutional Deinterleaver` subsystem creates a packet valid flag and passes it downstream.



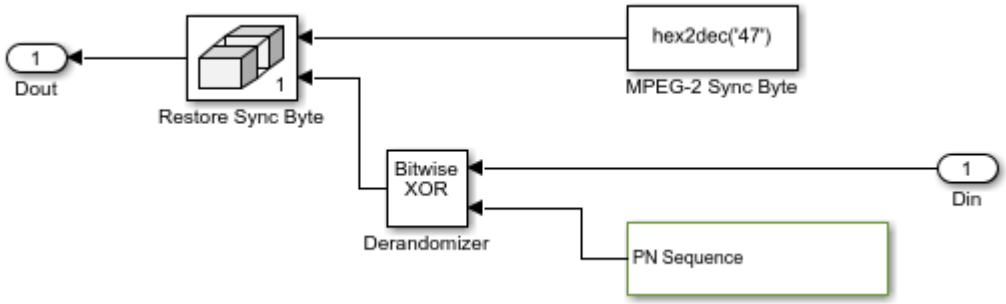
- Reed-Solomon Decoder

The Integer-Output RS Decoder block corresponds to the Integer-Input RS Encoder block at the transmitter and both blocks have the same configuration. The block has a second output port to indicate the number of bytes that have been corrected for the processed packet.



- Derandomizer

This subsystem corresponds to the `Randomizer` subsystem at the transmitter. The block that generates the pseudo random byte sequence is the same as the block in the `Randomizer` subsystem. The MPEG-2 sync byte is inserted into each packet after the derandomization to form an MPEG-2 Transport Packet.



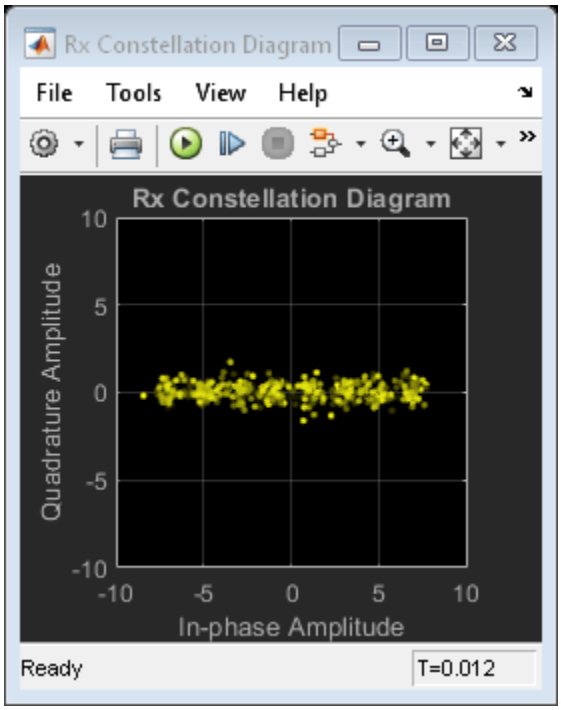
Results and Displays

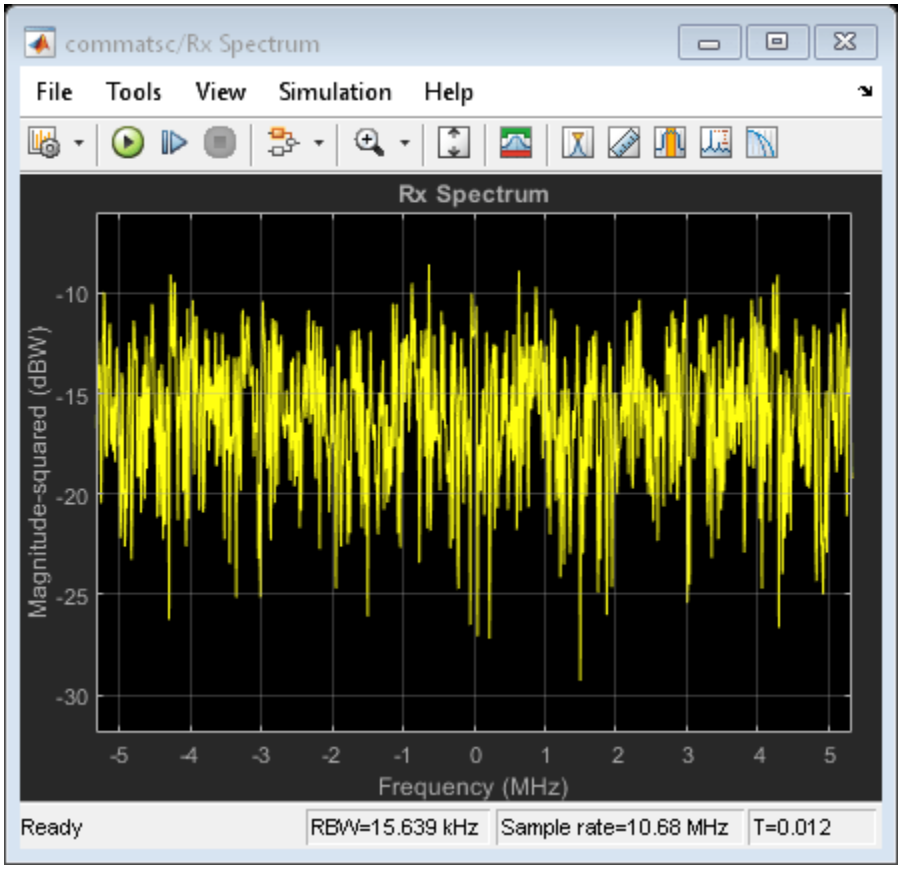
The Error Rate Calculation block measures the system byte error rate by comparing the transmitted and decoded MPEG-2 Transport Packet data. Note that the system has 54 packets, i.e., 10152 bytes, of delay in total, which specifies the `Receive delay` parameter of the block.

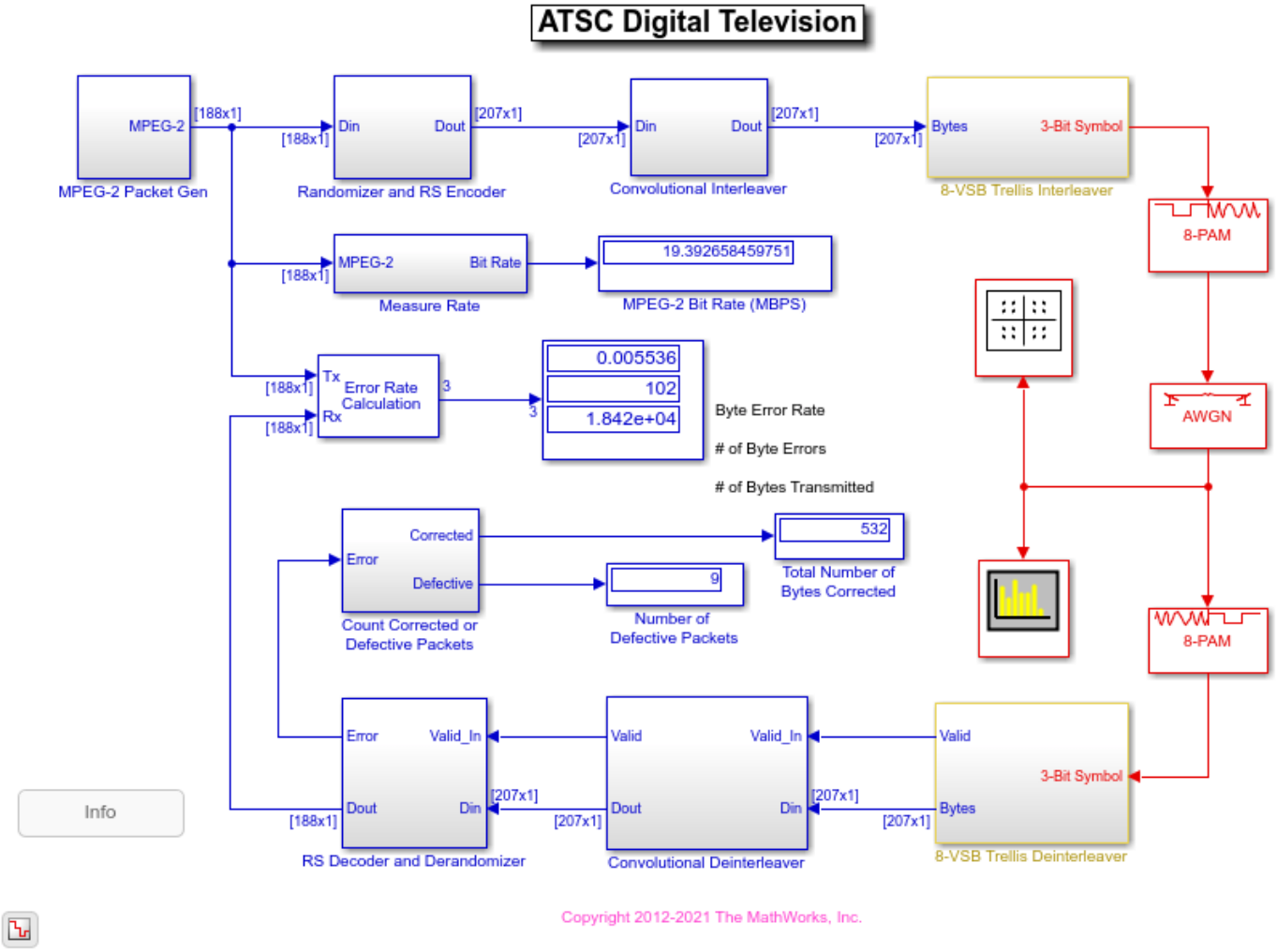
To examine the performance of the system, use the included visualization blocks, as listed below:

- `MPEG-2 Bit Rate (Mbit/s)` display
- `Receiver 8-PAM Constellation Diagram` scope
- `Receiver Spectrum` scope
- `Number of Corrected Bytes` display
- `Number of Defective Packets` display
- `System Byte Error Rate` display

Run the model. Scopes show the ATSC received constellation and the spectrum.




commatsc/Rx Spectrum
File
Tools
View
Simulation
Help
Rx Spectrum
Magnitude-squared (dBW)
-10
-15
-20
-25
-30
-5
-4
-3
-2
-1
0
1
2
3
4
5
Frequency (MHz)
Ready
RBW=15.639 kHz
Sample rate=10.68 MHz
T=0.012



Further Exploration

Upon loading the model, you can set a different signal to noise ratio (SNR) by changing the `EsNo` field value of the `prmATSC` workspace variable and observe the system performance. The following components are not modeled in the system, but you can try to include them:

- Data Segment and Data Field synchronization
- Channel impairments such as multipath fading channels and frequency offsets
- Receiver carrier recovery and equalization

Selected Bibliography

1 Advanced Television Systems Committee, *ATSC Digital Television Standard A/53, Part 2 - RF/ Transmission System Characteristics*, Washington, D.C., Jan. 3, 2007.

DVB-S.2 Link, Including LDPC Coding

This example shows the application of low density parity check (LDPC) codes in the second generation Digital Video Broadcasting standard (DVB-S.2), which is deployed by DIRECTV in the United States. The example uses communications System objects to simulate a transmitter-receiver chain that includes LDPC encoding and decoding.

Introduction

The ETSI (European Telecommunications Standards Institute) EN 302 307 standard for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S.2) [1] uses a state-of-the-art coding scheme to increase the channel capacity. The concatenation of LDPC (Low-Density Parity-Check) and BCH codes is the basis of this coding scheme. LDPC codes, invented by Gallager in his seminal doctoral thesis in 1960, can achieve extremely low error rates near channel capacity by using a low-complexity iterative decoding algorithm [2]. The outer BCH codes are used to correct sporadic errors made by the LDPC decoder.

The channel codes for DVB-S.2 provide a significant capacity gain over DVB-S under the same transmission conditions. Depending on the transmission mode, DVB-S.2 provides Quasi-Error-Free operation (packet error rate below 10^ -7) at about 0.7 dB to 1 dB from the Shannon limit.

This example simulates the BCH encoder, LDPC encoder, interleaver, modulator, as well as their counterparts in the receiver, according to the DVB-S.2 standard. The example collects the error rate at the demodulator, LDPC decoder, and BCH decoder outputs, determines the distribution of the number of iterations performed by the LDPC decoder, and shows the received symbol constellation. For more information regarding system structure, simplifications, and assumptions, see the “DVB-S.2 Link, Including LDPC Coding in Simulink” on page 8-372 example.

Initialization

The configureDVBS2Demo.m script initializes some simulation parameters and generates a structure, dvb. The fields of this structure are the parameters of the DVB-S.2 system at hand. It also creates the System objects making up the DVB-S.2 system.

```
subsystemType = '16APSK 2/3'; % Constellation and LDPC code rate
EsNodB        = 9;            % Energy per symbol to noise PSD ratio in dB
numFrames     = 20;           % Number of frames to simulate

% Initialize
configureDVBS2Demo

% Display system parameters
dvb
```

```
dvb =

  struct with fields:

                 CodeRate: '2/3'
                   EsNodB: 9
          ModulationType: '16APSK'
        NumBytesPerPacket: 188
         NumBitsPerPacket: 1504
        BCHCodewordLength: 43200
```

```
          BCHMessageLength: 43040
          BCHGeneratorPoly: [1 0 1 1 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 1 ... ]
          BCHPrimitivePoly: [1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1]
      NumPacketsPerBBFrame: 28
    NumInfoBitsPerCodeword: 42112
                BitPeriod: 2.3746e-05
       LDPCCodewordLength: 64800
     LDPCParityCheckMatrix: [21600x64800 logical]
         LDPCNumIterations: 50
           InterleaveOrder: [64800x1 double]
             Constellation: [16x1 double]
             SymbolMapping: [12 14 15 13 4 0 8 10 2 6 7 3 11 9 1 5]
               PhaseOffset: [0.7854 0.2618]
             BitsPerSymbol: 4
           ModulationOrder: 16
             SequenceIndex: 2
        NumSymsPerCodeword: 16200
                  NoiseVar: 0.1259
               NoiseVarEst: 0.3227
            RecDelayPreBCH: 43040
```

The example uses these System objects and functions.

Simulation objects:

```
enc            - BCH encoder
dec            - BCH decoder
intrlvr        - Block interleaver
deintrlvr      - Block deinterleaver
pskModulator   - PSK modulator
pskDemodulator - PSK demodulator
chan           - AWGN channel
```

Performance measurement objects:

```
PER       - Packet error rate calculator
BERLDPC   - LDPC decoder output error rate calculator
BERMod    - Demodulator output error rate calculator
constDiag - Scatter plot of channel output
meanCalc  - Average of the noise variance
```

Simulation functions:

```
dvbsapskmod   - DVBSAPSK modulator
dvbsapskdemod - DVBSAPSK demodulator
ldpcEncode    - LDPC encoder
ldpcDecode    - LDPC decoder
```

LDPC Encoder and Decoder Configuration Objects

Create LDPC encoder and decoder configuration objects based on the parity check matrix according to Section 5.3.1 of the DVB-S.2 standard [1].

```
encldpcCfg = ldpcEncoderConfig(dvb.LDPCParityCheckMatrix);
decldpcCfg = ldpcDecoderConfig(dvb.LDPCParityCheckMatrix);
```

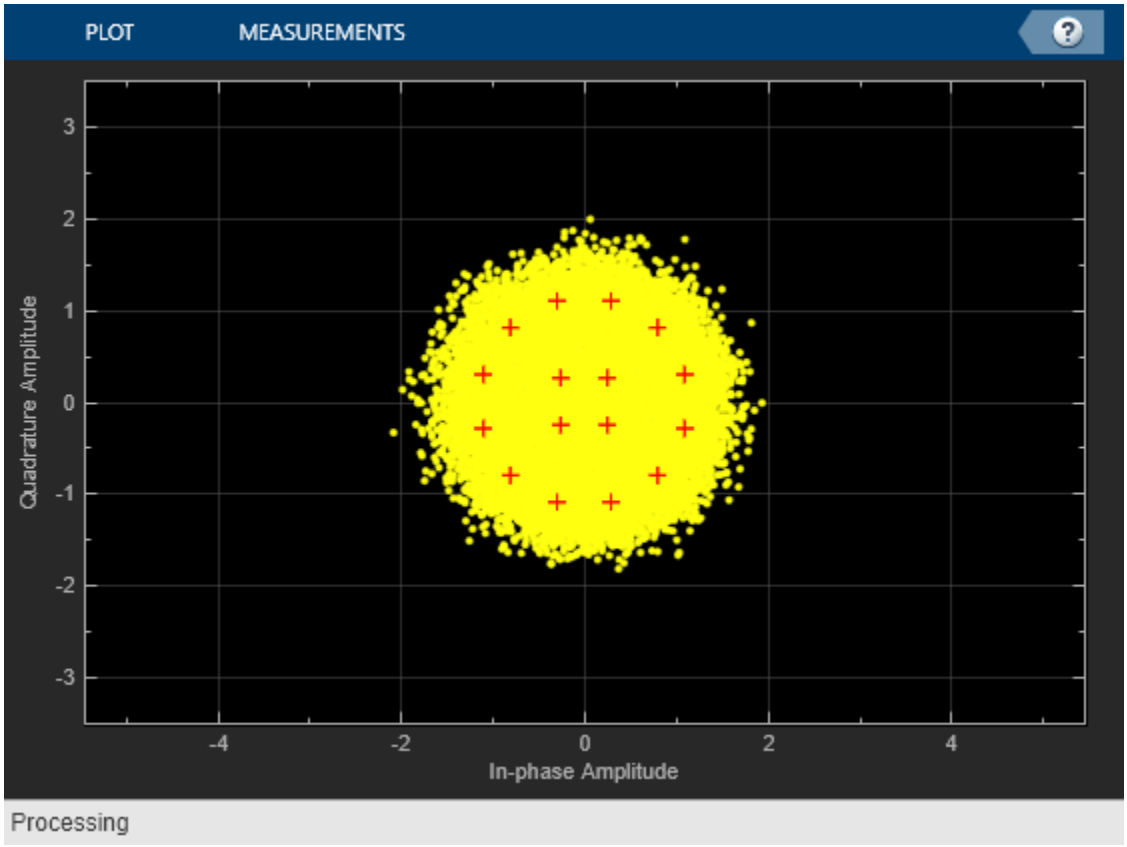
Stream Processing Loop

This section of the code calls the processing loop for a DVB-S.2 system. The main loop processes the data frame-by-frame, where the system parameter `dvb.NumPacketsPerBBFrame` determines the number of data packets per BB frame. The first part of the for-loop simulates the system. The simulator encodes each frame using BCH and LDPC encoders as inner and outer codes, respectively. The encoded bits pass through an interleaver. The modulator maps the interleaved bits to symbols from the predefined constellation. The modulated symbols pass through an AWGN channel. The demodulator employs an approximate log-likelihood algorithm to obtain soft bit estimates. The LDPC decoder decodes the deinterleaved soft bit values and generates hard decisions. The BCH decoder works on these hard decisions to create the final estimate of the received frame.

The second part of the for-loop collects performance measurements such as the bit error rate and a scatter plot. It also estimates the received SNR value.

```
bbFrameTx  = false(encbch.MessageLength,1);
numIterVec = zeros(numFrames,1);
falseVec   = false(dvb.NumPacketsPerBBFrame,1);

for frameCnt=1:numFrames

    % Transmitter, channel, and receiver
    bbFrameTx(1:dvb.NumInfoBitsPerCodeword) = ...
        logical(randi([0 1],dvb.NumInfoBitsPerCodeword,1));

    bchEncOut = encbch(bbFrameTx);
    ldpcEncOut = ldpcEncode(bchEncOut,encldpcCfg);
    intrlvrOut = intrlv(ldpcEncOut,dvb.InterleaveOrder);

    if dvb.ModulationOrder == 4 || dvb.ModulationOrder == 8
        modOut = pskModulator(intrlvrOut);
    else
        modOut = dvbsapskmod(intrlvrOut,dvb.ModulationOrder,'s2', ...
            dvb.CodeRate,'InputType','bit','UnitAveragePower',true);
    end

    chanOut = chan(modOut);

    if dvb.ModulationOrder == 4 || dvb.ModulationOrder == 8
        demodOut = pskDemodulator(chanOut);
    else
        demodOut = dvbsapskdemod(chanOut,dvb.ModulationOrder,'s2', ...
            dvb.CodeRate,'OutputType','approxllr','NoiseVar', ...
            dvb.NoiseVar,'UnitAveragePower',true);
    end

    deintrlvrOut = deintrlv(demodOut,dvb.InterleaveOrder);
    % By default, ldpcDecode stops iterating when all parity checks are
    % satisfied, which reduces decoding time
    [ldpcDecOut, numIter] = ldpcDecode(deintrlvrOut,decldpcCfg,dvb.LDPCNumIterations);
    bchDecOut = decbch(ldpcDecOut);
    bbFrameRx = bchDecOut(1:dvb.NumInfoBitsPerCodeword,1);

    % Error statistics
    comparedBits = xor(bbFrameRx,bbFrameTx(1:dvb.NumInfoBitsPerCodeword));
    packetErr    = any(reshape(comparedBits,dvb.NumBitsPerPacket, ...
        dvb.NumPacketsPerBBFrame));
```

```
    per = PER(falseVec,packetErr');
    berMod = BERMod(demodOut<0,intrlvrOut);
    berLDPC = BERLDPC(logical(ldpcDecOut),bchEncOut);

    % LDPC decoder iterations
    numIterVec(frameCnt) = numIter;

    % Noise variance estimate
    noiseVar = meanCalc(var(chanOut - modOut));

    % Scatter plot
    constDiag(chanOut);
end
```



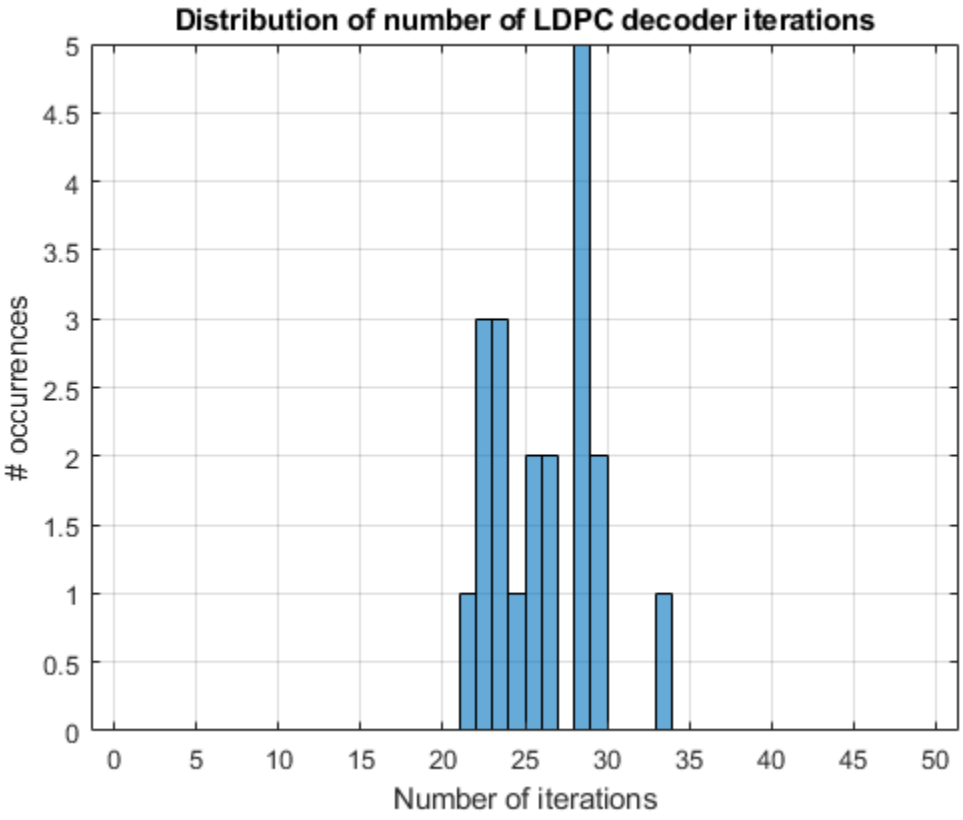
Executing the error rate measurement objects (hPER, hBERMod, and hBERLDPC), outputs a 3-by-1 vector containing updates of the measured error rate value, the number of errors, and the total number of transmissions (packets or bits). Display the BER at the demodulator output, the BER at the LDPC decoder output, and the packet error rate of the end-to-end system together with the measured SNR at the receiver input. While the demodulator output presents an error rate of more than 10%, the LDPC decoder is able to correct all of the errors and provide error free packets.

```
fprintf('Measured SNR : %1.2f dB\n',10*log10(1/noiseVar))
fprintf('Modulator BER: %1.2e\n',berMod(1))
fprintf('LDPC BER     : %1.2e\n',berLDPC(1))
fprintf('PER          : %1.2e\n',per(1))
```

```
Measured SNR : 8.98 dB
Modulator BER: 8.25e-02
LDPC BER     : 0.00e+00
PER          : 0.00e+00
```

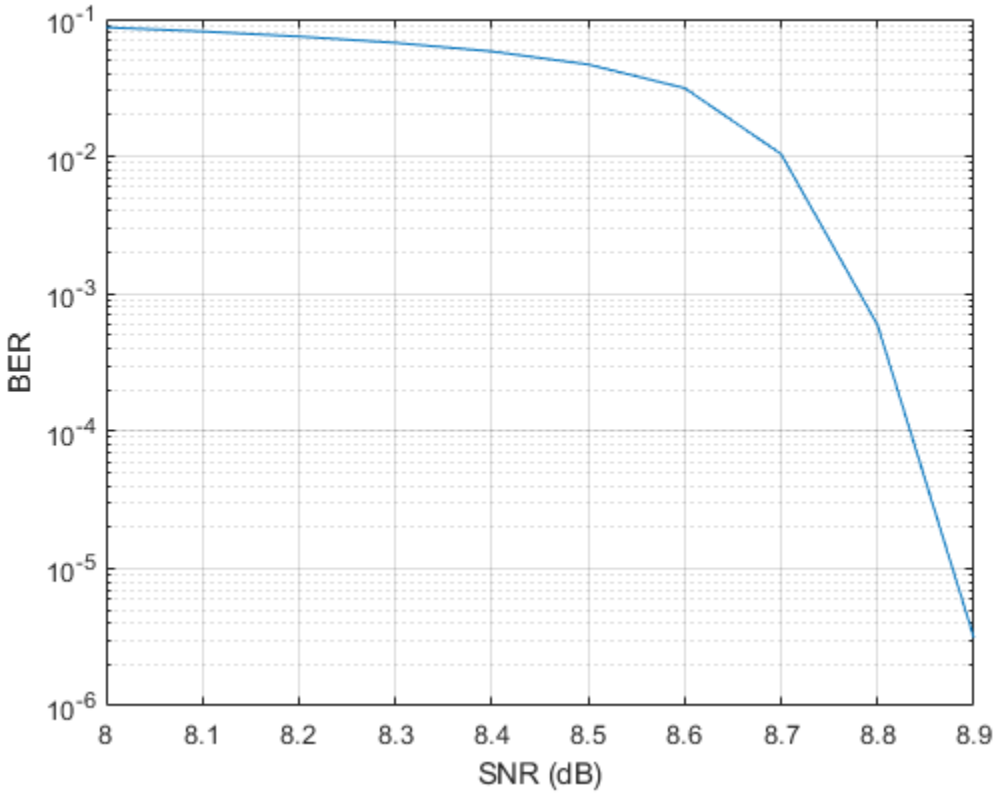
The figure shows the distribution of the number of iterations performed by the LDPC decoder. The decoder was able to decode all the frames without an error before reaching the maximum iteration count of 50.

```
distFig = figure;
histogram(numIterVec,1:dvb.LDPCNumIterations-1);
xlabel('Number of iterations'); ylabel('# occurrences'); grid on;
title('Distribution of number of LDPC decoder iterations')
```



We ran the stream processing loop for 32.4e6 bits for several SNR values. Since this simulation takes a long time, in this example we only provide the result of the simulation stored in a MAT-file.

```
load berResultsDVBS2Demo.mat cBER_16APSK snrdB_16APSK
berFig = figure;
semilogy(snrdB_16APSK,cBER_16APSK(1,:)); xlim([8 8.9]);
xlabel('SNR (dB)'); ylabel('BER'); grid on
```



Summary

This example utilized several System objects to simulate part of the DVB-S.2 communication system over an AWGN channel. It showed how to model several parts of the DVB-S.2 system such as the LDPC coding. System performance was measured using the PER and BER values obtained with error rate measurement System objects.

Further Exploration

You can modify parts of this example to experiment with different subsystem types using various values for Es/No and maximum number of LDPC decoder iterations. This example supports the following subsystem types:

```
'QPSK 1/4', 'QPSK 1/3', 'QPSK 2/5', 'QPSK 1/2', 'QPSK 3/5', 'QPSK
2/3', 'QPSK 3/4', 'QPSK 4/5', 'QPSK 5/6', 'QPSK 8/9', 'QPSK 9/10'

'8PSK 3/5', '8PSK 4/5', '8PSK 2/3', '8PSK 3/4', '8PSK 5/6', '8PSK
8/9', '8PSK 9/10'

'16APSK 2/3', '16APSK 3/4', '16APSK 4/5', '16APSK 5/6', '16APSK
8/9', '16APSK 9/10'

'32APSK 3/4', '32APSK 4/5', '32APSK 5/6', '32APSK 8/9', '32APSK
9/10'
```

Appendix

This example uses the following scripts and helper function:

- configureDVBS2Demo.m
- getParamsDVBS2Demo.m
- createSimObjDVBS2Demo.m

Selected Bibliography

1. ETSI Standard EN 302 307 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-S2)*, European Telecommunications Standards Institute, Valbonne, France, 2005-03.
2. R. G. Gallager, *Low-Density Parity-Check Codes*, IEEE® Transactions on Information Theory, Vol. 8, No. 1, January 1962, pp. 21-28.
3. W. E. Ryan, *An introduction to LDPC codes*, in Coding and Signal Processing for Magnetic Recording Systems (Bane Vasic, ed.), CRC Press, 2004.

DVB-S.2 Link, Including LDPC Coding in Simulink

This model shows the state-of-the-art channel coding scheme used in the second generation Digital Video Broadcasting standard (DVB-S.2), which is deployed by DIRECTV in the United States. The coding scheme is based on concatenation of LDPC (Low-Density Parity-Check) and BCH codes. LDPC codes, invented by Gallager in his seminal doctoral thesis in 1960, can achieve extremely low error rates near channel capacity by using a low-complexity iterative decoding algorithm. The outer BCH codes are used to correct sporadic errors made by the LDPC decoder.

The channel codes for DVB-S.2 provide a significant capacity gain over DVB-S under the same transmission conditions and allow Quasi-Error-Free operation (packet error rate below 10^ -7) at about 0.7 dB to 1 dB from the Shannon limit, depending on the transmission mode.

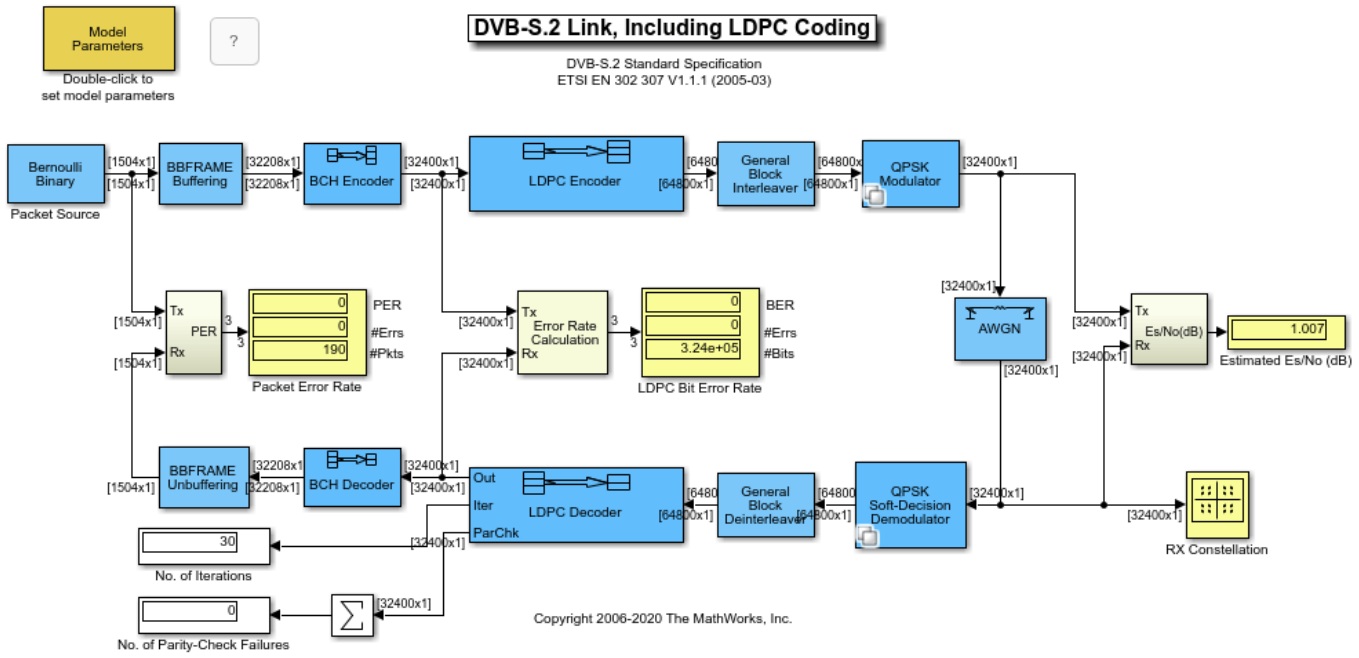
This example models the BCH encoder, LDPC encoder, interleaver, modulator, as well as their counterparts in the receiver, according to the DVB-S.2 standard.

Structure of the Example

The communication system in this example performs these tasks:

- Generation of BBFRAME by a random source
- BCH encoding, for all coding parameters and normal FECFRAME
- LDPC encoding, for all coding parameters and normal FECFRAME
- Interleaving
- Modulation (QPSK, 8PSK, 16APSK, or 32APSK)
- AWGN channel modeling
- Soft-decision demodulation
- Deinterleaving
- LDPC decoding, by means of the message passing algorithm
- BCH decoding
- BBFRAME unbuffering

```
modelname =  'commdvbs2' ;
open_system(modelname);
RX = [modelname '/RX Constellation']; % Define Simulink object as a variable
set_param(RX,'openScopeAtSimStart','off' ); % Set Simulink scope visibility parameter
T = evalc('sim(modelname)');
```



Furthermore, this model has blocks for measuring and displaying the packet error rate, LDPC bit error rate, and estimated Es/No. There is also a scatter plot scope displaying the received signal, which helps users visualize the channel distortions of the signal.

Simplifications and Assumptions

For simplicity, this example

- Assumes perfect synchronization between the transmitter and the receiver
- Uses a complex baseband model of an AWGN channel, rather than a full satellite channel
- Models BBHEADER and DATA FIELD in a BBFRAME using a Bernoulli binary random source, and does not perform baseband scrambling
- Supports only normal FECFRAME (i.e., the block length of LDPC codes is 64800)
- Processes one LDPC codeword in one unit of time in Simulink®
- Approximates the log-likelihood ratio of the channel output for LDPC decoding by considering only two points in the constellation nearest to the received signal during soft-decision demodulation
- Uses Es/No provided by the user for LDPC decoding, instead of estimating the Es/No from the received signal

Also, the example does not model these aspects of the DVB-S.2 standard:

- Short FECFRAME
- Physical Layer (PL) Framing
- PL Signalling and Pilot insertion
- PL Scrambler
- Baseband (BB) Filter and Quadrature Modulation

Parameters of the Model

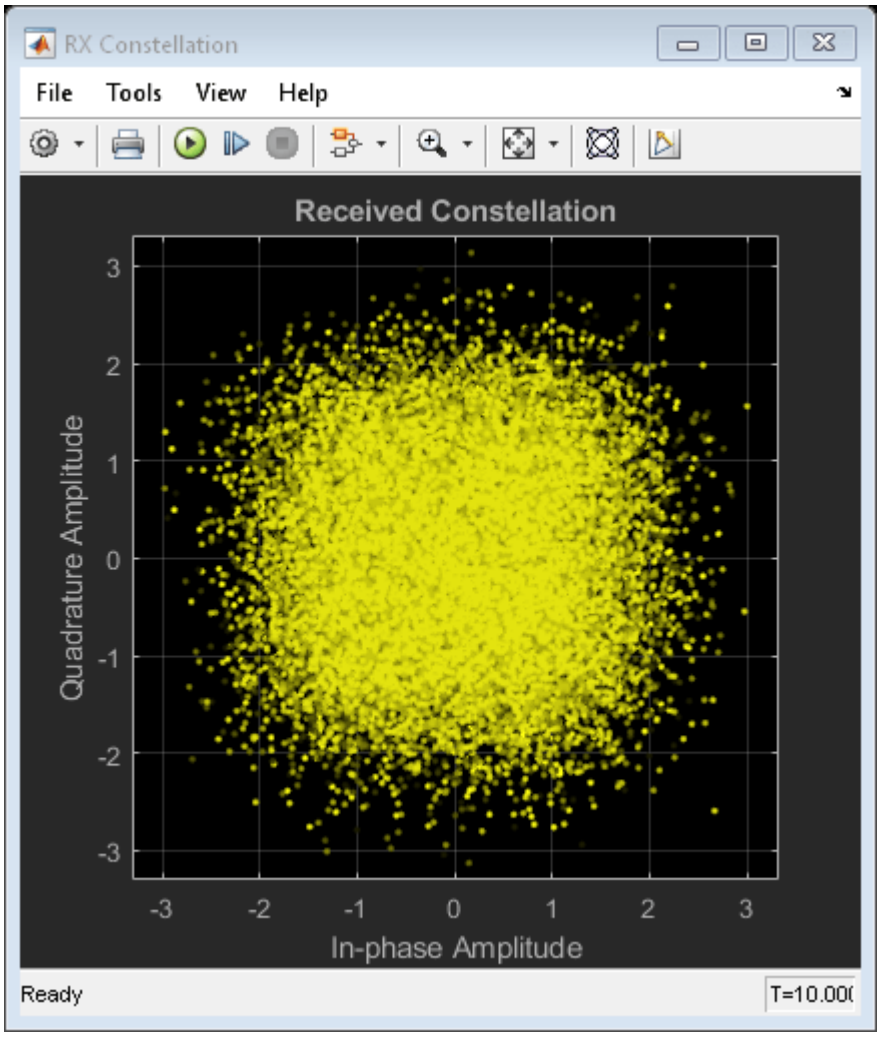
Double-clicking the Model Parameters block allows users to set the following parameters for the model:

Parameter	Description	Range of values
Mode	One of 17 combinations of modulation (QPSK or 8PSK) and coding rate, as specified in the DVB-S.2 standard.	Selectable from menu.
Es/No (dB)	Ratio of energy per symbol to noise power spectral density. Expressed in dB.	Any real number. Typically -5 to 20 dB.
Number of iterations for LDPC decoder	Number of message passing iterations (between bit nodes and check nodes) for LDPC decoding algorithm.	Integer >= 1. The default value is 50 for DVB-S.2.

Results and Displays

When the model starts, a window automatically comes up to display the scatter plot of the received signal. The LDPC bit error rate, packet error rate, and estimated Es/No from the received signal will be continuously updated.

```
% Set scope visibility for next display and run simulation
set_param(RX,'openScopeAtSimStart','on');
sim(modelname);
```



The power of LDPC codes can be readily observed using the default settings: QPSK, rate 1/2, Es/No = 1 dB, and 50 iterations in decoding. Even with such a low Es/No, the LDPC decoder will seldom make an error. The scatter plot vividly illustrates how noisy the channel is.

If Es/No is slightly decreased, for example, to 0.5 dB, the LDPC bit error rate will be much greater. This is consistent with typical steep performance curves of LDPC codes.

```
% Cleanup
%
% To clear the variables set above and close without saving the changes to
% the model, type the following commands into the MATLAB(R) command prompt.
%

close_system(modelname,0);
```

Selected Bibliography

[1] *DVB-S.2 Standard Specification*, ETSI EN 302 307 V1.1.1 (2005-03).

[2] R. G. Gallager, *Low-Density Parity-Check Codes*, IRE Transactions on Information Theory, Vol. 8, No. 1, January 1962, pp. 21-28.

[3] W. E. Ryan, *An introduction to LDPC codes*, in Coding and Signal Processing for Magnetic Recoding Systems (Bane Vasic, ed.), CRC Press, 2004.

Digital Video Broadcasting - Cable (DVB-C)

This example shows part of the ETSI (European Telecommunications Standards Institute) EN 300 429 standard for cable system transmission of digital television signals [1]. The example uses communications System objects to simulate the Digital Video Broadcasting - Cable (DVB-C) transmitter-receiver chain.

Introduction

The DVB-C standard describes transmission of digital television signals over cable lines using the MPEG-2 or MPEG-4 family of digital audio and video streams. In this example, we model a portion of that standard. The stream of data is transmitted using Reed-Solomon codes and single carrier QAM modulation. The standard prescribes the transmitter design and sets minimum performance requirements for the receiver.

The purpose of this example is to:

- Model the main portions of a possible transmit/receive design (operating in 64-QAM mode with MPEG-2 Transport Packet data)
- Illustrate the use of key Communications Toolbox™ System objects for DVB-C (or similar) system design
- Illustrate creation of higher level System objects that contain other System objects in order to model large components of the system under test
- Generate error statistics that will help to determine whether the model satisfies system performance requirements
- Illustrate creation of test harness that can support variable numbers of test runs. In this case, we use that support to support one mode where only a single EbNo is specified, and we observe spectra and scatterplots. We also support a mode where multiple EbNo are specified, in order to generate a BER curve.

Initialization

The commdvbc_init.m script initializes simulation parameters and generates a structure, prmDVBC. The fields of this structure are the parameters of the DVB-C system at hand.

```
commdvbc_init
% The fields of this structure are the parameters of the DVB-C system at
% hand.
prmDVBC
```

```
prmDVBC =

  struct with fields:

            bitsPerByte: 8
            bitsPerMTpl: 6
     MPEG2DatRateBitPerS: 9600000
     rawMPEG2DataPcktLen: 184
     MPEG2TrnsprtPcktLen: 188
    MPEG2TrnsprtFramePer: 1.5667e-04
     MPEG2PcktsPerSprFrm: 8
     MPEG2TrnsSuperFrame: 1504
      PRBSSeqPeriodBytes: 1503
```

```
       PRBSSeqPeriodBits: 12024
        RSCodewordLength: 204
       CableChanFrameLen: 272
      CableChanFrmPeriod: 1.5667e-04
      RCosineSampsPerSym: 8
       CableSymbolPeriod: 7.1998e-08
       RCosineFilterSpan: 16
     TxRxSymbolSampDelay: 288
     DeintrlvrAlignDelay: 192
        QAMSymbolMapping: [44 45 41 40 52 54 62 60 46 47 43 42 53 55 63 ... ]
     ConvIntlNumBranches: 12
       ConvIntlCellDepth: 17
```

Run System under Test

The main loop in the system under test processes the data packet-by-packet, where eight packets form a superframe. Set useCodegen=true in order to use the generated code instead of the MATLAB® code. Set the MATLAB variable compileIt to true in order to create the generated code.

Code Architecture for the System under Test

This example models the link from the cable operator to a customer's set top box. The model for that link is contained in a function named runDVBCSystemUnderTest. The data processing loop is divided into six main parts. A System object™ was used to model each of those six components in that link. Those objects are:

```
1) DVBCSource: generates the bitstream
2) DVBCTransmitter: contains the transmitter (encoding, modulation, filtering, etc.)
3) comm.AWGNChannel: models the channel
4) DVBCReceiver: contains the receiver
5) DVBCBER: calculates error rates
6) DVBCScopes: optional object that provides visualization
```

The inner loop of runDVBCSystemUnderTest makes use of these objects:

You can use a for-loop around the system under test to process a fixed number of super frames. Alternatively, you can use a while-loop to control the simulation length based on the number of simulated errors and transmitted bits. We have done the latter, targeting the number of errors to 100, and maximum number of transmissions to 1e6.

```
while (berEnd2End(2) < totalErrors) && (berEnd2End(3) < totalBits)
    txBytes = dvbcSource();                          % Source
    [txPckt, modTxPckt] = dvbcTX(txBytes);           % Transmitter
    chPckt = awgnChan(txPckt);                       % Channel
    [rxBytes, modRxPckt, rxPcakt] = dvbcRX(chPckt);  % Receiver
    [berEnd2End, berDemod] = ...
      dvbBER(txBytes,rxBytes,modTxPckt,modRxPckt);   % BER
    if useScopes
        runDVBCScopes(dvbcScope,txPckt,chPckt,rxPckt);
    end
end
```

Descriptions of the Individual Components

MPEG-2 Baseband Physical Interface - Data source

This section generates random data and header bits and appends a header synchronization byte. The first packet of each superframe uses the bit-complement of the header synchronization byte. The code for this component is contained in DVBCSource.m.

Transmitter Baseband Processing

This section randomizes the data using a pseudo-noise sequence. The transmitter applies RS encoding and convolutional interleaving. The function convertBytesToMTuplesDVBCDemo.m converts 8-bit bytes into 6-bit chunks for the 64-QAM modulator. It applies a square root raised cosine filter with 8x oversampling to the data stream after modulation. The code for this component is contained in DVBCTransmitter.m.

Channel

The signal is transmitted through an AWGN channel by using the comm.AWGNChannel System object.

Receiver Baseband Processing

This section demodulates received symbols and converts 6-bit chunks into bytes using the convertMTuplesToBytesDVBCDemo.m function. Since the filtering operation introduces a delay, the example synchronizes the received bytes to the packet edge using the delay System object, hPacketSync. Note that, the interleaver delay is a multiple of the packet size, so synchronizing to the packet edge is enough. The receiver deinterleaves the packet-synchronized bytes and decodes using the RS decoder System object. Because the example uses a single PN sequence generator, it synchronizes the decoded data to the superframe edge before derandomization. The example shows the transmitted and received channel signal spectrum. Finally, it compares transmitted bits and received bits as well as the modulator input and the demodulator output to obtain bit error rates. The code for this component is contained in DVBCReceiver.m.

BER Computations

This component compares the received, decoded bits and compares those to the transmitted bits in order to compute a bit error rate. The code for this component is contained in DVBCBER.m.

Visualization

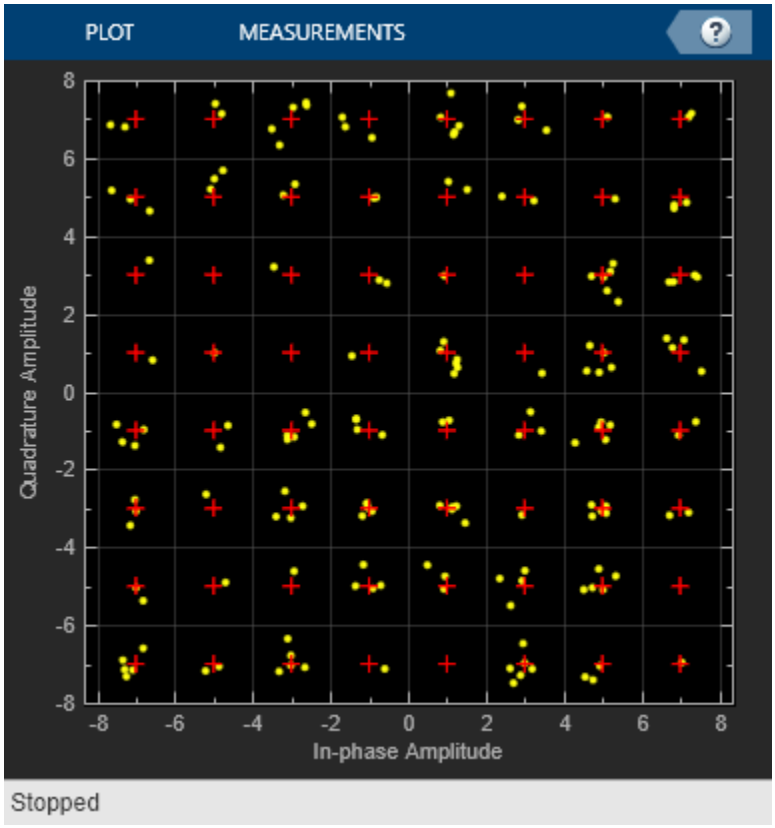
Optional instrumentation provides visualization. The code for this component is contained in DVBCScopes.m.

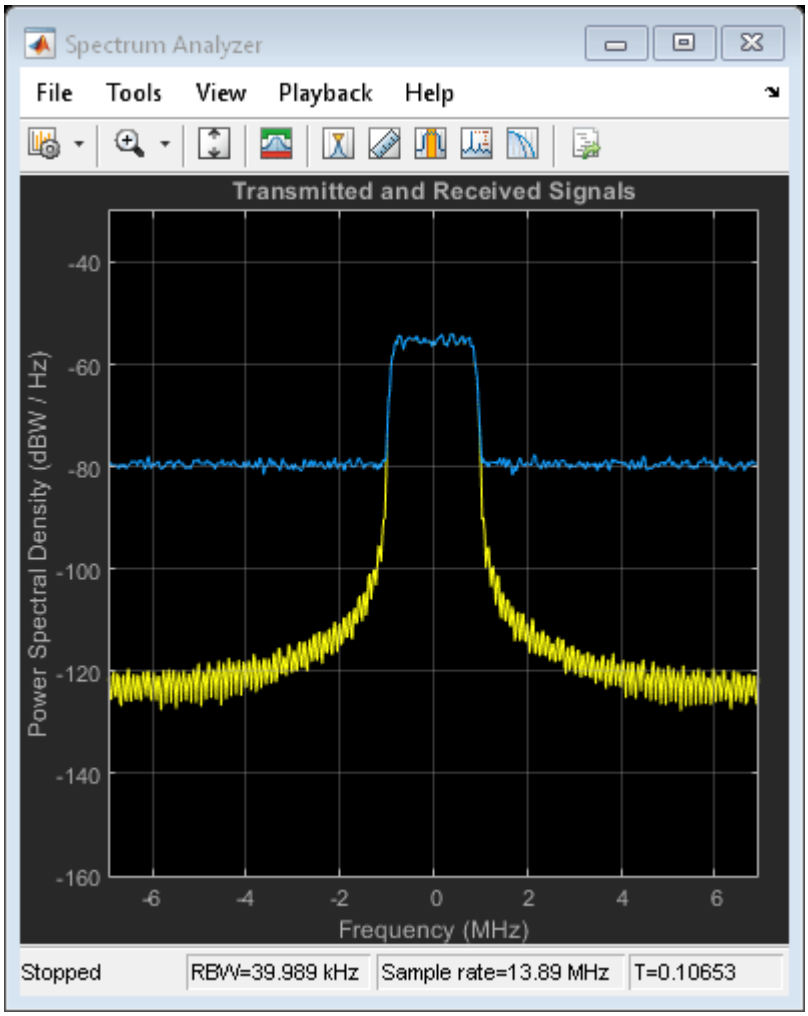
Running the System under Test

We first run the system under test with a single EbNo and visualization turned on in order to verify that it is working properly.

```
totalErrors = 100;
totalBits = 1e6;
EbNo = 16.5;
useScopes = true;
useCodegen = false;
compileIt = false;
if compileIt
    % Make EbNo input var-size row vector (max length = 100)
    codegen runDVBCSystemUnderTest -report -args {coder.Constant(useScopes),coder.Constant(prmDVB
end
if useCodegen
```

```
    % Constant inputs do not appear in call to generated code version
    [berEnd2End, berDemod] = runDVBCSystemUnderTest_mex(useScopes, prmDVBC, num, sigPower, EbNo,
else
    [berEnd2End, berDemod] = runDVBCSystemUnderTest(useScopes, prmDVBC, num, sigPower, EbNo, tota
end
```





BER Curves

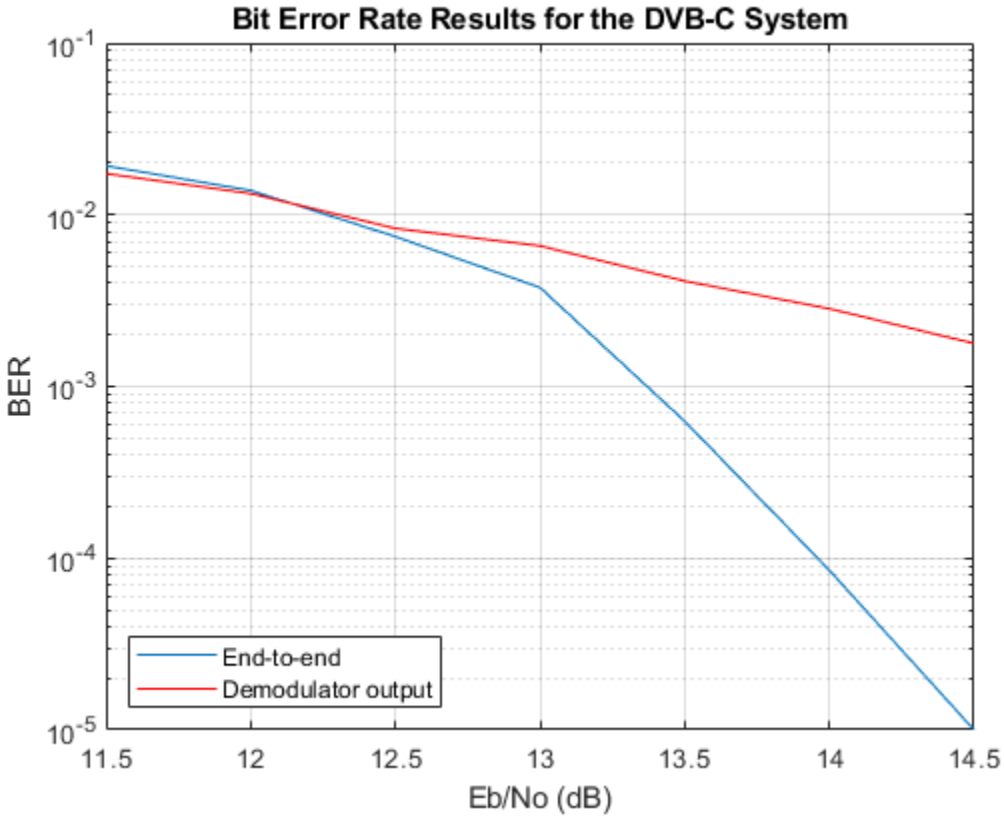
Next, we rerun the system under test with a vector of EbNo's and visualization turned off to generate a BER curve.

Calling the error rate measurement objects, berEnd2End and berDemod, output a 3-by-1 vector containing updates of the measured BER value, the number of errors, and the total number of bit transmissions. Display the BER at the output of the demodulator together with the end-to-end BER.

```
EbNo = 11.5:0.5:14.5;
useScopes = false;
useCodegen = false;
compileIt = false;
if compileIt
    % Make EbNo input var-size row vector (max length = 100)
    codegen runDVBCSystemUnderTest -report -args {coder.Constant(useScopes),coder.Constant(prmDVB
end
if useCodegen
    % Constant inputs do not appear in call to generated code version
    [berEnd2End, berDemod] = runDVBCSystemUnderTest_mex(useScopes, prmDVBC, num, sigPower, EbNo,
else
```

```
    [berEnd2End, berDemod] = runDVBCSystemUnderTest(useScopes, prmDVBC, num, sigPower, EbNo, tota
end
%
plotDVBCResults(EbNo, berEnd2End, berDemod);
```

```
berEnd2End =

    0.0193
    0.0139
    0.0075
    0.0038
    0.0006
    0.0001
    0.0000


berDemod =

    0.0174
    0.0133
    0.0083
    0.0066
    0.0041
    0.0028
    0.0018
```



Summary

This example utilized several System objects to simulate part of the DVB-C communication system over an AWGN channel. It showed how to model several parts of the DVB-C system such as the randomization, coding, and interleaving. The example also used the delay System objects to synchronize the transmitter and the receiver. System performance was measured using the BER curves obtained with error rate measurement System objects.

Appendix

This example uses the following scripts and helper functions:

- runDVBCSystemUnderTest.m
- DVBCSource.m
- DVBCTransmitter.m
- DVBCReceiver.m
- DVBCBER.m
- DVBCScopes.m
- convertBytesToMTuplesDVBCDemo.m
- convertMTuplesToBytesDVBCDemo.m
- createDVBCScopes.m
- runDVBCScopes.m
- plotDVBCResults.m

Selected Bibliography

1 ETSI Standard EN 300 429 V1.2.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems, European Telecommunications Standards Institute, Valbonne, France, 1998.

Digital Video Broadcasting - Cable (DVB-C)

This model shows part of the ETSI (European Telecommunications Standards Institute) EN 300 429 standard for cable system transmission of digital television signals [1]. The standard prescribes the transmitter design and sets minimum performance requirements for the receiver.

The purpose of this example is to:

- Model the main portions of a possible transmitter design (operating in 64-QAM mode with MPEG-2 Transport Packet data)
- Model the main portions of a possible receiver design (operating in 64-QAM mode with MPEG-2 Transport Packet data)
- Generate error statistics that will help determine whether the model satisfies the system performance requirements
- Illustrate the use of key Communications Toolbox™ library blocks for DVB-C (or similar) system design

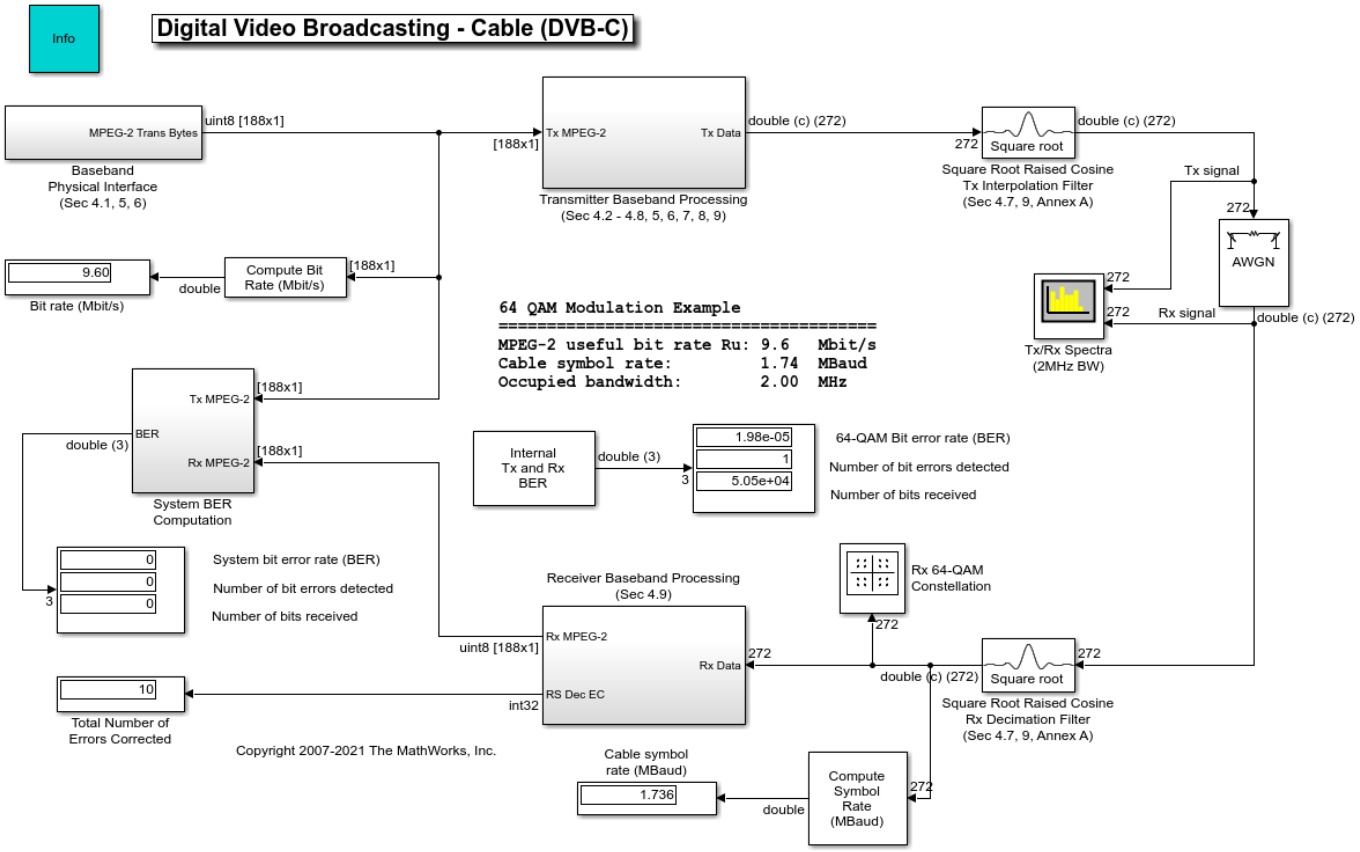
Available Example Versions

There are two different versions of this example.

Floating-point version: commdvbc.slx

Fixed-point version: commdvbc_fixpt.slx

Structure of the Example



MATLAB® Workspace Variable Parameter Definitions

When the example model is first loaded, it creates the MATLAB workspace variable `prmDVBC`, which is a structure with members that are used as parameters in the blocks in the model file. Note also that this workspace variable is cleared when the model is closed.

```
prmDVBC =

  struct with fields:

           bitsPerByte: 8
           bitsPerMTpl: 6
    MPEG2DatRateBitPerS: 9600000
     rawMPEG2DataPcktLen: 184
     MPEG2TrnsprtPcktLen: 188
    MPEG2TrnsprtFramePer: 1.5667e-04
     MPEG2PcktsPerSprFrm: 8
     MPEG2TrnsSuperFrame: 1504
      PRBSSeqPeriodBytes: 1503
       PRBSSeqPeriodBits: 12024
        RSCodewordLength: 204
       CableChanFrameLen: 272
      CableChanFrmPeriod: 1.5667e-04
      RCosineSampsPerSym: 8
```

```
  CableSymbolPeriod: 7.1998e-08
  RCosineFilterSpan: 16
TxRxSymbolSampDelay: 288
DeintrlvrAlignDelay: 192
   QAMSymbolMapping: [44 45 41 40 52 54 62 60 46 47 43 42 53 55 63 ... ]
ConvIntlNumBranches: 12
  ConvIntlCellDepth: 17
```

Baseband Physical Interface (Simulated MPEG-2 Data Source)

This portion of the model corresponds to sections 4.1, 5, and 6 in [1]. The MPEG-2 Transport Packet is defined in ISO®/IEC 13818-1 [2], and is comprised of 188-byte packets.

Communications Toolbox, DSP System Toolbox™, and Simulink® library blocks are used to simulate a MPEG-2 Transport Packet data stream for system simulation and BER performance measurement purposes.

Transmitter Baseband Processing

- Sync1 Inversion and Randomization

This subsystem corresponds to sections 4.2 and 7.1 in [1]. The MPEG-2 `Sync1` byte is inverted, and the data stream (other than the Sync bytes) is randomized for spectrum shaping purposes. A resettable `PN Sequence Generator` library block is used as part of the scrambler for this data randomization process.

- Shortened (204,188) Reed-Solomon Encoder

This library block corresponds to sections 4.3 and 7.2 in [1]. As described in the standard, this process adds 16 parity bytes to the MPEG-2 Transport Packet to give a (204,188) codeword. This allows up to eight (8) erroneous bytes per transport packet to be corrected by the corresponding receiver `Reed-Solomon Decoder` block.

- Convolutional Interleaver

This library block corresponds to sections 4.4 and 7.3 in [1]. The interleaving process is based on the Forney approach [3] and is compatible with the Ramsey type III approach [4], with I = 12.

- Byte (8-bit) to M-Tuple (6-bit) Conversion

A `MATLAB® Function` block is used to perform this processing. 8-bit data bytes are converted into 64-ary (6-bit) values. This block corresponds to sections 4.5 and 8 in [1].

- Differential Encoding

An example implementation of the Differential Encoding unit as described in sections 4.6 and 8 in [1] is shown using a `MATLAB Function` block. For the purposes of this example model, the Differential Encoding unit output is connected to a terminator (i.e., the unit is bypassed).

- 64-QAM Constellation Mapping

The `Rectangular QAM Modulator Baseband` library block maps the baseband 64-ary (M-tuple) values to complex (I and Q) 64-QAM constellation symbol values for transmission, as described in sections 4.7 and 9 in [1].

Square Root Raised Cosine Interpolation Filter

This library block performs the baseband shaping of the complex (I and Q) constellation symbol values for transmission, as described in sections 4.7, 9, and Annex A in [1].

AWGN Channel

The System FEC as specified by the standard is designed to improve the Bit Error Rate (BER) from 10^-4 to the range, 10^-10 to 10^-11 ("Quasi Error Free" operation). The `AWGN Channel` library block `Signal to Noise Ratio (Eb/No)` is set to 16.5 dB corresponding to an operating BER of approximately 10^-4.

Square Root Raised Cosine Rx Decimation Filter

This library block performs the matched decimation filtering of the received complex (I and Q) constellation symbol values, as described in sections 4.7, 9, and Annex A in [1].

Receiver Baseband Processing

- 64-QAM Constellation Demapping

The `Rectangular QAM Demodulator Baseband` library block demaps the received baseband complex (I and Q) 64-QAM constellation symbol values to 64-ary M-tuples, as described in sections 4.7 and 9 in [1].

- Differential Decoding

For the purposes of this example model, the Differential Decoding portion is omitted. Additionally, a more realistic receiver system implementation will likely have equalization and synchronization processing prior to this portion of the receiver model.

- M-Tuple (6-bit) to Byte (8-bit) Conversion

A `MATLAB Function` block is used to perform this processing, which is the inverse of the Byte to M-Tuple processing used in the transmitter. 64-ary (6-bit) M-tuple values are repacked into 8-bit data bytes.

- Convolutional Deinterleaver

The `Convolutional Deinterleaver` library block corresponds to the `Convolutional Interleaver` library block appearing in the transmitter subsystem implementation. The deinterleaving process is based on the Forney approach [3] and is compatible with the Ramsey type III approach [4], with I = 12.

For the sake of example model simplicity, a simple extra delay is used to synchronize the first sync byte into the "0" branch of the Convolutional Deinterleaver. A more realistic receiver system implementation will likely have additional upstream synchronization processing prior to this portion of the model.

- Shortened (204,188) Reed-Solomon Decoder

This library block performs the R-S decoding corresponding to the encoded data packets.

- Sync1 Inversion and Energy Dispersal Removal

This subsystem performs data descrambling to obtain the received MPEG-2 Transport Packet data bytes.

Results and Displays

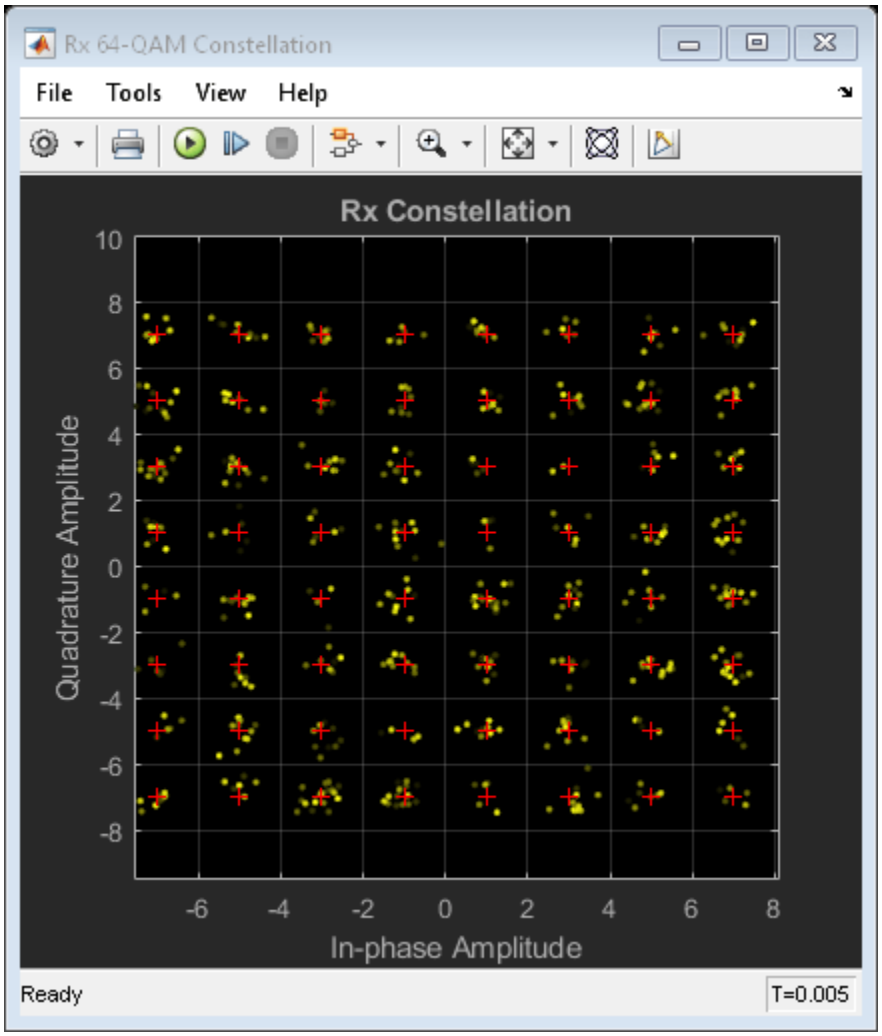
To examine the performance of the example, use the included visualization blocks, as listed below.

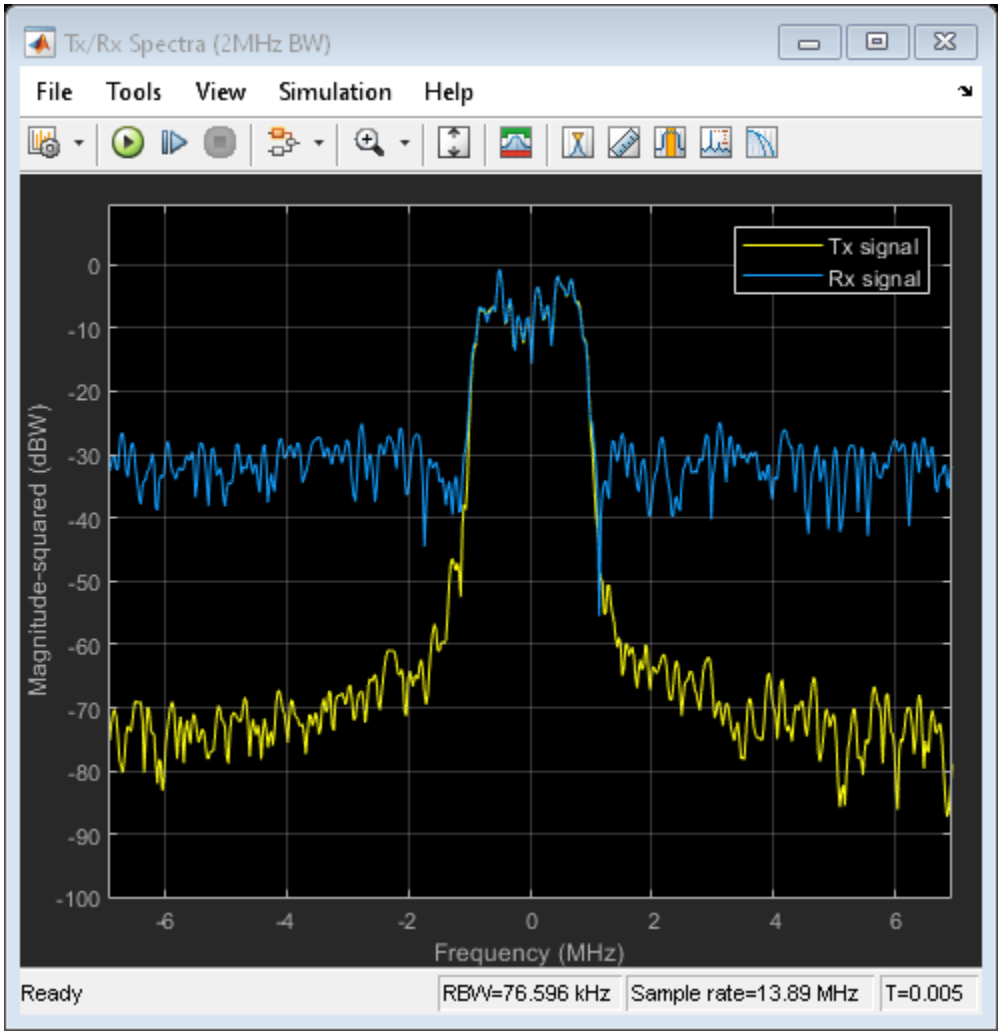
Overall System Results and Displays:

- `Bit rate (Mbit/s)` display
- `Cable symbol rate (MBaud)` display
- `64-QAM bit error rate (BER)` display
- `System bit error rate (BER)` display
- Various internal bit error rate (BER) displays (under the `Internal Tx and Rx BER` subsystem)

Transmitter/Receiver Results and Displays:

- `Rx 64-QAM Constellation` scatter plot
- `Tx/Rx Spectrum (2MHz BW)` scope
- `Total Number of Errors Corrected` display





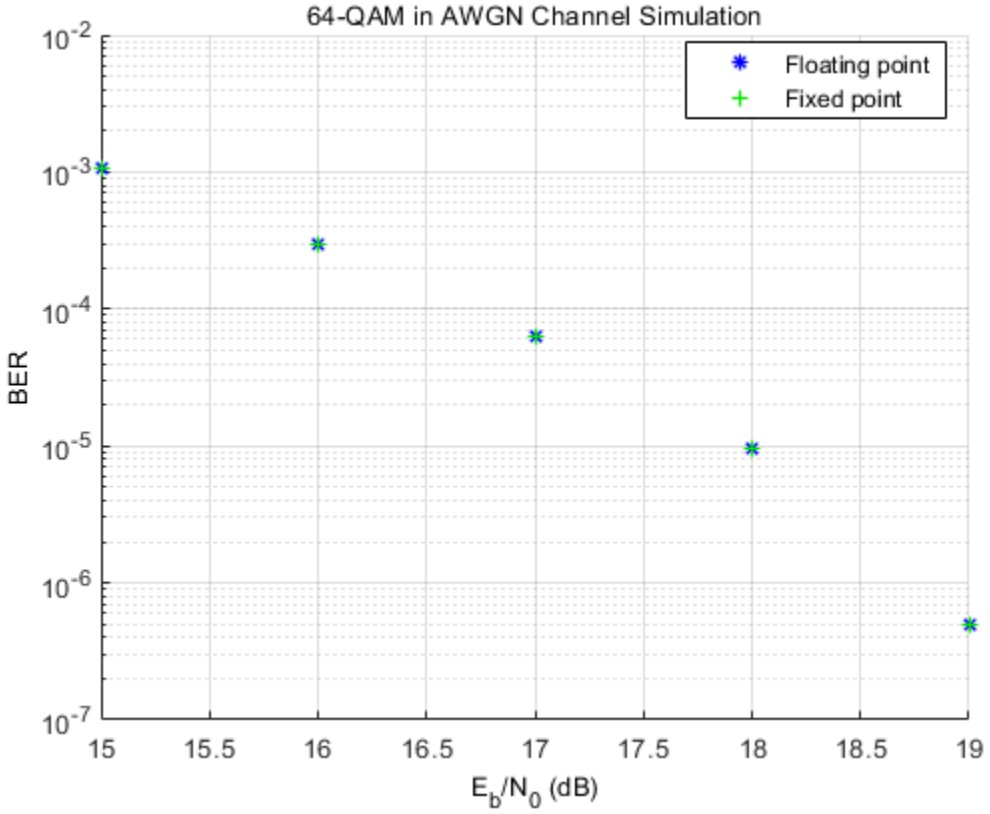
Differences Between the Fixed-Point and Floating-Point Example Versions

There are two different versions of this example -- a floating-point version and a fixed-point version. The examples are similar. In particular, most of the `Transmitter Baseband Processing` and `Receiver Baseband Processing` subsystems are identical, and mainly use unsigned integer data types in their signal paths.

The differences between the two versions are in how the signals are processed by the `Byte to M-tuple Conversion`, `64-QAM Constellation Mapping`, `Square Root Raised Cosine Tx Interpolation Filter`, `Square Root Raised Cosine Rx Decimation Filter`, `64-QAM Constellation Demapping`, and `M-Tuple to Byte Conversion` blocks. These blocks use floating-point (and built-in integer) arithmetic when their input and/or output signals are floating-point (i.e., data type `double` or `single`) or purely built-in integer (e.g., `uint8`), as is the case in the floating-point version (commdvbc.slx).

In the fixed-point version (commdvbc_fixpt.slx) however, these blocks use fixed-point arithmetic because their input and/or output signals are fixed-point data types (i.e., `sfix` or `ufix` in Simulink). Also note that a Fixed-Point Designer™ license is required to run the fixed-point version of the example.

The following simulation results show matching BER performance for the chosen settings when comparing the floating-point version with the fixed-point version.



Selected Bibliography

[1] *ETSI Standard EN 300 429 V1.2.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems*, European Telecommunications Standards Institute, Valbonne, France, 1998.

[2] *ISO/IEC 13818-1*, "Coding of moving pictures and associated audio."

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

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

Digital Video Broadcasting - Terrestrial

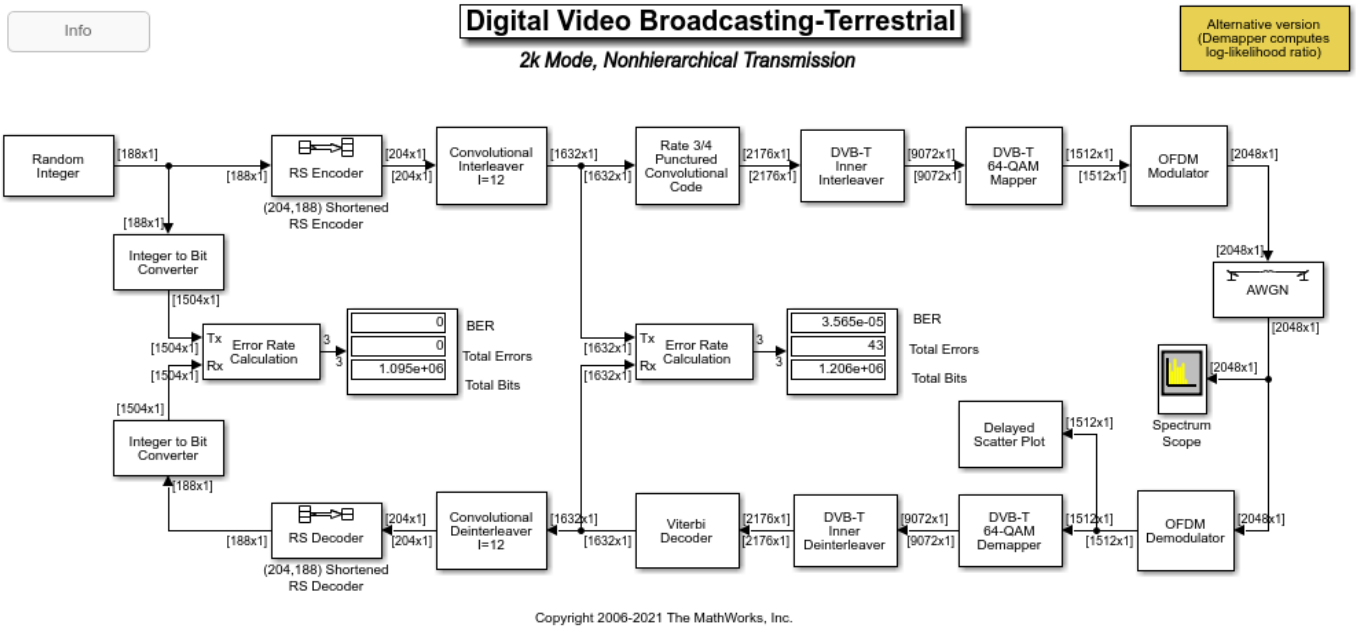
This model shows part of the ETSI (European Telecommunications Standards Institute) EN 300 744 standard for terrestrial transmission of digital television signals. The standard prescribes the transmitter design and sets minimum performance requirements for the receiver.

The purpose of this example is to

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

Structure of the Example

Using a list and a schematic, the standard shows the major processes that the data undergoes. The top row of blocks in the model mimics the structure of the schematic, by including subsystems that perform major processes.



The table below shows which subsystems correspond to processes from the schematic.

Process in Schematic	Subystem or Block in Demo
Outer coder	(204, 188) Shortened Reed-Solomon Encoder
Outer interleaver	Convolutional Interleaver I=12
Inner coder	Rate 3/4 Punctured Convolutional Code
Inner interleaver	DVB-T Inner Interleaver
Mapper	DVB-T 64-QAM Mapper
OFDM	OFDM Transmitter

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

Variables in the Model

The model uses variables as listed below.

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

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

Design of the Receiver

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

- A 64-QAM demapper that makes soft decisions, producing a set of six real numbers for each complex number in its input. These six numbers represent soft decisions on the real and imaginary components' first bit, second bit, and third bit. The Viterbi Decoder subsystem interprets the soft-decision numbers and uses them to decode the punctured convolutional code properly. To examine the exact mapping more closely, see the DVB-T 64-QAM Demapper subsystem, as well as the `dvbt_qam` variable.
- A traceback depth of 136 in the Viterbi Decoder library block. This library block appears within the top-level Viterbi Decoder subsystem.

Receive delay calculation

The DVB-T inner Interleaver and Deinterleaver contains the following frame size rebuffering and the corresponding delays:

- 2176 to 756 resulting in 756 sample delay
- 756 to 9072 resulting in 9072 sample delay
- 9072 to 756 resulting in 0 sample delay
- 756 to 2176 resulting in 2176 sample delay

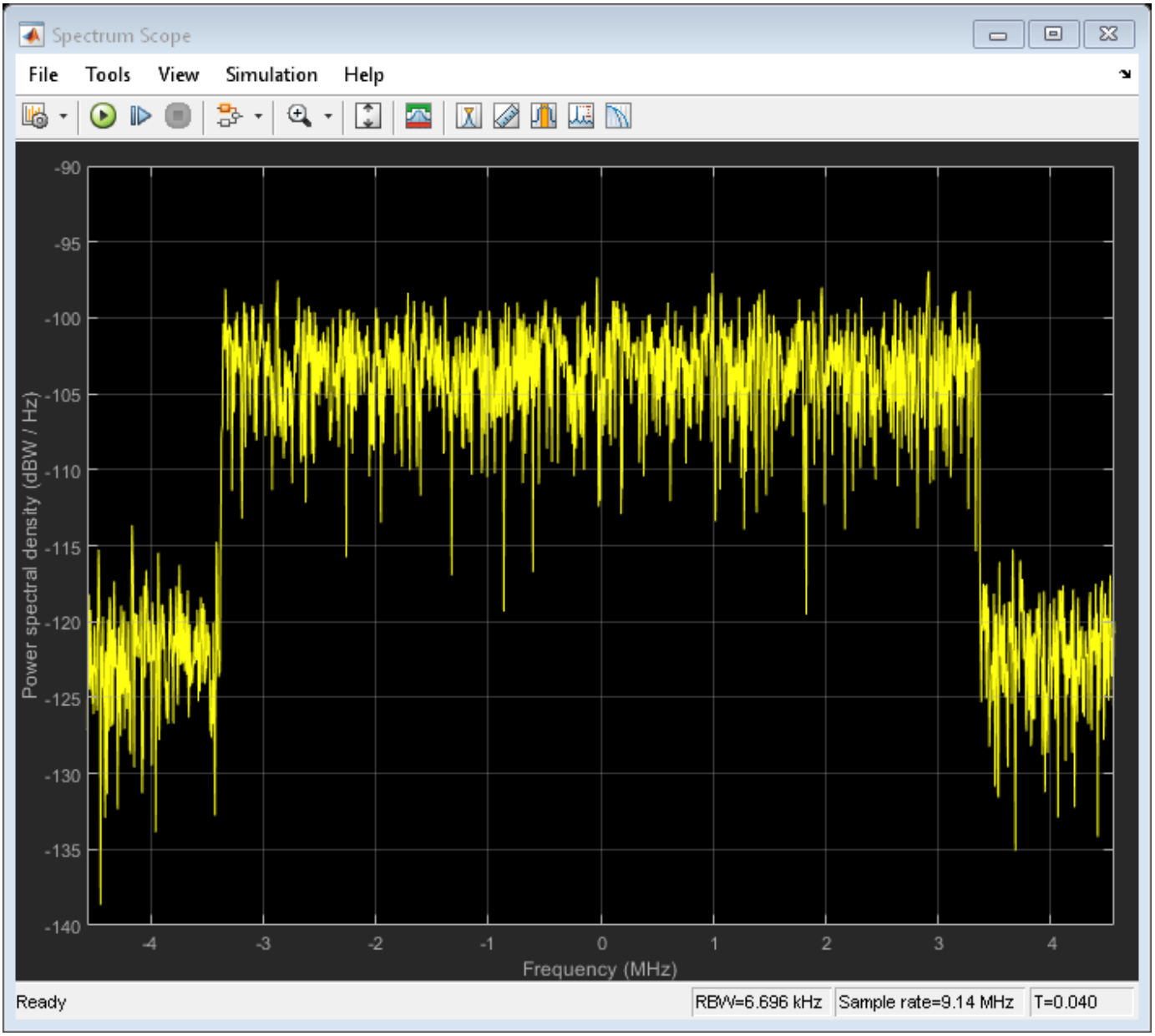
This results in a delay of 12004 samples. Since 2176 is the input frame size to the Viterbi Decoder mod(12004,2176) results in a delay of 1124 which corresponds to 1124*3/4 = 843 samples due to rate 3/4 coding. With a traceback depth of 136, the Viterbi decoder also adds a further delay of 136, bringing the total delay to 843+136 = 979. In order to align the actual codewords before feeding into the Convolutional Deinterleaver an extra delay of 1632-979 = 653 samples is added. Rate 3/4 coding also causes the 12004 delay to manifest as 12004*3/4 = 9003. Thus the total delay for the model excluding Convolutional Interleaving/Deinterleaving is 9003+136+653 = 9792 which is equal to 6 frames as the frame size at the 'inner' Error rate calculation block is 1632.

Convolutional Interleaving/Deinterleaving with 12 rows of shift registers adds a delay of 11 frames. Due to this the receive delay for the 'outer' error rate calculation block is a total of 6 + 11 = 17 frames.

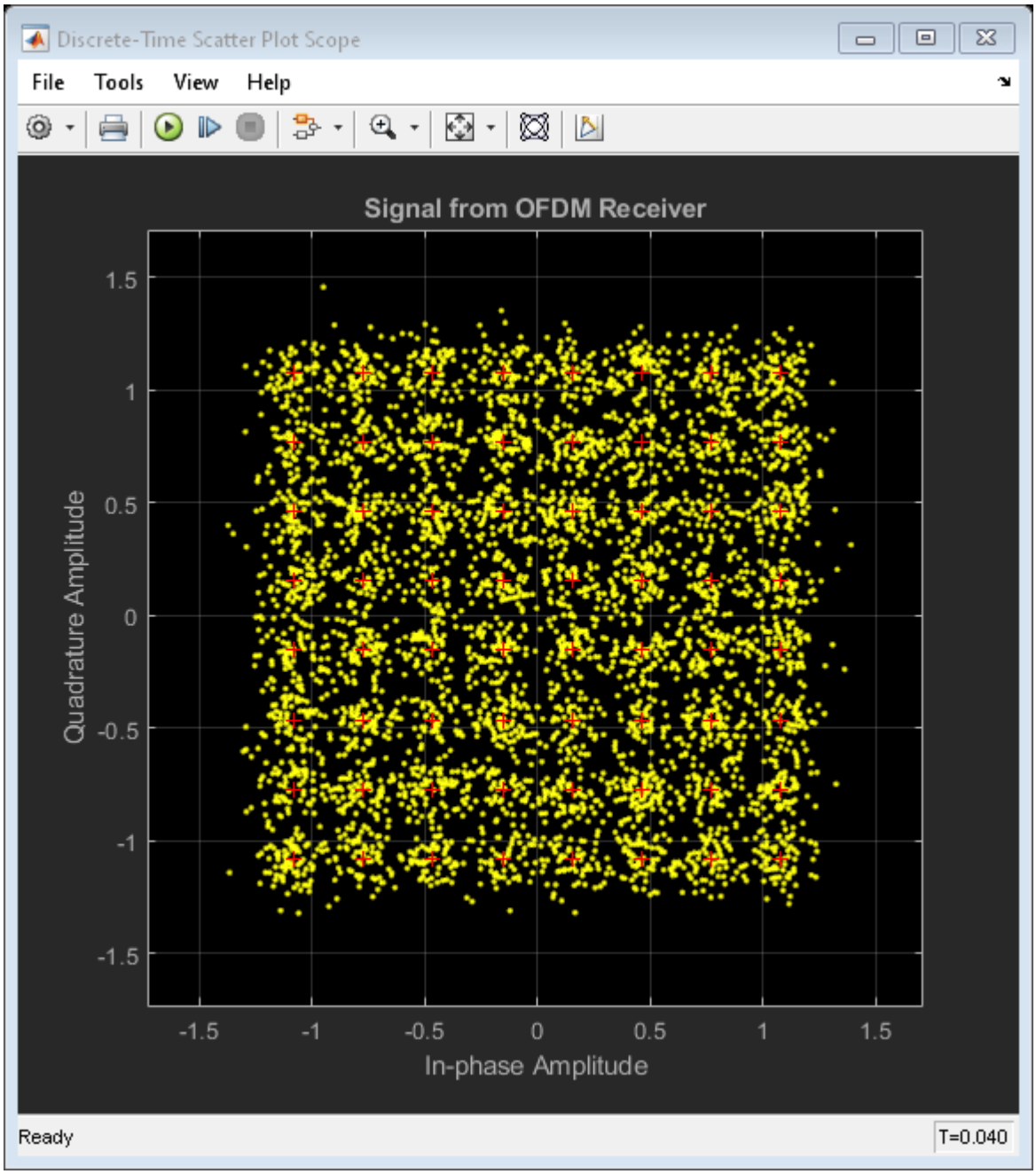
Results and Displays

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

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

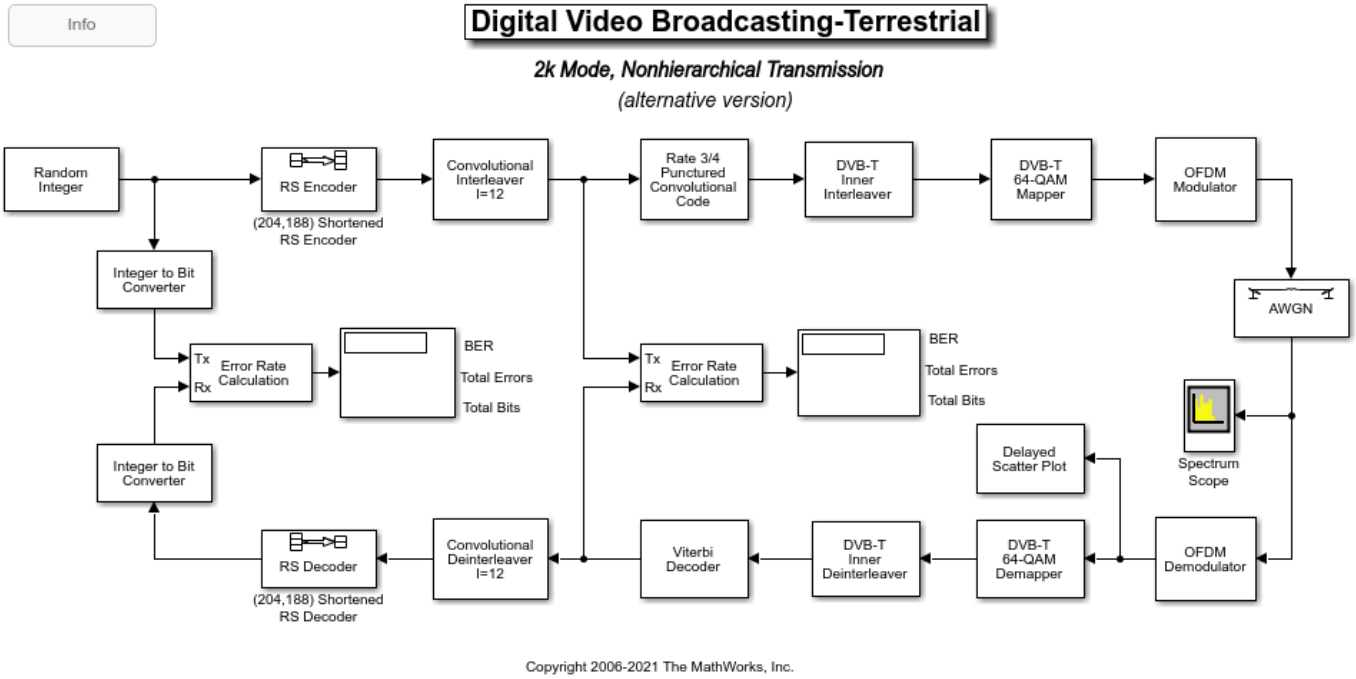


Spectrum Analyzer



Digital Video Broadcasting-Terrestrial, Alternate Form

The model `commdvbt_alt` illustrates an alternative way to model the 64-QAM Demapper in the receiver.



To see how the alternative version implements the 64-QAM Demapper, compare the alternative DVB-T 64-QAM Demapper subsystem in commdvbt_alt example with the original DVB-T 64-QAM Demapper subsystem in the commdvbt example.

Original: In the original form, soft decisions are computed using a subsystem-based implementation. In-phase and quadrature phase signal components are extracted after appropriately scaling the received signal, and then they are shifted to obtain soft decisions for various bits.

Alternative: In the alternative form, the built-in Rectangular QAM Demodulator block is configured to compute exact bitwise log-likelihood ratios (LLRs). Noise variance needs to be provided and it is computed using the received signal and the signal generated by the DVB-T 64-QAM Mapper. This approach makes derivation of soft decisions easy for any signal constellation through the use of the built-in block.

Selected Bibliography

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

IEEE 802.16-2009 WirelessMAN-OFDMA PHY Downlink PUSC

This model shows a downlink partial usage of subchannels (PUSC) Physical Layer communication from base station (BS) to two mobile stations (MS), according to the IEEE® 802.16-2009 standard [1].

Structure of the Example

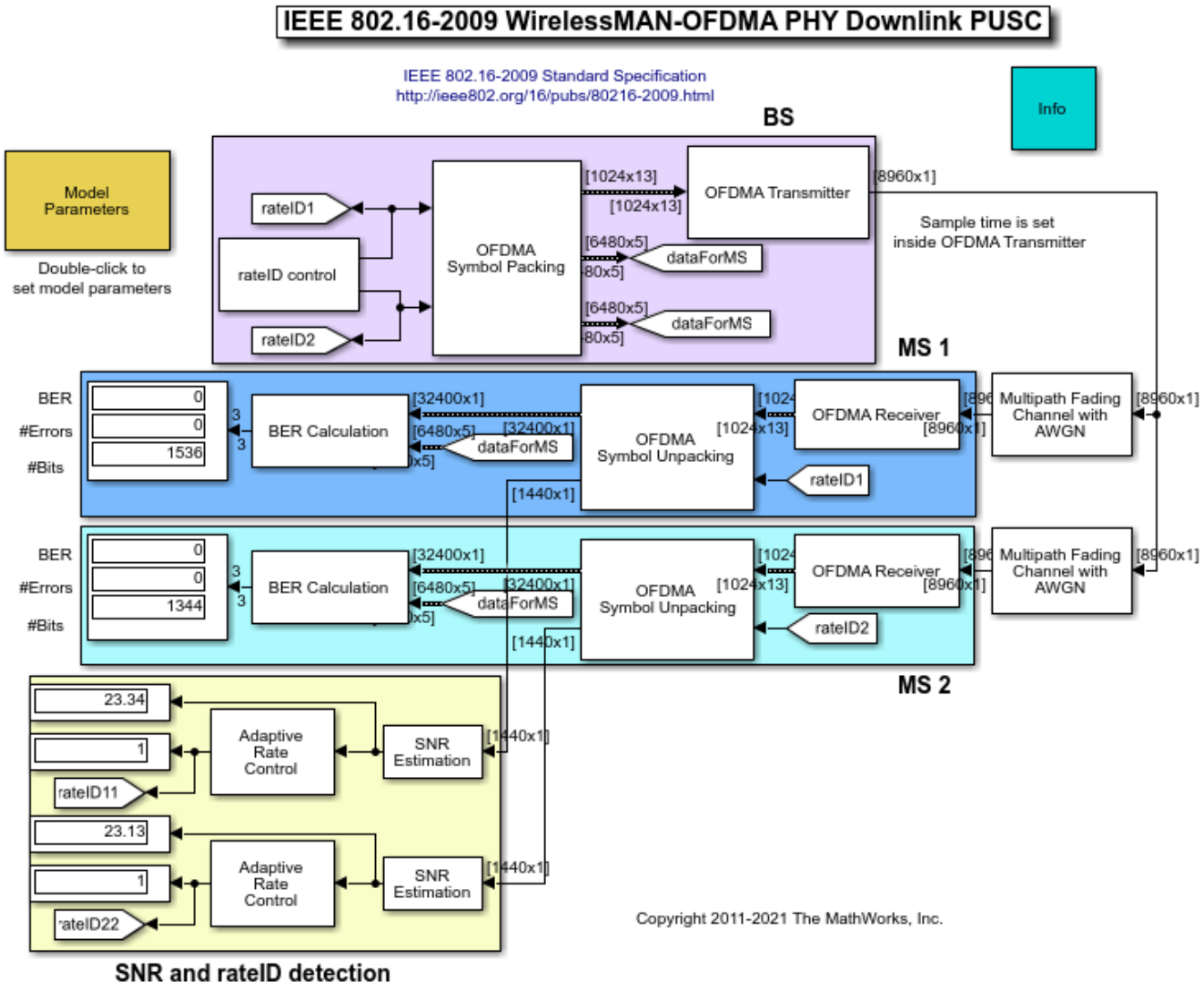
This example models the downlink PUSC of the WirelessMAN-OFDMA PHY. It supports all of the mandatory coding and modulation options. The purpose of this example is to showcase the variable-size capability of Simulink®, MATLAB® Function block, DSP System Toolbox™, and Communications Toolbox™. To simplify the implementation, the restriction of two MS (also referred to as users in the model) and 1024 FFT size are applied.

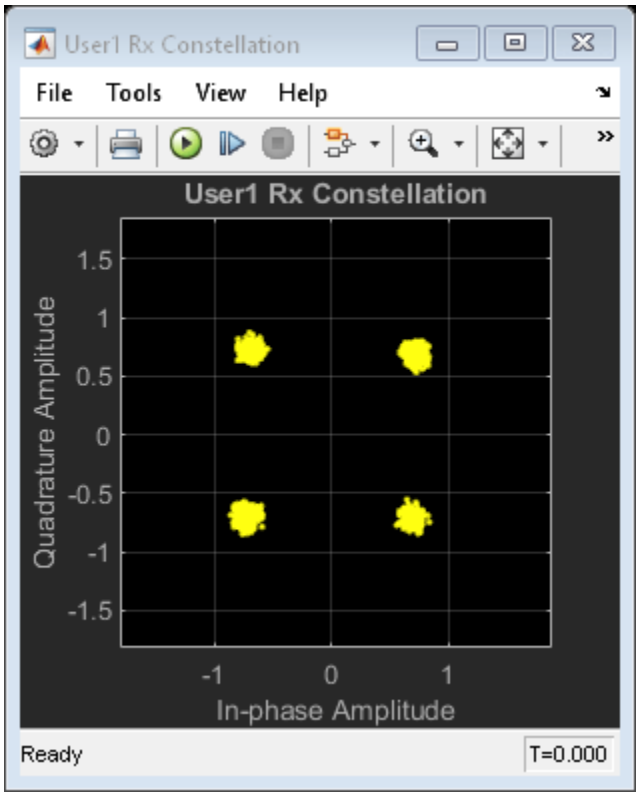
Out of 1024 frequency carriers (also called subcarriers), 720 subcarriers can be used to carry user data (the rest are reserved for pilots and guards). To properly allocate the data carriers to different MS, the standard organizes 720 subcarriers into 30 subchannels (each subchannel contains 24 subcarriers). A subchannel is the smallest unit that can be allocated to an MS.

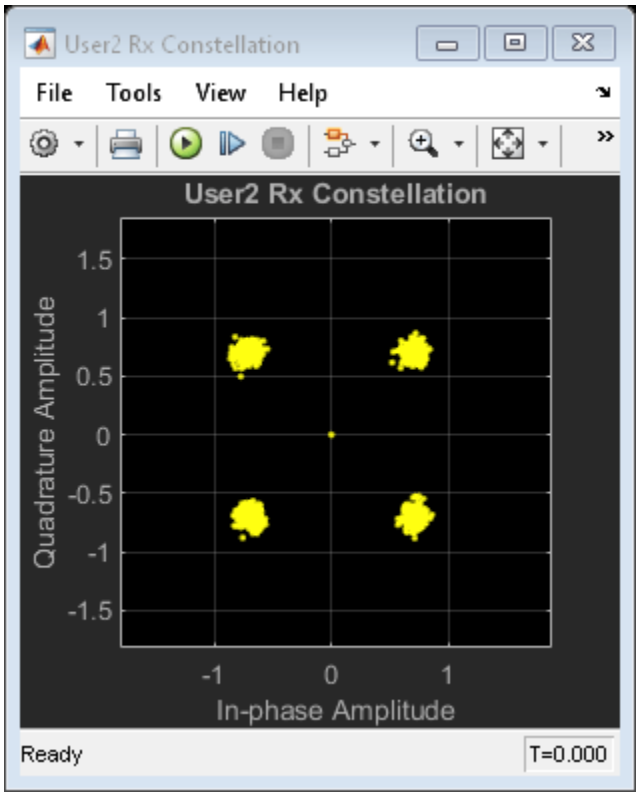
The standard allows frequency resources (in subchannels) be dynamically allocated to MS. This means while the model is running, BS can dynamically change the subchannel allocation to MS1 and MS2. For example, in one burst, subchannels 0~5 are allocated to MS1 and subchannels 6~25 are allocated to MS2. In another burst, the allocation may become 2~10 and 15~25 respectively. When more subchannels are allocated to one MS, more data can be transmitted to this MS in one burst. This dynamic change introduces variable-size signaling.

The variable-size features of the following tools are shown:

- Simulink blocks
- MATLAB Function block
- DSP System Toolbox blocks
- Communications Toolbox blocks and System objects


IEEE 802.16-2009 WirelessMAN-OFDMA PHY Downlink PUSC
IEEE 802.16-2009 Standard Specification
http://ieee802.org/16/pubs/80216-2009.html
Info
BS
Model Parameters
Double-click to set model parameters
rateID1
rateID control
rateID2
OFDMA Symbol Packing
[1024x13]
[1024x13]
OFDMA Transmitter
[8960x1]
Sample time is set inside OFDMA Transmitter
[6480x5]
dataForMS
[6480x5]
dataForMS
MS 1
BER
#Errors
#Bits
0
0
1536
BER Calculation
[32400x1]
[6480x5]
[32400x1]
dataForMS
OFDMA Symbol Unpacking
[1440x1]
[1024x13]
OFDMA Receiver
Multipath Fading Channel with AWGN
[8960x1]
rateID1
BER
#Errors
#Bits
0
0
1344
BER Calculation
[32400x1]
[6480x5]
[32400x1]
dataForMS
OFDMA Symbol Unpacking
[1440x1]
[1024x13]
OFDMA Receiver
Multipath Fading Channel with AWGN
[8960x1]
rateID2
MS 2
23.34
1
rateID11
Adaptive Rate Control
SNR Estimation
[1440x1]
23.13
1
rateID22
Adaptive Rate Control
SNR Estimation
[1440x1]
SNR and rateID detection
Copyright 2011-2021 The MathWorks, Inc.


User1 Rx Constellation
File
Tools
View
Help
User1 Rx Constellation
Quadrature Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-1
0
1
In-phase Amplitude
Ready
T=0.000


User2 Rx Constellation
File
Tools
View
Help
User2 Rx Constellation
Quadrature Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-1
0
1
In-phase Amplitude
Ready
T=0.000

OFDMA Symbol Packing Subsystem

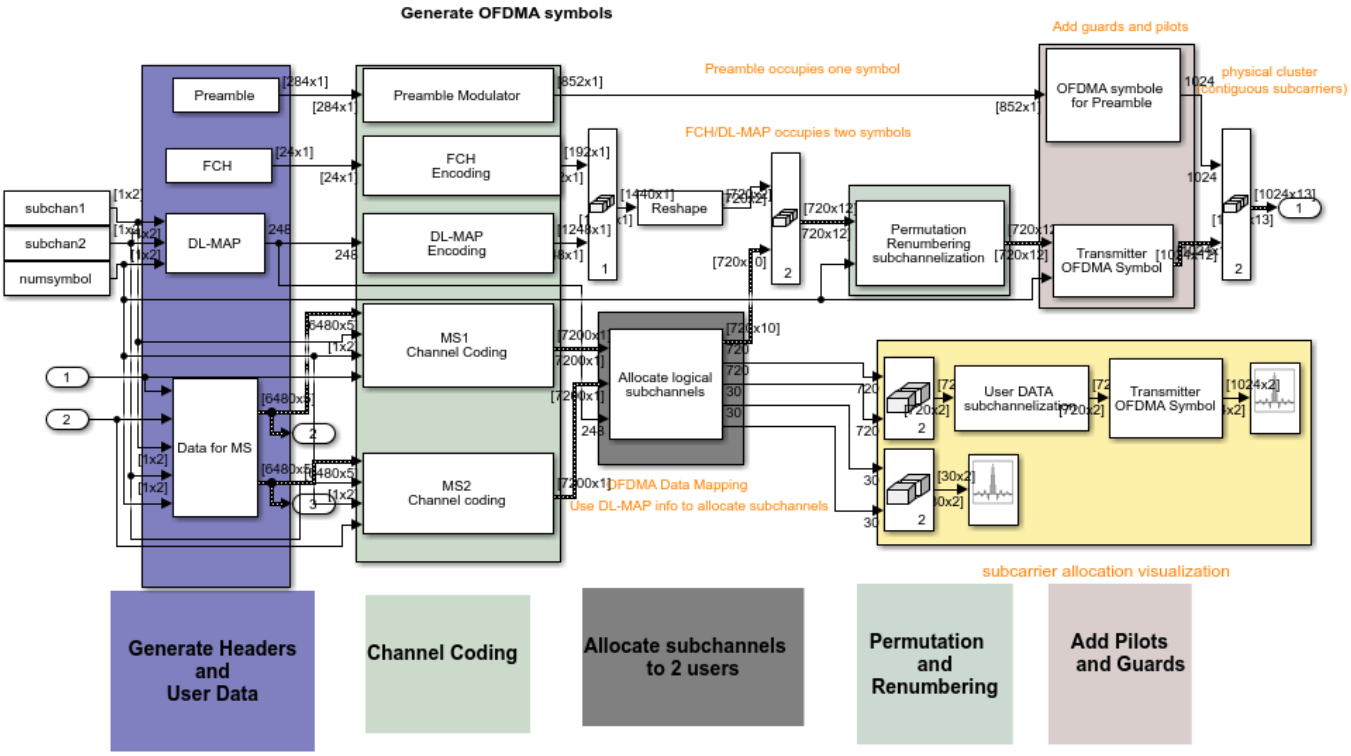
This subsystem, organized into five parts, generates data and packs it into OFDMA symbols:

- Generate Headers and User Data
- Channel Coding
- Allocate Subchannels
- Permutation and Renumbering
- Add Pilots and Guards

Variable-size processing happens in this subsystem. Signals output from **Data for MS** are variable-size because the two MS can be dynamically assigned subchannels for data transmission. The **MS1(2) Channel Coding** block includes Randomization, Interleaving, and all seven mandatory coding and modulation specified in the standard:

Rate_ID	Modulation Tail-biting CC rate
0	QPSK 1/2
1	QPSK 3/4
2	16-QAM 1/2
3	16-QAM 3/4
4	64-QAM 1/2
5	64-QAM 2/3
6	64-QAM 3/4

Channel coding is applied block by block. The block size is dependent of the number of subchannels allocated. The example illustrates how data is concatenated into blocks and how variable-size signals are processed by using blocks and System objects. To see more details, go to this block: **OFDMA Symbol Packing-->MS1 Channel Coding--> QPSK-1/2**.



OFDMA Transmitter/Receiver Subsystem

OFDMA Transmitter includes:

- Transforms signal from frequency domain to time domain (A gain block is used to scale the transmitted signal to unit power)
- Adds Cyclic Prefix
- Sets sample time for the model

In this model, data is driven by the transmitter port. To avoid sample time confusions, we try to set the system sample time at one place, which is at the output port of the **OFDMA Transmitter** block. Signal before this point is considered as data and data is drawn from the source to fit the sample time specified.

OFDMA Receiver includes:

- Removes cyclic prefix
- Transforms signal from time domain to frequency domain
- Implements the frequency domain equalization

According to the standard, a symbol is divided into 60 basic clusters. Two pilot carriers and 12 data carriers are allocated within each cluster. The receiver can estimate the response of the channel based on the known pilot information. Because the channel response may be different at different frequencies, the actual response for a data subcarrier is interpolated based on measurements of pilot subcarriers.

OFDMA Symbol Unpacking Subsystem

This subsystem unpacks the OFDMA symbols it receives by:

- Removing the DC and left/right guards from the Preamble symbol
- Separating FCH and DL-MAP from the user DATA
- Using FCH to detect DL-MAP message
- Using DL-MAP to separate user data for MS1 and MS2
- Performing channel decoding

Exploring the Example

1. Fixed Settings You cannot change the following default settings of the model:

- 1024 FFT size
- Two users

2. Channel Conditions Channel configuration can be set in the two Channel blocks.

The following channels can be simulated:

- AWGN only
- Flat Fading Channel with AWGN
- Frequency-selective Multipath fading with AWGN

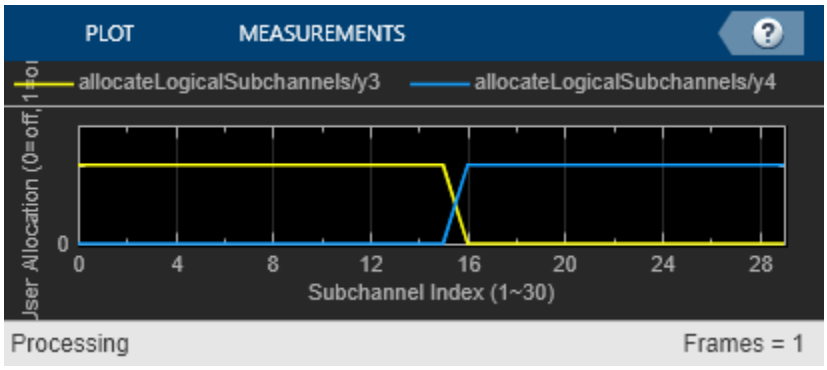
SNR and **Fading mode** are both tunable at run time.

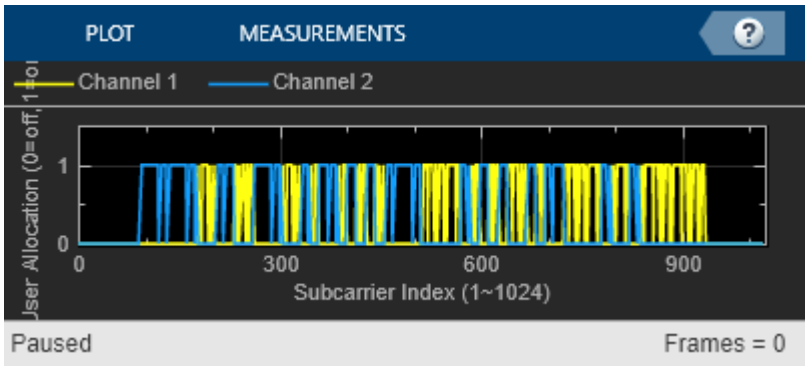
3. Other Model Parameters You can set all the other changeable parameters from the **Model Parameter** block.

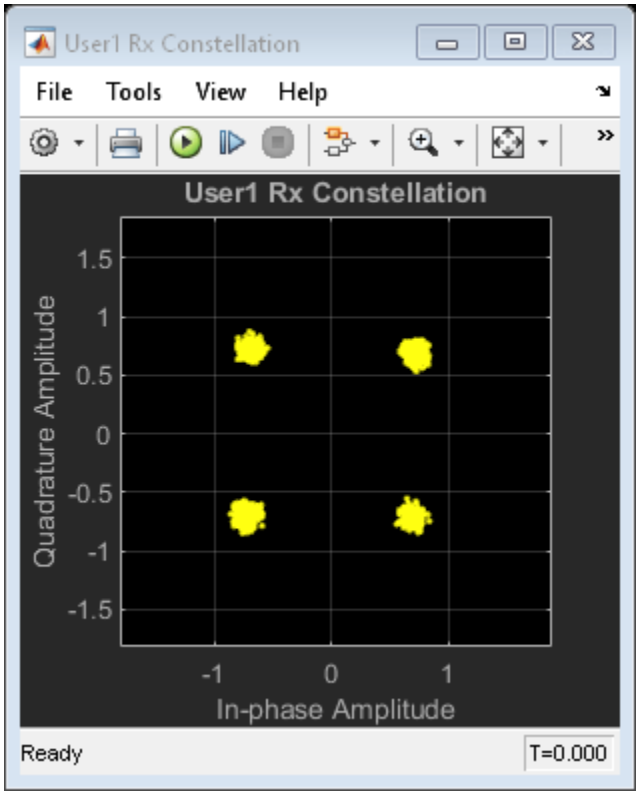
Among those parameters, the **Subchannels allocated to users** parameter is tunable at run time. Based on this parameter, DL-MAP message is packed and transmitted. Receivers use the detected DL-MAP message to decode information from the subchannels assigned to them. The subchannel allocation status is shown in **Subchannel allocation scope**; the subcarrier allocation status is shown in **Subcarrier allocation scope**.

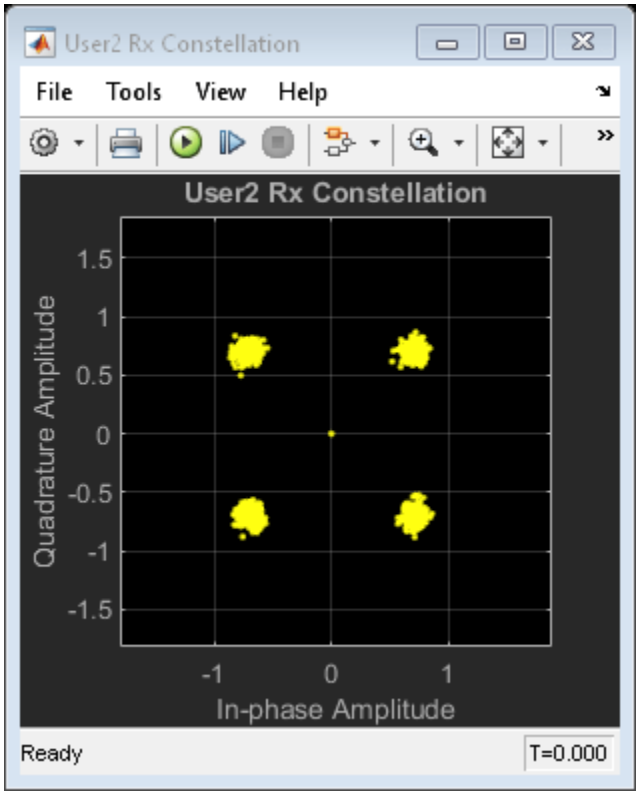
You can specify the modulation and coding rate or calculate them adaptively based on the channel conditions detected. When you select **Adapt modulation and coding to channel conditions**, you specify the **Adaptive rate control SNR thresholds (dB)**. When unchecked, you must specify the **Modulation** and **Coding rate** parameters.

To ensure the proper memory usage, this example limits the maximum number of OFDMA symbols in one burst to 13 (10 data symbols + 3 header symbols).


PLOT
MEASUREMENTS
allocateLogicalSubchannels/y3
allocateLogicalSubchannels/y4
User Allocation (0=off,1=on)
0
0
4
8
12
16
20
24
28
Subchannel Index (1~30)
Processing
Frames = 1


PLOT
MEASUREMENTS
Channel 1
Channel 2
User Allocation (0=off,1=on)
1
0
0
300
600
900
Subcarrier Index (1~1024)
Paused
Frames = 0


User1 Rx Constellation
File
Tools
View
Help
User1 Rx Constellation
Quadrature Amplitude
1.5
1
0.5
0
-0.5
-1
-1.5
-1
0
1
In-phase Amplitude
Ready
T=0.000



Notes

We make effort to follow the standard closely and make certain assumptions when needed. The following is a list of assumptions applied:

- The number of OFDMA symbol for both MS1 and MS2 in one burst are the same and not tunable at run time
- IDcell of '0' is used
- The first symbol is always Preamble
- The second and third symbols are FCH+DL-MAP (pad zeros at the end)
- User data starts in the fourth symbol
- Receivers use FCH and DL-MAP message to decode the received signal. If channels are too noisy, these message may corrupt easily. Since there is no resend request mechanism implemented, the model will error out. To avoid FCH and DL-MAP messages corruption, configure channels properly.

Selected References

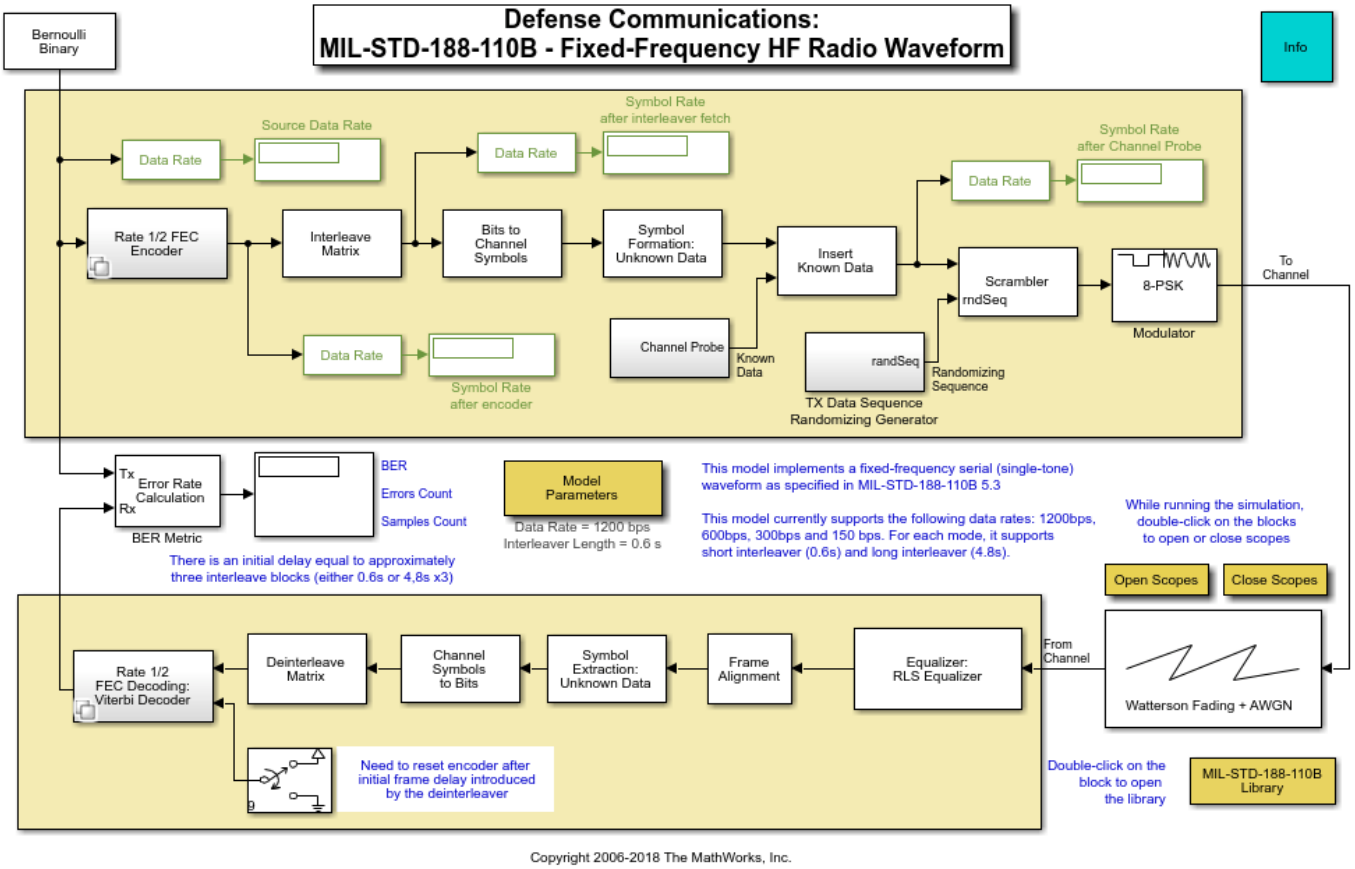
1 IEEE Standard 802.16-2009, "Part 16: Air Interface for Broadband Wireless Access Systems," May 2009. http://ieee802.org/16/published.html

Defense Communications: US MIL-STD-188-110B Baseband End-to-End Link

This model shows an end-to-end baseband communications system compliant with the U. S. MIL-STD-188-110B military standard. In particular, the model implements the data phase transmission, using a fixed-frequency serial (single-tone) waveform. This model supports these data rates: 150 bps, 300 bps, 600 bps, and 1200 bps. It also implements interleaver lengths of 0.6 s and 4.8 s.

The system described in this standard is intended for long-haul and tactical communications over HF (high frequency) channels. The system is compatible with the NATO standard STANAG 4539.

Structure of the Example



The communication system in this example performs these tasks:

- Generation of random binary data.
- Coding that depends on the data rate that you select in the Model Parameters block's dialog box. The Encoder block at the top level of the block diagram is a subsystem whose contents depend on the selected data rate. In all cases, this subsystem contains a convolutional encoder that uses a rate 1/2 code with constraint length 7. However, the subsystem can achieve rate 1/4 or 1/8 by following the encoder with a repetition operation.
- Interleaving using a matrix specified by the standard.

- Binary-to-Gray mapping.
- Appending the training sequence, also referred to as the known data or the channel probe symbols. By contrast the unknown symbols are the data that the user wants to transmit.
- Data scrambling, by adding the data to a randomizing sequence modulo 8.
- 8-PSK modulation.
- Watterson channel model, implemented using the SISO Fading Channel library block. Specifically, the block implements the moderate channel model described in [2], using a Gaussian Doppler spectrum.
- Receiver equalization using an RLS equalizer. Internally, the equalizer subsystem scrambles the training sequence so as to compare corresponding data sets, introduces delays to align frame boundaries, and descrambles the equalized signal.
- Channel symbol demapping.
- Deinterleaving.
- Viterbi decoding. The decoder is a subsystem that mirrors the encoder subsystem. The decoder includes a reset port, because it is necessary to reset the Viterbi decoder after an initial delay period elapses.

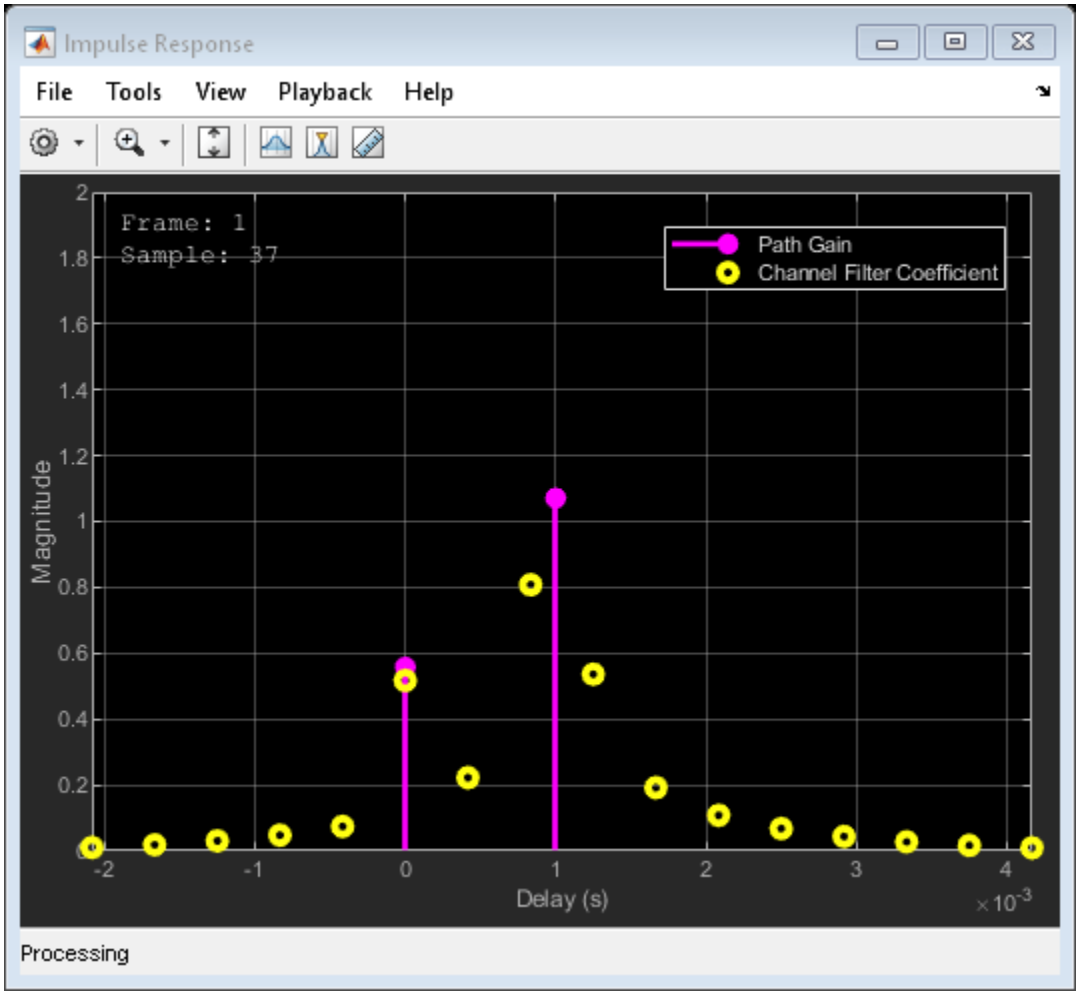
Other Features of the Example:

- Inside the encoder subsystem is an icon labeled "Compare FEC Encoder." You can double-click it to open another Simulink model that compares the block diagram appearing in the standard with the single Convolutional Encoder block in the Communications Toolbox™. The model illustrates that the two ways of modeling the convolutional code yield the same results.
- Inside the Interleave Matrix subsystem is an icon labeled "Interleave Mapping." You can double-click it to open a plot that shows the mapping, which depends on your choices in the Model Parameters dialog box.

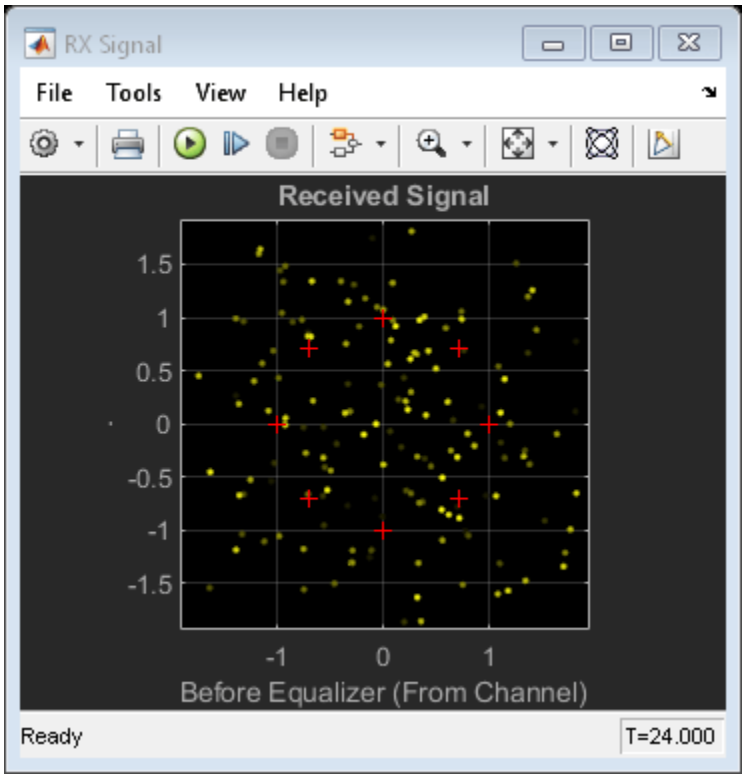
Results and Displays

When you run the simulation, it displays these numerical or graphical results:

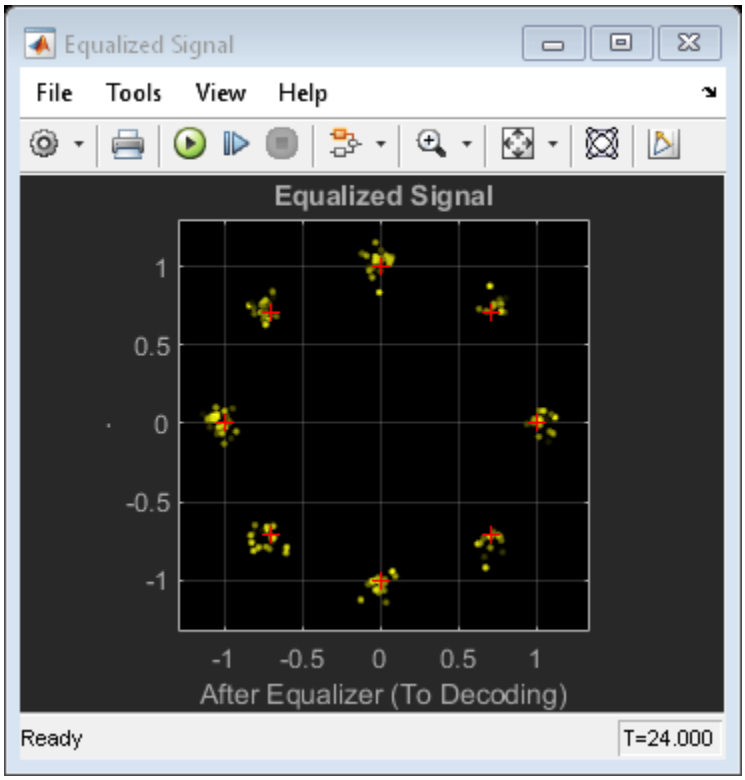
- The bit error rate of the entire system.
- The data rate at several points during the simulation. The source data rate is the one that you specify in the Model Parameters dialog box, while the last displayed data rate (before the Scrambler) is always 2400 bps. The other displayed data rates depend on your choices in the Model Parameters dialog box.
- The Watterson channel impulse response.



- Constellation diagram of the signal before equalization.



- Constellation diagram of the signal after equalization.



Simulink® Techniques Illustrated in the Example

The coding behavior in the standard depends on the data rate. This model varies the behavior of the coding and decoding subsystems depending on the **Information Rate** parameter that you select in the Model Parameters dialog box. Double-clicking the encoder or decoder icon enables you to see the contents of the subsystem based on the current value of the **Information Rate** parameter. When you change the **Information Rate** parameter, an initialization function associated with the Model Parameters block sets certain model parameters and also chooses the contents of the encoder and decoder subsystems.

Selected Bibliography

[1] MIL-STD-188110B: Interoperability and Performance Standards for Data Modems, U. S. Department of Defense, 2000.

[2] ITU-R Recommendation 520-2: Use of High Frequency Ionospheric Channel Simulators, 1978/1982/1992.

See Also

The Communications Toolbox example Defense Communications: US MIL-STD-188-110A Receiver shows a MIL-STD-188-110A receiver, with preamble detection, carrier synchronization, and symbol timing synchronization. It runs at a fixed rate of 1200 bps.

WCDMA End-to-End Physical Layer

This model shows part of the frequency division duplex (FDD) downlink physical layer of the third generation wireless communication system known as wideband code division multiple access (WCDMA).

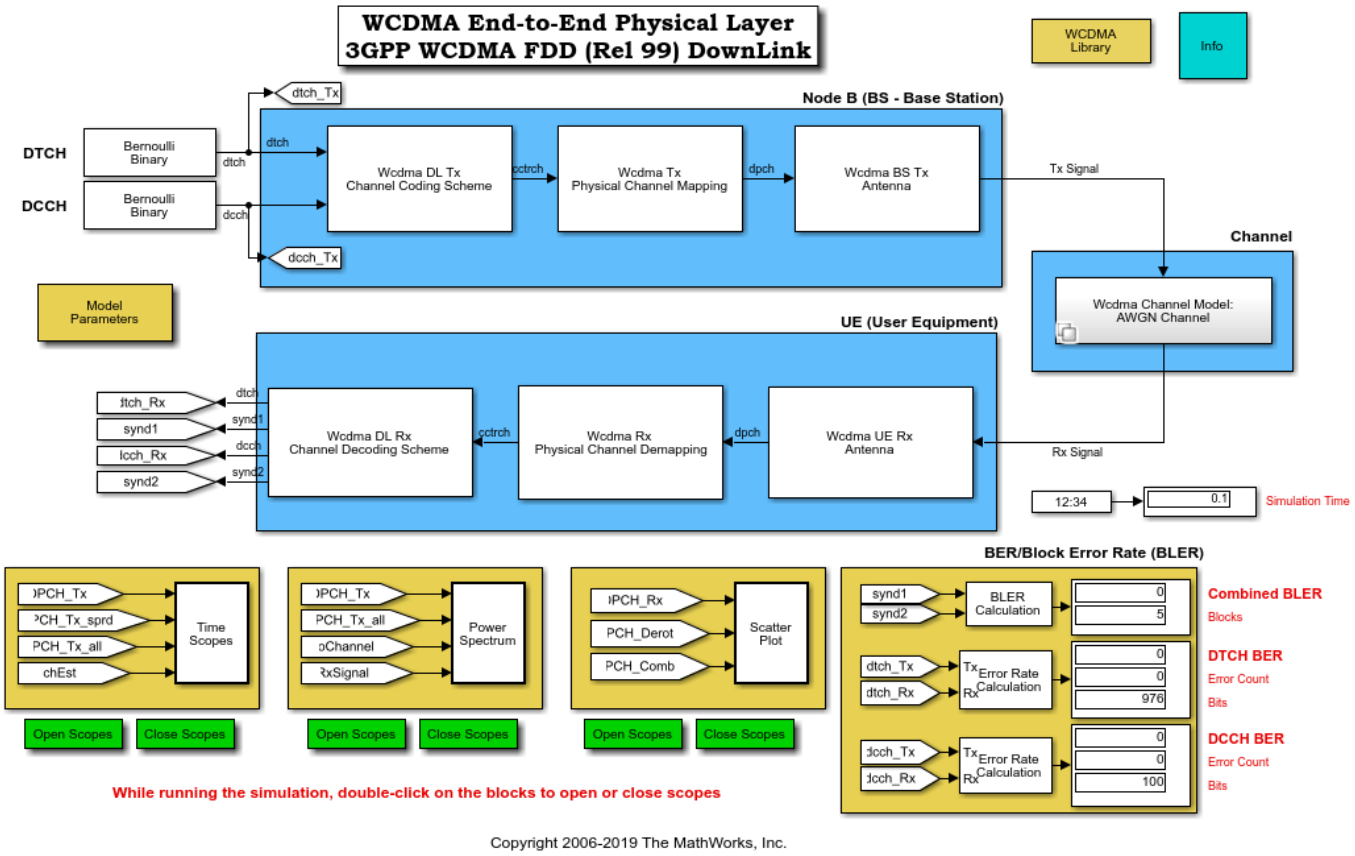
WCDMA is one of five air interfaces for the third generation of wireless communications developed within the framework of the International Mobile Telecommunications (IMT)-2000, as defined by the International Telecommunication Union (ITU). The WCDMA technology is officially known as IMT-2000 Direct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

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

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

Structure of the Example

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



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

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

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

- Cyclic redundancy code (CRC) attachment
- Transport block concatenation and segmentation

- Channel encoding
- Rate matching
- First interleaving
- Radio frame segmentation

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

WCDMA Tx Physical Mapping. This subsystem implements the following functions:

- Physical channel segmentation
- Second interleaver
- Slot builder

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

WCDMA BS Tx Antenna. The WCDMA BS Tx Antenna subsystem performs the following functions:

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

WCDMA Channel Model. The WCDMA Channel Model subsystem simulates a wireless link channel containing additive white Gaussian noise (AWGN) and, if selected, a set of multipath propagation conditions. You can modify the multipath profile with the Propagation conditions environment parameter, as described under **Exploring the Example**.

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

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

Exploring the Example

You can view or change parameters in the model by double-clicking the block labeled Model Parameters. This displays the **Block Parameters** dialog.

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

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

Parameter	Description
DL measurement channels	The down link (DL) measurement channels. There are four channels whose settings are specified by the standard: • 12.2 Kbps • 64 Kbps • 128 Kbps • 384 Kbps If you select one of these channels, the following parameters are inactive. To change these parameter settings, select `User Defined`.
Transport block set size	Integer row vector representing the transport block set size as defined by the standard associated with each transport channel.
Transport block size	Integer row vector representing the transport block size as defined by the standard associated with each transport channel.
TTI in ms	Integer row vector representing the transmission time interval (TTI) in ms as defined by the standard associated with each transport channel.
CRC Size	Integer row vector representing the CRC size in the number of bits associated with each transport channel.
Type of error protection	Integer row vector representing the coding scheme associated with each transport channel. The options are • 0 for no coding • 1 for ½ rate convolutional coding • 2 for 1/3 rate convolutional coding • 3 for 1/3 rate turbo coding
Rate matching attribute	Integer row vector representing the rate matching attribute as defined by the standard associated with each transport channel.
Position of TrCH in radio frame	Sets the position of the transport channels in the radio frame to be `Fixed` or `Flexible` as defined by the standard.
Number of PhCH	Integer from `1` to `3` corresponding to the number of physical channels used.
Slot Format (0...16)	Sets the corresponding slot format parameter as defined by the standard.

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

Parameter	Description
DPCH Code number	Integer from 0 to the value of the spreading factor minus 1, corresponding to the index of the orthogonal code assigned to the DPCH channel.
Scrambling code	Vector of two elements, corresponding to the index of the scrambling code assigned to the base station.
Number of filter taps for RRC	Number of filter coefficients for the root-raised cosine filter.
Number of coefficients for channel estimation filters	Number of filter coefficients for the lowpass filter implemented in channel estimation.
Oversampling factor	Integer value corresponding to the number of samples per symbol.

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

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

Results and Displays

The following blocks calculate various error rates in the example:

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

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

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

Accompanying Models

The following two models offer standalone implementation of some of the subsystems included in this example model:

commwcdmamuxandcoding.slx: shows the WCDMA DL Tx Channel Coding Scheme with Physical Channel Mapping and WCDMA Physical Channel Demapping with the Rx Channel Decoding Scheme.

commwcdmaspreadandmod.slx : shows WCDMA BS Tx Antenna and WCDMA UE Rx Antenna.

Selected Bibliography

https://www.3gpp.org

BER Simulations with Parallel Computing Toolbox

This example shows how to improve the execution speed of communication systems involving BER simulations. To improve the performance of these systems, one of the available options is to parallelize the simulations. This example introduces the usage of the Parallel Computing Toolbox™ (PCT) in BER simulations. It presents two possible ways of parallelizing BER simulations and recommends the better method.

License Check and Opening a Parallel Pool

This section checks for the availability of PCT. If available, it opens a parallel pool of workers and assigns the maximum number of available workers in the pool to the variable numWorkers. If not available it assigns numWorkers = 1, in which case the example runs on a single core.

```
[licensePCT,~] = license( 'checkout','Distrib_Computing_Toolbox');
if ( licensePCT && ~isempty(ver('parallel')))
    if isempty(gcp('nocreate'))
        parpool;
    end
    pool = gcp;
    numWorkers = pool.NumWorkers;
else
    numWorkers = 1;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
```

Initialization

This example parallelizes the “Spatial Multiplexing” on page 8-212 example to demonstrate the usage of PCT. The following are the parameters needed to simulate this example.

```
EbNo = 1:2:11;  % Eb/No in dB
N = 2;          % Number of transmit antennas
M = 2;          % Number of receive antennas
modOrd = 2;     % constellation size = 2^modOrd
numBits = 1e6;  % Number of bits
numErrs =  100; % Number of errors
lenEbNo = length(EbNo);
% Create a local random stream to be used for data generation for
% repeatability. Use the combined multiple recursive generator since it
% supports substreams.
hStr = RandStream('mrg32k3a'); % Setting the random stream
[berZF,berMMSE] = deal(zeros(lenEbNo,3));
[nerrsZF,nbitsZF,nerrsMMSE,nbitsMMSE] = deal(zeros(numWorkers,lenEbNo));
```

Parallelizing Across the Eb/No Range

The first method parallelizes across the Eb/No range, where one worker processes a single Eb/No value. Here the performance is limited by the time required to process the highest Eb/No value.

```
simIndex = 1;
str = 'Across the Eb/No range';
disp('Performing BER simulations with one worker processing one Eb/No value ...');
```

```
Performing BER simulations with one worker processing one Eb/No value ...
```

```
tic
parfor idx = 1:lenEbNo
    [BER_ZF,BER_MMSE] = simBERwithPCT(N,M,EbNo,modOrd, ...
        idx,hStr,numBits,numErrs);
    berZF(idx,:) = BER_ZF(idx,:);
    berMMSE(idx,:) = BER_MMSE(idx,:);
end
timeRange = toc;
clockBERwithPCT(simIndex,timeRange,timeRange,str);
```

Parallelizing Across the Number of Workers in the Parallel Pool

The second method parallelizes across the number of available workers, where each worker processes the full Eb/No range. However, each worker counts (total errors/numWorkers) errors before proceeding to the next Eb/No value. This method uses all available cores equally efficiently.

```
simIndex = simIndex + 1;
str = 'Across the number of available workers';
seed = 0:numWorkers-1;
disp('Performing BER simulations with each worker processing the entire range ...');
```

```
Performing BER simulations with each worker processing the entire range ...
```

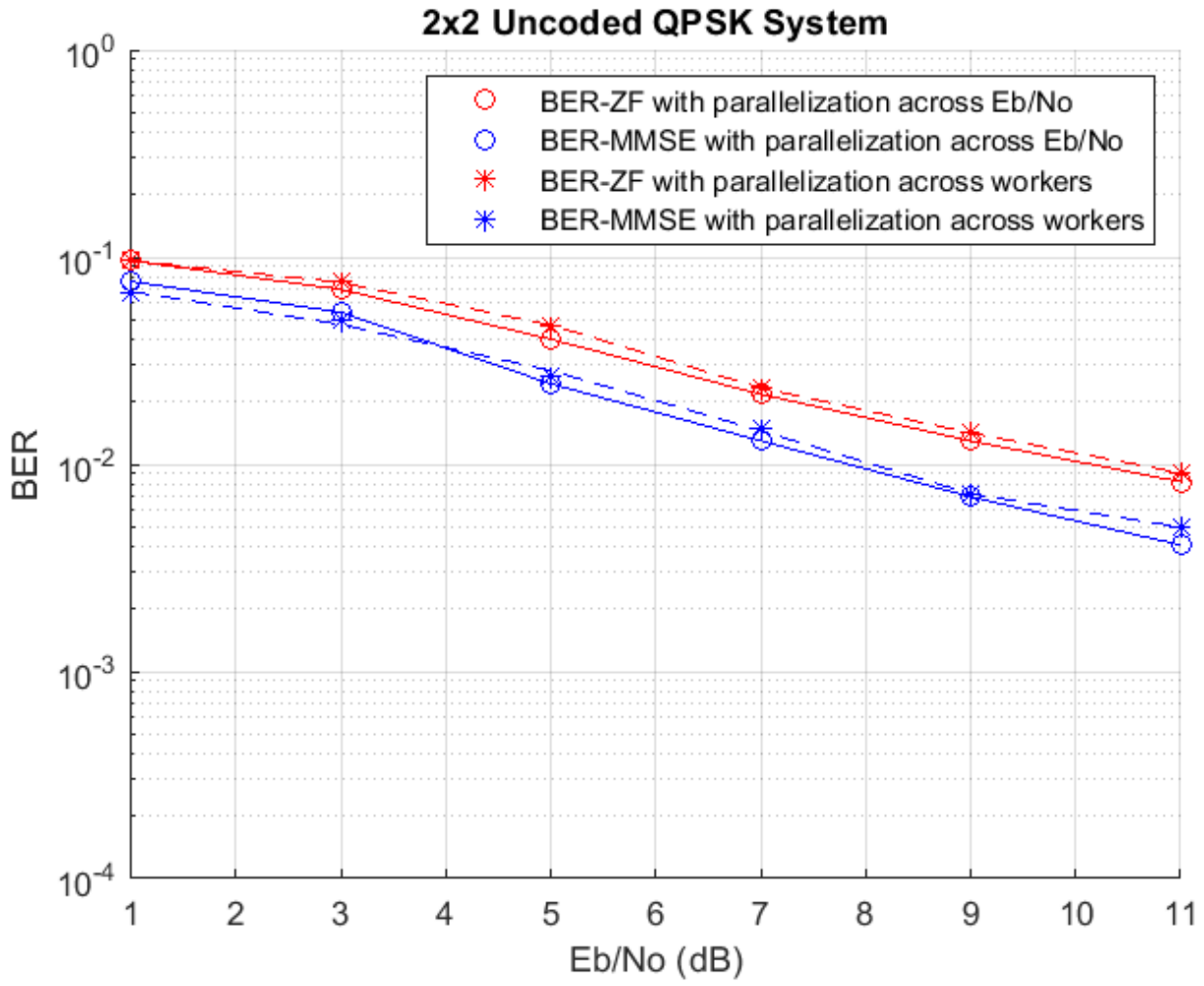
```
tic
parfor n = 1:numWorkers
    hStr = RandStream('mrg32k3a','Seed',seed(n));
    for idx = 1:lenEbNo
        [BER_ZF,BER_MMSE] = simBERwithPCT(N,M,EbNo,modOrd, ...
            idx,hStr,numBits/numWorkers,numErrs/numWorkers);
        nerrsZF(n,idx) = BER_ZF( idx,2);
        nbitsZF(n,idx) = BER_ZF( idx,3);
        nerrsMMSE(n,idx) = BER_MMSE(idx,2);
        nbitsMMSE(n,idx) = BER_MMSE(idx,3);
    end
end
bZF = sum(nerrsZF,1)./sum(nbitsZF,1);
bMMSE = sum(nerrsMMSE,1)./sum(nbitsMMSE,1);
timeWorker = toc;
```

Below are the results obtained on a Windows® 7, 64-bit, Intel® Xeon® CPU W3550, ~3.1GHz, 12.288GB RAM machine using four cores. The table shows the performance comparison of the above methods. We see that the second method performs better than the first. These are the results obtained on a single run and may vary from run to run.

```
-------------------------------------------------------------------------------------------
Type of Parallelization                          | Elapsed Time (sec)| Speedup Ratio
1. Across the Eb/No range                        |           89.7366 |       1.0000
2. Across the number of available workers        |           28.4443 |       3.1548
-------------------------------------------------------------------------------------------
```

The plot below shows the BER curves obtained for the zero forcing (ZF) and minimum mean squared error (MMSE) receivers using the different parallelization methods.

```
plotBERwithPCT(EbNo,berZF(:,1),berMMSE(:,1),bZF,bMMSE);
```



To generate a performance comparison table for your machine, uncomment the following line of code and run this entire script.

```
% clockBERwithPCT(simIndex,timeRange,timeWorker,str);
```

Appendix

The following functions are used in this example:

- simBERwithPCT.m
- plotBERwithPCT.m
- clockBERwithPCT.m

End to End System Simulation Acceleration Using GPUs

This example shows a comparison of four techniques which can be used to accelerate bit error rate (BER) simulations using System objects in the MATLAB® Communications Toolbox™ software. A small system, based on convolutional coding, illustrates the effect of code generation using the MATLAB® Coder™ product, parallel loop execution using `parfor` in the Parallel Computing Toolbox™ product, a combination of code generation and `parfor`, and GPU-based System objects.

The System objects this example features are accessible in the Communications Toolbox product. In order to run this example you must have a MATLAB Coder license, a Parallel Computing Toolbox license, and a sufficient GPU.

System Design and Simulation Parameters

This example uses a simple convolutional coding system to illustrate simulation acceleration strategies. The system generates random message bits using `randi`. A transmitter encodes these bits using a rate 1/2 convolutional encoder, applies a QPSK modulation scheme, and then transmits the symbols. The symbols pass through an AWGN channel, where signal corruption occurs. QPSK demodulation occurs at the receiver, and the corrupted bits are decoded using the Viterbi algorithm. Finally, the bit error rate is computed. The System objects used in this system are :

- comm.ConvolutionalEncoder - convolutional encoding
- comm.PSKModulator - QPSK modulation
- comm.AWGNChannel - AWGN channel
- comm.PSKDemodulator - QPSK demodulation (approx LLR)
- comm.ViterbiDecoder - Viterbi decoding

The code for the transceivers can be found in:

- viterbiTransceiverCPU.m
- viterbiTransceiverGPU.m

Each point along the bit error rate curve represents the result of many iterations of the transceiver code described above. To obtain accurate results in a reasonable amount of time, the simulation will gather at least 200 bit errors per signal-to-noise ratio (SNR) value, and at most 5000 packets of data. A packet represents 2000 message bits. The SNR ranges from 1 dB to 5 dB.

```
iterCntThreshold = 5000;
minErrThreshold = 200;
msgL = 2000;
snrdb = 1:5;
```

Initialization

Call the transceiver functions once to factor out setup time and object construction overhead. Objects are stored in persistent variables in each function.

```
errs  = zeros(length(snrdb),1);
iters = zeros(length(snrdb),1);

berplot = cell(1,5);
numframes = 500;    %GPU version runs 500 frames in parallel.
```

```
viterbiTransceiverCPU(-10,1,1);
viterbiTransceiverGPU(-10,1,1,numframes);

N=1; %N tracks which simulation variant is run
```

Workflow

The workflow for this example is:

1. Run a baseline simulation of System objects
2. Use MATLAB Coder to generate a MEX function for the simulation
3. Use parfor to run the bit error rate simulation in parallel
4. Combine the generated MEX function with parfor
5. Use the GPU-based System objects

```
fprintf(1,'Bit Error Rate Acceleration Analysis Example\n\n');
```

```
Bit Error Rate Acceleration Analysis Example
```

Baseline Simulation

To establish a reference point for various acceleration strategies, a bit error rate curve is generated using System objects alone. The code for the transceiver is in `viterbiTransceiverCPU.m`.

```
fprintf(1,'***Baseline - Standard System object simulation***\n');

% create random stream for each snrdb simulation
s = RandStream.create('mrg32k3a','NumStreams',1,...
    'CellOutput',true,'NormalTransform', 'Inversion');
RandStream.setGlobalStream(s{1});

ts = tic;
for ii=1:numel(snrdb)
    fprintf(1,'Iteration number %d, SNR (dB) = %d\n',ii, snrdb(ii));
    [errs(ii),iters(ii)] =viterbiTransceiverCPU(snrdb(ii), minErrThreshold, iterCntThreshold);
end
ber = errs./ (msgL* iters);
baseTime=toc(ts);
berplot{N} = ber;
desc{N} = 'baseline';
reportResultsCommSysGPU(N, baseTime,baseTime, 'Baseline');
```

```
***Baseline - Standard System object simulation***
Iteration number 1, SNR (dB) = 1
Iteration number 2, SNR (dB) = 2
Iteration number 3, SNR (dB) = 3
Iteration number 4, SNR (dB) = 4
Iteration number 5, SNR (dB) = 5
-------------------------------------------------------------------------------------------------
Versions of the Transceiver                          | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                          |           17.0205 |       1.0000
-------------------------------------------------------------------------------------------------
```

Code Generation

Using MATLAB Coder, a MEX file can be generated with optimized C code that matches the precompiled MATLAB code. Because the `viterbiTransceiverCPU` function conforms to the MATLAB code generation subset, it can be compiled into a MEX function without modification.

You must have a MATLAB Coder license to run this portion of the example.

```
fprintf(1,'\n***Baseline + codegen***\n');
N=N+1; %Increase simulation counter

% Create the coder object and turn off checks which will cause low
% performance.
fprintf(1,'Generating Code ...');
config_obj = coder.config('MEX');
config_obj.EnableDebugging = false;
config_obj.IntegrityChecks = false;
config_obj.ResponsivenessChecks = false;
config_obj.EchoExpressions = false;

% Generate a MEX file
codegen('viterbiTransceiverCPU.m', '-config', 'config_obj', '-args', {snrdb(1), minErrThreshold,
fprintf(1,'  Done.\n');

%Run once to eliminate startup overhead.
viterbiTransceiverCPU_mex(-10,1,1);

s = RandStream.getGlobalStream;
reset(s);

% Use the generated MEX function viterbiTransceiverCPU_mex in the
% simulation loop.
ts = tic;
for ii=1:numel(snrdb)
    fprintf(1,'Iteration number %d, SNR (dB) = %d\n',ii, snrdb(ii));
    [errs(ii),iters(ii)] = viterbiTransceiverCPU_mex(snrdb(ii), minErrThreshold, iterCntThreshol
end
ber = errs./ (msgL* iters);
trialtime=toc(ts);
berplot{N} = ber;
desc{N} = 'codegen';
reportResultsCommSysGPU(N, trialtime,baseTime, 'Baseline + codegen');
```

```
***Baseline + codegen***
Generating Code ...Code generation successful.

  Done.
Iteration number 1, SNR (dB) = 1
Iteration number 2, SNR (dB) = 2
Iteration number 3, SNR (dB) = 3
Iteration number 4, SNR (dB) = 4
Iteration number 5, SNR (dB) = 5
-------------------------------------------------------------------------------------------------
Versions of the Transceiver                          | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                          |           17.0205 |       1.0000
2. Baseline + codegen                                |           14.3820 |       1.1835
-------------------------------------------------------------------------------------------------
```

Parfor - Parallel Loop Execution

Using `parfor`, MATLAB executes the transceiver code against all SNR values in parallel. This requires opening the parallel pool and adding a `parfor` loop.

You must have a Parallel Computing Toolbox license to run this portion of the example.

```
fprintf(1,'\n***Baseline + parfor***\n');
fprintf(1,'Accessing multiple CPU cores ...\n');
if isempty(gcp('nocreate'))
    pool = parpool;
    poolWasOpen = false;
else
    pool = gcp;
    poolWasOpen = true;
end
nW=pool.NumWorkers;
N=N+1; %Increase simulation counter

snrN = numel(snrdb);

mT = minErrThreshold / nW;
iT = iterCntThreshold / nW;

errN =  zeros(nW, snrN);
itrN =  zeros(nW, snrN);

% replicate snrdb
snrdb_rep=repmat(snrdb,nW,1);

% create an independent stream for each worker
s = RandStream.create('mrg32k3a','NumStreams',nW,...
    'CellOutput',true,'NormalTransform', 'Inversion');

% pre-run
parfor jj=1:nW
    RandStream.setGlobalStream(s{jj});
    viterbiTransceiverCPU(-10, 1, 1);
end

fprintf(1,'Start parfor job ... ');
ts = tic;
parfor jj=1:nW
    for ii=1:snrN
        [err, itr] = viterbiTransceiverCPU(snrdb_rep(jj,ii), mT, iT);
        errN(jj,ii) = err;
        itrN(jj,ii) = itr;
    end
end
ber = sum(errN)./ (msgL*sum(itrN));
trialtime=toc(ts);
fprintf(1,'Done.\n');
berplot{N} = ber;
desc{N} = 'parfor';
reportResultsCommSysGPU(N, trialtime,baseTime, 'Baseline + parfor');


***Baseline + parfor***
```

```
Accessing multiple CPU cores ...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
Start parfor job ... Done.
-------------------------------------------------------------------------------------------
Versions of the Transceiver                        | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                        |           17.0205 |       1.0000
2. Baseline + codegen                              |           14.3820 |       1.1835
3. Baseline + parfor                               |            2.6984 |       6.3075
-------------------------------------------------------------------------------------------
```

Parfor and Code Generation

You can combine the last two techniques for additional acceleration. The compiled MEX function can be executed inside of a `parfor` loop.

You must have a MATLAB Coder license and a Parallel Computing Toolbox license to run this portion of the example.

```
fprintf(1,'\n***Baseline + codegen + parfor***\n');
N=N+1; %Increase simulation counter

% pre-run
parfor jj=1:nW
    RandStream.setGlobalStream(s{jj});
    viterbiTransceiverCPU_mex(1, 1, 1); % use the same mex file
end

fprintf(1,'Start parfor job ... ');
ts = tic;
parfor jj=1:nW
    for ii=1:snrN
        [err, itr] = viterbiTransceiverCPU_mex(snrdb_rep(jj,ii), mT, iT);
        errN(jj,ii) = err;
        itrN(jj,ii) = itr;
    end
end
ber = sum(errN)./ (msgL*sum(itrN));
trialtime=toc(ts);
fprintf(1,'Done.\n');
berplot{N} = ber;
desc{N} = 'codegen + parfor';
reportResultsCommSysGPU(N, trialtime,baseTime, 'Baseline + codegen + parfor');


***Baseline + codegen + parfor***
Start parfor job ... Done.
-------------------------------------------------------------------------------------------
Versions of the Transceiver                        | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                        |           17.0205 |       1.0000
2. Baseline + codegen                              |           14.3820 |       1.1835
3. Baseline + parfor                               |            2.6984 |       6.3075
4. Baseline + codegen + parfor                     |            2.7059 |       6.2902
-------------------------------------------------------------------------------------------
```

GPU

The System objects that the `viterbiTransceiverCPU` function uses are available for execution on the GPU. The GPU-based versions are:

- comm.gpu.ConvolutionalEncoder - convolutional encoding
- comm.gpu.PSKModulator - QPSK modulation
- comm.gpu.AWGNChannel - AWGN channel
- comm.gpu.PSKDemodulator - QPSK demodulation (approx LLR)
- comm.gpu.ViterbiDecoder - Viterbi decoding

A GPU is most effective when processing large quantities of data at once. The GPU-based System objects can processes multiple frames in a single call to the step method. The `numframes` variable represents the number of frames processed per call. This is analogous to `parfor` except that the parallelism is on a per object basis, rather than a per `viterbiTransceiverCPU` call basis.

You must have a Parallel Computing Toolbox license and a CUDA® 1.3 capable GPU to run this portion of the example.

```
fprintf(1,'\n***GPU***\n');
N=N+1; %Increase simulation counter

try
    dev = parallel.gpu.GPUDevice.current;
    fprintf(...
        'GPU detected (%s, %d multiprocessors, Compute Capability %s)\n',...
        dev.Name, dev.MultiprocessorCount, dev.ComputeCapability);

    sg = parallel.gpu.RandStream.create('mrg32k3a','NumStreams',1,'NormalTransform','Inversion');
    parallel.gpu.RandStream.setGlobalStream(sg);

    ts = tic;
    for ii=1:numel(snrdb)
        fprintf(1,'Iteration number %d, SNR (dB) = %d\n',ii, snrdb(ii));
        [errs(ii),iters(ii)] =viterbiTransceiverGPU(snrdb(ii), minErrThreshold, iterCntThreshold
    end
    ber = errs./ (msgL* iters);
    trialtime=toc(ts);
    berplot{N} = ber;
    desc{N} = 'GPU';
    reportResultsCommSysGPU(N, trialtime,baseTime, 'Baseline + GPU');

    fprintf(1,'  Done.\n');

catch %#ok<CTCH>

    % Report that the appropriate GPU was not found.
    fprintf(1, ['Could not find an appropriate GPU or could not ', ...
        'execute GPU code.\n']);

end
```

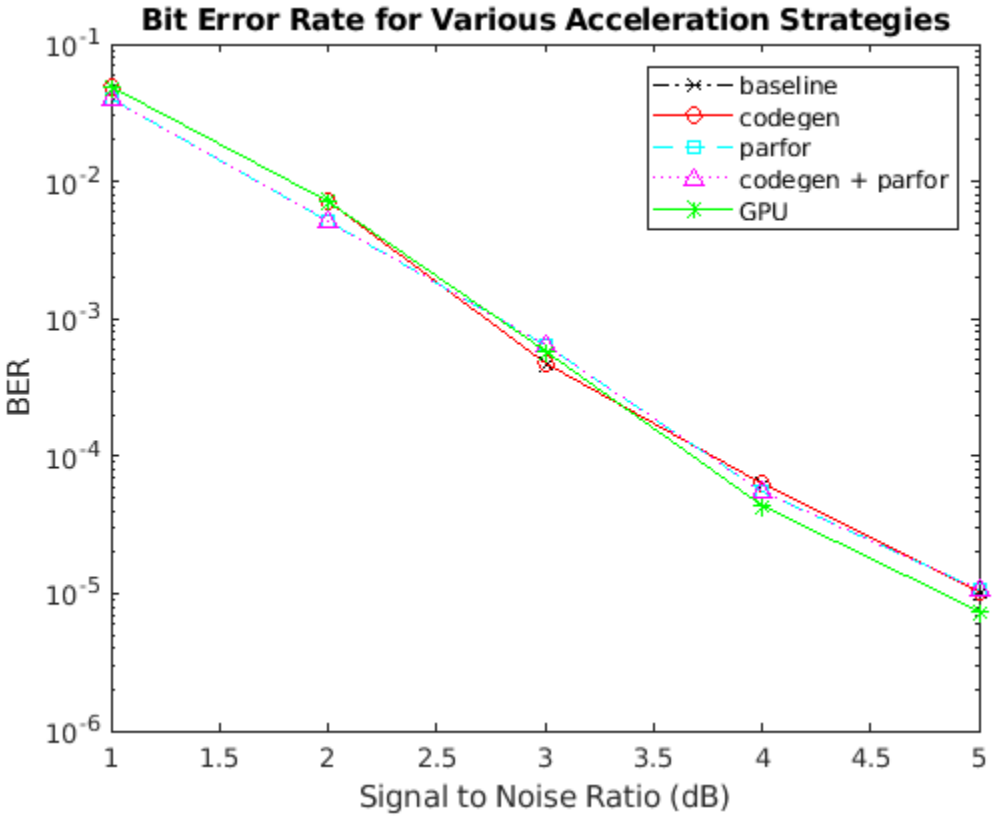
```
***GPU***
GPU detected (Tesla V100-PCIE-32GB, 80 multiprocessors, Compute Capability 7.0)
Iteration number 1, SNR (dB) = 1
Iteration number 2, SNR (dB) = 2
Iteration number 3, SNR (dB) = 3
Iteration number 4, SNR (dB) = 4
Iteration number 5, SNR (dB) = 5
------------------------------------------------------------------------------------------
```

```
Versions of the Transceiver                          | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                          |           17.0205 |       1.0000
2. Baseline + codegen                                |           14.3820 |       1.1835
3. Baseline + parfor                                 |            2.6984 |       6.3075
4. Baseline + codegen + parfor                       |            2.7059 |       6.2902
5. Baseline + GPU                                    |            0.1895 |      89.8137
-------------------------------------------------------------------------------------------
  Done.
```

Analysis

Comparing the results of these trials, it is clear that the GPU is significantly faster than any other simulation acceleration technique. This performance boost requires a very modest change to the simulation code. However, there is no loss in bit error rate performance as the following plot illustrates. The very slight differences in the curves are a result of different random number generation algorithms and/or effects of averaging different quantities of data for the same point on the curve.

```
lines = {'kx-.', 'ro-', 'cs--', 'm^:', 'g*-'};
for ii=1:numel(desc)
    semilogy(snrdb, berplot{ii}, lines{ii});
    hold on;
end
hold off;
title('Bit Error Rate for Various Acceleration Strategies');
xlabel('Signal to Noise Ratio (dB)');
ylabel('BER');
legend(desc{:});
```



Cleanup

Leave the parallel pool in the original state.

```
if ~poolWasOpen
    delete(gcp);
end
```

```
Parallel pool using the 'local' profile is shutting down.
```

Simulation Acceleration Using MATLAB Coder and Parallel Computing Toolbox

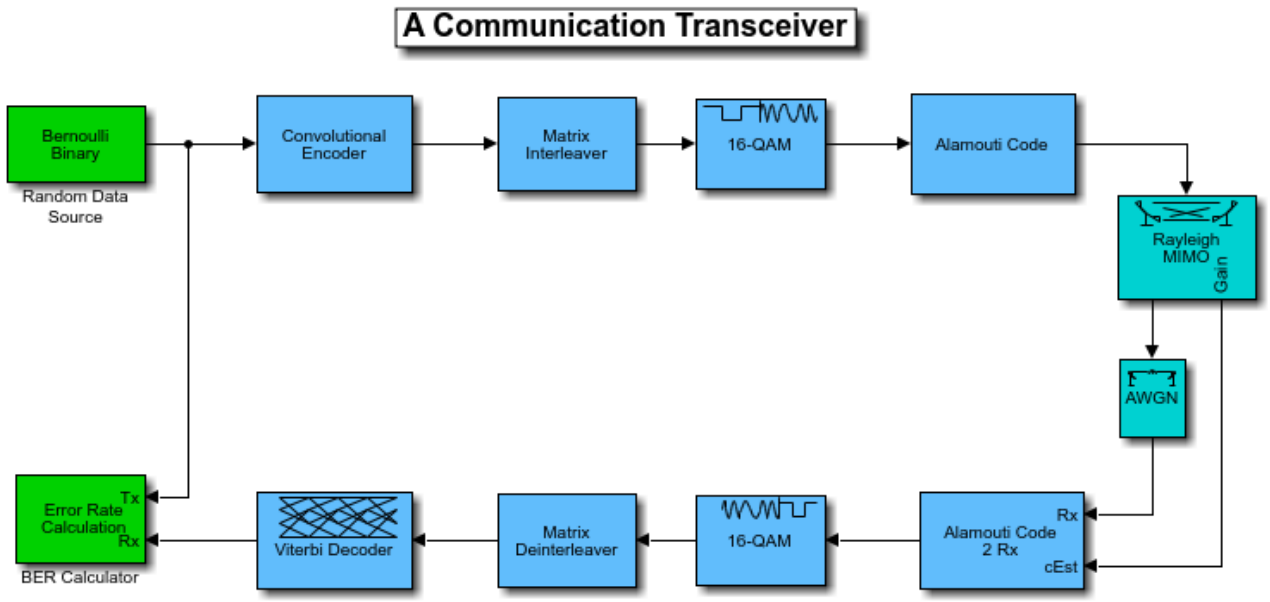
This example shows two ways to accelerate the simulation of communications algorithms in MATLAB®. It showcases the runtime performance effects of using MATLAB to C code generation and parallel processing runs (using the MATLAB `parfor` (Parallel Computing Toolbox) function). For a comprehensive look at all possible acceleration techniques, see Accelerating MATLAB Algorithms and Applications article.

The combined effect of using these methods may speed up a typical simulation time by an order of magnitude. The difference is tantamount to running the simulation overnight or within just a few hours.

To run the MATLAB to C code generation section of this example, you must have MATLAB Coder™ product. To run the parallel processing section of this example, you must have Parallel Computing Toolbox™ product.

Example Structure

This example examines various implementations of this transceiver system in MATLAB.



This system is composed of a transmitter, a channel model, and a receiver. The transmitter processes the input bit stream with a convolutional encoder, an interleaver, a modulator, and a MIMO space-time block encoder (see [1 on page 8-0], [2 on page 8-0]). The transmitted signal is then processed by a 2x2 MIMO block fading channel and an additive white gaussian noise (AWGN) channel. The receiver processes its input signal with a 2x2 MIMO space-time block decoder, a demodulator, a deinterleaver, and a Viterbi decoder to recover the best estimate of the input bit stream at the receiver.

The example follows this workflow:

1. Create a function that runs the simulation algorithms
2. Use the MATLAB Profiler GUI to identify speed bottlenecks
3. Accelerate the simulation with MATLAB to C code generation
4. Achieve even faster simulation using parallel processing runs

Create Function that Runs Simulation Algorithms

Start with a function that represents the first version or baseline implementation of this algorithm. The inputs to the `helperAccelBaseline` function are the E_b/N_o value of the current frame (`EbNo`), minimum number of errors (`minNumErr`) and the maximum number of bits processed (`maxNumBits`). E_b/N_o is the ratio of energy per bit to noise power spectral density. The function output is the bit error rate (BER) information for each E_b/N_o point.

```
type helperAccelBaseline
```

```
function ber = helperAccelBaseline(EbNo, minNumErr, maxNumBits)
%helperAccelBaseline Simulate a communications link
%   BER = helperAccelBaseline(EBNO,MINERR,MAXBIT) returns the bit error
%   rate (BER) of a communications link that includes convolutional coding,
%   interleaving, QAM modulation, an Alamouti space-time block code, and a
%   MIMO block fading channel with AWGN.  EBNO is the energy per bit to
%   noise power spectral density ratio (Eb/No) of the AWGN channel in dB,
%   MINERR is the minimum number of errors to collect, and MAXBIT is the
%   maximum number of simulated bits so that the simulations do not run
%   indefinitely if the Eb/No value is too high.

%   Copyright 2011-2021 The MathWorks, Inc.

M = 16;                                                 % Modulation Order
k = log2(M);                                            % Bits per Symbol
codeRate = 1/2;                                         % Coding Rate
adjSNR = EbNo - 10*log10(1/codeRate) + 10*log10(k);
trellis = poly2trellis(7,[171 133]);
tblen = 32;
dataFrameLen = 1998;

% Add 6 zeros to terminate the convolutional code
chanFrameLen=(dataFrameLen+6)/codeRate;
permvec=[1:3:chanFrameLen 2:3:chanFrameLen 3:3:chanFrameLen]';

ostbcEnc = comm.OSTBCEncoder(NumTransmitAntennas=2);
ostbcComb = comm.OSTBCCombiner(NumTransmitAntennas=2,NumReceiveAntennas=2);
mimoChan = comm.MIMOChannel(MaximumDopplerShift=0,PathGainsOutputPort=true);
berCalc = comm.ErrorRate;

% Run Simulation
ber = zeros(3,1);
while (ber(3) <= maxNumBits) && (ber(2) < minNumErr)
    data = [randi([0 1],dataFrameLen,1);false(6,1)];
    encOut = convenc(data,trellis);           % Convolutional Encoder
    intOut = intrlv(double(encOut),permvec'); % Interleaver
    modOut = qammod(intOut,M,...
      'InputType','bit');                     % QAM Modulator
    stbcOut = ostbcEnc(modOut);               % Alamouti Space-Time Block Encoder
    [chanOut, pathGains] = mimoChan(stbcOut); % 2x2 MIMO Channel
    chEst = squeeze(sum(pathGains,2));
```

```
    rcvd = awgn(chanOut,adjSNR,'measured');   % AWGN channel
    stbcDec = ostbcComb(rcvd,chEst);          % Alamouti Space-Time Block Decoder
    demodOut = qamdemod(stbcDec,M,...
      'OutputType','bit');                    % QAM Demodulator
    deintOut = deintrlv(demodOut,permvec');   % Deinterleaver
    decOut = vitdec(deintOut(:),trellis, ...  % Viterbi Decoder
        tblen,'term','hard');
    ber = berCalc(decOut(1:dataFrameLen),data(1:dataFrameLen));
end
```

As a starting point, measure the time it takes to run this baseline algorithm in MATLAB. Use the MATLAB timing functions (`tic` and `toc`) to record the elapsed runtime to complete processing of a for-loop that iterates over E_b/N_o values from 0 to 7 dB.

```
minEbNodB=0;
maxEbNodB=7;
EbNoVec = minEbNodB:maxEbNodB;
minNumErr=100;
maxNumBits=1e6;
N=1;
str='Baseline';
% Run the function once to load it into memory and remove overhead from
% runtime measurements
helperAccelBaseline(3,10,1e4);
berBaseline=zeros(size(minEbNodB:maxEbNodB));
disp('Processing the baseline algorithm.');
```

```
Processing the baseline algorithm.
```

```
tic;
for EbNoIdx=1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx);
  y=helperAccelBaseline(EbNo,minNumErr,maxNumBits);
  berBaseline(EbNoIdx)=y(1);
end
rtBaseline=toc;
```

The result shows the simulation time (in seconds) of the baseline algorithm. Use this timing measurement to compare with subsequent accelerated simulation runtimes.

```
helperAccelReportResults(N,rtBaseline,rtBaseline,str,str);
```

```
-------------------------------------------------------------------------------------------
Versions of the Transceiver                      | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                      |            4.4886 |       1.0000
-------------------------------------------------------------------------------------------
```

Identify Speed Bottlenecks by Using MATLAB Profiler App

Identify the processing bottlenecks and problem areas of the baseline algorithm by using the MATLAB Profiler. Obtain the profiler information by executing the following script:

```
profile on
y=helperAccelBaseline(6,100,1e6);
profile off
profile viewer
```

The Profiler report presents the execution time for each function call of the algorithm. You can sort the functions according to their self-time in a descending order. The first few functions that the

Profiler window depicts represent the speed bottleneck of the algorithm. In this case, the `vitdec` function is identified as the major speed bottleneck.

Accelerate Simulation with MATLAB to C Code Generation

MATLAB Coder generates portable and readable C code from algorithms that are part of the MATLAB code generation subset. You can create a MATLAB executable (MEX) of the `helperAccelBaseline`, function because it uses functions and System objects that support code generation. Use the `codegen` (MATLAB Coder) function to compile the `helperAccelBaseline` function into a MEX function. After successful code generation by codegen, you will see a MEX file in the workspace that appends '_mex' to the function, `helperAccelBaseline_mex`.

```
codegen('helperAccelBaseline.m','-args',{EbNo,minNumErr,maxNumBits})
```

```
Code generation successful.
```

Measure the simulation time for the MEX version of the algorithm. Record the elapsed time for running this function in the same for-loop as before.

```
N=N+1;
str='MATLAB to C code generation';
tag='Codegen';
helperAccelBaseline_mex(3,10,1e4);
berCodegen=zeros(size(berBaseline));
disp('Processing the MEX function of the algorithm.');
```

```
Processing the MEX function of the algorithm.
```

```
tic;
for EbNoIdx=1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx);
  y=helperAccelBaseline_mex(EbNo,minNumErr,maxNumBits);
  berCodegen(EbNoIdx)=y(1);
end
rt=toc;
```

The results here show the MEX version of this algorithm runs faster than the baseline versions of the algorithm. The amount of acceleration achieved depends on the nature of the algorithm. The best way to determine the acceleration is to generate a MEX-function using MATLAB Coder and test the speedup firsthand. If your algorithm contains single-precision data types, fixed-point data types, loops with states, or code that cannot be vectorized, you are likely to see speedups. On the other hand, if your algorithm contains MATLAB implicitly multithreaded computations such as `fft` and `svd`, functions that call IPP or BLAS libraries, functions optimized for execution in MATLAB on a PC such as FFTs, or algorithms where you can vectorize the code, speedups are less likely.

```
helperAccelReportResults(N,rtBaseline,rt,str,tag);
```

```
-------------------------------------------------------------------------------------------
Versions of the Transceiver                      | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                      |            4.4886 |       1.0000
2. MATLAB to C code generation                   |            1.6402 |       2.7367
-------------------------------------------------------------------------------------------
```

Achieve Even Faster Simulation Using Parallel Processing Runs

Utilize multiple cores to increase simulation acceleration by running tasks in parallel. Use parallel processing runs (`parfor` loops) in MATLAB to perform the work on the number of available workers.

Parallel Computing Toolbox enables you to run different iterations of the simulation in parallel. Use the `gcp` (Parallel Computing Toolbox) function to get the current parallel pool. If a pool is available but not open, the `gcp` opens the pool and reserves several MATLAB workers to execute iterations of a subsequent `parfor`-loop. In this example, six workers run locally on a MATLAB client machine.

```
pool = gcp
```

```
pool =

 ProcessPool with properties:

            Connected: true
           NumWorkers: 6
              Cluster: local
        AttachedFiles: {}
    AutoAddClientPath: true
          IdleTimeout: 30 minutes (5 minutes remaining)
          SpmdEnabled: true
```

Run Parallel Over Eb/No Values

Run E_b/N_o points in parallel using six workers using a `parfor`-loop rather than a for-loop as used in the previous cases. Measure the simulation time.

```
N=N+1;
str='Parallel runs with parfor over Eb/No';
tag='Parfor Eb/No';
helperAccelBaseline_mex(3,10,1e4);
berParfor1=zeros(size(berBaseline));
disp('Processing the MEX function of the algorithm within a parfor-loop.');
```

```
Processing the MEX function of the algorithm within a parfor-loop.
```

```
tic;
parfor EbNoIdx=1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx);
  y=helperAccelBaseline_mex(EbNo,minNumErr,maxNumBits);
  berParfor1(EbNoIdx)=y(1);
end
rt=toc;
```

The result adds the simulation time of the MEX version of the algorithm executing within a `parfor`-loop to the previous results. Note that by running the algorithm within a `parfor`-loop, the elapsed time to complete the simulation is shorter. The basic concept of a `parfor`-loop is the same as the standard MATLAB for-loop. The difference is that `parfor` divides the loop iterations into groups so that each worker executes some portion of the total number of iterations. Because several MATLAB workers can be computing concurrently on the same loop, a `parfor`-loop provides significantly better performance than a normal serial for-loop.

```
helperAccelReportResults(N,rtBaseline,rt,str,tag);
```

```
-------------------------------------------------------------------------------------------
Versions of the Transceiver                       | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                       |            4.4886 |       1.0000
2. MATLAB to C code generation                    |            1.6402 |       2.7367
3. Parallel runs with parfor over Eb/No           |            1.0943 |       4.1020
-------------------------------------------------------------------------------------------
```

Run Parallel Over Number of Bits

In the previous section, the total simulation time is mainly determined by the highest E_b/N_o point. You can further accelerate the simulations by dividing up the number of bits simulated for each E_b/N_o point over the workers. Run each E_b/N_o point in parallel using six workers using a `parfor`-loop. Measure the simulation time.

```
N=N+1;
str='Parallel runs with parfor over number of bits';
tag='Parfor # Bits';
helperAccelBaseline_mex(3,10,1e4);
berParfor2=zeros(size(berBaseline));
disp('Processing the MEX function of the second version of the algorithm within a parfor-loop.')
```

```
Processing the MEX function of the second version of the algorithm within a parfor-loop.
```

```
tic;
% Calculate number of bits to be simulated on each worker
minNumErrPerWorker = minNumErr / pool.NumWorkers;
maxNumBitsPerWorker = maxNumBits / pool.NumWorkers;
for EbNoIdx=1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx);
  numErr = zeros(pool.NumWorkers,1);
  parfor w=1:pool.NumWorkers
    y=helperAccelBaseline_mex(EbNo,minNumErrPerWorker,maxNumBitsPerWorker);
    numErr(w)=y(2);
    numBits(w)=y(3);
  end
  berParfor2(EbNoIdx)=sum(numErr)/sum(numBits);
end
rt=toc;
```

The result adds the simulation time of the MEX version of the algorithm executing within a `parfor`-loop where this time each worker simulates the same E_b/N_o point. Note that by running this version within a `parfor`-loop we get the fastest simulation performance. The difference is that `parfor` divides the number of bits that needs to be simulated over the workers. This approach reduces the simulation time of even the highest E_b/N_o value by evenly distributing load (specifically, the number of bits to simulate) over workers.

```
helperAccelReportResults(N,rtBaseline,rt,str,tag);
```

```
-------------------------------------------------------------------------------------------------
Versions of the Transceiver                          | Elapsed Time (sec)| Acceleration Ratio
1. Baseline                                          |            4.4886 |       1.0000
2. MATLAB to C code generation                       |            1.6402 |       2.7367
3. Parallel runs with parfor over Eb/No              |            1.0943 |       4.1020
4. Parallel runs with parfor over number of bits     |            0.6501 |       6.9043
-------------------------------------------------------------------------------------------------
```

Summary

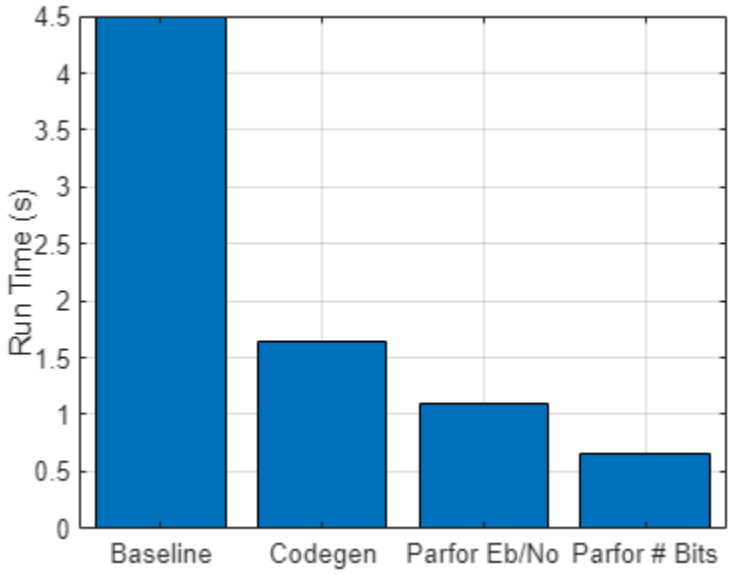
You can significantly speed up simulations of your communications algorithms with the combined effects of MATLAB to C code generation and Parallel processing runs.

- MATLAB to C code generation accelerates the simulation by locking-down datatypes and sizes of every variable and by reducing the overhead of the interpreted language that checks for the size and datatype of variables in every line of the code.

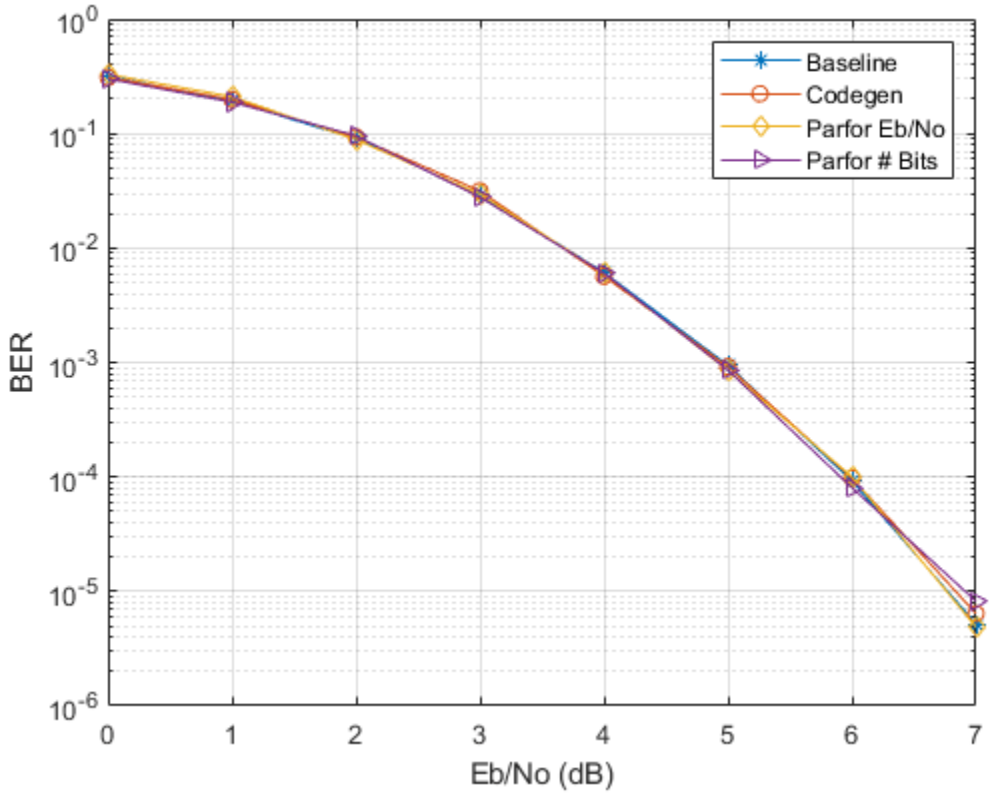
- Parallel processing runs can substantially accelerate simulation by computing different iterations of your algorithm concurrently across a number of MATLAB workers.
- Parallelizing each E_b/N_o point individually can accelerate further by speeding up even the longest running E_b/N_o point.

The following shows the run time of all four approaches as a bar graph. The results may vary based on the specific algorithm, available workers, and selection of minimum number of errors and maximum number of bits.

```
results = helperAccelReportResults;
```



This plot shows the BER curves for the different simulation processing approaches match each other closely. For each plotted E_b/N_0 each of the four versions of the algorithm ran with the maximum number of input bits set to ten million (`maxNumBits=1e7`) and the minimum number of bit errors set to five thousand (`minNumErr=5000`).



Further Exploration

This example uses the `gcp` function to reserve several MATLAB workers that run locally on your MATLAB client machine. By modifying the parallel configurations, you can accelerate the simulation even further by running the algorithm on a larger cluster of workers that are not on your MATLAB client machine. For a description of how to manage and use parallel configurations, see the “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox) topic.

The following functions are used in this example.

- helperAccelBaseline.m
- helperAccelReportResults.m

Selected References

1 S. M. Alamouti, "A simple transmit diversity technique for wireless communications," *IEEE® Journal on Selected Areas in Communications*, vol. 16, no. 8, pp. 1451-1458, Oct. 1998.

2 V. Tarokh, H. Jafarkhami, and A. R. Calderbank, "Space-time block codes from orthogonal designs," *IEEE Transactions on Information Theory*, vol. 45, no. 5, pp. 1456-1467, Jul. 1999.

Using GPUs to Accelerate Turbo Coding Bit Error Rate Simulations

This example shows how you can use GPUs to dramatically accelerate bit error rate simulations. Turbo Codes form the backbone of many modern communication systems. Because of the intense amount of computation involved in a Turbo Decoder and the massive amount of trials required for a valid bit error rate simulation, the Turbo Decoder is an ideal candidate for GPU acceleration. See the “Parallel Concatenated Convolutional Coding: Turbo Codes” on page 8-59 example, which explains the data processing chain, for more information on Turbo Codes.

You must have a Parallel Computing Toolbox™ license to use the Turbo Decoder GPU example.

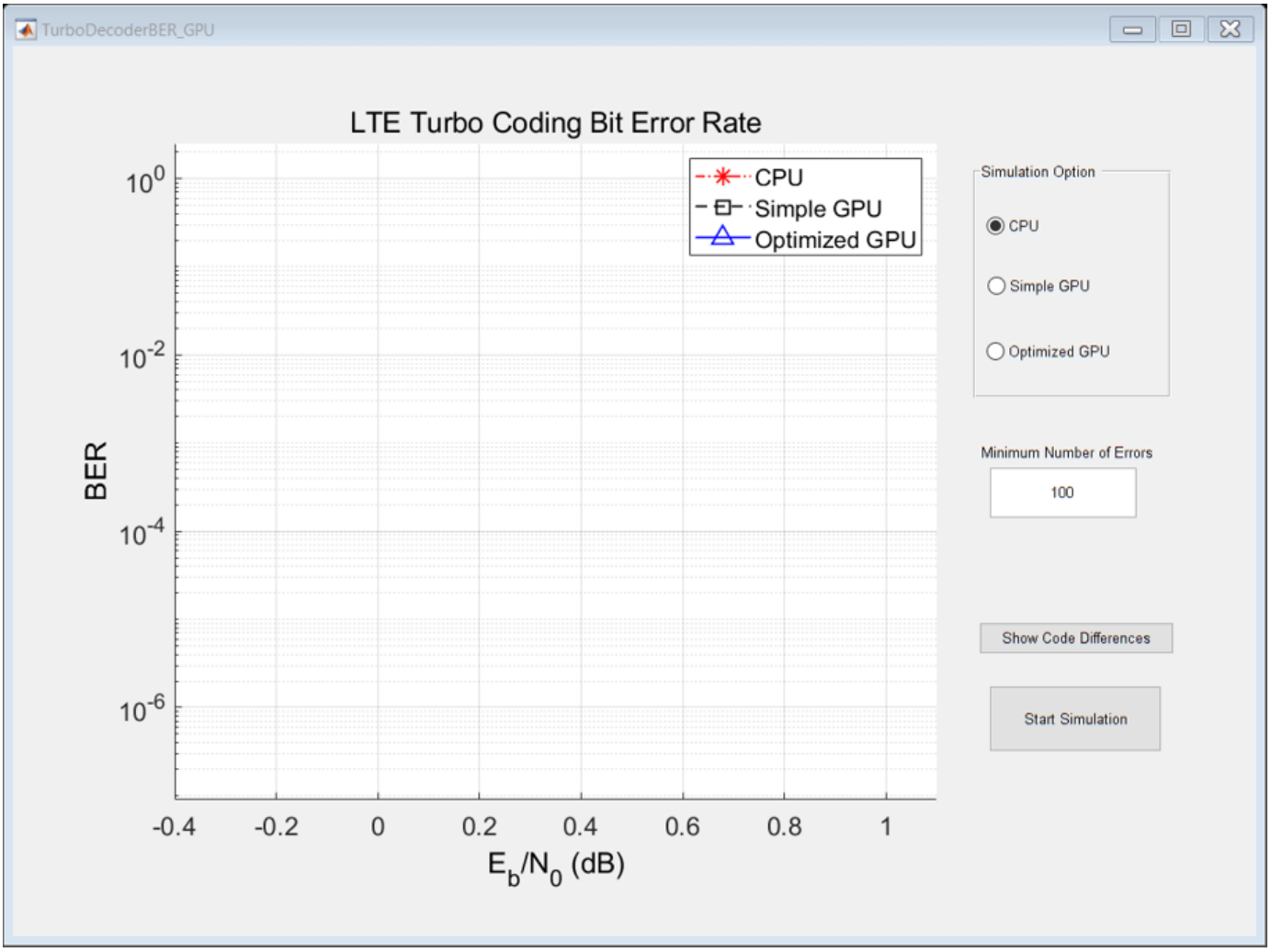
This example illustrates two approaches for GPU acceleration of the Turbo Coding bit error rate simulation. The baseline system consists of random message generation, a Turbo Encoder (comm.TurboEncoder), BPSK modulation using MATLAB® code, an AWGN channel (comm.AWGNChannel), BPSK demodulation using MATLAB code, a Turbo Decoder (comm.TurboDecoder), and finally bit error rate computation (comm.ErrorRate).

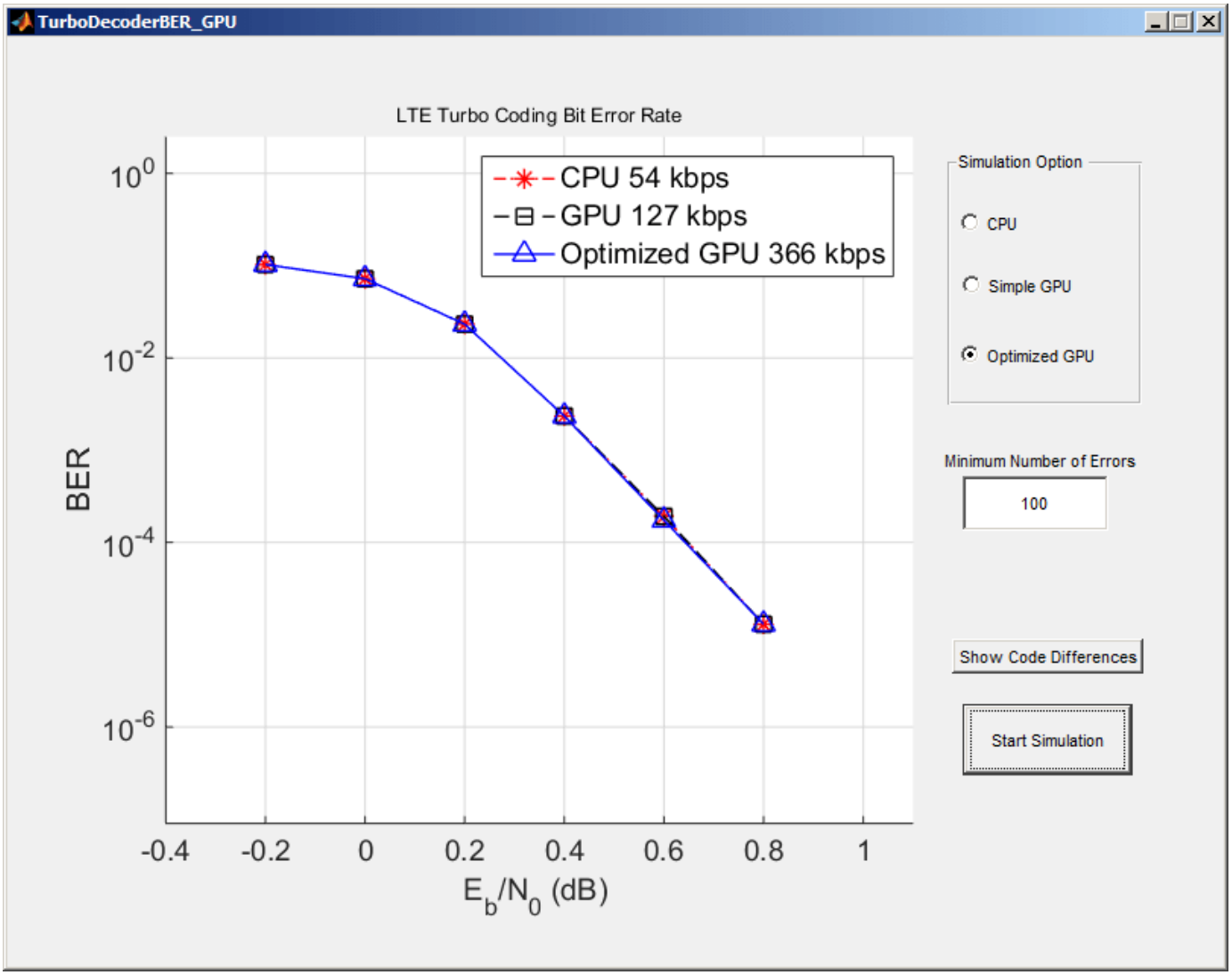
Notice: Supply of this software does not convey a license nor imply any right to use any Turbo codes patents owned by France Telecom, Telediffusion de France and/or Groupe des Ecoles des Telecommunications except in connection with use of the software for the purposes of design, simulation and analysis. Code generated from Turbo codes technology in this software is not intended and/or suitable for implementation or incorporation in any commercial products.

Please contact France Telecom for information about Turbo Codes Licensing program at the following address: France Telecom R&D - PIV/TurboCodes 38-40, rue du General Leclerc 92794 Issy-les-Moulineaux Cedex 9, France.

Launch the TurboDecoderBERsim GUI

```
TurboDecoderBER_GPU
```


TurboDecoderBER_GPU
LTE Turbo Coding Bit Error Rate
CPU
Simple GPU
Optimized GPU
BER
10^{0}
10^{-2}
10^{-4}
10^{-6}
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
E_b/N_0 (dB)
Simulation Option
CPU
Simple GPU
Optimized GPU
Minimum Number of Errors
100
Show Code Differences
Start Simulation



Overview of the Simulation

In the `Simulation options` button group select the `CPU` option for a CPU only simulation. The `Simple GPU` option makes very modest changes to the CPU version by replacing the CPU-based Turbo Decoder (comm.TurboDecoder) with the GPU implementation (comm.gpu.TurboDecoder).

The `Optimized GPU` option uses the comm.gpu.TurboDecoder object, and runs the BPSK modulation and demodulation code on the GPU, using gpuArray overloads. This option also uses the GPU-accelerated AWGN channel. As a GPU computing best practice, multiple frames of data are processed in each call to a System object®'s step method.

You should process multiple frames of data together (or in parallel) whenever possible on the GPU. In general, the GPU has far more compute power than necessary to process one frame of data. Giving the GPU multiple frames of data to process in one function call more efficiently utilizes the GPU's processing power. To use multiframe processing, a random message is created that is an integer multiple of the frame size in length. The Turbo Encoder encodes this long, multiframe vector one frame at a time. (There is no real advantage to multiframe processing on the CPU, and the CPU Turbo Encoder does not have a multiframe mode.) Data is then sent to the GPU using the gpuArray function.

The rest of the data processing chain is written as before because there is no notion of framing for the channel, modulator or demodulator. To have the Turbo Decoder run in multiframe mode, set the NumFrames property equal to the number of frames in the multiframe data vector (the default is one). The Turbo Decoder decodes each frame independently and in parallel in a single call to the step method (in particular, it does not treat the data as one long frame).

Code Differences

To see the changes in the original CPU source code necessary for the two GPU implementations, click on the appropriate GPU radio button (either `Simple GPU` or `Optimized GPU`) and then click the `Show Code Differences` button. This launches the comparison tool to view the changes necessary for GPU acceleration.

Error Rate Performance

You can plot the bit error rate curve for any of the three versions of the code. The number of errors required to plot a single point can be changed in the Minimum Number of Errors field. Enter the desired number of errors and click the `Start Simulation` button. Click the same button to stop the simulation early.

The bit error rate curves for the CPU and Simple GPU version match exactly. This indicates that the GPU version of the Turbo Decoder achieves exactly the same bit error rate as the CPU version at a much higher speed. In some cases, the Optimized GPU version may have a slightly different bit error rate because it runs multiple frames in parallel. Therefore, it may run a few frames more than necessary to pass the Minimum Number of Errors.

Results

As the simulation runs it displays number of message bits processed through the main simulation loop per second in the plot legend. This gives some measure of how quickly the simulation is running for each version of the code. Long simulations have been completed on a computer using an Intel® Xeon® X5650 processor and an NVIDIA® K20c GPU. Those simulations have shown that the `Simple GPU` is more than 2 times faster than the `CPU` version and that the `Optimized GPU` version is 6 times faster than the `CPU` version.

DVB-S.2 System Simulation Using a GPU-Based LDPC Decoder System Object

This example shows how to use a GPU-based LDPC Decoder System object™ to increase the speed of a communications system simulation. The performance improvement is illustrated by modeling part of the ETSI (European Telecommunications Standards Institute) EN 302 307 standard for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S.2) [1 on page 8-0]. For further information on using System objects to simulate the DVB-S.2 system see “DVB-S.2 Link, Including LDPC Coding in Simulink” on page 8-372. You must have a Parallel Computing Toolbox™ user license to use the GPU-based LDPC Decoder.

Introduction

The LDPC Decoding algorithm is computationally expensive and constitutes the vast majority of the time spent in a DVB-S.2 simulation. Using the comm.gpu.LDPCDecoder System object to execute the decoding algorithm on a GPU dramatically improves simulation run time. The example simulates the DVB-S.2 system, obtaining a benchmark for speed (run time), once with a CPU-based LDPC decoder function (`ldpcDecode`) and once with a GPU-based LDPC Decoder (`comm.gpu.LDPCDecoder`). The example captures the bit error rate for both versions, to show there is no loss in decoding performance using the GPU.

```
fprintf(...
    'DVB-S.2 Digital Video Broadcast Standard Bit Error Rate Simulation\n\n');
```

```
DVB-S.2 Digital Video Broadcast Standard Bit Error Rate Simulation
```

```
fprintf(...
    'Performance comparison of CPU- and GPU- accelerated decoders.\n');
```

```
Performance comparison of CPU- and GPU- accelerated decoders.
```

GPU Presence Detection

The example attempts to query the GPU to detect a Parallel Computing Toolbox user license and the presence of a supported GPU. If the GPU or the Parallel Computing Toolbox is unavailable, a CPU-only simulation can be performed.

```
try
    % Query the GPU
    dev = parallel.gpu.GPUDevice.current;

    % Print out information about the GPU that was found
    fprintf(...
        'GPU detected (%s, %d multiprocessors, Compute Capability %s)\n',...
        dev.Name,dev.MultiprocessorCount,dev.ComputeCapability);

    % Include a GPU-based simulation.
    doGPU = true;

catch % #ok<CTCH>

    % The GPU is not supported or not present, or the Parallel Computing
    %Toolbox was not present and licensed. Consider a CPU-only simulation.

    inp = input(['***NOTE: GPU not detected. ', ...
```

```
        'Continue with CPU-only simulation? [Y]/N '],'s');
    if strcmpi(inp, 'y') || isempty(inp)
        doGPU = false;
    else
        return;
    end
end
```

```
GPU detected (Tesla V100-PCIE-32GB, 80 multiprocessors, Compute Capability 7.0)
```

Initialization

The getParamsDVBS2Demo.m function generates a structure, dvb, which holds the configuration information for the DVB-S.2 system given the parameters below. Subsequently, the example includes creating and configuring System objects, based on the dvb structure.

The createSimObjDVBS2Demo.m script constructs most of the System objects used in DVB-S.2 and configures them based on the dvb structure.

Then an LDPC decoder configuration object and a GPU-based LDPC Decoder System object are created. The LDPC decoder configuration object is passed to the CPU-based `ldpcDecode` function which uses options equivalent to those used by the GPU-based LDPC Decoder System object.

```
% DVB-S.2 System Parameters
subsystemType = 'QPSK 1/2'; % Constellation and LDPC code rate
EsNodB = 0.75;              % Energy per symbol to noise PSD ratio in dB
numFrames = 10;             % Number of frames to simulate
maxNumLDPCIterations = 50;  % LDPC Decoder iterations

dvb = getParamsDVBS2Demo(subsystemType,EsNodB,maxNumLDPCIterations);

% Create and configure the BCH Encoder and Decoder, Modulator, Demodulator,
% AWGN Channel.

createSimObjDVBS2Demo;

% Construct an LDPC Encoder configuration object
encoderCfg = ldpcEncoderConfig(dvb.LDPCParityCheckMatrix);

% LDPC Decoder Configuration
ldpcPropertyValuePairs = { ...
    'MaximumIterationCount',dvb.LDPCNumIterations, ...
    'ParityCheckMatrix',dvb.LDPCParityCheckMatrix, ...
    'DecisionMethod','Hard Decision', ...
    'IterationTerminationCondition','Maximum iteration count', ...
    'OutputValue','Information part'};

% Construct an LDPC Decoder configuration object
decoderCfg = ldpcDecoderConfig(dvb.LDPCParityCheckMatrix);
if doGPU
    % Construct a GPU-based LDPC Decoder System object
    gpuLDPCDecoder = comm.gpu.LDPCDecoder(ldpcPropertyValuePairs{:});
end

% Create an ErrorRate object to analyze the differences in bit error rate
% between the CPU and GPU.

BER = comm.ErrorRate;
```

CPU and GPU Performance Comparison

This example simulates the DVB-S.2 system using the CPU-based LDPC Decoder function first, and then the GPU-based LDPC Decoder System object. The example obtains system benchmarks for each LDPC Decoder by passing several frames of data through the system and measuring the total system simulation time. The first frame of data incurs a large simulation initialization time, and so, it is excluded from the benchmark calculations. The per frame and average system simulation times are printed to the Command Window. The bit error rate (BER) of the system is also printed to the Command Window to illustrate that both CPU-based and GPU-based LDPC Decoders achieve the same BER.

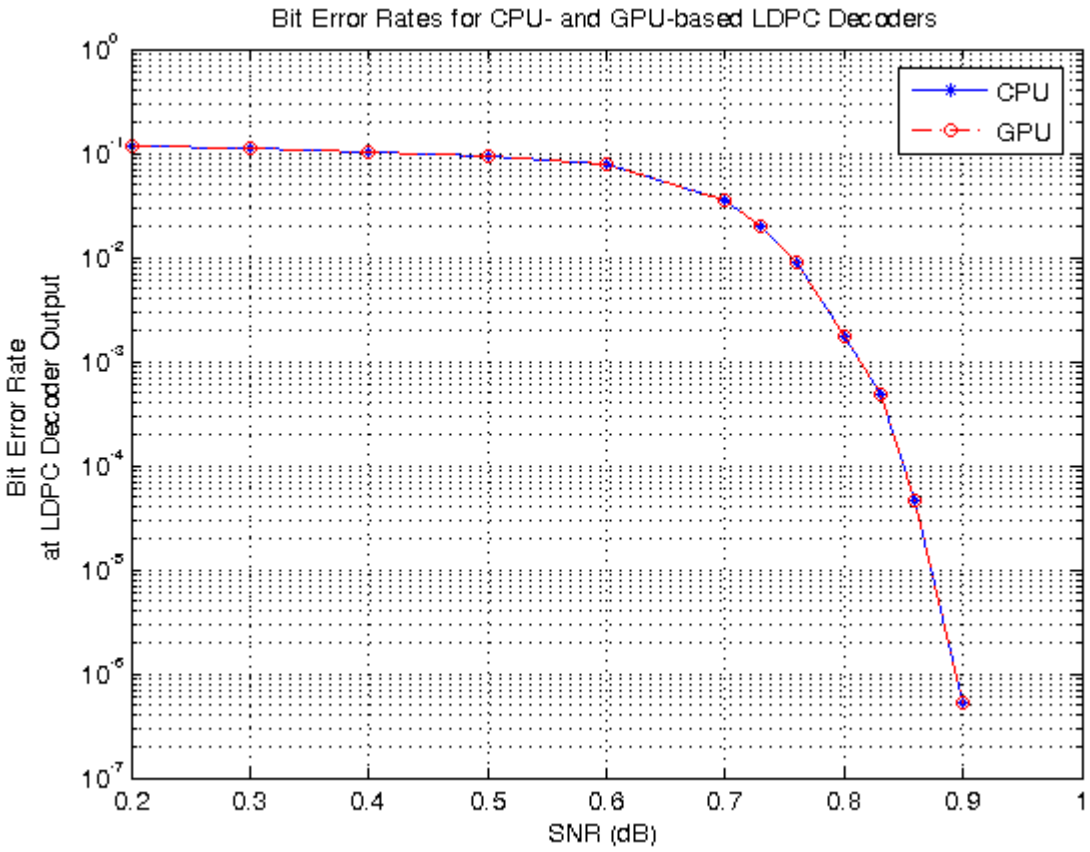
```
if doGPU
    architectures = 2;
else
    architectures = 1;
end

% Initialize run time results vectors
runtime = zeros(architectures,numFrames);
avgtime = zeros(1,architectures);

% Seed the random number generator used for the channel and message
% creation.  This will allow a fair BER comparison between CPU and GPU.
% Cache the original random stream to restore later.

original_rs = RandStream.getGlobalStream;
rs = RandStream.create('mrg32k3a','seed',25);
RandStream.setGlobalStream(rs);

% Loop for each processing unit - CPU and GPU
for ii = 1:architectures

    % Do some initial setup for the execution loop
    if (ii == 1)
        arch = 'CPU'; % Use CPU LDPC Decoder
    else
        arch = 'GPU';
        decoder = gpuLDPCDecoder;% Use GPU LDPC Decoder
    end

    % Reset the Error Rate object
    reset(BER);

    % Reset the random stream
    reset(rs);

    % Notice to the user that DVB-S.2 simulation is beginning.
    fprintf(['\nUsing ' arch '-based LDPC Decoder:\n']);
    dels = repmat('\b',1,fprintf('  Initializing ...'));


    % Main simulation loop. Run numFrames+1 times and ignore the first
    % frame (which has initialization overhead) for the run time
    % calculation. Use the first run for the BER calculation.
    for rr = 1:(numFrames+1)

        % Start timer
```

```
            ts = tic;

            % ***Create an input Message*** %
            msg = zeros(encbch.MessageLength, 1);
            msg(1:dvb.NumInfoBitsPerCodeword) = ...
                logical(randi([0 1],dvb.NumInfoBitsPerCodeword,1));

            % ***Transmit*** %
            bchencOut = encbch(msg);
            ldpcencOut = ldpcEncode(bchencOut,encoderCfg);
            xlvrOut = intrlv(ldpcencOut,dvb.InterleaveOrder);
            modOut = pskModulator(xlvrOut);

            % ***Corrupt with noise*** %
            chanOut = chan(modOut);

            % ***Receive*** %y
            demodOut = pskDemodulator(chanOut);
            dexlvrOut = deintrlv(demodOut,dvb.InterleaveOrder);

            % Use the appropriate LDPC Decoder.
            if strcmp(arch,'CPU')
                ldpcdecOut = logical(ldpcDecode(dexlvrOut,decoderCfg,dvb.LDPCNumIterations,'Decision
            else
                ldpcdecOut = decoder(dexlvrOut);
            end

            bchdecOut = decbch(ldpcdecOut);

            % ***Compute BER *** % Calculate BER at output of LDPC, not BCH.
            ber = BER(logical(bchencOut),ldpcdecOut);

            % Stop timer
            runtime(ii, rr) = toc(ts);

            % Don't report the first frame with the initialization overhead.
            if (rr > 1)
                fprintf(dels);
                newCharsToDelete = fprintf('  Frame %d decode : %.2f sec', ...
                    rr-1, runtime(ii,rr));
                dels = repmat('\b',1,newCharsToDelete);
            end
        end % end of running a frame through the DVB-S.2 system.


        % Report the run time results to the Command Window.
        fprintf(dels); % Delete the last line printed out.

        % Calculate the average run time. Don't include frame 1 because it
        % includes some System object initialization time.
        avgtime(ii) = mean(runtime(ii,2:end));

        fprintf('  %d frames decoded, %.2f sec/frame\n',numFrames,avgtime(ii));
        fprintf('  Bit error rate: %g \n',ber(1) );

    end % architecture loop

    Using CPU-based LDPC Decoder:
```

```
  Initializing ...
  Frame 1 decode : 0.29 sec  Frame 2 decode : 0.30 sec  Frame 3 decode : 0.32 sec  Frame 4 decode
  10 frames decoded, 0.28 sec/frame
  Bit error rate: 0.00785634
Using GPU-based LDPC Decoder:
  Initializing ...
  Frame 1 decode : 0.12 sec  Frame 2 decode : 0.12 sec  Frame 3 decode : 0.12 sec  Frame 4 decode
  10 frames decoded, 0.11 sec/frame
  Bit error rate: 0.00785634
```

```
% Reset the random stream to the cached object
RandStream.setGlobalStream(original_rs);
```

Using code similar to what is shown above, a bit error rate measurement was made offline. The bit error rate performance of the GPU- and CPU-based LDPC Decoders are identical as seen in this plot.



Summary

If a GPU was used, show the speedup based on the average run time of a DVB-S.2 system using a GPU LDPC Decoder vs a CPU LDPC Decoder.

```
if ~doGPU
    fprintf('\n*** GPU not present ***\n\n');
else
    %Calculate system-wide speedup
    fprintf(['\nFull system simulation runs %.2f times faster using ' ...
        'the GPU-based LDPC Decoder.\n\n'],avgtime(1) / avgtime(2));
end
```

```
Full system simulation runs 2.60 times faster using the GPU-based LDPC Decoder.
```

Appendix

This example uses the createSimObjDVBS2Demo.m script and getParamsDVBS2Demo.m helper function.

Selected Bibliography

1 ETSI Standard EN 302 307 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, New Gathering and other broadband satellite applications (DVB-S.2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

HDL Code Generation for Viterbi Decoder

This example shows HDL code generation support for the Viterbi Decoder block. It shows how to check, generate, and verify the HDL code you generate from a fixed-point Viterbi Decoder model. This example also discusses the settings you can use to alter the HDL code you generate.

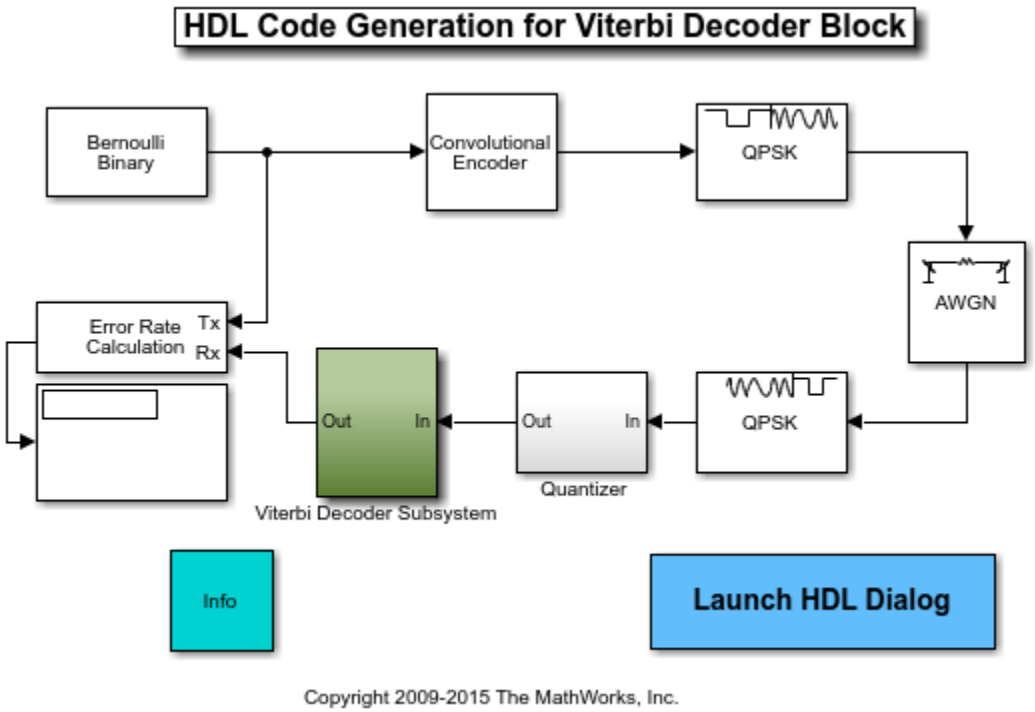
In order to run this example, you must have an HDL Coder™ license.

Introduction

The model shows HDL code generation for a fixed-point Viterbi Decoder block used in soft decision convolutional decoding. To learn more about HDL support for Viterbi Decoder, refer to the "HDL Code Generation" section of the block page in documentation.

To open the model, run the following commands:

```
modelname = 'commviterbihdl';
open_system(modelname);
```



In this model, the top-level subsystem *Viterbi Decoder Subsystem* contains the Viterbi Decoder block. To open this subsystem, run the following commands:

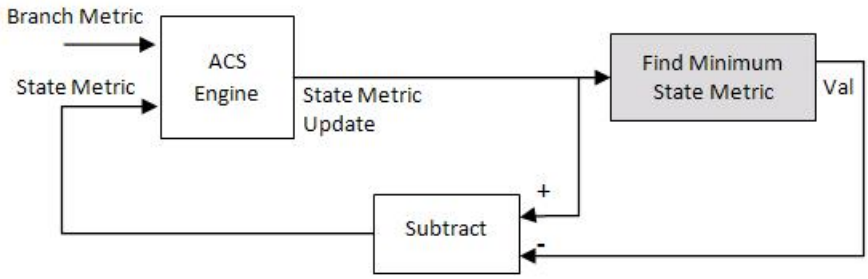
```
systemname = [modelname '/Viterbi Decoder Subsystem'];
open_system(systemname);
```

The Viterbi Decoding Algorithm

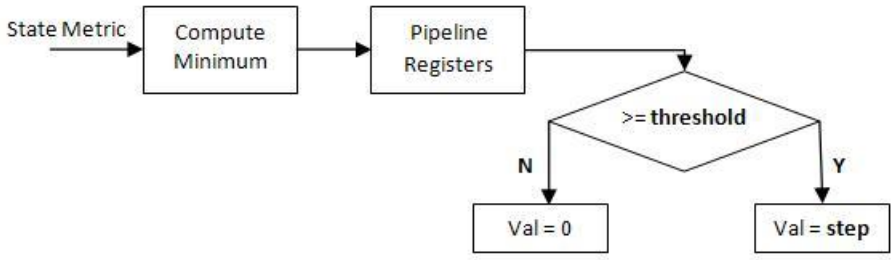
There are three main components to the Viterbi decoding algorithm. They are the branch metric computation (BMC), add-compare-select (ACS), and traceback decoding. The following diagram illustrates the three units in the Viterbi decoding algorithm:

The Renormalization Method

The Viterbi Decoder prevents the overflow of the state metrics in the ACS component by subtracting the minimum value of the state metrics at each time step, as shown in the following figure:



Obtaining the minimum value of all the state metric elements in one clock cycle results in a poor clock frequency for the circuit. The performance of the circuit may be improved by adding pipeline registers. However, simply subtracting the minimum value delayed by pipeline registers from the state metrics may still lead to overflow. The hardware architecture modifies the renormalization method and avoids the state metric overflow in three steps. First, the architecture calculates values for the threshold and step parameters, based on the trellis structure and the number of soft decision bits. Second, the delayed minimum value is compared to the threshold. Last, if the minimum value is greater than or equal to the threshold value, the implementation subtracts the step value from the state metric; otherwise no adjustment is performed. The following figure illustrates the modified renormalization method:



Optimal State Metric Word Length Calculation

The hardware implementation calculates the optimal word length of the state metric and compares it with the value you specify for the block. The hardware architecture uses the optimal value if it is smaller than the one you specify. A message is displayed to show the value during HDL code

generation. If the calculated value is larger than the value you specify, an error message is reported and the optimal value is displayed.

Applying the calculated optimal state metric word length in the hardware implementation may significantly reduce the hardware resource if the value you specify is too large. For example, if you set 16 bits as the state metric word length but only 9 bits are required to achieve the same numerical results, applying the calculated optimal state metric word length in the hardware architecture saves approximately 40 percent of the register resources. The calculated optimal state metric word length for some typical trellises is displayed in the following table:

Decoding rate	Free distance	Example Trellis	Number of soft decision bits	Optimal word length
1/2	10	7, [171 133]	3	8
1/2	10	7, [171 133]	1	5
1/3	15	7, [171 133 165]	3	9
1/3	15	7, [171 133 165]	1	6
1/4	24	9,[463 535 733 745]	3	9
1/4	24	9,[463 535 733 745]	1	7

Check and Generate Code for a Fixed-point Viterbi Model

This model decodes a DVB rate 1/2 , constraint length 7,(171,133) convolutional code with 3 bits soft decision. The decoder runs at continuous mode with the traceback depth of 32. The state metric word length is set to 16 bits. To validate the parameter settings of the Viterbi Decoder block, you can run the following commands:

- workingdir = tempname;
- checkhdl(systemname,'TargetDirectory',workingdir);

Running checkhdl generates messages that report:

- the default value of **TracebackStagesPerPipeline**. More information on this parameter can be found in the section **Pipelining the register-based traceback unit**,
- the state metric word length used in the HDL code compared with the one set on the block mask,
- the total delay introduced by the pipeline registers with respect to the original Viterbi block.

To generate HDL for the subsystem containing the Viterbi Decoder block, run the following commands: workingdir = tempname; makehdl(systemname,'TargetDirectory',workingdir);

The top level VHDL® file name matches the name of the block in the model. The Viterbi_Decoder component generated in the *Viterbi_Decoder.vhd* contains three components: BranchMetric, ACS, and Traceback. The ACS and Traceback components instantiate components ACSUnit and TracebackUnit multiple times respectively. Data type definitions are included in the package file *Viterbi_Decoder_Subsystem_pkg.vhd*.

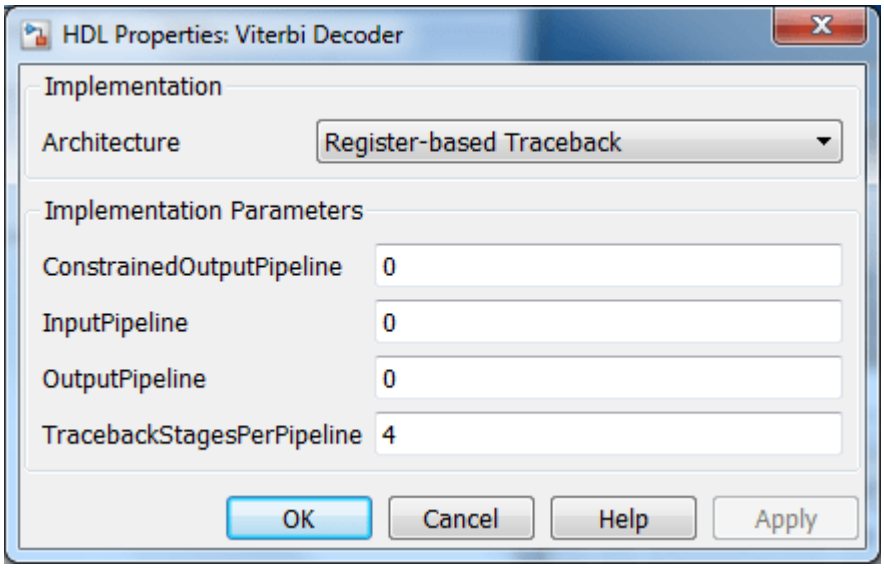
To generate a testbench for the subsystem containing the Viterbi Decoder block, run the following command: makehdltb(systemname,'TargetDirectory',workingdir);

Optimization of The Traceback Unit

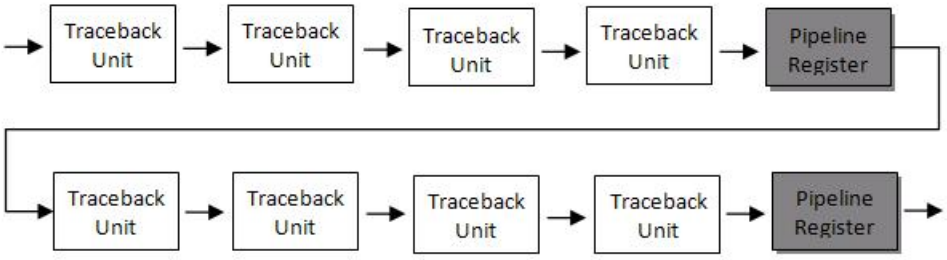
They are two methods to optimize the traceback unit: pipelining the register-based traceback or using the RAM-based traceback architecture.

- **Pipelining the register-based traceback unit**

The Viterbi Decoder block decodes every bit by tracing back through a traceback depth you define for the block. Because the block implements a complete traceback for each decision bit, registers are used to store the minimum state index and branch decision in the Traceback Decoding unit. This unit may be pipelined in order to improve the performance of the generated circuit. Pipeline registers can be added to the traceback unit by specifying the number of traceback stages per pipeline register. This can be done by setting the **TracebackStagesPerPipeline** implementation parameter for the Viterbi Decoder in the HDL block properties dialog. Right click the Viterbi Decoder block to navigate to the **HDL Block Properties** menu.



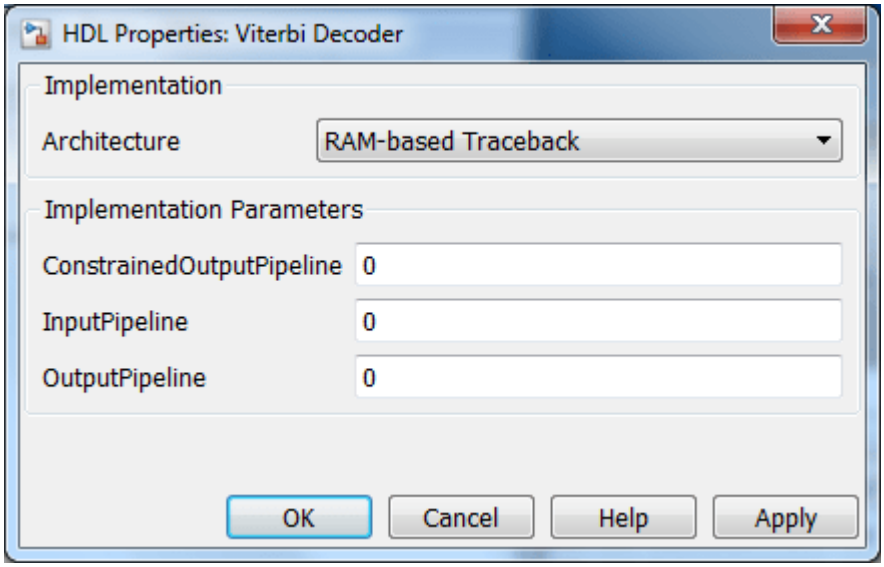
Setting the property value to 4 results in the insertion of a pipeline register for every four traceback units in the model, as illustrated in the following figure:



The TracebackStagesPerPipeline implementation parameter provides you a way of balancing 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 number results in a lower usage of registers along with a decrease in the circuit speed. In our experiment with the rate 1/2 , constraint length 7,(171,133) convolutional code, adjusting the TracebackStagesPerPipeline parameter from 4 to 8 reduces the pipeline register usage in half, with the circuit speed changing from 173MHz to 94 MHz.

- **RAM-based traceback**

Instead of using registers, you can choose to use RAMs to save the survivor branch information. This can be done by setting the HDL Architecture property of the Viterbi Decoder block to RAM-based Traceback.



There are two major differences between the register-based and the RAM-based traceback architectures.

Firstly, the register-based implementation combines the traceback and decode operations into one step and uses the best state found from the minimum operation as the decoding initial state. The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data.

Secondly, the register-based implementation decodes one bit after a complete trackback; while the 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.

Due to 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 of constraint length, is recommended in the RAM-based traceback to achieve 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. The following table summarizes the RAM usage for some typical trellis structures.

Constraint length	Example trellis	Traceback depth	Memory size(bits)	Block RAMs
3	3,[5 7]	30	3x30x4	1
4	4,[15 17])	40	3x40x8	1
5	5,[37 27 33 25 35]	50	3x50x16	1
6	6,[73 75 55 65 47 57]	60	3x60x32	1
7	7,[171 133]	70	3x70x64	2
8	(8,[225 331 367])	80	3x80x128	4
9	9,[463 535 733 745]	90	3x90x256	8

Our experiment with the rate 1/2, constraint length 7, (171, 133) convolutional code shows that the RAM-based traceback unit uses 90% fewer registers than the register-based traceback unit (with pipelining every 4 stages)) using similar clock constraints in synthesis. The two implementations provide a register-RAM tradeoff that can be tailored to the individual design.

Selected References

1. Clark, G. C. Jr. and J. Bibb Cain., *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.
2. G. Feygin and P. G. Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," *IEEE Transactions on Communications*, vol. 41, no. 3, pp. 425-429, March 1993.

Using HDL Optimized CRC Library Blocks

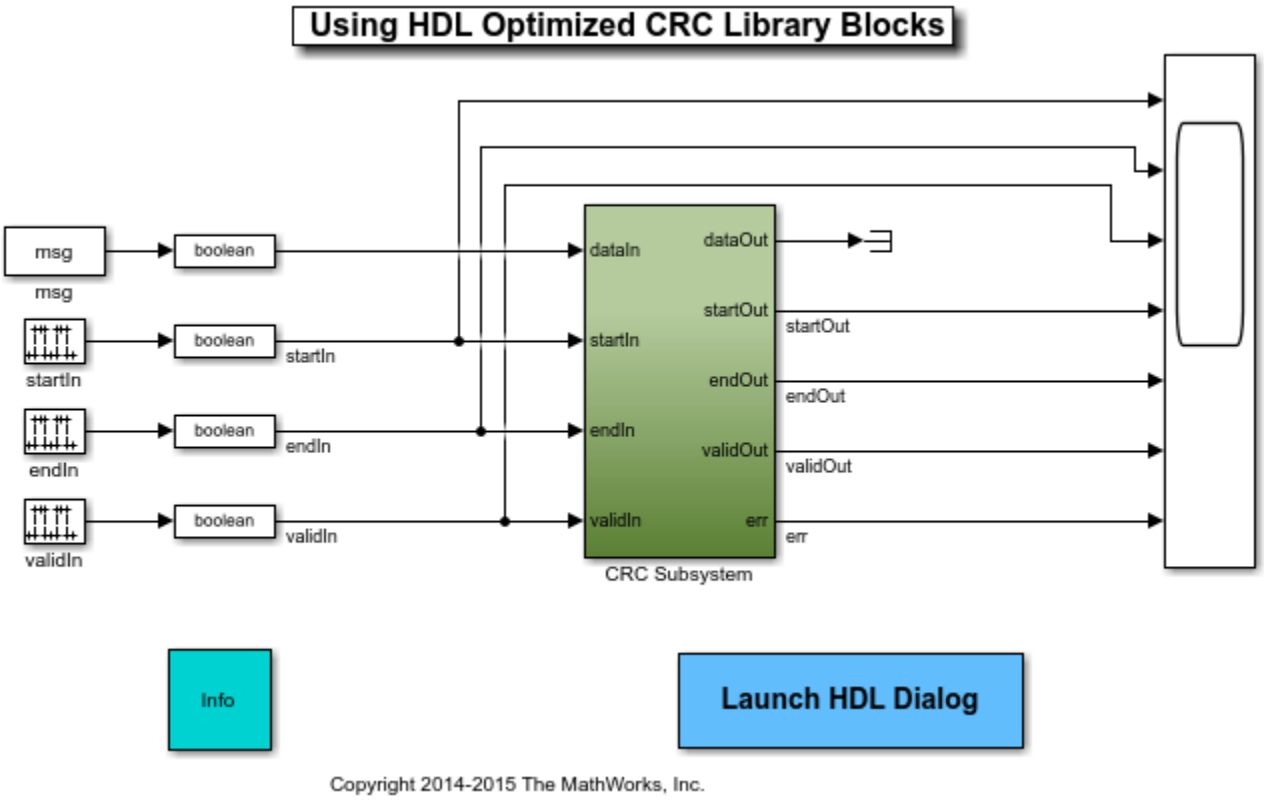
This example shows how to use the HDL Optimized CRC Generator and CRC Detector library blocks and then configure these blocks to meet the IEEE® 802.11 standard [1].

Introduction

The model shows how to use HDL Optimized CRC Generator and Detector library blocks for simulation and HDL Code generation. The 802.11 standard is used as the application. To learn more about HDL support for HDL Optimized CRC blocks, refer to the documentation. To learn more about the algorithm used in the blocks, refer to the paper in [2].

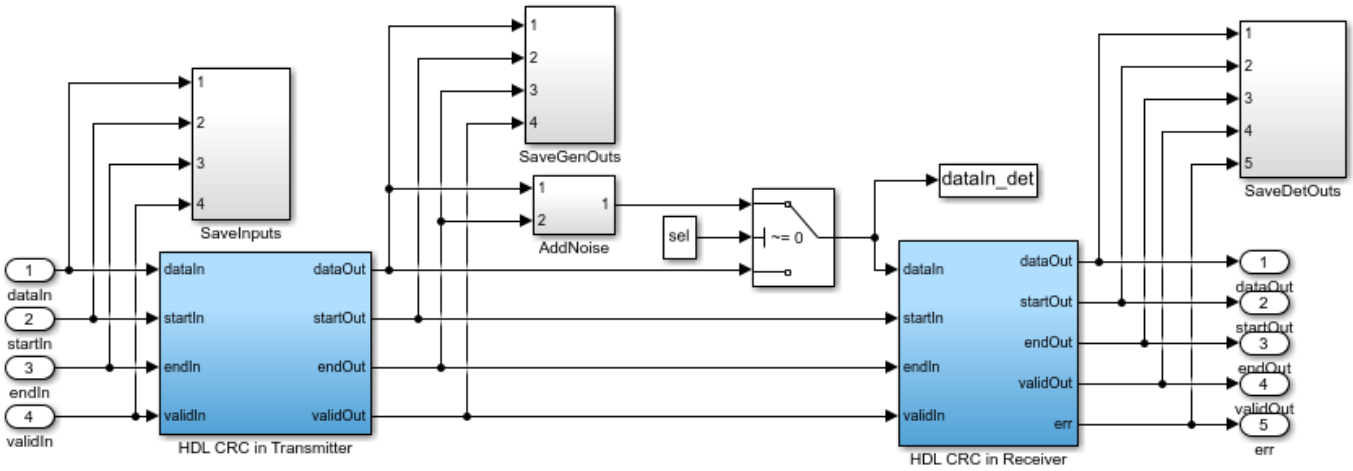
To open this example model, run the following commands:

```
modelname = 'commcrchdl';
open_system(modelname);
```



In this model, the top-level subsystem **CRC Subsystem** contains the HDL Optimized CRC Generator and Detector blocks. This subsystem also has an **AddNoise** subsystem that you can choose to add noise to the generated CRC checksum. To open this subsystem, run the following commands:

```
systemname = [modelname '/CRC Subsystem'];
open_system(systemname);
```



Parameter Settings

- **Polynomial**

CRC-CCITT is used in the IEEE® 802.11 standard to protect the SIGNAL, SERVICE and LENGTH fields. The row vector [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1] represent the polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

- **Initial State**

The HDL optimized CRC Generator block in the demo uses the Direct Method, i.e. it feeds the message into the most significant bit (MSB) of the checksum shift register and processes the message without padding zeros. The details of the CRC implementation can be found in this model. The diagram of the IEEE 802.11 CRC implementation is illustrated in Figure 15-2 of the 802.11 standard. The initial state is set as 1.

- **Final XOR Value**

The final XOR value is set as *0xFFFF* to implement the ones complement of the CRC Checksum.

Input Signals

The test vector in this model uses the example DBPSK signal specified in the 802.11 standard. The test data padded with CRC_length zeros is processed at 16 bits/sample in streaming mode. Parameter *dataIn_width*, which is the port width of the CRC Generator input port dataIn, defines the data processing speed. *mlen* defines the period in the controls signals startIn, endIn, and validIn. *dlen* defines the pulse width of the validIn signal. The input signals are configured in the **InitFcn** callback function in the **Model Properties** dialog box.

```
%DBPSK data
data = [0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0];
crc_len = 16;
% pad crc_len zero
msg = [data zeros(1,crc_len)];
```

```
dataIn_width = 16;
mlen = length(msg)/dataIn_width;
dlen = length(data)/dataIn_width;
```

You can alter the dataIn_width to 8,4,2,1 bit(s) in this example to meet your design requirements.For example, if you are processing data with length 56, aside from padding 8 bit zeros and using *dataIn_with* 16, you can choose *dataIn_width* to be 8 to ensure mlen and dlen are all integer numbers.

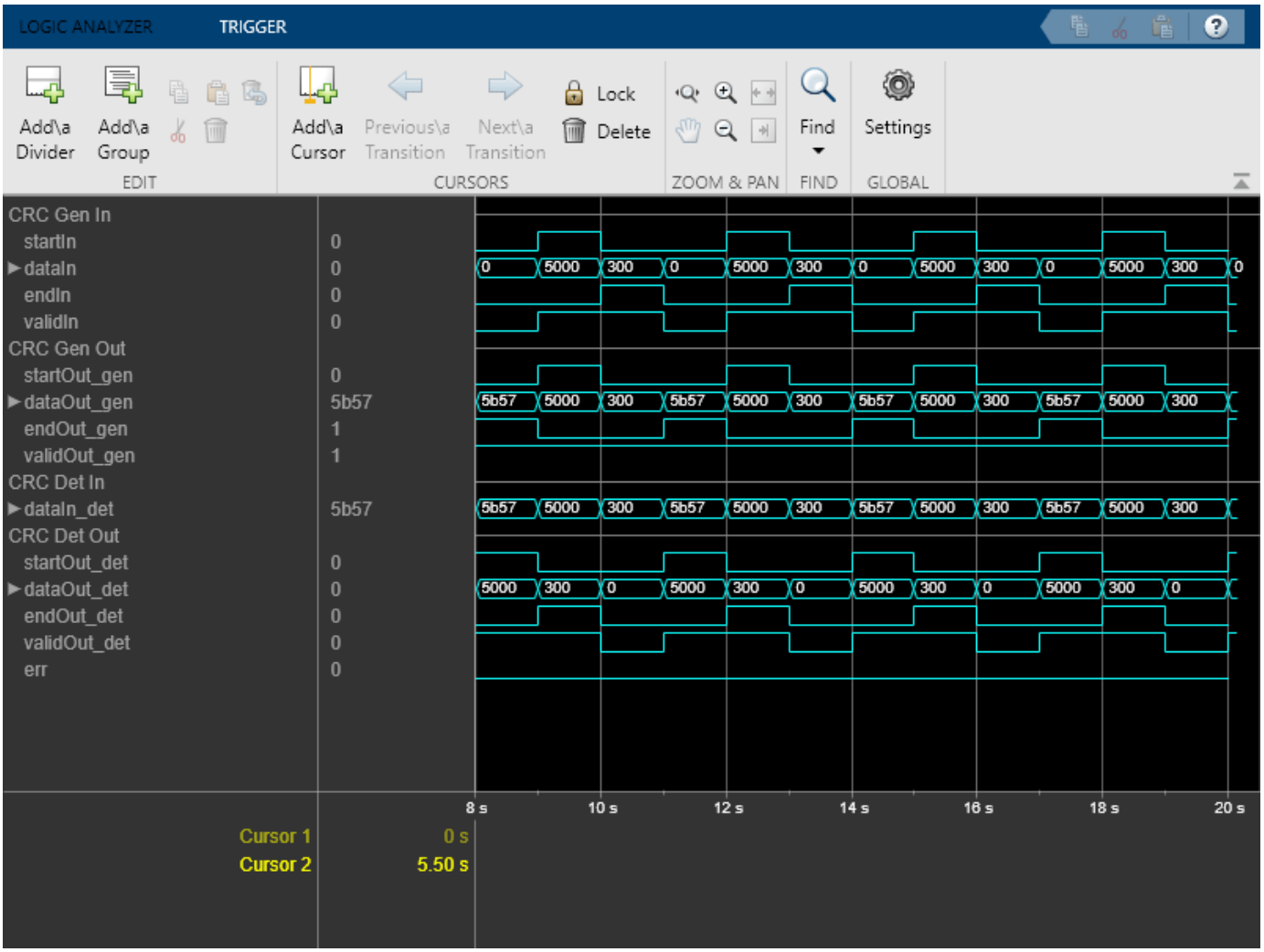
Output Signals

Run the model using the following command:

```
sim(modelname);
```

Several of the key signals have been logged into the workspace. These signals can be viewed in a Logic Analyzer window. The function commcrchdl_plot shows how to set the Logic Analyzer display. For further information on the Logic Analyzer System object™, refer to the `dsp.LogicAnalyzer`.

```
h = commcrchdl_plot(dataIn,startIn,endIn,validIn,...
                    dataOut_gen,startOut_gen,endOut_gen,validOut_gen,...
                    dataIn_det,dataOut_det,startOut_det,endOut_det,validOut_det,err);
```



dataIn, *startIn*, *endIn*, and *validIn* are input data and control signals to the HDL CRC generator. *dataOut_gen* (output of CRC generator) displays the message with the checksum appended every *dataIn_width* bits per sample. You can read the checksum when *endOut_gen* is high in the output waveform. The value **0x5B57** matches the CRC-16 FCS specified in 802.11 standard Section 15.2.3.6. *dataIn_det* shows the message with the corrupted checksum. *dataOut_det* displays the message output of the CRC detector. Error is detected when the *err* signal is high. *err* is valid when the *endOut_det* is active.

Initial delays are introduced at the output of the CRC generator and detector. You can calculate the initial delays using the following command:

```
initial_delay_gen =  crc_len/dataIn_width + 2;
initial_delay_det =  4*crc_len/dataIn_width + 4;
```

Check and Generate HDL Code

To check and generate HDL code of this example, you must have an HDL Coder™ license.

You can use the commands *makehdl(subsystemname)* and *makehdltb(subsystemname)* to generate the HDL code and testbench for the subsystems **HDL CRC in Transmitter** and **HDL CRC in Receiver**.

Specify the subsystemname as 'commcrchdl/CRC Subsystem/HDL CRC in Transmitter' or 'commcrchdl/CRC Subsystem/HDL CRC in Receiver'.

Selected References

1 IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. (2007 revision). IEEE-SA. 12 June 2007.

1 Giuseppe Campobello, Giuseppe Patane, Marco Russo. "Parallel CRC Realization," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1312-1319, October, 2003.

Using HDL Optimized RS Encoder/Decoder Library Blocks

This example shows how to implement encoder and decoder for the IEEE® 802.16 standard [1] using the HDL Optimized Reed-Solomon (RS) Encoder and Decoder library blocks.

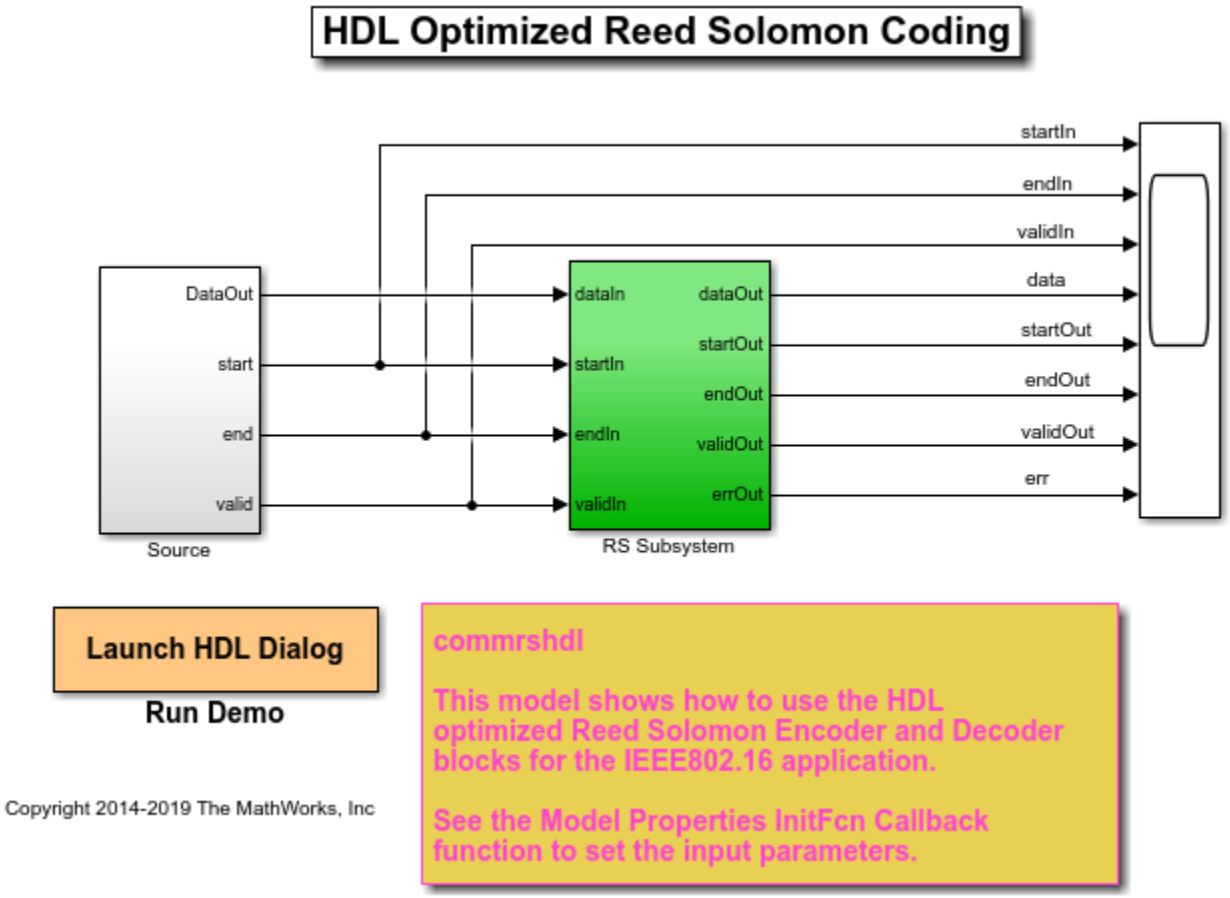
Introduction

The RS code is a nonbinary block code. A RS code that maps K information symbols into a codeword of symbol length N is denoted as RS(N, K) 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 IEEE 802.16 Broadband Wireless Access standard [1] employs a “Shortening, Puncturing, and Erasures” on page 16-25 of the RS(255,239) code generated on GF(256), i.e., $N = 255$, $K = 239$, and $M = 8$. RS encoder introduces $N - K = 16$ parity symbols, which are used by the RS decoder to detect and correct symbol errors. The code can correct up to $T = \text{floor}[(N - K)/2] = 8$ symbol errors in each codeword.

This model shows how to use HDL Optimized RS Encoder and Decoder library blocks for simulation and HDL Code generation. It implements the encoding and error correction for the IEEE 802.16 standard. For details about HDL support for HDL Optimized RS Encoder and Decoder blocks, refer to Integer-Input RS Encoder HDL Optimized or Integer-Output RS Decoder HDL Optimized. To learn more about the algorithm used in the blocks refer to [2].

To open this example model, run the following commands:

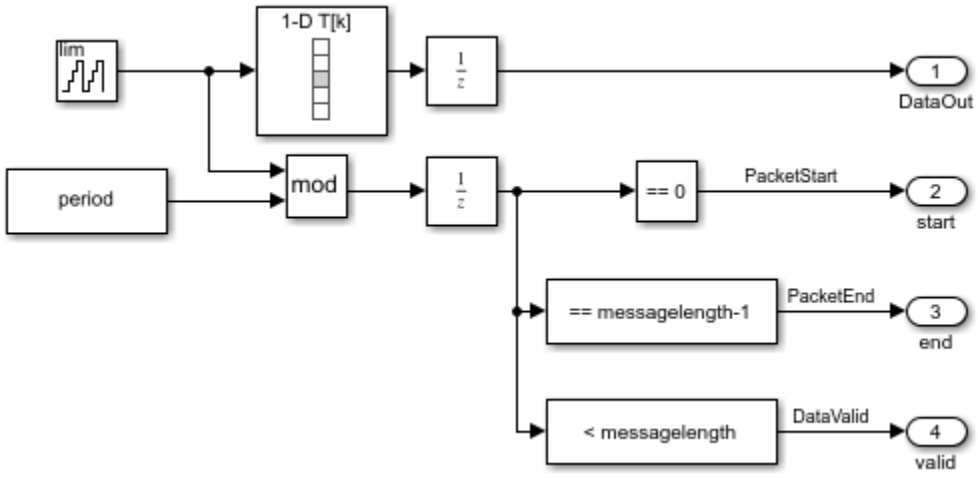
```
modelname = 'commrshdl';
open_system(modelname);
```



Source

The **Source** subsystem generates the information symbols for the RS Encoder. To open the **Source** subsystem, run the following commands:

```
systemname = [modelname '/Source'];
open_system(systemname);
```



One of the messages (information symbols) employed by the IEEE 802.16 standard contains the following 36 bytes (Randomized data specified on page 827 of [1]).

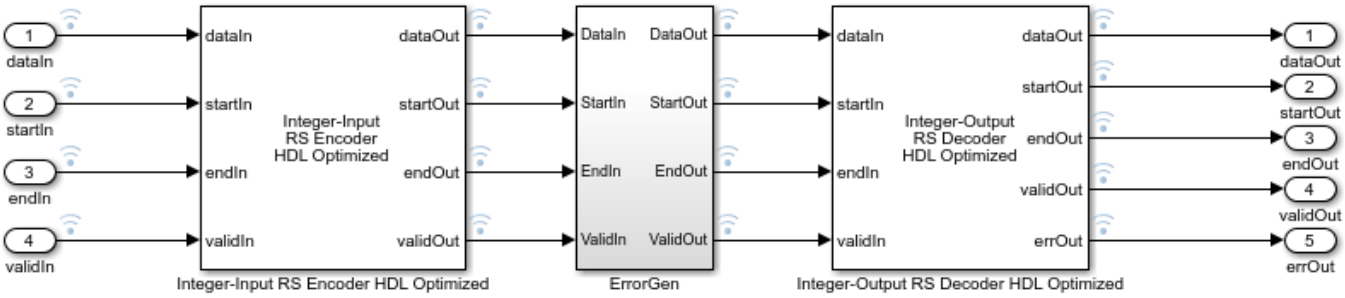
```
message = [D4 BA A1 12 F2 74 96 30 27 D4 88 9C 96 E3 A9 52 B3 15 AB FD
92 53 07 32 C0 62 48 F0 19 22 E0 91 62 1A C1 00].
```

The **Source** repeatedly transmits the message followed by a guard interval. The model has parameters **messagelength**, for the number of symbols in the message to encode; and **period**, which includes the **messagelength** and the length of the guard interval. The guard interval between messages accommodates the latency of the encoder adding parity check symbols to the message, and the decoder performing a Chien search. In the initFcn callback of the model, the **messagelength** is set to 36 and **period** is set to 236 (which suggest that the guard interval has a length of 200 symbols).

Note that the values of **messagelength** and **period** can be varied as desired.

The top-level **RS Subsystem** contains the HDL Optimized RS Encoder and Decoder blocks. To open the **RS subsystem**, run the following commands:

```
systemname = [modelname '/RS Subsystem'];
open_system(systemname);
```



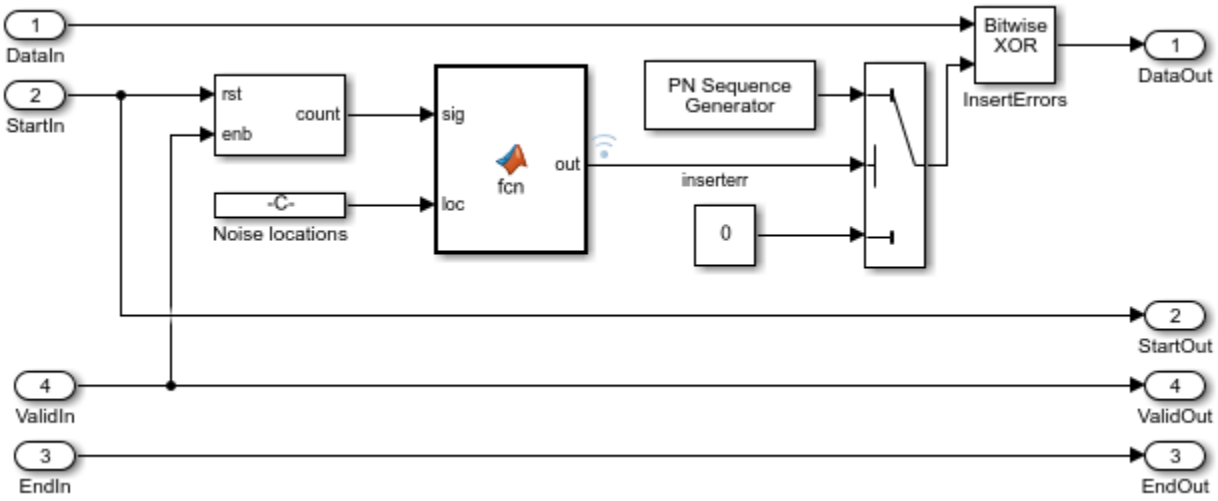
The values of N and K are set in the InitFcn callback of the model and are used to configure the HDL Optimized RS Encoder and Decoder blocks. The values of N and K cannot be changed in this model.

The RS encoder infers a shortened code if the message length is less than K symbols. In this case, it will pad the input message with $239 - 36 = 203$ zeros, encodes the padded message, and appends 16 parity check symbols. The block then removes the added zeros symbols, creating a $36 + 16 = 52$ symbol output.

The field generator polynomial employed by IEEE 802.16 standard is $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. Accordingly, for both RS encoder and decoder, the **Source of primitive polynomial** is set as **Property**, the **Primitive polynomial** is set as [1 0 0 0 1 1 1 0 1], the **Source of B which is the starting power for roots of the primitive polynomial** is set as **Property**, and the **B value** is set as **0**. The code generator polynomial used by IEEE 802.16 standard is $g(x) = (x + \lambda^0)(x + \lambda^1)(x + \lambda^2)\ldots(x + \lambda^{2T-1})$, where $\lambda = 02_{\text{hex}}$.

Restrictions on M and the codeword length N are detailed on the Integer-Input RS Encoder block reference page. The **ErrorGen** subsystem adds noise to the RS encoded message. To open the **ErrorGen** subsystem, run the following commands:

```
systemname = [modelname '/RS Subsystem/ErrorGen'];
open_system(systemname);
```



The **ErrorGen** subsystem implements the logic to add noise to the codewords at locations specified in the **Noise Locations** constant. The location can be changed as desired. In this example, the noise will be added to the 5th, 23rd, 34th, and 12th codewords, corresponding to the symbols F2, 07, 1A, and 9C. The MATLAB® function block outputs logical true only at these four time instances for each packet, and activates a bitwise XOR operation between the original symbols and the noise.

Output Signals

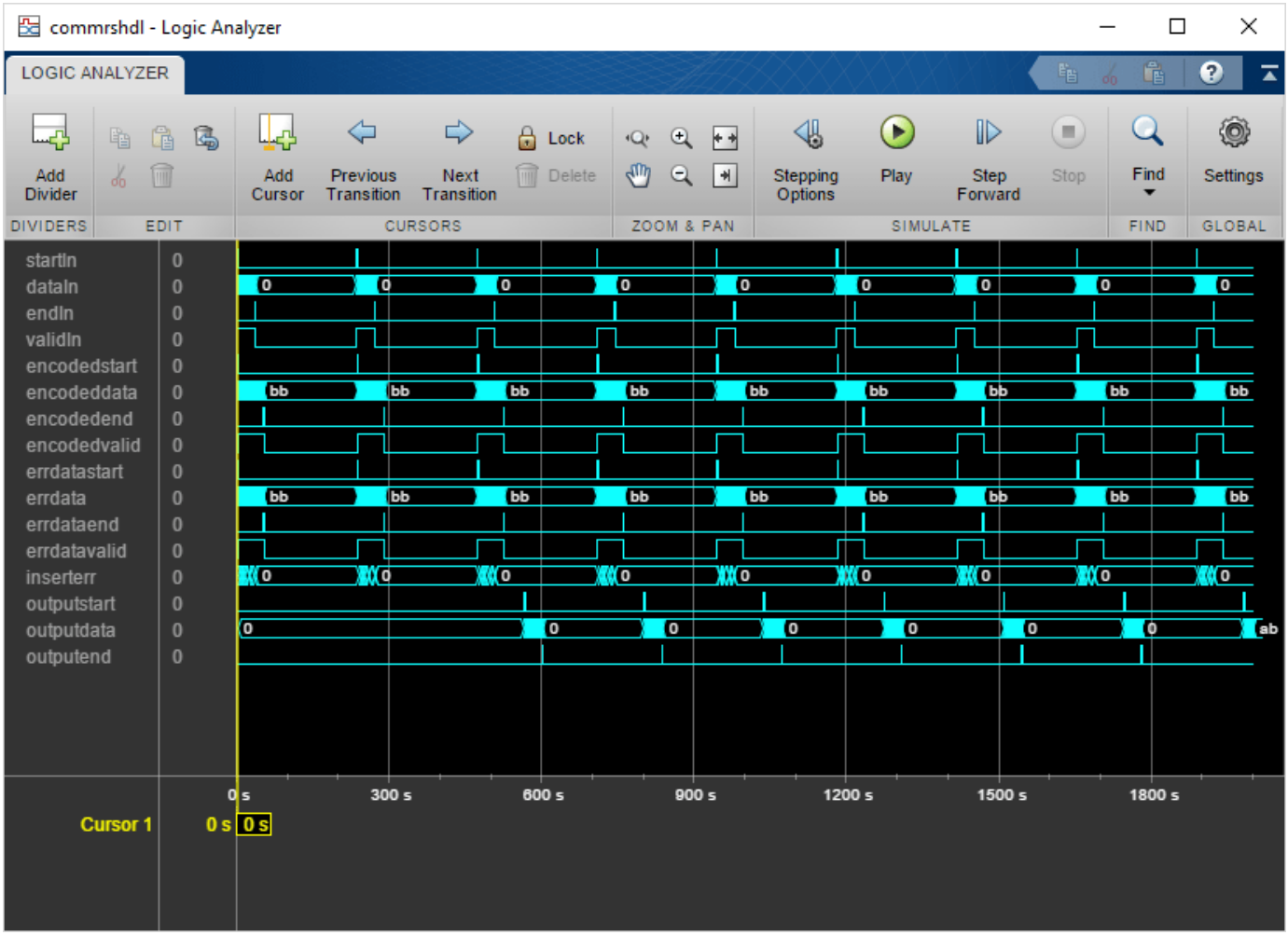
Run the model using the following command:

```
sim(modelname);
```

Viewing the Signals

The Logic Analyzer can be used to view multiple signals in one window and viewing signals this way makes it easier to observe transitions. Signals in this model at various stages, namely, before

encoding, after encoding, after adding noise, and after decoding are streamed. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolstrip.



Analysis of Results

In the Logic Analyzer output the inputdata signal represents the input of the RS encoder block and this is the 36 byte message given in the IEEE 802.16 specification. The encoded data shows the output of the RS encoder block. Note that the IEEE 802.16 specification performs puncturing of the parity bytes and retains only the first four bytes of the 16 bytes. In this demo all 16 bytes of parity are used and the first four bytes of parity are 49, 31, 40, and BF, matching the IEEE 802.16 specification.

The errdata signal represents the encoded data with noise added in the specified noise locations. These noise locations are marked with 1s in the inserterr signal.

The decoded and corrected message out of the RS decoder block is shown by the outputdata signal. Note that the RS decoder block introduces about 3 period lengths of latency. Observe outputdata to see that the errors induced by noise are corrected.

Generate HDL Code and Test Bench

To check and generate HDL code for this example, you must have an HDL Coder™ license.

Get a unique temporary directory name for the generated files,

```
workingdir = tempname;
```

To check whether there are any issues with the model for HDL code generation, you can run the following command:

```
checkhdl('commrshdl/RS Subsystem','TargetDirectory',workingdir);
```

Enter the following command to generate HDL code:

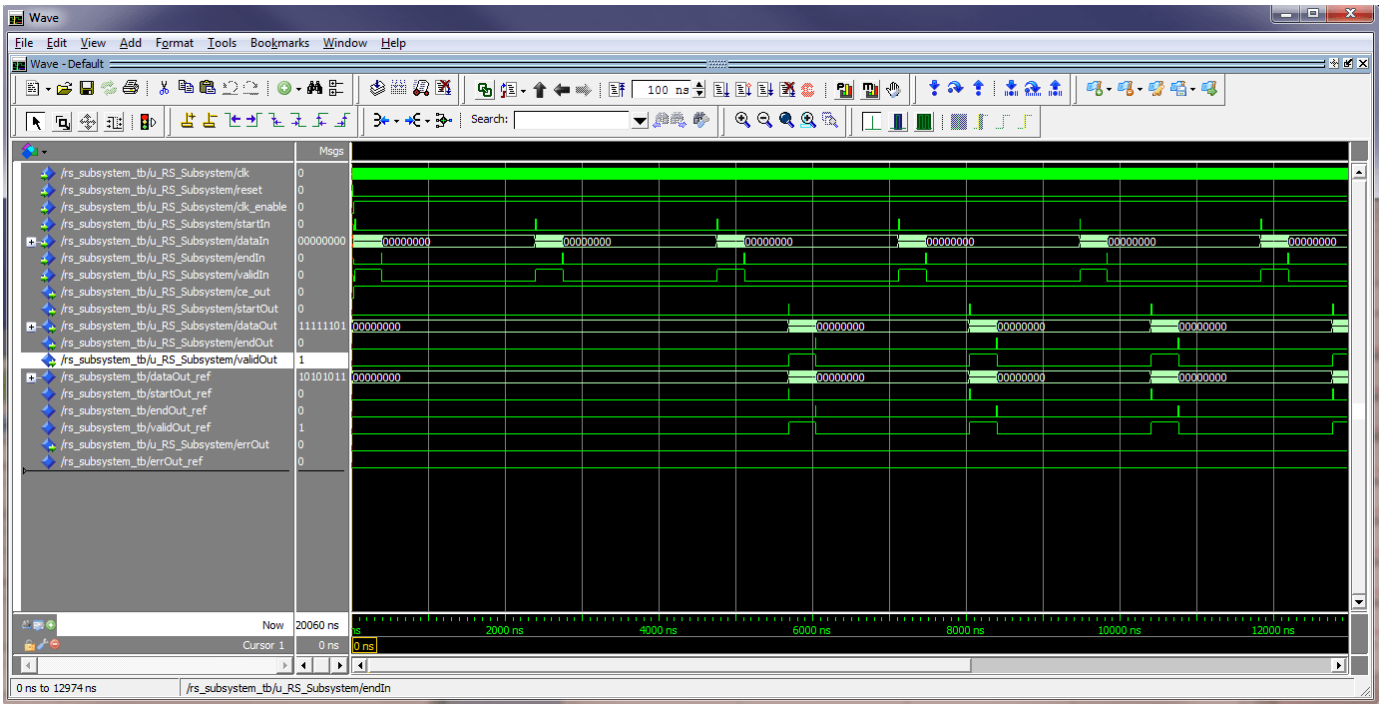
```
makehdl('commrshdl/RS Subsystem','TargetDirectory',workingdir);
```

Enter the following command to generate the test bench:

```
makehdltb('commrshdl/RS Subsystem','TargetDirectory',workingdir);
```

ModelSim® Output

The following figure shows the ModelSim HDL simulator after running the generated .do file scripts for the test bench. Compare the ModelSim result with the Simulink® result as plotted before.



Selected References

1. IEEE 802.16: IEEE Standard for Air Interface for Broadband Wireless Access Systems(Revision of IEEE Std 802.16-2009). IEEE-SA. 8 June 2012.

2. George C. Clark Jr, J. Bibb Cain, Error-Correction Coding for Digital Communications, New York: Springer, 1981.

Frequency Offset Calibration for Receivers

This example shows how to measure and calibrate for the frequency offset between a transmitter and a receiver at the receiver using MATLAB® and Communications Toolbox™. You can either use captured signals or receive signals in real time using the *Communications Toolbox Support Package for RTL-SDR Radio*. The receiver monitors the received signal, calculates the frequency offset between the transmitter and the receiver and displays it in the MATLAB® command window.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- *Communications Toolbox™*

To receive signals in real time, you also need the following hardware:

- RTL-SDR radio

and the following software

- *Communications Toolbox Support Package for RTL-SDR Radio*

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

If you choose to receive signals in real time using a radio, you need to tune to a known broadcast pilot tone or provide a signal source with a known center frequency to establish a baseline. If you do not have a signal generator available, you can use a low-cost Family Radio Service walkie-talkie as a source. Note that the signal source must be narrowband, with a sine wave being an ideal source.

Background

All radio receivers exhibit a frequency offset as compared to the transmitter. In some cases, the frequency offset may be more than the receiver algorithm can handle. Therefore, you may need to calibrate your receiver to minimize the frequency offset.

The example provides the following information about the communication link:

- The quantitative value of the frequency offset in Hz and PPM
- A graphical view of the qualitative SNR level of the received signal

If you have a transmitter, you can use it to generate a narrowband signal, such as a tone.

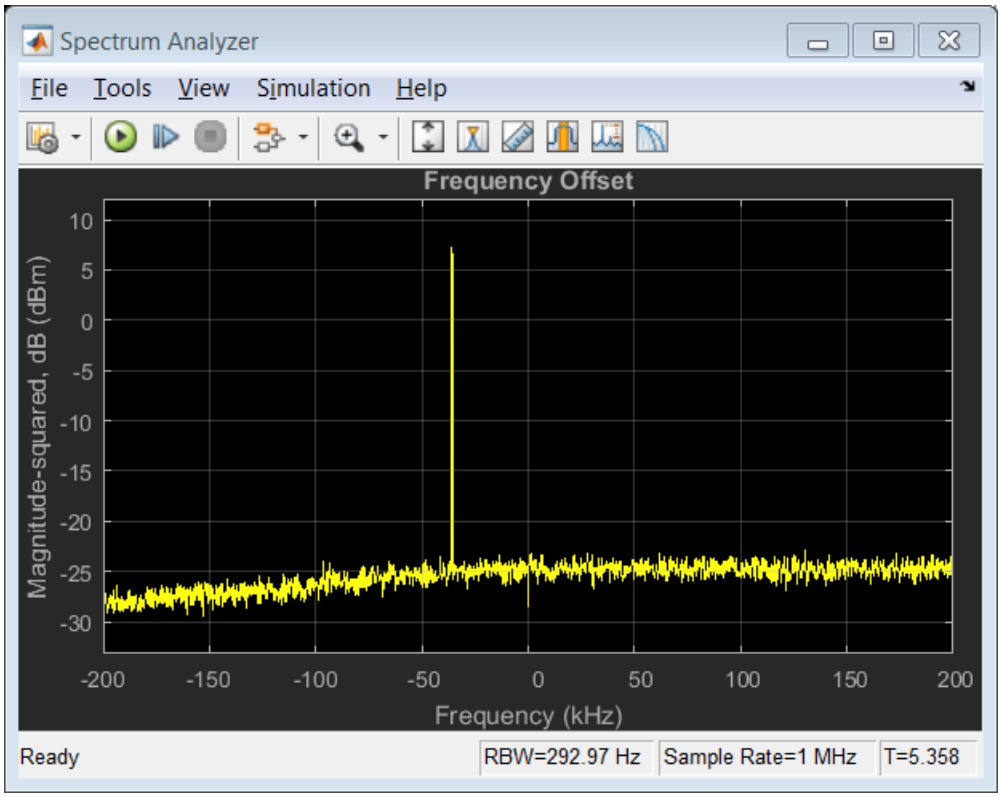
If you do not have a transmitter, you may be able to use a broadcast signal. For example, in USA, the ATSC digital TV signals include a narrowband pilot tone on the RF carrier. The pilot tone is usually at a nominal frequency of 309.440 kHz above the bottom edge of the channel. If such a signal is present in your area, you can set the expected center frequency value to the frequency of the tone. This example uses the pilot tone of channel 29, which is at approximately 560e6 + 309.440e3 Hz. For a list of channel number and frequency values, see *North American television frequencies*.

If you are using an RTL-SDR radio as the receiver, specify the displayed PPM correction value as the FrequencyCorrection property of the RTL-SDR Receiver System object™ to compensate for the frequency offset. Be sure to use the sign of the offset in your specification. Once you've done that, the spectrum displayed by the receiver's spectrum analyzer System object should have its maximum amplitude at roughly 0 Hz.

Run the Example

Begin transmitting with your known signal source. If you are in the USA, you can set the expected center frequency to the pilot tone of a near by digital TV transmitter. Then, type `FrequencyOffsetCalibrationForReceiversExample` in the MATLAB Command Window or click the 'Open example' button to open and run the example.

The example displays the spectrum of the received signal on a frequency range of -200 kHz to 200 kHz and prints the estimated frequency offset in Hz and PPM in the command window. In the case shown below, the frequency of the maximum received signal power is about -35 kHz.



Example Code

The receiver asks for user input and initializes variables. Then it calls the signal source, DC blocker, coarse tone frequency offset estimator, and spectrum analyzer in a loop. The loop also keeps track of the radio time using the frame duration.

```
% Request user input from command-line for application parameters
userInput = helperFrequencyCalibrationUserInput;

% Calculate system parameters based on the user input
[fcParam,sigSrc] = helperFrequencyCalibrationConfig(userInput);

% Create a DC blocker system object to remove the DC component of the
% received signal and increase accuracy of the frequency offset estimation.
dcBlocker = dsp.DCBlocker('Algorithm', 'Subtract mean');

% Create a coarse frequency offset estimation System Object to calculate
% the offset. The system object performs an FFT on its input signal and
```

```
% finds the frequency of maximum power. This quantity is the frequency
% offset.
CFO = comm.CoarseFrequencyCompensator( ...
    'FrequencyResolution',  25, ...
    'SampleRate',           fcParam.FrontEndSampleRate);

% Create a spectrum analyzer scope to visualize the signal spectrum
scope = dsp.SpectrumAnalyzer(...
    'Name',             'Actual Frequency Offset',...
    'Title',            'Actual Frequency Offset', ...
    'SpectrumType',     'Power',...
    'FrequencySpan',    'Full', ...
    'SampleRate',       fcParam.FrontEndSampleRate, ...
    'YLimits',          [-40,10],...
    'SpectralAverages', 50, ...
    'FrequencySpan',    'Start and stop frequencies', ...
    'StartFrequency',   -200e3, ...
    'StopFrequency',    200e3,...
    'Position',         figposition([50 30 30 40]));
```

Stream Processing

```
msgLength = 0;
radioTime = 0;
secondCounter = 1;
while radioTime < userInput.Duration
  rxSig = sigSrc();
  rxSig = dcBlocker(rxSig);
  [~, offset] = CFO(rxSig);
  freqCorrection = (-offset / fcParam.ExpectedFrequency) * fcParam.FrontEndSampleRate;

  % Visualize spectrum and print results
  scope(rxSig);
  if radioTime > secondCounter
    fprintf(repmat('\b', 1, msgLength));
    msg = sprintf(['Frequency offset = %f Hz,\n' ...
      'Frequency correction value (Hz) = %f \n' ...
      'Frequency correction value (PPM) = %f \n'], ...
      offset, -offset, freqCorrection);
    fprintf(msg);
    msgLength = numel(msg);
    secondCounter = secondCounter + 1;
  end

  % Update radio time
  radioTime = radioTime + fcParam.FrameDuration;
end

% Release all System objects
release(sigSrc);
release(dcBlocker);
release(CFO);
```

Conclusion

In this example, you used Communications Toolbox™ System objects to build a receiver that calculates the relative frequency offset between a transmitter and a receiver.

Airplane Tracking Using ADS-B Signals

This example shows you how to track planes by processing Automatic Dependent Surveillance-Broadcast (ADS-B) signals using MATLAB® and Communications Toolbox™. You can either use captured signals or receive signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio. The example can show the tracked planes on a map, if you have the Mapping Toolbox™.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Communications Toolbox™

To receive signals in real time, you also need one of the following SDR devices and the corresponding support package Add-On:

- RTL-SDR radio and the corresponding Communications Toolbox Support Package for RTL-SDR Radio software Add-On
- ADALM-PLUTO radio and the corresponding Communications Toolbox Support Package for ADALM-PLUTO Radio software Add-On

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

Background

ADS-B is a cooperative surveillance technology for tracking aircraft. This technology enables an aircraft to periodically broadcast its position information (altitude, GPS coordinates, heading, etc.) using the Mode-S signaling scheme.

Mode-S is a type of aviation transponder interrogation mode. When an aircraft receives an interrogation request, it sends back a transponder's *squawk code*. This is referred to as Mode 3A. Mode-S (Select) is another type of interrogation mode that is designed to help avoid interrogating the transponder too often. More details about Mode-S can be found in [1]. This mode is widely adopted in Europe and is being phased in for North America.

Mode-S signaling scheme uses squitter messages, which are defined as a non-solicited messages used in aviation radio systems. Mode-S has the following properties:

- Transmit Frequency: 1090 MHz
- Modulation: Pulse Position Modulation
- Data Rate: 1 Mbit/s
- Short Squitter Length: 56 microseconds
- Extended Squitter Length: 112 microseconds

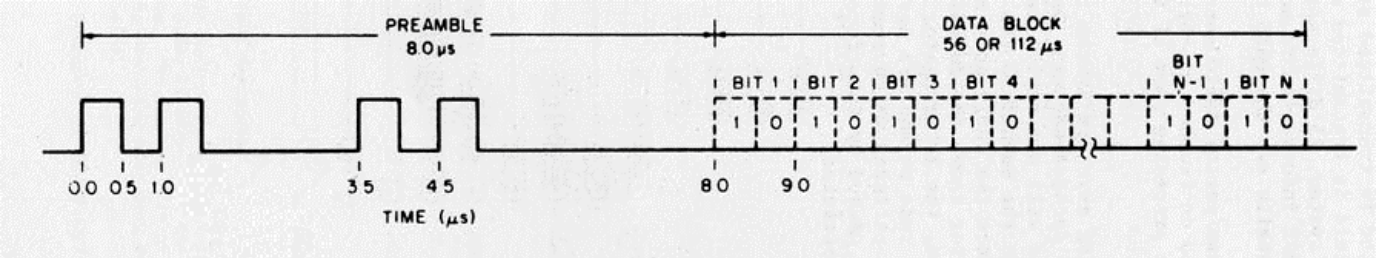
Short squitter messages contain the following information:

- Downlink Format (DF)
- Capability (CA)
- Aircraft ID (Unique 24-bit sequence)
- CRC Checksum

Extended squitter (ADS-B) messages contain all the information in a short squitter and one of these:

- Altitude
- Position
- Heading
- Horizontal and Vertical Velocity

The signal format of Mode-S has a sync pulse that is 8 microseconds long followed by either 56 or 112 microseconds of data as illustrated in the following figure.

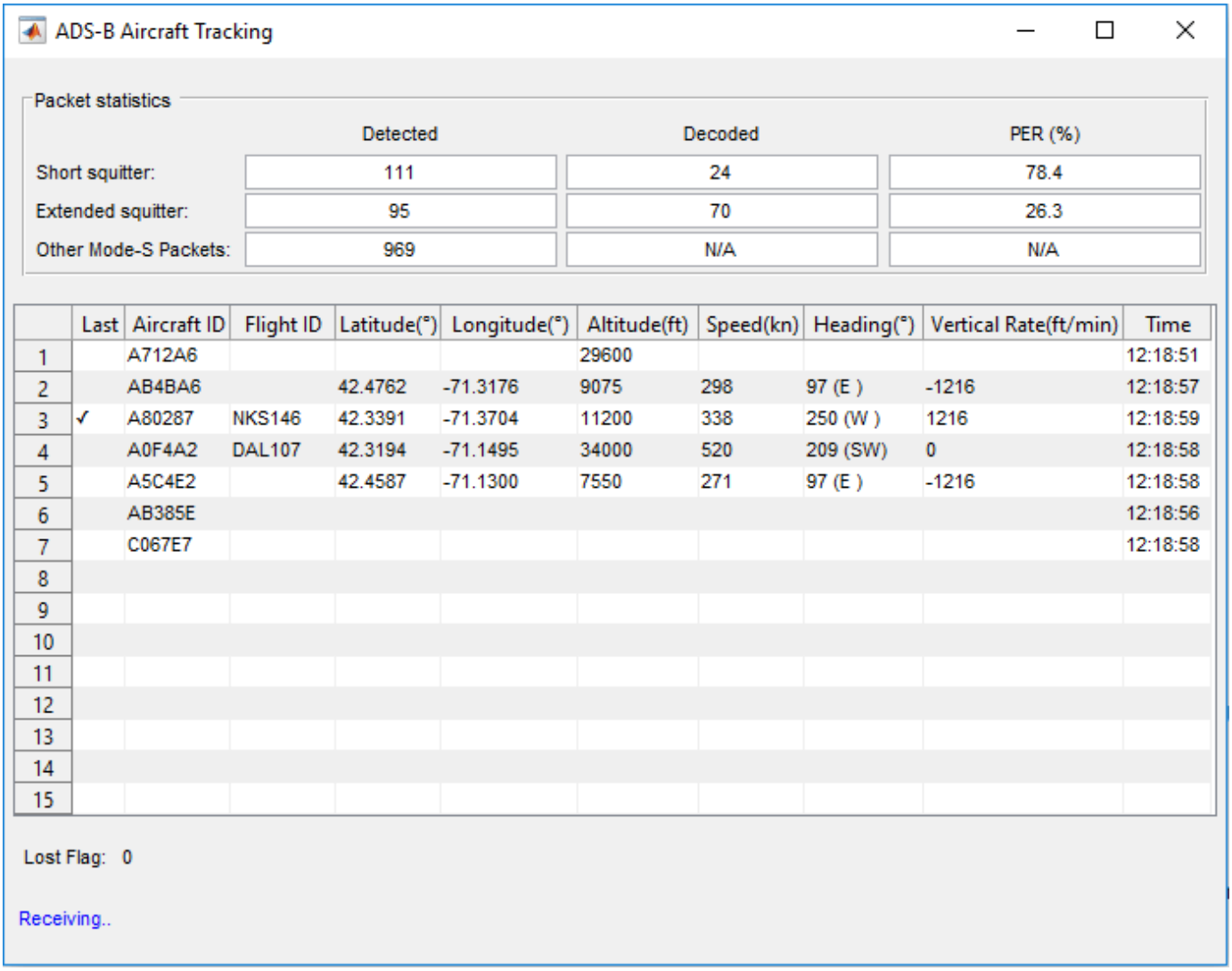


Run the Example

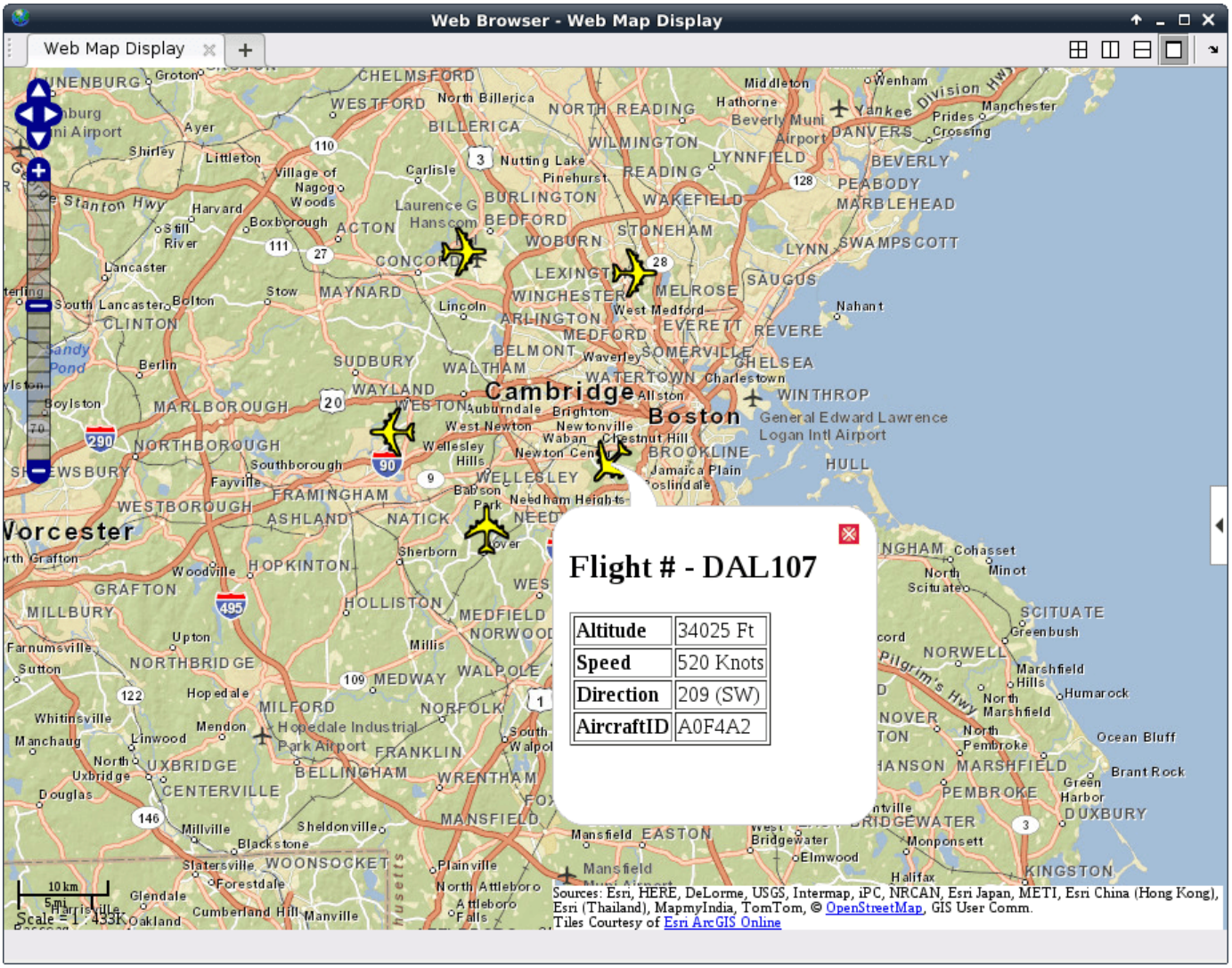
Type `ADSBExample` in the MATLAB command window or click the link to run the example. You need to enter the following information when you run the example:

1. Reception duration in seconds,
2. Signal source (captured data or RTL-SDR radio or ADALM-PLUTO radio),
3. Optional output methods (map and/or text file).

The example shows the information on the detected airplanes in a tabular form as shown in the following figure.

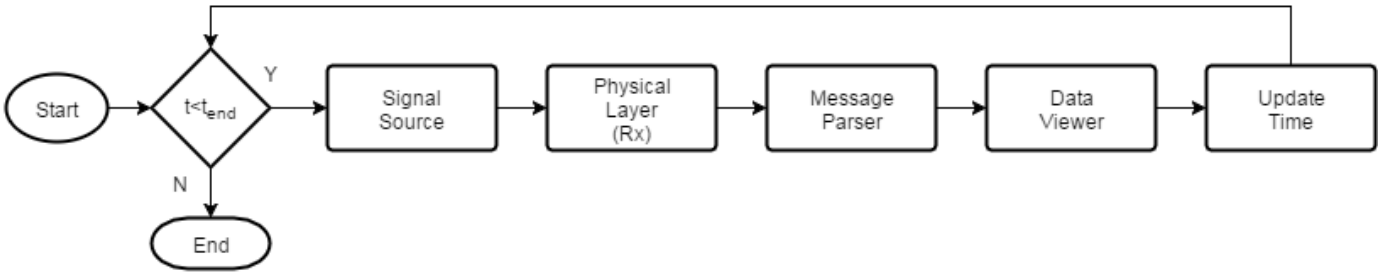


You can also observe the airplanes on a map, if you have a valid license for the Mapping Toolbox.



Receiver Structure

The following block diagram summarizes the receiver code structure. The processing has four main parts: Signal Source, Physical Layer, Message Parser, and Data Viewer.



Signal Source

This example can use three signal sources:

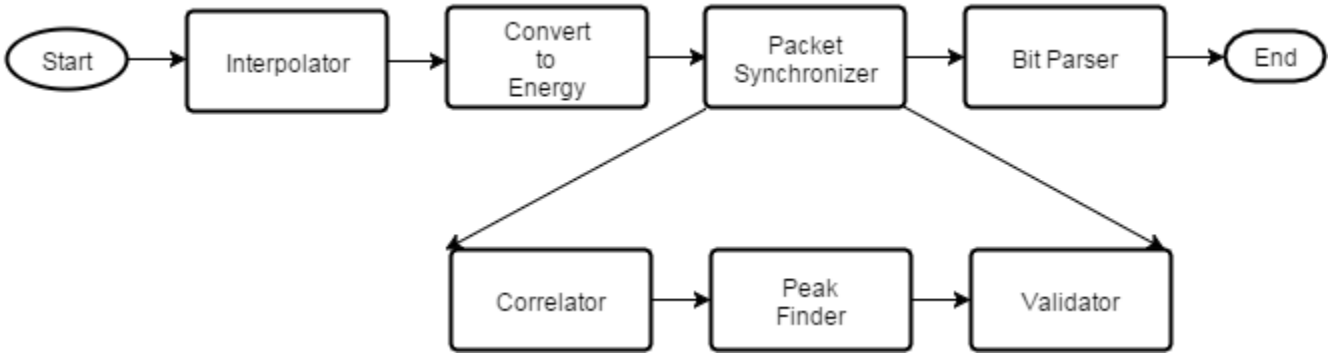
1. ''Captured Signal'': Over-the-air signals written to a file and sourced from a Baseband File Reader object at 2.4 Msps
2. ''RTL-SDR Radio'': RTL-SDR radio at 2.4 Msps
3. ''ADALM-PLUTO Radio'': ADALM-PLUTO radio at 12 Msps

If you assign ''RTL-SDR'' or ''ADALM-PLUTO'' as the signal source, the example searches your computer for the radio you specified, either an RTL-SDR radio at radio address '0' or an ADALM-PLUTO radio at radio address 'usb:0' and uses it as the signal source.

Here the extended squitter message is 120 micro seconds long, so the signal source is configured to process enough samples to contain 180 extended squitter messages at once, and set `SamplesPerFrame` of the signal property accordingly. The rest of the algorithm searches for Mode-S packets in this frame of data and outputs all correctly identified packets. This type of processing is defined as batch processing. An alternative approach is to process one extended squitter message at a time. This single packet processing approach incurs 180 times more overhead than the batch processing, while it has 180 times less delay. Since the ADS-B receiver is delay tolerant, batch processing was used.

Physical Layer

The baseband samples received from the signal source are processed by the physical (PHY) layer to produce packets that contain the PHY layer header information and raw message bits. The following diagram shows the physical layer structure.



The RTL-SDR radio is capable of using a sampling rate in the range [200e3, 2.8e6] Hz. When RTL-SDR radio is the source, the example uses a sampling rate of 2.4e6 Hz and interpolates by a factor of 5 to a practical sampling rate of 12e6 Hz.

The ADALM-PLUTO radio is capable of using a sampling rate in the range [520e3, 61.44e6] Hz. When the ADALM-PLUTO radio is the source, the example samples the input directly at 12 MHz.

With the data rate of 1 Mbit/s and a practical sampling rate of 12 MHz, there are 12 samples per symbol. The receive processing chain uses the magnitude of the complex symbols.

The packet synchronizer works on subframes of data equivalent to two extended squitter packets, that is, 1440 samples at 12 MHz or 120 micro seconds. This subframe length ensures that a whole extended squitter packet is contained in the subframe. The Packet synchronizer first correlates the received signal with the 8 microsecond preamble and finds the peak value. Then, it validates the synchronization point by checking if it matches the preamble sequence, [1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0], where a '1' represents a high value and a '0' represents a low value.

The Mode-S PPM modulation scheme defines two symbols. Each symbol has two chips, where one has a high value and the other has a low value. If the first chip is high followed by low chip, this corresponds to the symbol being a 1. Alternatively, if the first chip is low followed by high chip, then the symbol is 0. The bit parser demodulates the received chips and creates a binary message. The binary message is validated using a CRC checker. The output of bit parser is a vector of Mode-S physical layer header packets that contains the following fields:

- RawBits: Raw message bits
- CRCError: FALSE if CRC checks, TRUE if CRC fails
- Time: Time of reception in seconds from start of receiver
- DF: Downlink format (packet type)
- CA: Capability

Message Parser

The message parser extracts data from the raw bits based on the packet type as described in [2]. This example can parse short squitter packets and extended squitter packets that contain airborne velocity, identification, and airborne position data.

Data Viewer

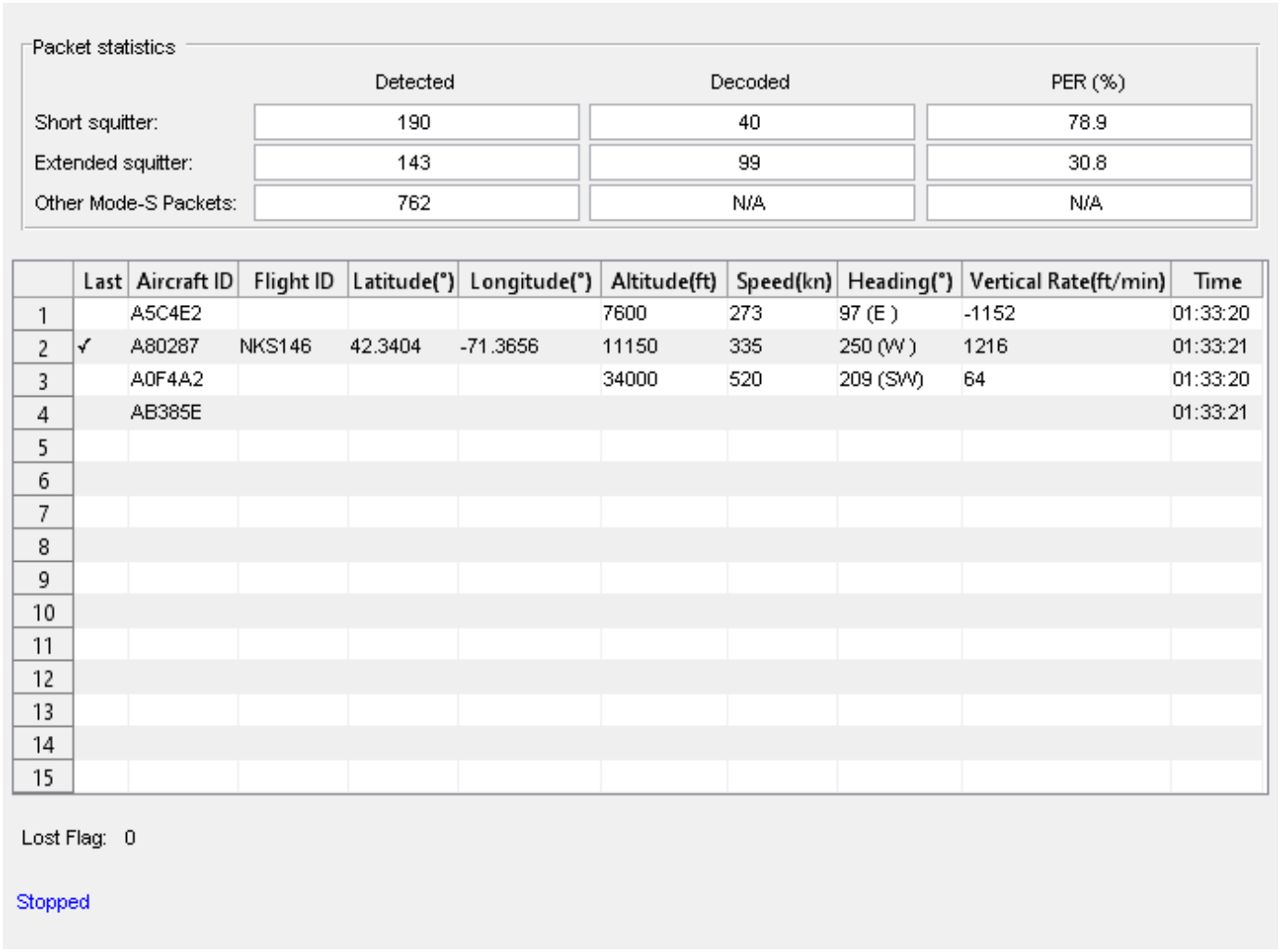
The data viewer shows the received messages on a graphical user interface (GUI). For each packet type, the number of detected packets, the number of correctly decoded packets, and the packet error rate (PER) is shown. As data is captured, the application lists information decoded from these messages in a tabular form.

Example Code

The receiver asks for user input and initializes variables. Then it calls the signal source, physical layer, message parser, and data viewer in a loop. The loop keeps track of the radio time using the frame duration.

```
%For the option to change default settings, set |cmdlineInput| to 1.
cmdlineInput = 0;
if cmdlineInput
    % Request user input from the command-line for application parameters
    userInput = helperAdsbUserInput;
else
    load('defaultinputsADSB.mat');
end

% Calculate ADS-B system parameters based on the user input
[adsbParam,sigSrc] = helperAdsbConfig(userInput);

% Create the data viewer object and configure based on user input
viewer = helperAdsbViewer('LogFileName',userInput.LogFilename, ...
    'SignalSourceType',userInput.SignalSourceType);
if userInput.LogData
  startDataLog(viewer);
end
if userInput.LaunchMap
  startMapUpdate(viewer);
end

% Create message parser object
```

```
msgParser = helperAdsbRxMsgParser(adsbParam);

% Start the viewer and initialize radio time
start(viewer)
radioTime = 0;

% Main loop
while radioTime < userInput.Duration


     if adsbParam.isSourceRadio
        if adsbParam.isSourcePlutoSDR
            [rcv,~,lostFlag] = sigSrc();
        else
            [rcv,~,lost] = sigSrc();
            lostFlag = logical(lost);
        end
    else
        rcv = sigSrc();
        lostFlag = false;
     end

  % Process physical layer information (Physical Layer)
  [pkt,pktCnt] = helperAdsbRxPhy(rcv,radioTime,adsbParam);

  % Parse message bits (Message Parser)
  [msg,msgCnt] = msgParser(pkt,pktCnt);

  % View results packet contents (Data Viewer)
  update(viewer,msg,msgCnt,lostFlag);

  % Update radio time
  radioTime = radioTime + adsbParam.FrameDuration;
end

% Stop the viewer and release the signal source
stop(viewer)
release(sigSrc)
```



Further Exploration

You can further explore ADS-B signals using the ADSBExampleApp app. This app allows you to select the signal source and change the duration. To launch the app, type `DSBExampleApp` in the MATLAB command window or click the link.

You can explore following helper functions for details of the physical layer implementation:

- helperAdsbRxPhy.m
- helperAdsbRxPhySync.m
- helperAdsbRxPhyBitParser.m
- helperAdsbRxMsgParser.m

Selected Bibliography

1 International Civil Aviation Organization, Annex 10, Volume 4. Surveillance and Collision Avoidance Systems.

2 Technical Provisions For Mode S Services and Extended Squitter (Doc 9871)

Automatic Meter Reading

This example shows you how to use Communications Toolbox™ to read utility meters by processing Standard Consumption Message (SCM) signals and Interval Data Message (IDM) signals which are emitted by Encoder-Receiver-Transmitter (ERT) compatible meters. You can either use recorded data from a file, or receive over-the-air signals in real time using an RTL-SDR or ADALM-PLUTO radio.

In Simulink®, you can explore the “Automatic Meter Reading in Simulink” on page 8-522 example.

Required Hardware and Software

To run this example using recorded data from a file, you need Communications Toolbox™.

To receive signals in real time, you also need one of these SDR devices and the corresponding support package Add-On:

- RTL-SDR radio and the corresponding Communications Toolbox Support Package for RTL-SDR Radio
- ADALM-PLUTO radio and the corresponding Communications Toolbox Support Package for ADALM-PLUTO Radio

For more information, see the *Software Defined Radio (SDR) discovery page*.

Background

Automatic Meter Reading (AMR) is a technology that autonomously collects the consumption and status data from utility meters (such as electric, gas, or water meters) and delivers the data to utility providers for billing or analysis purposes. The AMR system utilizes low power radio frequency (RF) communication to transmit meter readings to a remote receiver. The RF transmission properties include:

- Transmission frequency within range: 910-920 MHz
- Data rate: 32768 bps
- On-off keyed Manchester coded signaling

The SCM and IDM are two types of the conventional message types that the meters send out. The SCM packets are used with a fixed length of 96 bits, whereas IDM packets are used with a fixed length of 736 bits. These tables show the packet format of the SCM and IDM messages:

SCM Packet Format

Field	Length (bits)	Fixed Value	Notes
Sync bit	1	1	
Preamble	20	0xF2A60	
ERT ID MS bits	2		MSBs of the meter serial number
Reserved	1		
Physical tamper	2		
ERT type	4		determines the commodity type, e.g., electric
Encoder tamper	2		
Consumption data	24		Actual meter reading
ERT ID LS bits	24		remaining bits of the meter serial number
Checksum	16		A BCH(255, 239) shortened to message length of 59 with genrator polynomial g(X) = $(267543)_8$

IDM Packet Format

Field	Length (bytes)	Fixed Value	Notes
Training sync	2	0x5555	
Frame sync	2	0x16A3	
Packet type	1	0x1C	
Packet length	2	0x5CC6	The first byte is the number of total bytes and the last byte is the Hamming code of the first byte.
Application version	1	0x04	
ERT Type	1	0x17	The last 4 bits determines the commodity type, e.g., electric
ERT serial number	4		
Consumption interval count	1		Incrementing interval counter that increments at the end of each interval and rolls over to 0 after 255
Module programming state	1		
Tamper count	6		
Async count	2		
Power outage flags	6		
Last consumption count	4		
Differential consumption intervals*	53		47 intervals, represented by 9-bit integers
Transmit time offset	2		Time elapsed since the last interval ended
Serial number CRC	2		CRC-16-CCITT of ERT serial number
Packet CRC	2		CRC-16-CCITT of packet starting at Packet type

* Each integer shows the amount of the metered quantity that was consumed during each interval. Units will vary based on meter type and version. The leftmost interval is the most recent.

Meters capable of sending both SCM and IDM messages transmit them on the same channel with separation of roughly 275 msec. Each meter transmits the SCM and IDM messages over multiple frequencies using a hopping pattern. The actual transmission frequencies, the frequency hopping pattern, and the time interval between transmissions are random to avoid interference from other transmissions. For more information, see reference [1].

Run Example

When you run the example:

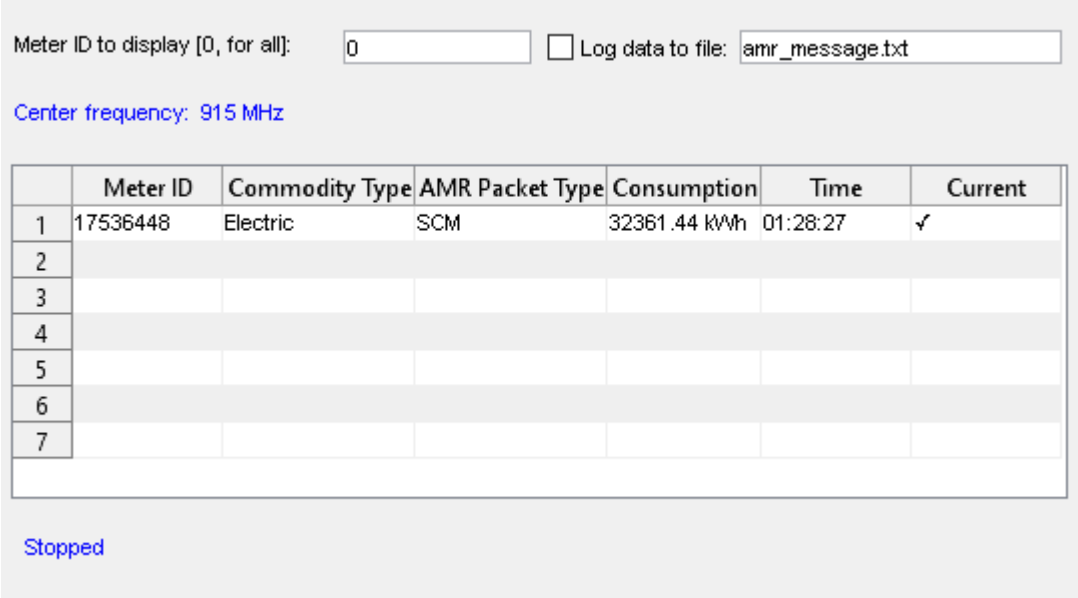
- The receiver initializes the simulation parameters and calculates the AMR parameters.
- A data viewer display shows the meter ID, consumption information, and commodity type.
- The simulation loop calls the signal source, physical layer, message parser, and data viewer.
- The processing loop keeps track of the radio time using the frame duration.
- The display updates for each data capture, showing unique meter IDs with the latest consumption information.

Initialize Parameters

The default signal source is 'File', which runs the example using the recorded baseband signal file `amr_capture_01.bb`. To run the example using your RTL or ADALM-PLUTO SDR, change the setting for `signalSource` when you call the helperAMRInit.m file. Valid options for `signalSource` are 'File', 'RTL-SDR', and 'ADALM-PLUTO'.

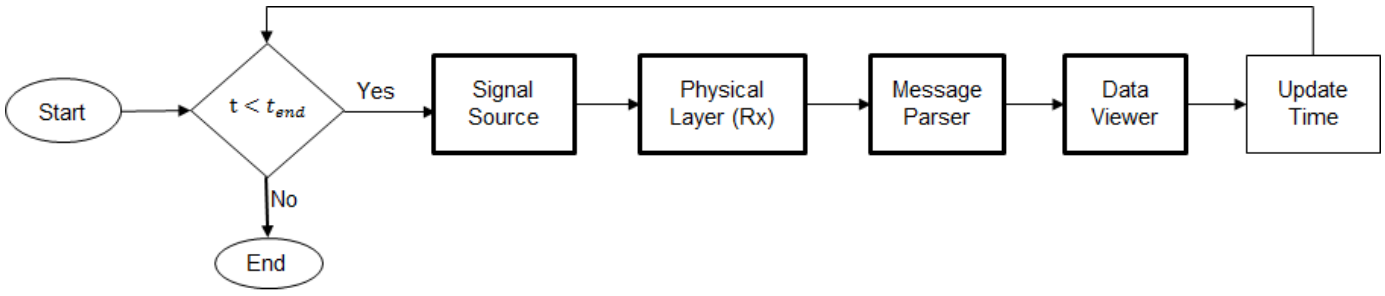
```
signalSource = 'File';
initParam = helperAMRInit(signalSource);

% Calculate AMR system parameters based on the initialized parameters
[amrParam,sigSrc] = helperAMRConfig(initParam);

% Create the data viewer object
viewer = helperAMRViewer('MeterID',initParam.MeterID, ...
    'LogData',initParam.LogData, ...
    'LogFilename',initParam.LogFilename, ...
    'Fc',amrParam.CenterFrequency, ...
    'SignalSourceType',initParam.SignalSourceType);
```

```
start(viewer);
radioTime = 0; % Initialize the radio time

% Main Processing Loop
while radioTime < initParam.Duration
    rcvdSignal = sigSrc();
    amrBits = helperAMRRxPHY(rcvdSignal,amrParam);
    amrMessages = helperAMRMessageParser(amrBits,amrParam);
    update(viewer,amrMessages);
    radioTime = radioTime + amrParam.FrameDuration;
end

stop(viewer); % Stop the viewer
release(sigSrc); % Release the signal source
```



Receiver Code Structure

The flow chart summarizes the receiver code structure. The processing has four main parts: Signal Source, Physical Layer, Message Parser and Data Viewer.



Signal Source

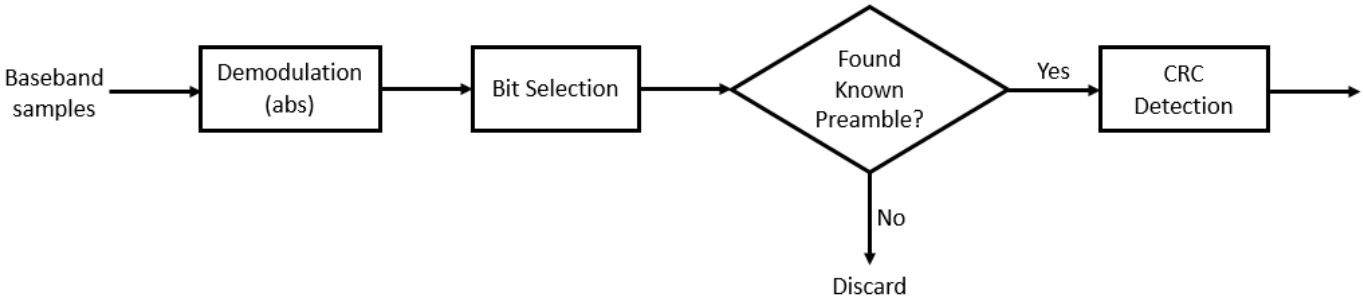
This example can use three signal sources:

1. ''File'': Over-the-air signals written to a file and read using a Baseband File Reader object at 1.0 Msps
2. ''RTL-SDR'': RTL-SDR radio at a sample rate of 1.0 Msps
3. ''ADALM-PLUTO'': ADALM-PLUTO radio at a sample rate of 1.0 Msps

If you assign ''RTL-SDR'' or ''ADALM-PLUTO'' as the signal source, the example searches your computer for the radio you specified, either an RTL-SDR radio at radio address '0' or an ADALM-PLUTO radio at radio address 'usb:0' and uses it as the signal source.

Physical Layer

The baseband samples received from the signal source are processed by the physical layer (PHY) to produce packets that contain the SCM or IDM information. This diagram shows the physical layer receive processing.



The RTL-SDR radio is capable of using a sampling rate in the range of 225-300 kHz or 900-2560 kHz. The ADALM-PLUTO radio is capable of using a sampling rate in the range of 520 kHz-61.44 MHz. A sampling rate of 1.0 Msps is used to produce a sufficient number of samples per Manchester encoded data bit. For each frequency in the hopping pattern, every AMR data packet is transmitted. The frequency hopping allows for increased reliability over time. Since every packet is transmitted on each frequency hop, it is sufficient to monitor only one frequency for this example. The radio is tuned to a center frequency of 915 MHz for the entire simulation runtime.

The received complex samples are amplitude demodulated by extracting their magnitude. The on-off keyed Manchester coding implies the bit selection block includes clock recovery. This block outputs bit sequences (ignoring the idle times in the transmission) which are subsequently checked for the known preamble. If the preamble matches, the bit sequence is further decoded, otherwise, it is discarded and the next sequence is processed.

When the known SCM preamble is found for a bit sequence, the received message bits are decoded using a shortened (255,239) BCH code which can correct up to two bit errors. In the case where the known IDM preamble is found, the receiver performs a cyclic redundancy check (CRC) of the meter serial number and of the whole packet starting at the Packet type (the 5th byte) to determine if the packet is valid. Valid, corrected messages are passed onto the AMR message parser.

Message Parser

For a valid message, the bits are then parsed into the specific fields of the SCM or the IDM format.

Data Viewer

The data viewer shows the decoded packets on a separate MATLAB figure. For each successfully decoded packet, the meter ID, commodity type, AMR packet type, consumption information and the

capture time is shown. As data is captured and decoded, the application lists the information decoded from these messages in a tabular form. The table lists only the unique meter IDs with their latest consumption information.

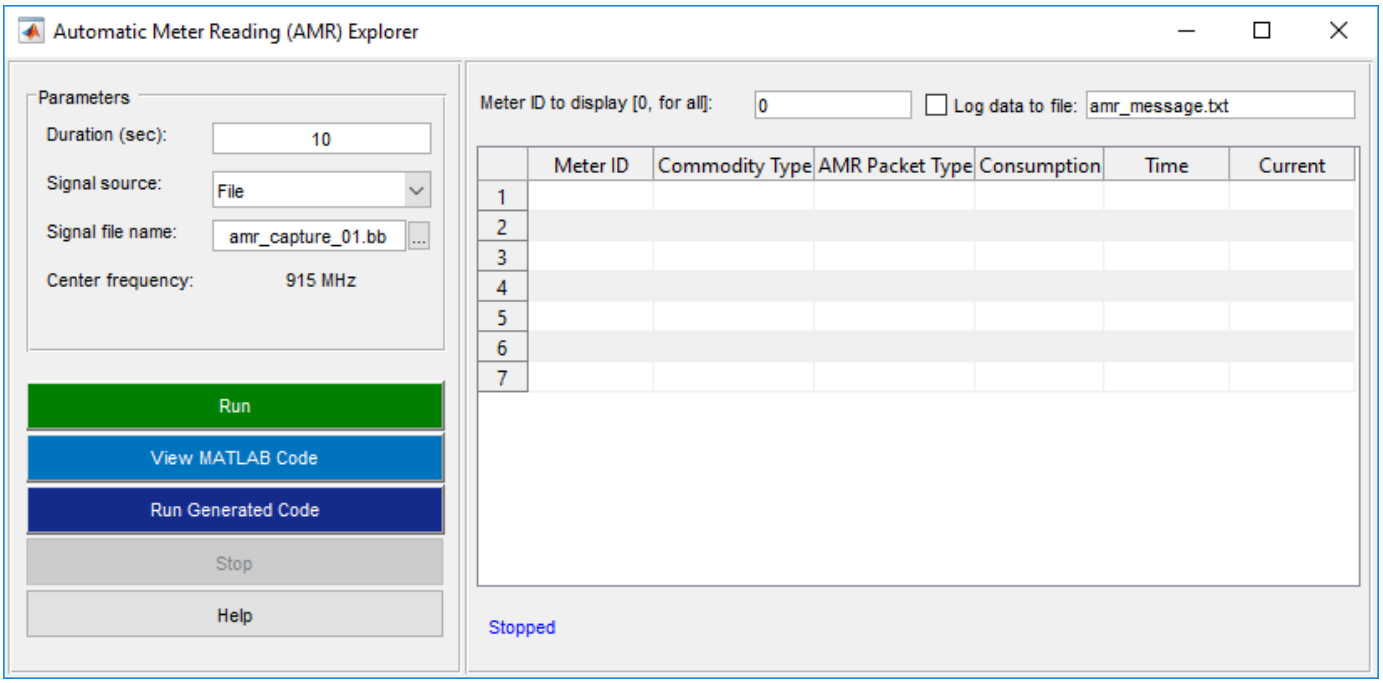
You can also change the meter ID and start text file logging using the data viewer.

- **Meter ID** - Change the meter ID from 0, which is the default value and is reserved for displaying all detected meters, to a specific meter ID which you would like to be displayed.
- **Log data to file** - Save the decoded messages in a TXT file. You can use the saved data for post processing.

Further Exploration

The data file accompanying the example has only one meter reading and has been captured at center frequency of 915 MHz. Using RTL-SDR or ADALM-PLUTO, the example will display readings from multiple meters when it is run for a longer period in a residential neighborhood.

You can further explore AMR signals using the AMRExampleApp user interface. This app allows you to select the signal source and change the center frequency of the RTL-SDR or ADALM-PLUTO. This link launchs the AMRExampleApp app shown here.



You can also explore the following functions for details of the physical layer, AMR message formats:

- helperAMRRxPHY.m
- helperAMRRxDiscreteEvent.m
- helperAMRRxBitParser.m
- helperAMRMessageParser.m

A version of the example that works with multiple radios is AMRMultipleRadios.m. This allows you to examine the frequency hop patterns for a meter by setting different center frequencies per radio device available. The script is set for two radios, but can be extended for any number.

Selected Bibliography

1. Automatic meter reading, https://en.wikipedia.org/wiki/Encoder_receiver_transmitter, 2016.
2. Itron Electricity meters, https://www.itron.com/solutions/who-we-serve/electricity, 2017.

Packetized Modem with Data Link Layer

This example shows you how to implement a packetized modem with Data Link Layer [1] using MATLAB® and Communications Toolbox™. The modem features a packet-based physical layer and an ALOHA-based Data Link Layer. You can either simulate the system or run with radios using the *Communications Toolbox Support Package for USRP® Radio*.

Required Hardware and Software

To simulate the system performance, you need the following software:

- *Communications Toolbox™*

To measure system performance with radios, you also need the following hardware:

- USRP® radios (B2xx, N2xx, or X3xx)

and the following software

- *Communications Toolbox Support Package for USRP® Radio*

For a full list of Communications Toolbox supported SDR platforms, refer to the Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

Introduction

Packetized wireless modems are communications systems that transmit information in bursts called packets through a wireless channel. Each modem, also called a node, features a physical layer where packets are modulated, transmitted and received on a shared frequency band, and demodulated. Since the same frequency band is used by all nodes, a medium access control (MAC) algorithm is required to reduce the packet loss due to collisions (i.e. simultaneous transmissions). Data Link Layer includes a MAC sublayer and a logical link control sublayer to share the same channel and provides an error-free link between two nodes. Data Link Layer is also called Layer 2 and sits between Network Layer (Layer 3) and Physical Layer (Layer 1).

Run the Example

The example code creates three packetized modem node objects and connects them through a channel object. Each node can send packets to the other two nodes. `ACKTimeout` determines the timeout duration before a node decides the DATA packet transmission was not successful. `ACKTimeout` must be greater than the round trip duration for a DATA-ACK exchange, which is 0.21 seconds for this example. The simulation is time-based and simulates the full physical layer processing together with the data link layer.

Set simulation parameters

```
runDuration = 10;         % Seconds
numPayloadBits = 19530;   % Bits
packetArrivalRate = 0.2;  % Packets per second
ackTimeOut = 0.25;        % ACK time out in seconds
maxBackoffTime = 10;      % Maximum backoff time in ackTimeOut durations
mMaxDataRetries = 5;      % Maximum DATA retries
queueSize = 10;           % Data Link Layer queue size in packets
samplesPerFrame = 2000;   % Number of samples processed every iteration
verbose = true;           % Print packet activity to command line
sampleRate = 200e3;
```

Fix random number generation seed for repeatable simulations.

```
rng(12345)
```

Create packetized modem nodes by using the helperPacketizedModemNode object.

```
node1 = helperPacketizedModemNode('Address',1, ...
    'DestinationList',[2, 3],'NumPayloadBits',numPayloadBits, ...
    'PacketArrivalRate',packetArrivalRate,'ACKTimeOut',ackTimeOut, ...
    'MaxBackoffTime',maxBackoffTime,'MaxDataRetries',mMaxDataRetries, ...
    'QueueSize',queueSize,'CarrierDetectorThreshold',1e-5, ...
    'AGCMaxPowerGain',65,'SamplesPerFrame',samplesPerFrame, ...
    'Verbose',verbose,'SampleRate',sampleRate);
node2 = helperPacketizedModemNode('Address',2, ...
    'DestinationList',[1 3],'NumPayloadBits',numPayloadBits, ...
    'PacketArrivalRate',packetArrivalRate,'ACKTimeOut',ackTimeOut, ...
    'MaxBackoffTime',maxBackoffTime,'MaxDataRetries',mMaxDataRetries, ...
    'QueueSize',queueSize,'CarrierDetectorThreshold',1e-5, ...
    'AGCMaxPowerGain',65,'SamplesPerFrame',samplesPerFrame, ...
    'Verbose',verbose,'SampleRate',sampleRate);
node3 = helperPacketizedModemNode('Address',3, ...
    'DestinationList',[1 2],'NumPayloadBits',numPayloadBits, ...
    'PacketArrivalRate',packetArrivalRate,'ACKTimeOut',ackTimeOut, ...
    'MaxBackoffTime',maxBackoffTime,'MaxDataRetries',mMaxDataRetries, ...
    'QueueSize',queueSize,'CarrierDetectorThreshold',1e-5, ...
    'AGCMaxPowerGain',65,'SamplesPerFrame',samplesPerFrame, ...
    'Verbose',verbose,'SampleRate',sampleRate);
```

Configure the propagation channel by using the helperMultiUserChannel object.

```
channel = helperMultiUserChannel( ...
    'NumNodes',3,'EnableTimingSkew',true,'DelayType','Triangle', ...
    'TimingError',20,'EnableFrequencyOffset',true, ...
    'PhaseOffset',47,'FrequencyOffset',2000,'EnableAWGN',true, ...
    'EbNo',25,'BitsPerSymbol',2,'SamplesPerSymbol',4, ...
    'EnableRicianMultipath', true, ...
    'PathDelays',[0 node1.SamplesPerSymbol/node1.SampleRate], ...
    'AveragePathGains',[15 0],'KFactor',15,'MaximumDopplerShift',10, ...
    'SampleRate',node1.SampleRate);
```

Main simulation loop

```
radioTime = 0;
nodeInfo = info(node1);
frameDuration = node1.SamplesPerFrame/node1.SampleRate;
[rcvd1,rcvd2,rcvd3] = deal(complex(zeros(node1.SamplesPerFrame,1)));
while radioTime < runDuration
    trans1 = node1(rcvd1, radioTime);
    trans2 = node2(rcvd2, radioTime);
    trans3 = node3(rcvd3, radioTime);

    % Multi-user channel
    [rcvd1,rcvd2,rcvd3] = channel(trans1,trans2,trans3);

    % Update radio time.
    radioTime = radioTime + frameDuration;
end
```

```
|   Time    |     Link    |   Action   |  Seq #  | Backoff (Node 1)
-------------------------------------------------------------
|   Time    |     Link    |   Action   |  Seq #  | Backoff (Node 2)
-------------------------------------------------------------
|   Time    |     Link    |   Action   |  Seq #  | Backoff (Node 3)
-------------------------------------------------------------
| 4.46000 s |   3 ->>   1 |       DATA | #     0 |
| 4.67000 s |   1 <<-   3 |       DATA | #     0 |
| 4.67000 s |   1 ->>   3 |        ACK | #     0 |
| 4.68000 s |   3 <<-   1 |        ACK | #     0 |
| 5.04000 s |   1 ->>   3 |       DATA | #     0 |
| 5.16000 s |   2 ->>   3 |       DATA | #     0 |
| 5.30000 s |   1 ->>   3 |   Back Off | #     0 | 1.00000 s |
| 5.42000 s |   2 ->>   3 |   Back Off | #     0 | 1.00000 s |
| 6.31000 s |   1 ->>   3 |       DATA | #     0 |
| 6.43000 s |   2 ->>   3 |       DATA | #     0 |
| 6.57000 s |   1 ->>   3 |   Back Off | #     0 | 2.25000 s |
| 6.69000 s |   2 ->>   3 |   Back Off | #     0 | 1.75000 s |
| 8.45000 s |   2 ->>   3 |       DATA | #     0 |
| 8.66000 s |   3 <<-   2 |       DATA | #     0 |
| 8.66000 s |   3 ->>   2 |        ACK | #     0 |
| 8.67000 s |   2 <<-   3 |        ACK | #     0 |
| 8.83000 s |   1 ->>   3 |       DATA | #     0 |
| 9.09000 s |   1 ->>   3 |   Back Off | #     0 | 2.25000 s |
| 9.52000 s |   3 ->>   2 |       DATA | #     1 |
| 9.73000 s |   2 <<-   3 |       DATA | #     1 |
| 9.73000 s |   2 ->>   3 |        ACK | #     1 |
| 9.74000 s |   3 <<-   2 |        ACK | #     1 |
```

Results

The packetized modem node objects collect statistics on the performance of the data link layer algorithm. Call the `info` method of the Node object to access these statistics. Sample results for a 10 second simulated time with a packet arrival rate of 0.2 packets/second are shown here. Each data packet is 200 msec long.

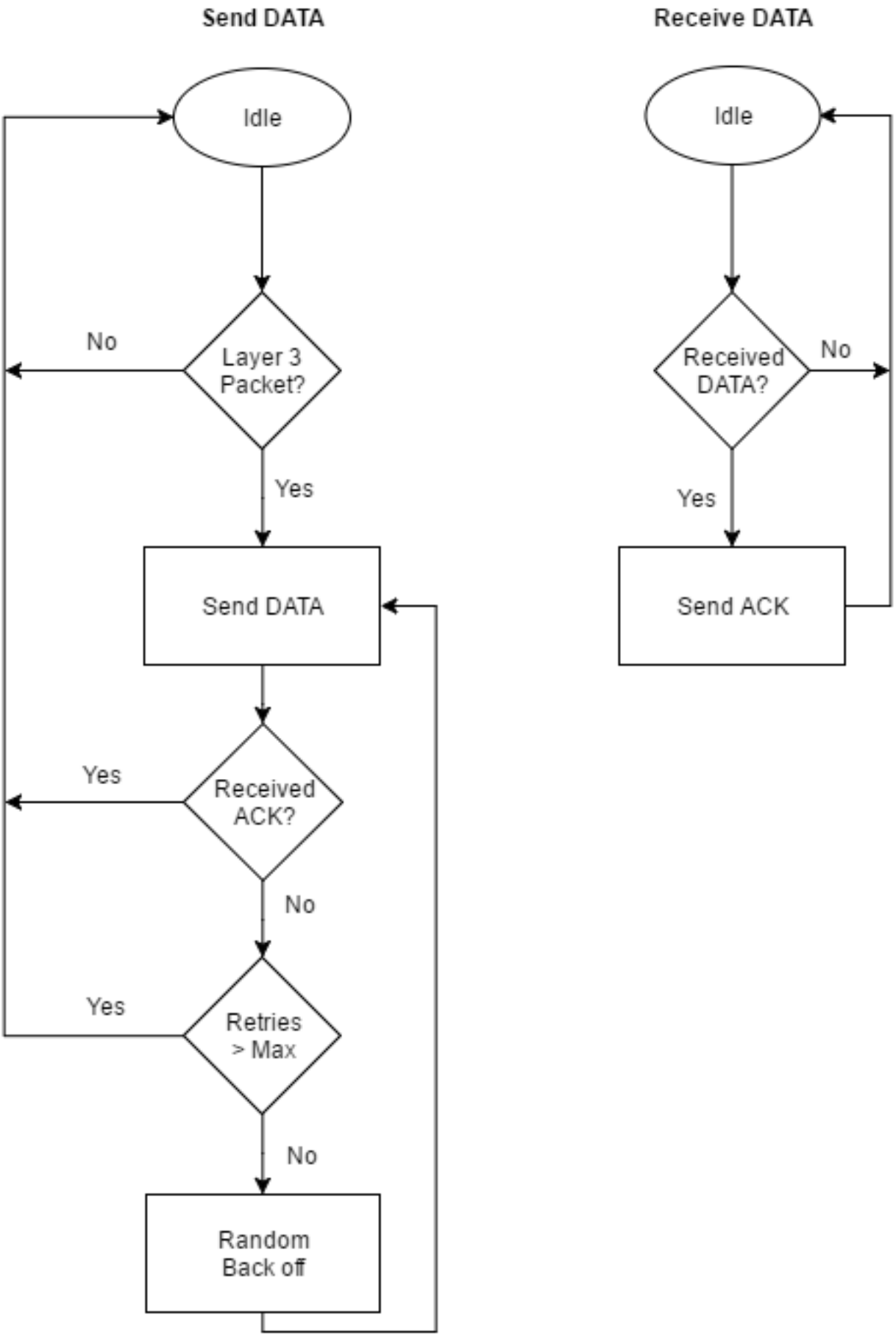
Display statistics

```
nodeInfo(1) = info(node1);
nodeInfo(2) = info(node2);
nodeInfo(3) = info(node3);

for p=1:length(nodeInfo)
    fprintf('\nNode %d:\n', p);
    fprintf('\tNumGeneratedPackets: %d\n', nodeInfo(p).NumGeneratedPackets)
    fprintf('\tNumReceivedPackets: %d\n', nodeInfo(p).NumReceivedPackets)
    fprintf('\tAverageRetries: %f\n', nodeInfo(p).Layer2.AverageRetries)
    fprintf('\tAverageRoundTripTime: %f\n', ...
        nodeInfo(p).Layer2.AverageRoundTripTime)
    fprintf('\tNumDroppedPackets: %d\n', ...
        nodeInfo(p).Layer2.NumDroppedPackets)
    fprintf('\tNumDroppedPackets (Max retries): %d\n', ...
        nodeInfo(p).Layer2.NumDroppedPacketsDueToRetries)
    fprintf('\tThroughput: %d\n', ...
        numPayloadBits / nodeInfo(p).Layer2.AverageRoundTripTime)
    fprintf('\tLatency: %d\n', nodeInfo(p).Layer2.AverageLatency)
end
```

```
Node 1:
    NumGeneratedPackets: 2
    NumReceivedPackets: 1
    AverageRetries: NaN
    AverageRoundTripTime: NaN
    NumDroppedPackets: 0
    NumDroppedPackets (Max retries): 0
    Throughput: NaN
    Latency: Inf

Node 2:
    NumGeneratedPackets: 1
    NumReceivedPackets: 1
    AverageRetries: 2.000000
    AverageRoundTripTime: 3.509844
    NumDroppedPackets: 0
    NumDroppedPackets (Max retries): 0
    Throughput: 5.564350e+03
    Latency: 2.104687e-01

Node 3:
    NumGeneratedPackets: 2
    NumReceivedPackets: 1
    AverageRetries: 0.000000
    AverageRoundTripTime: 0.220254
    NumDroppedPackets: 0
    NumDroppedPackets (Max retries): 0
    Throughput: 8.867039e+04
    Latency: 1.749922e+00
```

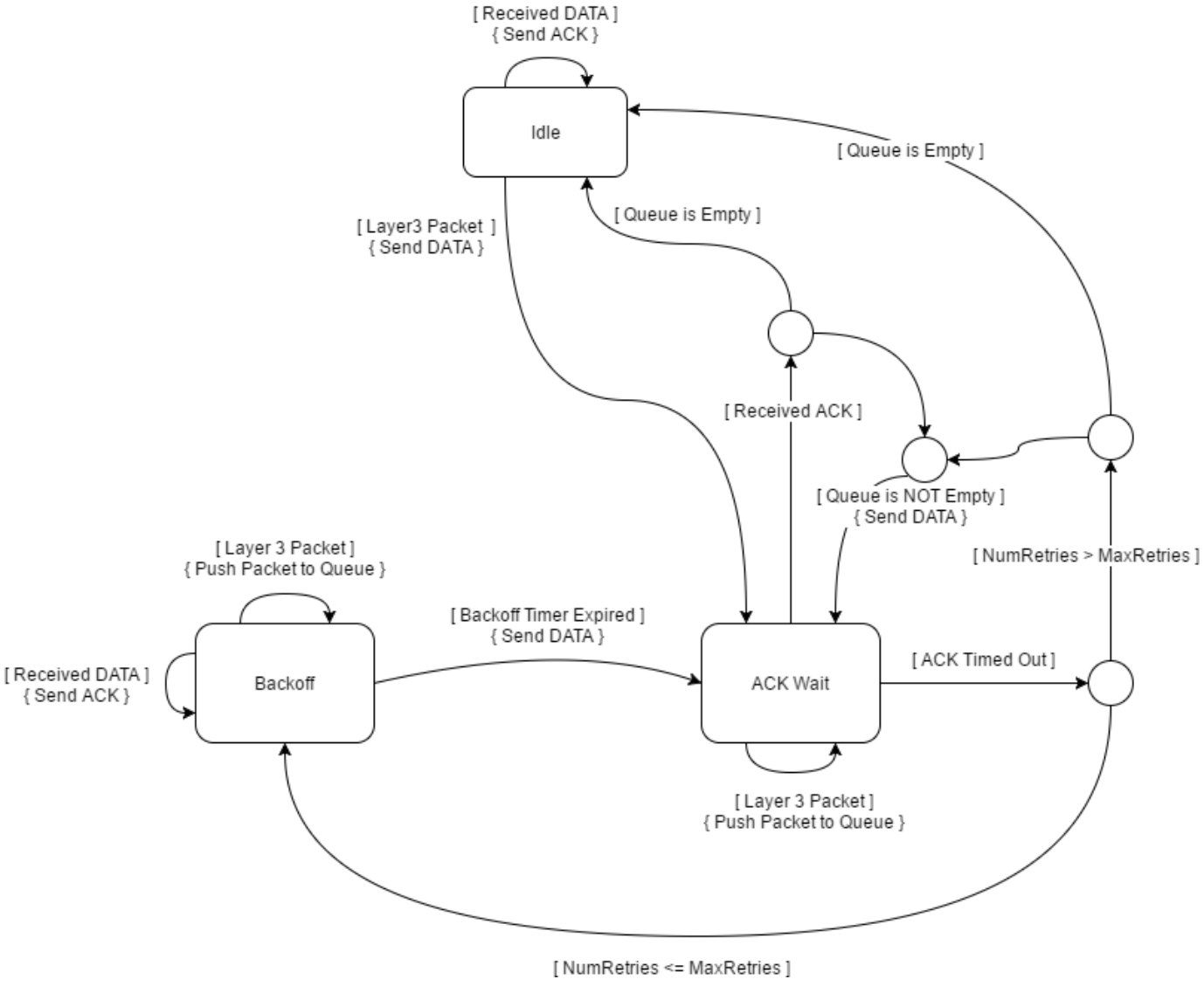
Data Link Layer (Layer 2)

This example implements a Data Link Layer based on the ALOHA random access protocol [2]. The following flow diagram shows how the ALOHA protocol transmits and receives data packets.


Send DATA
Idle
Layer 3 Packet?
No
Yes
Send DATA
Received ACK?
Yes
No
Retries > Max
Yes
No
Random Back off
Receive DATA
Idle
Received DATA?
No
Yes
Send ACK

When Data Link Layer has a Layer 3 packet to transmit, it starts a new session and sends the packet right away using a DATA packet. The algorithm waits for an acknowledgment (ACK) packet. If an ACK is not received before the timeout period, it backs off a random amount of time and sends the DATA packet again. If it fails to receive an ACK after a number of retries, it drops the packet. If during this session, a new Layer 3 packet is received, the Layer 3 packet is put in a first-in-first-out (FIFO) queue. If the FIFO queue is full, packet is dropped.

The algorithm is implemented in the helperPacketizedModemDataLinkLayer helper System object™. The `helperPacketizedModemDataLinkLayer` System object defines a state machine with three states: IDLE, ACK_WAIT, and BACKOFF. The following state machine describes how the data link layer algorithm is implemented in this object. Statements in brackets, [...], and curly braces, {...}, are conditions and actions, respectively. Small circles are passthrough states used to represent multiple conditions.



The original ALOHA protocol uses a hub/star topology. The uplink and downlink utilizes two separate frequency bands. The following example employs a mesh network topology where nodes transmit and receive using the same frequency band.

Modem Structure

The modem code structure executes these six main processing parts:

1. Source Controller
2. Message Generator
3. PHY Decoder
4. Data Link Layer
5. Message Parser
6. PHY Encoder

The Data Link Layer processes outputs of the Message Generator and PHY Decoder, so it must run after those two operations. The Message Parser and PHY Encoder process outputs of the Data Link Layer. This sequence ensures that the modem can receive packets and respond to them in the same time interval. The helperPacketizedModemNode object implements the modem.

Source Controller

The Source Controller generates an enable signal and a random destination address based on the user-selected packet arrival distribution.

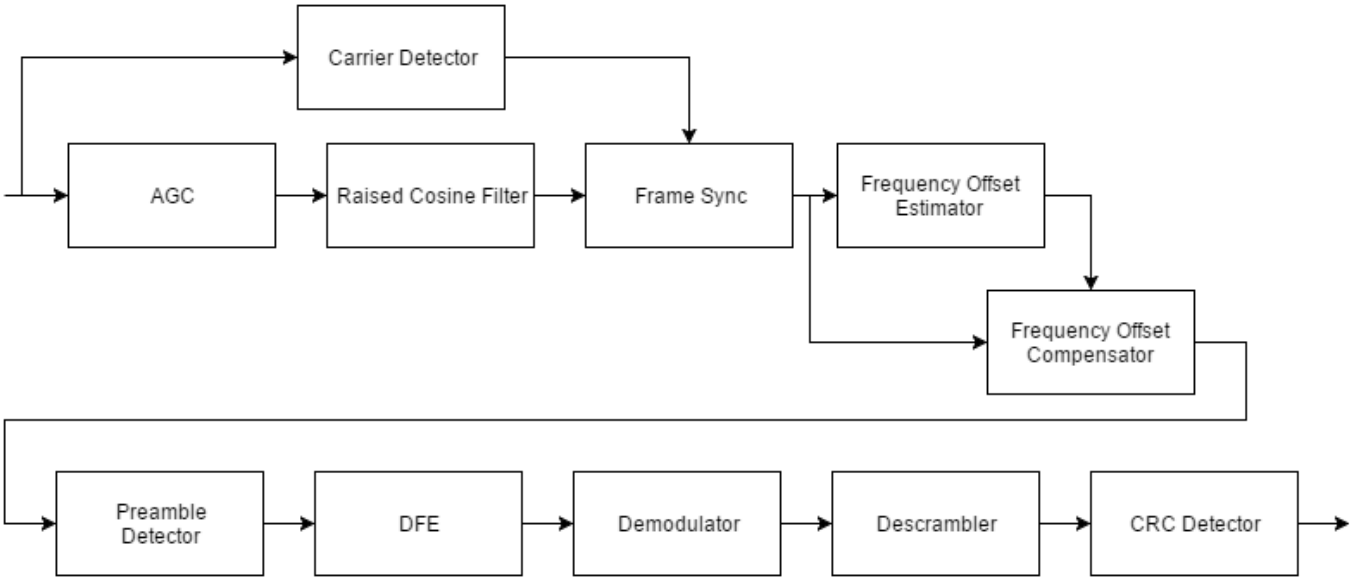
Message Generator

The Message Generator starts creating layer 3 data packets when enabled by the source controller. The packets contain a digitized text message. If the message does not fit into one packet, the generator creates multiple packets. The packet structure is as follows:

- To Address: 8 bits
- From Address: 8 bits
- Packet Number: 16 bits
- Payload: M bits

PHY Decoder

The PHY Decoder receives baseband I/Q samples and creates layer 2 packets. PHY Decoder can correct for amplitude variations using an AGC, frequency offsets with a frequency offset estimator and compensator, and timing skews and multipath using a fractionally spaced decision feedback equalizer (DFE). The block diagram of the physical layer (Layer 1) receiver is as follows:



When data payload size is set to 19530 bits, the total packet length of the modem is 39956 samples. The modem processes `SamplesPerFrame` samples, which is 2000 samples for this example, at each iteration. A smaller `SamplesPerFrame` results in smaller latency but increases the overhead of the modem algorithm. An increased overhead may increase the processing time such that the modem does not run in real-time anymore.

Data Link Layer

Data Link Layer provides a link between two neighboring nodes. It employs the ALOHA-based protocol described in the **Data Link Layer (Layer 2)** section. The packet structure contains these fields:

- Type: 4 bits
- Version: 2 bits
- Reserved: 2 bits
- To Address: 8 bits
- From Address: 8 bits
- Sequence Number: 8 bits
- Time stamp: 32 bits
- Payload: N (= M+32) bits

The data link layer also collects these statistics:

- Number of successful packet transfers, which is defined as the number of successfully received ACK packets
- Average retries
- Average round trip time in seconds
- Number of dropped packets due to layer 3 packet queue being full
- Number of dropped packets due to retries
- Throughput defined as successful data delivery rate in bits per second

- Average latency in seconds defined as the time between the generation of the layer 3 data packet and reception of it at the destination node

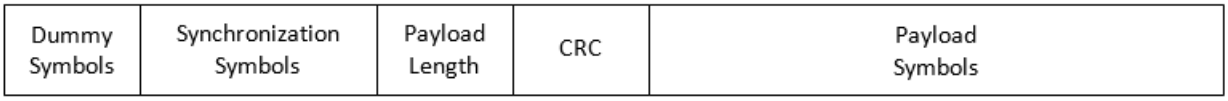
Message Parser

The message parser parses the received layer 2 payload and creates layer 3 packet. The message parser collects these statistics:

- Number of received packets
- Number of received duplicate packets

PHY Encoder

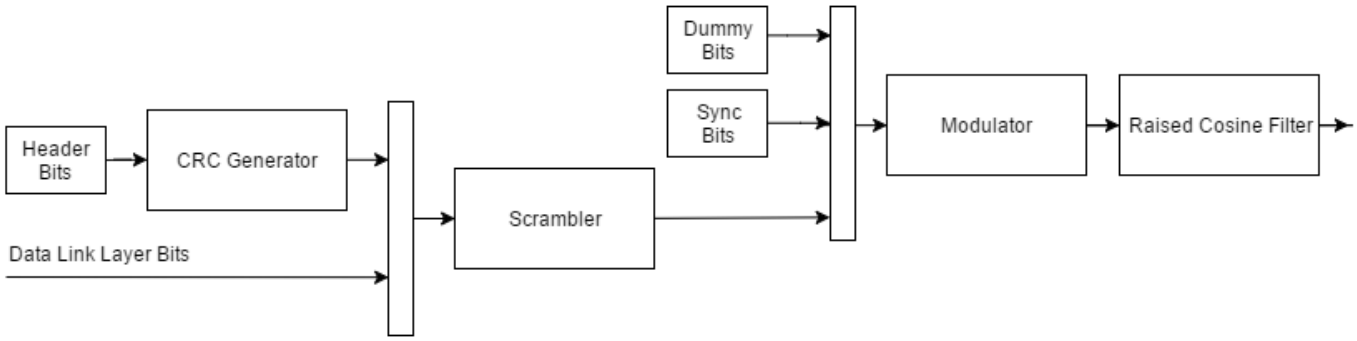
The PHY encoder creates physical layer packets by modulating the layer 2 packets into baseband I/Q samples. The packet structure is shown here.



The dummy symbols are used to train the AGC and for carrier detection. The synchronization symbols are a modulated PN-sequence. The header has these fields:

- Payload length: 16 bits
- CRC: 16 bits

This image shows the block diagram of the physical layer (Layer 1) transmitter.

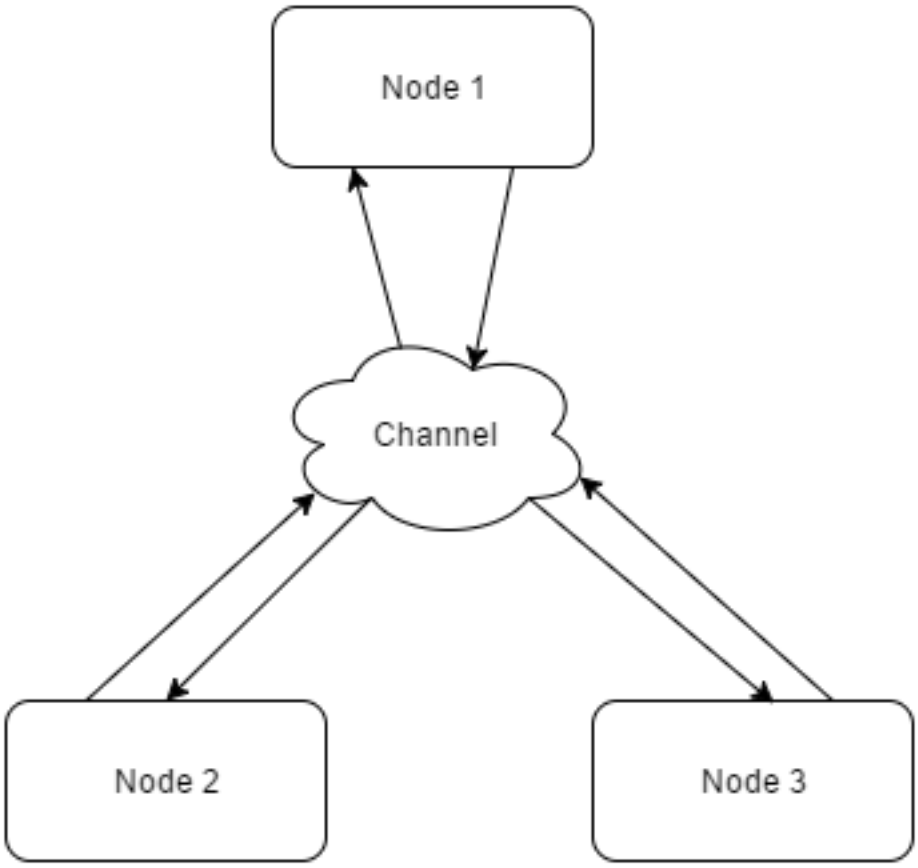


Channel Model

This example simulates a three-node network but any number of nodes can be simulated. The output of each node is passed to the channel simulator. The channel adds baseband signals from all three nodes after imposing the these channel impairments:

- Timing skew
- Frequency offset
- Rician multipath
- AWGN

In addition to these impairments, the signals from neighboring nodes are applied a path loss of 20 dB, while the self-interference is added directly.



Running Using Radios

You can also run this example using radios instead of a simulated channel. The combination of an SDR hardware and a host computer that runs a MATLAB session comprises a node. The following steps show you how to set up a three-node network. This example uses USRP® B200 and B210 radios.

1) Connect a USRP® radio to host computer A, which we will call Node 1. Follow the instruction in “Installation and Setup” (Communications Toolbox Support Package for USRP Radio) to install and setup your host computer for use with USRP® radios. Start a MATLAB session.

2) Set up Node 1 as a transmitter for initialization. The helperPacketizedModemInitializeRadio initializes the connected USRP® radio. Run `helperPacketizedModemInitializeRadio('tx', PLATFORM, ADDRESS, FC, RT)`, where PLATFORM is the type of the USRP® radio, ADDRESS is the serial number or IP address, FC is the center frequency, and RT is run time in seconds. This example uses 915 MHz for the center frequency. Assuming that your radio is a B200 with serial number 'ABCDE', the function call will be `helperPacketizedModemInitializeRadio('tx', 'B200', 'ABCDE', 915e6, 120)`. This function will run the transmitter for 120 seconds. If you need more time to finish the initialization, rerun the command with a longer run time.

3) Repeat step 1 for a second radio and host computer and call this node Node 2.

4) Set up Node 2 as a receiver for initialization. Run `[CDT, MAXGAIN, RXGAIN] = helperPacketizedModemInitializeRadio('rx', PLATFORM, ADDRESS, FC, RT)`. Assuming that your radio is a B210 with serial number '12345', the function call will be `[CDT1, MAXGAIN1,`

`RXGAIN1] = helperPacketizedModemInitializeRadio('rx', 'B210', '12345', 915e6, 120)`. The function will run until it determines the best values for carrier detector threshold (CDT), maximum AGC gain (MAXGAIN), and radio receive gain (RXGAIN) or until RT seconds have elapsed. If the initialization algorithm cannot determine suitable parameters, it may suggest increasing or decreasing the transmitter power and retrying the initialization.

5) Run the same experiment with Node 1 as the receiver and Node 2 as the transmitter to determine best receiver parameters for Node 1. In most cases the channel should be dual and the parameters will be very close.

6) Repeat steps 1-5 for all other pairs of radios, i.e. Node 1 and Node 3, Node 3 and Node 2. Obtain CDT, MAXGAIN, and RXGAIN values for each node. If you get different values for the same node while initializing for different links, choose the maximum values for MAXGAIN and RXGAIN, and minimum of CDT.

7) Start Node 1 by running the helperPacketizedModemRadio helper function. Use the command `helperPacketizedModemRadio(P,RA,NA,DA,FC,CDT,MAXG,RGAIN,D)`, where P is platform, RA is radio address, NA is node address, DA is destination address list, FC is center frequency, CDT is carrier detection threshold, MAXG is maximum AGC gain, RGAIN is radio receiver gain, and D is duration. For example, run as `helperPacketizedModemRadio('B200','ABCDE',1,[2 3],915e6,CDT1,MAXGAIN1,RXGAIN1,120)`.

8) Start Node 2 by running `helperPacketizedModemRadio('B210','12345',2,[1 3],915e6,CDT2,MAXGAIN2,RXGAIN2,120)`.

9) Start Node 3 by running `helperPacketizedModemRadio('B200','A1B2C',3,[1 2],915e6,CDT3,MAXGAIN3,RXGAIN3,120)`.

10) Once the session ends, each node prints out its statistics.

A three network setup operated for two hours. Each node generated packets at a rate of 0.2 packets/second according to a Poisson distribution. The nodes were placed approximately equal distance. One of the links had line-of-sight while other two did not. The following are the results collected on all three nodes. Since the round trip time of a DATA-ACK exchange using the B2xx radios connected over USB can be as high as 800 msec, the average round trip time of the network is greater than 3 sec. The algorithm minimizes packet loss and provides a fair access to the shared channel to all nodes.

```
Node 1:
 NumGeneratedPackets: 1440
 NumReceivedPackets: 1389
 AverageRetries: 0.533738
 AverageRoundTripTime: 3.725093
 NumDroppedPackets: 95
 NumDroppedPackets (Max retries): 23
 Throughput: 5.242823e+03

Node 2:
 NumGeneratedPackets: 1440
 NumReceivedPackets: 1340
 AverageRetries: 0.473157
 AverageRoundTripTime: 3.290775
 NumDroppedPackets: 31
 NumDroppedPackets (Max retries): 9
 Throughput: 5.934772e+03

Node 3:
 NumGeneratedPackets: 1440
```

```
NumReceivedPackets: 1385
AverageRetries: 0.516129
AverageRoundTripTime: 3.558408
NumDroppedPackets: 107
NumDroppedPackets (Max retries): 29
Throughput: 5.488410e+03
```

Discussions

The simulation code from previous sections and the helperPacketizedModemRadio helper function both utilize the helperPacketizedModemNode System object to implement the modem node. In this example, the same code is used to evaluate a system, first using a simulated channel, then using SDR hardware and over-the-air channels.

Even though the code using simulated channels is time-based, the modem node object could be used to run an event-based simulation. This example does not provide an event-based simulation kernel.

Further Exploration

You can vary these parameters to investigate their effect on data link layer performance:

- PacketArrivalRate
- ACKTimeOut
- MaxBackoffTime
- MaxDataRetries
- QueueSize

You can also explore the helper functions for implementation details of the algorithms:

- helperPacketizedModemNode.m
- helperPacketizedSourceController.m
- helperPacketizedModemMessageGenerator.m
- helperPacketizedModemDataLinkLayer.m
- helperPacketizedModemPHYEncoder.m
- helperPacketizedModemPHYDecoder.m
- helperPacketizedModemMessageParser.m
- helperMultiUserChannel.m

You can examine the physical layer only performance using the PacketizedModemPhysicalLayerTxRxExample script.

Selected Bibliography

1 Data Link Layer

2 ALOHANet

Copyright Notice

FM Broadcast Receiver

This example shows how to build an FM mono or stereo receiver using MATLAB® and Communications Toolbox™. You can either use captured signals, or receive signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Communications Toolbox

To receive signals in real time, you also need one of the following hardware:

- RTL-SDR radio and the corresponding software Communications Toolbox Support Package for RTL-SDR Radio
- ADALM-PLUTO radio and the corresponding software Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to the "MATLAB and Simulink Hardware Support for SDR" section of Software-Defined Radio (SDR).

Background

FM broadcasting uses frequency modulation (FM) to provide high-fidelity sound transmission over broadcast radio channels. Pre-emphasis and de-emphasis filters are used to reduce the effect of noise on high audio frequencies. Stereo encoding enables simultaneous transmission of both left and right audio channels over the same FM channel [1].

Run the Example

Type `FMReceiverExample` in the MATLAB Command Window or click the ‘Open example’ button to open and run the example. You need to enter the following information:

1. Reception duration in seconds
2. Signal source (captured data, RTL-SDR radio or ADALM-PLUTO radio)
3. FM channel frequency

The example plays the received audio over your computer's speakers.

NOTE: This example utilizes a center frequency that is outside the default PlutoSDR tuning range. Click configurePlutoRadio('AD9364') to use your ADALM-PLUTO radio outside the qualified tuning range.

Receiver Structure

The FM Broadcast Demodulator Baseband System object™ converts the input sampling rate of the 228 kHz to 45.6 kHz, the sampling rate for your host computer's audio device. According to the FM broadcast standard in the United States, the de-emphasis lowpass filter time constant is set to 75 microseconds. This example processes the received mono signals. The demodulator can also process stereo signals.

To perform stereo decoding, the FM Broadcast Demodulator Baseband object uses a peaking filter which picks out the 19 kHz pilot tone from which the 38 kHz carrier is created. Using the resulting

carrier signal, the FM Broadcast Demodulator Baseband block downconverts the L-R signal, centered at 38 kHz, to baseband. Afterwards, the L-R and L+R signals pass through a 75 microsecond de-emphasis filter . The FM Broadcast Demodulator Baseband block separates the L and R signals and converts them to the 45.6 kHz audio signal.

Example Code

The receiver asks for user input and initializes variables. Then, it calls the signal source and FM broadcast receiver in a loop. The loop also keeps track of the radio time using the frame duration and lost samples reported by the signal source.

The latency output of the signal source is an indication of when the samples were actually received and can be used to determine how close to real time the receiver is running. A latency value of 1 and a lost samples value of 0 indicates that the system is running in real time. A latency value of greater than one indicates that the receiver was not able to process the samples in real time. Latency is reported in terms of the number of frames. It can be between 1 and 128. If latency is greater than 128, then samples are lost.

```
% Request user input from the command-line for application parameters
userInput = helperFMUserInput;

% Calculate FM system parameters based on the user input
[fmRxParams,sigSrc] = helperFMConfig(userInput);

% Create FM broadcast receiver object and configure based on user input
fmBroadcastDemod = comm.FMBroadcastDemodulator(...
    'SampleRate', fmRxParams.FrontEndSampleRate, ...
    'FrequencyDeviation', fmRxParams.FrequencyDeviation, ...
    'FilterTimeConstant', fmRxParams.FilterTimeConstant, ...
    'AudioSampleRate', fmRxParams.AudioSampleRate, ...
    'Stereo', false);

% Create audio player
player = audioDeviceWriter('SampleRate',fmRxParams.AudioSampleRate);

% Initialize radio time
radioTime = 0;

% Main loop
while radioTime < userInput.Duration
  % Receive baseband samples (Signal Source)
  if fmRxParams.isSourceRadio
      if fmRxParams.isSourcePlutoSDR
          rcv = sigSrc();
          lost = 0;
          late = 1;
      else
          [rcv,~,lost,late] = sigSrc();
      end
  else
    rcv = sigSrc();
    lost = 0;
    late = 1;
  end

  % Demodulate FM broadcast signals and play the decoded audio
  audioSig = fmBroadcastDemod(rcv);
```

```
  player(audioSig);

  % Update radio time. If there were lost samples, add those too.
  radioTime = radioTime + fmRxParams.FrontEndFrameTime + ...
    double(lost)/fmRxParams.FrontEndSampleRate;
end

% Release the audio and the signal source
release(sigSrc)
release(fmBroadcastDemod)
release(player)
```

Further Exploration

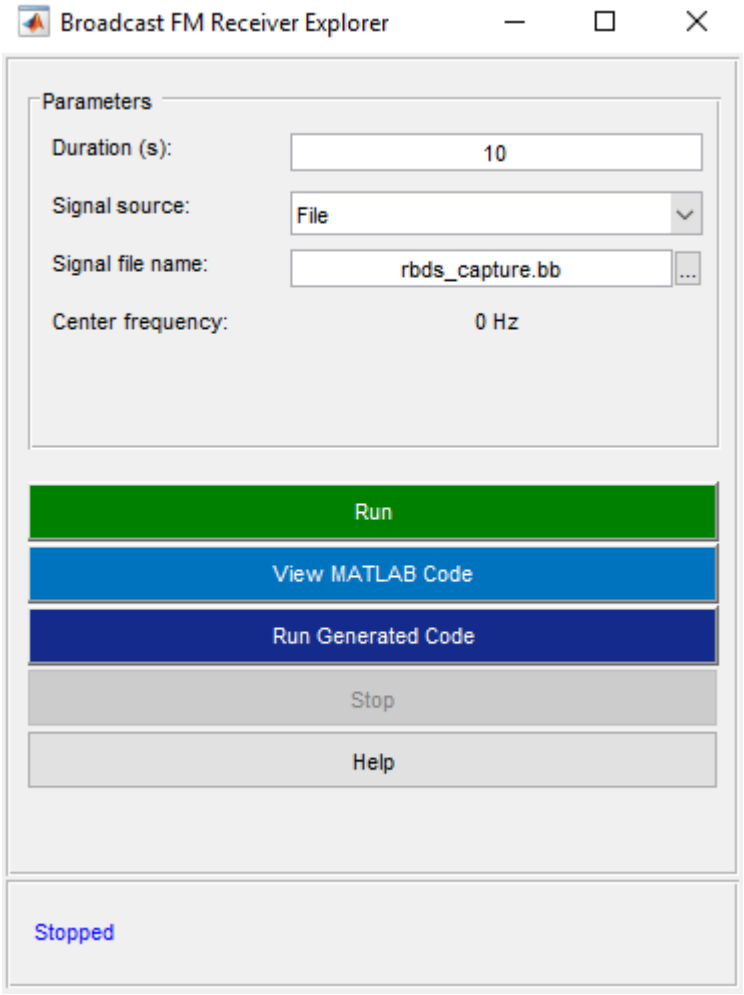
To further explore the example, you can vary the center frequency of the RTL-SDR radio or ADALM-PLUTO radio and listen to other radio stations.

You can set the Stereo property of the FM demodulator object to true to process the signals in stereo fashion and compare the sound quality.

You can explore following function for details of the system parameters:

- helperFMConfig.m

You can further explore the FM signals using FMReceiverExampleApp user interface. This app allows you to select the signal source and change the center frequency of the RTL-SDR radio or ADALM-PLUTO radio. To launch the app, type FMReceiverExampleApp in the MATLAB Command Window. This user interface is shown in the following figure



Selected Bibliography

1 https://en.wikipedia.org/wiki/FM_broadcasting

RDS/RBDS and RadioText Plus (RT+) FM Receiver

This example shows how you can use MATLAB® and the Communications Toolbox™ to extract program or song information from FM radio stations using the RDS or RBDS standard and, optionally, the RadioText Plus (RT+) standard. You can either use captured signals or receive signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio.

Required Hardware and Software

To run this example using captured signals, you need the Communications Toolbox™. To receive signals in real time, you also need one of the following hardware:

- RTL-SDR radio and the corresponding software Communications Toolbox Support Package for RTL-SDR Radio
- ADALM-PLUTO radio and the corresponding software Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of Software Defined Radio (SDR).

Background

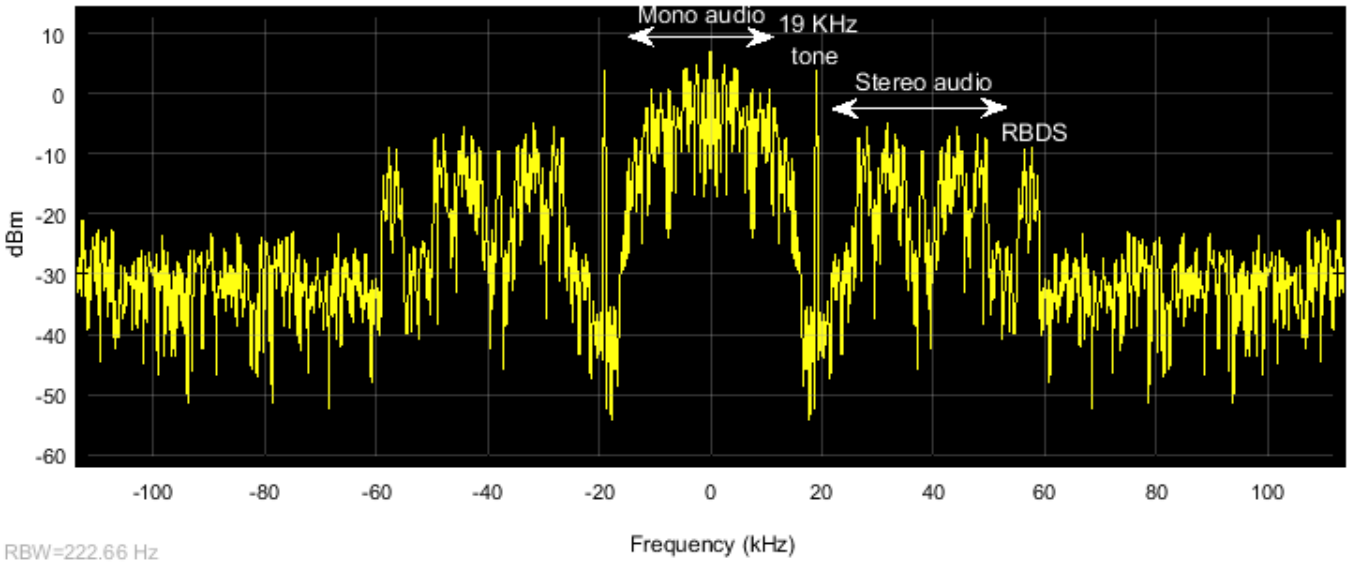
RBDS and RDS are very similar standards specifying how to supplement FM radio signals with additional information. RBDS is used in North America, while RDS was originally used in Europe and evolved to an international standard. RBDS and RDS comprise 3 layers:

- Physical Layer (Layer 1)
- Data-link Layer (Layer 2)
- Session and presentation Layer (Layer 3)

Physical Layer (Layer 1)

The RDS/RBDS PHY decoder receives the captured signal from a file or the live signal from the radio and performs the following steps:

- **FM demodulation**: Once the FM signal is demodulated, the RDS/RBDS signal resides at the 57 kHz +/- 2.4 kHz band:

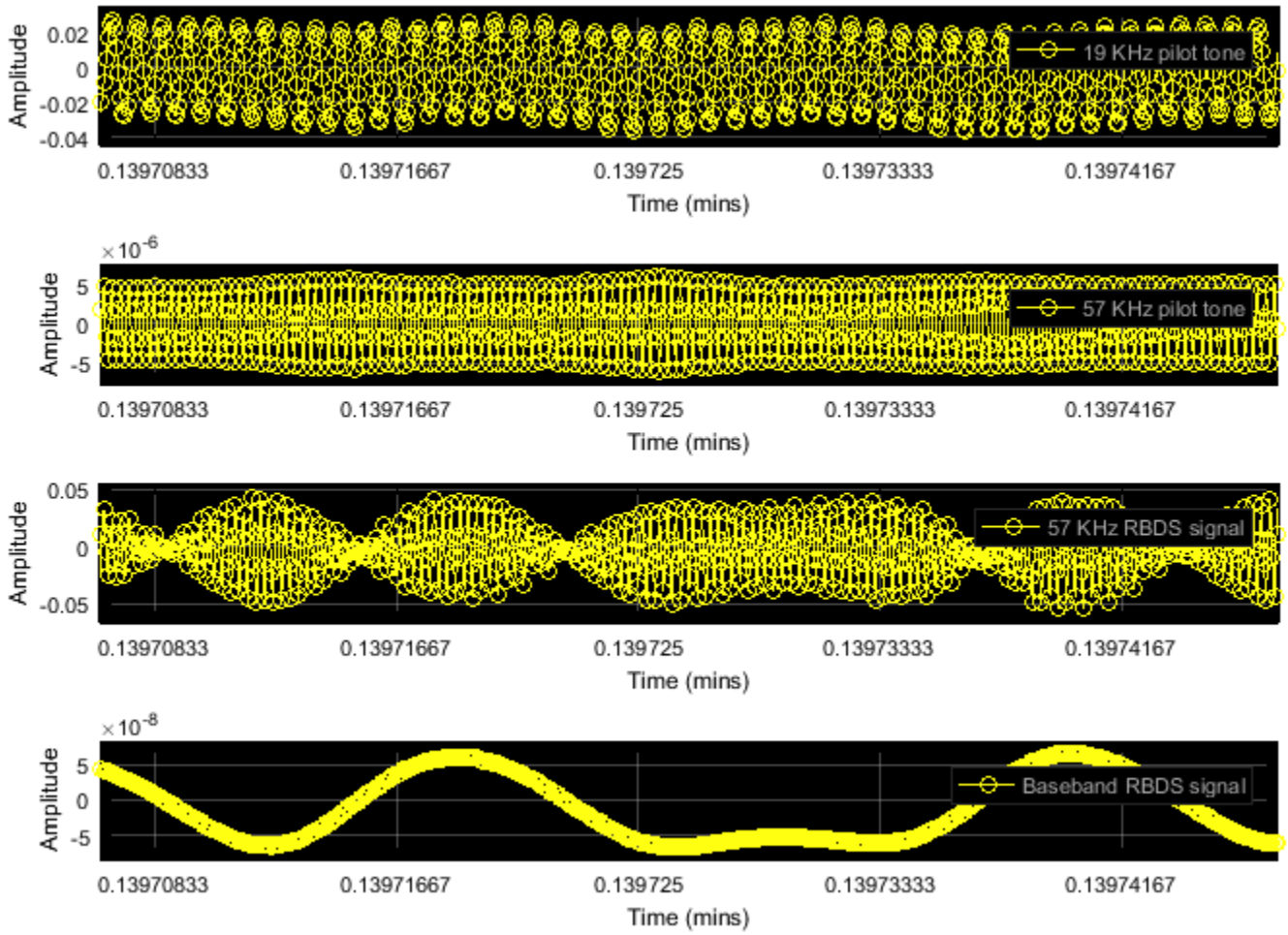


Be aware that the RDS and RBDS signals are transmitted with relatively low power, so it is not always visible in the FM spectrum as in the above figure.

FM signals contain a pilot tone at 19 kHz, which can be used as a phase and frequency reference for coherent demodulation of the RDS/RBDS signal at 57 kHz and the stereo audio at 38 kHz. Pilot tones at 38 kHz and 57 kHz can be generated by doubling and tripling the frequency of the 19 kHz pilot tone [2].

Processing steps for coherent demodulation of the RDS/RBDS signal are:

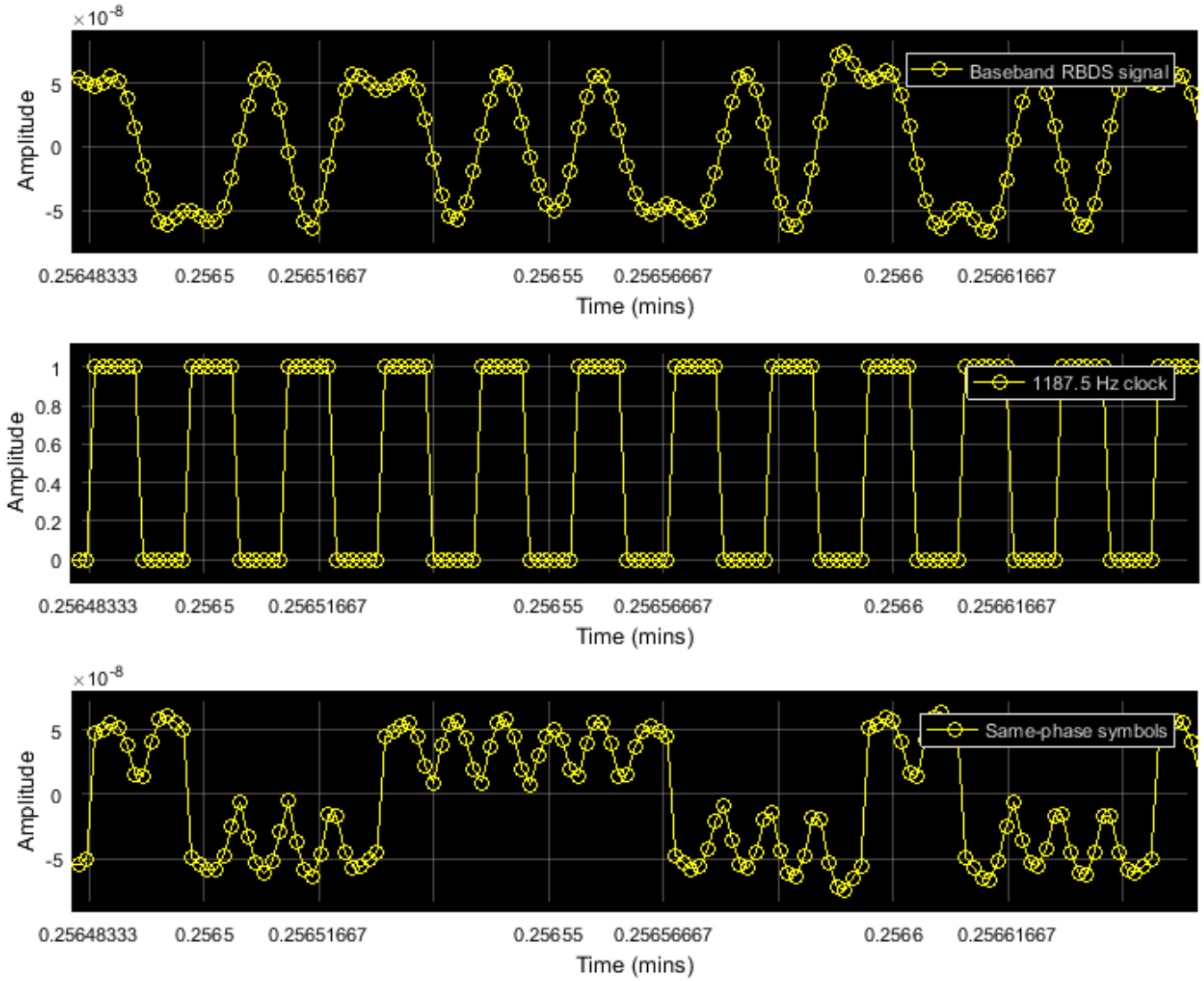
- **Bandpass filtering**: The PHY receiver conducts bandpass filtering at 19 kHz and 57 kHz, to isolate the pilot tone and the RDS/RBDS signal, respectively.
- **Frequency tripling**: Raise the complex representation of the 19 kHz pilot tone to the 3rd power to triple its frequency and obtain a 57 kHz pilot tone.
- **AM Demodulation**: RDS and RBDS symbols are generated at an 1187.5 Hz rate and are AM-modulated to a 57 kHz carrier. The 57 kHz RDS/RBDS signal can be coherently demodulated with a 57 kHz carrier that is locked in frequency and phase. Typically, the frequency-tripled 19 kHz pilot tone suffices for coherent demodulation. The next figures show the 19 kHz and 57 kHz pilot tones, the 57 kHz RDS/RBDS signal, and the AM-demodulated baseband RDS/RBDS signal.



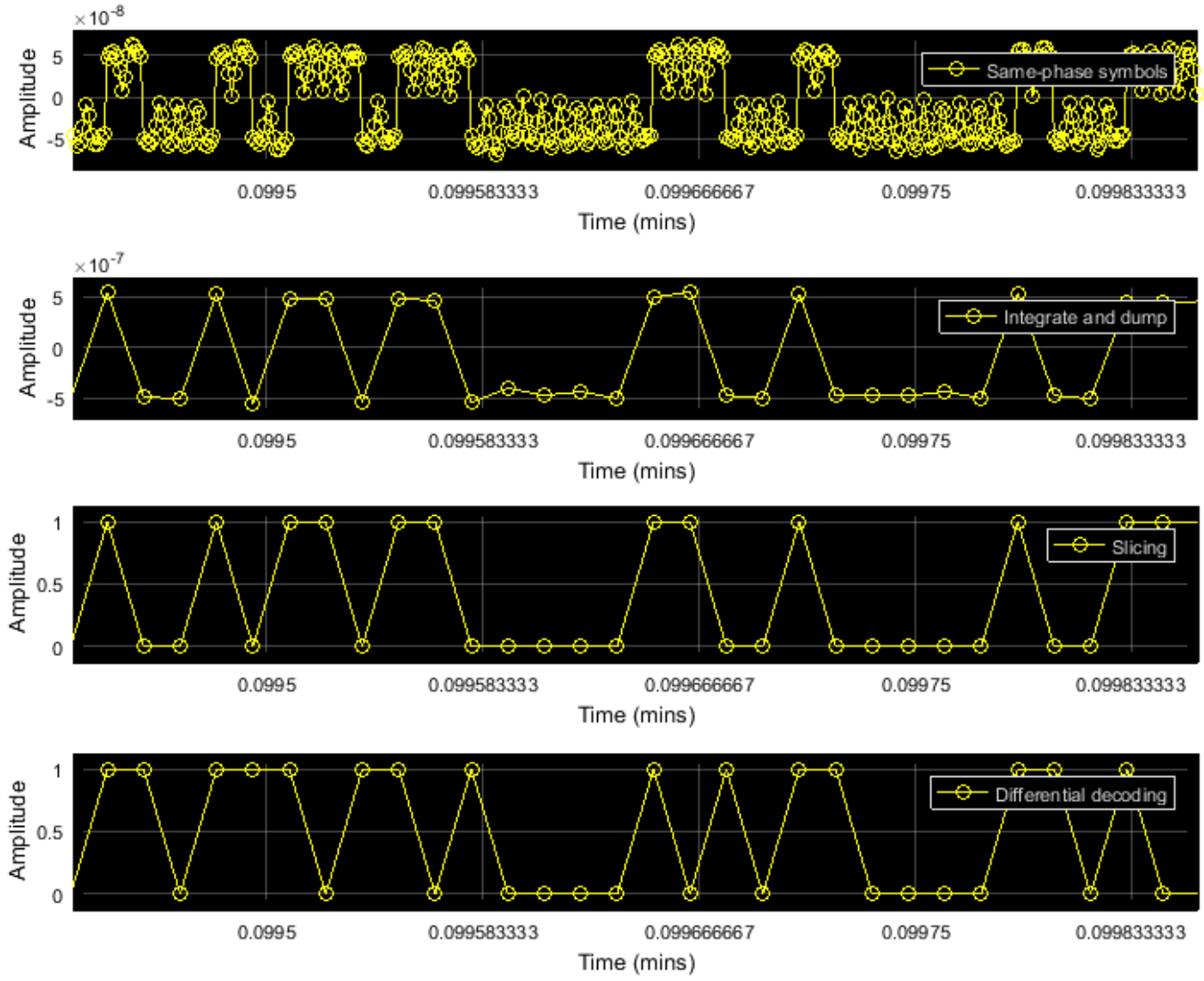
At the same time, there exist several FM stations whose 57 kHz RDS/RBDS signal exhibits a time-varying phase offset from the 19 kHz pilot tone and its frequency-tripled version. The PHY receiver contains a Costas loop to compensate for such time-varying phase offsets.

- **Costas loop**: The Costas loop performs 2 orthogonal AM demodulations, one demodulation with a 57 kHz sine and another with a 57 kHz cosine. The sampling rate of the received signal is carefully chosen as 228 kHz, which provides 4 samples per 57 kHz cycle. Therefore, a one sample delay of the 57 kHz pilot tone results to a one quarter wavelength phase offset, and allows us to generate a cosine wave from a sine wave. The sine-demodulated signal corresponds to the coherent demodulation output. The cosine-demodulated signal is used for detection of phase error. The products of the 57 kHz RDS/RBDS signal with the sine/cosine waves are low-pass filtered with the filter specified in Sec. 1.7 of [1]. The product of the two filter outputs is an error signal. The larger it is, the more the 19 kHz pilot tone is delayed to behave more like the cosine-based demodulator.
- **Clock extraction**: To perform biphase symbol decoding, a clock matching the RDS/RBDS symbol rate of 1187.5 Hz is extracted from the 19 kHz pilot tone. Note, 1187.5 Hz x 16 = 19 kHz. To account for frequency offsets, frequency division is used to extract the clock from the 19 kHz pilot tone. Since the frequency division operation provides multiple correct answers, the baseband RDS/RBDS signal serves as training data that aid in the determination of the desired output.

- **Biphase symbol decoder**: RDS and RBDS use bi-phase-level (bi-ϕ-L) coding, which is commonly known as Manchester coding. In each clock cycle, the RDS/RBDS symbol takes two opposite amplitude values, either a positive followed by a negative, or a negative followed by a positive. The biphase symbol decoder negates the second amplitude level, so that each symbol holds the same amplitude level throughout the entire clock cycle. The new clock-wide amplitude level corresponds to the symbol's bit representation. The following two screenshots correspond to the waveforms #1-6 in Figure 2 of [1].



To obtain each symbol's bit value, the waveform is integrated over each clock cycle, and the outcome is compared to zero (slicer).



- **Differential decoding**: Finally, the bits are differentially decoded to revert the differential encoding at the transmitter.

Data-link Layer (Layer 2)

Layer 2 is implemented using the RBDSDataLinkDecoder System object™. This layer is responsible for synchronization and error correction.

The bit output of the PHY layer is logically organized in 104-bit groups comprising four 26-bit blocks. Each block contains a 16-bit information word and 10 parity bits (see Figure 8 in [1]). A distinct 10-bit offset word is modulo-2 added to the parity bits of each block.

- **Synchronization**: Initially, block and group boundaries are sought exhaustively using a sliding window of 104 bits. For each 104-bit window, the 4 offset words are sought at the last 10 bits of each 26-bit block. An offset word is identified if no bit errors are detected in its block. Once the offset words are identified, group-level synchronization is attained and the exhaustive sliding-window processing stops. Subsequently, the next 104 bits will be treated as the next group.

If future groups contain bit errors and the offset words cannot be identified at their expected position, synchronization may be lost. In this case, Layer 2 first examines the possibility of 1-bit synchronization slips, exploiting the fact that the first information word (16 bits) is always the same for all bit groups. If the first information word is found dislocated by 1 bit (either leftward or rightward), synchronization is retained and the group boundaries are adjusted accordingly. If bit errors persist for 25 group receptions and at the same time synchronization cannot be reestablished using such leftward/rightward 1-bit shifts, then synchronization is lost and Layer 2 re-enters the exhaustive, sliding-window-based search for synchronization.

- **Error correction**: The RDS/RBDS error correction code is a (26, 16) cyclic code shortened from (341, 331). The error correction implementation uses the shift-register scheme described in Annex B of [1].

Session and Presentation Layer (Layer 3)

Layer 2 removes the parity/offset bits, therefore Layer 3 receives groups of 64-bits, comprising four 16-bit blocks. There exist up to 32 different group types, each labeled with a number from 0 to 15 and the letter 'A' or 'B', for example, 0B, 2A, 3A. The format of each group can be fixed or it can be abstract if this group is allocated for an Open Data Application (ODA, see list in [3]).

Layer 3 is implemented using the RBDSSessionDecoder System object. This object supports decoding of the 0A, 0B, 2A, 2B, 3A, 4A, 10A fixed-format group types.

- 0A and 0B convey an 8-character string, which typically changes in a scrolling-text fashion.
- 2A and 2B convey longer 64- or 32-character strings.
- 3A registers ODAs and specifies their dedicated abstract-format group type.
- 4A conveys the system time.
- 10A further categorizes the program type (e.g., 'Football' for 'Sports' program type).

For ODAs, the RDS/RBDS receiver supports decoding of RadioText Plus (RT+). This ODA can break down the long 32- or 64-character string from group types 2A or 2B into two specific content types (for example, artist and song).

Registering ODA Implementations: RadioText Plus (RT+)

The RDS/RBDS receiver is extensible. ODA implementations can be specified using the registerODA function of the RBDSSessionDecoder System object. This function accepts the hexadecimal ID of the ODA (ODA IDs can be found in [3]), and handles to the functions that process the main ODA group type, as well as the ODA-specific part of the 3A group type. For example, the sessionDecoder RBDSSessionDecoder object can be extended for RadioText Plus (RT+) using this code:

```
rtID = '4BD7'; % hexadecimal ID of RadioText Plus (RT+)
registerODA(sessionDecoder, rtID, @RadioTextPlusMainGroup, @RadioTextPlus3A);
```

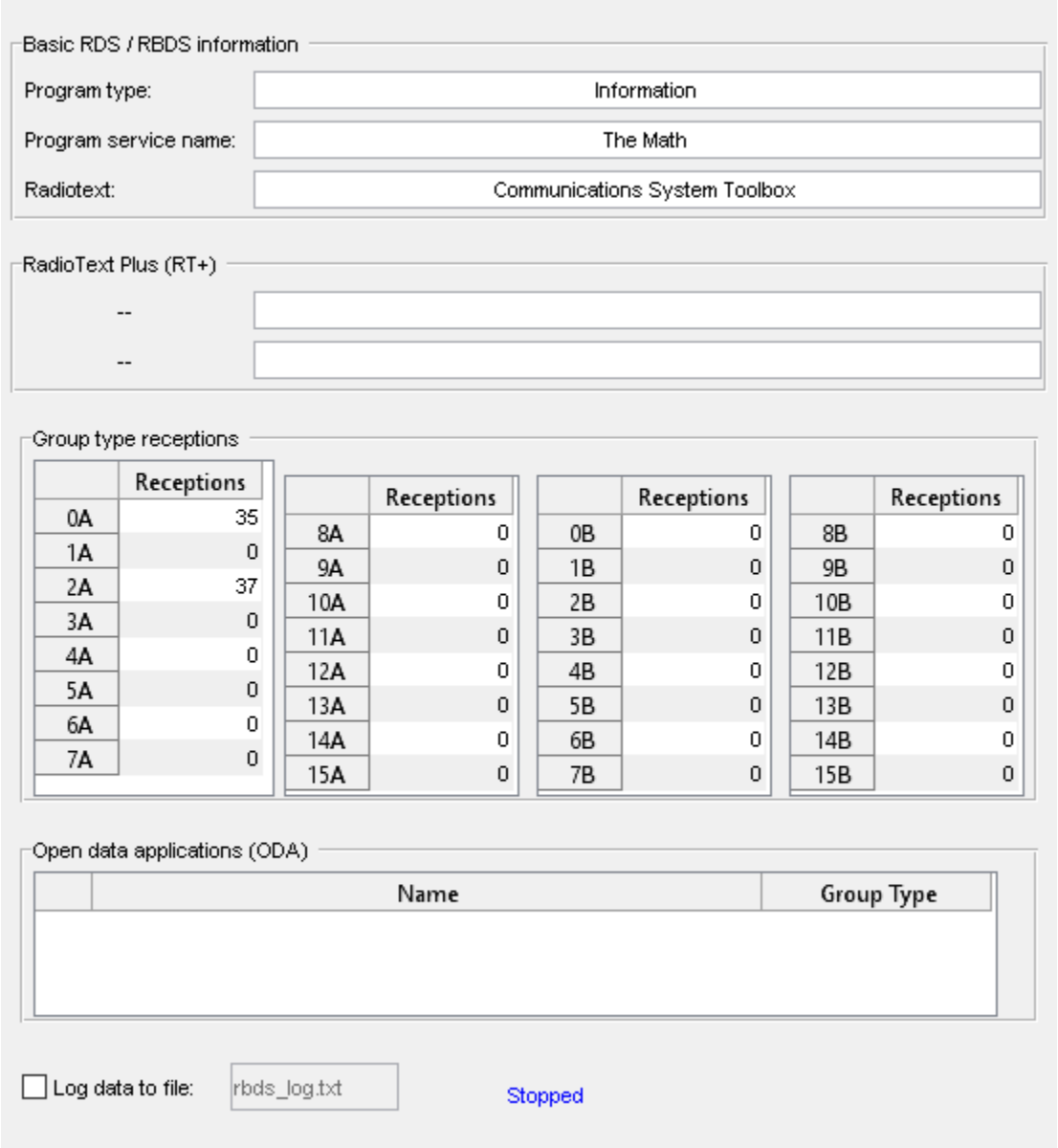
Run the Example Code

Type RBDSExample in the MATLAB Command Window or click this link to run the example.

```
% Set RDS/RBDS system parameters
userInput = helperRBDSInit();
userInput.Duration = 10.8;
userInput.SignalSource = 'File';
userInput.SignalFilename = 'rbds_capture.bb';
% userInput.SignalSource = 'RTL-SDR';
```

```
% userInput.CenterFrequency = 98.5e6;
% userInput.SignalSource = 'ADALM-PLUTO';
% userInput.CenterFrequency = 98.5e6;


[rbdsParam, sigSrc] = helperRBDSConfig(userInput);

% Create FM broadcast receiver object and configure based on RDS/RBDS parameters
fmBroadcastDemod = comm.FMBroadcastDemodulator(...
    'SampleRate',         rbdsParam.FrontEndSampleRate, ...
    'FrequencyDeviation', rbdsParam.FrequencyDeviation, ...
    'FilterTimeConstant', rbdsParam.FilterTimeConstant, ...
    'AudioSampleRate',    rbdsParam.AudioSampleRate, ...
    'Stereo', true);

% Create audio player
player = audioDeviceWriter('SampleRate', rbdsParam.AudioSampleRate);

% Layer 2 object
datalinkDecoder = RBDSDataLinkDecoder();

% Layer 3 object
sessionDecoder  = RBDSSessionDecoder();
% register processing implementation for RadioText Plus (RT+) ODA:
rtID = '4BD7';
registerODA(sessionDecoder, rtID, @RadioTextPlusMainGroup, @RadioTextPlus3A);

% Create the data viewer object
viewer = helperRBDSViewer();

% Start the viewer and initialize radio time
start(viewer)
radioTime = 0;

% Main loop
while radioTime < rbdsParam.Duration
  % Receive baseband samples (Signal Source)
  rcv = sigSrc();

  % Demodulate FM broadcast signals and play the decoded audio
  audioSig = fmBroadcastDemod(rcv);
  player(audioSig);

  % Process physical layer (Layer 1)
  bitsPHY = RBDSPhyDecoder(rcv, rbdsParam);

  % Process data-link layer (Layer 2)
  [enabled, iw1, iw2, iw3, iw4] = datalinkDecoder(bitsPHY);

  % Process session and presentation layer (Layer 3)
  outStruct = sessionDecoder(enabled, iw1, iw2, iw3, iw4);

  % View results packet contents (Data Viewer)
  update(viewer, outStruct);

  % Update radio time
  radioTime = radioTime + rbdsParam.FrameDuration;
end
```

```
% Stop the viewer and release the signal source and audio writer
stop(viewer);
release(sigSrc);
release(player);
```



Viewing Results

The above screenshot illustrates the graphical display of the processed RDS/RBDS data.

- **Basic RDS/RBDS information**:

1 The first field corresponds to the program type, which is conveyed by the second information word of all group types. If 10A group types are received, the first field also provides further characterization, such as, Sports \ **Football**.

2 The second field illustrates the 8-character text conveyed by 0A/0B groups.

3 The third field illustrates the longer 32/64-character text conveyed by 2A/2B group types.

- **RadioText Plus (RT+)**: This section is used if any 3A groups indicate that the RadioText Plus (RT +) ODA uses an abstract-format group type, e.g., 11A. Then, upon receptions of this abstract group type, the 32/64-character text conveyed by groups 2A/2B will be split to two substrings. Moreover, the two labels will be updated to characterize the substrings (such as Artist and Song).

- **Group type receptions**: The tables act as a histogram illustrating which group types have been received from a station and with what frequency. As a result, users may want to look at the logged data for further information that is not depicted in the graphical viewer (specifically, system time in 4A, alternate frequencies in 0A etc.).

- **Open data applications (ODA)**: If any 3A group types are received, then the list of encountered ODAs is updated with the ODA name and their dedicated group type.

Further Exploration

You can further explore RDS/RBDS signals using the RBDSExampleApp user interface. You can launch it by clicking at this link or by typing *RBDSExampleApp* in the command window:

This user interface allows you to:

- Select the source of the signal (capture file or RTL-SDR or ADALM-PLUTO)

- Specify the station frequency (for RTL-SDR or ADALM-PLUTO)

- Run Layers 1 and 2 of the RDS/RBDS receiver though generated C code. These are the most time-consuming parts of the RDS/RBDS chain and generating code can help you achieve real-time processing.

- Disable audio playback

- Open scopes, such as a Spectrum Analyzer and Time Scopes, that analyze the received signal and illustrate the decoding process. Enabling scopes requires extra computational effort and may preclude real-time decoding. In this case, RDS/RBDS decoding may only be successful for captured signals loaded from a file.

Moreover, you can enable the 'Log data to file' checkbox in order to log further fields from all group types.

You can also explore the implementation of the following functions and System objects:

- RBDSPhyDecoder.m
- RBDSCostasLoop.m
- RBDSDataLinkDecoder.m
- RBDSSessionDecoder.m

Selected Bibliography

1 National Radio Systems Committee, United States RBDS standard, April 1998

2 Der, Lawrence. "Frequency Modulation (FM) Tutorial". Silicon Laboratories Inc.

3 National Radio Systems Committee, List of ODA Applications in RDS

4 RadioText Plus (RT+) Specification

5 Joseph P. Hoffbeck, "Teaching Communication Systems with Simulink® and the USRP", ASEE Annual Conference, San Antonio, TX, June 2012

FRS/GMRS Walkie-Talkie Receiver

This example shows how to build a walkie-talkie receiver using MATLAB® and Communications Toolbox™. The specific radio standard this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System). You can use simulated signals, captured signals, or received signals from a commercial walkie-talkie using the Communications Toolbox Support Package for RTL-SDR Radio.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found in the reference list below. Operation in other countries may or may not work.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Communications Toolbox

To receive signals in real time, you also need the following hardware:

- RTL-SDR radio
- Walkie-talkie

and the following software

- Communications Toolbox Support Package for RTL-SDR Radio

For a full list of Communications Toolbox supported SDR platforms, refer to the "MATLAB and Simulink Hardware Support for SDR" section of Software-Defined Radio (SDR).

Background

Walkie-talkies provide a subscription-free method of communicating over short distances. Although their popularity has been diminished by the rise of cell phones, walkie-talkies are still useful when lack of reception or high per-minute charges hinders cell phone use.

Modern walkie-talkies operate on the FRS/GMRS standards. Both standards use frequency modulation (FM) at 462 or 467 MHz, which is in the UHF (Ultra High Frequency) band.

Run the Example

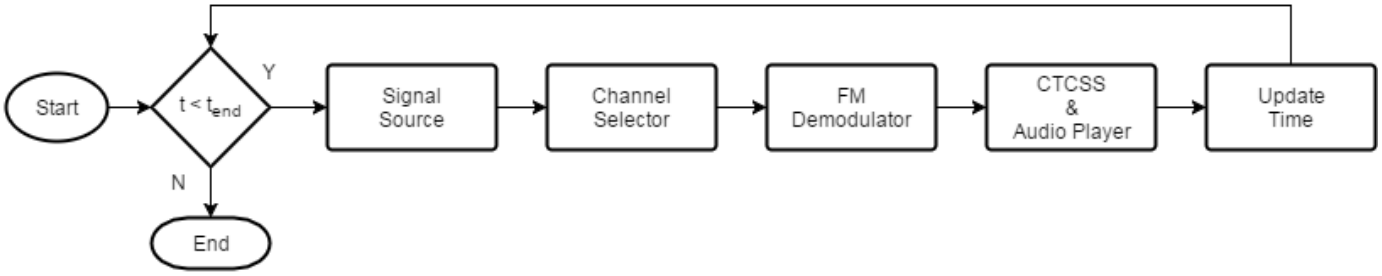
Type `FRSReceiverExample` in the MATLAB Command Window or click the 'Open example' button to open and run the example. You need to enter the following information:

1. Reception duration in seconds
2. Signal source (simulated signal, captured signal or RTL-SDR radio)
3. Channel number (1-14)
4. CTCSS code (1-38, 0 no CTCSS filtering)
5. Detection threshold for received signal

The example plays the received audio over your computer's speakers.

Receiver Structure

The following block diagram summarizes the receiver code structure. The processing has four main parts: Signal Source, Channel Selector, FM Demodulator, and CTCSS processing.



Signal Source

This example can use three signal sources:

1. ''Simulated Signal'': Simulated FRS/GMRS signal at 240e3 samples/sec
2. ''Captured Signal'': Over-the-air signals written to a file and sourced from a baseband file reader object at 240e3 samples/sec
3. ''RTL-SDR Radio'': RTL-SDR radio at 240e3 samples/sec

If you choose ''RTL-SDR Radio'' as the signal source, this application will search your computer for RTL-SDR radios and ask you to choose one of them as the signal source.

Channel Selector

The receiver removes the DC component and applies a variable gain to the received signal to obtain an approximately known amplitude signal with reduced interference. The receiver then applies a low pass channel separation filter to reduce the signals from adjacent channels. The gap between adjacent channels is 25 kHz, which means the baseband bandwidth is, at most, 12.5 kHz. Thus, we choose the cutoff frequency to be 10 kHz.

Next, a channel selector computes the average power of the filtered signal. If it is greater than a threshold (set to a default of 10%), the channel selector determines that the received signal is from the correct channel and allows the signal to pass through. In the case of an out-of-band signal, although the channel separation filter reduces its magnitude, it is still FM modulated and the modulating signal will be present after FM demodulation. To completely reject such a signal, the channel selector outputs all zeros.

FM Demodulator

This example uses the FM Demodulator Baseband System object™ whose sample rate and maximum frequency deviation are set to 240 kHz and 2.5 kHz, respectively.

CTCSS

First, a decimation filter converts the sampling rate from 240 kHz to 8 kHz. This rate is one of the native sampling rates of your host computer's output audio device. Then, the CTCSS decoder computes the power at each CTCSS tone frequency using the Goertzel algorithm and outputs the code with the largest power. The Goertzel algorithm provides an efficient method to compute the

frequency components at predetermined frequencies, that is, the tone code frequencies used by FRS/GMRS.

The script compares the estimated received code with the preselected code. If the two codes match, the signals are passed to the audio device. When the preselected code is zero, it indicates no squelch system is used and the decision block passes the signal at the channel to the audio device no matter which code is used.

Finally, a high pass filter with a cutoff frequency of 260 Hz filters out the CTCSS tones, which have a maximum frequency of 250 Hz. Use an audioDeviceWriter System object™ to play the received signals through your computer's speakers. If you do not hear any sound, select another device using the DeviceName property of the audio device writer object, audioPlayer.

Example Code

The receiver asks for user input and initializes variables. Then it calls the signal source, channel selector, FM demodulator, and CTCSS processor in a loop. The loop also keeps track of the radio time using the frame duration and lost samples reported by the signal source.

The latency output of the signal source is an indication of when the samples were actually received and can be used to determine how close to real time the receiver is running. A latency value of 1 and a lost samples value of 0 indicates that the system is running in real-time. A latency value of greater than one indicates that the receiver was not able to process the samples in real time. Latency is reported in terms of the number of frames. It can be between 1 and 128. If latency is greater than 128, then samples are lost.

```
% Request user input from the command-line for application parameters
userInput = helperFRSReceiverUserInput;

% Calculate FRS receiver parameters based on the user input
[frsRxParams,sigSrc] = helperFRSReceiverConfig(userInput);

% Create channel selector components
dcBlocker = dsp.DCBlocker('Algorithm', 'Subtract mean');
agc = comm.AGC;
channelFilter = frsRxParams.ChannelFilter;

% Create FM demodulator
fmDemod = comm.FMDemodulator(...
  'SampleRate', frsRxParams.FrontEndSampleRate, ...
  'FrequencyDeviation', frsRxParams.FrequencyDeviation);

% Create CTCSS and audio output components
decimator = dsp.FIRDecimator(...
  frsRxParams.DecimationFactor, ...
  frsRxParams.DecimationNumerator);
decoder = helperFRSCTCSSDecoder( ...
  'MinimumBlockLength', frsRxParams.CTCSSDecodeBlockLength, ...
  'SampleRate', frsRxParams.AudioSampleRate);
audioFilter = frsRxParams.AudioFilter;
audioPlayer = audioDeviceWriter(frsRxParams.AudioSampleRate);

% Initialize radio time
radioTime = 0;

% Main loop
```

```
while radioTime < userInput.Duration
  % Receive baseband samples (Signal Source)
  if frsRxParams.isSourceRadio
    [rcv,~,lost,late] = sigSrc();
  else
    rcv = sigSrc();
    lost = 0;
    late = 1;
  end

  % Channel selector
  rcv = dcBlocker(rcv);
  outAGC = agc(rcv);
  outChanFilt = channelFilter(outAGC);
  rxAmp = mean(abs(outChanFilt));
  if rxAmp > frsRxParams.DetectionThreshold
    x = outChanFilt;
  else
    x = complex(single(zeros(frsRxParams.FrontEndSamplesPerFrame, 1)));
  end

  % FM demodulator
  y = fmDemod(x);

  % CTCSS decoder and audio output
  outRC = decimator(y);
  rcvdCode = decoder(outRC);
  if (rcvdCode == frsRxParams.CTCSSCode) || (frsRxParams.CTCSSCode == 0)
    rcvdSig = outRC;
  else
    rcvdSig = single(zeros(frsRxParams.AudioFrameLength, 1));
  end
  audioSig = audioFilter(rcvdSig);
  audioPlayer(audioSig);

  % Update radio time. If there were lost samples, add those too.
  radioTime = radioTime + frsRxParams.FrontEndFrameTime + ...
    double(lost)/frsRxParams.FrontEndSampleRate;
end

% Release the resources
release(fmDemod)
release(audioPlayer)
release(sigSrc)
```

Further Exploration

The CTCSS decoding computes the DFT (Discrete Fourier Transform) of the incoming signal using the Goertzel algorithm and computes the power at the tone frequencies. Because the tone frequencies are very close to each other (only 3-4 Hz apart) the block length of the DFT should be large enough to provide enough resolution for the frequency analysis. However, long block lengths cause decoding delay. For example, a block length of 16384 will cause 2 seconds of delay because the CTCSS decoder operates at an 8 kHz sampling rate. This creates a trade-off between detection performance and processing latency. The optimal block length may depend on the quality of the transmitter and receiver, the distance between the transmitter and receiver, and other factors. You are encouraged to change the block length in the initialization function by navigating to the helperFRSReceiverConfig function and changing the value of the CTCSSDecodeBlockLength field.

This will enable you to observe the trade-off and find the optimal value for your transmitter/receiver pair.

You can explore following functions and System objects for details of the physical layer implementation:

- helperFRSReceiverConfig.m
- helperFRSCTCSSDecoder.m
- helperFRSSignalGenerator.m

References

- Family Radio Service
- General Mobile Radio Service
- Continuous Tone-Coded Squelch System
- Goertzel Algorithm

Frequency Offset Calibration for Receivers in Simulink

This example shows how to measure and calibrate for transmitter/receiver frequency offset at the receiver using Simulink® and Communications Toolbox™. You can either use captured signals or receive signals in real time using the Communications Toolbox Support Package for RTL-SDR Radio. The receiver monitors the received signal, calculates and display the transmitter/receiver frequency offset.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Simulink
- Communications Toolbox™

To receive signals in real time, you also need the following hardware:

- RTL-SDR radio

and the following software

- Communications Toolbox Support Package for RTL-SDR Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of Software Defined Radio (SDR) discovery page.

Introduction

For an introduction on the frequency offset calibration for receivers, refer to the “Frequency Offset Calibration for Receivers” on page 8-461 example.

Running the Example

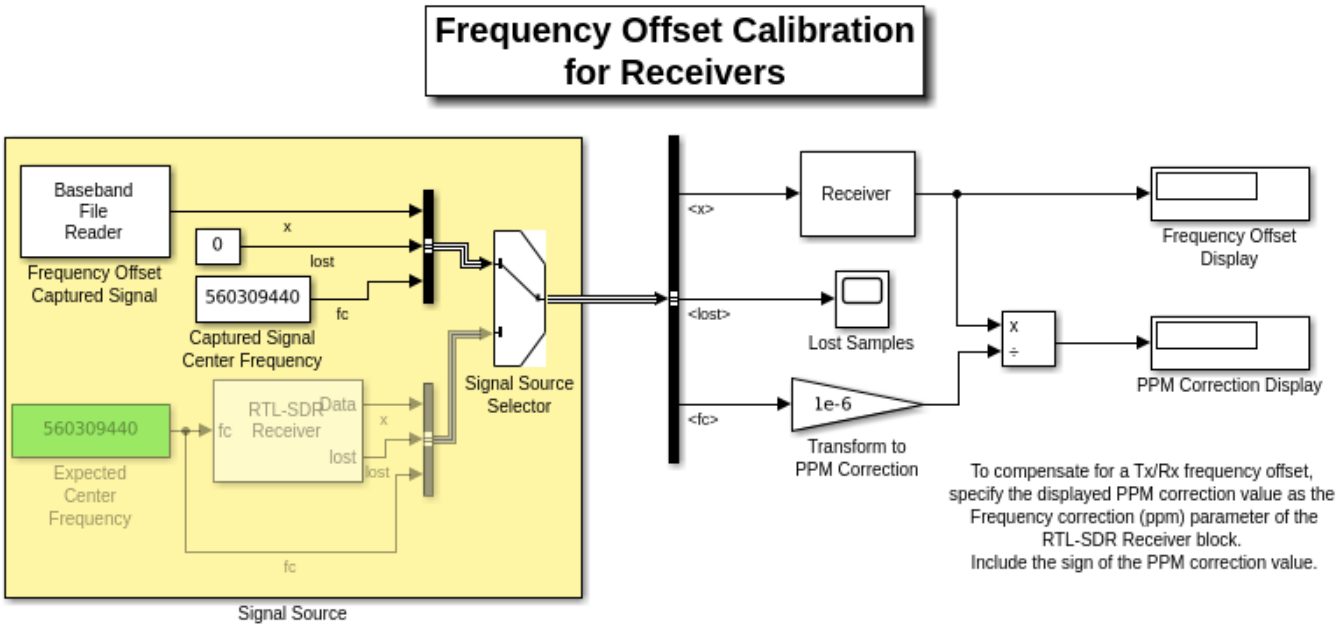
To run the example using captured signals, select the `Frequency Offset Captured Signal` block as the source using the `Signal Source Selector` block. Then click the run button. The model processes signals that were captured with an RTL-SDR radio at a center frequency of 560309440 Hz. This value corresponds to the pilot tone of channel 29 of digital TV signals in the USA.

To run the example using the RTL-SDR radio as the source, select the `RTL-SDR Receiver` block as the source using the `Signal Source Selector` block. Double-click the `Expected Center Frequency` block and set to the expected tone frequency. Begin transmitting with your known signal source. If you are in the USA, you can set the expected center frequency to the pilot tone of a near by digital TV transmitter. For a list of channel number and frequency values, see North American television frequencies. Then click the run button.

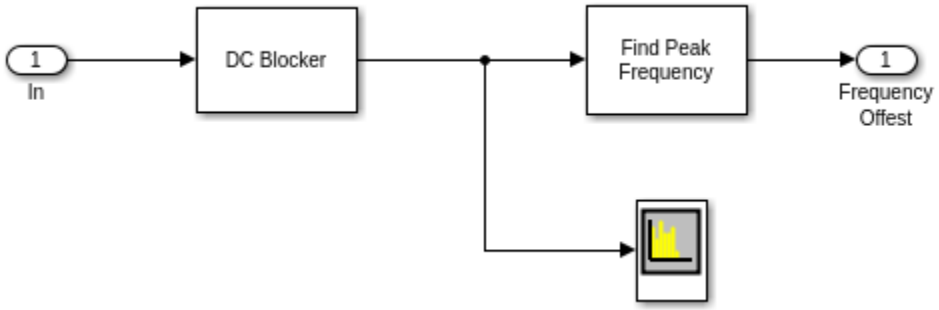
If you use the RTL-SDR radio as the source, to compensate for a transmitter/receiver frequency offset, specify the displayed PPM correction value as the Frequency correction (ppm) parameter of the RTL-SDR Receiver block. Be sure to use the sign of the offset in your specification. The spectrum displayed by the Spectrum Analyzer block should then have its maximum at 0 Hz.

Structure of the Example

The following figure shows the receiver model:



The following figure shows the detailed structure of the **Receiver** subsystem:



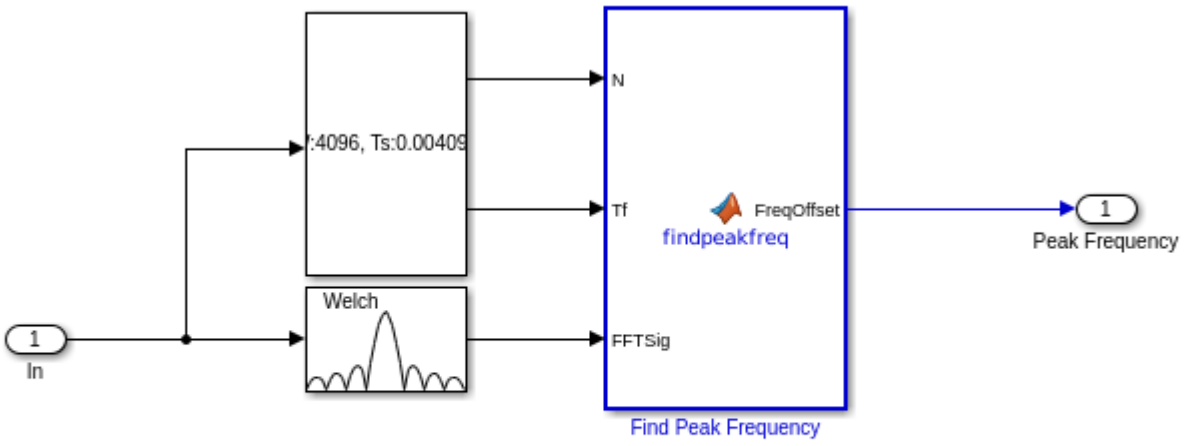
- The **Find Peak Frequency** block - uses an FFT to find the frequency with the maximum power in the received signal.
- The **Spectrum Analyzer** block - computes and displays the power spectral density of the received signal.

Find Peak Frequency

The **Find Peak Frequency** subsystem finds the frequency with the maximum power in the received signal, which equals the frequency offset. The following diagram shows the subsystem. In this subsystem, the Periodogram block returns the PSD estimate of the received signal. The Probe block finds the frame size and the frame sample time. With this information, this subsystem finds the index of the maximum amplitude across the frequency band and converts the index to the frequency value according to

Foffset = IndexofMaxAmplitude * FrameSize / (FFTLength * FrameSampleTime)

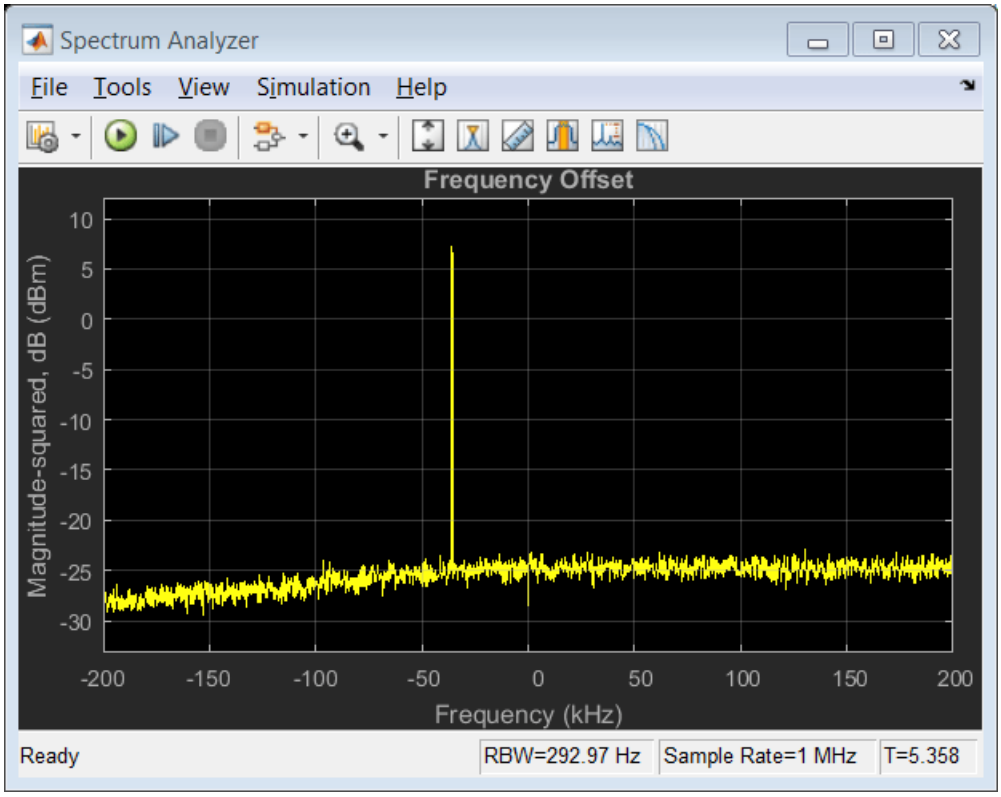
The MATLAB function findpeakfreq.m performs this conversion.



This subsystem does the following:
1. Finds the index of the maximum amplitude across the frequency band
2. Converts the index to the frequency offset according to
Foffset = IndexofMaxAmplitude * FrameSize / (FFTLength * FrameSampleTime)

Spectrum Analyzer

The following figure shows the output of the Spectrum Analyzer on a frequency range of -200 kHz to 200 kHz. In the case shown below, the frequency with the maximum power of the received signal is about -35 kHz.



Airplane Tracking Using ADS-B Signals in Simulink

This example shows you how to track planes by processing Automatic Dependent Surveillance-Broadcast (ADS-B) signals using Simulink® and Communications Toolbox™. You can either use captured and saved signals, or receive signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio. The example can show the tracked planes on a map, if you have the Mapping Toolbox™.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Simulink
- Communications Toolbox™

To receive signals in real time, you also need one of the following SDR devices and the corresponding support package Add-On:

- RTL-SDR radio and the corresponding Communications Toolbox Support Package for RTL-SDR Radio Add-On
- ADALM-PLUTO radio and the corresponding Communications Toolbox Support Package for ADALM-PLUTO Radio Add-On

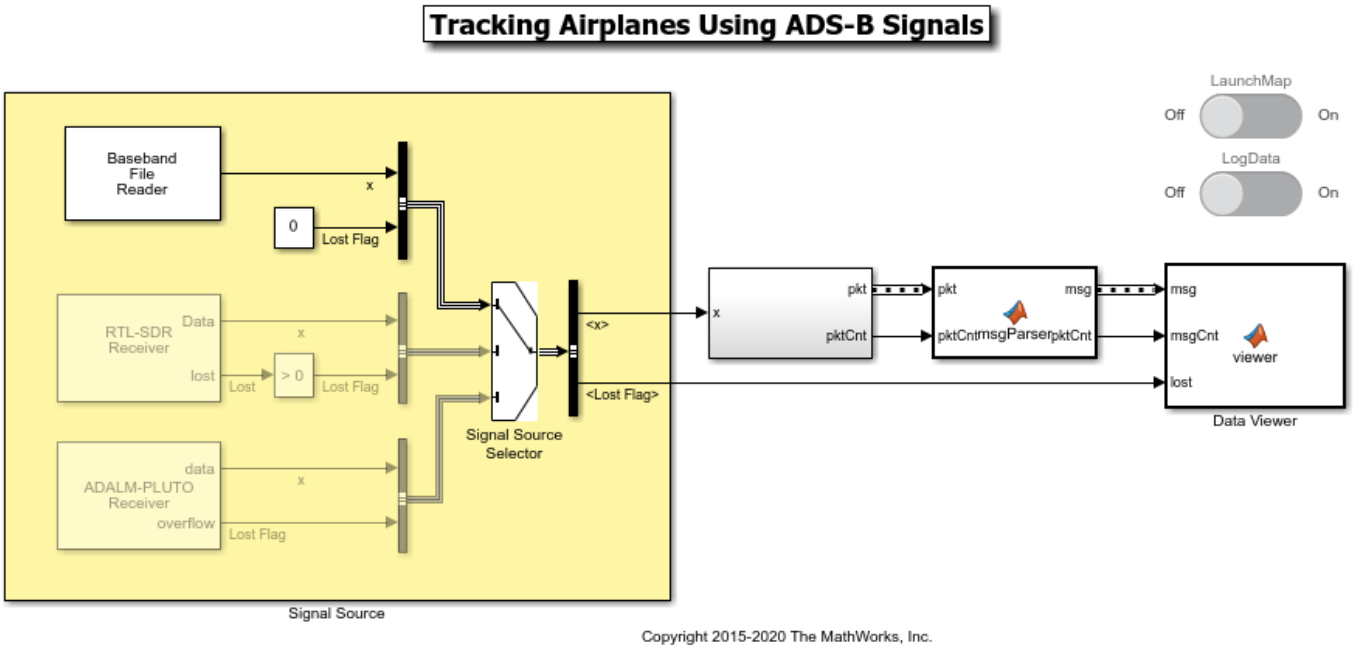
For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

Introduction

For an introduction on the Mode-S signaling scheme and ADS-B technology for tracking aircraft, refer to the “Airplane Tracking Using ADS-B Signals” on page 8-464 MATLAB® example.

Receiver Structure

The following block diagram summarizes the receiver code structure. The processing has four main parts: Signal Source, Physical Layer, Message Parser, and Data Viewer.



Signal Source

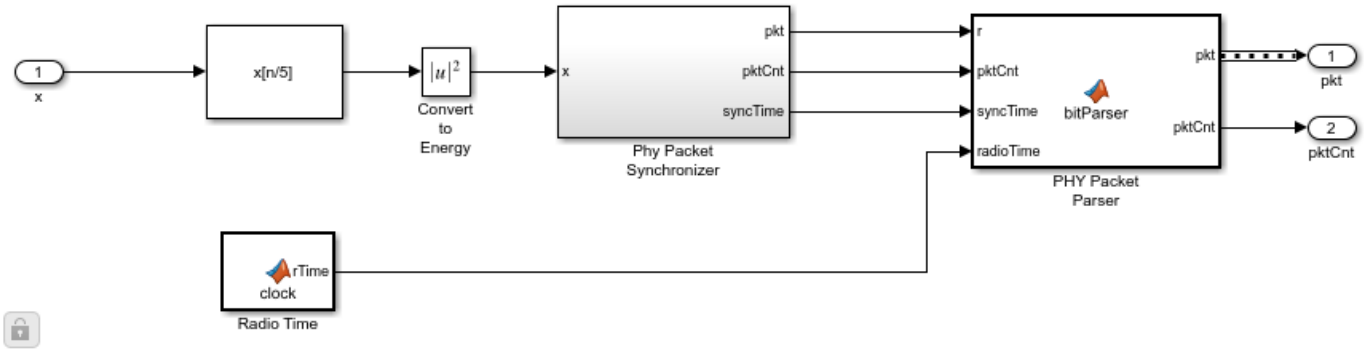
This example can use signal sources from a:

1. ''Captured Signal'': Over-the-air signals written to a file and sourced using a baseband file reader block at 2.4 Msps
2. ''RTL-SDR Radio'': RTL-SDR radio at 2.4 Msps
3. ''ADALM-PLUTO'': ADALM-PLUTO radio at a sample rate of 12 Msps

Here the extended squitter message is 120 micro seconds long, so the signal source is configured to process enough samples to contain 180 extended squitter messages at once, and set `SamplesPerFrame` of the signal property accordingly. The rest of the algorithm searches for Mode-S packets in this frame of data and outputs all correctly identified packets. This type of processing is defined as batch processing. An alternative approach is to process one extended squitter message at a time. This single packet processing approach incurs 180 times more overhead than the batch processing, while it has 180 times less delay. Since the ADS-B receiver is delay tolerant, batch processing was used.

Physical Layer

The baseband samples received from the signal source are processed by the physical (PHY) layer to produce packets that contain the PHY layer header information and the raw message bits. The following diagram shows the physical layer structure.



The RTL-SDR radio is capable of using a sampling rate in the range [200e3, 2.8e6] Hz. When RTL-SDR radio is the source, the example uses a sampling rate of 2.4e6 Hz and interpolates by a factor of 5 to a practical sampling rate of 12e6 Hz.

The ADALM-PLUTO radio is capable of using a sampling rate in the range [520e3, 61.44e6] Hz. When the ADALM-PLUTO radio is the source, the example samples the input directly at 12 MHz.

With the data rate of 1 Mbit/s and a practical sampling rate of 12 MHz, there are 12 samples per symbol. The receive processing chain uses the magnitude of the complex symbols.

The packet synchronizer works on subframes of data that is equivalent to two extended squitter packets, i.e. 1440 samples at 12 MHz or 120 micro seconds. This subframe length ensures that a whole extended squitter packet can be found in the subframe. Packet synchronizer first correlates the received signal with the 8 microsecond preamble and find the peak value. Then, it validates the found synchronization point by checking if it confirms to the preamble sequence, [1 0 0 0 0 0 1 0 1 0 0 0 0 0 0], where a '1' represents a high value and a '0' represents a low value.

The Mode-S PPM modulation scheme defines two symbols. Each symbol has two chips, where one has a high value and the other has a low value. If the first chip is high followed by low chip, this corresponds to the symbol being a 1. Alternatively, if the first chip is low followed by high chip, then the symbol is 0. The bit parser demodulates the received chips and creates a binary message. The binary message is validated using a CRC checker. The output of bit parser is a vector of Mode-S physical layer header packets that contains the following fields:

- RawBits: Raw message bits
- CRCError: FALSE if CRC checks, TRUE if CRC fails
- Time: Time of reception in seconds from start of receiver
- DF: Downlink format (packet type)
- CA: Capability

Message Parser

The message parser processes the raw bits based on the packet type as described in [2]. This example can parse short squitter packets and extended squitter packets that contain airborne velocity, identification, and airborne position data.

Data Viewer

The data viewer shows the received messages on a graphical user interface (GUI). For each packet type, the number of detected packets, the number of correctly decoded packets and the packet error

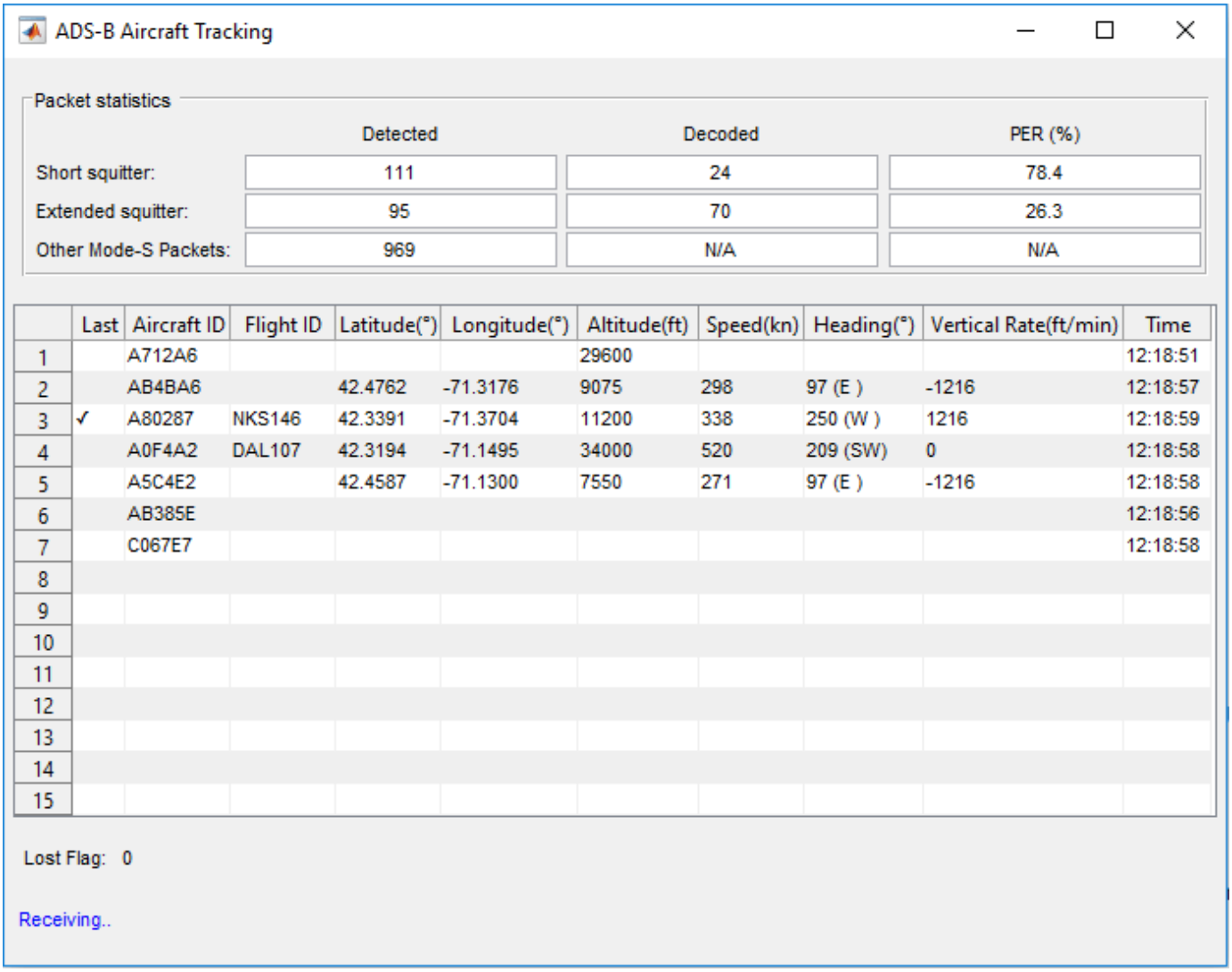
rate (PER) is shown. As data is captured, the application lists information decoded from these messages in a tabular form.

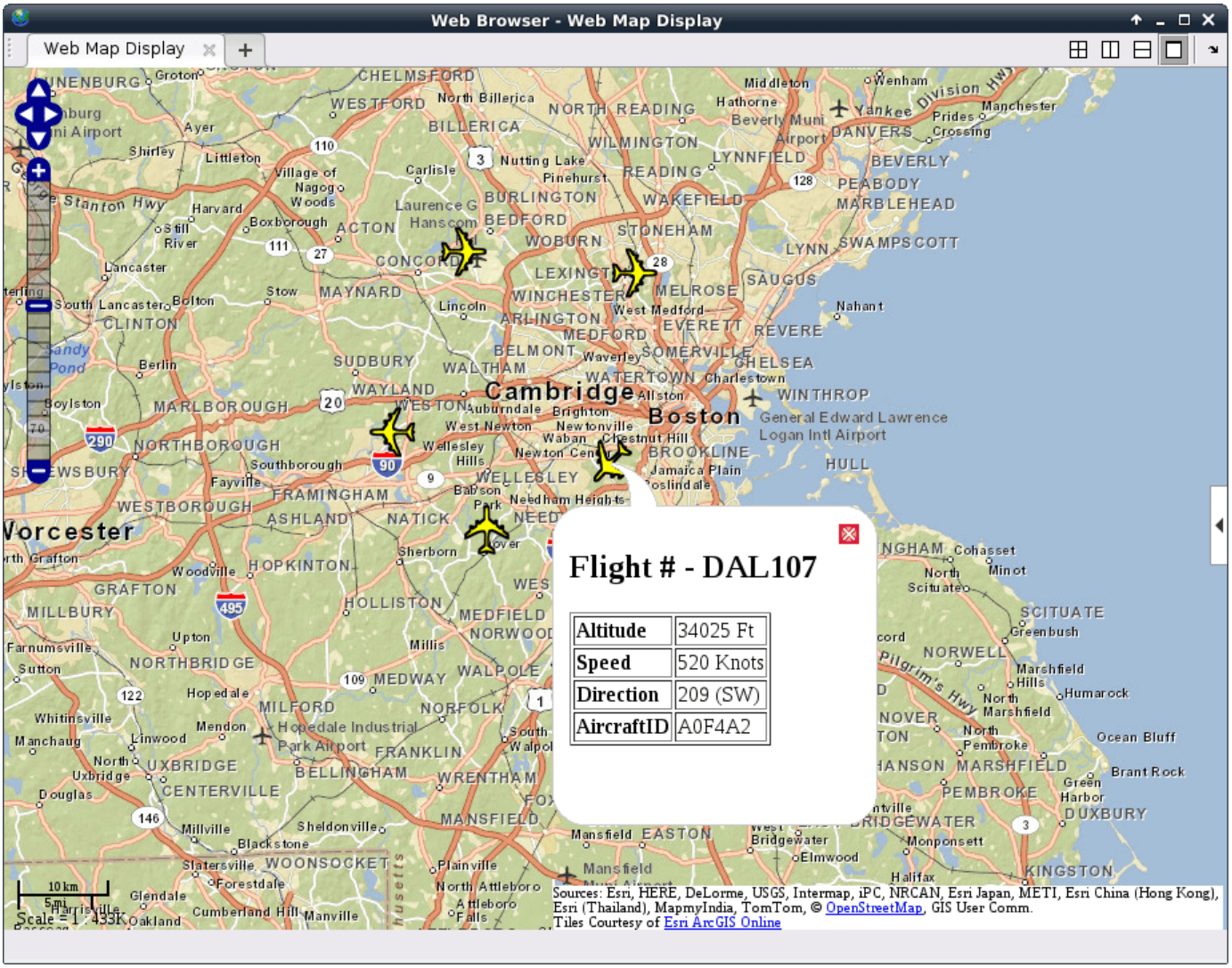
Launch Map and Log Data

You can also launch the map and start text file logging using the two slider switches(Launch Map and Log Data).

- **Log Data*** - When Log Data is On, it Saves the captured data in a TXT file. You can use the saved data for later for post processing.
- **Launch Map** - When Launch Map is On, map will be launched where the tracked flights can be viewed. **NOTE:** You must have a valid license for the Mapping Toolbox if you want to use this feature.

The following figures illustrate how the application tracks and lists flight details and displays them on a map.





Selected Bibliography

1 International Civil Aviation Organization, Annex 10, Volume 4. Surveillance and Collision Avoidance Systems.

2 Technical Provisions For Mode S Services and Extended Squitter (Doc 9871)

Airplane Tracking Using ADS-B Signals with Raspberry Pi and RTL-SDR

This example shows you how to create a remote sensing station that tracks planes using a Raspberry Pi™ and RTL-SDR radio. You will learn how to deploy a Simulink® model that processes Automatic Dependent Surveillance-Broadcast (ADS-B) signals and sends the demodulated data to a host PC using UDP packets for visualization.

Required Hardware and Software

To run this example, you need the following hardware:

- RTL-SDR radio
- Raspberry Pi

and the following software

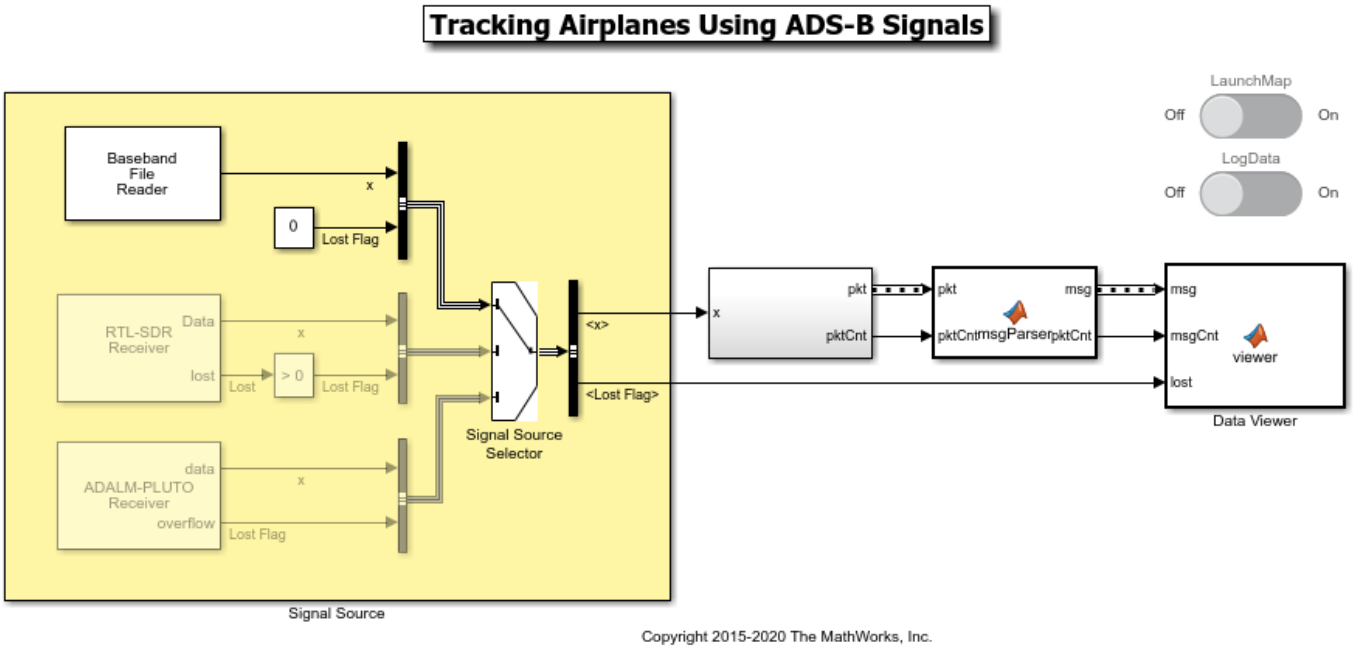
- *Simulink*
- *Communications Toolbox™*
- *Communications Toolbox Support Package for RTL-SDR Radio*
- *Simulink Support Package for Raspberry Pi Hardware*
- Optionally, Mapping Toolbox™ (to track planes on a map)

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

Introduction

For an introduction on implementing a ADS-B receiver in Simulink, refer to the “Airplane Tracking Using ADS-B Signals in Simulink” on page 8-512 example, pictured below. We also recommend completing “Getting Started with MATLAB Support Package for Raspberry Pi Hardware” (MATLAB Support Package for Raspberry Pi Hardware) example.

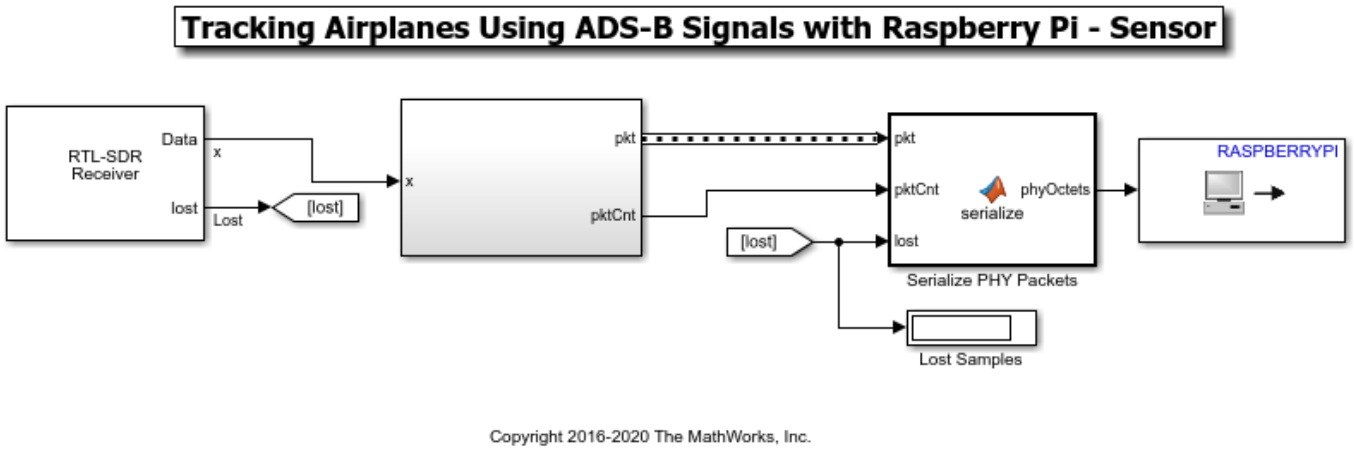
```
modelName = 'ADSBSimulinkExample';
open_system(modelName);
set_param(modelName, 'SimulationCommand', 'update');
```



Remote ADS-B Receiver with Raspberry Pi

You can set up a remote sensing station for airplane tracking using the Raspberry Pi hardware with an RTL-SDR radio attached to it. You can run the **PHY Layer** block of the ADS-B receiver on the Raspberry Pi and send the received data over the Internet using UDP packets. You can receive these UDP packets on your local computer and run the **Message Parser** and **Data Viewer** blocks to visualize the results. The following is the modified remote ADS-B receiver model that runs on Raspberry Pi.

```
close_system(modelName)
modelName = 'ADSBRaspberryPiSimulinkExample';
open_system(modelName);
set_param(modelName, 'SimulationCommand', 'update');
```



Run ADS-B Receiver Model on Raspberry Pi Hardware

Follow the following steps to run the ADS-B receiver model on the Raspberry Pi hardware.

1. Attach an RTL-SDR radio to one of the USB ports of the Raspberry Pi hardware

2. Open the Tracking Airplanes Using ADS-B Signals with Raspberry Pi - Sensor model

3. Double-click on the **UDP Send** block. Open the block mask and enter the `IP address of your host computer` in the **Remote IP address** edit box. For example, if the IP address of your host computer is 10.10.10.1, enter '10.10.10.1' in the block mask. Do not change the **Remote IP port** parameter. Click OK to save and close the block mask.

4. In your Simulink model, click the **Deploy To Hardware** button on the toolbar.

5. The model running on Raspberry Pi hardware will start sending UDP packets to port 25000 of your host computer.

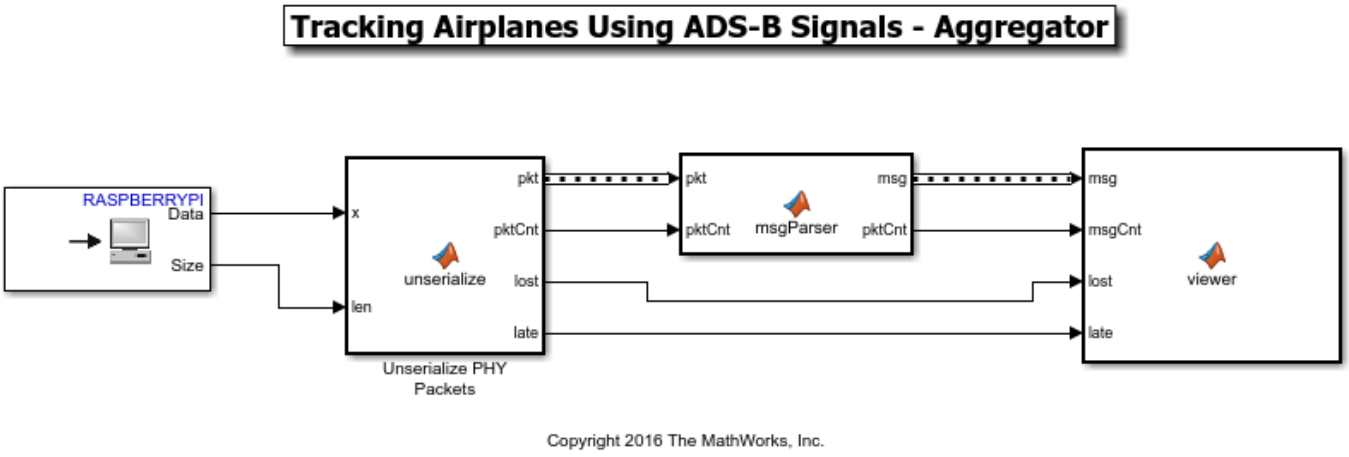
Run ADS-B Aggregator Model on the Host Computer

Follow the following steps to run the host model that receives the UDP packets sent by the model running on Raspberry Pi hardware.

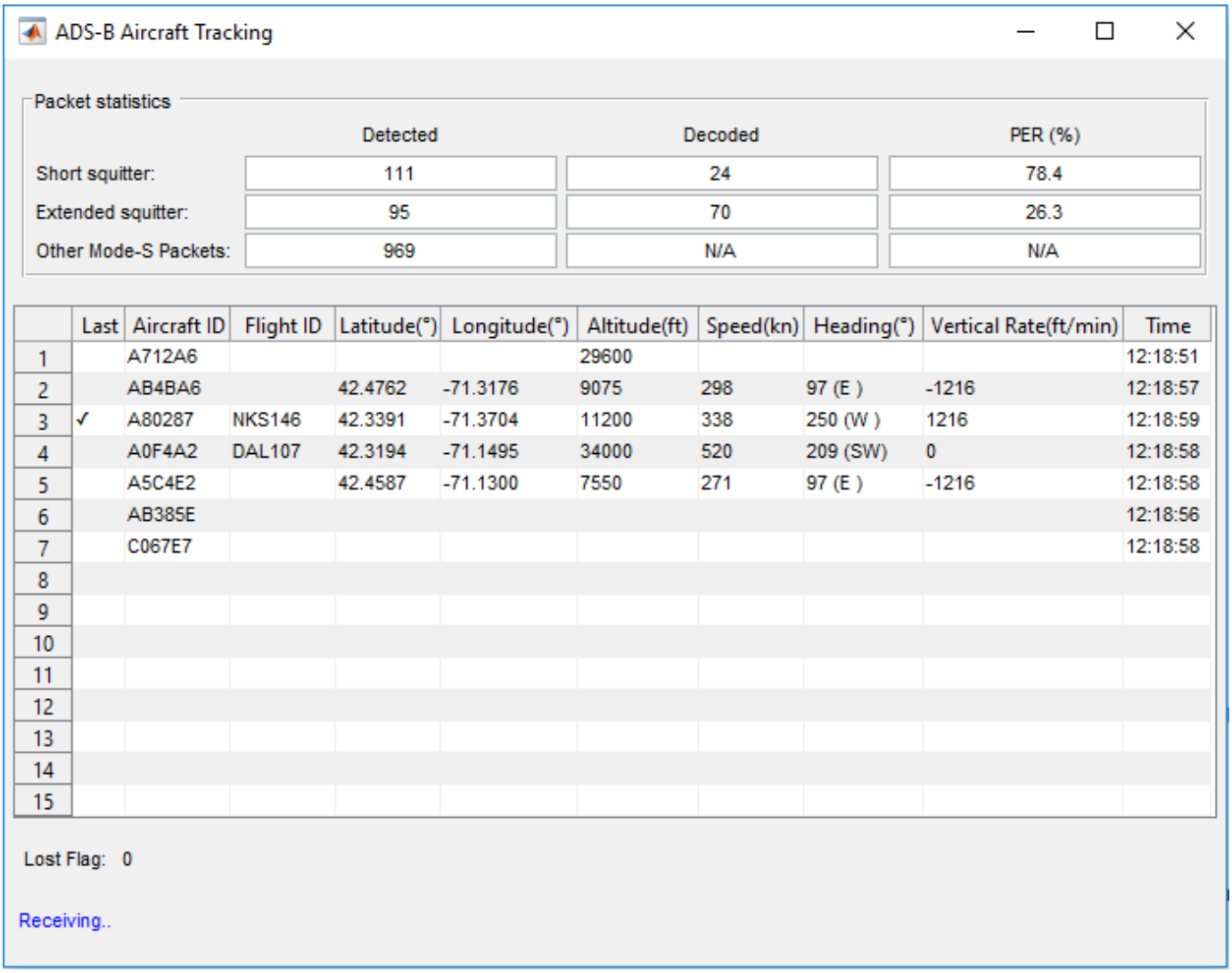
1. Open the Tracking Airplanes Using ADS-B Signals - Aggregator. This model has a **UDP Receive** block that is configured to receive UDP packets sent by the model running on Raspberry Pi hardware. Double-click on the **UDP Receive** block mask. Note that the Local IP port is set to 25000, and the output data type is set to "uint8".

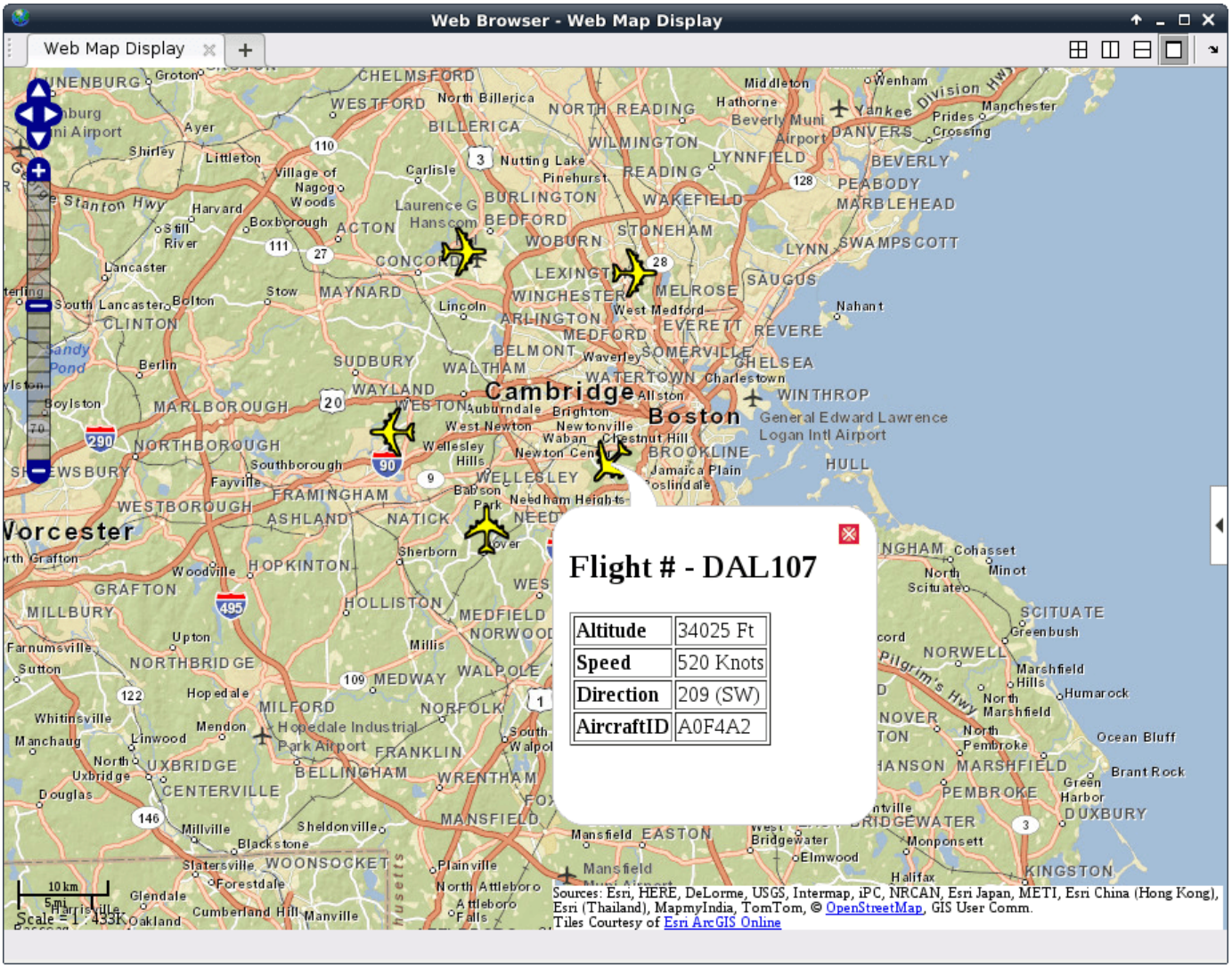
2. Click the Play button to start the model.

```
close_system(modelName)
modelName = 'ADSBAggregatorSimulinkExample';
open_system(modelName);
set_param(modelName, 'SimulationCommand', 'update');
```



The following figures illustrate how the application tracks and lists flight details and displays them on a map.





Stop the Model Running on Raspberry Pi Hardware

When you want to stop the model running on Raspberry Pi, execute the following on MATLAB® command line.

```
rPi = raspberrypi;
stop(rPi,'ADSBRaspberryPiSimulinkExample');
```

Troubleshooting

If you cannot receive any data on the host model, make sure that the Raspberry Pi and your host computer are on the same local area network. In other words, make sure that the first three numbers of the IP addresses are the same.

Also, make sure that your Internet security software allows the transmission and reception of UDP packets on port 25000.

```
close_system(modelName)
```

Automatic Meter Reading in Simulink

This example shows you how to use Simulink® and Communications Toolbox™ to read utility meters by processing Standard Consumption Message (SCM) or Interval Data Message (IDM) signals emitted by meters. You can either use recorded data from a file, or receive over-the-air signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio.

Required Hardware and Software

To run this example using recorded data from a file, you need the following software:

- Simulink
- Communications Toolbox™

To receive signals in real time, you also need one of the following SDR devices and the corresponding support package Add-On:

- RTL-SDR radio and the corresponding Communications Toolbox Support Package for RTL-SDR Radio
- ADALM-PLUTO radio and the corresponding Communications Toolbox Support Package for ADALM-PLUTO Radio

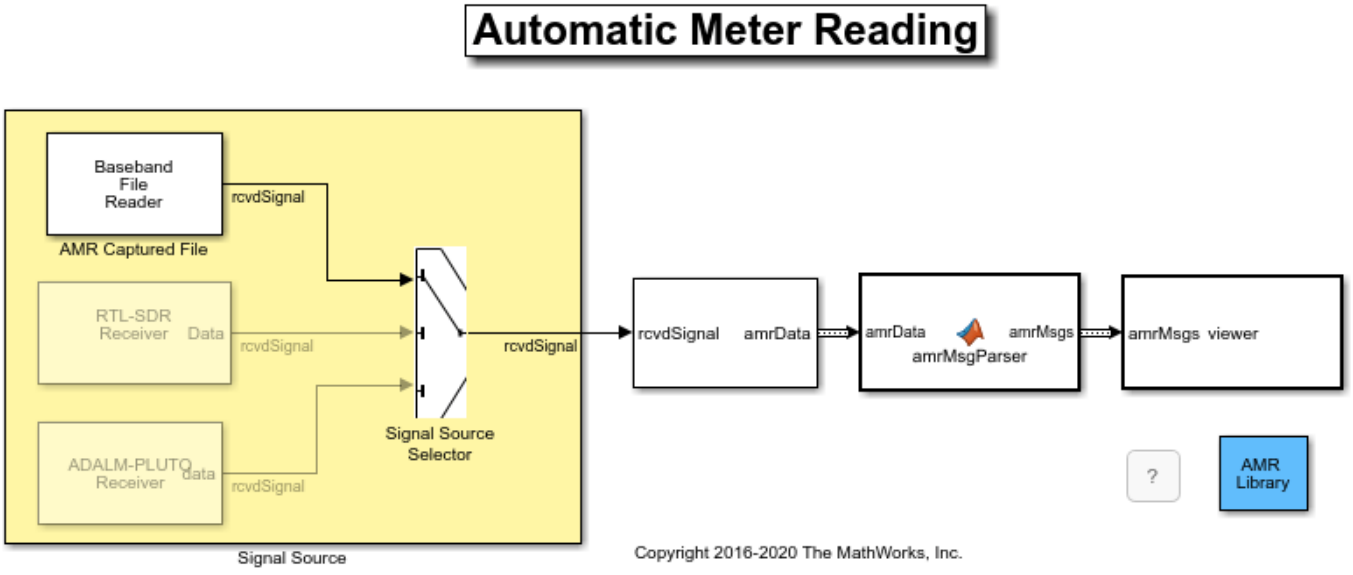
For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of the *Software Defined Radio (SDR) discovery page*.

Introduction

For an introduction to the SCM/IDM signaling scheme and AMR technology for reading utility meters, refer to the “Automatic Meter Reading” on page 8-472 example in MATLAB®.

Receiver Model Structure

The following block diagram summarizes the receiver structure. The processing has four main parts: Signal Source, Physical Layer, Message Parser, and Data Viewer.



Signal Source

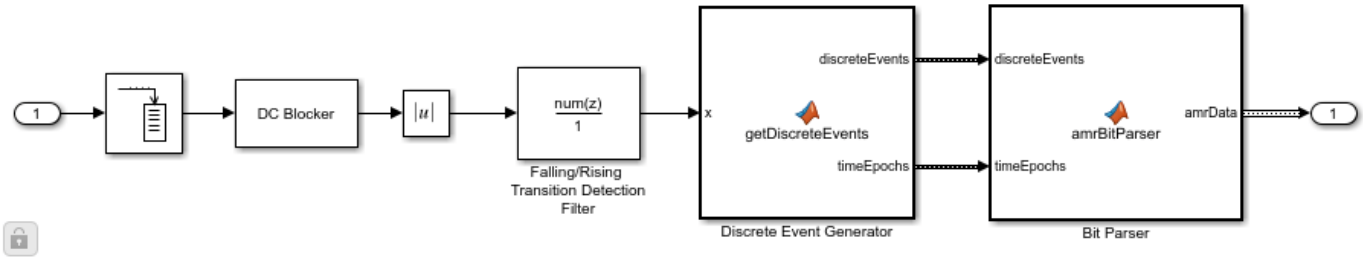
This example can use three signal sources:

1. ''File'': Over-the-air signals written to a file and read using a Baseband File Reader block at 1.0 Msps
2. ''RTL-SDR Radio'': RTL-SDR radio at a sample rate of 1.0 Msps
3. ''ADALM-PLUTO Radio'': ADALM-PLUTO radio at a sample rate of 1.0 Msps

If you assign ''RTL-SDR'' or ''ADALM-PLUTO'' as the signal source, the example searches your computer for the radio you specified, either an RTL-SDR radio at radio address '0' or an ADALM-PLUTO radio at radio address 'usb:0' and uses the radio as the signal source.

Physical Layer

The baseband samples received from the signal source are processed by the physical layer (PHY) to produce packets that contain the SCM or IDM information. This diagram shows the physical layer receive processing.



The RTL-SDR radio is capable of using a sampling rate in the range of 225-300 kHz or 900-2560 kHz and ADALM-PLUTO radio is capable of using a sampling rate in the range of 520 kHz-61.44 MHz. A sampling rate of 1.0 Msps is used to produce a sufficient number of samples per Manchester encoded data bit. For each frequency in the hopping pattern, every AMR data packet is transmitted. The frequency hopping allows for increased reliability over time. Since every packet is transmitted on each frequency hop, it is sufficient to monitor only one frequency for this example. The radio is tuned to a center frequency of 915 MHz for the entire simulation runtime.

The received complex samples are amplitude demodulated by extracting their magnitude. The on-off keyed Manchester coding implies that the bit selection block includes clock recovery. The output of this block is bit sequences (ignoring the idle times in the transmission) which are subsequently checked for the known preamble. If the preamble matches, the bit sequence is further decoded, otherwise, it is discarded and the next sequence is processed.

When the known SCM preamble is found for a bit sequence, the received message bits are decoded using a shortened (255,239) BCH code which can correct up to two bit errors. In the case where the known IDM preamble is found, the receiver performs a cyclic redundancy check (CRC) of the meter serial number and of the whole packet starting at the Packet type (the 5th byte) to determine if the packet is valid. Valid, corrected messages are passed onto the AMR Message parser.

Message parser

For a valid message, the bits are then parsed into the specific fields of either the IDM or SCM format. This example can parse both the SCM format and the IDM format.

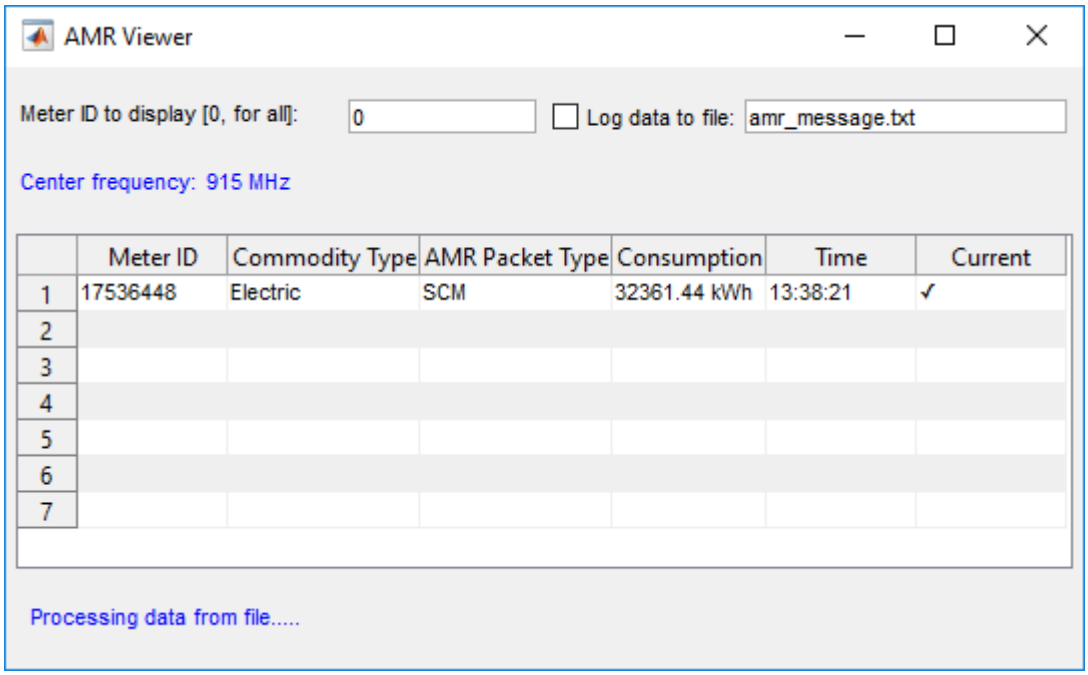
Data Viewer

The data viewer shows the decoded SCM or IDM messages on a user interface. For each successfully decoded SCM/IDM, the commodity type, meter ID, consumption information and the capture time is shown. As data is captured and decoded, the application lists the information decoded from these messages in a tabular form. The table lists only the unique meter IDs with their latest consumption information.

You can also change the meter ID and start text file logging using the user interface.

- **Meter ID** - The default value, `0`, is reserved for displaying all detected meters. You can enter the ID of a specific meter to display readings from only that meter ID.
- **Log data to file** - Save the decoded messages in a TXT file. You can use the saved data for post processing.

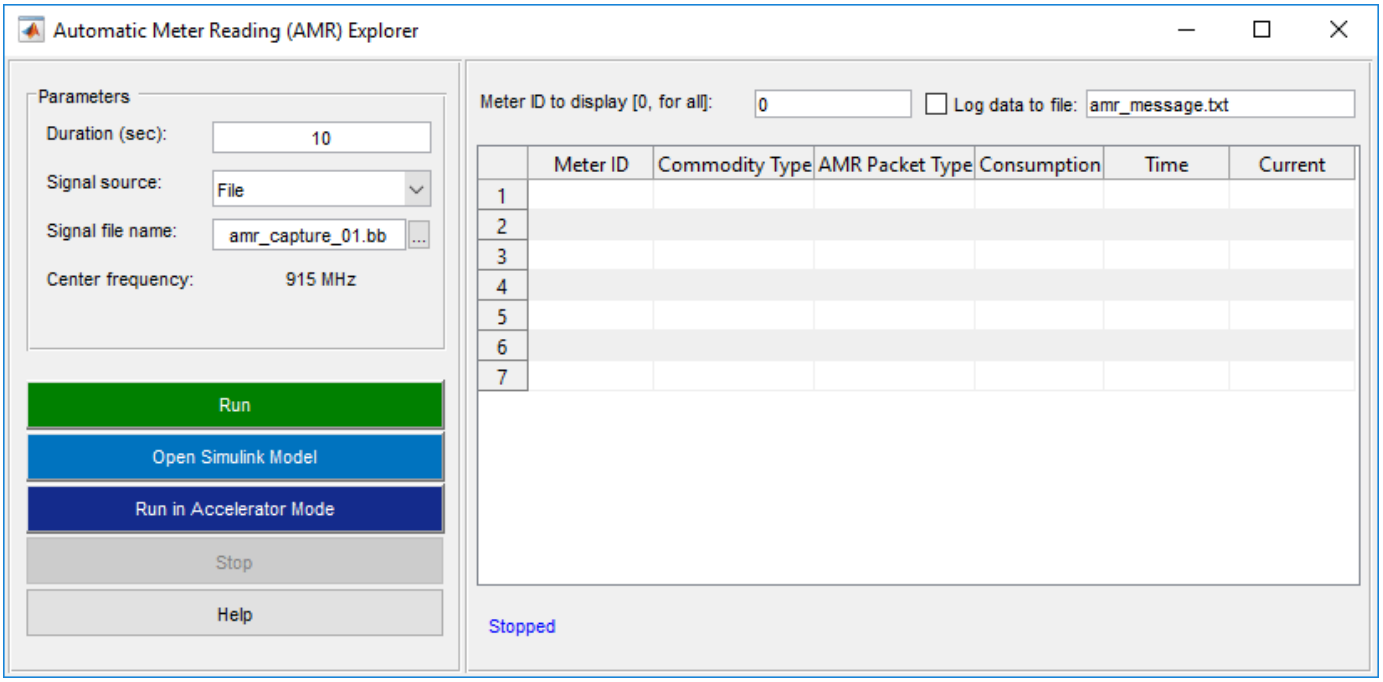
This figures shows the meter readings displayed in the user interface.



Further Exploration

The data file accompanying the example has only one meter reading and has been captured at center frequency of 915 MHz. Using RTL-SDR or ADALM-PLUTO radio, the example will display readings from multiple meters when it is run for a longer period in a residential neighborhood.

You can further explore AMR signals using the AMRSimulinkExampleApp app. The app allows you to set the run duration, select the signal source, change the center frequency of the radio, and run to log meter readings.



Selected Bibliography

1 Automatic meter reading, https://en.wikipedia.org/wiki/Encoder_receiver_transmitter, 2016.

2 Itron Electricity meters, https://www.itron.com/solutions/who-we-serve/electricity, 2017.

FM Broadcast Receiver

This example shows how to build an FM mono or stereo receiver using Simulink® and Communications Toolbox™. You can either use captured signals, or receive signals in real time using the RTL-SDR or ADALM-PLUTO.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- *Simulink*
- *Communications Toolbox™*

To receive signals in real time, you also need one of the following hardware:

- RTL-SDR radio and the corresponding software *Communications Toolbox Support Package for RTL-SDR Radio*
- ADALM-PLUTO radio and the corresponding software *Communications Toolbox Support Package for ADALM-PLUTO Radio*

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR)*.

Introduction

For an introduction to the FM broadcasting technology and demodulation of these signals, refer to the “FM Broadcast Receiver” on page 8-490 example.

Running the Example

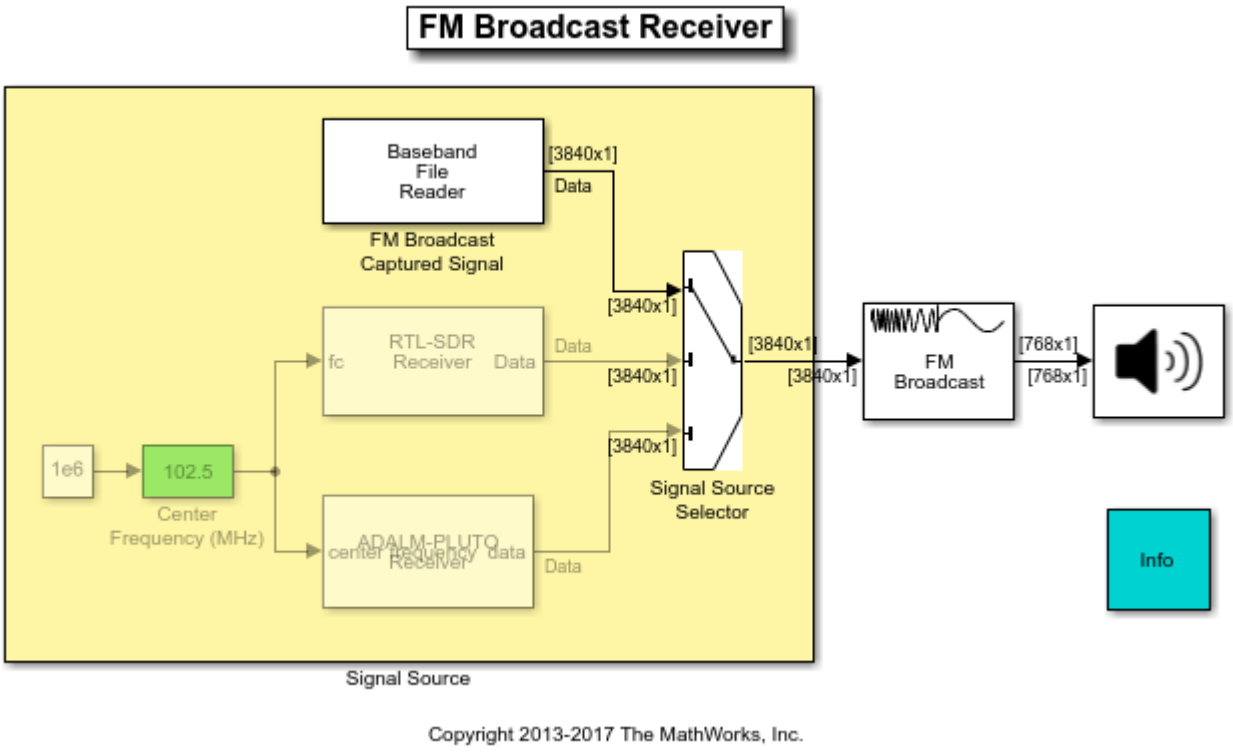
To run the example using captured signals, select the `FM Broadcast Captured Signal` block as the source using the `Signal Source Selector` block. Then click the run button.

To run the example using the RTL-SDR radio or ADALM-PLUTO radio as the source, select the corresponding `RTL-SDR Receiver` or `ADALM-PLUTO Radio Receiver` block as the source using the `Signal Source Selector` block. Double-click the `Center Frequency (MHz)` block and select the value to the center frequency to a broadcast FM radio station near you.

If you hear some dropouts or delay in the sound, run the model in Accelerator mode. From the model menu, select Simulation->Accelerator, then click the run button. If you still experience dropouts or delay in Accelerator mode, try running the model in Rapid Accelerator mode.

Receiver Structure

The following block diagram summarizes the receiver structure. The processing has three main parts: signal source, FM broadcast demodulation, and audio output.



Signal Source

This example can use three signal sources:

1. ''Captured Signal'': Over-the-air signals written to a file and sourced using a Baseband File Reader block at 228e3 samples/sec.
2. ''RTL-SDR Radio'': RTL-SDR radio running at 228e3 samples/sec. Set the center frequency to a broadcast FM radio station near you.
3. ''ADALM-PLUTO Radio Receiver'': ADALM-PLUTO radio running at 228e3 samples/sec. Set the center frequency to a broadcast FM radio station near you.

FM Broadcast Demodulation

The baseband samples received from the signal source are processed by the `FM Broadcast Demodulation Baseband` block. This block converts the input sampling rate of 228 kHz to 45.6 kHz, the sampling rate for your host computer's audio device. According to the FM broadcast standard in the United States, the de-emphasis lowpass filter time constant is set to 75 microseconds. This example processes received mono signals. The demodulator can also process stereo signals.

To perform stereo decoding, the FM Broadcast Demodulator Baseband object uses a peaking filter which picks out the 19 kHz pilot tone from which the 38 kHz carrier is created. Using the resulting carrier signal, the FM Broadcast Demodulator Baseband block downconverts the L-R signal, centered at 38 kHz, to baseband. Afterwards, the L-R and L+R signals pass through a 75 microsecond de-emphasis filter. The FM Broadcast Demodulator Baseband block separates the L and R signals and converts them to the 45.6 kHz audio signal.

Audio Device Writer

Play the demodulated audio signals through your computer's speakers using the `Audio Device Writer` block.

Further Exploration

To further explore the example, you can vary the center frequency of the RTL-SDR radio or ADALM-PLUTO radio and listen to other radio stations using the `Center Frequency (MHz)` block.

You can set the Stereo property of the `FM Broadcast Demodulator Baseband` block to true to process the signals in stereo fashion and compare the sound quality.

Selected Bibliography

https://en.wikipedia.org/wiki/FM_broadcasting

FM Reception with RTL-SDR Radio on Raspberry Pi Hardware

This example shows how to build an FM mono receiver using a Raspberry Pi™ and RTL-SDR radio. You will learn how to deploy a Simulink® model that processes FM broadcast signals and play the audio through the Raspberry Pi's speaker.

Required Hardware and Software

To run this example, you need the following hardware:

- RTL-SDR radio
- Raspberry Pi

and the following software

- *Simulink*
- *Communications Toolbox™*
- *Communications Toolbox Support Package for RTL-SDR Radio*
- *Simulink Support Package for Raspberry Pi Hardware*

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of *Software Defined Radio (SDR) discovery page*.

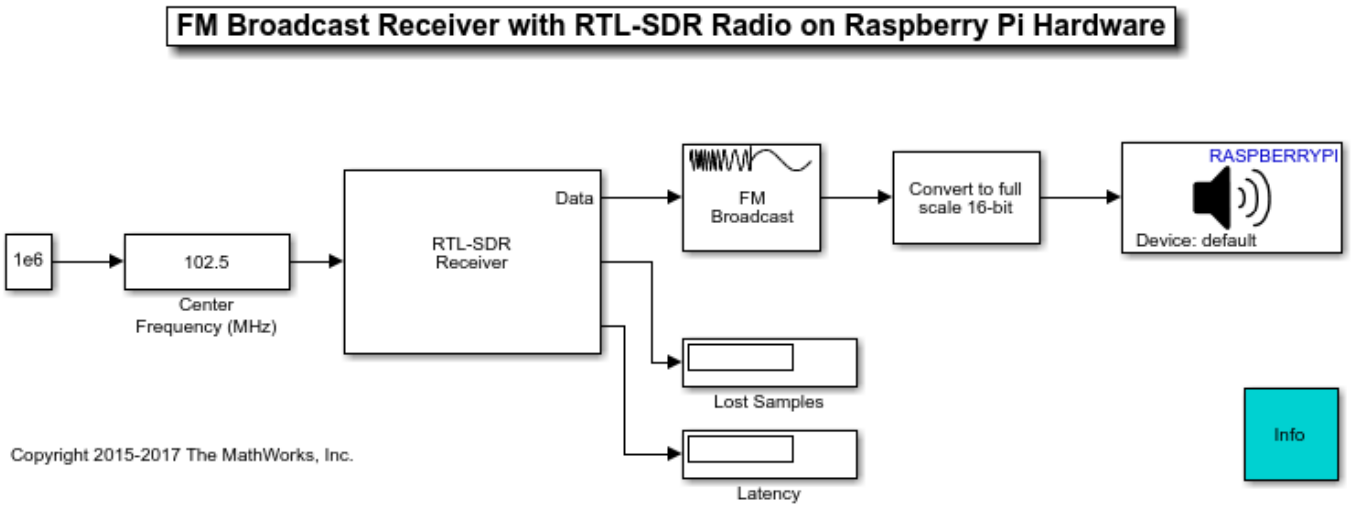
Introduction

Simulink Support Package for Raspberry Pi Hardware enables you to create and run Simulink models on Raspberry Pi hardware. Communications Toolbox Support Package for RTL-SDR Radio enables you to receive radio signals from the RTL-SDR radio. You can use these two support packages together to receive and process RF signals on the Raspberry Pi hardware using Simulink software. In this example, you will learn how to run an FM receiver model as a standalone application on the Raspberry Pi hardware.

For an introduction on implementing an FM broadcast receiver in Simulink, refer to the “FM Broadcast Receiver” on page 8-526 example. We also recommend completing “Getting Started with MATLAB Support Package for Raspberry Pi Hardware” (MATLAB Support Package for Raspberry Pi Hardware) example.

FM Receiver

The following shows the FM receiver model. The model uses the `RTL-SDR Receiver` block to receive radio signals and sends them to the `FM Broadcast Demodulator Baseband` block. The FM demodulator block demodulates the received signal and generates mono audio. The mono signals are sent to the `ALSA Audio Playback` block optimized for the Raspberry Pi hardware.



Configure and Run the Model as a Standalone Application

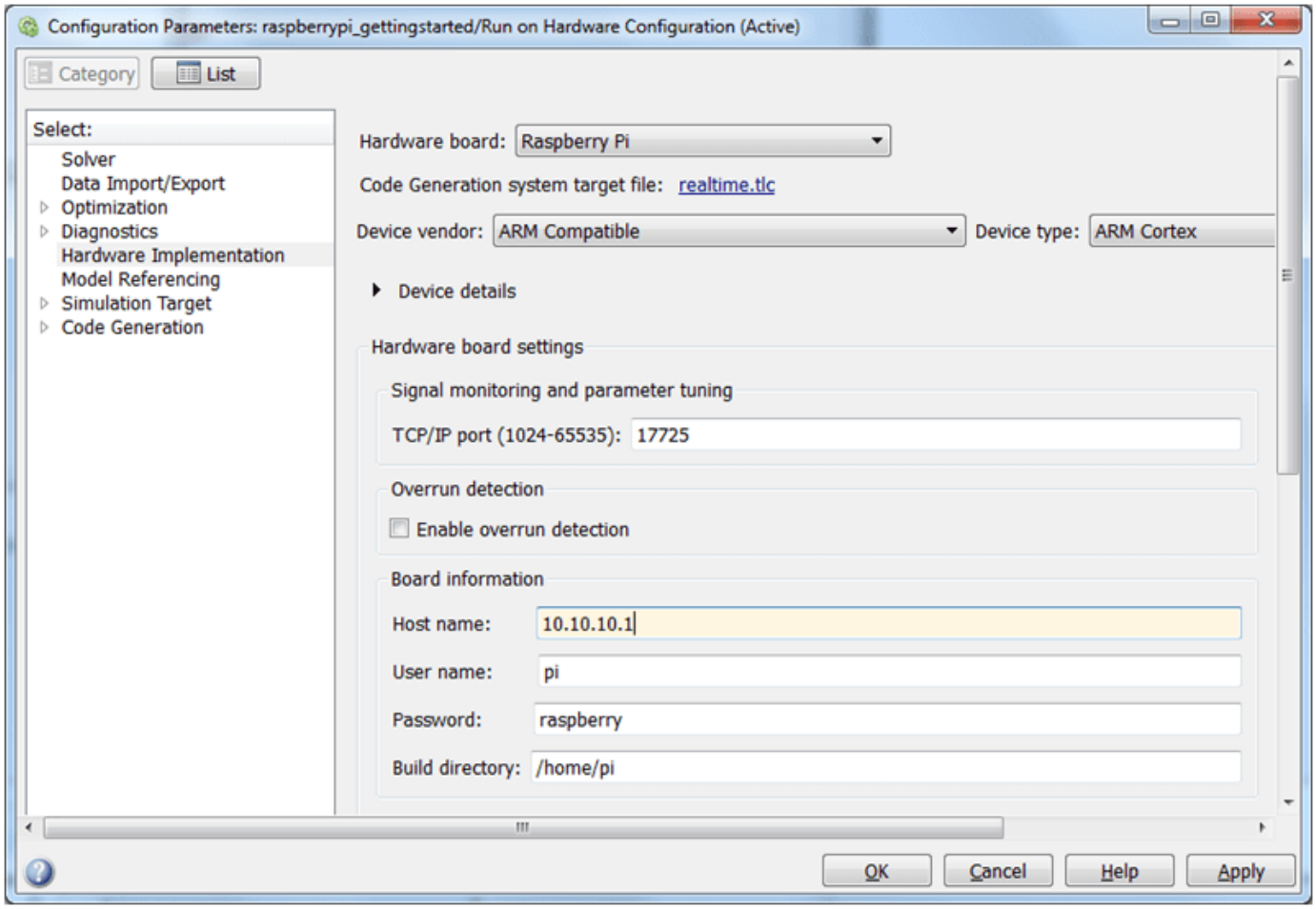
The following steps show you how to configure the model to run on the Raspberry Pi hardware.

1. If your Raspberry Pi hardware is not connected to an Ethernet network, follow the instructions in “Configure Network Settings on Raspberry Pi Hardware” (Simulink Support Package for Raspberry Pi Hardware).

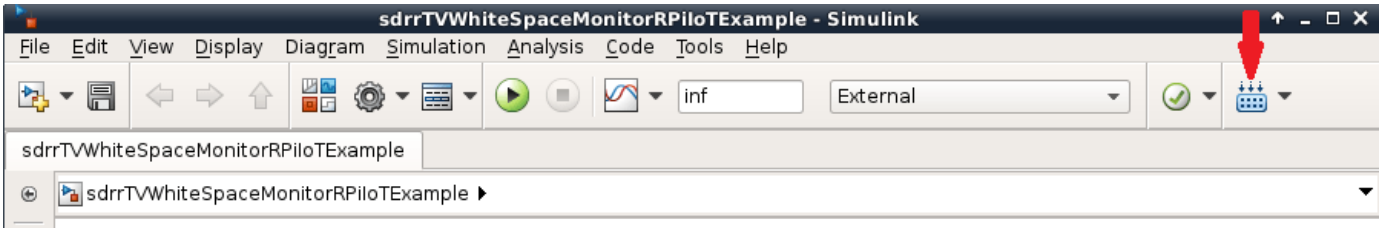
2. In the model, set simulation stop time to 'inf' to run the simulation until you explicitly pause or stop the model.

3. In the Simulink model, click **Tools > Run on Target Hardware> Options...**.

4. When the **Configuration Parameters** page opens up, set the **Target hardware** parameter to Raspberry Pi. Review the other parameters on that page. If you performed a Firmware Update, Board information will be automatically populated with the IP address, user name and password of your Raspberry Pi hardware. Also, notice the **TCP/IP port** edit box under Signal monitoring and parameter tuning. The default value of **TCP/IP port** is 17725. Simulink uses this TCP/IP port to communicate with Raspberry Pi hardware. Leave the TCP/IP port parameter at its default value. Click **OK** when you are done.



5. In the Simulink model, click the **Deploy to Hardware** button on the toolbar or press Ctrl+B.



6. The model will now run on the Raspberry Pi hardware. A system command window will open that shows the messages coming from the model running on Raspberry Pi hardware.

7. Connect speakers to the audio output of the Raspberry Pi hardware to listen to the radio.

8. Stop the model running on the Raspberry Pi hardware by executing the following on the MATLAB® command line

```
h = raspberrypi;
stopModel(h,'FMReceiverRaspberryPiSimulinkExample');
```

Running and Stopping the Model on Raspberry Pi Hardware

Simulink Support Package for Raspberry Pi Hardware generates a Linux® executable for each Simulink model you run on the Raspberry Pi hardware.

1. To run/stop a Simulink model, you use the run and stop methods of the raspberrypi communication object. First, create a communication object to the Raspberry Pi hardware:

```
rpi = raspberrypi;
```

This command generates a Raspberry Pi object that is your gateway to communicating with your Raspberry Pi hardware from MATLAB command line.

2. Execute the following on the MATLAB command line to stop the Simulink model you ran in previous section:

```
stopModel(rpi, 'FMReceiverRaspberryPiSimulinkExample')
```

3. To run a previously built Simulink model on your board, you use runModel method. In order to run the FMReceiverRaspberryPiSimulinkExample model, execute the following on the MATLAB command line:

```
runModel(rpi, 'FMReceiverRaspberryPiSimulinkExample')
```

Summary

This example introduced the workflow for receiving radio signals with an RTL-SDR radio and processing the received signals using a Simulink model running on Raspberry Pi hardware.

RDS/RBDS and RadioText Plus (RT+) FM Receiver

This example shows how you can use Simulink® and the Communications Toolbox™ to extract program or song information from FM radio stations using the RDS or RBDS standard and, optionally, the RadioText Plus (RT+) standard. You can either use captured signals or receive signals in real time using the RTL-SDR Radio or ADALM-PLUTO Radio.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Simulink
- Communications Toolbox™

To receive signals in real time, you also need one of the following hardware:

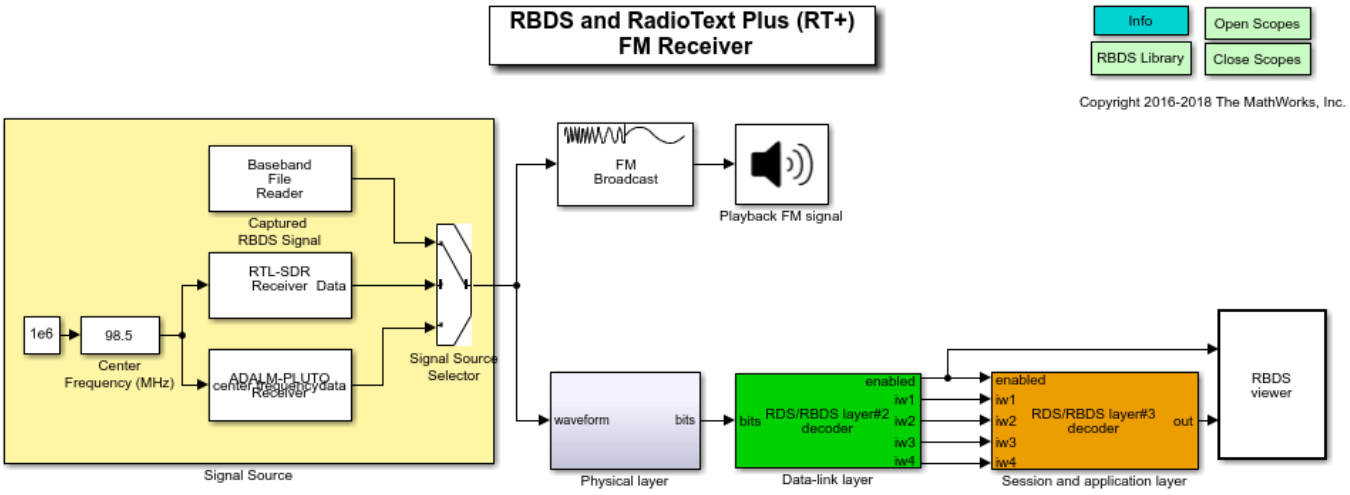
- RTL-SDR radio and the corresponding software Communications Toolbox Support Package for RTL-SDR Radio
- ADALM-PLUTO radio and the corresponding software Communications Toolbox Support Package for ADALM-PLUTO Radio

For a full list of Communications Toolbox supported SDR platforms, refer to Supported Hardware section of Software Defined Radio (SDR).

Background

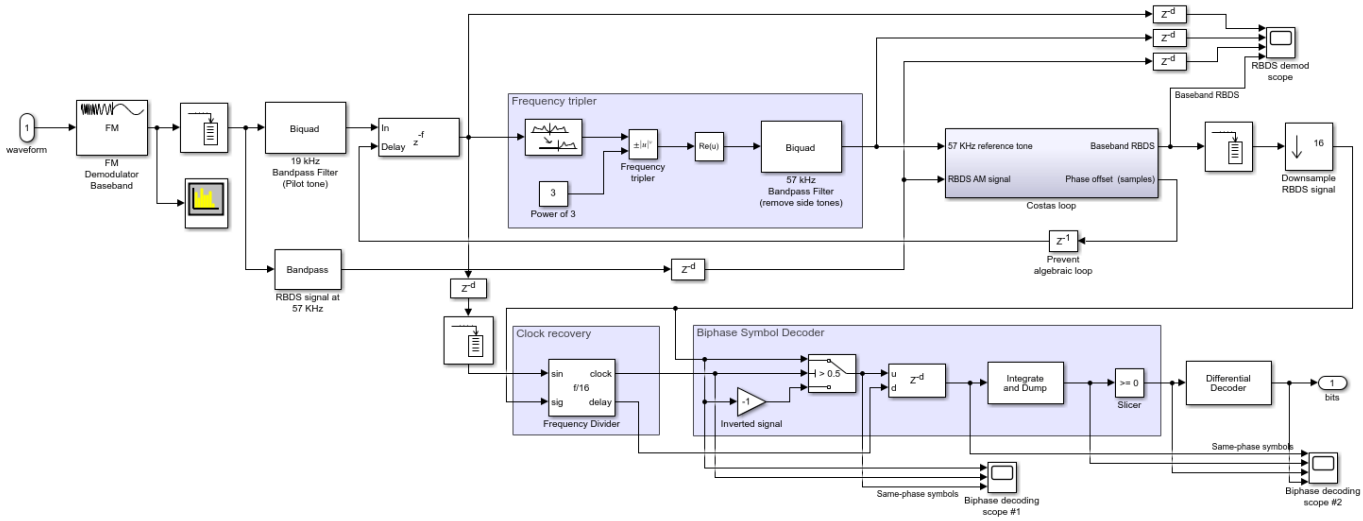
RBDS and RDS are very similar standards specifying how to supplement FM radio signals with additional information. RBDS is used in North America, while RDS was originally used in Europe and evolved to an international standard. RBDS and RDS comprise 3 layers:

- Physical layer (Layer 1)
- Data-link layer (Layer 2)
- Session and presentation layer (Layer 3)

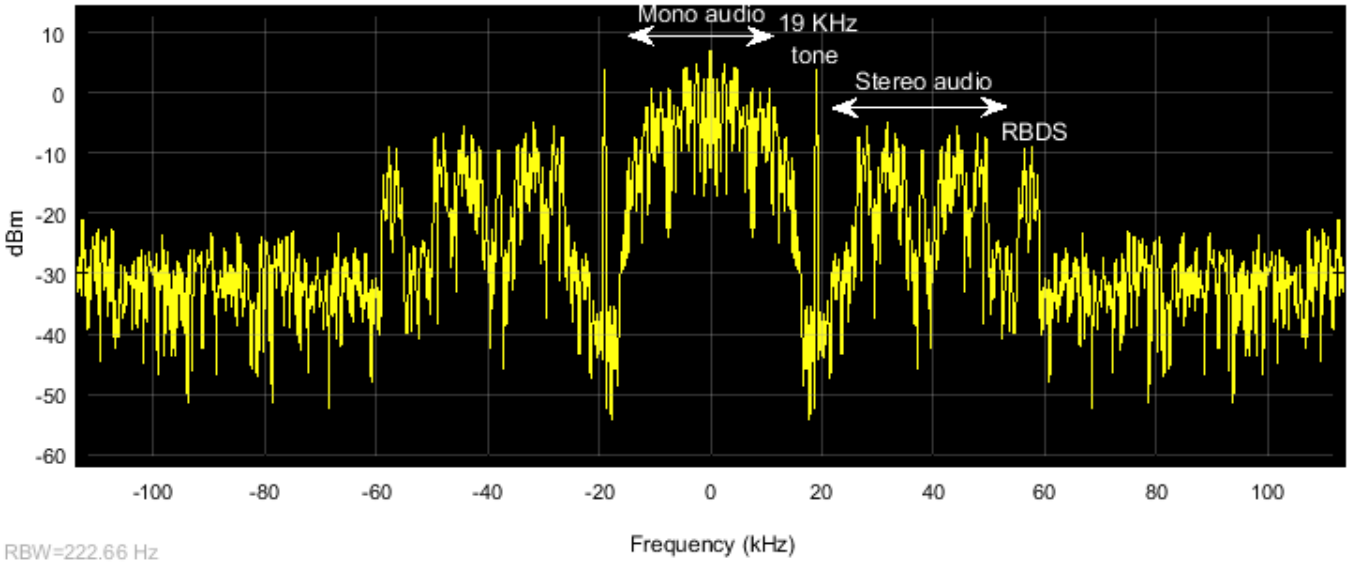


Physical Layer (Layer 1)

The physical layer subsystem receives the captured signal from a file or the live signal from the radio and performs the following steps:



- **FM demodulation**: Once the FM signal is demodulated, the RDS/RBDS signal resides at the 57 kHz +/- 2.4 kHz band:

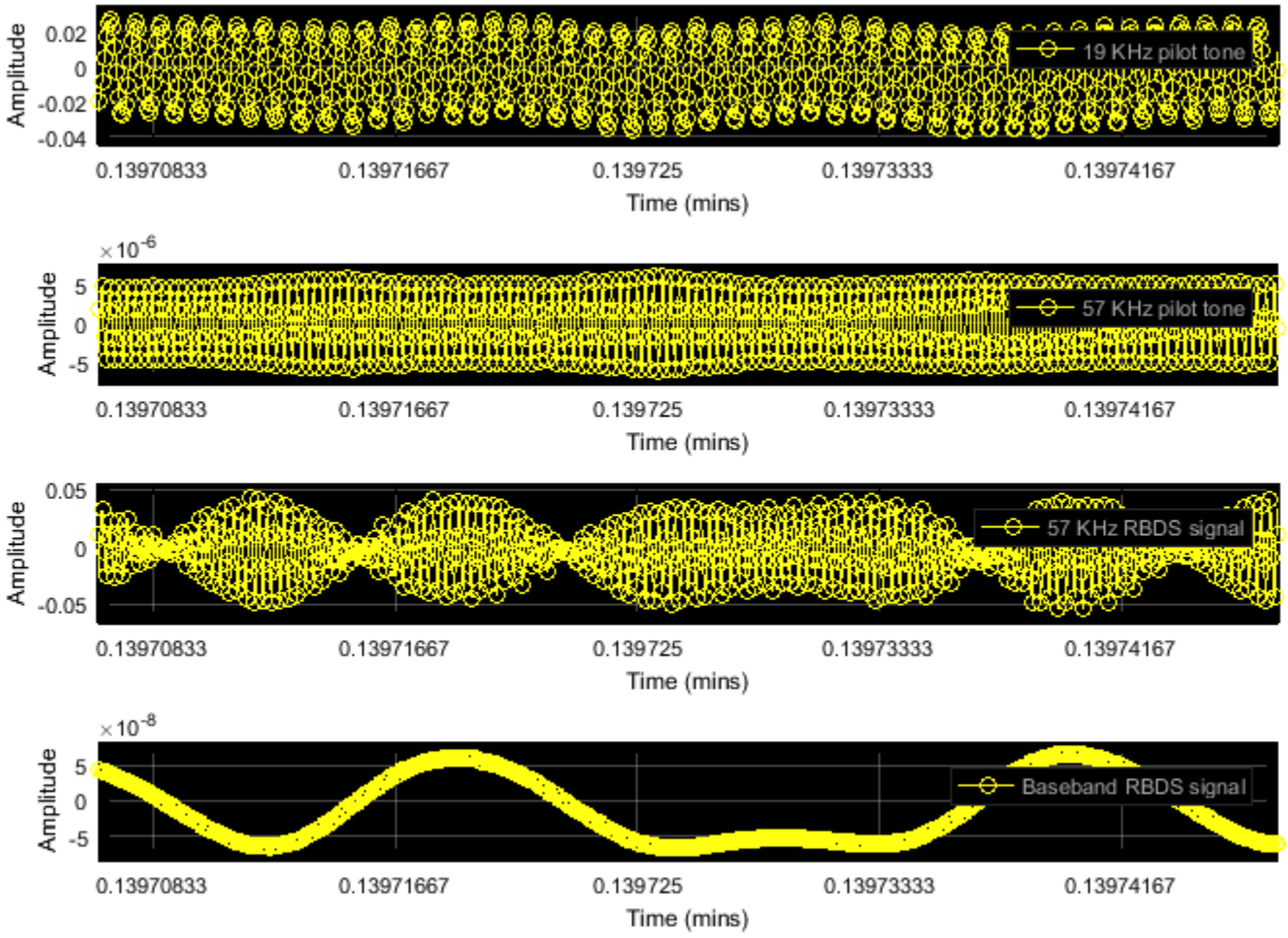


Be aware that the RDS/RBDS signal is transmitted with relatively low power, so it is not always visible in the FM spectrum as in the above figure.

FM signals contain a pilot tone at 19 kHz, which can be used as a phase and frequency reference for coherent demodulation of the RDS/RBDS signal at 57 kHz and the stereo audio at 38 kHz. Pilot tones at 38 kHz and 57 kHz can be generated by doubling and tripling the frequency of the 19 kHz pilot tone [2].

Processing steps for coherent demodulation of the RDS/RBDS signal are:

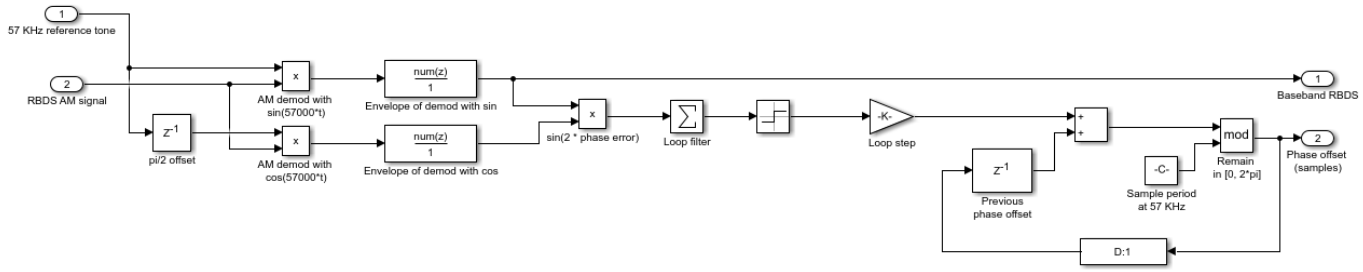
- **Bandpass filtering**: The PHY receiver conducts bandpass filtering at 19 kHz and 57 kHz, to isolate the pilot tone and the RDS/RBDS signal, respectively.
- **Frequency tripling**: Raise the complex representation of the 19 kHz pilot tone to the 3rd power to triple its frequency and obtain a 57 kHz pilot tone.
- **AM Demodulation**: RDS and RBDS symbols are generated at an 1187.5 Hz rate and are AM-modulated to a 57 kHz carrier. The 57 kHz RDS/RBDS signal can be coherently demodulated with a 57 kHz carrier that is locked in frequency and phase. Typically, the frequency-tripled 19 kHz pilot tone suffices for coherent demodulation. The next figures show the 19 kHz and 57 kHz pilot tones, the 57 kHz RDS/RBDS signal, and the AM-demodulated baseband RDS/RBDS signal.



At the same time, there exist several FM stations whose 57 kHz RDS/RBDS signal exhibits a time-varying phase offset from the 19 kHz pilot tone and its frequency-tripled version. The PHY receiver contains a Costas loop to compensate for such time-varying phase offsets.

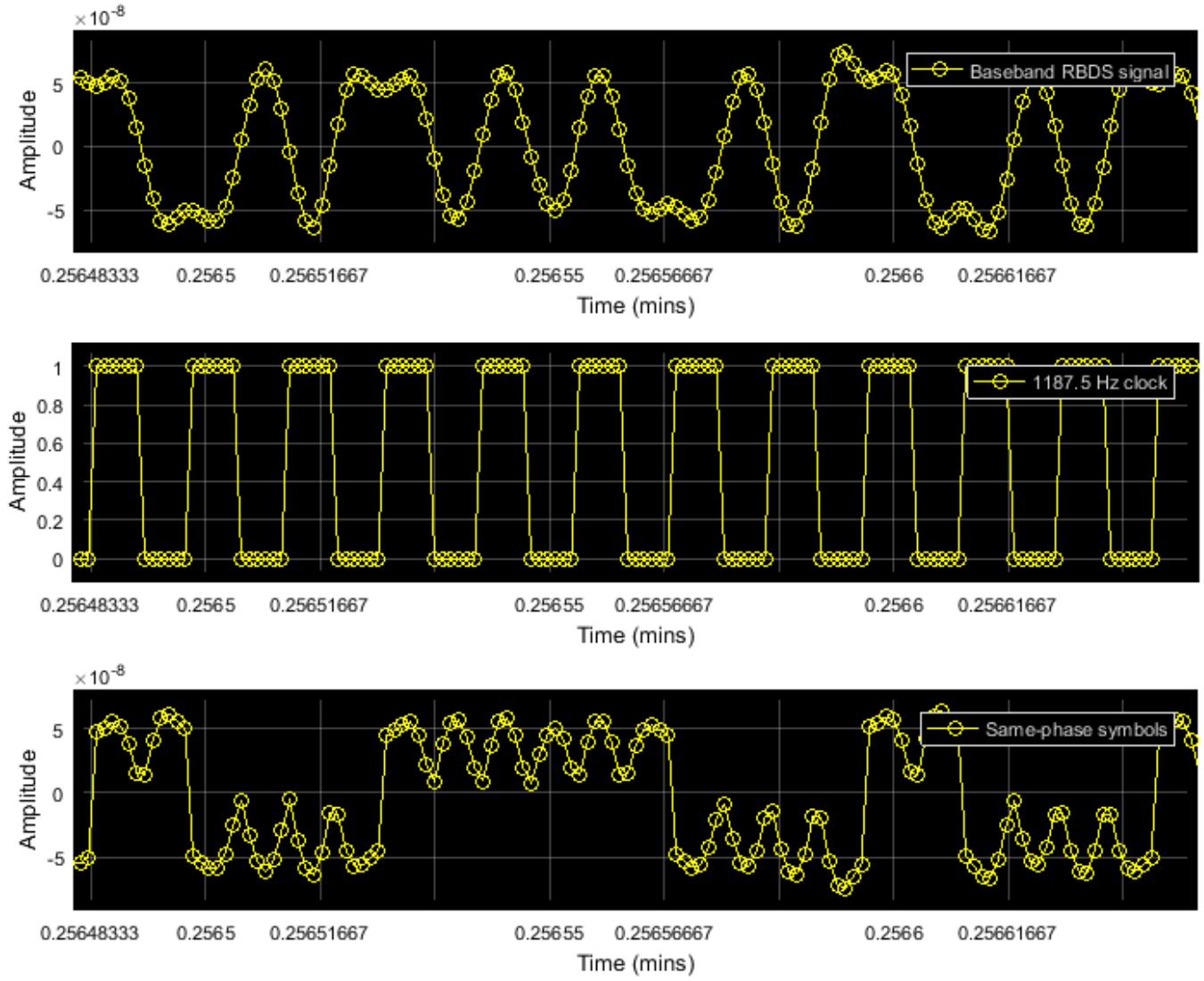
- **Costas loop**: The Costas loop performs 2 orthogonal AM demodulations, one demodulation with a 57 kHz sine and another with a 57 kHz cosine. The sampling rate of the received signal is carefully chosen as 228 kHz, which provides 4 samples per 57 kHz cycle. Therefore, a one sample

delay of the 57 kHz pilot tone results to a one quarter wavelength phase offset, and allows us to generate a cosine wave from a sine wave. The sine-demodulated signal corresponds to the coherent demodulation output. The cosine-demodulated signal is used for detection of phase error. The products of the 57 kHz RDS/RBDS signal with the sine/cosine waves are low-pass filtered with the filter specified in Sec. 1.7 of [1]. The product of the two filter outputs is an error signal. The larger it is, the more the 19 kHz pilot tone is delayed to behave more like the cosine-based demodulator.

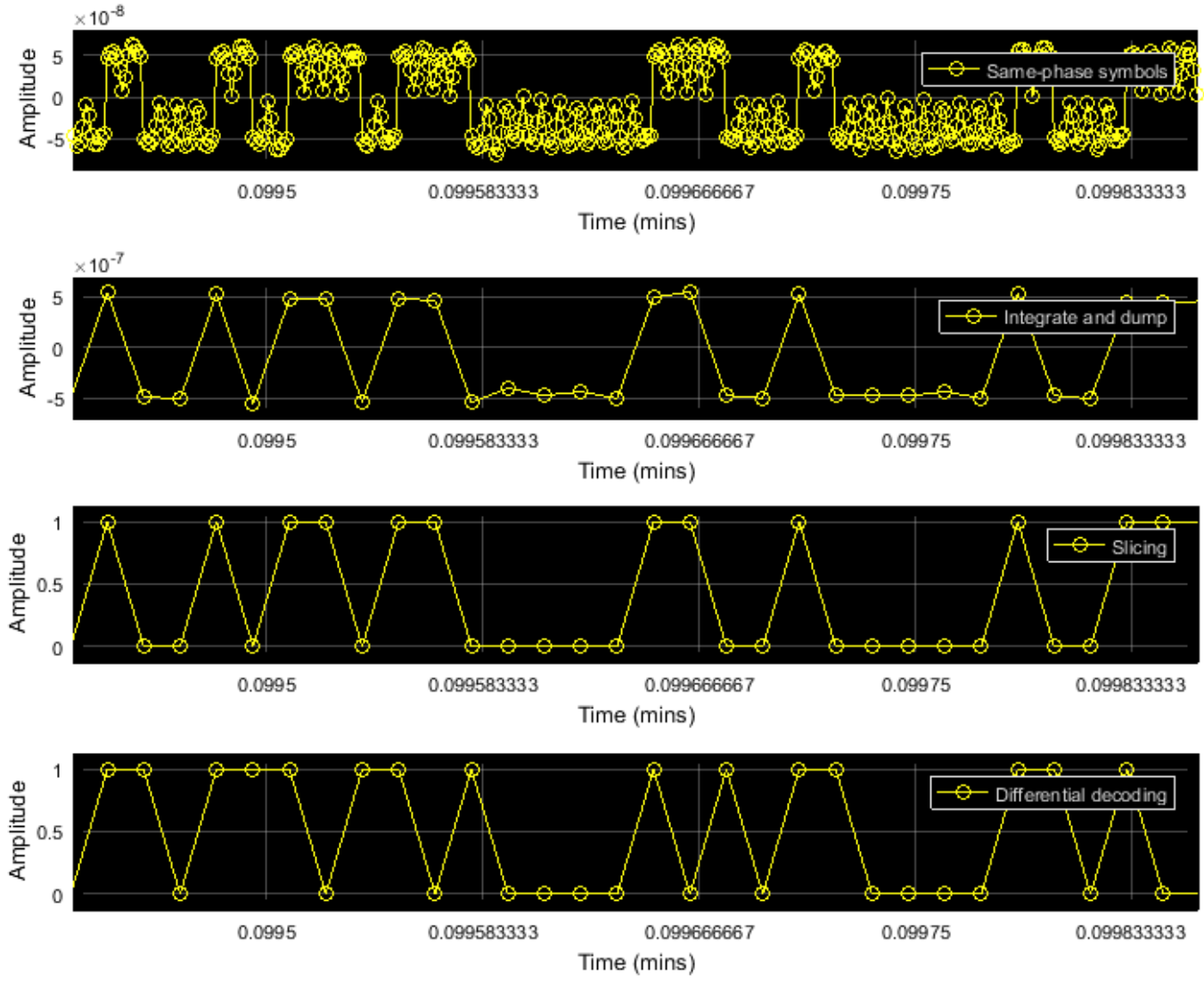


- **Clock extraction**: To perform biphase symbol decoding, a clock matching the RDS/RBDS symbol rate of 1187.5 Hz is extracted from the 19 kHz pilot tone. Note, 1187.5 Hz x 16 = 19 kHz. To account for frequency offsets, frequency division is used to extract the clock from the 19 kHz pilot tone. Since the frequency division operation provides multiple correct answers, the baseband RDS/RBDS signal serves as training data that aid in the determination of the desired output.

- **Biphase symbol decoder**: RDS and RBDS use bi-phase-level (bi- ϕ -L) coding, which is commonly known as Manchester coding. In each clock cycle, the RDS/RBDS symbol takes two opposite amplitude values, either a positive followed by a negative, or a negative followed by a positive. The biphase symbol decoder negates the second amplitude level, so that each symbol holds the same amplitude level throughout the entire clock cycle. The new clock-wide amplitude level corresponds to the symbol's bit representation. The following two screenshots correspond to the waveforms #1-6 in Figure 2 of [1].



To obtain each symbol's bit value, the waveform is integrated over each clock cycle, and the outcome is compared to zero (slicer).



- **Differential decoding**: Finally, the bits are differentially decoded to revert the differential encoding at the transmitter.

Data-link Layer (Layer 2)

Layer 2 is implemented using the RBDSDataLinkDecoder System block. This layer is responsible for synchronization and error correction.

The bit output of the PHY layer is logically organized in 104-bit groups comprising four 26-bit blocks. Each block contains a 16-bit information word and 10 parity bits (see Figure 8 in [1]). A distinct 10-bit offset word is modulo-2 added to the parity bits of each block.

- **Synchronization**: Initially, block and group boundaries are sought exhaustively using a sliding window of 104 bits. For each 104-bit window, the 4 offset words are sought at the last 10 bits of each 26-bit block. An offset word is identified if no bit errors are detected in its block. Once the offset words are identified, group-level synchronization is attained and the exhaustive sliding-window processing stops. Subsequently, the next 104 bits will be treated as the next group.

If future groups contain bit errors and the offset words cannot be identified at their expected position, synchronization may be lost. In this case, Layer 2 first examines the possibility of 1-bit synchronization slips, exploiting the fact that the first information word (16 bits) is always the same for all bit groups. If the first information word is found dislocated by 1 bit (either leftward or rightward), synchronization is retained and the group boundaries are adjusted accordingly. If bit errors persist for 25 group receptions and at the same time synchronization cannot be reestablished using such leftward/rightward 1-bit shifts, then synchronization is lost and Layer 2 re-enters the exhaustive, sliding-window-based search for synchronization.

- **Error correction**: The RDS and RBDS error correction code is a (26, 16) cyclic code shortened from (341, 331). The error correction implementation uses the shift-register scheme described in Annex B of [1].

Session and Presentation Layer (Layer 3)

Layer 2 removes the parity/offset bits, therefore Layer 3 receives groups of 64-bits, comprising four 16-bit blocks. There exist up to 32 different group types, each labeled with a number from 0 to 15 and the letter 'A' or 'B', for example, 0B, 2A, 3A. The format of each group can be fixed or it can be abstract if this group is allocated for an Open Data Application (ODA, see list in [3]).

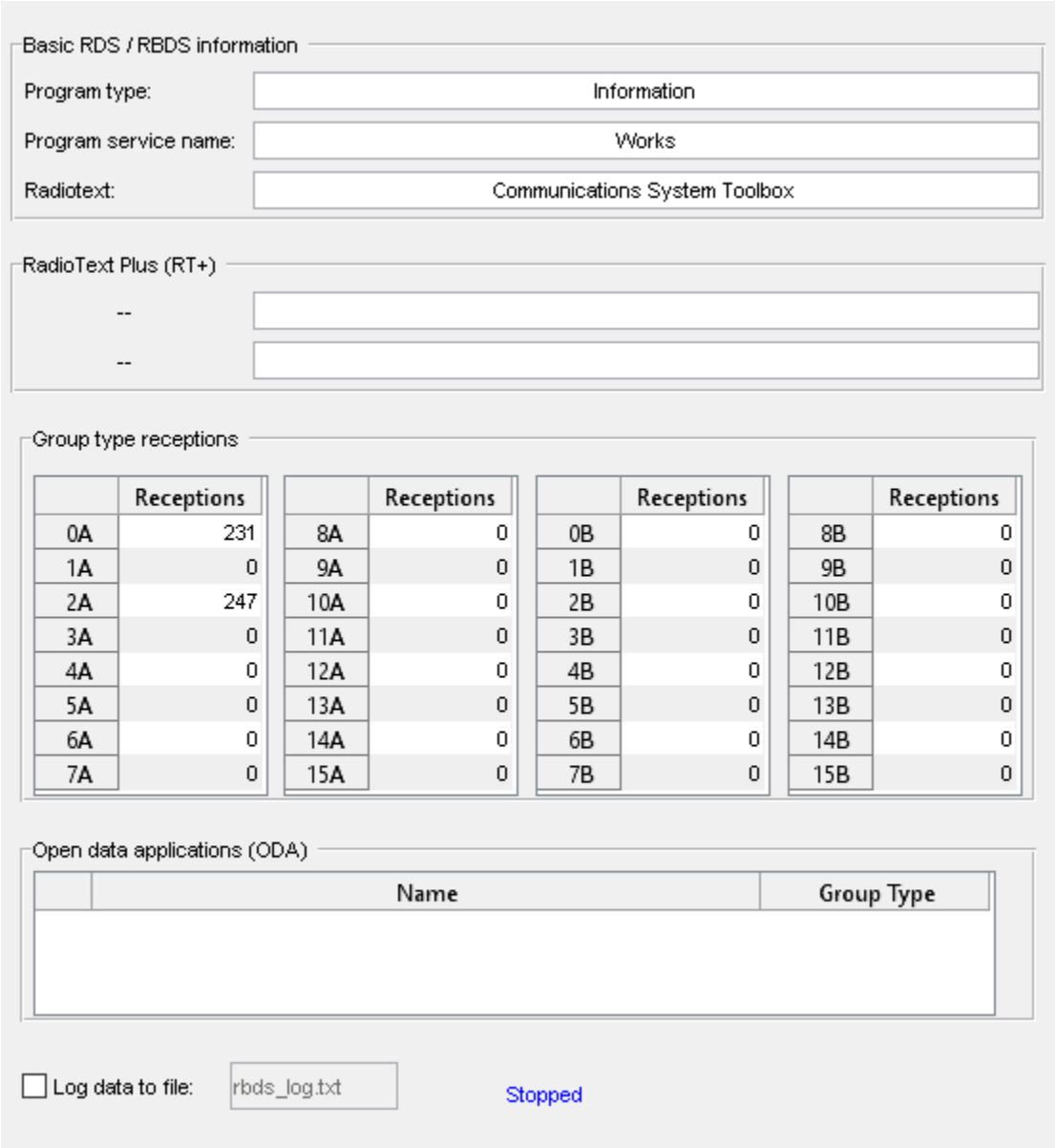
Layer 3 is implemented using the RBDSSessionDecoder System block. This block supports decoding of the 0A, 0B, 2A, 2B, 3A, 4A, 10A fixed-format group types.

- 0A and 0B convey an 8-character string, which typically changes in a scrolling-text fashion.
- 2A and 2B convey longer 64- or 32-character strings.
- 3A registers ODAs and specifies their dedicated abstract-format group type.
- 4A conveys the system time.
- 10A further categorizes the program type (e.g., 'Football' for 'Sports' program type).

For ODAs, the RDS/RBDS receiver supports decoding of RadioText Plus (RT+) [4]. This ODA can break down the long 32 or 64-character string from group types 2A and 2B into two specific content types (for example, artist and song).

Viewing Results

The following screenshot illustrates the graphical display of the processed RDS/RBDS data:



- **Basic RDS/RBDS information**:

1. The first field corresponds to the program type, which is conveyed by the second information word of all group types. If 10A group types are received, the first field also provides further characterization, e.g., Sports \ **Football**.
2. The second field illustrates the 8-character text conveyed by 0A/0B groups.
3. The third field illustrates the longer 32/64-character text conveyed by 2A/2B group types.

- **RadioText Plus (RT+)**: This section is used if any 3A groups indicate that the RadioText Plus (RT +) ODA [4] uses a certain abstract-format group type, e.g., 11A. Then, upon receptions of this abstract group type, the 32/64-character text conveyed by groups 2A/2B will be split to two substrings. Moreover, the two labels will be updated to characterize the substrings (e.g., Artist and Song).

- **Group type receptions**: The tables act as a histogram illustrating which group types have been received from a station and with what frequency. As a result, users may want to look at the logged data for further information that is not depicted in the graphical viewer (e.g., system time in 4A, alternate frequencies in 0A etc.).
- **Open data applications (ODA)**: If any 3A group types are received, then the list of encountered ODAs is updated with the ODA name and their dedicated group type.

Moreover, you can enable the 'Log data to file' checkbox in order to log further fields from all group types.

Selected Bibliography

1. National Radio Systems Committee, United States RBDS standard, April 1998
2. Der, Lawrence. "Frequency Modulation (FM) Tutorial". Silicon Laboratories Inc.
3. National Radio Systems Committee, List of ODA Applications in RDS
4. RadioText Plus (RT+) Specification
5. Joseph P. Hoffbeck, "Teaching Communication Systems with Simulink® and the USRP", ASEE Annual Conference, San Antonio, TX, June 2012

FRS/GMRS Receiver in Simulink

This example shows how to implement a walkie-talkie receiver using Simulink® and Communications Toolbox™. The specific radio standard that this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System). You can use simulated signals, captured signals, or received signals from a commercial walkie-talkie using the Communications Toolbox Support Package for RTL-SDR Radio.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found in the reference list below. Operation in other countries may or may not work.

Required Hardware and Software

To run this example using captured signals, you need the following software:

- Simulink
- Communications Toolbox™

To receive signals in real time, you also need the following hardware:

- RTL-SDR radio
- Walkie-talkie

and the following software

- Communications Toolbox Support Package for RTL-SDR Radio

For a full list of Communications Toolbox supported SDR platforms, refer to the "MATLAB and Simulink Hardware Support for SDR" section of Software-Defined Radio (SDR).

Introduction

For an introduction on FRS/GMRS technology and demodulation of these signals, refer to the “FRS/GMRS Walkie-Talkie Receiver” on page 8-504 example.

Running the Example

To run the example using simulated signals, select the `FRS/GMRS Signal Generator` block as the source using the `Signal Source Selector` block. Double click the `FRS/GMRS Signal Generator` block to select the `CTCSS code` and source type as one of 'Single tone', 'Chirp', or 'Audio'. Then click the run button.

To run the example using captured signals, select the `FRS/GMRS Captured Signal` block as the source using the `Signal Source Selector` block. Then click the run button.

To run the example using the RTL-SDR radio as the source, select the `RTL-SDR Receiver` block as the source using the `Signal Source Selector` block. Then click the run button. Turn on your walkie-talkie, set the channel to be one of the 14 channels (numbered 1 to 14) and the private code to be either one of the 38 private codes (numbered 1 to 38) or 0, in which case no squelch system is used and all received messages are accepted. Note that the private codes above 38 are digital codes and are not implemented in this example.

Double-click the `Channel Number` block and select the same channel number as the walkie-talkie. Double-click the `CTCSS Code` block and set the CTCSS Code to the private code you set in the

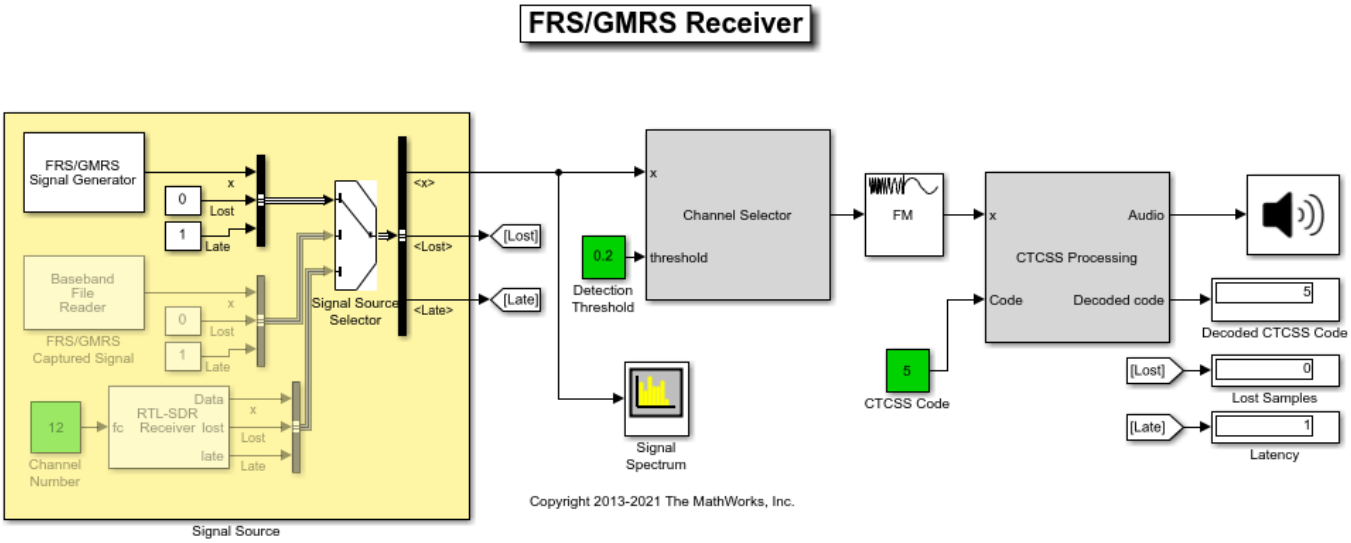
walkie-talkie. Run the model, and see if you can hear your voice come out of the computer speakers. If not, try adjusting the `Detection Threshold` block value downward slightly. You can change the channel and private code without stopping and restarting the model.

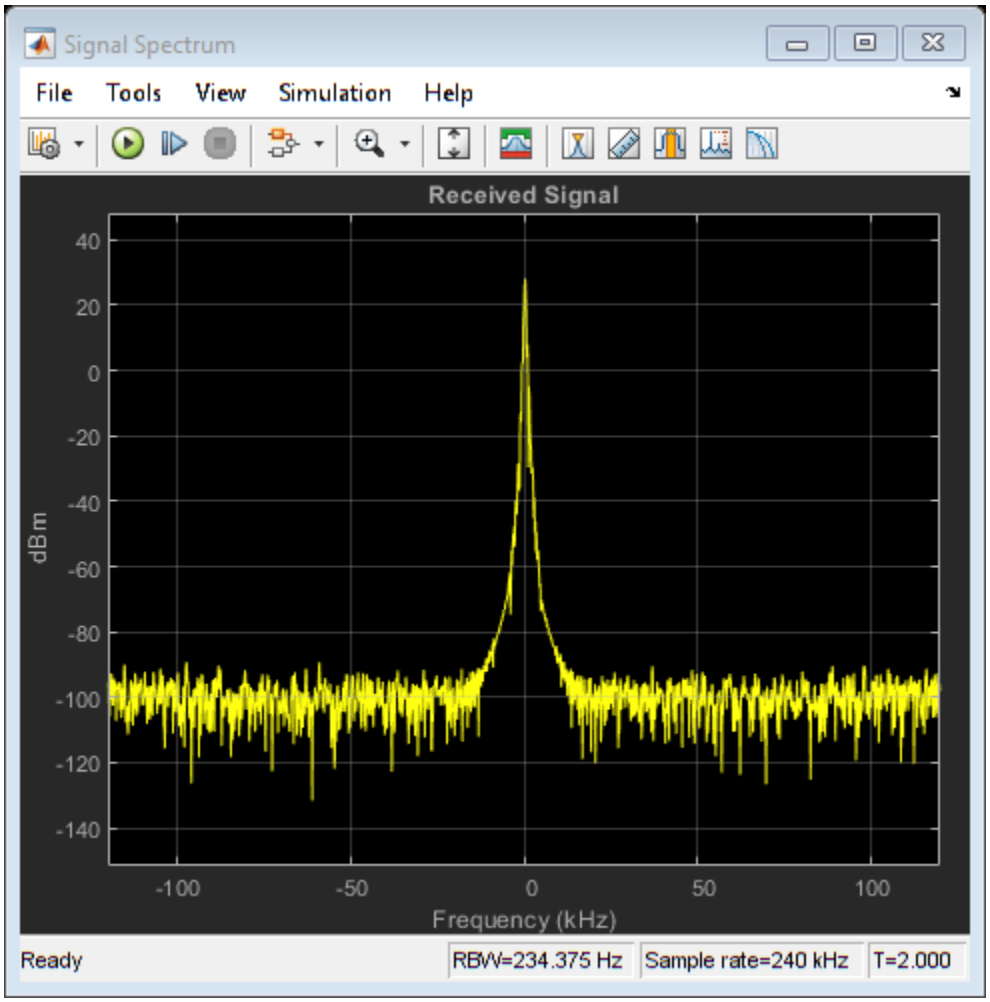
If you hear some dropouts or delay in the sound, run the model in Accelerator mode. From the model menu, select Simulation->Accelerator, then click the run button. If you still experience dropouts or delay in Accelerator mode, try running the model in Rapid Accelerator mode.

The "Signal Spectrum" shows the spectrum of the received signal at the input of the `Channel Selector` block. You can observe how the spectrum changes as you speak into your walkie-talkie.

Receiver Structure

The following block diagram summarizes the receiver structure. The processing has four main parts: Signal Source, Channel Selector, FM Demodulator, and CTCSS processing.





Signal Source

This example can use three signal sources:

1. ''Simulated Signal'': Simulated FRS/GMRS signal at 240e3 samples/sec
2. ''Captured Signal'': Over-the-air signals written to a file and sourced using a baseband file reader block at 240e3 samples/sec
3. ''RTL-SDR Radio'': RTL-SDR radio at 240e3 samples/sec. Use a walkie-talkie as a transmitter. Set the channel number to the channel number of your walkie-talkie.

Channel Selector

The receiver removes the DC component and applies a variable gain to the received signal to obtain an approximately known amplitude signal with reduced interference. The receiver then applies a low pass channel separation filter to reduce the signals from adjacent channels. The gap between adjacent channels is 25 kHz, which means the baseband bandwidth is, at most, 12.5 kHz. Thus, we choose the cutoff frequency to be 10 kHz.

Next, a channel selector computes the average power of the filtered signal. If it is greater than a threshold (set to a default of 10%), the channel selector determines that the received signal is from

the correct channel and allows the signal to pass through. In the case of an out-of-band signal, although the channel separation filter reduces its magnitude, it is still FM modulated and the modulating signal will be present after FM demodulation. To completely reject such a signal, the channel selector outputs all zeros.

FM Demodulator

This example uses the `FM Demodulator Baseband` block whose sample rate and maximum frequency deviation are set to 240 kHz and 2.5 kHz, respectively.

CTCSS

First, a decimation filter converts the sampling rate from 240 kHz to 8 kHz. This rate is one of the native sampling rates of your host computer's output audio device. Then, the CTCSS decoder computes the power at each CTCSS tone frequency using the Goertzel algorithm and outputs the code with the largest power. The Goertzel algorithm provides an efficient method to compute the frequency components at predetermined frequencies, that is, the tone code frequencies used by FRS/GMRS.

The model compares the estimated received code with the preselected code and then sends the signal to the audio device if the two codes match. When the preselected code is zero, it indicates no squelch system is used and the decision block passes the signal at the channel to the audio device no matter which code is used.

Finally, a high pass filter with a cutoff frequency of 260 Hz filters out the CTCSS tones, which have a maximum frequency of 250 Hz. Use an `Audio Device Writer` block to play the received signals through your computer's speakers. If you do not hear any sound, please select another device using the DeviceName parameter of the `Audio Device Writer` block.

Audio Output

Before the audio device, a high pass filter with a cutoff frequency of 260 Hz is used to filter out the CTCSS tones (which have a maximum frequency of 250 Hz) so that they are not heard.

The `Audio Device Writer` block is set up by default to output to the current audio device in your system preferences.

Exploring the Example

The CTCSS decoding computes the DTFT (Discrete-Time Fourier Transform) of the incoming signal using the Goertzel algorithm and computes the power at the tone frequencies. Because the tone frequencies are very close to each other (only 3-4 Hz apart) the block length of the DTFT should be large enough to provide enough resolution for the frequency analysis. However, long block lengths cause decoding delay. For example, a block length of 16384 will cause 2 seconds of delay, since the CTCSS decoder operates at an 8 kHz sampling rate. This creates a tradeoff between detection performance and processing latency. The optimal block length may depend on the quality of the transmitter and receiver, the distance between the transmitter and receiver, and other factors. You are encouraged to change the block length in the initialization function by navigating to the helperFRSReceiverConfig function and changing the value of the CTCSSDecodeBlockLength field. This will enable you to observe the tradeoff and find the optimal value for your transmitter/receiver pair.

When the `FRS/GMRS Signal Generator` is selected as the source, you can change the `CTCSS tone amplitude` parameter of this block and observe how this affects the signal spectrum.

Appendix

The following script is used in this example:

- helperFRSReceiverConfig.m

References

- Family Radio Service
- General Mobile Radio Service
- Continuous Tone-Coded Squelch System
- Goertzel Algorithm

ALOHA and CSMA/CA Packetized Wireless Networks

This example shows how to simulate a basic ALOHA or CSMA/CA MAC using Simulink®, Stateflow® and the Communications Toolbox™.

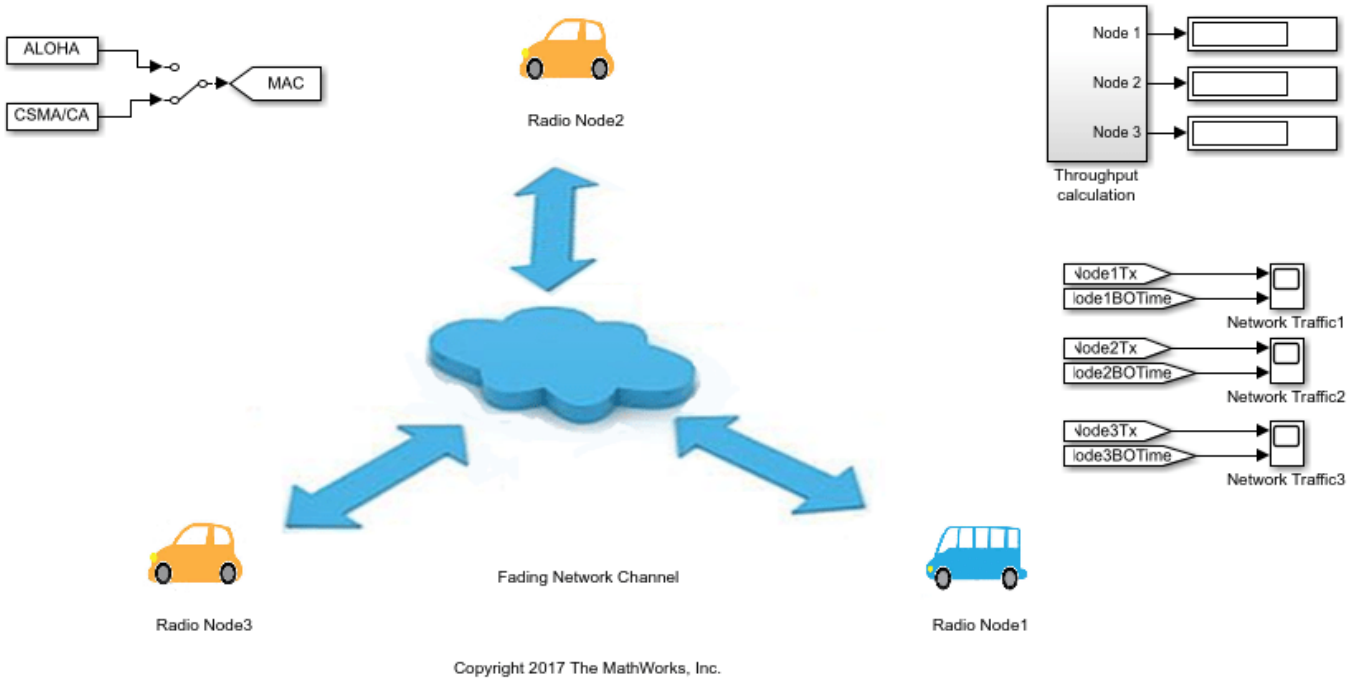
Background

ALOHA: ALOHA is a seminal random-access protocol that became operational in 1971. In ALOHA, nodes transmit packets as soon as these are available, without sensing the wireless carrier. As a result, wireless packets may collide at a receiver if they are transmitted simultaneously. Hence, successful packet reception is acknowledged by transmitting a short acknowledgment packet. If an acknowledgment is not received timely enough, then the data packet is resent at a later instant determined, e.g., by binary exponential backoff.

CSMA/CA: Carrier Sense Multiple Access with Collision Avoidance is an improved random-access scheme, according to which wireless nodes first sense the wireless medium before transmitting their data packets. If the medium is sensed busy, then transmissions are deferred, e.g., according to a binary exponential backoff. Collision avoidance is enabled by: *(i)* waiting for an interframe spacing (IFS) duration after the channel has been sensed idle, *(ii)* transmitting only after a certain number of (not necessarily contiguous) sensed idle time slots, chosen randomly from the contention window (i.e., an adaptive range of possible backoff durations), *(iii)* exchanging Request-to-Send and Clear-to-Send frames (RTS and CTS). Out of these three methods, this example models the first two (IFS and contention window). CSMA/CA has been employed in Ethernet, IEEE® 802.11, and IEEE 802.15.4, among other standards.

Overview

This example models a three-node PHY/MAC network. All nodes are within range; transmissions between two nodes can be received by and interfere with the third one.



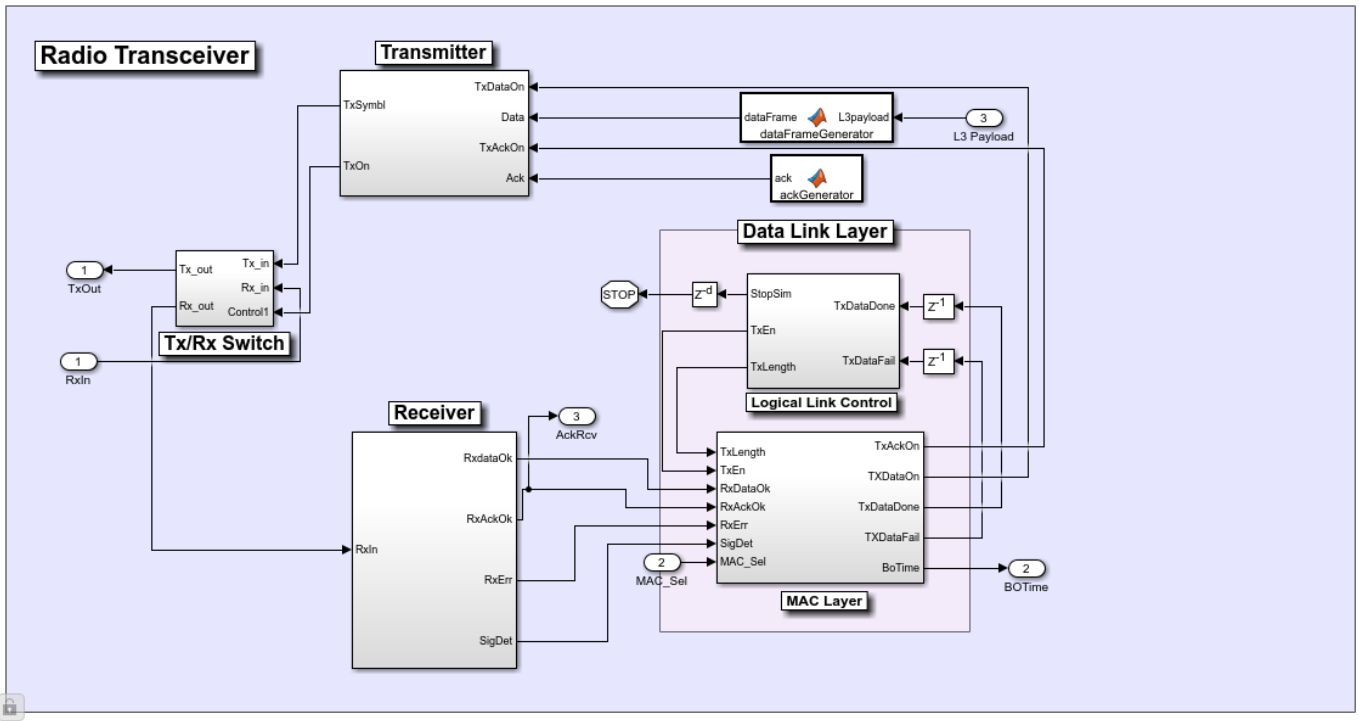
The default configuration enables data frame transmissions from node 1 to node 3, from node 3 to node 2, and from node 2 to node 1. Acknowledgment frames are transmitted from node 3 to node 1, from node 2 to node 3, and from node 1 to node 2.

The MAC scheme can be either ALOHA or CSMA/CA, as determined by the top-level switch. MAC frames are encoded to or decoded from a PHY waveform using a QPSK-based PHY layer.

The MAC layer operates at a very fine timescale (every 0.8 microseconds), as the backoff duration is typically much shorter than the duration of a data frame. As a result, the Simulink model is scalar-based (i.e., the length of most signals is equal to 1) and the MAC/PHY layers do not process frames, i.e., batches of samples.

Radio Transceiver

Each radio transceiver is a joint PHY and MAC implementation enabling both receive and transmit operations. The left side of the next diagram corresponds to the PHY layer, while the right side corresponds to the Data Link Layer (MAC and Logical Link Control).

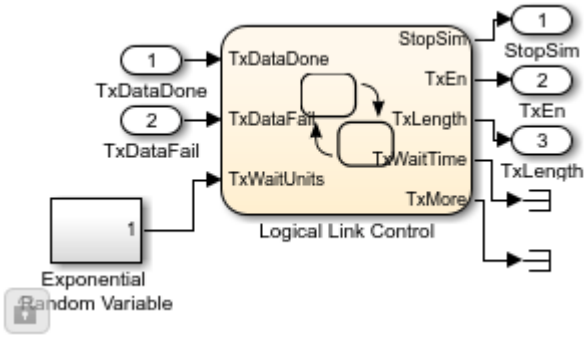


On the receive-side chain, the transceiver decodes the PHY layer of the received waveforms and passes the corresponding MAC Protocol Data Unit (MPDU) to the MAC layer, which processes data and acknowledgment frames.

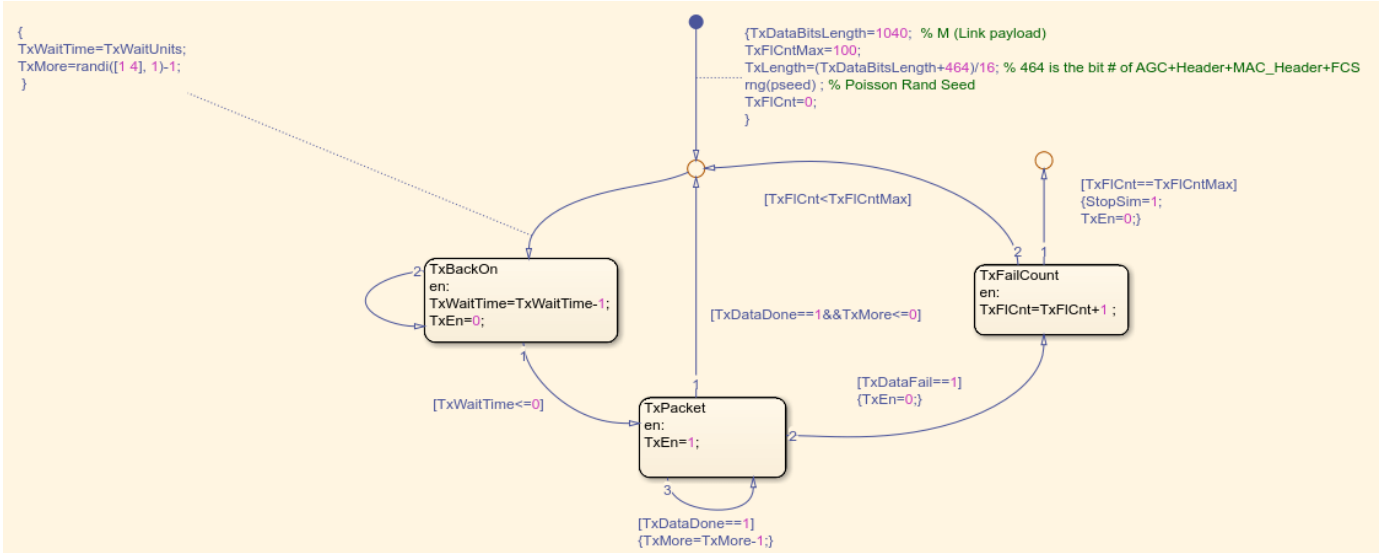
On the transmit-side chain, the Data Link layer initiates MAC frame transmissions either when the Logical Link Control sublayer determines that a new data frame is injected or when the MAC sublayer needs to transmit an acknowledgment for a received data frame. The data MAC frames are generated by prepending a MAC header and appending a CRC MAC footer to a payload that is the input from the higher, third layer (network layer). The acknowledgment MAC frames do not contain a payload; they only contain the MAC header and CRC footer.

Logical Link Control

The Logical Link Control (LLC) sublayer is responsible for injecting data packets into the transceiver. It is mainly implemented using a Stateflow chart. The packet interarrival time is exponentially distributed, which corresponds to a Poisson process.

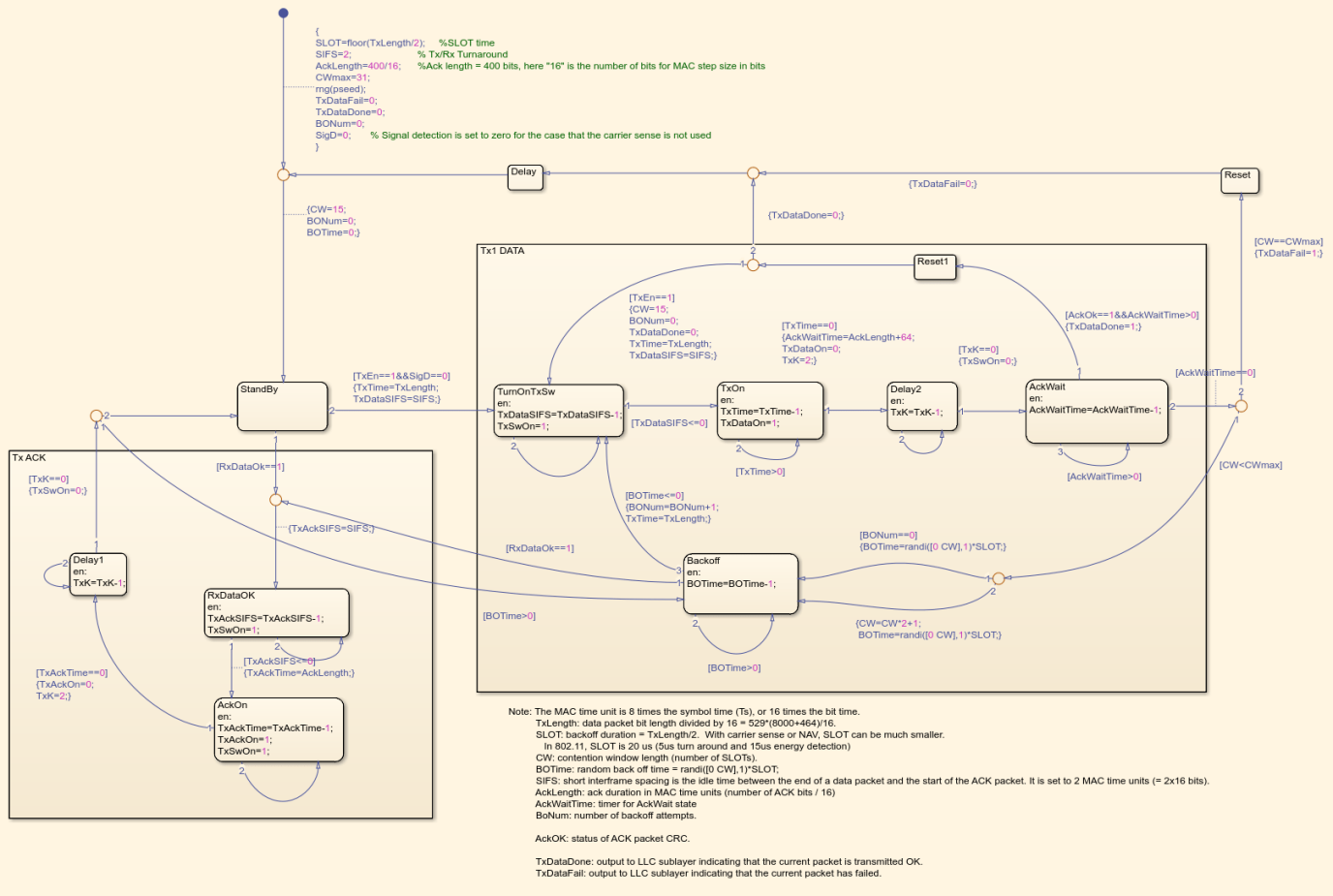


Then, the Stateflow chart counts down the packet inter-arrival time until the next packet arrives. This chart also models the segmentation of large packets into smaller data frames by determining the number of additional frame transmissions ("TxMore").



ALOHA MAC Layer

When the top-level MAC switch is set to ALOHA, the MAC subsystem of the Data link layer essentially operates as the following Stateflow chart:

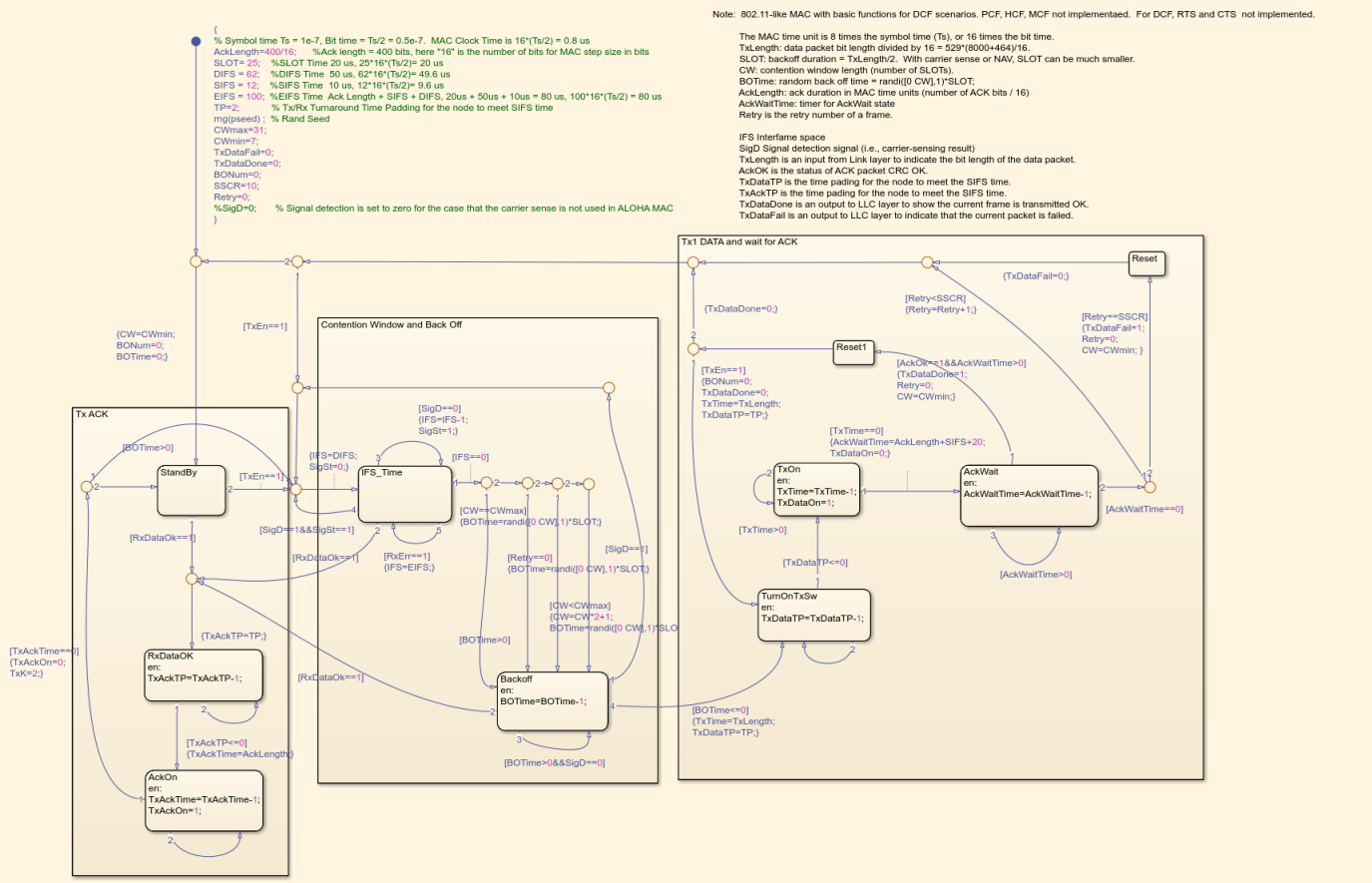


The left side of the chart is responsible for acknowledging a received data frame. Before transmitting the acknowledgment, the transmitter first waits for a short interframe spacing (SIFS). Then, it outputs a positive 'TxAckOn' signal for the duration of the acknowledgment frame.

The right side of the chart is responsible for transmitting a data frame. Before transmitting the data frame, the transmitter first waits for a short interframe spacing (SIFS). Then, it transmits the signal, **without** sensing the wireless medium, by outputting a positive 'TxDataOn' signal for the duration of the data frame. Subsequently, the node awaits to receive an acknowledgment within a certain time interval. If the acknowledgment is received before timeout, the current data frame transmission is concluded. If it is not, then the node enters a backoff state and it doubles its contention window (CW) every time except for the first backoff instance. The backoff duration is randomly chosen from the [0, CW] interval. If the maximum number of backoff attempts is reached, then the transceiver declares a failure in transmitting this data frame.

CSMA/CA MAC Layer

When the top-level MAC switch is set to CSMA/CA, the MAC subsystem of the Data link layer essentially operates as the following Stateflow chart:

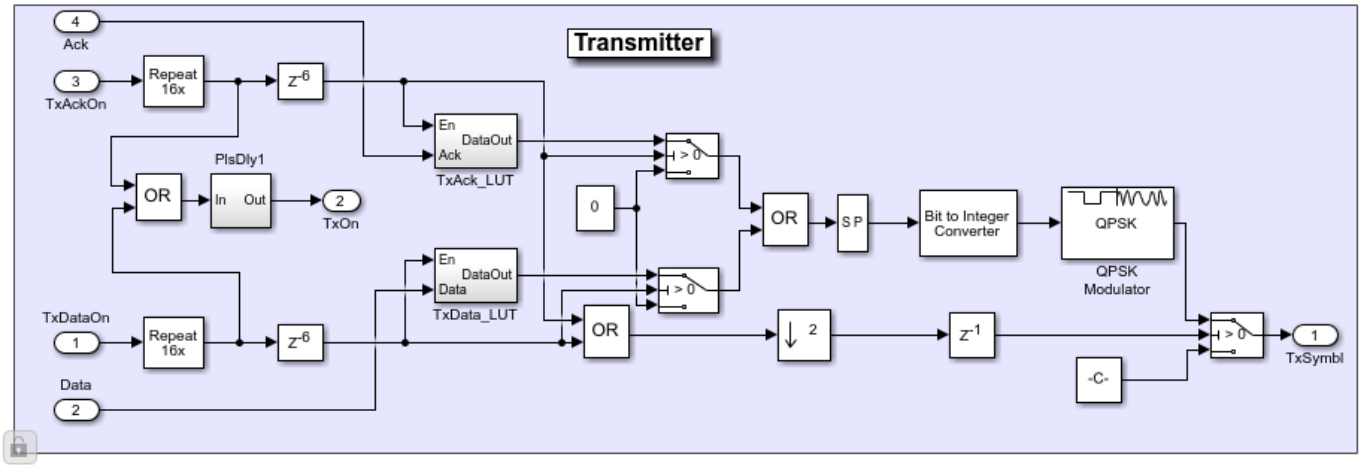


The CSMA/CA chart has some similarities with the ALOHA chart, but it also has some differences:

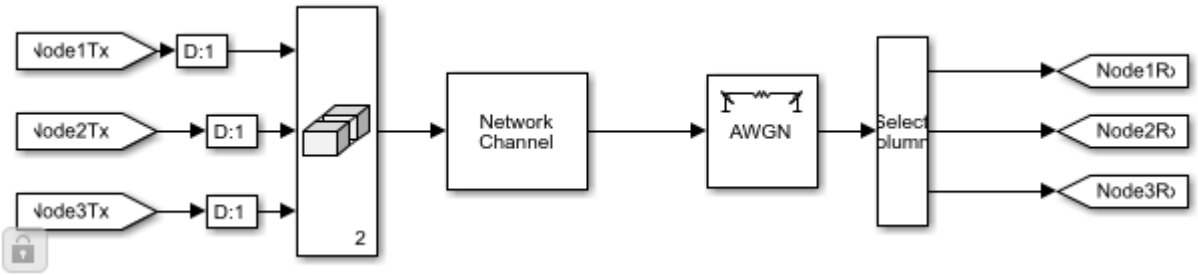
- The transceiver senses the wireless medium.
- Data frames are not transmitted before an interframe spacing (IFS) duration elapses since the wireless medium has been sensed as idle.
- The backoff counter decrements only when the medium is sensed idle.

Physical Layer

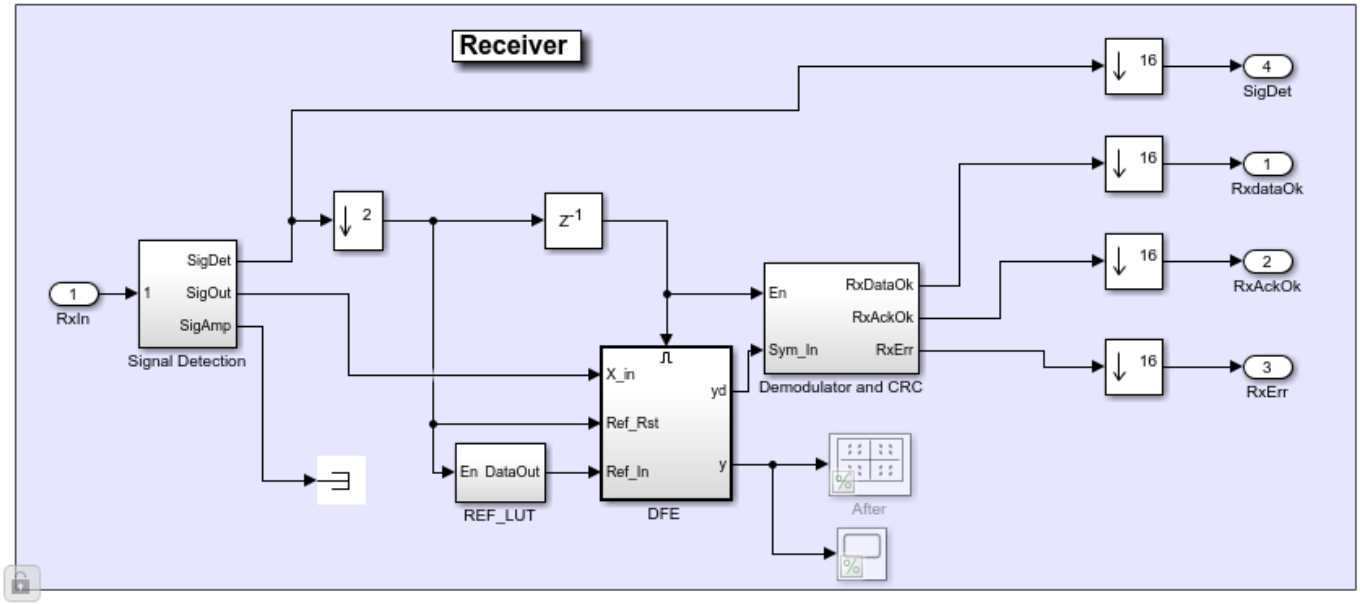
Transmitter: The transmitter performs QPSK modulation on the MPDU bits. The bit rate is 20 MHz and the symbol rate is 10 MHz. The QPSK symbols are subsequently filtered with the raised cosine filter of the "Tx/Rx Switch" subsystem.



Channel: The filtered PHY waveform passes through a network channel, which imposes multipath fading and white gaussian noise. The network channel allows each node to receive superimposed signals transmitted by multiple other nodes. Multipath fading is applied using the NetworkChannel System block. White noise is added using the multichannel capability of the AWGN Channel block.

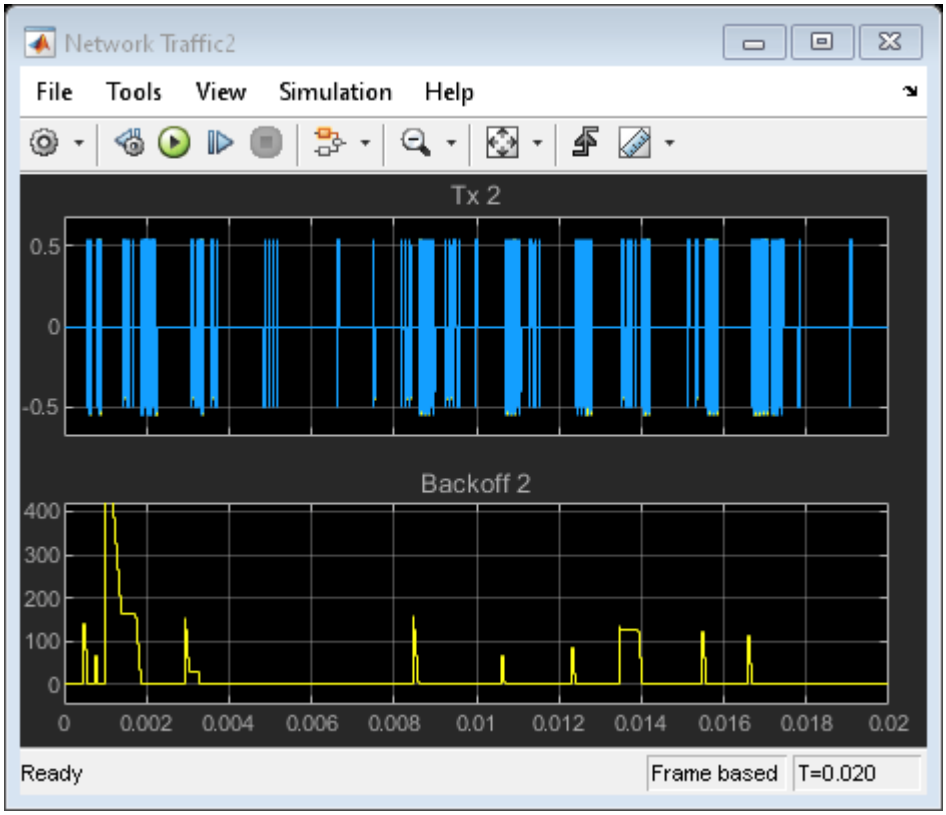


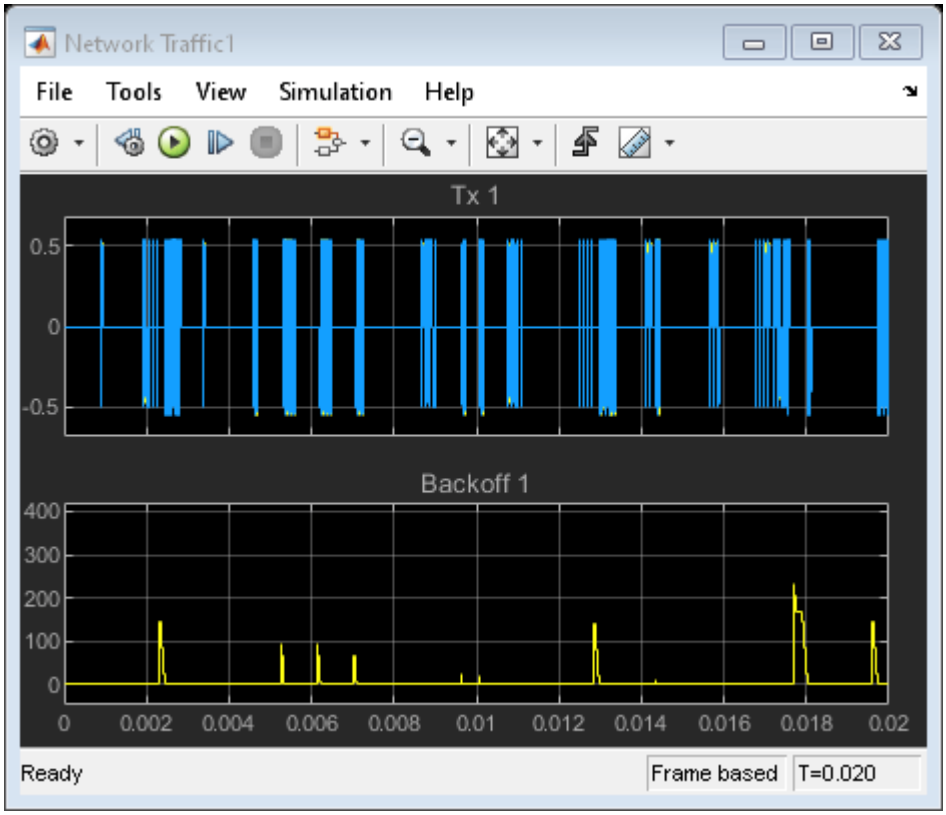
Receiver: Transceivers process the signal waveform only when its amplitude exceeds a certain threshold (see Signal Detection subsystem). Subsequently, the received waveform is equalized using a Decision Feedback Equalizer (DFE); this component reduces intersymbol interference (ISI) caused by multipath fading, corrects small symbol timing offsets and carrier offsets, and its fast convergence suits packetized networks. Next, the equalized QPSK symbols are demodulated. The corresponding bits are passed to a CRC detector in order to identify the frame start, the PHY payload length and the frame type (data or acknowledgment).

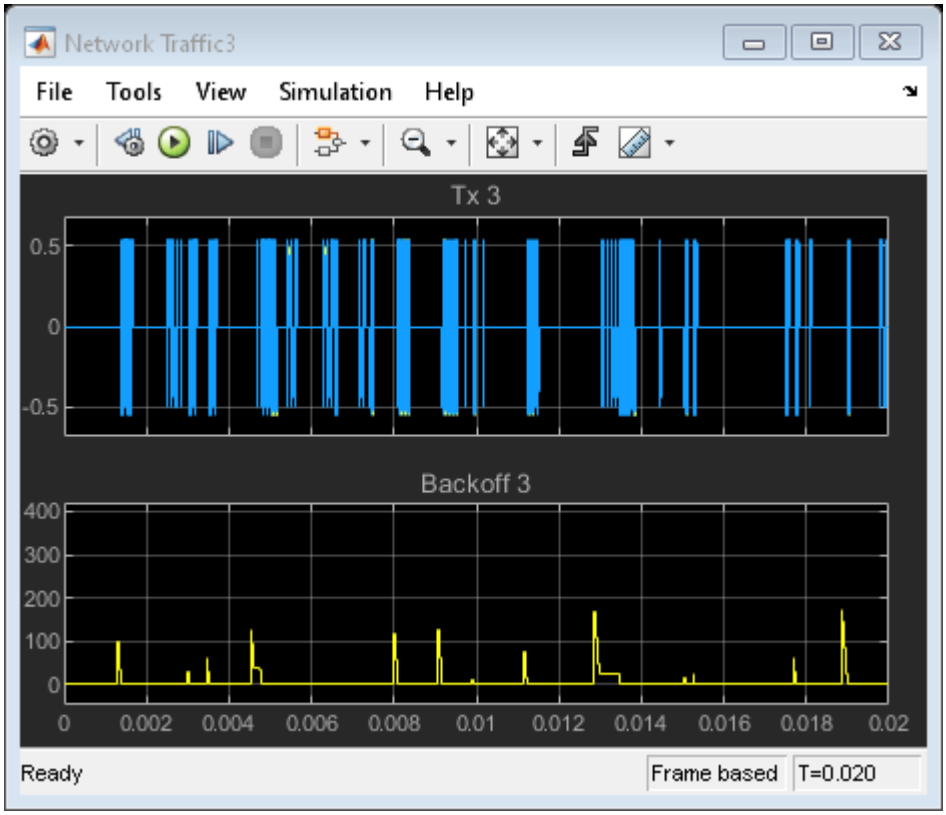


Simulation Results

Model simulation shows one scope for each transceiver. Each scope depicts the transmitted signal (top axes) and the backoff counter (bottom axes) for each transceiver.




Network Traffic1
File Tools View Simulation Help
Tx 1
0.5
0
-0.5
Backoff 1
400
300
200
100
0
0 0.002 0.004 0.006 0.008 0.01 0.012 0.014 0.016 0.018 0.02
Ready
Frame based
T=0.020


Network Traffic3
File Tools View Simulation Help
Tx 3
0.5
0
-0.5
Backoff 3
400
300
200
100
0
0 0.002 0.004 0.006 0.008 0.01 0.012 0.014 0.016 0.018 0.02
Ready
Frame based
T=0.020

At the same time, the top-level model depicts per-node throughput in three display blocks. Throughput is calculated by measuring the number of successfully acknowledged data packets.

Further Exploration

- The used MAC scheme can be toggled between ALOHA and CSMA/CA (default). Changing the MAC scheme to ALOHA yields lower node throughput for the default packet arrival rates. This is because ALOHA packets collide more frequently as nodes do not sense the wireless carrier.
- The packet arrival rates can be customized through the dialog mask of each node. The network saturation point can be empirically and iteratively found, e.g., by gradually increasing the same packet arrival rate for each node. Increasing low arrival rates can increase node throughput; increasing high arrival rates (past the saturation point) can actually have a detrimental effect on throughput as packets collide and nodes backoff more frequently.
- If the arrival rates are disproportional for each node, then unfairness scenarios can be established. For example, one node may be capturing the medium very frequently and maintain a low contention window, while other nodes may back off for a long time and only sporadically access the medium.
- You can change the random seed of the nodes at their block mask to enable different random-access scenarios. For example, for a given packet arrival rate, the random seed determines how soon the first transmission occurs.

Selected Bibliography

1 N. Abramson, The ALOHA System Final Technical Report, NASA Advanced Research Projects Agency, October 11, 1974

2 IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Nov. 1997. P802.11

Multicore Simulation of Comparing Demodulation Types

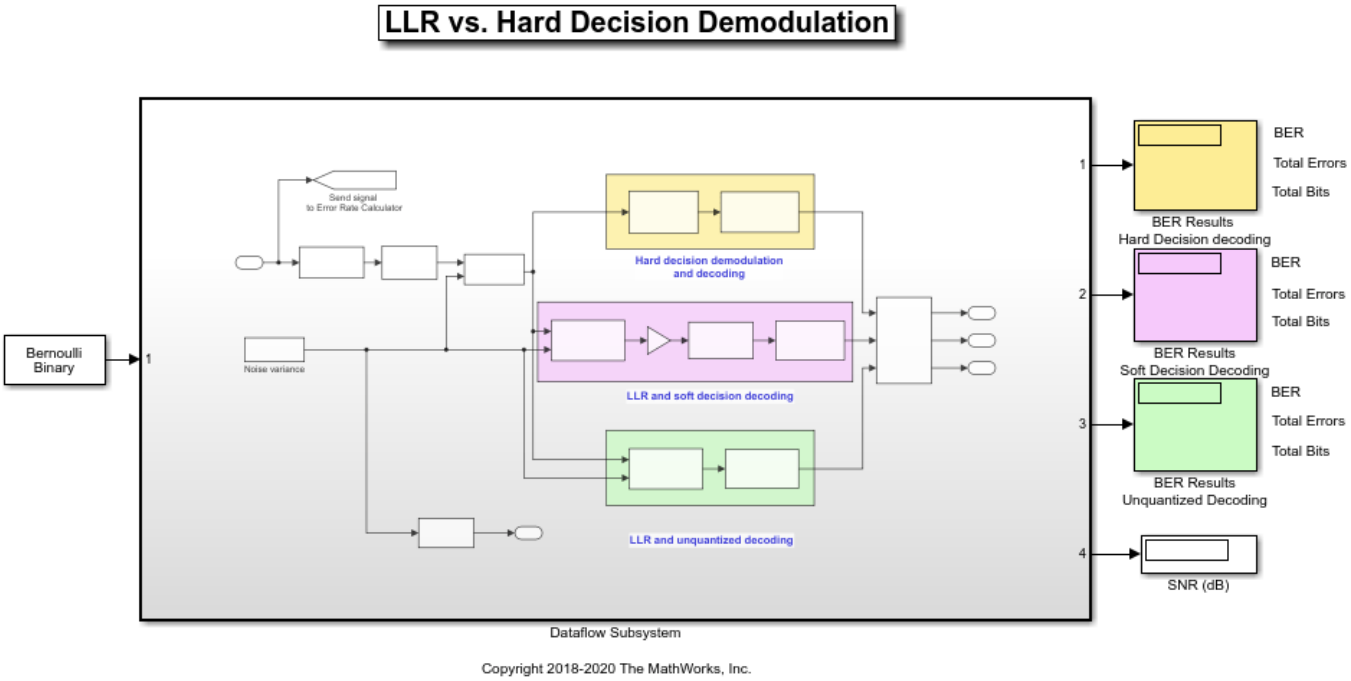
This example compares an LLR and hard decision demodulation. It uses the dataflow domain in Simulink® to automatically partition the data-driven portions of the communications system into multiple threads and thereby improving the performance of the simulation by executing it on your desktop's multiple cores.

Introduction

The dataflow execution domain allows you to make use of multiple cores in the simulation of computationally intensive systems. This example shows how dataflow as the execution domain of a subsystem improves simulation performance of the model. To learn more about dataflow and how to run Simulink models using multiple threads, see “Multicore Execution using Dataflow Domain”.

LLR vs Hard Decision Demodulation

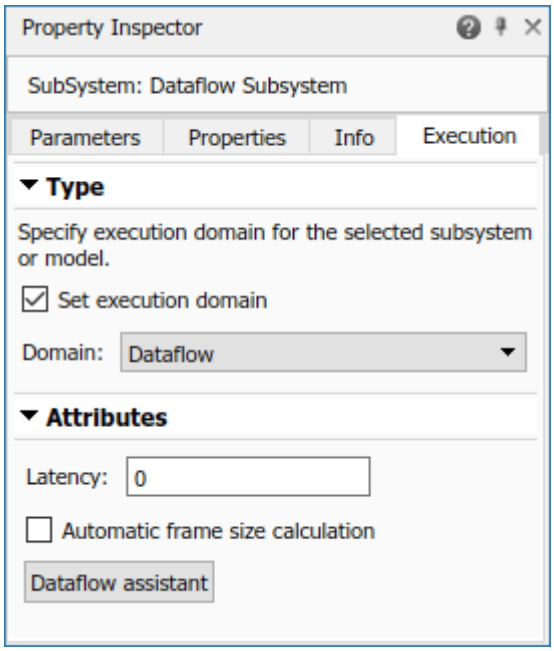
This example shows a communication system that compares BER performance when using LLR instead of hard decision demodulation in the decoder. This example has one transmitter, an AWGN channel and three receivers. The three receivers use different decoding techniques to compare the BER of each approach. Bit error rate computation is shown in Display blocks for comparing the performance of the three receivers.

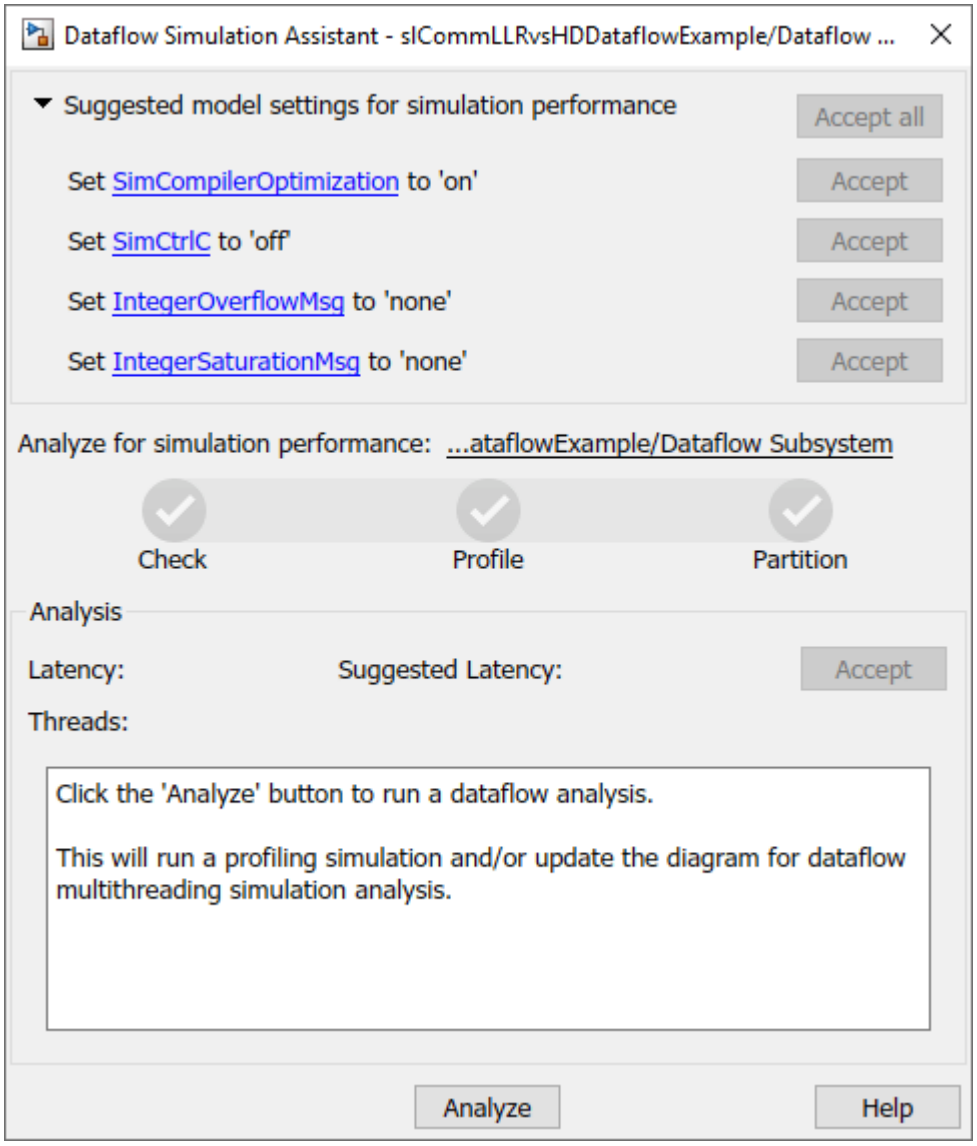


Setting up Dataflow Subsystem

This example uses dataflow domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. The Domain parameter of the Dataflow Subsystem in this model is set as **Dataflow**. You can view this by selecting the subsystem and then selecting **View>Property Inspector**. Dataflow domains automatically partition your model and simulate the system using multiple threads for better simulation performance. Once you set the Domain parameter to Dataflow, you can use Dataflow Simulation Assistant to analyze your model to get better performance. You can

open the Dataflow Simulation Assistant by clicking on the **Dataflow assistant** button below the **Automatic frame size calculation** parameter in Property Inspector.





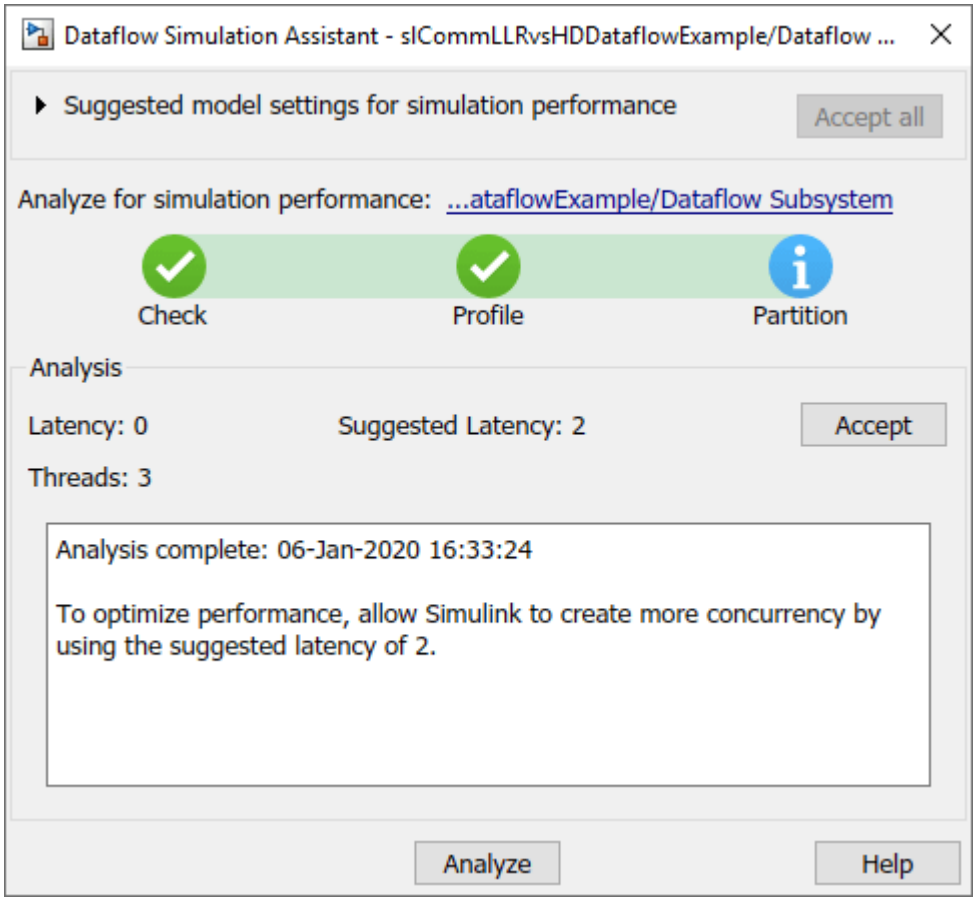
Analyzing Concurrency in Dataflow Subsystem

The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. In this example the model settings are already optimal. In the Dataflow Simulation Assistant, click the **Analyze** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Dataflow Simulation Assistant shows how many threads the dataflow subsystem will use during simulation.

After analyzing the model, the assistant shows three threads because the three different receiver types can run independently in parallel. When Latency used is zero, dataflow can only use this inherent parallelism in the model. The three receivers are data dependent on one transmitter. This causes bottleneck since the transmitter needs to complete its processing before any receivers start processing. Without pipeline delays only the inherent parallelism in the model can be utilized to run Dataflow Subsystem using multiple threads. By pipelining the data dependent blocks, the Dataflow

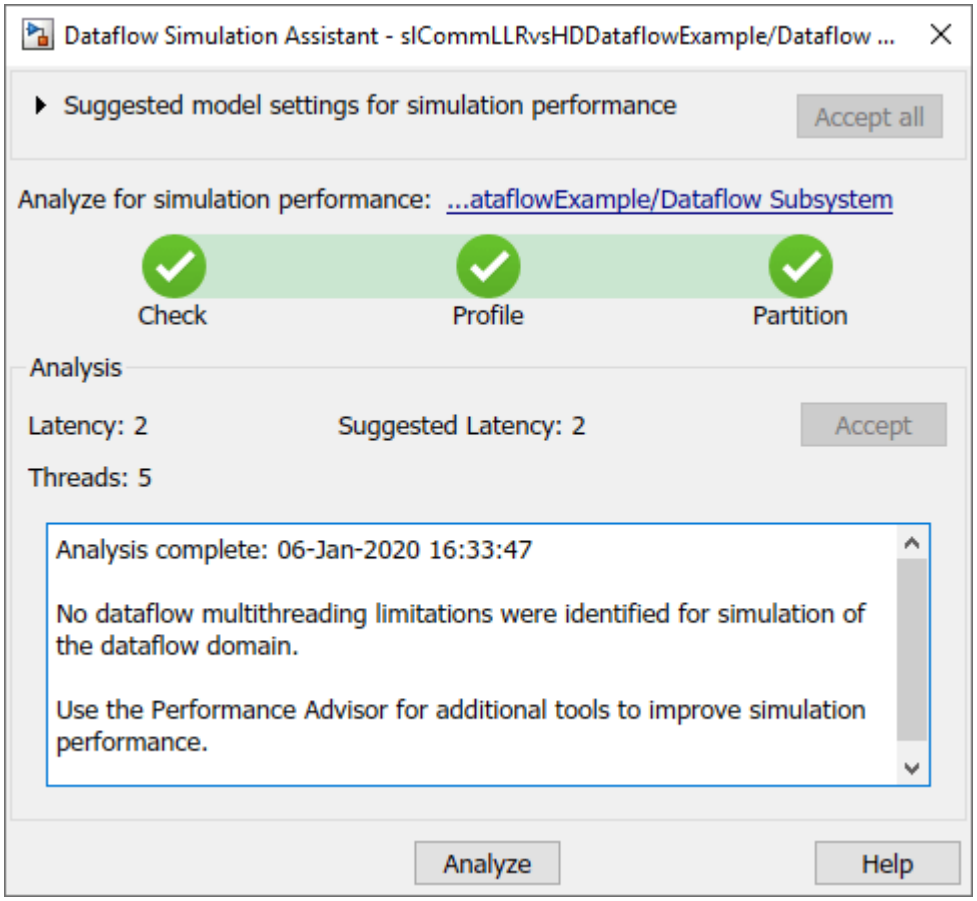
Subsystem can increase concurrency for higher data throughput. Dataflow Simulation Assistant shows the recommended number of pipeline delays as Suggested Latency. The suggested latency value is computed to give the best performance.

The following diagram shows the Dataflow Simulation Assistant where the Dataflow Subsystem currently specifies a latency value of zero, and the suggested latency for the system is two. Using the **Suggested Latency** value introduces pipeline delays in the model and enables more blocks to run in parallel.



Click the **Accept** button next to **Suggested Latency** in the Dataflow Simulation Assistant to use the recommended latency for the Dataflow Subsystem.

Dataflow Simulation Assistant now shows the number of threads as five implying that the blocks inside the dataflow subsystem simulate in parallel using five threads. Use of two pipeline delays increased the number of blocks that can be run in parallel inside Dataflow Subsystem. Latency value can also be entered directly in the Property Inspector for "Latency" parameter. Simulink shows the latency parameter value using Z^{-1} tags at the output ports of the dataflow subsystem.



Multicore Simulation Performance

We measure the performance improvement of using multiple cores by comparing the execution time taken for running model using multiple threads with the time taken when the model does not use dataflow. Execution time is measured using the sim command, which returns the simulation execution time of the model. These numbers and analysis were published on a Windows desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor.

```
Simulation execution time for multithreaded model = 4.37s
Simulation execution time for single-threaded model = 10.64s
Actual speedup with dataflow: 2.4x
```

Summary

This example shows how dataflow execution domain can improve performance in a communication system model using multiple cores on the desktop.

Input, Output, and Display

Learn how to input, output and display data and signals with Communications Toolbox.

Signal Terminology

This section defines important Communications Toolbox terms related to matrices, vectors, and scalars, as well as frame-based and sample-based processing.

Matrices, Vectors, and Scalars

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

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

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

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

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

Export Data to MATLAB

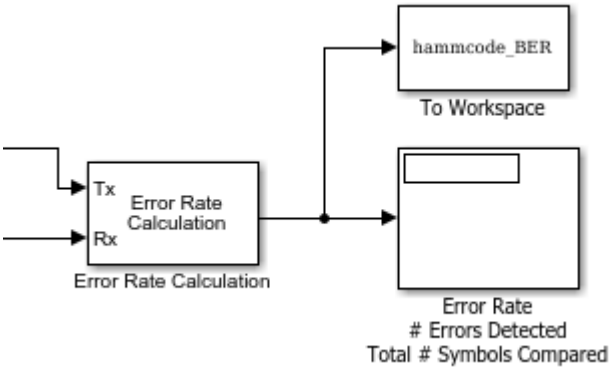
In this section...

Use a To Workspace Block

This section explains how to send data from a Simulink model to the MATLAB® workspace so you can analyze the results of simulations in greater detail.

You can use a To Workspace block, from the DSP System Toolbox™/Sinks library to send data to the MATLAB workspace as a vector. For example, you can send the error rate data from the Hamming code model, described in the section Reducing the Error Rate Using a Hamming Code on page 16-63. To insert a To Workspace block into the model, follow these steps:

1. To open the model, at the MATLAB prompt, enter `doc_hamming`.
2. To add a To Workspace block, begin typing the name 'to workspace' in the model window and select the To Workspace block from the DSP System Toolbox/Sinks library. Connect it as shown.



Tip More than one To Workspace block exists. Select the To Workspace block from the DSP System Toolbox / Sinks sublibrary.

Configure the To Workspace Block

To configure the To Workspace block, follow these steps:

1. Double-click the block to display its dialog box.
2. Type `hammcode_BER` in the **Variable name** field.

3. Type `1` in the **Limit data points to last** field. This limits the output vector to the values at the final time step of the simulation.
4. Click **OK**.

When you run a simulation, the model sends the output of the Error Rate Calculation block to the workspace as a vector of size 3, called `hamming_BER`. The entries of this vector are the same as those shown by the Error Rate Display block.

View Error Rate Data in Workspace

After running a simulation, you can view the output of the To Workspace block by typing the following commands at the MATLAB prompt:

```
format short e
hammcode_BER
```

The vector output is the following:

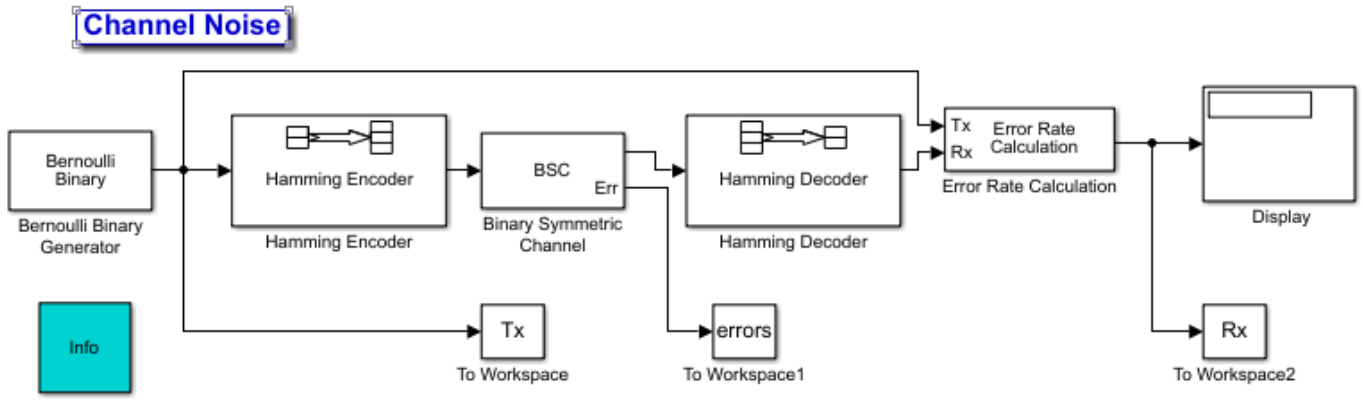
```
hammcode_BER =
5.4066e-003  1.0000e+002  1.8496e+004
```

The command `format short e` displays the entries of the vector in exponential form. The entries are as follows:

- The first entry is the error rate.
- The second entry is the total number of errors.
- The third entry is the total number of comparisons made.

Send Signal and Error Data to Workspace

To analyze the error-correction performance of the Hamming code, send the transmitted signal, the received signal, and the error vectors, created by the Binary Symmetric Channel block, to the workspace. An example of this is shown in the following figure.



1. Type `doc_channel` at the MATLAB command line to open the starter model.
2. Double-click the Binary Symmetric Channel block to open its dialog box, and select **Output error vector**. This creates an output port for the error data.

3. Move blocks to make room so that you can insert Hamming Encoder and Hamming Decoder blocks. To find them, start typing `Hamming` in the model window. Select them from the options presented. These Hamming Encoder and Hamming Decoder blocks are in the Communications Toolbox/Error Detection and Correction /Block sublibrary.
4. Add three To Workspace blocks into the model window and connect them as shown in the preceding figure.

   Tip More than one To Workspace block exists. Select the To Workspace block from the DSP System Toolbox / Sinks sublibrary.

5. Double-click the left To Workspace block.
   - Type `Tx` in the **Variable name** field in the block's dialog box. The block sends the transmitted signal to the workspace as an array called `Tx`.
   - In the **Save 2-D signals as** field, select `3-D array (concatenate along third dimension)`. This preserves each frame as a separate column of the array `Tx`.
   - Click **OK**.
6. Double-click the middle To Workspace block:
   - Type `errors` in the **Variable name** field.
   - In the **Save 2-D signals as** field, select `3-D array (concatenate along third dimension)`. This preserves each frame as a separate column of the array `Tx`.
   - Click **OK**.
7. Double-click the right To Workspace block:
   - Type `Rx` in the **Variable name** field.
   - In the **Save 2-D signals as** field, select `3-D array (concatenate along third dimension)`. This preserves each frame as a separate column of the array `Tx`.
   - Click **OK**.

View Signal and Error Data in Workspace

After running a simulation, you can display individual frames of data. For example, to display the tenth frame of `Tx`, at the MATLAB prompt type

```
Tx(:,:,10)
```

This returns a column vector of length 4, corresponding to the length of a message word. Usually, you should not type `Tx` by itself, because this displays the entire transmitted signal, which is very large.

To display the corresponding frame of errors, type

```
errors(:,:,10)
```

This returns a column vector of length 7, corresponding to the length of a codeword.

To display frames 1 through 5 of the transmitted signal, type

```
Tx(:,:,1:5)
```

Analyze Signal and Error Data

You can use MATLAB to analyze the data from a simulation. For example, to identify the differences between the transmitted and received signals, type

```
diffs = Tx~=Rx;
```

The vector `diffs` is the XOR of the vectors `Tx` and `Rx`. A 1 in `diffs` indicates that `Tx` and `Rx` differ at that position.

You can determine the indices of frames corresponding to message words that are incorrectly decoded with the following MATLAB command:

```
error_indices = find(diffs);
```

A 1 in the vector `not_equal` indicates that there is at least one difference between the corresponding frame of `Tx` and `Rx`. The vector `error_indices` records the indices where `Tx` and `Rx` differ. To view the first incorrectly decoded word, type

```
Tx(:,:,error_indices(1))
```

To view the corresponding frame of errors, type

```
errors(:,:,error_indices(1))
```

Analyze this data to determine the error patterns that lead to incorrect decoding.

Sources and Sinks

In this section...

Communications Toolbox provides sources, sinks, and display devices that facilitate analysis of communication system performance.

Data Sources

Use the functions and blocks listed in “Sources and Sinks” to generate random data to simulate a signal source.

Random Symbols

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

For example, the command below generates a 5-by-4 matrix whose entries are random, independently chosen, and uniformly distributed in the set {1,3,5}.

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

```
a = 5×4

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

To skew the distribution so that 1 is twice as likely to occur as either 3 or 5, use the command below. The third input argument has two rows, one of which indicates the possible values of `b` and the other indicates the probability of each value.

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

```
b = 5×4

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

Random Integers

In MATLAB®, the `randi` function generates random integer matrices whose entries are in a range that you specify. A special case generates random binary matrices.

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

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

```
c = 5×4

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

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

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

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

Random Bit Error Patterns

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

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

```
f = 5×4

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

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

On the other hand, to perturb each codeword by introducing one error with probability 0.4 and two errors with probability 0.6, use the command below instead. Each row has one '1' with probability 0.4, otherwise two '1's

```
g = randerr(5,4,[1,2; 0.4,0.6])
```

```
g = 5×4

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

Adding the random matrix `g` to your code matrix (modulo 2) introduces one or two errors into each codeword with the specified probability of occurrence for each. The probability matrix that is the third argument of `randerr` affects only the number of 1s in each row, not their placement.

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

```
binarymatrix1 = randsrc(100,1,[0 1]); % Possible values are 0,1
binarymatrix2 = randi([0 1],100,1); % Two possible values
binarymatrix3 = randerr(100,1,[0 1; 0.5 0.5]); % No 1s, or one 1
```

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

Noise Sources

Construct noise generator blocks in Simulink to simulate communication links.

Random Noise Generators

You can construct random noise generators to simulate channel noise by using the MATLAB Function block with random number generating functions. Construct different types of channel noise by using the following combinations.

Distribution	Block	Function
Gaussian	MATLAB Function	`wgn`
Rayleigh	MATLAB Function	`randn`
Rician	MATLAB Function	`randn`
Uniform on a bounded interval	MATLAB Function	`rand`

See “Random Noise Generators in Simulink” on page 12-29 for an example of how Rayleigh and Rician distributed noise is created.

Gaussian Noise Generator

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

For example, the command below generates a column vector of length 50 containing real white Gaussian noise whose power is 2 dBW. By default, the power type in dBW and load impedance is 1 ohm.

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

To generate complex white Gaussian noise whose power is 2 watts, across a load of 60 ohms, use either of the commands below.

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

To send a signal through an additive white Gaussian noise channel, use the `awgn` function. See “AWGN Channel” on page 22-2 for more information.

Sequence Generators

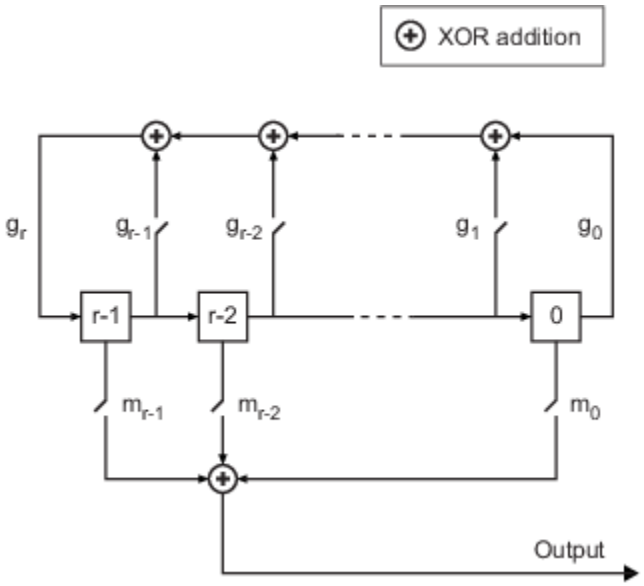
Use the functions, System objects, and blocks listed in “Sources and Sinks” to generate sequences for spreading or synchronization in a communication system. You can generate pseudorandom sequences, synchronization codes, and orthogonal codes. For examples comparing correlation properties of these sequence generators, see “Spreading Sequences” on page 9-21.

Pseudorandom Sequences

You can generate pseudorandom or pseudonoise (PN) sequences using these System objects in MATLAB and these blocks in Simulink. The applications of these sequences range from multiple-access spread spectrum communication systems to ranging, synchronization, and data scrambling.

Sequence	System object™	Block
Gold sequences	`comm.GoldSequence`	Gold Sequence Generator
Kasami sequences	`comm.KasamiSequence`	Kasami Sequence Generator
PN sequences	`comm.PNSequence`	PN Sequence Generator

To generate pseudorandom sequences, the underlying code implements shift registers, as illustrated in this diagram.



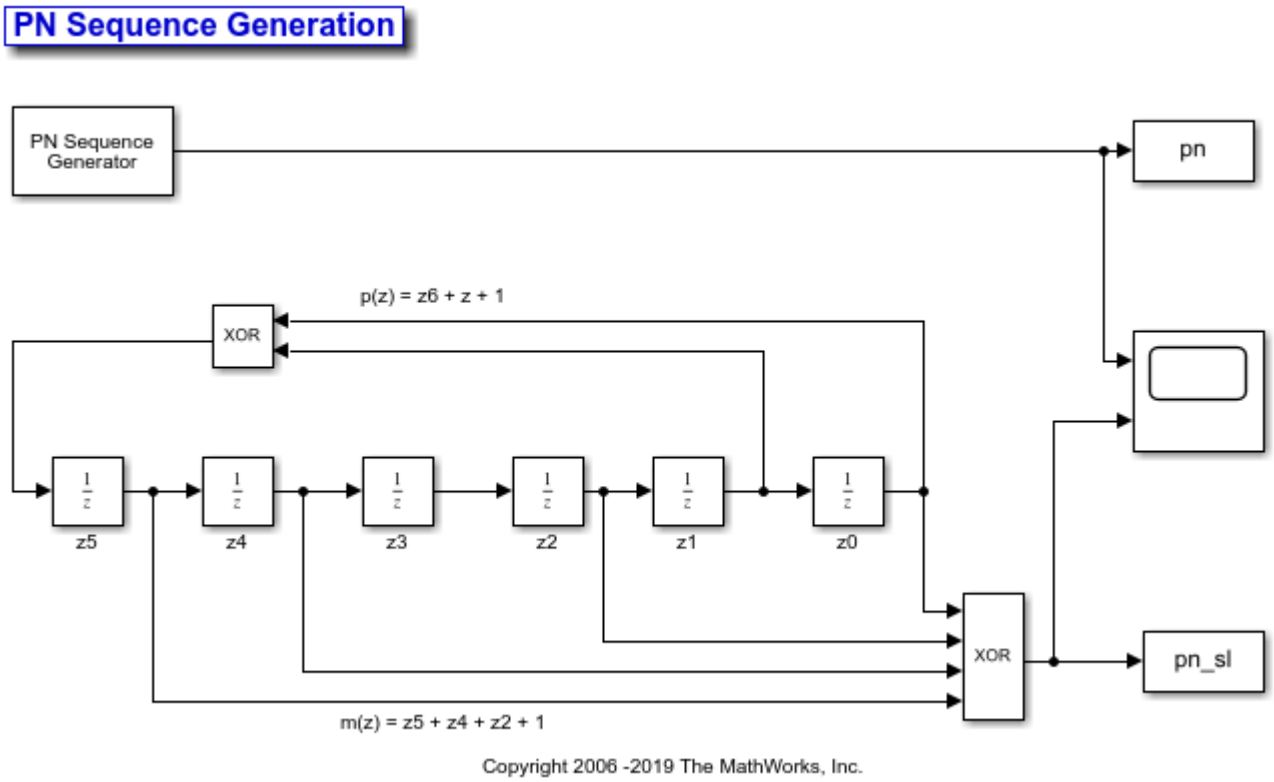
All r registers in the generator update their values at each time step according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The shift register can be described by a binary polynomial in z, $g_r z^r + g_{r-1} z^{r-1} + \ldots + g_0$. The coefficient g_i is 1 if there is a connection, or 0 if there is no connection, from the ith shift register to the adder.

The coefficient m_i is 1 if there is a delay, or a 0 if there is no delay, from the ith shift register to the adder preceding the output. If the shift is zero, the m_0 switch is closed while all other m_k switches are open.

The Kasami and PN sequence generators use this polynomial description for their generator polynomial. The Gold sequence generator uses this polynomial description for the preferred first and second generator polynomial PN sequences.

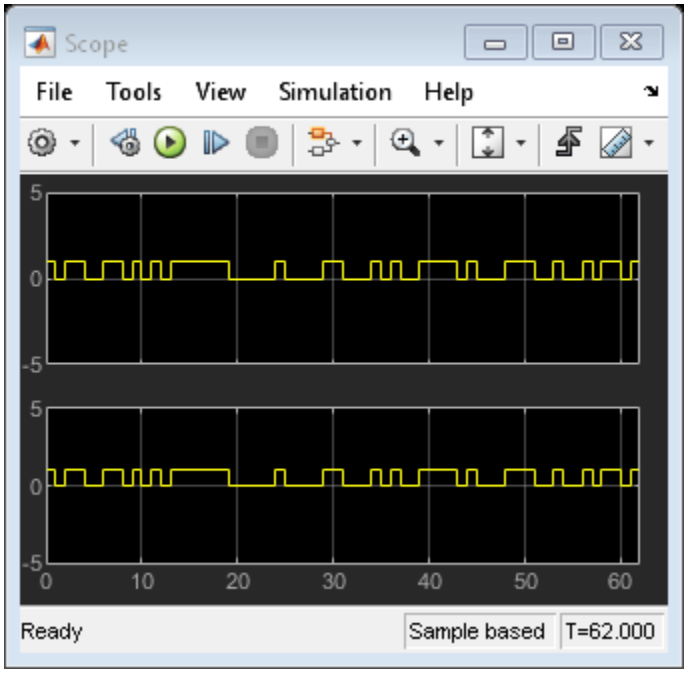
Model PN Sequence Generation With Linear Feedback Shift Register

This example shows that sequences output from the `PN Sequence Generator` can be modeled using a linear feedback shift register (LFSR) built with primitive Simulink® blocks.



For the chosen generator polynomial, p(z)=z^6+z+1, the model generates a PN sequence of period 63, by using the `PN Sequence Generator` block and by modeling a LFSR using primitive Simulink blocks. The two parameters, `Initial states` and `Output mask vector (or scalar shift value)`, are interpreted in the LFSR model schematic. The `PreLoadFcn` callback function is used to initialize runtime parameters. To view the callback functions, go to `MODELING> SETUP> Model Settings> Model Properties`, and select the `Callbacks` tab.

The scope output shows the two implementations produce matching PN sequences.



Using the PN Sequence Generator block allows you to easily generate PN sequences of large periods. To experiment further, open the model. Modify settings to see how the performance varies for different path delays or adjust the PN sequence generator parameters. You can experiment with different initial states, by changing the value of Initial states prior to running the simulation. For all values, the two generated sequences are the same.

Synchronization Codes

Use the `comm.BarkerCode` System object and Barker Code Generator block to generate Barker codes to perform synchronization. Barker codes are subsets of PN sequences. They are short codes, with a length at most 13, which are low-correlation sidelobes. A correlation sidelobe is the correlation of a codeword with a time-shifted version of itself.

Orthogonal Codes

Orthogonal codes are used for spreading to benefit from their perfect correlation properties. When used in multi-user spread spectrum systems, where the receiver is perfectly synchronized with the transmitter, the despreading operation is ideal.

Code	System object	Block
Hadamard codes	`comm.HadamardCode`	Hadamard Code Generator
OVSF codes	`comm.OVSFCode`	OVSF Code Generator
Walsh codes	`comm.WalshCode`	Walsh Code Generator

Scopes

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

- “Eye Diagrams” on page 9-14

- “Scatter Plots” on page 9-14
- “Signal Trajectories” on page 9-14

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

Block Name	Plots
Eye Diagram Scope	Eye diagram of a signal
Constellation Diagram	Constellation diagram and signal trajectory of a signal

Eye Diagrams

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

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

Examples appear in “View a Sinusoid” on page 9-14 and “View a Modulated Signal” on page 9-16.

Scatter Plots

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

The Constellation Diagram block produces a constellation diagram from discrete-time signals. An example appears in “View a Sinusoid” on page 9-14.

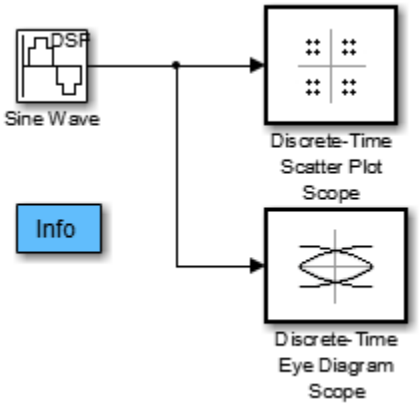
Signal Trajectories

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

The Constellation Diagram block produces signal trajectories. The Constellation Diagram block produces signal trajectories when the `ShowTrajectory` property is set to true. A signal trajectory connects all points of the input signal, irrespective of the specified decimation factor (`Samples per symbol`).

View a Sinusoid

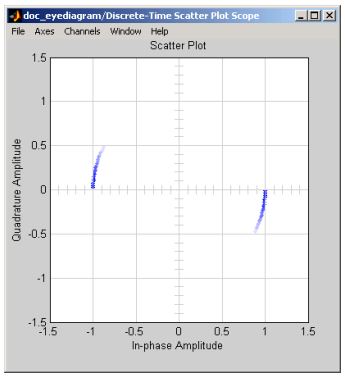
The following model produces a constellation diagram and an eye diagram from a complex sinusoidal signal. Because the decision time interval is almost, but not exactly, an integer multiple of the period of the sinusoid, the eye diagram exhibits drift over time. More specifically, successive traces in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



To open the model, enter `doc_eyediagram` at the MATLAB command line. To build the model, gather and configure these blocks:

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

Connect the blocks as shown in the preceding figure. In the **Simulate** section, set **Stop time** to `250`. The **Simulate** section appears on multiple tabs. Running the model produces the following scatter diagram plot.

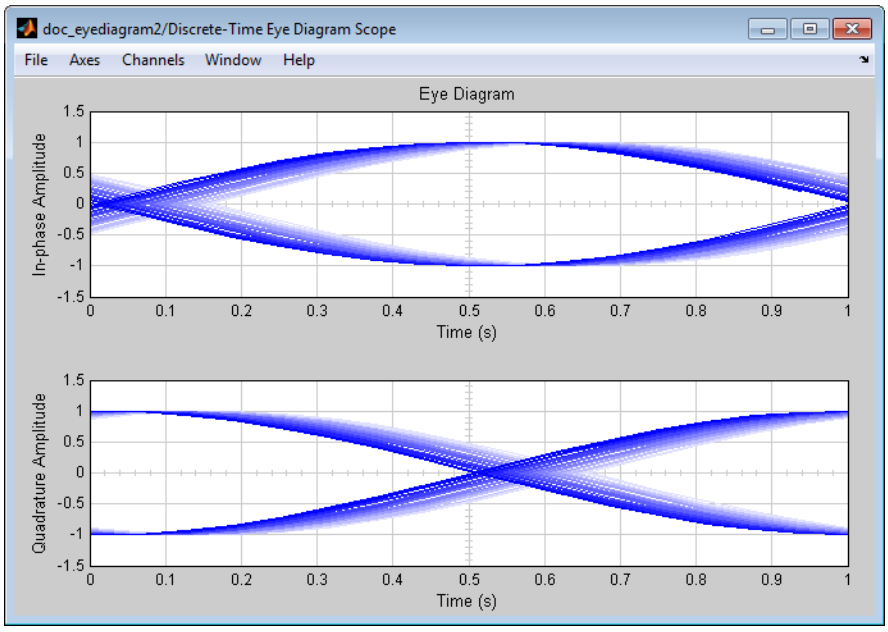


The points of the scatter plot lie on a circle of radius 1. Note that the points fade as time passes. This is because the box next to **Color fading** is checked under **Rendering Properties**, which causes the

scope to render points more dimly the more time that passes after they are plotted. If you clear this box, you see a full circle of points.

The Constellation Diagram block displays a circular trajectory.

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



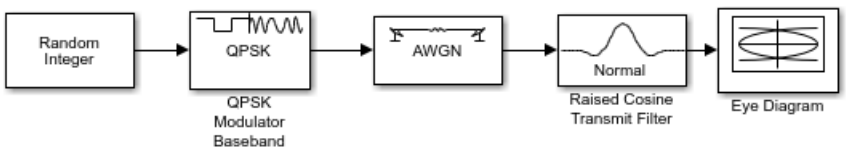
View a Modulated Signal

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

- “Eye Diagram of a Modulated Signal” on page 9-16
- “Constellation Diagram of a Modulated Signal” on page 9-18
- “Signal Trajectory of a Modulated Signal” on page 9-19

Eye Diagram of a Modulated Signal

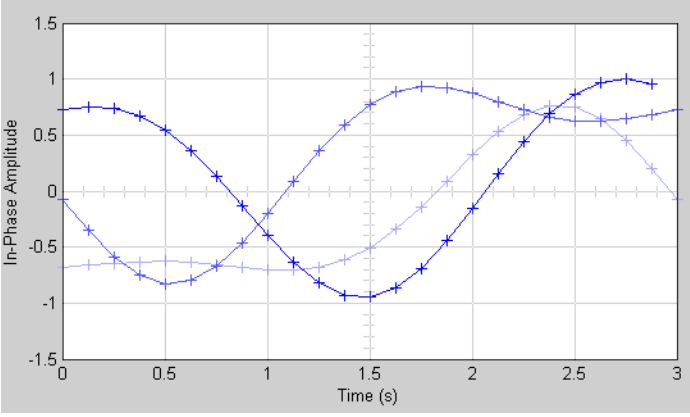
The following model modulates a random signal using QPSK, filters the signal with a raised cosine filter, and creates an eye diagram from the filtered signal.



To open the model, enter `doc_signaldisplays` at the MATLAB command line. To build the model, gather and configure the following blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to 4.
  - Set **Sample time** to `0.01`.
- QPSK Modulator Baseband, in PM in the Digital Baseband sublibrary of the Modulation library of Communications Toolbox, with default parameters
- AWGN Channel, in the Channels library of Communications Toolbox, with the following changes to the default parameter settings:
  - Set **Mode** to `Signal-to-noise ratio (SNR)`.
  - Set **SNR (dB)** to 15.
- Raised Cosine Transmit Filter, in the Comm Filters library
  - Set **Filter shape** to `Normal`.
  - Set **Rolloff factor** to `0.5`.
  - Set **Filter span in symbols** to 6.
  - Set **Output samples per symbol** to 8.
  - Set **Input processing** to `Elements as channels (sample based)`.
- Eye Diagram Scope, in the Comm Sinks library
  - Set **Samples per symbol** to 8.
  - Set **Symbols per trace** to 3. This specifies the number of symbols that are displayed in each trace of the eye diagram. A *trace* is any one of the individual lines in the eye diagram.
  - Set **Traces displayed** to 3.
  - Set **New traces per display** to 1. This specifies the number of new traces that appear each time the diagram is refreshed. The number of traces that remain in the diagram from one refresh to the next is **Traces displayed** minus **New traces per display**.
  - On the **Rendering Properties** panel, set **Markers** to + to indicate the points plotted at each sample. The default value of **Markers** is empty, which indicates no marker.
  - On the **Figure Properties** panel, set **Eye diagram to display** to `In-phase only`.

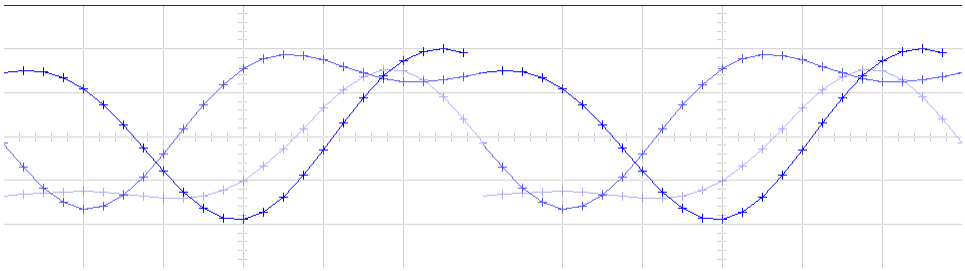
When you run the model, the Eye Diagram displays the following diagram. Your exact image varies depending on when you pause or stop the simulation.



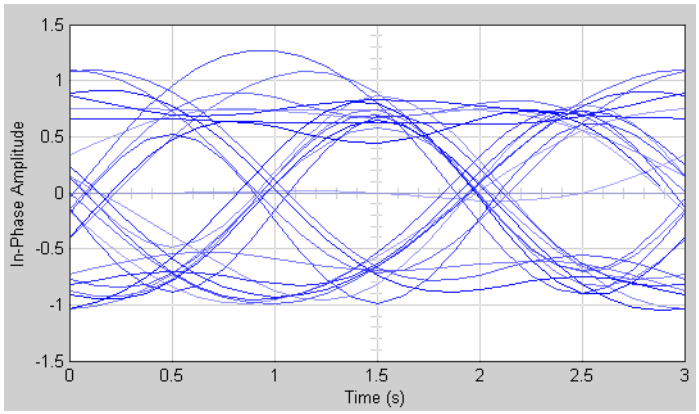
Three traces are displayed. Traces 2 and 3 are faded because **Color fading** under **Rendering Properties** is selected. This causes traces to be displayed less brightly the older they are. In this

picture, Trace 1 is the most recent and Trace 3 is the oldest. Because **New traces per display** is set to 1, only Trace 1 is appearing for the first time. Traces 2 and 3 also appear in the previous display.

Because **Symbols per trace** is set to 3, each trace contains three symbols, and because **Samples per trace** is set to 8, each symbol contains eight samples. Note that trace 1 contains 24 points, which is the product of **Symbols per trace** and **Samples per symbol**. However, traces 2 and 3 contain 25 points each. The last point in trace 2, at the right border of the scope, represents the same sample as the first point in trace 1, at the left border of the scope. Similarly, the last point in trace 3 represents the same sample as the first point in trace 2. These duplicate points indicate where the traces would meet if they were displayed side by side, as illustrated in the following picture.



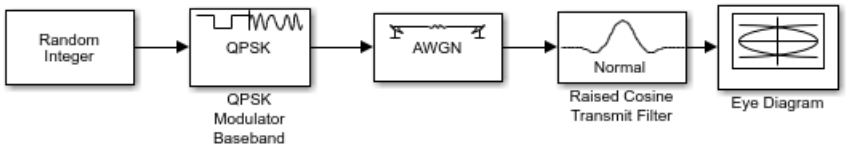
You can view a more realistic eye diagram by changing the value of **Traces displayed** to 40 and clearing the **Markers** field.



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

Constellation Diagram of a Modulated Signal

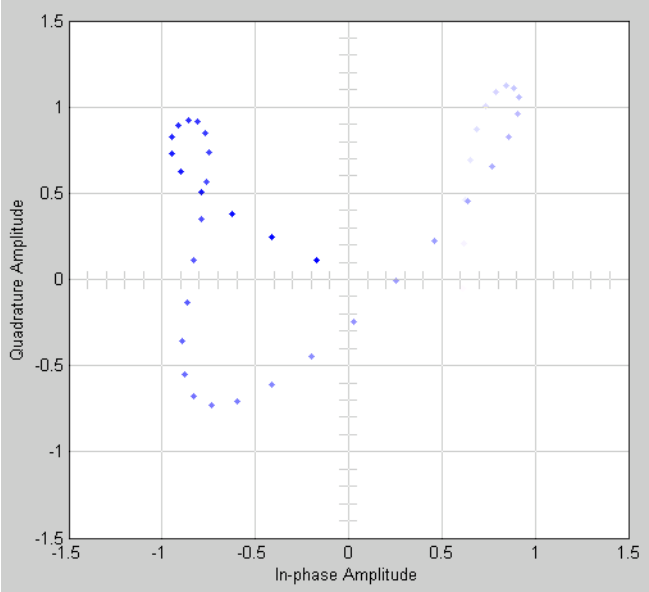
The following model creates a scatter plot of the same signal considered in “Eye Diagram of a Modulated Signal” on page 9-16.



To build the model, follow the instructions in “Eye Diagram of a Modulated Signal” on page 9-16 but replace the Eye Diagram block with the following block:

- Constellation Diagram, in the Communications Toolbox/Comm Sinks library
  - Set **Samples per symbol** to 2
  - Set **Offset** to `0`. This specifies the number of samples to skip before plotting the first point.
  - Set **Symbols to display** to `40`.

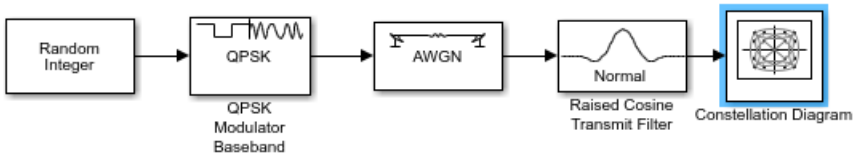
When you run the simulation, the Constellation Diagram block displays the following plot.



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

Signal Trajectory of a Modulated Signal

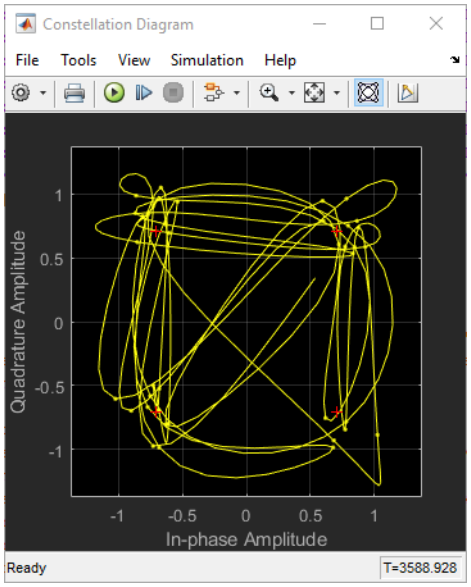
The following model creates a signal trajectory plot of the same signal considered in “Eye Diagram of a Modulated Signal” on page 9-16.



To build the model, follow the instructions in “Eye Diagram of a Modulated Signal” on page 9-16 but replace the Eye Diagram block with the following block:

- Constellation Diagram , in the Communications Toolbox/Comm Sinks library
  - Set **Samples per symbol** to 8.
  - Set **Symbols to display** to `40`. This specifies the number of symbols displayed in the signal trajectory. The total number of points displayed is the product of **Samples per symbol** and **Symbols to display**.

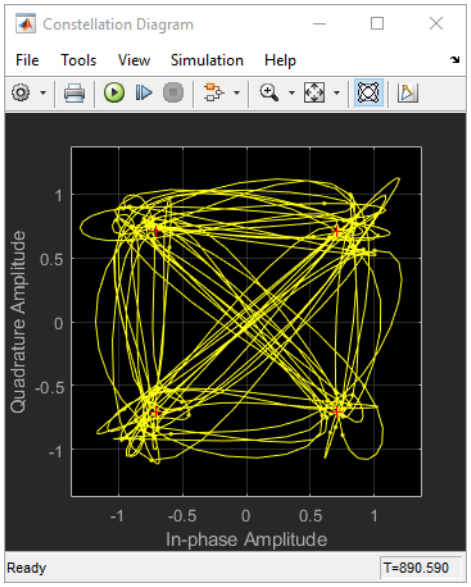
When you run the model, the Constellation Diagram displays a trajectory like the one below.



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

See “Constellation Diagram of a Modulated Signal” on page 9-18 to compare the preceding signal trajectory to the scatter plot of the same signal. The Constellation Diagram block connects the points displayed by the Constellation Diagram block to display the signal trajectory.

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



See Also

“Spreading Sequences” on page 9-21

Spreading Sequences

Spreading consists of multiplying input data bits by a pseudorandom or pseudonoise (PN) sequence. The ratio of the PN sequence bit rate to the data rate is called the *spreading factor*. When the PN sequence has a bit rate higher than the data bit rate, the spreading factor is greater than 1. When the spreading factor is greater than 1, spreading input data adds redundancy to the transmission signal.

Spreading input data by using spreading sequences with low cross-correlation properties enables the receiver to resolve individual user data after despreading the received signal. Using spreading sequences with low cross-correlation properties helps resolve individual user data in a multipath environment in the presence of interference signals.

After signal synchronization on the receiver side, the received signal is multiplied by the same PN that was used by the transmitter. This operation removes the spreading from the received signal. Ideally, after this despreading, the signal for the user of interest is recovered with no further contribution by the signals of interferers. In CDMA systems, each transmitter is assigned distinct spreading codes that have low cross-correlation properties, such as the ideal orthogonal codes or any one of the PN, Gold, or Kasami sequences.

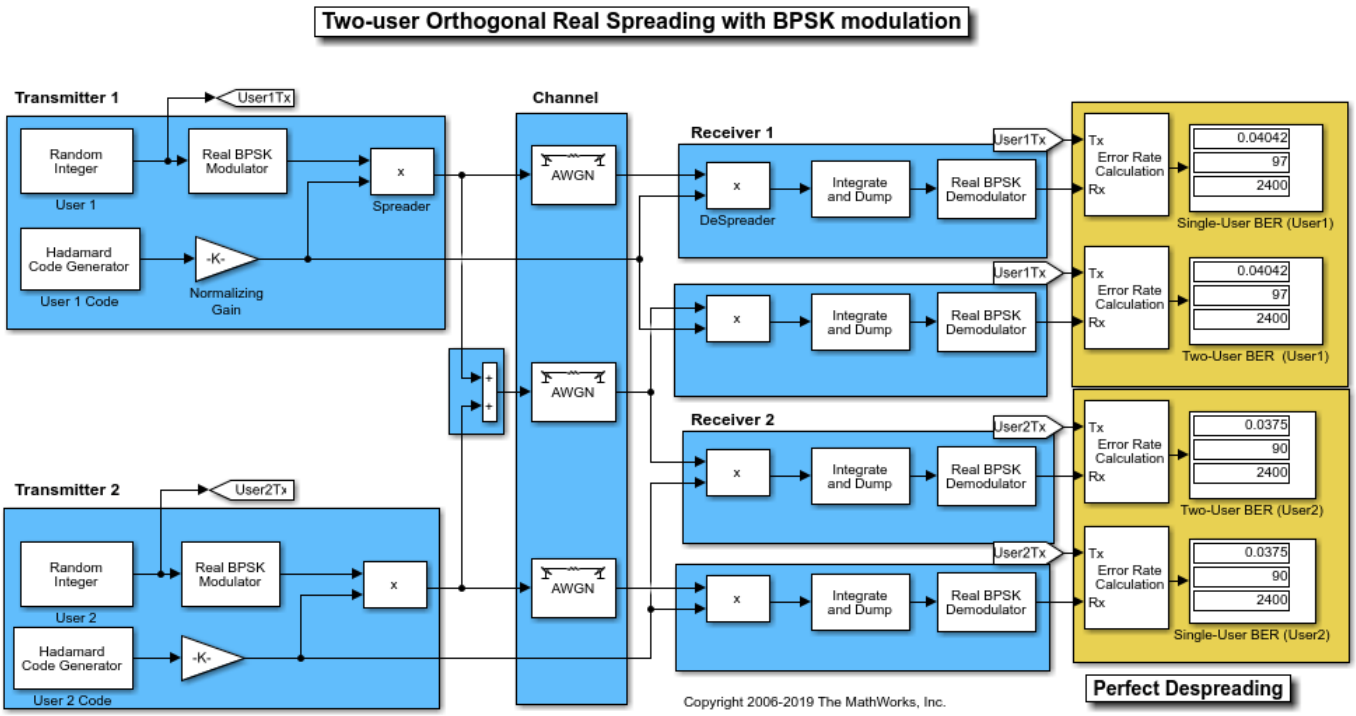
Spread-spectrum communication systems spread the transmission signal over a wide frequency band, typically much wider than the minimum bandwidth required to transmit the data. The spreading uses a waveform that appears random to anyone except the intended receiver of the transmitted signal. The waveform is actually pseudorandom in the sense that it can be generated by precise rules, yet has the statistical properties of a truly random sequence.

The following sections highlight various spreading sequences, their properties, and characteristic performance in single-user or multiuser and single-path or multipath transmission environments.

Orthogonal Spreading for Multiuser System in Single-Path Channel

This model compares data recovery for a single-user system versus a two-user system. Transmission data passes through a single-path AWGN channel in two data streams that are independently spread by different orthogonal codes.

The model uses random binary data, which is BPSK modulated (real), spread by orthogonal codes of length 64, and then transmitted over an AWGN channel. The receiver consists of a despreader followed by a BPSK demodulator.



Using the same transmission data, the model calculates the BER performance for recovery of the single-user and two-user transmissions through identically configured AWGN channels.

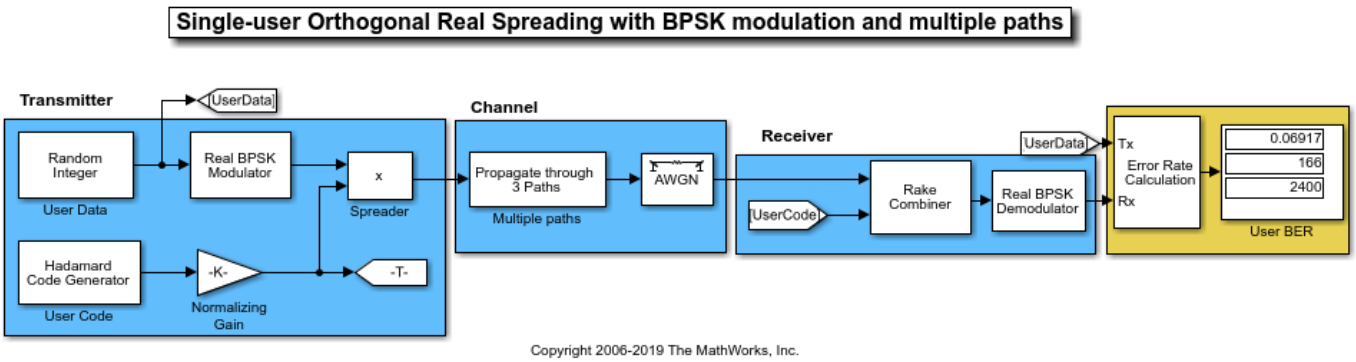
The bit error rate results are exactly the same for the individual users in both cases. The matching error rates result from perfect despreading due to the ideal cross-correlation properties of the orthogonal codes selected.

To experiment further, open the model. Modify the settings to see how the performance varies with different Hadamard codes for the individual users.

Orthogonal Spreading for Single-User System in Multipath Channel

This model simulates orthogonal spreading for a single-user system in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently by using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by orthogonal codes of length 64, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.



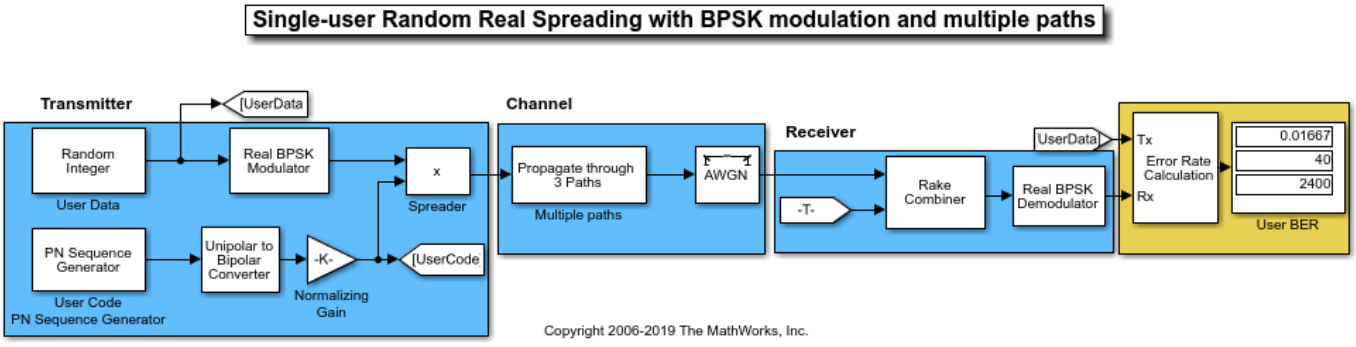
The non-ideal, auto-correlation values of the chosen orthogonal spreading codes prevent perfect resolution of the individual paths. As a consequence, BER performance is not improved by using diversity combining in the receiver. For a multipath example that uses PN sequences when spreading user data and uses diversity combining in the receiver, see “PN Spreading for Single-User System in Multipath Channel” on page 9-23.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different Hadamard codes.

PN Spreading for Single-User System in Multipath Channel

This model simulates pseudo-random spreading for a single-user system in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently by using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by PN sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator. The receiver achieves gains from diversity combining due to the ideal auto-correlation properties of the PN sequences used when spreading the data.

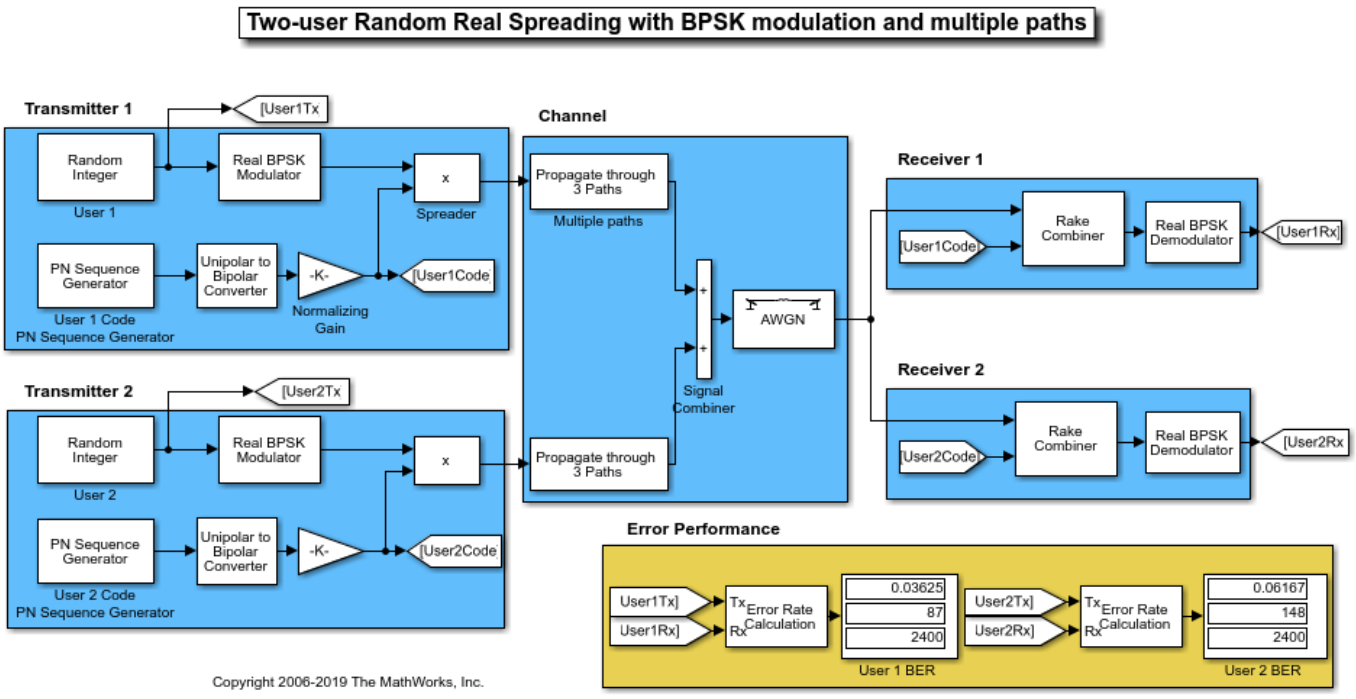


To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or adjust the PN sequence generator parameters.

PN Spreading for Multiuser System in Multipath Channel

This model simulates pseudo-random spreading for two users in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by PN sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.



Using the same transmission data, the model calculates the performance for two-user transmissions through identically configured, multipath AWGN channels.

Because the transmissions for the individual users were spread using different PN sequences, the error rate computed for the users are different. Due to the higher cross-correlation properties of the nonorthogonal PN sequences used to spread the data, BER performance is degraded in a multipath environment. Sequences with high orthogonality, such as Hadamard and Kasami, are a better choice for multipath environments. For a multipath example that uses Hadamard code sequences when spreading user data, see “Orthogonal Spreading for Multiuser System in Single-Path Channel” on page 9-21. For a multipath example that uses Kasami code sequences when spreading user data, see “Kasami Spreading for Multiuser System in Multipath Channel” on page 9-25.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different PN sequences for the individual users.

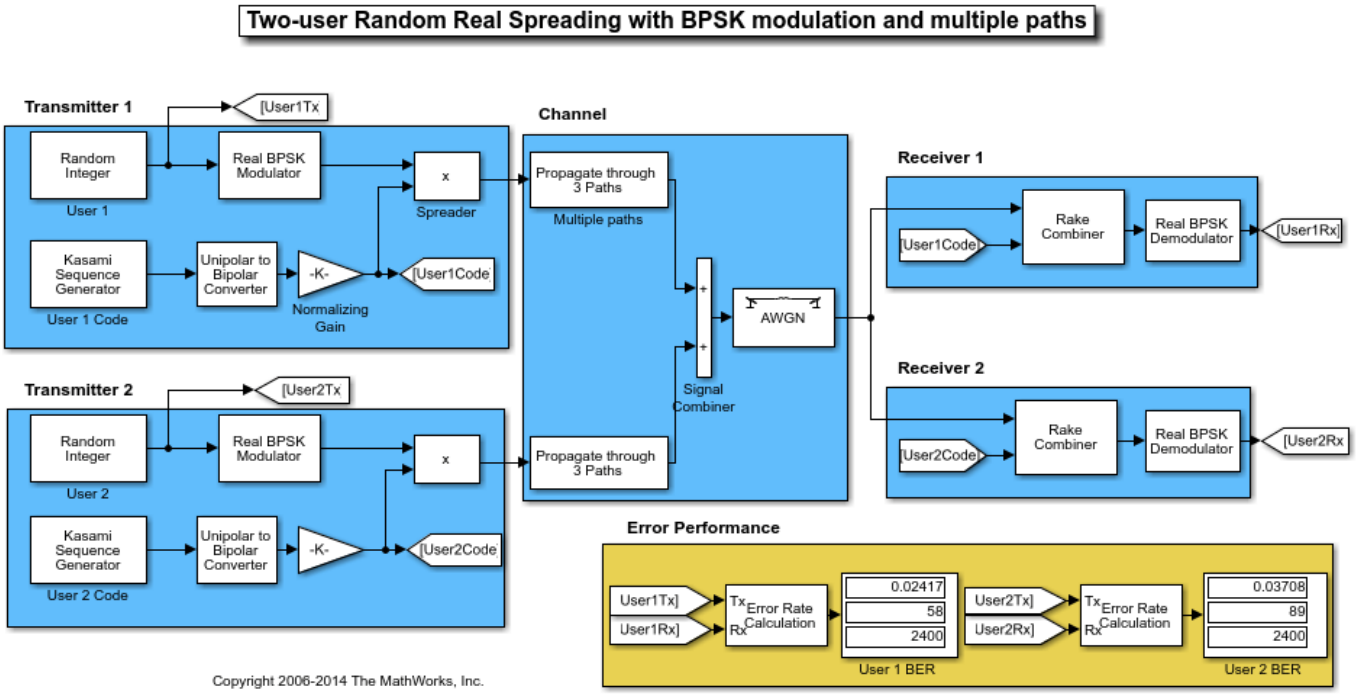
Benefits of Diversity Combining for Nonorthogonal Sequence Spreading

For the “PN Spreading for Multiuser System in Multipath Channel” on page 9-24 example, the individual user performance is degraded for the same channel conditions that were used in the “PN Spreading for Single-User System in Multipath Channel” on page 9-23 example. This is primarily due to the higher cross-correlation values between the two sequences, which prevent ideal separation. However, there are still advantages to diversity combining when using nonorthogonal sequence spreading, because the error rate for a multipath AWGN channel received using RAKE with diversity combining is nearly as good as the AWGN-only case in the “Orthogonal Spreading for Multiuser System in Single-Path Channel” on page 9-21 example.

Kasami Spreading for Multiuser System in Multipath Channel

This model simulates Kasami sequence spreading for two users in a multipath transmission environment. This is similar to a mobile channel environment where the signals are received over multiple paths. Each path can have different amplitudes and delays. The receiver combines the independent paths coherently using diversity reception to realize gains from the multipath transmissions received. The modeled system does not simulate fading effects and the receiver gets perfect knowledge of the number of paths and their respective delays.

The model uses random binary data, which is BPSK modulated (real), spread by Kasami sequences, and then transmitted over a multipath AWGN channel. The receiver consists of a despreader, a diversity combiner, and a BPSK demodulator.



Using the same transmission data, the model calculates the performance for two-user transmissions through identically configured multipath AWGN channels.

The computed BER indicates transmission data spread using Kasami sequences exhibit low cross-correlation. The Kasami sequences provide a balance between the ideal cross-correlation properties of orthogonal codes and the ideal auto-correlation properties of PN sequences.

To experiment further, open the model. Modify the settings to see how the performance varies for different path delays or with different Kasami sequence generator settings for the individual users.

See Also

Data and Signal Management

Matrices, Vectors, and Scalars

Simulink supports matrix signals, one-dimensional arrays, sample-based processing, and frame-based processing. This section describes how Communications Toolbox processes certain kinds of matrices and signals.

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

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

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

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

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

Processing Rules

The following rules indicate how the blocks in the Communications Toolbox process scalar, vector, and matrix signals.

- In their numerical computations, blocks that process scalars do not distinguish between one-dimensional scalars and one-by-one matrices. If the block produces a scalar output from a scalar input, the block preserves dimension.
- For vector input signals:
  - The numerical computations do not distinguish between one-dimensional arrays and M-by-1 matrices.
  - Most blocks do not process row vectors and do not support multichannel functionality.
  - The block output preserves dimension and orientation.
  - The block treats elements of the input vector as a collection that arises naturally from the block's operation (for example, a collection of symbols that jointly represent a codeword) or as successive samples from a single time series.
- Most blocks do not process matrix signals that have more than one row and more than one column. For blocks that do, a signal in the shape of an *N*-by-*M* matrix represents a series of *N* successive samples from *M* channels. An **Input processing** parameter on the block determines whether each element or column of the input signal is a channel.
- Some blocks, such as the digital baseband modulation blocks, can produce multiple output values for each value of a scalar input signal. A **Rate options** parameter on the block determines if the additional samples are output by increasing the rate of the output signal or by increasing the size of the output signal.

- Blocks that process continuous-time signals do not process frame-based inputs. Such blocks include the analog phase-locked loop blocks.

To learn which blocks processes scalar signals, vector signals, or matrices, refer to each block's individual Help page.

Sample-Based and Frame-Based Processing

In frame-based processing, blocks process data one frame at a time. Each frame of data contains sequential samples from an independent channel. For more information, see “Sample- and Frame-Based Concepts”.

In sample-based processing, blocks process signals one sample at a time. Each element of the input signal represents one sample of a distinct channel. For more information, see “What Is Sample-Based Processing?”.

Floating-Point and Fixed-Point Data Types

The inputs and outputs of the communications blocks support various data types. For some blocks, changing to single outputs can lead to different results when compared with double outputs for the same set of parameters. Some blocks may naturally output different data types than what they receive (e.g. digital modulators) a signal. Refer to the individual block reference pages for details.

For more information, see “Floating-Point Numbers” (Fixed-Point Designer) and “Fixed-Point Signal Processing”.

Access the Block Support Table

The Simulink Block Data Type Support for Communications Toolbox table provides details regarding capabilities and limitations pertaining to code generation, variable-sizing, and supported data types for all Communications Toolbox blocks. To access the table, type `showcommblockdatatypetable` at the MATLAB command line.

Delays

In this section...

Section Overview

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

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

Other References for Delays

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

- “Group Delay” on page 24-4
- Find Delay block reference page
- Delay block reference page
- Viterbi Decoder block reference page
- Derepeat block reference page

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

- Example: A Rate 2/3 Feedforward Encoder. on page 16-44.
- Example: Soft-Decision Decoding on page 16-48. (See Delay in Received Data on page 16-51.)
- Example: Delays from Demodulation on page 16-147.

Sources of Delays

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

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

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

The following discussions include some of these sources of delay.

ADSL Example Model

This section examines the “256-Channel ADSL” on page 8-115 example and shows how to compute the correct value for the **Receive delay** parameter in one of the Error Rate Calculation blocks in the model. The model includes delays from convolutional interleaving and an explicit delay block.

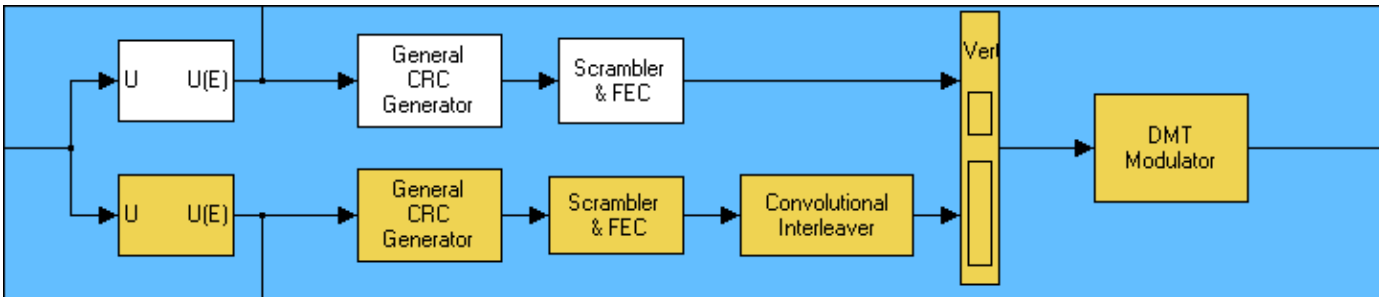
In the ADSL example, data follows two parallel paths that lead to Error Rate Calculation blocks near the end of each path. The first path has no interleaver and has a delay of zero. The second path has a delay compared to the first path due to a convolutional interleaver and deinterleaver pair and a fixed delay. The **Receive delay** parameter in the Error Rate Calculation block must reflect the delay of the given path. The sections that follow make an observation about frame periods in the model, and then consider delays for the interleaved data path.

Frame Periods in the Model

Before searching for individual delays, first observe that most signal lines throughout the model share the same frame period. On the **Debug** tab, expand **Information Overlays**. In the **Sample Time** section, select **Colors** to color blocks and signals according to their frame periods (or sample periods, in the case of sample-based signals). All signal lines at the top level of the model are the same color, which indicates they have the same frame period. Since there is a common frame period, frames are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by the period of each frame) reduces to a sum.

Path for Interleaved Data

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



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

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

Interleaving

In the second path, the delay due to the Convolutional Interleaver block in the transmitter and the Convolutional Deinterleaver block in the receiver is **Rows of shift registers** × **Register length step** × (**Rows of shift registers** – 1). As configured, the delay due to the interleaver and deinterleaver pair in the ADSL example is 5 × 2 × (5 – 1) = 40.

Delay Block

The receiver portion of the interleaved path also contains a Delay block. This block is set to insert a delay of 800 samples. The Delay block has the same sample time as the interleaver and deinterleaver blocks. Therefore, the total delay from interleaving, deinterleaving, and the explicit delay is 840 samples. These 840 samples make up one frame of data leaving the Delay block.

Summing the Delays

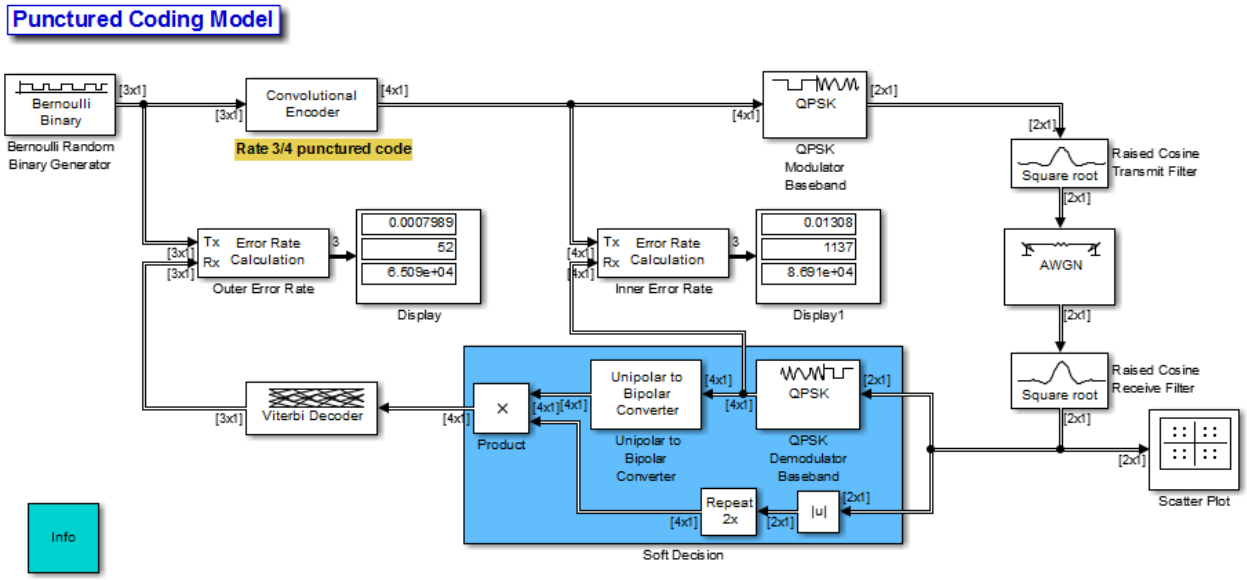
No other blocks in the interleaved path of the ADSL example cause any delays. Adding the delays from the interleaver and deinterleaver pair and the Delay block indicates that the total delay in the interleaved path is one frame.

Total Delay Relative to Error Rate Calculation Block

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

Punctured Coding Model

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



Frame Periods in the Model

Before searching for individual delays, if the **Timing Legend** pane is not already open, on the **Debug** tab, expand **Information Overlays**. In the **Sample Time** section, select **Legend**. In the **Timing Legend** pane **>Highlight>All**. Only the rightmost portion of the model differs in color from the rest of the model. This means that all signals and blocks in the model except those in the rightmost edge share the same frame period. Consequently, frames at this predominant frame rate are a convenient unit for measuring delays in the blocks that process these signals. In the computation of the cumulative delay along a path, the weighted average (of numbers of frames, weighted by each frame's period) reduces to a sum.

The yellow blocks represent multirate systems, while the AWGN Channel block runs at a higher frame rate than all the other blocks in the model.

Inner Error Rate Block

The block labeled Inner Error Rate, located near the center of the model, is a copy of the Error Rate Calculation block from the Comm Sinks library. It computes the bit error rate for the portion of the model that excludes the punctured convolutional code. In the portion of the model between this block's two input signals, delays come from the Tx Filter and the Rx Filter. This section explains why the Inner Error Rate block's **Receive delay** parameter is the total delay on page 10-10 value of 16.

Tx Filter Block

The block labeled Tx Filter is a copy of the Raised Cosine Transmit Filter block. It interpolates the input signal by a factor of 8 and applies a square-root raised cosine filter. For the filter, the value of the **Filter span in symbols** parameter is 6, which means its group delay is 3 symbols. Since this block's sample rate increases from input port to output port, it must output an initial frame of zeros at the beginning of the simulation. Since its input frame size is 2, the total delay of the block is 2 + 3 = 5 symbols. This corresponds to 5 samples at the block's input port.

Rx Filter Block

The block labeled Rx Filter is a copy of the Raised Cosine Receive Filter block. It decimates its input signal by a factor of 8 and applies another square-root raised cosine filter. For the filter, the value of the **Filter span in symbols** parameter is 6, which means its group delay is 3 symbols. At the output of the filter block, the 3 symbols correspond to 3 samples.

QPSK Demodulator Block

The block labeled QPSK Demodulator Baseband receives complex QPSK signals and outputs 2 bits for each complex input. This conversion to output bits doubles the cumulative delay at the input of the block.

Summing the Delays

No other blocks in the portion of the model between the Inner Error Rate block's two input signals cause any delays. The total delay is then (2 + 3 + 3) × 2 = 16 samples. This value can be used as the **Receive Delay** parameter in the Inner Error Rate block.

Outer Error Rate Block

The block labeled Outer Error Rate, located at the left of the model, is a copy of the Error Rate Calculation block from the Comm Sinks library. It computes the bit error rate for the entire model, including the punctured convolutional code. Delays come from the Tx Filter, Rx Filter, and Viterbi Decoder blocks. This section explains why the **Receive delay** parameter of the Outer Error Rate block is the total delay on page 10-10 value of `108`.

Filter and Downsample Blocks

The Tx Filter, Rx Filter, and Downsample blocks have a combined delay of 16 samples. For details, see “Inner Error Rate Block” on page 10-9.

Viterbi Decoder Block

Because the Viterbi Decoder block decodes a rate `3/4` punctured code, it actually reduces the delay seen at its input. This reduction is given as 16 × 3/4 = 12 samples.

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

Total Delay Relative to Outer Error Rate Block

The Outer Error Rate block requires a **Receive delay** parameter value that is the sum of all delays in the system. This total delay is `12 + 96 = 108` samples.

Use the Find Delay Block

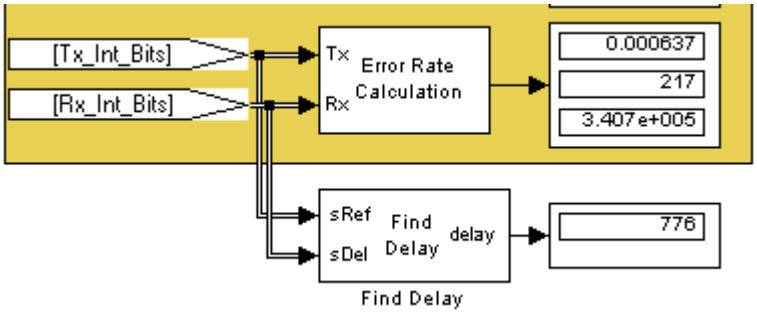
The preceding discussions explained why certain Error Rate Calculation blocks in the models had specific **Receive delay** parameter values. You could have arrived at those numbers independently by using the Find Delay block. This section explains how to find the signal delay using the ADSL example

model, `commadsl`, as an example. Applying the technique to the punctured convolutional coding example, discussed in “Punctured Coding Model” on page 10-8, would be similar.

Using the Find Delay Block to Determine the Correct Receive Delay

Recall from “Path for Interleaved Data” on page 10-7 that the delay in the path for interleaved data is 776 samples. To have the Find Delay block compute that value for you, use this procedure:

1. Insert a Find Delay block and a Display block in the model near the Error Rate Calculation block that computes the value labeled `Interleaved BER`.
2. Connect the blocks as shown below.



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

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

4. Run the simulation.

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

Manipulate Delays

- “Delays and Alignment Problems” on page 10-11
- “Observing the Problem” on page 10-12
- “Aligning Words of a Block Code” on page 10-14
- “Aligning Words for Interleaving” on page 10-15
- “Aligning Words of a Concatenated Code” on page 10-17
- “Aligning Words for Nonlinear Digital Demodulation” on page 10-19

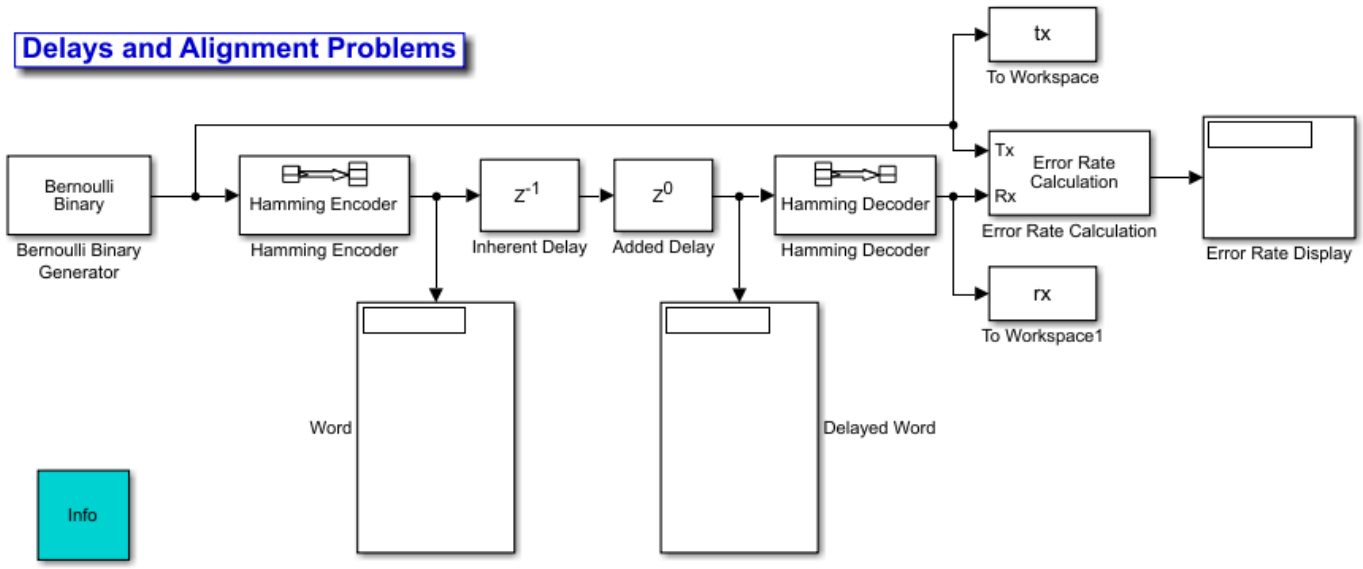
Delays and Alignment Problems

Some models require you not only to compute delays but to manipulate them. For example, if a model incurs a delay between a block encoder and its corresponding decoder, the decoder might misinterpret the boundaries between the codewords that it receives and, consequently, return meaningless results. More generally, such a situation can arise when the path between paired components of a block-oriented operation (such as interleaving, block coding, or bit-to-integer conversions) includes a delay-causing operation (such as those listed in “Sources of Delays” on page 10-6).

To avoid this problem, you can insert an additional delay of an appropriate amount between the encoder and decoder. If the model also computes an error rate, then the additional delay affects that

process, as described in “Delays” on page 10-6. This section uses examples to illustrate the purpose, methods, and implications of manipulating delays in a variety of circumstances.

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



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

- The Inherent Delay block represents any delay-causing blocks that might occur in a model between the encoder and decoder. See “Sources of Delays” on page 10-6 for a list of possibilities that might occur in a more realistic model.
- The Added Delay block is an explicit delay that you insert to produce an appropriate amount of total delay between the encoder and decoder. For example, the `commadsl` model contains a Delay block that serves this purpose.

Observing the Problem

By default, the **Delay** parameters in the Inherent Delay and Added Delay blocks are set to `1` and `0`, respectively. This represents the situation in which some operation causes a one-bit delay between the encoder and decoder, but you have not yet tried to compensate for it. The total delay between the encoder and decoder is one bit. You can see from the blocks labeled Word and Delayed Word that the codeword that leaves the encoder is shifted downward by one bit by the time it enters the decoder. The decoder receives a signal in which the boundary of the codeword is at the second bit in the frame, instead of coinciding with the beginning of the frame. That is, the codewords and the frames that hold them are not aligned with each other.

This nonalignment is problematic because the Hamming Decoder block assumes that each frame begins a new codeword. As a result, it tries to decode a word that consists of the last bit of one output frame from the encoder followed by the first six bits of the next output frame from the encoder. You

can see from the Error Rate Display block that the error rate from this decoding operation is close to 1/2. That is, the decoder rarely recovers the original message correctly.

To use an analogy, suppose someone corrupts a paragraph of prose by moving each period symbol from the end of the sentence to the end of the first word of the next sentence. If you try to read such a paragraph while assuming that a new sentence begins after a period, you misunderstand the start and end of each sentence. As a result, you might fail to understand the meaning of the paragraph.

To see how delays of different amounts affect the decoder's performance, vary the values of the **Delay** parameter in the Added Delay block and the **Receive delay** parameter in the Error Rate Calculation block and then run the simulation again. Many combinations of parameter values produce error rates that are close to 1/2. Furthermore, if you examine the transmitted and received data by entering

```
[tx rx]
```

at the MATLAB command line, you might not detect any correlation between the transmitted and received data.

Correcting the Delays

Some combinations of parameter values produce error rates of zero because the delays are appropriate for the system. For example:

- In the Added Delay block, set **Delay** to `6`.
- In the Error Rate Calculation block, set **Receive delay** to `4`.
- Run the simulation.
- Enter `[tx rx]` at the MATLAB command line.

The top number in the Error Rate Display block shows that the error rate is zero. The decoder recovered each transmitted message correctly. However, the Word and Displayed Word blocks do not show matching values. It is not immediately clear how the encoder's output and the decoder's input are related to each other. To clarify the matter, examine the output in the MATLAB command window. The sequence along the first column (`tx`) appears in the second column (`rx`) four rows later. To confirm this, enter

```
isequal(tx(1:end-4),rx(5:end))
```

at the MATLAB command line and observe that the result is `1` (true). This last command tests whether the first column matches a shifted version of the second column. Shifting the MATLAB vector `rx` by four rows corresponds to the Error Rate Calculation block's behavior when its **Receive delay** parameter is set to `4`.

To summarize, these special values of the **Delay** and **Receive delay** parameters work for these reasons:

- Combined, the Inherent Delay and Added Delay blocks delay the encoded signal by a full codeword rather than by a partial codeword. Thus the decoder is correct in its assumption that a codeword boundary falls at the beginning of an input frame and decodes the words correctly. However, the delay in the encoded signal causes each recovered message to appear one word later, that is, four bits later.
- The Error Rate Calculation block compensates for the one-word delay in the system by comparing each word of the transmitted signal with the data four bits later in the received signal. In this way, it correctly concludes that the decoder's error rate is zero.

Note These are not the only parameter values that produce error rates of zero. Because the code in this model is a (7, 4) block code and the inherent delay value is 1, you can set the **Delay** and **Receive delay** parameters to 7k-1 and 4k, respectively, for any positive integer k. It is important that the sum of the inherent delay (1) and the added delay (7k-1) is a multiple of the codeword length (7).

Aligning Words of a Block Code

The ADSL example, discussed in “ADSL Example Model” on page 10-7, illustrates the need to manipulate the delay in a model so that each frame of data that enters a block decoder has a codeword boundary at the beginning of the frame. The need arises because the path between a block encoder and block decoder includes a delay-causing convolutional interleaving operation. This section explains why the model uses a Delay block to manipulate the delay between the convolutional deinterleaver and the block decoder, and why the Delay block is configured as it is. To open the ADSL example model, enter `commadsl` at the MATLAB command line.

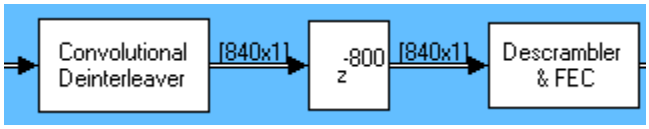
Misalignment of Codewords

In the ADSL example, the Convolutional Interleaver and Convolutional Deinterleaver blocks appear after the Scrambler & FEC subsystems but before the Descrambler & FEC subsystems. These two subsystems contain blocks that perform Reed-Solomon coding, and the coding blocks expect each frame of input data to start on a new word rather than in the middle of a word.

As discussed in “Path for Interleaved Data” on page 10-7, the delay of the interleaver and deinterleaver pair is 40 samples. However, the input to the Descrambler & FEC subsystem is a frame of size 840, and 40 is not a multiple of 840. Consequently, the signal that exits the Convolutional Deinterleaver block is a frame whose first entry does *not* represent the beginning of a new codeword. As described in “Observing the Problem” on page 10-12, this misalignment, between codewords and the frames that contain them, prevents the decoder from decoding correctly.

Inserting a Delay to Correct the Alignment

The ADSL example solves the problem by moving the word boundary from the 41st sample of the 840-sample frame to the first sample of a successive frame. Moving the word boundary is equivalent to delaying the signal. To this end, the example contains a Delay block between the Convolutional Deinterleaver block and the Descrambler & FEC subsystem.



The **Delay** parameter in the Delay block is `800` because that is the minimum number of samples required to shift the 41st sample of one 840-sample frame to the first sample of the next 840-sample frame. In other words, the sum of the inherent 40-sample delay (from the interleaving/deinterleaving process) and the artificial 800-sample delay is a full frame of data, not a partial frame.

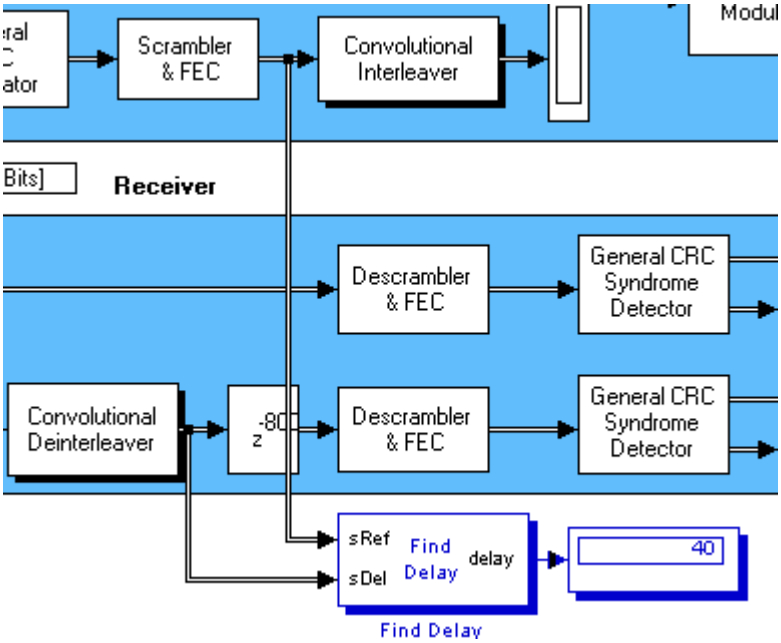
This 800-sample delay has implications for other parts of the model, specifically, the **Receive delay** parameter in one of the Error Rate Calculation blocks. For details about how the delay influences the value of that parameter, see “Path for Interleaved Data” on page 10-7.

Using the Find Delay Block

The preceding discussion explained why an 800-sample delay is necessary to correct the misalignment between codewords and the frames that contain them. Knowing that the Descrambler

& FEC subsystem requires frame boundaries to occur on word boundaries, you could have arrived at the number 800 independently by using the Find Delay block. Use this procedure:

1. Insert a Find Delay block and a Display block in the model.
2. Create a branch line that connects the input of the Convolutional Interleaver block to the `sRef` input of the Find Delay block.
3. Create another branch line that connects the output of the Convolutional Deinterleaver block to the `sDel` input of the Find Delay block.
4. Connect the `delay` output of the Find Delay block to the new Display block. The modified part of the model now looks like the following image (which also shows drop shadows on key blocks to emphasize the modifications).

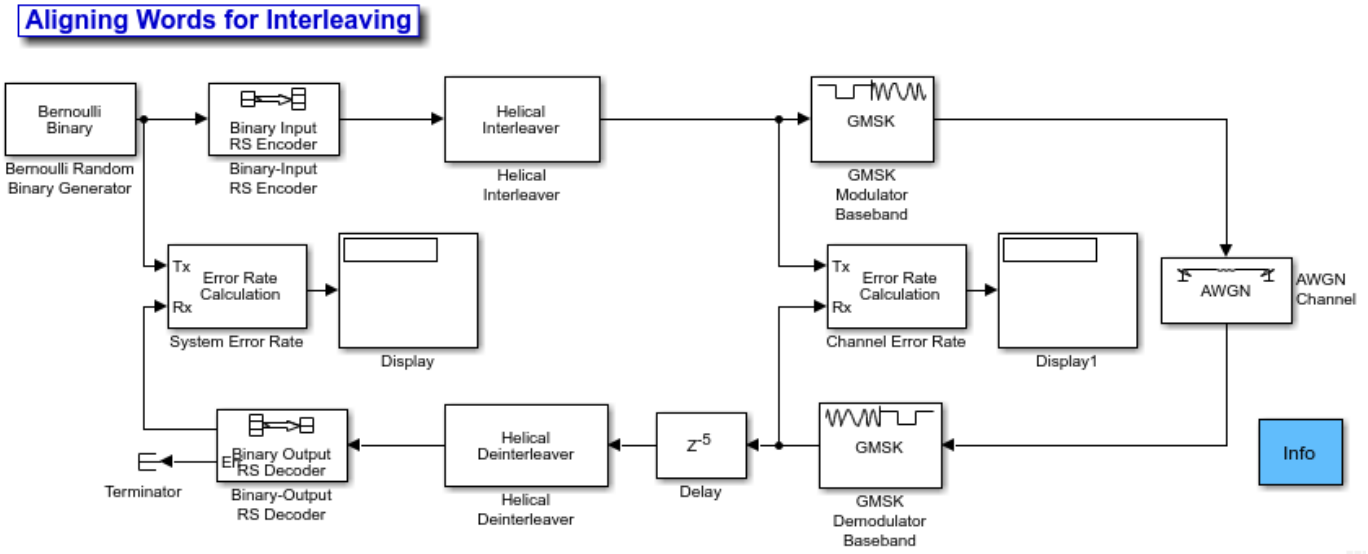


5. Show the dimensions of each signal in the model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**.
6. Run the simulation.

The new Display block now shows the value 40. Also, the display of signal dimensions shows that the output from the Convolutional Deinterleaver block is a frame of length 840. These results indicate that the sequence of blocks between the Convolutional Interleaver and Convolutional Deinterleaver, inclusive, delays an 840-sample frame by 40 samples. An additional delay of 800 samples brings the total delay to 840. Because the total delay is now a multiple of the frame length, the delayed deinterleaved data can be decoded.

Aligning Words for Interleaving

This section describes an example that manipulates the delay before a deinterleaver, because the path between the interleaver and deinterleaver includes a delay from demodulation. To open the model, enter `doc_gmskint` at the MATLAB command line.



The model includes block coding, helical interleaving, and GMSK modulation. The table below summarizes the individual block delays in the model.

Block	Delay, in Output Samples from Individual Block	Reference
GMSK Demodulator Baseband	16	“Delays in Digital Modulation” on page 16-146
Helical Deinterleaver	42	“Delays of Convolutional Interleavers” on page 16-121
Delay	5	Delay reference page

Misalignment of Interleaved Words

The demodulation process in this model causes a delay between the interleaver and deinterleaver. Because the deinterleaver expects each frame of input data to start on a new word, it is important to ensure that the total delay between the interleaver and deinterleaver includes one or more full frames but no partial frames.

The delay of the demodulator is 16 output samples. However, the input to the Helical Deinterleaver block is a frame of size 21, and 16 is not a multiple of 21. Consequently, the signal that exits the GMSK Demodulator Baseband block is a frame whose first entry does *not* represent the beginning of a new word. As described in “Observing the Problem” on page 10-12, this misalignment between words and the frames that contain them hinders the deinterleaver.

Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 17th sample of the 21-sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by five samples. The Delay block between the GMSK Demodulator Baseband block and the Helical Deinterleaver block accomplishes such a delay. The Delay block has its **Delay** parameter set to 5.

Combining the effects of the demodulator and the Delay block, the total delay between the interleaver and deinterleaver is a full 21-sample frame of data, not a partial frame.

Checking Alignment of Block Codewords

The interleaver and deinterleaver cause a combined delay of 42 samples measured at the output from the Helical Deinterleaver block. Because the delayed output from the deinterleaver goes next to a Reed-Solomon decoder, and because the decoder expects each frame of input data to start on a new word, it is important to ensure that the total delay between the encoder and decoder includes one or more full frames but no partial frames.

In this case, the 42-sample delay is exactly two frames. Therefore, it is not necessary to insert a Delay block between the Helical Deinterleaver block and the Binary-Output RS Decoder block.

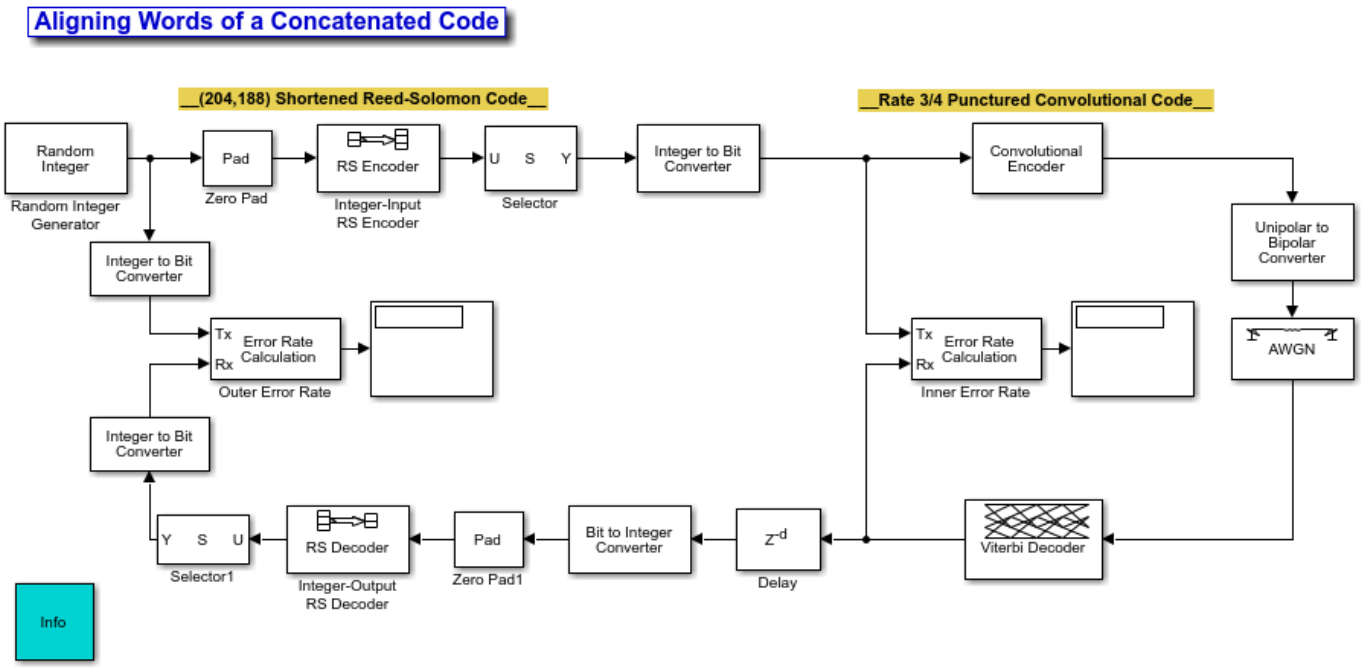
Computing Delays to Configure the Error Rate Calculation Blocks

The model contains two Error Rate Calculation blocks, labeled Channel Error Rate and System Error Rate. Each of these blocks has a **Receive delay** parameter that must reflect the delay of the path between the block's Tx and Rx signals. The following table explains the **Receive delay** values in the two blocks.

Block	Receive Delay Value	Reason
Channel Error Rate	16	Delay of GMSK Demodulator Baseband block, in samples
System Error Rate	15*3	Three fifteen-sample frames: one frame from the GMSK Demodulator Baseband and Delay blocks, and two frames from the interleaver and deinterleaver pair

Aligning Words of a Concatenated Code

This section describes an example that manipulates the delay between the two portions of a concatenated code decoder, because the first portion includes a delay from Viterbi decoding while the second portion expects frame boundaries to coincide with word boundaries. To open the model, enter `doc_concat` at the MATLAB command line. It uses the block and convolutional codes from the `commdvbt` example, but simplifies the overall design a great deal.



The model includes a shortened block code and a punctured convolutional code. All signals and blocks in the model share the same frame period. The following table summarizes the individual block delays in the model.

Block	Delay, in Output Samples from Individual Block
Viterbi Decoder	136
Delay	1496 (that is, 1632 - 136)

Misalignment of Block Codewords

The Viterbi decoding process in this model causes a delay between the Integer to Bit Converter block and the Bit to Integer Converter block. Because the latter block expects each frame of input data to start on a new 8-bit word, it is important to ensure that the total delay between the two converter blocks includes one or more full frames but no partial frames.

The delay of the Viterbi Decoder block is 136 output samples. However, the input to the Bit to Integer Converter block is a frame of size 1632. Consequently, the signal that exits the Viterbi Decoder block is a frame whose first entry does *not* represent the beginning of a new word. As described in “Observing the Problem” on page 10-12, this misalignment between words and the frames that contain them hinders the converter block.

Note The outer decoder in this model (Integer-Output RS Decoder) also expects each frame of input data to start on a new codeword. Therefore, the misalignment issue in this model affects many concatenated code designs, not just those that convert between binary-valued and integer-valued signals.

Inserting a Delay to Correct the Alignment

The model moves the word boundary from the 137th sample of the 1632-sample frame to the first sample of the next frame. Moving the word boundary is equivalent to delaying the signal by 1632-136 samples. The Delay block between the Viterbi Decoder block and the Bit to Integer Converter block accomplishes such a delay. The Delay block has its **Delay** parameter set to `1496`.

Combining the effects of the Viterbi Decoder block and the Delay block, the total delay between the interleaver and deinterleaver is a full 1632-sample frame of data, not a partial frame.

Computing Delays to Configure the Error Rate Calculation Blocks

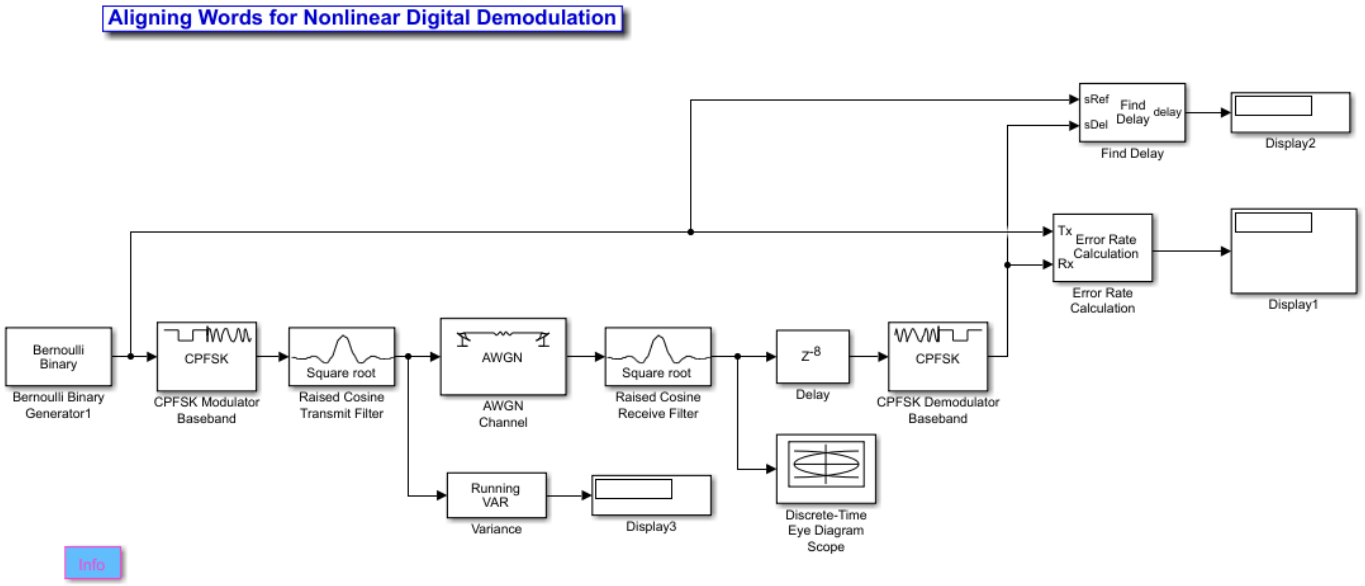
The model contains two Error Rate Calculation blocks, labeled Inner Error Rate and Outer Error Rate. Each of these blocks has a **Receive delay** parameter that must reflect the delay of the path between the block's `Tx` and `Rx` signals. The table below explains the **Receive delay** values in the two blocks.

Block	Receive Delay Value	Reason
Inner Error Rate	`136`	Delay of Viterbi Decoder block, in samples
Outer Error Rate	`1504 (188*8 bits)`	One 188-sample frame, from the combination of the inherent delay of the Viterbi Decoder block and the added delay of the Delay block

Aligning Words for Nonlinear Digital Demodulation

This example manipulates delay in order obtain the correct symbol synchronization of a signal so that symbol boundaries correctly align before demodulation occurs.

To open this model, type `doc_nonlinear_digital_demod` at the MATLAB command line.



This model includes a CPFSK modulation scheme and pulse shaping filter. For the demodulation to work properly, the input signal to the CPFSK demodulator block must have the correct alignment. Various blocks in this model introduce processing delays. Because of these delays, the input signal to the CPFSK demodulator block is not in the correct alignment.

Both the Raised Cosine Transmit and Receive Filter blocks introduce a delay. The delay is defined as: $GroupDelay \cdot Ts$

where Ts represents the input sample time of the Raised Cosine Transmit Filter block.

The input sample time of the Raised Cosine Transmit Filter block equals the output sample time of the Raised Cosine Receive Filter block. Therefore, the total delay at the output of the Raised Cosine Receive Filter is:

$2 \cdot GroupDelay \cdot Ts$

or $8 \cdot Ts$

as GroupDelay = 4

The CPFSK demodulator block receives this delayed signal, and then it processes each collection of 8 samples per symbol to compute 1 output symbol. You must ensure that the CPFSK demodulator receives input samples in the correct collection of samples. For binary CPFSK with a **Modulation index** of 1/2, the demodulator input must align along even numbers of symbols. Note that this requirement applies only to binary CPFSK with a modulation index of 1/2. Other CPM schemes with different M-ary values and modulation indexes have different requirements.

To ensure that the CPFSK demodulator in this model receives the correct collection of input samples with the correct alignment, introduce a delay of 8 samples (in this example, $8 \cdot Ts$). The total delay at the input of the CPFSK demodulator is $16 \cdot Ts$, which equates to two symbol delays (2.T, where T is the symbol period).

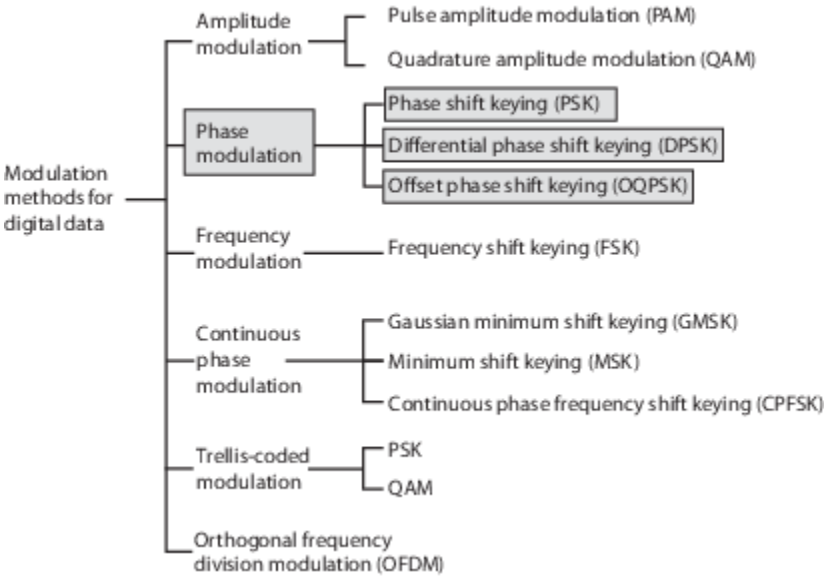
In sample-based mode, the CPFSK demodulator introduces a delay of **Traceback length** + 1 samples at its output. In this example, **Traceback length** equals 16. Therefore, the total **Receiver delay** in the Error rate calculation block equals 17+2 or 19. For more information, see “Delays in Digital Modulation” on page 16-146.

Digital Modulation

Phase Modulation

Phase modulation is a linear baseband modulation technique in which the message modulates the phase of a constant amplitude signal. Communications Toolbox provides modulators and demodulators for these phase modulation techniques:

- Phase shift keying (PSK) — Binary, quadrature, and general PSK
- Differential phase shift keying (DPSK) — Binary, quadrature, and general DPSK
- Offset QPSK (OQPSK)



To modulate input data with these techniques, you can use MATLAB functions, System objects, or Simulink blocks.

Modulation Scheme	MATLAB functions	System objects	Simulink blocks
Binary PSK (BPSK)		• `comm.BPSKModulator` • `comm.BPSKDemodulator`	• BPSK Modulator Baseband • BPSK Demodulator Baseband
Quadrature PSK (QPSK)		• `comm.QPSKModulator` • `comm.QPSKDemodulator`	• QPSK Modulator Baseband • QPSK Demodulator Baseband
General PSK	• `pskmod` • `pskdemod`	• `comm.PSKModulator` • `comm.PSKDemodulator`	• M-PSK Modulator Baseband • M-PSK Demodulator Baseband

Modulation Scheme	MATLAB functions	System objects	Simulink blocks
Differential BPSK (DBPSK)		• `comm.DBPSKModulator` • `comm.DBPSKDemodulator`	• DBPSK Modulator Baseband • DBPSK Demodulator Baseband
Differential QPSK (DQPSK)		• `comm.DQPSKModulator` • `comm.DQPSKDemodulator`	• DQPSK Modulator Baseband • DQPSK Demodulator Baseband
General DPSK	• `dpskmod` • `dpskdemod`	• `comm.DPSKModulator` • `comm.DPSKDemodulator`	• M-DPSK Modulator Baseband • M-DPSK Demodulator Baseband
OQPSK		• `comm.OQPSKModulator` • `comm.OQPSKDemodulator`	• OQPSK Modulator Baseband • OQPSK Demodulator Baseband

Baseband and Passband Simulation

Communications Toolbox supports *baseband* and *passband* simulation methods; however, the phase shift keying techniques support baseband simulation only.

A general passband waveform can be represented as

$$Y_1(t)\sqrt{2}\cos(2\pi f_c t + \theta) - Y_2(t)\sqrt{2}\sin(2\pi f_c t + \theta)\,,$$

where f_c is the carrier frequency and θ is the initial phase of the carrier signal. This equation is equal to the real part of

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

In a baseband simulation, only the expression within the square brackets is modeled. The vector y is a sampling of the complex signal

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

BPSK

In binary phase shift keying (BPSK), the phase of a constant amplitude signal switches between two values corresponding to binary 1 and binary 0. The passband waveform of a BPSK signal is

$$s_n(t) = \sqrt{\frac{2E_b}{T_b}}\cos(2\pi f_c t + \phi_n),$$

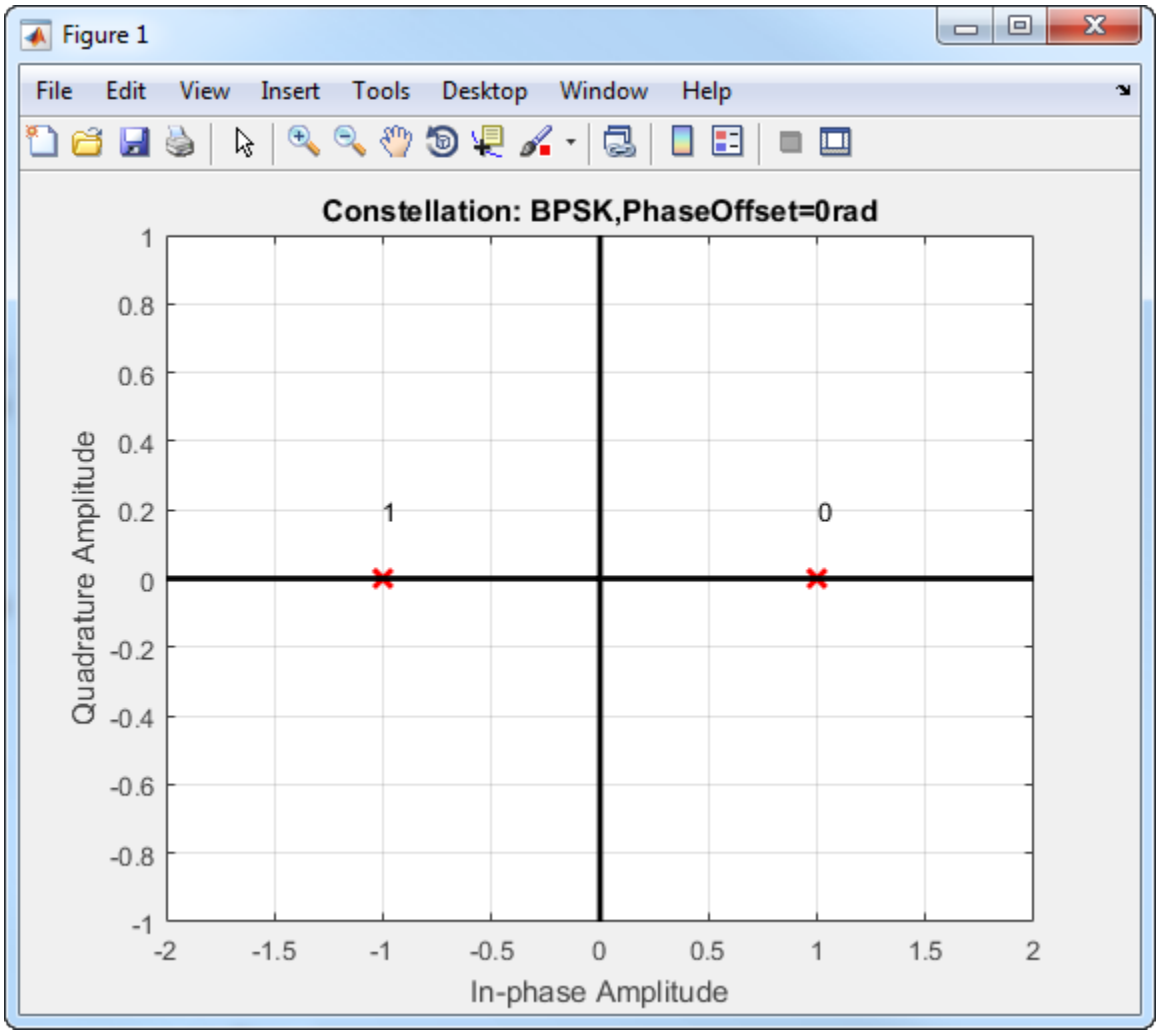
where:

- 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

$$s_n(t) = e^{-i\phi_n} = \cos(\pi n).$$

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.

QPSK

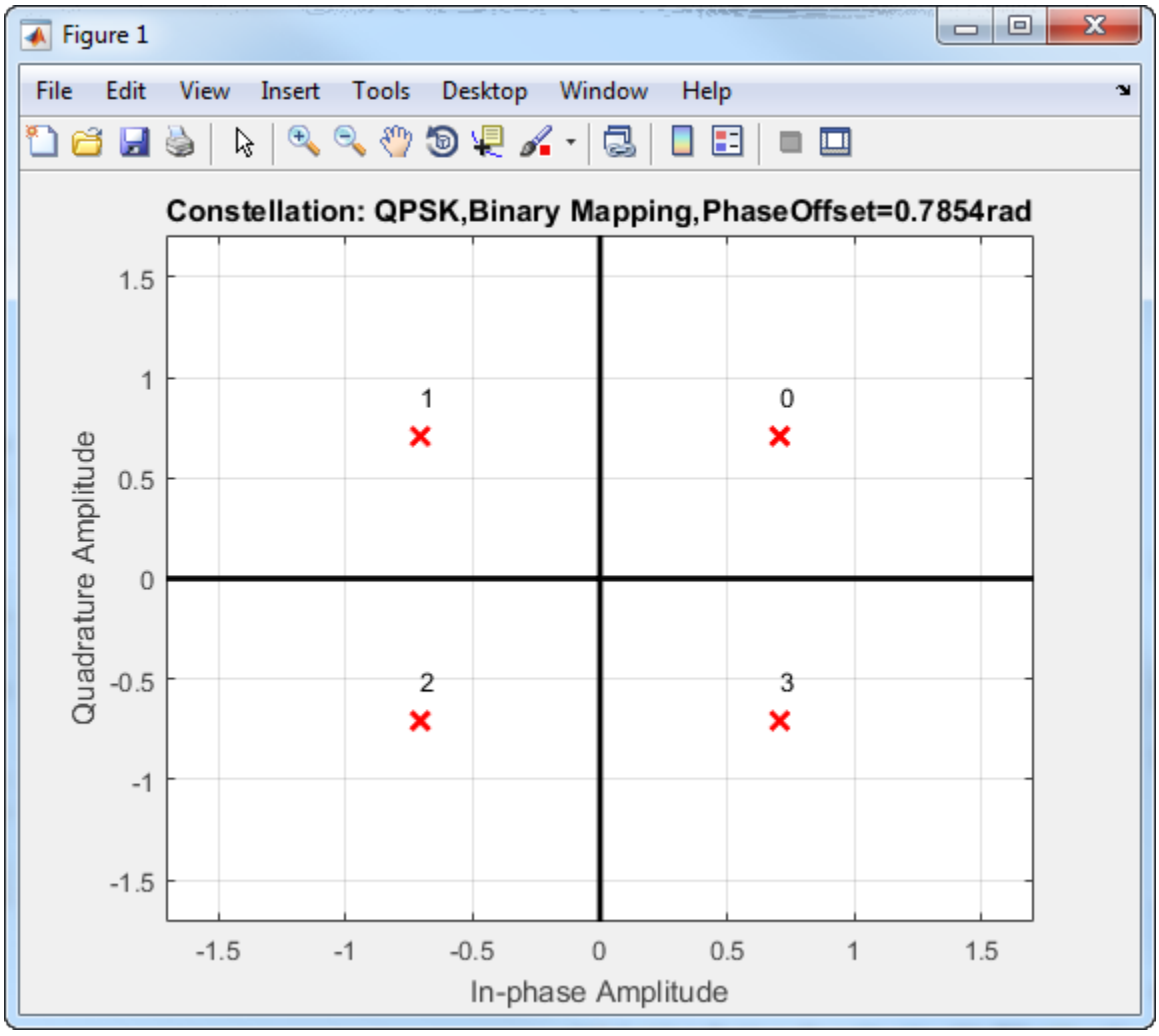
In quadrature phase shift keying, the message bits are grouped into 2-bit symbols, which are transmitted as one of four phases of a constant amplitude baseband signal. This grouping provides a bandwidth efficiency that is twice as great as the efficiency of BPSK. The general QPSK signal is expressed as

$$s_n(t) = \sqrt{\frac{2E_s}{T_s}}\cos\left(2\pi f_c t + (2n+1)\frac{\pi}{4}\right); \quad n \in \left\{0, 1, 2, 3\right\},$$

where E_s is the energy per symbol and T_s is the symbol duration. The complex baseband representation of a QPSK signal is

$$s_n(t) = \exp\left(j\pi\left(\frac{2n+1}{4}\right)\right); \quad n \in \left\{0, 1, 2, 3\right\}.$$

In this QPSK constellation diagram, each 2-bit sequence is mapped to one of four possible states. The states correspond to phases of $\pi/4$, $3\pi/4$, $5\pi/4$, and $7\pi/4$.

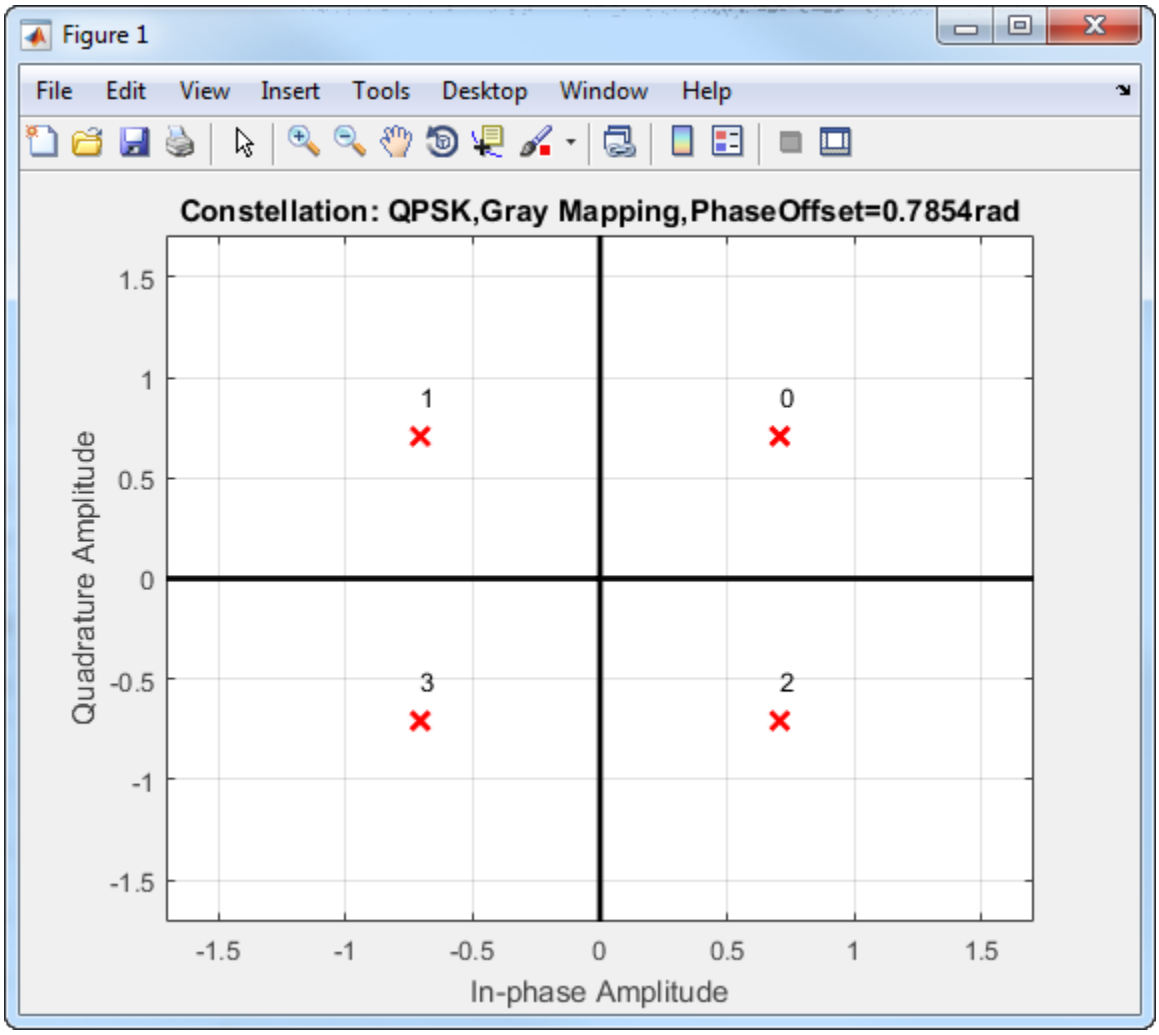


To improve bit error rate performance, the incoming bits can be mapped to a Gray-coded ordering.

Binary-to-Gray Mapping

Binary Sequence	Gray-coded Sequence
00	00
01	01
10	11
11	10

The primary advantage of the Gray code is that only one of the two bits changes when moving between adjacent constellation points. Gray codes can be applied to higher-order modulations, as shown in this Gray-coded QPSK constellation.



The bit error probability for QPSK in AWGN with Gray coding is

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right),$$

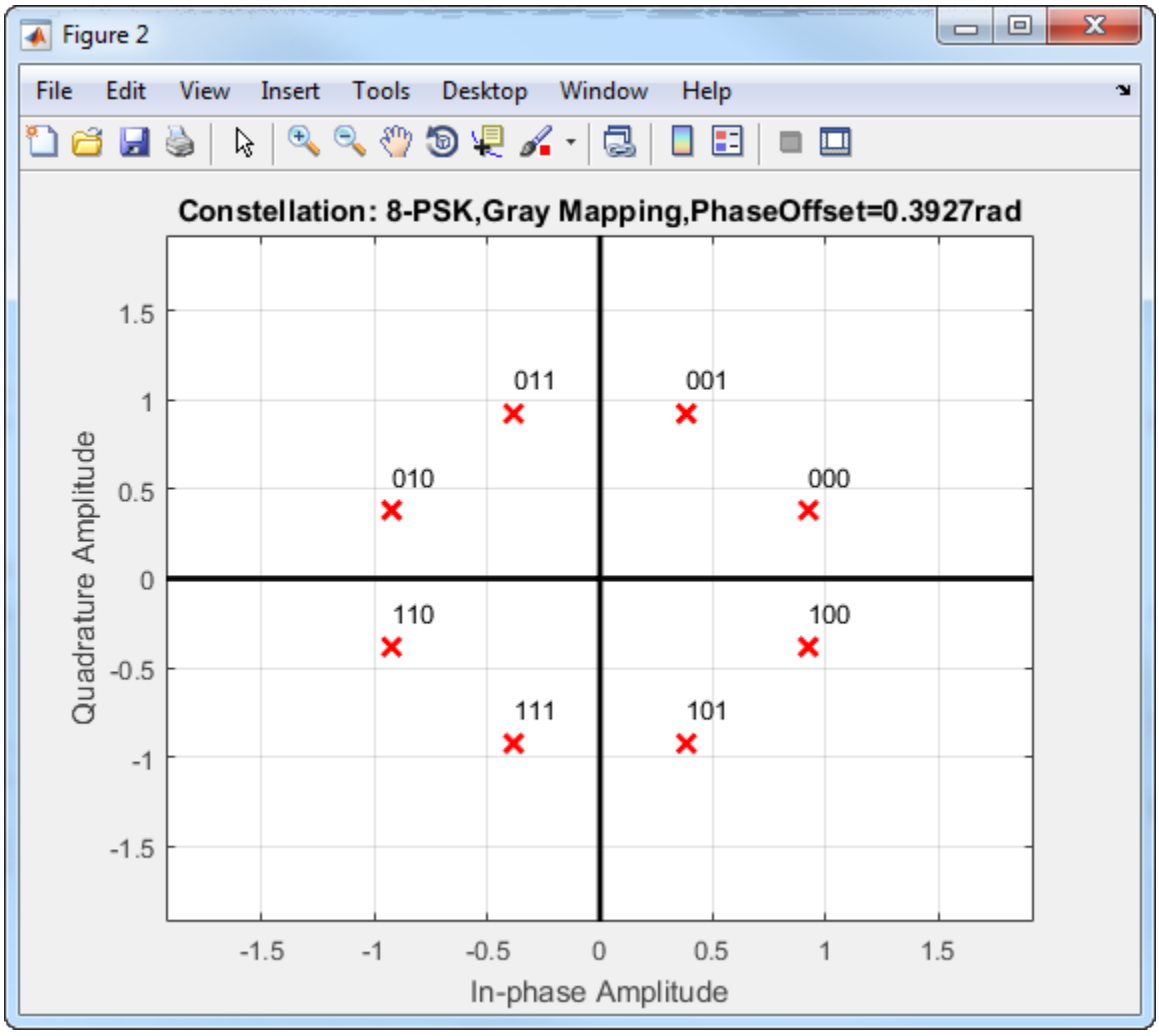
which is the same as the expression for BPSK. As a result, QPSK provides the same performance with twice the bandwidth efficiency.

Higher-Order PSK

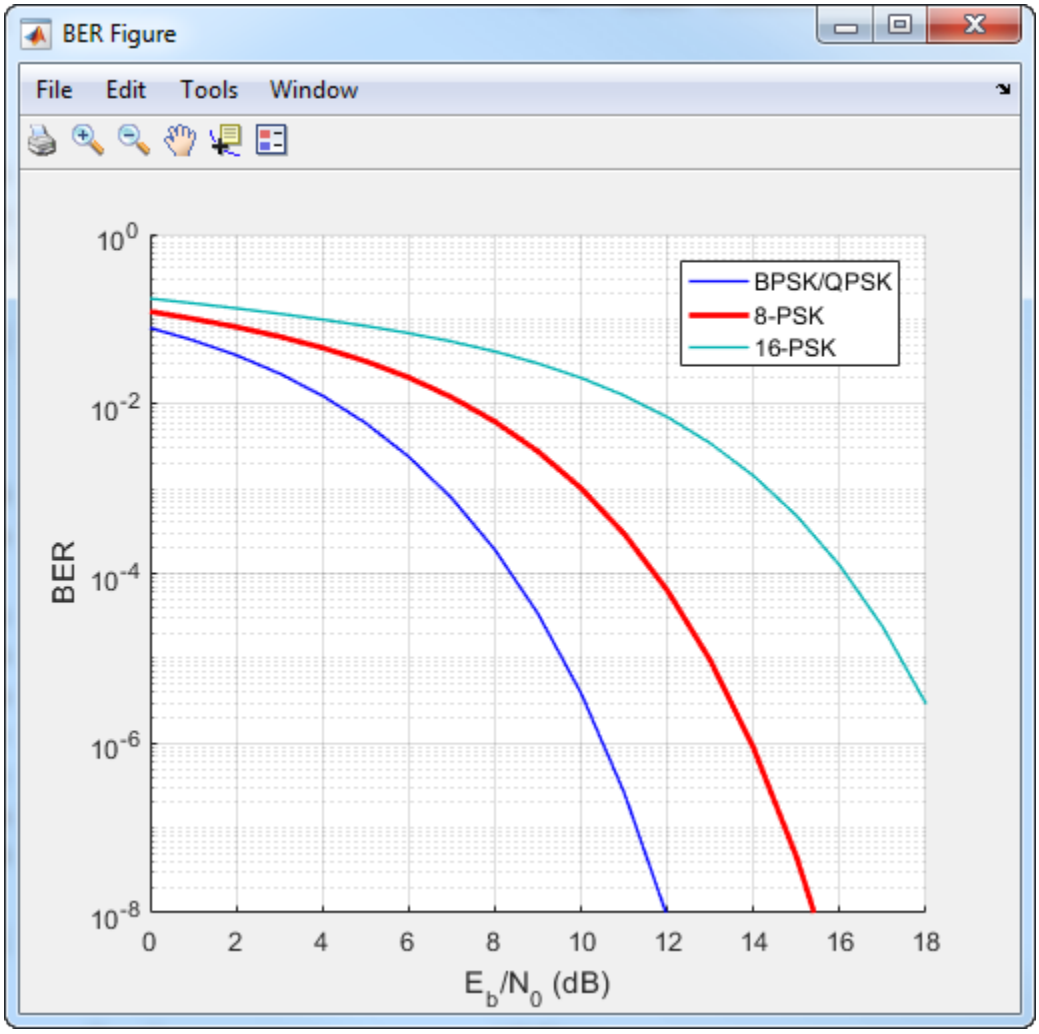
In MATLAB, you can modulate and demodulate higher-order PSK constellations. The complex baseband form for an M-ary PSK signal using natural binary-ordered symbol mapping is

$$s_n(t) = \exp\left(j\pi\left(\frac{2n+1}{M}\right)\right); \quad n \in \{0, 1, \ldots, M-1\}.$$

This 8-PSK constellation uses Gray-coded symbol mapping.



For modulation orders beyond 4, the bit error rate performance of PSK in AWGN worsens. In the following figure, the QPSK and BPSK curves overlap one another.



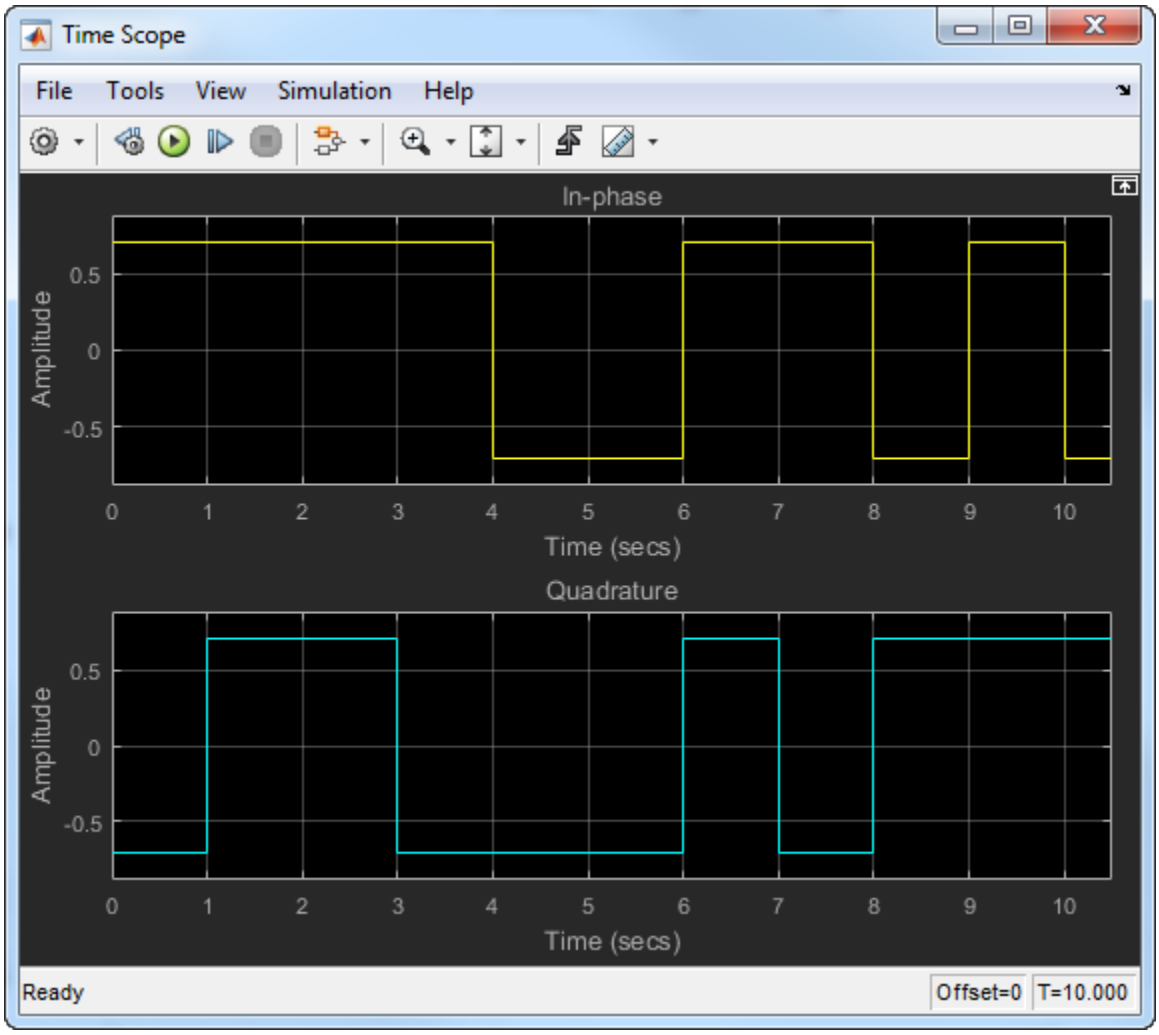
DPSK

DPSK is a noncoherent form of phase shift keying that does not require a coherent reference signal at the receiver. With DPSK, the difference between successive input symbols is mapped to a specific phase. As an example, for binary DPSK (DBPSK), the modulation scheme operates such that the difference between successive bits is mapped to a binary 0 or 1. When the input bit is 1, the differentially encoded symbol remains the same as the previous symbol, while an incoming 0 toggles the output symbol.

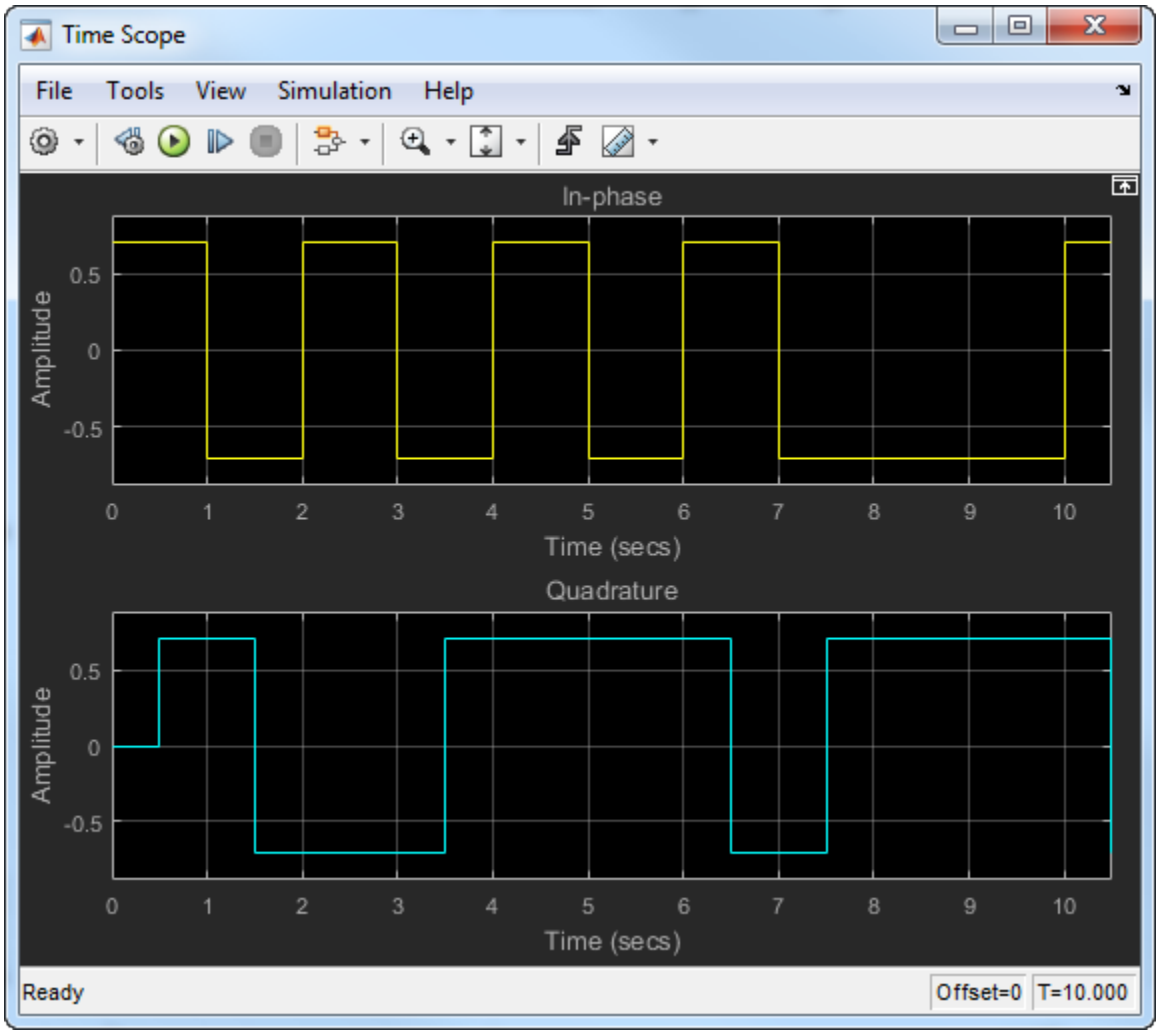
The disadvantage of DPSK is that it is approximately 3 dB less energy efficient than coherent PSK. The bit error probability for DBPSK in AWGN is $P_b = 1/2 \exp(E_b/N_0)$.

OQPSK

Offset QPSK is similar to QPSK except that the time alignment of the in-phase and quadrature bit streams differs. In QPSK, the in-phase and quadrature bit streams transition at the same time. In OQPSK, the transitions have an offset of a half-symbol period as shown.



The in-phase and quadrature signals transition only on boundaries between symbols. These transitions occur at 1-second intervals because the sample rate is 1 Hz. The following figure shows the in-phase and quadrature signals for an OQPSK signal.



For OQPSK, the quadrature signal has a 1/2 symbol period offset (0.5 s).

The BER for an OQPSK signal in AWGN is identical to that of a QPSK signal. The BER is

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right),$$

where E_b is the energy per bit and N_0 is the noise power spectral density.

Soft-Decision Demodulation

All Communications Toolbox demodulator functions, System objects and blocks can demodulate binary data using either hard decisions or soft decisions. Two soft-decision algorithms are available: exact log-likelihood ratio (LLR) and approximate LLR. Exact LLR provides the greatest accuracy but is slower, while approximate LLR is less accurate but more efficient.

Exact LLR Algorithm

The log-likelihood ratio (LLR) is the logarithm of the ratio of probabilities of a 0 bit being transmitted versus a 1 bit being transmitted for a received signal. The LLR for a bit, b, is defined as:

$$L(b) = \log\left(\frac{\Pr(b = 0 \mid r = (x, y))}{\Pr(b = 1 \mid r = (x, y))}\right)$$

Assuming equal probability for all symbols, the LLR for an AWGN channel can be expressed as:

$$L(b) = \log\left(\frac{\sum_{s \in S_0} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}{\sum_{s \in S_1} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}\right)$$

Variable	Description
r	Received signal with coordinates (x, y)
b	Transmitted bit (one of the K bits in an M-ary symbol, assuming all M symbols are equally probable)
S_0	Ideal symbols or constellation points with bit 0, at the given bit position
S_1	Ideal symbols or constellation points with bit 1, at the given bit position
s_x	In-phase coordinate of ideal symbol or constellation point
s_y	Quadrature coordinate of ideal symbol or constellation point
σ^2	Noise variance of baseband signal
σ_x^2	Noise variance along in-phase axis
σ_y^2	Noise variance along quadrature axis

Note Noise components along the in-phase and quadrature axes are assumed to be independent and of equal power, that is, $\sigma_x^2 = \sigma_y^2 = \sigma^2/2$.

Approximate LLR Algorithm

Approximate LLR is computed by using only the nearest constellation point to the received signal with a 0 (or 1) at that bit position, rather than all the constellation points as done in exact LLR. It is defined in [2] as:

$$L(b) = -\frac{1}{\sigma^2}\left(\min_{s \in S_0}\left((x - s_x)^2 + (y - s_y)^2\right) - \min_{s \in S_1}\left((x - s_x)^2 + (y - s_y)^2\right)\right)$$

References

[1] Rappaport, Theodore S. *Wireless Communications: Principles and Practice.* Upper Saddle River, NJ: Prentice Hall, 1996, pp. 238–248.

[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*. Vol. 16, No. 2, Feb. 1998, pp. 260–264

See Also

Related Examples

- “Estimate BER of QPSK in AWGN with Reed-Solomon Coding” on page 19-13
- “Gray-Coded Binary Ordering” on page 17-11
- “Log-Likelihood Ratio (LLR) Demodulation” on page 8-69

Various User Guide Topic Examples

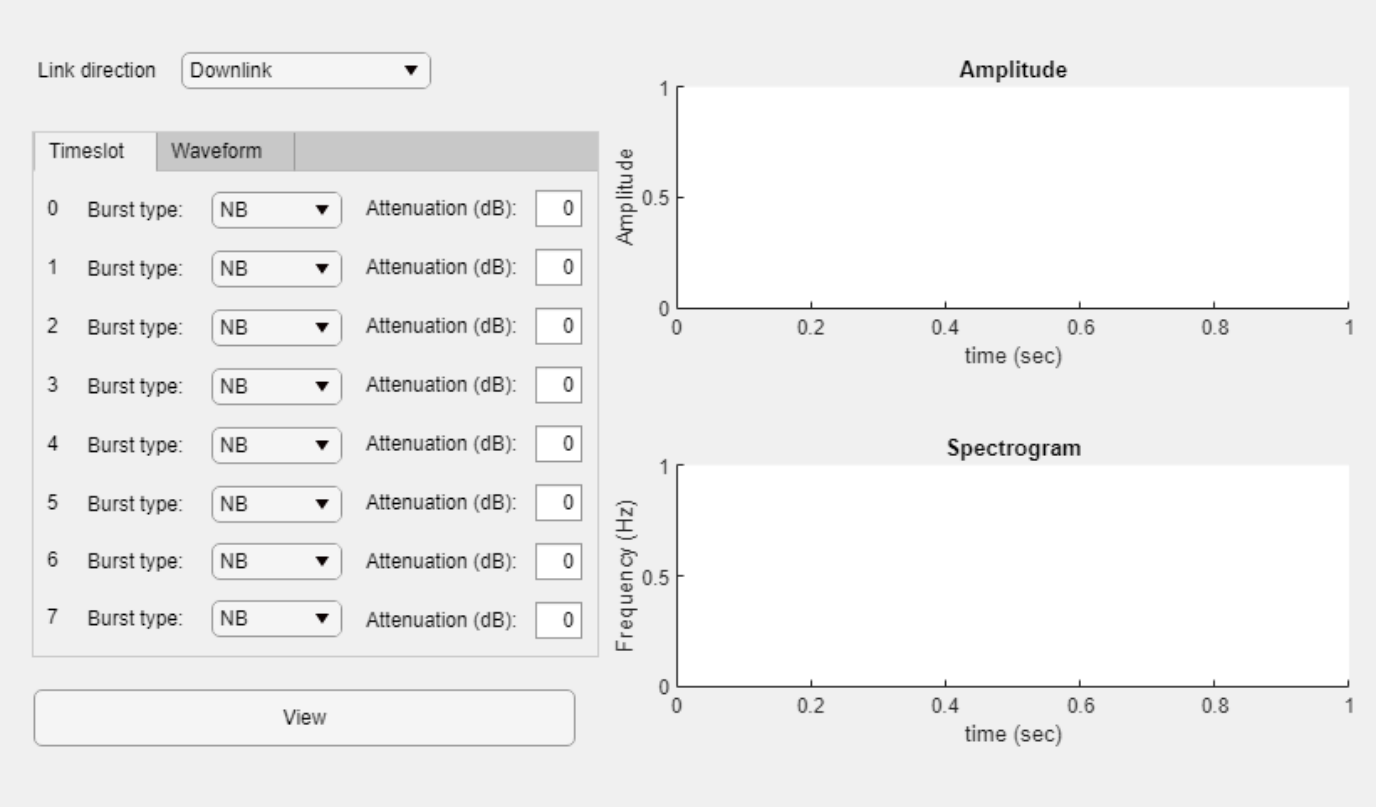
Create a Standalone GSM Waveform Explorer Application with MATLAB Compiler

This example shows how to use MATLAB Compiler™ to create a standalone application from the custom MATLAB™ app, GSMWaveformExplorer, which was created by using App Designer. By installing “MATLAB Runtime” (MATLAB Compiler) you can run standalone applications on systems that do not have MATLAB installed. For more information, see “Create and Run a Simple App Using App Designer”.

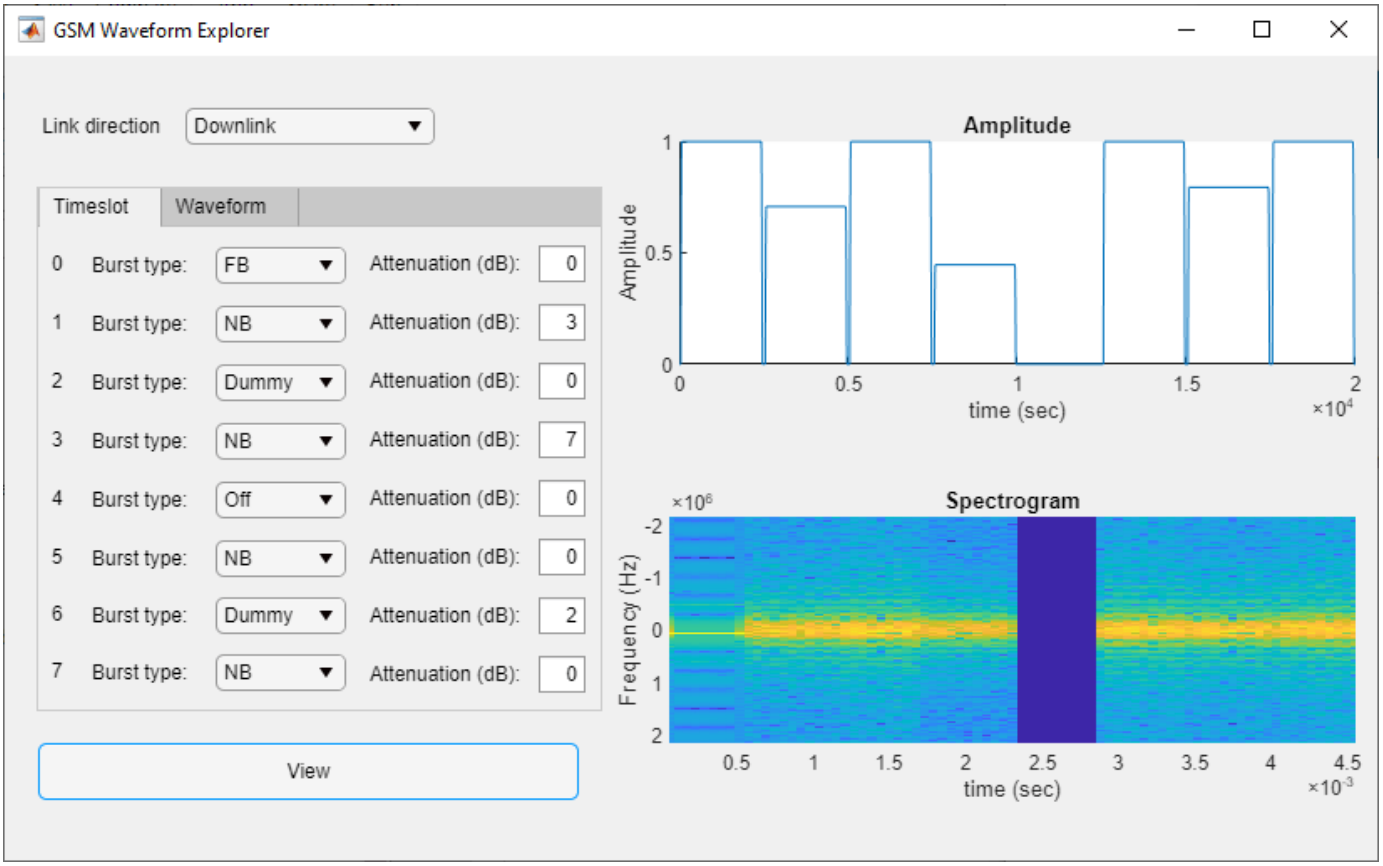
MATLAB Simulation

Open the `GSMWaveformExplorer` app in MATLAB by entering:

```
GSMWaveformExplorer
```



The `GSMWaveformExplorer` app allows you to explore GSM TDMA frame configurations by using the `gsmUplinkConfig` and `gsmDownlinkConfig` objects and the `gsmFrame` function. You can select the **Link direction** as **Uplink** or **Downlink**. In the **Timeslot** tab, you can adjust the burst type and attenuation of individual timeslots. In the **Waveform** tab, you can adjust the samples per symbol and burst shape. Select **View** to visualize the time domain and spectrogram plots of the waveform.



Compile the MATLAB Function into a Standalone Application

Compile `GSMWaveformExplorer` into a standalone application by using the `mcc` (MATLAB Compiler) function and specifying the `'-m'` option. This step takes a few minutes to complete. The first message shown below appears only if your have a network installation and the second message appears only if you are running MATLAB Compiler with a demo license.

```
mcc('-m','GSMWaveformExplorer');
```

```
Disregard cmd.exe warnings about UNC directory pathnames.
DEMO Compiler license.
  The generated application will expire 30 days from today,
  on Fri Jul 10 15:13:33 2020.
```

You can also use the interactive `applicationCompiler` (MATLAB Compiler) app to generate the standalone application.

Run the Standalone Application

Before deploying the standalone app, you can test it on a machine that has MATLAB installed by running commands in the MATLAB Command Window. You can run the standalone `GSMWaveformExplorer` app on a machine that has MATLAB installed by using the `system` command.

- If you are running in the Windows or Linux operating system, type:

```
status = system(fullfile(pwd,'GSMWaveformExplorer'));
```

- If you are running in the Mac operating system, type:

```
status =
system(fullfile('GSMWaveformExplorer.app','Contents','MacOS','GSMWaveformExpl
orer'));
```

Running the standalone application with the system command uses the MATLAB environment and any library files needed from this installation of MATLAB. As with running the app in MATLAB, running the standalone version of the `GSMWaveformExplorer` app opens an GSM Waveform Explorer window that allows you to adjust the GSM TDMA frame configurations and view the waveform.

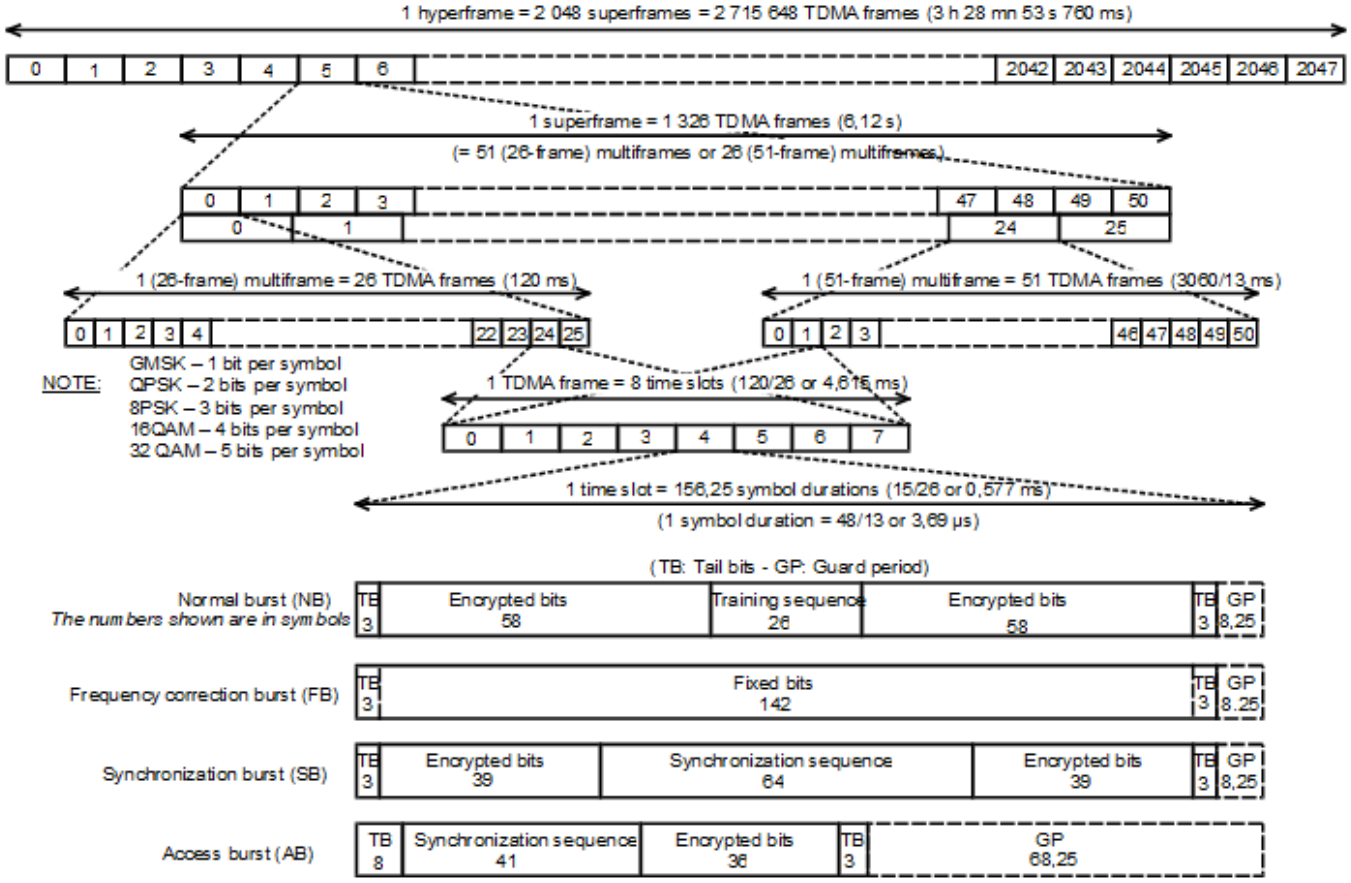
To deploy this application on a machine that does not have MATLAB installed, see “MATLAB Runtime” (MATLAB Compiler).

GSM TDMA Frame Parameterization for Waveform Generation

This example shows how to parameterize and generate different GSM TDMA frames and multiframe structures.

Introduction

The GSM standard [1 on page 12-0] specifies a TDMA frame as a combination of 8 time slots. Each time slot has a duration of 3/5200 seconds (about 0.577 ms) and a time slot number (TN) from 0 to 7. GSM frames use GMSK modulation, where one symbol is equivalent to one bit. Each time slot is 156.25 bits long. The content of a time slot is called a burst. The transmission timing of a burst within a time slot is defined in terms of the bit number. The bit number (BN) refers to a particular bit period within a time slot. The bit with the lowest bit number is transmitted first. BN0 is the first bit period and BN156 is the last quarter-bit period. This figure shows time frames, time slots, and bursts for a GSM system [1 on page 12-0].



A TDMA contains eight time slots with each timeslot separated by a guard period. Each time slot can carry only one type of burst. Available burst type are: normal burst (NB), frequency correction burst (FB), synchronization burst (SB), access burst (AB), or dummy burst [2 on page 12-0]. The different burst types and the guard period are described in these next sections.

Normal Burst (NB)

The normal burst consists of these bit fields and can appear in uplink or downlink frames. All tail bits are zero. Based on the specified training sequence code (TSC), the training sequence field contains one of eight possible training sequences.

```
normalBurstDescription()
```

```
ans=6×3 table
     BitNumber     LengthOfField        ContentsOfField
    ___________    _____________    ________________________

    "0 - 2"         {[     3]}      "tail bits"
    "3 - 60"        {[    58]}      "encrypted bits"
    "61 - 86"       {[    26]}      "training sequence bits"
    "87 - 144"      {[    58]}      "encrypted bits"
    "145 - 147"     {[     3]}      "tail bits"
    "148 - 156"     {[8.2500]}      "guard period (bits)"
```

Access Burst (AB)

The access burst consists of these bit fields and can appear in uplink frames only. All tail bits are zero.

```
accessBurstDescription()
```

```
ans=5×3 table
    BitNumber     LengthOfField       ContentsOfField
    __________    _____________    ____________________

    "0 - 7"         {[      8]}     "extended tail bits"
    "8 - 48"        {[     41]}     "synch. sequence bits"
    "49 - 84"       {[     36]}     "encrypted bits"
    "85 - 87"       {[      3]}     "tail bits"
    "88 - 156"      {[68.2500]}     "guard period (bits)"
```

Frequency Correction Burst (FB)

The frequency correction burst consists of these bit fields and can appear in downlink frames only. All tail bits and fixed bits are zero. Modulating all zeros with the GMSK modulator results in a constant phase rotation of -90 degrees for each symbol duration. Therefore, this burst generates an unmodulated carrier with a positive frequency offset of 1625/24 kHz.

```
frequencyCorrectionBurstDescription()
```

```
ans=4×3 table
     BitNumber     LengthOfField       ContentsOfField
    ___________    _____________    ____________________

    "0 - 2"         {[     3]}      "tail bits"
    "3 - 144"       {[   142]}      "fixed bits"
    "145 - 147"     {[     3]}      "tail bits"
    "148 - 156"     {[8.2500]}      "guard period (bits)"
```

Synchronization Burst (SB)

The synchronization burst consists of these bit fields and can appear in downlink frames only. All tail bits are zero.

```
synchronizationBurstDescription()
```

```
ans=6×3 table
     BitNumber     LengthOfField             ContentsOfField
    ___________    _____________    ____________________________________

    "0 - 2"         {[     3]}      "tail bits"
    "3 - 41"        {[    39]}      "encrypted bits"
    "42 - 105"      {[    64]}      "extended training sequence bits"
    "106 - 144"     {[    39]}      "encrypted bits"
    "145 - 147"     {[     3]}      "tail bits"
    "148 - 156"     {[8.2500]}      "guard period (bits)"
```

Dummy Burst

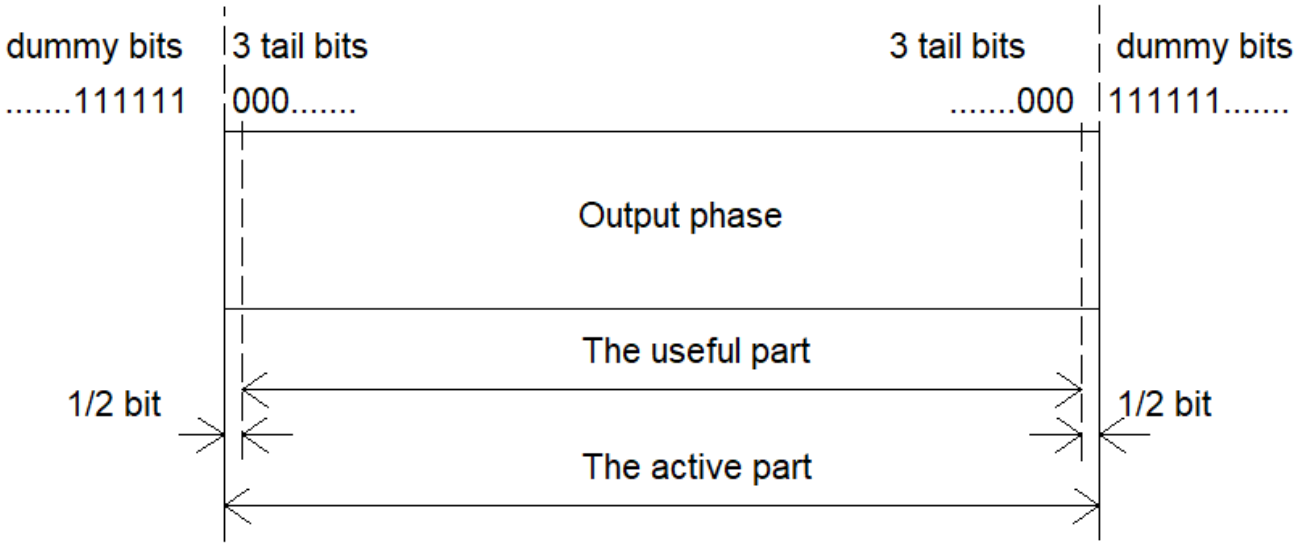
The dummy burst burst consists of these bit fields and can appear in downlink frames only. All tail bits are zero. Mixed bits contain a predetermined sequence of ones and zeros.

```
dummyBurstDescription()
```

```
ans=4×3 table
     BitNumber     LengthOfField       ContentsOfField
    ___________    _____________    ____________________

    "0 - 2"         {[     3]}      "tail bits"
    "3 - 144"       {[   142]}      "mixed bits"
    "145 - 147"     {[     3]}      "tail bits"
    "148 - 156"     {[8.2500]}      "guard period (bits)"
```

Guard Period

The GSM standard, [3 on page 12-0], requires mobile stations to attenuate their transmission during the period between bursts. The ramp-up and ramp-down of the signal power level occurs during the guard periods. The useful part of a burst starts half way through the bit number 0. The useful part ends halfway through BN87 for ABs and BN147 for NBs, FBs, SBs,and dummy bursts. This figure shows the useful and active parts of a burst.



Generate Single Uplink Frame

Configure an uplink GSM TDMA frame using the `gsmUplinkConfig` object.

```
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

   Read-only properties:
    No properties.
```

Set time slots 2 and 5 to carry access bursts. Since MATLAB array indices start from 1, but time slots start from 0, set the third and sixth elements of the `BurstType` to `"AB"`.

```
cfg.BurstType([2 5] +1) = "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

   Read-only properties:
    No properties.
```

Assign training sequence codes 3, 5, 1, 7, 0, and 2 to time slots 0, 1, 3, 4, 6, and 7, respectively.

```
cfg.TSC([0 1 3 4 6 7] +1) = [3 5 1 7 0 2]
```

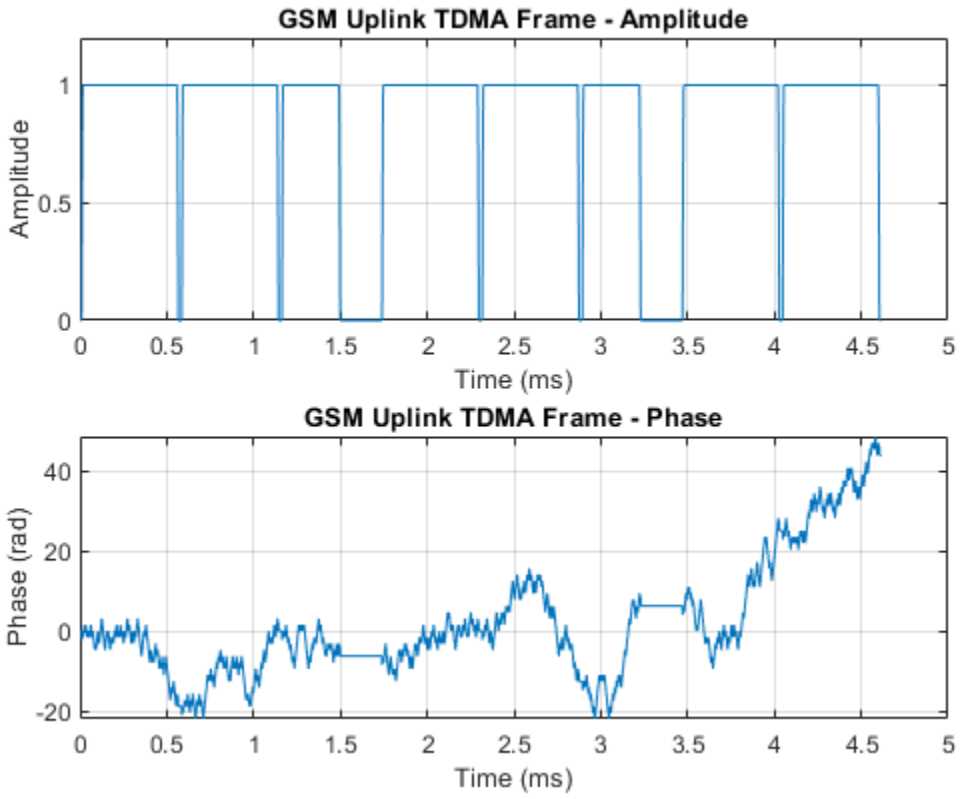
```
cfg =
  gsmUplinkConfig with properties:

          BurstType: [NB    NB    AB    NB    NB    AB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [3 5 2 1 7 5 0 2]
        Attenuation: [0 0 0 0 0 0 0 0]
           RiseTime: 2
          RiseDelay: 0
           FallTime: 2
          FallDelay: 0

   Read-only properties:
    No properties.
```

Generate the baseband samples of the frame using the `gsmFrame` function.

```
x = gsmFrame(cfg);
```

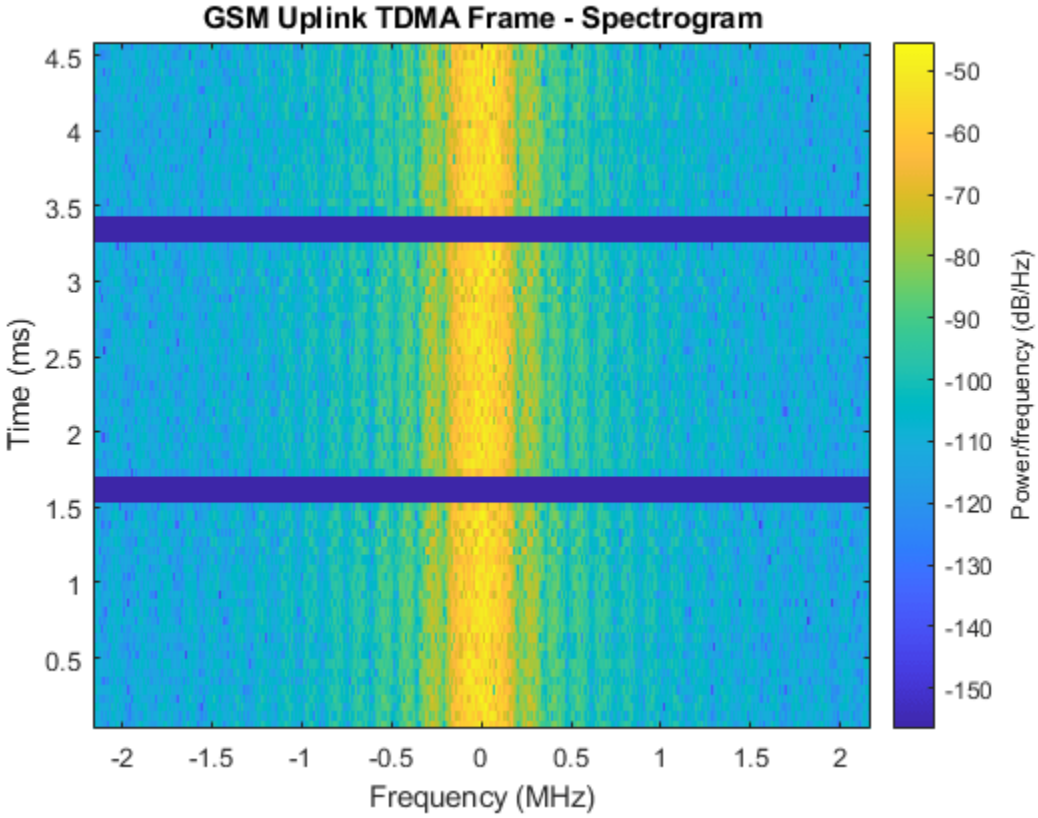
Plot the frame. Get the sample rate of the generated waveform by using the `gsmInfo` function, and then calculate the time axis values in ms. The plot shows 8 bursts in the frame, with guard periods between each burst. As described in the Access Burst (AB) on page 12-0 section, ABs are short burst and have a wider guard period than other bursts.

```
wfInfo = gsmInfo(cfg);
Rs = wfInfo.SampleRate;
t = (0:length(x) - 1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(x))
grid on
axis([0 5 0 1.2])
title('GSM Uplink TDMA Frame - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(x)))
grid on
title('GSM Uplink TDMA Frame - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Plot the spectrogram of the frame.

```
figure
spectrogram(x,500,[],[],Rs,'centered')
title('GSM Uplink TDMA Frame - Spectrogram')
```



Generate Single Downlink Frame

Configure a downlink GSM TDMA frame using the `gsmDownlinkConfig` object.

```
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

   Read-only properties:
    No properties.
```

Set time slots 0 to carry a frequency correction burst, set time slots 4 and 6 to carry dummy bursts, and set time slot 2 to be empty.

```
cfg.BurstType(0 +1) = "FB";
cfg.BurstType([4 6] +1) = "Dummy";
cfg.BurstType(2 +1) = "Off"
```

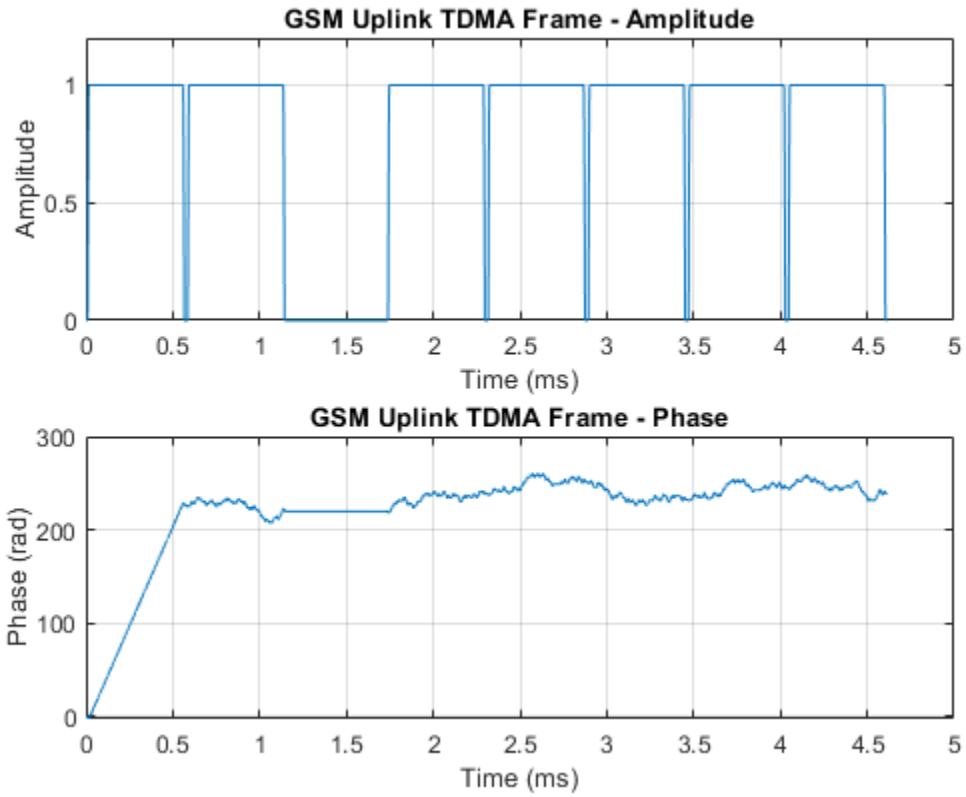
```
cfg =
  gsmDownlinkConfig with properties:

           BurstType: [FB    NB    Off    NB    Dummy    NB    Dummy    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

   Read-only properties:
    No properties.
```

Generate the baseband samples of the frame using the `gsmFrame` function. This function inserts random bits instead of encrypted bits.

```
x = gsmFrame(cfg);
```

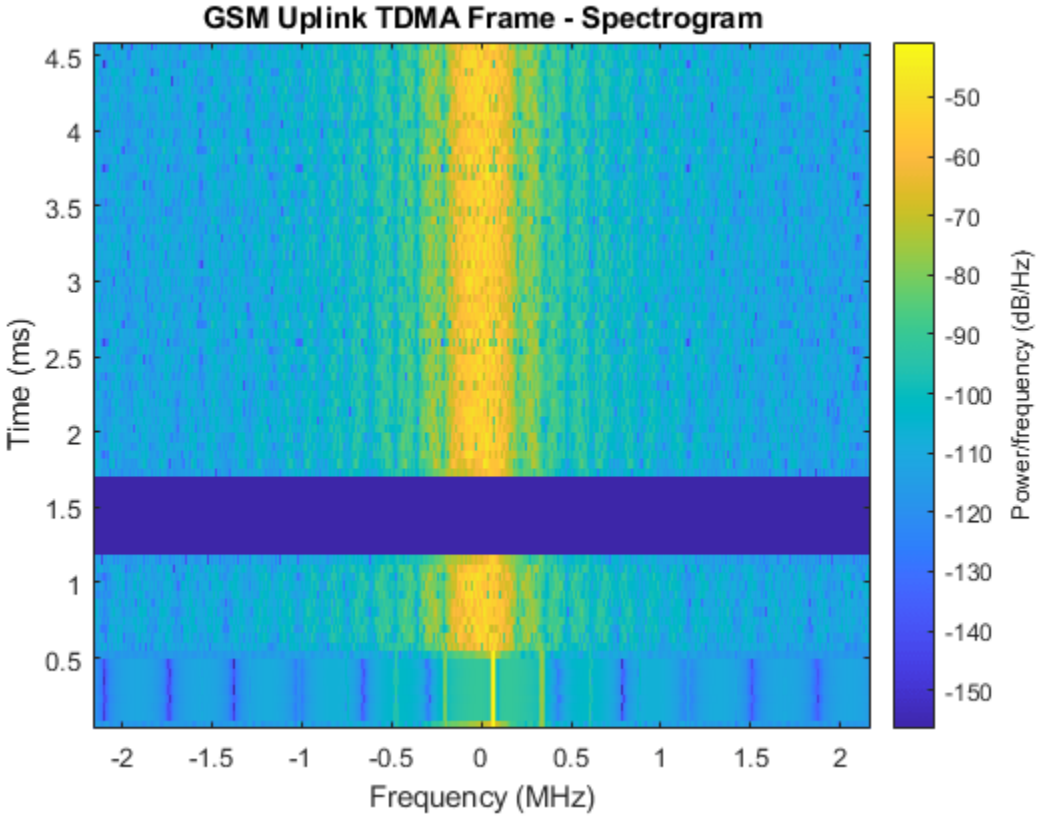
Plot the frame.

```
wfInfo = gsmInfo(cfg);
Rs = wfInfo.SampleRate;
t = (0:length(x) - 1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(x))
grid on
axis([0 5 0 1.2])
title('GSM Uplink TDMA Frame - Amplitude')
xlabel('Time (ms)');ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(x)))
grid on
title('GSM Uplink TDMA Frame - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Plot the spectrogram of the frame. This plot shows the single tone during time slot 0 due to the FB.

```
figure
spectrogram(x,500,[],[],Rs,'centered')
title('GSM Uplink TDMA Frame - Spectrogram')
```



Generate Multiframe Structure

Create a 51-frame multiframe structure, as shown in the figure in the Introduction on page 12-0 section. Create three `gsmDownlinkConfig` objects with specified burst configurations. To assembly the 51-frame multiframe, use the first and second `gsmDownlinkConfig` objects once and repeat the third `gsmDownlinkConfig` objects for the next 49 frames. Repeat the multiframe structure 3 times.

```
cfg1 = gsmDownlinkConfig('BurstType',["FB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
cfg2 = gsmDownlinkConfig('BurstType',["SB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
cfg3 = gsmDownlinkConfig('BurstType',["NB" "NB" "NB" "NB" "NB" "Dummy" "NB" "NB"]);
wfInfo = gsmInfo(cfg);
frameLength = wfInfo.FrameLengthInSamples;
x = zeros(frameLength*51*3,1);
for p=1:3
    x1 = gsmFrame(cfg1);
    x2 = gsmFrame(cfg2);
    x3 = gsmFrame(cfg3,49);
    x((p-1)*frameLength*51+1:p*frameLength*51) = [x1;x2;x3];
end
```

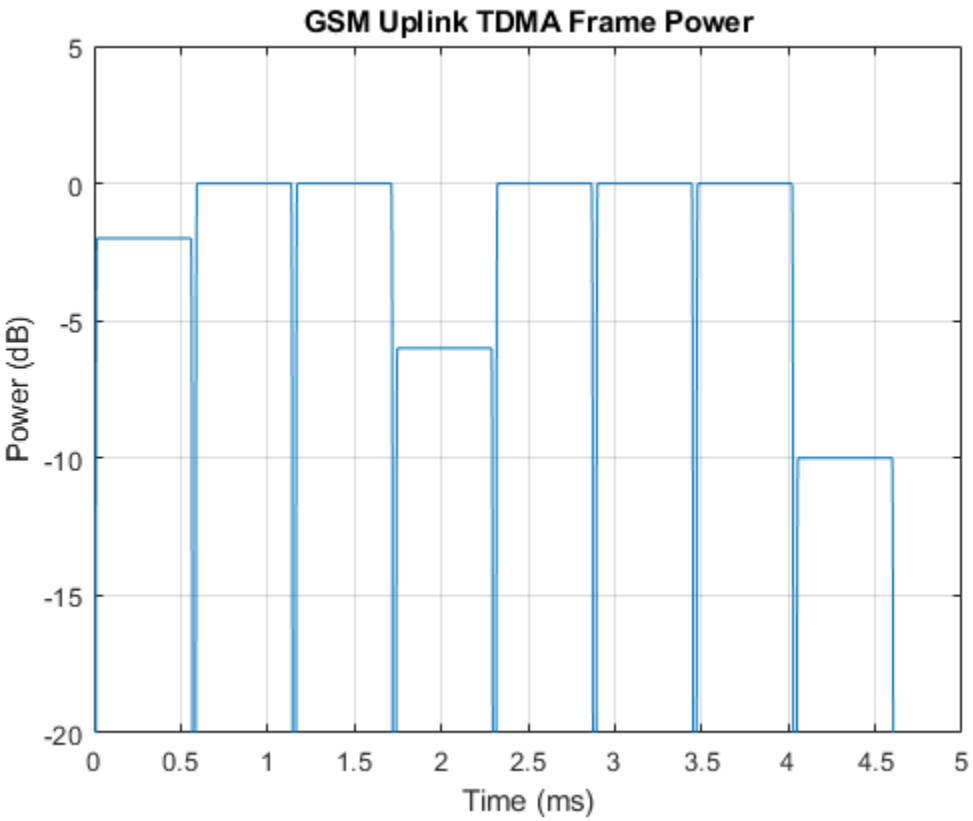
Simulate Power Control and Propagation Loss Effects

Due to power control and unique propagation loss for each user, the power of each time slot within a frame might be different. Set the power attenuation for time slots 0, 3, and 7 to 2, 6, and 10 dB, respectively.

```
cfg = gsmUplinkConfig;
cfg.Attenuation([0 3 7] +1) = [2 6 10]
```

```
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: [2 0 0 6 0 0 0 10]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0

   Read-only properties:
    No properties.
```

```
x = gsmFrame(cfg);
wfInfo = gsmInfo(cfg);
Rs = wfInfo.SampleRate;
t = (0:length(x) - 1)/Rs*1e3;
plot(t, 20*log10(abs(x)))
axis([0 5 -20 5])
grid on
title('GSM Uplink TDMA Frame Power')
xlabel('Time (ms)')
ylabel('Power (dB)')
```



Adjust Ramp-Up and Ramp-Down Behavior

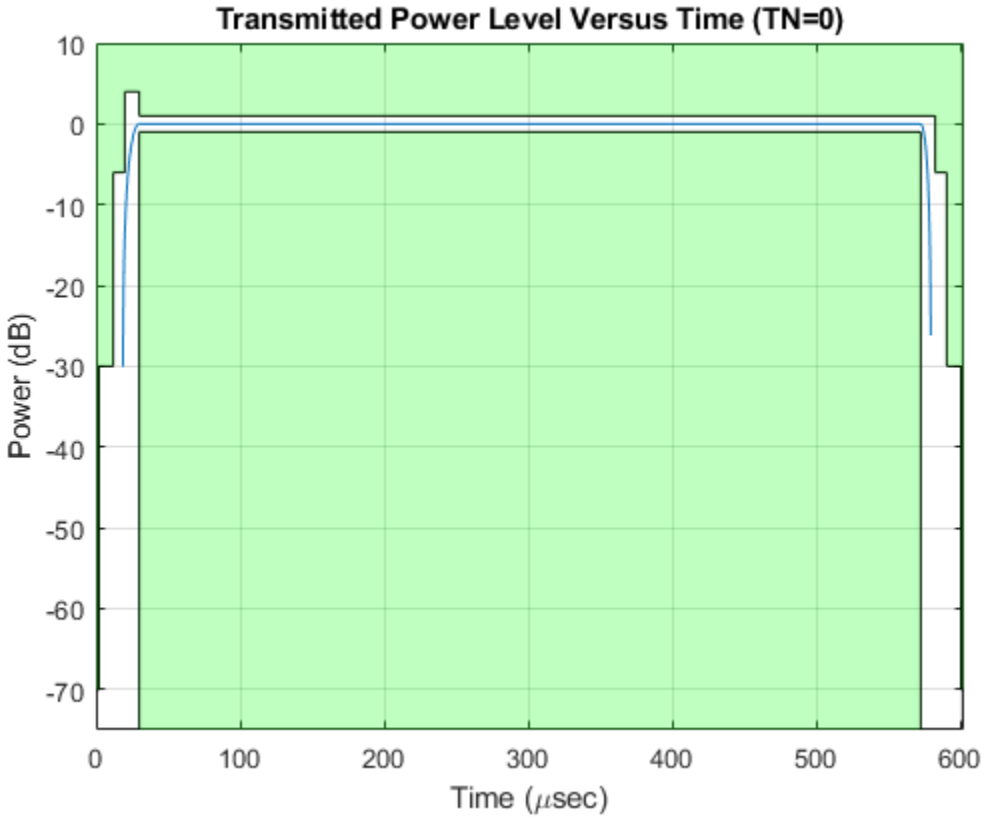
GSM bursts must ramp up and ramp down during guard periods [2] on page 12-0 . The `gsmFrame` function implements the rise and fall characteristics of the bursts as a sinusoid. The burst ramps up from zero to full amplitude in the number of symbol durations specified by the `RiseTime` property value. The resolution of `RiseTime` is $1/N_{sps}$, where N_{sps} represents the `SamplesPerSymbol` property value of the `gsmDownlinkConfig` object.

Adjust the ramp-up characteristics of the bursts. Since `SamplesPerFrame` is 16, you can specify `RiseTime` with a symbol duration resolution of 0.0625. Set `RiseTime` to a duration of 3.125 symbols.

```
cfg = gsmDownlinkConfig;
cfg.RiseTime = 3.125;
```

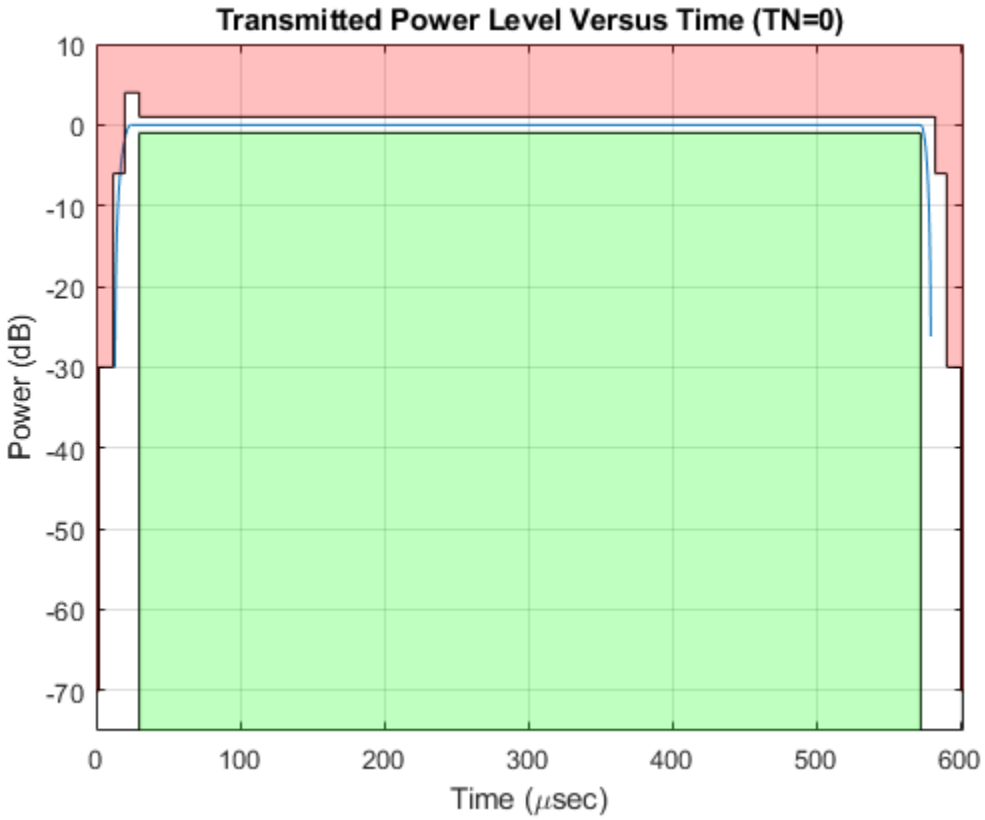
Visualize and check if the rise-time characteristics are within the GSM specifications by using the `gsmCheckTimeMask` function.

```
gsmCheckTimeMask(cfg)
```



Move the start of the rise time duration to the left by 1.5 symbols by setting the `RiseDelay` to `-1.5`. When `RiseDelay` is `0`, the burst reaches full amplitude at the start of the useful part of the burst.

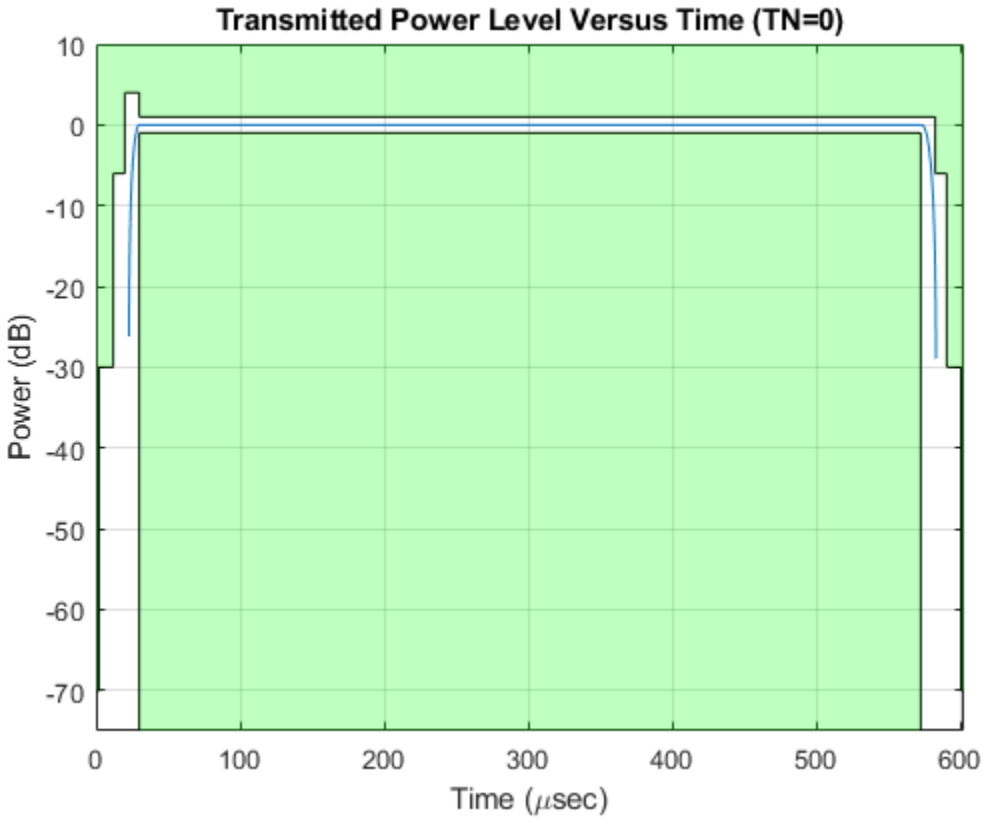
```
cfg.RiseDelay = -1.5;
gsmCheckTimeMask(cfg)
```



The burst ramps down from full amplitude to zero in the number of symbol durations specified by the Fall`Time` property. The resolution of `FallTime` is $1/N_{sps}$, where N_{sps} represents the `SamplesPerSymbol` property value of the `gsmDownlinkConfig` object. Set `FallTime` to a duration of 2.75 symbols.

Move the start of the fall time to the right by 0.25 symbols durations by setting the `FallDelay` to `0.25`. When Fall`Delay` is `0`, the burst starts to ramp down from full amplitude at the end of the useful part of the burst.

```
cfg = gsmDownlinkConfig;
cfg.FallTime = 2.75;
cfg.FallDelay = 0.25;
gsmCheckTimeMask(cfg)
```



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." *General description*." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*

Helper Functions

normalBurstDescription

This function formats a table to show information about normal burst fields.

```
function d = normalBurstDescription()
BitNumber = ["0 - 2";"3 - 60";"61 - 86";...
    "87 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;58;26;58;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "encrypted bits";...
    "training sequence bits";...
    "encrypted bits";...
    "tail bits";...
```

```
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```

frequencyCorrectionBurstDescription

This function formats a table to show information about frequency correction burst fields.

```
function d = frequencyCorrectionBurstDescription()
BitNumber = ["0 - 2";"3 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;142;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "fixed bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```

synchronizationBurstDescription

This function formats a table to show information about synchronization burst fields.

```
function d = synchronizationBurstDescription()
BitNumber = ["0 - 2";"3 - 41";"42 - 105";...
    "106 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;39;64;39;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "encrypted bits";...
    "extended training sequence bits";...
    "encrypted bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```

dummyBurstDescription

This function formats a table to show information about dummy burst fields.

```
function d = dummyBurstDescription()
BitNumber = ["0 - 2";"3 - 144";"145 - 147";"148 - 156"];
LengthOfField = {3;142;3;8.25};
ContentsOfField = [...
    "tail bits";...
    "mixed bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```

accessBurstDescription

This function formats a table to show information about access burst fields.

```
function d = accessBurstDescription()
BitNumber = ["0 - 7";"8 - 48";"49 - 84";...
    "85 - 87";"88 - 156"];
LengthOfField = {8;41;36;3;68.25};
ContentsOfField = [...
    "extended tail bits";...
    "synch. sequence bits";...
    "encrypted bits";...
    "tail bits";...
    "guard period (bits)"...
    ];
d = table(BitNumber,LengthOfField,ContentsOfField);
end
```

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.

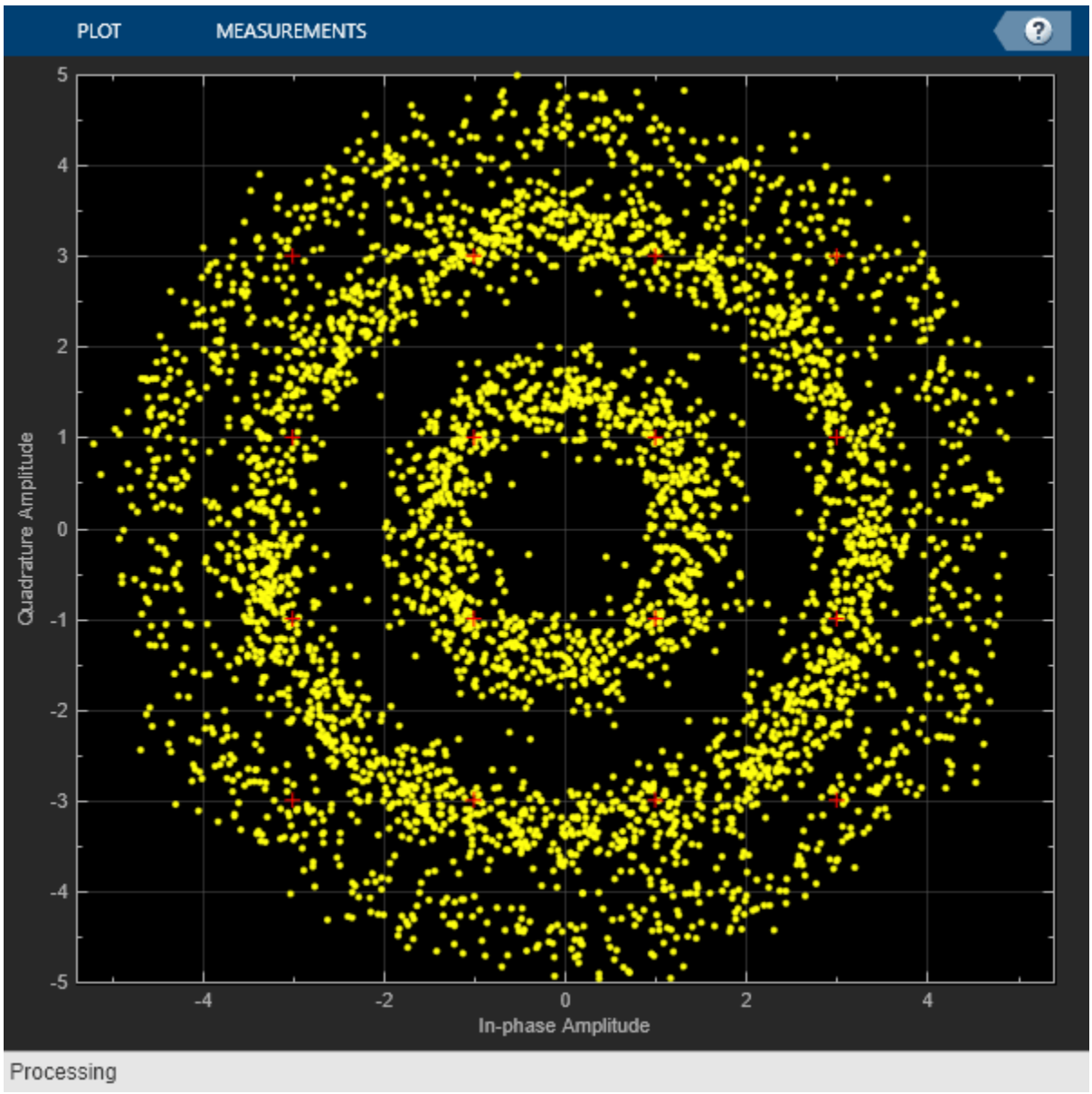
```
freqOffsetSig = phaseFreqOffset(txSig);
rxSig = channel(freqOffsetSig);
```

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.

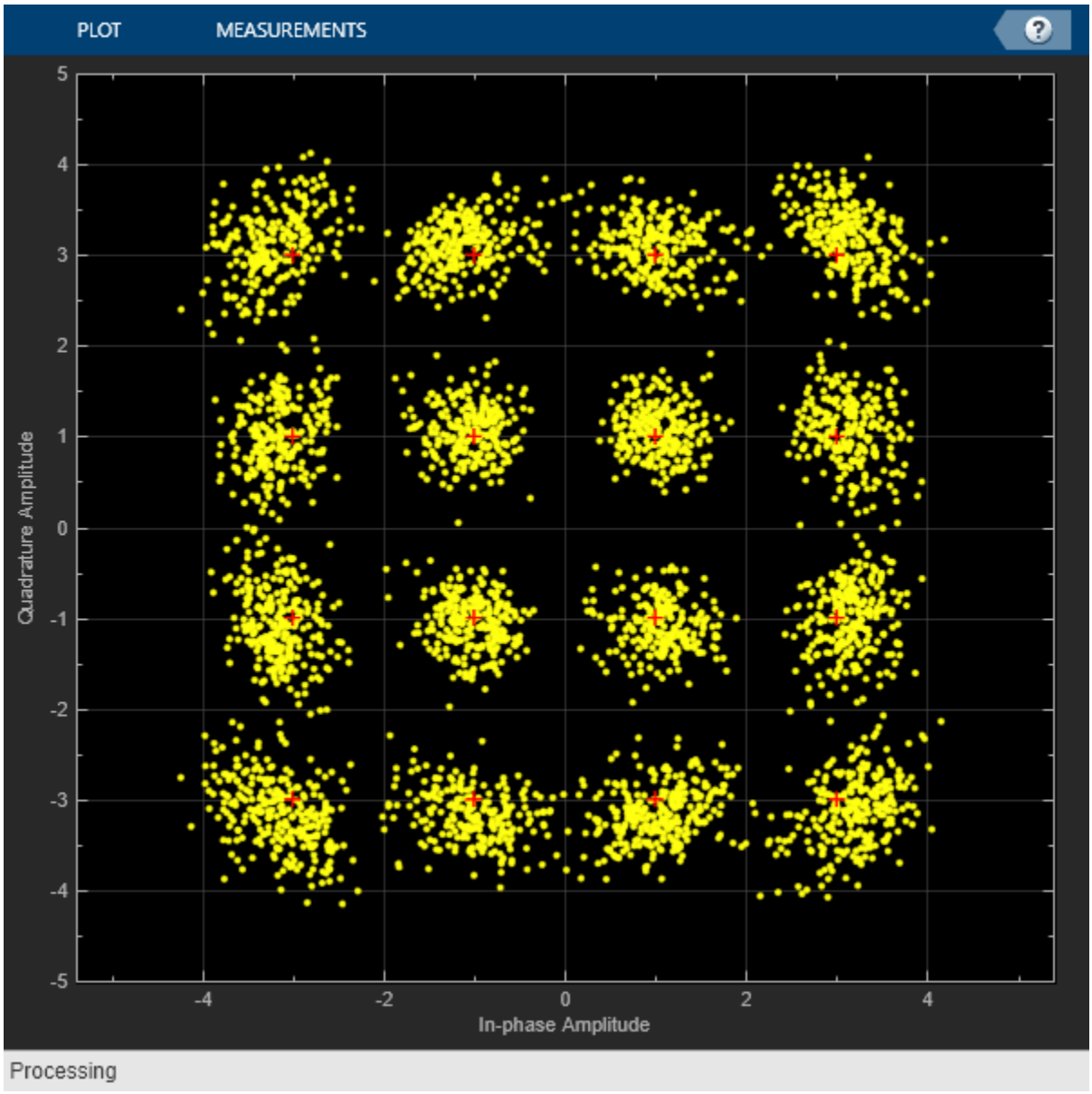
```
coarseSync = comm.CoarseFrequencyCompensator('Modulation','QAM','FrequencyResolution',1,'SampleRa
```

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.

See Also

`comm.CoarseFrequencyCompensator` | `comm.CarrierSynchronizer`

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.

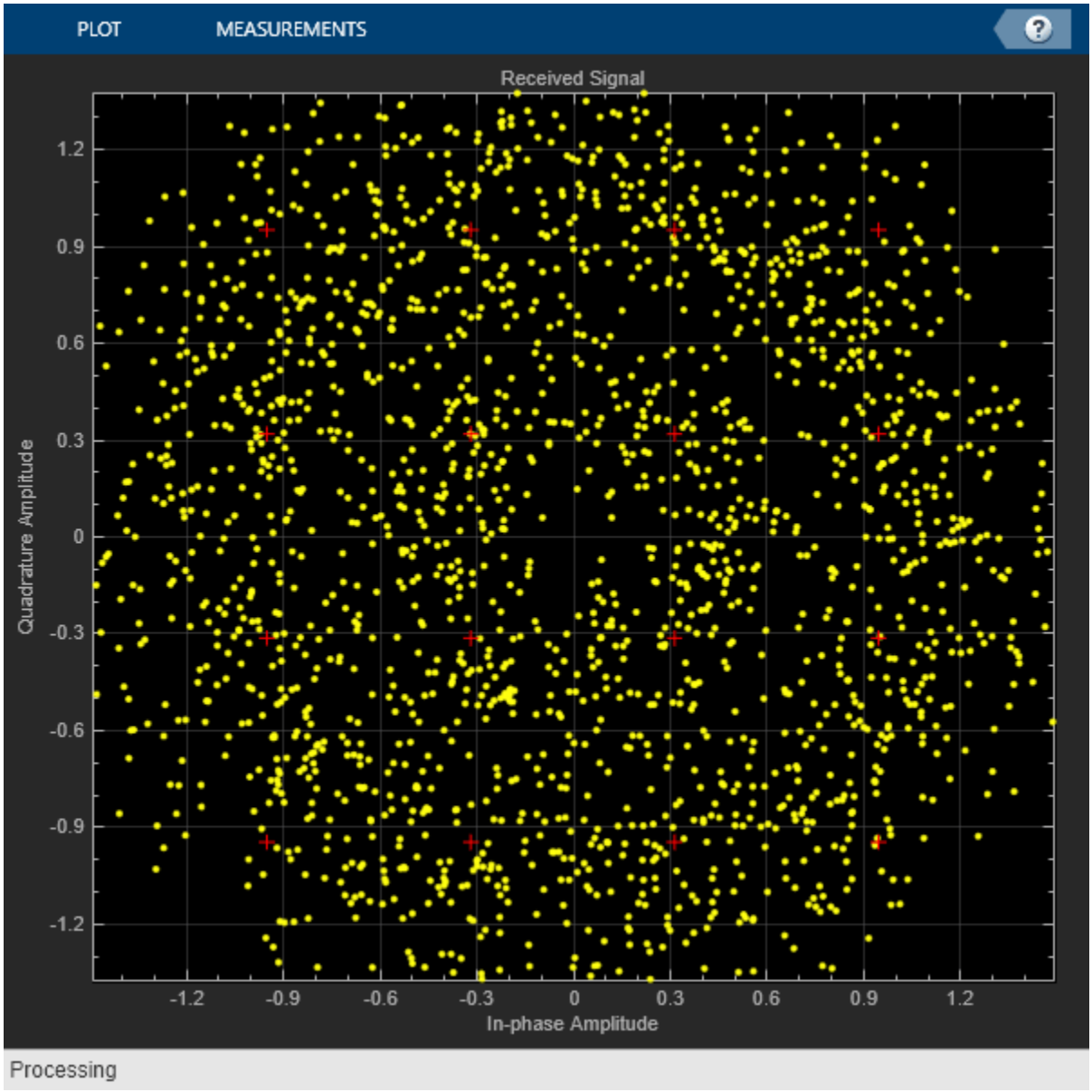
- Applies frequency and timing offsets.
- Passes the transmitted signal through an AWGN channel.
- Filters the received signal.
- Corrects the Doppler shift.
- Corrects the timing offset.

```
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
```

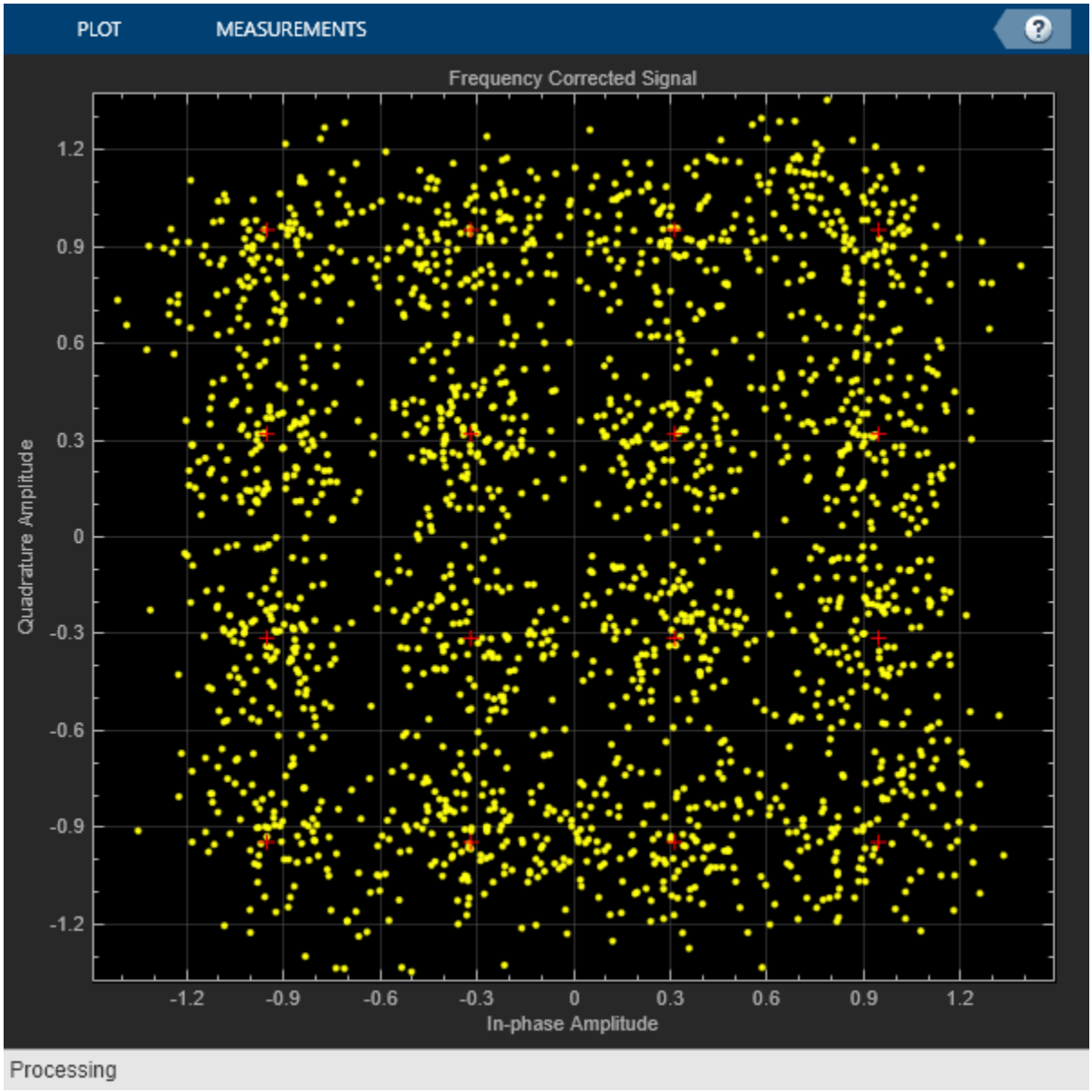
Visualization

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.

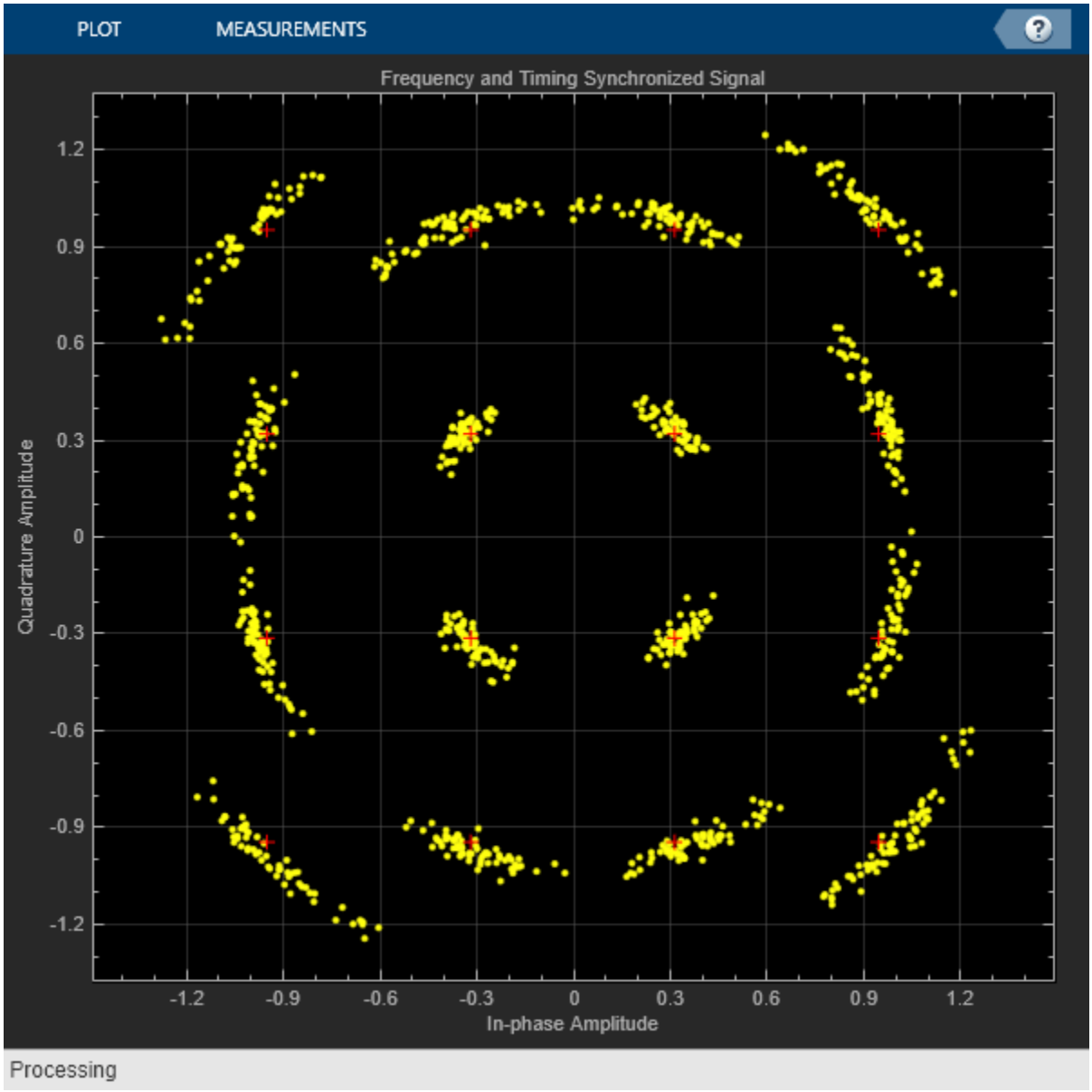
```
cdReceive(rxSig)
```




```
cdDoppler(rxCorr)
```




```
cdTiming(rxData)
```

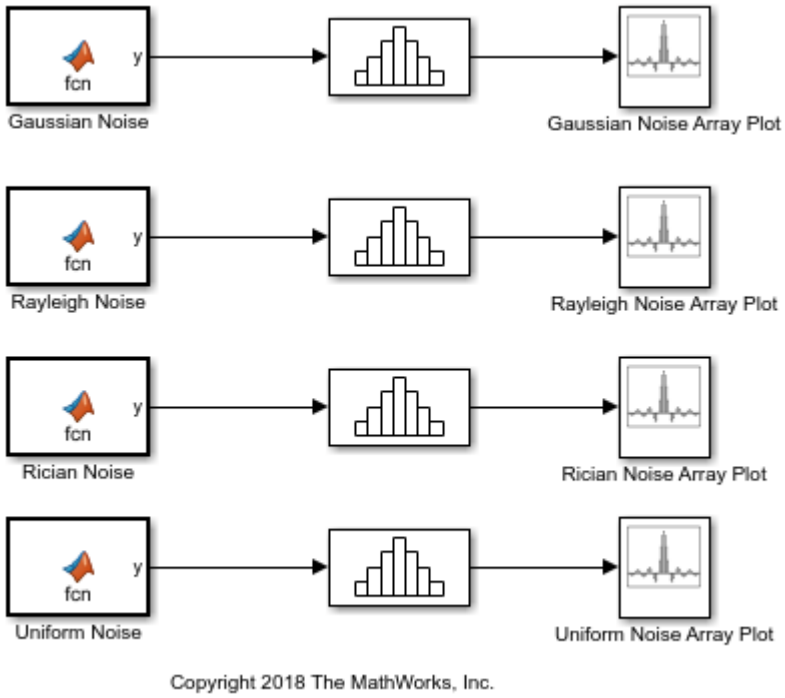


See Also

`comm.CarrierSynchronizer` | `comm.SymbolSynchronizer`

Random Noise Generators in Simulink

You can generate noise for communication system modeling using the MATLAB® Function block with a random number generator. This example generates and displays histogram plots of Gaussian, Rayleigh, Rician, and Uniform noise.



The noise generators output 1e5-by-1 vectors every second, which is equivalent to a 0.00001 second sample time. In this model, each MATLAB Function block defines a specific noise generator using its underlying function. To view the underlying code for a MATLAB Function block in the MATLAB Editor, open the model, select the desired MATLAB Function block, and then press **Ctrl+u**. Each MATLAB function block contains block mask parameters that map to the function arguments in the underlying code.

For each MATLAB Function block the **Samples per frame** parameter maps to its underlying function argument `spf`. Similarly, **Seed** maps to `seed`.

The **Gaussian Noise** MATLAB Function block maps the **Power (dBW)** parameter to `p`, and defines the function

$$y = wgn(spf, 1, p)$$

The **Rayleigh Noise** MATLAB Function block maps the **Sigma** parameter to `alpha`, and defines the function

$$y = abs(alpha * (randn(spf, 1) + 1i * randn(spf, 1)))$$

The **Rician Noise** MATLAB Function block maps the **Rician K-factor** parameter to `K` and the **Sigma** parameter to `s`, and defines the function

$$m1 = sqrt(2 * K). * s$$

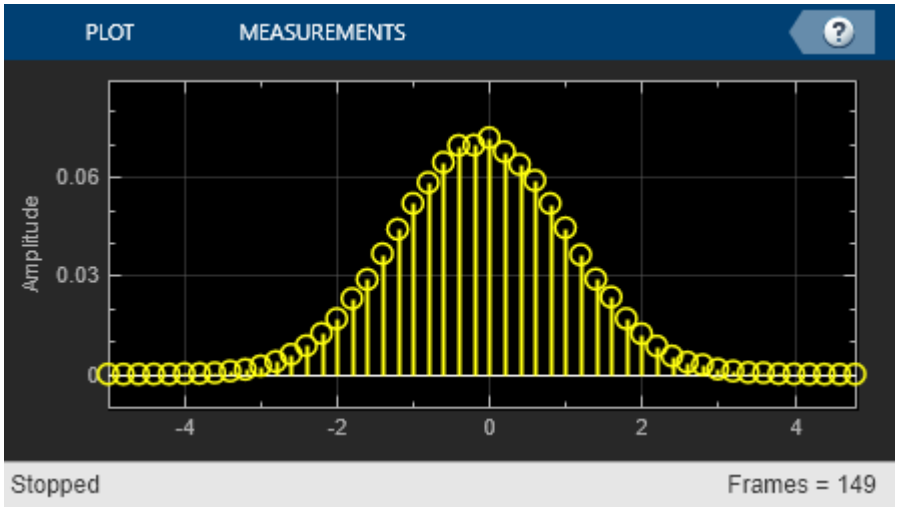
$$m2 = 0$$

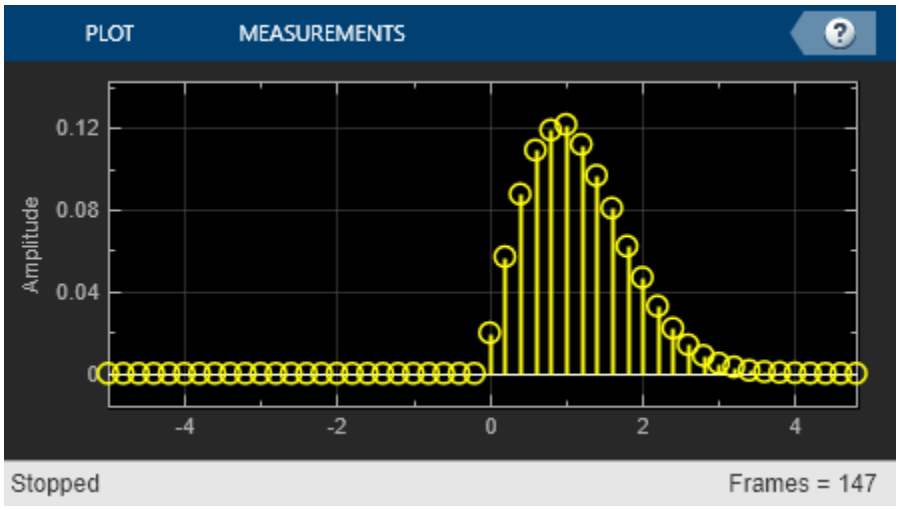
$$y = sqrt((s^2 * randn(spf, 1) + m1)^2 + (s^2 * randn(spf, 1) + m2)^2)$$

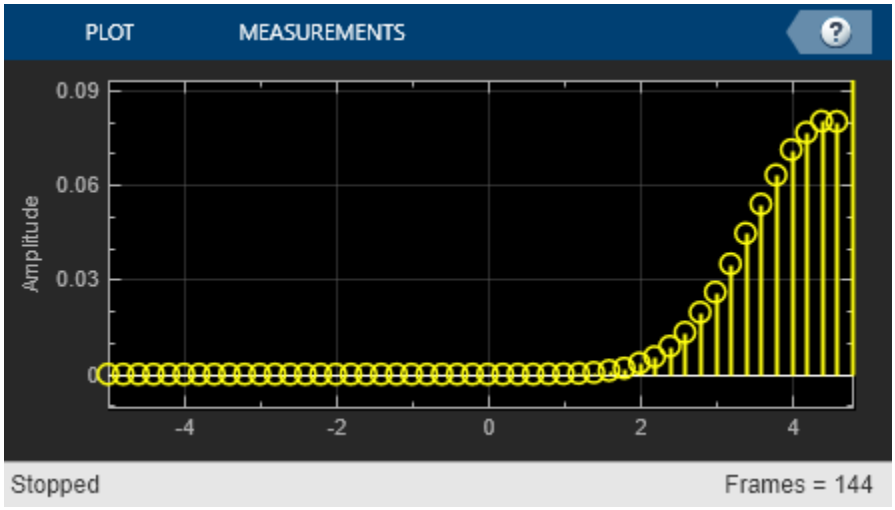
The **Uniform Noise** MATLAB Function block maps the **Noise lower bound** parameter to `lb` and the **Noise upper bound** parameter to `ub`, and defines the function

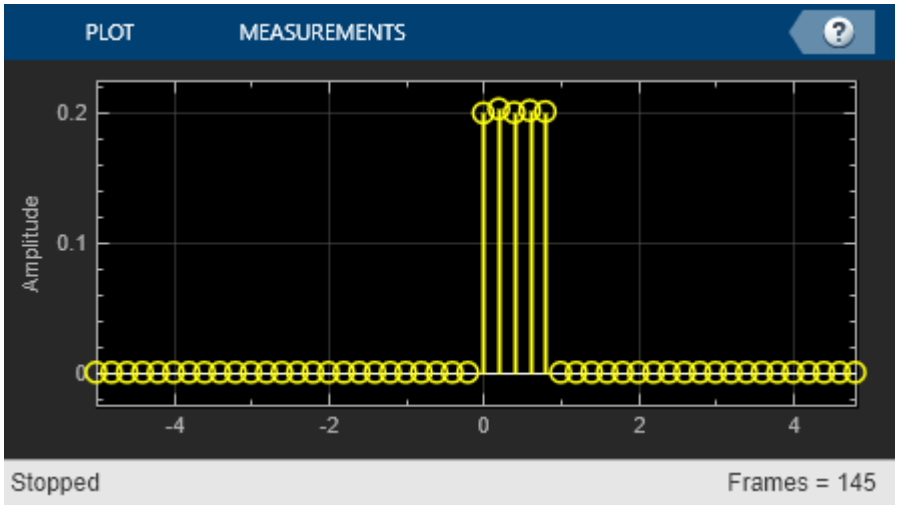
$$y = lb + (ub - lb). * rand(spf, 1)$$

The model generates these histogram plots to show the noise distribution across the spectrum for each noise generator.









For further exploration, open the model and adjust one of the noise generation settings. For example, the Rician noise generator has a K-factor of 10, which causes the mean value of the noise to be larger than that of the Rayleigh distributed noise. Double-click the Rician Noise MATLAB Function block to open the block mask and change the K-factor from 10 to 2. Rerun the model to see the noise spectrum shift.

Visualize Effects of Frequency-Selective Fading

FSK Modulation in Fading Channel

Pass an FSK signal through a Rayleigh multipath fading channel. Change the signal bandwidth to observe the impact of the fading channel on the FSK spectrum.

FSK Modulation in Flat Fading

Set modulation order to 4, the modulated symbol rate to 45 bps, and the frequency separation to 200 Hz.

```
M = 4;           % Modulation order
symbolRate = 45; % Symbol rate (bps)
freqSep = 200;   % Frequency separation (Hz)
```

Calculate the samples per symbol parameter, `sampPerSym`, as a function of the modulation order, frequency separation, and symbol rate. To avoid output signal aliasing, the product of `sampPerSym` and `symbolRate` must be greater than the product of `M` and `freqSep`. Calculate the sample rate of the FSK output signal.

```
sampPerSym = ceil(M*freqSep/symbolRate);
fsamp = sampPerSym*symbolRate;
```

Create an FSK modulator.

```
fskMod = comm.FSKModulator(M, ...
    'FrequencySeparation',freqSep, ...
    'SamplesPerSymbol',sampPerSym, ...
    'SymbolRate',symbolRate);
```

Set the path delays and average path gains for the fading channel.

```
pathDelays = [0 3 10]*1e-6; % Discrete delays of three-path channel (s)
avgPathGains = [0 -3 -6];   % Average path gains (dB)
```

By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 m difference in path length. The path delays and path gains specify the average delay profile of the channel.

Create a Rayleigh channel using the defined parameters. Set the `Visualization` property to display the impulse and frequency responses.

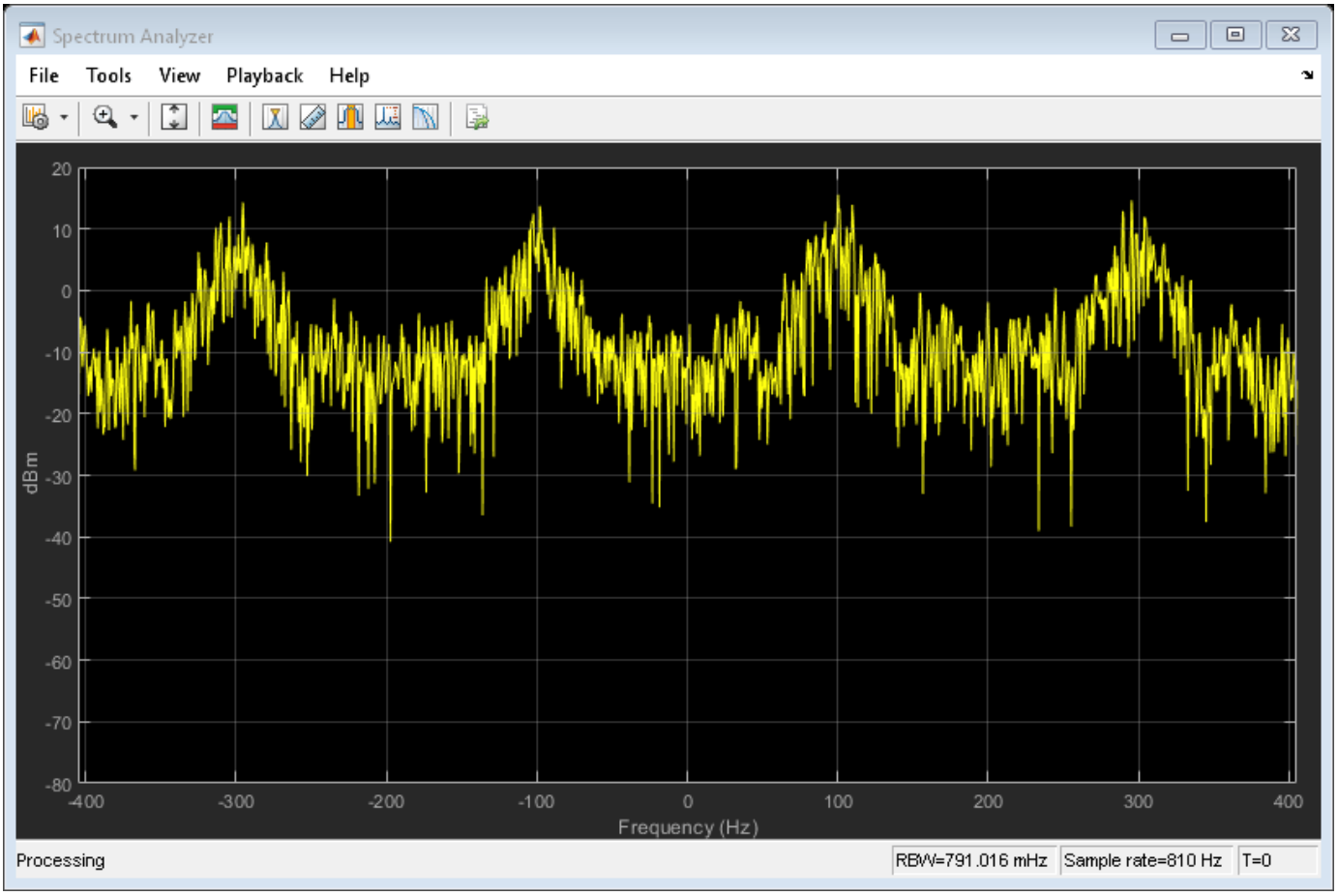
```
channel = comm.RayleighChannel(...
    'SampleRate',fsamp, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',0.01, ...
    'Visualization','Impulse and frequency responses', ...
    'SamplesToDisplay','10%');
```

Generate random data symbols and apply FSK modulation.

```
data = randi([0 3],2000,1);
modSig = fskMod(data);
```

Plot the spectrum of the FSK modulated signal.

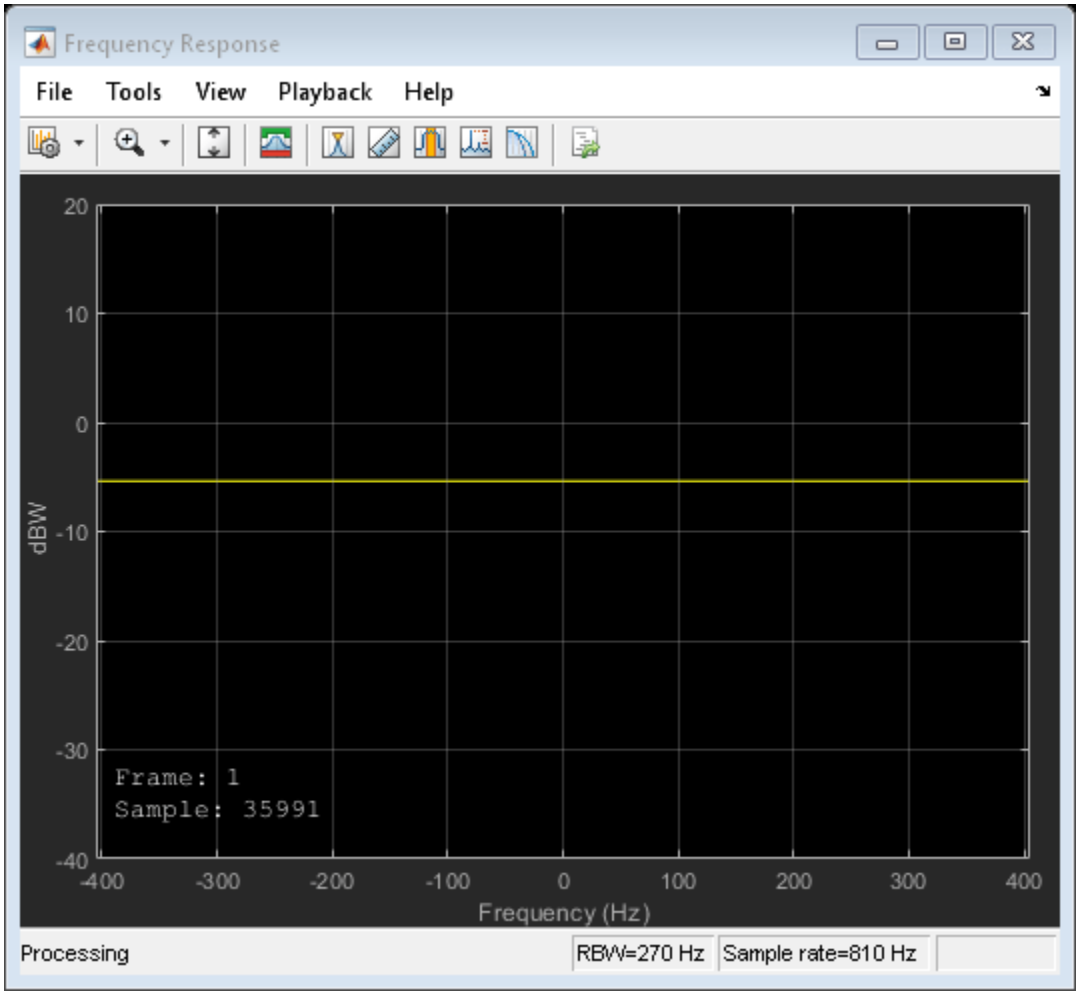
```
spectrum = dsp.SpectrumAnalyzer('SampleRate',fsamp);
spectrum(modSig)
```

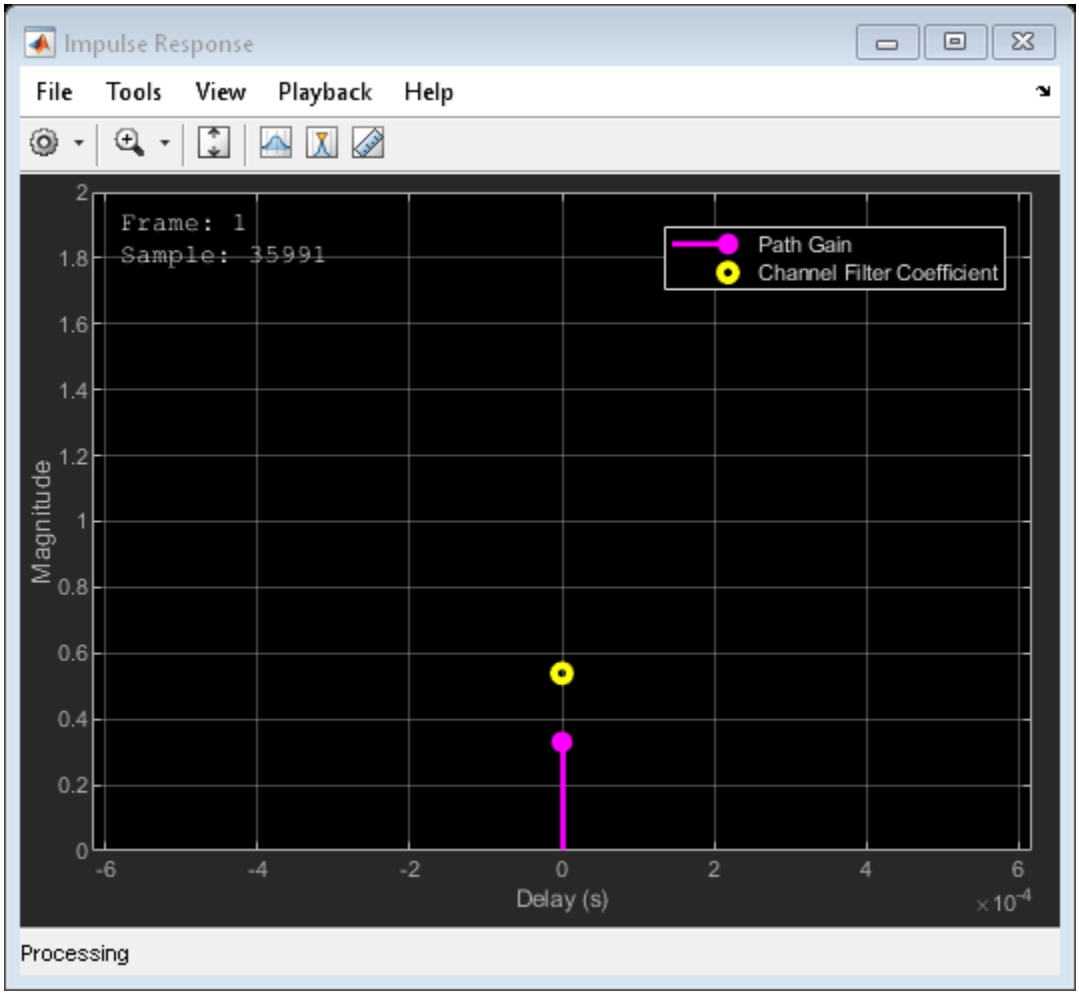


The modulated signal is composed of four tones each having approximately 20 dBm peak power separated by 200 Hz.

Pass the signal through the Rayleigh fading channel and apply AWGN having a 25 dB signal-to-noise ratio.

```
snrdB = 25;
rxSig = awgn(channel(modSig),snrdB);
```

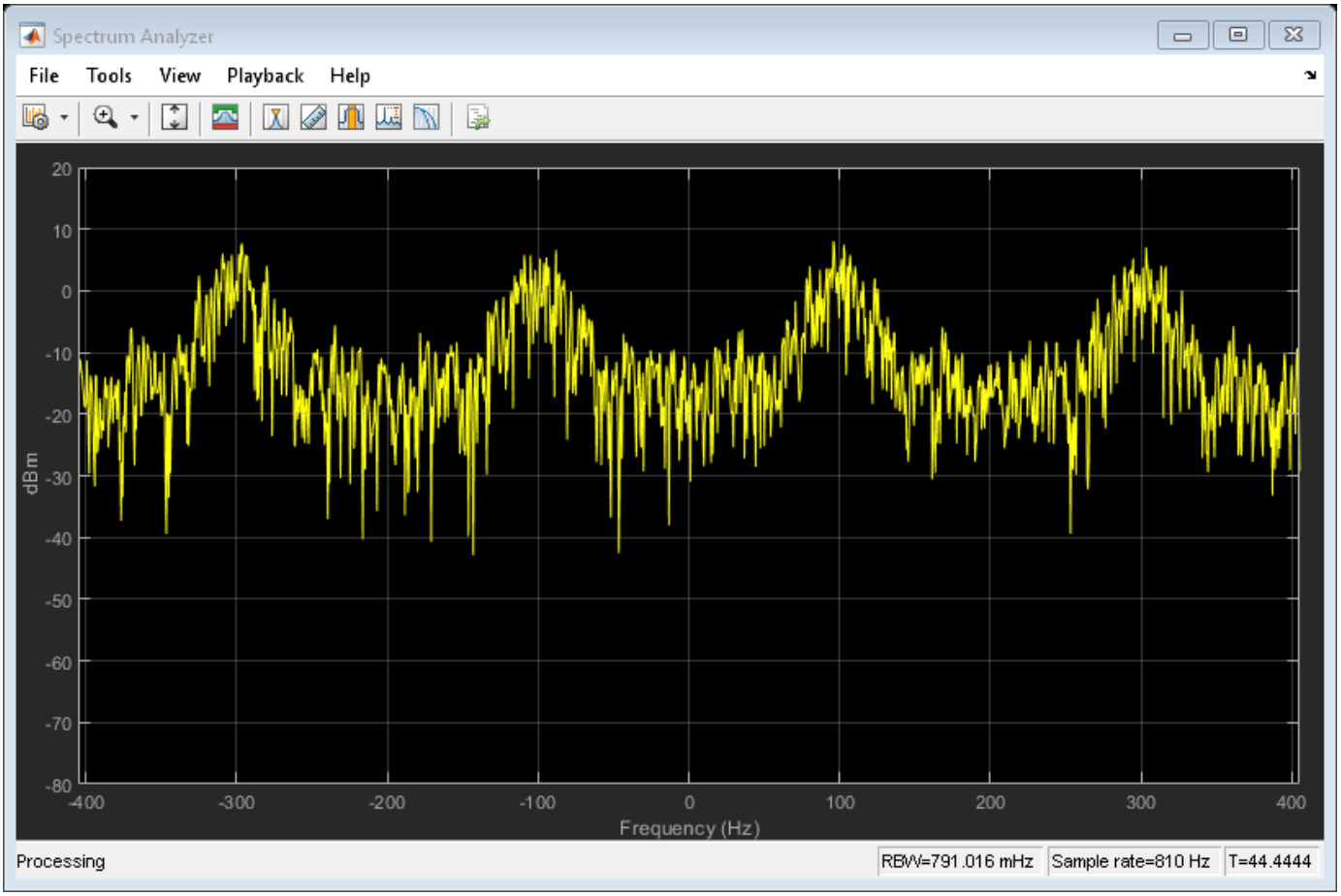

Frequency Response
File
Tools
View
Playback
Help
20
10
0
-10
-20
-30
-40
dBW
Frame: 1
Sample: 35991
-400
-300
-200
-100
0
100
200
300
400
Frequency (Hz)
Processing
RBW=270 Hz
Sample rate=810 Hz



The impulse and frequency responses show that the channel behaves as though it were flat. This is because the signal bandwidth, 800 Hz, is much smaller than the coherence bandwidth, 50 kHz.

Plot the received signal spectrum.

```
spectrum(rxSig)
```



The four tones comprising the FSK signal maintain the same frequency separation and peak power levels relative to each other. The absolute peak power levels have decreased due to the fading channel.

FSK Modulation in Frequency-Selective Fading

Increase the symbol rate to 45 kbps and the frequency separation to 200 kHz. Calculate the new samples per symbol and sample rate parameters.

```
symbolRate = 45e3;
freqSep = 200e3;
sampPerSym = ceil(M*freqSep/symbolRate);
fsamp = sampPerSym*symbolRate;
```

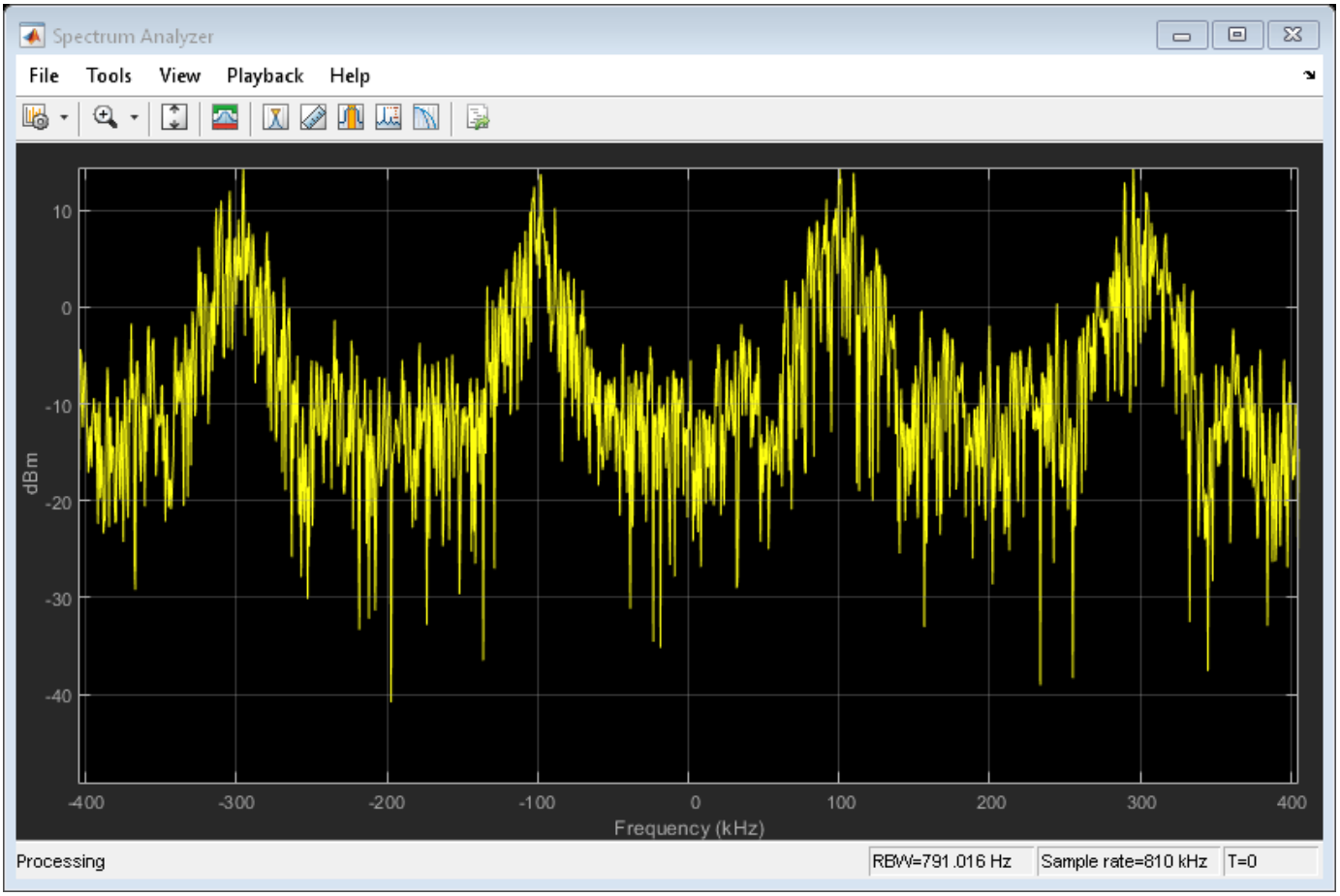
Update the FSK modulator properties.

```
release(fskMod)
fskMod.SymbolRate = symbolRate;
fskMod.FrequencySeparation = freqSep;
```

Update the spectrum analyzer sample rate property, `sa.SampleRate`. Apply FSK modulation and plot the resulting spectrum.

```
release(spectrum)
spectrum.SampleRate = sampPerSym*symbolRate;
```

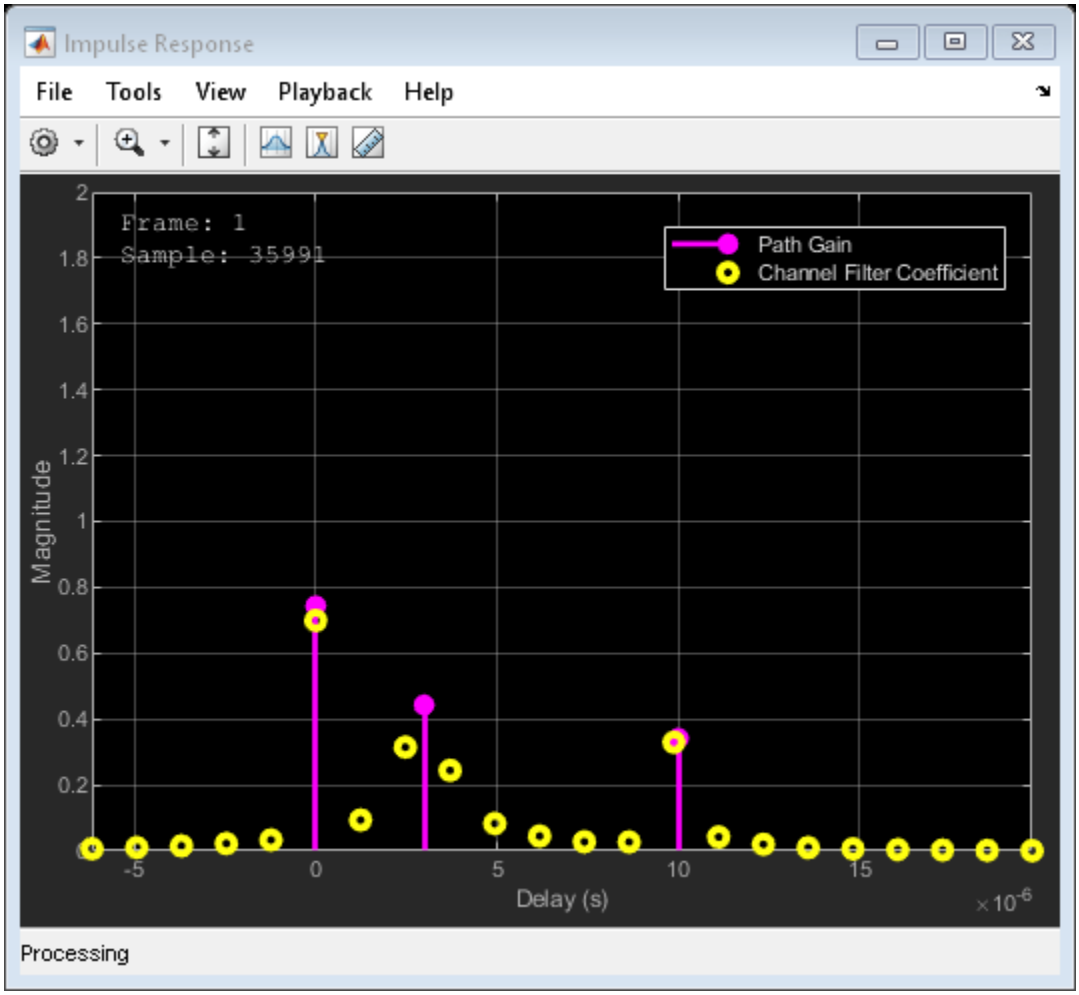
```
modSig = fskMod(data);
spectrum(modSig)
```

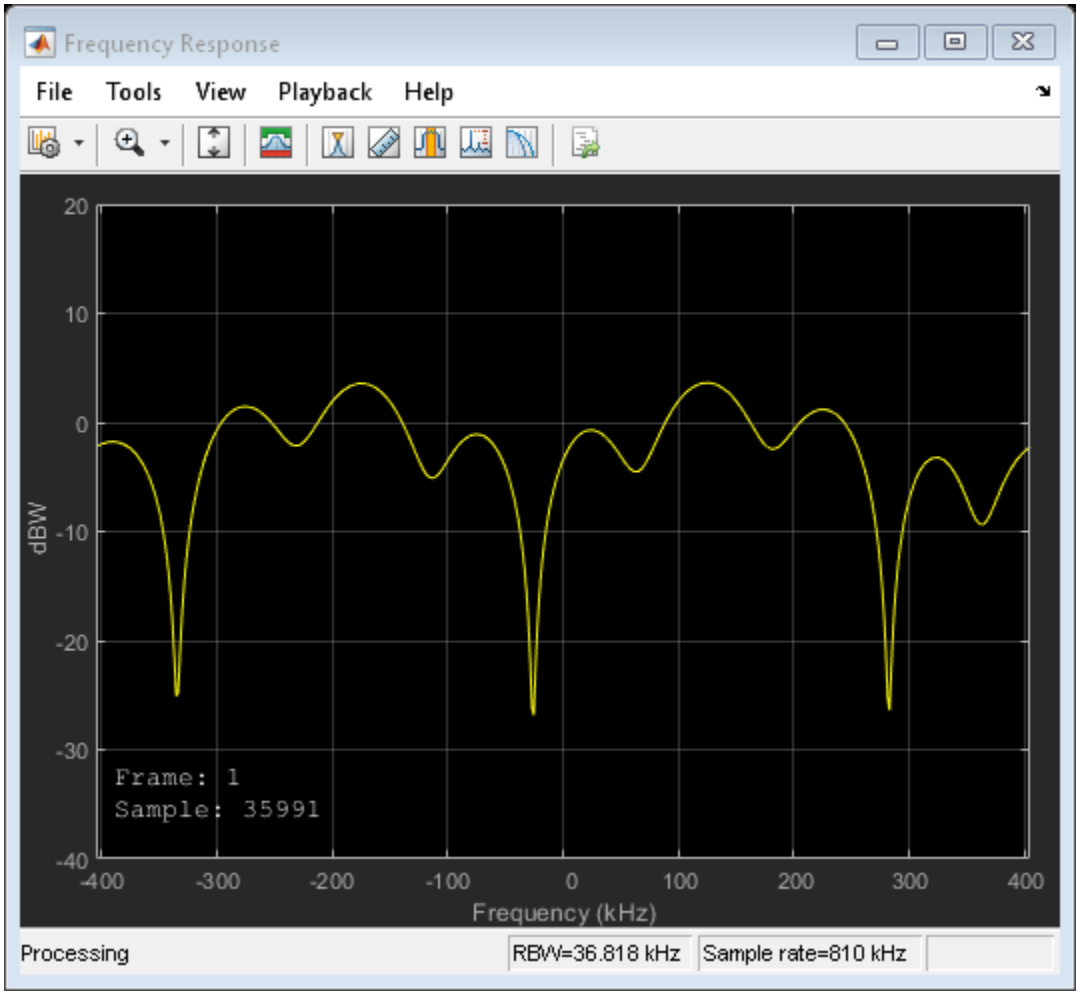


The spectrum has the same shape as in the flat-fading case but the four tones are now separated by 200 kHz.

Update the channel sample rate property. Pass the signal through the Rayleigh fading channel and apply AWGN.

```
release(channel)
channel.SampleRate = fsamp;

rxSig = awgn(channel(modSig),25);
```

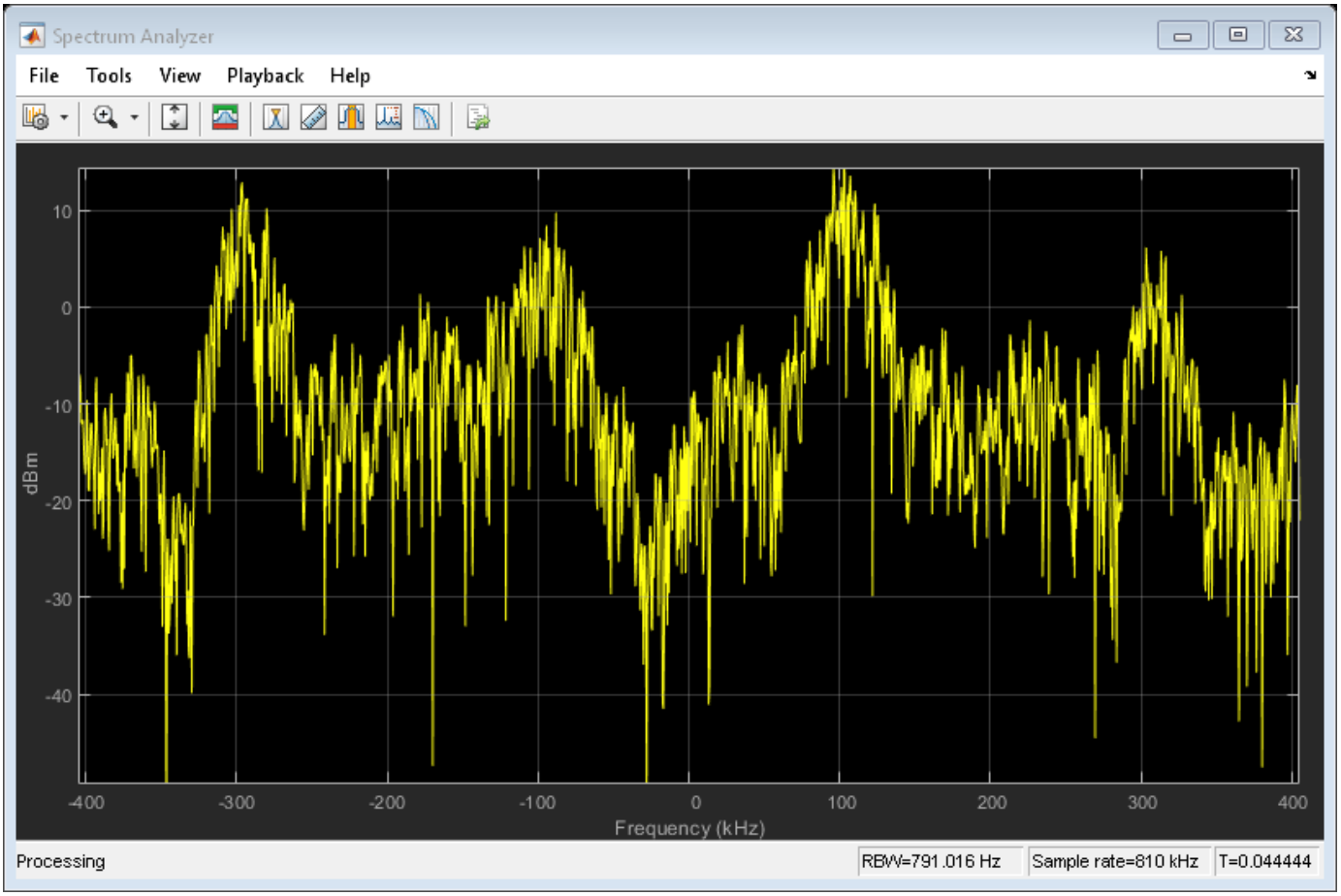

Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 35991
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-6
Processing



The impulse and frequency responses show that the multipath fading is frequency selective.

Plot the received signal spectrum.

```
spectrum(rxSig)
```



There are still four identifiable tones but their relative peak power levels differ due to the frequency-selective fading. The signal bandwidth, 800 kHz, is larger than the coherence bandwidth, 50 kHz.

QPSK Modulation in Fading Channel

Pass a QPSK signal through a Rayleigh multipath fading channel. Change the signal bandwidth to observe the impact of the fading channel on the QPSK constellation.

QPSK Modulation in Flat Fading

Set the symbol rate parameter to 500 bps.

```
symbolRate = 500;
```

Generate random data symbols and apply QPSK modulation.

```
data = randi([0 3],10000,1);
modSig = pskmod(data,4,pi/4,'gray');
```

Set the path delays and average path gains for the fading channel.

```
pathDelays = [0 3 10]*1e-6;     % Discrete delays of three-path channel (s)
avgPathGains = [0 -3 -6];       % Average path gains (dB)
```

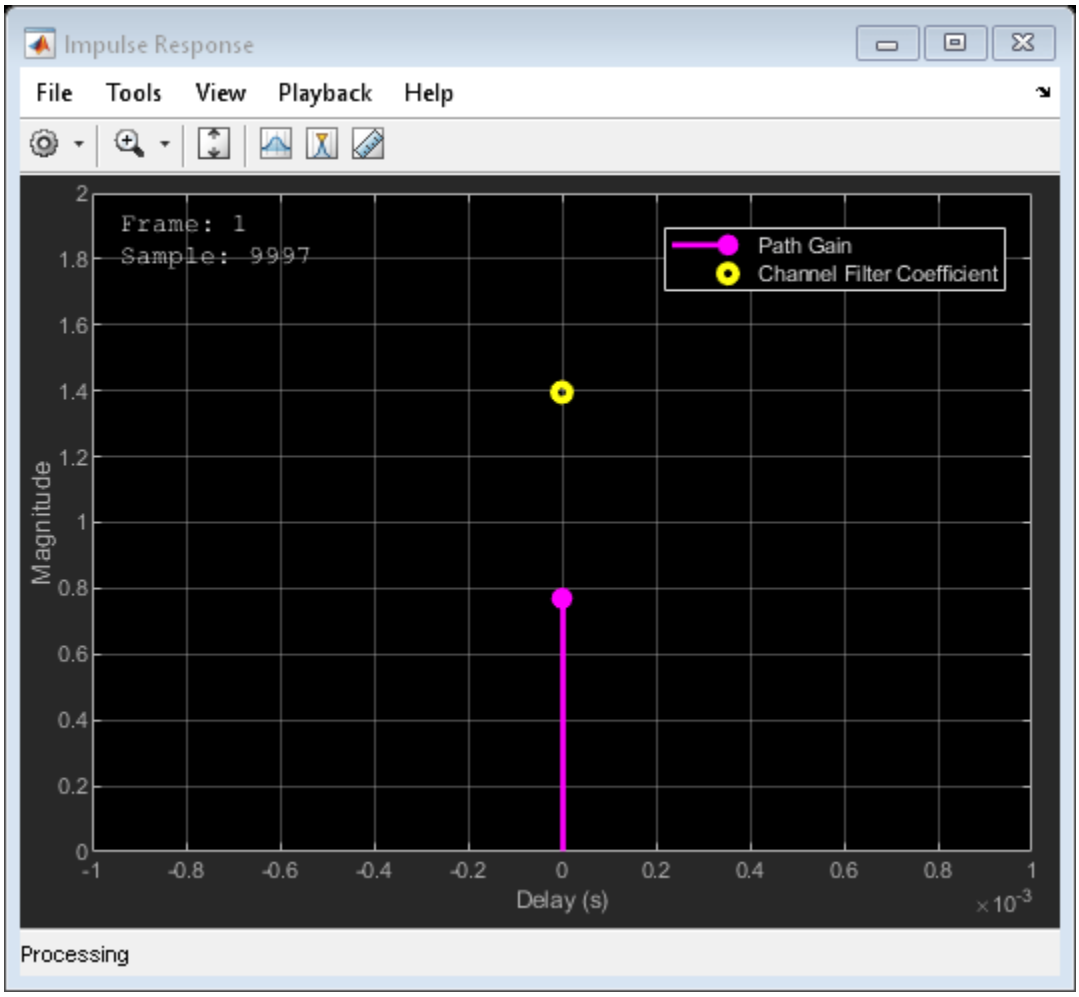
By convention, the delay of the first path is typically set to zero. For subsequent paths, a 1 microsecond delay corresponds to a 300 meter difference in path length. The path delays and path gains specify the average delay profile of the channel.

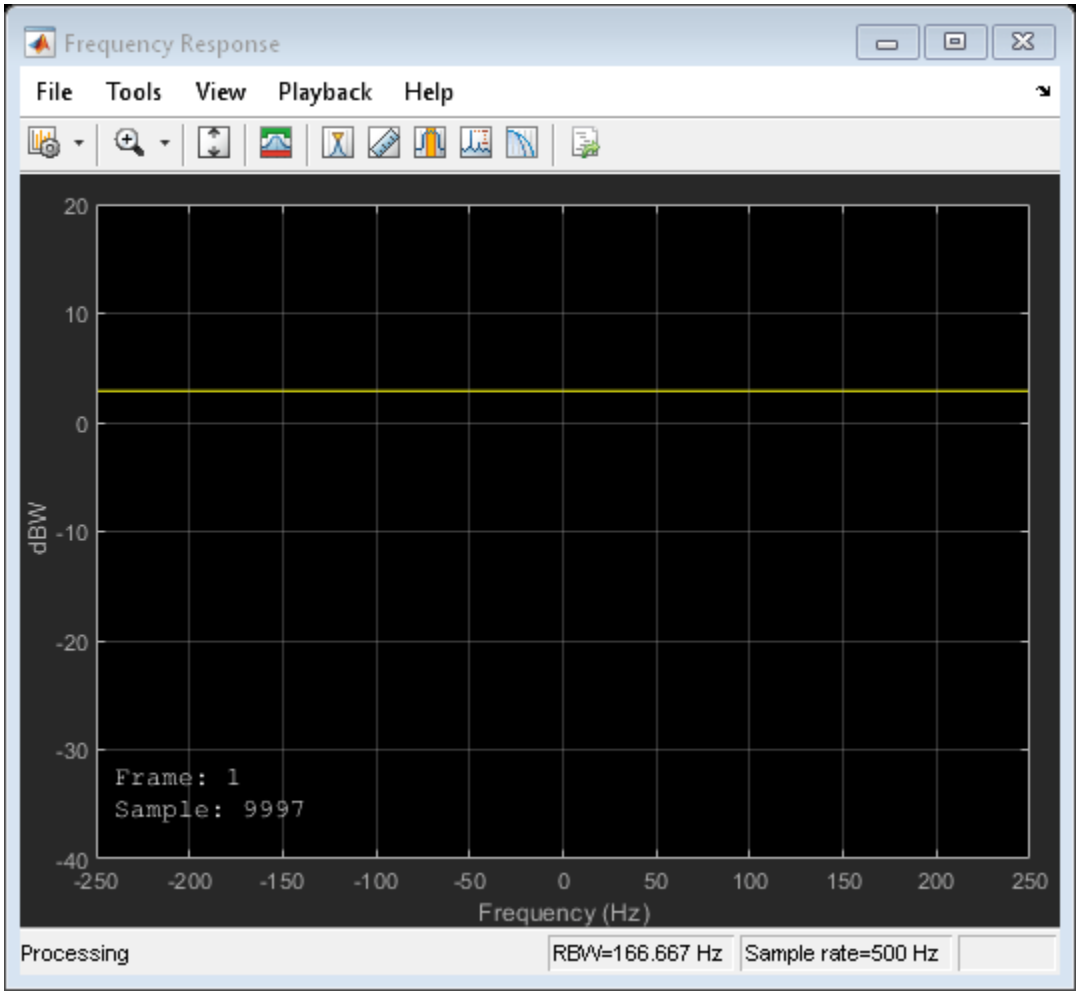
Create a Rayleigh channel using the defined parameters. Set the `Visualization` property to display the impulse and frequency responses.

```
fsamp = symbolRate;
channel = comm.RayleighChannel(...
    'SampleRate',fsamp, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',0.01, ...
    'Visualization','Impulse and frequency responses');
```

Pass the signal through the Rayleigh channel and apply AWGN.

```
rxSig = awgn(channel(modSig),25);
```

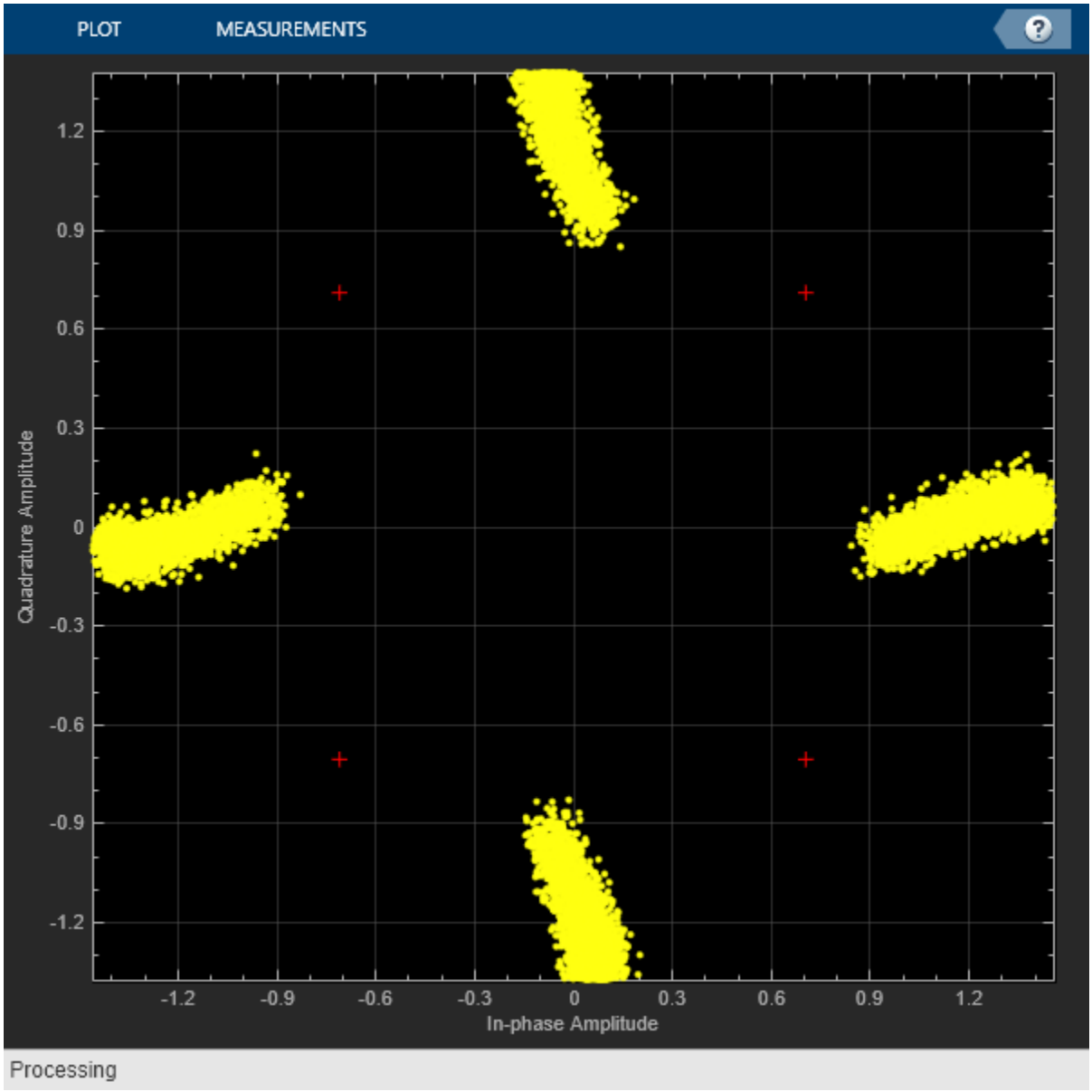




The impulse and frequency responses show that the channel behaves as though it were flat. This is because the signal bandwidth, 500 Hz, is much smaller than the coherence bandwidth, 50 kHz. Alternatively, the delay span of the channel (10 microseconds) is much smaller than the QPSK symbol period (2 milliseconds) so the resultant bandlimited impulse response is approximately flat.

Plot the constellation.

```
constDiagram = comm.ConstellationDiagram;
constDiagram(rxSig)
```

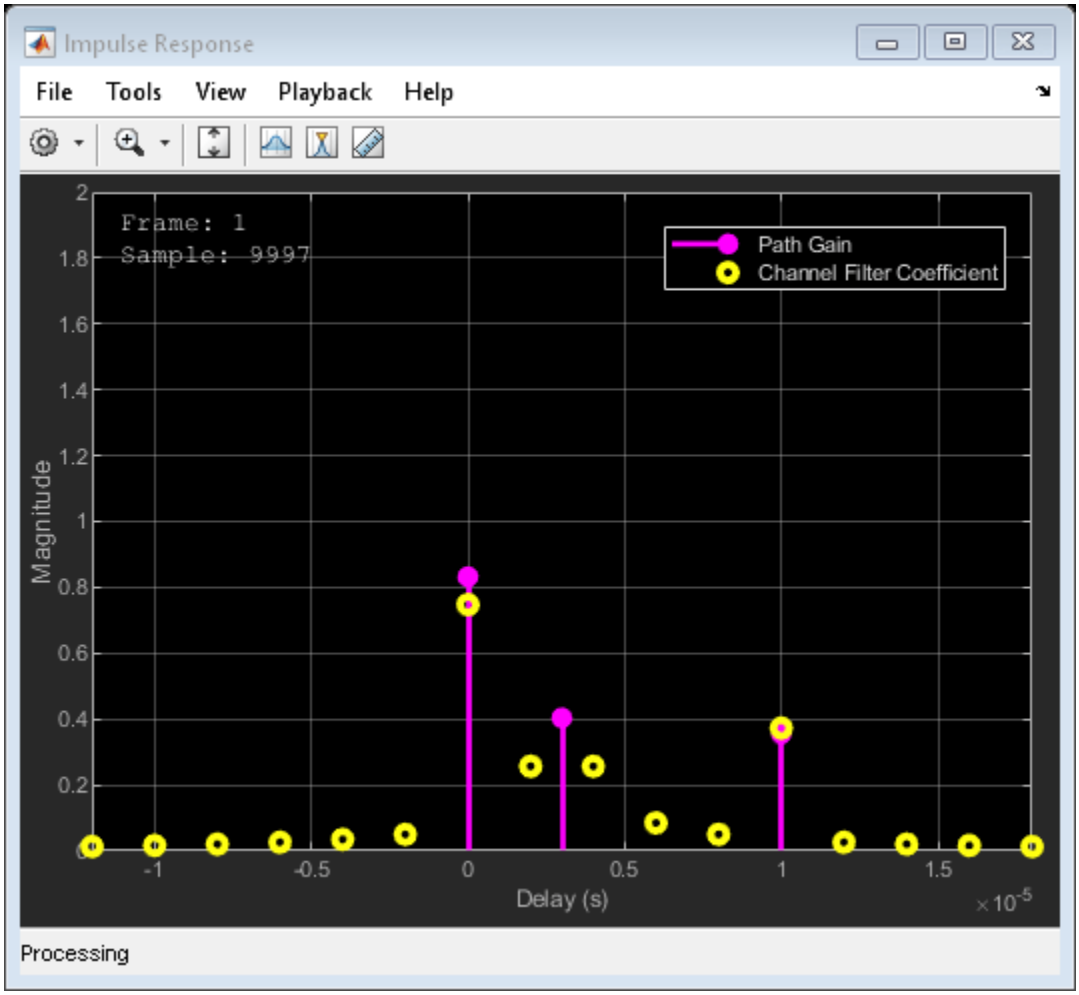


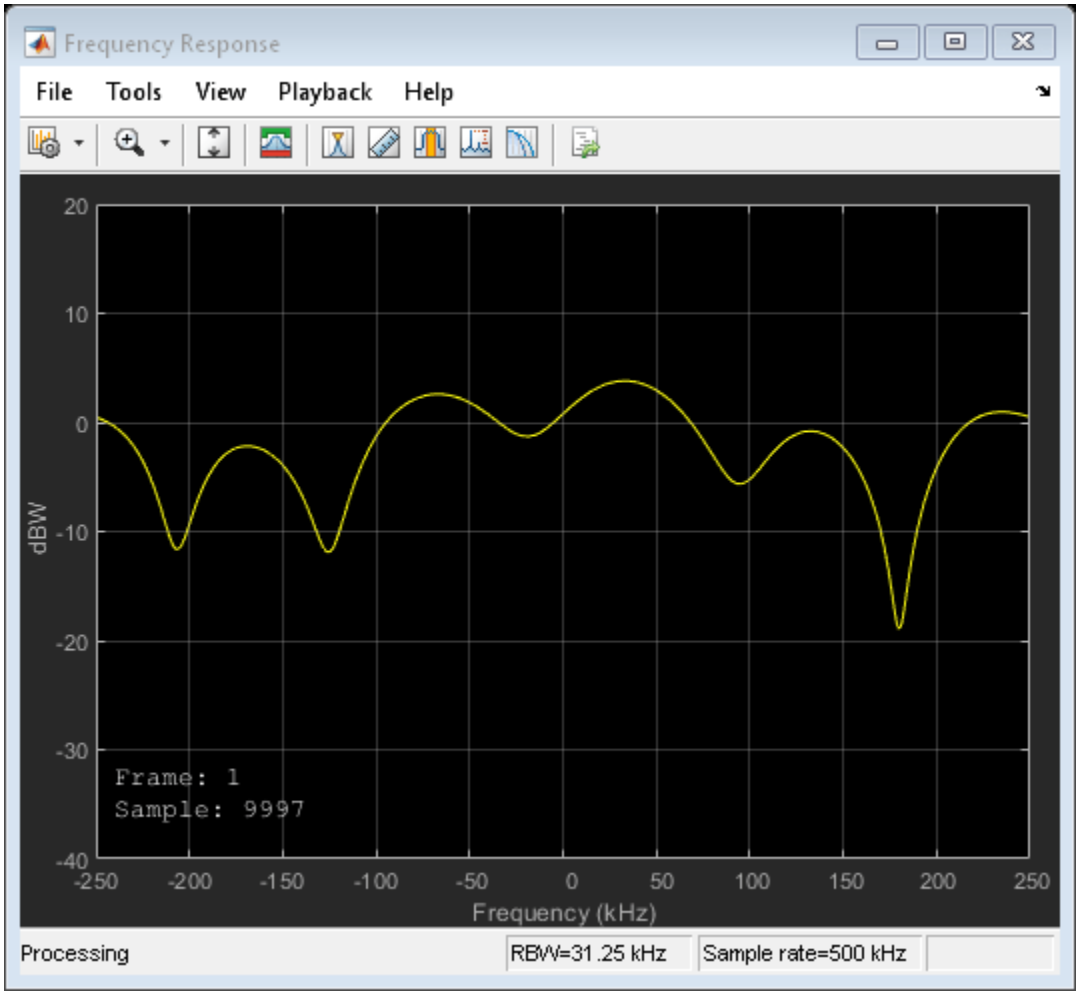
The QPSK constellation shows the effects of the fading channel; however, the signal still has four identifiable states.

QPSK Modulation in Frequency-Selective Fading

Increase the symbol rate to 500 kbps and update the related channel property. Pass the signal through the Rayleigh channel and apply AWGN.

```
symbolRate = 500e3;

release(channel)
channel.SampleRate = symbolRate;

rxSig = awgn(channel(modSig),25);
```

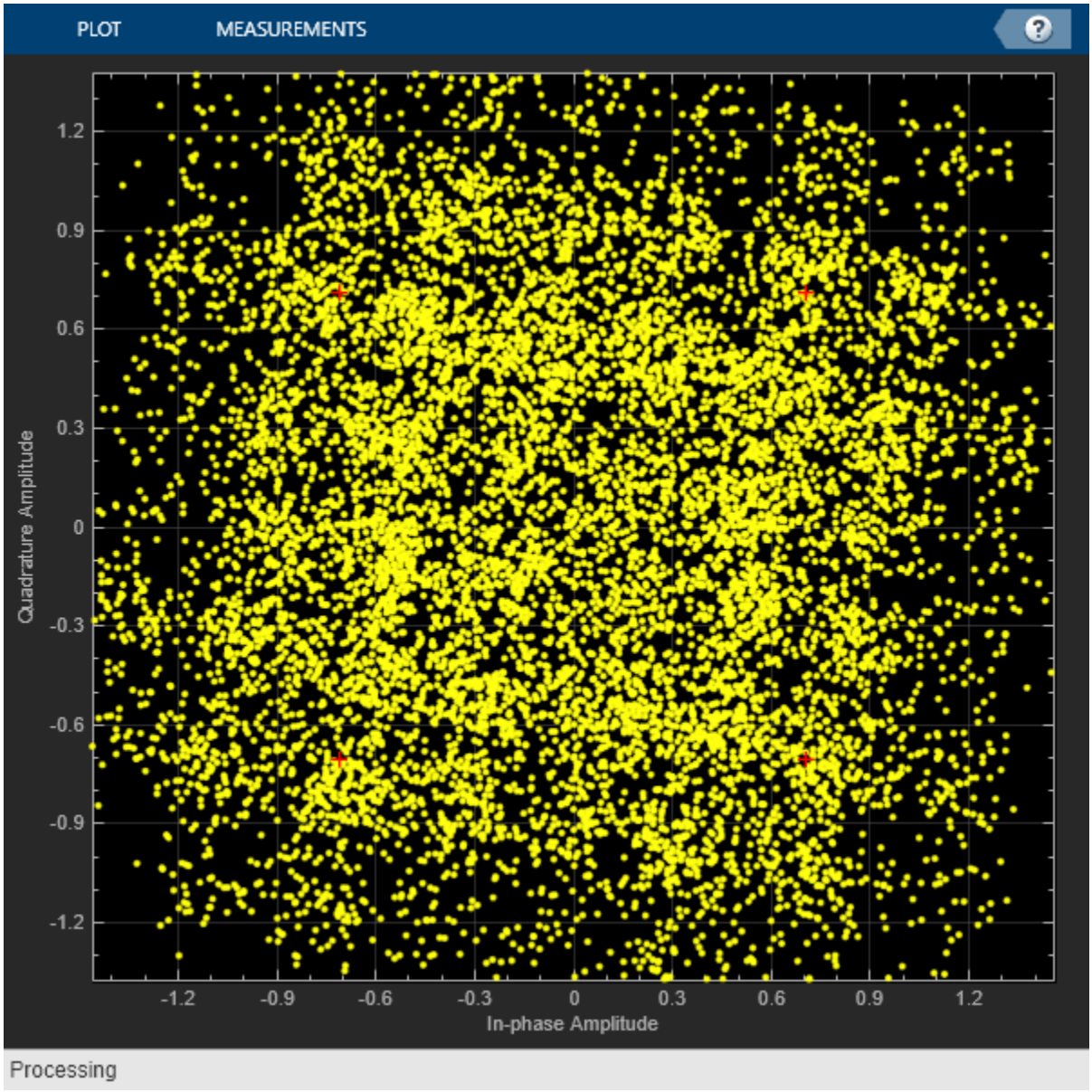

Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 9997
Path Gain
Channel Filter Coefficient
Magnitude
Delay (s)
×10^-5
Processing



The impulse and frequency responses show that the multipath fading is frequency selective.

Plot the constellation.

```
constDiagram(rxSig)
```



As the signal bandwidth is increased from 500 Hz to 500 kHz, the signal becomes highly distorted. This distortion is due to the intersymbol interference (ISI) that comes from time dispersion of the wideband signal. The delay span of the channel (10 microseconds) is now larger than the QPSK symbol period (2 microseconds) so the resultant bandlimited impulse response is no longer flat. Alternatively, the signal bandwidth is much larger than the coherence bandwidth, 50 kHz.

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.

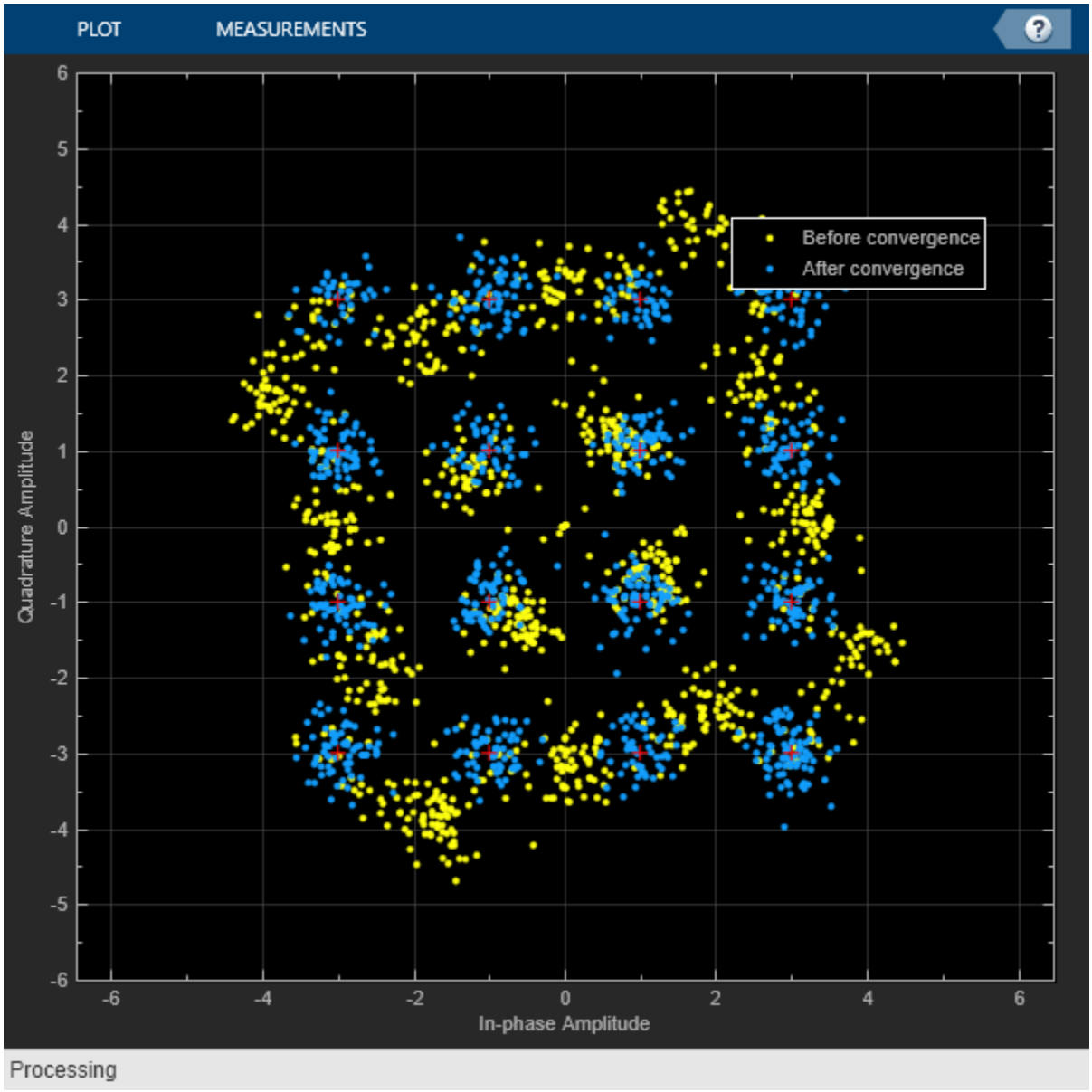
```
syncCoarse = coarse(rxSig);
```

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
```

Adjust Carrier Synchronizer Damping Factor to Correct Frequency Offset

Attempt to correct for a frequency offset using the carrier synchronizer object. Increase the damping factor of the synchronizer and determine if the offset was corrected.

Set the modulation order, sample rate, frequency offset, and signal-to-noise ratio parameters.

```
M = 8;
fs = 1e6;
foffset = 1000;
snrdb = 20;
```

Create a phase frequency offset object to introduce a frequency offset to a modulated signal. Create a constellation diagram object.

```
pfo = comm.PhaseFrequencyOffset('SampleRate',fs, ...
    'FrequencyOffset',foffset);
constDiagram = comm.ConstellationDiagram( ...
    'ReferenceConstellation',pskmod(0:M-1,M,pi/M));
```

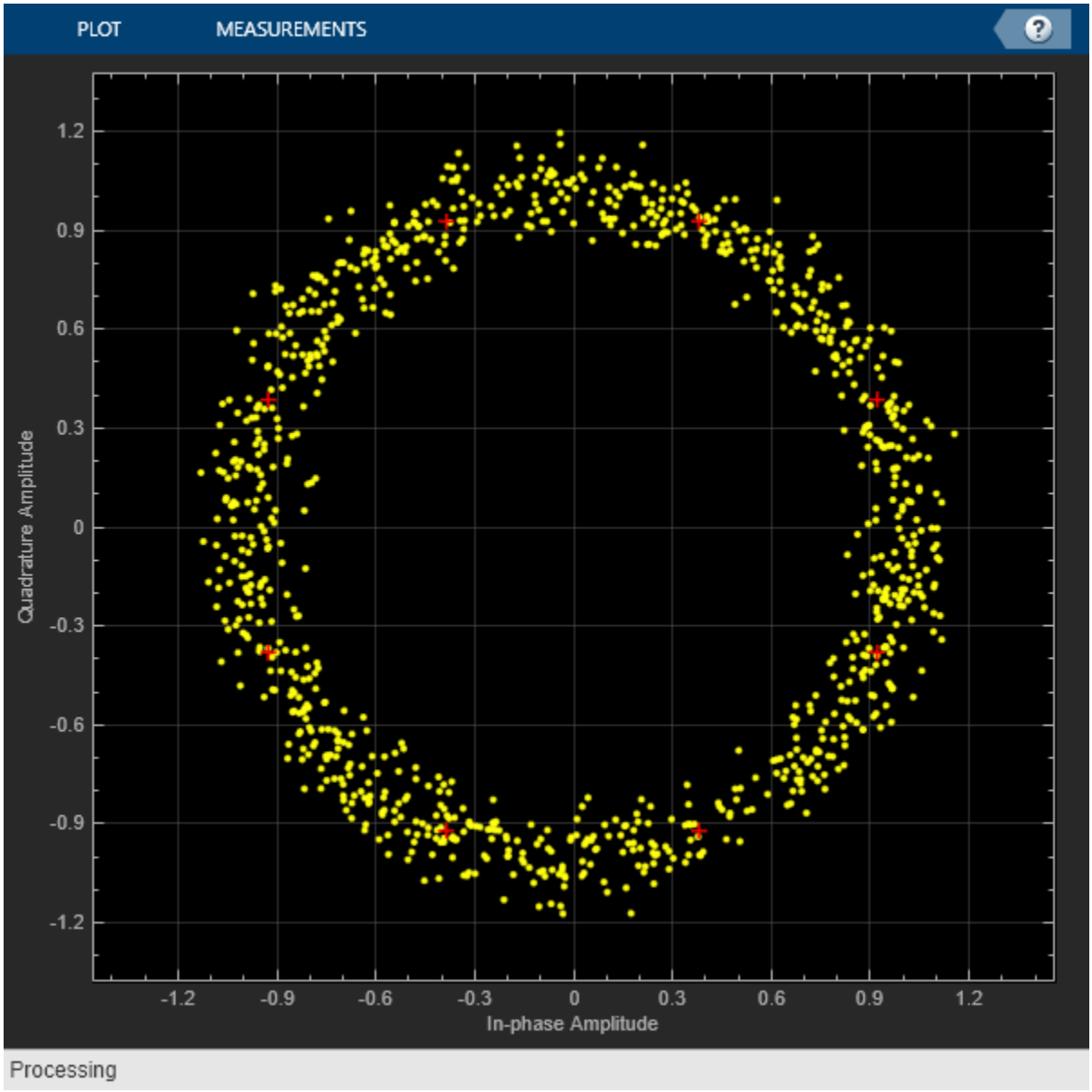
Create a carrier synchronizer object to correct for the frequency offset.

```
carriersync = comm.CarrierSynchronizer('Modulation','8PSK', ...
    'DampingFactor',0.05,'NormalizedLoopBandwidth',0.01);
```

The main processing loop includes these steps:

- Generate random data.
- Apply 8-PSK modulation.
- Introduce a frequency offset.
- Pass the signal through an AWGN channel.
- Correct for the frequency offset.
- Display the constellation diagram.

```
for k = 1:200
    data = randi([0 M-1],1000,1);
    modSig = pskmod(data,M);
    txSig = pfo(modSig);
    rxSig = awgn(txSig,snrdb);
    syncOut = carriersync(rxSig);
    constDiagram(syncOut)
end
```



The constellation points cannot be clearly identified indicating that the carrier synchronizer is unable to compensate for the frequency offset.

Determine the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay by using the `info` function.

```
syncInfo = info(carriersync)
```

```
syncInfo = struct with fields:
    NormalizedPullInRange: 0.0044
    MaxFrequencyLockDelay: 78.9568
        MaxPhaseLockDelay: 130
```

Convert the normalized pull-in range from radians to cycles. Compare the normalized frequency offset to the pull-in range.

```
[foffset/fs syncInfo.NormalizedPullInRange/(2*pi)]
```

```
ans = 1×2
10^-3 ×

    1.0000    0.7071
```

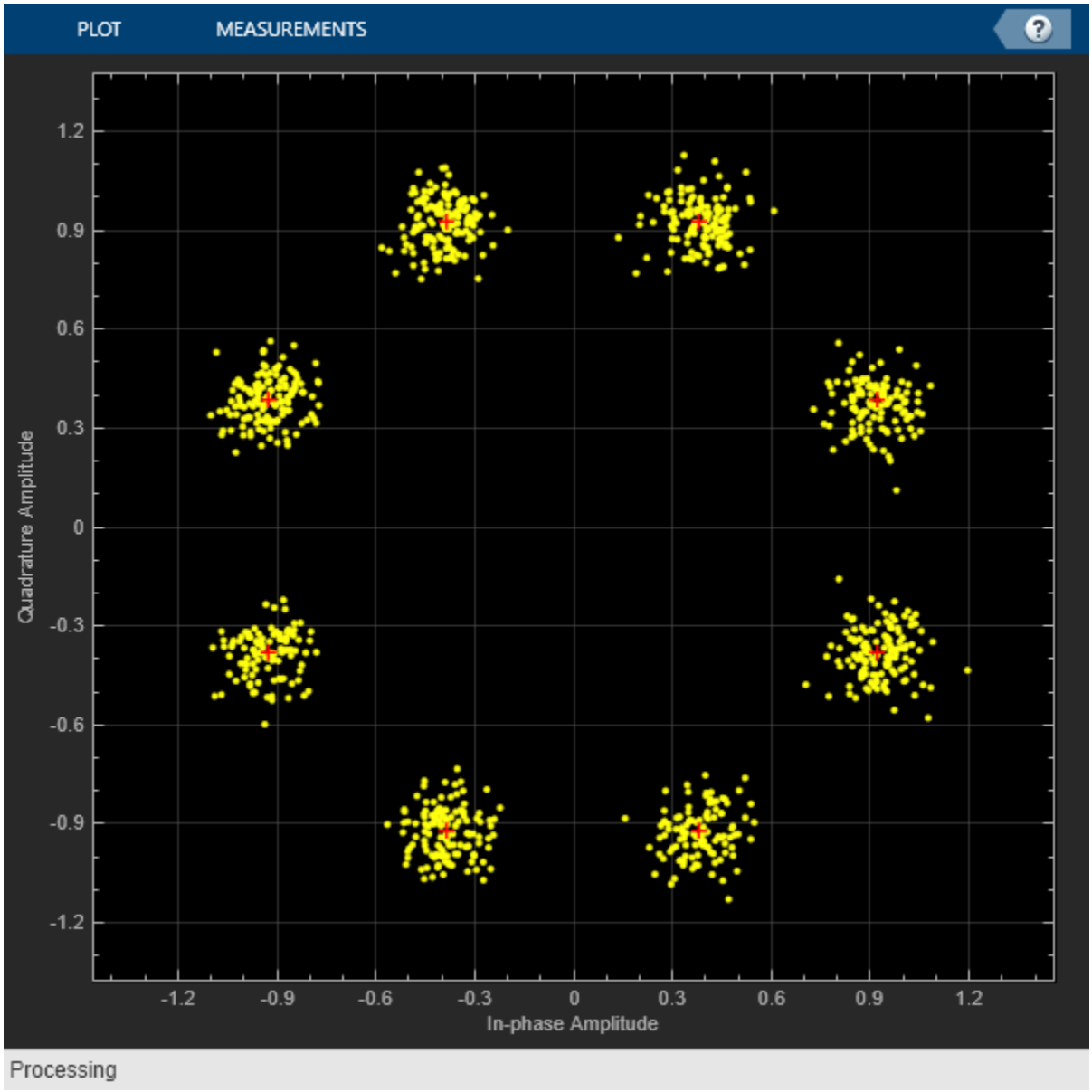
The offset is greater than the pull-in range. This is reason that the carrier synchronizer failed to correct the frequency offset.

Change the damping factor of the synchronizer to `0.707`.

```
carriersync.DampingFactor = 0.707;
```

Repeat the main processing loop.

```
for k = 1:200
    data = randi([0 M-1],1000,1);
    modSig = pskmod(data,M);
    txSig = pfo(modSig);
    rxSig = awgn(txSig,snrdb);
    syncOut = carriersync(rxSig);
    constDiagram(syncOut)
end
```



There are now eight observable clusters, which shows that the frequency offset was corrected.

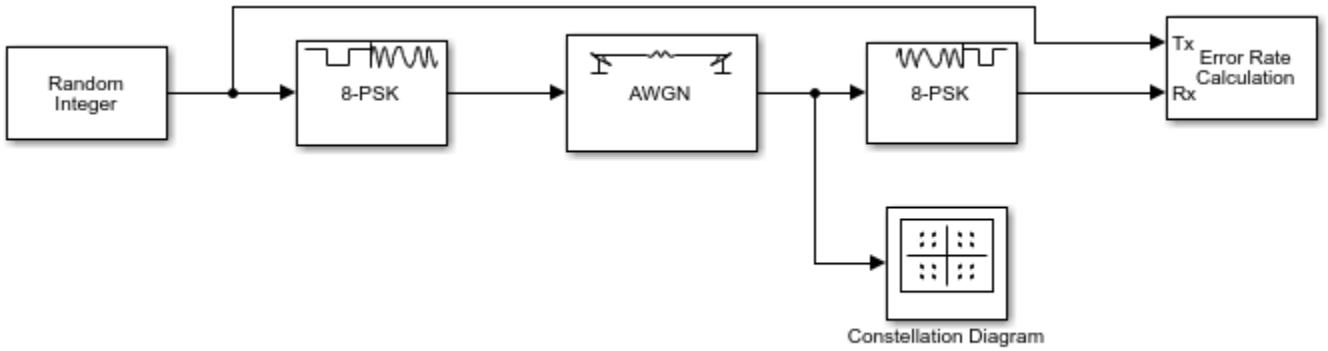
Determine the new pull-in range. The normalized offset is less than the pull-in range. This explains why the carrier synchronizer was able to correct the offset.

```
syncInfo = info(carriersync);
[foffset/fs syncInfo.NormalizedPullInRange/(2*pi)]
```

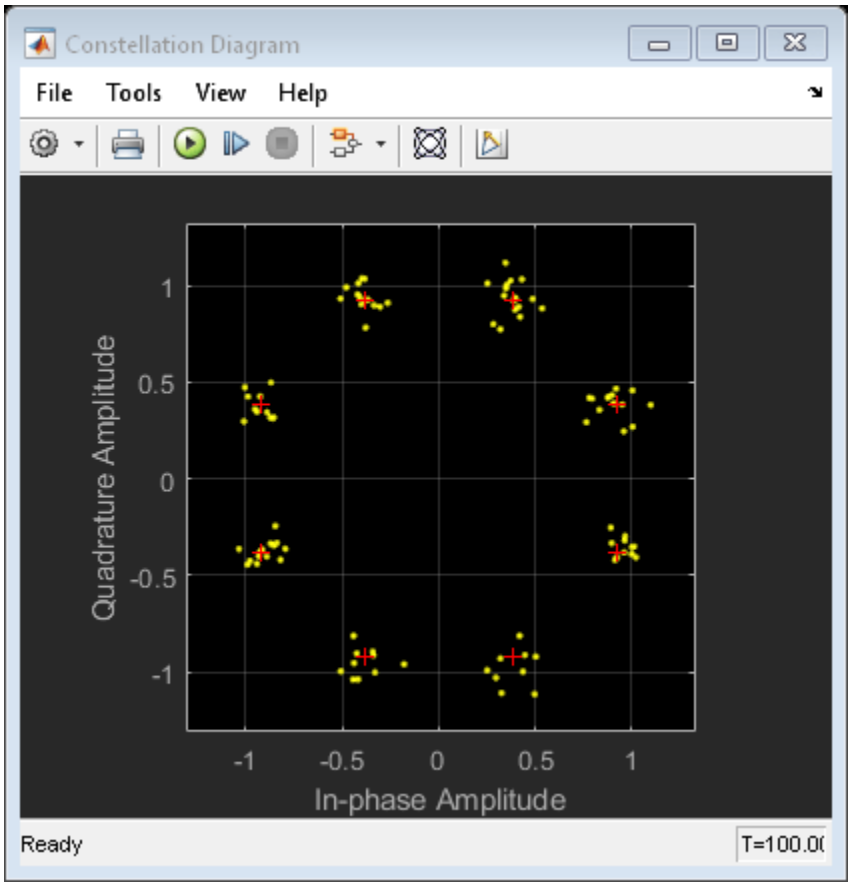
```
ans = 1×2

    0.0010    0.0100
```

Modulate and Demodulate 8-PSK Signal

Use the **Open model** button to open the `doc_8psk_model` model. The model generates an 8-PSK signal, applies white noise, displays the resulting constellation diagram, and computes the error statistics.



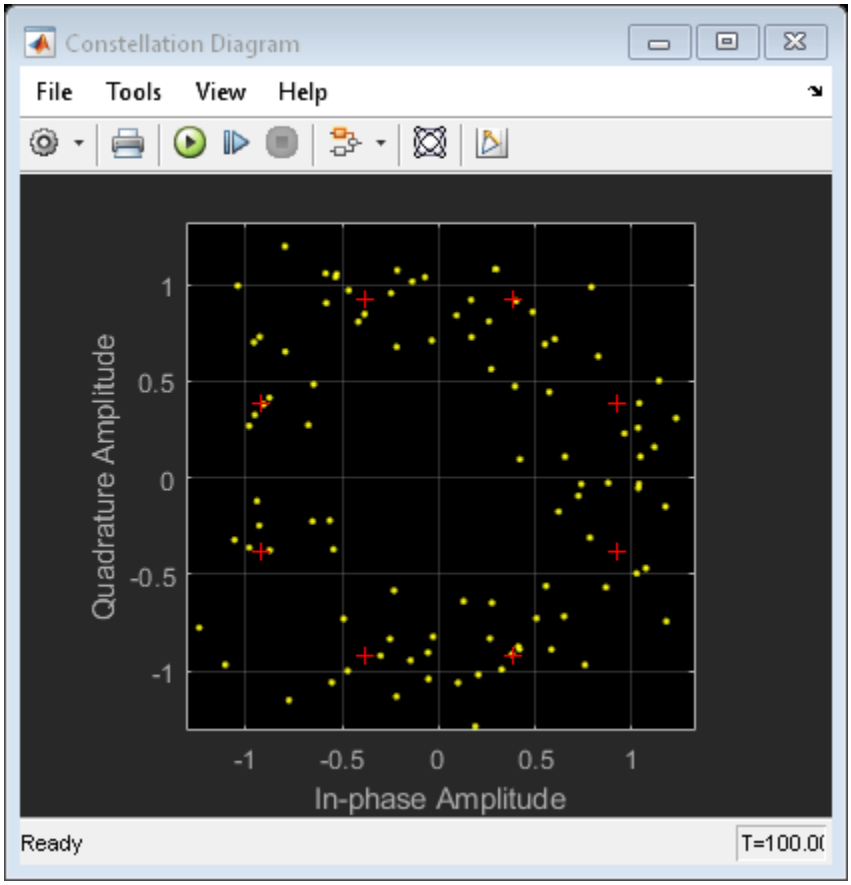
Run the model.



The error statistics are collected in vector `ErrorVec`. Because Eb/No is 15 dB, there are no measured symbol errors.

```
Number of symbol errors = 0
```

Change the Eb/No of the AWGN Channel block from 15 dB to 5 dB. The increase in the noise is shown in the constellation diagram.



Because of the increase in the noise level, the number of symbol errors is greater than zero.

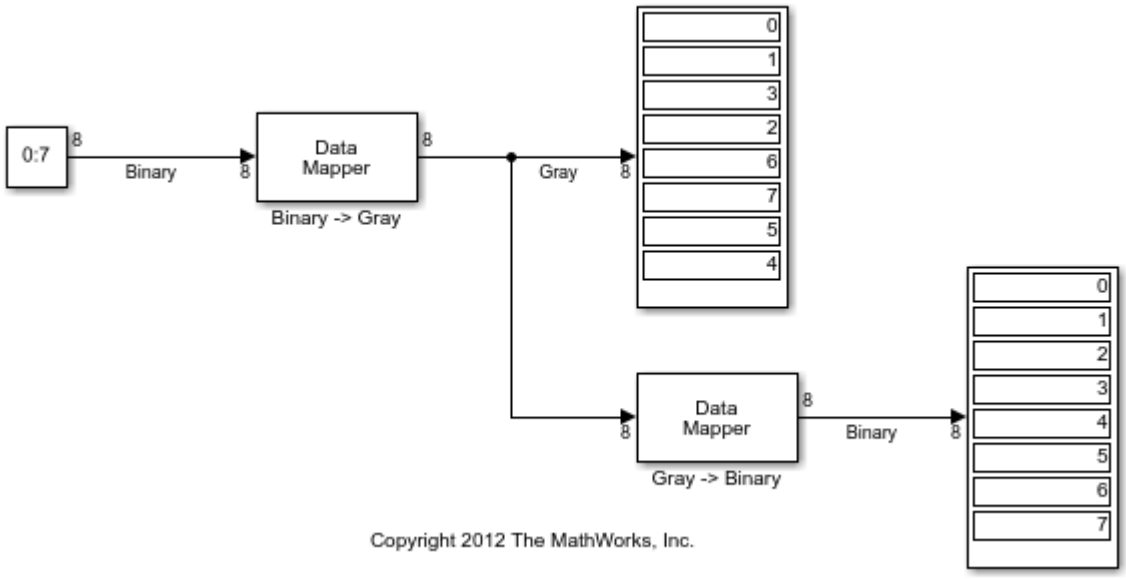
```
Number of symbol errors = 13
```

Binary to Gray Conversion in Simulink

Use the **Open model** button to open the Binary-to-Gray model. The model converts a binary sequence to a Gray-coded sequence and vice versa by using `Data Mapper` blocks.

Run the model.

The `Display` blocks show the natural binary and Gray-coded sequence ordering.



Bluetooth Tutorials

What Is Bluetooth?

Bluetooth wireless technology is the air interface intended to replace the cables connecting portable and fixed electronic equipment. Bluetooth device manufacturers have the flexibility to include optional core specification features to optimize and differentiate product offers.

Bluetooth is equated with the implementation specified by the Bluetooth Core Specification group of standards maintained by the Bluetooth Special Interest Group (SIG) industry consortium. Communications Toolbox Library for the Bluetooth Protocol functionality enables you to model Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR) communications system links, as specified in the Core System Package [Low Energy Controller volume], Specification Volume 6. It also enables you to explore variations on implementations for future evolution of the standard. Bluetooth BR/EDR and BLE devices operate in the same unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band as Wi-Fi®.

In Bluetooth BR/EDR, the radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth BR/EDR channel has a bandwidth of 1 MHz. Each frequency is located at (2402 + k) MHz, where k = 0,1,...78.

In BLE, the operating radio frequency is in the range 2.4000 GHz to 2.4835 GHz, inclusive. The channel bandwidth is 2 MHz and the operating band is divided into 40 channels, k = 0, ..., 39. The center frequency of the k^{th} channel is located at 2402 + k × 2 MHz.

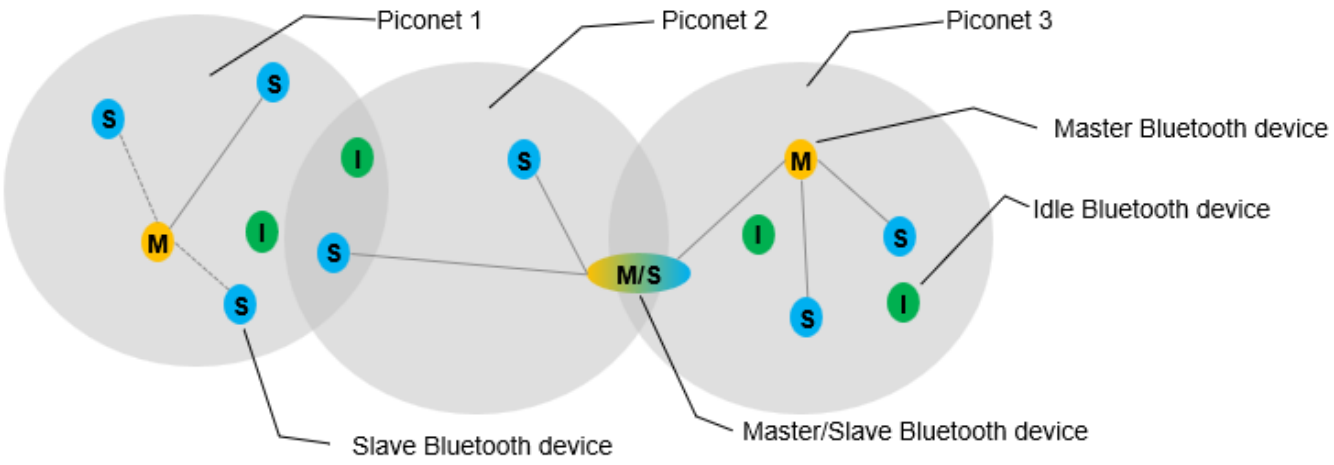
Network Architecture

Network topologies supported in Bluetooth include point-to-point, broadcast, and mesh connectivity. Point-to-point connectivity is available for devices that operate in Bluetooth BR/EDR or BLE. The broadcast and mesh connection topologies are only supported for BLE devices.

Point-to-Point and Point-to-Multipoint Connection Topology

Devices using point-to-point communication operate in a piconet. Point-to-point piconets define one-to-one device communication links. Examples of point-to-point links are links between PCs or mobile phones and peripherals such as headsets, printers, and fitness trackers. Multiple piconets connect to one another in a *scatternet* topology. Point-to-multipoint piconets define one to more than one device communication links.

This image shows a scatternet of three piconets. Each piconet shows one device in the role of *master* (M), with other devices in the *slave* (S) or *idle* (I) roles. The image also shows one device (M/S) assigned the master role in one piconet and slave in another piconet.



- A scatternet is an ad hoc network consisting of two or more piconets.
- A *piconet* is defined as a connection between two or more Bluetooth devices. Piconets nets are limited to a maximum of eight devices, with one master taking the master role at any given time and seven slaves.
- The individual Bluetooth devices assume the role of master, slave, or idle peer devices in a given piconet. An individual Bluetooth device can take the role of a slave in one piconet while taking the role of master in another piconet.
    - The master device provides the synchronization reference.
    - The slaves are other devices that synchronize to the clock and frequency hopping pattern of the master.
    - Other idle devices may be located in a piconet but are not active.

Broadcast Connection Topology

Broadcast piconets establish one-to-many communication links for BLE devices. Examples of broadcast links are retail point-of-interest information, indoor navigation, and asset tracking.

Mesh Connection Topology

Mesh networks establish the option of many-to-many communication links for BLE devices. Mesh topology enables the creation of large-scale device networks. Mesh is ideally suited for control, monitoring, and automation systems that require reliable and secure communication between thousands of devices.

The Bluetooth SIG specifies mesh networking requirements to enable an interoperable many-to-many (m:m) mesh networking solution for Bluetooth Low Energy (LE) wireless technology. Mesh networks are ideally suited for large-scale device networks supporting building automation, sensor networks, asset tracking, and other solutions requiring reliable and secure communication between multiple devices. Bluetooth SIG adopted these specifications.

- Mesh Profile Specification – Defines fundamental requirements to enable an interoperable mesh networking solution for Bluetooth LE wireless technology.
- Mesh Model Specification – Introduces models, used to define basic functionality of nodes on a mesh network.

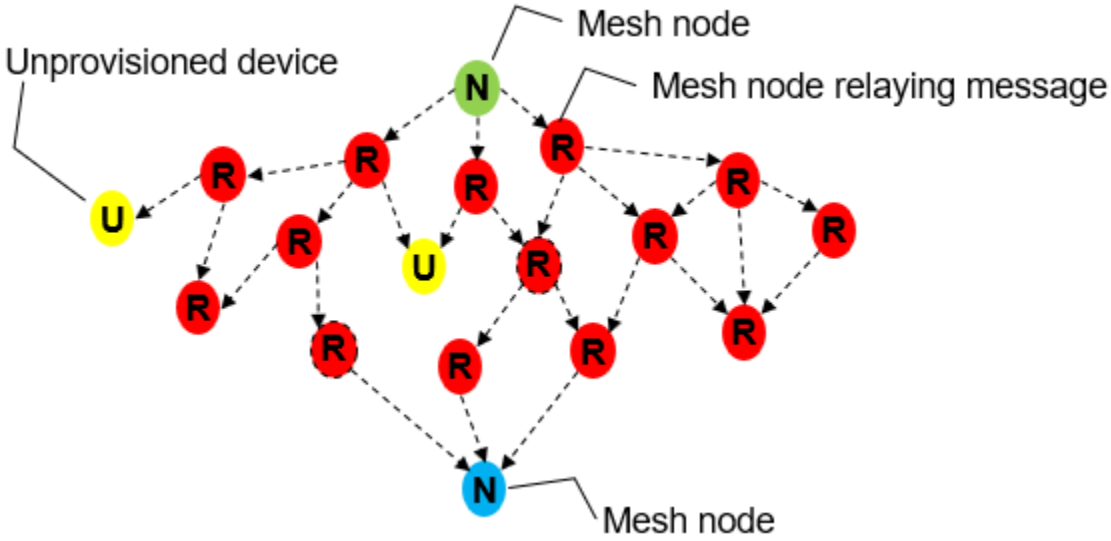
- Mesh Device Properties – Defines device properties required for the Mesh Model specification.

The Bluetooth Mesh Model Specification defines categories of BLE mesh models. Model categories include

- Foundation models
- Generic models
- Sensors
- Time and scenes
- Lighting

All devices must implement the foundation models for configuration server and health server. All other models in the model categories are optional and implemented based on the service the BLE device performs.

As described in the Mesh Profile Bluetooth specification, mesh networks operate as managed-flood-based networks. Devices use broadcast channels to transmit messages to other devices, and the messages are relayed forward to other devices extending the range of the original message.



A device that is not a member of a mesh network is referred to as an *unprovisioned device*. A device that is a member of a mesh network is known as a *node*. Devices are added to a mesh network by a *Provisioner*. Nodes in a mesh network share network keys that enable them to receive and to relay messages from other nodes in their network or subnet. Network keys are used to secure and authenticate messages at the network layer. Unprovisioned devices cannot receive messages because they do not have the network key to recover the message.

Bluetooth Products

The Bluetooth SIG defines a Bluetooth product as any product containing an implementation of Bluetooth wireless technology. Bluetooth products are classified as:

- Bluetooth End Product

- Bluetooth Host Subsystem Product
- Bluetooth Controller Subsystem Product
- Bluetooth Profile Subsystem Product
- Bluetooth Component Product
- Bluetooth Development Tool
- Bluetooth Test Equipment

The Communications Toolbox Library for the Bluetooth Protocol provides features enabling you to model Bluetooth Host and Controller Subsystem Products fully compliant Bluetooth links with the low energy (LE) core configuration.

Bluetooth Low Energy Core Configuration

The Bluetooth Core Specification, Volume 0, Part B, Section 4.4 specifies a set of required features that must be implemented to model fully compliant Bluetooth links with the low energy (LE) core configuration.

The LE core configuration defines three main layers - Application, Host, and Controller. The Communications Toolbox Library for the Bluetooth Protocol provides features to model the host and controller layers. Requirements defined in the Bluetooth Core Specification for the host and controller include

Layer	Sublayer	Bluetooth Specification Volume	Required Features
Host	Logical link control and adaptation protocol (L2CAP)	Volume 3, part A	If the GAP Peripheral or Central role is supported, L2CAP LE Signaling Channel (CID 0x0005) and all mandatory features associated with it.
	Generic access profile (GAP)	Volume 3, part C	All mandatory features for at least one of the LE GAP roles (Broadcaster, Observer, Peripheral, or Central) in sections 9–12 and section 15.
	Attribute profile (ATT)	Volume 3, part F	If the GAP Peripheral or Central role is supported, all mandatory features.
	Generic attribute profile (GATT)	Volume 3, part G	GATT is mandatory when ATT is supported. When supported, all mandatory features.

Layer	Sublayer	Bluetooth Specification Volume	Required Features
	Security manager (SM)	Volume 3, part H	If the GAP Peripheral or Central role is supported, all mandatory features.
Controller	Physical (PHY)	Volume 6, part A	All mandatory features.
	Link layer (LL)	Volume 6, part B	All mandatory features.

For a description of the mapping between Bluetooth protocol stack functionality and the OSI reference model, see “Bluetooth Protocol Stack” on page 13-7.

References

[1] https://www.bluetooth.com/

[2] "Bluetooth Core Specification." *Bluetooth Special Interest Group (SIG)*.

[3] "Supplement to the Bluetooth Core Specification, CSS Version 7." *Bluetooth Special Interest Group (SIG)*.

[4] "Bluetooth Core Specification Addendum 6." *Bluetooth Special Interest Group (SIG)*.

[5] "Mesh Profile Bluetooth Specification." *Bluetooth Special Interest Group (SIG)*.

[6] "Mesh Model Bluetooth Specification." *Bluetooth Special Interest Group (SIG)*.

[7] "Mesh Device Properties Bluetooth Specification." *Bluetooth Special Interest Group (SIG)*.

See Also

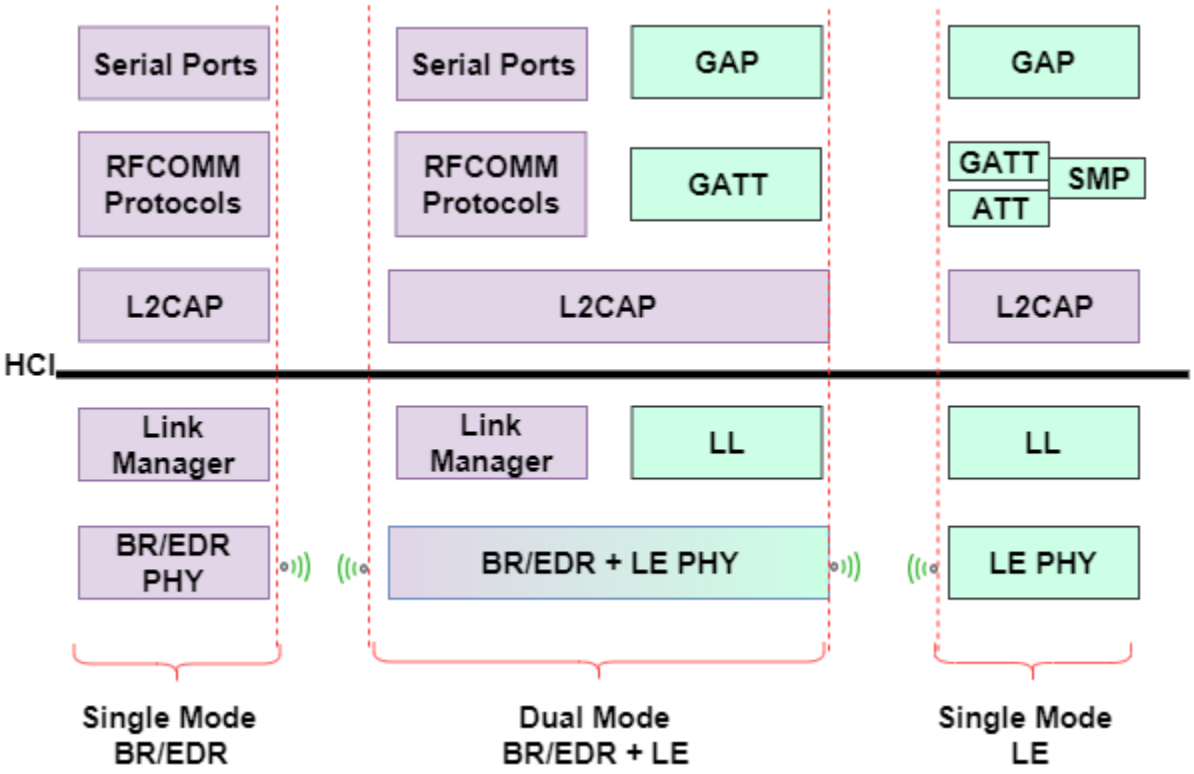
More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Location and Direction Finding” on page 13-37
- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Bluetooth Protocol Stack

The Bluetooth Special Interest Group (SIG) [1] and [2] defines the protocol stack for Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR) technology. The fundamental objectives of these specifications is to develop interactive services and applications over interoperable radio components and data communication protocols.

This figure shows the architecture of the Bluetooth stack.



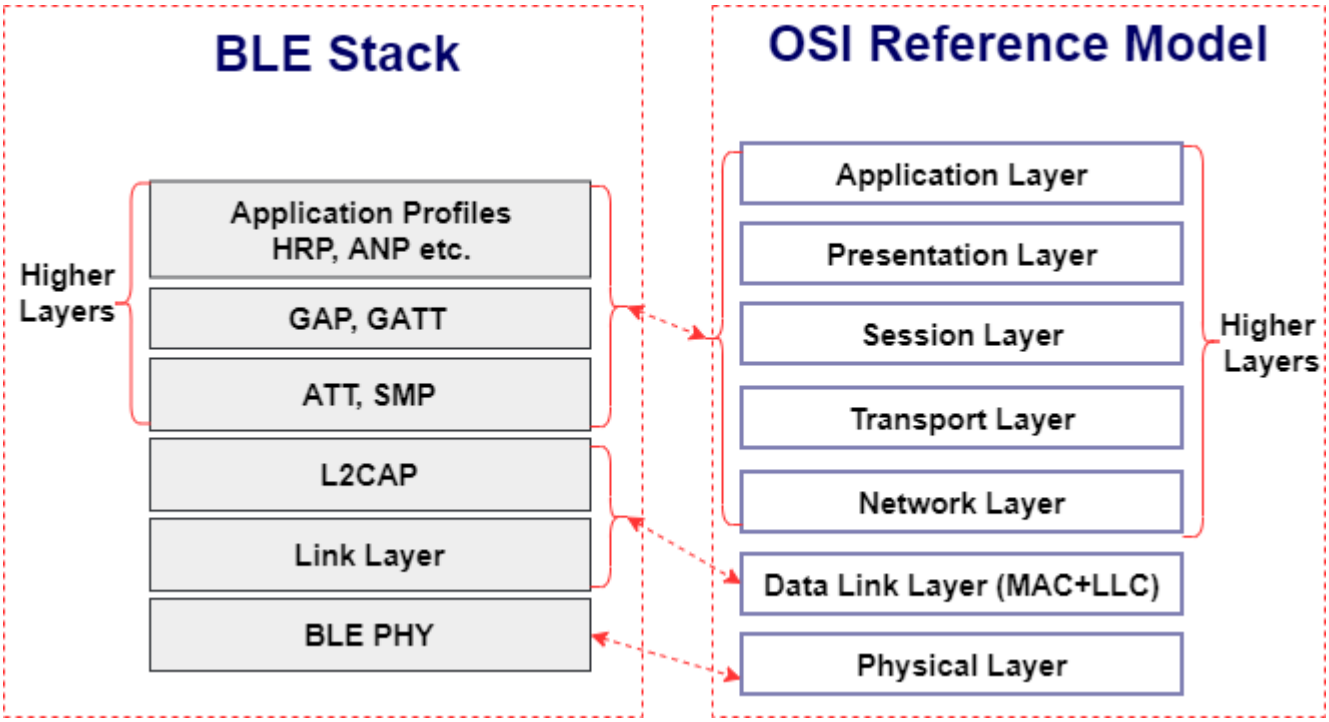
Bluetooth devices can be one of these two types:

- Single mode – Supports a BR/EDR or LE profile
- Dual mode – Supports BR/EDR and LE profiles

The subsequent sections provide details about the architecture of “BLE Protocol Stack” on page 13-7 and “Bluetooth BR/EDR Protocol Stack” on page 13-12.

BLE Protocol Stack

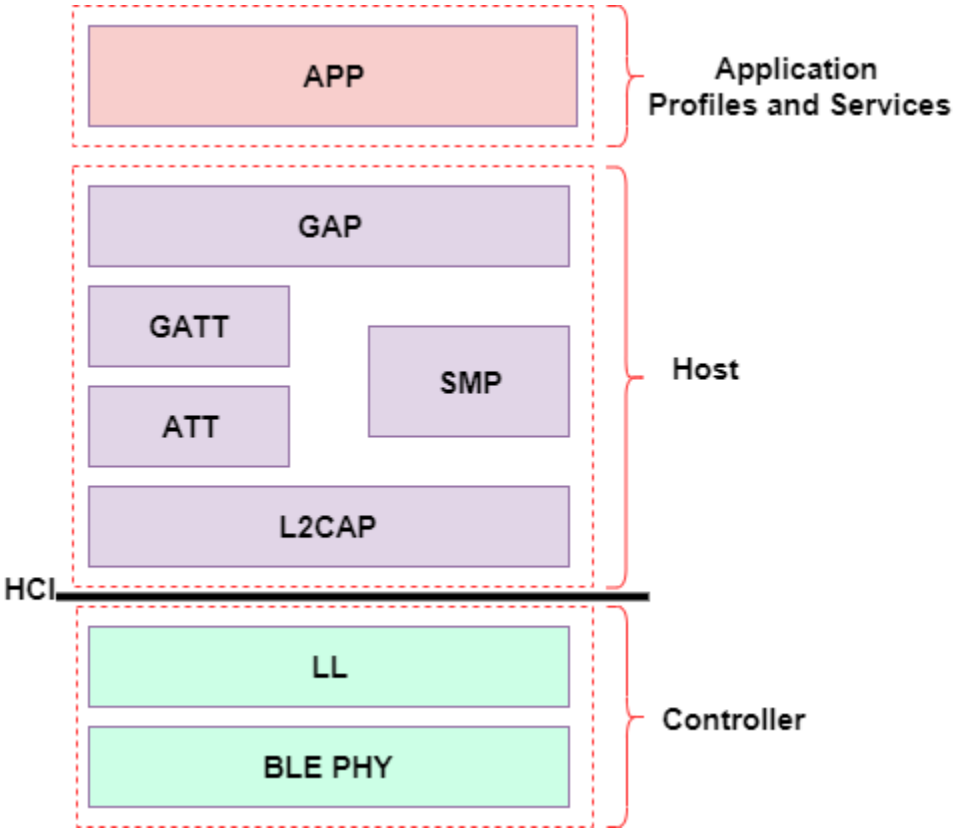
This figure compares the BLE protocol stack to the Open System Interconnection (OSI) reference model.



In the preceding figure, the BLE protocol stack is shown along with the OSI reference model.

- There is one-to-one mapping at the physical layer (PHY)
- The OSI data link layer (DLL) maps to the BLE logical link control and adaptation protocol (L2CAP) and link layer (LL)
- In the BLE stack, the higher layers provide application layer services, device roles and modes, connection management, and security protocol

The functionality of the BLE protocol stack is divided between three main layers: the Controller, the Host, and Application Profiles and Services.



Controller

The controller layer includes the BLE PHY, the LL, and the controller-side host controller interface (HCI).

BLE PHY

The BLE PHY air interface operates in the same unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band as Wi-Fi. The BLE PHY air interface also includes these characteristics:

- Operating radio frequency (RF) is in the range 2.4000 GHz to 2.4835 GHz, inclusive.
- The channel bandwidth is 2 MHz. The operating band is divided into 40 channels, k = 0, ..., 39. The center frequency of the kth channel is 2402 + k × 2 MHz.
  - User data packets are transmitted using channels in the range [0, 36].
  - Advertising data packets are transmitted in channels 37, 38, and 39.
- Gaussian frequency shift-keying (GFSK) modulation scheme is implemented.
- The BLE PHY uses frequency-hopping spread spectrum (FHSS) to reduce interference and to counter the impact of fading channels. The time between frequency hops can vary from 7.5 ms to 4 s and is set at the connection time for each slave.
- Support for throughput at 1 Mbps is mandatory for specification version 4.x compliant devices. At a data rate of 1 Mbps, the transmission is uncoded.
- Optionally, devices compliant with the Bluetooth Core Specification version 5.1 support these additional data rates:

- Coded transmission at bit rates of 500 kbps or 125 kbps
- Uncoded transmission at a bit rate of 2 Mbps

LL

The LL performs tasks similar to the medium access control (MAC) layer of the OSI model. In Bluetooth, the LL interfaces directly with the BLE PHY and manages the link state of the radio to define the role of a device as master, slave, advertiser, or, scanner.

Controller-Side HCI

The HCI on the controller side handles the interface between the host and the controller. The HCI defines a set of commands and events for transmission and reception of packet data. When receiving packets from the controller, the HCI extracts raw data at the controller to send to the host.

Host

The host includes the host-side HCI, L2CAP, attribute protocol (ATT), generic attribute profile (GATT), security manager protocol (SMP), and generic access profile (GAP).

Host-Side HCI

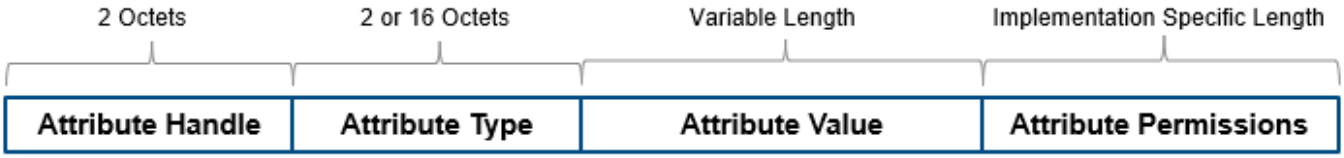
The HCI on the host side handles the interface between the host and the controller. The HCI defines a set of commands and events for transmission and reception of packet data. When transmitting data, the HCI translates raw data into packets to send them from the host to the controller.

L2CAP

The L2CAP encapsulates data from the BLE higher layers into the standard BLE packet format for transmission or extracts data from the standard BLE LL packet on reception according to the link configuration specified at the ATT and SMP layers.

ATT

The ATT transfers attribute data between clients and servers in GATT-based profiles. The ATT defines the roles of the client-server architecture. The roles typically correspond to the master and the slave as defined in the link layer. In general, a device could be a client, a server, or both, irrespective of whether it is a master or a slave. The ATT also performs data organization into attributes as shown in this figure.



Device attributes are represented as:

- The attribute handle is a 16-bit identifier value assigned by the server to enable a client to reference those attributes.
- The attribute type is a universally unique identifier (UUID) defined by Bluetooth SIG. For example, UUID 0x2A37 represents a heart-rate measurement.
- The attribute value is a variable length field. The UUID associated with and the service class of the service record containing the attribute value, determine the length of the attribute value field.

- Attribute permissions are sets of permission values associated with each attribute. These permissions specify read and write privileges for an attribute, and the security level required for read and write permission.

GATT

The GATT provides a reference framework for all GATT-based profiles. The GATT encapsulates the ATT and is responsible for coordinating the exchange of profiles in a BLE link. Profiles include information and data such as handle assignment, a UUID, and a set of permissions.

For devices that implement the GATT profile,

- The client is the device that initiates commands and requests toward the server. The client can receive responses, indications, and notifications.
- The server is the device that accepts incoming commands and requests from the client. The server sends responses, indications, and notifications to the client.

The GATT uses client-server architecture. The roles are not fixed and are determined when a device initiates a defined procedure. Roles are released when the procedure ends.

The terminology used in the GATT includes:

- Service — A collection of data and associated behaviors used to accomplish a particular function or feature
- Characteristic — A value used in a service along with appropriate permissions
- Characteristic descriptor — A description of the associated characteristic behavior
- GATT-Client — A GATT-Client initiates commands and requests towards the server and can receive responses, indications, and notifications sent by the server
- GATT-Server — A GATT-Server accepts incoming commands and requests from a client and sends responses, indications, and notifications to the client

SMP

The SMP applies security algorithms to encrypt and decrypt data packets. This layer defines the initiator and the responder, corresponding to the master and the slave, once the connection is established.

GAP

The GAP specifies roles, modes, and procedures of a device. It also manages the connection establishment and security. The GAP interfaces directly with the Application Profiles and Services (App) layer.

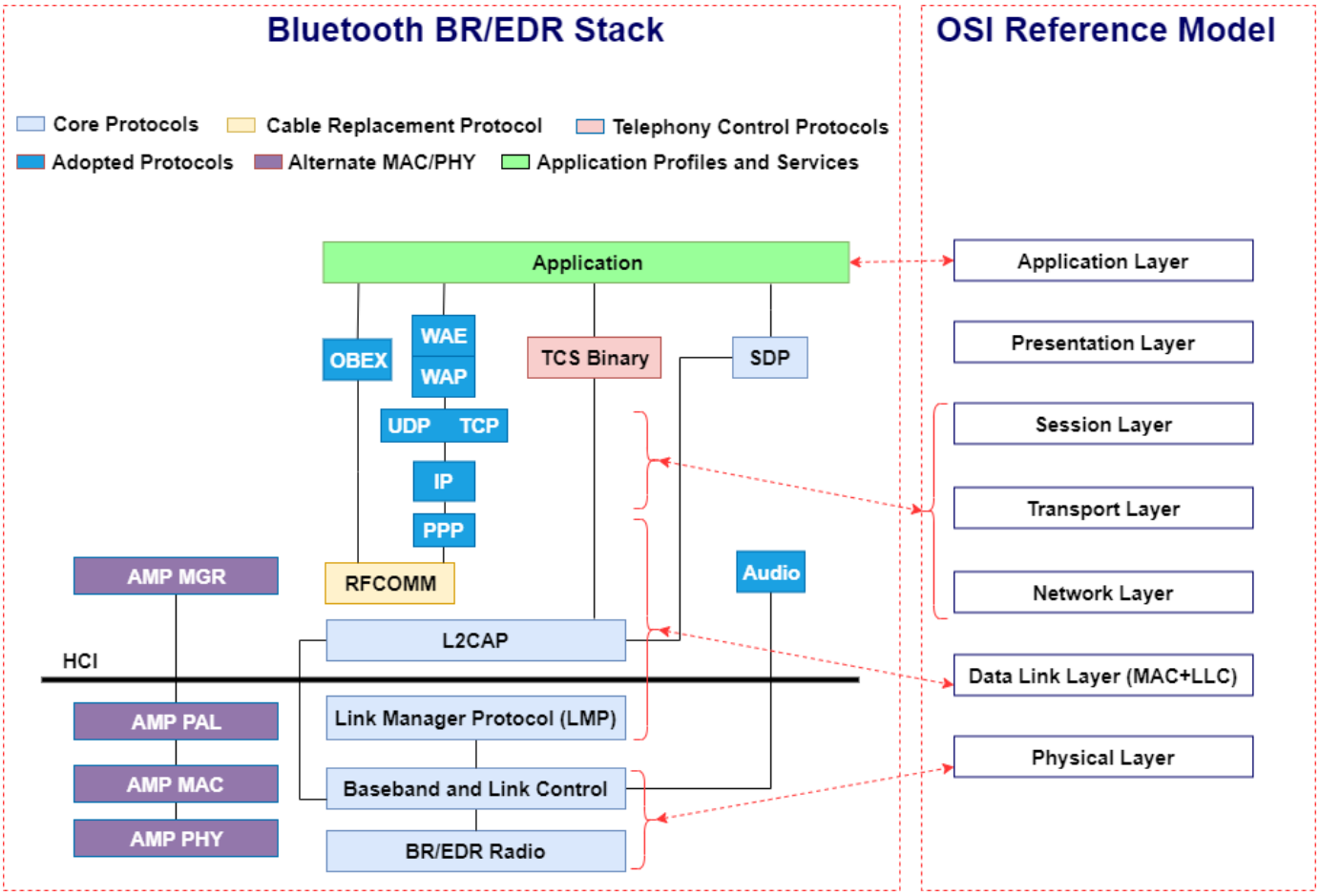
APP Layer

The App layer is the direct user interface defining profiles that afford interoperability between various applications. The Bluetooth core specification enables vendors to define proprietary profiles for use cases not defined by SIG profiles.

Note For more information about the BLE protocol stack architecture, see volume 3, Part C, sections 2 and 2.1 of the Bluetooth Core Specification [1].

Bluetooth BR/EDR Protocol Stack

This figure compares the block diagram of the Bluetooth BR/EDR protocol stack and with the OSI reference model.



The mapping of BR/EDR stack to the OSI reference model is as shown below:

- The “BR/EDR Radio” on page 13-13 and “Baseband and Link Control” on page 13-13 layers of the Bluetooth BR/EDR stack map to the OSI PHY layer.
- The “Link Manager Protocol (LMP)” on page 13-13, “L2CAP” on page 13-13, “Cable Replacement Protocol” on page 13-13 (RFCOMM), and “PPP” on page 13-14 layers of the Bluetooth BR/EDR stack map to the OSI data link layer.
- The user datagram protocol (UDP), transmission control protocol (TCP), and internet protocol (IP) layers of Bluetooth BR/EDR stack map to a combined, network, transport and session layers of the OSI reference model.
- There is one-to-one mapping at the application layer.

Core Protocols

The Bluetooth core protocols and the Bluetooth radio are required by most of the Bluetooth devices. The core protocols include these layers.

BR/EDR Radio

The BR/EDR radio is the lowest defined layer of the Bluetooth specification. The BR mode is mandatory, whereas the EDR mode is optional. This layer defines the requirements of the Bluetooth transceiver device operating in the 2.4 GHz ISM frequency band. It implements a 1600 hops/sec FHSS technique. The radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth channel has a bandwidth of 1 MHz. Each frequency is located at (2402 + k) MHz, where k = 0,1,...78. The modulation technique for BR and EDR mode is GFSK and differential phase shift keying (DPSK), respectively. The baud rate is 1 Msymbols/s. The Bluetooth BR/EDR radio uses the time division duplex (TDD) topology in which data transmission occurs in one direction at one time. The transmission alternates in two directions, one after the other.

Baseband and Link Control

The baseband and link control layer enables the PHY RF link between different Bluetooth devices, forming a piconet. The baseband handles the channel processing and timing, and the link control handles the channel access control. This layer provides these two different types of PHY RF links with their corresponding baseband packets:

- Synchronous connection-oriented (SCO) – Supports real-time audio traffic
- Asynchronous connection-oriented (ACL) – Supports data packet transmission

Link Manager Protocol (LMP)

The LMP layer is primarily responsible for link setup and link configuration between different Bluetooth devices. These processes include establishing security functions such as authentication and encryption by generating, exchanging, and checking link and encryption keys. Furthermore, this layer controls the power modes and duty cycles of the Bluetooth radio device and the connection states of a Bluetooth unit in a piconet.

L2CAP

The L2CAP adapts higher-layer protocols over the baseband. It shields the higher-layer protocols from the details of the lower-layer protocols. The L2CAP provides connection-oriented and connectionless services to the higher-layer protocols. This includes protocol multiplexing capability, segmentation and reassembly operations, and group abstractions.

SDP

Discovery services are an important aspect of the Bluetooth framework. The service discovery protocol (SDP) provides a means for applications to query services and the characteristics of services, following which a connection can be established between two or more Bluetooth devices. The SDP is quite different from service discovery in traditional network-based environments. The SDP is built on top of the L2CAP.

Cable Replacement Protocol

The cable replacement protocol in the Bluetooth BR/EDR stack uses RFCOMM to provide emulation of serial ports over L2CAP. RFCOMM emulates RS-232 control and data signals over the Bluetooth baseband and provides transport capabilities for higher-layer services that use a serial interface as a transport mechanism. RFCOMM also provides multiple simultaneous connections to one device and enables connections to multiple devices.

Telephony Control Protocols

The telephony control protocol specification, binary (TCS binary), defines the call control signaling to establish data and voice calls between Bluetooth devices. It is built on top of the L2CAP. Moreover, TCS binary defines mobility management procedures for handling Bluetooth devices.

Adopted Protocols

In addition to the core protocols, the Bluetooth BR/EDR stack includes protocols adopted from other standard bodies. These adopted protocols are defined in specifications issued by other standard-making organizations and are incorporated into the Bluetooth framework.

PPP

The point-to-point protocol (PPP) is an Internet Engineering Task Force (IETF) [3] standard protocol for transporting IP datagrams over a point-to-point link. The PPP runs over the RFCOMM to realize point-to-point connections.

TCP, UDP, and IP

These layers are the IETF-defined foundation protocols of the TCP/IP protocol suite.

- TCP – This protocol provides a reliable virtual connection between devices to realize data communication. The TCP treats the data as a stream of bytes and transmits them without any errors or duplication.
- UDP – This protocol is an alternative to the TCP and provides an unreliable datagram connection between devices. As there is no end-to-end connection in UDP, data is transmitted link-by-link without any guarantee of service.
- IP – This layer is a network layer protocol that enables a datagram service between devices, supporting both the TCP and UDP.

The use of the TCP, UDP, and IP in the Bluetooth BR/EDR stack enables communication with any other device connected to the Internet.

OBEX

The object exchange (OBEX) protocol is a session-level protocol developed by the Infrared Data Association (IrDA) to exchange objects. The OBEX protocol provides functionality similar to that of HTTP, but in a simpler manner. HTTP is an application layer protocol and layered above the TCP/IP. The OBEX protocol provides the client with a reliable transport for connecting to a server. It also provides a model for representing objects and operations.

WAE and WAP

Bluetooth BR/EDR stack incorporates the wireless application environment (WAE) and wireless application protocol (WAP) into its architecture. The advantages of using WAE/WAP features in the Bluetooth stack are:

- Build application gateways that act as an interface between WAP servers and some other application on the PC
- Provide functions such as remote control and data fetching from the PC to the Bluetooth handset
- Reuse the upper software application developed for the WAP application environment

Application Profiles and Services

For more information, refer “APP Layer” on page 13-11.

Alternate MAC/PHY

The alternate MAC/PHY (AMP) manager is a secondary controller in the Bluetooth core system. After an L2CAP connection is established between two devices over the BR/EDR radio, the AMP manager can discover the AMPs that are available on the other device. If an AMP is common between two devices, the Bluetooth core system provides mechanisms for moving data traffic from the BR/EDR controller to an AMP controller.

Each AMP manager consists of a protocol adaptation layer (PAL) on top of a MAC and PHY. The PAL maps the Bluetooth protocols to the specific protocols of the underlying MAC and PHY.

L2CAP channels can be created on, or moved to, an AMP. If an AMP physical link has a link supervision timeout, then L2CAP channels can be moved back to BR/EDR radio. To minimize power consumption in the device, AMPs are enabled or disabled as required.

HCI

The HCI provides a command interface to the BR/EDR radio, baseband controller, and the link manager. It is a single standard interface for accessing the Bluetooth baseband capabilities, the hardware status, and the control registers.

Note For more information about the Bluetooth BR/EDR protocol stack architecture, see volume 1, Part A, sections 2 and 2.1 of the Bluetooth Core Specification [1].

References

[1] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[2] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 6, 2019. https://www.bluetooth.com/.

[3] IETF. “Internet Standards.” Accessed November 6, 2019. https://www.ietf.org/.

[4] Bluetooth Protocol Stack - an Overview | ScienceDirect Topics. Accessed November 15, 2019. https://www.sciencedirect.com/.

See Also

More About

- “What Is Bluetooth?” on page 13-2
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Location and Direction Finding” on page 13-37
- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Bluetooth Parameterization

Communications Toolbox Library for the Bluetooth Protocol configuration objects initialize, store, and validate configuration properties.

Configuration Objects in Communications Toolbox Library for the Bluetooth Protocol

The configuration objects are designed specifically as containers to store properties. They also provide some level of data validation for the function inputs that they maintain. Functions perform further data validation across input settings based on run-time conditions.

The configuration objects are optimized for iterative computations that process large streams of data, such as communications systems.

Communications Toolbox Library for the Bluetooth Protocol configuration objects define and configure format-specific and function-specific properties. The property page of each object contains descriptions, valid settings, ranges, and other information about the object properties.

- `bleLLDataChannelPDUConfig` — The Link Layer Data Channel Protocol Data Unit configuration object defines and configures LL data channel PDUs.
- `bleLLAdvertisingChannelPDUConfig` — The Link Layer Advertising Channel Protocol Data Unit configuration object defines and configures LL advertising channel PDUs.
- `bleL2CAPFrameConfig` — The Layer 2 CAP Frame configuration object defines and configures Layer 2 CAP frame.
- `bleGAPDataBlockConfig` — The GAP Data Block configuration object defines and configures GAP data block.
- `bleATTPDUConfig` — The ATT Protocol Data Unit configuration object defines and configures ATT PDUs.

See Also

More About

- “What Is Bluetooth?” on page 13-2
- “Create Configuration Objects” on page 13-17
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Location and Direction Finding” on page 13-37
- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Create Configuration Objects

Communications Toolbox Library for the Bluetooth Protocol uses value objects to organize properties required for generation of higher layer Bluetooth PDUs. After you create the various configuration objects described here, you can use them to generate waveforms.

Create Link Layer Data Channel PDU Configuration Object

This example shows how to create a BLE link layer data channel PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using `Name,Value` pairs when creating the object.

Create Object and Then Modify Properties

Create a BLE link layer data channel PDU configuration object with default settings.

```
lldatapdu = bleLLDataChannelPDUConfig
```

```
lldatapdu =
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (continuation fragment/empty)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: '012345'

   Read-only properties:
    No properties.
```

Modify the defaults to specify a `'Data (start fragment/complete)'` PDU.

```
lldatapdu.LLID = 'Data (start fragment/complete)'
```

```
lldatapdu =
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (start fragment/complete)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: '012345'

   Read-only properties:
    No properties.
```

Override Default Property Values During Object Creation

Create a BLE link layer data channel PDU configuration object, using `Name,Value` pairs to specify a `'Control'` PDU.

```
lldatapdu2 = bleLLDataChannelPDUConfig('LLID','Control')
```

```
lldatapdu2 =
  bleLLDataChannelPDUConfig with properties:
```

```
                LLID: 'Control'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
   CRCInitialization: '012345'
       ControlConfig: [1x1 bleLLControlPDUConfig]

  Read-only properties:
   No properties.
```

Create Link Layer Advertising Channel PDU Configuration Object

This example shows how to create a BLE link layer advertising channel PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using `Name,Value` pairs when creating the object.

Create Object and Then Modify Properties

Create a BLE link layer advertising channel PDU configuration object with default settings.

```
lladvertpdu = bleLLAdvertisingChannelPDUConfig
```

```
lladvertpdu =
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Advertising indication'
        ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
          AdvertisingData: [3x2 char]

  Read-only properties:
   No properties.
```

Modify the defaults to specify a `'Advertising direct indication'` PDU.

```
lladvertpdu.PDUType = 'Advertising direct indication'
```

```
lladvertpdu =
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Advertising direct indication'
        ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
        TargetAddressType: 'Random'
            TargetAddress: '0123456789CD'

  Read-only properties:
   No properties.
```

Override Default Property Values During Object Creation

Create a BLE link layer advertising channel PDU configuration object, using `Name,Value` pairs to specify a `'Scan response'` PDU using channel section algorithm 2.

```
lldatapdu2 = bleLLAdvertisingChannelPDUConfig('PDUType','Scan response','ChannelSelection','Algo
```

```
lldatapdu2 =
  bleLLAdvertisingChannelPDUConfig with properties:

                 PDUType: 'Scan response'
       ChannelSelection: 'Algorithm2'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
         ScanResponseData: [3x2 char]

   Read-only properties:
    No properties.
```

Create L2CAP Frame Configuration Object

This example shows how to create a BLE logical link control and adaptation protocol (L2CAP) frame configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using `Name,Value` pairs when creating the object.

Create Object and Then Modify Properties

Create a BLE L2CAP frame configuration object with default settings.

```
l2capframe = bleL2CAPFrameConfig
```

```
l2capframe =
  bleL2CAPFrameConfig with properties:

         ChannelIdentifier: '0005'
               CommandType: 'Credit based connection request'
          SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 1

   Read-only properties:
    No properties.
```

Modify the defaults setting the channel identifier to `'0004'` to specify an ATT channel.

```
l2capframe.ChannelIdentifier = '0004'
```

```
l2capframe =
  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'

   Read-only properties:
```

```
    No properties.
```

Override Default Property Values During Object Creation

Create a BLE L2CAP frame configuration object, using `Name,Value` pairs to specify a `'Command reject'` signaling channel command with the reject reason `'Invalid CID in request'`.

```
l2capframe = bleL2CAPFrameConfig ('CommandType','Command reject','CommandRejectReason','Invalid C
```

```
l2capframe =
  bleL2CAPFrameConfig with properties:

              ChannelIdentifier: '0005'
                    CommandType: 'Command reject'
               SignalIdentifier: '01'
           CommandRejectReason: 'Invalid CID in request'
         SourceChannelIdentifier: '0040'
    DestinationChannelIdentifier: '0040'

   Read-only properties:
    No properties.
```

Create GAP Data Block Configuration Object

This example shows how to create a BLE GAP data block configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using `Name,Value` pairs when creating the object.

Create Object and Then Modify Properties

Create a BLE GAP data block configuration object with default settings.

```
gapDataBlk = bleGAPDataBlockConfig
```

```
gapDataBlk =
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {'Flags'}
       LEDiscoverability: 'General'
                   BREDR: 0

   Read-only properties:
    No properties.
```

Modify the defaults to specify for an advertising data block for `'Flags'` and `'Tx power level'` advertising data types.

```
gapDataBlk.AdvertisingDataTypes = {'Flags'; 'Tx power level'}
```

```
gapDataBlk =
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
       LEDiscoverability: 'General'
```

```
                  BREDR: 0
            TxPowerLevel: 0

   Read-only properties:
    No properties.
```

```
gapDataBlk.AdvertisingDataTypes
```

```
ans = 2x1 cell
    {'Flags'         }
    {'Tx power level'}
```

Override Default Property Values During Object Creation

Create a BLE GAP data block configuration object, using `Name,Value` pairs to specify `'Advertising interval'` and `'Local name'` advertising data types.

```
gapDataBlk = bleGAPDataBlockConfig  ('AdvertisingDataTypes', {'Advertising interval', 'Local name
```

```
gapDataBlk =
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'Bluetooth'
     LocalNameShortening: 0
     AdvertisingInterval: 32

   Read-only properties:
    No properties.
```

```
gapDataBlk.AdvertisingDataTypes
```

```
ans = 2x1 cell
    {'Advertising interval'}
    {'Local name'          }
```

Create Attribute PDU Configuration Object

This example shows how to create a BLE attribute (ATT) PDU configuration object. It also shows how to change the default property settings by using dot notation or by overriding the default settings by using `Name,Value` pairs when creating the object.

Create Object and Then Modify Properties

Create a BLE ATT PDU configuration object with default settings.

```
attpdu = bleATTPDUConfig
```

```
attpdu =
  bleATTPDUConfig with properties:

              Opcode: 'Read request'
    AttributeHandle: '0001'
```

```
   Read-only properties:
    No properties.
```

Modify the defaults to specify a `'Read blob request'` operation code.

```
attpdu.Opcode = 'Read blob request'
```

```
attpdu =
  bleATTPDUConfig with properties:

              Opcode: 'Read blob request'
    AttributeHandle: '0001'
              Offset: 0

   Read-only properties:
    No properties.
```

Override Default Property Values During Object Creation

Create a BLE ATT PDU configuration object, using `Name,Value` pairs to specify `'Information request'` for the operation code of a request PDU that caused an error.

```
lldatapdu2 = bleATTPDUConfig('RequestedOpcode','Information request')
```

```
lldatapdu2 =
  bleATTPDUConfig with properties:

              Opcode: 'Read request'
    AttributeHandle: '0001'

   Read-only properties:
    No properties.
```

See Also

More About

- “What Is Bluetooth?” on page 13-2
- “Bluetooth Parameterization” on page 13-16
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Location and Direction Finding” on page 13-37
- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Bluetooth Packet Structure

The Bluetooth Special Interest Group (SIG) [1] and [2] defines different packet structures for Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR) devices.

BLE Packet Structure

Bit Ordering in BLE Packets

When defining packets and messages in the baseband specification, the bit ordering follows the little-endian format. In this format, these rules apply:

- The least significant bit (LSB) corresponds to b_0.
- LSB is the first bit sent over the air.
- When illustrating the packet structure, the LSB is shown on the left side.

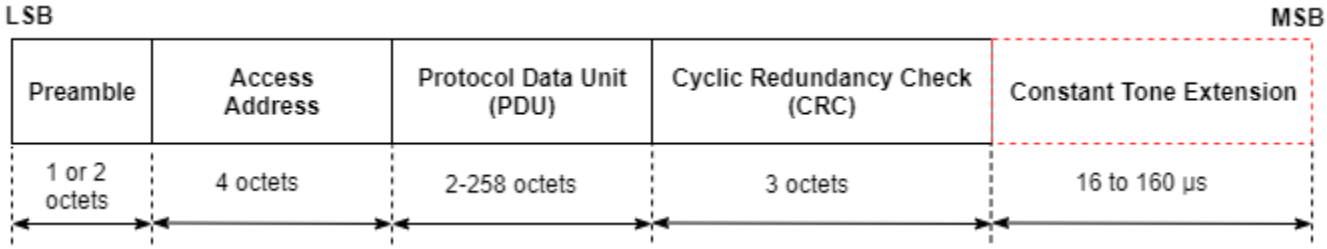
Moreover, data fields generated internally at the baseband level (packet header and payload header length), must be transmitted with the LSB first. For example, a 3-bit parameter is sent as: $b_0b_1b_2 = 110$ over the air, where 1 is sent first and 0 is sent last.

The BLE devices use packet formats for: “BLE Uncoded Physical Layer (PHY)” on page 13-23, “BLE Coded PHY” on page 13-24, “Advertising Physical Channel PDU” on page 13-26, “Data Physical Channel PDU” on page 13-27, and “Constant Tone Extension and In-Phase Quadrature (IQ) Sampling” on page 13-30.

Note For more information about BLE packet structure, see volume 6, Part B, Section 2 of the Bluetooth Core Specification [2].

BLE Uncoded Physical Layer (PHY)

The Bluetooth Core Specification [2] defines two physical layer (PHY) transmission modes (LE 1M and LE 2M) for uncoded PHY. This figure shows the packet structure for the BLE uncoded PHY operating on LE 1M and LE 2M.



Each packet contains four mandatory fields (preamble, access-address, protocol data unit (PDU), and cyclic redundancy check (CRC)) and one optional field (constant tone extension (CTE)). The preamble is transmitted first, followed by the access address, PDU, CRC, and CTE (if present) in that order. The entire packet is transmitted at the same symbol rate of 1 Msym/s or 2 Msym/s.

Preamble

All link layer (LL) packets contain a preamble, which is used in the receiver to perform frequency synchronization, automatic gain control (AGC) training, and symbol timing estimation. The preamble

is a fixed sequence of alternating 0 and 1 bits. For the BLE packets transmitted on the LE 1M PHY and LE 2M PHY, the preamble size is 1 octet and 2 octets, respectively.

Access address

The access address is a 4-octet value. Each LL connection between any two devices and each periodic advertising train has a distinct access address. Each time the BLE device needs a new access address, the LL generates a new random value adhering to these requirements:

- The value must not be the access address for any existing LL connection on this device.
- The value must not be the access address for any enabled periodic advertising train.
- The value must have no more than six successive 1s or 0s.
- The value must not be the access address for any advertising channel packets.
- The value must not be a sequence that differs from the access address of advertising physical channel packets by only 1 bit.
- All four octets for the value must not be equal.
- The value must have a minimum of two transitions in the most significant 6 bits.

If the random value does not satisfy the above requirements, a new random value is generated until it meets all of the requirements.

PDU

When a BLE packet is transmitted on either the primary or secondary advertising physical channel or the periodic physical channel, the PDU is defined as the “Advertising Physical Channel PDU” on page 13-26. When a packet is transmitted on the data physical channel, the PDU is defined as the “Data Physical Channel PDU” on page 13-27.

CRC

The size of the CRC is 3 octets and is calculated on the PDU of all LL packets. If the PDU is encrypted, then the CRC is calculated after encryption of the PDU is complete. The CRC polynomial has the form $x^{24}+x^{10}+x^{9}+x^{6}+x^{4}+x^{3}+x+1$.

For more information about CRC generation, see volume 6, Part B, Section 3.1.1 of the Bluetooth Core Specification [2].

CTE

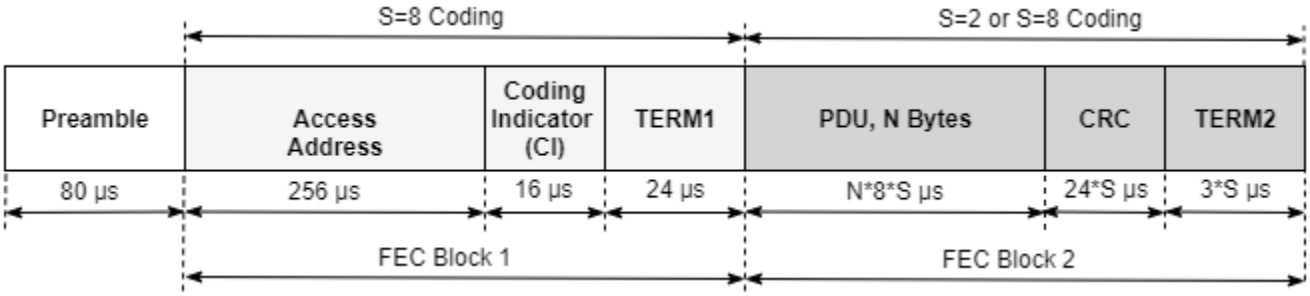
The CTE consists of a constantly modulated series of unwhitened 1s. This field has a variable length that ranges from 16 µs to 160 µs.

For more information about the CTE, see volume 6, Part B, Section 2.5.1 of the Bluetooth Core Specification [2].

Note For more information about BLE uncoded PHY packet structure, refer to Vol 6, Part B, Section 2.1 of Bluetooth Core Specification [2].

BLE Coded PHY

This figure shows the packet structure for the BLE coded PHY and is implemented for BLE packets on all the physical channels.



Each BLE packet consists of a preamble and these two forward error correcting (FEC) blocks:

- FEC block 1— This block contains three fields: access address, coding indicator (CI), and TERM1. This block implements an *S*=8 coding scheme, where each bit represents eight symbols. This gives a data rate of 125 Kbps.
- FEC block 2— This block contains these three fields: PDU, CRC, and TERM2. This block implements an *S*=8 or *S*=2 coding scheme. In the *S*=2 coding scheme, each bit represents two symbols. Therefore, the data rate is 500 Kbps.

The BLE coded PHY does not contain the CTE.

Preamble

The BLE coded PHY preamble is 80 symbols in length and contains 10 repetitions of the symbol pattern '00111100' (in the transmission order).

Access address

The length of BLE coded PHY access address is 256 symbols. For more information, see “Access address” on page 13-24. In addition to the requirements listed in the access address subsection of the “BLE Uncoded Physical Layer (PHY)” on page 13-23 section, the new value for the access address of the BLE coded PHY must also meet these requirements:

- The value must have at least three 1s in the last significant bits.
- The value must have no more than 11 transitions in the least significant 16 bits.

CI

The CI field consists of two bits as shown in this table:

Bits in CI	Description
`00b`	FEC block 2 coded using *S*=8
`01b`	FEC block 2 coded using *S*=2
All other values	Reserved for future use

PDU

The PDU in the BLE coded PHY packet structure has the same formatting as the “PDU” on page 13-24 in the BLE uncoded PHY packet.

CRC

The CRC in the BLE coded PHY packet structure has the same formatting as the “CRC” on page 13-24 in the BLE uncoded PHY packet.

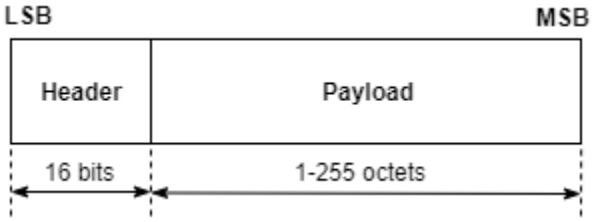
TERM1 and TERM2

Each FEC block contains a terminator at the end of the block. That terminator is referred to as TERM1 and TERM2. Each terminator is 3 bits long and forms the termination sequence during the FEC encoding process.

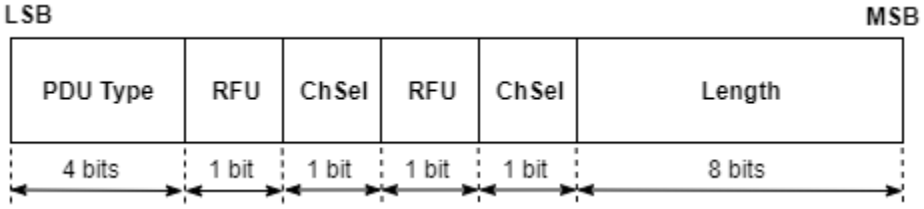
Note For more information about BLE coded PHY packet structure, see volume 6, Part B, Section 2.2 of the Bluetooth Core Specification [2].

Advertising Physical Channel PDU

The packet structure format of the advertising physical channel PDU is shown in this figure.



The advertising physical channel PDU has a 16-bit header and a variable-size payload. The 16-bit header field of the advertising physical channel PDU is shown in this figure.



The PDU type field in the advertising channel PDU header defines different types of PDUs that can be transmitted on the BLE coded PHY. This table maps different types of PDUs with the physical channels and the PHYs on which the BLE packet might appear. The table also indicates the PHY transmission modes supported for each type of advertising physical channel PDU.

PDU Type	PDU Name	Physical Channel	LE 1M Support	LE 2M Support	LE Coded Support
0000b	ADV_IND	Primary Advertising	Yes		
0001b	ADV_DIRECT_IND	Primary Advertising	Yes		
0010b	ADV_NONCONN_IND	Primary Advertising	Yes		
0011b	SCAN_REQ	Primary Advertising	Yes		
	AUX_SCAN_REQ	Secondary Advertising	Yes	Yes	Yes
0100b	SCAN_RSP	Primary Advertising	Yes		
0101b	CONNECT_IND	Primary Advertising	Yes		
	AUX_CONNECT_REQ	Secondary Advertising	Yes	Yes	Yes

PDU Type	PDU Name	Physical Channel	LE 1M Support	LE 2M Support	LE Coded Support
0110b	ADV_SCAN_IND	Primary Advertising	Yes		
0111b	ADV_EXT_IND	Primary Advertising	Yes		Yes
	AUX_ADV_IND	Secondary Advertising	Yes	Yes	Yes
	AUX_SCAN_RSP	Secondary Advertising	Yes	Yes	Yes
	AUX_SYNC_IND	Periodic	Yes	Yes	Yes
	AUX_CHAIN_IND	Secondary Advertising and Periodic	Yes	Yes	Yes
1000b	AUX_CONNECT_RSP	Secondary Advertising	Yes	Yes	Yes
All other values	Reserved for future use				

The RFU field is reserved for future use. The `ChSel`, `TxAdd`, and `RxAdd` fields of the advertising physical channel PDU header contain information specific to the type of PDU defined for each advertising physical channel PDU separately. If the `ChSel`, `TxAdd`, or `RxAdd` fields are not defined as used in a given PDU, then they are considered as reserved for future use.

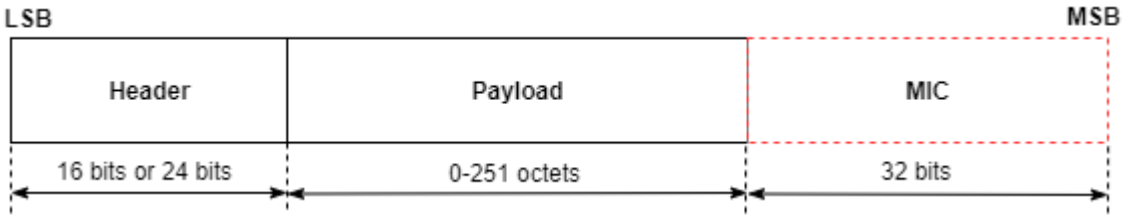
The `Length` field of the advertising physical channel PDU header denotes the length of the payload in octets. The valid range of the Length field is 1 to 255 octets.

The `Payload` field in the advertising physical channel PDU packet structure is specific to the type of PDUs listed in the preceding table.

Note For more information about advertising physical channel PDUs, see volume 6, Part B, Section 2.3 of the Bluetooth Core Specification [2].

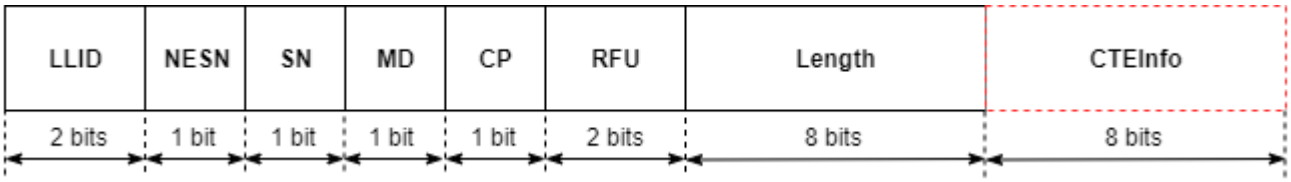
Data Physical Channel PDU

The packet structure format of the data physical channel PDU is shown in this figure.



The data physical channel PDU has a 16-bit or 24-bit header, a variable length payload in the range [0, 251] octets, and can include a 32-bit message integrity check (MIC) field. The MIC is not included in an unencrypted LL connection or in an encrypted LL connection with a data channel PDU containing an empty payload. The MIC is included in an encrypted LL connection with a data channel PDU containing a nonzero length payload. In this case, the MIC is calculated as specified in volume 6, Part E, Section 1 of the Bluetooth Core Specification [2].

The header field of the data physical channel PDU is shown in this figure.



The data physical channel PDU header includes these fields:

- Link layer identifier (LLID) — This field indicates whether the packet is an LL data PDU or LL control PDU.
  - `00b` — Reserved for future use
  - `01b` — LL Data PDU, which can be a continuation fragment of an logical link control and adaptation (L2CAP) message or an empty PDU
  - `10b` — LL Data PDU, which can be a start of an L2CAP message or a complete L2CAP message with no fragmentation
  - `11b` — LL control PDU
- Next expected sequence number (NESN): The LL uses this field to either acknowledge the last data physical channel PDU sent by the peer or to request the peer to resend the last data physical channel PDU. For more information about NESN, see volume 6, Part B, Section 4.5.9 of the Bluetooth Core Specification [2].
- Sequence number (SN): The LL uses this field to identify the BLE packets sent by it. For more information about the SN, see volume 6, Part B, Section 4.5.9 of the Bluetooth Core Specification [2].
- More data (MD): This field indicates that the BLE device has more data to send. If neither of master and slave BLE device has set the MD bit in their packets, the packet from the slave closes the connection event. If the master and slave devices have set the MD bit, the master can continue the connection event by sending another packet, and the slave must listen after sending its packet. For more information about MD, see volume 6, Part B, Section 4.5.6 of the Bluetooth Core Specification [2].
- CTEInfo present (CP): This field indicates whether the data physical channel PDU header has a CTEInfo field and, subsequently whether the data physical channel packet has a CTE. For more information about the packet structure of the CTEInfo field, see volume 6, Part B, Section 2.5.2 of the Bluetooth Core Specification [2].
- Length: This field indicates the size, in octets, of the payload and MIC, if present. The size of this field is in the range [0, 255] octets.
- CTEInfo: This field indicates the type and length of the CTE.

The two types of data physical channel PDUs are: “LL Data PDU” on page 13-28 and “LL Control PDU” on page 13-29.

LL Data PDU

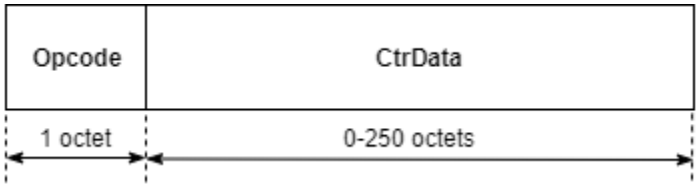
The LL uses the LL data PDU to send L2CAP data. The LLID field in the LL data channel PDU header is set to either `01b` or `10b`. An LL data PDU is referred to as an empty PDU if

- The LLID field of the LL data channel PDU header is set to `01b`.
- The Length field of the LL data channel PDU header is set to `00000000b`.

An LL data PDU with the LLID field in the header set to `10b` does not have the Length field set to `00000000b`.

LL Control PDU

The LL uses the LL data PDU to control the LL connection. If the LLID field of data physical channel PDU header is set to `11b`, the data physical channel PDU contains an LL control PDU. This figure shows the LL control PDU payload.



The Opcode field defines different types of LL control PDUs as shown in this table.

Opcode	LL Control PDU
0x00	LL_CONNECTION_UPDATE_IND
0x01	LL_CHANNEL_MAP_IND
0x02	LL_TERMINATE_IND
0x03	LL_ENC_REQ
0x04	LL_ENC_RSP
0x05	LL_START_ENC_REQ
0x06	LL_START_ENC_RSP
0x07	LL_UNKNOWN_RSP
0x08	LL_FEATURE_REQ
0x09	LL_FEATURE_RSP
0x0A	LL_PAUSE_ENC_REQ
0x0B	LL_PAUSE_ENC_RSP
0x0C	LL_VERSION_IND
0x0D	LL_REJECT_IND
0x0E	LL_SLAVE_FEATURE_REQ
0x0F	LL_CONNECTION_PARAM_REQ
0x10	LL_CONNECTION_PARAM_RSP
0x11	LL_REJECT_EXT_IND
0x12	LL_PING_REQ
0x13	LL_PING_RSP
0x14	LL_LENGTH_REQ
0x15	LL_LENGTH_RSP
0x16	LL_PHY_REQ
0x17	LL_PHY_RSP

Opcode	LL Control PDU
0x18	LL_PHY_UPDATE_IND
0x19	LL_MIN_USED_CHANNELS_IND
0x1A	LL_CTE_REQ
0x1B	LL_CTE_RSP
0x1C	LL_PERIODIC_SYNC_IND
0x1D	LL_CLOCK_ACCURACY_REQ
0x1E	LL_CLOCK_ACCURACY_RSP
All other values	Reserved for future use

The CtrData field in the LL control PDU is specific to the value of the Opcode field. For more information about different LL control PDUs and their corresponding CtrData field structure, see volume 6, Part B, Sections 2.4.2.1 to 2.4.2.28 of the Bluetooth Core Specification [2].

Constant Tone Extension and In-Phase Quadrature (IQ) Sampling

The length of the CTE is variable and in the range [16, 160] µs. This field contains a constantly modulated series of 1s with no whitening applied. The CTE is of two types: antenna switching during CTE transmission (AoD) and antenna switching during CTE reception (AoA). When receiving a packet containing an AoD CTE, the receiver does not need to switch antennae. When receiving a packet containing an AoA CTE, the receiver performs antenna switching according to the switching pattern configured by the host. In both cases, the receiver takes an IQ sample at each microsecond during the reference period and an IQ sample each sample slot. The controller reports the IQ samples to the host. The receiver samples the entire CTE regardless of its length, unless this conflicts with other activities. For more information about CTE, see volume 6, Part B, Sections 2.5.1 to 2.5.3 of the Bluetooth Core Specification [2].

When requested by the host, the receiver performs IQ sampling when receiving a valid BLE packet with a CTE. However, when receiving a BLE packet with a CTE and an incorrect CRC, the receiver might perform IQ sampling. For more information about IQ sampling, see volume 6, Part B, Section 2.5.4 of the Bluetooth Core Specification [2].

Note For more information about data physical channel PDUs, see volume 6, Part B, Section 2.4 of the Bluetooth Core Specification [2].

Bluetooth BR/EDR Packet Structure

Bit Ordering in Bluetooth BR/EDR Packets

The bit ordering in Bluetooth BR/EDR packets follows the same format as the “Bit Ordering in BLE Packets” on page 13-23.

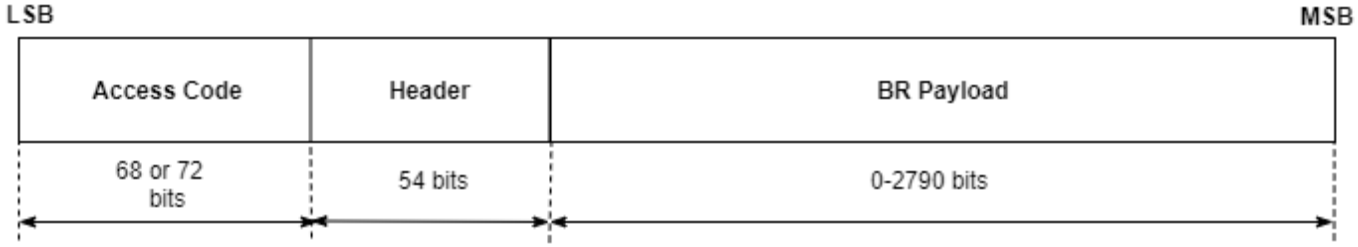
Bluetooth BR/EDR devices use packet formats for: “BR Mode” on page 13-31, “EDR Mode” on page 13-31, “Access Code” on page 13-31, “Packet Header” on page 13-33, “Packet Types” on page 13-34, and “Payload Format” on page 13-35.

Note For more information about Bluetooth BR/EDR packet structure, see volume 2, Part B, Section 6 of the Bluetooth Core Specification [2].

General Format

BR Mode

The general format of Bluetooth BR packets is shown in this figure. Each packet consists of these fields: the access code (68 or 72 bits), header (54 bits), and payload in the range [0, 2790] bits.

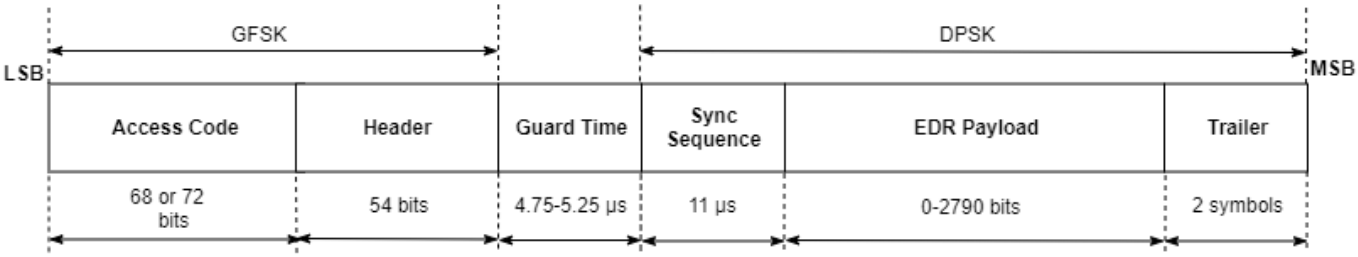


The Bluetooth Core Specification [2] defines different types of packets. A packet can consist of:

- The shortened access code only
- The access code and the packet header
- The access code, the packet header, and the payload

EDR Mode

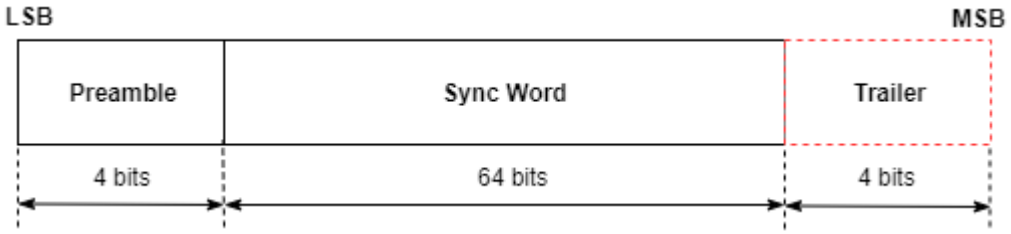
The general format of Bluetooth EDR packets is shown in this figure.



The format and modulation of the access code and the packet header fields are similar to that of BR packets. Following the header field, the EDR packets have a guard time in the range [4.75, 5.25] µs, a sync sequence (11 µs), payload in the range [0, 2790] bits, and trailer (two symbols) fields.

Access Code

Each packet starts with an access code. If a packet header follows, the access code is 72 bits long. Otherwise, the length of the access code is 68 bits. In this case, the access code is referred to as a shortened access code. The shortened access code does not contain a trailer. The access code is used for synchronization, DC offset compensation, and identification of all packets exchanged on the physical channel. The shortened access code is used in paging and inquiry. In this case, the access code itself is used as a signaling message, and neither a header nor a payload is present. This figure shows the packet structure of the access code.



Different access code types use different lower address parts (LAPs) to construct the sync word. A summary of different access code types is shown in this table.

Access Code Type	LAP	Access Code Length (Bits)	Description
Channel access code (CAC)	Master	72	This access code is used in the connection state, synchronization train substate, and synchronization scan substate. It is derived from the LAP of the Master's `BD_ADDR` .
Device access code (DAC)	Paged device	68 or 72	This access code is used during page, page scan, and page response substates. It is derived from the paged devices's `BD_ADDR`.
Dedicated inquiry access code (DIAC)	Dedicated	68 or 72	This access code is used in the inquiry substate for dedicated inquiry operations.
General inquiry access code (GIAC)	Reserved	68 or 72	This access code is used in the inquiry substate for general inquiry operations.

For DAC, DIAC, and GIAC access code types, the access code length of 72 bits is used only in combination with frequency hopping sequence (FHS) packets. When used as self-contained messages without a header, the DAC, DIAC and GIAC do not include trailer bits and are of length 68 bits.

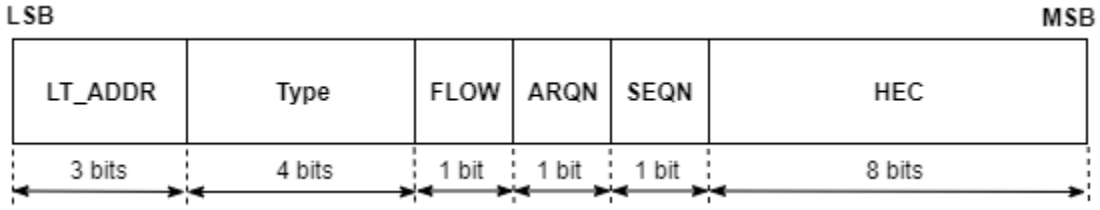
The CAC consists of a preamble, sync word, and trailer.

- Preamble: It is a fixed 4-symbol pattern of 1s and 0s that facilitates DC compensation. If the LSB of the following sync word is `1` or `0`, the preamble sequence is `1010` or `0101` (in transmission order), respectively.
- Sync word: It is a 64-bit code word derived from a 24-bit LAP address. The construction guarantees a large Hamming distance between sync words based on different LAPs. The autocorrelation properties of the sync word improve timing acquisition.
- Trailer: It is appended to the sync word as soon as the packet header follows the access code. The trailer is a fixed 4-symbol pattern of 1s and 0s. The trailer together with the three MSBs of the sync word form a 7-bit pattern of alternating 1s and 0s which is used for extended DC compensation. The trailer sequence is either `1010` or `0101` (in transmission order) depending on whether the MSB of the sync word is `0` or `1`, respectively.

Note For more information about access code in Bluetooth BR/EDR, see volume 2, Part B, Section 6.3 of the Bluetooth Core Specification [2].

Packet Header

The structure of the Bluetooth BR/EDR packet header is shown in this figure.



This table provides a brief description about the packet header fields.

Packet Header Field	Size of the Field (Bits)	Description
Logical transport address (LT_ADDR)	3	This field indicates the destination slave(s) for a packet in a master-to-slave transmission slot and indicates the source slave for a slave-to-master transmission slot.
Type	4	This field specifies the type of packet used. The Bluetooth Core Specification [2] defines 16 different types of BR/EDR packets. The value in this field depends on the value of LT_ADDR field in the packet. This field determines the number of slots occupied by the current packet.
Flow control (FLOW)	1	This field implements the flow control of BR/EDR packets over the asynchronous connection-oriented logical (ACL) transport. When the receive buffer for the ACL logical transport is full, a 'STOP' indication (FLOW = 0) is returned to stop the other device from transmitting data temporarily. When the receive buffer can accept data, a 'GO' indication (FLOW = 1) is returned.
Automatic repeat request number (ARQN)	1	This field informs the source of a successful transfer of payload data with the CRC. This field is reserved for future use on the connectionless slave broadcast (CSB) logical transport.
Sequence number (SEQN)	1	This field provides a sequential numbering scheme to order the data packet stream. This field is reserved for future use on the CSB logical transport.

Packet Header Field	Size of the Field (Bits)	Description
Header error check (HEC)	8	This field checks the packet header integrity. Before generating the HEC, the HEC generator is initialized with an 8-bit value. These 8 bits correspond to the upper address part (UAP). After the initialization, the HEC generator calculates the HEC value for the 10 header bits. Before checking the HEC, the receiver initializes the HEC check circuitry with the appropriate 8-bit UAP. If the HEC does not check the packet header integrity, the entire packet is discarded.

Note For more information about packet header used in Bluetooth BR/EDR, see volume 2, Part B, Section 6.4 of the Bluetooth Core Specification [2].

Packet Types

The packets used in the piconet are related to these logical transports on which they are used.

- Synchronous connection-oriented (SCO): It is a circuit-switched connection that reserved slots between the master and a specific slave.
- Extended SCO (eSCO): Similar to SCO, it reserves slots between the master and a specific slave. eSCO supports a retransmission window following the reserved slots. Together, the reserved slots and the retransmission window form the complete eSCO window.
- ACL: It provides a packet-switched connection between the master and all active slaves participating in the piconet. ACL supports asynchronous and isochronous services. Between a master and a slave, only a single ACL logical transport must exist.
- CSB: It is used to transport profile broadcast data from a master to multiple slaves. A CSB logical transport is unreliable.

This table summarizes the packets defined for the SCO, eSCO, ACL, and CSB logical transport types.

Note The column entries followed by "D" means data field only. "C.1" implies that the MIC value is mandatory when encryption with AES-CCM is enabled. Otherwise, MIC is excluded. For more information about different packet types used in Bluetooth BR/EDR, see volume 2, Part B, Sections 6.5 and 6.7 of the Bluetooth Core Specification [2].

Packet Type	TYPE Code	Slot Occupancy	Payload Header (Bytes)	User Payload (Bytes)	FEC	MIC	CRC	Logical Transport Types Supported
ID	N/A	1	N/A	N/A	N/A	N/A	N/A	N/A
NULL	0000	1	N/A	N/A	N/A	N/A	N/A	SCO, eSCO, ACL, CSB

Packet Type	TYPE Code	Slot Occupancy	Payload Header (Bytes)	User Payload (Bytes)	FEC	MIC	CRC	Logical Transport Types Supported
POLL	0001	1	N/A	N/A	N/A	N/A	N/A	SCO, eSCO, ACL
FHS	0010	1	N/A	18	2/3	N/A	Yes	SCO, ACL
DM1	0011	1	1	0-17	2/3	C.1	Yes	SCO, ACL, CSB
DH1	0100	1	1	0-27	No	C.1	Yes	ACL, CSB
DM3	1010	3	2	0-121	2/3	C.1	Yes	ACL, CSB
DH3	1011	3	2	0-183	No	C.1	Yes	ACL, CSB
DM5	1110	5	2	0-224	2/3	C.1	Yes	ACL, CSB
DH5	1111	5	2	0-339	No	C.1	Yes	ACL, CSB
2-DH1	0100	1	2	0-54	No	C.1	Yes	ACL, CSB
2-DH3	1010	3	2	0-367	No	C.1	Yes	ACL, CSB
2-DH5	1110	5	2	0-679	No	C.1	Yes	ACL, CSB
3-DH1	1000	1	2	0-83	No	C.1	Yes	ACL, CSB
3-DH3	1011	3	2	0-552	No	C.1	Yes	ACL, CSB
3-DH5	1111	5	2	0-1021	No	C.1	Yes	ACL, CSB
HV1	0101	1	N/A	10	1/3	No	No	SCO
HV2	0110	1	N/A	20	2/3	No	No	SCO
HV3	0111	1	N/A	30	No	No	No	SCO
DV	1000	1	1 D	10+(0-9) D	2/3 D	No	Yes D	SCO
EV3	0111	1	N/A	1-30	No	No	Yes	eSCO
EV4	1100	3	N/A	1-120	2/3	No	Yes	eSCO
EV5	1101	3	N/A	1-180	No	No	Yes	eSCO
2-EV3	0110	1	N/A	1-60	No	No	Yes	eSCO
2-EV5	1100	3	N/A	1-360	No	No	Yes	eSCO
3-EV3	0111	1	N/A	1-90	No	No	Yes	eSCO
3-EV5	1101	3	N/A	1-540	No	No	Yes	eSCO

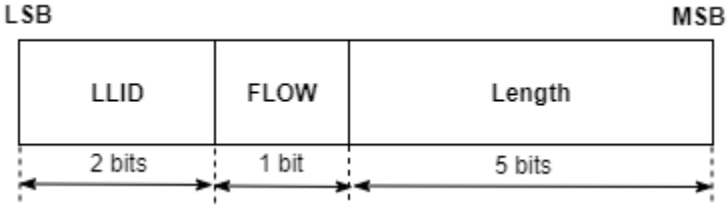
Payload Format

The Buetooth Core Specification [2] defines two types of payload field formats: synchronous data field (for ACL packets) and asynchronous data field (for SCO and eSCO packets). However, the DV packets contain both the synchronous and asynchronous data fields.

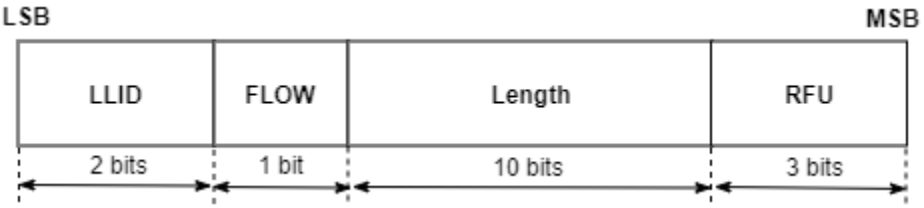
- Synchronous data field: In SCO, which supports only the BR mode, the length of the synchronous data field is fixed. The synchronous data field contains only the synchronous data body portion and does not have a payload header. In BR eSCO, the synchronous data field consists of these two segments: a synchronous data body and a CRC code. In this case, no payload header is present. In

EDR eSCO, the synchronous data field consists of a guard time, synchronization sequence, synchronous data body, CRC code, and trailer. In this case, no payload header is present.

- Asynchronous data field: The BR ACL packets have an asynchronous data field consisting of payload header, payload body, MIC (if applicable), and CRC (if applicable). This figure shows the 8-bit payload header format for BR single-slot ACL packets.



EDR ACL packets have an asynchronous data field consisting of guard time, synchronization sequence, payload header, payload body, MIC (if applicable), CRC (if applicable), and trailer. This figure shows the 16-bit payload header format for EDR multislot ACL packets.



Note For more information about the payload format, see volume 2, Part B, Sections 6.6.1 and 6.6.2 of the Bluetooth Core Specification [2].

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/.

See Also

More About

- “What Is Bluetooth?” on page 13-2
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Location and Direction Finding” on page 13-37
- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Bluetooth Location and Direction Finding

Bluetooth technology [1] uses low-power radio frequency to enable short-range communication at a low cost. The Bluetooth Core Specification [2] provided by the Bluetooth Special Interest Group (SIG) added a location and direction finding feature in the Bluetooth low energy (BLE). The communication in BLE is realized using these two distinct physical layers (PHYs).

- LE Uncoded: This PHY is further segregated into LE 1M PHY and LE 2M PHY. LE 1M is the default PHY and provides a symbol rate of 1 Msym/s. Support for LE 1M is mandatory in all devices that support BLE. LE 2M provides a symbol rate of 2 Msym/s. The support for LE 2M is optional for the devices supporting the BLE controller.
- LE Coded: This PHY is equipped for longer range communication. It has the potential to quadruple the range that can be achieved whilst reducing the data rate. Support for LE Coded PHY is optional for devices supporting the BLE controller.

Bluetooth direction finding can use either LE 1M or LE 2M PHY, but not the LE Coded PHY.

Location and Direction Finding Services in Bluetooth

For several years, Bluetooth has been used to provide different types of location and direction finding services. On a high-level, these services can be split into two categories.

- Proximity solutions: This category consists of point of interest (PoI) information applications (for example, museums that provide the user information about the artefacts in the room). This category also includes item-finding solutions such as Bluetooth tags that help to find lost or misplaced items. In these solutions, the Bluetooth tags periodically transmit BLE broadcast frames. The access point (AP) scans these frames to obtain the Bluetooth tag information and sends it to the location server through the access controller (AC). In PoI proximity applications, determining what point or PoIs are in close proximity of the calculated location is necessary.
- Positioning systems: This category includes location-based services to leverage Bluetooth to find the physical position of the device. The prominent use case examples in this category are real-time locating systems used for asset tracking, people tracking, and indoor positioning systems used to enable pathfinding solutions that help people navigate through intricate indoor scenarios. Indoor positioning use cases need applications that estimate the accurate location of the beacons they encounter so that the location of the tracked device corresponding to the known location of the beacon can be calculated. In some cases, the position of a beacon might need to be determined in three dimensions, considering its x- and y-coordinates in horizontal plane and its elevation above or below some reference altitude. The application can determine the position of its host device only if it knows the direction from which the received signal is coming, the approximate distance to that beacon, and the location of the beacon.

  In applications involving smartphones, when calculating the direction of the signal, the application must consider the orientation in three dimensional space of the phone.

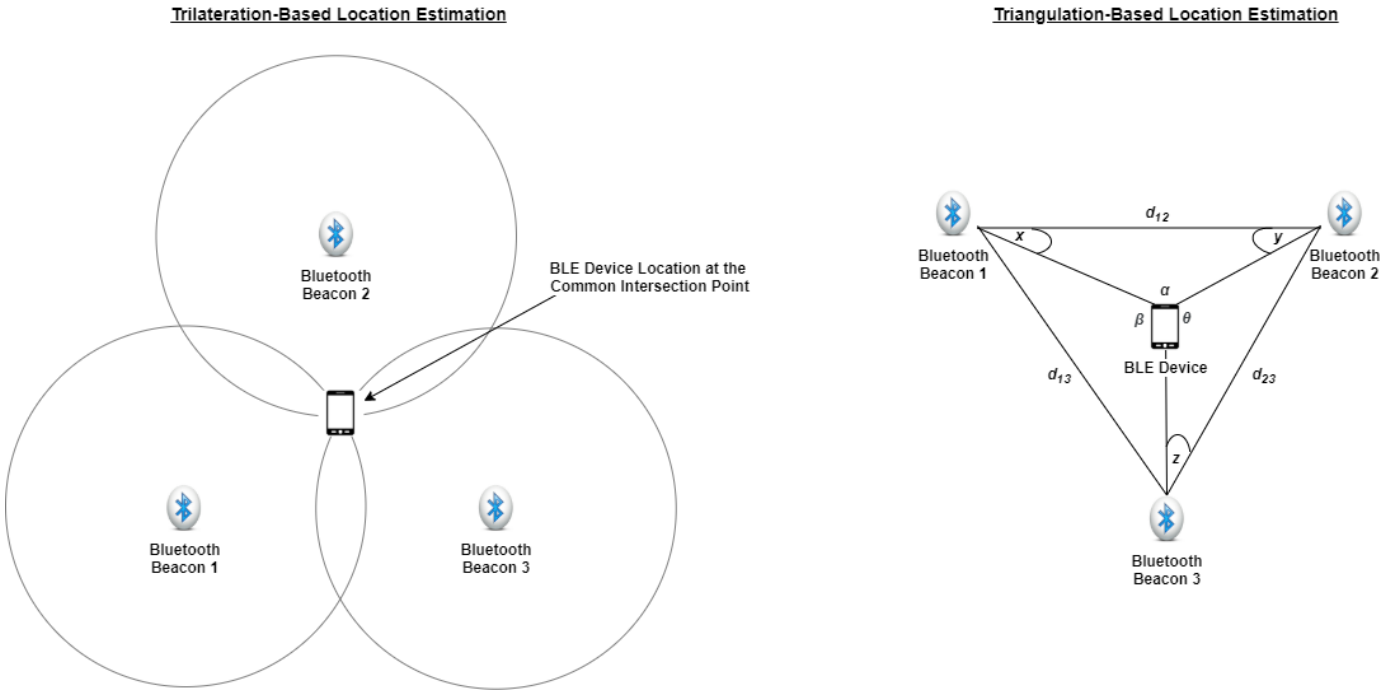
Beyond the previously mentioned location-finding services, the applications themselves must undertake these common considerations.

- Determining antenna array details: To accurately receive and process IQ sample data, applications must have details of the antenna array in the local device (for angle of arrival (AoA)) or remote device (angle of departure (AoD)). Application profiles describe how applications can obtain the antenna array description from remote devices. Expect the APIs to emerge for retrieving details of antenna arrays in local and remote devices.

- Configuring constant tone extension (CTE) parameters: Parameters such as the length of the CTE, the length of antenna switching pattern, and the number of packets that include the CTE to transmit per periodic advertising event govern the CTE production. These parameters can be set through new host controller interface (HCI) commands.
- Configuring and enabling IQ sampling: The Bluetooth Core Specification [2] defines a series of parameters to configure and initiate IQ sampling. These parameters include sample slot duration (either 1 µs or 2 µs), the length of the switching pattern, and the IDs of the antennas to be included in the sampling pattern.
- Developing algorithms and calculating angles from IQ sample data: The Bluetooth SIG does not designate any one particular algorithm as the standard direction-finding algorithm. The choice of algorithm is left to the application layer to address. Generally, this is the area in which manufacturers and developers compete.

Location Estimation Techniques in Bluetooth

Bluetooth beacon technology is an application of the BLE standard. A beacon broadcasts a distinct ID. An application on a BLE device receiving that ID looks into a database to recognize the transmitting beacon and then provides the user with information related to the location of that beacon. This figure shows the techniques used to estimate the distance between the BLE device and the beacon.



- Trilateration-based location estimation: Trilateration is one of the most commonly used technique to estimate the device location. In this technique, the locations of at least two reference Bluetooth beacons and the distance between them must be known. However, to accurately determine the relative location of a node, three beacons are needed. The trilateration technique uses the received signal strength indicator (RSSI) value to compute the distance between the Bluetooth beacons and the BLE device. The RSSI value helps to determine the proximity of two BLE devices by providing meter-level accuracy. The RSSI value indicates the strength of the beacon's signal as seen from the receiving BLE device. As the RSSI value increases, the beacon signal strengthens. This relationship helps indicate when the BLE device is in close proximity of the beacon. Because

the direction of the beacon signal cannot be determined by trilateration, the location of BLE device can be at any point on the circumference of the circle. However, the ideal location of the BLE device is at the common intersection point of the three circles. Due to lack of information related to the direction of the beacon signal, the three circles might not always have a common intersection point.

In trilateration, the advantage of using the RSSI value is that it does not need any additional hardware or incur any additional communication overhead. On the contrary, the accuracy of the RSSI-based approach is impeded by the accuracy of the path-loss model you select. Also, this approach is not accurate enough for several use cases. Even if the reference RSSI value is efficiently calibrated when first installing the Bluetooth beacon, the calculated RSSI value is influenced by the environmental conditions such as the presence of people and humidity levels. The RSSI-based approach gives poor accuracy, particularly in indoor scenarios that are filled with obstacles such as walls and furniture. These obstacles are the source of multipath fading and make the relation between distance and RSSI inaccurate.

- Triangulation-based location estimation: Triangulation is a technique of calculating the position of a point that relies on a known distance between two or three reference points and the angles measured using the Bluetooth direction finding feature between those reference points to that point. These angles can be AoA or AoD. For more information, see “Angle of Arrival (AoA)” on page 13-40 and “Angle of Departure (AoD)” on page 13-40. Unlike trilateration, which implements only the distance measurements, the triangulation technique uses angle measurements. With this technique, you can calculate the location of any point in 2-D given the three angles between the point and other three reference points. However, in 2-D space, a minimum of two angles is required to estimate the location of any point. With reference to the preceding figure, d_{12}, d_{23}, and d_{13} denote the distances between the Bluetooth beacons 1–2, 2–3, and 1–3 respectively. Angles x, y, and z are the known angle measurements between the BLE device and Bluetooth beacons 1, 2, and 3, respectively. Using these known measurements, the triangulation technique enables you to compute angles α, β, and θ. Consequently, the location of the BLE device is obtained. Triangulation is a complex technique that requires information about the location and spatial rotation of the Bluetooth beacons. However, due to AoA and AoD capabilities, triangulation gives a more accurate location of the BLE device as compared to the trilateration technique.

To accurately determine the BLE device location, more advanced solutions must implement multiple Bluetooth beacons and complex algorithms based on trilateration and triangulation techniques.

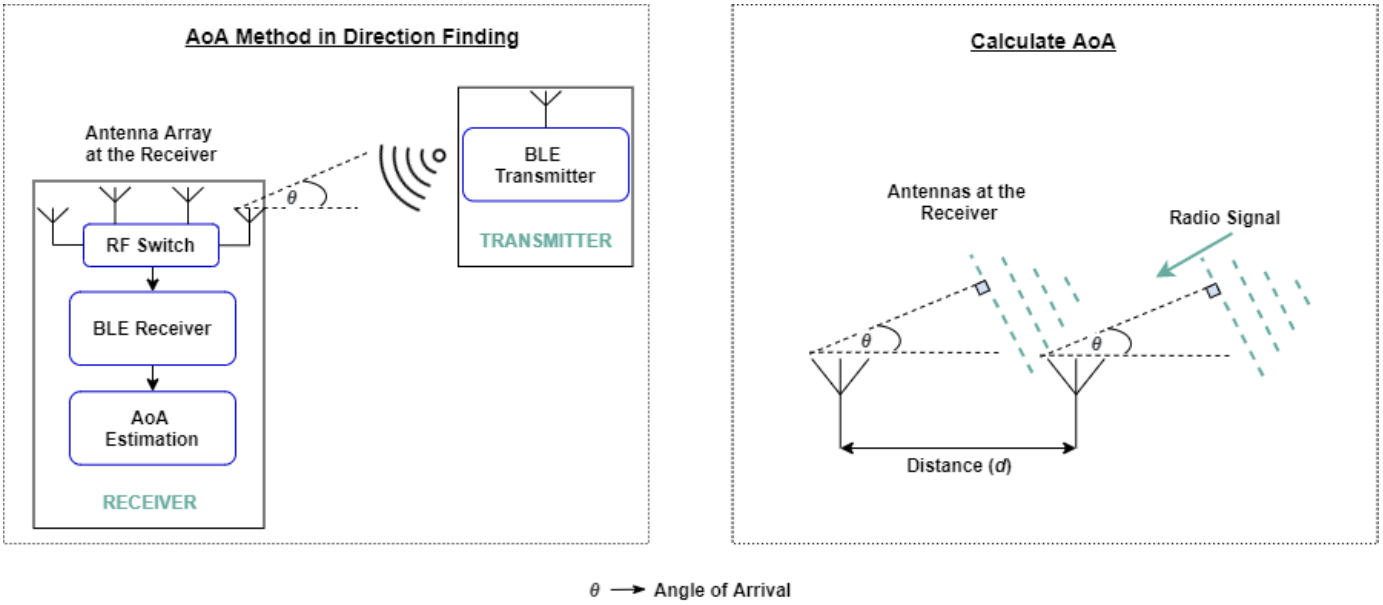
Bluetooth Direction-Finding Capabilities

The Bluetooth Core Specification [2] introduced new features that support high-accuracy direction finding. The controller specification is enhanced so that the specialized hardware that incorporates an antenna array can be used to determine the direction of a received BLE signal. The HCI is modified so that data acquired by the controller can be made available to higher layers of the stack where direction calculations can take place. Bluetooth direction finding offers two distinct methods, each of which exploits the same underlying basis. These direction finding methods are – AoA and AoD.

Note Bluetooth direction-finding capabilities, AoA and AoD, are introduced in the Bluetooth Core Specification 5.1 [2].

Angle of Arrival (AoA)

A BLE device can send its direction-related information to another peer BLE device by transmitting direction-finding enabled packets using a single antenna. The peer BLE device consisting of an RF switch and an antenna array switches antennas and captures the received in-phase (I) and quadrature (Q) samples. The BLE device uses these I and Q samples to compute the phase difference in the radio signal received by various elements of the antenna array. Consequently, the calculated phase difference is used to estimate the AoA. This figure illustrates the concept of the AoA method.



The transmitter device uses a single antenna, whereas the receiver device uses an antenna array handled by the RF switch. At the receiver, d denotes the distance between two antennas. The phase difference, ψ, between the signals arriving at the two antennas is calculated as:

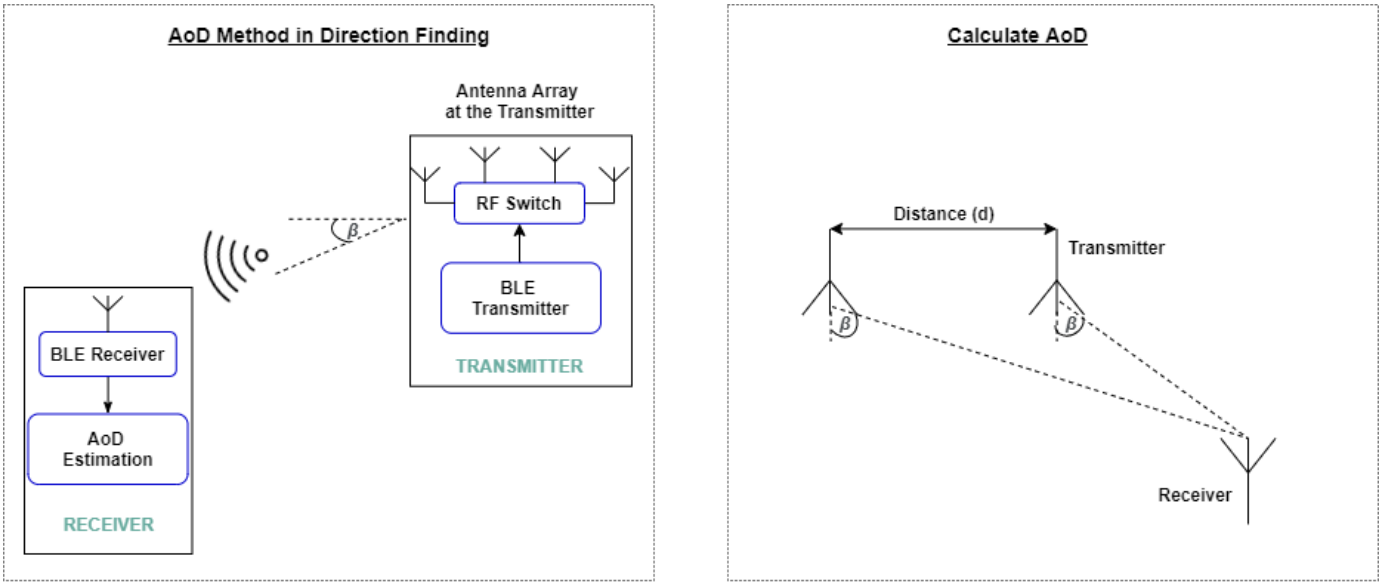
$$\psi = \frac{2\pi d \cos\theta}{\lambda}$$

λ is the signal wavelength and θ is the AoA. To avoid the aliasing effect, the maximum value of d must be λ/2. Rearranging the above equation, the AoA is calculated as:

$$\theta = \cos^{-1}\left(\frac{\psi\lambda}{2\pi d}\right)$$

Angle of Departure (AoD)

Unlike in AoA, the AoD method consists of a single antenna at the receiver and multiple antennas at the transmitter. A BLE transmitter consisting of an RF switch and antenna array can make its AoD detectable by sending direction-finding packets and then switching antennas in the antenna array during the transmission. The BLE receiver receives the packets using a single antenna and captures the I and Q samples. The direction of the signal is determined from different propagation delays of the BLE signal between multiple antennas of the antenna array and the single receiving antenna. This figure illustrates the concept of the AoD method.



The receiver device uses a single antenna, whereas the transmitter device has an antenna array handled by the RF switch. At the transmitter, d denotes the distance between two antennas. The phase difference, ψ, between the signals arriving at the two antennas is calculated as:

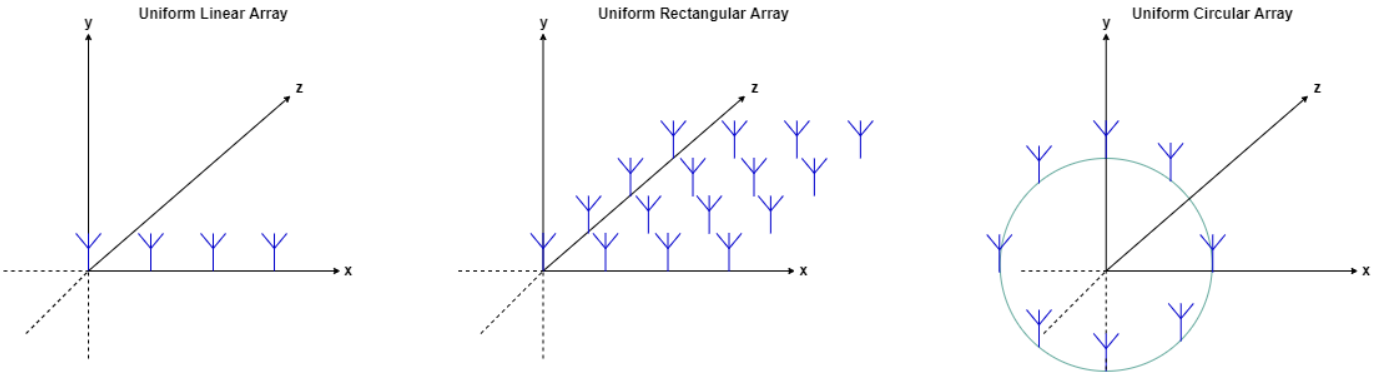
$$\psi = \frac{2\pi d \sin\beta}{\lambda}$$

λ is the signal wavelength and β is the AoD. By rearranging the above equation, AoD is calculated as:

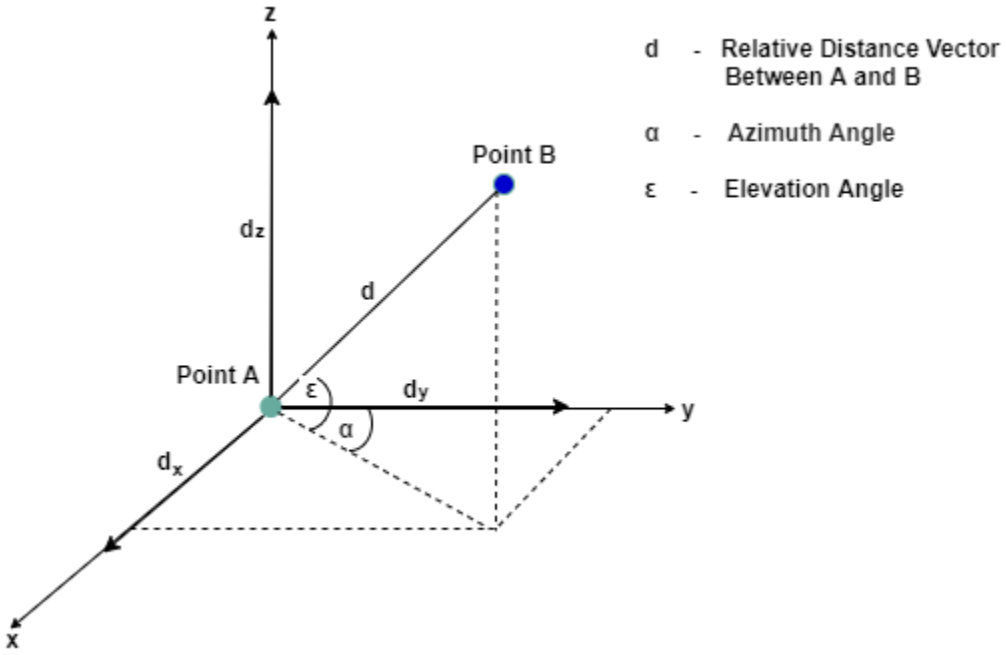
$$\beta = \sin^{-1}\left(\frac{\psi\lambda}{2\pi d}\right)$$

Antenna Arrays

The fundamental use of antenna arrays is to direct a radiated signal toward a desired angular sector. The number, geometrical design, relative amplitudes, and relative phases of the elements of the antenna array depend on the desired angular pattern. Once the antenna array is designed to focus in a specific direction, the array can also be steered in another direction by changing the relative phases of the array elements. This figure shows some commonly used antenna array designs.



In the uniform linear array (ULA) case, antenna elements are located in a single line. In the uniform rectangular array (URA) case, antenna elements are positioned along a rectangular grid. The uniform circular array (UCA) enables antenna elements to be placed along the circumference of the circle. The geometrical designs of ULAs are simple and enable only a single angle to be calculated from a signal. More complex antenna array designs can enable two or three angles to be determined. Calculating the elevation and azimuth angles of the signal relative to a reference plane is common in these antenna arrays. This figure shows the concept of elevation and azimuth angles.



d is the relative distance vector between points A and B. d_x, d_y, and d_z denote the components of d along x-, y-, and z-axis, respectively. Using this information, the azimuth angle (α) and elevation angle (ϵ) between points A and B is calculated as:

$$\alpha = \tan^{-1}(\frac{d_y}{d_x})$$

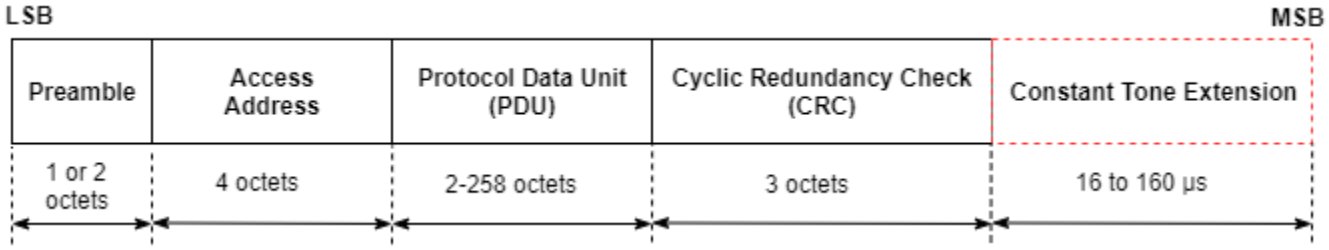
$$\varepsilon = \tan^{-1}(\frac{d_z}{\sqrt{d_x^2 + d_y^2}})$$

Bluetooth Direction-Finding Signals

Bluetooth direction-finding signals are an important part of the Bluetooth direction-finding technique. Direction-finding signals provide a source of constant signal material to which the IQ sampling can be applied. New link layer (LL) protocol data units (PDUs) are identified for direction finding between two connected BLE devices. Moreover, the Bluetooth Core Specification [2] enables you to use existing advertising PDUs for connectionless direction-finding purposes. In these cases, additional information known as CTE is appended to the PDUs. To calculate AoA and AoD, the Bluetooth direction-finding signals use these BLE packet structure fields.

Constant Tone Extension (CTE)

This figure shows the packet structure for the BLE uncoded PHY operating on LE 1M and LE 2M. The CTE field is appended at the end of the packet structure.



CTE contains a series of symbols, each representing a binary 1. The number of symbols in the CTE field are configured by the higher layers so that a suitable amount of data and time is available for IQ sampling.

Note For more information about the CTE, see volume 6, Part B, Section 2.5.1 of the Bluetooth Core Specification [2].

Frequency Deviation

In a given radio channel, Bluetooth uses two frequencies, one to denote digital 0s and the other to denote digital 1s. These two frequencies are computed by adding or subtracting the *frequency deviation* to or from the center frequency of the channel. Any change in the frequency also changes the wavelength. The wavelength is an important factor in calculating direction from IQ samples. Therefore, CTE consists solely of digital 1s. This implies that the entire CTE is transmitted at a single frequency and has a constant wavelength.

Cyclic Redundancy Check (CRC)

Each BLE packet contains a CRC field that is used in error detection. The BLE transmitter calculates a CRC value from the remainder of the packet to be transmitted, appends the CRC to the end of the packet, and transmits the packet. The BLE receiver performs the same calculation and compares the computed CRC value with the appended CRC value. If the CRC values are unequal, a communication error has occurred. This causes a change in one or more of the transmitted bits. In this case, the packet is ignored by the BLE receiver and can be retransmitted by the BLE transmitter.

Note The CTE value in the direction-finding packets is not included in the CRC calculation.

Message Integrity Check (MIC)

If a connection between the BLE transmitter and receiver is encrypted and authenticated, the LL PDU includes a MIC field. The MIC value is used to authenticate the sender of the PDU.

Note The CTE value in the direction-finding packets is not included in the MIC calculation.

Whitening

Whitening refers to the process of scrambling the bits to avoid lengthy sequences of 1s and 0s in the transmitted bit stream. The lengthy sequences of 1s and 0s might cause the receiver to lose its frequency lock and act as though the center frequency has moved up or down. BLE uses whitening to scramble the PDU and CRC fields of all LL packets.

Note The CTE value in the direction-finding packets is not subject to the whitening process.

Connectionless and Connection-Oriented Direction Finding

The Bluetooth Core Specification [2] enables the AoA and AoD to be used in either connectionless or connection-oriented communication. However, in typical use cases, the AoD is used with connectionless communication and the AoA is used with connection-oriented communication. This table shows four possible permutations of using the AoA and AoD with connectionless and connection-oriented communication.

Type of Connection	AoA	AoD
Connectionless	BLE controller support is optional.	BLE controller support is optional. Using the AoD with connectionless communication is typical.
Connection-oriented	BLE controller support is optional. Using the AoA with connection-oriented communication is typical.	BLE controller support is optional.

Connectionless direction finding implements Bluetooth periodic advertising. The CTE is appended to otherwise standard AUX_SYNC_IND PDUs. Connection-oriented direction finding conveys the CTE using new LL_CTE_RSP packets that are transmitted over the connection as an acknowledgment to LL_CTE_REQ PDUs. In both of these cases, a variety of setup and configuration steps must be completed before IQ sampling is initiated and the CTE-bearing packets are generated.

With the Bluetooth-direction finding capability, the proximity and positioning systems operating at submeter accuracy can be developed for use cases such as indoor positioning, path finding, asset tracking, and directional discovery. The Bluetooth direction-finding capability elevates proven engineering techniques for signal direction. This capability also standardizes the interfaces, interactions, and prominent intrinsic operations of the BLE stack. Precise direction finding is now interoperable across different manufacturers and can be widely adopted to create a new generation of advanced Bluetooth location and direction finding services.

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] Suryavanshi, Nitesh B., K. Viswavardhan Reddy, and Vishnu R. Chandrika. “Direction Finding Capability in Bluetooth 5.1 Standard.” *In Ubiquitous Communications and Network*

Computing, edited by Navin Kumar and R. Venkatesha Prasad, 53–65. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Cham: Springer International Publishing, 2019.

See Also

Functions
`bleAngleEstimate` | `bleWaveformGenerator` | `bleIdealReceiver`

Objects
`bleAngleEstimateConfig`

More About

- “What Is Bluetooth?” on page 13-2
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23

Bluetooth Mesh Networking

Bluetooth technology [1] uses low-power radio frequency to enable short-range communication at a low cost. In 2010, Bluetooth 4.0 introduced a new variant known as Bluetooth low energy (BLE), or Bluetooth Smart. BLE supports a point-to-multipoint or broadcast communication that is useful for a short-range navigation beacon mode such as real-time locating systems used for asset-tracking and people tracking. Auxiliary improvements to Bluetooth technology were released with the introduction of Bluetooth 5.0 [2]. In addition to Bluetooth 5.0, the Bluetooth Special Interest Group (SIG) defined a new connectivity model for BLE, known as the Mesh Profile [4]. The BLE Mesh Profile establishes the option of many-to-many communication links for BLE devices and is optimized for creating large scale Internet of Things (IoT) networks. The mesh stack defined by the BLE Mesh Profile is located on top of the BLE core specification. For more information about the mesh stack, see “Bluetooth Mesh Stack” on page 13-46. This new mesh networking capability of Bluetooth is ideally suited for building automation, large scale sensor networks, and other IoT solutions that require tens and hundreds of devices to be reliably and securely set up.

Motivation for Bluetooth Mesh Networking

As mesh networking topologies offer the best way to satisfy different increasingly common and popular communications requirements, Bluetooth mesh networking was introduced. Some of the fundamental requirements include:

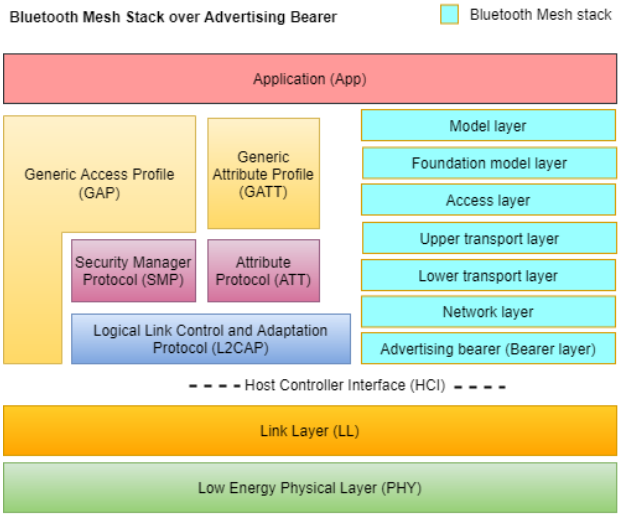
- Low energy consumption
- Extension of the coverage area through multihop communication
- Increased network scalability through efficient use of radio resources
- Interoperability with different standards
- Increased communications security through authentication and encryption
- Improved system reliability through multipath message-oriented communication
- Ability to deliver optimal network responsiveness

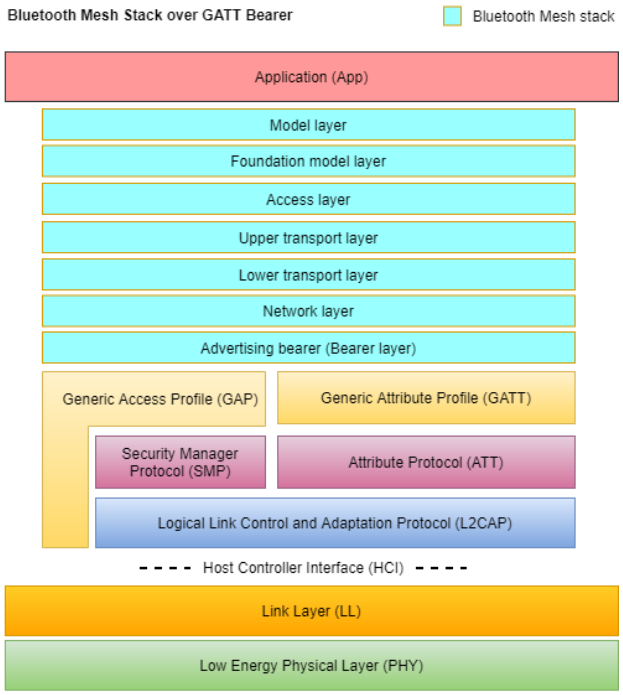
Other low-power wireless communication technologies, such as ZigBee® and Thread, also support mesh networking topologies. However, these technologies often face issues such as low data rates, restricted number of *hops* when relaying data across the mesh, limitations in scalability often caused by the way radio channels are used, and delays when following procedures to change the device composition of the mesh topology. Additionally, these wireless communication technologies are not supported by standard smartphones, tablets, and PCs. The Bluetooth mesh meets the previously mentioned requirements and creates an industry-standard mesh communications technology based on BLE.

Bluetooth mesh networking integrates the trusted, global interoperability with an evolved, trusted ecosystem to create industrial-grade device networks.

Bluetooth Mesh Stack

This figure illustrates how the Bluetooth mesh stack fits into the standard BLE protocol stack. The figure shows the Bluetooth mesh stack over the BLE advertising bearer and generic attribute (GATT) bearer.





The Bluetooth mesh stack consists of these layers:

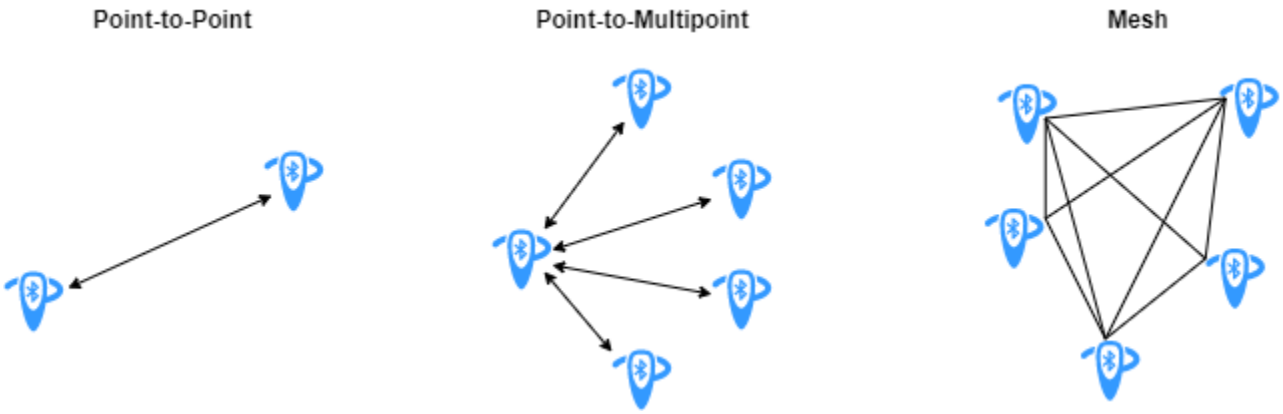
- Model layer: This layer defines the models, messages, and states required for use-case scenarios. For example, to change the state of a light to On or Off, the Bluetooth nodes use the Generic OnOff Set message from the Generic OnOff model.
- Foundation model layer: This layer defines the models, messages, and states required to configure and manage the mesh network. This layer also configures element, publish, and subscription addresses.
- Access layer: This layer defines the interface to the upper transport layer and the format of the application data. It also controls the encryption and decryption of the application data in the upper transport layer.
- Upper transport layer: This layer defines functionalities such as encryption, decryption, and authentication of the application data and is designed to provide confidentiality of access messages. This layer is also responsible for generating transport control messages (*Friendship* and heartbeat) internally and transmits those messages to a peer upper transport layer. The network layer encrypts and authenticates these messages.
- Lower transport layer: This layer defines functionalities such as segmentation and reassembly of upper transport layer messages into multiple lower transport layer messages to deliver large upper transport layer messages to other nodes. This layer also defines the friend queue used by the Friend node to store the lower transport layer messages for a Low Power node.
- Network layer: This layer defines functionalities such as encryption, decryption, and authentication of the lower transport layer messages. This layer transmits the lower transport layer messages over the bearer layer and relays the mesh messages when the *Relay* feature is enabled. The network layer also defines the message cache containing all recently seen network messages. If the received message is found to be in the cache, then it is discarded.
- Bearer layer: This layer defines the interface between the Bluetooth mesh stack and the BLE protocol stack. This layer is also responsible for creating a mesh network by provisioning the mesh

nodes. The two types of bearers supported by the Bluetooth mesh are advertising bearer and GATT bearer.

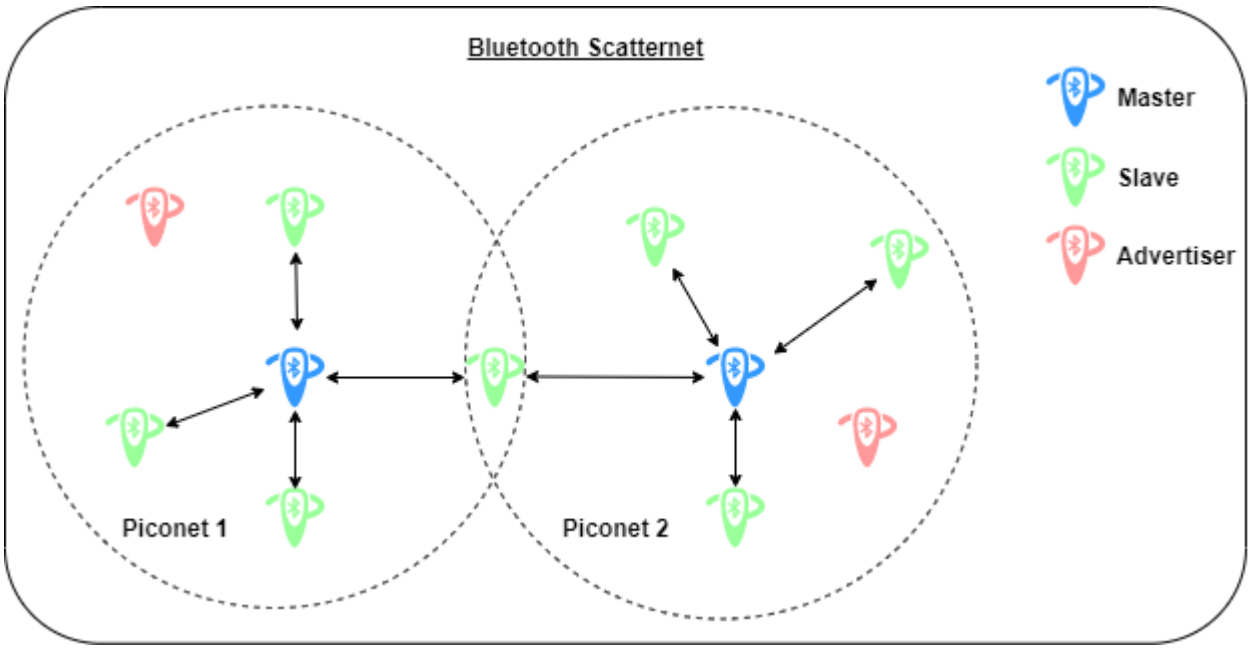
BLE is the wireless communications protocol stack of which the Bluetooth mesh makes use. For more information about BLE protocol stack, see “BLE Protocol Stack” on page 13-7.

Bluetooth Connection Topologies

Most BLE devices communicate with each other using a simple point-to-point (one-to-one communication) or point-to-multipoint (one-to-many communication) topology as shown in this figure.



Devices using one-to-one communication operate in a Bluetooth *piconet*. As shown in this figure, each piconet consists of a device in the role of *Master* (M), with other devices in the *Slave* (S) or *Advertiser* roles. Before joining the piconet, each S node is in an advertiser role. Multiple piconets connect to each other, forming a Bluetooth *scatternet*.



For example, a smartphone with an established one-to-one connection to a heart rate monitor over which it can transfer data is an example of point-to-point connection.

On the contrary, the Bluetooth mesh enables you to set up many-to-many communication links between Bluetooth devices. In a Bluetooth mesh, devices can relay data to remote devices that are not in the direct communication range of the source device. This enables a Bluetooth mesh network to extend its radio range and encompass a large geographical area containing a large number of devices. Another advantage of the Bluetooth mesh over point-to-point and point-to-multipoint topologies is the capability of self healing. The self-healing capability of the Bluetooth mesh implies that the network does not have any single point of failure. If a Bluetooth device disconnects from the mesh network, other devices can still send and receive messages from each other, which keeps the network functioning.

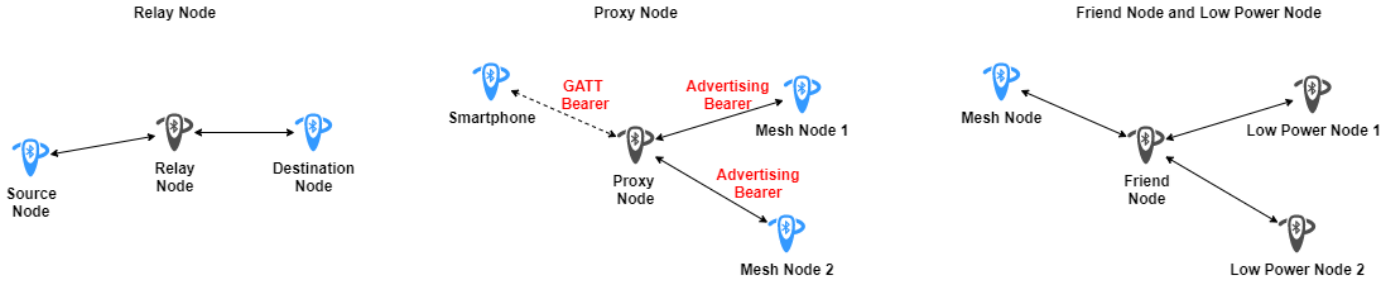
Fundamentals of Bluetooth Mesh Networking

These concepts of Bluetooth mesh networking serve as the foundation to study the functionality of a Bluetooth mesh network.

Devices and Nodes

Devices that belong to a Bluetooth mesh network are known as *nodes*. Devices that are not a part of a Bluetooth mesh network are known as *unprovisioned devices*. The process of transforming an unprovisioned Bluetooth device into a node is called “Provisioning” on page 13-53. Each node can send and receive messages either directly or through relaying from node to node. This figure shows a web of Bluetooth nodes spread across the MathWorks, Natick office.

Each Bluetooth mesh node might possess some optional features enabling them to acquire additional, special capabilities. These features include the *Relay*, *Proxy*, *Friend*, and the *Low Power* features. The Bluetooth mesh nodes possessing these features are known as Relay nodes, Proxy nodes, Friend nodes, and Low Power nodes (LPNs), respectively.

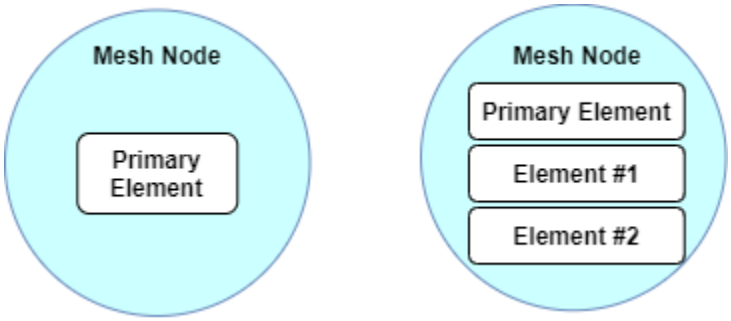


- Relay nodes: Bluetooth mesh nodes that possess the Relay feature are known as Relay nodes. These nodes use the relaying mechanism to retransmit the received messages through multiple hops. Depending on the power source and computational capacity, a mesh node might become a Relay node.
- Proxy nodes: To enable communication between a BLE device that do not possess a Bluetooth mesh stack and the nodes in the mesh network, proxy nodes can be used. A Proxy node acts as an intermediary and utilizes the proxy protocol with generic attribute profile (GATT) operations. For example, as shown in the preceding figure, a smartphone that does not support the Bluetooth mesh stack interacts with mesh nodes by a Proxy node through GATT operations.
- Friend node: Bluetooth mesh nodes that do not have any power constraints are good exemplars for being Friend nodes. LPNs work in collaboration with Friend nodes. A Friend node stores messages destined to an LPN and delivers the messages to the LPN whenever the LPN polls the Friend node for the *waiting messages*. The relationship between an LPN and a Friend node is called “Friendship” on page 13-54.
- Low Power node: Bluetooth mesh nodes that are power constrained can use the Low Power feature to minimize the On time of the radio and conserve energy. Such nodes are known as LPNs. LPNs are predominantly concerned with sending messages but have a need to occasionally receive messages. For example, a temperature monitoring sensor that is powered by a small coin cell battery sends a temperature reading once per minute whenever the temperature is above or below the configured threshold values. If the temperature stays within the thresholds, the LPN sends no message.

Note For more information about Bluetooth mesh features, see sections 3.4.6, 3.6.6.3, 3.6.6.4, and 7.2 of the Bluetooth Mesh Profile [4].

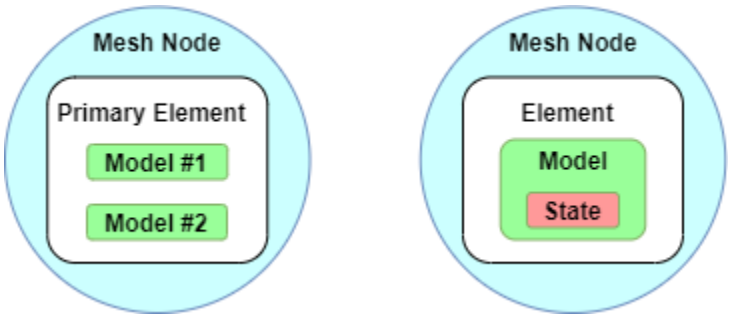
Elements, Models, and States

Some Bluetooth mesh nodes possess one or more independent constituent parts known as *elements*. This figure shows that a mesh node must have at least one element (primary element) but can have multiple elements.



Elements consists of entities that define the functionality of a node and the condition of the element. Each element in a mesh node has a unique *unicast address* that enables each element to be addressed.

This figure shows the mesh node and its constituents. The basic functionality of a mesh node is defined and implemented by models. Models reside inside elements, and each element must have at least one model.



Bluetooth Core Specifications [2] defines these three categories of models.

- Server model: This model defines a collection of states, state transitions, state bindings, and messages that the element containing the model can send or receive. It also defines behaviors pertaining to messages, states, and state transitions.
- Client model: This model defines the messages that it can send or receive in order to acquire values of multiple states defined in the corresponding server model.
- Control model: This model comprises of a server model and client model. The server model enables communication with other client models, and the client model enables communication with server models.

A *state* is a value of a certain type that defines the condition of elements. Additionally, states also have associated behaviors. For example, consider a simple light bulb that can be either On or Off. The Bluetooth mesh defines a state called generic *OnOff*. When the light bulb acquires this state and a value of *On*, the bulb is illuminated. Similarly, when the light bulb acquires the generic *OnOff* state and a value of *Off*, the bulb is switched off.

Note For more information about Bluetooth mesh models and states, see section 4 of the Bluetooth Mesh Profile [4].

Addresses and Messages

Bluetooth Core Specifications [2] defines these four types of addresses.

- Unassigned address: Nonconfigured elements or elements without any designated addresses have an unassigned address. Mesh nodes with unassigned addresses are not involved in messaging.
- Unicast address: During provisioning, a provisioner assigns a unicast address to each element in a node. Unicast addresses can appear in the source address field of a message, the destination address field of a message, or both. Messages sent to unicast addresses are processed by only one element. For more information about provisioning, see “Provisioning” on page 13-53.
- Virtual address: A virtual address represents a set of destination addresses. Each virtual address logically represents a 128-bit label universally unique identifier (UUID). The Bluetooth nodes can publish or subscribe to these addresses.
- Group address: Group addresses are types of multicast addresses that represent multiple elements from one or more nodes. Group addresses can be fixed (allocated by Bluetooth SIG) or dynamically assigned.

Communication in Bluetooth mesh networks is realized through messages. A message can be a control message or an access message.

- Control message: These messages are involved in the actual functioning of the Bluetooth mesh network. For example, heartbeat and friend request messages are types of control messages.
- Access message: These messages enable client models to retrieve or set the values of states in server models. Access messages can be acknowledged or unacknowledged. Acknowledged messages are transmitted to each receiving element. The receiving element acknowledges the messages by sending a status message. No response is sent to an unacknowledged message. Bluetooth mesh network status messages are an example of unacknowledged messages.

For every state, the server model supports a set of messages. For example, these message can include a client model requesting the value of a state or requesting to change a state and a server model sending messages about the states or a change in the state.

Messages are identified by opcodes and have associated parameters. A unique opcode defines these three types of mesh messages:

- GET message: This mesh message requests the state value from one or more nodes.
- SET message: This mesh message changes the value of a given state.
- STATUS message: This mesh message is used is these scenarios.
  - In response to a GET message containing the state value
  - In response to an unacknowledged SET message
  - Sent independently to report the status of an element

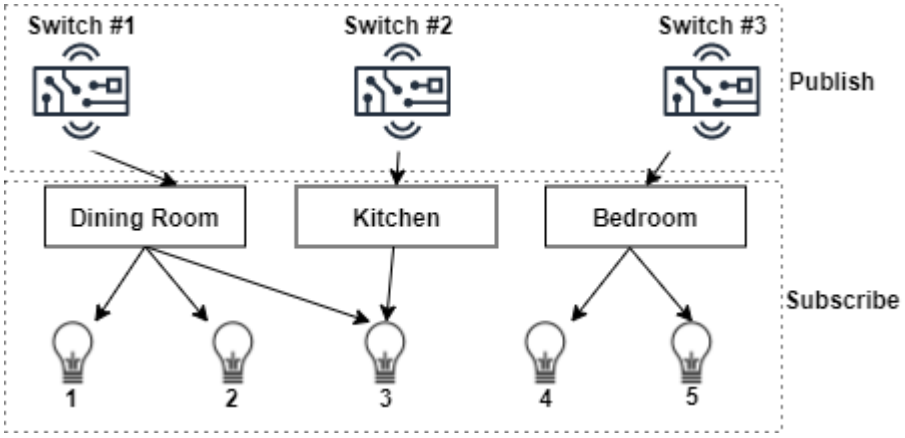
Note For more information about Bluetooth mesh addresses, see section 3.4.2 of the Bluetooth Mesh Profile [4].

Publish/Subscribe Message-Oriented Communication System

Bluetooth mesh networking implements a publish/subscribe message-oriented communication system. Such an approach ensures that different types of products can coexist in a mesh network

without being affected by messages from devices they do not need to listen to. The act of sending a message is known as *publishing*. Based on the configuration, the mesh nodes select messages sent to specific addresses for processing. This technique is known as *subscribing*. A publisher node sends messages to those nodes that have subscribed to the publisher. Typically, mesh messages are addressed to group or virtual addresses.

Consider the example shown in this figure. Each room can subscribe to messages from the specific light bulbs for that room. Additionally, these messages can be unicast, multicast, broadcast, or any combination of these three options.



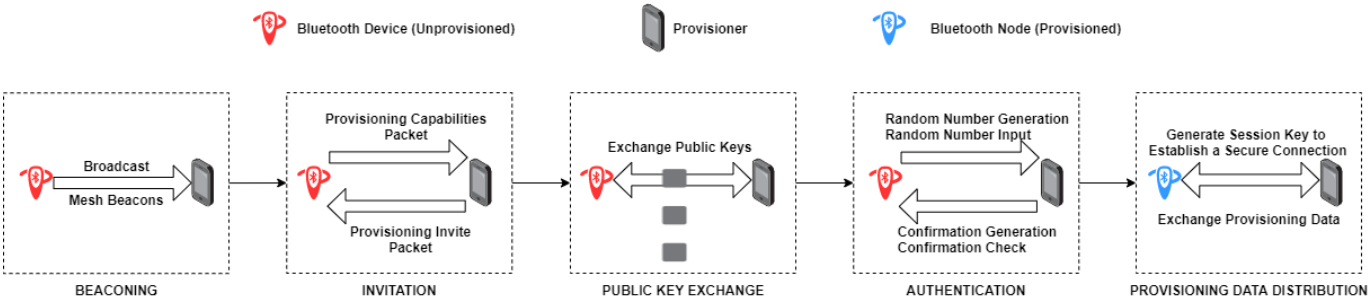
Switch #1 publishes to the group address Dinning Room. Light bulbs 1, 2, and 3 each subscribe to the Dinning Room address and therefore process messages published to this address. Switch #2 publishes to the group address Kitchen. Light bulb 3 subscribes to the Kitchen address and is the only bulb controlled by Switch #2. Similarly, Switch #3 publishes to the group address Bedroom and hence controls light bulbs 4 and 5. This example also shows that the mesh nodes can subscribe to messages addressed to more than one unique address.

The group and virtual addresses used in the publish/subscribe communication system enables removing, replacing, or adding new nodes to the mesh network without any reconfiguration. For example, an additional light bulb can be added in Kitchen using the “Provisioning” on page 13-53 process and then configured to subscribe to the Kitchen group address. In this process, no other light bulbs are impacted.

Note For more information about the Bluetooth mesh publish/subscribe communication, see section 2.3.8 of the Bluetooth Mesh Model Specification [5].

Provisioning

Provisioning is the process by which a Bluetooth device (unprovisioned device) joins the mesh network and becomes a Bluetooth node. This process is controlled by a *provisioner*. A provisioner and the unprovisioned device follow a fixed procedure as defined in the Bluetooth Mesh Profile [4]. A provisioner is typically a smartphone running a provisioning application. The process of provisioning can be accomplished by one or more provisioners. This figure shows the five steps of provisioning.

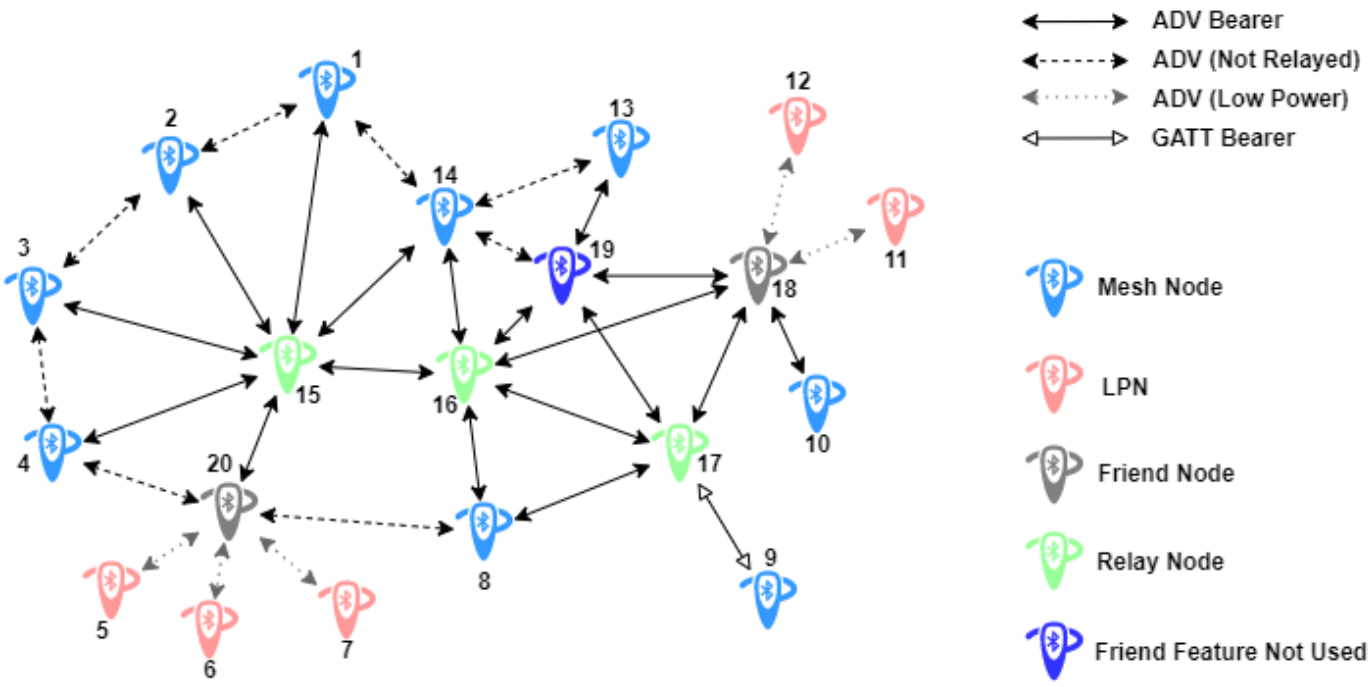


1. Beaconing: In this step, the unprovisioned Bluetooth device advertises its availability to be provisioned by sending the *mesh beacon* advertisements in the advertisement packets. A typical way to trigger beaconing is through a specified sequence of button clicks on the unprovisioned Bluetooth device.
2. Invitation: In this step, the provisioner invites the unprovisioned Bluetooth device for provisioning by sending a *provisioning invite protocol data unit* (PDU). The unprovisioned Bluetooth device responds with information about its capabilities by sending a *provisioning capabilities PDU*.
3. Public key exchange: In this step, the provisioner and the unprovisioned device exchange their public keys. These public keys can be static or ephemeral, either directly or using an out-of-band (OOB) method.
4. Authentication: In this step, the unprovisioned device outputs a random, single or multidigit number to the user in some form, using an action appropriate to its capabilities. The authentication method depends on the capabilities of both devices used. Irrespective of the authentication method that the Bluetooth node uses, the authentication also includes a confirmation value generation step and a confirmation check step.
5. Provisioning data distribution: After successfully completing the authentication step, the provisioner and the unprovisioned device generate a sesion key by using their private keys and the exchanged peer public keys. The provisioner and the unprovisioned device use the session key to secure the subsequent exchange of data needed to complete the provisioning process. This process includes the distribution of a security key called the *network key* (NetKey). After provisioning is completed, the provisioned device acquires the NetKey, a mesh security parameter called *IV Index*, and a unicast address assigned by the provisioner. At this point, the Bluetooth device can be termed as a Bluetooth node.

Note For more information about the Bluetooth mesh provisioning, see section 5 of the Bluetooth Mesh Profile [4].

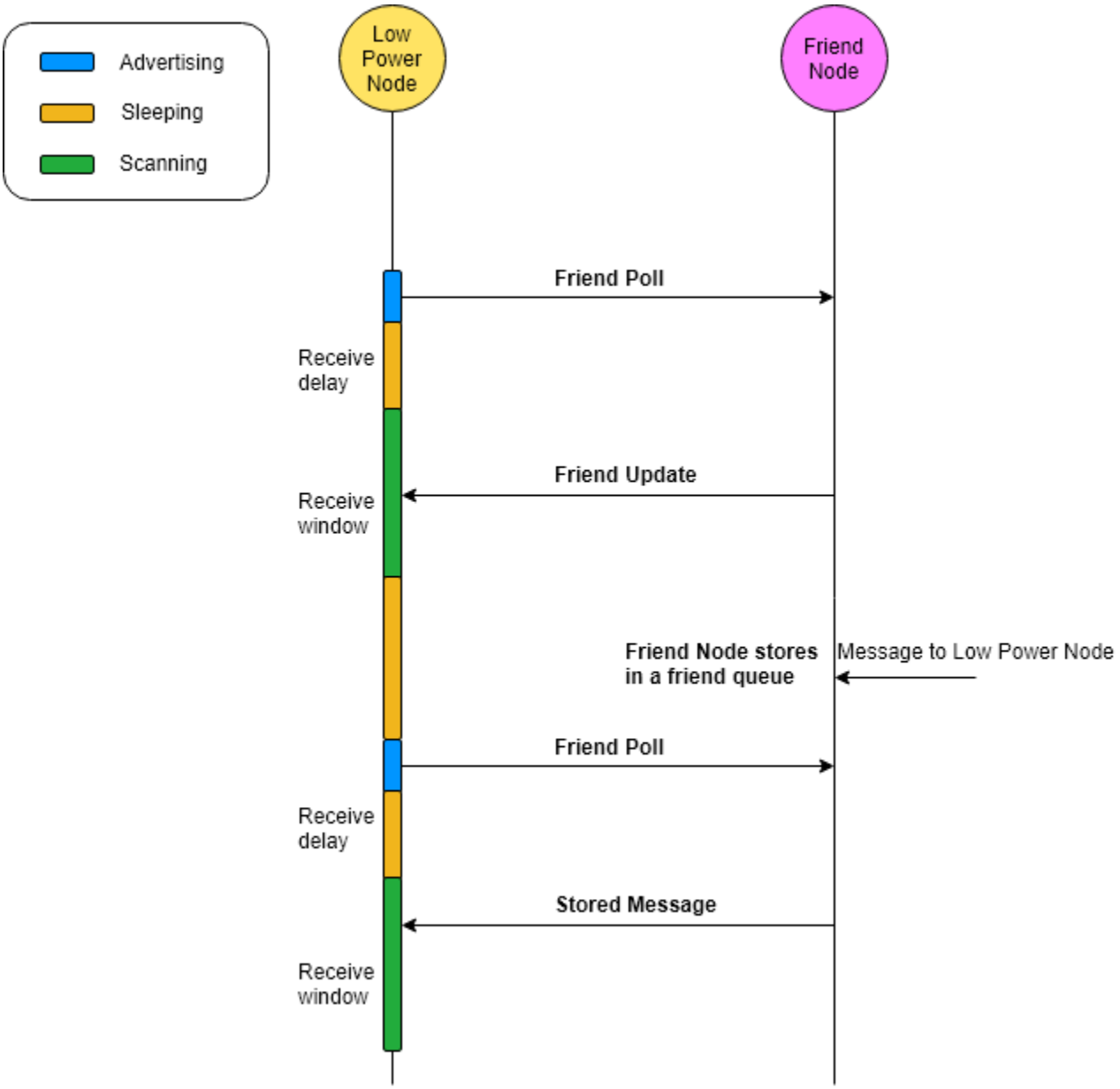
Friendship

To reduce the duty cycles of the LPN and conserve energy, the LPN must establish a *Friendship* with a mesh node supporting the Friend feature. This figure from [4] shows the relationship between LPNs and Friend nodes.



LPNs 5, 6, and 7 have a Friendship relationship with Friend node 20. Friend node 18 has Friendship with LPNs 11 and 12. Subsequently, Friend node 20 stores and forwards messages addressed to LPNs 5, 6, and 7. Similarly, Friend node 18 stores and forwards messages addressed to LPNs 11 and 12. Forwarding by the Friend node occurs only when the LPN wakes up and polls the Friend node for messages awaiting delivery. This mechanism enables all of the LPNs to conserve energy and operate for longer durations.

This figure shows the Bluetooth mesh messages exchanged between an LPN and a Friend node to establish Friendship.



The Bluetooth nodes use these timing parameters to establish Friendship:

- Receive delay: This parameter specifies the time between when an LPN sends a request and listens for a response from the Friend node. The LPN is in sleep state for the complete duration of the receive delay.
- Receive window: This parameter specifies the time for which an LPN listens for a response from a Friend node. The LPN is in the scanning state for the complete duration of the receive window.
- Poll timeout: This parameter specifies the maximum time between two successive requests from an LPN. Within the poll timeout, if the Friend node or the LPN fails to a receive request or response from the other node, the Friendship is terminated.

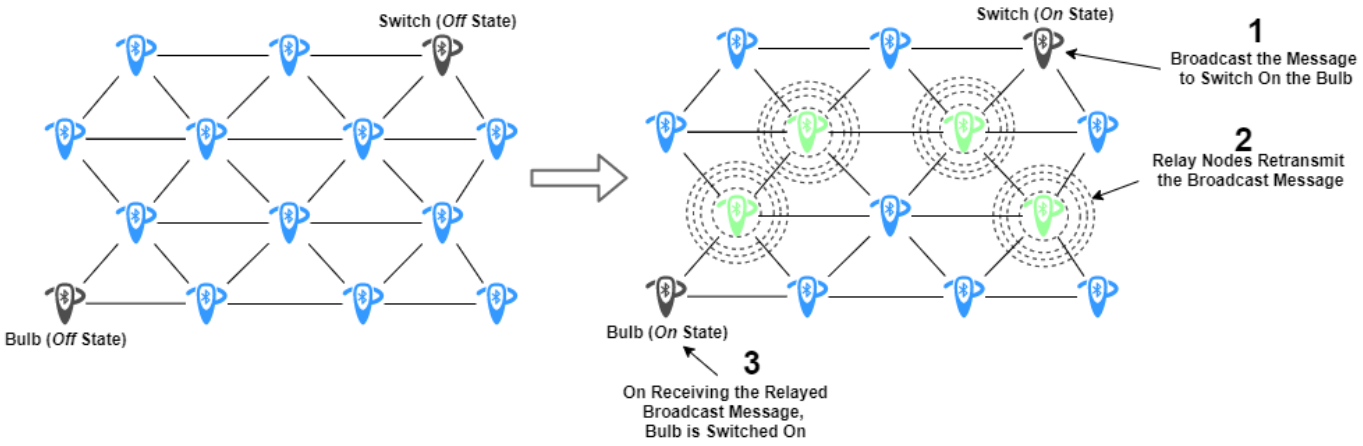
  Periodically, LPNs poll Friend nodes for any data messages stored in the friend queue. After polling the Friend node, the LPN enters a sleep state for the duration of receive delay. The Friend

node uses the receive delay to prepare the response for the LPN. After the receive delay, the Friend node responds to the LPN before the sum of the receive delay and the receive window. For more information about Friendship, see “Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks” on page 3-136 example.

Note For more information about Friendship, see section 3.6.6 of the Bluetooth Mesh Profile [4].

Managed Flooding

Many mesh networks implement routing mechanisms to relay messages in the network. Another mechanism to relay messages is to flood the network with messages being relayed without any consideration of the optimal routes to reach their respective destinations. Bluetooth mesh networking uses an approach known as *managed flooding* that comprises of both of these mechanisms. Bluetooth mesh networking leverages the strengths of the flooding approach and optimizes its operations such that it is both reliable and efficient. This figure demonstrates the process of managed flooding in the Bluetooth mesh.



The figure illustrates communication between a switch and a connected light bulb in a Bluetooth mesh. Initially, the switch and bulb are in the *Off* state. Changing the switch to the *On* state broadcasts a message to turn on the bulb. All of the mesh nodes in range of the switch hear the message, but only the relay nodes retransmit the message. The message is relayed in this manner across the network until it reaches the bulb and turns on the bulb. This process is termed as managed flooding. To optimize this process, Bluetooth mesh implements these measures.

- Heartbeat messages: These messages are transmitted by mesh nodes periodically to indicate to other mesh nodes that the node sending the heartbeat is still active. Heartbeat messages contain information that enables the receiving nodes to determine the number of hops between it and the sending node.
- Time to live (TTL): This field is present in all Bluetooth mesh PDUs. It manages the maximum number of hops over which a message is relayed. Setting the TTL value enables mesh nodes to have control over relaying and to conserve energy. Heartbeat messages enable nodes to determine the optimal TTL value required for each published message.
- Message cache: Every mesh node contains a message cache to determine whether it has seen a message before. If the node has seen the message before, the node discards the message and avoids unnecessary processing higher up the stack.

- Friendship: For more information about the Friendship mechanism, see “Friendship” on page 13-54.

For more information about managed flooding in Bluetooth mesh networks, refer “Bluetooth Mesh Flooding in Wireless Sensor Networks” on page 3-159 example.

Applications of Bluetooth Mesh Networking

The addition of mesh capabilities to Bluetooth creates opportunities for applying Bluetooth mesh networking in automation and IoT domains. These are some of the prominent applications of Bluetooth mesh networking.

- Smart home automation — Bluetooth mesh networking can be used to simplify the smart home automation processes by enabling a mesh network of devices (such as smart bulbs, thermostats, and vents) readily established and provisioned with the user's smartphone. A Bluetooth mesh network of such connected devices can be used to relay messages through multiple paths, thus increasing the communication reliability and network scalability. Bluetooth mesh does not have any single point of failure. This prevents service outages if a mesh node fails. For example, consider a home scenario with a Bluetooth mesh network of all lighting devices. If some of the lighting devices in the mesh network fail, the messages from the rest of the mesh can still reach the user’s control device.
- Beaconing — One of the prominent use case of Bluetooth mesh networking is the *beaconing*. In beaconing, an external event triggers a mesh node to transmit data. This data can include sensor information, location information, or point-of-interest information. Any mesh node can integrate one or more beacon standards (such as iBeacon of Apple or EddyStone from Google) and can be transformed into a virtual Bluetooth beacon while operating as a Bluetooth mesh node. This approach can enable new use-case scenarios, such as indoor positioning, asset tracking, and point-of-interest information delivery. Use cases involving Bluetooth direction finding (introduced in Bluetooth Core Specifications 5.1 [3]) implement beaconing to support high-accuracy direction finding. Bluetooth mesh combined with direction-finding features such as angle of arrival and angle of departure can pave way for many commercial IoT-based use case scenarios. For more information about beaconing and Bluetooth direction-finding capabilities, refer “Bluetooth Location and Direction Finding” on page 13-37 topic.
- Automated irrigation systems and plant lighting — Bluetooth mesh networks can be used to develop intelligent solutions in automated irrigation systems and plant lighting. For example, as land resources decline, many plants are grown in greenhouses for higher harvesting yields. Automated indoor planting needs a combination of appropriate light source with a smart control system and a Bluetooth mesh network. In this scenario, Bluetooth mesh modules are placed into the light sources. This mechanism enables the mesh modules to automate the control of the light source, soil moisture, air temperature, moisture, humidity, and automatic irrigation.
- Low latency — Bluetooth mesh networks can be useful in low-latency use-case scenarios. In networks where round-trip time is of a high significance, specific functional nodes and relay nodes can be used to minimize the communication delay while maintaining coverage and reliability.

Mesh networking with the Bluetooth standard can be used in intelligent IoT solutions to facilitate home, commercial, and industrial automation. In summary, Bluetooth mesh networking is comparable to all other Bluetooth connectivity and establishes hub-less networks that expand the coverage and reliability of Bluetooth systems.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed March 22, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.0. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[4] Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Profile." Version 1.0.1. https://www.bluetooth.com/.

[5] Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Model Specification." Version 1.0.1. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks” on page 3-136
- “Bluetooth Mesh Flooding in Wireless Sensor Networks” on page 3-159

Bluetooth-WLAN Coexistence

Due to the ubiquitous deployment of wireless networks and devices on the unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band, multiple homogenous and heterogeneous networks (Bluetooth, Wi-Fi, and ZigBee) operating in this band are likely to coexist in a physical scenario. The wireless personal area network (WPAN) represented by the Bluetooth [1] and wireless local area network (WLAN) represented by IEEE® 802.11 standard both operate in the 2.4 GHz ISM frequency band. Bluetooth and WLAN radios often operate in the same physical scenario and some times in the same device. In these cases, Bluetooth and WLAN transmissions can interfere with each other, impacting the performance and reliability of both networks.

IEEE 802.15.2 Task Group [3] considers proposals for mechanisms to improve the level of coexistence between Bluetooth and WLAN devices and publishes the recommended practices derived from these.

Bluetooth and IEEE 802.11 WLAN Specifications

Bluetooth technology uses low-power radio frequency to enable short-range communication. Bluetooth is equated with the implementation specified by the Bluetooth Core Specification [2] group of standards maintained by the Bluetooth Special Interest Group (SIG) industry consortium. The Communications Toolbox Library for the Bluetooth Protocol enables you to model Bluetooth low energy (BLE), BLE mesh, and Bluetooth basic rate/enhanced data rate (BR/EDR) communications system links, as specified in the Bluetooth Core Specification. Bluetooth BR/EDR and BLE devices operate in the unlicensed 2.4 GHz ISM frequency band.

The Bluetooth BR mode is mandatory, whereas EDR mode is optional. The Bluetooth BR/EDR radio implements a 1600 hops/sec frequency-hopping spread spectrum (“FHSS” on page 13-61) technique. The radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth channel has a bandwidth of 1 MHz. Each frequency is located at (2402 + k) MHz, where k = 0,1,..., 78. The modulation technique for BR and EDR mode is Gaussian frequency shift-keying (GFSK) and differential phase shift-keying (DPSK), respectively. The baud rate is 1 Msymbols/s. The Bluetooth BR/EDR radio uses the time division duplex (TDD) topology in which data transmission occurs in one direction at one time. The transmission alternates in two directions, one after the other.

In BLE, the operating band is divided into 40 channels, k = 0, 1, ..., 39, with a channel bandwidth of 2 MHz. The range of RF center frequencies is [2402, 2480] MHz. The user data packets are transmitted using channels in the range [0, 36]. The advertising data packets are transmitted on channels 37, 38, and 39. BLE also implements GFSK modulation. The BLE physical layer (PHY) uses FHSS to reduce interference and to counter the impact of fading channels. The time between frequency hops can vary from 7.5 ms to 4 s and is set at the connection establishment for each Slave with the Master. The Master device provides the synchronization reference. The Slave device synchronizes to the clock and frequency-hopping pattern of the Master device. The support for the data rate at 1 Mbps is mandatory for specification version 4.x compliant devices. At a data rate of 1 Mbps, the data transmission is uncoded. Optionally, devices compliant with the Bluetooth Core Specification 5.x support these additional data rates:

- Coded transmission at bit rates of 500 kbps or 125 kbps
- Uncoded transmission at a bit rate of 2 Mbps

To explore the Bluetooth BR/EDR and BLE protocol stack, see “Bluetooth Protocol Stack” on page 13-7. For information about different packet structures implemented in Bluetooth BR/EDR and BLE transmissions, see “Bluetooth Packet Structure” on page 13-23. To study the fundamentals of Bluetooth mesh networking and its applications, see “Bluetooth Mesh Networking” on page 13-46.

The IEEE 802.11 (Wi-Fi) standard is a wireless technology that connects devices and an infrastructure in a WLAN. WLAN is compliant with various IEEE 802.11 standards. Some of the prominent and widely implemented standards are 802.11 a/b/g/n/ac/ax. The 802.11a standard uses the 5 GHz unlicensed national information infrastructure (U-NII) band and provides at least 23 nonoverlapping 20 MHz wide channels instead of three nonoverlapping 20 MHz-wide channels offered by the 2.4 GHz band. The 802.11ac standard also operates in only the 5 GHz frequency band. As per Part 15 of the U.S. Federal Communications Commission (FCC) Rules and Regulations, 802.11b, 802.11g, and 802.11n standards use the 2.4 GHz. Devices that use these standards suffer interference in the 2.4 GHz band from Bluetooth devices. To mitigate this interference, devices that use 802.11b, 802.11g, or 802.11n standards implement direct-sequence spread spectrum (“DSSS” on page 13-62), “Orthogonal Frequency-Division Multiplexing” on page 13-63 (OFDM), and multiple-input, multiple-output (MIMO) OFDM signaling techniques, respectively. Devices that use the 802.11n or 802.11ax (Wi-Fi-6) standard operate in dual-band at 2.4 GHz and 5 GHz. The 802.11ax standard enhances the existing 802.11 a/b/g/n/ac standards even if they are not fully upgraded to 802.11ax. The OFDM-based channel access technique of 802.11ax standard is completely backward-compatible with traditional enhanced distributed channel access/carrier-sense multiple access (EDCA/CSMA). IEEE 802.11ax provides maximum compatibility, coexisting efficiently with 802.11a/n/ac devices.

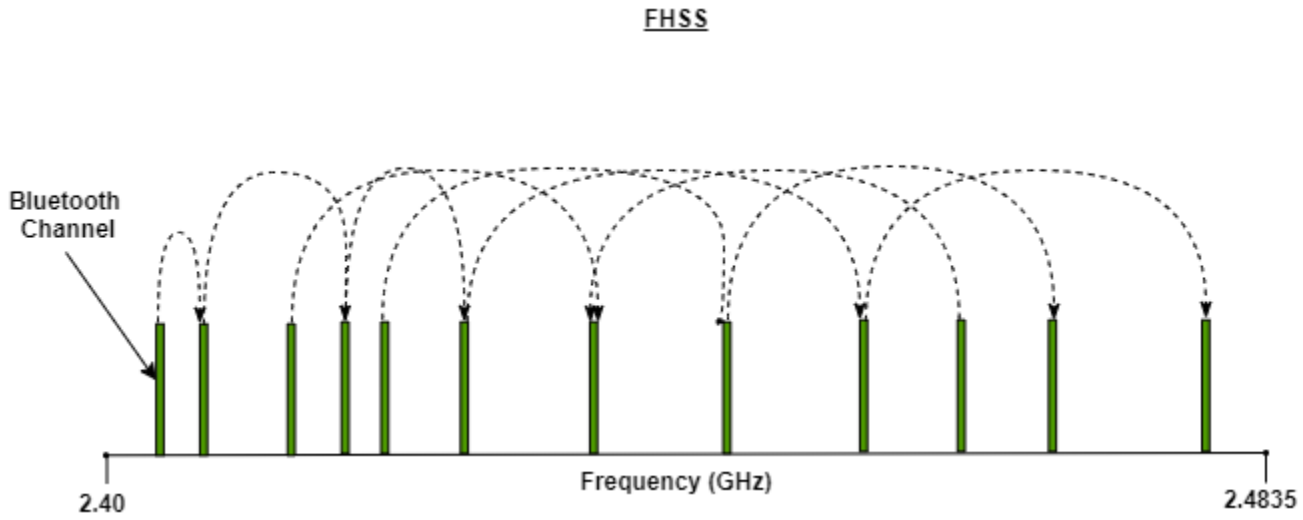
For more information about WLAN radio frequency channels, see “WLAN Radio Frequency Channels” (WLAN Toolbox). For more information about WLAN packet structures, see “WLAN PPDU Structure” (WLAN Toolbox) and “Packet Size and Duration Dependencies” (WLAN Toolbox).

Spread Spectrum Techniques

Bluetooth and WLAN technologies operate using the spread spectrum signal structuring. This signal structuring technique enables a narrowband signal such as a stream of 1s and 0s, to spread across a given frequency spectrum and transform into a wideband signal. Bluetooth devices implement the basic FHSS technique defined in the Bluetooth Core Specification [2]. This basic frequency-hopping technique is modified into an adaptive frequency hopping (AFH) technique to mitigate interference. WLAN devices use the DSSS technique.

FHSS

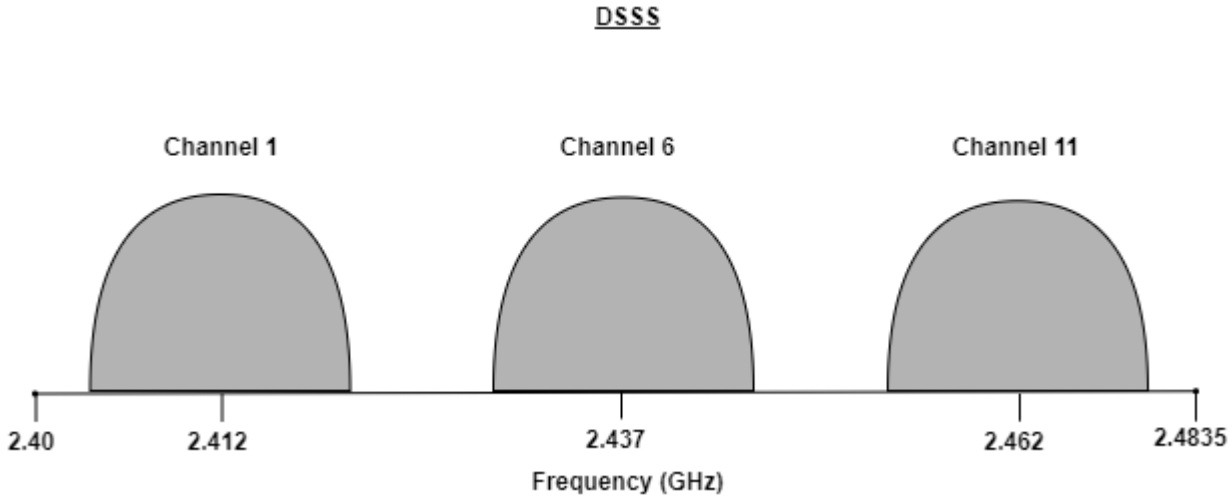
The basic Bluetooth frequency-hopping technique or the FHSS spreads the narrowband signal by *hopping* across different channels on the 2.4 GHz frequency spectrum. This figure shows how the FHSS transmits a Bluetooth signal on different frequencies at specific intervals to spread the signal across a relatively wide operating band.



The transmitting and receiving Bluetooth devices adhere to a specific hopping sequence during a particular session so that the receiving device can anticipate the frequency of the next transmission. In this case, Bluetooth makes full use of the 2.4 GHz frequency spectrum.

DSSS

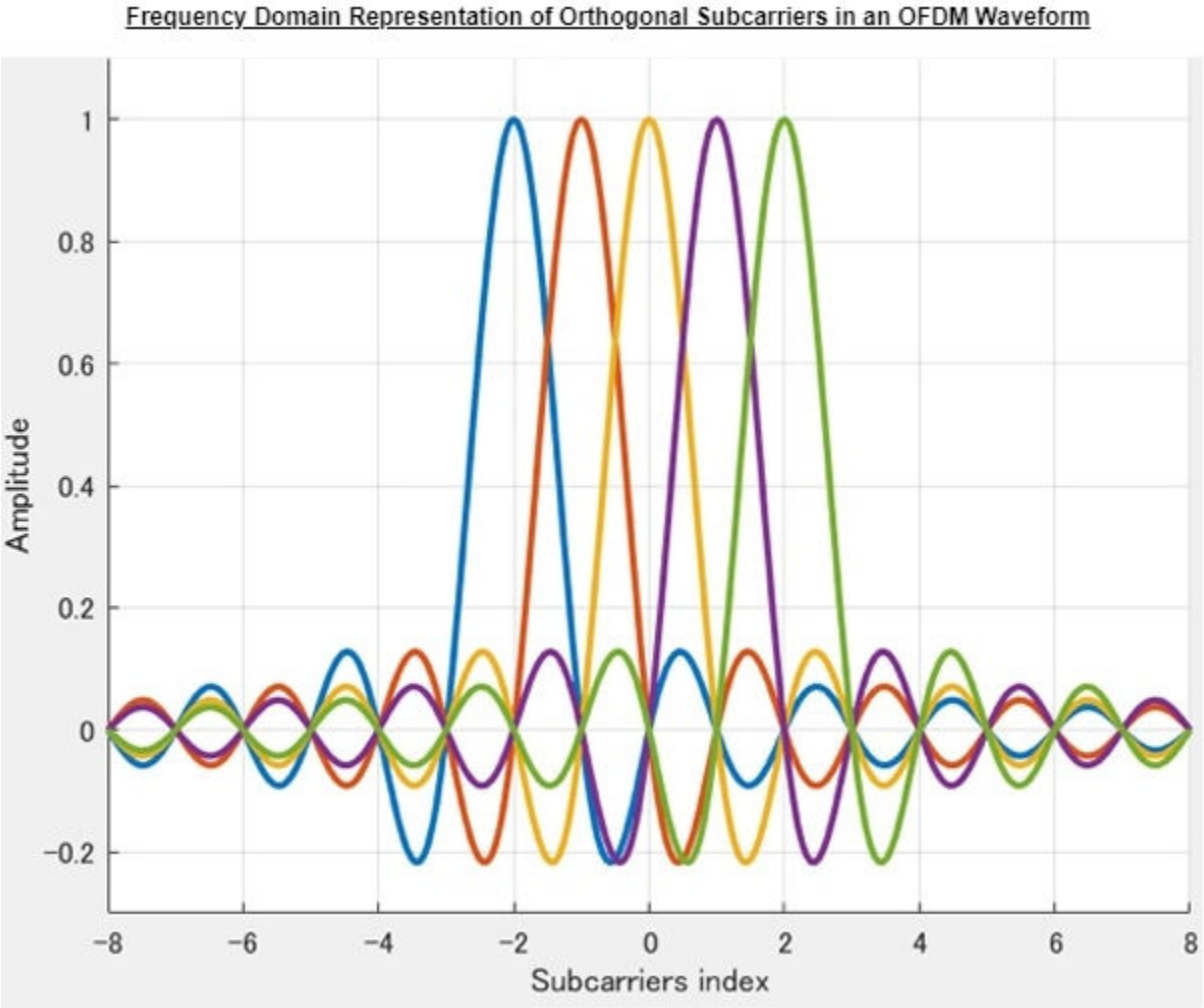
With the DSSS, the narrowband data signal is divided and simultaneously transmitted on multiple frequencies within a specific frequency band. This figure shows how the DSSS continually transmits the data signal across different channels.



The DSSS adds redundant data bits known as *chips*, to the data signal to denote 1s and 0s. The ratio of chips to data is called the *spreading ratio*. Increasing the ratio increases the immunity of the WLAN signal to interference. This is because if part of the transmission is corrupted, the data can still be recovered from the remaining part of the chipping code. The DSSS technique provides greater transmission rates than the FHSS. The DSSS also protects against data loss through redundant simultaneous data transmission. However, because DSSS floods the channel with redundant transmissions, it is more vulnerable to interference from Bluetooth devices operating on the same frequency band.

Orthogonal Frequency-Division Multiplexing

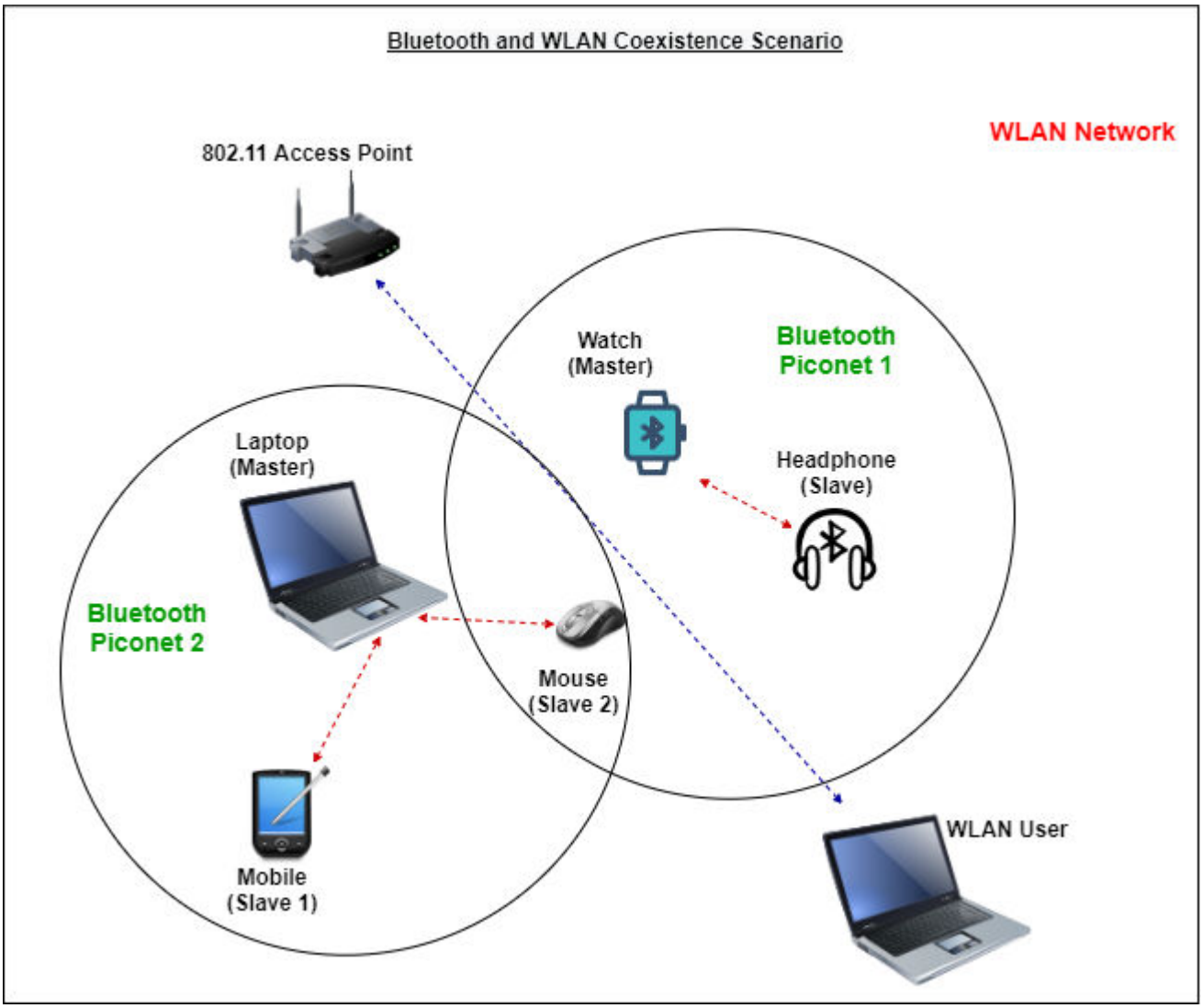
OFDM is a flexible, multicarrier modulation technique implemented by IEEE standards 802.11g/n/ac/ax. OFDM partitions the channel bandwidth into multiple narrow-band orthogonal subcarriers to carry the information. This partitioning enables the removal of guard bands. However, because the orthogonal subcarriers are unrelated, they can overlap each other. Therefore, OFDM is bandwidth efficient. This figure shows the frequency domain representation of the orthogonal subcarriers in an OFDM waveform.



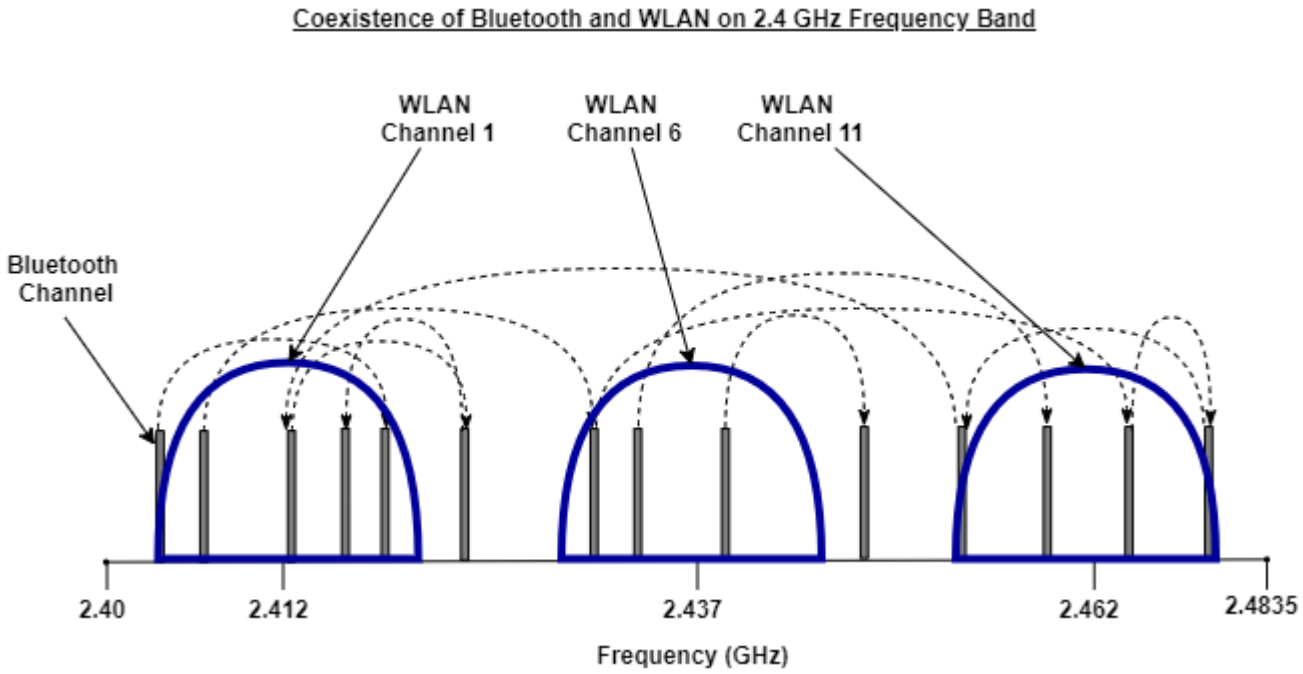
The use of narrow-band subchannels (compared to a single wideband channel) helps mitigate channel fading. As each subchannel operates at a low data rate, OFDM is very resilient to intersymbol interference and interframe interference. As data is transmitted simultaneously on multiple orthogonal subcarriers, OFDM can provide very high throughput. To further maximize the throughput, you can use OFDM with MIMO, extended rate physical (ERP), and multiuser (MU) technologies.

Bluetooth-WLAN Coexistence Problem

As Bluetooth and WLAN devices operate in the same 2.4 GHz frequency band, a mutual interference exist between the two wireless networks. This interference results in performance degradation. For example, consider the scenario shown in this figure. The scenario consists of two Bluetooth piconets collocated with a WLAN.



If the transmission in piconet 1 is overlapped in time and frequency by transmissions from piconet 2 and/or the WLAN, a Bluetooth packet can be lost. This figure shows how the Bluetooth and WLAN devices share the 2.4 GHz frequency spectrum.



If the Bluetooth packets transmitted through the FHSS hops to the portion of the frequency spectrum occupied by the DSSS WLAN transmitter, then mutual interference occurs. This interference results in packet collisions. Factors such as the distance between WLAN and Bluetooth devices, the data traffic present in these two networks, power levels of the devices, and data rate of the WLAN network impact the level of interference. Additionally, different types of data traffic have different levels of sensitivity to the interference. For example, voice traffic can be more sensitive to interference than data traffic.

Bluetooth in Presence of 802.11b WLAN Interferer

A transmission that uses one spread spectrum technique interferes with a receiver that uses different spread spectrum technique. 802.11b WLAN devices operate in 22 MHz bandwidth. In Bluetooth, 22 of the 79 hopping channels are subject to interference. A frequency-hopping system like Bluetooth is vulnerable to interference from the adjacent channels as well. This vulnerability increases the total number of interference channels from 22 to 24. Based on these assumptions, the results shown in [3] quantify the packet error rate (PER) in Bluetooth transmissions with a 802.11b WLAN interferer. The results show that the network throughput decreases and network delay increases for Bluetooth in the presence of 802.11b interference.

To study the impact of WLAN interference on BLE transmission, see “BLE Coexistence Model with WLAN Signal Interference” on page 3-175 and “Statistical Modeling of WLAN Interference on BLE Network” on page 3-198 examples.

802.11b WLAN in Presence of Bluetooth Interferer

When a Bluetooth device hops into the 802.11b passband, a packet collision can occur with the WLAN device. This collision occurs because 22 of the 79 Bluetooth channels fall within the WLAN passband. As 802.11b devices support four data rates (1, 2, 5, and 11 Mbps), the transmission time of the WLAN packets may vary significantly for packets carrying the exact same data. Increasing the duration of the WLAN packet increases the likelihood that the packet collides with an interfering Bluetooth packet. If automatic data rate scaling is implemented and enabled in the WLAN device, the Bluetooth

interference can cause the WLAN device to scale to a lower data rate. Lower data rate increase the temporal duration of the WLAN packets. This increase in packet duration can lead to frequent packet collisions with the interfering Bluetooth packets. In some implementations, the frequent packet collisions can result in WLAN scaling down its data rate to 1 Mbps. In this case, to ensure reliable packet delivery, the IEEE 802.11 medium access (MAC) layer incorporates an acknowledgement (ACK) and retransmission mechanism.

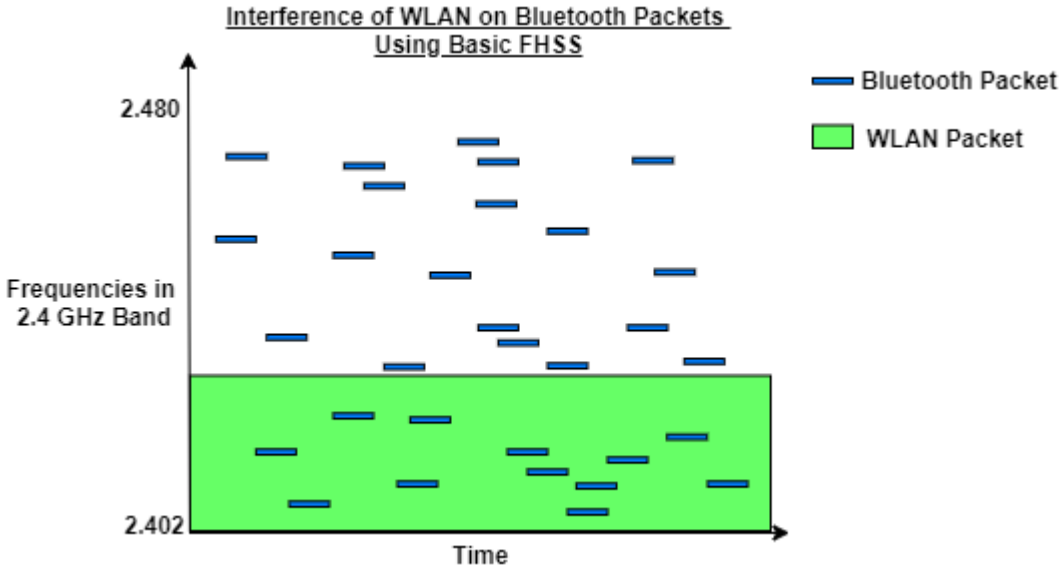
Coexistence Mechanisms

Interference between Bluetooth and WLAN can be addressed by two coexistence mechanisms – noncollaborative and collaborative.

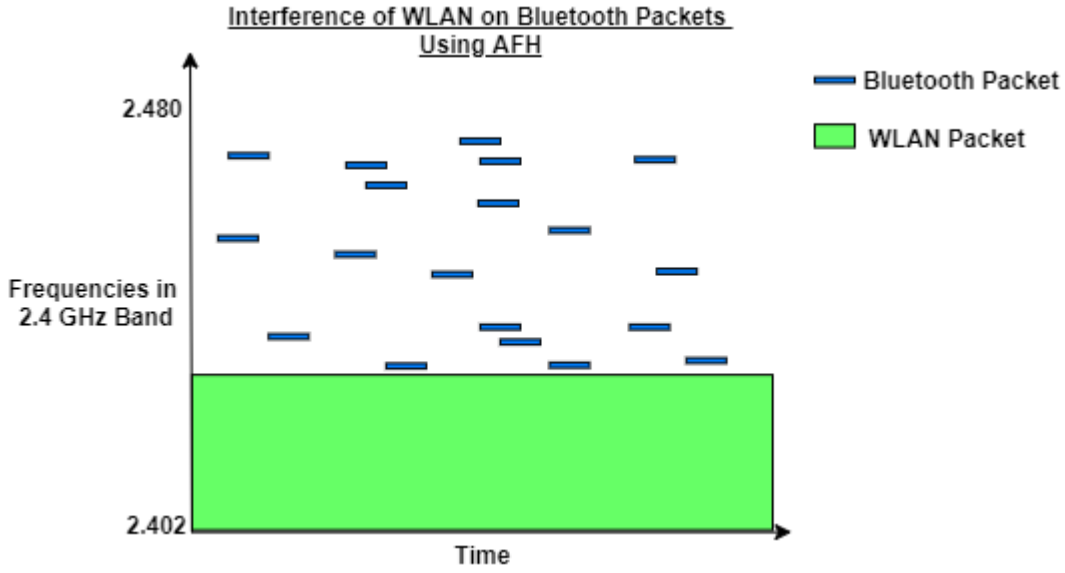
Noncollaborative Coexistence

Noncollaborative mechanisms do not exchange information between two wireless networks. These coexistence mechanisms are applicable only after a WLAN or Bluetooth piconet is established and the data is to be transmitted. These coexistence mechanisms do not help in the process of establishing a WLAN or Bluetooth piconet. As per the recommended practices mentioned in [3], these noncollaborative coexistence mechanisms are used to mitigate interference between Bluetooth and WLAN.

- Adaptive frequency hopping (AFH) — Prior to the emergence of AFH, Bluetooth devices implemented the basic FHSS signal structuring scheme. The FHSS scheme often resulted in Bluetooth and WLAN packet transmissions interfering with each other, as shown in this figure.



On the contrary, AFH enables Bluetooth to adapt to its environment by identifying fixed sources of WLAN interference and excluding them from the list of available channels. This figure shows the previous scenario with AFH enabled.



AFH dynamically alters the frequency hopping sequence to avoid the interference observed by the Bluetooth devices. AFH operates through these four processes.

- AFH capability discovery: This process informs the Master about the Slave(s) that support AFH and the associated parameters.
- Channel classification: This process classifies the channels as *good* or *bad*. Channel classification takes place in the Master and optionally in the Slave(s).
- Channel classification information exchange: This process uses AFH link manager protocol (LMP) commands to exchange information between the Master and the supporting Slave(s) in the piconet.
- Adaptive hopping: This process adaptively selects *good* channels for frequency hopping.

For more information about how AFH mitigates interference and enables coexistence between Bluetooth and WLAN, see “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76.

Note For more information about AFH, see Annex B of IEEE 802.15.2 Task Group [3].

- Adaptive interference suppression — This mechanism is exclusively related to signal processing in the WLAN physical layer (PHY). The adaptive interference suppression mechanism requires a Bluetooth receiver collocated with a WLAN receiver. The WLAN receiver has no prior knowledge of the timing or frequency used by the Bluetooth network. The WLAN receiver uses an adaptive filter to estimate and cancel the interfering signal.

Note For more information about adaptive interference suppression, see Clause 8 of IEEE 802.15.2 Task Group [3].

- Adaptive packet selection and scheduling — Bluetooth transmissions involve various packet types with different configurations such as packet length and degree of error protection used. By selecting the best packet type according to the channel condition of the upcoming frequency hop, better throughput and network performance can be achieved. Additionally, packet transmissions can be scheduled efficiently so that the Bluetooth devices transmit during hops that are outside of

the WLAN frequencies and refrain from transmitting while in-band. This type of packet transmission scheduling minimizes mutual interference and also increases the throughput of Bluetooth networks.

Note For more information about adaptive packet selection and scheduling, see Clause 9 of IEEE 802.15.2 Task Group [3].

- Packet scheduling for synchronous connection-oriented (SCO) links — Voice applications are among the most sought-after applications for Bluetooth devices but are vulnerable to interference. Interference from an in-band WLAN network degrades the voice quality of the Bluetooth SCO link, making it inaudible to the users. This noncollaborative coexistence mechanism recommends improvements that can significantly improve the quality-of-service (QoS) for SCO links. The fundamental idea is to enable the SCO link the flexibility of selecting hops that are out-of-band with the collocating WLAN spectrum for transmission. The duty cycle of the SCO link does not change.

Note For more information about packet scheduling for SCO links, see Annex A of IEEE 802.15.2 Task Group [3].

- Packet scheduling for asynchronous connection-oriented logical (ACL) links — This mechanism defines a procedure to minimize the impact of WLAN interference on Bluetooth devices by using these two components.

  - Channel classification: It is performed on every Bluetooth receiver and is based on the measurements conducted per frequency or channel to locate the presence of interference. A channel is considered as *good* if it can correctly decode a received packet. Otherwise, the channel is considered as *bad*. Good and bad channels are classified based on different criteria such as the received signal strength indicator (RSSI), PER, or negative ACKs.
  - Master delay policy: It uses the information available in the channel classification table to avoid packet transmission in a *bad* channel. Because the Master device controls and manages all transmissions in a piconet, the delay rule must be implemented in the Master device only. Also, a Slave transmission must follow each Master transmission. Therefore, the Master checks the receiving frequency of the Slave and its own receiving frequency before choosing to transmit a packet in a given frequency hop.

Note For more information about packet scheduling for ACL links, see Clause 10 of IEEE 802.15.2 Task Group [3].

Collaborative Coexistence

In collaborative coexistence mechanisms, two wireless networks collaborate and exchange network-related information. As per the recommended practices stated in [3], the three collaborative coexistence mechanisms are:

- Alternating wireless medium access (AWMA) — In the AWMA mechanism, a WLAN radio and a Bluetooth radio are collocated in the same physical unit, enabling a wired connection between the two radios. The collaborative coexistence mechanism uses this wired connection to coordinate access to the wireless medium between WLAN and Bluetooth. The AWMA mechanism uses part of the wireless IEEE 802.11 beacon interval for the Bluetooth operations. From a timing perspective, the medium assignment alternates between usage following the IEEE 802.11 procedures and usage following the Bluetooth procedures. Each wireless network limits its transmissions to the appropriate time segment, thus preventing mutual interference between the two networks.

Note For more information about AWMA, see Clause 5 and Annex I of IEEE 802.15.2 Task Group [3].

- Packet traffic arbitration (PTA) — In the PTA mechanism, the WLAN station and the Bluetooth device are collocated. The PTA control entity provides per-packet authorization of all transmissions. This mechanism can deny permission for transmission if it has chances of collisions. The PTA mechanism dynamically coordinates sharing of the wireless medium based on the traffic load of WLAN and Bluetooth. If a collision occurs, the PTA mechanism prioritizes transmission based on the priorities of different packets. Using the PTA mechanism in case of high variability in the WLAN and Bluetooth traffic load or whenever a Bluetooth SCO link needs to be supported.

Note For more information about PTA, see Clause 6 and Annex J of IEEE 802.15.2 Task Group [3].

- Deterministic interference suppression — In this mechanism, a null is inserted in the WLAN receiver at the frequency of the Bluetooth signal. Because Bluetooth devices hop to a new frequency for each packet transmission, the WLAN receiver must know the hopping pattern and timing of the Bluetooth device. The hopping pattern and timing is obtained by using a Bluetooth receiver as part of the WLAN receiver. Deterministic interference suppression is a collocated, collaborative coexistence mechanism.

Note For more information about deterministic interference suppression, see Clause 7 and Annex K of IEEE 802.15.2 Task Group [3].

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed April 17, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

[3] *P802.15.2/D09 - IEEE Draft Recommended Practice for Information Technology Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks Specific Requirements - Part 15.2: Coexistence of Wireless Personal Area Networks With Other Wireless Devices Operating in Unlicensed Frequency Bands.* LAN/MAN Standards Committee, IEEE Computer Society, 2003, https://ieeexplore.ieee.org/document/4040972.

[4] Macleod, M D. “14 - Coding.” *In Telecommunications Engineer’s Reference Book*, edited by Fraidoon Mazda, 14–1. Butterworth-Heinemann, 1993. https://www.sciencedirect.com/science/article/pii/B9780750611626500204.

[5] Xiao, Yang, and Yi Pan. “Coexistence of Bluetooth Piconets and Wireless LAN.” *In Emerging Wireless LANs, Wireless PANs, and Wireless MANs: IEEE 802.11, IEEE 802.15, 802.16 Wireless Standard Family*, 151–85. Wiley, 2009. https://ieeexplore.ieee.org/abstract/document/8040602.

[6] “IEEE SA - The IEEE Standards Association - Home.” Accessed May 4, 2020. https://standards.ieee.org/.

[7] IEEE P802.11ax/D4.1. “*Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 1: Enhancements for High Efficiency WLAN*.” Draft

Standard for Information technology — Telecommunications and information exchange between systems. Local and metropolitan area networks — Specific requirements.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “Statistical Modeling of WLAN Interference on BLE Network” on page 3-198
- “WLAN Protocol Stack” (WLAN Toolbox)
- “WLAN PPDU Structure” (WLAN Toolbox)
- “Packet Size and Duration Dependencies” (WLAN Toolbox)

Parameterize BLE Direction Finding Features

Bluetooth technology [1] uses low-power radio frequency to enable short-range communication at a low cost. The Bluetooth Core Specification 5.1 [2] provided by the Bluetooth Special Interest Group (SIG) added direction finding features in Bluetooth low energy (BLE) technology. The Bluetooth direction-finding capabilities, angle of arrival (AoA) and angle of departure (AoD), are introduced in the Bluetooth Core Specification 5.1 [2]. For more information about BLE direction finding, see the “Bluetooth Location and Direction Finding” on page 13-37 topic and “Bluetooth Low Energy Based Positioning Using Direction Finding” on page 3-38 example.

The Communications Toolbox Library for the Bluetooth Protocol enables you to configure these simulation and configuration parameters of BLE direction finding features.

Set Simulation Parameters for BLE Location and Direction Finding

Specify the dimension in which to determine the BLE node position and the number of BLE locators. To estimate the 2-D or 3-D position of a BLE node, specify at least two or three locators, respectively.

```
numDimensions = 2;
numLocators = 3;
```

Specify the bit energy to noise density ratio (Eb/No) range (in dB) and the number of iterations to simulate each Eb/No point.

```
EbNo = 6:2:16;
numIterations = 200;
```

Specify the direction finding method, the direction finding packet type, and the physical layer (PHY) transmission mode. The PHY transmission mode must be LE1M or LE2M for a connection-oriented constant tone extension (CTE) and LE1M for a connectionless CTE.

```
dfMethod = AoA;
dfPacketType = ConnectionCTE;
phyMode = LE1M;
```

Specify the antenna array parameters. The antenna array size must be a scalar or vector for 2-D or 3-D positioning, respectively. The scalar or vector array size represents a uniform linear array (ULA) or uniform rectangular array (URA), respectively. Specify the normalized element spacing between the antenna elements with respect to the wavelength. Specify the antenna switching pattern as a 1-by-M row vector, where M is in the range [2, $\frac{74}{\text{slotDuration}} + 1$].

```
arraySize = 16;
elementSpacing = 0.5;
switchingPattern = 1:prod(arraySize);
```

Specify the BLE waveform generation parameters. The length of the CTE must be in microseconds, in the range [16, 160], and have a step size of 8 microseconds.

```
slotDuration = 2;          % Slot duration in microseconds
cteLength = 160;
sps = 8;
chanIndex = 17;
crcInit = '555551';
accAddress = '01234567';
payloadLength = 1;         % Payload length in bytes
```

Create BLE Angle Estimation Configuration Object

Create a default BLE angle estimation configuration object by using the `bleAngleEstimateConfig` object. This object enables you to configure different parameters for BLE angle estimation.

```
cfg = bleAngleEstimateConfig
```

```
cfg =
  bleAngleEstimateConfig with properties:

           ArraySize: 4
      ElementSpacing: 0.5000
        SlotDuration: 2
    SwitchingPattern: [1 2 3 4]

   Read-only properties:
    No properties.
```

Specify a URA antenna design by setting the antenna array size of the configuration object to [4 4]. Set the row element spacing and column element spacing to 0.4 and 0.3, respectively. Specify the value of the antenna switching pattern.

```
cfg.ArraySize = [4 4];
cfg.ElementSpacing = [0.4 0.3];
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]

   Read-only properties:
    No properties.
```

Generate Random Positions for BLE Locators

A BLE locator represents a receiving device and a transmitting device in AoA and AoD calculation, respectively. To place a BLE node at the origin and the locators randomly in the 2-D or 3-D space, use helperBLEGeneratePositions function. Specify the number of locators as 3 and the number of dimensions as 2. The function returns the 2-D position of the BLE node at the origin, a matrix representing the position of the three locators, and the AoA or AoD (in degrees) between the BLE node and the locators.

```
[nodePos,locatorPos,angle] = helperBLEGeneratePositions(3,2)
```

```
nodePos = 2×1

     0
     0


locatorPos = 2×3

  -23.7249  -57.5071  -12.5811
  -77.9415   69.9823   -1.7241


angle = 3×1

   73.0701
  -50.5887
    7.8032
```

Generate BLE Direction Finding Packet

Set the simulation parameters to generate a BLE direction finding packet.

```
dfPacketType = ConnectionCTE;
cteLength = 160;
dfMethod = AoA;
payloadLength = 1;                  % Payload length in bytes
crcInit = '555551';
slotDuration = 2;                   % Slot duration in microseconds
```

Derive the type of CTE based on the slot duration and the direction finding method.

```
if strcmp(dfMethod,'AoA')
    cteType = [0;0];
else
    cteType = [0;1];
    if slotDuration == 1
        cteType = [1;0];
    end
end
```

To generate a direction finding packet corresponding to the type of CTE, use helperBLEGenerateDFPDU function.

```
dfPacket = helperBLEGenerateDFPDU(dfPacketType,cteLength,cteType,payloadLength,crcInit);
```

Perform Antenna Steering and Switching on BLE Waveform

Set BLE direction finding simulation parameters to perform antenna steering and switching on a BLE waveform.

```
dfPacketType = ConnectionCTE;
cteLength = 160;
dfMethod = AoA;
payloadLength = 1;          % Payload length in bytes
crcInit = '555551';
slotDuration = 2;           % Slot duration in microseconds
phyMode = LE1M;
sps = 8;
chanIndex = 17;
```

Derive the type of CTE based on the slot duration and the direction finding method.

```
if strcmp(dfMethod,'AoA')
    cteType = [0;0];
else
    cteType = [0;1];
    if slotDuration == 1
        cteType = [1;0];
    end
end
```

Create a default BLE angle estimation configuration object. Specify the antenna slot duration.

```
obj = bleAngleEstimateConfig;
obj.SlotDuration = slotDuration;
```

Generate a direction finding packet corresponding to the type of CTE by using the helperBLEGenerateDFPDU function.

```
dfPacket = helperBLEGenerateDFPDU(dfPacketType,cteLength,cteType,payloadLength,crcInit);
```

Using the direction finding packet, generate the BLE waveform.

```
bleWaveform = bleWaveformGenerator(dfPacket,'Mode',phyMode,'SamplesPerSymbol',sps, ...
    'ChannelIndex',chanIndex,'DFPacketType',dfPacketType);
```

Use the helperBLESwitchAntenna function to steer the BLE waveform by 45 degrees in azimuth and 0 degrees in elevation and switch between the antennas according to the switching pattern.

```
dfWaveform = helperBLESwitchAntenna(bleWaveform,45,...
    phyMode,sps,dfPacketType,payloadLength,cteLength,obj);
```

Decode BLE Waveform with Connection-Oriented CTE

Set the simulation parameters to decode the BLE waveform.

```
dfPacketType = ConnectionCTE;
cteLength = 160;
dfMethod = AoA;
payloadLength = 1;          % Payload length in bytes
crcInit = '555551';
slotDuration = 2;           % Slot duration in microseconds
phyMode = LE1M;
sps = 8;
chanIndex = 17;
```

Derive the type of CTE based on the slot duration and the direction finding method.

```
if strcmp(dfMethod,'AoA')
    cteType = [0;0];
else
    cteType = [0;1];
    if slotDuration == 1
        cteType = [1;0];
    end
end
```

Create a default BLE angle estimation configuration object. Specify the antenna slot duration.

```
obj = bleAngleEstimateConfig;
obj.SlotDuration = slotDuration;
```

To generate a direction finding packet corresponding to the type of CTE, use the helperBLEGenerateDFPDU function.

```
dfPacket = helperBLEGenerateDFPDU(dfPacketType,cteLength,cteType,payloadLength,crcInit);
```

Using the direction finding packet, generate the BLE waveform.

```
bleWaveform = bleWaveformGenerator(dfPacket,'Mode',phyMode,'SamplesPerSymbol',sps,...
    'ChannelIndex',chanIndex,'DFPacketType',dfPacketType);
```

Get the in-phase and quadrature (IQ) samples by decoding the BLE waveform.

```
[bits,accAddr,iqSamples] = bleIdealReceiver(bleWaveform,'Mode',phyMode,...
    'SamplesPerSymbol',sps,'ChannelIndex',chanIndex,'DFPacketType',dfPacketType,'SlotDuration',s
```

Estimate AoA of BLE Waveform

Create a BLE angle estimation configuration object, specifying the values of the antenna array size, slot duration, and antenna switching pattern.

```
obj = bleAngleEstimateConfig('ArraySize',2,'SlotDuration',2, ...
    'SwitchingPattern',[1 2]);
```

Estimate the AoA of the BLE waveform by using the `bleAngleEstimate` function. The function accepts IQ samples and the BLE angle estimation configuration object as inputs. You can either use the IQ samples obtained by decoding the BLE waveform or use the IQ samples corresponding to the connection data channel protocol data unit (PDU).

Specify the IQ samples of a connection data PDU with an AoA 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 waveform.

```
angle = bleAngleEstimate(IQsamples,obj)
```

```
angle = 70
```

Estimate Unknown Position of BLE Node Using Triangulation

Specify the number of locators as 3 and the number of dimensions as 2. The helperBLEGeneratePositions function returns the 2-D position of the BLE node at the origin, a matrix representing the position of the three locators, and the AoA or AoD (in degrees) between the BLE node and the locators.

```
[nodePos,locatorPos,angle] = helperBLEGeneratePositions(3,2);
```

Get the the unknown position of the BLE node by using the helperBLETriangulation function.

```
nodePosEstimate = helperBLETriangulation(locatorPos,angle)
```

```
nodePosEstimate = 2×1
10^-13 ×

   -0.0822
    0.2334
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 1, 2020. 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). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

Functions
`bleAngleEstimate` | `bleWaveformGenerator` | `bleIdealReceiver`

Objects
`bleAngleEstimateConfig`

More About

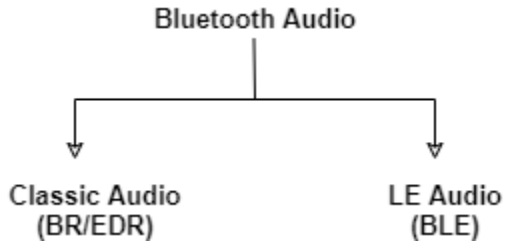
- “What Is Bluetooth?” on page 13-2
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Location and Direction Finding” on page 13-37

Bluetooth Low Energy Audio

The Bluetooth Core Specification 5.2 [2] defined by the Bluetooth Special Interest Group (SIG) introduced the next generation of Bluetooth audio called the low energy (LE) audio. LE audio operates on the Bluetooth low energy (BLE) standard. For more information about the BLE stack, see “Bluetooth Protocol Stack” on page 13-7.

What Is LE Audio?

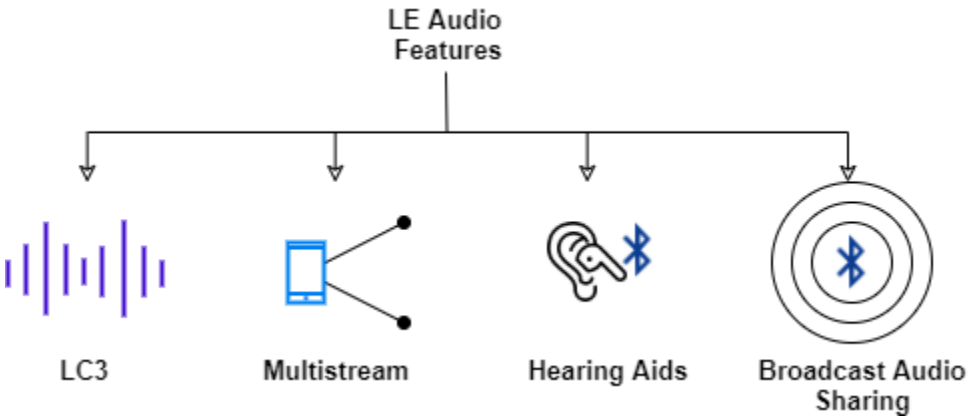
This figure shows the taxonomy of Bluetooth audio.



Bluetooth audio can be classified as – classic audio (operates on the basic rate/enhanced data rate (BR/EDR) physical layer (PHY)) and LE audio (operates on the BLE PHY). LE audio is the next generation of Bluetooth audio, which supports development of the same audio products and use cases as the classic audio. It also enables creation of new products and use cases and presents additional features and capabilities to help improve the performance of classic audio products. Some of the key features and use cases of LE audio include enabling audio sharing, providing multistream audio, and supporting hearing aids. For more information about LE audio features and use cases, see “Features of LE Audio” on page 13-78 and “Use Cases of LE Audio” on page 13-88, respectively.

Features of LE Audio

This figure illustrates the salient features of LE audio.



Low Complexity Communication Codec (LC3)

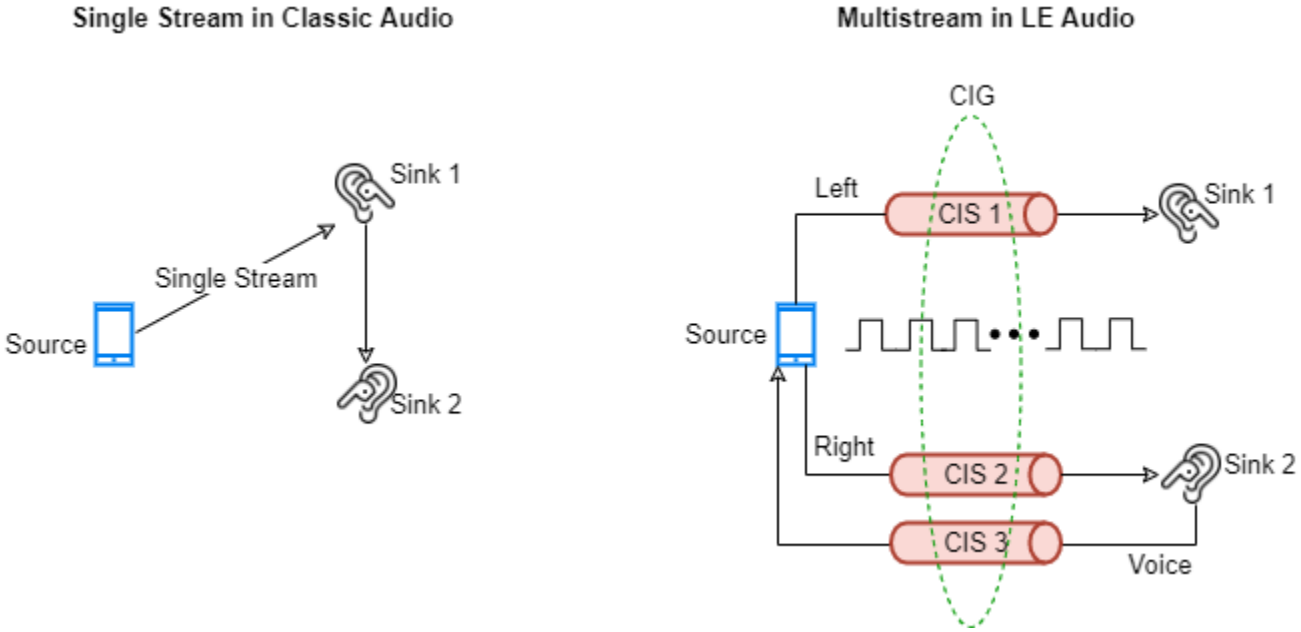
LE audio includes a new high quality, low power codec, known as LC3. It supports a wide range of sample rates, bit rates, and frame rates giving the product developers maximum flexibility to optimize their products to deliver the best possible audio experience to end users. As compared to the subband

codec (SBC) implemented by classic audio, LC3 is much more efficient in processing and delivering audio. A comparison between LC3 and SBC related to the standard stereo listening test [1] verifies that LC3 delivers high quality audio at low data rates. The results shown in [1] show that even at half of the bit rate, LC3 provides far superior audio experience than SBC.

The intrinsic shortcomings of SBC resulted in the manufacturers of audio equipment such as Bluetooth headphones turning to proprietary solutions such as audio codec 3 (AC3) and AptX. Such proprietary solutions need specific hardware support and add costs over standards-based implementations. The introduction of LC3 removes the dependency on the proprietary solutions, resulting in lower device costs. LC3 enables the product developers to have an efficient tradeoff between sound quality and power consumption. The high quality, low power characteristic of LC3 enables the product developers to optimize the longevity of the device battery.

Multistream Audio

Multistream audio enables you to transmit multiple, independent, and synchronized audio streams between an audio source device, such as a smartphone, and one or more audio sink devices like earbuds or earphones. To support multistream audio, the Bluetooth Core Specification 5.2 [2] introduced the connected isochronous stream (CIS) and connected isochronous group (CIG). For more information about the CIS and CIG, see “CIS and CIG” on page 13-81. This figure shows how LE audio enables you to send multiple audio streams between a source and sink.



Classic Bluetooth audio supports only a single point-to-point audio stream over the advanced audio distribution profile (A2DP). However, LE audio enables you to handle multiple isochronous audio streams with synchronization between them. The multistream support of LE audio can improve the performance of truly wireless earphones by providing a better stereo imaging experience, making the use of voice assistant services more seamless, and making switching between multiple audio source devices smoother [1].

Hearing Aids Support

LE audio provides exclusive support for hearing aids. Typically, hearing aid devices require low and efficient power consumption. LE audio supports high quality, low power capability of LC3 and the

efficient power consumption characteristic of the BLE standard. LE audio-supported hearing aids are interoperable, enabling you to connect to most smartphones, TVs, and laptops and making these devices much more accessible to people with hearing loss.

Broadcast Audio Sharing

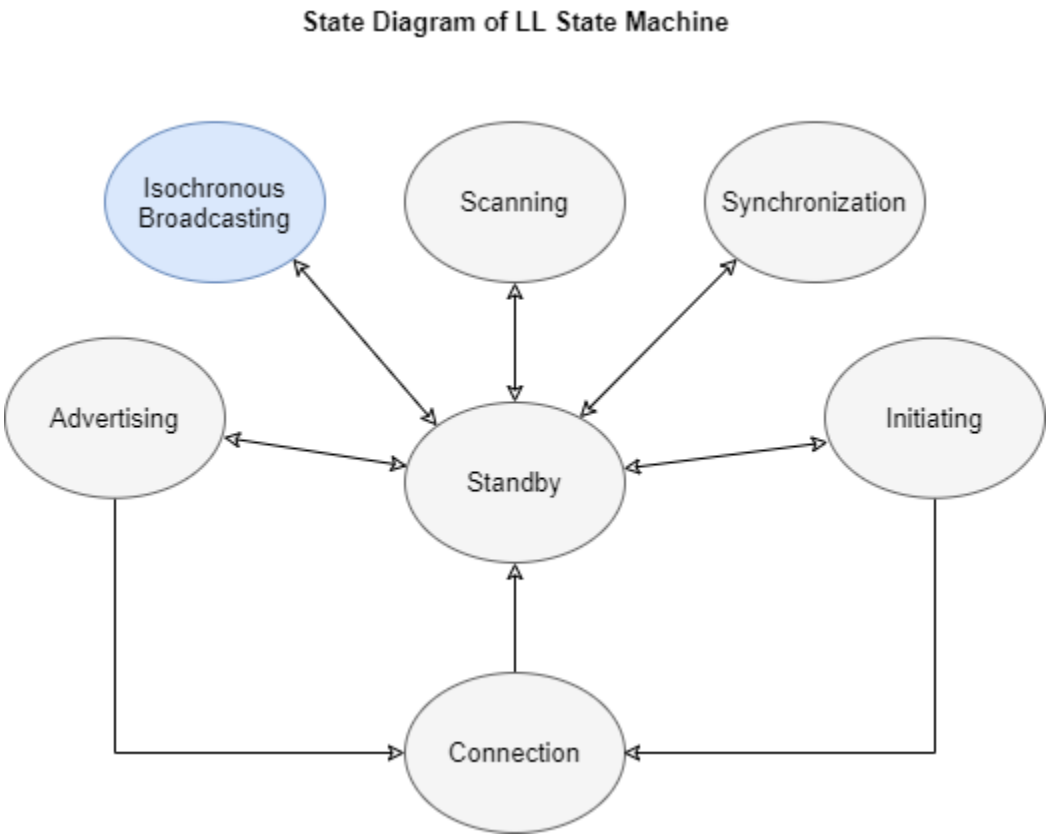
LE audio now supports the ability to broadcast one or more audio streams to an unlimited number of audio sink devices. Broadcast audio opens significant new opportunities for innovation, such as a new Bluetooth use case, audio sharing. Broadcast audio sharing can be personal (share audio with people around you) or location-based (share audio in public places like airports).

Support For LE Audio In Bluetooth Core Specification

The Bluetooth Core Specification 5.2 [2] introduced these updates pertaining to LE audio.

Changes to Link Layer (LL) State Machine

The functioning of the LL is described in terms of a state machine. This figure shows the state diagram of the LL state machine.



The Bluetooth Core Specification 5.2 [2] added a new state, Isochronous Broadcasting, to the LL state machine. In the Isochronous Broadcasting state, the LL transmits the isochronous data packets on an isochronous physical channel. The Isochronous Broadcasting state can be entered from the Standby state. If a device is in the Isochronous Broadcasting state, then it is referred to as an isochronous broadcaster.

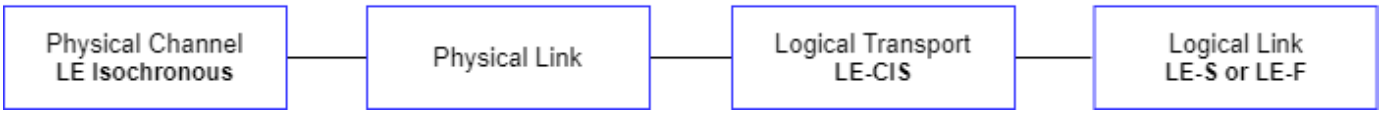
Note For more information about different states of the LL state machine, see Volume 6, Part B, Section 1.1 of the Bluetooth Core Specification 5.2 [2].

LE Isochronous Channels and the Bluetooth Data Transport Architecture

The LE isochronous channels feature enables you to transfer latency-sensitive data between the devices. This feature provides a mechanism to ensure synchronization between multiple sink devices receiving data from the same source. The expired data (data that violates the time-bound validity period) that is not transmitted, is discarded. Consequently, the receiving devices receive data that is valid with respect to its age and acceptable latency.

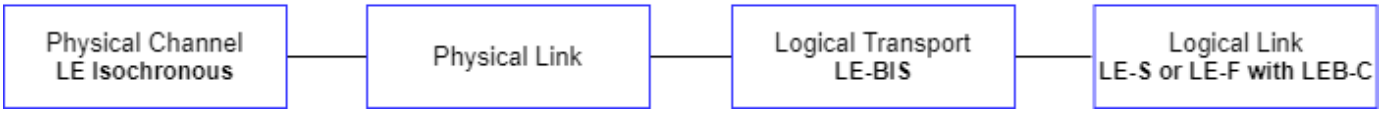
The Bluetooth data transport architecture now supports LE isochronous channels. The LE isochronous channels can be connection-oriented or connectionless. In both cases, the isochronous communication is realized using the new LE isochronous physical channel. This physical channel uses frequency hopping and specifies the timing of the first packet. This timing acts as a reference point for the timing of the subsequent packets. The LE isochronous physical channel can operate on an LE Uncoded (LE1M and LE2M) or LE Coded BLE PHY. The LE isochronous physical channel uses LE-Stream (LE-S) and LE-Frame (LE-F) logical links to transmit audio data and framed data packets, respectively. The connection-oriented isochronous channels use LE-CIS logical transport and support bidirectional communication. This figure shows the procedure of connection-oriented isochronous channel data transport.

Connection-oriented Isochronous Channel Data Transport



A single LE-CIS stream provides point-to-point isochronous communication between two connected devices. A flushing period is specified for the LE-CIS logical transport. Any packet that has not been transmitted within the flushing period is discarded. This figure shows the procedure of connectionless isochronous channel data transport.

Connectionless Isochronous Channel Data Transport

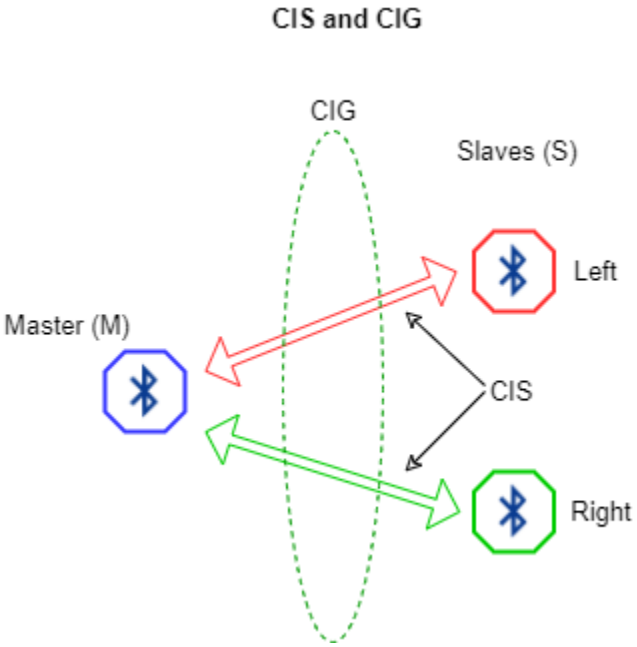


Connectionless isochronous communication uses broadcast isochronous streams (BIS) and supports only unidirectional communication. The BIS uses LE-S or LE-F logical links over the LE isochronous physical channel for user data, with the new LE-broadcast control (LEB-C) logical link used for control requirements. A single BIS can stream identical copies of data to multiple receiver devices.

CIS and CIG

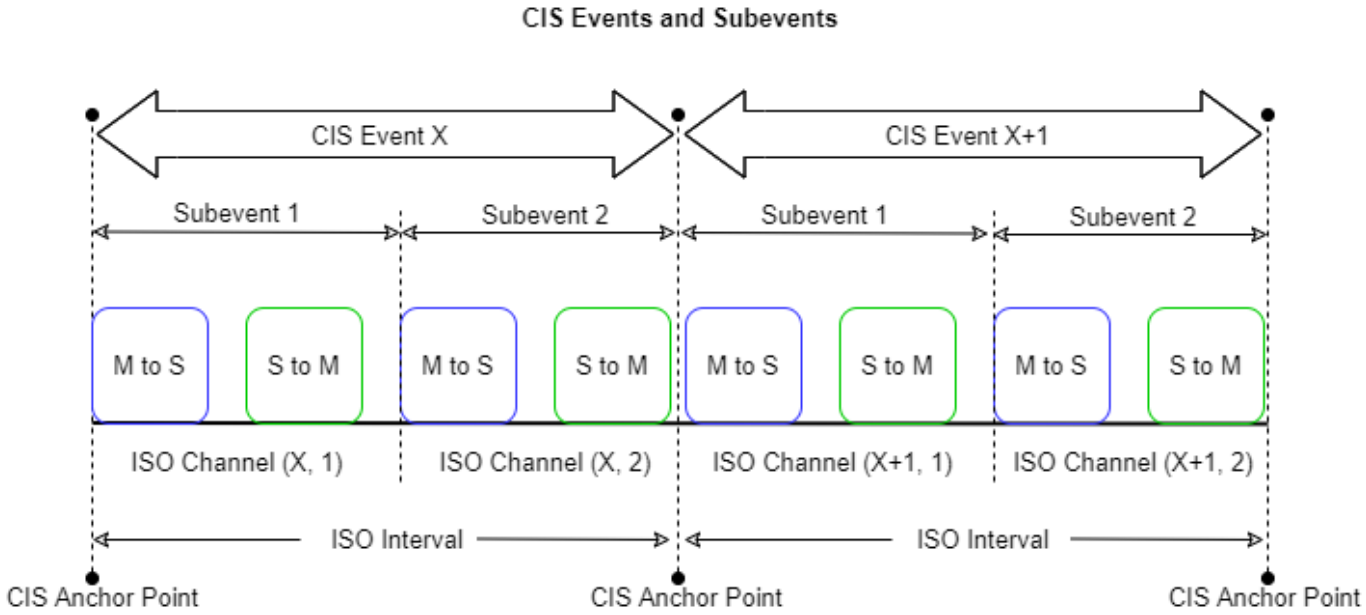
A CIS is a logical transport that enables connected devices to transfer isochronous data unidirectionally and bidirectionally. The isochronous data can be transferred either in an LE-S or LE-

F logical link by using the CIS logical transport. Each CIS is associated with an LE asynchronous connection (ACL). A CIS supports variable-size packets and the transmission of one or more packets in each isochronous event. This capability enables LE audio to support a range of data rates. A CIG consists of two or more CISs that have the same ISO interval (time between the anchor points of adjacent CISs) and that are expected to have a time relationship at the application layer, or of a single CIS. This figure shows the CIS and CIG servicing left and right stereo ear buds.



The maximum number of CISs in a CIG is 31. A CIG comes into existence when its first CIS is created, and it ceases to exist when all of its constituent CISs are destroyed.

This figure illustrates the concept of CIS events and subevents.



Each CIS event occurs at a regular ISO Interval, which is in the range from 5 ms to 4 s in multiples of 1.25 ms. Each CIS event is partitioned into one or more subevents. In a CIS, during a subevent, the Master (M) transmits once and the Slave (S) responds as shown in the preceding figure. A CIS event is an opportunity for the M and S to exchange CIS protocol data units (PDUs). A subevent ends at the end of the S's packet, if any, and at the end of the M's packet. The isochronous channel is changed at the end of each subevent. The LL closes a CIS event at the end of its last subevent.

All CISes in a CIG has the same M but may have different S's. A CIG event consists of the corresponding CIS events of the CISes currently making up that CIG. Each CIG event starts at the anchor point of the earliest (in transmission order) CIS of the CIG and ends at the end of the last subevent of the latest CIS of the same CIG event. Any two CIG events on the same CIG do not overlap because the last CIS event of a given CIG event ends before the first CIS anchor point of the next CIG event.

Consider a use case where an audio stream from a smartphone (the source) is to be played in the left and right buds (the two sinks) of LE earphones. The left and right buds each establish a CIS stream with the source device. Both the CIS streams are part of the same CIG. A fragment of audio produced by the source is encoded into a packet and a copy is transmitted to each sink device over its stream, one at a time during a series of consecutive CIS events. The audio playback must not start until all devices in the CIG have received the packet.

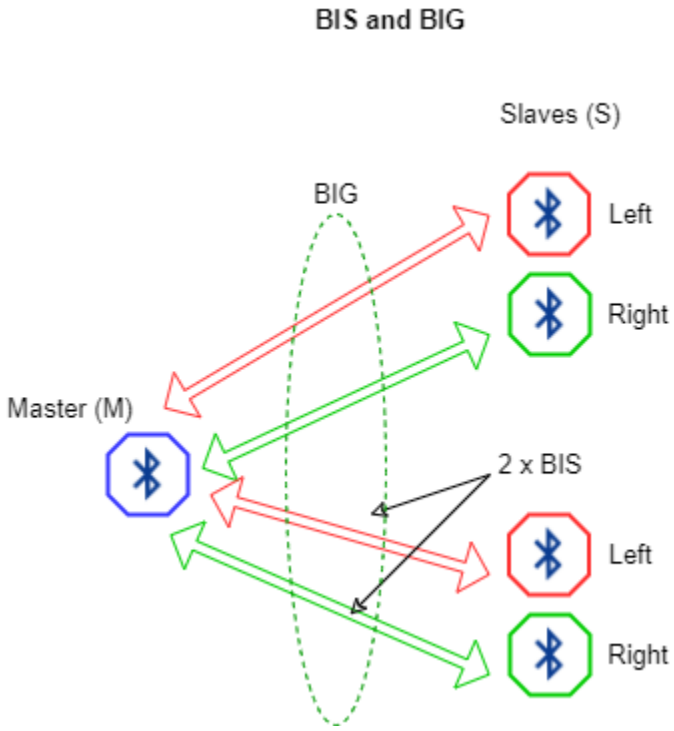
Note

- For more information about CIS, see Volume 6, Part B, Section 4.5.13 of the Bluetooth Core Specification 5.2 [2].
- For more information about CIG, see Volume 6, Part B, Section 4.5.14 of the Bluetooth Core Specification 5.2 [2].

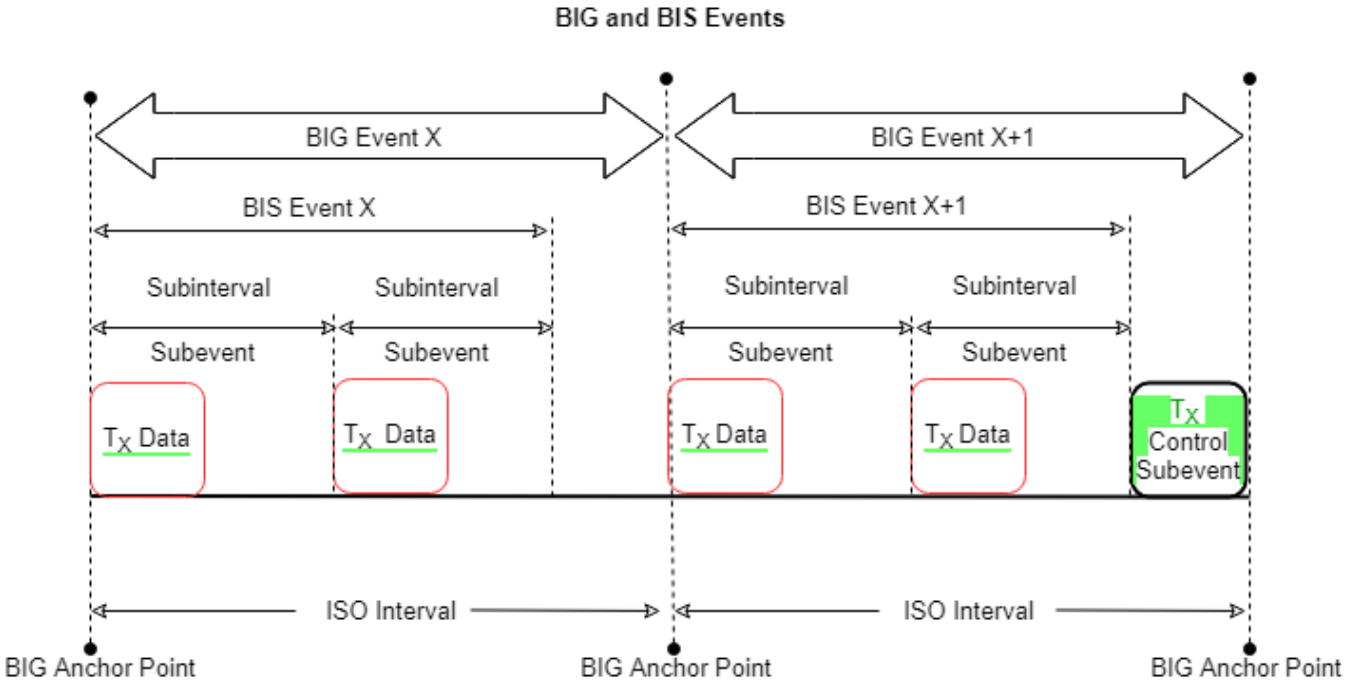
BIS and Broadcast Isochronous Group (BIG)

A BIS is a logical transport that enables a device to transfer isochronous data (framed or unframed). A BIS supports variable-size packets and the transmission of one or more packets in each isochronous event, enabling LE audio to support a range of data rates. The data traffic is unidirectional from the broadcasting device. Therefore, no acknowledgment protocol exists, making broadcast isochronous traffic unreliable. To improve the reliability of packet delivery, the BIS supports multiple retransmissions.

A BIG contains two or more BISs that have the same ISO interval and that are expected to have a time relationship at the application layer, or of a single BIS. The maximum number of BISs in a BIG is 31. This figure shows the BIS and BIG servicing a pair of left and right stereo ear buds.



For each BIS within a BIG, a schedule of transmission time slots (known as events and subevents) exist. This figure shows the concept of BIS and BIG events and subevents.



Each BIS event starts at the BIS anchor point and ends after its last subevent. Each BIG event starts at the BIG anchor point and ends after the control subevent, if one exists. If a control subevent does not exist, the BIG event ends at the end of the last BIS event. A BIS subevent enables an isochronous

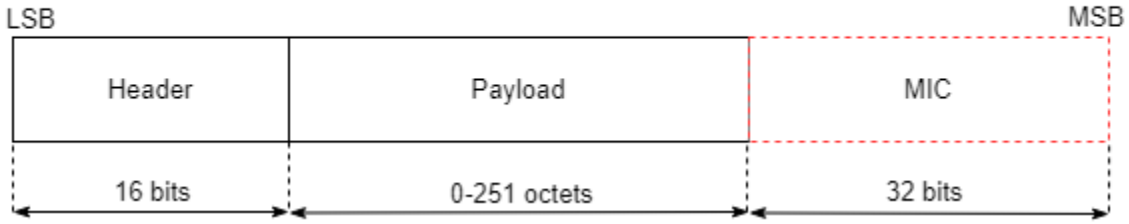
broadcaster to transmit a BIS PDU and enables a synchronized receiver to receive it. The LL must transmit one BIS data PDU at the start of each subevent of the isochronous broadcasting event. For each BIS event, the source of the data must send a burst of data consisting of burst number (BN) payloads. The subevents of each BIS event are each partitioned into groups of BN subevents. Each BIG event contains an optional control subevent. If a control subevent is present, the LL transmits a single BIG control PDU at the start of the control subevent to send control information about the BIG. The LL does not transmit a BIG Control PDU at any other time.

Note For more information about BIG and BIS, see Volume 6, Part B, Section 4.4.6 of the Bluetooth Core Specification 5.2 [2].

Isochronous Physical Channel Protocol Data Unit (PDU)

The isochronous physical channel PDU contains a 16-bit header, a variable size payload, and an optional message integrity check (MIC) field. This figure shows the packet structure of isochronous physical channel PDU.

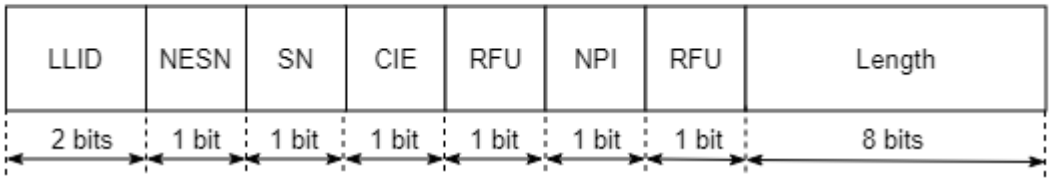
Isochronous Physical Channel PDU



The format of the Header and Payload fields depend on the type of isochronous physical channel PDU that is being used. The isochronous physical channel PDU is a CIS PDU or a BIS PDU when used on a CIS or BIS, respectively. The MIC field is included in all PDUs that contain a nonzero Payload transmitted on an encrypted CIS or BIS. If a PDU is sent on an nonencrypted CIS or BIS or has a zero-length Payload, then the MIC field is not present.

This figure shows the packet structure of an isochronous physical channel PDU Header for a CIS PDU.

Isochronous Physical Channel PDU Header for CIS PDU

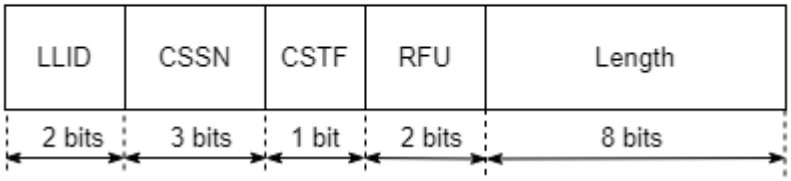


A CIS PDU can be a CIS data PDU or a CIS null PDU. A CIS Data PDU carries isochronous data, whereas a CIS null PDU is used when no data exists to be sent. This table explains the contents of the Header field.

Header Field Name	Description
LL identifier (LLID)	The LLID field indicates the type of content of the Payload field of the CIS Data PDU. These are the valid values of this field. • 0b00 – Unframed CIS data PDU (end fragment of an service data unit (SDU) or a complete SDU) • 0b01 – Unframed CIS data PDU (start or continuation fragment of an SDU) • 0b10 – Framed CIS data PDU (one or more segments of an SDU) • 0b11 – Reserved for future use For a CIS null PDU, the LLID is reserved for future use (RFU).
Next expected sequence number (NESN)	The LL uses this field to either acknowledge the last PDU sent by the peer device, or to request the peer device to resend the last PDU sent.
Sequence number (SN)	This field sets the identification number for LL packets. For a CIS null PDU, the SN is RFU.
Close isochronous event (CIE)	The device uses this field to close a CIS event early.
Null PDU indicator (NPI)	This field indicates whether the CIS PDU is a CIS data PDU or a CIS null PDU. If the CIS PDU is a CIS null PDU, then LL sets this field.
Length	This field indicates the size (in octets) of the Payload and MIC, if included.

This figure shows the packet structure of an isochronous physical channel PDU Header for a BIS PDU.

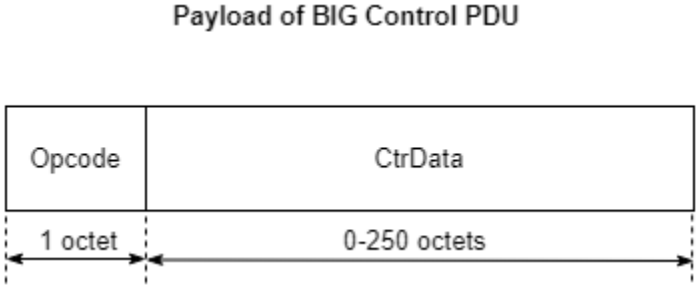
Isochronous Physical Channel PDU Header for BIS PDU



A BIS PDU can be a BIS data PDU or a BIG control PDU. A BIS data PDU carries isochronous data. A BIG control PDU sends control information for a BIG. This table explains the contents of the Header field.

Header Field Name	Description
LLID	The LLID field indicates the type of content of the Payload field of the BIS data PDU. These are the valid values of this field. • 0b00 – Unframed BIS data PDU (end fragment of an SDU or a complete SDU) • 0b01 – Unframed BIS data PDU (start or continuation fragment of an SDU) • 0b10 – Framed BIS data PDU (one or more segments of an SDU) • 0b11 – BIG control PDU
Control subevent sequence number (CSSN)	The LL uses this field to indicate the start of a BIG event that contains the first transmission of a new BIG control PDU.
Control subevent transmission flag (CSTF)	The LL uses this field to indicate whether it has scheduled a BIG control PDU to be transmitted in a BIG event.
Length	This field indicates the size (in octets) of the Payload and MIC, if included.

This figure shows the packet structure of the Payload field in a BIG control PDU.



The Opcode field specifies different types of BIG control PDUs. The Opcode field specifies the CtrData field in the Payload of BIG control PDU. For a given Opcode, the length of the CtrData field is fixed.

Note

- For more information about CIS PDU, see Volume 6, Part B, Section 2.6.1 of the Bluetooth Core Specification 5.2 [2]
- For more information about BIS PDU, see Section 2.6.2 of the Bluetooth Core Specification 5.2 [2]
- For more information about BIG control PDU, see Volume 6, Part B, Section 2.6.3 of the Bluetooth Core Specification 5.2 [2].

Isochronous Adaptation Layer (ISOAL)

To support LE audio, the Bluetooth Core Specification 5.2 [2] introduced the ISOAL in the Bluetooth stack that is present in the controller above the LL. The ISOAL enables the lower and upper layers of the stack to work together. This flexibility enables the size of isochronous data packets created and used by the upper layers to be distinct from the size used by the CIS or BIS logical transport in the LL. The ISOAL provides segmentation, fragmentation, reassembly, and recombination services for the

conversion of the SDUs from the upper layer to the PDUs of the baseband resource manager and vice versa. The ISOAL enables the upper layer to use timing intervals that differ from those used by the LL so that the rate of SDUs exchanged with the upper layers is not the same as the rate with which they are exchanged with the LL. The isochronous communication mechanism uses the host controller interface (HCI) as the interface from the upper layer to the ISOAL. The SDUs are transferred to and from the upper layer by either using HCI ISO data packets or over an implementation-specific transport.

Note For more information about ISOAL, see Volume 6, Part G of the Bluetooth Core Specification 5.2 [2].

Use Cases of LE Audio

This table shows some prominent use cases of LE audio.

Use Case	Description
Personal audio sharing	With personal audio sharing, people can share their Bluetooth audio experience with others around them. For example, group of friends can simultaneously enjoy music playing on one smartphone through their LE supported headphones. This is an example of a private group of audio sink devices sharing a single audio source.
Public assisted hearing	The dialogue of a theater play can be broadcast such that all LE hearing aid users in the audience can hear the dialogue.
Public television	At the gymnasium, all attendees with LE headphones or ear buds can listen to the television audio stream.
Multi-language flight announcements	Passengers at the airport or in an aircraft can connect their LE headphones to the flight information system, specify their preferred language, and listen to the flight information in that language.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 14, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification Version 5.2 Feature Overview. https://www.bluetooth.com/.

See Also

More About

- “What Is Bluetooth?” on page 13-2
- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23

- “Bluetooth Mesh Networking” on page 13-46
- “Bluetooth-WLAN Coexistence” on page 13-60

Comparison of Bluetooth BR/EDR and BLE Specifications

Bluetooth technology [1], operating on the 2.4 GHz unlicensed industrial, scientific, and medical (ISM) frequency band, uses low-power radio frequency to enable short-range communication at a low cost. The two variants of the Bluetooth technology are –

- Bluetooth basic rate/enhanced data rate (BR/EDR) or classic Bluetooth
- Bluetooth low energy (BLE) or Bluetooth Smart

The Bluetooth Core Specification [2], specified by the Special Interest Group (SIG) consortium, defines the technologies required to create interoperable Bluetooth BR/EDR and BLE devices.

Bluetooth BR/EDR radio is primarily designed for low power, high data throughput operations. In Bluetooth BR/EDR, the radio hops in a pseudo-random way on 79 designated Bluetooth channels. Each Bluetooth BR/EDR channel has a bandwidth of 1 MHz. Each frequency is located at (2402 + k) MHz, where k = 0,1, ..., 78.

In 2010, the SIG introduced BLE with the Bluetooth 4.0 version. The BLE radio is designed and optimized to support applications and use cases that have a relatively low duty cycle. For example, suppose a person wears a heart rate monitoring device for several hours. Because this device transmits only a few bytes of data every second, its radio is in the 'on' state for a very short period of time. In BLE, the operating radio frequency is in the range from 2.4000 GHz to 2.4835 GHz. The channel bandwidth is 2 MHz, and the operating band is divided into 40 channels, (k = 0, 1, ..., 39). The center frequency of the kth channel is located at (2402 + k × 2) MHz.

This table summarizes and compares different features of Bluetooth BR/EDR and BLE.

Feature	Bluetooth BR/EDR	BLE
Frequency band	Operates on a 2.4 GHz Industrial, Scientific, and Medical (ISM) band, with the values in the range from 2.4000 GHz to 2.4835 GHz	Operates on 2.4 GHz ISM band, with the values in the range from 2.4000 GHz to 2.4835 GHz
Channels	79 channels	40 channels (37 data channels and 3 advertising channels)
Channel bandwidth	1 MHz	2 MHz
Spread spectrum technique	1600 hops/sec frequency-hopping spread spectrum (FHSS)	FHSS
Modulation scheme	• Gaussian frequency shift keying (GFSK) • π/4 differential quadrature phase shift keying (DQPSK) • 8 differential phase shift keying (DPSK)	GFSK
Power usage	1 W (reference value)	~0.01x W to 0.5x W of reference (depending on the use case scenario)

Feature	Bluetooth BR/EDR	BLE
Maximum transmission power	• Class 1: 100 mW (20 dBm) • Class 2: 2.5 mW (4 dBm) • Class 3: 1 mW (0 dBm)	• Class 1: 100 mW (20 dBm) • Class 1.5: 10 mW (10 dBm) • Class 2: 2.5 mW (4 dBm) • Class 3: 1 mW (0 dBm)
Data rate	• BR PHY (GFSK): 1 Mb/s • EDR PHY (π/4 DQPSK): 2 Mb/s • EDR PHY (8 DPSK): 3 Mb/s	• LE Coded PHY (S = 8): 125 Kb/s • LE Coded PHY (S = 2): 500 Kb/s • LE 1M PHY: 1 Mb/s • LE 2M PHY: 2 Mb/s
Device discovery	Inquiry or paging	Advertising
Device address privacy	None	Private device addressing supported
Encryption algorithm	E0/SAFER+	AES-CCM
Audio capable	Yes	Yes (BLE audio is introduced in Bluetooth Core Specification 5.2)
Network topology	Point-to-point (including piconet)	• Point-to-point (including piconet) • Broadcast • Mesh

This table summarizes prominent applications of Bluetooth BR/EDR and BLE.

Application	Bluetooth BR/EDR	BLE
Audio streaming applications such as: • Bluetooth headphones or earbuds • Bluetooth speakers • Bluetooth watches	Supported	Supported
Location and direction finding applications such as: • Asset tracking • Indoor navigation services • Beacon-based services	Not supported	Supported
Data transmission applications such as: • Medical and health equipments • Sports and fitness equipments • Peripherals and accessories	Not supported	Supported

Application	Bluetooth BR/EDR	BLE
Device network applications such as: • Monitoring systems and services • Automation systems • Control systems	Not supported	Supported

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 14, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

More About

• “What Is Bluetooth?” on page 13-2
• “Bluetooth Protocol Stack” on page 13-7
• “Bluetooth Packet Structure” on page 13-23
• “Bluetooth Mesh Networking” on page 13-46
• “Bluetooth-WLAN Coexistence” on page 13-60
• “Bluetooth Location and Direction Finding” on page 13-37

Create, Configure, and Visualize BLE Mesh Network

Communications Toolbox Library for the Bluetooth Protocol features enable you to create, configure, and visualize a Bluetooth low energy (BLE) mesh network. For information about BLE mesh networking, see “Bluetooth Mesh Networking” on page 13-46.

Create, Configure, and Visualize BLE Mesh Network

Specify the total number of BLE mesh nodes.

```
totalNodes = 6;
```

Create a BLE mesh node by using the helperBLEMeshNode object. This helper function creates a BLE mesh node object and models the complete protocol stack in a mesh node.

```
meshNodes(1,totalNodes) = helperBLEMeshNode();                % Create a list of mesh nodes
for nodeIdx = 1:totalNodes
    meshNode = helperBLEMeshNode();                           % Object for a mesh node
    meshNode.Identifier = nodeIdx;                            % Unique identifier for a mesh node
    meshNodes(nodeIdx) = meshNode;                            % Assign node to the list
end
```

Configure the mesh nodes as source, destination, and relay. Node 6 is termed as the end node.

```
sourceDestinationPairs = [1 4; 2 5];
relayNodeIDs = 3;
```

Assign positions to the mesh nodes by using one of these options.

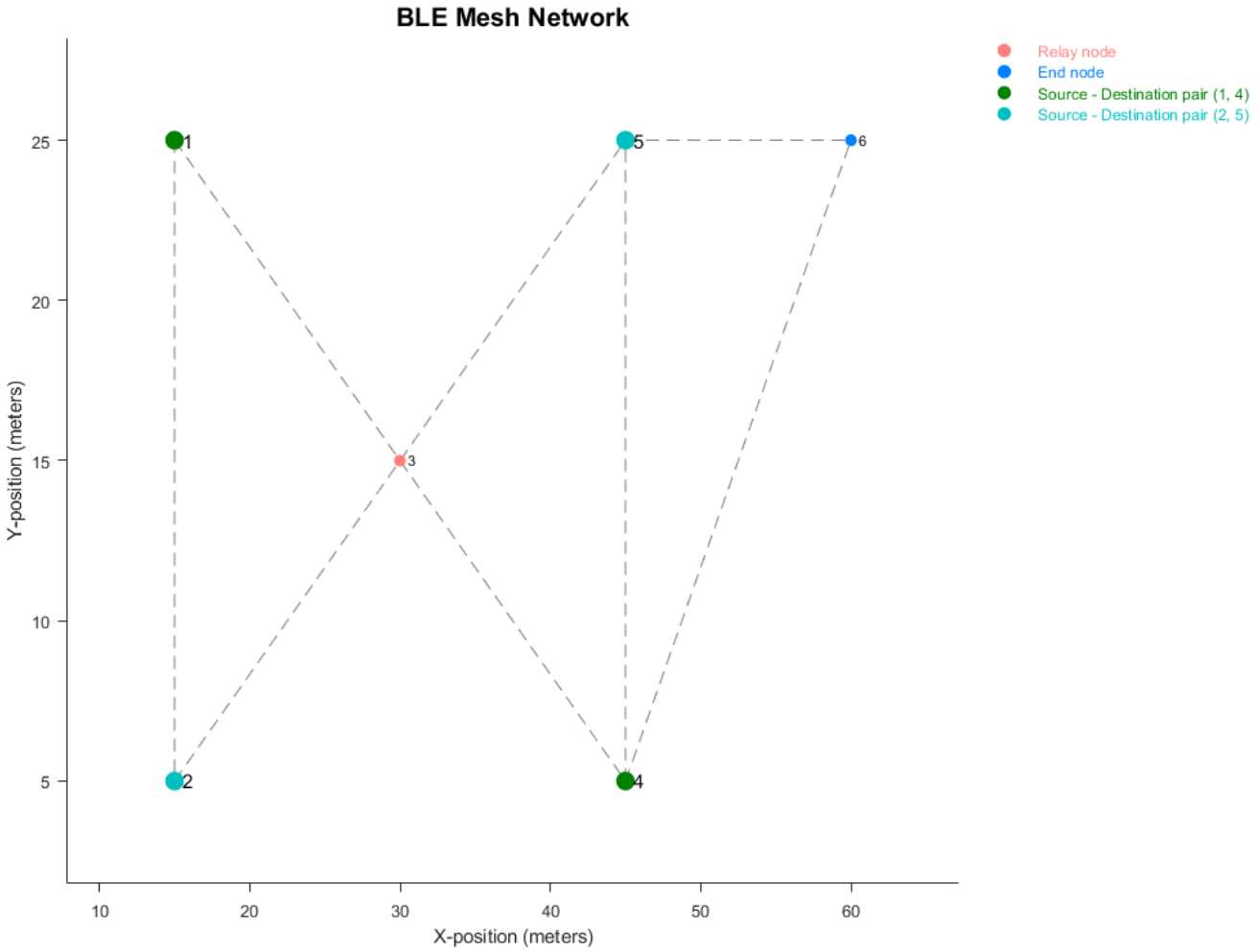
- Specify an *n*-by-2 matrix, where *n* is the total number of mesh nodes. Each row in the matrix represents the *x*- and *y*-coordinate of the mesh node. To use this option, the `NodePositionType` property of the `helperBLEMeshVisualizeNetwork` object must be set to `'UserInput'`.
- Load a `.mat` file containing node positions into the workspace. To use this option, you must set the `NodePositionType` property of the `helperBLEMeshVisualizeNetwork` to `'UserInput'`.

For this example, assign positions to the mesh nodes by specifying six (x, y) coordinates as a matrix.

```
bleMeshNodesPositions = [15 25; 15 5; 30 15; 45 5; 45 25; 60 25];
```

Visualize the BLE mesh network by using the helperBLEMeshVisualizeNetwork function. This helper function creates a BLE mesh network visualization object with configurable properties.

```
meshNetworkGraph = helperBLEMeshVisualizeNetwork();          % Object for BLE mesh network visual
meshNetworkGraph.NumberOfNodes = totalNodes;                 % Total number of mesh nodes
meshNetworkGraph.NodePositionType = 'UserInput';             % Option to assign node position
meshNetworkGraph.Positions = bleMeshNodesPositions;          % List of all node positions
meshNetworkGraph.VicinityRange = 25;                         % Transmission and reception range o
meshNetworkGraph.Title = 'BLE Mesh Network';                 % Title of plot
meshNetworkGraph.SrcDstPairs = sourceDestinationPairs;       % Source-destination pair
meshNetworkGraph.NodeState = [1 1 2 1 1 1];                  % State of mesh node
meshNetworkGraph.DisplayProgressBar = false;                 % Display progress bar
meshNetworkGraph.createNetwork();                            % Display mesh network
```



References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 25, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Profile." Version 1.0.1. https://www.bluetooth.com/.

[4] Bluetooth Special Interest Group (SIG). "Bluetooth Mesh Model Specification." Version 1.0.1. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth Mesh Networking” on page 13-46
- “Energy Profiling of Bluetooth Mesh Nodes in Wireless Sensor Networks” on page 3-136
- “Bluetooth Mesh Flooding in Wireless Sensor Networks” on page 3-159
- “Estimate Packet Delivery Ratio in Bluetooth Mesh Network” on page 3-117

Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform

Communications Toolbox Library for the Bluetooth Protocol features enable you to model a wireless channel that is shared between Bluetooth basic rate/enhanced data rate (BR/EDR) and WLAN. The library also provides functionalities to add WLAN interference to the Bluetooth BR/EDR waveform. For information about how Bluetooth and WLAN coexists in the unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band, see “Bluetooth-WLAN Coexistence” on page 13-60.

Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform

This example shows you how to create a Bluetooth BR/EDR channel and configure its applicable properties. Then, specify the source of WLAN interference and add the WLAN signal to the Bluetooth BR/EDR channel. Finally, generate a Bluetooth BR/EDR waveform and pass the waveform through the channel.

Create and Configure Bluetooth BR/EDR Channel with WLAN Interference

Configure a Bluetooth BR/EDR channel by using the helperBluetoothChannel object, which creates a Bluetooth BR/EDR channel model object with configurable properties.

```
bluetoothBREDRChannel = helperBluetoothChannel
```

```
bluetoothBREDRChannel =
  helperBluetoothChannel with properties:

    ChannelIndex: 0
            FSPL: 1
    NodePosition: [0 0 0]
            EbNo: 10
             SIR: 0
```

Set the ratio of energy per bit to noise power spectral density (Eb/No) for the additive white Gaussian noise (AWGN) channel to 22 dB. Specify signal to interference ratio (SIR) as -15 dB.

```
bluetoothBREDRChannel.EbNo = 22;
bluetoothBREDRChannel.SIR = -15;
```

Specify the source of WLAN interference by using the `wlanInterference` property. Use one of these options to specify the source of the WLAN interference.

- `'Generated'`: To add a WLAN (802.11b) signal (requires the WLAN Toolbox™ software), select this option.
- `'BasebandFile'`: To add a WLAN signal from a baseband file (`.bb`), select this option. You can specify the file name using the `wlanBBFilename` input argument. If you do not specify the `.bb` file, the example uses the default `.bb` file, `'WLANNonHTDSSS.bb'`, to add the WLAN signal.
- `'None'`: To disable WLAN interference, select this option.

Specify the source of WLAN interference as `'BasebandFile'`and specify a baseband file.

```
wlanInterference = BasebandFile;
wlanBBFilename =  'WLANNonHTDSSS.bb';
```

Generate the WLAN signal interference, by using the helperBluetoothGenerateWLANWaveform function. Add the generated WLAN signal interference to the Bluetooth BR/EDR channel.

```
if ~strcmpi(wlanInterference,'None')
    wlanWaveform = helperBluetoothGenerateWLANWaveform(wlanInterference,wlanBBFilename);
    addWLANWaveform(bluetoothBREDRChannel,wlanWaveform);
end
```

Generate and Pass Bluetooth BR/EDR Waveform Through the Channel

Create a Bluetooth BR/EDR signal structure, specifying different configurable properties of the waveform.

```
bluetoothSignal = struct(...
    'PacketType','DM1',...                  % Packet type
    'Waveform',[],...                       % Waveform
    'NumSamples',[],...                     % Number of samples
    'SampleRate',1e6,...                    % Sample rate
    'SamplesPerSymbol',8,...                % Samples per symbol
    'Payload',zeros(1,3200), ...            % Payload
    'PayloadLength',0, ...                  % Payload length
    'SourceID',0,...                        % Source identifier
    'Bandwidth',1,...                       % Bandwidth
    'NodePosition',[0 0 0],...              % Node position
    'CenterFrequency',2402,...              % Center frequency
    'StartTime',0,...                       % Waveform start time
    'EndTime',0,...                         % Waveform end time
    'Duration',0);...                       % Waveform duration
```

Create a Bluetooth BR/EDR waveform configuration object. Specify the packet type as HV1.

```
cfg = bluetoothWaveformConfig;
cfg.PacketType = 'HV1';
```

Create a bit vector containing concatenated payloads.

```
numBits = getPayloadLength(cfg)*8;              % Byte to bit conversion
message = randi([0 1],numBits,1);
```

Generate a Bluetooth BR/EDR waveform.

```
txWaveform = bluetoothWaveformGenerator(message,cfg);
```

Pass the generated waveform through the Bluetooth BR/EDR channel.

```
bluetoothSignal.Waveform = txWaveform;
bluetoothSignal.NumSamples = numel(txWaveform);
bluetoothSignal = run(bluetoothBREDRChannel,bluetoothSignal,cfg.Mode);
wirelessWaveform = bluetoothSignal.Waveform;
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 17, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

[3] *P802.15.2/D09 - IEEE Draft Recommended Practice for Information Technology Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks Specific Requirements - Part 15.2: Coexistence of Wireless Personal Area Networks With Other Wireless Devices Operating in Unlicensed Frequency Bands.* LAN/MAN Standards Committee, IEEE Computer Society, 2003, https://ieeexplore.ieee.org/document/4040972.

See Also

Functions
`bluetoothWaveformGenerator`

Objects
`bluetoothWaveformConfig`

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Bluetooth-WLAN Coexistence” on page 13-60
- “Configure BLE Channel and Pass Waveform” on page 13-99
- “BLE Coexistence Model with WLAN Signal Interference” on page 3-175
- “Statistical Modeling of WLAN Interference on BLE Network” on page 3-198
- “End-to-End Bluetooth BR/EDR PHY Simulation with WLAN Interference and Adaptive Frequency Hopping” on page 3-76

Configure BLE Channel and Pass Waveform

Communications Toolbox Library for the Bluetooth Protocol features enable you to create and configure a Bluetooth low energy (BLE) channel.

Configure BLE Channel and Pass Waveform

Configure a BLE channel by using the helperBLEChannel object. This helper object enables you to configure the applicable properties of the BLE channel.

Create a BLE channel model object with default properties.

```
bleChannel = helperBLEChannel
```

```
bleChannel =
  helperBLEChannel with properties:

            ChannelIndex: 37
                 RxRange: 10
    RangePropagationLoss: 1
               FSPLModel: 1
            NodePosition: [0 0 0]
```

Set the (*x*, *y*, *z*) position coordinates of the nodes. Specify the receiving range (in meters) of the nodes.

```
bleChannel.RxRange = 15;
bleChannel.NodePosition = [5 0 0];
```

Create an input message column vector of length 2056 containing random binary values.

```
message = randi([0 1],2056,1);
symbolRate = 1e6;
```

Set the BLE waveform frequency (in MHz). Specify the position coordinates of the transmitter.

```
waveformFrequency = 2402;
transmitterPos = [18 0 0];
```

Generate the BLE waveform.

```
txWaveform = bleWaveformGenerator(message);
```

Pass the generated BLE waveform through the BLE channel.

```
wirelessWaveform = run(bleChannel,txWaveform,waveformFrequency,transmitterPos);
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 17, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Configure Bluetooth BR/EDR Channel with WLAN Interference and Pass the Waveform” on page 13-96

Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH

Communications Toolbox Library for the Bluetooth Protocol features enable you to create and configure a Bluetooth piconet. The library provides functionalities to configure asynchronous connection-oriented (ACL) link, synchronous connection-oriented (SCO) link, or both between the Master and the Slave. You can also configure the frequency hopping techniques as basic frequency hopping or adaptive frequency hopping (AFH).

Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH

Configure the simulation parameters of the Bluetooth piconet by creating a structure. Specify the number of Slaves in the piconet. A piconet can contain a maximum of seven Slaves.

```
simulationParameters = struct;
simulationParameters.NumSlaves = 2 ;
```

Calculate the total number of nodes in the piconet (one Master and multiple Slaves).

```
numNodes = simulationParameters.NumSlaves + 1;
```

Specify the type of logical link between the Master and Slaves. Valid logical link values depend on how many Slaves are connected to the Master.

- If the Master is connected to one Slave, you must specify the logical link value as a one-element vector of 1 (ACL link), 2 (SCO link), or 3 (ACL and SCO links).
- If the Master is connected to multiple Slaves, you must specify the logical link value as an *n*-element row vector, where *n* is the number of Slaves. Each element must be 1 (ACL link), 2 (SCO link), or 3 (ACL and SCO links).

To enable an ACL logical transport, set the logical link traffic to 1 or 3. You can specify the ACL packet type as `'DM1'`, `'DH1'`, `'DM3'`, `'DH3'`, `'DM5'`, or `'DH5'`. To enable an SCO logical transport, set the logical link traffic to 2 or 3. You can specify the SCO packet type as `'HV1'`, `'HV2'`, or `'HV3'` for the respective Slave that has SCO link traffic.

Enable ACL and SCO traffic, specifying the type of ACL and SCO packet as `'DM1'` and `'HV3'`, respectively.

```
simulationParameters.LinkTraffic = [1 2];
simulationParameters.ACLPacketType = DM1 ;
simulationParameters.SCOPacketType = {2,'HV3'};
```

To enable AFH, set the sequence type to Connection adaptive.

```
simulationParameters.SequenceType = Connection adaptive ;
```

Initialize a cell array to store the Bluetooth nodes.

```
btNodes = cell(1,numNodes);
```

Specify the distance (in meters) between two Bluetooth nodes.

```
interNodeDistance = 10;
```

Set the positions of the Bluetooth nodes.

```
simulationParameters.NodePositions = zeros(numNodes,3);
for nodeIdx = 1:numNodes
    simulationParameters.NodePositions(nodeIdx,:) = [nodeIdx*interNodeDistance 0 0];      % Set n
end
```

Set the node configuration parameters related to the wireless channel and channel classification.

```
simulationParameters.EbNo = 22;                              % Ratio of energy per bit (Eb) to spec
simulationParameters.WLANInterference = 'None';
simulationParameters.SIR = [-15 -16 -14 -13 -12 -11 -10];  % Signal to interference ratio in dB
simulationParameters.PERThreshold = 40;                      % Packet error rate
simulationParameters.ClassificationInterval = 3000;          % Classification interval in slots
simulationParameters.RxStatusCount = 10;                     % Status of maximum number of received
simulationParameters.MinRxCountToClassify = 4;               % Status of minimum number of received
simulationParameters.PreferredMinimumGoodChannels = 20;      % Preferred number of good channels re
simulationParameters.TxPower = 20;                           % Transmit power in dBm
simulationParameters.ReceiverRange = 40;                     % Bluetooth node receiver range in met
```

Create a Bluetooth piconet by using the `helperBluetoothCreatePiconet` function.

```
helperBluetoothCreatePiconet(simulationParameters);
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 17, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Packet Distribution in Bluetooth Piconet” on page 13-106
- “Bluetooth Full Duplex Data and Voice Transmission in MATLAB” on page 3-51

Generate BLE Waveform and Add RF Impairments

Communications Toolbox Library for the Bluetooth Protocol features enable you to add radio frequency (RF) impairments to a Bluetooth low energy (BLE) or Bluetooth basic rate/enhanced data rate (BR/EDR) waveform.

Generate BLE Waveform and Add RF Impairments

Specify the data length. Create a message column vector of the specified data length containing random binary values.

```
dataLength = 2056;
message = randi([0 1],dataLength,1);
symbolRate = 1e6;
```

Specify the values of the physical layer (PHY) mode, channel index, samples per symbol, and access address.

```
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].';
```

Generate the BLE waveform.

```
txWaveform = bleWaveformGenerator(message,'Mode',phyMode,'SamplesPerSymbol',sps,'ChannelIndex',ch
```

Initialize the RF impairments for the specified PHY mode and samples per symbol by using the helperBLEImpairmentsInit function. The helper function returns a structure with phase frequency offset and variable fractional delay fields.

```
initRFImp = helperBLEImpairmentsInit(phyMode,sps)
```

```
initRFImp = struct with fields:
         pfo: [1x1 comm.PhaseFrequencyOffset]
    varDelay: [1x1 dsp.VariableFractionalDelay]
```

Specify the values of the frequency offset and phase offset.

```
initRFImp.pfo.FrequencyOffset = 150;
initRFImp.pfo.PhaseOffset = -1;
```

Specify the values of static timing offset, timing drift, variable timimg offset, and DC offset.

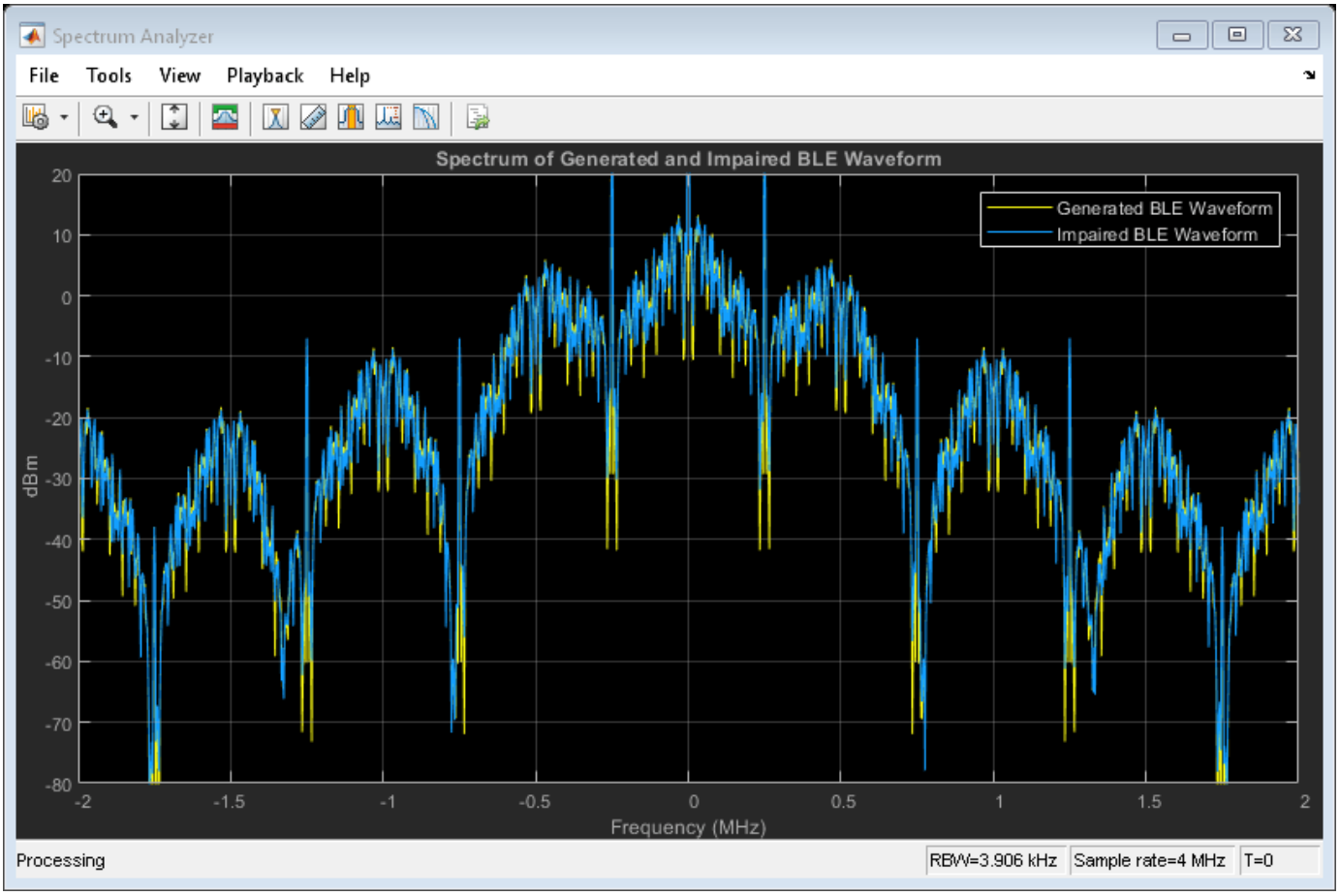
```
staticTimingOff = 0.15*sps;
timingDrift = 10;
initRFImp.vdelay = (staticTimingOff:timingDrift:staticTimingOff + timingDrift * (length(txWavefo
initRFImp.dc = 6 20;
```

Add RF impairments to the generated BLE waveform by using the helperBLEImpairmentsAddition function.

```
txImpairedWaveform = helperBLEImpairmentsAddition(txWaveform,initRFImp);
```

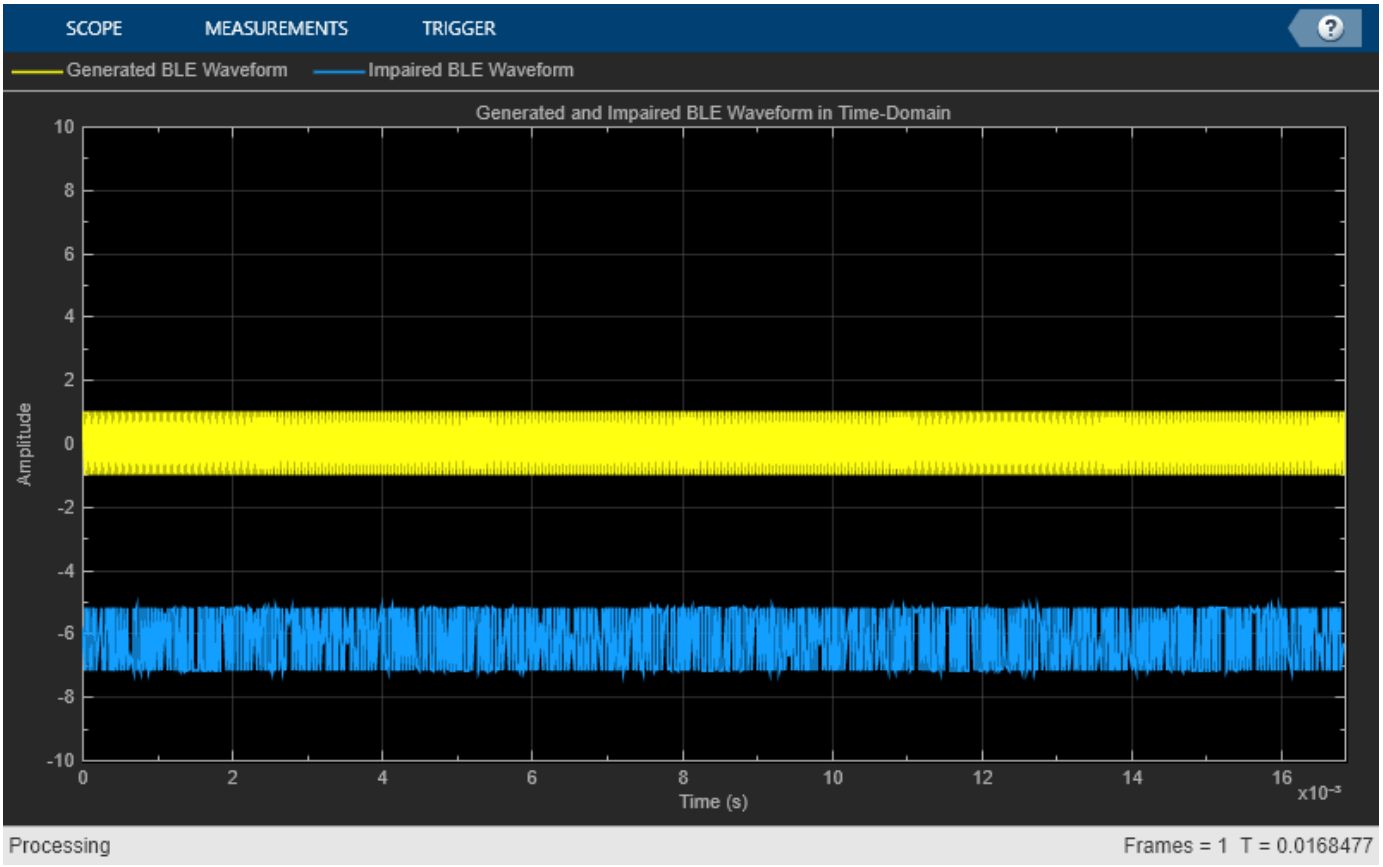
Create a default `dsp.SpectrumAnalyzer` System object. Then, set the sample rate of the frequency spectrum. Visualize the generated and impaired BLE waveform in the spectrum analyzer.

```
scope = dsp.SpectrumAnalyzer;
scope.SampleRate = sps*symbolRate;
scope.NumInputPorts = 2;
scope.Title = 'Spectrum of Generated and Impaired BLE Waveform';
scope.ShowLegend = true;
scope.ChannelNames = {'Generated BLE Waveform','Impaired BLE Waveform'};
scope(txWaveform,txImpairedWaveform);
```



Visualize the generated and impaired BLE waveform in time-domain by using the `timescope` object.

```
timeScope = timescope('SampleRate',symbolRate*sps,'TimeSpanSource','Auto','ShowLegend',true);
timeScope.Title = 'Generated and Impaired BLE Waveform in Time-Domain';
timeScope.ChannelNames = {'Generated BLE Waveform','Impaired BLE Waveform'};
timeScope(real(txWaveform),real(txImpairedWaveform));
```



References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 27, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “End-to-End BLE PHY Simulation Using Path Loss Model, RF Impairments, and AWGN” on page 3-90
- “End-to-End Bluetooth BR/EDR PHY Simulations with RF Impairments and Corrections” on page 3-110
- “End-to-End Bluetooth Low Energy PHY Simulation with RF Impairments and Corrections” on page 3-275

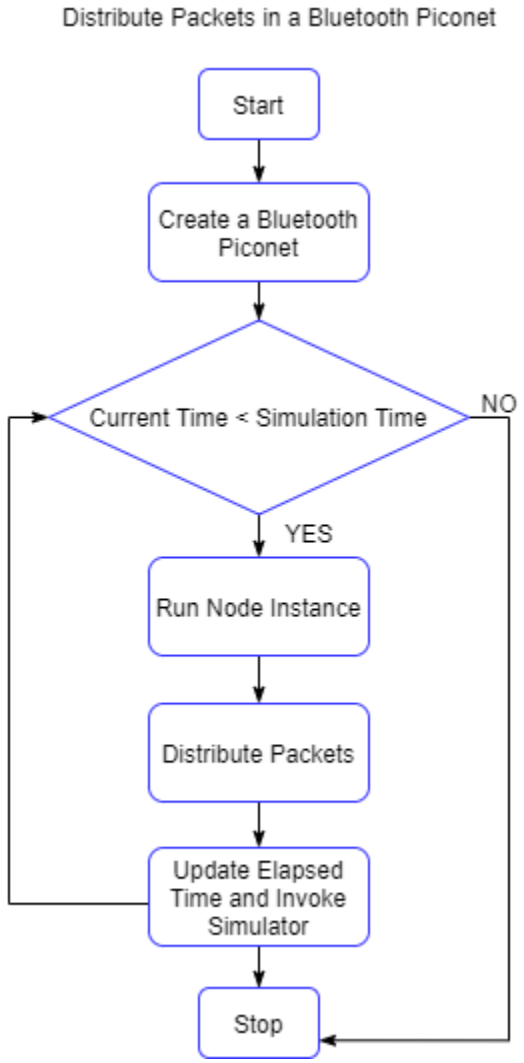
Packet Distribution in Bluetooth Piconet

Communications Toolbox Library for the Bluetooth Protocol features enable you to distribute packets in a Bluetooth piconet using discrete time simulation (DTS). In DTS, you can call the node only if it has an operation to perform. To increase the speed of the simulation, the DTS implements these two core time values.

- Next invoke time — At this time, the simulator runs all of the node instances. This value is given by each node through discrete time operations such as sending data, receiving data, retransmissions, or transferring data from the higher layer to lower layer. The simulator is called at the time that is the minimum of the next invoke time values given by each node.
- Elapsed time — This value is the time elapsed between the last and current call of the simulator.

Packet Distribution

To distribute packets in a Bluetooth piconet using DTS, follow these steps.



Create Bluetooth Piconet

Create a Bluetooth piconet and configure the nodes as Master and Slaves. For information about how to create a Bluetooth piconet, see “Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH” on page 13-101. In that example, a cell array, `btNodes`, is created representing the Bluetooth piconet. `btNodes` contains all of the nodes in the piconet with the complete stack enabled.

Run Node Instance

Set the simulation time, current time, elapsed time, and next invoke time.

```
simulationTime = 2*1e6;                                                    % In microseconds
currentTime = 0;
nextInvokeTime = zeros(1,numel(btNodes));
```

Simulate the Bluetooth nodes by running node instance for each Bluetooth node.

```
while(curTime < simulationTimeInUs)
  for nodeIdx = 1:numel(btNodes)                                            % Simulate all the Bl
    nextInvokeTimes(nodeIdx) = runNode(btNodes{nodeIdx},elapsedTime);      % Run the Bluetooth no
  end
```

Distribute Packets

To distribute packets from each node to the receiving buffer of other nodes, use `helperBluetoothDistributePackets` function. This helper function accepts `btNodes` as an input and returns the transmission flag, `isPacketDistributed`, indicating whether the channel is free or not.

```
isPacketDistributed = helperBluetoothDistributePackets(btNodes);
```

Update Elapsed Time and Invoke Simulator

Based on the transmission flag, update the elapsed time. If no packets are to be distributed, update the elapsed time to the next event at a node.

```
  if isPacketDistributed
    elapsedTime = 0;
  else
    elapsedTime = min(nextInvokeTimes(nextInvokeTimes ~= -1));
  end
end
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 27, 2020. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.2. https://www.bluetooth.com/.

See Also

More About

- “Bluetooth Protocol Stack” on page 13-7
- “Bluetooth Packet Structure” on page 13-23
- “Create Bluetooth Piconet by Enabling ACL Traffic, SCO Traffic, and AFH” on page 13-101

Equalization

Equalization

In this section...

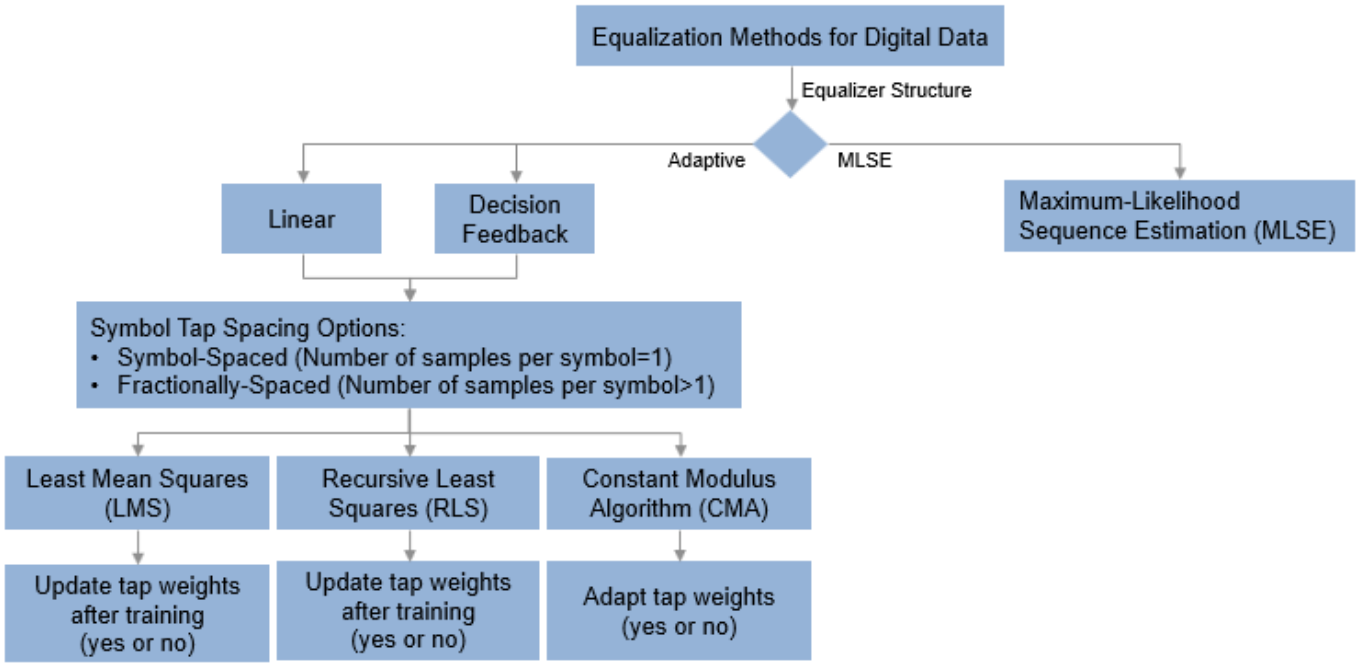
In a multipath fading scattering environment, the receiver typically detects several constantly changing, delayed versions of the transmitted signal. These time-dispersive channels cause intersymbol interference (ISI) that occurs when symbols received from multiple paths are delayed and overlap in time. ISI causes high error rates because the symbols from multiple received paths interfere with each other and become indistinguishable by the receiver.

Equalizers attempt to mitigate ISI and improve the receiver performance. Equalizer structures are filters that attempt to match the propagation channel response. For time-varying propagation channels, adapting the equalization filter tap weights so that they maintain a match to the channel over time improves the error rate performance.

Equalizer Structure Options

The Communications Toolbox includes System objects and blocks to recover transmitted data using by linear, decision-feedback, or maximum-likelihood sequence estimation (MLSE) equalization structures. For more information, see “Selected References for Equalizers” on page 14-3.

This figure shows the high-level configuration options for each equalization structure.



For each equalizer structure, you can configure structural settings (such as the number of taps and initial set of tap weights), algorithmic settings (such as the step size), and the signal constellation used by the modulator in your design. You also specify adaptability of the equalizer tap weights throughout the simulation.

- Linear and decision-feedback filter equalizer structures adapt tap weights by using the LMS, RLS, or CMA adaptive algorithm. When using these equalizer structures, the number of samples per symbol determines whether symbols are processed using whole or fractional symbol spacing.
  - When using LMS and RLS adaptive algorithms, the equalizer begins operating in tap weights training mode. Configure the equalizer to operate adaptively in decision-directed mode or without further adjustment of taps after training is completed.
  - When using the CMA adaptive algorithm, the equalizer has no training mode. You can configure the equalizer to operate adaptively in decision-directed mode or in nonadaptive mode.

  To explore the linear and decision-feedback filter equalizer capabilities, see “Adaptive Equalizers” on page 14-5.
- Maximum-Likelihood Sequence Estimation (MLSE) equalizers use the Viterbi algorithm. The MLSE equalization structure provides the optimal match to the received symbols but it requires an accurate channel estimate and is the most computationally complex structure. To explore MLSE equalizer capabilities, see “MLSE Equalizers” on page 14-36.

The computational complexity of each equalization structure grows with the length of the channel time dispersion. Considering the Doppler and frequency selectivity characteristics of the channel, use the information in this table when selecting which equalization structure to use in your simulation.

Equalizer Structure	Doppler Speed	Is Channel Frequency Selective?	Computational Complexity
Linear RLS	High	No	Medium
Linear LMS	Low	No	Lowest
Linear CMA	Low	No	Lowest
DFE RLS	High	Yes	Medium
DFE LMS	Low	Yes	Lowest
DFE CMA	Low	Yes	Lowest
MLSE	Low	Yes	Highest

Selected References for Equalizers

[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.

[5] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, John Wiley & Sons, 1996.

See Also

Objects
`comm.LinearEqualizer` | `comm.DecisionFeedbackEqualizer` | `comm.MLSEEqualizer`

Blocks
Linear Equalizer | Decision Feedback Equalizer | MLSE Equalizer

More About

- “Adaptive Equalizers” on page 14-5
- “MLSE Equalizers” on page 14-36

Adaptive Equalizers

In this section...

Adaptive equalizer structures provide suboptimal equalization of time variations in the propagation channel characteristics. However, these equalizers are appealing because their computational complexity is lower than “MLSE Equalizers” on page 14-36.

In Communications Toolbox, the `comm.LinearEqualizer` and `comm.DecisionFeedbackEqualizer` System objects and the Linear Equalizer and Decision Feedback Equalizer blocks use tap delay line filters to equalize a linearly modulated signal through a dispersive channel. These features output the estimate of the signal by using an estimate of the channel modeled as a finite input response (FIR) filter.

To decode a received signal, the adaptive equalizer:

1. Applies the FIR filter to the symbols in the input signal. The FIR filter tap weights correspond to the channel estimate.
2. Outputs the signal estimate and uses the signal estimate to update the tap weights for the next symbol. The signal estimate and updating of weights depends on the adaptive equalizer structure and algorithm.

Adaptive equalizer structure options are linear or decision-feedback. Adaptive algorithm options are least mean square (LMS), recursive mean square (RMS), or constant modulus algorithm (CMA). For background material on adaptive equalizers, see “Selected References for Equalizers” on page 14-3.

Number of Taps

For the linear equalizer, the number of taps must be greater than or equal to the number of input samples per symbol. For the decision feedback equalizer, the number of forward taps must be greater than or equal to the number of input samples per symbol.

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.

Note The MLSE equalizer supports fractional symbol spacing but using it is not recommended. The MLSE computational complexity and burden grows exponentially with the length of the channel time dispersion. Oversampling the input means multiplying the exponential term by the number of samples per symbol.

Linear Equalizers

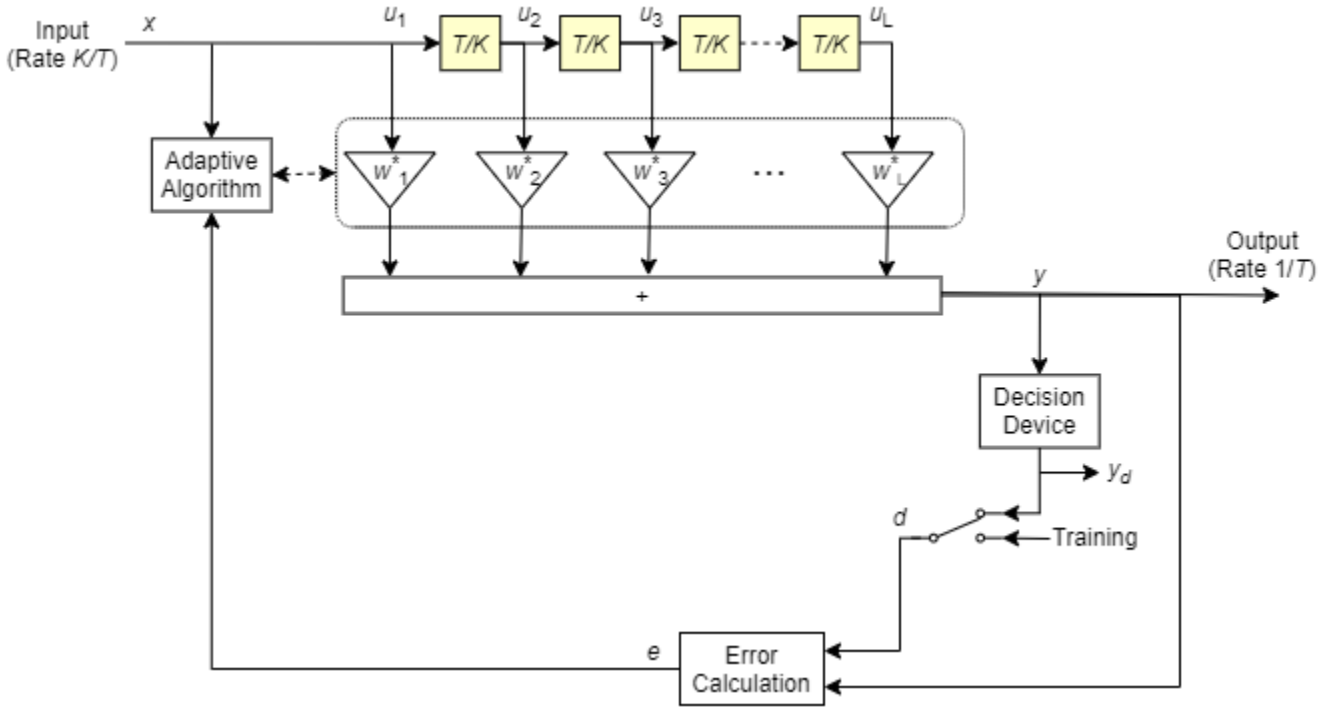
Linear equalizers can remove intersymbol interference (ISI) when the frequency response of a channel has no null. If a null exists in the frequency response of a channel, linear equalizers tend to enhance the noise. In this case, use decision feedback equalizers to avoid enhancing the noise.

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.

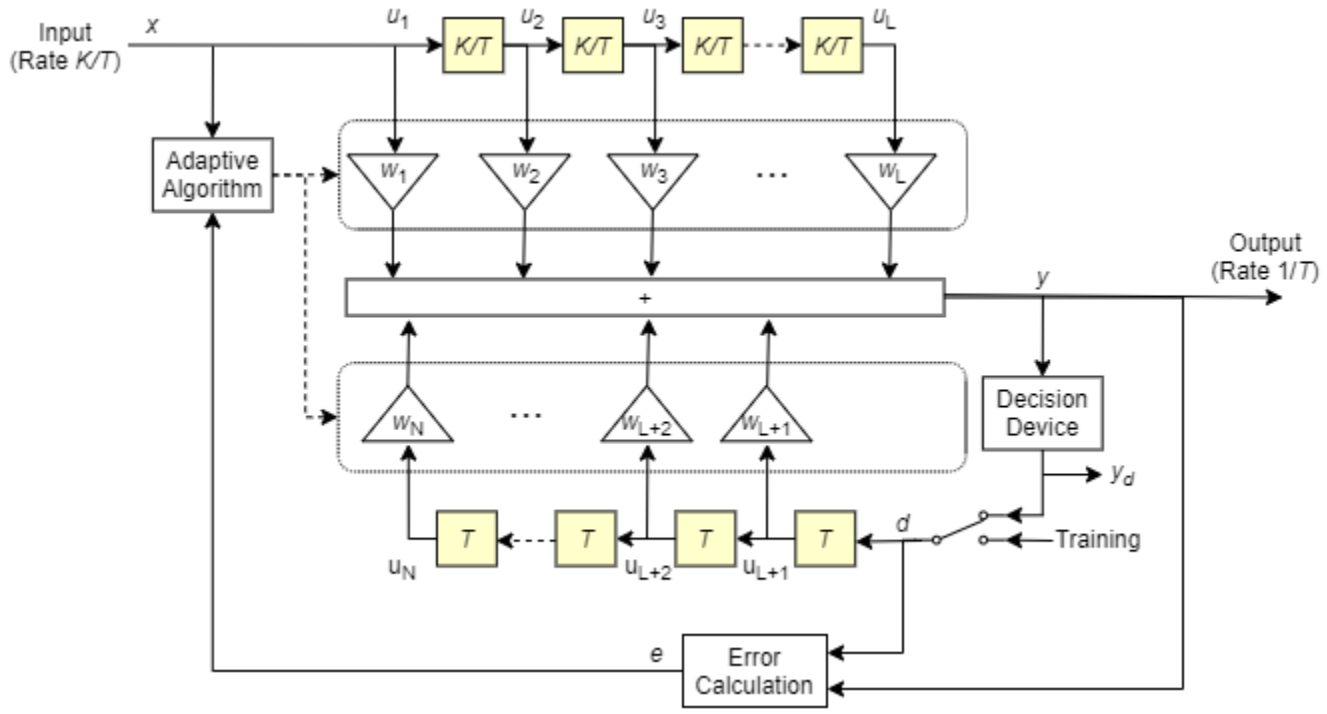
Decision-Feedback Equalizers

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.

Reference Signal and Operating Modes

In default applications, the equalizer first operates in training mode to gather information about the channel. The equalizer later switches to decision-directed mode.

- When the equalizer is operating in training mode, the reference signal is a preset, known transmitted sequence.
- When the equalizer is operating in decision-directed mode, the reference signal is a detected version of the output signal, denoted by y_d in the schematic.

The CMA algorithm has no training mode. Training mode applies only when the equalizer is configured to use the LMS or RLS algorithm.

Error Calculation

The error calculation operation produces a signal given by this expression, where R is a constant related to the signal constellation.

$$e = \begin{cases} d - y & \text{LMS or RLS} \\ y(R - |y|^2) & \text{CMA} \end{cases}$$

Updating Tap Weights

- “Least Mean Square Algorithm” on page 14-9
- “Recursive Least Square Algorithm” on page 14-9
- “Constant Modulus Algorithm” on page 14-10

For linear and decision-feedback equalizer structures, the choice of LMS, RLS, or CMA determines the algorithms that are used to set the tap weights and perform the error calculation. The new set of tap weights depends on:

- The current set of tap weights
- The input signal
- The output signal
- The reference signal, d, for LMS and RLS adaptive algorithms only. The reference signal characteristics depend on the operating mode of the equalizer.

Least Mean Square 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 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

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

$$w_{new} = w_{current} + (StepSize)\ u^*e.$$

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.

Configuring Adaptive Equalizers

Choose the linear or decision-feedback equalizer structure. Decide which adaptive algorithm to use — LMS, RLS, or CMA. Specify settings for structure and algorithm-specific operation modes.

Configuring an equalizer involves selecting a linear or decision-feedback structure, selecting an adaptive algorithm, and specifying the structure and algorithm specific operation modes.

- “Specify an Adaptive Equalizer” on page 14-10
- “Equalizer Training” on page 14-13
- “Managing Delays When Using Equalizers” on page 14-24

When deciding which adaptive algorithm best fits your needs, consider:

- The LMS algorithm executes quickly but converges slowly. Its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly. Its complexity grows approximately with the square of the number of weights. This algorithm can also be unstable when the number of weights is large.
- The constant modulus algorithm (CMA) is useful when no training signal is available. It works best for constant modulus modulations such as PSK.
    - If CMA has no additional side information, it can introduce phase ambiguity. For example, the weights found by the CMA might produce a perfect QPSK constellation but introduce a phase rotation of 90, 180, or 270 degrees. In this case, employ a phase ambiguity correction algorithm or choose a differential modulation scheme. Differential modulation schemes are insensitive to phase ambiguity.

To view or change any properties of an adaptive equalizer, use the syntax described for channel objects in “Displaying and Changing Object Properties” on page 22-14.

For more information about adaptive algorithms, see the references listed in “Selected References for Equalizers” on page 14-3.

Specify an Adaptive Equalizer

- “Defining an Equalizer Object” on page 14-11
- “Adaptive Algorithm Assignment” on page 14-12

To create an adaptive equalizer object for use in MATLAB, select the `comm.LinearEqualizer` or `comm.DecisionFeedbackEqualizer` System object. For Simulink, use the Linear Equalizer or

Decision Feedback Equalizer block. Based on the propagation channel characteristics in your simulation, use the criteria in “Equalization” on page 14-2 to select the equalizer structure.

The equalizer object has many properties that record information about the equalizer. Properties can be related to:

- The structure of the equalizer, such as the number of taps.
- The adaptive algorithm that the equalizer uses, such as the step size in the LMS or CMA algorithm.
- Information about the current state of the equalizer. The equalizer object can output the values of the weights.

To view or change any properties of an equalizer object, use the syntax described for channel objects in “Displaying and Changing Object Properties” on page 22-14.

Defining an Equalizer Object

The code creates equalizer objects for these configurations:

- A symbol-spaced linear RLS equalizer with 10 weights.
- A fractionally spaced linear RLS equalizer with 10 weights, a BPSK constellation, and two samples per symbol.
- A decision-feedback RLS equalizer with three weights in the feedforward filter and two weights in the feedback filter.

All three equalizer objects specify the RLS adaptive algorithm with a forgetting factor of 0.3.

Create equalizer objects of different types. The default settings are used for properties not set using `'Name,Value'` pairs.

```
eqlin = comm.LinearEqualizer('Algorithm','RLS','NumTaps',10,'ForgettingFactor',0.3)
```

```
eqlin =
  comm.LinearEqualizer with properties:

                         Algorithm: 'RLS'
                           NumTaps: 10
                 ForgettingFactor: 0.3000
    InitialInverseCorrelationMatrix: 0.1000
                     Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i ... ]
                      ReferenceTap: 3
                        InputDelay: 0
              InputSamplesPerSymbol: 1
              TrainingFlagInputPort: false
                 AdaptAfterTraining: true
               InitialWeightsSource: 'Auto'
                 WeightUpdatePeriod: 1
```

```
eqfrac = comm.LinearEqualizer('Algorithm','RLS','NumTaps',10,'ForgettingFactor',0.3, ...
    'Constellation',[-1 1],'InputSamplesPerSymbol',2)
```

```
eqfrac =
  comm.LinearEqualizer with properties:

                         Algorithm: 'RLS'
```

```
                          NumTaps: 10
                 ForgettingFactor: 0.3000
    InitialInverseCorrelationMatrix: 0.1000
                    Constellation: [-1 1]
                     ReferenceTap: 3
                       InputDelay: 0
              InputSamplesPerSymbol: 2
              TrainingFlagInputPort: false
                 AdaptAfterTraining: true
               InitialWeightsSource: 'Auto'
                 WeightUpdatePeriod: 1
```

```
eqdfe = comm.DecisionFeedbackEqualizer('Algorithm','RLS','NumForwardTaps',3, ...
    'NumFeedbackTaps',2,'ForgettingFactor',0.3)
```

```
eqdfe =
  comm.DecisionFeedbackEqualizer with properties:

                        Algorithm: 'RLS'
                  NumForwardTaps: 3
                 NumFeedbackTaps: 2
                 ForgettingFactor: 0.3000
    InitialInverseCorrelationMatrix: 0.1000
                    Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i ... ]
                     ReferenceTap: 3
                       InputDelay: 0
              InputSamplesPerSymbol: 1
              TrainingFlagInputPort: false
                 AdaptAfterTraining: true
               InitialWeightsSource: 'Auto'
                 WeightUpdatePeriod: 1
```

Adaptive Algorithm Assignment

Use the `Algorithm` property to assign the adaptive algorithm used by the equalizer.

Algorithm Assignment

When creating the equalizer object, assign the adaptive algorithm.

```
eqlms = comm.LinearEqualizer('Algorithm','LMS');
```

Create the equalizer object with default property settings. LMS is the default adaptive algorithm.

```
eqrls = comm.LinearEqualizer;
eqrls.Algorithm
```

```
ans =
'LMS'
```

Update `eqrls` to use the RLS adaptive algorithm.

```
eqrls.Algorithm = 'RLS';
eqrls.Algorithm
```

```
ans =
'RLS'
```

Cloning and Duplicating Objects

Configure a new equalizer object by cloning an existing equalizer object, and then changing its properties. Clone `eqlms` to create an independent equalizer, `eqcma`, then update the algorithm to `'CMA'`.

```
eqcma = clone(eqlms);
eqcma.Algorithm
```

```
ans =
'LMS'
```

```
eqcma.Algorithm = 'CMA';
eqcma.Algorithm
```

```
ans =
'CMA'
```

If you want an independent duplicate, use the `clone` command.

```
eqlms.NumTaps
```

```
ans = 5
```

```
eq2 = eqlms;
eq2.NumTaps = 6;
eq2.NumTaps
```

```
ans = 6
```

```
eqlms.NumTaps
```

```
ans = 6
```

The `clone` command creates a copy of `eqlms` that is independent of `eqlms`. By contrast, the command `eqB = eqA` creates `eqB` as a reference to `eqA`, so that `eqB` and `eqA` always have identical property settings.

Equalizer Training

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.

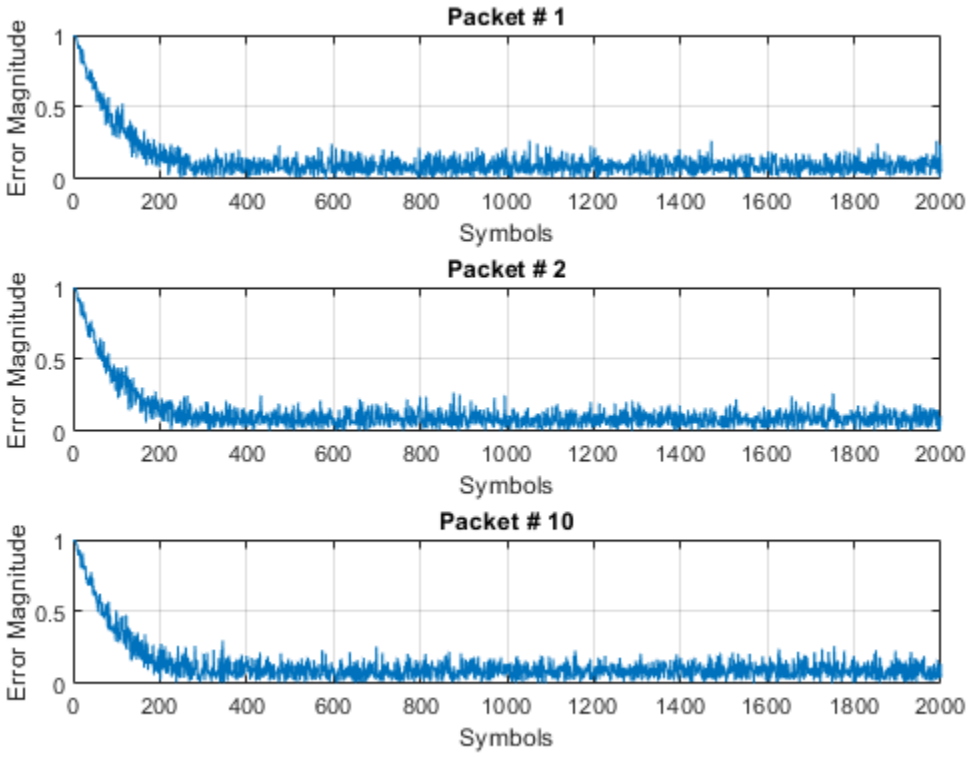
```
M = 4;
numTrainSymbols = 200;
numDataSymbols = 1800;
SNR = 20;
trainingSymbols = pskmod(randi([0 M-1],numTrainSymbols,1),M,pi/4);
```

```
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
```

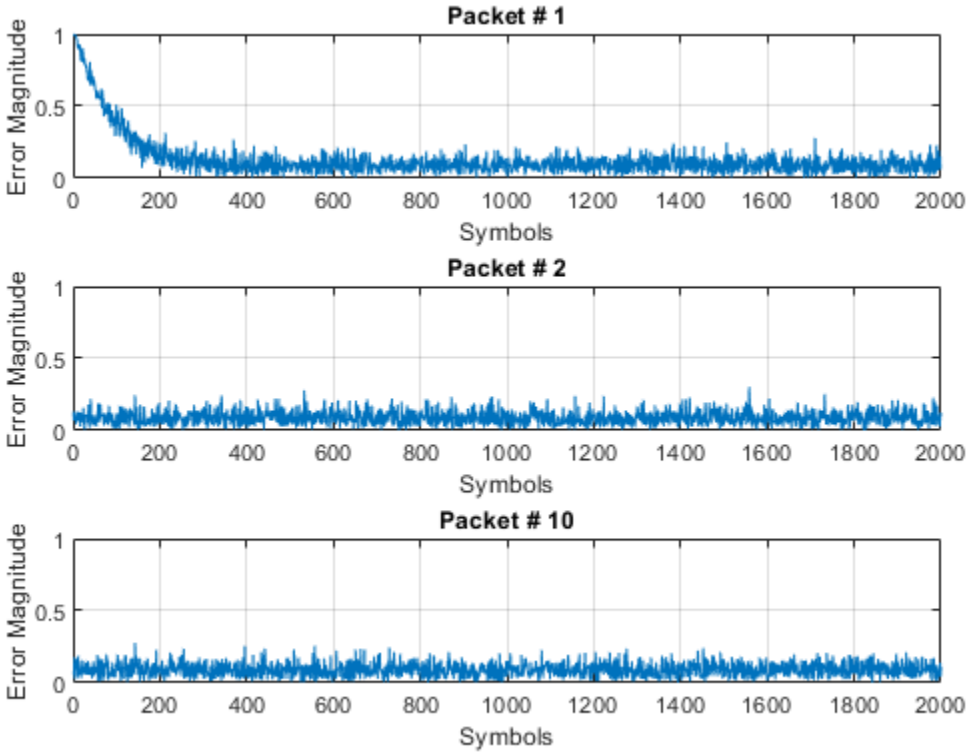


Train the Equalizer at the Beginning of Each Packet Without Reset

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);
```

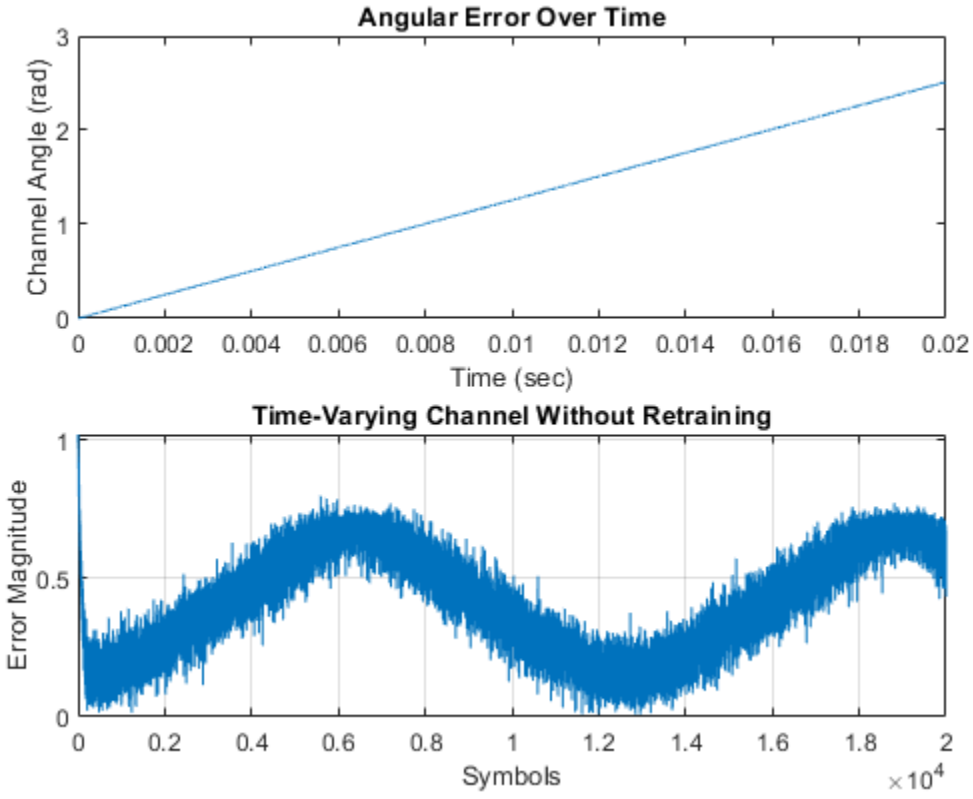
Set the `AdaptAfterTraining` property to `false` to stop the equalizer tap weight updates after the training phase.

```
release(lineq)
lineq.AdaptAfterTraining = false
```

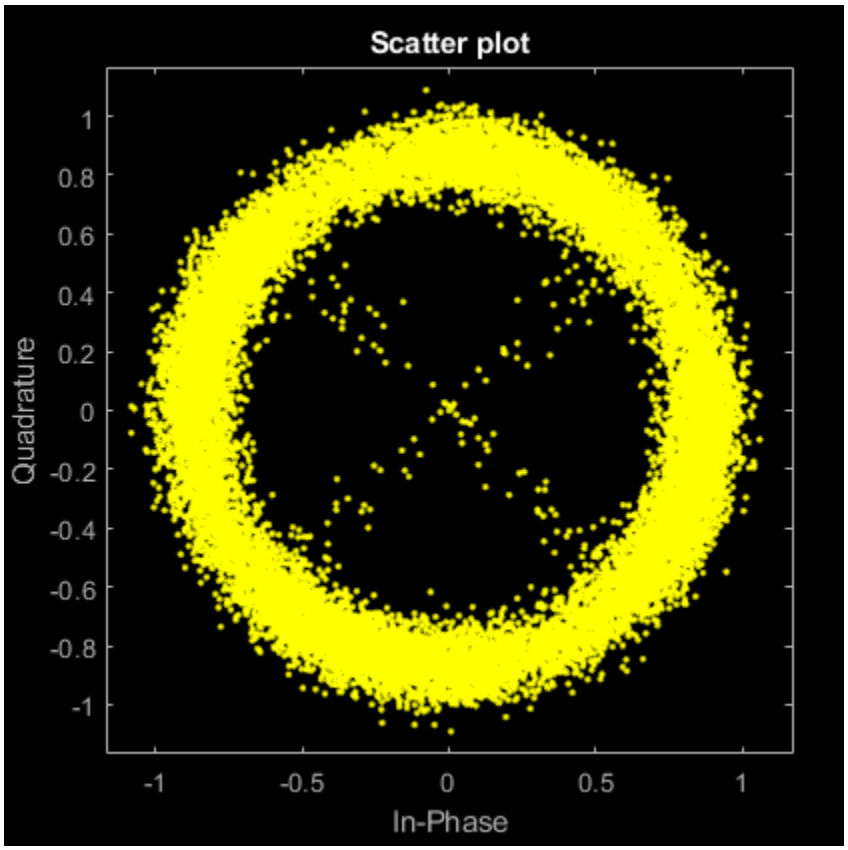
```
lineq =
  comm.LinearEqualizer with properties:

               Algorithm: 'LMS'
                 NumTaps: 5
                StepSize: 0.0100
           Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i ... ]
            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')
```



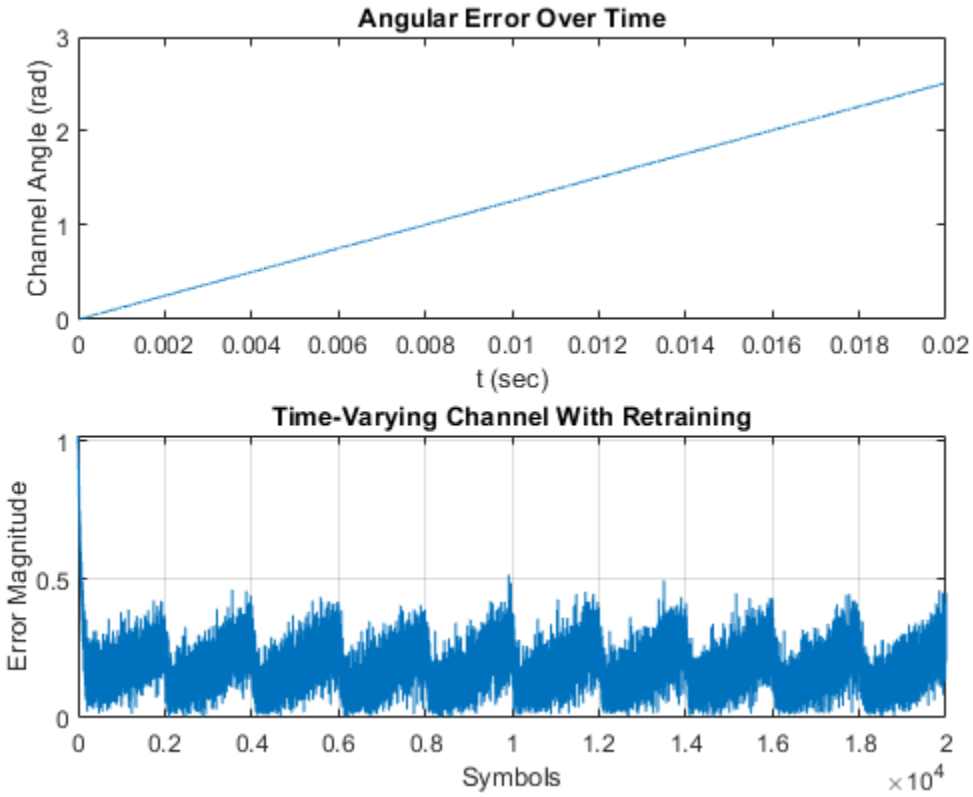

```
scatterplot(y)
```



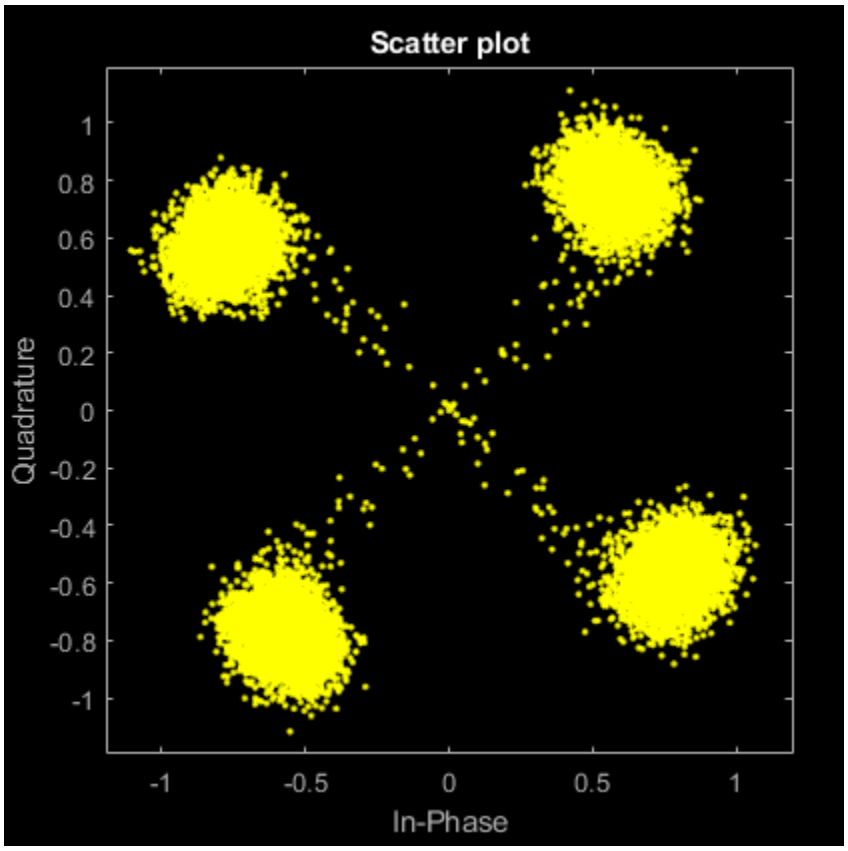
Set the `TrainingInputPort` property to `true` to configure the equalizer to retrain the taps when signaled by the `trainFlag` input. The equalizer trains only when `trainFlag` is `true`. After every 2000 symbols, the equalizer retrains the taps and keeps lock on variations of the channel. Plot the angular 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 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, ...
```

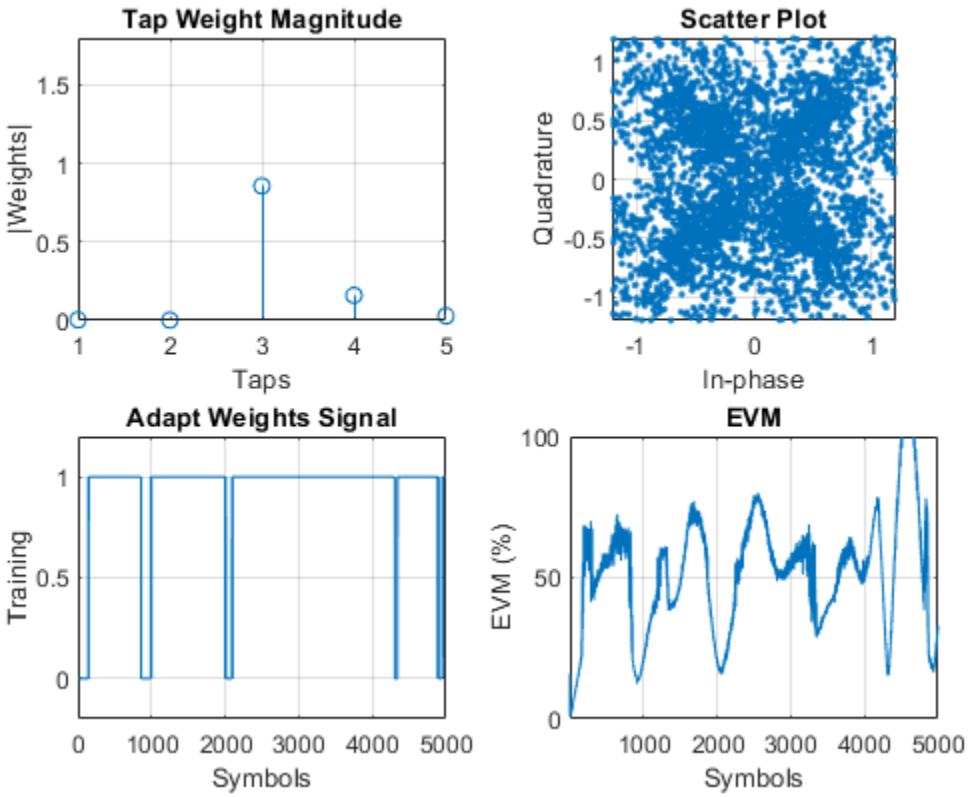
```
    'StepSize',0.03, ...
    'AdaptWeightsSource','Input port')

lineq =
  comm.LinearEqualizer with properties:

               Algorithm: 'CMA'
                 NumTaps: 5
                StepSize: 0.0300
           Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i ... ]
            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
```

Managing Delays When Using Equalizers

For proper equalization, you must determine and account for system delays. As shown in the following example, you can use the `finddelay` function to determine the system delay. This example

uses LMS linear equalization but the same approach is valid for the RLS and CMA adaptive algorithms and for decision feedback equalizers.

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.

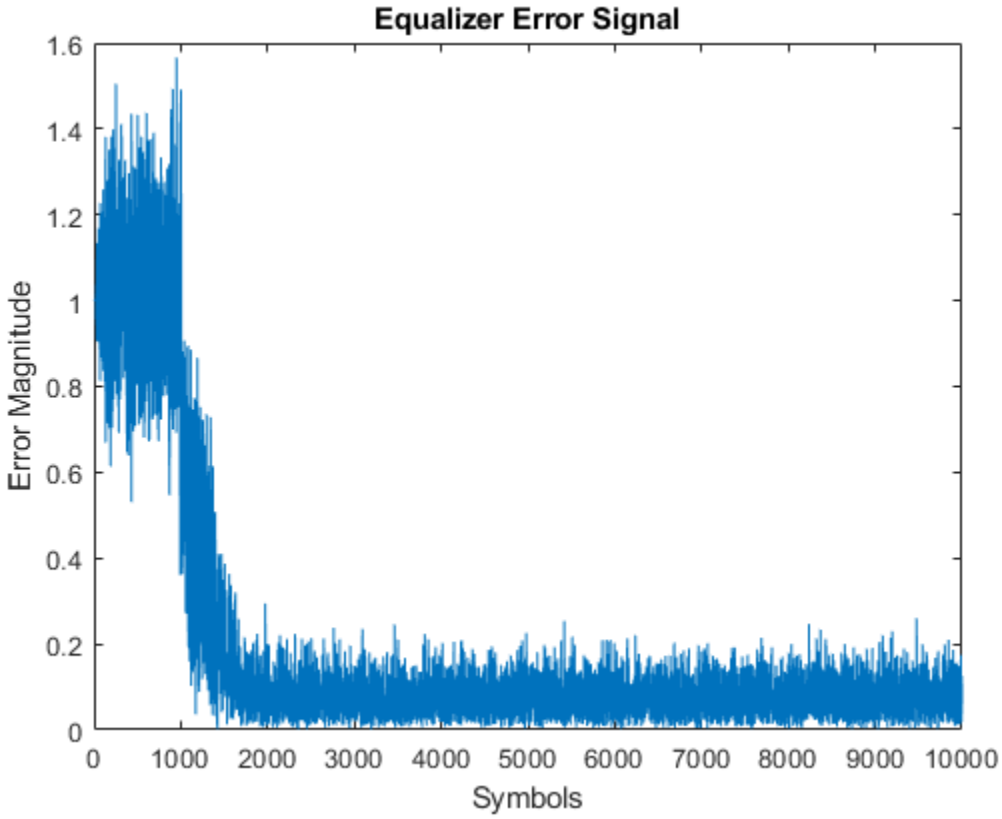
```
eqInfo = info(lineq)
```

```
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.

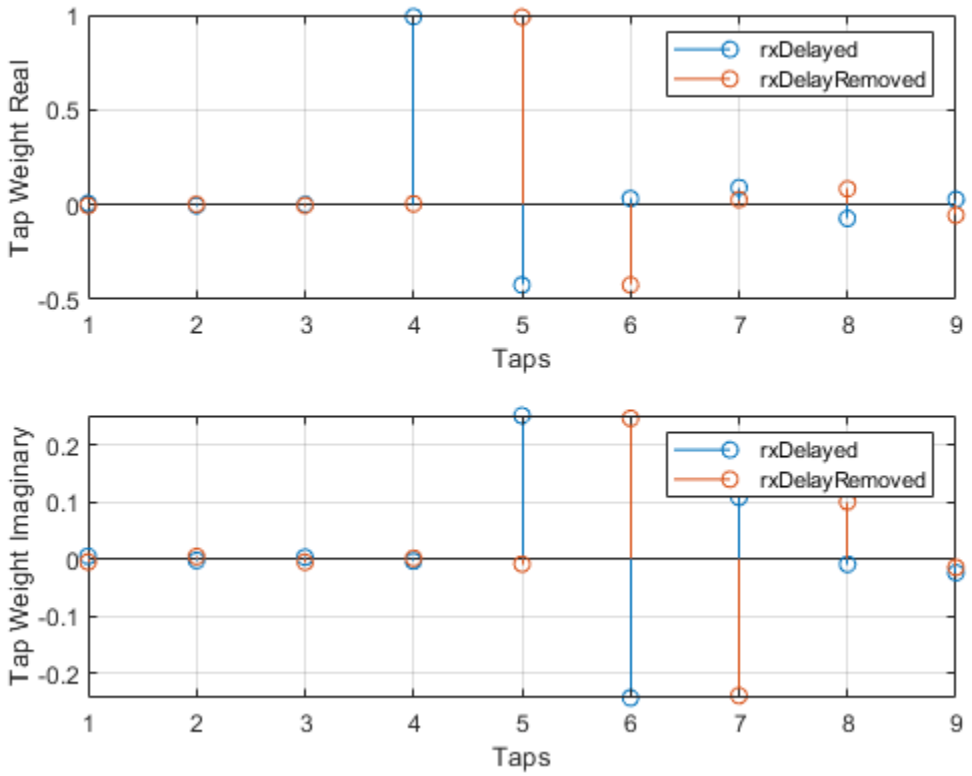
```
release(lineq)
lineq.InputDelay = rxDelay
```

```
lineq =
  comm.LinearEqualizer with properties:

               Algorithm: 'LMS'
                 NumTaps: 9
                StepSize: 0.0100
           Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i ... ]
            ReferenceTap: 5
              InputDelay: 20
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: true
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

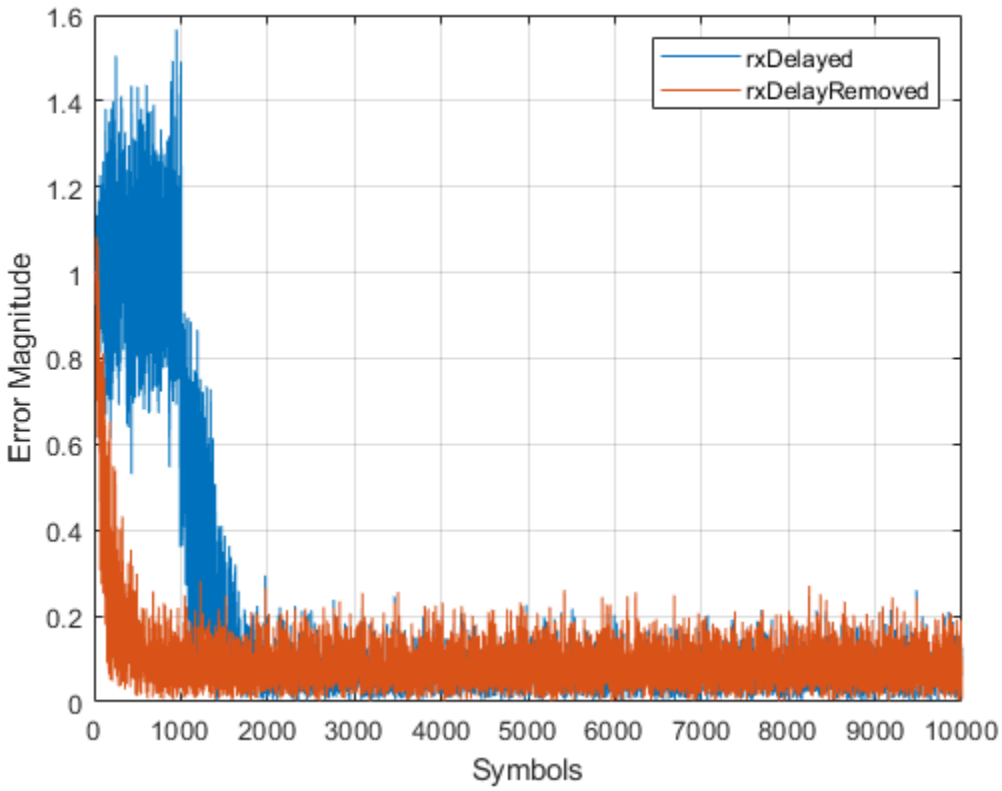
```
[y2,err2,wts2] = lineq(rx,tx(1:numTrainingSymbols,1));
```

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
```

Using Adaptive Equalizers in Simulink

Adaptive Equalization with Filtering and Fading Channel

This model shows the behavior of the selected adaptive equalizer in a communication link that has a fading channel. The transmitter and receiver have root raised cosine pulse shaped filtering. A subsystem block enables you to select between linear or decision feedback equalizers that usie the least mean square (LMS) or recursive least square (RLS) adaptive algorithm.

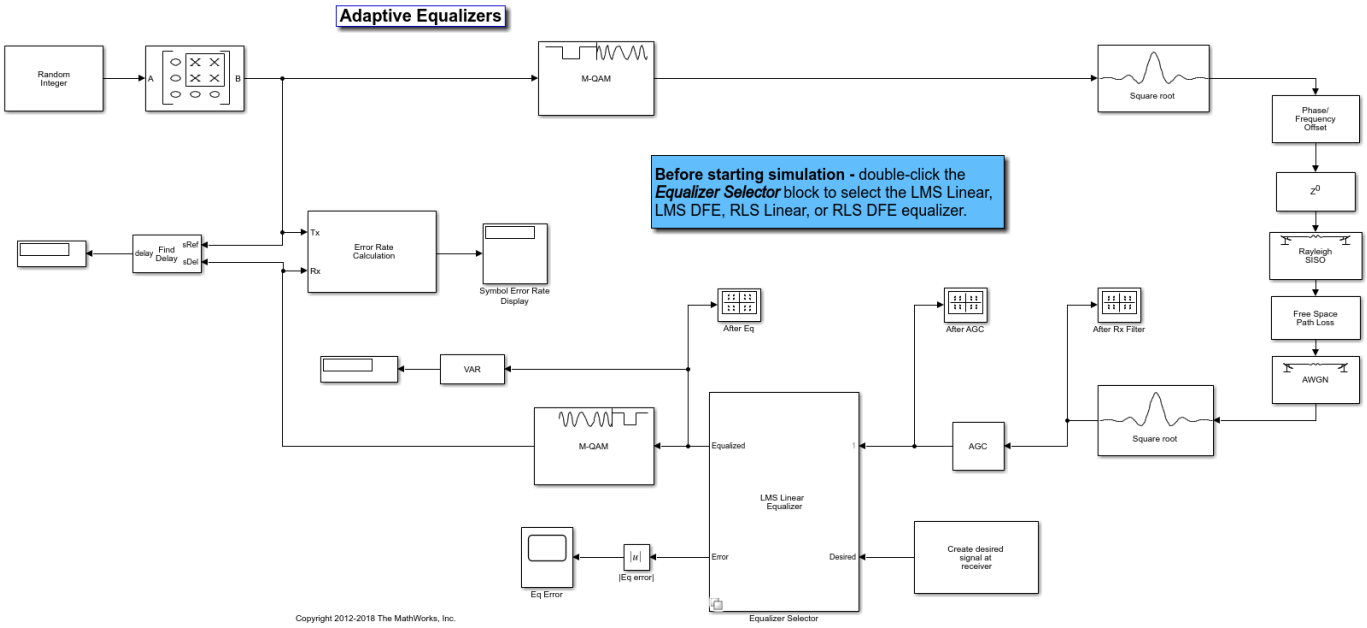
Model Structure

- The transmitter generates 16QAM random signal data that includes a training sequence and applies root raised cosine pulse shaped filtering.
- Channel impairments include multipath fading, Doppler shift, carrier frequency offset, variable integer delay, free space path loss, and AWGN.
- The receiver applies root raised cosine pulse shaped filtering, adjusts the gain, includes equalizer mode control to enable training and enables you to select the equalizer algorithm from these choices.

Selection	Equalizer Algorithm
LMS Linear	Linear least mean square equalizer
LMS DFE	Decision feedback least mean square equalizer
RLS Linear	Linear recursive least square equalizer
RLS DFE	Decision feedback recursive least square equalizer

- Scopes help you understand how the different equalizers and adaptive algorithms behave.

Explore Example Model



Experimenting with the model

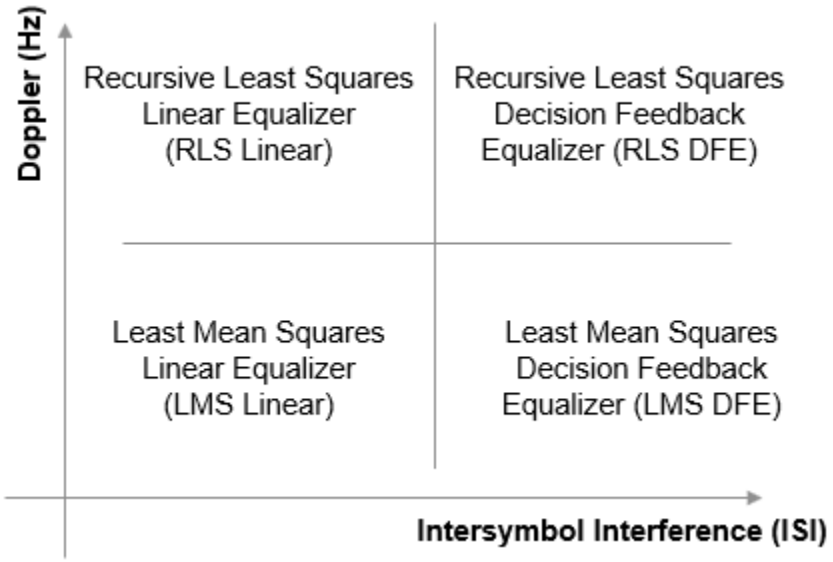
This model provides several ways for you to change settings and observe the results. The `InitFcn` found in `File>Model Properties>Callbacks` calls `cm_ex_adaptive_eq_with_fading_init` to initialize the model. This file enables you to vary settings in the model, including:

- System parameters, such as SNR.
- Pulse shaping filter parameters, such as rolloff and filter length
- Path loss value.
- Channel conditions: Rayleigh or Rician fading, channel path gains, channel path delays, and Doppler shift.
- Equalizer choice and configuration.

Model Considerations

This non-standards-based communication link is representative of a modern communications system.

- The optimal equalizer configuration depends on the channel conditions. The initialization file sets the Doppler shift and multipath fading channel parameters that highlight the capabilities of different equalizers.



- The decision feedback equalizer structure performs better than the linear equalizer structure for higher intersymbol interference.
- The RLS algorithm performs better than the LMS algorithm for higher Doppler frequencies.
- The LMS algorithm executes quickly, converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, its complexity grows approximately as the square of the number of weights. It can be unstable when the number of weights is large.
- The channels exercised for different equalizers have the following characteristics.

Selection	Channel Characteristics
LMS Linear	3 tap multipath fading channel with 10 Hz Doppler shift
LMS DFE	5 tap multipath fading channel with 25 Hz Doppler shift
RLS Linear	2 tap multipath fading channel with 70 Hz Doppler shift
RLS DFE	5 tap multipath fading channel with 100 Hz Doppler shift

- Initial settings for other channel impairments are the same for all equalizers. Carrier frequency offset value is set to 50 Hz. Free space path loss is set to 60 dB. Variable integer delay is set to 2 samples, which requires the equalizers to perform some timing recovery.

Deep channel fades and path loss can cause the equalizer input signal level to be much less than the desired output signal level and result in unacceptably long equalizer convergence time. The AGC block adjusts the magnitude of received signal to reduce the equalizer convergence time. You must adjust the optimal gain output power level based on the modulation scheme selected. For 16QAM, a desired output power of 10 W is used.

Training of the equalizer is performed at the beginning of the simulation.

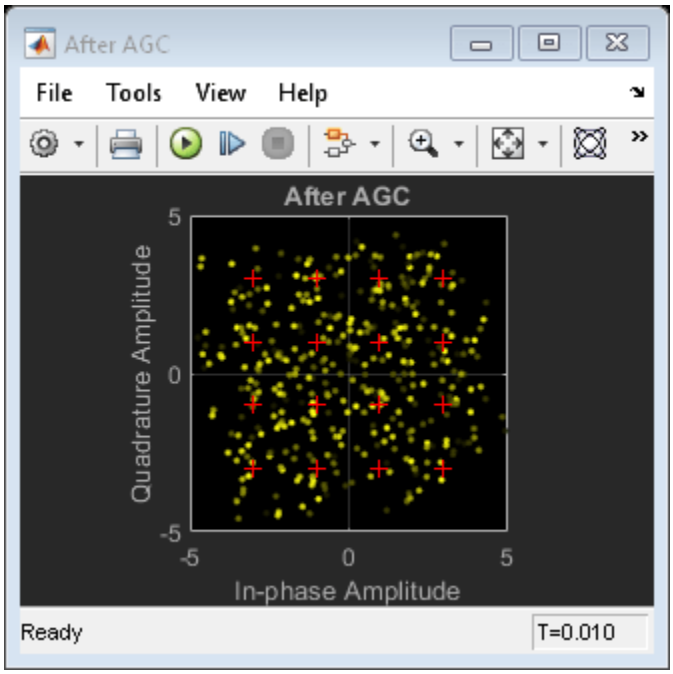
Running the Simulation

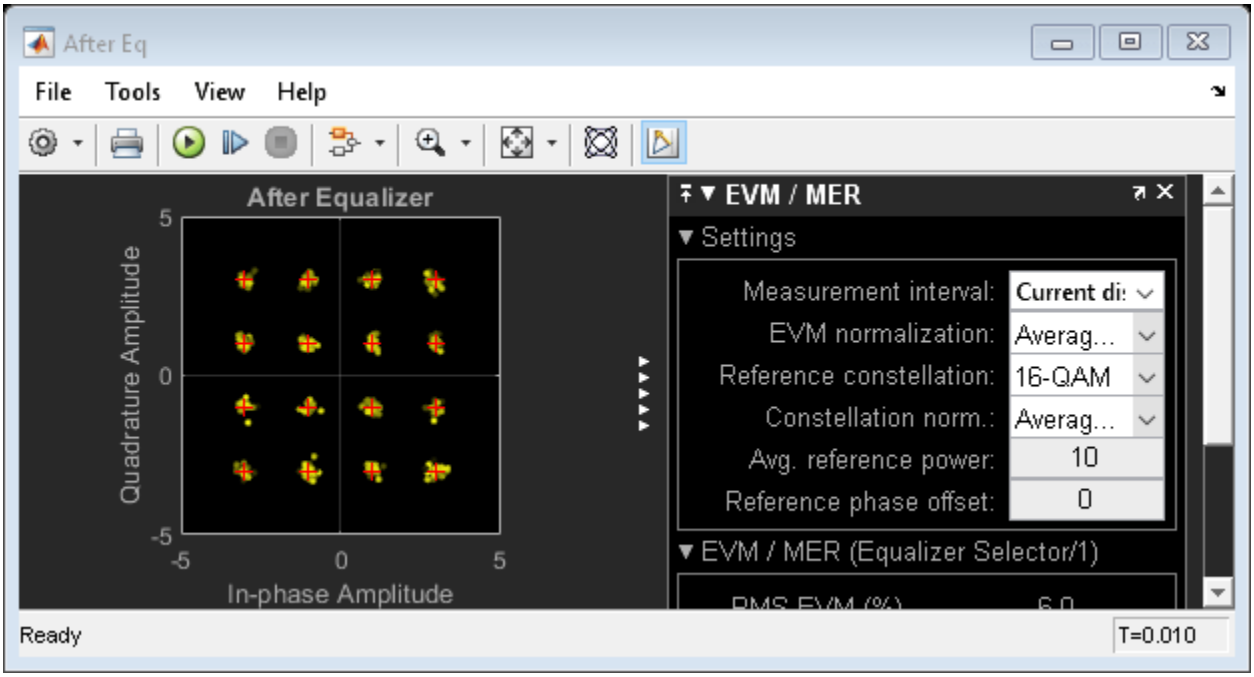
Running the simulation computes symbol error statistics and produces these figures:

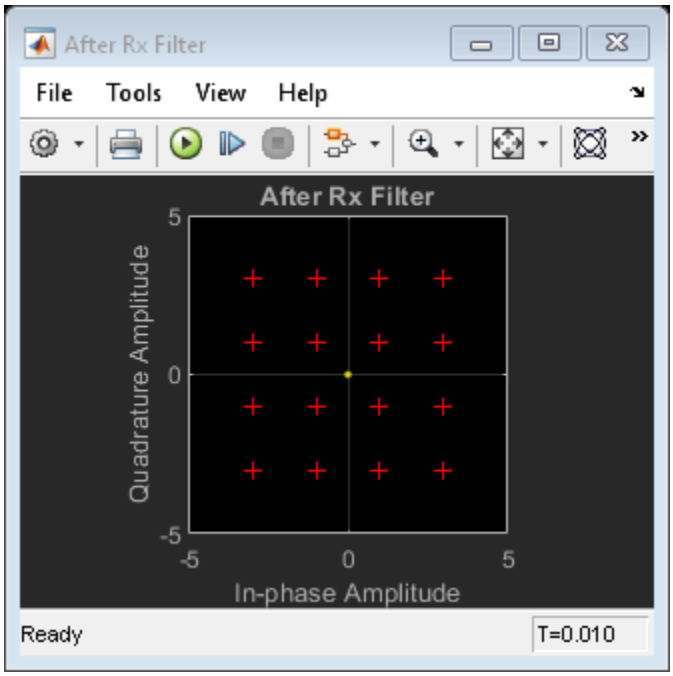
- A constellation diagram of the signal after the receive filter.
- A constellation diagram of the signal after adjusting gain.
- A constellation diagram of the signal after equalization with signal quality measurements shown.
- An equalizer error plot.

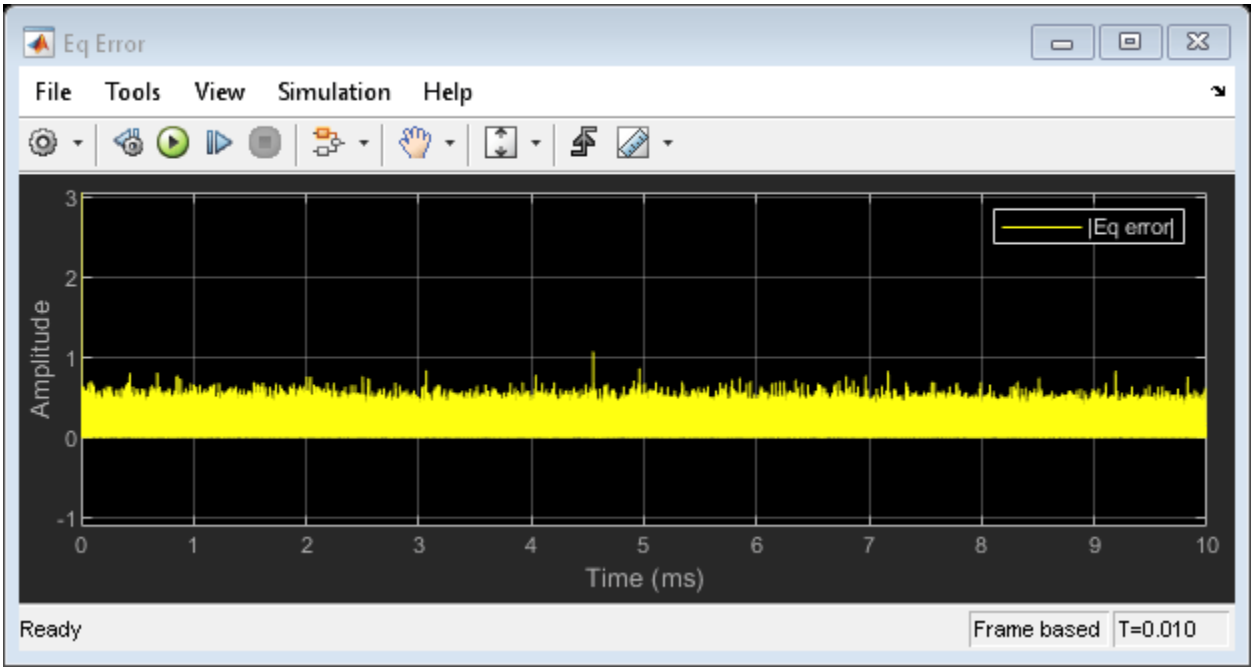
For the plots shown here, the equalizer algorithm selected is `RLS Linear`. Monitoring these figures, you can see that the received signal quality fluctuates as simulation time progresses.

The *After Rx Filter* and *After AGC* constellation plots show the signal before equalization. *After AGC* shows the impact of the channel conditions on the transmitted signal. The *After Eq* plot shows the signal after equalization. The signal plotted in the constellation diagram after equalization shows the variation in signal quality based on the effectiveness of the equalization process. Throughout the simulation, the signal constellations plotted before equalization deviate noticeably from a 16QAM signal constellation. The *After Eq* constellation improves or degrades as the equalizer error signal varies. The `Eq error` plotted in the *Eq Error* plot, indicates poor equalization at the start of the simulation. The error degrades at first then improves as the equalizer converges.




After Eq
File Tools View Help
After Equalizer
Quadrature Amplitude
In-phase Amplitude
5
0
-5
-5
0
5
EVM / MER
Settings
Measurement interval:
Current di:
EVM normalization:
Averag...
Reference constellation:
16-QAM
Constellation norm.:
Averag...
Avg. reference power:
10
Reference phase offset:
0
EVM / MER (Equalizer Selector/1)
Ready
T=0.010


After Rx Filter
File Tools View Help
After Rx Filter
Quadrature Amplitude
In-phase Amplitude
5
0
-5
-5
0
5
Ready
T=0.010



Further Exploration

Double-click the `Equalizer Selector` block and select a different equalizer. Run the simulation to see the performance of the various equalizer options. You can use the signal logger to compare the results from this experimentation. In the block diagram, right-click on signal wires and select `Log Selected Signals`. If you have enabled signal logging, after the simulation run finishes, open the Simulation Data Inspector to view the logged signals.

At the MATLAB™ command prompt, enter `edit cm_ex_adaptive_eq_with_fading_init.m` to open the initialization file, then modify a parameter and rerun the simulation. For example, adjust the channel characteristics (params.maxDoppler|, `params.pathDelays`, and `params.pathGains`). The RLS adaptive algorithm performs better than the LMS adaptive algorithm as the maximum Doppler is increased.

See Also

Objects
`comm.LinearEqualizer` | `comm.DecisionFeedbackEqualizer` | `comm.MLSEEqualizer`

Blocks
Linear Equalizer | Decision Feedback Equalizer | MLSE Equalizer

More About

- “Equalization” on page 14-2
- “MLSE Equalizers” on page 14-36

MLSE Equalizers

In this section...

Maximum-Likelihood Sequence Estimation (MLSE) equalizers provide optimal equalization of time variations in the propagation channel characteristics. However, MLSE equalizers are sometimes less appealing because their computational complexity is higher than “Adaptive Equalizers” on page 14-5.

In Communications Toolbox, the `mlseeq` function, `comm.MLSEEqualizer` System object, and MLSE Equalizer block use the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. These features output the maximum likelihood sequence estimate of the signal by using an estimate of the channel modeled as a finite input response (FIR) filter.

To decode a received signal, the MLSE equalizer:

1. Applies the FIR filter to the symbols in the input signal. The FIR filter tap weights correspond to the channel estimate.
2. Uses the Viterbi algorithm to compute the traceback paths and the state metric. These values are assigned to the symbols at each step of the Viterbi algorithm. The metrics are based on Euclidean distance.
3. Outputs the maximum likelihood sequence estimate of the signal as a sequence of complex numbers corresponding to the constellation points of the modulated signal.

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

For background material on MLSE equalizers, see “Selected References for Equalizers” on page 14-3.

Equalize a Vector Signal in MATLAB

You can use the `mlseeq` function or `comm.MLSEEqualizer` System object for MLSE equalization in MATLAB. The examples in this section call the `mlseeq` function. A similar workflow applies when using the `comm.MLSEEqualizer` System object. For examples that use the System object, see the `comm.MLSEEqualizer` System object reference page.

The `mlseeq` function has two operation modes:

- Continuous operation mode enables you to process a series of vectors by using repeated calls to `mlseeq`. The function saves its internal state information from one call to the next. To learn more, see Equalizing in Continuous Operation Mode on page 14-37.
- Reset operation mode enables you to specify a preamble and postamble that accompany your data. To learn more, see Using a Preamble or Postamble on page 14-40.

If you are not processing a series of vectors and do not need to specify a preamble or postamble, the operational modes are nearly identical. They differ in that continuous operation mode incurs a delay,

while reset operation mode does not. The following example uses reset operation mode. If you modify the example to run using continuous operation mode, there will be delay in the equalized output. To learn more about this delay, see Delays in Continuous Operation Mode on page 14-38.

Use mlseeq to Equalize a Vector Signal

In its simplest form, the `mlseeq` function equalizes a vector of modulated data when you specify:

- The estimated coefficients of the channel (modeled as an FIR filter).
- The signal constellation for the modulation type.
- The Viterbi algorithm traceback depth. Larger values for the traceback depth can improve the results from the equalizer but increase the computation time.

Generate a PSK modulated signal with modulation order set to four.

```
M = 4;
msg = pskmod([1 2 2 0 3 1 3 3 2 1 0 2 3 0 1]',M);
```

Filter the modulated signal through a distortion channel.

```
chcoeffs = [.986; .845; .237; .12345+.31i];
filtmsg = filter(chcoeffs,1,msg);
```

Define the reference constellation, traceback length, and channel estimate for the MLSE equalizer. In this example, the exact channel is provided as the channel estimate.

```
const = pskmod([0:M-1],M);
tblen = 10;
chanest = chcoeffs;
```

Equalize the received signal.

```
msgEq = mlseeq(filtmsg,chanest,const,tblen,'rst');
isequal(msg,msgEq)
```

```
ans = logical
   1
```

Equalizing Signals in Continuous Operation Mode

If your data is partitioned into a series of vectors (that you process within a loop, for example), continuous operation mode is an appropriate way to use the `mlseeq` function. In continuous operation mode, `mlseeq` can save its internal state information for use in a subsequent invocation and can initialize by using previously stored state information. To choose continuous operation mode when invoking `mlseeq`, specify `'cont'` as an input argument.

Note Continuous operation mode incurs a delay, as described in Delays in Continuous Operation Mode on page 14-38. This mode cannot accommodate a preamble or postamble.

Procedure for Continuous Operation Mode

When using continuous operation mode within a loop, preallocate three empty matrix variables to store the state metrics, traceback states, and traceback inputs for the equalizer before the equalization loop starts. Inside the loop, invoke `mlseeq` using a syntax such as:

```
sm = [];
ts = [];
ti = [];
for ...
    [y,sm,ts,ti] = mlseeq(x,chcoeffs,const,tblen,'cont',nsamp,sm,ts,ti);
...
end
```

Using `sm`, `ts`, and `ti` as input arguments causes `mlseeq` to continue operating from where it finished in the previous iteration. Using `sm`, `ts`, and `ti` as output arguments causes `mlseeq` to update the state information at the end of the current iteration. In the first iteration, `sm`, `ts`, and `ti` start as empty matrices, so the first invocation of the `mlseeq` function initializes the metrics of all states to 0.

Delays in Continuous Operation Mode

Continuous operation mode with a traceback depth of `tblen` incurs an output delay of `tblen` symbols. The first `tblen` output symbols are unrelated to the input signal and the last `tblen` input symbols are unrelated to the output signal. For example, this command uses a traceback depth of `3`. The first three output symbols are unrelated to the input signal of `ones(1,10)`.

```
y = mlseeq(ones(1,10),1,[-7:2:7],3,'cont')
```

```
y =
    -7    -7    -7     1     1     1     1     1     1     1
```

It is important to keep track of delays introduced by different portions of the communications system. The “Use mlseeq to Equalize a Vector in Continuous Operation Mode” on page 14-38 example illustrates how to account for the delay when computing an error rate.

Use mlseeq to Equalize a Vector in Continuous Operation Mode

This example shows the procedure for using the continuous operation mode of the `mlseeq` function within a loop.

Initialize Variables

Specify run-time variables.

```
numsym = 200; % Number of symbols in each iteration
numiter = 25; % Number of iterations

M = 4; % Use 4-PSK modulation
qpskMod = comm.QPSKModulator('PhaseOffset',0);

chcoeffs = [1 ; 0.25]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
```

To initialize the equalizer, define parameters for the reference constellation, traceback length, number of samples per symbol, and the state variables `sm`, `ts`, and `ti`.

```
const = qpskMod((0:M-1)');
tblen = 10;
nsamp = 1;
sm = [];
```

```
ts = [];
ti = [];
```

Define variables to accumulate results from each iteration of the loop.

```
fullmodmsg = [];
fullfiltmsg = [];
fullrx = [];
```

Simulate the System by Using a Loop

Run the simulation in a loop that generates random data, modulates the data by using baseband PSK modulation, and filters the data. The `mlseeq` function equalizes the filtered data. The loop also updates the variables that accumulate results from each iteration of the loop.

```
for jj = 1:numiter
    msg = randi([0 M-1],numsym,1); % Random signal vector
    modmsg = qpskMod(msg); % PSK-modulated signal
    filtmsg = filter(chcoeffs,1,modmsg); % Filtered signal
    % Equalize the signal.
    [rx,sm,ts,ti] = mlseeq(filtmsg,chanest,const, ...
        tblen,'cont',nsamp,sm,ts,ti);
    % Update vectors with cumulative results.
    fullmodmsg = [fullmodmsg; modmsg];
    fullfiltmsg = [fullfiltmsg; filtmsg];
    fullrx = [fullrx; rx];
end
```

Computing an Error Rate and Plotting Results

Compute the symbol error rate from all iterations of the loop. The `symerr` function compares selected portions of the received and transmitted signals, not the entire signals. Because continuous operation mode incurs a delay whose length in samples is the traceback depth (`tblen`) of the equalizer, exclude the first `tblen` samples from the received signal and the last `tblen` samples from the transmitted signal. Excluding samples that represent the delay of the equalizer ensures that the symbol error rate calculation compares samples from the received and transmitted signals that are meaningful and that truly correspond to each other.

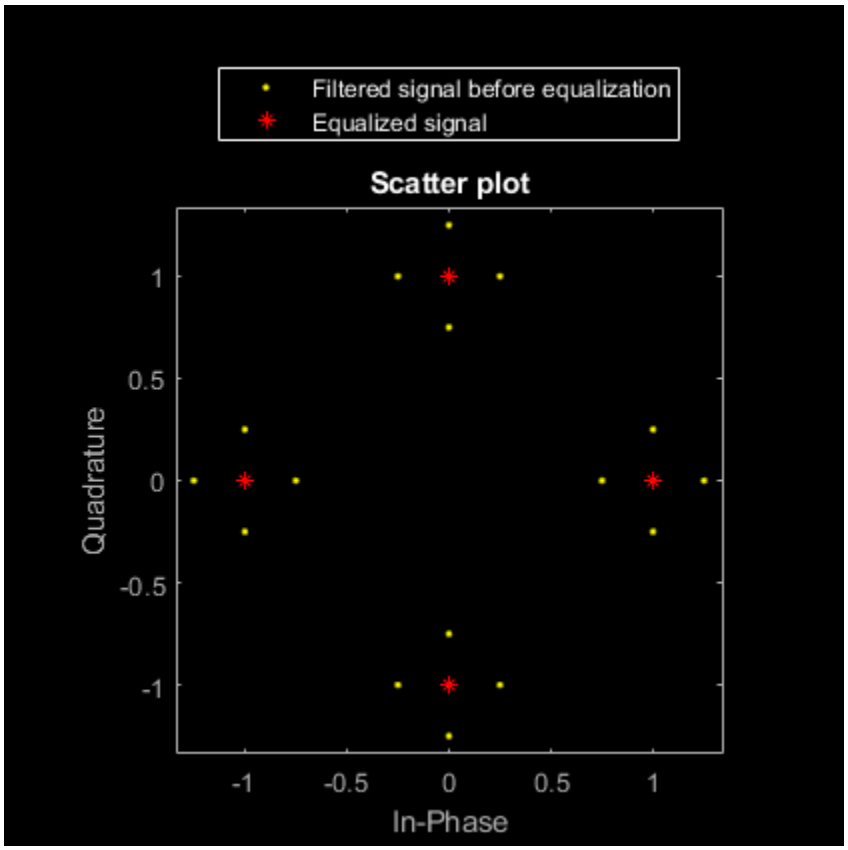
Taking the delay into account, compute the total number of symbol errors.

```
hErrorCalc = comm.ErrorRate('ReceiveDelay',10);
err = step(hErrorCalc, fullmodmsg, fullrx);
numsymerrs = err(1)
```

```
numsymerrs = 0
```

Plot the signal constellation before and after equalization. The points in the equalized signal coincide with the points of the ideal signal constellation for 4-PSK.

```
h = scatterplot(fullfiltmsg);
hold on;
scatterplot(fullrx,1,0,'r*',h);
legend('Filtered signal before equalization','Equalized signal',...
   'Location','NorthOutside');
hold off;
```



Use a Preamble or a Postamble

Some systems include a sequence of known symbols at the beginning or end of a set of data. The known sequence at the beginning or end is called a *preamble* or *postamble*, respectively. The `mlseeq` function can accommodate a preamble and postamble that are already incorporated into its input signal. When you invoke the function, you specify the preamble and postamble as integer vectors that represent the sequence of known symbols by indexing into the signal constellation vector. For example, a preamble vector of `[1 4 4]` and a 4-PSK signal constellation of `[1 j -1 -j]` indicates that the modulated signal begins with `[1 -j -j]`.

If your system uses a preamble without a postamble, use a postamble vector of `[]` when invoking `mlseeq`. If your system uses a postamble without a preamble, use a preamble vector of `[]`.

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 Equalizers in Simulink

The MLSE Equalizer block uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The block outputs the maximum likelihood sequence estimate (MLSE) of the signal by using your estimate of the channel modeled as a finite input response (FIR) filter. When using the MLSE Equalizer block, you specify the channel estimate and the signal constellation of the input signal. You can also specify an expected input signal preamble and postamble as input parameters to the MLSE Equalizer block.

MLSE Equalization with Dynamically Changing Channel

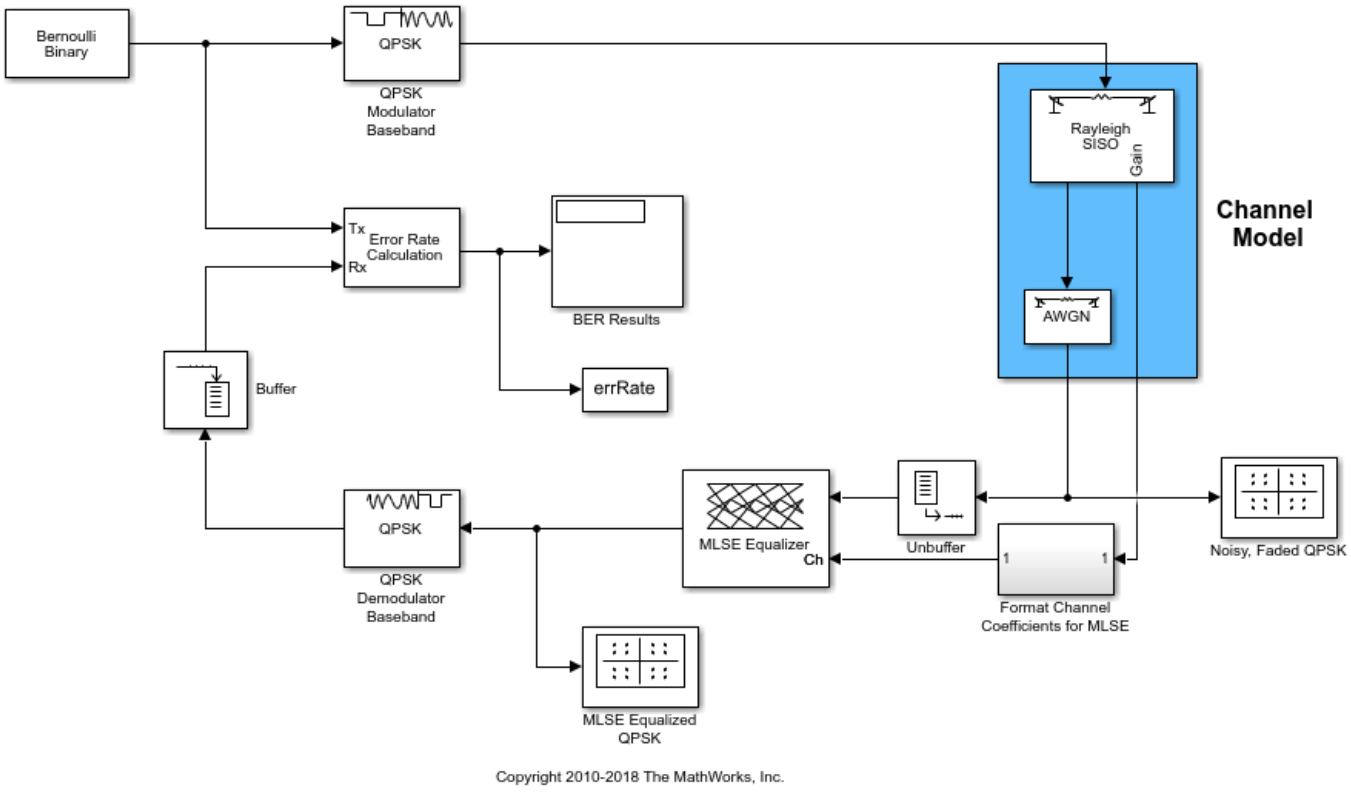
Use a Maximum Likelihood Sequence Estimation (MLSE) equalizer to equalize the effects of a multipath Rayleigh fading channel. The MLSE equalizer inputs data that has passed through a time varying dispersive channel and an estimate of the channel. The channel estimate contains dynamically evolving channel coefficients of a two-path Rayleigh fading channel.

Model Structure

- The transmitter generates QPSK random signal data.
- Channel impairments include multipath fading and AWGN.

- The receiver applies MLSE equalization and QPSK demodulation.
- The model uses scopes and a BER calculation to show the system behavior.

Explore Example Model



Experimenting with the model

The `Bernoulli Binary Generator` block sample time of 5e-6 seconds corresponds to a bit rate of 200 kbps and a QPSK symbol rate of 100 ksym/sec.

The `Multipath Rayleigh Fading Channel` block settings are:

- Maximum Doppler shift is 30 Hz.
- Discrete path delay is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This delay reflects the simplest delay vector for a two-path channel.
- Average path gain is [0 -10].
- Average path gains are normalized to 0 dB so that the average power input to the `AWGN` block is 1 W.

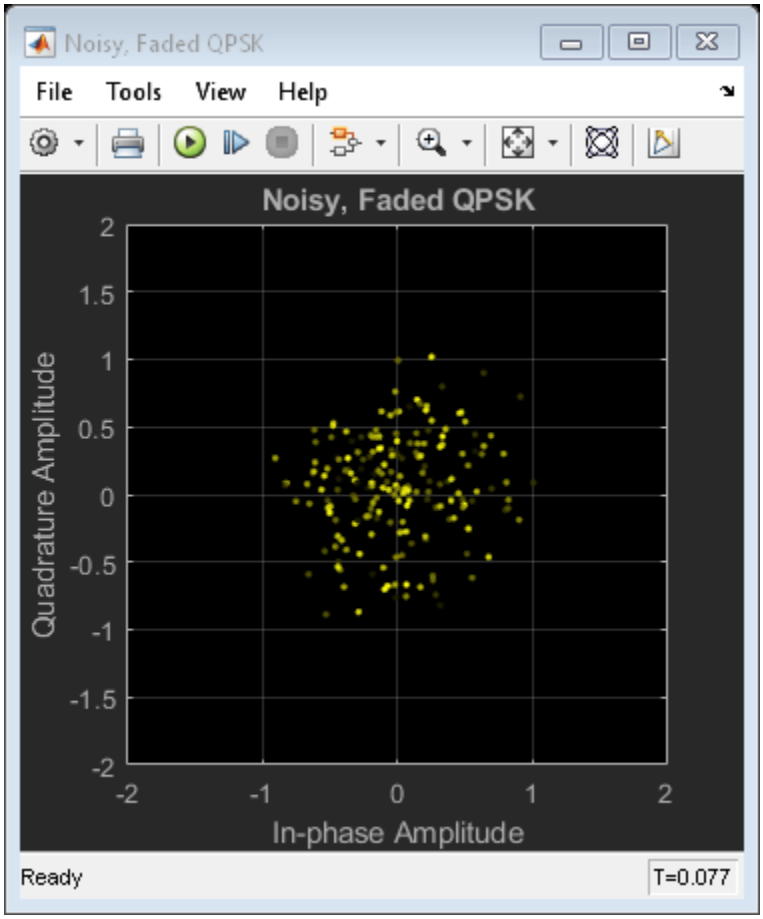
The `MLSE Equalizer` block has the Traceback depth set to 10. Vary this depth to study its effect on Bit Error rate (BER).

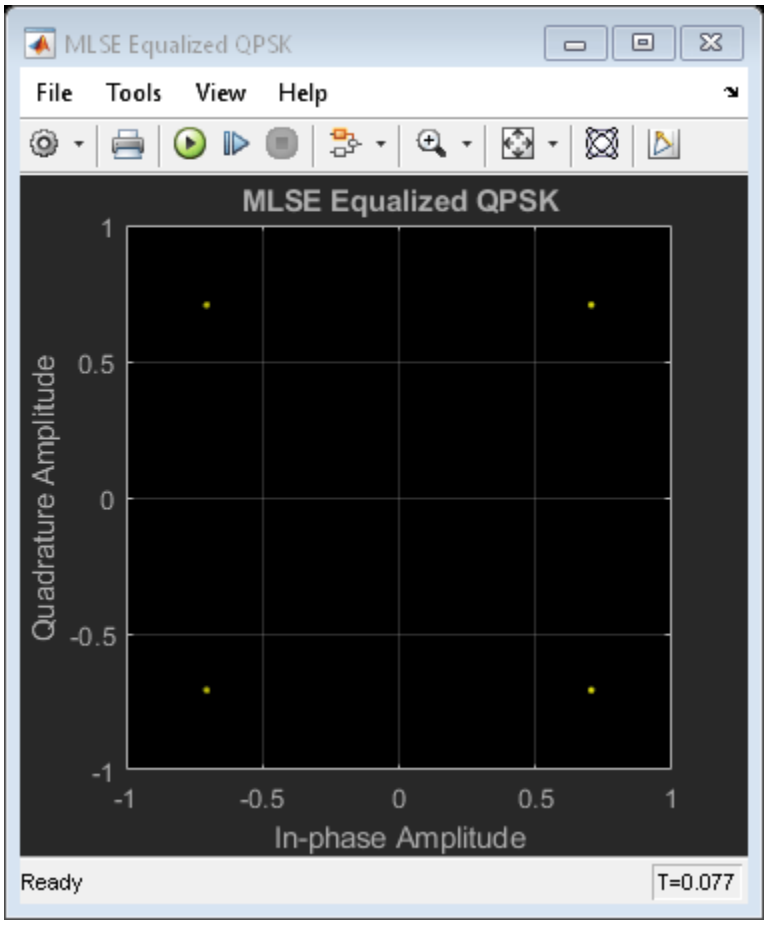
The QPSK demodulator accepts an N-by-1 input frame and generates a 2N-by-1 output frame. This output frame and a traceback depth of 10 results in a delay of 20 bits. The model performs frame-based processing on frames that have 100 samples per frame. Due to the frame-based processing,

there is a inherent delay of 100 bits in the model. The combined receive delay of 120 is set in the Receive delay parameter of the Error Rate Calculation block, aligning the samples.

The computed BER is displayed. Constellation plots show the constellation before and after equalization.

```
BER = 0.033645
```





See Also

Objects
`comm.LinearEqualizer` | `comm.DecisionFeedbackEqualizer` | `comm.MLSEEqualizer`

Blocks
Linear Equalizer | Decision Feedback Equalizer | MLSE Equalizer

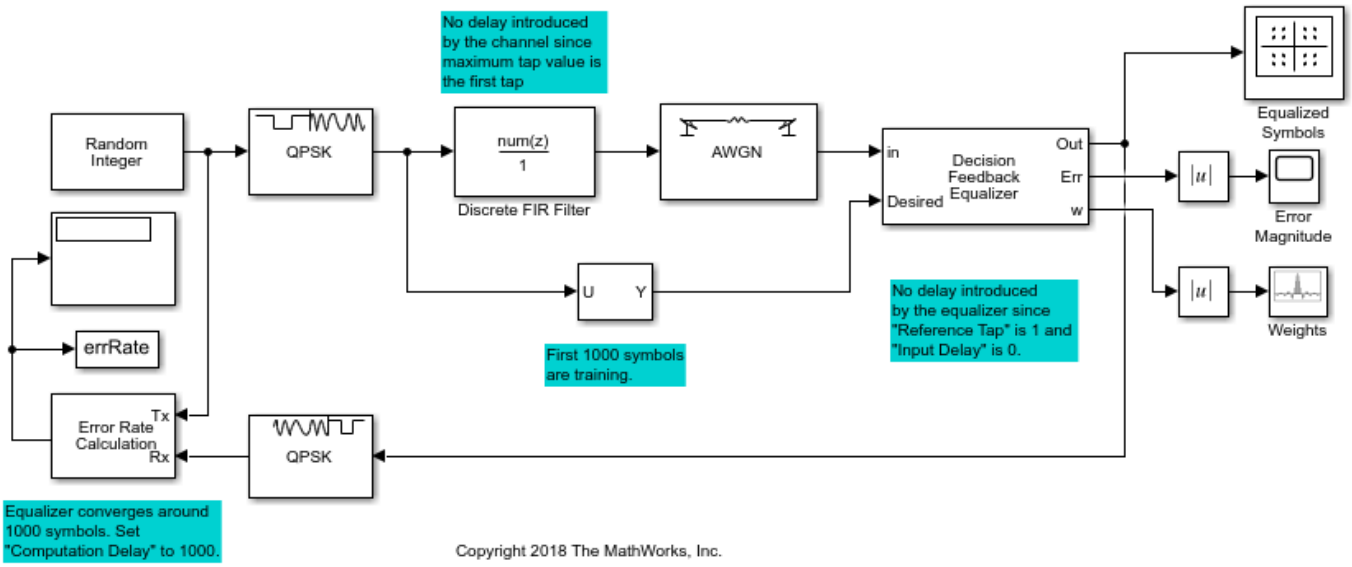
More About

- “Equalization” on page 14-2
- “Adaptive Equalizers” on page 14-5

Equalizer Examples (new & old)

DF Equalize QPSK-Modulated Signal in Simulink

Apply decision feedback equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel.

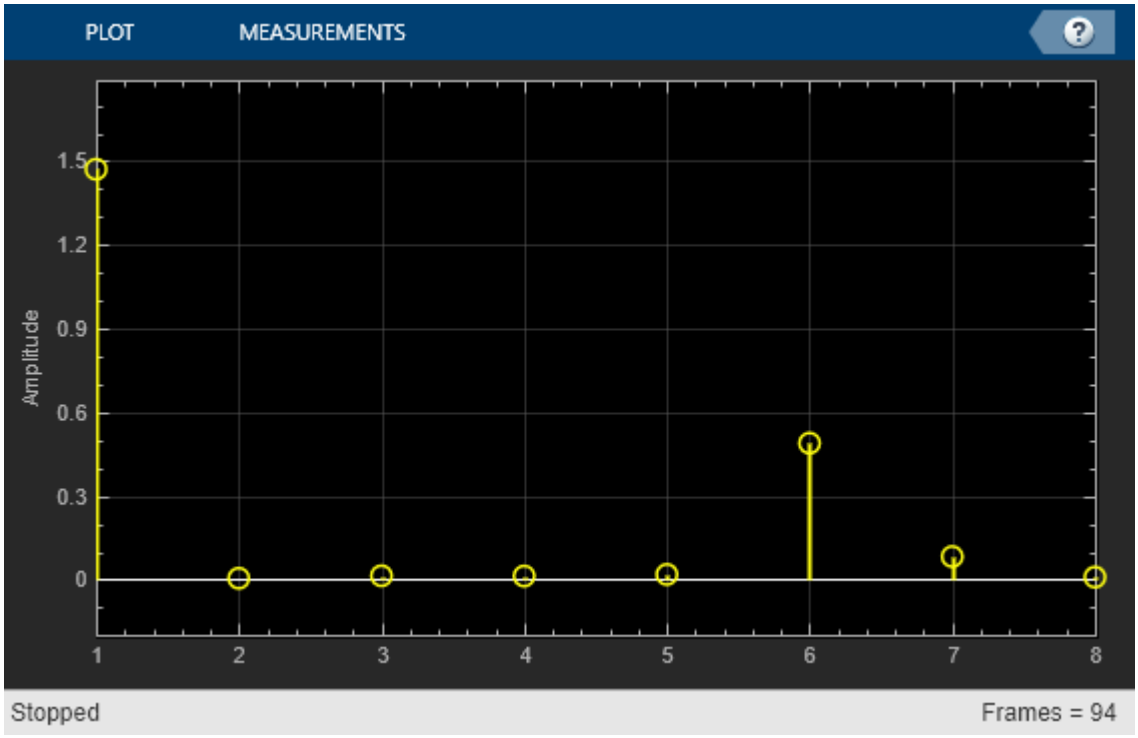


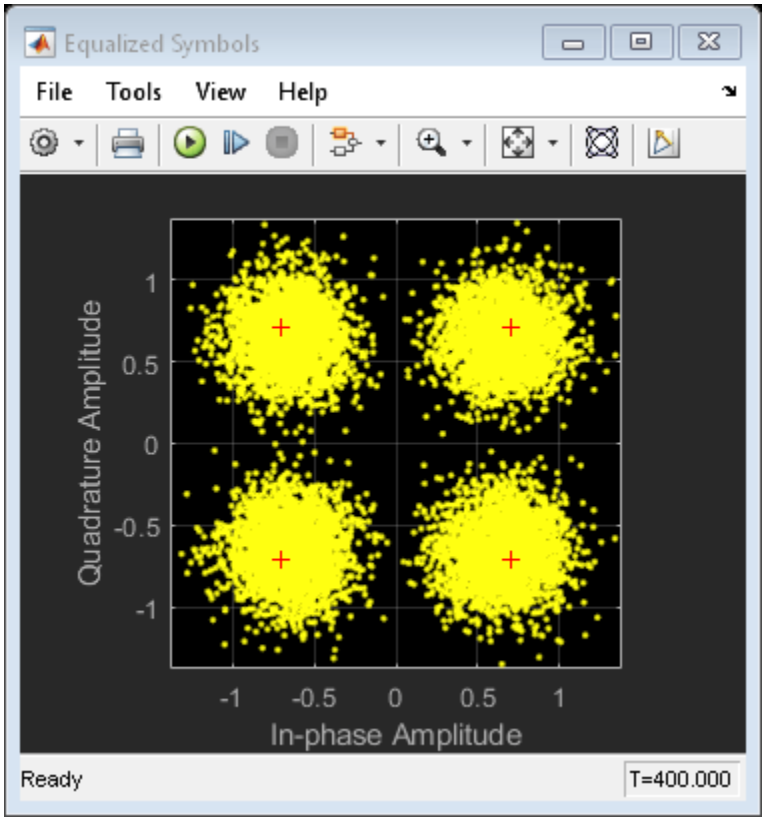
The `slex_dfeq_qpsk_signal` model generates an M=4 sequence using the `Random Integer Generator` block. The sequence is modulated with the `M-PSK Modulator Baseband` block, filtered with the `Discrete FIR Filter` block, and then impaired with the `AWGN` block. The `Decision Feedback Equalizer` block equalizes the data sequence, the data is demodulated with the `M-PSK Demodulator Baseband` block, and the bit error rate is computed. The signal path out of the modulator is split to a `Selector` block, which provides the first 1000 symbols of the modulated signal to the equalizer as an initial training sequence.

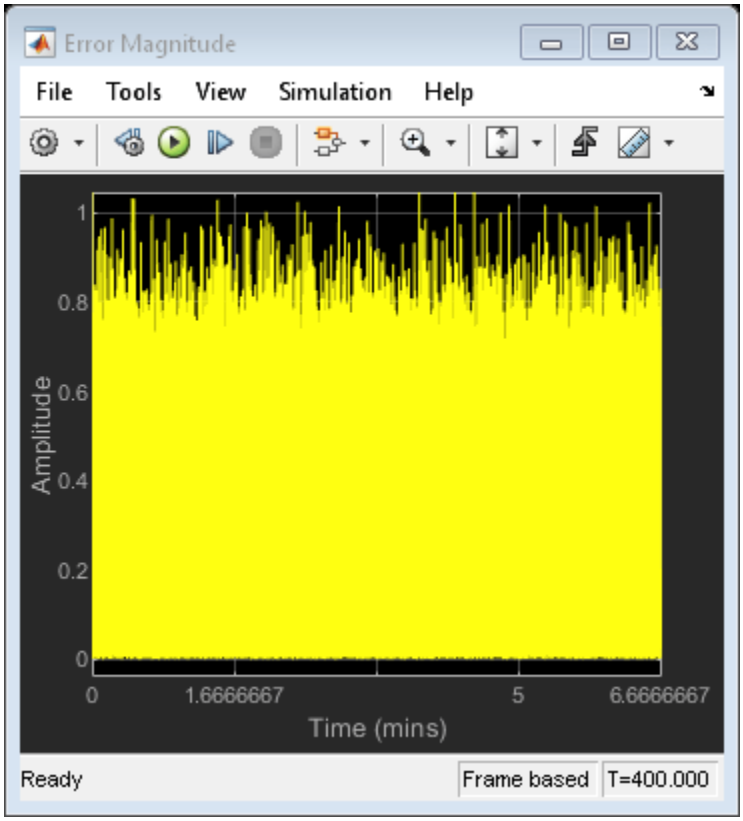
No delay is introduced between the transmitted and received signal because the maximum tap value is the first tap of the discrete FIR filter and the equalizer reference tap is `1`. The equalizer converges after around 1000 symbols so this value is used for the computation delay of the `Error Rate Calculation` block.

The computed error rate is displayed and plots show the equalized constellation, equalized tap weights, and signal error magnitude.

```
Computed error rate = 0.0011111
```

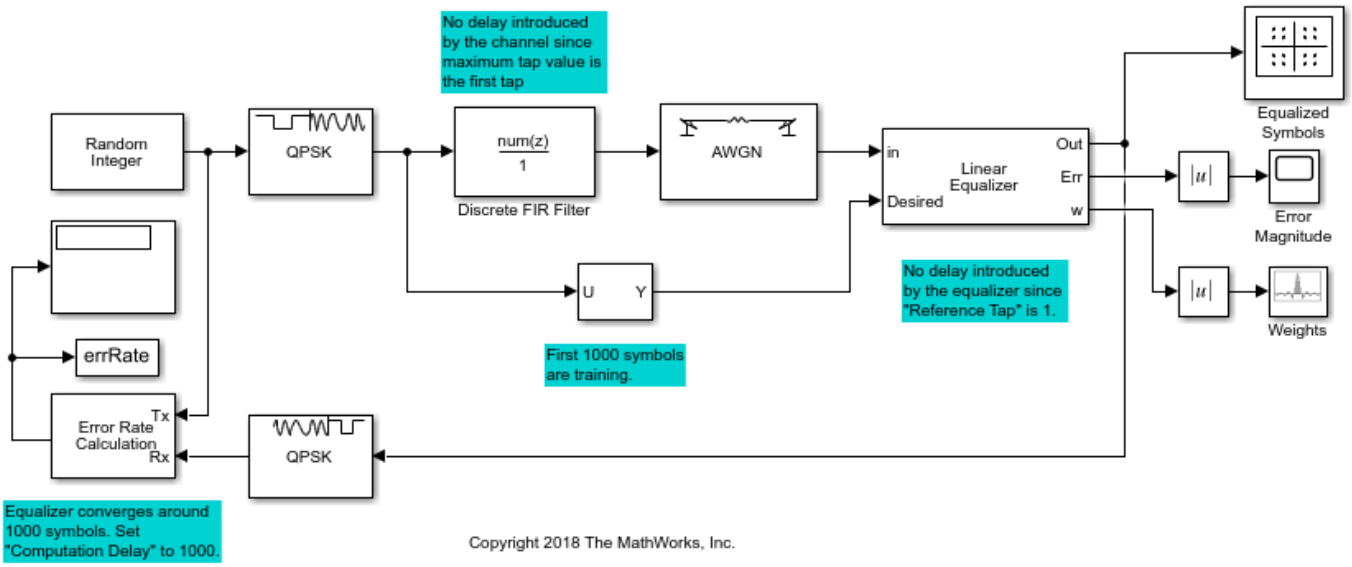

PLOT
MEASUREMENTS
Amplitude
1.5
1.2
0.9
0.6
0.3
0
1
2
3
4
5
6
7
8
Stopped
Frames = 94


Equalized Symbols
File
Tools
View
Help
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=400.000


Error Magnitude
File
Tools
View
Simulation
Help
1
0.8
0.6
0.4
0.2
0
Amplitude
0
1.6666667
5
6.6666667
Time (mins)
Ready
Frame based
T=400.000

Linearly Equalize QPSK-Modulated Signal in Simulink

Apply linear equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel.

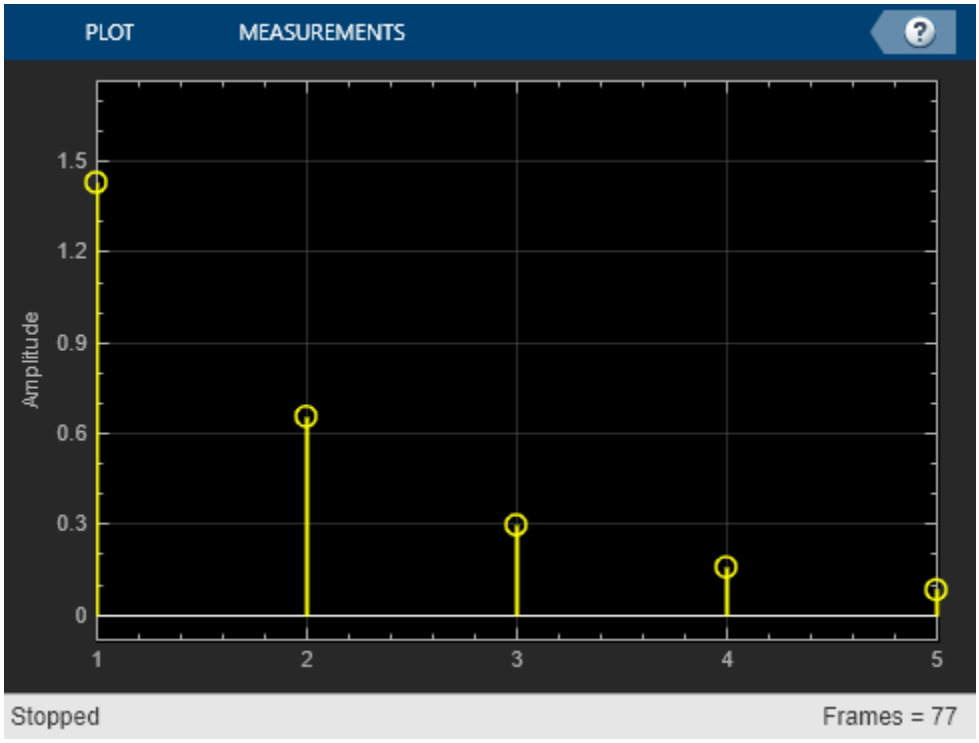


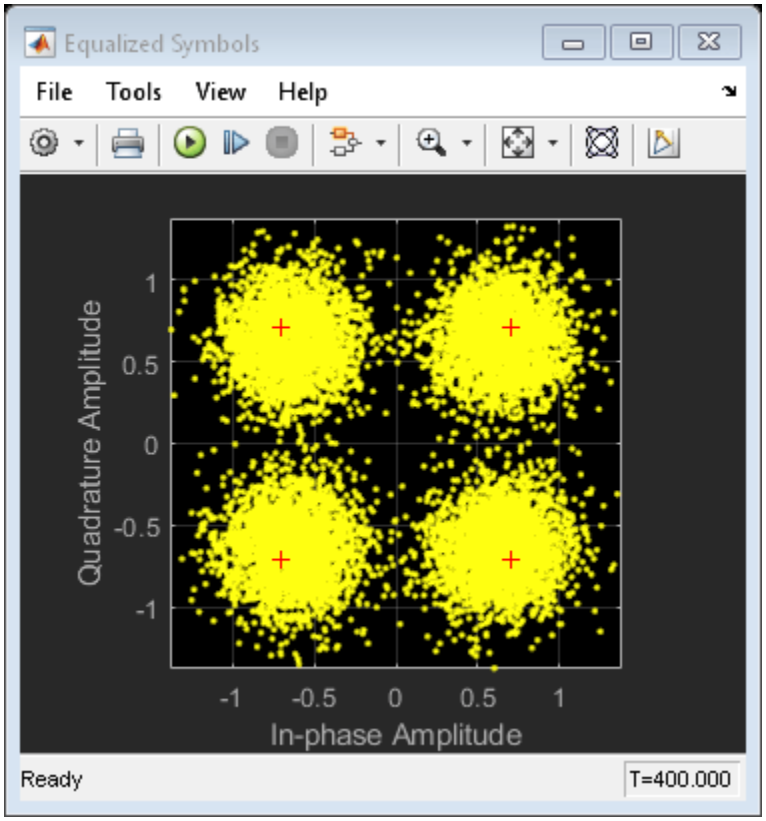
The `slex_lineq_qpsk_signal` model generates an M=4 sequence using the `Random Integer Generator` block. The sequence is modulated with the `M-PSK Modulator Baseband` block, filtered with the `Discrete FIR Filter` block, and then impaired with the `AWGN` block. The `Linear Equalizer` block equalizes the data sequence, the data is demodulated with the `M-PSK Demodulator Baseband` block, and the bit error rate is computed. The signal path out of the modulator is split to a `Selector` block, which provides the first 1000 symbols of the modulated signal to the equalizer as an initial training sequence.

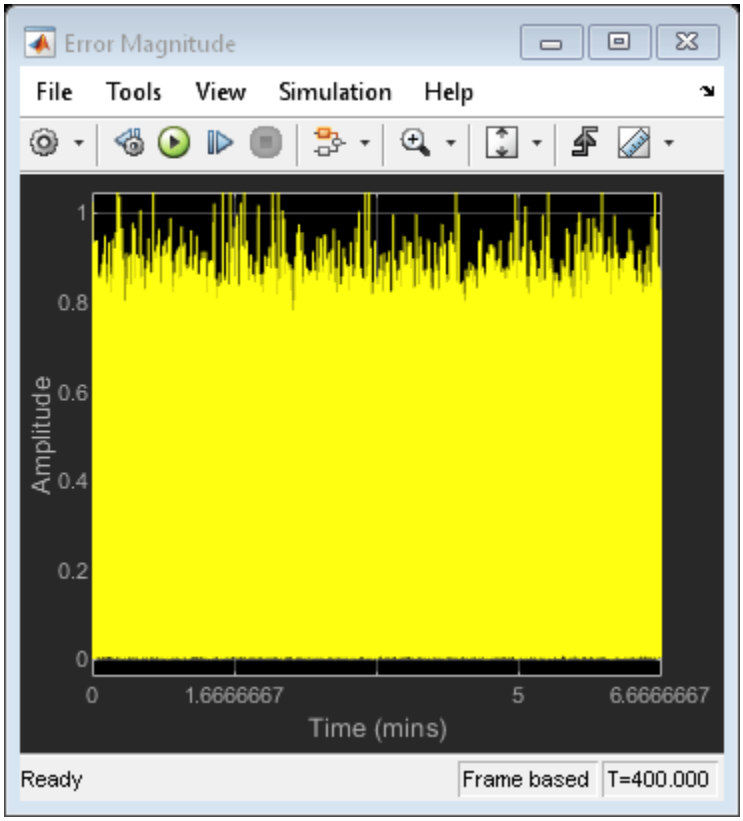
No delay is introduced between the transmitted and received signal because the maximum tap value is the first tap of the discrete FIR filter and the equalizer reference tap is `1`. The equalizer converges after around 1000 symbols so this value is used for the computation delay of the `Error Rate Calculation` block.

The computed error rate is displayed and plots show the equalized constellation, equalized tap weights, and signal error magnitude.

```
Computed error rate = 0.0024444
```


PLOT
MEASUREMENTS
Amplitude
1.5
1.2
0.9
0.6
0.3
0
1
2
3
4
5
Stopped
Frames = 77


Equalized Symbols
File
Tools
View
Help
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=400.000


Error Magnitude
File
Tools
View
Simulation
Help
1
0.8
0.6
0.4
0.2
0
Amplitude
0
1.6666667
5
6.6666667
Time (mins)
Ready
Frame based
T=400.000

Adaptive Equalization with Filtering and Fading Channel

This model shows the behavior of the selected adaptive equalizer in a communication link that has a fading channel. The transmitter and receiver have root raised cosine pulse shaped filtering. A subsystem block enables you to select between linear or decision feedback equalizers that usie the least mean square (LMS) or recursive least square (RLS) adaptive algorithm.

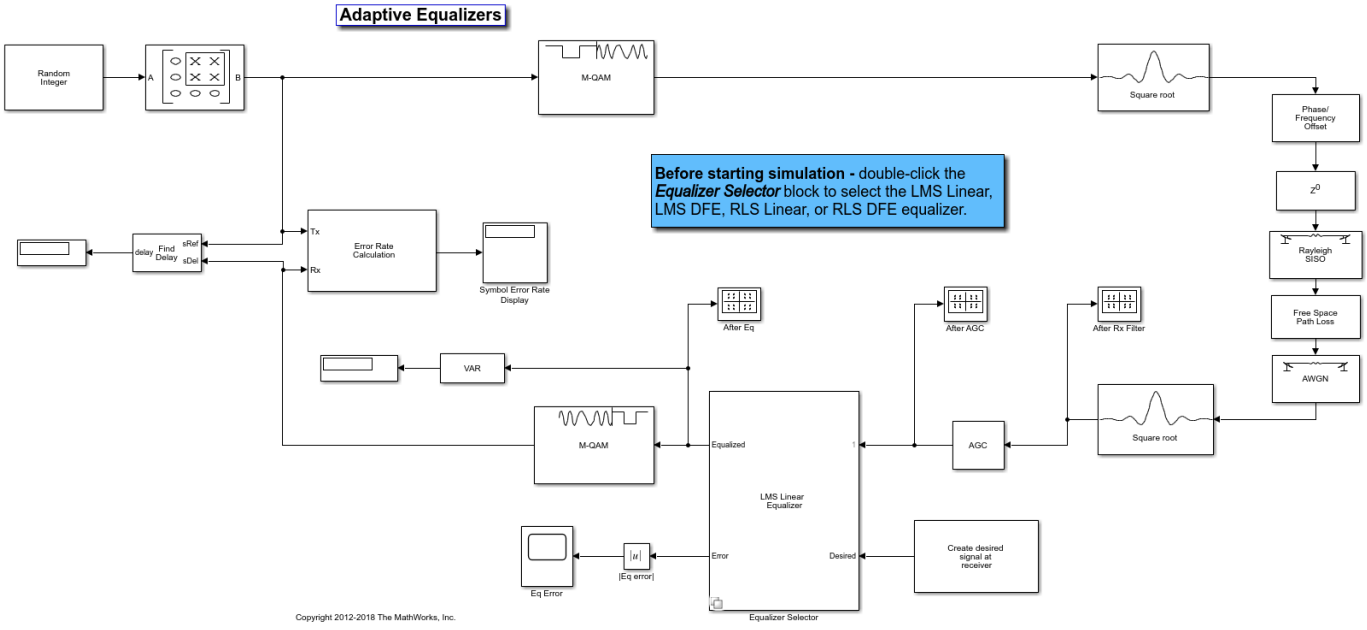
Model Structure

- The transmitter generates 16QAM random signal data that includes a training sequence and applies root raised cosine pulse shaped filtering.
- Channel impairments include multipath fading, Doppler shift, carrier frequency offset, variable integer delay, free space path loss, and AWGN.
- The receiver applies root raised cosine pulse shaped filtering, adjusts the gain, includes equalizer mode control to enable training and enables you to select the equalizer algorithm from these choices.

Selection	Equalizer Algorithm
LMS Linear	Linear least mean square equalizer
LMS DFE	Decision feedback least mean square equalizer
RLS Linear	Linear recursive least square equalizer
RLS DFE	Decision feedback recursive least square equalizer

- Scopes help you understand how the different equalizers and adaptive algorithms behave.

Explore Example Model



Experimenting with the model

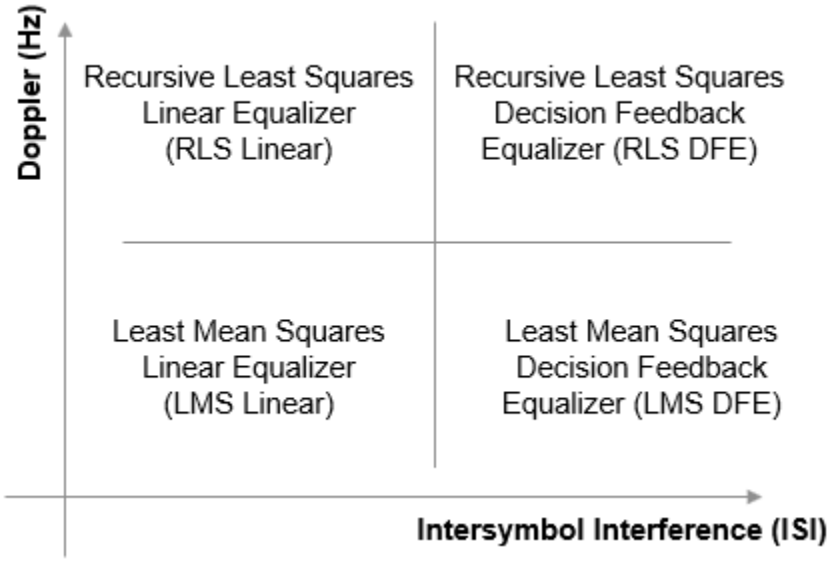
This model provides several ways for you to change settings and observe the results. The `InitFcn` found in `File>Model Properties>Callbacks` calls `cm_ex_adaptive_eq_with_fading_init` to initialize the model. This file enables you to vary settings in the model, including:

- System parameters, such as SNR.
- Pulse shaping filter parameters, such as rolloff and filter length
- Path loss value.
- Channel conditions: Rayleigh or Rician fading, channel path gains, channel path delays, and Doppler shift.
- Equalizer choice and configuration.

Model Considerations

This non-standards-based communication link is representative of a modern communications system.

- The optimal equalizer configuration depends on the channel conditions. The initialization file sets the Doppler shift and multipath fading channel parameters that highlight the capabilities of different equalizers.



- The decision feedback equalizer structure performs better than the linear equalizer structure for higher intersymbol interference.
- The RLS algorithm performs better than the LMS algorithm for higher Doppler frequencies.
- The LMS algorithm executes quickly, converges slowly, and its complexity grows linearly with the number of weights.
- The RLS algorithm converges quickly, its complexity grows approximately as the square of the number of weights. It can be unstable when the number of weights is large.
- The channels exercised for different equalizers have the following characteristics.

Selection	Channel Characteristics
LMS Linear	3 tap multipath fading channel with 10 Hz Doppler shift
LMS DFE	5 tap multipath fading channel with 25 Hz Doppler shift
RLS Linear	2 tap multipath fading channel with 70 Hz Doppler shift
RLS DFE	5 tap multipath fading channel with 100 Hz Doppler shift

- Initial settings for other channel impairments are the same for all equalizers. Carrier frequency offset value is set to 50 Hz. Free space path loss is set to 60 dB. Variable integer delay is set to 2 samples, which requires the equalizers to perform some timing recovery.

Deep channel fades and path loss can cause the equalizer input signal level to be much less than the desired output signal level and result in unacceptably long equalizer convergence time. The `AGC` block adjusts the magnitude of received signal to reduce the equalizer convergence time. You must adjust the optimal gain output power level based on the modulation scheme selected. For 16QAM, a desired output power of 10 W is used.

Training of the equalizer is performed at the beginning of the simulation.

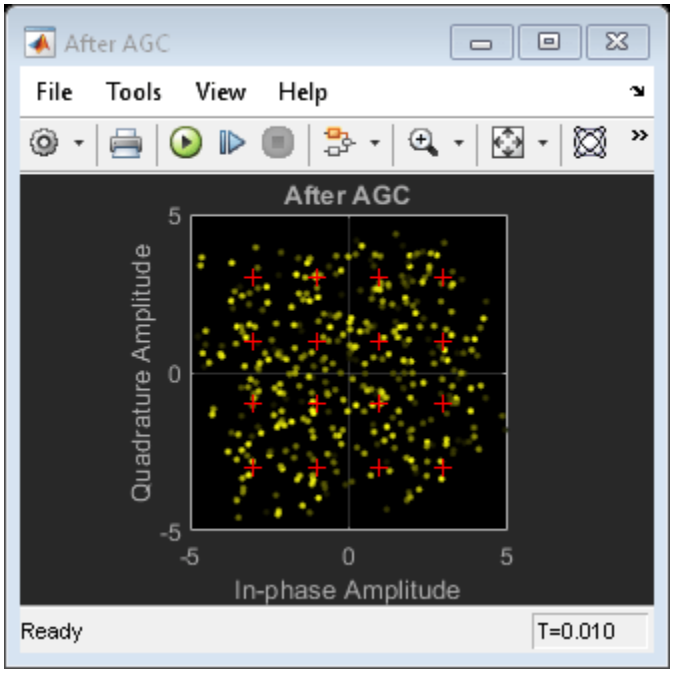
Running the Simulation

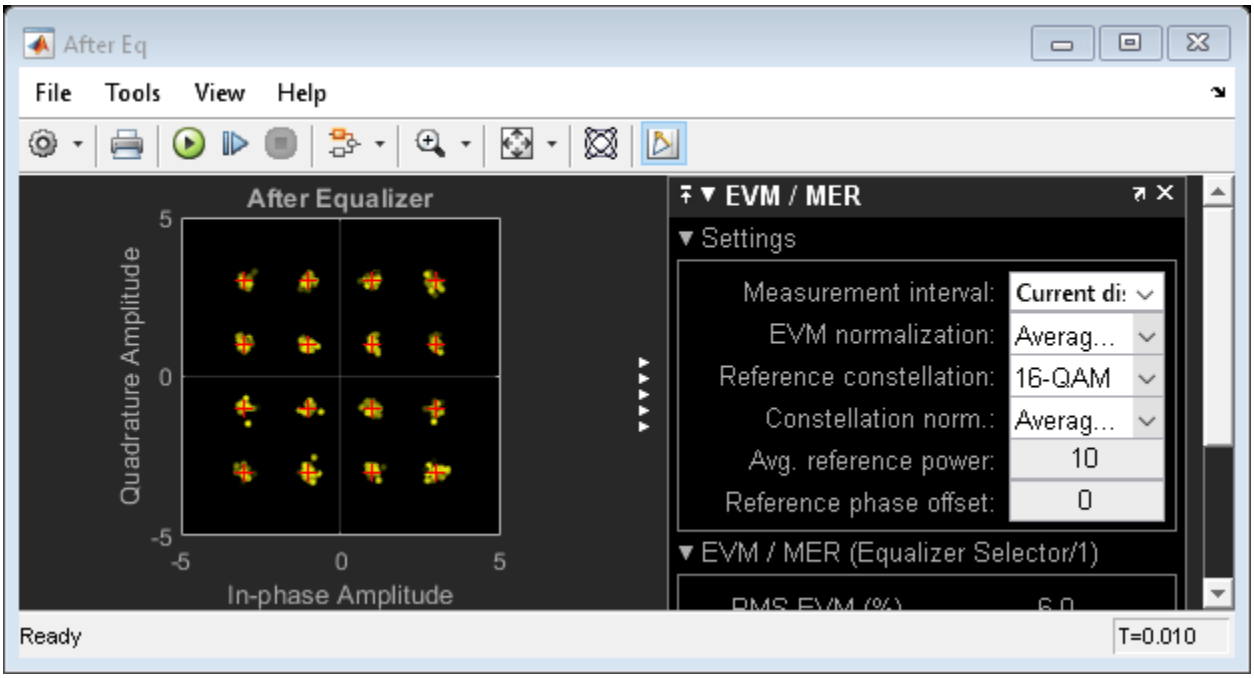
Running the simulation computes symbol error statistics and produces these figures:

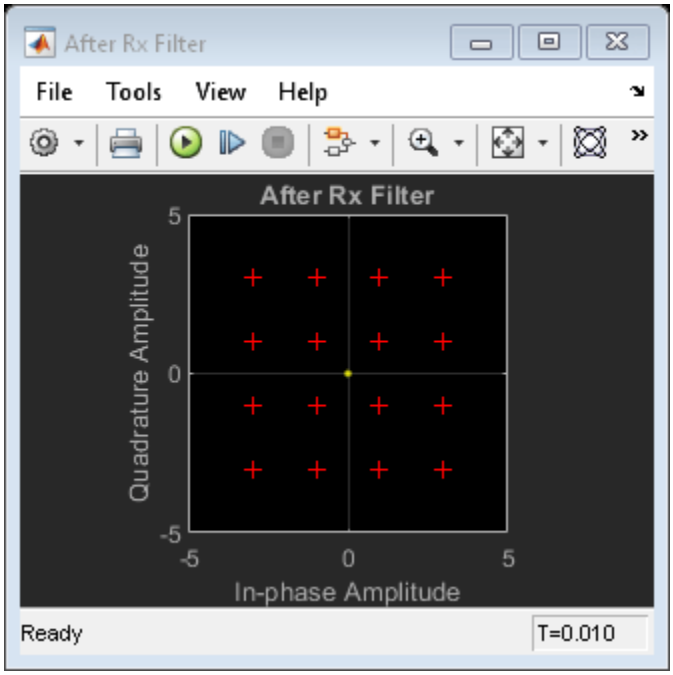
- A constellation diagram of the signal after the receive filter.
- A constellation diagram of the signal after adjusting gain.
- A constellation diagram of the signal after equalization with signal quality measurements shown.
- An equalizer error plot.

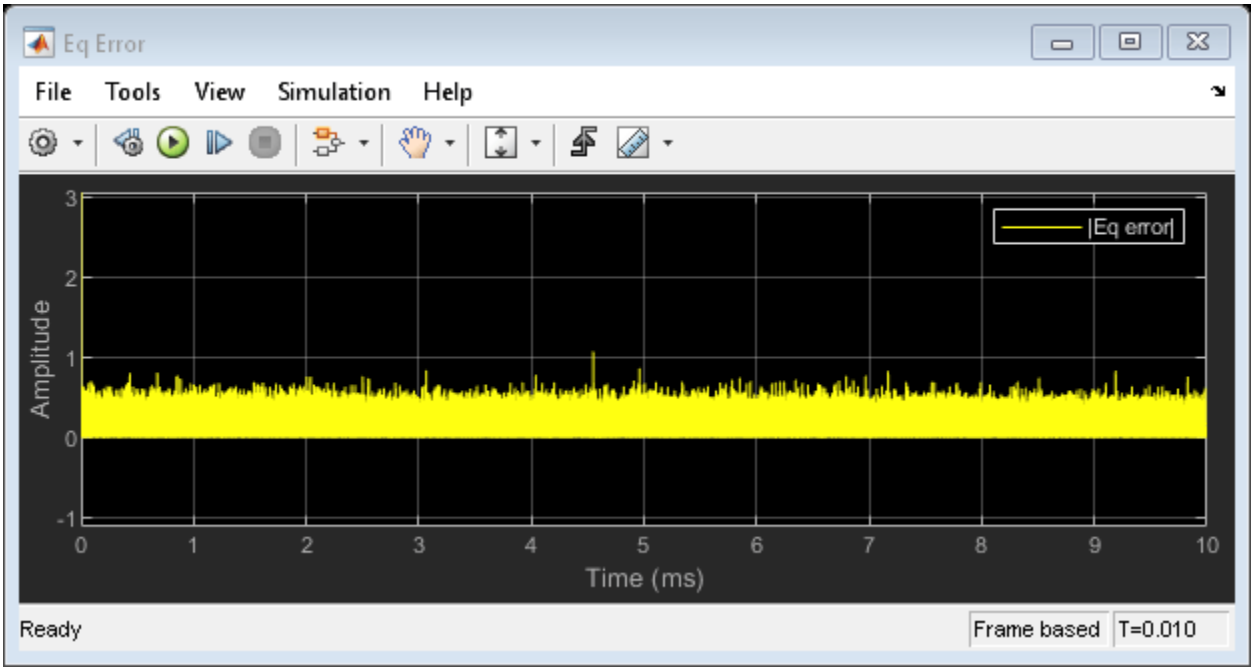
For the plots shown here, the equalizer algorithm selected is `RLS Linear`. Monitoring these figures, you can see that the received signal quality fluctuates as simulation time progresses.

The *After Rx Filter* and *After AGC* constellation plots show the signal before equalization. *After AGC* shows the impact of the channel conditions on the transmitted signal. The *After Eq* plot shows the signal after equalization. The signal plotted in the constellation diagram after equalization shows the variation in signal quality based on the effectiveness of the equalization process. Throughout the simulation, the signal constellations plotted before equalization deviate noticeably from a 16QAM signal constellation. The *After Eq* constellation improves or degrades as the equalizer error signal varies. The `Eq error` plotted in the *Eq Error* plot, indicates poor equalization at the start of the simulation. The error degrades at first then improves as the equalizer converges.


After AGC
File
Tools
View
Help
After AGC
Quadrature Amplitude
In-phase Amplitude
5
0
-5
Ready
T=0.010


After Eq
File
Tools
View
Help
After Equalizer
Quadrature Amplitude
In-phase Amplitude
EVM / MER
Settings
Measurement interval:
Current di
EVM normalization:
Averag...
Reference constellation:
16-QAM
Constellation norm.:
Averag...
Avg. reference power:
10
Reference phase offset:
0
EVM / MER (Equalizer Selector/1)
Ready
T=0.010





Further Exploration

Double-click the `Equalizer Selector` block and select a different equalizer. Run the simulation to see the performance of the various equalizer options. You can use the signal logger to compare the results from this experimentation. In the block diagram, right-click on signal wires and select `Log Selected Signals`. If you have enabled signal logging, after the simulation run finishes, open the Simulation Data Inspector to view the logged signals.

At the MATLAB™ command prompt, enter `edit cm_ex_adaptive_eq_with_fading_init.m` to open the initialization file, then modify a parameter and rerun the simulation. For example, adjust the channel characteristics (params.maxDoppler|, `params.pathDelays`, and `params.pathGains`). The RLS adaptive algorithm performs better than the LMS adaptive algorithm as the maximum Doppler is increased.

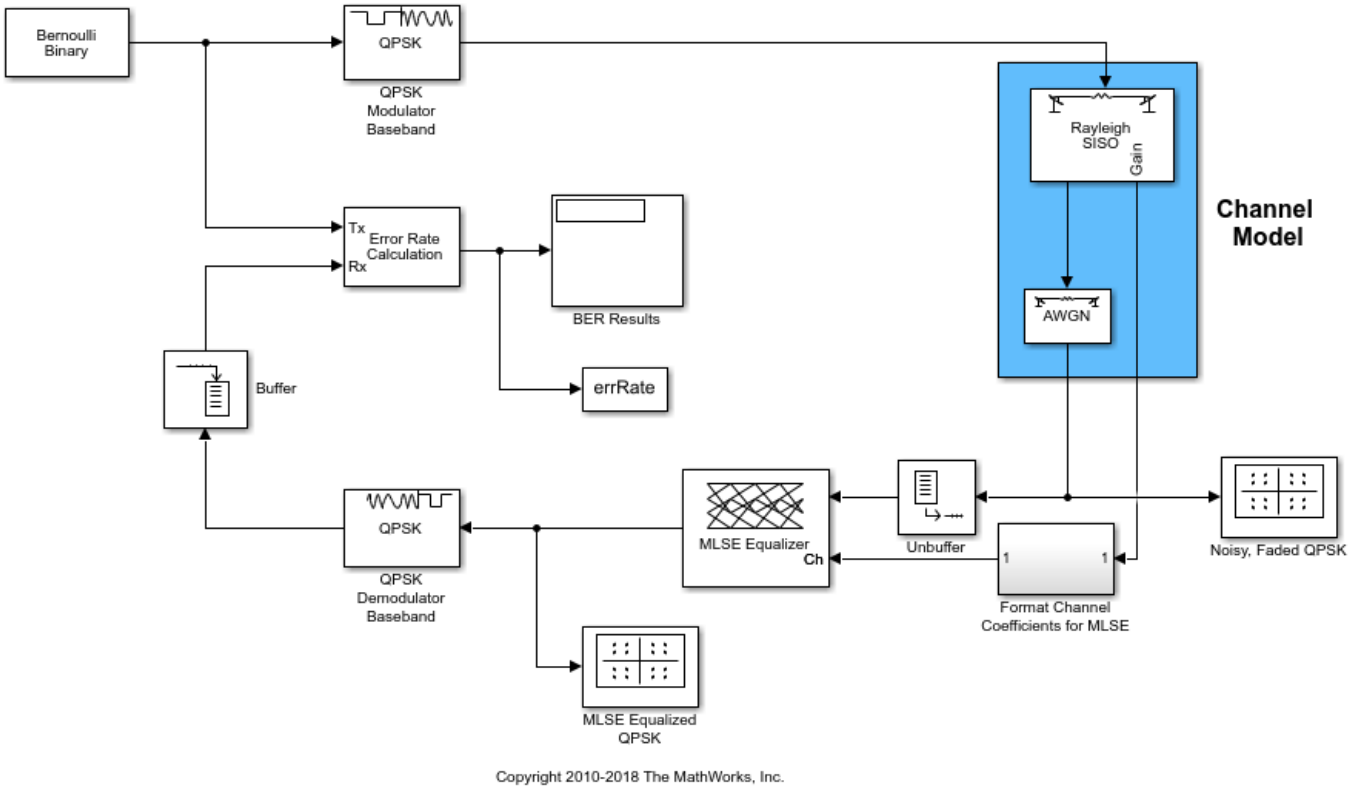
MLSE Equalization with Dynamically Changing Channel

Use a Maximum Likelihood Sequence Estimation (MLSE) equalizer to equalize the effects of a multipath Rayleigh fading channel. The MLSE equalizer inputs data that has passed through a time varying dispersive channel and an estimate of the channel. The channel estimate contains dynamically evolving channel coefficients of a two-path Rayleigh fading channel.

Model Structure

- The transmitter generates QPSK random signal data.
- Channel impairments include multipath fading and AWGN.
- The receiver applies MLSE equalization and QPSK demodulation.
- The model uses scopes and a BER calculation to show the system behavior.

Explore Example Model



Experimenting with the model

The `Bernoulli Binary Generator` block sample time of 5e-6 seconds corresponds to a bit rate of 200 kbps and a QPSK symbol rate of 100 ksym/sec.

The `Multipath Rayleigh Fading Channel` block settings are:

- Maximum Doppler shift is 30 Hz.
- Discrete path delay is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This delay reflects the simplest delay vector for a two-path channel.

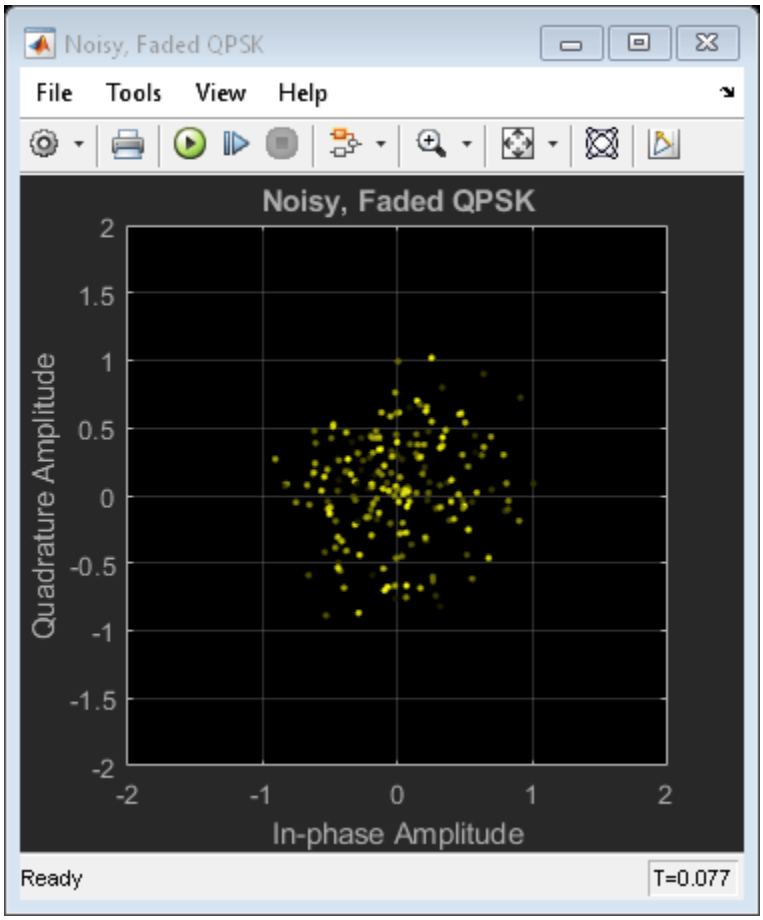
- Average path gain is [0 -10].
- Average path gains are normalized to 0 dB so that the average power input to the `AWGN` block is 1 W.

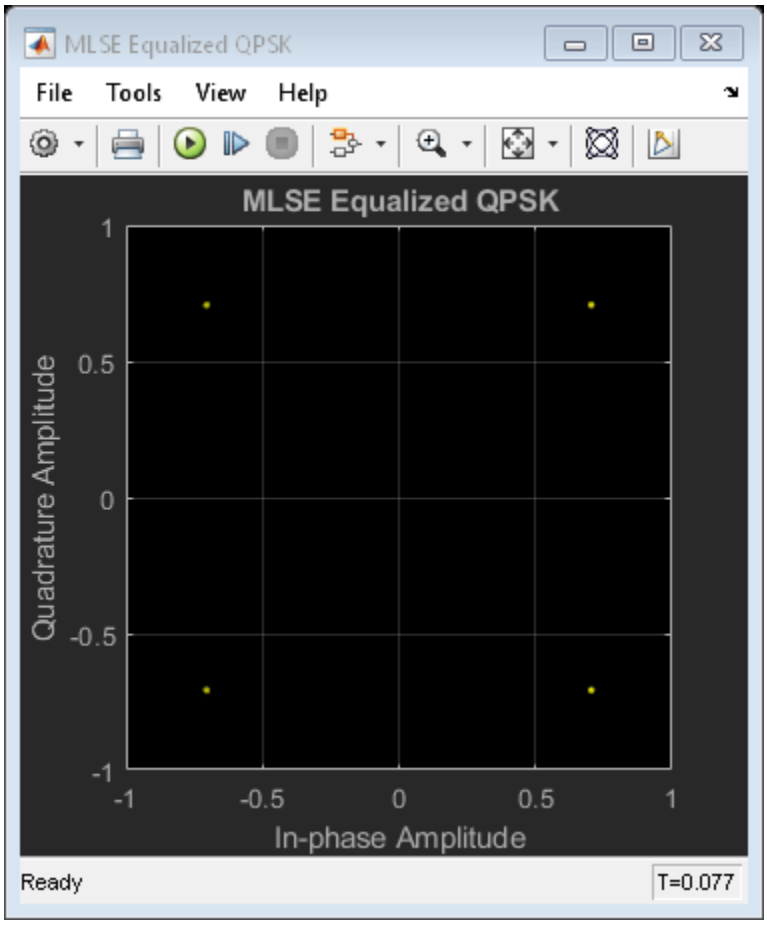
The `MLSE Equalizer` block has the Traceback depth set to 10. Vary this depth to study its effect on Bit Error rate (BER).

The QPSK demodulator accepts an N-by-1 input frame and generates a 2N-by-1 output frame. This output frame and a traceback depth of 10 results in a delay of 20 bits. The model performs frame-based processing on frames that have 100 samples per frame. Due to the frame-based processing, there is a inherent delay of 100 bits in the model. The combined receive delay of 120 is set in the Receive delay parameter of the Error Rate Calculation block, aligning the samples.

The computed BER is displayed. Constellation plots show the constellation before and after equalization.

```
BER = 0.033645
```




MLSE Equalized QPSK
File
Tools
View
Help
MLSE Equalized QPSK
Quadrature Amplitude
In-phase Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
Ready
T=0.077

Adaptive Equalization

This example shows how to a model a communication link with PSK modulation, raised cosine pulse shaping, multipath fading, and adaptive equalization.

The example sets up three equalization scenarios, and calls a separate script to execute the processing loop multiple times for each scenario. Each call corresponds to a transmission block. The pulse shaping and multipath fading channel retain state information from one block to the next. For visualizing the impact of channel fading on adaptive equalizer convergence, the simulation resets the equalizer state every block.

To experiment with different simulation settings, you can edit the example. For instance, you can set the `ResetBeforeFiltering` property of the equalizer object to 0, which will cause the equalizer to retain state from one block to the next.

Transmission Block

Set parameters related to the transmission block which is composed of three parts: training sequence, payload, and tail sequence. All three use the same PSK scheme; the training and tail sequences are used for equalization. We use the default random number generator to ensure the repeatability of the results.

```
Rsym      = 1e6;  % Symbol rate (Hz)
nTrain    = 100;  % Number of training symbols
nPayload  = 400;  % Number of payload symbols
nTail     = 20;   % Number of tail symbols
% Set random number generator for repeatability
hStream   = RandStream.create('mt19937ar', 'seed', 12345);
```

PSK Modulation

Configure the PSK modulation and demodulation System objects.

```
bitsPerSym = 2;                                % Number of bits per PSK symbol
M = 2^bitsPerSym;                              % Modulation order
hPSKMod   = comm.PSKModulator(M, ...
    'PhaseOffset',0, ...
    'SymbolMapping','Binary');
hPSKDemod = comm.PSKDemodulator(M, ...
    'PhaseOffset',0, ...
    'SymbolMapping','Binary');

PSKConstellation = constellation(hPSKMod).'; % PSK constellation
```

Training and Tail Sequences

Generate the training and tail sequences.

```
xTrainData = randi(hStream, [0 M-1], nTrain, 1);
xTailData  = randi(hStream, [0 M-1], nTail, 1);
xTrain     = step(hPSKMod,xTrainData);
xTail      = step(hPSKMod,xTailData);
```

Transmit and Receive Filters

Configure raised cosine transmit and receive filter System objects. The filters incorporate upsampling and downsampling, respectively.

```
chanFilterSpan = 8;  % Filter span in symbols
sampPerSymChan = 4;  % Samples per symbol through channels
hTxFilt = comm.RaisedCosineTransmitFilter( ...
    'RolloffFactor',0.25, ...
    'FilterSpanInSymbols',chanFilterSpan, ...
    'OutputSamplesPerSymbol',sampPerSymChan);

hRxFilt = comm.RaisedCosineReceiveFilter( ...
    'RolloffFactor',0.25, ...
    'FilterSpanInSymbols',chanFilterSpan, ...
    'InputSamplesPerSymbol',sampPerSymChan, ...
    'DecimationFactor',sampPerSymChan);

% Calculate the samples per symbol after the receive filter
sampPerSymPostRx = sampPerSymChan/hRxFilt.DecimationFactor;
% Calculate the delay in samples from both channel filters
chanFilterDelay = chanFilterSpan*sampPerSymPostRx;
```

AWGN Channel

Configure an AWGN channel System object with the `NoiseMethod` property set to `Signal to noise ratio (Es/No)` and `Es/No` set to `20` dB.

```
hAWGNChan = comm.AWGNChannel( ...
    'NoiseMethod','Signal to noise ratio (Es/No)', ...
    'EsNo',20, ...
    'SamplesPerSymbol',sampPerSymChan);
```

Simulation 1: Linear Equalization for Frequency-Flat Fading

Begin with single-path, frequency-flat fading channel. For this channel, the receiver uses a simple 1-tap LMS (least mean square) equalizer, which implements automatic gain and phase control.

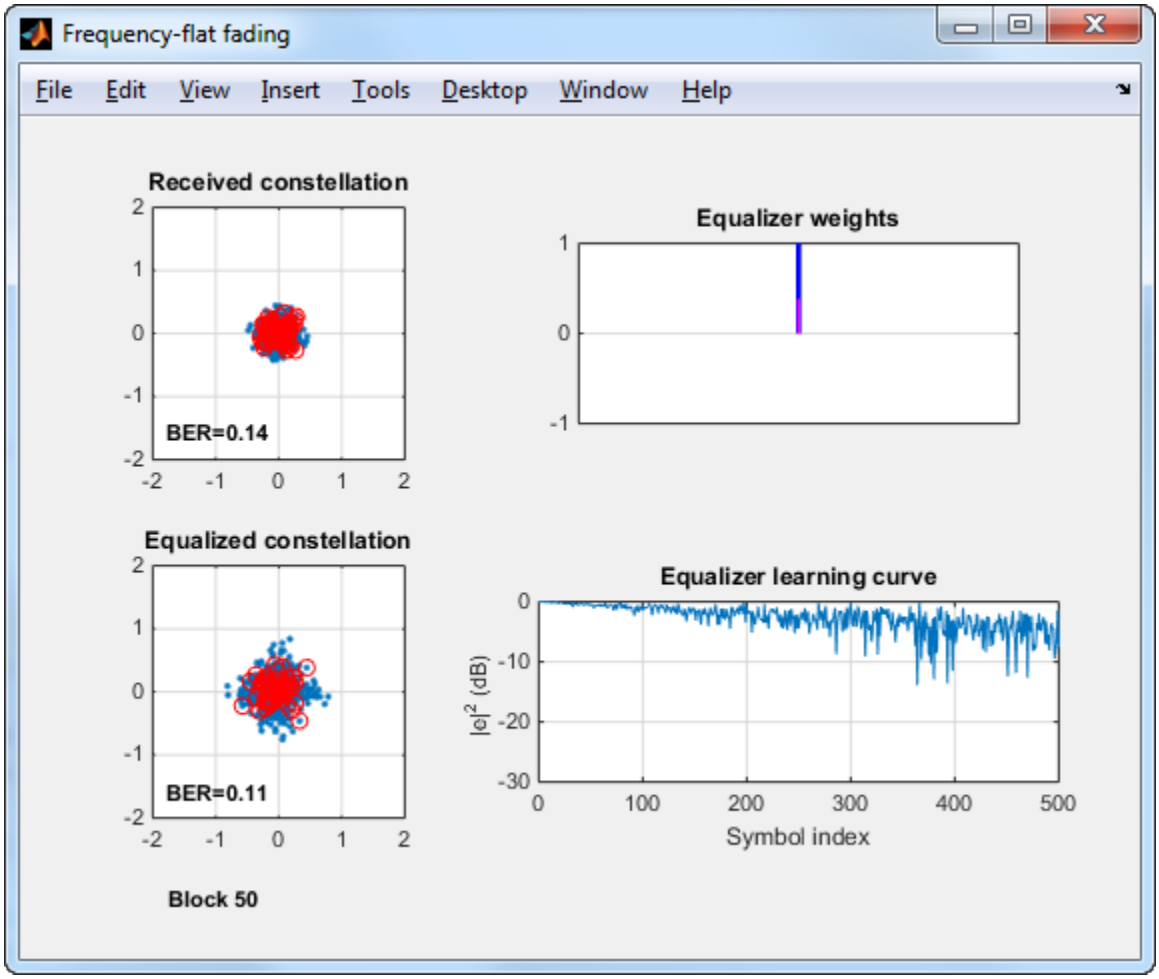
The script `commadapteqloop.m` runs multiple times. Each run corresponds to a transmission block. The equalizer resets its state and weight every transmission block. To retain state from one block to the next, you can set the `ResetBeforeFiltering` property of the equalizer object to `false`.

Before the first run, `commadapteqloop.m` displays the Rayleigh channel System object and the properties of the equalizer object. For each run, a MATLAB figure shows signal processing visualizations. The red circles in the signal constellation plots correspond to symbol errors. In the "Weights" plot, blue and magenta lines correspond to real and imaginary parts, respectively.

```
simName = 'Linear equalization for frequency-flat fading';  % Used to label figure window

% Configure a frequency-flat Rayleigh channel System object with the
% RandomStream property set to 'mt19937ar with seed' for repeatability.
hRayleighChan = comm.RayleighChannel( ...
    'SampleRate',Rsym*sampPerSymChan, ...
    'MaximumDopplerShift',30);

% Configure an adaptive equalizer object
nWeights = 1;  % Single weight
stepSize = 0.1; % Step size for LMS algorithm
alg = lms(stepSize);  % Adaptive algorithm object
eqObj = lineareq(nWeights,alg,PSKConstellation);  % Equalizer object
% Delay in symbols from the equalizer
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;
```

```
% Link simulation
nBlocks = 50;  % Number of transmission blocks in simulation
for block = 1:nBlocks
    commadapteqloop;
end
```

```
System: comm.RayleighChannel

  Properties:
            SampleRate: 4000000
            PathDelays: 0
      AveragePathGains: 0
    NormalizePathGains: true
    MaximumDopplerShift: 30
        DopplerSpectrum: [1x1 struct]
           RandomStream: 'mt19937ar with seed'
                   Seed: 73
    PathGainsOutputPort: false

                 EqType: 'Linear Equalizer'
                AlgType: 'LMS'
               nWeights: 1
             nSampPerSym: 1
                  RefTap: 1
               SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i -1.0000 + 0.0000i -0.0000 - 1.0000i]
               StepSize: 0.1000
          LeakageFactor: 1
                Weights: 0
           WeightInputs: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```



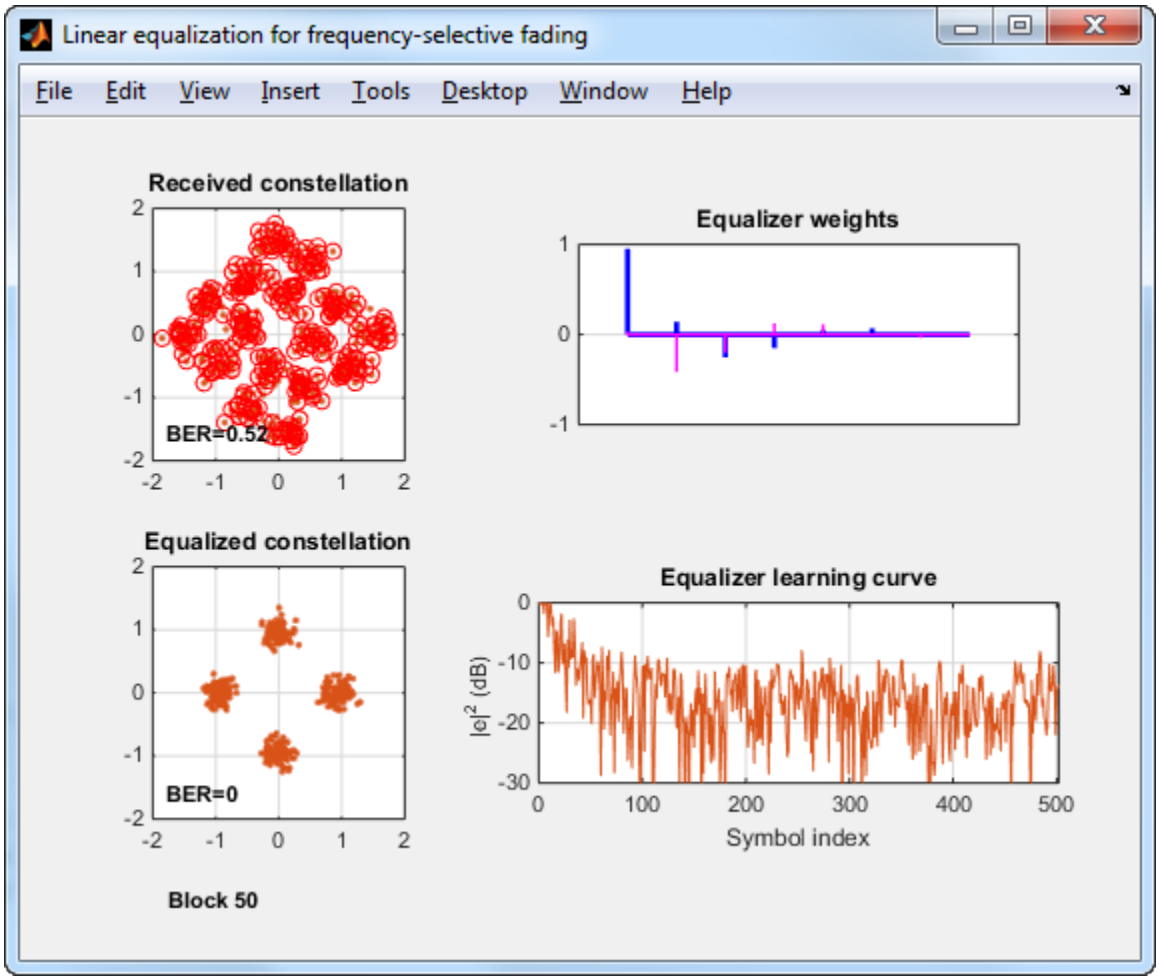
Simulation 2: Linear Equalization for Frequency-Selective Fading

Simulate a three-path, frequency-selective Rayleigh fading channel. The receiver uses an 8-tap linear RLS (recursive least squares) equalizer with symbol-spaced taps.

```
simName = 'Linear equalization for frequency-selective fading';

% Reset transmit and receive filters
reset(hTxFilt);
reset(hRxFilt);

% Set the Rayleigh channel System object to be frequency-selective
release(hRayleighChan);
hRayleighChan.PathDelays = [0 0.9 1.5]/Rsym;
hRayleighChan.AveragePathGains = [0 -3 -6];

% Configure an adaptive equalizer
nWeights = 8;
forgetFactor = 0.99;  % RLS algorithm forgetting factor
alg = rls(forgetFactor);  % RLS algorithm object
eqObj = lineareq(nWeights,alg,PSKConstellation);
eqObj.RefTap = 3;  % Reference tap
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;
```

```
% Link simulation and store BER values
BERvect = zeros(1,nBlocks);
for block = 1:nBlocks
    commadapteqloop;
    BERvect(block) = BEREq;
end
avgBER2 = mean(BERvect)
```

```
System: comm.RayleighChannel

  Properties:
            SampleRate: 4000000
            PathDelays: [0 9e-07 1.5e-06]
      AveragePathGains: [0 -3 -6]
    NormalizePathGains: true
    MaximumDopplerShift: 30
        DopplerSpectrum: [1x1 struct]
           RandomStream: 'mt19937ar with seed'
                   Seed: 73
    PathGainsOutputPort: false

                 EqType: 'Linear Equalizer'
                AlgType: 'RLS'
               nWeights: 8
             nSampPerSym: 1
                  RefTap: 3
               SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i -1.0000 + 0.0000i -0.0000 - 1.0000i]
           ForgetFactor: 0.9900
            InvCorrInit: 0.1000
          InvCorrMatrix: [8x8 double]
                Weights: [0 0 0 0 0 0 0 0]
           WeightInputs: [0 0 0 0 0 0 0 0]
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0


avgBER2 =

   3.0000e-04
```



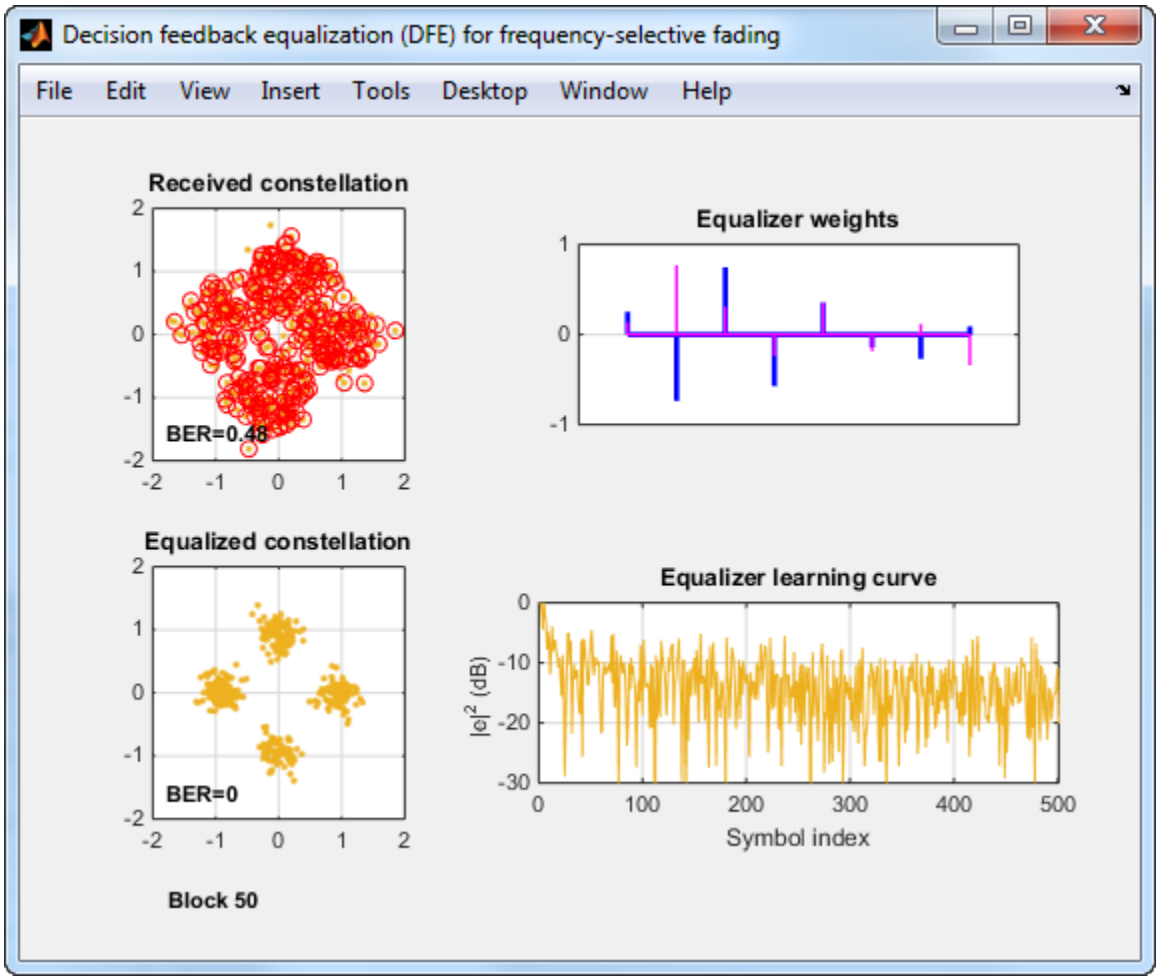
Simulation 3: Decision feedback Equalization (DFE) for Frequency-Selective Fading

The receiver uses a DFE with a six-tap fractionally spaced forward filter (two samples per symbol) and two feedback weights. The DFE uses the same RLS algorithm as in Simulation 2. The receive filter structure is reconstructed to account for the increased number of samples per symbol.

```
simName = 'Decision feedback equalization (DFE) for frequency-selective fading';

% Reset transmit filter and adjust receive filter decimation factor
reset(hTxFilt);
release(hRxFilt);
hRxFilt.DecimationFactor = 2;
sampPerSymPostRx = sampPerSymChan/hRxFilt.DecimationFactor;
chanFilterDelay = chanFilterSpan*sampPerSymPostRx;

% Reset fading channel
reset(hRayleighChan);

% Configure an adaptive equalizer object
nFwdWeights = 6;  % Number of feedforward equalizer weights
nFbkWeights = 2;  % Number of feedback filter weights
eqObj = dfe(nFwdWeights, nFbkWeights,alg,PSKConstellation,sampPerSymPostRx);
eqObj.RefTap = 3;
```

```
eqDelayInSym = (eqObj.RefTap-1)/sampPerSymPostRx;

for block = 1:nBlocks
    commadapteqloop;
    BERvect(block) = BEREq;
end
avgBER3 = mean(BERvect)
```

```
System: comm.RayleighChannel

  Properties:
            SampleRate: 4000000
            PathDelays: [0 9e-07 1.5e-06]
      AveragePathGains: [0 -3 -6]
    NormalizePathGains: true
   MaximumDopplerShift: 30
       DopplerSpectrum: [1x1 struct]
          RandomStream: 'mt19937ar with seed'
                  Seed: 73
   PathGainsOutputPort: false

                EqType: 'Decision Feedback Equalizer'
               AlgType: 'RLS'
              nWeights: [6 2]
            nSampPerSym: 2
                 RefTap: 3
              SigConst: [1.0000 + 0.0000i 0.0000 + 1.0000i -1.0000 + 0.0000i -0.0000 - 1.0000i]
           ForgetFactor: 0.9900
            InvCorrInit: 0.1000
          InvCorrMatrix: [8x8 double]
                Weights: [0 0 0 0 0 0 0 0]
           WeightInputs: [0 0 0 0 0 0 0 0]
   ResetBeforeFiltering: 1
    NumSamplesProcessed: 0


avgBER3 =

     0
```



Summary

This example showed the relative performance of linear and decision feedback equalizers in both frequency-flat and frequency-selective fading channels. It showed how a one-tap equalizer is sufficient to compensate for a frequency-flat channel, but that a frequency-selective channel requires an equalizer with multiple taps. Finally, it showed that a decision feedback equalizer is superior to a linear equalizer in a frequency-selective channel.

Appendix

This example uses the following script and helper functions:

- commadapteqloop.m
- commadapteq_checkvars.m
- commadapteq_graphics.m

Equalize BSPK Signal

Equalize a BPSK signal using a linear equalizer with a least mean square (LMS) algorithm.

Generate random binary data and apply BPSK modulation.

```
M = 2;
data = randi([0 1],1000,1);
modData = pskmod(data,M);
```

Apply two-tap static fading to the modulated signal and add AWGN noise.

```
rxSig = conv(modData,[0.02+0.5i 0.05]);
rxSig = awgn(rxSig,30);
```

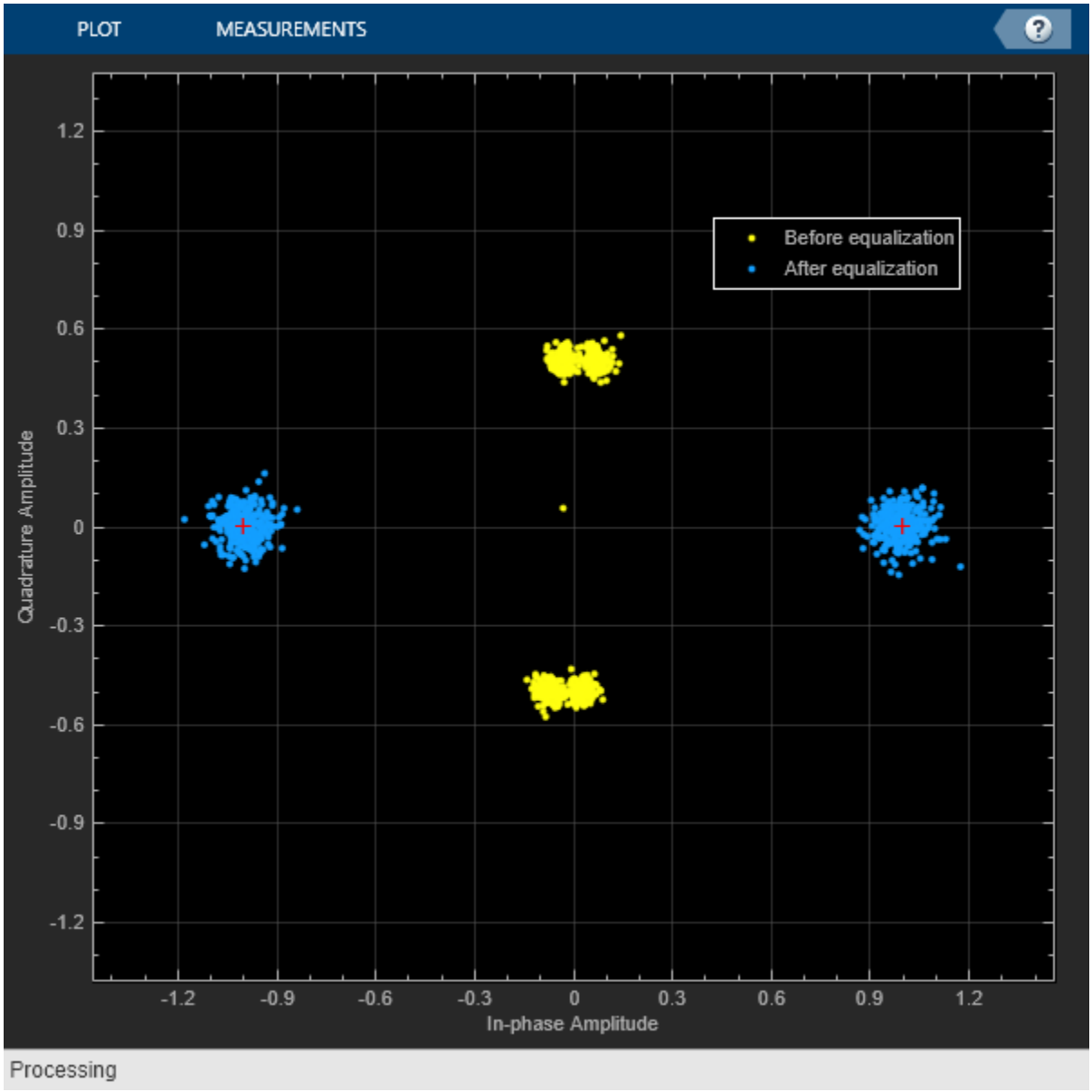
Create a linear equalizer System object™ configured to use the LMS adaptive algorithm, 8 taps, 0.1 step size, and the 4th tap as the reference tap. Set the constellation to match the modulation of the transmitted signal.

```
lineq = comm.LinearEqualizer( ...
    NumTaps=8, ...
    StepSize=0.1, ...
    Constellation=complex([-1 1]), ...
    ReferenceTap=4)
```

```
lineq =
  comm.LinearEqualizer with properties:

               Algorithm: 'LMS'
                 NumTaps: 8
                StepSize: 0.1000
           Constellation: [-1.0000 + 0.0000i 1.0000 + 0.0000i]
            ReferenceTap: 4
              InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: true
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

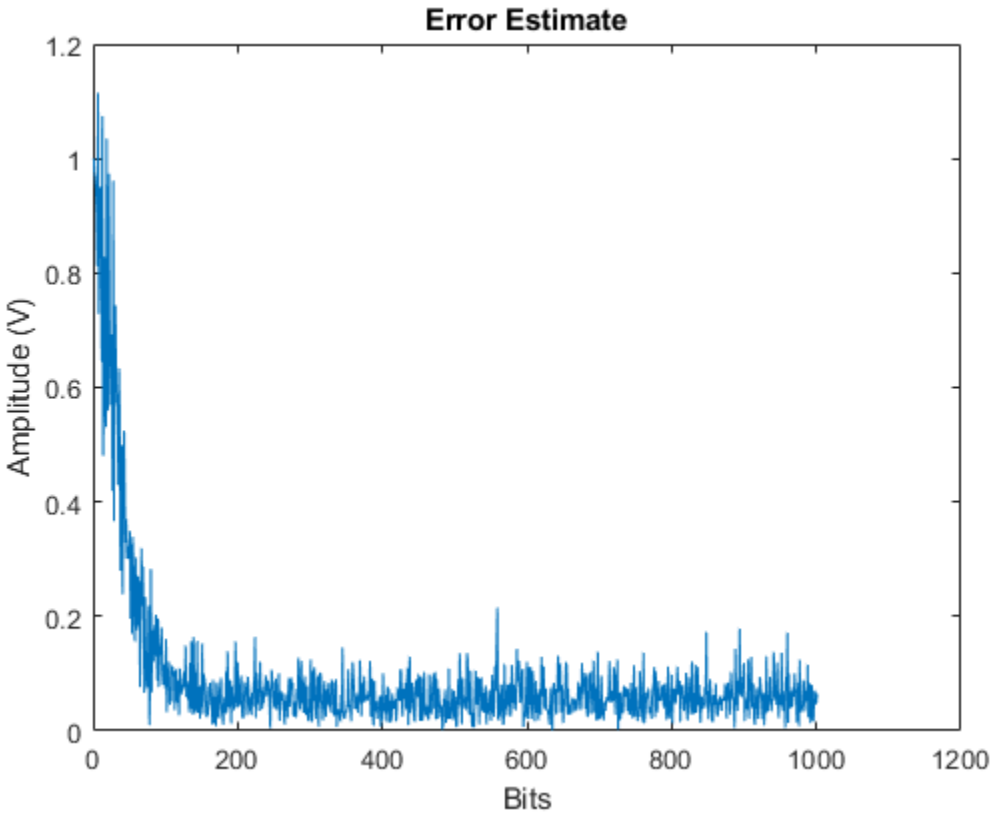
Equalize the received signal, `rxSig`. Use the first 200 data bits as a training sequence. Display a constellation diagram showing the received signal before and after equalization.

```
trSeq = modData(1:200);
[eqSig,err] = lineq(rxSig,trSeq);

constdiag = comm.ConstellationDiagram( ...
    NumInputPorts=2, ...
    ChannelNames={'Before equalization','After equalization'}, ...
    ReferenceConstellation=pskmod([0 M-1],M));
constdiag(rxSig(400:end),eqSig(400:end))
```



Plot the magnitude of the error estimate. As shown by the decrease and stabilizing of the error signal, the equalization converges in less than 200 bits.

```
plot(abs(err))
title('Error Estimate')
xlabel('Bits')
ylabel('Amplitude (V)')
```


Error Estimate
Amplitude (V)
Bits
0
0.2
0.4
0.6
0.8
1
1.2
0
200
400
600
800
1000
1200

Compare RLS and LMS Algorithms

Equalize a QAM signal passed through a frequency-selective fading channel using RLS and LMS algorithms. Compare the performance of the two algorithms.

Specify the modulation order. Generate the corresponding QAM reference constellation.

```
M = 16;
sigConst = qammod(0:M-1,M,'UnitAveragePower',true);
```

Create a frequency-selective static channel having three taps.

```
rchan = comm.RayleighChannel('SampleRate',1000, ...
    'PathDelays',[0 1e-3 2e-3],'AveragePathGains',[0 -3 -6], ...
    'MaximumDopplerShift',0, ...
    'RandomStream','mt19937ar with seed','Seed',73);
```

RLS Equalizer

Create an RLS equalizer object.

```
eqrls = lineareq(6,rls(0.99,0.1));
eqrls.SigConst = sigConst;
eqrls.ResetBeforeFiltering = 0;
```

Generate and QAM modulate a random training sequence. Pass the sequence through the Rayleigh fading channel. Pass the received signal and the training signal through the equalizer to set the equalizer tap weights.

```
trainData = randi([0 M-1],200,1);
trainSig = qammod(trainData,M,'UnitAveragePower',true);
rxSig = rchan(trainSig);
[~,~,errorSig] = equalize(eqrls,rxSig,trainSig);
```

Plot the magnitude of the error estimate.

```
plot(abs(errorSig))
title('Error Estimate, RLS Equalizer')
xlabel('Symbols')
ylabel('Amplitude')
```

The error is nearly eliminated within 200 symbols.

Transmit a QAM signal through a frequency-selective channel. Equalize the received signal using the previously 'trained' RLS equalizer. Measure the time required to execute the processing loop.

```
tic
for k = 1:20
   data = randi([0 M-1],1000,1); % Random message
   txSig = qammod(data,M,'UnitAveragePower',true);

   % Introduce channel distortion.
   rxSig = rchan(txSig);

   % Equalize the received signal.
   eqSig = equalize(eqrls,rxSig);
```

```
end
rlstime = toc;
```

Plot the constellation diagram of the received and equalized signals.

```
h = scatterplot(rxSig,1,0,'c.');
hold on
scatterplot(eqSig,1,0,'b.',h)
legend('Received Signal','Equalized Signal')
title('RLS Equalizer')
hold off
```

The equalizer removed the effects of the fading channel.

LMS Equalizer

Repeat the equalization process with an LMS equalizer. Create an LMS equalizer object.

```
eqlms = lineareq(6,lms(0.03));
eqlms.SigConst = sigConst;
eqlms.ResetBeforeFiltering = 0;
```

Train the LMS equalizer.

```
trainData = randi([0 M-1],1000,1);
trainSig = qammod(trainData,M,'UnitAveragePower',true);
rxSig = rchan(trainSig);
[~,~,errorSig] = equalize(eqlms,rxSig,trainSig);
```

Plot the magnitude of the error estimate.

```
plot(abs(errorSig))
title('Error Estimate, LMS Equalizer')
xlabel('Symbols')
ylabel('Amplitude')
```

Training the LMS equalizer requires 1000 symbols.

Transmit a QAM signal through the same frequency-selective channel. Equalize the received signal using the previously 'trained' LMS equalizer. Measure the time required to execute the processing loop.

```
tic
for k = 1:20
   data = randi([0 M-1],1000,1); % Random message
   txSig = qammod(data,M,'UnitAveragePower',true);

   % Introduce channel distortion.
   rxSig = rchan(txSig);

   % Equalize the received signal.
   eqSig = equalize(eqlms,rxSig);

end
lmstime = toc;
```

Plot the constellation diagram of the received and equalized signals.

```
h = scatterplot(rxSig,1,0,'c.');
hold on
```

```
scatterplot(eqSig,1,0,'b.',h)
legend('Received Signal','Equalized Signal')
title('LMS Equalizer')
```

The equalizer removes the effects of the fading channel.

Compare the loop execution time for the two equalizer algorithms.

```
[rlstime lmstime]
```

The LMS algorithm is more computationally efficient as it took 50% of the time to execute the processing loop. However, the training sequence required by the LMS algorithm is 5 times longer.

System Design

Source Coding

In this section...

Represent Partitions

Scalar quantization is a process that maps all inputs within a specified range to a common value. This process maps inputs in a different range of values to a different common value. In effect, scalar quantization digitizes an analog signal. Two parameters determine a quantization: a partition on page 16-2 and a codebook on page 16-2.

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

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

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

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

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

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

Represent Codebooks

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

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

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

Determine Which Interval Each Input Is In

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

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

The output is

```
index =

     0
     6
     5
```

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

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

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

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

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

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

Optimize Quantization Parameters

Section Overview

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

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

Example: Optimizing Quantization Parameters

The `lloyds` function optimizes the partition and codebook according to the Lloyd algorithm. The code below optimizes the partition and codebook for one period of a sinusoidal signal, starting from a rough initial guess. Then it uses these parameters to quantize the original signal using the initial

guess parameters as well as the optimized parameters. The output shows that the mean square distortion after quantizing is much less for the optimized parameters. The `quantiz` function automatically computes the mean square distortion and returns it as the third output parameter.

```
% Start with the setup from 2nd example in "Quantizing a Signal."
t = [0:.1:2*pi];
sig = sin(t);
partition = [-1:.2:1];
codebook = [-1.2:.2:1];
% Now optimize, using codebook as an initial guess.
[partition2,codebook2] = lloyds(sig,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
[index2,quant2,distor2] = quantiz(sig,partition2,codebook2);
% Compare mean square distortions from initial and optimized
[distor, distor2] % parameters.
```

The output is

```
ans =

    0.0148    0.0024
```

Differential Pulse Code Modulation

Section Overview

The quantization in the section “Quantize a Signal” on page 16-11 requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM).

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

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in “Represent Partitions” on page 16-2 and “Represent Codebooks” on page 16-2, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

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

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

Represent Predictors

If the guess for the kth value of the signal x, based on earlier values of x, is

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

then the corresponding predictor vector for toolbox functions is

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

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

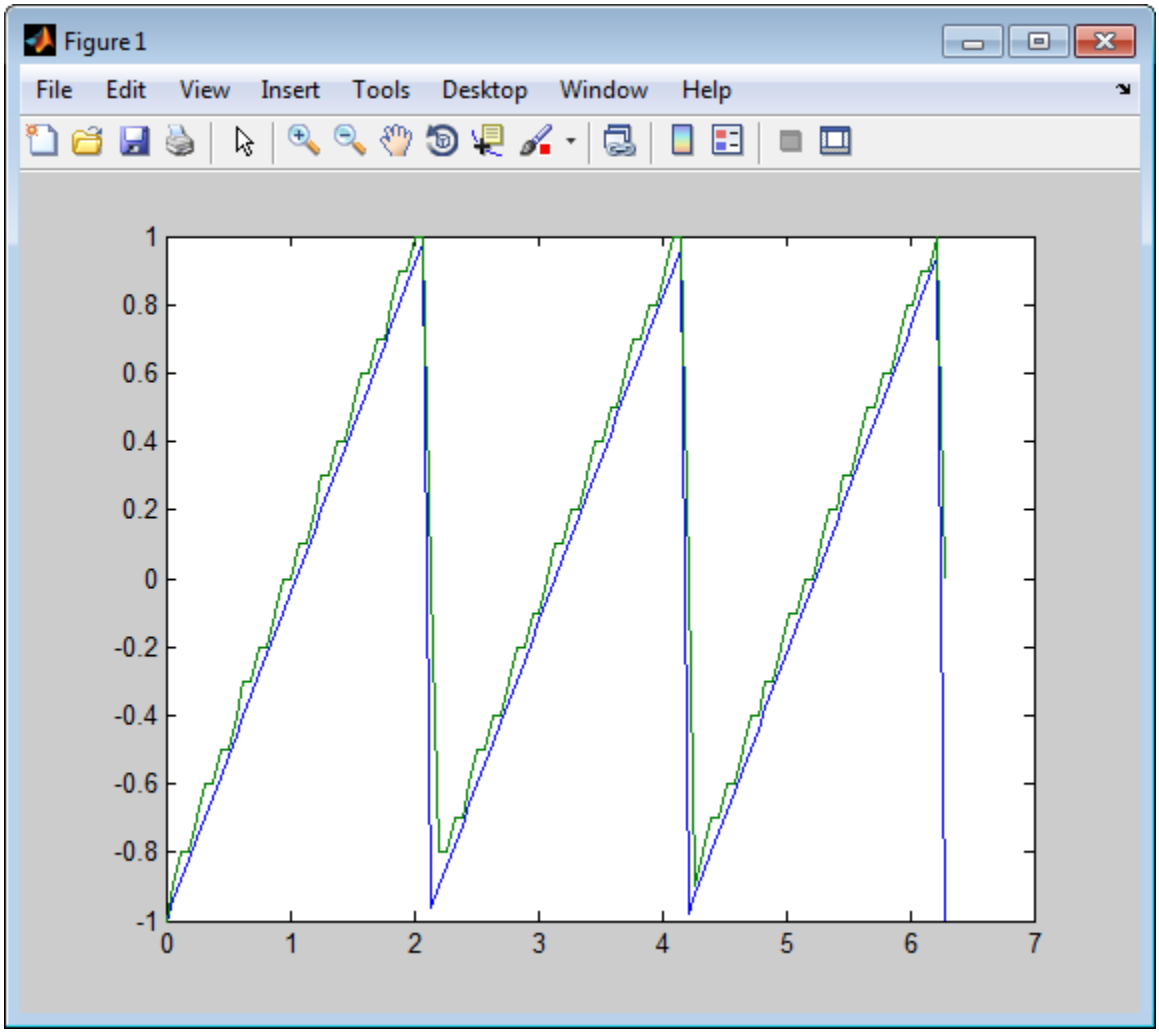
Example: DPCM Encoding and Decoding

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

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

The output is

```
distor =

    0.0327
```



Optimize DPCM Parameters

Section Overview

The section "Optimize Quantization Parameters" on page 16-3 describes how to use training data with the `lloyds` function to help find quantization parameters that will minimize signal distortion.

This section describes similar procedures for using the `dpcmopt` function in conjunction with the two functions `dpcmenco` and `dpcmdeco`, which first appear in the previous section.

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

Example: Comparing Optimized and Nonoptimized DPCM Parameters

This example is similar to the one in the last section. However, where the last example created `predictor`, `partition`, and `codebook` in a straightforward but haphazard way, this example uses the same codebook (now called `initcodebook`) as an initial guess for a new *optimized* codebook parameter. This example also uses the predictive order, 1, as the desired order of the new optimized predictor. The `dpcmopt` function creates these optimized parameters, using the sawtooth signal `x` as training data. The example goes on to quantize the training data itself; in theory, the optimized parameters are suitable for quantizing other data that is similar to `x`. Notice that the mean square distortion here is much less than the distortion in the previous example.

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

The output is

```
distor =

    0.0063
```

Compand a Signal

Section Overview

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

For more information, see `quantiz` and `compand`.

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
```

Huffman Coding

Section Overview

Huffman coding offers a way to compress data. The average length of a Huffman code depends on the statistical frequency with which the source produces each symbol from its alphabet. A Huffman code dictionary, which associates each data symbol with a codeword, has the property that no codeword in the dictionary is a prefix of any other codeword in the dictionary.

The `huffmandict`, `huffmanenco`, and `huffmandeco` functions support Huffman coding and decoding.

Note For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding. To learn how to use arithmetic coding, see “Arithmetic Coding” on page 16-10.

Huffman coding requires statistical information about the source of the data being encoded. In particular, the `p` input argument in the `huffmandict` function lists the probability with which the source produces each symbol in its alphabet.

For example, consider a data source that produces 1s with probability 0.1, 2s with probability 0.1, and 3s with probability 0.8. The main computational step in encoding data from this source using a Huffman code is to create a dictionary that associates each data symbol with a codeword. The example here creates such a dictionary and then shows the codeword vector associated with a particular value from the data source.

Create a Huffman Code Dictionary Using MATLAB

This example shows how to create a Huffman code dictionary using the `huffmandict` function.

Create a vector of data symbols and assign a probability to each.

```
symbols = [1 2 3];
prob = [0.1 0.1 0.8];
```

Create a Huffman code dictionary. The most probable data symbol, 3, is associated with a one-digit codeword, while less probable data symbols are associated with two-digit codewords.

```
dict = huffmandict(symbols,prob)
```

```
dict=3×2 cell array
    {[1]}    {[1 1]}
    {[2]}    {[1 0]}
    {[3]}    {[  0]}
```

Display the second row of the dictionary. The output also shows that a Huffman encoder receiving the data symbol 2 substitutes the sequence `1 0`.

```
dict{2,:}
```

```
ans = 2
```

```
ans = 1×2

     1     0
```

Create and Decode a Huffman Code Using MATLAB

The example performs Huffman encoding and decoding using a source whose alphabet has three symbols. Notice that the `huffmanenco` and `huffmandeco` functions use the dictionary created by `huffmandict`.

Generate a data sequence to encode.

```
sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
```

Define the set of data symbols and the probability associated with each element.

```
symbols = [1 2 3];
p = [0.1 0.1 0.8];
```

Create the Huffman code dictionary.

```
dict = huffmandict(symbols,p);
```

Encode and decode the data. Verify that the original data, `sig`, and the decoded data, `dhsig`, are identical.

```
hcode = huffmanenco(sig,dict);
dhsig = huffmandeco(hcode,dict);
isequal(sig,dhsig)
```

```
ans = logical
   1
```

Arithmetic Coding

Section Overview

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

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

Represent Arithmetic Coding Parameters

Arithmetic coding requires statistical information about the source of the data being encoded. In particular, the `counts` input argument in the `arithenco` and `arithdeco` functions lists the frequency with which the source produces each symbol in its alphabet. You can determine the frequencies by studying a set of test data from the source. The set of test data can have any size you choose, as long as each symbol in the alphabet has a nonzero frequency.

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

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

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

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

Create and Decode an Arithmetic Code Using MATLAB

Encode and decode a sequence from a source having three symbols.

Create a sequence vector containing symbols from the set of {1,2,3}.

```
seq = [3 3 1 3 3 3 3 3 2 3];
```

Set the `counts` vector to define an encoder that produces 10 ones, 20 twos, and 70 threes from a typical 100-symbol set of test data.

```
counts = [10 20 70];
```

Apply the arithmetic encoder and decoder functions.

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

Verify that the decoder output matches the original input sequence.

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

```
ans = logical
   1
```

Quantize a Signal

Scalar Quantization Example 1

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

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

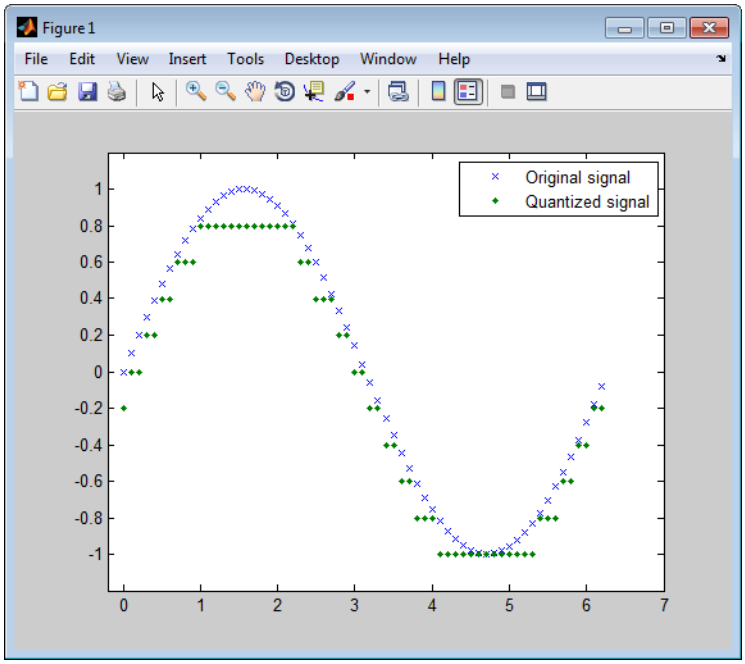
The output is below.

```
quantized =

  Columns 1 through 6

   -1.0000   -1.0000   -1.0000   -1.0000    0.5000    0.5000

  Columns 7 through 12

    2.0000    2.0000    2.0000    2.0000    2.0000    3.0000

  Column 13

    3.0000
```

Scalar Quantization Example 2

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

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



See Also

Functions

`lloyds` | `quantiz` | `dpcmenco` | `dpcmdeco` | `dpcmopt`

Error Detection and Correction

In this section...

Cyclic Redundancy Check Codes

CRC-Code Features

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

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

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

Communications Toolbox supports CRC Coding using Simulink blocks, System objects, or MATLAB objects. These CRC coding features are listed in “Error Detection and Correction”.

CRC Non-Direct Algorithm

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

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

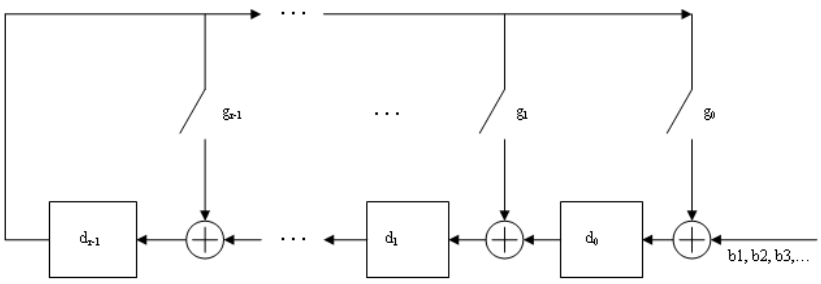
The CRC generation feature, which implements the transmission phase of the CRC algorithm, does the following:

1. Left shifts the input data vector by *r* bits and divides the corresponding polynomial by *P*.
2. Sets the checksum equal to the binary vector of length *r*, corresponding to the remainder from step 1.
3. Appends the checksum to the input data vector. The result is the output vector.

The CRC detection feature computes the checksum for its entire input vector, as described above.

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

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

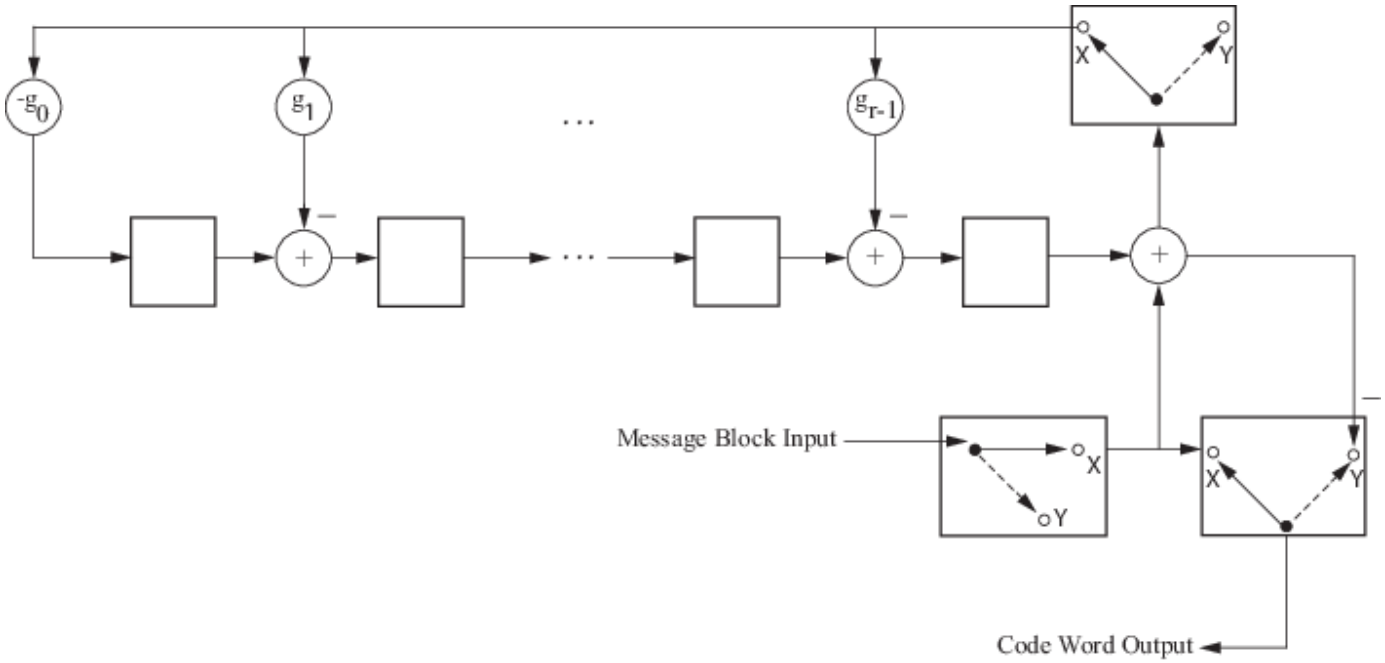


Bits enter the linear feedback shift register (LFSR) from the lowest index bit to the highest index bit. The sequence of input message bits represents the coefficients of a message polynomial in order of decreasing powers. The message vector is augmented with *r* zeros to flush out the LFSR, where *r* is the degree of the generator polynomial. If the output from the leftmost register stage d(1) is a 1, then the bits in the shift register are XORed with the coefficients of the generator polynomial. When the augmented message sequence is completely sent through the LFSR, the register contains the checksum [d(1) d(2) . . . d(r)]. This is an implementation of binary long division, in which the message sequence is the divisor (numerator) and the polynomial is the dividend (denominator). The CRC checksum is the remainder of the division operation.

Example Using CRC Non-Direct Algorithm

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

CRC Direct Algorithm



where

Message Block Input is

$$m_0, m_1, \ldots, m_{k-1}$$

Code Word Output is

$$c_0, c_1, \ldots, c_{n-1} = \underbrace{m_0, m_1, \ldots, m_{k-1}}_{X}, \underbrace{d_0, d_1, \ldots, d_{n-k-1}}_{Y}$$

The initial step of the direct CRC encoding occurs with the three switches in position X. The algorithm feeds *k* message bits to the encoder. These bits are the first *k* bits of the code word output. Simultaneously, the algorithm sends *k* bits to the linear feedback shift register (LFSR). When the system completely feeds the *k*th message bit to the LFSR, the switches move to position Y. Here, the LFSR contains the mathematical remainder from the polynomial division. These bits are shifted out of the LFSR and they are the remaining bits (checksum) of the code word output.

Selected Bibliography for CRC Coding

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

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

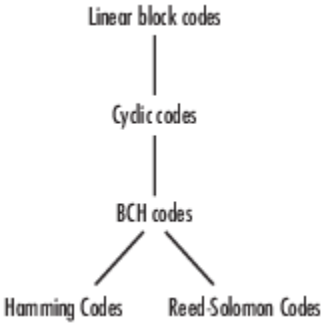
Block Codes

Block-Coding Features

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

Block coding is a special case of error-control coding. Block-coding techniques map a fixed number of message symbols to a fixed number of code symbols. A block coder treats each block of data independently and is a memoryless device. Communications Toolbox contains block-coding capabilities by providing Simulink blocks, System objects, and MATLAB functions.

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



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

Note The blocks and functions in Communications Toolbox are designed for error-control codes that use an alphabet having 2 or 2^m symbols.

Communications Toolbox Support Functions

Functions in Communications Toolbox can support simulation blocks by

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

For more information about error-control coding capabilities, see Block Codes on page 16-17.

Terminology

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

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

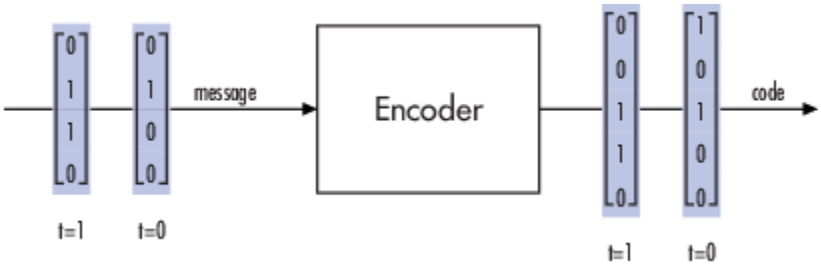
Data Formats for Block Coding

Each message or codeword is an ordered grouping of symbols. Each block in the Block Coding sublibrary processes one word in each time step, as described in the following section, “Binary Format (All Coding Methods)” on page 16-18. Reed-Solomon coding blocks also let you choose between binary and integer data, as described in “Integer Format (Reed-Solomon Only)” on page 16-19.

Binary Format (All Coding Methods)

You can structure messages and codewords as binary *vector* signals, where each vector represents a message word or a codeword. At a given time, the encoder receives an entire message word, encodes it, and outputs the entire codeword. The message and code signals operate over the same sample time.

This example illustrates the encoder receiving a four-bit message and producing a five-bit codeword at time 0. It repeats this process with a new message at time 1.



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

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

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

Using Serial Signals

If you prefer to structure messages and codewords as scalar signals, where several samples jointly form a message word or codeword, you can use the Buffer and Unbuffer blocks. Buffering involves latency and multirate processing. If your model computes error rates, the initial delay in the coding-buffering combination influences the **Receive delay** parameter in the Error Rate Calculation block.

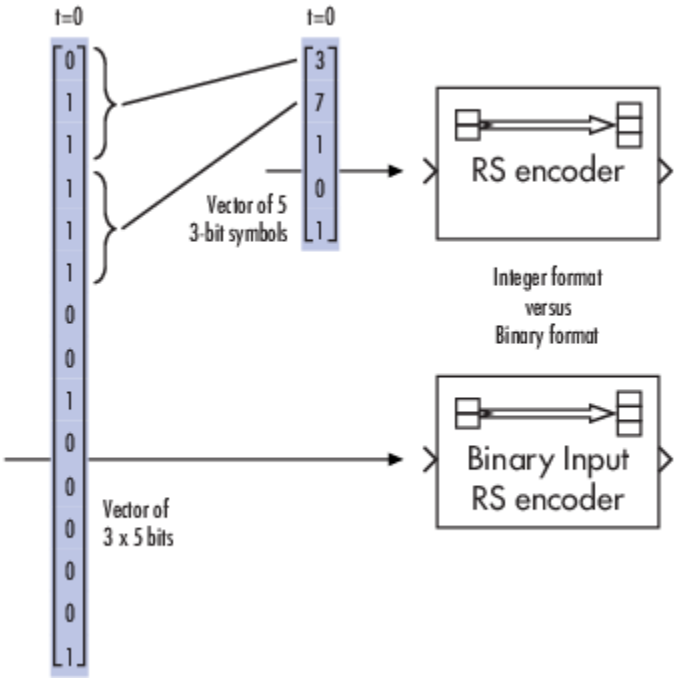
You can display the sample times of signals in your model. On the **Debug** tab, expand **Information Overlays**. In the **Sample Time** section, select **Colors**. Alternatively, you can attach Probe blocks to connector lines to help evaluate sample timing, buffering, and delays.

Integer Format (Reed-Solomon Only)

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

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

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



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

Using Block Encoders and Decoders Within a Model

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

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

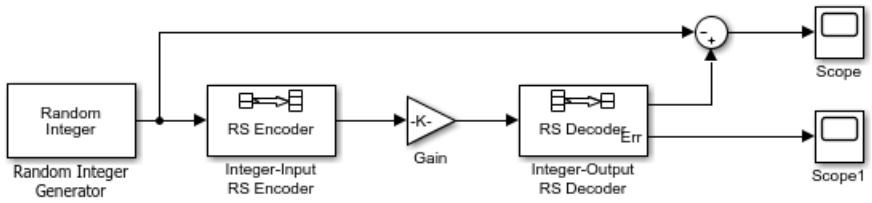
  The Reed-Solomon and BCH blocks have an error counter as a second output.
- Be sure that the signal sizes are appropriate for the mask parameters. For example, if you use the Binary Cyclic Encoder block and set **Message length K** to 4, the input signal must be a vector of length 4.

  You can display the size of signals in your model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**.

Examples of Block Coding

Example: Reed-Solomon Code in Integer Format

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



Open the model by typing `doc_rscoding` at the MATLAB command line. To build the model, gather and configure these blocks:

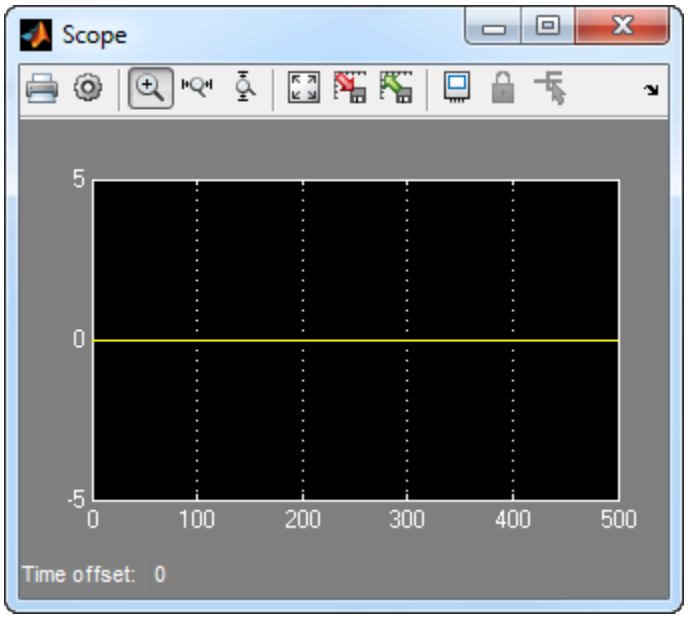
- Random Integer Generator, in the Comm Sources library
  - Set **M-ary number** to `15`.
  - Set **Initial seed** to a positive number, `randn` is chosen here.
  - Check the **Frame-based outputs** check box.
  - Set **Samples per frame** to `5`.
- Integer-Input RS Encoder
  - Set **Codeword length N** to `15`.
  - Set **Message length K** to `5`.
- Gain, in the Simulink Math Operations library
  - Set **Gain** to `[0; 0; 0; 0; 0; ones(10,1)]`.
- Integer-Output RS Decoder
  - Set **Codeword length N** to `15`.
  - Set **Message length K** to `5`.
- Scope, in the Simulink Sinks library. Get two copies of this block.
- Add, in the Simulink Math Operations library
  - Set **List of signs** to `|-+`

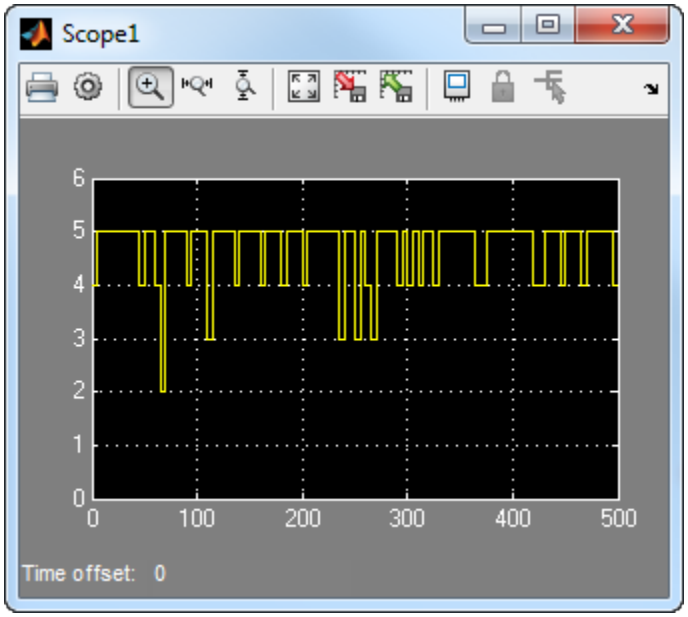
Connect the blocks as shown in the preceding figure. On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `500`. The **Simulate** section appears on multiple tabs.

You can display the vector length of signals in your model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**.

The encoder accepts a vector of length 5 (which is K in this case) and produces a vector of length 15 (which is N in this case). The decoder does the opposite.

Running the model produces the following scope images. The plotted error count will vary based on the **Initial Seed** value used in the Random Integer Generator block. You can adjust the axis range exactly match that of the first scope. Right-click the plot area in the second scope and select **Configuration Properties**. On the **Display** tab, adjust the axes limits.





Number of Errors Before Correction

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

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

Notes on Specific Block-Coding Techniques

Although the Block Coding sublibrary is somewhat uniform in its look and feel, the various coding techniques are not identical. This section describes special options and restrictions that apply to parameters and signals for the coding technique categories in this sublibrary. Coding techniques discussed below include - Generic Linear Block code, Cyclic code, Hamming code, BCH code, and Reed-Solomon code.

Generic Linear Block Codes

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

Generator Matrix - The process of encoding a message into an [N,K] linear block code is determined by a K-by-N generator matrix *G*. Specifically, a 1-by-K message vector *v* is encoded into the 1-by-N codeword vector *vG*. If *G* has the form [I_k, *P*] or [*P*, I_k], where *P* is some K-by-(N-K) matrix and I_k is the K-by-K identity matrix, *G* is said to be in *standard form*. (Some authors, such as Clark and Cain [2], use the first standard form, while others, such as Lin and Costello [3], use the second.) The linear block-coding blocks in this product require the **Generator matrix** mask parameter to be in standard form.

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

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

You can avoid specifying a decoding table explicitly, by setting the **Error-correction truth table** parameter to `0`. When **Error-correction truth table** is `0`, the block computes a decoding table using the `syndtable` function.

Cyclic Codes

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

Generator Polynomials - Cyclic codes have special algebraic properties that allow a polynomial to determine the coding process completely. This so-called generator polynomial is a degree-(N-K) divisor of the polynomial x^N-1. Van Lint [5] explains how a generator polynomial determines a cyclic code.

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

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

Hamming Codes

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

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

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

BCH Codes

BCH codes are cyclic error-correcting codes that are constructed using finite fields. For these codes, the codeword length N must have the form 2^M-1, where M is an integer from 3 to 9. The message length K is restricted to particular values that depend on N. To see which values of K are valid for a given N, see the `comm.BCHEncoder` System object reference page. No known analytic formula describes the relationship among the codeword length, message length, and error-correction capability for BCH codes.

Narrow-Sense BCH Codes

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.

Reed-Solomon Codes

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

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

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

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

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

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

Shortening, Puncturing, and Erasures

Many standards utilize punctured codes, and digital receivers can easily output erasures. BCH and RS performance improves significantly in fading channels where the receiver generates erasures.

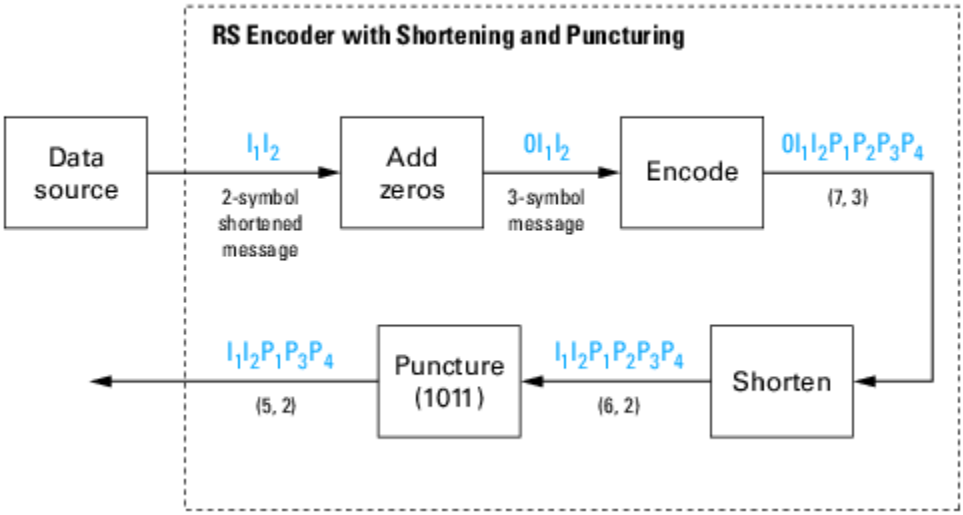
A *punctured codeword* has only parity symbols removed, and a *shortened codeword* has only information symbols removed. A codeword with erasures can have those erasures in either information symbols or parity symbols.

Reed Solomon Examples with Shortening, Puncturing, and Erasures

In this section, a representative example of Reed Solomon coding with shortening, puncturing, and erasures is built with increasing complexity of error correction.

Encoder Example with Shortening and Puncturing

The following figure shows a representative example of a (7,3) Reed Solomon encoder with shortening and puncturing.



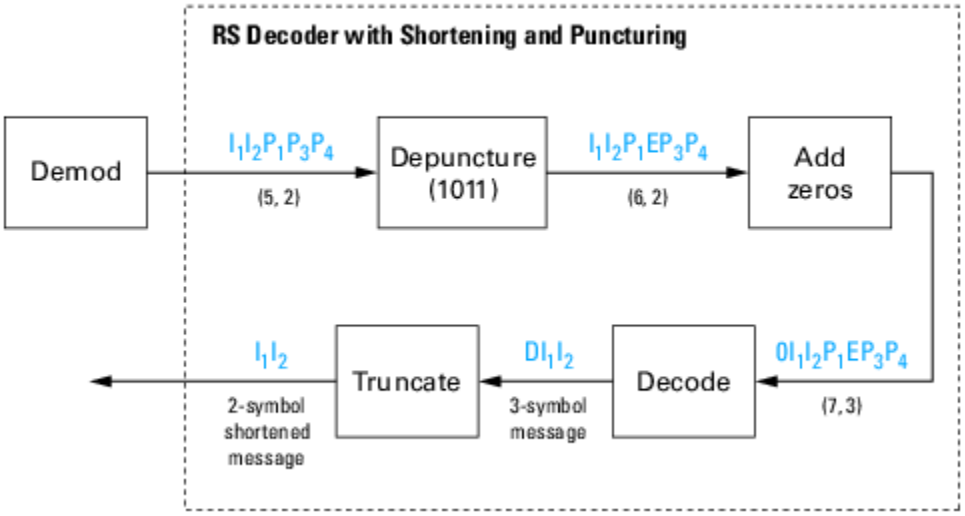
In this figure, the message source outputs two information symbols, designated by I_1I_2. (For a BCH example, the symbols are binary bits.) Because the code is a shortened (7,3) code, a zero must be added ahead of the information symbols, yielding a three-symbol message of $0I_1I_2$. The modified message sequence is RS encoded, and the added information zero is then removed, which yields a result of $I_1I_2P_1P_2P_3P_4$. (In this example, the parity bits are at the end of the codeword.)

The puncturing operation is governed by the puncture vector, which, in this case, is 1011. Within the puncture vector, a `1` means that the symbol is kept, and a `0` means that the symbol is thrown away. In

this example, the puncturing operation removes the second parity symbol, yielding a final vector of $I_1I_2P_1P_3P_4$.

Decoder Example with Shortening and Puncturing

The following figure shows how the RS decoder operates on a shortened and punctured codeword.

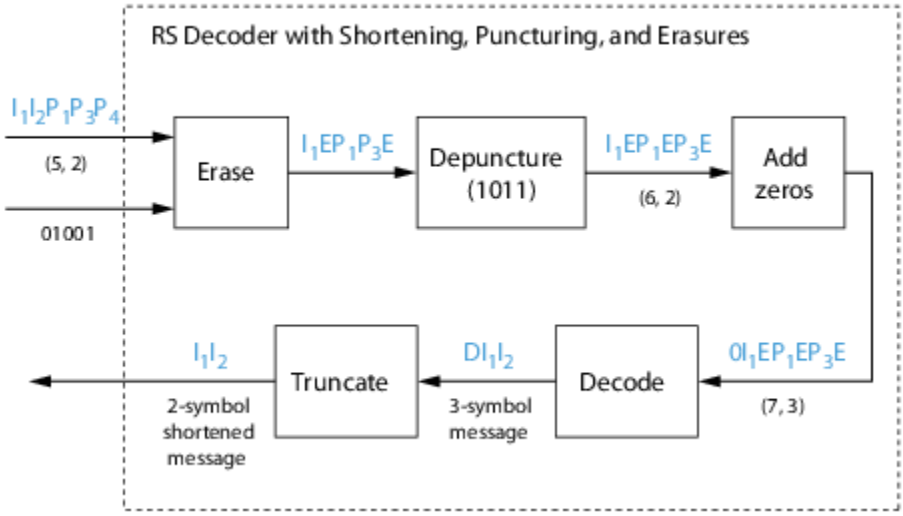


This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a (5,2) codeword, because it has been shortened from a (7,3) codeword by one symbol, and one symbol has also been punctured.

As a first step, the decoder adds an erasure, designated by E, in the second parity position of the codeword. This corresponds to the puncture vector 1011. Adding a zero accounts for shortening, in the same way as shown in the preceding figure. The single erasure does not exceed the erasure-correcting capability of the code, which can correct four erasures. The decoding operation results in the three-symbol message DI_1I_2. The first symbol is truncated, as in the preceding figure, yielding a final output of I_1I_2.

Decoder Example with Shortening, Puncturing, and Erasures

The following figure shows the decoder operating on the punctured, shortened codeword, while also correcting erasures generated by the receiver.



In this figure, demodulator receives the $I_1I_2P_1P_3P_4$ vector that the encoder sent. The demodulator declares that two of the five received symbols are unreliable enough to be erased, such that symbols 2 and 5 are deemed to be erasures. The 01001 vector, provided by an external source, indicates these erasures. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered.

The decoder blocks receive the codeword and the erasure vector, and perform the erasures indicated by the vector 01001. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered. The resulting codeword vector is $I_1EP_1P_3E$, where E is an erasure symbol.

The codeword is then depunctured, according to the puncture vector used in the encoding operation (i.e., 1011). Thus, an erasure symbol is inserted between P_1 and P_3, yielding a codeword vector of $I_1EP_1EP_3E$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0I_1EP_1EP_3E$, such that a (7,3) codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of DI_1I_2 (where D refers to a dummy symbol). Finally, the removal of the D symbol from the message vector accounts for the shortening and yields the original I_1I_2 vector.

For additional information, see the “Reed-Solomon Coding with Erasures, Punctures, and Shortening” MATLAB example or the “Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink” on page 19-3 example.

Reed-Solomon Code in Integer Format

To open an example model that uses a Reed-Solomon code in integer format, type `doc_rscoding` at the MATLAB command line. For more information about the model, see “Example: Reed-Solomon Code in Integer Format” on page 16-20

Find a Generator Polynomial

To find a generator polynomial for a cyclic, BCH, or Reed-Solomon code, use the `cyclpoly`, `bchgenpoly`, or `rsgenpoly` function, respectively. The commands

```
genpolyCyclic = cyclpoly(15,5) % 1+X^5+X^10
genpolyBCH = bchgenpoly(15,5)  % x^10+x^8+x^5+x^4+x^2+x+1
genpolyRS = rsgenpoly(15,5)
```

find generator polynomials for block codes of different types. The output is below.

```
genpolyCyclic =

     1     0     0     0     0     1     0     0     0     0     1


genpolyBCH = GF(2) array.

Array elements =

     1     0     1     0     0     1     1     0     1     1     1


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

Array elements =

     1     4     8    10    12     9     4     2    12     2     7
```

The formats of these outputs vary:

- `cyclpoly` represents a generator polynomial using an integer row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable.
- `bchgenpoly` and `rsgenpoly` represent a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable.
- `rsgenpoly` uses coefficients in a Galois field other than the binary field GF(2). For more information on the meaning of these coefficients, see “How Integers Correspond to Galois Field Elements” on page 16-84 and “Polynomials over Galois Fields” on page 16-98.

Nonuniqueness of Generator Polynomials

Some pairs of message length and codeword length do not uniquely determine the generator polynomial. The syntaxes for functions in the example above also include options for retrieving generator polynomials that satisfy certain constraints that you specify. See the functions' reference pages for details about syntax options.

Algebraic Expression for Generator Polynomials

The generator polynomials produced by `bchgenpoly` and `rsgenpoly` have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where A is a primitive element for an appropriate Galois field, and b and t are integers. See the functions' reference pages for more information about this expression.

Performing Other Block Code Tasks

This section describes functions that compute typical parameters associated with linear block codes, as well as functions that convert information from one format to another.

- **Error Correction Versus Error Detection for Linear Block Codes**

You can use a linear block code to detect $d_{\min}$ -1 errors or to correct $t = \left[\frac{1}{2}(d_{\min} - 1)\right]$ errors.

If you compromise the error correction capability of a code, you can detect more than t errors. For example, a code with $d_{\min} = 7$ can correct $t = 3$ errors or it can detect up to 4 errors and correct up to 2 errors.

- **Finding the Error-Correction Capability**

  The `bchgenpoly` and `rsgenpoly` functions can return an optional second output argument that indicates the error-correction capability of a BCH or Reed-Solomon code. For example, the commands

  ```
  [g,t] = bchgenpoly(31,16);
  t
  t =

       3
  ```

  find that a [31, 16] BCH code can correct up to three errors in each codeword.

- **Finding Generator and Parity-Check Matrices**

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

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

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

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

- **Converting Between Parity-Check and Generator Matrices**

  The `gen2par` function converts a generator matrix into a parity-check matrix, and vice versa. The reference page for `gen2par` contains examples to illustrate this.

Selected Bibliography for Block Coding

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

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

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

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

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

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

[7] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

[8] Ryan, William E., "An introduction to LDPC codes," *Coding and Signal Processing for Magnetic Recoding Systems* (Vasic, B., ed.), CRC Press, 2004.

Convolutional Codes

Convolutional Code Features

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

Communications Toolbox provides convolutional coding capabilities as Simulink blocks, System objects, and MATLAB functions. This product supports feedforward and feedback convolutional codes that can be described by a trellis structure or a set of generator polynomials. It uses the Viterbi algorithm to implement hard-decision and soft-decision decoding.

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

For background information about convolutional coding, see the works listed in Selected Bibliography for Convolutional Coding on page 16-54.

Block Parameters for Convolutional Coding

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

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

- If you design your encoder using a diagram with shift registers and modulo-2 adders, you can compute the code generator polynomial matrix and subsequently use the `poly2trellis` function (in Communications Toolbox) to generate the corresponding trellis structure mask parameter automatically. For an example, see "Design a Rate 2/3 Feedforward Encoder Using Simulink" on page 16-44.

- If you design your encoder using a trellis diagram, you can construct the trellis structure in MATLAB and use it as the mask parameter.

For more information about these representations, see Polynomial Description of a Convolutional Code on page 16-31 and Trellis Description of a Convolutional Code on page 16-33.

Using the Polynomial Description in Blocks

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

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

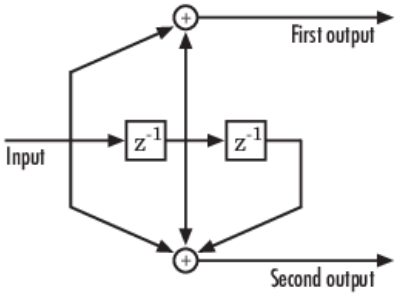
To use this encoder with one of the convolutional-coding blocks, simply place a `poly2trellis` command such as

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

in the **Trellis structure** parameter field.

Polynomial Description of a Convolutional Code

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



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

- Constraint lengths on page 16-31
- Generator polynomials on page 16-32
- Feedback connection polynomials on page 16-32 (for feedback encoders only)

Constraint Lengths

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

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

Generator Polynomials

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

For *systematic* bits of a systematic feedback encoder, match the entry in the code generator matrix with the corresponding element of the feedback connection vector. See “Feedback Connection Polynomials” on page 16-32 below for details.

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

1. Build a binary number representation by placing a 1 in each spot where a connection line from the shift register feeds into the adder, and a 0 elsewhere. The leftmost spot in the binary number represents the current input, while the rightmost spot represents the oldest input that still remains in the shift register.
2. Convert this binary representation into an octal representation by considering consecutive triplets of bits, starting from the rightmost bit. The rightmost bit in each triplet is the least significant. If the number of bits is not a multiple of three, place zero bits at the left end as necessary. (For example, interpret 1101010 as 001 101 010 and convert it to 152.)

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

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

For a table of some good convolutional code generators, refer to [2] in the section “Selected Bibliography for Block Coding” on page 16-29, especially that book's appendices.

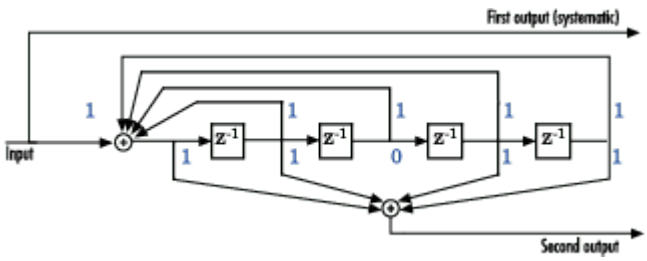
Feedback Connection Polynomials

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

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

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
```

Using the Polynomial Description in MATLAB

To use the polynomial description with the functions `convenc` and `vitdec`, first convert it into a trellis description using the `poly2trellis` function. For example, the command below computes the trellis description of the encoder pictured in the section Polynomial Description of a Convolutional Code on page 16-31.

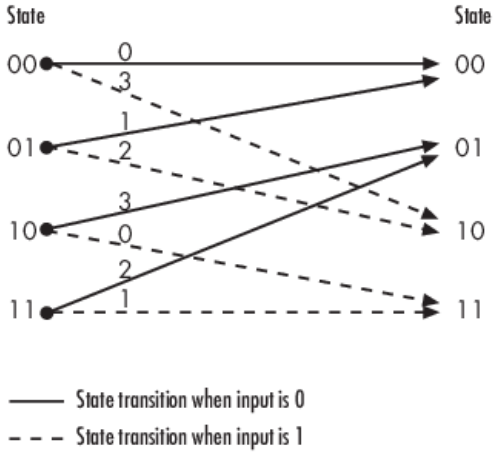
```
trellis = poly2trellis(3,[6 7]);
```

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

Trellis Description of a Convolutional Code

A trellis description of a convolutional encoder shows how each possible input to the encoder influences both the output and the state transitions of the encoder. This section describes trellises, and how to represent on page 16-34 trellises in MATLAB, and gives an example of a MATLAB trellis on page 16-35.

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



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

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

Specifying a Trellis in MATLAB

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

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

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

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

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

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

How to Create a MATLAB Trellis Structure

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

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

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

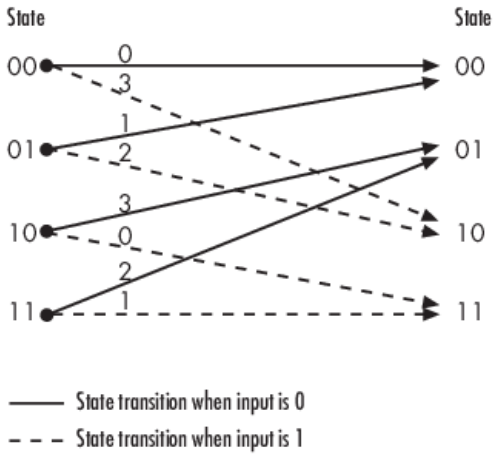
  The reference page for the `istrellis` function illustrates this approach.
- Collect all field names and their values in a single `struct` command. For example:

  ```
  s = struct('numInputSymbols',2,'numOutputSymbols',2,...
     'numStates',2,'nextStates',[0 1;0 1],'outputs',[0 0;1 1]);
  ```
- Start with a polynomial description of the encoder and use the `poly2trellis` function to convert it to a valid trellis structure. For more information , see Polynomial Description of a Convolutional Code on page 16-31.

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

Example: A MATLAB Trellis Structure

Consider the trellis shown below.



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

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

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

Create and Decode Convolutional Codes

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

Encoding

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

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

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

To check whether your trellis corresponds to a catastrophic convolutional code, use the `iscatastrophic` function.

Hard-Decision Decoding

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

```
Define a trellis.

t = poly2trellis([4 3],[4 5 17;7 4 2]);
Encode a vector of ones.

code = convenc(ones(100,1),t);
Set the traceback length for decoding and decode using vitdec.

tb = 2;
decoded = vitdec(code,t,tb,'trunc','hard');
Verify that the decoded data is a vector of 100 ones.

isequal(decoded,ones(100,1))
```

```
ans = logical
   1
```

Soft-Decision Decoding

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

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

Input Values for 3-bit Soft Decisions

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

Implement Soft-Decision Decoding Using MATLAB

The script below illustrates decoding with 3-bit soft decisions. First it creates a convolutional code with `convenc` and adds white Gaussian noise to the code with `awgn`. Then, to prepare for soft-decision decoding, the example uses `quantiz` to map the noisy data values to appropriate decision-value integers between 0 and 7. The second argument in `quantiz` is a partition vector that determines which data values map to 0, 1, 2, etc. The partition is chosen so that values near 0 map to 0, and values near 1 map to 7. (You can refine the partition to obtain better decoding performance if your application requires it.) Finally, the example decodes the code and computes the bit error rate. When comparing the decoded data with the original message, the example must take the decoding delay into account. The continuous operation mode of `vitdec` causes a delay equal to the traceback length, so `msg(1)` corresponds to `decoded(tblen+1)` rather than to `decoded(1)`.

```
s = RandStream.create('mt19937ar', 'seed',94384);
prevStream = RandStream.setGlobalStream(s);
msg = randi([0 1],4000,1); % Random data
t = poly2trellis(7,[171 133]); % Define trellis.
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(t);
% Create an AWGNChannel System object.
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 6);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(t, 'InputFormat', 'Soft', ...
    'SoftInputWordLength', 3, 'TracebackDepth', 48, ...
    'TerminationMethod', 'Continuous');
% Create a ErrorRate Calculator System object. Account for the receive
% delay caused by the traceback length of the viterbi decoder.
hErrorCalc = comm.ErrorRate('ReceiveDelay', 48);
```

```
ber = zeros(3,1); % Store BER values
code = step(hConvEnc,msg); % Encode the data.
hChan.SignalPower = (code'*code)/length(code);
ncode = step(hChan,code); % Add noise.

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

tblen = 48; delay = tblen; % Traceback length
decoded = step(hVitDec,qcode); % Decode.

% Compute bit error rate.
ber = step(hErrorCalc, msg, decoded);
ratio = ber(1)
number = ber(2)
RandStream.setGlobalStream(prevStream);
```

The output is below.

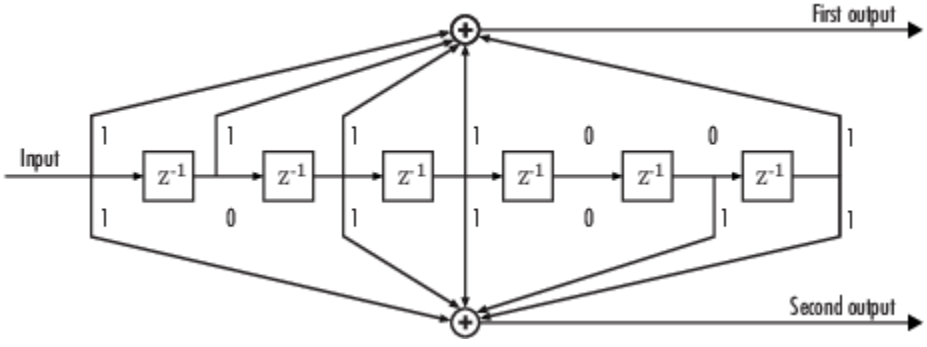
```
number =

     5


ratio =

    0.0013
```

Implement Soft-Decision Decoding Using Simulink

This example creates a rate 1/2 convolutional code. It uses a quantizer and the Viterbi Decoder block to perform soft-decision decoding. To open the model, enter `doc_softdecision` at the MATLAB command line. For a description of the model, see Overview of the Simulation on page 16-48.

Defining the Convolutional Code

The feedforward convolutional encoder in this example is depicted below.



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

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

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

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

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

Mapping the Received Data

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

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

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

Decoding the Convolutional Code

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

Soft-Decision Interpretation of Data

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

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

Traceback and Decoding Delay

The traceback depth 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 Estimate

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).

The **Traceback depth** parameter in the Viterbi Decoder block represents the length of the decoding delay. Some hardware implementations offer options of 48 and 96. This example chooses 48 because that is closer to the estimated target for a rate ½ code with a constraint length of 7.

Delay in Received Data

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

In this case, the Receive delay value is equal to the Traceback depth value (48).

Comparing Simulation Results with Theoretical Results

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

- **Computing Theoretical Bounds for the Bit Error Rate**

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

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

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

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

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

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

  Values for the coefficients c_d and the free distance f are in published articles such as "Convolutional Codes with Optimum Distance Spectrum" [3]. The free distance for this code is $f = 10$.

  The following commands calculate the values of P_b for E_b/N_0 values in the range from 1 to 4, in increments of 0.5:

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

- **Simulating Multiple Times to Collect Bit Error Rates**

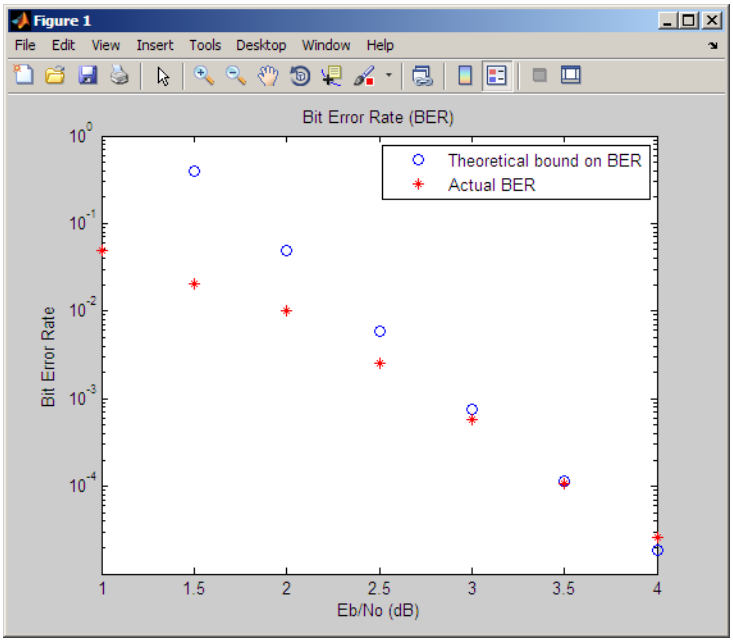
  You can efficiently vary the simulation parameters by using the `sim` function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB, in increments of 0.5 dB. It collects all bit error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

Note To model the model, enter `doc_softdecision` at the MATLAB command line. Then execute these commands, which might take a few minutes.

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

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

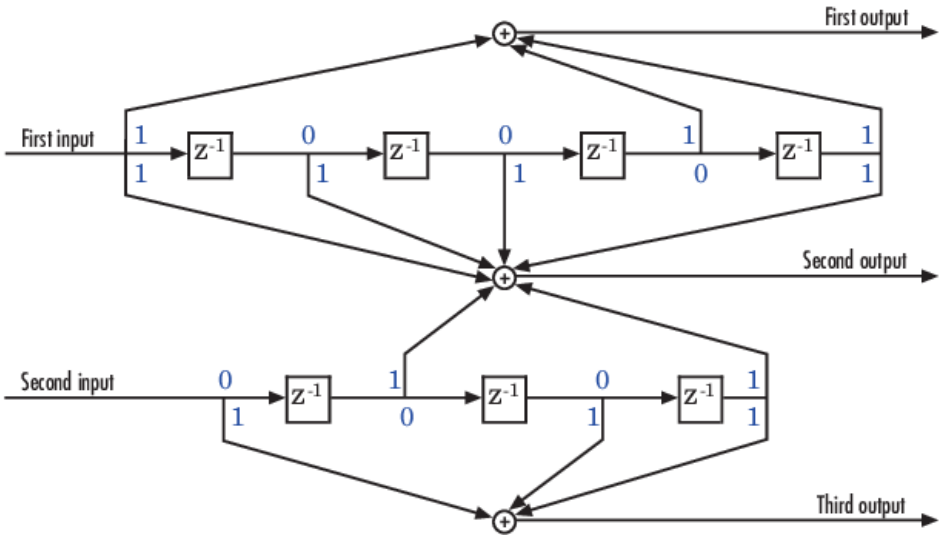
Note The estimate for P_b assumes that the decoder uses unquantized data, that is, an infinitely fine quantization. By contrast, the simulation in this example uses 8-level (3-bit) quantization. Because of this quantization, the simulated bit error rate is not quite as low as the bound when the signal-to-noise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

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



Design a Rate-2/3 Feedforward Encoder Using MATLAB

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



Determining Coding Parameters

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

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

To determine the code generator parameter as a 2-by-3 matrix of octal numbers, use the element in the ith row and jth column to indicate how the ith input contributes to the jth output. For example, to compute the element in the second row and third column, the leftmost and two rightmost elements in the second shift register of the diagram feed into the sum that forms the third output. Capture this information as the binary number 1011, which is equivalent to the octal number 13. The full value of the code generator matrix is [23 35 0; 0 5 13].

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

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

Using the Encoder

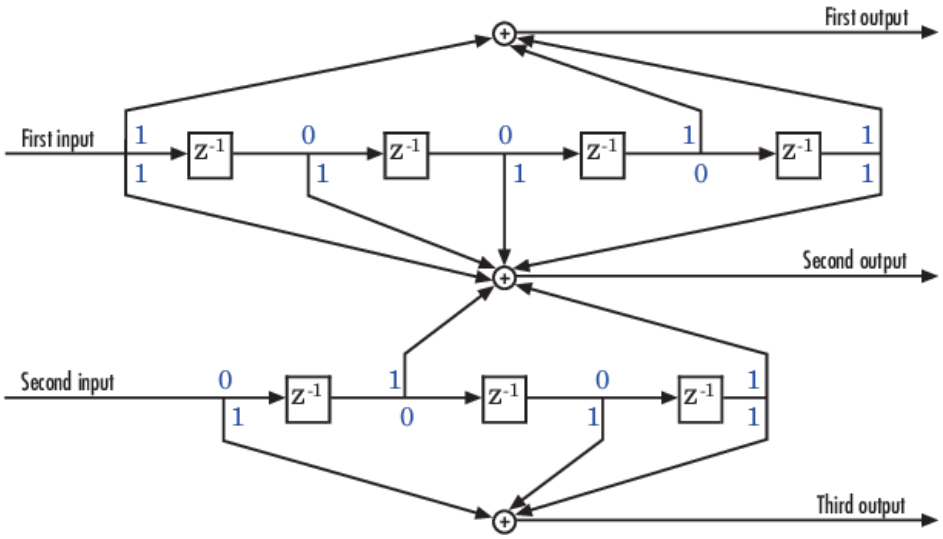
Below is a script that uses this encoder.

```
len = 1000;

msg = randi([0 1],2*len,1); % Random binary message of 2-bit symbols
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Trellis
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(trel);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(trel, 'InputFormat', 'hard', ...
    'TracebackDepth', 34, 'TerminationMethod', 'Continuous');
% Create a ErrorRate Calculator System object. Since each symbol represents
% two bits, the receive delay for this object is twice the traceback length
% of the viterbi decoder.
hErrorCalc = comm.ErrorRate('ReceiveDelay', 68);
ber = zeros(3,1); % Store BER values
code = step(hConvEnc,msg); % Encode the message.
ncode = rem(code + randerr(3*len,1,[0 1;.96 .04]),2); % Add noise.
decoded = step(hVitDec, ncode); % Decode.
ber = step(hErrorCalc, msg, decoded);
```

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

Design a Rate 2/3 Feedforward Encoder Using Simulink

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



How to Determine Coding Parameters

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

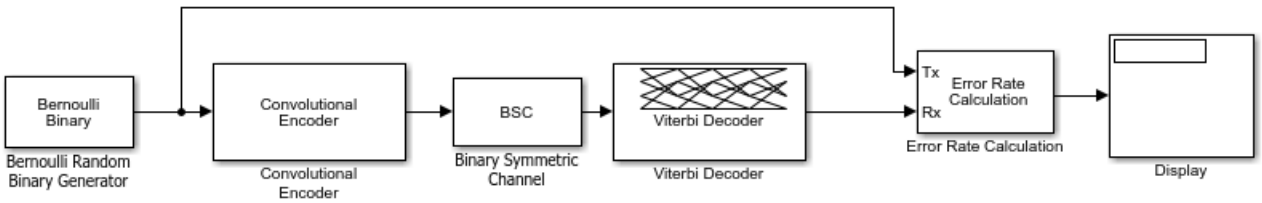
The encoder's constraint length is a vector of length 2 since the encoder has two inputs. The elements of this vector indicate the number of bits stored in each shift register, including the current input bits. Counting memory spaces in each shift register in the diagram and adding one for the current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

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

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

How to Simulate the Encoder

The following model simulates this encoder.



To open the completed model, enter `doc_convcoding` at the MATLAB command line. To build the model, gather and configure these blocks:

- Bernoulli Binary Generator, in the Comm Sources library

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

Connect the blocks as shown in the preceding figure. On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `inf`. The **Simulate** section appears on multiple tabs.

Notes on the Model

You can display the matrix size of signals in your model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**.

The encoder accepts a 2-by-1 column vector and produces a 3-by-1 column vector, while the decoder does the opposite. The **Samples per frame** parameter in the Bernoulli Binary Generator block is 2 because the block must generate a message word of length 2.

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

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

errors occur, because **Target number of errors** is set to `100` in the Error Rate Calculation block. The error rate is much less than `0.02`, the **Error probability** in the Binary Symmetric Channel block.

Puncture a Convolutional Code Using MATLAB

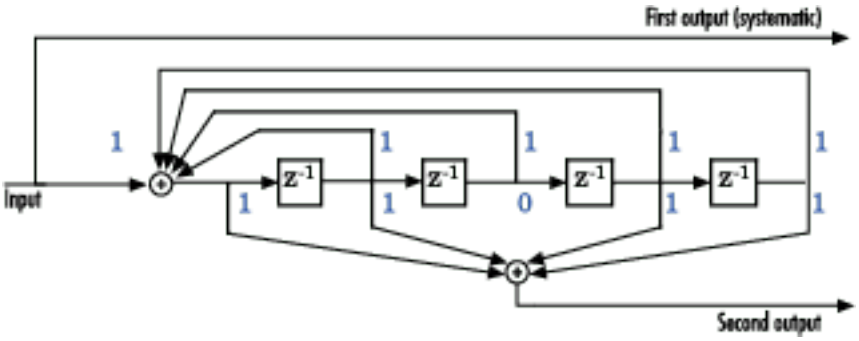
This example processes a punctured convolutional code. It begins by generating 30,000 random bits and encoding them using a rate-3/4 convolutional encoder with a puncture pattern of [1 1 1 0 0 1]. The resulting vector contains 40,000 bits, which are mapped to values of -1 and 1 for transmission. The punctured code, `punctcode`, passes through an additive white Gaussian noise channel. Then `vitdec` decodes the noisy vector using the `'unquant'` decision type.

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

```
len = 30000; msg = randi([0 1], len, 1); % Random data
t = poly2trellis(7, [133 171]); % Define trellis.
% Create a ConvolutionalEncoder System object
hConvEnc = comm.ConvolutionalEncoder(t, ...
    'PuncturePatternSource', 'Property', ...
    'PuncturePattern', [1;1;1;0;0;1]);
% Create an AWGNChannel System object.
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 3);
% Create a ViterbiDecoder System object
hVitDec = comm.ViterbiDecoder(t, 'InputFormat', 'Unquantized', ...
    'TracebackDepth', 96, 'TerminationMethod', 'Truncated', ...
    'PuncturePatternSource', 'Property', ...
    'PuncturePattern', [1;1;1;0;0;1]);
% Create a ErrorRate Calculator System object.
hErrorCalc = comm.ErrorRate;
berP = zeros(3,1); berPE = berP; % Store BER values
punctcode = step(hConvEnc,msg); % Length is (2*len)*2/3.
tcode = 1-2*punctcode; % Map "0" bit to 1 and "1" bit to -1
hChan.SignalPower = (tcode'*tcode)/length(tcode);
ncode = step(hChan,tcode); % Add noise.

% Decode the punctured code
decoded = step(hVitDec,ncode); % Decode.
berP = step(hErrorCalc, msg, decoded);% Bit error rate
% Erase the least reliable 100 symbols, then decode
release(hVitDec); reset(hErrorCalc)
hVitDec.ErasuresInputPort = true;
[dummy idx] = sort(abs(ncode));
erasures =  zeros(size(ncode)); erasures(idx(1:100)) = 1;
decoded = step(hVitDec,ncode, erasures); % Decode.
berPE = step(hErrorCalc, msg, decoded);% Bit error rate

fprintf('Number of errors with puncturing: %d\n', berP(2))
fprintf('Number of errors with puncturing and erasures: %d\n', berPE(2))
```

Implement a Systematic Encoder with Feedback Using Simulink

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



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

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

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

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

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

For more information on setting the mask parameters for the Convolutional Encoder block, see Polynomial Description of a Convolutional Code on page 16-31.

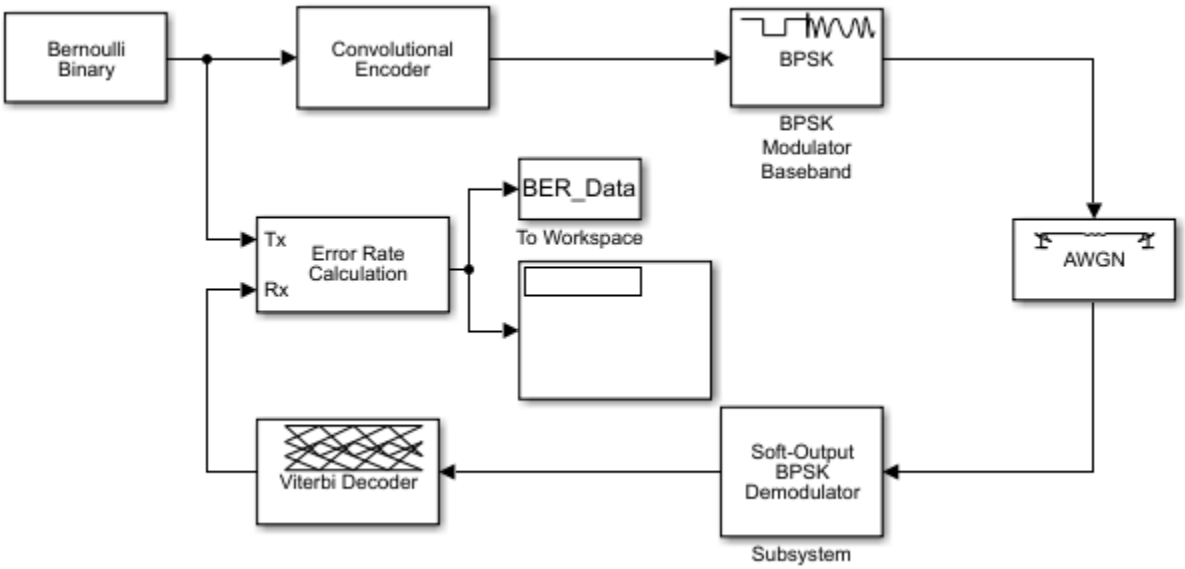
Soft-Decision Decoding

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

Overview of the Simulation

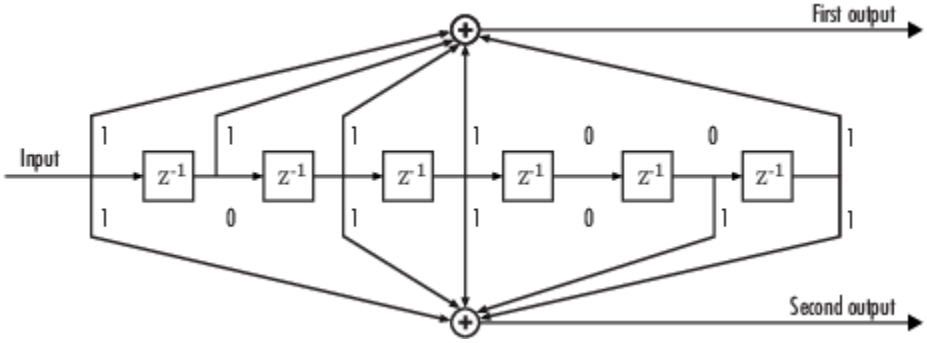
The model is in the following figure. To open the model, enter `doc_softdecision` at the MATLAB command line. The simulation creates a random binary message signal, encodes the message into a convolutional code, modulates the code using the binary phase shift keying (BPSK) technique, and adds white Gaussian noise to the modulated data in order to simulate a noisy channel. Then, the simulation prepares the received data for the decoding block and decodes. Finally, the simulation

compares the decoded information with the original message signal in order to compute the bit error rate. The 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). The simulation ends after processing 100 bit errors or 10^7 message bits, whichever comes first.



Defining the Convolutional Code

The feedforward convolutional encoder in this example is depicted below.



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

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

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

the most significant bit. The second output corresponds to the binary number 1011011, which is equivalent to the octal number 133. The code generator is therefore [171 133].

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

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

Mapping the Received Data

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

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

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

Decoding the Convolutional Code

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

Soft-Decision Interpretation of Data

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

Decision Value	Interpretation
0	Most confident 0
1	Second most confident 0
2	Third most confident 0

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

Traceback and Decoding Delay

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

Delay in Received Data

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

In this case, the Receive delay value is equal to the Traceback depth value (48).

Comparing Simulation Results with Theoretical Results

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

Computing Theoretical Bounds for the Bit Error Rate

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

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

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

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

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

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

Values for the coefficients c_d and the free distance f are in published articles such as "Convolutional Codes with Optimum Distance Spectrum" [3]. The free distance for this code is $f = 10$.

The following commands calculate the values of P_b for E_b/N_0 values in the range from 1 to 4, in increments of 0.5:

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

Simulating Multiple Times to Collect Bit Error Rates

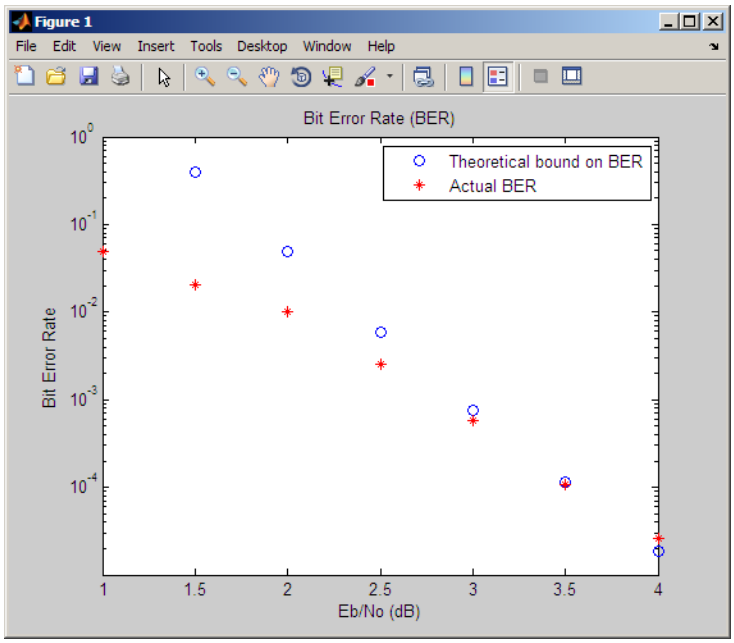
You can efficiently vary the simulation parameters by using the `sim` function to run the simulation from the MATLAB command line. For example, the following code calculates the bit error rate at bit energy-to-noise ratios ranging from 1 dB to 4 dB, in increments of 0.5 dB. It collects all bit error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

Note To open the model, enter `doc_softdecision` at the MATLAB command line. Then execute these commands, which might take a few minutes.

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

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

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

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



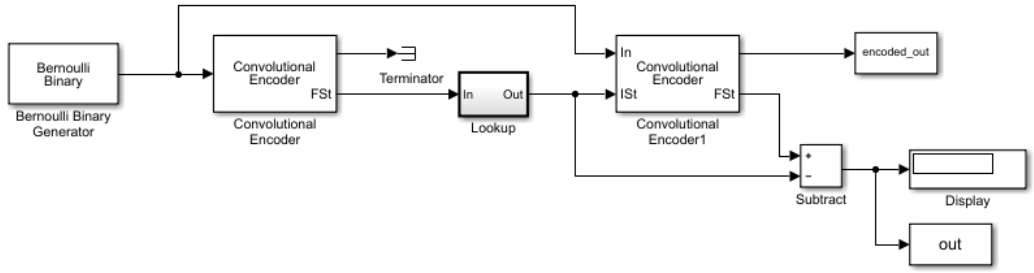
Tailbiting Encoding Using Feedback Encoders

This example demonstrates Tailbiting encoding using feedback encoders. For feedback encoders, the ending state depends on the entire block of data. To accomplish tailbiting, you must calculate for a given information vector (of N bits), the initial state, that leads to the same ending state after the block of data is encoded.

This is achieved in two steps:

- The first step is to determine the zero-state response for a given block of data. The encoder starts in the all-zeros state. The whole block of data is input and the output bits are ignored. After N bits, the encoder is in a state X_N [zs]. From this state, we calculate the corresponding initial state X_0 and initialize the encoder with X_0.
- The second step is the actual encoding. The encoder starts with the initial state X_0, the data block is input and a valid codeword is output which conforms to the same state boundary condition.

Refer to [8] for a theoretical calculation of the initial state X_0 from X_N [zs] using state-space formulation. This is a one-time calculation which depends on the block length and in practice could be implemented as a look-up table. Here we determine this mapping table by simulating all possible entries for a chosen trellis and block length.



To open the model, enter `doc_mtailbiting_wfeedback` at the MATLAB command line.

```
function mapStValues = getMapping(blkLen, trellis)
% The function returns the mapping value for the given block
length and trellis to be used for determining the initial
state from the zero-state response.

% All possible combinations of the mappings
mapStValuesTab = perms(0:trellis.numStates-1);

% Loop over all the combinations of the mapping entries:
for i = 1:length(mapStValuesTab)
mapStValues = mapStValuesTab(i,:);

% Model parameterized for the Block length
sim('mtailbiting_wfeedback');

% Check the boundary condition for each run
% if ending and starting states match, choose that mapping set
if unique(out)==0
        return
    end
end
```

Selecting the returned `mapStValues` for the **Table data** parameter of the `Direct Lookup Table (n-D)` block in the Lookup subsystem will perform tailbiting encoding for the chosen block length and trellis.

Selected Bibliography for Convolutional Coding

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

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

[3] Frenger, P., P. Orten, and T. Ottosson. “Convolutional Codes with Optimum Distance Spectrum.” *IEEE Communications Letters* 3, no. 11 (November 1999): 317–19. https://doi.org/10.1109/4234.803468.

Linear Block Codes

- “Create and Decode Linear Block Codes” on page 16-60

Represent Words for Linear Block Codes

The cyclic, Hamming, and generic linear block code functionality in this product offers you multiple ways to organize bits in messages or codewords. These topics explain the available formats:

- “Use MATLAB to Create Messages and Codewords in Binary Vector Format” on page 16-55
- “Use MATLAB to Create Messages and Codewords in Binary Matrix Format” on page 16-56
- “Use MATLAB to Create Messages and Codewords in Decimal Vector Format” on page 16-56

To learn how to represent words for BCH or Reed-Solomon codes, see “Represent Words for BCH Codes” on page 16-68 or “Represent Words for Reed-Solomon Codes” on page 16-73.

Use MATLAB to Create Messages and Codewords in Binary Vector Format

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

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

The output is below.

```
ans =

  Columns 1 through 5

     1     0     0     1     1

  Columns 6 through 10

     0     1     0     1     0

  Columns 11 through 12

     1     1


ans =

  Columns 1 through 5

     1     1     1     0     0

  Columns 6 through 10

     1     0     0     1     0

  Columns 11 through 15

     1     0     0     1     1
```

```
  Columns 16 through 18

     0            1            1
```

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

Use MATLAB to Create Messages and Codewords in Binary Matrix Format

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

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

The output is below.

```
msg =

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


code =

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

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

Use MATLAB to Create Messages and Codewords in Decimal Vector Format

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

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

Note When you use the decimal vector format, `encode` expects the *leftmost* bit to be the least significant bit.

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

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

The output is below.

```
code =

    39
    20
    54
```

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

Configure Parameters for Linear Block Codes

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

Parameters Used in Block Coding Techniques

Parameter	Block Coding Technique
“Generator Matrix” on page 16-57	Generic linear block
“Parity-Check Matrix” on page 16-57	Generic linear block
“Generator Polynomial” on page 16-59	Cyclic
“Use Decoding Table in MATLAB” on page 16-59	Generic linear block, Hamming

Generator Matrix

The process of encoding a message into an [n,k] linear block code is determined by a k-by-n generator matrix G. Specifically, the 1-by-k message vector v is encoded into the 1-by-n codeword vector vG. If G has the form $[I_k\ P]$ or $[P\ I_k]$, where P is some k-by-(n-k) matrix and I_k is the k-by-k identity matrix, G is said to be in *standard form*. (Some authors, e.g., Clark and Cain [2], use the first standard form, while others, e.g., Lin and Costello [3], use the second.) Most functions in this toolbox assume that a generator matrix is in standard form when you use it as an input argument.

Some examples of generator matrices are in the next section, “Parity-Check Matrix” on page 16-57.

Parity-Check Matrix

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

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

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

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

Examples

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

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

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

genmat =

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

The next example finds parity-check and generator matrices for a [7,3] cyclic code. The `cyclpoly` function is mentioned below in “Generator Polynomial” on page 16-59.

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

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

genmat =

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

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

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

parmat =

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

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

Generator Polynomial

Cyclic codes have algebraic properties that allow a polynomial to determine the coding process completely. This so-called *generator polynomial* is a degree-(n-k) divisor of the polynomial x^n-1. Van Lint [5] explains how a generator polynomial determines a cyclic code.

The `cyclpoly` function produces generator polynomials for cyclic codes. `cyclpoly` represents a generator polynomial using a row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable. For example, the command

```
genpoly = cyclpoly(7,3)

genpoly =

     1     0     1     1     1
```

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

Use Decoding Table in MATLAB

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

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

This example uses a Hamming decoding table to correct an error in a received message. The `hammgen` function produces the parity-check matrix and the `syndtable` function produces the decoding table. To determine the syndrome, the transpose of the parity-check matrix is multiplied on the left by the received codeword. The decoding table helps determine the correction vector. The corrected codeword is the sum (modulo 2) of the correction vector and the received codeword.

Set parameters for a [7,4] Hamming code.

```
m = 3;
n = 2^m-1;
k = n-m;
```

Produce a parity-check matrix and decoding table.

```
parmat = hammgen(m);
trt = syndtable(parmat);
```

Specify a vector of received data.

```
recd = [1 0 0 1 1 1 1]

recd = 1×7

     1     0     0     1     1     1     1
```

Calculate the syndrome, and then display the decimal and binary value for the syndrome.

```
syndrome = rem(recd * parmat',2);
syndrome_int = bit2int(syndrome',m); % Convert to decimal.
```

```
disp(['Syndrome = ',num2str(syndrome_int),...
      ' (decimal), ',num2str(syndrome),' (binary)'])
```

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

Determine the correction vector by using the decoding table and syndrome, and then compute the corrected codeword by using the correction vector.

```
corrvect = trt(1+syndrome_int,:)
```

```
corrvect = 1×7

     0     0     0     0     1     0     0
```

```
correctedcode = rem(corrvect+recd,2)
```

```
correctedcode = 1×7

     1     0     0     1     0     1     1
```

Create and Decode Linear Block Codes

The functions for encoding and decoding cyclic, Hamming, and generic linear block codes are `encode` and `decode`. This section discusses how to use these functions to create and decode generic linear block on page 16-60 codes, cyclic on page 16-61 codes, and Hamming on page 16-62 codes.

Generic Linear Block Codes

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

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

encodes the information in `msg` using the [`n`,`k`] code that the generator matrix `genmat` determines. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See “Represent Words for Linear Block Codes” on page 16-55 or the reference page for `encode` for a description of the formats of `msg` and `code`.

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

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

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

Example: Generic Linear Block Coding

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

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

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

```
err_words =

     1
     2
```

Cyclic Codes

A cyclic code is a linear block code with the property that cyclic shifts of a codeword (expressed as a series of bits) are also codewords. An alternative characterization of cyclic codes is based on its generator polynomial, as mentioned in “Generator Polynomial” on page 16-59 and discussed in [5].

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

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

encodes the information in `msg` using the [`n`,`k`] code determined by the generator polynomial `genpoly`. `genpoly` is an optional argument for `encode`. The default generator polynomial is `cyclpoly(n,k)`. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See “Represent Words for Linear Block Codes” on page 16-55 or the reference page for `encode` for a description of the formats of `msg` and `code`.

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

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

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

Example

You can modify the example in “Generic Linear Block Codes” on page 16-60 so that it uses the cyclic coding technique, instead of the linear block code with the generator matrix `genmat`. Make the changes listed below:

- Replace the second line by

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

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

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

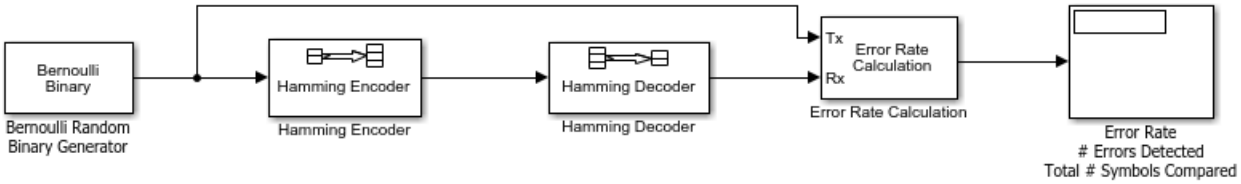
Hamming Codes

The reference pages for `encode` and `decode` contain examples of encoding and decoding Hamming codes. Also, the section “Use Decoding Table in MATLAB” on page 16-59 illustrates error correction in a Hamming code.

Hamming Codes

Create a Hamming Code in Binary Format Using Simulink

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



Open this model by entering `doc_hamming` at the MATLAB command line. To build the model, gather and configure these blocks:

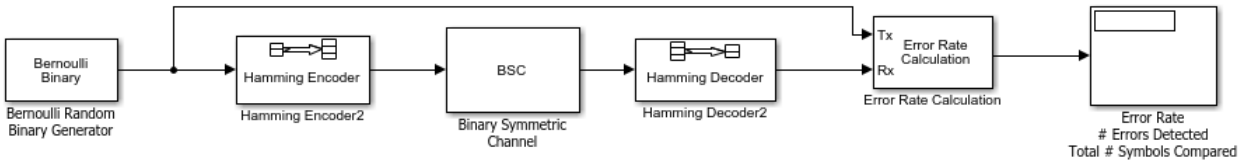
- Bernoulli Binary Generator, in the Comm Sources library
  - Set **Probability of a zero** to `.5`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randn` function.
  - Check the **Frame-based outputs** check box.
  - Set **Samples per frame** to 4.
- Hamming Encoder, with default parameter values
- Hamming Decoder, with default parameter values
- Error Rate Calculation, in the Comm Sinks library, with default parameter values

Connect the blocks as in the preceding figure. You can display the vector length of signals in your model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**. After updating the diagram, if necessary, press **Ctrl+D** to compile the model and check error statistics.

The connector lines show relevant signal attributes. The connector lines are double lines to indicate frame-based signals, and the annotations next to the lines show that the signals are column vectors of appropriate sizes.

Reduce the Error Rate Using a Hamming Code

This section describes how to reduce the error rate by adding an error-correcting code. This figure shows model that uses a Hamming code.

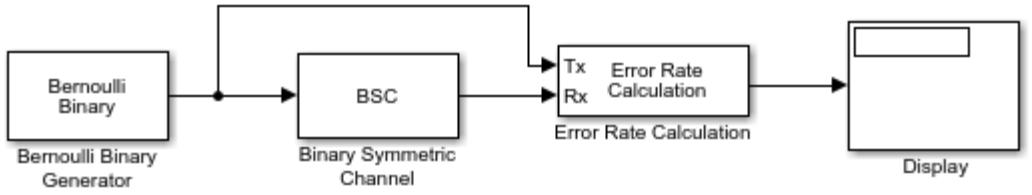


To open a complete version of the model, enter `doc_hamming` at the MATLAB prompt.

Building the Hamming Code Model

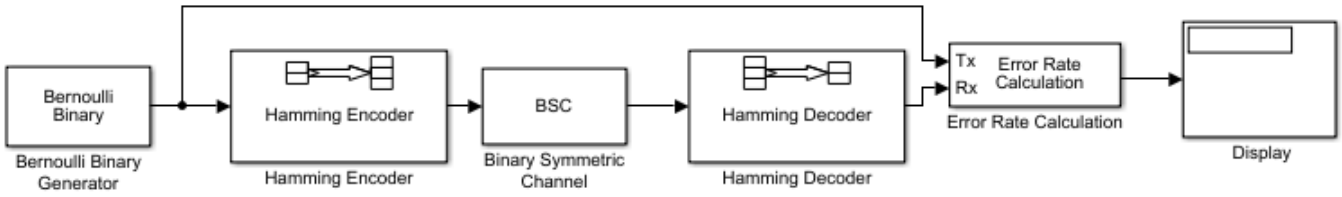
You can build the Hamming code model by following these steps:

1. Type `doc_channel` at the MATLAB command line to open the channel noise model.



   Then save the model as `my_hamming` in the folder where you keep your work files.
2. From the Simulink Library Browser drag the Hamming Encoder and Hamming Decoder blocks from the Error Detection and Correction/Block sublibrary into the model window.
3. Click the right border of the model and drag it to the right to widen the model window.
4. Move the Binary Symmetric Channel, Error Rate Calculation, and Display blocks to the right by clicking and dragging.
5. Create enough space between the Bernoulli Binary Generator and Binary Symmetric Channel blocks to fit the Hamming Encoder between them.
6. Click and drag the Hamming Encoder block on top of the line between the Bernoulli Binary Generator block and the Binary Symmetric Channel block, to the right of the branch point, as shown in the following figure. Then release the mouse button. The Hamming Encoder block should automatically connect to the line from the Bernoulli Binary Generator block to the Binary Symmetric Channel block.
7. Move blocks again to create enough space between the Binary Symmetric Channel and the Error Rate Calculation blocks to fit the Hamming Decoder between them.
8. Click and drag the Hamming Decoder block on top of the line between the Binary Symmetric Channel block and the Error Rate Calculation block.

The model should now resemble this figure.



Using the Hamming Encoder and Decoder Blocks

The Hamming Encoder block encodes the data before it is sent through the channel. The default code is the [7,4] Hamming code, which encodes message words of length 4 into codewords of length 7. As a result, the block converts frames of size 4 into frames of size 7. The code can correct one error in each transmitted codeword.

For an [n,k] code, the input to the Hamming Encoder block must consist of vectors of size k. In this example, k = 4.

The Hamming Decoder block decodes the data after it is sent through the channel. If at most one error is created in a codeword by the channel, the block decodes the word correctly. However, if more than one error occurs, the Hamming Decoder block might decode incorrectly.

To learn more about block coding features, see Block Codes on page 16-17.

Setting Parameters in the Hamming Code Model

Double-click the Bernoulli Binary Generator block and make the following changes to the parameter settings in the block's dialog box, as shown in the following figure:

1. Set **Samples per frame** to 4. This converts the output of the block into frames of size 4, in order to meet the input requirement of the Hamming Encoder Block. See “Sample-Based and Frame-Based Processing” on page 10-4 for more information about frames.

   Note Many blocks, such as the Hamming Encoder block, require their input to be a vector of a specific size. If you connect a source block, such as the Bernoulli Binary Generator block, to one of these blocks, set **Samples per frame** to the required value. For this model the **Samples per frame** parameter of the Bernoulli Binary Generator block must be a multiple of the **Message Length K** parameter of the Hamming Encoder block.

Labeling the Display Block

You can change the label that appears below a block to make it more informative. For example, to change the label below the Display block to `'Error Rate Display'`, first select the label with the mouse. This causes a box to appear around the text. Enter the changes to the text in the box.

Running the Hamming Code Model

To run the model, select **Simulation** > **Run**. The model terminates after 100 errors occur. The error rate, displayed in the top window of the Display block, is approximately .001. You get slightly different results if you change the **Initial seed** parameters in the model or run a simulation for a different length of time.

You expect an error rate of approximately .001 for the following reason: The probability of two or more errors occurring in a codeword of length 7 is

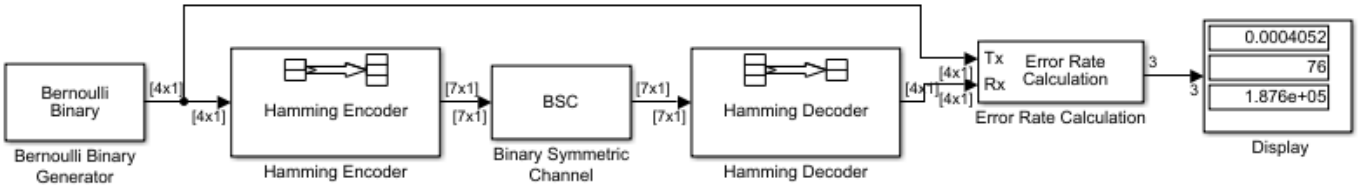
$1 - (0.99)^7 - 7(0.99)^6(0.01) = 0.002$

If the codewords with two or more errors are decoded randomly, you expect about half the bits in the decoded message words to be incorrect. This indicates that .001 is a reasonable value for the bit error rate.

To obtain a lower error rate for the same probability of error, try using a Hamming code with larger parameters. To do this, change the parameters **Codeword length** and **Message length** in the Hamming Encoder and Hamming Decoder block dialog boxes. You also have to make the appropriate changes to the parameters of the Bernoulli Binary Generator block and the Binary Symmetric Channel block.

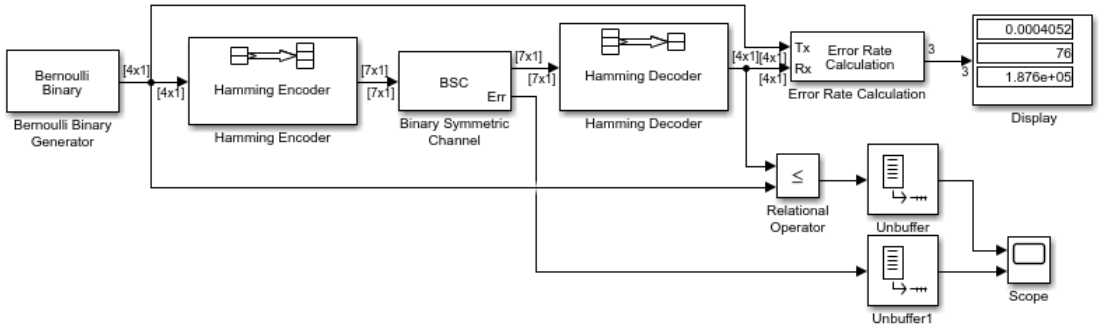
Displaying Frame Sizes

You can display the sizes of data frames in different parts in your model. On the **Debug** tab, expand **Information Overlays**. In the **Signals** section, select **Signal Dimensions**. The line leading out of the Bernoulli Binary Generator block is labeled `[4x1]`, indicating that its output consists of column vectors of size 4. Because the Hamming Encoder block uses a [7,4] code, it converts frames of size 4 into frames of size 7, so its output is labeled `[7x1]`.



Adding a Scope to the Model

To display the channel errors produced by the Binary Symmetric Channel block, add a Scope block to the model. This is a good way to see whether your model is functioning correctly. The example shown in the following figure shows where to insert the Scope block into the model.



To build this model from the one shown in the figure “Reduce the Error Rate Using a Hamming Code” on page 16-63, follow these steps:

1 Drag the following blocks from the Simulink Library Browser into the model window:

- Relational Operator block, from the Simulink Logic and Bit Operations library
- Scope block, from the Simulink Sinks library

- Two copies of the Unbuffer block, from the Buffers sublibrary of the Signal Management library in DSP System Toolbox

2 Double-click the Binary Symmetric Channel block to open its dialog box, and select **Output error vector**. This creates a second output port for the block, which carries the error vector.

3 Double-click the Scope block, under **View > Configuration Properties**, set **Number of input ports** to 2. Select **Layout** and highlight two blocks vertically. Click **OK**.

4 Connect the blocks as shown in the preceding figure.

Setting Parameters in the Expanded Model

Make the following changes to the parameters for the blocks you added to the model.

- **Error Rate Calculation Block** – Double-click the Error Rate Calculation block and clear the box next to **Stop simulation** in the block's dialog box.
- **Scope Block** – The Scope block displays the channel errors and uncorrected errors. To configure the block,

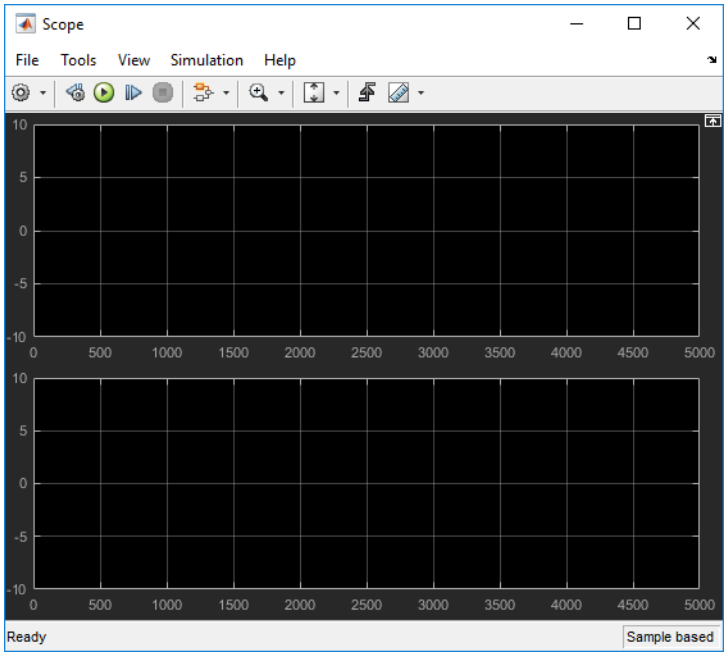
  1 Double-click the Scope block, select **View > Configuration Properties**.

  2 Select the **Time** tab and set **Time span** to `5000`.

  3 Select the **Logging** tab and set **Limit data points to last** to `30000`.

  4 Click **OK**.

  5 The scope should now appear as shown.



  6 To configure the axes, follow these steps:

    a Right-click the vertical axis at the left side of the upper scope.

    b In the context menu, select **Configuration Properties**.

    c Set **Y-limits (Minimum)** to `-1`.

    d Set **Y-limits (Maximum)** to 2, and click **OK**.

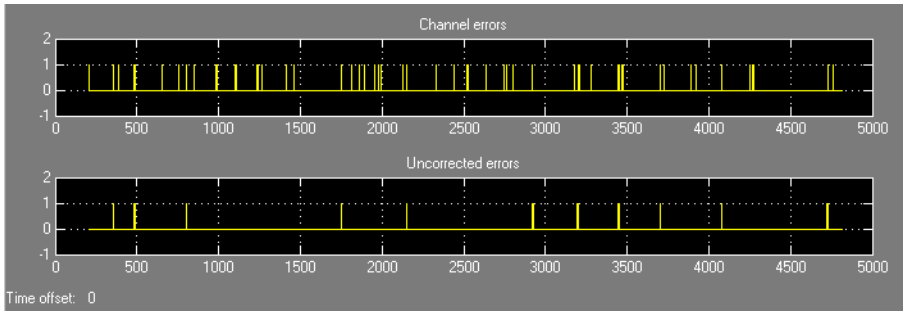
e Repeat the same steps for the vertical axis of the lower scope.

f Widen the scope window until it is roughly three times as wide as it is high. You can do this by clicking the right border of the window and dragging the border to the right, while pressing the left-mouse button.

- **Relational Operator** – Set **Relational Operator** to ~= in the block's dialog box. The Relational Operator block compares the transmitted signal, coming from the Bernoulli Random Generator block, with the received signal, coming from the Hamming Decoder block. The block outputs a 0 when the two signals agree and a 1 when they disagree.

Observing Channel Errors with the Scope

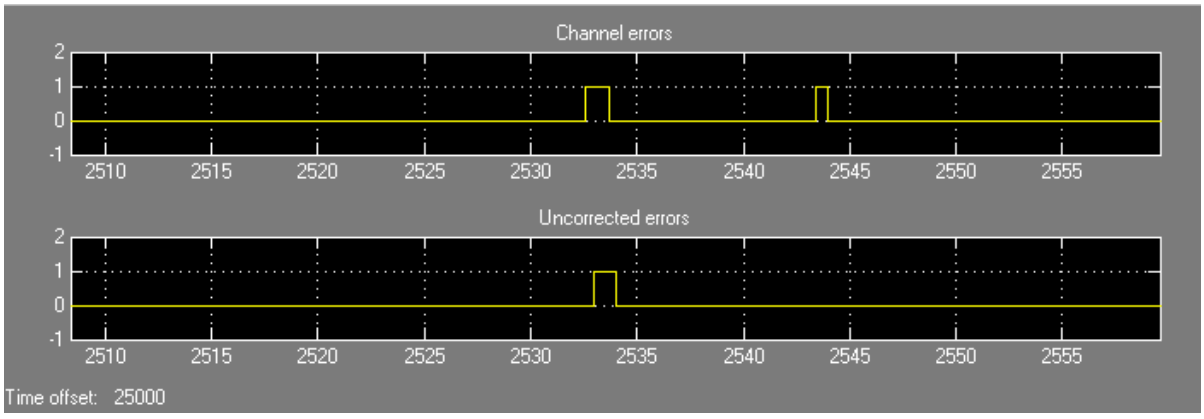
When you run the model, the scope displays the error data. At the end of each 5000 time steps, the scope appears as shown this figure. The scope then clears the displayed data and displays the next 5000 data points.



The upper scope shows the channel errors generated by the Binary Symmetric Channel block. The lower scope shows errors that are not corrected by channel coding.

Click the **Stop** button on the toolbar at the top of the model window to stop the scope.

You can see individual errors by zooming in on the scope. First click the middle magnifying glass button at the top left of the Scope window. Then click one of the lines in the lower scope. This zooms in horizontally on the line. Continue clicking the lines in the lower scope until the horizontal scale is fine enough to detect individual errors. A typical example of what you might see is shown in the figure below.



The wider rectangular pulse in the middle of the upper scope represents two 1s. These two errors, which occur in a single codeword, are not corrected. This accounts for the uncorrected errors in the

lower scope. The narrower rectangular pulse to the right of the upper scope represents a single error, which is corrected.

When you are done observing the errors, select **Simulation > Stop**.

“Export Data to MATLAB” on page 9-3 explains how to send the error data to the MATLAB workspace for more detailed analysis.

BCH Codes

Represent Words for BCH Codes

A message for an [n,k] BCH code must be a k-column binary Galois field array. The code that corresponds to that message is an n-column binary Galois field array. Each row of these Galois field arrays represents one word.

The example below illustrates how to represent words for a [15, 11] BCH code.

```
msg = [1 0 0 1 0; 1 0 1 1 1]; % Messages in a Galois array
obj = comm.BCHEncoder;
c1 = step(obj, msg(1,:)');
c2 = step(obj, msg(2,:)');
cbch = [c1 c2].'
```

The output is

```
  Columns 1 through 5

      1            0            0            1            0
      1            0            1            1            1

  Columns 6 through 10

      0            0            1            1            1
      0            0            0            0            1

  Columns 11 through 15

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

Parameters for BCH Codes

BCH codes use special values of n and k:

- n, the codeword length, is an integer of the form 2^m-1 for some integer $m > 2$.
- k, the message length, is a positive integer less than n. However, only some positive integers less than n are valid choices for k. See the BCH Encoder block reference page for a list of some valid values of k corresponding to values of n up to 511.

Create and Decode BCH Codes

The `BCH Encoder` and `BCH Decoder` System objects create and decode BCH codes, using the data described in “Represent Words for BCH Codes” on page 16-68 and “Parameters for BCH Codes” on page 16-68.

The topics are

Example: BCH Coding Syntaxes

The example below illustrates how to encode and decode data using a [15, 5] BCH code.

```
n = 15; k = 5; % Codeword length and message length
msg = randi([0 1],4*k,1); % Four random binary messages

% Simplest syntax for encoding
enc = comm.BCHEncoder(n,k);
dec = comm.BCHDecoder(n,k);
c1 = step(enc,msg); % BCH encoding
d1 = step(dec,c1); % BCH decoding

% Check that the decoding worked correctly.
chk = isequal(d1,msg)

% The following code shows how to perform the encoding and decoding
% operations if one chooses to prepend the parity symbols.

% Steps for converting encoded data with appended parity symbols
% to encoded data with prepended parity symbols
c11 = reshape(c1, n, []);
c12 = circshift(c11,n-k);
c1_prepend = c12(:); % BCH encoded data with prepended parity symbols

% Steps for converting encoded data with prepended parity symbols
% to encoded data with appended parity symbols prior to decoding
c21 = reshape(c1_prepend, n, []);
c22 = circshift(c21,k);
c1_append = c22(:); % BCH encoded data with appended parity symbols

% Check that the prepend-to-append conversion worked correctly.
d1_append = step(dec,c1_append);
chk = isequal(msg,d1_append)
```

The output is below.

```
chk =

     1
```

Detect and Correct Errors in a BCH Code Using MATLAB

The following example illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and attempts to decode the noisy code using the `BCH Decoder` System object.

```
n = 15; k = 5; % Codeword length and message length
[gp,t] = bchgenpoly(n,k); % t is error-correction capability.
nw = 4; % Number of words to process
msgw = randi([0 1], nw*k, 1); % Random k-symbol messages
enc = comm.BCHEncoder(n,k,gp);
dec = comm.BCHDecoder(n,k,gp);
c = step(enc, msgw); % Encode the data.
noise = randerr(nw,n,t); % t errors per codeword
noisy = noise';
noisy = noisy(:);
cnoisy = mod(c + noisy,2); % Add noise to the code.
[dc, nerrs] = step(dec, cnoisy); % Decode cnoisy.

% Check that the decoding worked correctly.
chk2 = isequal(dc,msgw)
nerrs % Find out how many errors have been corrected.
```

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

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

```
chk2 =

     1


nerrs =

     3
     3
     3
     3
```

Excessive Noise in BCH Codewords

In the previous example, the `BCH Decoder` System object corrected all the errors. However, each BCH code has a finite error-correction capability. To learn more about how the `BCH Decoder` System object behaves when the noise is excessive, see the analogous discussion for Reed-Solomon codes in “Excessive Noise in Reed-Solomon Codewords” on page 16-76.

Algorithms for BCH and RS Errors-only Decoding

Overview

The errors-only decoding algorithm used for BCH and RS codes can be described by the following steps (sections 5.3.2, 5.4, and 5.6 in [2]).

1. Calculate the first $2t$ terms of the infinite degree syndrome polynomial, $S(z)$.
2. If those $2t$ terms of $S(z)$ are all equal to 0, then the code has no errors , no correction needs to be performed, and the decoding algorithm ends.
3. If one or more terms of $S(z)$ are nonzero, calculate the error locator polynomial, $\Lambda(z)$, via the Berlekamp algorithm.
4. Calculate the error evaluator polynomial, $\Omega(z)$, via

$$\Lambda(z)S(z) = \Omega(z) \mathrm{mod} z^{2t}$$

5 Correct an error in the codeword according to

$$e_{i_m} = \frac{\Omega(\alpha^{-i_m})}{\Lambda'(\alpha^{-i_m})}$$

where e_{i_m} is the error magnitude in the i_mth position in the codeword, m is a value less than the error-correcting capability of the code, $\Omega(z)$ is the error magnitude polynomial, $\Lambda'(z)$ is the formal derivative [5] of the error locator polynomial, $\Lambda(z)$, and α is the primitive element of the Galois field of the code.

Further description of several of the steps is given in the following sections.

Syndrome Calculation

For narrow-sense codes, the $2t$ terms of $S(z)$ are calculated by evaluating the received codeword at successive powers of α (the field's primitive element) from 0 to $2t$-1. In other words, if we assume one-based indexing of codewords $C(z)$ and the syndrome polynomial $S(z)$, and that codewords are of the form $[c_1\ c_1\ \ldots\ c_N]$, then each term S_i of $S(z)$ is given as

$$S_i = \sum_{i=1}^{N} c_i \alpha^{N-1-i}$$

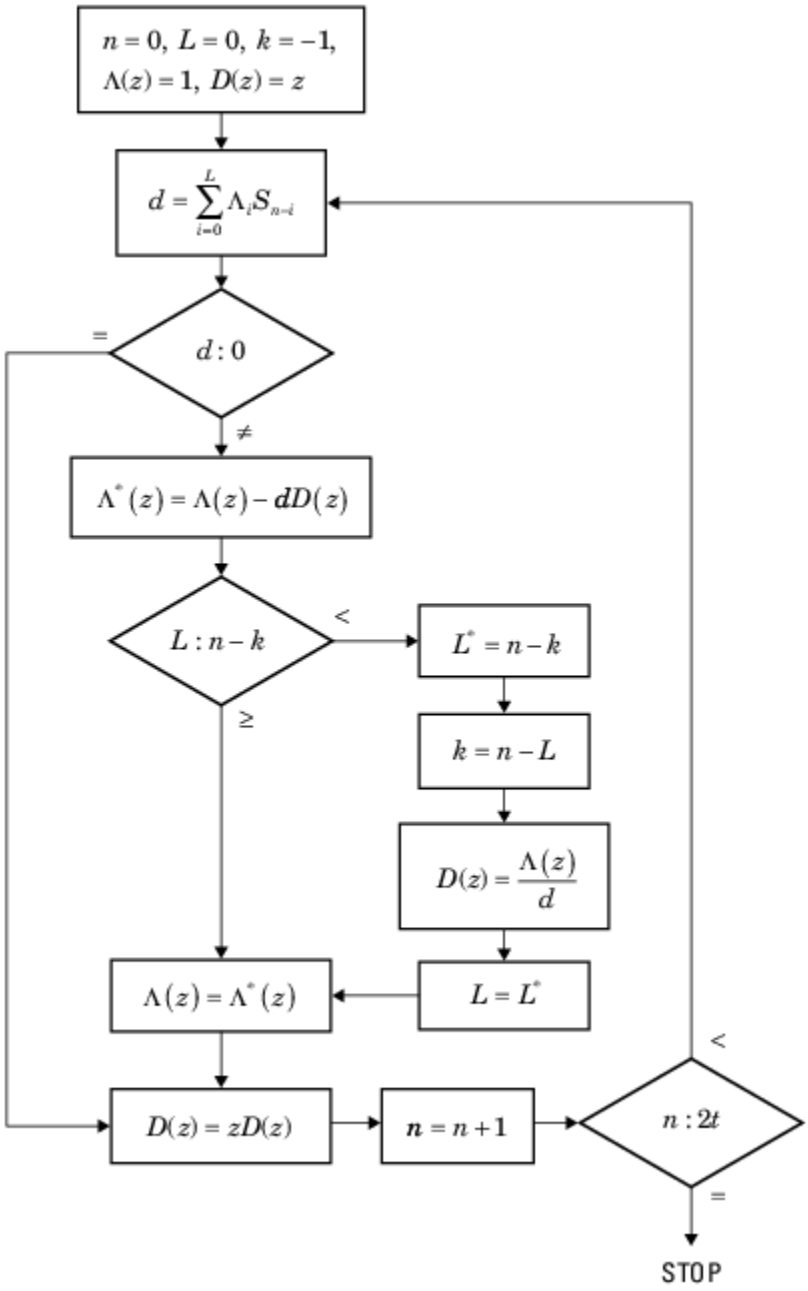
Error Locator Polynomial Calculation

The error locator polynomial, $\Lambda(z)$, is found using the Berlekamp algorithm. A complete description of this algorithm is found in [2], but we summarize the algorithm as follows.

We define the following variables.

Variable	Description
n	Iterator variable
k	Iterator variable
L	Length of the feedback register used to generate the first $2t$ terms of $S(z)$
$D(z)$	Correction polynomial
d	Discrepancy

The following diagram shows the iterative procedure (i.e., the Berlekamp algorithm) used to find $\Lambda(z)$.



Error Evaluator Polynomial Calculation

The error evaluator polynomial, $\Omega(z)$, is simply the convolution of $\Lambda(z)$ and $S(z)$.

Reed-Solomon Codes

Represent Words for Reed-Solomon Codes

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

Note For information about Galois field arrays and how to create them, see “Representing Elements of Galois Fields” on page 16-82 or the reference page for the `gf` function.

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

```
n = 7; k = 3; % Codeword length and message length
m = 3; % Number of bits in each symbol
msg = [1 6 4; 0 4 3]; % Message is a Galois array.
obj = comm.RSEncoder(n, k);
c1 = step(obj, msg(1,:)');
c2 = step(obj, msg(2,:)');
c = [c1 c2].'
```

The output is

```
C =

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

Parameters for Reed-Solomon Codes

This section describes several integers related to Reed-Solomon codes and discusses how to find generator polynomials on page 16-73.

Allowable Values of Integer Parameters

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

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

Generator Polynomial

The `rsgenpoly` function produces generator polynomials for Reed-Solomon codes. `rsgenpoly` represents a generator polynomial using a Galois row vector that lists the polynomial's coefficients in

order of *descending* powers of the variable. If each symbol has m bits, the Galois row vector is in the field GF(2^m). For example, the command

```
r = rsgenpoly(15,13)

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

Array elements =

     1     6     8
```

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

Algebraic Expression for Generator Polynomials

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

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

Create and Decode Reed-Solomon Codes

The `RS Encoder` and `RS Decoder` System objects create and decode Reed-Solomon codes, using the data described in “Represent Words for Reed-Solomon Codes” on page 16-73 and “Parameters for Reed-Solomon Codes” on page 16-73.

This section illustrates how to use the `RS Encoder` and `RS Decoder` System objects. The topics are

Reed-Solomon Coding Syntaxes in MATLAB

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

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

This example also shows that corresponding syntaxes of the `RS Encoder` and `RS Decoder` System objects use the same input arguments, except for the first input argument.

```
m = 4; % Number of bits in each symbol
n = 2^m-1; k = 13; % Codeword length and message length
msg = randi([0 m-1],4*k,1); % Four random integer messages

% Simplest syntax for encoding
hEnc = comm.RSEncoder(n,k);
hDec = comm.RSDecoder(n,k);
c1 = step(hEnc, msg);
d1 = step(hDec, c1);


% Vary the generator polynomial for the code.
release(hEnc), release(hDec)
hEnc.GeneratorPolynomialSource = 'Property';
hDec.GeneratorPolynomialSource = 'Property';
hEnc.GeneratorPolynomial       = rsgenpoly(n,k,19,2);
hDec.GeneratorPolynomial       = rsgenpoly(n,k,19,2);
c2 = step(hEnc, msg);
d2 = step(hDec, c2);

% Vary the primitive polynomial for GF(16).
release(hEnc), release(hDec)
hEnc.PrimitivePolynomialSource = 'Property';
hDec.PrimitivePolynomialSource = 'Property';
hEnc.GeneratorPolynomialSource = 'Auto';
hDec.GeneratorPolynomialSource = 'Auto';
hEnc.PrimitivePolynomial       = [1 1 0 0 1];
hDec.PrimitivePolynomial       = [1 1 0 0 1];
c3 = step(hEnc, msg);
d3 = step(hDec, c3);

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

% The following code shows how to perform the encoding and decoding
% operations if one chooses to prepend the parity symbols.

% Steps for converting encoded data with appended parity symbols
% to encoded data with prepended parity symbols
c31 = reshape(c3, n, []);
c32 = circshift(c31,n-k);
c3_prepend = c32(:); % RS encoded data with prepended parity symbols

% Steps for converting encoded data with prepended parity symbols
% to encoded data with appended parity symbols prior to decoding
c34 = reshape(c3_prepend, n, []);
c35 = circshift(c34,k);
c3_append = c35(:); % RS encoded data with appended parity symbols

% Check that the prepend-to-append conversion worked correctly.
d3_append = step(hDec,c3_append);
chk = isequal(msg,d3_append)
```

The output is

```
chk =

     1
```

Detect and Correct Errors in a Reed-Solomon Code Using MATLAB

The example below illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and attempts to decode the noisy code using the RS `Decoder` System object.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Codeword length and message length
t = (n-k)/2; % Error-correction capability of the code
nw = 4; % Number of words to process
msgw = randi([0 n],nw*k,1); % Random k-symbol messages
hEnc = comm.RSEncoder(n,k);
hDec = comm.RSDecoder(n,k);
c = step(hEnc, msgw); % Encode the data.
noise = (1+randi([0 n-1],nw,n)).*randerr(nw,n,t); % t errors per codeword
noisy = noise';
noisy = noisy(:);
cnoisy = gf(c,m) + noisy; % Add noise to the code under gf(m) arithmetic.
[dc nerrs] = step(hDec, cnoisy.x); % Decode the noisy code.
% Check that the decoding worked correctly.
isequal(dc,msgw)
nerrs % Find out how many errors hDec corrected.
```

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

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

```
ans =

     1

nerrs =

     2
     2
     2
     2
```

Excessive Noise in Reed-Solomon Codewords

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

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

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

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

```
noise = (1+randi([0 n-1],nw,n)).*randerr(nw,n,t+1); % t+1 errors/row
```

Create Shortened Reed-Solomon Codes

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

- $n_1 = 2^m - 1$, where m is the number of bits per symbol
- $k_1 = k + (n_1 - n)$

The `RS Encoder` System object supports shortened codes using the same syntaxes it uses for nonshortened codes. You do not need to indicate explicitly that you want to use a shortened code.

```
hEnc = comm.RSEncoder(7,5);
ordinarycode = step(hEnc,[1 1 1 1 1]');
hEnc = comm.RSEncoder(5,3);
shortenedcode = step(hEnc,[1 1 1 ]');
```

How the `RS Encoder` System Object Creates a Shortened Code

When creating a shortened code, the `RS Encoder` System object performs these steps:

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

The following example illustrates this process.

```
n = 12; k = 8; % Lengths for the shortened code
m = ceil(log2(n+1)); % Number of bits per symbol
msg = randi([0 2^m-1],3*k,1); % Random array of 3 k-symbol words
hEnc = comm.RSEncoder(n,k);
code = step(hEnc, msg); % Create a shortened code.

% Do the shortening manually, just to show how it works.
n_pad = 2^m-1; % Codeword length in the actual encoder
k_pad = k+(n_pad-n); % Messageword length in the actual encoder
hEnc = comm.RSEncoder(n_pad,k_pad);
mw = reshape(msg,k,[]); % Each column vector represents a messageword
msg_pad = [zeros(n_pad-n,3); mw]; % Prepend zeros to each word.
msg_pad = msg_pad(:);
code_pad = step(hEnc,msg_pad); % Encode padded words.
cw = reshape(code_pad,2^m-1,[]); % Each column vector represents a codeword
code_eqv = cw(n_pad-n+1:n_pad,:); % Remove extra zeros.
code_eqv = code_eqv(:);
ck = isequal(code_eqv,code); % Returns true (1).
```

Find a Generator Polynomial

To find a generator polynomial for a cyclic, BCH, or Reed-Solomon code, use the `cyclpoly`, `bchgenpoly`, or `rsgenpoly` function, respectively. The commands

```
genpolyCyclic = cyclpoly(15,5) % 1+X^5+X^10
genpolyBCH = bchgenpoly(15,5)  % x^10+x^8+x^5+x^4+x^2+x+1
genpolyRS = rsgenpoly(15,5)
```

find generator polynomials for block codes of different types. The output is below.

```
genpolyCyclic =

     1     0     0     0     0     1     0     0     0     0     1


genpolyBCH = GF(2) array.

Array elements =

     1     0     1     0     0     1     1     0     1     1     1


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

Array elements =

     1     4     8    10    12     9     4     2    12     2     7
```

The formats of these outputs vary:

- `cyclpoly` represents a generator polynomial using an integer row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable.
- `bchgenpoly` and `rsgenpoly` represent a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable.
- `rsgenpoly` uses coefficients in a Galois field other than the binary field GF(2). For more information on the meaning of these coefficients, see “How Integers Correspond to Galois Field Elements” on page 16-84 and “Polynomials over Galois Fields” on page 16-98.

Nonuniqueness of Generator Polynomials

Some pairs of message length and codeword length do not uniquely determine the generator polynomial. The syntaxes for functions in the example above also include options for retrieving generator polynomials that satisfy certain constraints that you specify. See the functions' reference pages for details about syntax options.

Algebraic Expression for Generator Polynomials

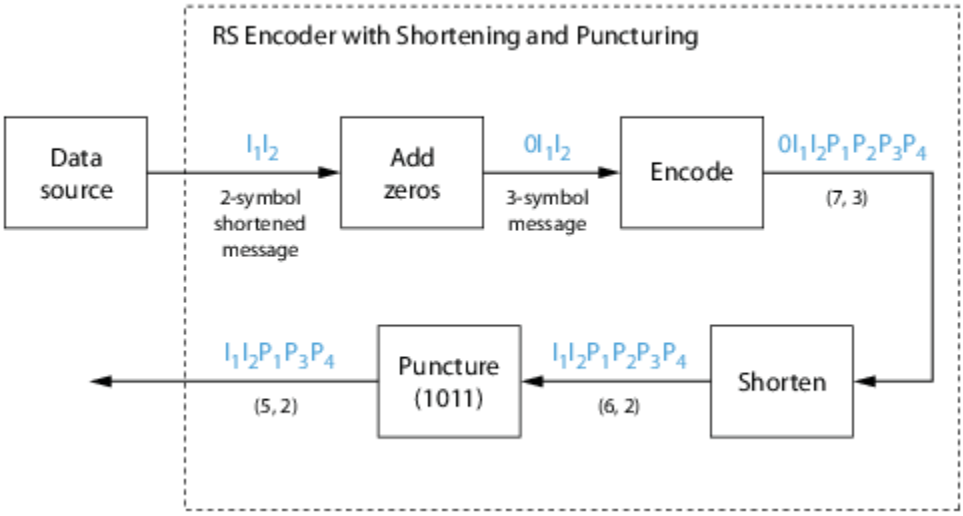
The generator polynomials produced by `bchgenpoly` and `rsgenpoly` have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where A is a primitive element for an appropriate Galois field, and b and t are integers. See the functions' reference pages for more information about this expression.

Reed Solomon Examples with Shortening, Puncturing, and Erasures

In this section, a representative example of Reed Solomon coding with shortening, puncturing, and erasures is built with increasing complexity of error correction.

Encoder Example with Shortening and Puncturing

The following figure shows a representative example of a (7,3) Reed Solomon encoder with shortening and puncturing.

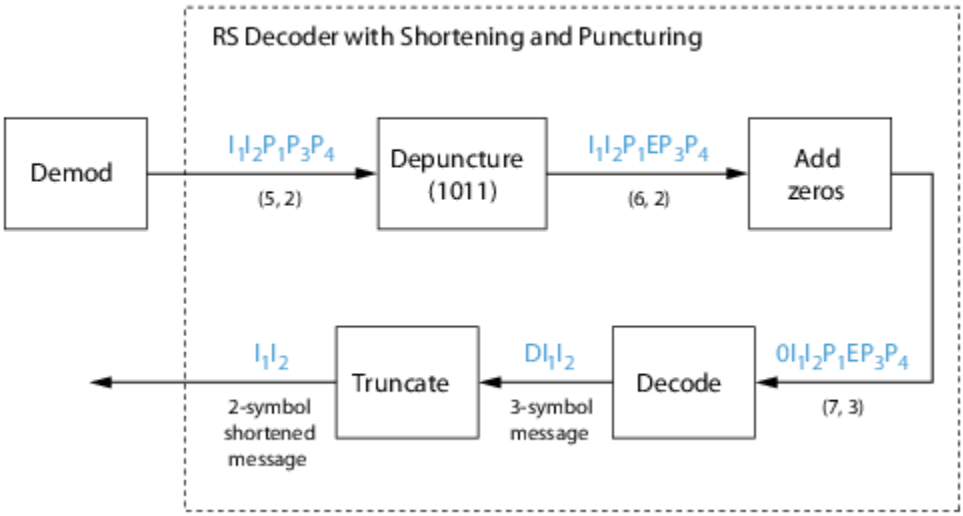


In this figure, the message source outputs two information symbols, designated by I_1I_2. (For a BCH example, the symbols are simply binary bits.) Because the code is a shortened (7,3) code, a zero must be added ahead of the information symbols, yielding a three-symbol message of $0I_1I_2$. The modified message sequence is then RS encoded, and the added information zero is subsequently removed, which yields a result of $I_1I_2P_1P_2P_3P_4$. (In this example, the parity bits are at the end of the codeword.)

The puncturing operation is governed by the puncture vector, which, in this case, is 1011. Within the puncture vector, a 1 means that the symbol is kept, and a 0 means that the symbol is thrown away. In this example, the puncturing operation removes the second parity symbol, yielding a final vector of $I_1I_2P_1P_3P_4$.

Decoder Example with Shortening and Puncturing

The following figure shows how the RS encoder operates on a shortened and punctured codeword.

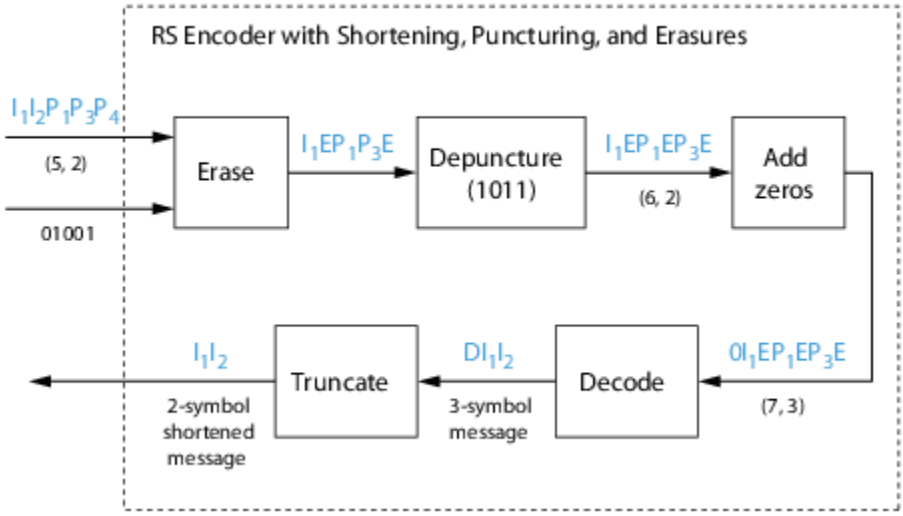


This case corresponds to the encoder operations shown in the figure of the RS encoder with shortening and puncturing. As shown in the preceding figure, the encoder receives a (5,2) codeword, because it has been shortened from a (7,3) codeword by one symbol, and one symbol has also been punctured.

As a first step, the decoder adds an erasure, designated by E, in the second parity position of the codeword. This corresponds to the puncture vector 1011. Adding a zero accounts for shortening, in the same way as shown in the preceding figure. The single erasure does not exceed the erasure-correcting capability of the code, which can correct four erasures. The decoding operation results in the three-symbol message DI_1I_2. The first symbol is truncated, as in the preceding figure, yielding a final output of I_1I_2.

Encoder Example with Shortening, Puncturing, and Erasures

The following figure shows the decoder operating on the punctured, shortened codeword, while also correcting erasures generated by the receiver.



In this figure, demodulator receives the $I_1I_2P_1P_3P_4$ vector that the encoder sent. The demodulator declares that two of the five received symbols are unreliable enough to be erased, such that symbols 2 and 5 are deemed to be erasures. The 01001 vector, provided by an external source, indicates these erasures. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered.

The decoder blocks receive the codeword and the erasure vector, and perform the erasures indicated by the vector 01001. Within the erasures vector, a 1 means that the symbol is to be replaced with an erasure symbol, and a 0 means that the symbol is passed unaltered. The resulting codeword vector is $I_1EP_1P_3E$, where E is an erasure symbol.

The codeword is then depunctured, according to the puncture vector used in the encoding operation (i.e., 1011). Thus, an erasure symbol is inserted between P_1 and P_3, yielding a codeword vector of $I_1EP_1EP_3E$.

Just prior to decoding, the addition of zeros at the beginning of the information vector accounts for the shortening. The resulting vector is $0I_1EP_1EP_3E$, such that a (7,3) codeword is sent to the Berlekamp algorithm.

This codeword is decoded, yielding a three-symbol message of DI_1I_2 (where D refers to a dummy symbol). Finally, the removal of the D symbol from the message vector accounts for the shortening and yields the original I_1I_2 vector.

For additional information, see the “Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink” on page 19-3 example.

LDPC Codes

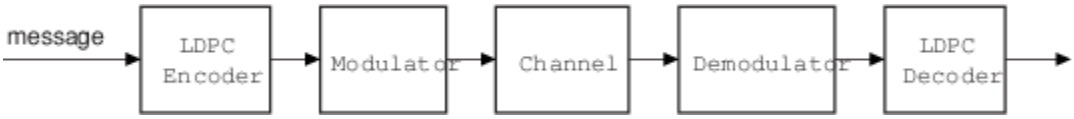
Low-Density Parity-Check (LDPC) codes are linear error control codes with:

- Sparse parity-check matrices
- Long block lengths that can attain performance near the Shannon limit (see LDPC Encoder and LDPC Decoder)

Communications Toolbox performs LDPC Coding using Simulink blocks and MATLAB objects.

The decoding process is done iteratively. If the number of iterations is too small, the algorithm may not converge. You may need to experiment with the number of iterations to find an appropriate value for your model. For details on the decoding algorithm, see Decoding Algorithm.

Unlike some other codecs, you cannot connect an LDPC decoder directly to the output of an LDPC encoder, because the decoder requires log-likelihood ratios (LLR). Thus, you may use a demodulator to compute the LLRs.



Also, unlike other decoders, it is possible (although rare) that the output of the LDPC decoder does not satisfy all parity checks.

Galois Field Computations

A *Galois field* is an algebraic field that has a finite number of members. Galois fields having 2^m members are used in error-control coding and are denoted GF(2^m). This chapter describes how to work with fields that have 2^m members, where m is an integer between 1 and 16. The sections in this chapter are as follows.

If you need to use Galois fields having an odd number of elements, see “Galois Fields of Odd Characteristic” on page 16-104.

For more details about specific functions that process arrays of Galois field elements, see the online reference pages in the documentation for MATLAB or for Communications Toolbox software.

Note Please note that the Galois field objects do not support the `copy` method.

MATLAB functions whose generalization to Galois fields is straightforward to describe do not have reference pages in this manual because the entries would be identical to those in the MATLAB documentation.

Galois Field Terminology

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

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

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

Representing Elements of Galois Fields

Section Overview

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

Creating a Galois field array

To begin working with data from a Galois field GF(2^m), you must set the context by associating the data with crucial information about the field. The `gf` function performs this association and creates a Galois field array in MATLAB. This function accepts as inputs

- The Galois field data, `x`, which is a MATLAB array whose elements are integers between 0 and `2^m-1`.
- (*Optional*) An integer, `m`, that indicates `x` is in the field GF(2^m). Valid values of `m` are between 1 and 16. The default is 1, which means that the field is GF(2).
- (*Optional*) A positive integer that indicates which primitive polynomial for GF(2^m) you are using in the representations in `x`. If you omit this input argument, `gf` uses a default primitive polynomial for GF(2^m). For information about this argument, see “Primitive Polynomials and Element Representations” on page 16-85.

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

When MATLAB Implicitly Creates a Galois field array

Some operations on Galois field arrays require multiple arguments. If you specify one argument that is a Galois field array and another that is an ordinary MATLAB array, MATLAB interprets both as Galois field arrays in the same field. It implicitly invokes the `gf` function on the ordinary MATLAB array. This implicit invocation simplifies your syntax because you can omit some references to the `gf` function. For an example of the simplification, see “Example: Addition and Subtraction” on page 16-88.

Example: Creating Galois Field Variables

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

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

b = a + a % Add a to itself, creating b.
```

The output is

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

Array elements =

     0     1     2     3


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

Array elements =

     0     0     0     0
```

The output shows the values of the Galois field arrays named `a` and `b`. Each output section indicates

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

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

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

Example: Representing Elements of GF(8)

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

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

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

How Integers Correspond to Galois Field Elements

Building on the GF(8) example above on page 16-84, this section explains the interpretation of array elements in a Galois field array in greater generality. The field GF(`2^m`) has `2^m` distinct elements, which this toolbox labels as 0, 1, 2,..., `2^m-1`. These integer labels correspond to elements of the Galois field via a polynomial expression involving a primitive element of the field. More specifically, each integer between 0 and `2^m-1` has a binary representation in `m` bits. Using the bits in the binary representation as coefficients in a polynomial, where the least significant bit is the constant term, leads to a binary polynomial whose order is at most `m-1`. Evaluating the binary polynomial at a primitive element of GF(`2^m`) leads to an element of the field.

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

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

$$X = b_{m-1} \cdot 2^{m-1} + \cdots + b_2 \cdot 4 + b_1 \cdot 2 + b_0$$

$$\leftrightarrow b_{m-1} \cdot A^{m-1} + \cdots + b_2 \cdot A^2 + b_1 \cdot A + b_0$$

Example: Representing a Primitive Element

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

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

The output is

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

Array elements =

     2
```

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

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

Primitive Polynomials and Element Representations

This section builds on the discussion in “Creating a Galois field array” on page 16-83 by describing how to specify your own primitive polynomial when you create a Galois field array. The topics are

If you perform many computations using a nondefault primitive polynomial, see “Speed and Nondefault Primitive Polynomials” on page 16-103.

Specifying the Primitive Polynomial

The discussion in “How Integers Correspond to Galois Field Elements” on page 16-84 refers to a primitive element, which is a root of a primitive polynomial of the field. When you use the `gf` function to create a Galois field array, the function interprets the integers in the array with respect to a specific default primitive polynomial for that field, unless you explicitly provide a different primitive polynomial. A list of the default primitive polynomials is on the reference page for the `gf` function.

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

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

instead of

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

The extra input argument, `25` in this case, specifies the primitive polynomial for the field GF(`2^m`) in a way similar to the representation described in “How Integers Correspond to Galois Field Elements” on page 16-84. In this case, the integer 25 corresponds to a binary representation of 11001, which in turn corresponds to the polynomial $D^4 + D^3 + 1$.

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

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

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

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

Array elements =

     1     2     3
```

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

Finding Primitive Polynomials

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

```
m = 4;
defaultprimpoly = primpoly(m) % Default primitive poly for GF(16)
allprimpolys = primpoly(m,'all') % All primitive polys for GF(16)
i1 = isprimitive(25) % Can 25 be the prim_poly input in gf(...)?
i2 = isprimitive(21) % Can 21 be the prim_poly input in gf(...)?
```

The output is below.

```
Primitive polynomial(s) =

D^4+D^1+1

defaultprimpoly =

    19

Primitive polynomial(s) =

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

allprimpolys =

    19
    25

i1 =

     1

i2 =

     0
```

Effect of Nondefault Primitive Polynomials on Numerical Results

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

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

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

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

Array elements =

     0


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

Array elements =

     6
```

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

Arithmetic in Galois Fields

Section Overview

You can perform arithmetic operations on Galois field arrays by using familiar MATLAB operators, listed in the table below. Whenever you operate on a pair of Galois field arrays, both arrays must be in the same Galois field.

Operation	Operator
Addition	+
Subtraction	-

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

For multiplication and division of polynomials over a Galois field, see “Addition and Subtraction of Polynomials” on page 16-98.

Example: Addition and Subtraction

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

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

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

The output is below.

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

Array elements =

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

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

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

Simplifying the Syntax

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

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

The output is below.

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

Array elements =

     4     4     4
     4     4     4
```

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

Example: Multiplication

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

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

Multiplication Table for GF(8)

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

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

The output is below.

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

Array elements =

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

Example: Division

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

Elementwise Division

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

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

Matrix Division

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

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

Example: Exponentiation

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

Elementwise Exponentiation

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

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

The output is below.

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

Array elements =

     0
```

Matrix Exponentiation

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

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

Example: Elementwise Logarithm

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

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

The output is

```
expformat =

     0     1     3     2     6     4     5
```

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

- A^5, using the last element of `expformat`
- A^2+A+1, using the binary representation of 7 as 111. See “Example: Representing Elements of GF(8)” on page 16-84 for more details.

Logical Operations in Galois Fields

Section Overview

You can apply logical tests to Galois field arrays and obtain a logical array. Some important types of tests are testing for the equality of two Galois field arrays and testing for nonzero values on page 16-92 in a Galois field array.

Testing for Equality

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

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

The output is below.

```
lg1 =

     1     1     1


lg2 =

     1     1     1


lg3 =

     0     1     1
```

Comparison of isequal and ==

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

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

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

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

Testing for Nonzero Values

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

```
m = 3; randels = gf(randi([0 2^m-1],6,1),m);
if all(randels) % If all elements are invertible
    invels = randels .\ 1; % Compute inverses of elements.
else
    disp('At least one element was not invertible.');
end
alph = gf(2,4);
poly = 1 + alph + alph^3;
if any(poly) % If poly contains a nonzero value
    disp('alph is not a root of 1 + D + D^3.');
end
code = [0:4 4 0; 3:7 4 5]
if all(code,2) % Is each row entirely nonzero?
    disp('Both codewords are entirely nonzero.');
else
    disp('At least one codeword contains a zero.');
end
```

Matrix Manipulation in Galois Fields

Basic Manipulations of Galois Field Arrays

Basic array operations on Galois field arrays are in the table below. The functionality of these operations is analogous to the MATLAB operations having the same syntax.

Operation	Syntax
Index into array, possibly using colon operator instead of a vector of explicit indices	`a(vector)` or `a(vector,vector1)`, where `vector` and/or `vector1` can be `":"` instead of a vector
Transpose array	`a'`
Concatenate matrices	`[a,b]` or `[a;b]`
Create array having specified diagonal elements	`diag(vector)` or `diag(vector,k)`
Extract diagonal elements	`diag(a)` or `diag(a,k)`
Extract lower triangular part	`tril(a)` or `tril(a,k)`
Extract upper triangular part	`triu(a)` or `triu(a,k)`
Change shape of array	`reshape(a,k1,k2)`

The code below uses some of these syntaxes.

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

Basic Information About Galois Field Arrays

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

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

Positions of Nonzero Elements

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

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

Linear Algebra in Galois Fields

Inverting Matrices and Computing Determinants

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

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

The code below illustrates matrix inversion and determinant computation.

```
m = 4;
randommatrix = gf(randi([0 2^m-1],4,4),m);
gfid = gf(eye(4),m);
if det(randommatrix) ~= 0
    invmatrix = inv(randommatrix);
    check1 = invmatrix * randommatrix;
    check2 = randommatrix * invmatrix;
    if (isequal(check1,gfid) & isequal(check2,gfid))
        disp('inv found the correct matrix inverse.');
    end
else
    disp('The matrix is not invertible.');
end
```

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

```
inv found the correct matrix inverse.
The matrix is not invertible.
```

Computing Ranks

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

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

The output is

```
ans =

     2     3
```

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

Factoring Square Matrices

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

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

Solving Linear Equations

To find a particular solution of a linear equation in a Galois field, use the `\` or `/` operator on Galois field arrays. The table below indicates the equation that each operator addresses, assuming that `A` and `B` are previously defined Galois field arrays.

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

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

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

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

Example: Solving Linear Equations

The examples below illustrate how to find particular solutions of linear equations over a Galois field.

```
m = 4;
A = gf(magic(3),m); % Square nonsingular matrix
```

```
Awide=[A, 2*A(:,3)]; % 3-by-4 matrix with redundancy on the right
Atall = Awide'; % 4-by-3 matrix with redundancy at the bottom
B = gf([0:2]',m);
C = [B; 2*B(3)];
D = [B; B(3)+1];
thesolution = A \ B; % Solution of A * x = B
thesolution2 = B' / A; % Solution of x * A = B'
ck1 = all(A * thesolution == B) % Check validity of solutions.
ck2 = all(thesolution2 * A == B')
% Awide * x = B has infinitely many solutions. Find one.
onesolution = Awide \ B;
ck3 = all(Awide * onesolution == B) % Check validity of solution.
% Atall * x = C has a solution.
asolution = Atall \ C;
ck4 = all(Atall * asolution == C) % Check validity of solution.
% Atall * x = D has no solution.
notasolution = Atall \ D;
ck5 = all(Atall * notasolution == D) % It is not a valid solution.
```

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

Signal Processing Operations in Galois Fields

Section Overview

You can perform some signal-processing operations on Galois field arrays, such as filtering on page 16-96, convolution on page 16-97, and the discrete Fourier transform on page 16-97.

This section describes how to perform these operations.

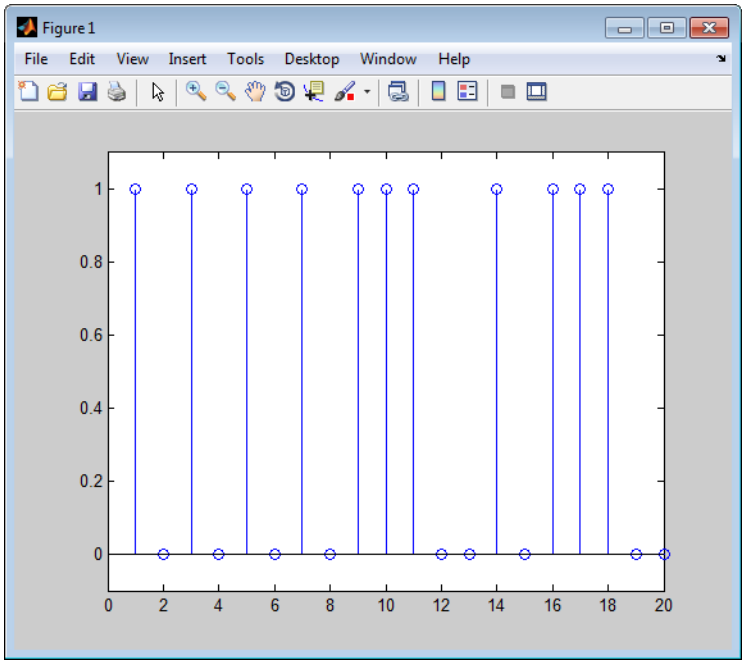
Other information about the corresponding operations for ordinary real vectors is in the Signal Processing Toolbox™ documentation.

Filtering

To filter a Galois vector, use the `filter` function. It behaves like the ordinary MATLAB `filter` function when given exactly three input arguments.

The code and diagram below give the impulse response of a particular filter over GF(2).

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



Convolution

Communications Toolbox software offers two equivalent ways to convolve a pair of Galois vectors:

- Use the `conv` function, as described in “Multiplication and Division of Polynomials” on page 16-99. This works because convolving two vectors is equivalent to multiplying the two polynomials whose coefficients are the entries of the vectors.
- Use the `convmtx` function to compute the convolution matrix of one of the vectors, and then multiply that matrix by the other vector. This works because convolving two vectors is equivalent to filtering one of the vectors by the other. The equivalence permits the representation of a digital filter as a convolution matrix, which you can then multiply by any Galois vector of appropriate length.

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

Example

Computes the convolution matrix for a vector `b` in GF(4). Represent the numerator coefficients for a digital filter, and then illustrate the two equivalent ways to convolve `b` with `x` over the Galois field.

```
m = 2; b = gf([1 2 3]',m);
n = 3; x = gf(randi([0 2^m-1],n,1),m);
C = convmtx(b,n); % Compute convolution matrix.
v1 = conv(b,x); % Use conv to convolve b with x
v2 = C*x; % Use C to convolve b with x.
```

Discrete Fourier Transform

The discrete Fourier transform is an important tool in digital signal processing. This toolbox offers these tools to help you process discrete Fourier transforms:

- `fft`, which transforms a Galois vector
- `ifft`, which inverts the discrete Fourier transform on a Galois vector
- `dftmtx`, which returns a Galois field array that you can use to perform or invert the discrete Fourier transform on a Galois vector

In all cases, the vector being transformed must be a Galois vector of length 2^m-1 in the field GF(2^m). The following example illustrates the use of these functions. You can check, using the `isequal` function, that `y` equals `y1`, `z` equals `z1`, and `z` equals `x`.

```
m = 4;
x = gf(randi([0 2^m-1],2^m-1,1),m); % A vector to transform
alph = gf(2,m);
dm = dftmtx(alph);
idm = dftmtx(1/alph);
y = dm*x; % Transform x using the result of dftmtx.
y1 = fft(x); % Transform x using fft.
z = idm*y; % Recover x using the result of dftmtx(1/alph).
z1 = ifft(y1); % Recover x using ifft.
```

Tip If you have many vectors that you want to transform (in the same field), it might be faster to use `dftmtx` once and matrix multiplication many times, instead of using `fft` many times.

Polynomials over Galois Fields

Section Overview

You can use Galois vectors to represent polynomials in an indeterminate quantity x, with coefficients in a Galois field. Form the representation by listing the coefficients of the polynomial in a vector in order of descending powers of x. For example, the vector

```
gf([2 1 0 3],4)
```

represents the polynomial $Ax^3 + 1x^2 + 0x + (A+1)$, where

- A is a primitive element in the field GF(2^4).
- x is the indeterminate quantity in the polynomial.

You can then use such a Galois vector to perform arithmetic with, evaluate on page 16-99, and find roots on page 16-100 of polynomials. You can also find minimal polynomials on page 16-101 of elements of a Galois field.

Addition and Subtraction of Polynomials

To add and subtract polynomials, use `+` and `-` on equal-length Galois vectors that represent the polynomials. If one polynomial has lower degree than the other, you must pad the shorter vector with

zeros at the beginning so the two vectors have the same length. The example below shows how to add a degree-one and a degree-two polynomial.

```
lin = gf([4 2],3); % A^2 x + A, which is linear in x
linpadded = gf([0 4 2],3); % The same polynomial, zero-padded
quadr = gf([1 4 2],3); % x^2 + A^2 x + A, which is quadratic in x
% Can't do lin + quadr because they have different vector lengths.
sumpoly = [0, lin] + quadr; % Sum of the two polynomials
sumpoly2 = linpadded + quadr; % The same sum
```

Multiplication and Division of Polynomials

To multiply and divide polynomials, use `conv` and `deconv` on Galois vectors that represent the polynomials. Multiplication and division of polynomials is equivalent to convolution and deconvolution of vectors. The `deconv` function returns the quotient of the two polynomials as well as the remainder polynomial. Examples are below.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
bpoly = gf([1 1],m); % x + 1
xpoly = gf([1 0],m); % x
% Product is A^2 x^3 + x^2 + (A^2 + A) x + (A + 1).
cpoly = conv(apoly,bpoly);
[a2,remd] = deconv(cpoly,bpoly); % a2==apoly. remd is zero.
[otherpol,remd2] = deconv(cpoly,xpoly); % remd is nonzero.
```

The multiplication and division operators in “Arithmetic in Galois Fields” on page 16-87 multiply elements or matrices, not polynomials.

Evaluating Polynomials

To evaluate a polynomial at an element of a Galois field, use `polyval`. It behaves like the ordinary MATLAB `polyval` function when given exactly two input arguments. The example below evaluates a polynomial at several elements in a field and checks the results using `.^` and `.*` in the field.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
x0 = gf([0 1 2],m); % Points at which to evaluate the polynomial
y = polyval(apoly,x0)

a = gf(2,m); % Primitive element of the field, corresponding to A.
y2 = a.^2.*x0.^2 + (a.^2+1).*x0 + (a+1) % Check the result.
```

The output is below.

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

Array elements =

     3     2    10


y2 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10
```

The first element of `y` evaluates the polynomial at `0` and, therefore, returns the polynomial's constant term of `3`.

Roots of Polynomials

To find the roots of a polynomial in a Galois field, use the `roots` function on a Galois vector that represents the polynomial. This function finds roots that are in the same field that the Galois vector is in. The number of times an entry appears in the output vector from `roots` is exactly its multiplicity as a root of the polynomial.

Note If the Galois vector is in GF(2^m), the polynomial it represents might have additional roots in some extension field GF($(2^m)^k$). However, `roots` does not find those additional roots or indicate their existence.

The examples below find roots of cubic polynomials in GF(8).

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
   root = rts(ii);
   rootsquared = gfmul(root,root,field);
   rootcubed = gfmul(root,rootsquared,field);
   answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
   % Recall that 1 is really alpha to the zero power.
   % If answer = -Inf, then the variable root represents
   % a root of the polynomial.
end
answer
```

Roots of Binary Polynomials

In the special case of a polynomial having binary coefficients, it is also easy to find roots that exist in an extension field. This is because the elements `0` and `1` have the same unambiguous representation in all fields of characteristic two. To find roots of a binary polynomial in an extension field, apply the `roots` function to a Galois vector in the extension field whose array elements are the binary coefficients of the polynomial.

The example below seeks the roots of a binary polynomial in various fields.

```
gf2poly = gf([1 1 1],1); % x^2 + x + 1 in GF(2)
noroots = roots(gf2poly); % No roots in the ground field, GF(2)
gf4poly = gf([1 1 1],2); % x^2 + x + 1 in GF(4)
roots4 = roots(gf4poly); % The roots are A and A+1, in GF(4).
gf16poly = gf([1 1 1],4); % x^2 + x + 1 in GF(16)
roots16 = roots(gf16poly); % Roots in GF(16)
checkanswer4 = polyval(gf4poly,roots4); % Zero vector
checkanswer16 = polyval(gf16poly,roots16); % Zero vector
```

The roots of the polynomial do not exist in GF(2), so `noroots` is an empty array. However, the roots of the polynomial exist in GF(4) as well as in GF(16), so `roots4` and `roots16` are nonempty.

Notice that `roots4` and `roots16` are not equal to each other. They differ in these ways:

- `roots4` is a GF(4) array, while `roots16` is a GF(16) array. MATLAB keeps track of the underlying field of a Galois field array.
- The array elements in `roots4` and `roots16` differ because they use representations with respect to different primitive polynomials. For example, `2` (which represents a primitive element) is an element of the vector `roots4` because the default primitive polynomial for GF(4) is the same polynomial that `gf4poly` represents. On the other hand, `2` is not an element of `roots16` because the primitive element of GF(16) is not a root of the polynomial that `gf16poly` represents.

Minimal Polynomials

The minimal polynomial of an element of GF(2^m) is the smallest degree nonzero binary-coefficient polynomial having that element as a root in GF(2^m). To find the minimal polynomial of an element or a column vector of elements, use the `minpol` function.

The code below finds that the minimal polynomial of `gf(6,4)` is $D^2 + D + 1$ and then checks that `gf(6,4)` is indeed among the roots of that polynomial in the field GF(16).

```
m = 4;
e = gf(6,4);
em = minpol(e) % Find minimal polynomial of e. em is in GF(2).

emr = roots(gf([0 0 1 1 1],m)) % Roots of D^2+D+1 in GF(2^m)
```

The output is

```
em = GF(2) array.

Array elements =

     0     0     1     1     1


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

Array elements =

     6
     7
```

To find out which elements of a Galois field share the same minimal polynomial, use the `cosets` function.

Manipulating Galois Variables

Section Overview

This section describes techniques for manipulating Galois variables or for transferring information between Galois field arrays and ordinary MATLAB arrays.

Note These techniques are particularly relevant if you write MATLAB file functions that process Galois field arrays. For an example of this type of usage, enter `edit gf/conv` in the Command Window and examine the first several lines of code in the editor window.

Determining Whether a Variable Is a Galois Field Array

To find out whether a variable is a Galois field array rather than an ordinary MATLAB array, use the `isa` function. An illustration is below.

```
mlvar = eye(3);
gfvar = gf(mlvar,3);
no = isa(mlvar,'gf'); % False because mlvar is not a Galois array
yes = isa(gfvar,'gf'); % True because gfvar is a Galois array
```

Extracting Information from a Galois Field Array

To extract the array elements, field order, or primitive polynomial from a variable that is a Galois field array, append a suffix to the name of the variable. The table below lists the exact suffixes, which are independent of the name of the variable.

Information	Suffix	Output Value
Array elements	`.x`	MATLAB array of type `uint16` that contains the data values from the Galois field array.
Field order	`.m`	Integer of type `double` that indicates that the Galois field array is in GF(`2^m`).
Primitive polynomial	`.prim_poly`	Integer of type `uint32` that represents the primitive polynomial. The representation is similar to the description in “How Integers Correspond to Galois Field Elements” on page 16-84.

Note If the output value is an integer data type and you want to convert it to `double` for later manipulation, use the `double` function.

The code below illustrates the use of these suffixes. The definition of `empr` uses a vector of binary coefficients of a polynomial to create a Galois field array in an extension field. Another part of the example retrieves the primitive polynomial for the field and converts it to a binary vector representation having the appropriate number of bits.

```
% Check that e solves its own minimal polynomial.
e = gf(6,4); % An element of GF(16)
emp = minpol(e); % The minimal polynomial, emp, is in GF(2).
empr = roots(gf(emp.x,e.m)); % Find roots of emp in GF(16).

% Check that the primitive element gf(2,m) is
% really a root of the primitive polynomial for the field.
primpoly_int = double(e.prim_poly);
```

```
mval = e.m;
primpoly_vect = gf(de2bi(primpoly_int,mval+1,'left-msb'),mval);
containstwo = roots(primpoly_vect); % Output vector includes 2.
```

Converting Galois Field Array to Doubles

```
a = gf([1,0])
b = double(a.x) %a.x is in uint16
```

MATLAB returns the following:

```
a = GF(2) array.

Array elements =

           1           0


b =

     1     0
```

Speed and Nondefault Primitive Polynomials

"Primitive Polynomials and Element Representations" on page 16-85 describes how to represent elements of a Galois field with respect to a primitive polynomial of your choice. This section describes how you can increase the speed of computations involving a Galois field array that uses a primitive polynomial other than the default primitive polynomial. The technique is recommended if you perform many such computations.

The mechanism for increasing the speed is a data file, `userGftable.mat`, that some computational functions use to avoid performing certain computations repeatedly. To take advantage of this mechanism for your combination of field order (`m`) and primitive polynomial (`prim_poly`):

1. Navigate in the MATLAB application to a folder to which you have write permission. You can use either the `cd` function or the Current Folder feature to navigate.
2. Define `m` and `prim_poly` as workspace variables. For example:

   ```
   m = 3; prim_poly = 13; % Examples of valid values
   ```
3. Invoke the `gftable` function:

   ```
   gftable(m,prim_poly); % If you previously defined m and prim_poly
   ```

The function revises or creates `userGftable.mat` in your current working folder to include data relating to your combination of field order and primitive polynomial. After you initially invest the time to invoke `gftable`, subsequent computations using those values of `m` and `prim_poly` should be faster.

Note If you change your current working directory after invoking `gftable`, you must place `userGftable.mat` on your MATLAB path to ensure that MATLAB can see it. Do this by using the `addpath` command to prefix the directory containing `userGftable.mat` to your MATLAB path. If you have multiple copies of `userGftable.mat` on your path, use `which('userGftable.mat','-all')` to find out where they are and which one MATLAB is using.

To see how much `gftable` improves the speed of your computations, you can surround your computations with the `tic` and `toc` functions. See the `gftable` reference page for an example.

Selected Bibliography for Galois Fields

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, MA, Addison-Wesley, 1993.

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

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

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

Galois Fields of Odd Characteristic

A *Galois field* is an algebraic field having p^m elements, where p is prime and m is a positive integer. This chapter describes how to work with Galois fields in which p is *odd*. To work with Galois fields having an even number of elements, see Galois Field Computations on page 16-81. The sections in this chapter are as follows.

Galois Field Terminology

Throughout this section, p is an odd prime number and m is a positive integer.

Also, this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [5].

- A *primitive element* of GF(p^m) is a cyclic generator of the group of nonzero elements of GF(p^m). This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power. Primitive elements are called A throughout this section.
- A *primitive polynomial* for GF(p^m) is the minimal polynomial of some primitive element of GF(p^m). As a consequence, it has degree m and is irreducible.

Representing Elements of Galois Fields

Section Overview

This section discusses how to represent Galois field elements using this toolbox's exponential on page 16-105 format and polynomial on page 16-106 format. It also describes a way to list all elements on page 16-106 of the Galois field, because some functions use such a list as an input argument. Finally, it discusses the nonuniqueness on page 16-107 of representations of Galois field elements.

The elements of GF(p) can be represented using the integers from 0 to p-1.

When m is at least 2, GF(p^m) is called an extension field. Integers alone cannot represent the elements of GF(p^m) in a straightforward way. MATLAB technical computing software uses two main conventions for representing elements of GF(p^m): the exponential format and the polynomial format.

Note Both the exponential format and the polynomial format are relative to your choice of a particular primitive element A of GF(p^m).

Exponential Format

This format uses the property that every nonzero element of GF(p^m) can be expressed as A^c for some integer c between 0 and p^m-2. Higher exponents are not needed, because the theory of Galois fields implies that every nonzero element of GF(p^m) satisfies the equation $x^{q-1} = 1$ where $q = p^m$.

The use of the exponential format is shown in the table below.

Element of GF(p^m)	MATLAB Representation of the Element
0	`-Inf`
$A^0 = 1$	`0`
A^1	`1`
...	...
A^{q-2} where `q` = p^m	`q-2`

Although `-Inf` is the standard exponential representation of the zero element, all negative integers are equivalent to `-Inf` when used as *input* arguments in exponential format. This equivalence can be useful; for example, see the concise line of code at the end of the section “Default Primitive Polynomials” on page 16-107.

Note The equivalence of all negative integers and `-Inf` as exponential formats means that, for example, -1 does *not* represent A^{-1}, the multiplicative inverse of A. Instead, -1 represents the zero element of the field.

Polynomial Format

The polynomial format uses the property that every element of GF(p^m) can be expressed as a polynomial in A with exponents between 0 and m-1, and coefficients in GF(p). In the polynomial format, the element

`A(1) + A(2)` A + `A(3)` A^2 + ... + `A(m)` A^{m-1}

is represented in MATLAB by the vector

```
[A(1) A(2) A(3) ... A(m)]
```

Note The Galois field functions in this toolbox represent a polynomial as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

List of All Elements of a Galois Field

Some Galois field functions in this toolbox require an argument that lists all elements of an extension field GF(p^m). This is again relative to a particular primitive element A of GF(p^m). The proper format for the list of elements is that of a matrix having p^m rows, one for each element of the field. The matrix has m columns, one for each coefficient of a power of A in the polynomial format shown in “Polynomial Format” on page 16-106 above. The first row contains only zeros because it corresponds to the zero element in GF(p^m). If k is between 2 and p^m, then the kth row specifies the polynomial format of the element A^{k-2}.

The minimal polynomial of A aids in the computation of this matrix, because it tells how to express A^m in terms of lower powers of A. For example, the table below lists the elements of GF(3^2), where A is a root of the primitive polynomial $2 + 2x + x^2$. This polynomial allows repeated use of the substitution

$A^2 = -2 - 2A = 1 + A$

when performing the computations in the middle column of the table.

Elements of GF(9)

Exponential Format	Polynomial Format	Row of MATLAB Matrix of Elements
A^{-Inf}	0	`0 0`
A^0	1	`1 0`
A^1	A	`0 1`
A^2	1+A	`1 1`
A^3	$A + A^2 = A + 1 + A = 1 + 2A$	`1 2`
A^4	$A + 2A^2 = A + 2 + 2A = 2$	`2 0`
A^5	2A	`0 2`
A^6	$2A^2 = 2 + 2A$	`2 2`
A^7	$2A + 2A^2 = 2A + 2 + 2A = 2 + A$	`2 1`

Example

An automatic way to generate the matrix whose rows are in the third column of the table above is to use the code below.

```
p = 3; m = 2;
% Use the primitive polynomial 2 + 2x + x^2 for GF(9).
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

The `gftuple` function is discussed in more detail in “Converting and Simplifying Element Formats” on page 16-107.

Nonuniqueness of Representations

A given field has more than one primitive element. If two primitive elements have different minimal polynomials, then the corresponding matrices of elements will have their rows in a different order. If the two primitive elements share the same minimal polynomial, then the matrix of elements of the field is the same.

Note You can use whatever primitive element you want, as long as you understand how the inputs and outputs of Galois field functions depend on the choice of *some* primitive polynomial. It is usually best to use the same primitive polynomial throughout a given script or function.

Other ways in which representations of elements are not unique arise from the equations that Galois field elements satisfy. For example, an exponential format of 8 in GF(9) is really the same as an exponential format of 0, because $A^8 = 1 = A^0$ in GF(9). As another example, the substitution mentioned just before the table Elements of GF(9) shows that the polynomial format [0 0 1] is really the same as the polynomial format [1 1].

Default Primitive Polynomials

This toolbox provides a *default* primitive polynomial for each extension field. You can retrieve this polynomial using the `gfprimdf` function. The command

```
prim_poly = gfprimdf(m,p); % If m and p are already defined
```

produces the standard row-vector representation of the default minimal polynomial for GF(p^m).

For example, the command below shows that the default primitive polynomial for GF(9) is $2 + x + x^2$, *not* the polynomial used in “List of All Elements of a Galois Field” on page 16-106.

```
poly1=gfprimdf(2,3);


poly1 =

     2     1     1
```

To generate a list of elements of GF(p^m) using the default primitive polynomial, use the command

```
field = gftuple([-1:p^m-2]',m,p);
```

Converting and Simplifying Element Formats

Converting to Simplest Polynomial Format

The `gftuple` function produces the simplest polynomial representation of an element of GF(p^m), given either an exponential representation or a polynomial representation of that element. This can be useful for generating the list of elements of GF(p^m) that other functions require.

Using `gftuple` requires three arguments: one representing an element of GF(p^m), one indicating the primitive polynomial that MATLAB technical computing software should use when computing the output, and the prime p. The table below indicates how `gftuple` behaves when given the first two arguments in various formats.

Behavior of gftuple Depending on Format of First Two Inputs

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Exponential format; c = any integer	Integer m > 1	Polynomial format of A^c, where A is a root of the *default* primitive polynomial for GF(p^m)
Example: `tp = gftuple(6,2,3); % c = 6 here`		
Exponential format; c = any integer	Vector of coefficients of primitive polynomial	Polynomial format of A^c, where A is a root of the *given* primitive polynomial
Example: `polynomial = gfprimdf(2,3); tp = gftuple(6,polynomial,3); % c = 6 here`		
Polynomial format of any degree	Integer m > 1	Polynomial format of degree < m, using *default* primitive polynomial for GF(p^m) to simplify
Example: `tp = gftuple([0 0 0 0 0 0 1],2,3);`		
Polynomial format of any degree	Vector of coefficients of primitive polynomial	Polynomial format of degree < m, using the *given* primitive polynomial for GF(p^m) to simplify
Example: `polynomial = gfprimdf(2,3); tp = gftuple([0 0 0 0 0 0 1],polynomial,3);`		

The four examples that appear in the table above all produce the same vector `tp = [2, 1]`, but their different inputs to `gftuple` correspond to the lines of the table. Each example expresses the fact that $A^6 = 2+A$, where A is a root of the (default) primitive polynomial $2 + x + x^2$ for GF(3^2).

Example

This example shows how `gfconv` and `gftuple` combine to multiply two polynomial-format elements of GF(3^4). Initially, `gfconv` multiplies the two polynomials, treating the primitive element as if it were a variable. This produces a high-order polynomial, which `gftuple` simplifies using the polynomial equation that the primitive element satisfies. The final result is the simplest polynomial format of the product.

```
p = 3; m = 4;
a = [1 2 0 1]; b = [2 2 1 2];
notsimple = gfconv(a,b,p) % a times b, using high powers of alpha
simple = gftuple(notsimple,m,p) %Highest exponent of alpha is m-1
```

The output is below.

```
notsimple =

     2     0     2     0     0     1     2


simple =

     2     1     0     1
```

Example: Generating a List of Galois Field Elements

This example applies the conversion functionality to the task of generating a matrix that lists all elements of a Galois field. A matrix that lists all field elements is an input argument in functions such as `gfadd` and `gfmul`. The variables `field1` and `field2` below have the format that such functions expect.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field1 = gftuple([-1:p^m-2]',m,p);

prim_poly = gfprimdf(m,p); % Or any primitive polynomial
% for GF(p^m)
field2 = gftuple([-1:p^m-2]',prim_poly,p);
```

Converting to Simplest Exponential Format

The same function `gftuple` also produces the simplest exponential representation of an element of GF(p^m), given either an exponential representation or a polynomial representation of that element. To retrieve this output, use the syntax

```
[polyformat, expformat] = gftuple(...)
```

The input format and the output `polyformat` are as in the table Behavior of gftuple Depending on Format of First Two Inputs. In addition, the variable `expformat` contains the simplest exponential format of the element represented in `polyformat`. It is *simplest* in the sense that the exponent is either `-Inf` or a number between 0 and p^m-2.

Example

To recover the exponential format of the element 2 + A that the previous section considered, use the commands below. In this case, `polyformat` contains redundant information, while `expformat` contains the desired result.

```
[polyformat, expformat] = gftuple([2 1],2,3)


polyformat =

     2     1

expformat =
```

```
     6
```

This output appears at first to contradict the information in the table Elements of GF(9) , but in fact it does not. The table uses a different primitive element; two plus that primitive element has the polynomial and exponential formats shown below.

```
prim_poly = [2 2 1];
[polyformat2, expformat2] = gftuple([2 1],prim_poly,3)
```

The output below reflects the information in the bottom line of the table.

```
polyformat2 =

     2     1


expformat2 =

     7
```

Arithmetic in Galois Fields

Section Overview

You can add, subtract, multiply, and divide elements of Galois fields using the functions `gfadd`, `gfsub`, `gfmul`, and `gfdiv`, respectively. Each of these functions has a mode for prime fields on page 16-110 and a mode for extension fields on page 16-111.

Arithmetic in Prime Fields

Arithmetic in GF(p) is the same as arithmetic modulo p. The functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` accept two arguments that represent elements of GF(p) as integers between 0 and p-1. The third argument specifies p.

Example: Addition Table for GF(5)

The code below constructs an addition table for GF(5). If `a` and `b` are between 0 and 4, then the element `gfp_add(a+1,b+1)` represents the sum `a+b` in GF(5). For example, `gfp_add(3,5) = 1` because 2+4 is 1 modulo 5.

```
p = 5;
row = 0:p-1;
table = ones(p,1)*row;
gfp_add = gfadd(table,table',p)
```

The output for this example follows.

```
gfp_add =

     0     1     2     3     4
     1     2     3     4     0
     2     3     4     0     1
```

```
     3     4     0     1     2
     4     0     1     2     3
```

Other values of `p` produce tables for different prime fields GF(`p`). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(`p`).

Arithmetic in Extension Fields

The same arithmetic functions can add elements of GF(p^m) when m > 1, but the format of the arguments is more complicated than in the case above. In general, arithmetic in extension fields is more complicated than arithmetic in prime fields; see the works listed in “Selected Bibliography for Galois Fields” on page 16-115 for details about how the arithmetic operations work.

When working in extension fields, the functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` use the first two arguments to represent elements of GF(p^m) in exponential format. The third argument, which is required, lists all elements of GF(p^m) as described in “List of All Elements of a Galois Field” on page 16-106. The result is in exponential format.

Example: Addition Table for GF(9)

The code below constructs an addition table for GF(3^2), using exponential formats relative to a root of the default primitive polynomial for GF(9). If `a` and `b` are between -1 and 7, then the element `gfpm_add(a+2,b+2)` represents the sum of A^a and A^b in GF(9). For example, `gfpm_add(4,6) = 5` because

$A^2 + A^4 = A^5$

Using the fourth and sixth rows of the matrix `field`, you can verify that

$A^2 + A^4 = (1 + 2A) + (2 + 0A) = 3 + 2A = 0 + 2A = A^5$ modulo 3.

```
p = 3; m = 2; % Work in GF(3^2).
field = gftuple([-1:p^m-2]',m,p); % Construct list of elements.
row = -1:p^m-2;
table = ones(p^m,1)*row;
gfpm_add = gfadd(table,table',field)
```

The output is below.

```
gfpm_add =

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

Note If you used a different primitive polynomial, then the tables would look different. This makes sense because the ordering of the rows and columns of the tables was based on that particular choice of primitive polynomial and not on any natural ordering of the elements of GF(9).

Other values of `p` and `m` produce tables for different extension fields GF(`p^m`). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(`p^m`).

Polynomials over Prime Fields

Section Overview

A polynomial over GF(p) is a polynomial whose coefficients are elements of GF(p). Communications Toolbox software provides functions for

- Changing polynomials in cosmetic on page 16-112 ways
- Performing polynomial arithmetic on page 16-112
- Characterizing polynomials as primitive or irreducible on page 16-113
- Finding roots on page 16-113 of polynomials in a Galois field

Note The Galois field functions in this toolbox represent a polynomial over GF(p) for odd values of p as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

Cosmetic Changes of Polynomials

To display the traditionally formatted polynomial that corresponds to a row vector containing coefficients, use `gfpretty`. To truncate a polynomial by removing all zero-coefficient terms that have exponents *higher* than the degree of the polynomial, use `gftrunc`. For example,

```
polynom = gftrunc([1 20 394 10 0 0 29 3 0 0])
gfpretty(polynom)
```

The output is below.

```
polynom =

     1    20   394    10     0     0    29     3


                                 2       3       6      7
                  1 + 20 X + 394 X  + 10 X  + 29 X  + 3 X
```

Note If you do not use a fixed-width font, then the spacing in the display might not look correct.

Polynomial Arithmetic

The functions `gfadd` and `gfsub` add and subtract, respectively, polynomials over GF(p). The `gfconv` function multiplies polynomials over GF(p). The `gfdeconv` function divides polynomials in GF(p), producing a quotient polynomial and a remainder polynomial. For example, the commands below show that $2 + x + x^2$ times $1 + x$ over the field GF(3) is $2 + 2x^2 + x^3$.

```
a = gfconv([2 1 1],[1 1],3)
[quot, remd] = gfdeconv(a,[2 1 1],3)
```

The output is below.

```
a =

     2     0     2     1


quot =

     1     1

remd =

     0
```

The previously discussed functions `gfadd` and `gfsub` add and subtract, respectively, polynomials. Because it uses a vector of coefficients to represent a polynomial, MATLAB does not distinguish between adding two polynomials and adding two row vectors elementwise.

Characterization of Polynomials

Given a polynomial over GF(p), the `gfprimck` function determines whether it is irreducible and/or primitive. By definition, if it is primitive then it is irreducible; however, the reverse is not necessarily true. The `gfprimdf` and `gfprimfd` functions return primitive polynomials.

Given an element of GF(p^m), the `gfminpol` function computes its minimal polynomial over GF(p).

Example

For example, the code below reflects the irreducibility of all minimal polynomials. However, the minimal polynomial of a nonprimitive element is not a primitive polynomial.

```
p = 3; m = 4;
% Use default primitive polynomial here.

prim_poly = gfminpol(1,m,p);
ckprim = gfprimck(prim_poly,p);
% ckprim = 1, since prim_poly represents a primitive polynomial.

notprimpoly = gfminpol(5,m,p);
cknotprim = gfprimck(notprimpoly,p);
% cknotprim = 0 (irreducible but not primitive)
% since alpha^5 is not a primitive element when p = 3.

ckreducible = gfprimck([0 1 1],p);
% ckreducible = -1 since the polynomial is reducible.
```

Roots of Polynomials

Given a polynomial over GF(p), the `gfroots` function finds the roots of the polynomial in a suitable extension field GF(p^m). There are two ways to tell MATLAB the degree m of the extension field GF(p^m), as shown in the following table.

Formats for Second Argument of gfroots

Second Argument	Represents
A positive integer	m as in GF(p^m). MATLAB uses the default primitive polynomial in its computations.
A row vector	A primitive polynomial for GF(p^m). Here m is the degree of this primitive polynomial.

Example: Roots of a Polynomial in GF(9)

The code below finds roots of the polynomial $1 + x^2 + x^3$ in GF(9) and then checks that they are indeed roots. The exponential format of elements of GF(9) is used throughout.

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
   root = rts(ii);
   rootsquared = gfmul(root,root,field);
   rootcubed = gfmul(root,rootsquared,field);
   answer(ii)= gfadd(gfadd(0,rootsquared,field),rootcubed,field);
   % Recall that 1 is really alpha to the zero power.
   % If answer = -Inf, then the variable root represents
   % a root of the polynomial.
end
answer
```

The output shows that A^0 (which equals 1), A^5, and A^7 are roots.

```
roots =

     0
     5
     7


answer =

  -Inf  -Inf  -Inf
```

See the reference page for `gfroots` to see how `gfroots` can also provide you with the polynomial formats of the roots and the list of all elements of the field.

Other Galois Field Functions

See the online reference pages for information about these other Galois field functions in Communications Toolbox software:

- `gfcosets`, which produces cyclotomic cosets
- `gffilter`, which filters data using GF(p) polynomials
- `gfprimfd`, which finds primitive polynomials

- `gfrank`, which computes the rank of a matrix over GF(p)
- `gfrepcov`, which converts one binary polynomial representation to another

Selected Bibliography for Galois Fields

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, Mass., Addison-Wesley, 1983.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, Mass., Addison-Wesley, 1993.

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

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

Interleaving

In this section...

Block Interleaving

Block Interleaving Features

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

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

Each interleaver function has a corresponding deinterleaver function. In typical usage of the interleaver/deinterleaver pairs, the inputs of the deinterleaver match those of the interleaver, except for the data being rearranged.

A block interleaver accepts a set of symbols and rearranges them, without repeating or omitting any of the symbols in the set. The number of symbols in each set is fixed for a given interleaver.

The set of block interleavers in this toolbox includes a general block interleaver as well as several special cases. Each special-case interleaver function uses the same computational code that the general block interleaver function uses, but provides a syntax that is more suitable for the special case. The interleaver functions are described below.

Type of Interleaver	Interleaver Function	Description
General block interleaver	`intrlv`	Uses the permutation table given explicitly as an input argument.
Algebraic interleaver	`algintrlv`	Derives a permutation table algebraically, using the Takeshita-Costello or Welch-Costas method. These methods are described in [4].
Helical scan interleaver	`helscanintrlv`	Fills a matrix with data row by row and then sends the matrix contents to the output in a helical fashion.

Type of Interleaver	Interleaver Function	Description
Matrix interleaver	`matintrlv`	Fills a matrix with data elements row by row and then sends the matrix contents to the output column by column.
Random interleaver	`randintrlv`	Chooses a permutation table randomly using the initial state input that you provide.

Types of Block Interleavers

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

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

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

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

Improve Error Rate Using Block Interleaving in MATLAB

The following example illustrates how an interleaver improves the error rate in a communication system whose channel produces a burst of errors. A random interleaver rearranges the bits of numerous codewords before two adjacent codewords are each corrupted by three errors.

Three errors exceed the error-correction capability of the Hamming code. However, the example shows that when the Hamming code is combined with an interleaver, this system is able to recover the original message despite the 6-bit burst of errors. The improvement in performance occurs because the interleaving effectively spreads the errors among different codewords so that the number of errors per codeword is within the error-correction capability of the code.

```
st1 = 27221; st2 = 4831; % States for random number generator
n = 7; k = 4; % Parameters for Hamming code
msg = randi([0 1],k*500,1); % Data to encode
code = encode(msg,n,k,'hamming/binary'); % Encoded data
% Create a burst error that will corrupt two adjacent codewords.
errors = zeros(size(code)); errors(n-2:n+3) = [1 1 1 1 1 1];

% With Interleaving
%------------------
inter = randintrlv(code,st2); % Interleave.
inter_err = bitxor(inter,errors); % Include burst error.
deinter = randdeintrlv(inter_err,st2); % Deinterleave.
decoded = decode(deinter,n,k,'hamming/binary'); % Decode.
```

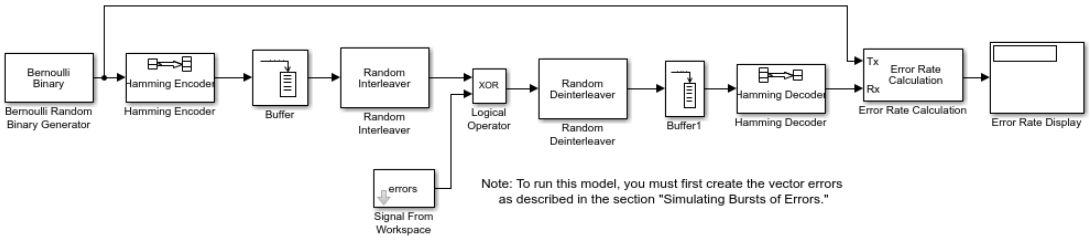
```
disp('Number of errors and error rate, with interleaving:');
[number_with,rate_with] = biterr(msg,decoded) % Error statistics

% Without Interleaving
%---------------------
code_err = bitxor(code,errors); % Include burst error.
decoded = decode(code_err,n,k,'hamming/binary'); % Decode.
disp('Number of errors and error rate, without interleaving:');
[number_without,rate_without] = biterr(msg,decoded) % Error statistics
```

The output from the example follows.

```
Number of errors and error rate, with interleaving:

number_with =

     0


rate_with =

     0

Number of errors and error rate, without interleaving:

number_without =

     4


rate_without =

    0.0020
```

Improve Error Rate Using Block Interleaving in Simulink

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



Before running the model, you must create a binary vector that simulates bursts of errors, as described in “Improve Error Rate Using Block Interleaving in Simulink” on page 16-118. The Signal From Workspace block imports this vector from the MATLAB workspace into the model, where the Logical Operator block performs an XOR of the vector with the signal.

To open the completed model, enter `doc_interleaver` at the MATLAB command line. To build the model, gather and configure these blocks:

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

On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `length(errors)`. The **Simulate** section appears on multiple tabs.

Creating the Vector of Errors

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

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

```
errors(n:n+2)=[1 1 1];
end
```

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

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

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

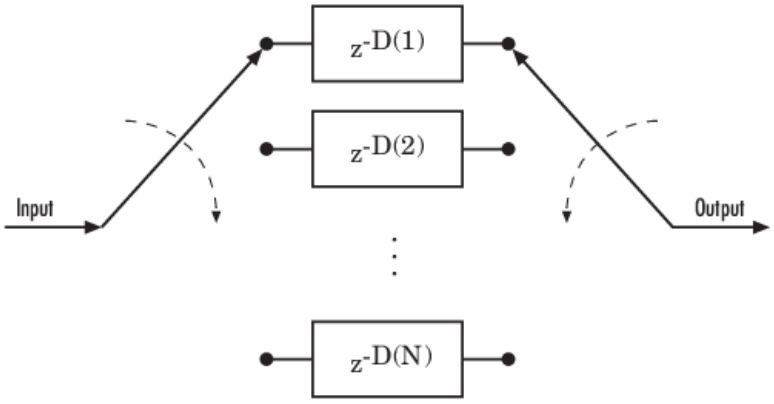
When you run a simulation with the model, the error rate is approximately .019, which shows the improvement due to error correction and interleaving. You can see the effect of interleaving by deleting the Random Interleaver and Random Deinterleaver blocks from the model, connecting the lines, and running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

Convolutional Interleaving

Convolutional Interleaving Features

A convolutional interleaver consists of a set of shift registers, each with a fixed delay. In a typical convolutional interleaver, the delays are nonnegative integer multiples of a fixed integer (although a general multiplexed interleaver allows unrestricted delay values). Each new symbol from an input vector feeds into the next shift register and the oldest symbol in that register becomes part of the output vector. A convolutional interleaver has memory; that is, its operation depends not only on current symbols but also on previous symbols.

The schematic below depicts the structure of a general convolutional interleaver by showing the set of shift registers and their delay values D(1), D(2),..., D(N). The kth shift register holds D(k) symbols, where k = 1,2,...,N. The convolutional interleaving functions in this toolbox have input arguments that indicate the number of shift registers and the delay for each shift register.



Communications Toolbox implements convolutional interleaving functionality using Simulink blocks, System objects, and MATLAB functions.

The set of convolutional interleavers in this product includes a general interleaver/deinterleaver pair as well as several special cases. Each special-case function uses the same computational code that its more general counterpart uses, but provides a syntax that is more suitable for the special case. The special cases are described below.

Type of Interleaver	Interleaving Function	Description
General multiplexed interleaver	`muxintrlv`	Allows unrestricted delay values for the set of shift registers.
Convolutional interleaver	`convintrlv`	The delay values for the set of shift registers are nonnegative integer multiples of a fixed integer that you specify.
Helical interleaver	`helintrlv`	Fills an array with input symbols in a helical fashion and empties the array row by row.

The `helscanintrlv` function and the `helintrlv` function both use a helical array for internal computations. However, the two functions have some important differences:

- `helintrlv` uses an unlimited-row array, arranges input symbols in the array along columns, outputs some symbols that are not from the current input, and leaves some input symbols in the array without placing them in the output.
- `helscanintrlv` uses a fixed-size matrix, arranges input symbols in the array across rows, and outputs all the input symbols without using any default values or values from a previous call.

Types of Convolutional Interleavers

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

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

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

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

Delays of Convolutional Interleavers

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

measured in symbols, between the original and restored sequences is indicated in the table below. The variable names in the second column (`delay`, `nrows`, `slope`, `col`, `ngrp`, and `stp`) refer to the inputs named on each function's reference page.

Delays of Interleaver/Deinterleaver Pairs

Interleaver/Deinterleaver Pair	Delay Between Original and Restored Sequences
`muxintrlv`, `muxdeintrlv`	`length(delay)*max(delay)`
`convintrlv`, `convdeintrlv`	`nrows*(nrows-1)*slope`
`helintrlv`, `heldeintrlv`	`col*ngrp*ceil(stp*(col-1)/ngrp)`

Delays of Convolutional Interleavers

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

Number of shift registers × Maximum delay among all shift registers

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

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

Convolutional Interleaver block

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

B × (N-1)

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

Helical Interleaver block

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

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

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

Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB

If you use a convolutional interleaver followed by a corresponding convolutional deinterleaver, then a nonzero delay means that the recovered data (that is, the output from the deinterleaver) is not the same as the original data (that is, the input to the interleaver). If you compare the two data sets

directly, then you must take the delay into account by using appropriate truncating or padding operations.

Here are some typical ways to compensate for a delay of D in an interleaver/deinterleaver pair:

- Interleave a version of the original data that is padded with D extra symbols at the end. Before comparing the original data with the recovered data, omit the first D symbols of the recovered data. In this approach, all the original symbols appear in the recovered data.
- Before comparing the original data with the recovered data, omit the last D symbols of the original data and the first D symbols of the recovered data. In this approach, some of the original symbols are left in the deinterleaver's shift registers and do not appear in the recovered data.

The following code illustrates these approaches by computing a symbol error rate for the interleaving/deinterleaving operation.

```
x = randi([0 63],20,1); % Original data
nrows = 3; slope = 2; % Interleaver parameters
D = nrows*(nrows-1)*slope; % Delay of interleaver/deinterleaver pair
hInt   = comm.ConvolutionalInterleaver('NumRegisters', nrows, ...
                    'RegisterLengthStep', slope);
hDeint = comm.ConvolutionalDeinterleaver('NumRegisters', nrows, ...
                    'RegisterLengthStep', slope);

% First approach.
x_padded = [x; zeros(D,1)]; % Pad x at the end before interleaving.
a1 = step(hInt, x_padded); % Interleave padded data.

b1 = step(hDeint, a1)
% Omit input padding and the first D symbols of the recovered data and
% compare
servec1 = step(comm.ErrorRate('ReceiveDelay',D),x_padded,b1);
ser1 = servec1(1)

% Second approach.
release(hInt); release(hDeint)
a2 = step(hInt,x); % Interleave original data.
b2 = step(hDeint,a2)
% Omit the last D symbols of the original data and the first D symbols of
% the recovered data and compare.
servec2 = step(comm.ErrorRate('ReceiveDelay',D),x,b2);
ser2 = servec2(1)
```

The output is shown below. The zero values of `ser1` and `ser2` indicate that the script correctly aligned the original and recovered data before computing the symbol error rates. However, notice from the lengths of `b1` and `b2` that the two approaches to alignment result in different amounts of deinterleaved data.

```
b1 =

     0
     0
     0
     0
     0
     0
     0
     0
```

```
     0
     0
     0
     0
    59
    42
     1
    28
    52
    54
    43
     8
    56
     5
    35
    37
    48
    17
    28
    62
    10
    31
    61
    39


ser1 =

     0


b2 =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
    59
    42
     1
    28
    52
    54
    43
     8


ser2 =

     0
```

Combining Interleaving Delays and Other Delays

If you use convolutional interleavers in a script that incurs an additional delay, d, between the interleaver output and the deinterleaver input (for example, a delay from a filter), then the restored sequence lags behind the original sequence by the sum of d and the amount from the table Delays of Interleaver/Deinterleaver Pairs on page 16-122. In this case, d must be an integer multiple of the number of shift registers, or else the convolutional deinterleaver cannot recover the original symbols properly. If d is not naturally an integer multiple of the number of shift registers, then you can adjust the delay manually by padding the vector that forms the input to the deinterleaver.

Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB

The example below illustrates convolutional interleaving and deinterleaving using a sequence of consecutive integers. It also illustrates the inherent delay of the interleaver/deinterleaver pair.

```
x = [1:10]'; % Original data
delay = [0; 1; 2]; % Set delays of three shift registers.
hInt = comm.MultiplexedInterleaver('Delay', delay);
hDeint = comm.MultiplexedDeinterleaver('Delay', delay);
y = step(hInt,x) % Interleave.
z = step(hDeint,y) % Deinterleave.
```

In this example, the `muxintrlv` function initializes the three shift registers to the values `[]`, `[0]`, and `[0 0]`, respectively. Then the function processes the input data `[1:10]'`, performing internal computations as indicated in the table below.

Current Input	Current Shift Register	Current Output	Contents of Shift Registers
1	1	1	[] [0] [0 0]
2	2	0	[] [2] [0 0]
3	3	0	[] [2] [0 3]
4	1	4	[] [2] [0 3]
5	2	2	[] [5] [0 3]
6	3	0	[] [5] [3 6]
7	1	7	[] [5] [3 6]

Current Input	Current Shift Register	Current Output	Contents of Shift Registers
8	2	5	[] [8] [3 6]
9	3	3	[] [8] [6 9]
10	1	10	[] [8] [6 9]

The output from the example is below.

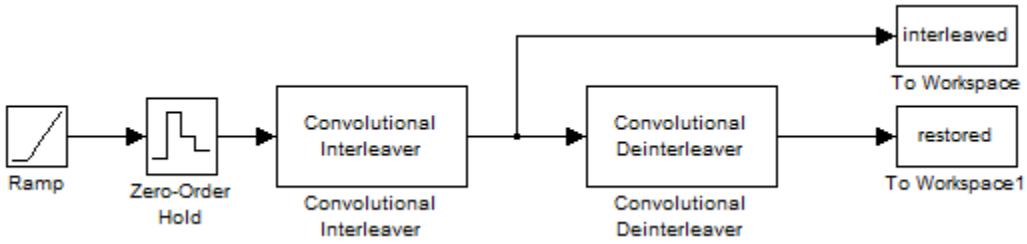
```
y =

     1
     0
     0
     4
     2
     0
     7
     5
     3
    10


state_y =

    value: {3x1 cell}
    index: 2


z =

     0
     0
     0
     0
     0
     0
     1
     2
     3
     4
```

Notice that the “Current Output” column of the table above agrees with the values in the vector `y`. Also, the last row of the table above indicates that the last shift register processed for the given data set is the first shift register. This agrees with the value of 2 for `state_y.index`, which indicates that any additional input data would be directed to the second shift register. You can optionally check that the state values listed in `state_y.value` match the “Contents of Shift Registers” entry in the last row of the table by typing `state_y.value{:}` in the Command Window after executing the example.

Another feature to notice about the example output is that `z` contains six zeros at the beginning before containing any of the symbols from the original data set. The six zeros illustrate that the delay of this convolutional interleaver/deinterleaver pair is `length(delay)*max(delay) = 3*2 = 6`. For more information about delays, see “Delays of Convolutional Interleavers” on page 16-121.

Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in Simulink

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



To open the model, enter `doc_convinterleaver` at the MATLAB command line. To build the model, gather and configure these blocks:

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

Connect the blocks as shown in the preceding figure. On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `20`. The **Simulate** section appears on multiple tabs. Run the simulation and execute the following command:

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

comparison =

     0     0    -1
     1    -2    -2
     2    -3    -3
     3     3    -1
     4    -2    -2
```

```
 5    -3    -3
 6     6    -1
 7     1    -2
 8    -3    -3
 9     9    -1
10     4    -2
11    -3    -3
12    12     0
13     7     1
14     2     2
15    15     3
16    10     4
17     5     5
18    18     6
19    13     7
20     8     8
```

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

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

For a similar example that also indicates the contents of the shift registers at each step of the process, see “Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB” on page 16-125.

Selected Bibliography for Interleaving

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

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

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

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

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

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

Digital Modulation

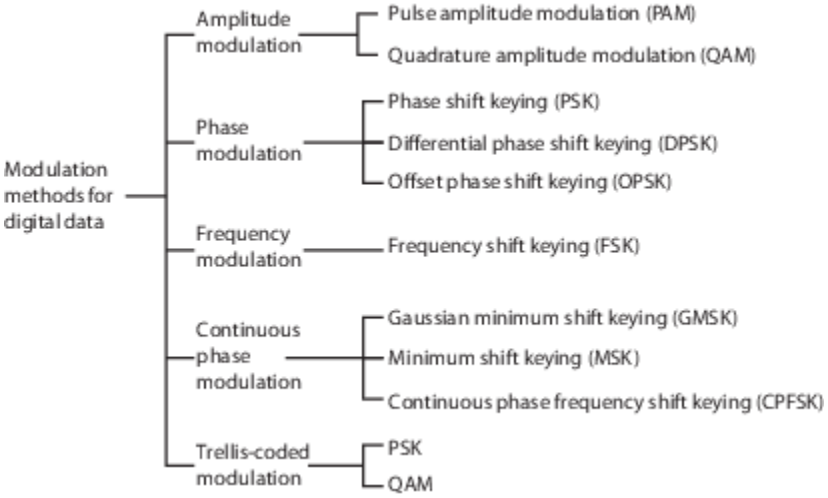
In most media for communication, only a fixed range of frequencies is available for transmission. One way to communicate a message signal whose frequency spectrum does not fall within that fixed frequency range, or one that is otherwise unsuitable for the channel, is to alter a transmittable signal according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

In this section...

Digital Modulation Features

Modulation Techniques

The Communications Toolbox supports these modulation techniques for digital data. All the methods at the far right are implemented in library blocks.



Like analog modulation, digital modulation alters a transmittable signal according to the information in a message signal. However, for digital modulation the message signal is restricted to a finite set. Modulation functions output the complex envelope of the modulated signal. Communications Toolbox features enable you to modulate and demodulate signals using various digital modulation techniques. Constellation plots enable you to visualize the constellation diagram of the modulation symbols.

Note Unless otherwise indicated, the modulation and demodulation functions do not perform pulse shaping or filtering. See Combining Pulse Shaping and Filtering with Modulation on page 16-142 for more information about filtering.

The available methods of modulation depend on whether the input signal is analog or digital. The “Modulation” category lists the digital and analog modulation techniques supported.

Accessing Digital Modulation Blocks

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

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

Kind of Modulation	Icon in Digital Baseband Library
Amplitude modulation	AM
Phase modulation	PM
Frequency modulation	FM
Continuous phase modulation	CPM
Trellis-coded modulation	TCM

Some digital modulation sublibraries contain blocks that implement specific modulation techniques. These specific-case modulation blocks use the same computational code that their general counterparts use, but provide an interface that is either simpler or more suitable for the specific case. This table lists general modulators along with the conditions under which is general modulator is equivalent to a specific modulator. The situation is analogous for demodulators.

General and Specific Blocks

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

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

Baseband and Passband Simulation

For a given modulation technique, two ways to simulate modulation techniques are called *baseband* and *passband*. Baseband simulation, also known as the *lowpass equivalent method*, requires less computation. The Communications Toolbox supports baseband simulation for digital modulation and passband simulation for analog modulation.

Baseband Modulated Signals Defined

If you use baseband modulation to produce the complex envelope y of the modulation of a message signal x, then y is a *complex-valued* signal that is related to the output of a passband modulator. If the modulated signal has the waveform

$$Y_1(t)\sqrt{2}\cos(2\pi f_c t + \theta) - Y_2(t)\sqrt{2}\sin(2\pi f_c t + \theta)\,,$$

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

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

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

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

If you prefer to work with passband signals instead of baseband signals, then you can build functions that convert between the two. Be aware that passband modulation tends to be more computationally intensive than baseband modulation because the carrier signal typically needs to be sampled at a high rate.

Modulation Terminology

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

The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. The Nyquist sampling theorem requires that the simulation sampling rate, F_s, be greater than two times the sum of the carrier frequency and the highest frequency of the modulated signal in order for the demodulator to recover the message correctly.

Representing Digital Signals

To modulate a signal using digital modulation with an alphabet having M symbols, start with a real message signal whose values are integers from 0 to M-1. Represent the signal by listing its values in a vector, `x`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

For example, if the modulation uses an alphabet with eight symbols, then the vector `[2 3 7 1 0 5 5 2 6]'` is a valid single-channel input to the modulator. As a multichannel example, the two-column matrix

```
[2 3;
 3 3;
 7 3;
 0 3;]
```

defines a two-channel signal in which the second channel has a constant value of `3`.

Signals and Delays

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

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

Integer-Valued Signals and Binary-Valued Signals

Some digital modulation blocks can accept either integer-valued or binary–valued signals. The corresponding demodulation blocks can output either integers or groups of individual bits that represent integers. This section describes how modulation blocks process integer or binary inputs; the case for demodulation blocks is the reverse. You should note that modulation blocks have an **Input type** parameter and that demodulation blocks have an **Output type** parameter.

When you set the **Input type** parameter to `Integer`, the block accepts integer values from `0` to 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. Since $M = 2^K$, K is commonly referred to as the modulation order.

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.

Constellation Ordering (or Symbol Set Ordering)

Depending on the modulation scheme, the **Constellation ordering** or **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 $M = 8$, **Constellation ordering** (or **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 **Constellation ordering** (or **Symbol set ordering** or **Symbol mapping**) 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

Gray to Binary Mapping for Integers

Binary Code	Gray Code
0	0
1	1
2	3
3	2
4	6
5	7
6	5
7	4

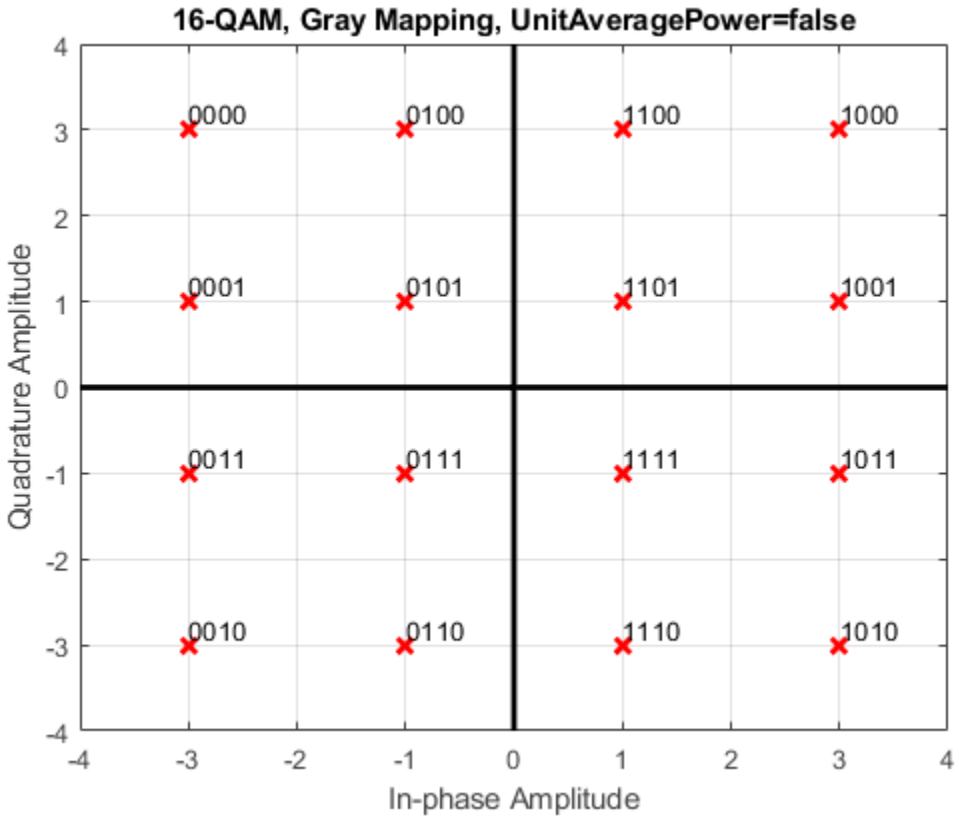
Gray Encoding a Modulated Signal

For the PSK, DPSK, FSK, QAM, and PAM modulation types, Gray constellations are obtained by setting the symbol mapping to Gray-encoding in the corresponding modulation function or System object®.

For modulation functions, you can specify 'gray' for the symbol order input argument to obtain Gray-encoded modulation.

The following example uses the `qammod` function with Gray-encoded symbol mapping.

```
y = [0:15];
y = de2bi(y);
M = 16;
symorder = 'gray';
xmap = qammod(y,M,symorder,'InputType','bit','PlotConstellation',true);
```



Checking the constellation plot, you can see the modulated symbols are Gray-encoded because all adjacent elements differ by only one bit.

Upsample Signals and Rate Changes

Some digital modulation blocks can output an upsampled version of the modulated signal, while their corresponding digital demodulation blocks can accept an upsampled version of the modulated signal as input. In both cases, the **Rate options** parameter represents the upsampling factor, which must be a positive integer. Depending on whether the input signal is single-rate mode or multirate mode, the block either changes the signal's vector size or its sample time, as the following table indicates. Only the OQPSK blocks deviate from the information in the table, in that S is replaced by 2S in the scaling factors.

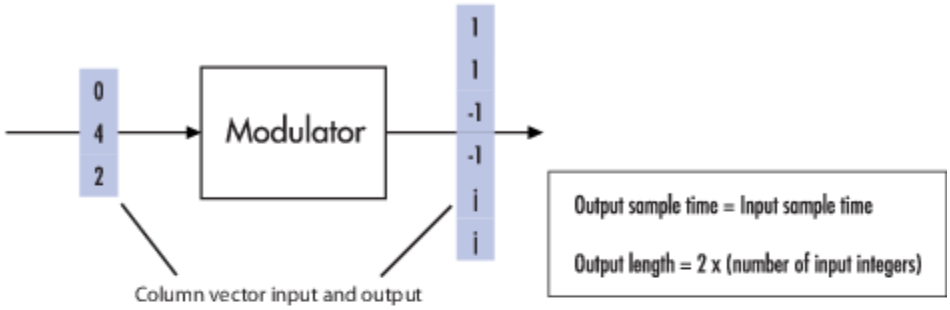
Process Upsampled Modulated Data (Except OQPSK Method)

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

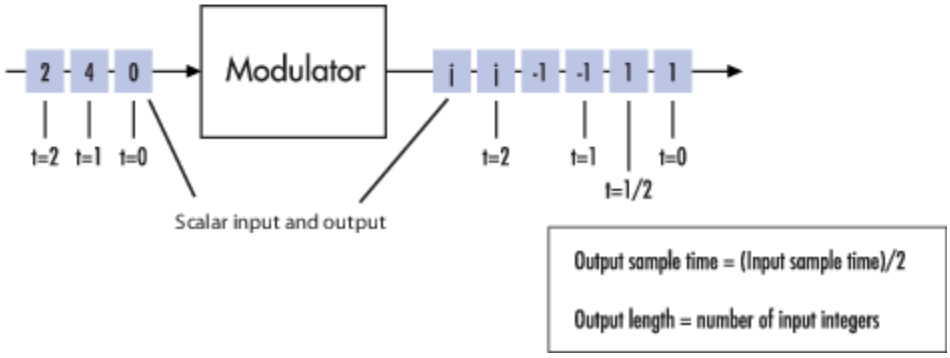
Illustrations of Size or Rate Changes

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

Upsample Output: Single-Rate Processing

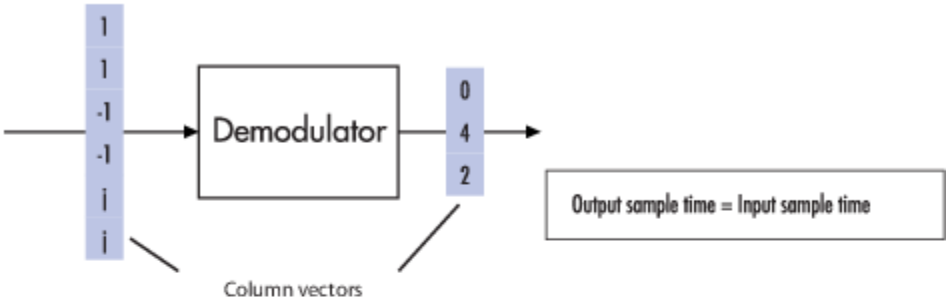


Upsample Output: Multirate Processing

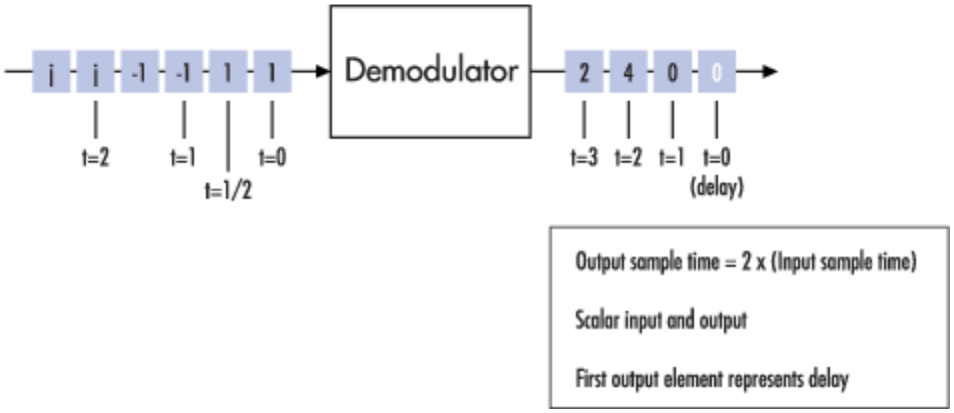


The following schematics illustrate how a demodulator (other than OQPSK or one from the CPM sublibrary) processes three doubly sampled symbols using both frame-based and sample-based inputs. In both cases, the **Samples per symbol** parameter is 2. The sample-based schematic includes an output delay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

Upsampled Input: Single Rate Processing



Upsampled Input: Multirate Processing

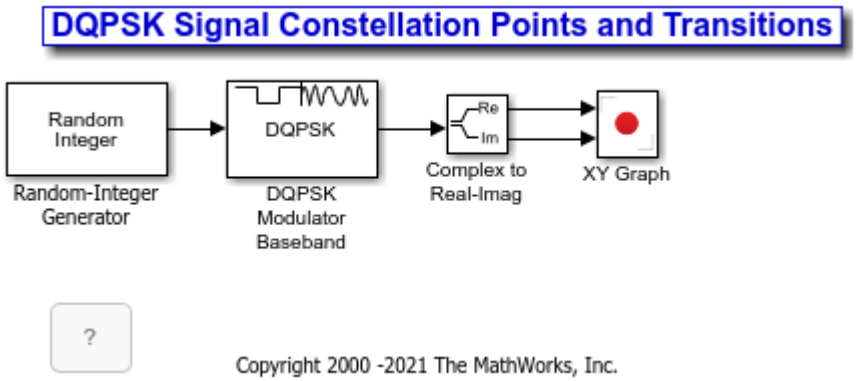


For more information on delays, see Delays in Digital Modulation on page 16-146.

PM Modulation

DQPSK Signal Constellation Points and Transitions

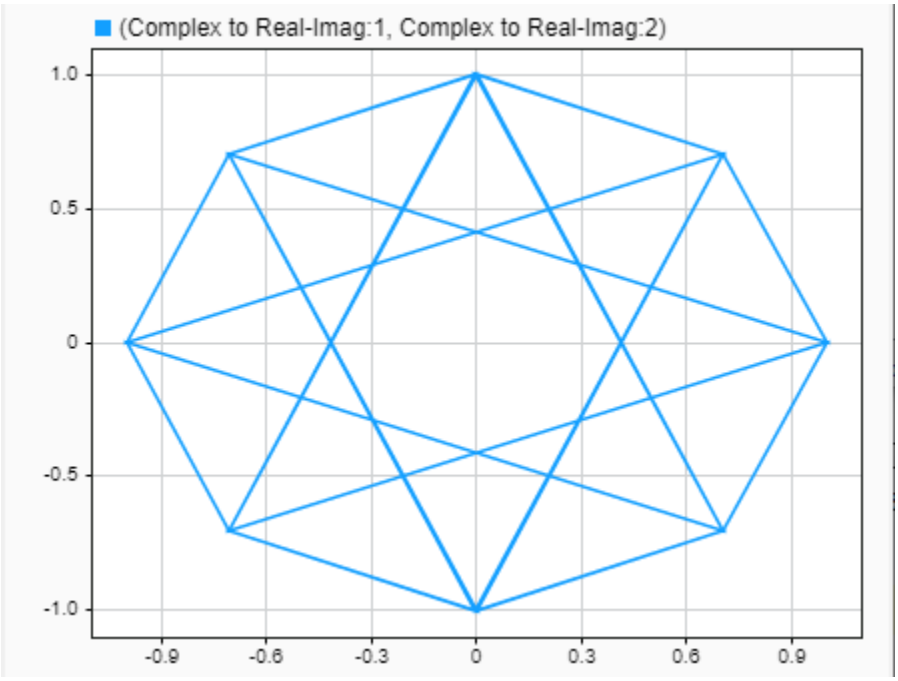
This model plots the output of the DQPSK Modulator Baseband block. The image shows the possible transitions from each symbol in the DQPSK signal constellation to the next symbol.



To build the model, gather and configure these blocks:

- Random Integer Generator
- DQPSK Modulator Baseband
- Complex to Real-Imag (Simulink)
- XY Graph (for more information, see “Visualize Simulation Data on an XY Plot” (Simulink))

For the Random Integer Generator block, set the M-ary number to 4, set the initial seed to any positive integer scalar (to randomize results you can use the output of the `randn` function), and set the sample time to .01.



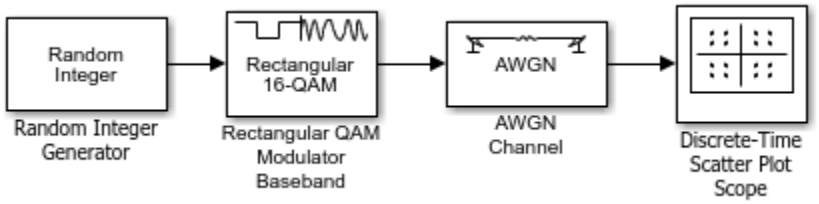
The plot illustrates pi/4-DQPSK modulation, because the default Phase offset parameter in the DQPSK Modulator Baseband block is pi/4. To see how the phase offset influences the signal constellation,

change the Phase offset parameter in the DQPSK Modulator Baseband block to pi/8 or another value. Run the model again and observe how the plot changes.

AM Modulation

Rectangular QAM Modulation and Scatter Diagram

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

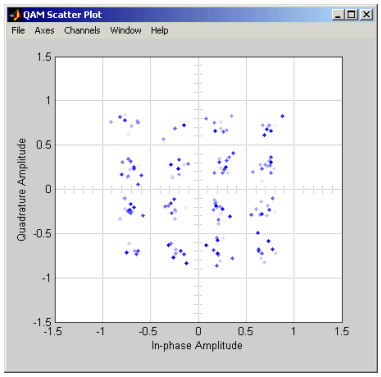


To open this model, enter `doc_qam_scatter` at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `16`.
  - Set **Initial seed** to any positive integer scalar, preferably the output of the `randn` function.
  - Set **Sample time** to `.1`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation
  - Set **Normalization method** to `Peak Power`.
- AWGN Channel, in the Channels library
  - Set **Es/No** to `20`.
  - Set **Symbol period** to `.1`.
- Constellation Diagram, in the Comm Sinks library
  - Set **Symbols to display** to `160`.

Connect the blocks as shown in the preceding figure. On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `250`. The **Simulate** section appears on multiple tabs.

Running the model produces a scatter diagram like the following one. Your plot might look somewhat different, depending on your **Initial seed** value in the Random Integer Generator block. Because the modulation technique is 16-QAM, the plot shows 16 clusters of points. If there were no noise, the plot would show the 16 exact constellation points instead of clusters around the constellation points.



Compute Symbol Error Rate

The example generates a random digital signal, modulates it, adds noise, demodulates the noisy signal, and computes the symbol error rate. The noisy, modulated data is plotted in a constellation diagram. Numerical results and plot may vary due to the random input data.

Create a random digital message and a constellation diagram System object.

```
M = 16; % Alphabet size, 16-QAM
x = randi([0 M-1],5000,1);

cpts = qammod(0:M-1,M);
constDiag = comm.ConstellationDiagram('ReferenceConstellation',cpts, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Apply 16-QAM modulation and transmit signal through an AWGN channel.

```
y = qammod(x,M);
ynoisy = awgn(y,15,'measured');
```

Demodulate ynoisy to recover the message and check the symbol error rate.

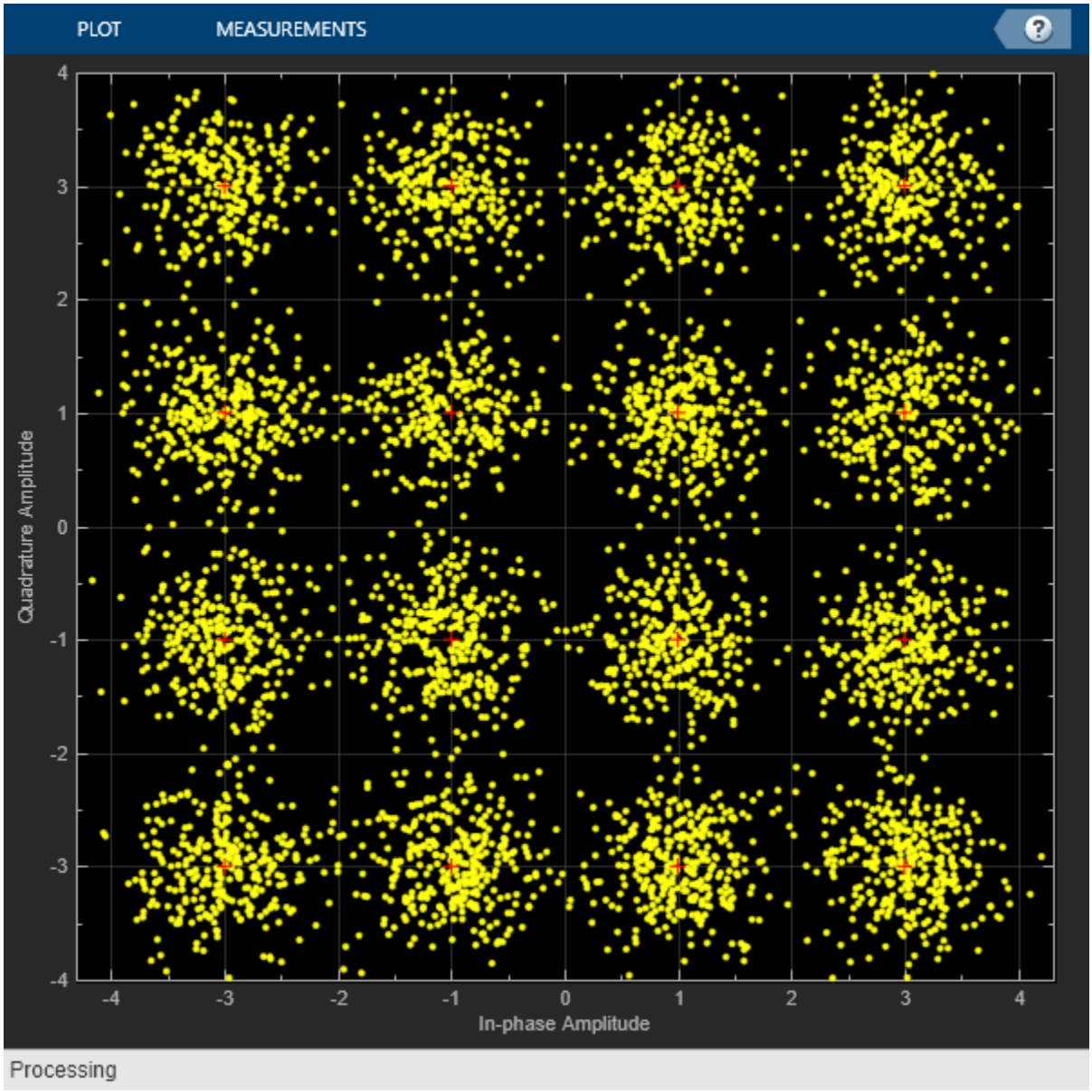
```
z = qamdemod(ynoisy,M);
[num,rt] = symerr(x,z)
```

```
num = 79
```

```
rt = 0.0158
```

Create constellation diagram from noisy data. The signal reference constellation has 16 precisely located points but the transmitted symbols with the noise added causes the scatter plot to have a small cluster of points scattered around each reference constellation point.

```
constDiag(ynoisy)
```



Combine Pulse Shaping and Filtering with Modulation

Modulation is often followed by pulse shaping, and demodulation is often preceded by a filtering or an integrate-and-dump operation. This section presents an example involving rectangular pulse shaping. For an example that uses raised cosine pulse shaping, see “Pulse Shaping Using a Raised Cosine Filter” on page 24-6.

Rectangular Pulse Shaping

Rectangular pulse shaping repeats each output from the modulator a fixed number of times to create an upsampled signal. Although it is less realistic than other kinds of pulse shaping, rectangular pulse shaping can be a first step or an exploratory step in algorithm development. If the transmitter upsamples the modulated signal, then the receiver should downsample the received signal before demodulating. The code below uses the `rectpulse` function for rectangular pulse shaping at the

transmitter and the `intdump` function for downsampling at the receiver. The “integrate and dump” operation is one way to downsample the received signal.

```
% Create a random digital message and a constellation diagram System
% object.
M = 16;                     % Alphabet size, 16-QAM
x = randi([0 M-1],5000,1);  % Message signal
Nsamp = 4;                  % Oversampling rate

% Apply 16-QAM modulation and rectangular pulse shaping. Transmit signal
% through an AWGN channel.
y = qammod(x,M);
ypulse = rectpulse(y,Nsamp);
ynoisy = awgn(ypulse,15,'measured');

% Downsample at the receiver.
ydownsamp = intdump(ynoisy,Nsamp);

% Demodulate to recover the message.
z = qamdemod(ydownsamp,M);
```

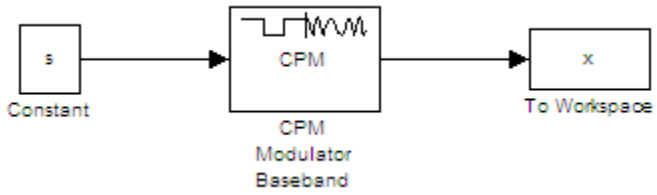
CPM Modulation

Phase Tree for Continuous Phase Modulation

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

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

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



To open the model, enter `doc_phasetree` at the MATLAB command line. To build the model, gather and configure these blocks:

- Constant, in the Simulink Commonly Used Blocks library
  - Set **Constant value** to `s` (which will appear in the MATLAB workspace).

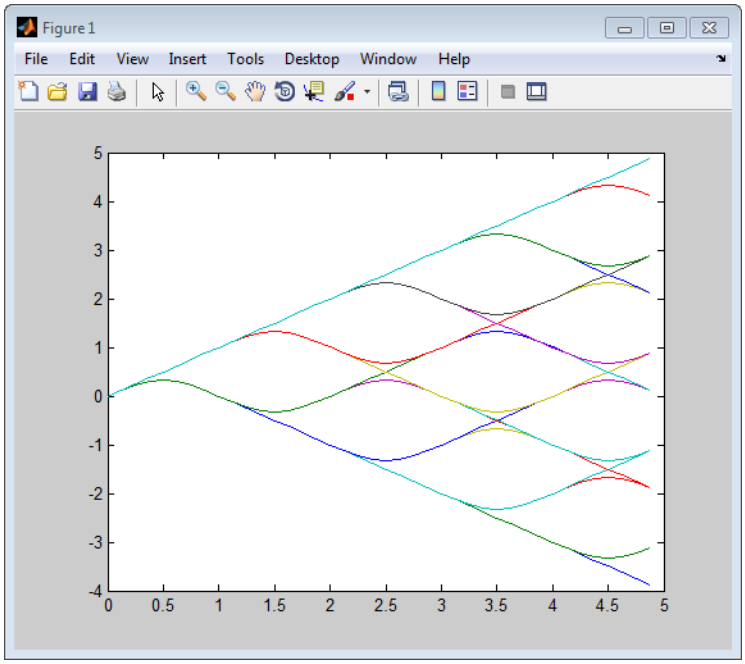
  - Set **Sampling mode** to `Frame-based`.
  - Set **Frame period** to `1`.
- CPM Modulator Baseband
  - Set **M-ary number** to `2`.
  - Set **Modulation index** to `2/3`.
  - Set **Frequency pulse shape** to `Raised Cosine`.
  - Set **Pulse length** to `2`.
- To Workspace, in the Simulink Sinks library
  - Set **Variable name** to `x`.
  - Set **Save format** to `Array`.

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

The second step of this example is to execute the following MATLAB code:

```
% Parameters from the CPM Modulator Baseband block
M_ary_number = 2;
modulation_index = 2/3;
pulse_length = 2;
samples_per_symbol = 8;

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

This code defines the parameters for the CPM Modulator, applies symbol mapping, and plots the results. Each curve represents a different instance of simulating the CPM Modulator Baseband block with a distinct (constant) input signal.



Exact LLR Algorithm

The log-likelihood ratio (LLR) is the logarithm of the ratio of probabilities of a 0 bit being transmitted versus a 1 bit being transmitted for a received signal. The LLR for a bit b is defined as:

$$L(b) = \log\left(\frac{\Pr(b = 0 \,|\, r = (x, y))}{\Pr(b = 1 \,|\, r = (x, y))}\right)$$

Assuming equal probability for all symbols, the LLR for an AWGN channel can be expressed as:

$$L(b) = \log\left(\frac{\sum_{s \in S_0} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}{\sum_{s \in S_1} e^{-\frac{1}{\sigma^2}\left((x - s_x)^2 + (y - s_y)^2\right)}}\right)$$

where the variables represent the values shown in the following table.

Variable	What the Variable Represents
r	Received signal with coordinates (x, y).
b	Transmitted bit (one of the K bits in an M-ary symbol, assuming all M symbols are equally probable.
S_0	Ideal symbols or constellation points with bit 0, at the given bit position.
S_1	Ideal symbols or constellation points with bit 1, at the given bit position.
s_x	In-phase coordinate of ideal symbol or constellation point.
s_y	Quadrature coordinate of ideal symbol or constellation point.

Variable	What the Variable Represents
σ^2	Noise variance of baseband signal.
σ_x^2	Noise variance along in-phase axis.
σ_y^2	Noise variance along quadrature axis.

Note Noise components along the in-phase and quadrature axes are assumed to be independent and of equal power (i.e., $\sigma_x^2 = \sigma_y^2 = \sigma^2/2$).

Approximate LLR Algorithm

Approximate LLR is computed by taking into consideration only the nearest constellation point to the received signal with a 0 (or 1) at that bit position, rather than all the constellation points as done in exact LLR. It is defined as [8]:

$$L(b) = -\frac{1}{\sigma^2}\left(\min_{s \in S_0}\left((x - s_x)^2 + (y - s_y)^2\right) - \min_{s \in S_1}\left((x - s_x)^2 + (y - s_y)^2\right)\right)$$

Delays in Digital Modulation

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

Delays Resulting from Digital Modulation or Demodulation

Modulation or Demodulation Type	Situation in Which Delay Occurs	Amount of Delay
FM demodulator	Sample-based processing	One output period
All demodulators in CPM sublibrary	Multirate processing, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking` D = **Traceback length** parameter	D+1 output periods
	Single-rate processing, D = **Traceback depth** parameter	D output periods
OQPSK demodulator	Single-rate processing	For more information, see OQPSK Demodulator Baseband.
	Multirate processing, and the model uses a fixed-step solver with **Tasking Mode** parameter set to `Auto` or `MultiTasking`.	
	Multirate processing processing, and the model uses a variable-step solver or the **Tasking Mode** parameter is set to Single`Tasking`.	
All decoders in TCM sublibrary	**Operation mode** set to `Continuous`, Tr = **Traceback depth** parameter, and code rate k/n	Tr*k output bits

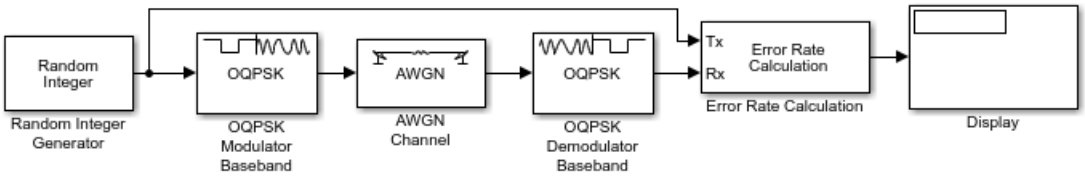
As a result of delays, data that enters a modulation or demodulation block at time T appears in the output at time T+delay. In particular, if your simulation computes error statistics or compares transmitted with received data, it must take the delay into account when performing such computations or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

First Output Sample in DPSK Demodulation

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

Example: Delays from Demodulation

Demodulation in the model below causes the demodulated signal to lag, compared to the unmodulated signal. When computing error statistics, the model accounts for the delay by setting the Error Rate Calculation block's **Receive delay** parameter to `0`. If the **Receive delay** parameter had a different value, then the error rate showing at the top of the Display block would be close to 1/2.



To open this model, enter `doc_oqpsk_modulation_delay` at the MATLAB command line. To build the model, gather and configure these blocks:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library
  - Set **M-ary number** to `4`.
  - Set **Initial seed** to any positive integer scalar.
- OQPSK Modulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- AWGN Channel, in the Channels library
  - Set **Es/No** to `6`.
- OQPSK Demodulator Baseband, in the PM sublibrary of the Digital Baseband sublibrary of Modulation
- Error Rate Calculation, in the Comm Sinks library
  - Set **Receive delay** to `1`.
  - Set **Computation delay** to `0`.
  - Set **Output data** to `Port`.
- Display, in the Simulink Sinks library
  - Drag the bottom edge of the icon to make the display big enough for three entries.

Connect the blocks as shown in the preceding figure. On the **Simulation** tab, in the **Simulate** section, set **Stop time** to `1000`. The **Simulate** section appears on multiple tabs. Then run the model and observe the error rate at the top of the Display block's icon. Your error rate will vary depending on your **Initial seed** value in the Random Integer Generator block.

Selected Bibliography for Digital Modulation

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

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

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

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

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

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

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

[8] Viterbi, A. J., "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, No. 2, pp 260–264, Feb. 1998

Analog Passband Modulation

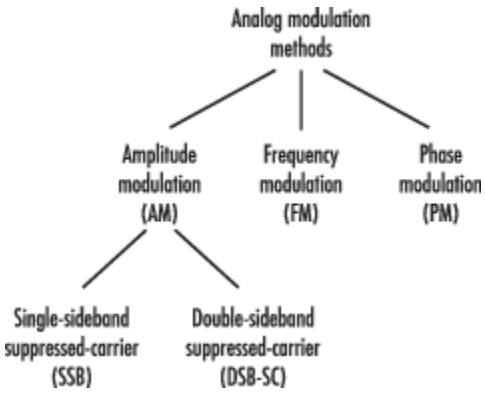
In this section...

Analog Modulation Features

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

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

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



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

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

Represent Signals for Analog Modulation

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

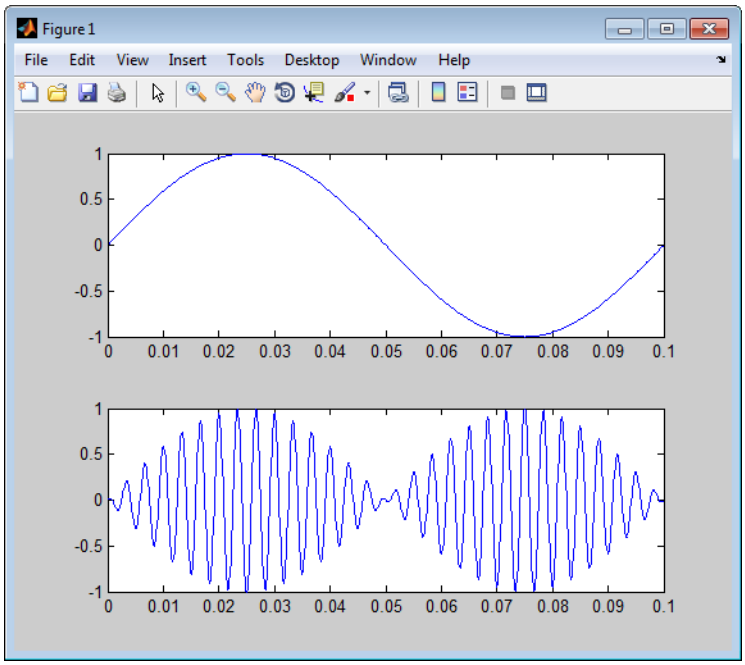
All analog demodulators in this product produce discrete-time, not continuous-time, output.

Representing Analog Signals Using MATLAB

To modulate an analog signal using MATLAB, start with a real message signal and a sampling rate `Fs` in hertz. Represent the signal using a vector `x`, the entries of which give the signal's values in time increments of 1/`Fs`. Alternatively, you can use a matrix to represent a multichannel signal, where each column of the matrix represents one channel.

For example, if `t` measures time in seconds, then the vector `x` below is the result of sampling a sine wave 8000 times per second for 0.1 seconds. The vector `y` represents the modulated signal.

```
Fs = 8000; % Sampling rate is 8000 samples per second.
Fc = 300; % Carrier frequency in Hz
t = [0:.1*Fs]'/Fs; % Sampling times for .1 second
x = sin(20*pi*t); % Representation of the signal
y = ammod(x,Fc,Fs); % Modulate x to produce y.
figure;
subplot(2,1,1); plot(t,x); % Plot x on top.
subplot(2,1,2); plot(t,y)% Plot y below.
```



As a multichannel example, the code below defines a two-channel signal in which one channel is a sinusoid with zero initial phase and the second channel is a sinusoid with an initial phase of `pi/8`.

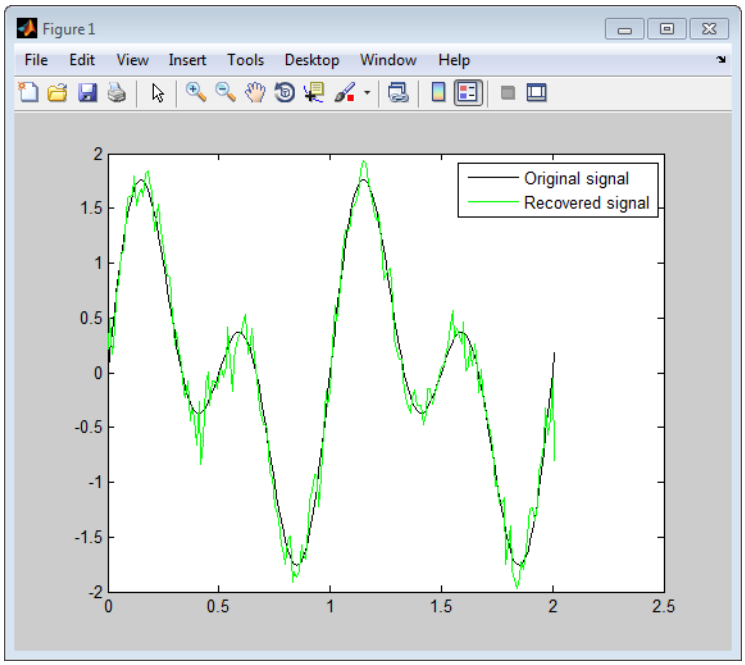
```
Fs = 8000;
t = [0:.1*Fs]'/Fs;
x = [sin(20*pi*t), sin(20*pi*t+pi/8)];
```

Analog Modulation with Additive White Gaussian Noise (AWGN) Using MATLAB

This example illustrates the basic format of the analog modulation and demodulation functions. Although the example uses phase modulation, most elements of this example apply to other analog modulation techniques as well.

The example samples an analog signal and modulates it. Then it simulates an additive white Gaussian noise (AWGN) channel, demodulates the received signal, and plots the original and demodulated signals.

```
% Prepare to sample a signal for two seconds,
% at a rate of 100 samples per second.
Fs = 100; % Sampling rate
t = [0:2*Fs+1]'/Fs; % Time points for sampling

% Create the signal, a sum of sinusoids.
x = sin(2*pi*t) + sin(4*pi*t);

Fc = 10; % Carrier frequency in modulation
phasedev = pi/2; % Phase deviation for phase modulation

y = pmmod(x,Fc,Fs,phasedev); % Modulate.
y = awgn(y,10,'measured',103); % Add noise.
z = pmdemod(y,Fc,Fs,phasedev); % Demodulate.

% Plot the original and recovered signals.
figure; plot(t,x,'k-',t,z,'g-');
legend('Original signal','Recovered signal');
```



Other examples using analog modulation functions appear in the reference pages for `ammod`, `amdemod`, `ssbdemod`, and `fmmod`.

Sampling Issues in Analog Modulation

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

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

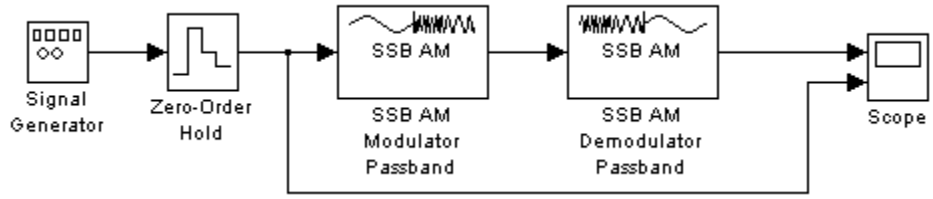
Filter Design Issues

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

Varying Filter's Cutoff Frequency Using Simulink

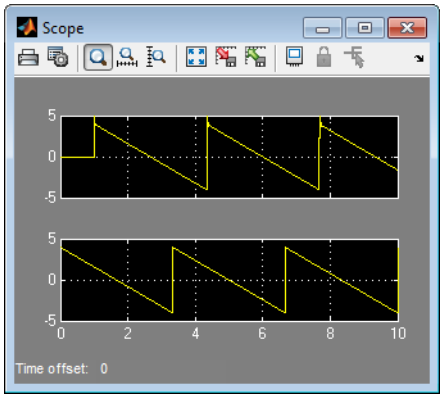
In many situations, a suitable cutoff frequency is half the carrier frequency. Because the carrier frequency must be higher than the bandwidth of the message signal, a cutoff frequency chosen in this way properly filters out unwanted frequency components. If the cutoff frequency is too high, the unwanted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

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



To open this model, enter `doc_filtercutoffs` at the MATLAB command line.

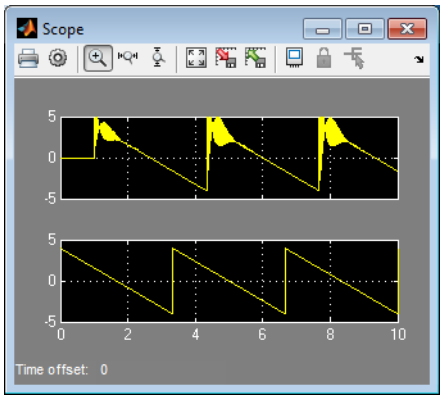
This example generates the following output:



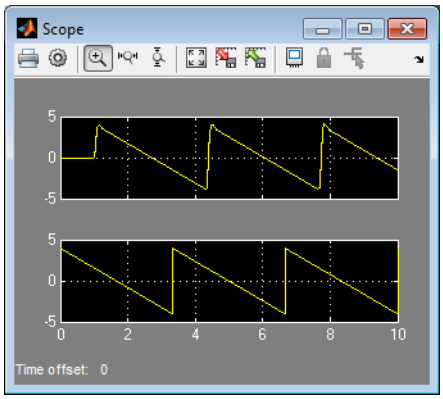
There is invariably a delay between a demodulated signal and the original received signal. Both the filter order and the filter parameters directly affect the length of this delay.

Other Filter Cutoffs

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



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



Phase-Locked Loops

A phase-locked loop combines a voltage-controlled oscillator and a phase comparator as a feedback system to adjust the oscillator frequency or phase to track an applied frequency-modulated or phase-modulated signal.

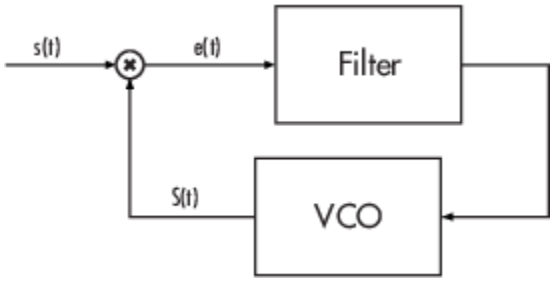
Voltage-Controlled Oscillator Blocks

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

Overview of PLL Simulation

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

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



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

Supported PLLs in Components Library

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

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

- You specify the filter's transfer function in the block mask using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each of these parameters is a vector that lists the coefficients of the respective polynomial in order of descending exponents of the variable *s*. To

design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox.

- You specify the key VCO characteristics in the block mask. All four PLL blocks use a **VCO input sensitivity** parameter. Some blocks also use **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.
- The phase detector for each of the PLL blocks is a feature that you cannot change from the block mask.

Implementing an Analog Baseband PLL

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

- Baseband PLL
- Linearized Baseband PLL

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

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

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

Implementing a Digital PLL

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

Selected Bibliography for Synchronization

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

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

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

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

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

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

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

[8] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009.

Multiple-Input Multiple-Output (MIMO)

In this section...

The use of Multiple-Input Multiple-Output (MIMO) techniques for sending and receiving multiple data signals simultaneously over the same radio channel by exploiting multipath propagation that provide potential gains in capacity when using multiple antennas at both transmitter and receiver ends of a communications system. New techniques, which account for the extra spatial dimension, have been adopted to realize these gains in new systems and previously existing systems.

MIMO technology has been adopted in multiple wireless systems, including Wi-Fi, WiMAX, LTE, and LTE-Advanced.

The Communications Toolbox product offers components to model:

- OSTBC (orthogonal space-time block coding technique)
- MIMO Fading Channels
- Spherical Decoding

and demos highlighting the use of these components in applications.

For background material on the subject of MIMO systems, see the works listed in Selected Bibliography for MIMO systems on page 16-158.

Orthogonal Space-Time Block Codes (OSTBC)

Model Orthogonal Space Time Block Coding (OSTBC) which is a MIMO technique offering full spatial diversity gain with extremely simple single-symbol maximum likelihood decoding as described in [4], [6], and [8].

In Simulink, the OSTBC Encoder and OSTBC Combiner blocks, residing in the MIMO block library, implement the orthogonal space time block coding technique. These two blocks offer a variety of specific codes (with different rates) for up to 4 transmit and 8 receive antenna systems. The encoder block is used at the transmitter to map symbols to multiple antennas while the combiner block is used at the receiver to extract the soft information per symbol using the received signal and the channel state information. You access the MIMO library by double clicking the icon in the main Communications Toolbox block library. Alternatively, you can type `commmimo` at the MATLAB command line.

The OSTBC technique is an attractive scheme because it can achieve the full (maximum) spatial diversity order and have symbol-wise maximum-likelihood (ML) decoding. For more information about the algorithmic details and the specific codes implemented, see OSTBC Combining Algorithms on the OSTBC Combiner block help page and OSTBC Encoding Algorithms on the OSTBC Encoder block help page. Similar functionality is available in MATLAB by using the `comm.OSTBCCombiner` and `comm.OSTBCEncoder` System objects.

MIMO Fading Channel

Model a MIMO fading channel using the `comm.MIMOChannel` System object in MATLAB or the MIMO Fading Channel block in Simulink. Using them you model the fading channel characteristics of MIMO links with Rayleigh and Rician fading, and use the Kronecker model for the spatial correlation between the links as described in [1].

Spherical Decoding

Model a sphere decoder using the `comm.SphereDecoder` System object in MATLAB or the Sphere Decoder block in Simulink. You can use them to find the maximum-likelihood solution for a set of received symbols over a MIMO channel with any number transmit antennas and receive antennas.

Selected Bibliography for MIMO Systems

[1] C. Oestges and B. Clerckx, *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.

[2] George Tsoulos, Ed., "*MIMO System Technology for Wireless Communications*", CRC Press, Boca Raton, FL, 2006.

[3] L. M. Correira, Ed., *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

[4] M. Jankiraman, "*Space-time codes and MIMO systems*", Artech House, Boston, 2004.

[5] G. J. Foschini, M. J. Gans, "On the limits of wireless communications in a fading environment when using multiple antennas", *IEEE Wireless Personal Communications*, Vol. 6, Mar. 1998, pp. 311-335.

[6] S. M. Alamouti, "A simple transmit diversity technique for wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 8, pp. 1451–1458, Oct. 1998.

[7] V. Tarokh, N. Seshadri, and A. R. Calderbank, "Space–time codes for high data rate wireless communication: Performance analysis and code construction," *IEEE Transactions on Information Theory*, vol. 44, no. 2, pp. 744–765, Mar. 1998.

[8] V. Tarokh, H. Jafarkhani, and A. R. Calderbank, "Space-time block codes from orthogonal designs," *IEEE Transactions on Information Theory*, vol. 45, no. 5, pp. 1456–1467, Jul. 1999.

[9] 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, 3GPP TS 36.104, v10.0.0, 2010-09.

[10] 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, 3GPP TS 36.101, v10.0.0, 2010-10.

Differential Pulse Code Modulation

In this section...

Section Overview

The quantization in the section Quantizing a Signal on page 16-11 requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM).

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

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in “Represent Partitions” on page 16-2 and “Represent Codebooks” on page 16-2, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

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

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

Represent Predictors

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

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

then the corresponding predictor vector for toolbox functions is

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

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

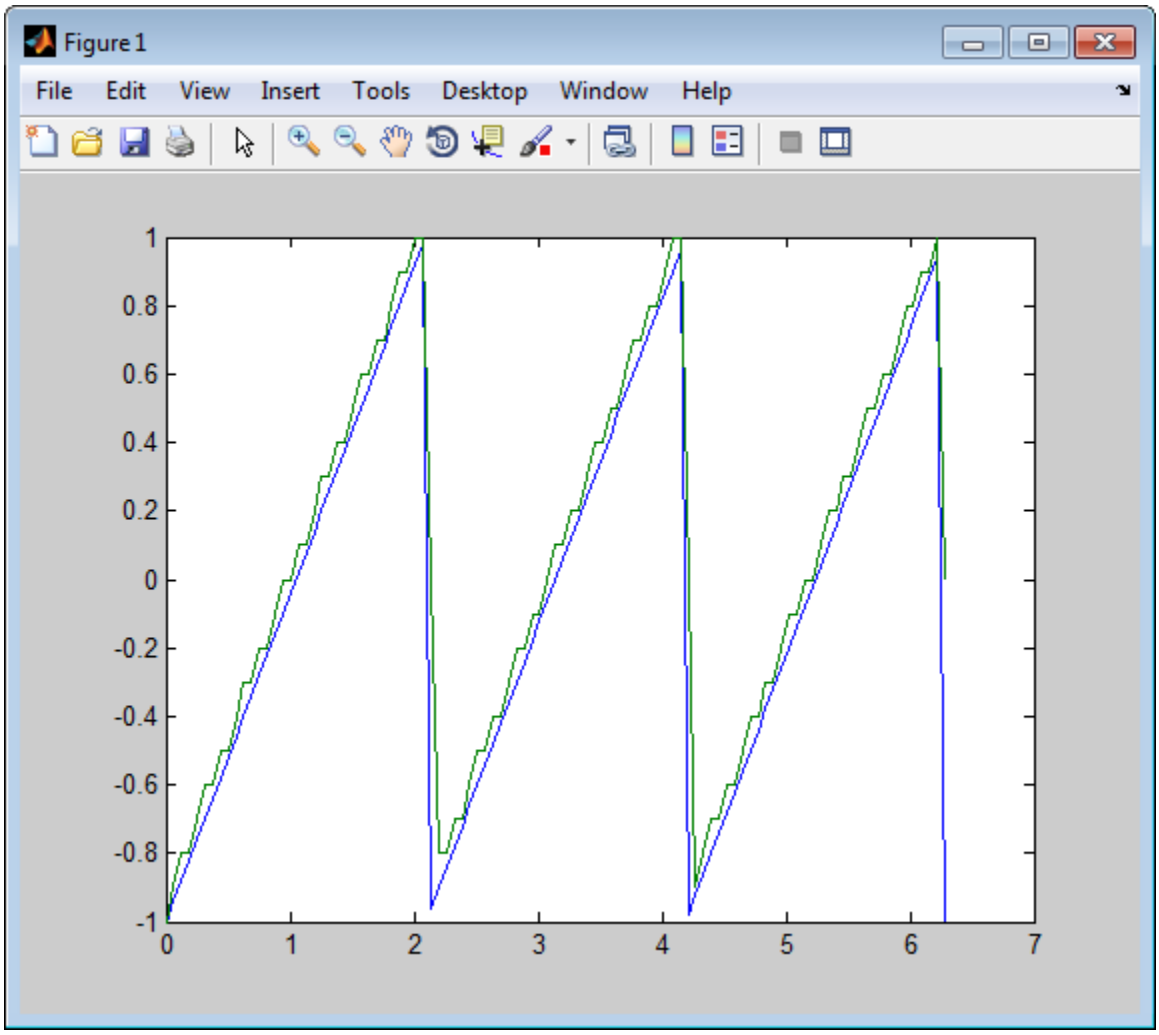
Example: DPCM Encoding and Decoding

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

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

The output is

```
distor =

    0.0327
```



Optimize DPCM Parameters

Section Overview

The section "Optimize Quantization Parameters" on page 16-3 describes how to use training data with the `lloyds` function to help find quantization parameters that will minimize signal distortion.

This section describes similar procedures for using the `dpcmopt` function in conjunction with the two functions `dpcmenco` and `dpcmdeco`, which first appear in the previous section.

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

Example: Comparing Optimized and Nonoptimized DPCM Parameters

This example is similar to the one in the last section. However, where the last example created `predictor`, `partition`, and `codebook` in a straightforward but haphazard way, this example uses the same codebook (now called `initcodebook`) as an initial guess for a new *optimized* codebook parameter. This example also uses the predictive order, 1, as the desired order of the new optimized predictor. The `dpcmopt` function creates these optimized parameters, using the sawtooth signal `x` as training data. The example goes on to quantize the training data itself; in theory, the optimized parameters are suitable for quantizing other data that is similar to `x`. Notice that the mean square distortion here is much less than the distortion in the previous example.

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

The output is

```
distor =

    0.0063
```

Quantize and Compand an Exponential Signal

When transmitting signals with a high dynamic range, quantization using equal length intervals can result in loss of precision and signal distortion. Companding is a operation that applies a logarithmic computation to compress the signal before quantization on the transmit side and applies an inverse operation to expand the signal to restore it to full scale on the receive side. Companding avoids signal distortion without the need to specify many quantization levels. Compare distortion when using 6-bit quantization on an exponential signal with and without companding. Plot the original exponential signal, the quantized signal and the expanded signal.

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;
[~,qsig,distortion] = quantiz(sig,partition,codebook);
```

Compress the signal by using the `compand` function configured to apply the mu-law method. Apply quantization and expand the quantized signal. Calculate the mean square distortion of the companded signal.

```
mu = 255; % mu-law parameter
csig_compressed = compand(sig,mu,V,'mu/compressor');
[~,quants] = quantiz(csig_compressed,partition,codebook);
csig_expanded = compand(quants,mu,max(quants),'mu/expander');
distortion2 = sum((csig_expanded - sig).^2)/length(sig);
```

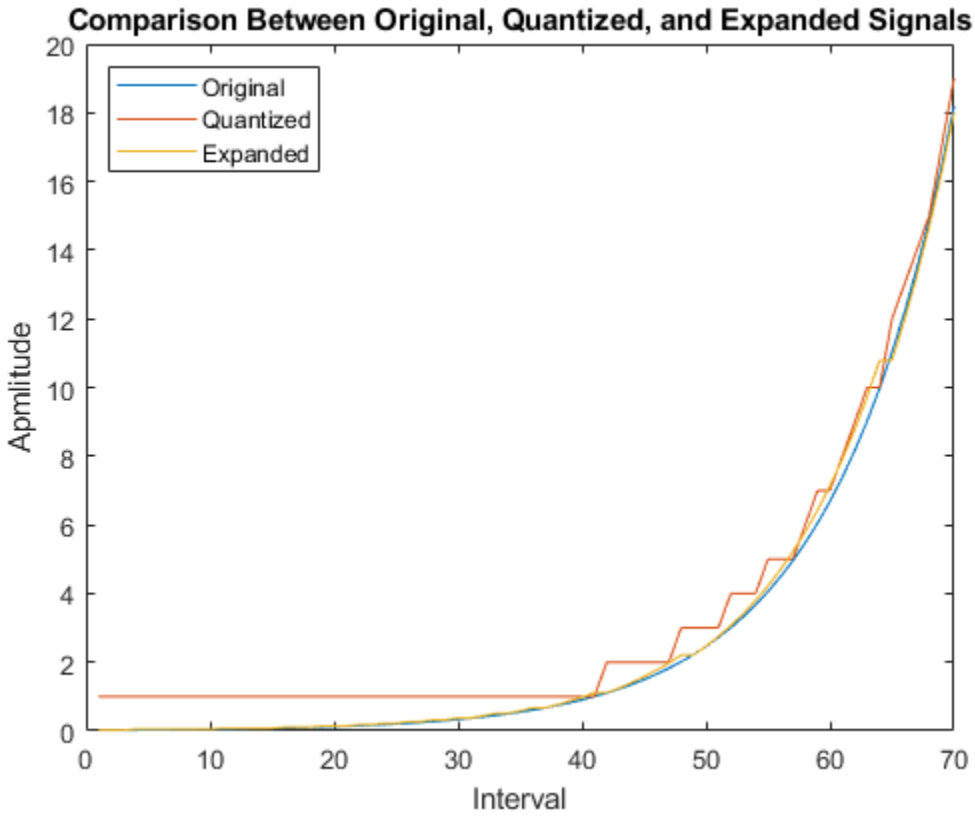
Compare the mean square distortion for quantization versus combined companding and quantization. The distortion for the companded and quantized signal is an order of magnitude lower than the distortion of the quantized signal. Equal-length intervals are well suited to the logarithm of an exponential signal but not well suited to an exponential signal itself.

```
[distortion, distortion2]
```

```
ans = 1×2

    0.5348    0.0397
```

Plot the original exponential signal, the quantized signal, and the expanded signal. Zoom in on axis to highlight the quantized signal error at lower signal levels.

```
plot([sig' qsig' csig_expanded']);
title('Comparison Between Original, Quantized, and Expanded Signals');
xlabel('Interval');
ylabel('Apmlitude');
legend('Original','Quantized','Expanded','location','nw');
axis([0 70 0 20])
```



See Also

Functions
`compand` | `quantiz`

Quantization

In this section...

Represent Partitions

Scalar quantization is a process that maps all inputs within a specified range to a common value. This process maps inputs in a different range of values to a different common value. In effect, scalar quantization digitizes an analog signal. Two parameters determine a quantization: a partition on page 16-2 and a codebook on page 16-2.

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

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

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

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

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

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

Represent Codebooks

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

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

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

Determine Which Interval Each Input Is In

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

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

The output is

```
index =

     0
     6
     5
```

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

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

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

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

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

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

Optimize Quantization Parameters

Section Overview

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

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

Example: Optimizing Quantization Parameters

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

```
% Start with the setup from 2nd example in "Quantizing a Signal."
t = [0:.1:2*pi];
sig = sin(t);
partition = [-1:.2:1];
```

```
codebook = [-1.2:.2:1];
% Now optimize, using codebook as an initial guess.
[partition2,codebook2] = lloyds(sig,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
[index2,quant2,distor2] = quantiz(sig,partition2,codebook2);
% Compare mean square distortions from initial and optimized
[distor, distor2] % parameters.
```

The output is

```
ans =

    0.0148    0.0024
```

Quantize a Signal

- “Scalar Quantization Example 1” on page 16-167
- “Scalar Quantization Example 2” on page 16-167

Scalar Quantization Example 1

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

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

The output is below.

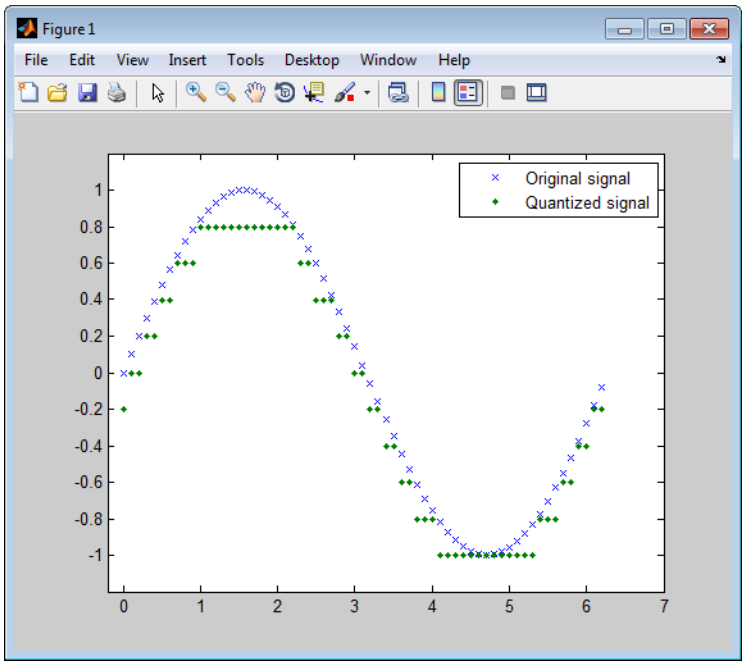
```
quantized =

  Columns 1 through 6

   -1.0000   -1.0000   -1.0000   -1.0000    0.5000    0.5000

  Columns 7 through 12

    2.0000    2.0000    2.0000    2.0000    2.0000    3.0000

  Column 13

    3.0000
```

Scalar Quantization Example 2

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

```
t = [0:.1:2*pi]; % Times at which to sample the sine function
sig = sin(t); % Original signal, a sine wave
partition = [-1:.2:1]; % Length 11, to represent 12 intervals
codebook = [-1.2:.2:1]; % Length 12, one entry for each interval
```

```
[index,quants] = quantiz(sig,partition,codebook); % Quantize.
plot(t,sig,'x',t,quants,'.')
legend('Original signal','Quantized signal');
axis([-.2 7 -1.2 1.2])
```



See Also

Functions
quantiz

OFDM Modulation

OFDM with User-Specified Pilot Indices

This example shows how to construct an orthogonal frequency division modulation (OFDM) modulator/demodulator pair and to specify their pilot indices. The OFDM modulator System object enables you to specify pilot subcarrier indices consistent with the constraints described in `comm.OFDMModulator.info`. In this example, for OFDM transmission over a 3x2 channel, pilot indices are created for each of the three transmit antennas. Additionally, the pilot indices differ between odd and even symbols.

Create an OFDM modulator object having five symbols, three transmit antennas, and length six windowing.

```
ofdmMod = comm.OFDMModulator('FFTLength',256, ...
    'NumGuardBandCarriers',[12; 11], ...
    'NumSymbols',5, ...
    'NumTransmitAntennas',3, ...
    'PilotInputPort',true, ...
    'Windowing',true, ...
    'WindowLength',6);
```

Specify pilot indices for even and odd symbols for the first transmit antenna.

```
pilotIndOdd = [20; 58; 96; 145; 182; 210];
pilotIndEven = [35; 73; 111; 159; 197; 225];

pilotIndicesAnt1 = cat(2,pilotIndOdd,pilotIndEven,pilotIndOdd, ...
    pilotIndEven,pilotIndOdd);
```

Generate pilot indices for the second and third antennas based on the indices specified for the first antenna. Concatenate the indices for the three antennas and assign them to the `PilotCarrierIndices` property.

```
pilotIndicesAnt2 = pilotIndicesAnt1 + 5;
pilotIndicesAnt3 = pilotIndicesAnt1 - 5;

ofdmMod.PilotCarrierIndices = ...
    cat(3,pilotIndicesAnt1,pilotIndicesAnt2,pilotIndicesAnt3);
```

Create on OFDM demodulator with two receive antennas based on the existing OFDM modulator System object. Determine the data and pilot dimensions using the `info` function.

```
ofdmDemod = comm.OFDMDemodulator(ofdmMod);
ofdmDemod.NumReceiveAntennas = 2;

dims = info(ofdmMod)
```

```
dims = struct with fields:
     DataInputSize: [215 5 3]
    PilotInputSize: [6 5 3]
        OutputSize: [1360 3]
```

Generate data and pilot symbols for the OFDM modulator given the array sizes specified in `modDim`.

```
dataIn = ...
    complex(randn(dims.DataInputSize), ...
    randn(dims.DataInputSize));
```

```
pilotIn = ...
    complex(randn(dims.PilotInputSize), ...
    randn(dims.PilotInputSize));
```

Apply OFDM modulation to the data and pilots.

```
modOut = ofdmMod(dataIn,pilotIn);
```

Pass the modulated data through a 3x2 random channel.

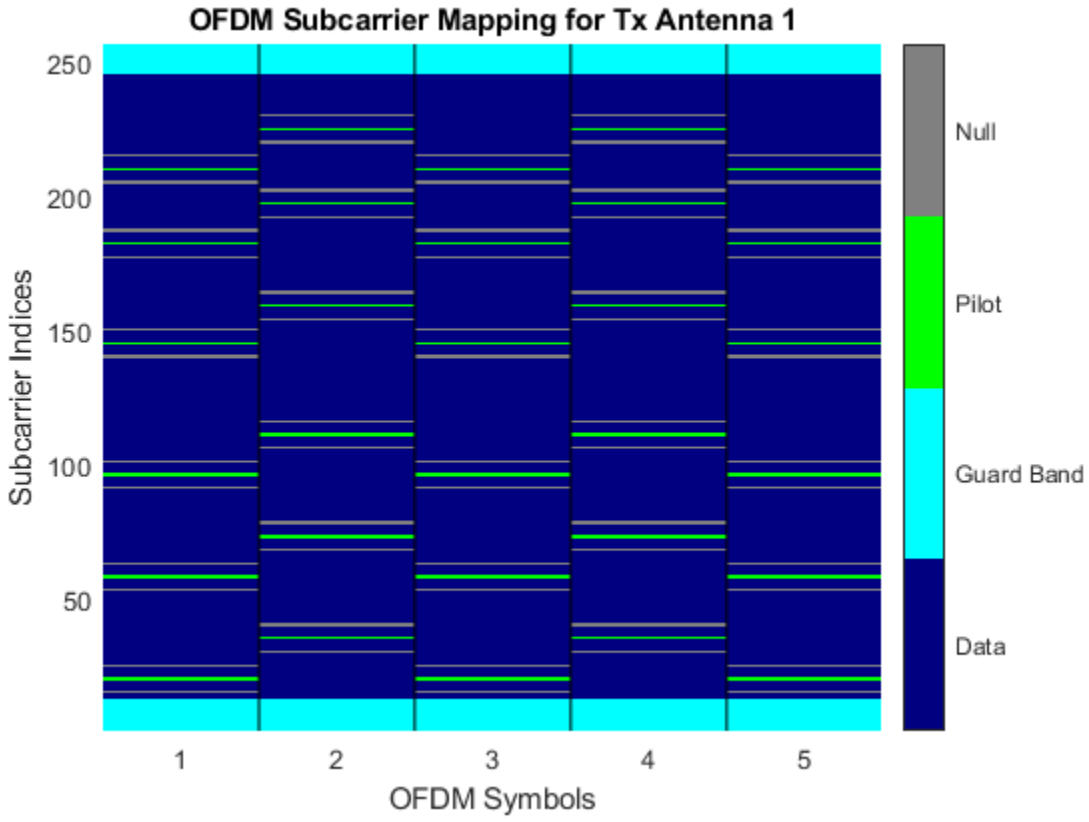
```
chanGain = complex(randn(3,2),randn(3,2));
chanOut = modOut * chanGain;
```

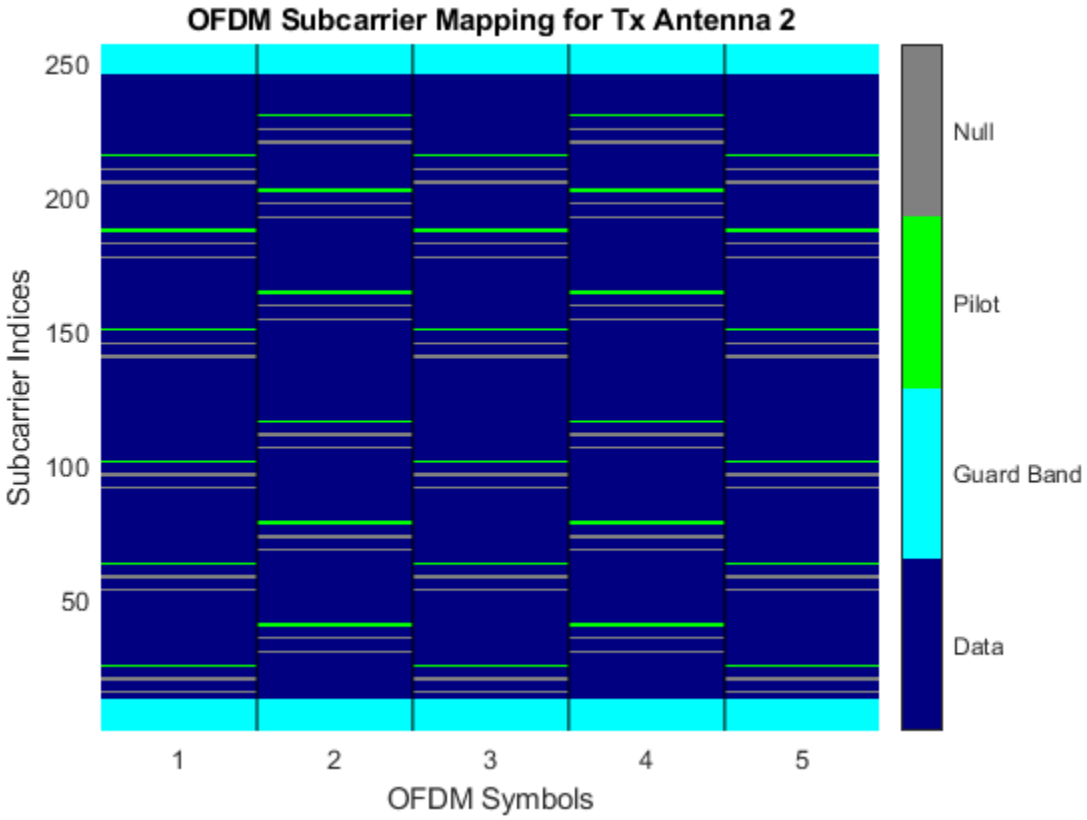
Demodulate the received data using the OFDM demodulator object.

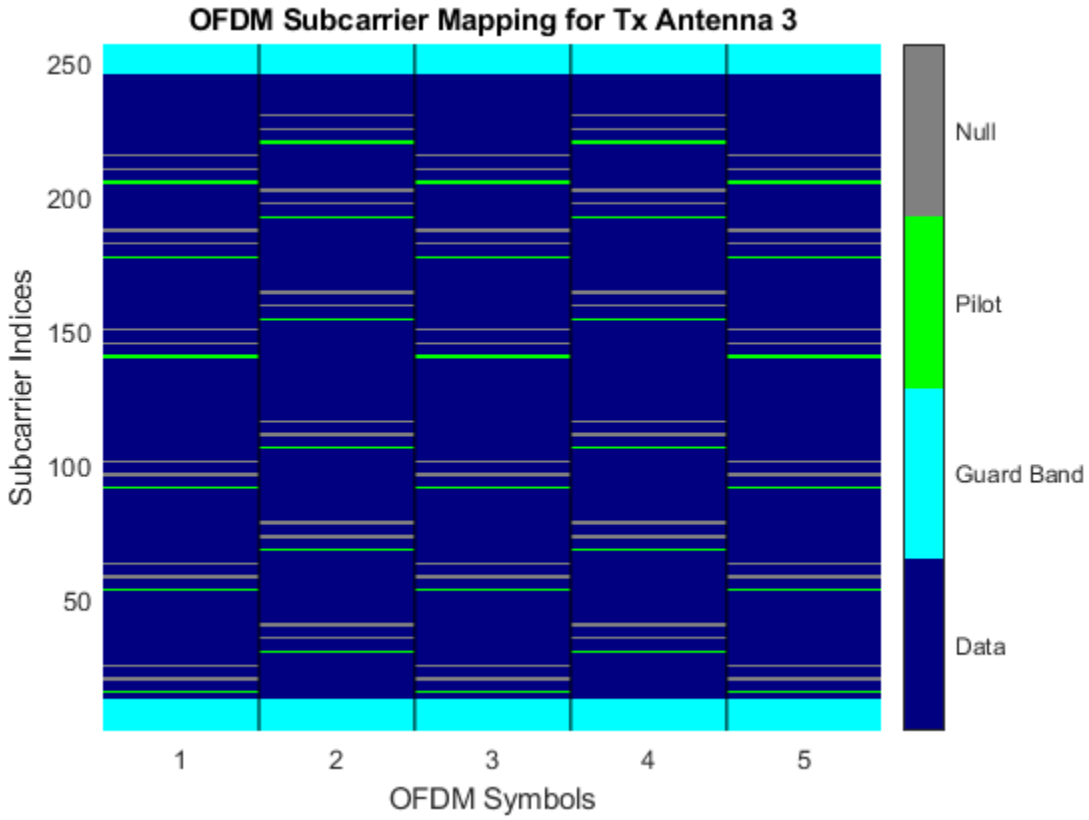
```
[dataOut,pilotOut] = ofdmDemod(chanOut);
```

Show the resource mapping for the three transmit antennas. The gray lines in the figure show the placement of custom nulls to avoid interference among antennas.

```
showResourceMapping(ofdmMod)
```




OFDM Subcarrier Mapping for Tx Antenna 2
Subcarrier Indices
250
200
150
100
50
1
2
3
4
5
OFDM Symbols
Null
Pilot
Guard Band
Data



For the first transmit and first receive antenna pair, demonstrate that the input pilot signal matches the input pilot signal.

```
pilotCompare = ...
    abs(pilotIn(:,:,1)*chanGain(1,1)) - abs(pilotOut(:,:,1,1));
max(pilotCompare(:) < 1e-10)
```

```
ans = logical
   1
```

SER Simulation for OFDM Link

This example shows how to perform a symbol error rate (SER) simulation of an over-the-air OFDM communication link.

A basic communications link using OFDM modulation with QPSK symbols is simulated. There is a single transmit and a single receive antenna.

Create QPSK modulator and demodulator objects.

```
qpskMod = comm.QPSKModulator;
qpskDemod = comm.QPSKDemodulator;
```

Create a default OFDM modulator and demodulator pair.

```
ofdmMod = comm.OFDMModulator;
ofdmDemod = comm.OFDMDemodulator;
```

Use the `info` function to determine the required input dimensions for the OFDM modulator.

```
modDim = info(ofdmMod)
```

```
modDim = struct with fields:
    DataInputSize: [53 1]
       OutputSize: [80 1]
```

Set the number of frames. Determine the number of OFDM symbols per frame from the `modDim.DataInputSize` array.

```
nFrames = 10000;
nSymbolsPerFrame = modDim.DataInputSize(1);
```

Create an error rate counter with a reset input port. Initialize the symbol error rate vector, SER.

```
errRate = comm.ErrorRate('ResetInputPort',true);
SER = zeros(nFrames,1);
```

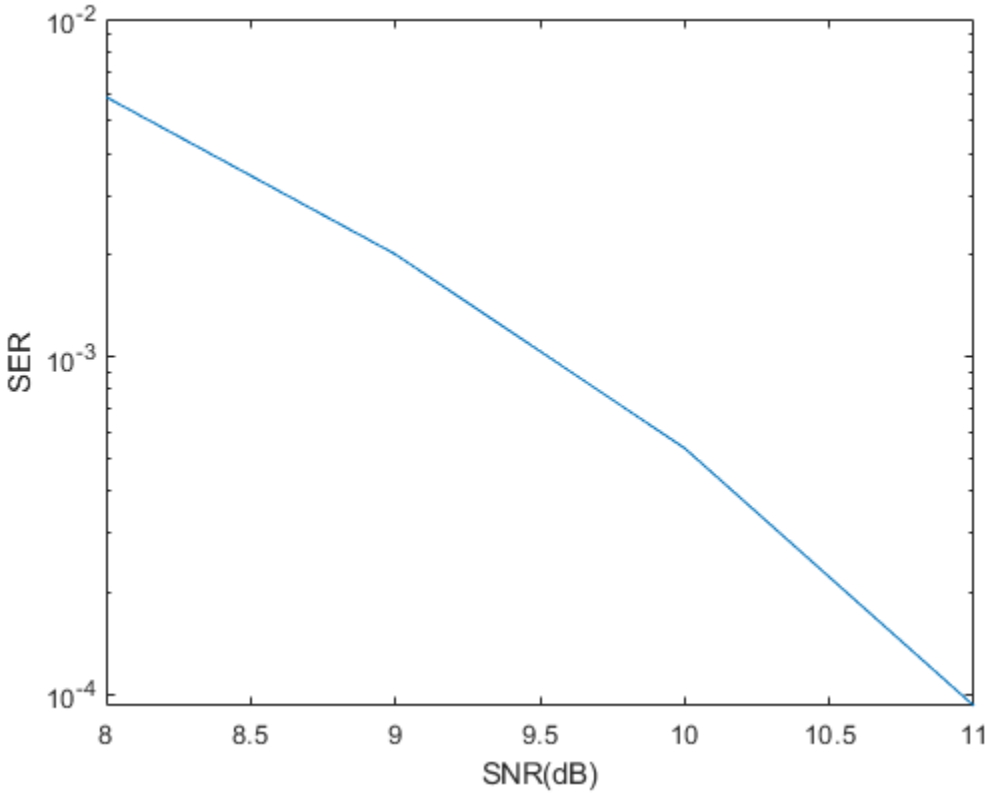
Run the simulation over 10000 OFDM frames (530000 symbols). During loop execution, generate a random data vector with length equal to the required number of symbols per frame, Apply QPSK modulation and then apply OFDM modulation. Pass the OFDM modulated data through the AWGN channel and then apply OFDM demodulation. Demodulate the resultant QPSK data and compare it with the original data to determine the symbol error rate.

```
snr = [8 9 10 11];
meanSER = zeros(size(snr));
for j = 1:length(snr)
    for k = 1:nFrames
    % Generate random data for each OFDM frame
    data = randi([0 3],nSymbolsPerFrame,1);

    % Apply QPSK modulation
    txQPSK = qpskMod(data);

    % Apply OFDM modulation
    txSig = ofdmMod(txQPSK);
```

```
    % Pass OFDM signal through AWGN channel
    rxSig = awgn(txSig,snr(j),'measured');

    % Demodulate OFDM data
    rxQPSK = ofdmDemod(rxSig);

    % Demodulate QPSK data
    rxData = qpskDemod(rxQPSK);

    % Compute BER
    errors = errRate(data,rxData,1);
    SER(k) = errors(1);
    end
    meanSER(j) = mean(SER);
end
```

Display the symbol error rate (SER) curve.

```
semilogy(snr,meanSER)
xlabel('SNR(dB)')
ylabel('SER')
```



OFDM with MIMO Simulation

This example shows how to use an OFDM modulator and demodulator in a simple, 2x2 MIMO error rate simulation. The OFDM parameters are based on the 802.11n standard.

Create a QPSK modulator and demodulator pair.

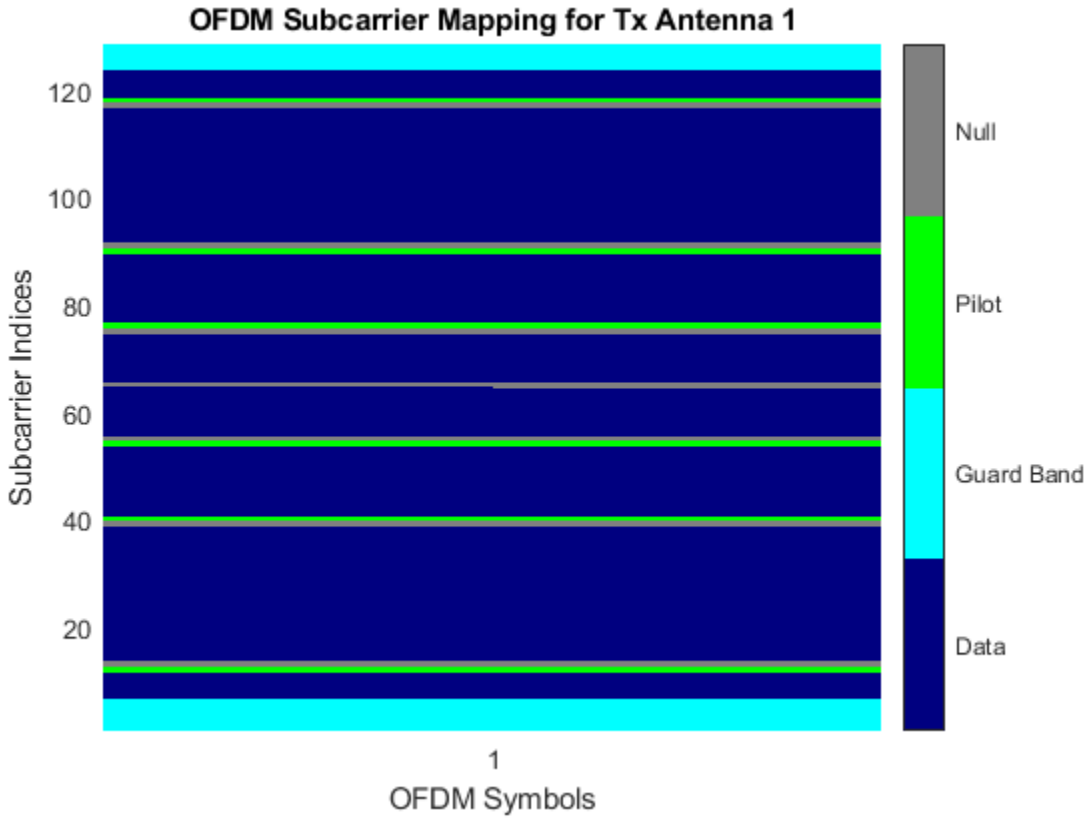
```
qpskMod = comm.QPSKModulator;
qpskDemod = comm.QPSKDemodulator;
```

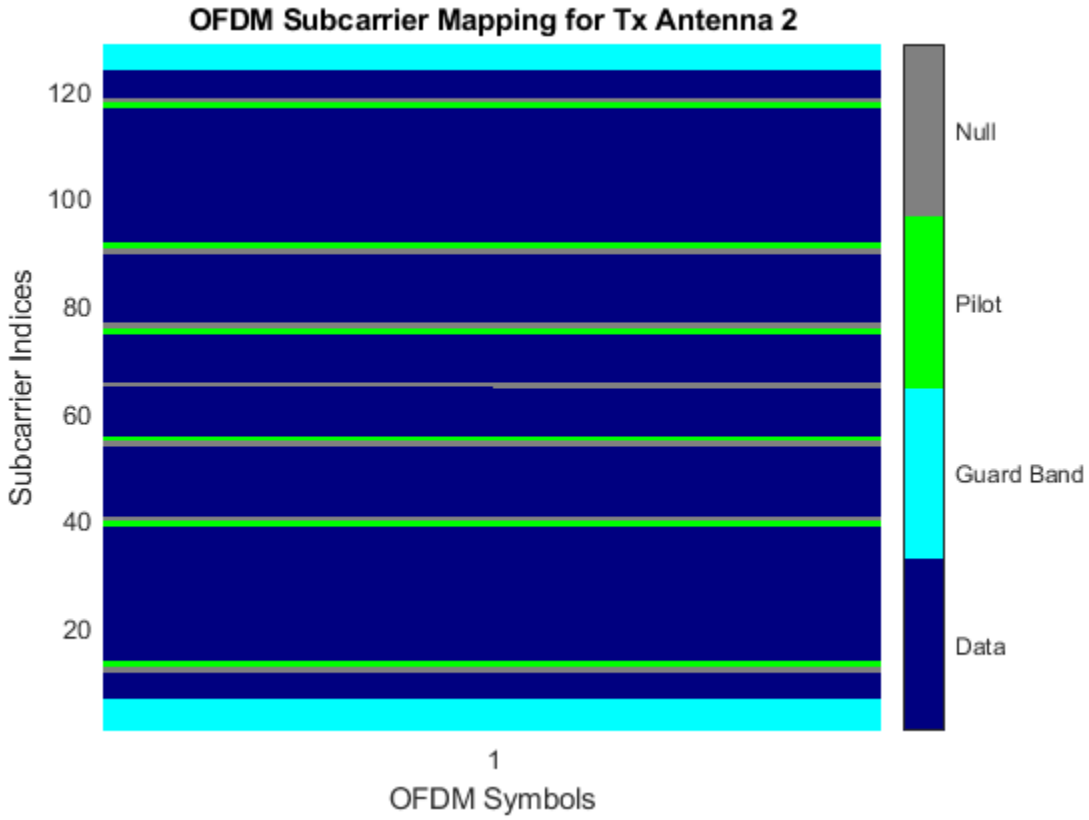
Create an OFDM modulator and demodulator pair with user-specified pilot indices, an inserted DC null, two transmit antennas, and two receive antennas. Specify pilot indices that vary across antennas.

```
ofdmMod = comm.OFDMModulator('FFTLength',128, ...
    'PilotInputPort',true, ...
    'PilotCarrierIndices', ...
    cat(3,[12; 40; 54; 76; 90; 118],[13; 39; 55; 75; 91; 117]), ...
    'InsertDCNull',true, ...
    'NumTransmitAntennas',2);
ofdmDemod = comm.OFDMDemodulator(ofdmMod);
ofdmDemod.NumReceiveAntennas = 2;
```

Show the resource mapping of pilot subcarriers for each transmit antenna. The gray lines in the figure denote the insertion of null subcarriers to minimize pilot signal interference.

```
showResourceMapping(ofdmMod)
```





Determine the dimensions of the OFDM modulator by using the `info` method.

```
ofdmModDim = info(ofdmMod);

numData = ofdmModDim.DataInputSize(1);   % Number of data subcarriers
numSym = ofdmModDim.DataInputSize(2);    % Number of OFDM symbols
numTxAnt = ofdmModDim.DataInputSize(3);  % Number of transmit antennas
```

Generate data symbols to fill 100 OFDM frames.

```
nframes = 100;
data = randi([0 3],nframes*numData,numSym,numTxAnt);
```

Apply QPSK modulation to the random symbols and reshape the resulting column vector to match the OFDM modulator requirements.

```
modData = qpskMod(data(:));
modData = reshape(modData,nframes*numData,numSym,numTxAnt);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate;
```

Simulate the OFDM system over 100 frames assuming a flat, 2x2, Rayleigh fading channel. Remove the effects of multipath fading using a simple, least squares solution, and demodulate the OFDM waveform and QPSK data. Generate error statistics by comparing the original data with the demodulated data.

```
for k = 1:nframes

    % Find row indices for kth OFDM frame
    indData = (k-1)*ofdmModDim.DataInputSize(1)+1:k*numData;

    % Generate random OFDM pilot symbols
    pilotData = complex(rand(ofdmModDim.PilotInputSize), ...
        rand(ofdmModDim.PilotInputSize));

    % Modulate QPSK symbols using OFDM
    dataOFDM = ofdmMod(modData(indData,:,:),pilotData);

    % Create flat, i.i.d., Rayleigh fading channel 2-by-2 channel
    chGain = complex(randn(2,2),randn(2,2))/sqrt(2);

    % Pass OFDM signal through Rayleigh and AWGN channels
    receivedSignal = awgn(dataOFDM*chGain,30);

    % Apply least squares solution to remove effects of fading channel
    rxSigMF = chGain.' \ receivedSignal.';

    % Demodulate OFDM data
    receivedOFDMData = ofdmDemod(rxSigMF.');

    % Demodulate QPSK data
    receivedData = qpskDemod(receivedOFDMData(:));

    % Compute error statistics
    dataTmp = data(indData,:,:);
    errors = errorRate(dataTmp(:),receivedData);
end
```

Display the error statistics.

```
fprintf('\nSymbol error rate = %d from %d errors in %d symbols\n',errors)
```

```
Symbol error rate = 9.471154e-02 from 1970 errors in 20800 symbols
```

Gray-Coded Binary Ordering

Gray coding is a technique that multilevel modulation schemes often use to minimize the bit error rate. It consists of ordering modulation symbols so that the binary representations of adjacent symbols differ by only one bit. This section shows a communications system with Gray-coded 8-ary phase shift keying (8-PSK) modulation to compare the error rate performance of Gray and natural binary coded bit ordering.

In this section...

Introduction

The example in this section modulates gray and natural binary coded data frames using the 8-PSK method. The data frames passes through an AWGN channel, and are demodulated using an 8-PSK demodulator. Error rate calculator System objects measure the symbol and bit error rates.

In this communications system, PSK Modulator System objects:

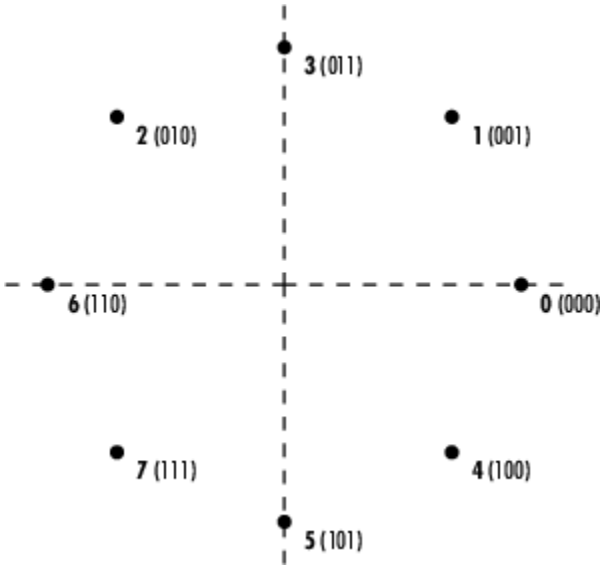
- Accept binary-valued inputs that represent integers between 0 and M – 1. M is the modulation order and is equal to 8 for 8-PSK modulation.
- Map binary representations to constellation points using Gray-coded and natural binary-coded ordering.
- Produce unit-magnitude complex phasor outputs, with evenly spaced phases between 0 and $2\pi(M – 1)/M$.

This table indicates the relationship between Gray-coded binary representations in the input and phasors in the output. The second column of the table is an intermediate representation that the System object uses in its computations.

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

This table sorts the first two columns from the previous table, according to the output values. This sorting makes it clearer that there is only a 1 bit difference between neighboring symbols. In the following figure, notice that the numbers in the second column of the table appear in counterclockwise order.

Modulator Output	Modulator Input
exp(0)	000
exp($j\pi/4$)	001
exp($j\pi/2$) = exp($j2\pi/4$)	011
exp($j3\pi/4$)	010
exp($j\pi$) = exp($j4\pi/4$)	110
exp($j5\pi/4$)	111
exp($j3\pi/2$) = exp($j6\pi/4$)	101
exp($j7\pi/4$)	100



Compare Error Rate for Gray- and Binary-Coded Ordering

Compare Gray coding with natural binary coding by using appropriately configured PSK modulator and PSK demodulator System objects. This simulation iterates over a range of bit energy to noise power spectral density, E_b/N_0, values and runs until either the specified maximum number of bit errors (`maxNumErrs`) or the maximum number of bits (`maxNumBits`) is reached for Gray coding for each E_b/N_0 point.

Initialization

Initialize the system variables and create System objects for modulation, demodulation, AWGN channel, and error rate operations. Since the `comm.AWGNChannel` System object™ and the `randi` function use the default random stream, set the random number generator seed to ensure repeatable results. The state of the random number generator is stored before setting the random stream seed, and restored at the end of the example.

```
M = 8; % Modulation order for 8-PSK
SamplesPerFrame = 10000;
maxNumErrs=100;
maxNumBits=1e8;
```

```
prevState = rng;
rng(529558);
```

Create PSK modulator and demodulator System objects to map the binary input data to 8-PSK Gray- and binary-coded constellations.

```
pskmod = comm.PSKModulator('ModulationOrder',M, ...
    'SymbolMapping','Gray', ...
    'PhaseOffset',0, ...
    'BitInput',true);
pskdemod = comm.PSKDemodulator('ModulationOrder',M, ...
    'SymbolMapping','Gray', ...
    'PhaseOffset',0, ...
    'BitOutput',true, ...
    'OutputDataType','uint8', ...
    'DecisionMethod','Hard decision');
pskmodb = comm.PSKModulator('ModulationOrder',M, ...
    'SymbolMapping','Binary', ...
    'PhaseOffset',0, ...
    'BitInput',true);
pskdemodb = comm.PSKDemodulator('ModulationOrder',M, ...
    'SymbolMapping','Binary', ...
    'PhaseOffset',0, ...
    'BitOutput',true, ...
    'OutputDataType','uint8', ...
    'DecisionMethod','Hard decision');
```

Create an AWGN channel System object to add noise to the modulated signal. The noise method is configured to E_b/N_0 for the processing loop. The PSK modulator generates symbols with 1 W of power, so the signal power property of the AWGN channel object is set to 1 W also.

```
awgnchan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'BitsPerSymbol',log2(M),'SignalPower',1);
```

Create a symbol error rate and bit error rate calculator System objects to compare the demodulated integer and bit data with the original source data. This comparison yields symbol error and bit error statistics. The output of the error rate calculator System object is a three-element vector containing the calculated error rate, the number of errors observed, and the amount of data processed. The simulation uses the three-element vector to determine when to stop the simulation.

```
symerror = comm.ErrorRate;
biterror = comm.ErrorRate;
biterrorb = comm.ErrorRate;
```

Frame Processing Loop

Configure a frame processing loop where data is coded, modulated, and demodulated using 8-PSK modulation. The loop simulates the communications system for E_b/N_0 values in the range 0 dB to 12 dB in steps of 2 dB.

For each E_b/N_0 value, the simulation stops when either the maximum number of errors (`maxNumErrs`) or the maximum number of bits (`maxNumBits`) processed by the bit error rate calculator System object is reached for the Gray coded bits.

```
EbNoVec = 0:2:12; % Eb/No values to simulate
SERVec = zeros(size(EbNoVec)); % SER history
BERVec = zeros(size(EbNoVec)); % BER history for Gray ordered
```

```
BERVecb = zeros(size(EbNoVec)); % BER history for binary ordered
for p = 1:length(EbNoVec)
  % Reset System objects
  reset(symerror);
  reset(biterror);
  reset(biterrorb);
  awgnchan.EbNo = EbNoVec(p);
  % Reset SER / BER for the current Eb/No value
  SER = zeros(3,1);
  BER = zeros(3,1);
  while (BER(2)<maxNumErrs) && (BER(3)<maxNumBits)
    % Generate random data
    txSym = randi([0 M-1],SamplesPerFrame,1,'uint8');
    txBits = reshape(de2bi(txSym,log2(M),'left-msb')',[],1); % Convert symbols to bits

    tx = pskmod(txBits);
    txb = pskmodb(txBits);
    rx = awgnchan(tx);
    rxb = awgnchan(txb);
    rxBits = pskdemod(rx);
    rxBitsb = pskdemodb(rxb);
    rxSym = bi2de(reshape(rxBits,log2(M),[])','left-msb');

    SER = symerror(txSym,rxSym); % Symbol error rate for Gray-coded data
    BER = biterror(txBits,rxBits); % Bit error rate for Gray-coded data
    BERb = biterrorb(txBits,rxBitsb); % Bit error rate for natural binary-coded data
  end
  % Save history of SER and BER values
  SERVec(p) = SER(1);
  BERVec(p) = BER(1);
  BERVecb(p) = BERb(1);
end
```

Restore the default stream.

```
rng(prevState)
```

Results Analysis

Analyze the data from the example and compare theoretical performance with simulation performance. The theoretical symbol error probability of MPSK is

$$P_{\mathrm{E}}(M) = \mathrm{erfc}\left(\sqrt{\frac{E_S}{N_0}}\sin\left(\frac{\pi}{M}\right)\right)$$

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

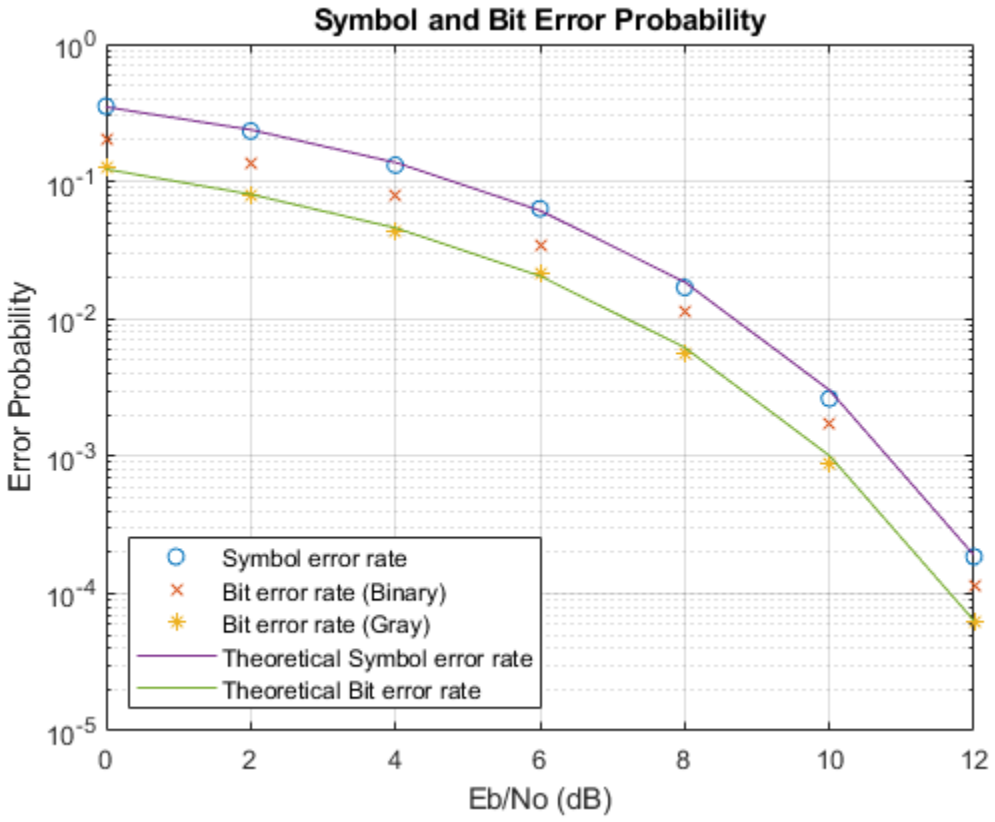
To determine the bit error probability, convert the symbol error probability, P_{E}, to its bit error equivalent. There is no general formula for the symbol to bit error conversion. Nevertheless, upper and lower limits are easy to establish. The actual bit error probability, P_b, can be shown to be bounded by

$$\frac{P_{\mathrm{E}}(M)}{\log_2 M} \le P_b \le \frac{M/2}{M-1}P_{\mathrm{E}}(M)$$

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

Calculate theoretical error probabilities by using the use the `berawgn` function. Plot the simulated symbol error rate for Gray coding, bit error rate for Gray and natural binary coding, and the theoretical symbol error and bit error probabilities for Gray coding.

```
[theorBER,theorSER] = berawgn(EbNoVec,'psk',M,'nondiff');

figure;
semilogy(EbNoVec,SERVec,'o',EbNoVec,BERVecb,'x',EbNoVec,BERVec,'*', ...
         EbNoVec,theorSER,'-',EbNoVec,theorBER,'-');
legend('Symbol error rate','Bit error rate (Binary)','Bit error rate (Gray)', ...
          'Theoretical Symbol error rate','Theoretical Bit error rate', ...
          'Location','SouthWest');
xlabel('Eb/No (dB)'); ylabel('Error Probability');
title('Symbol and Bit Error Probability');
grid on;
```



CPM Phase Tree

In this section...

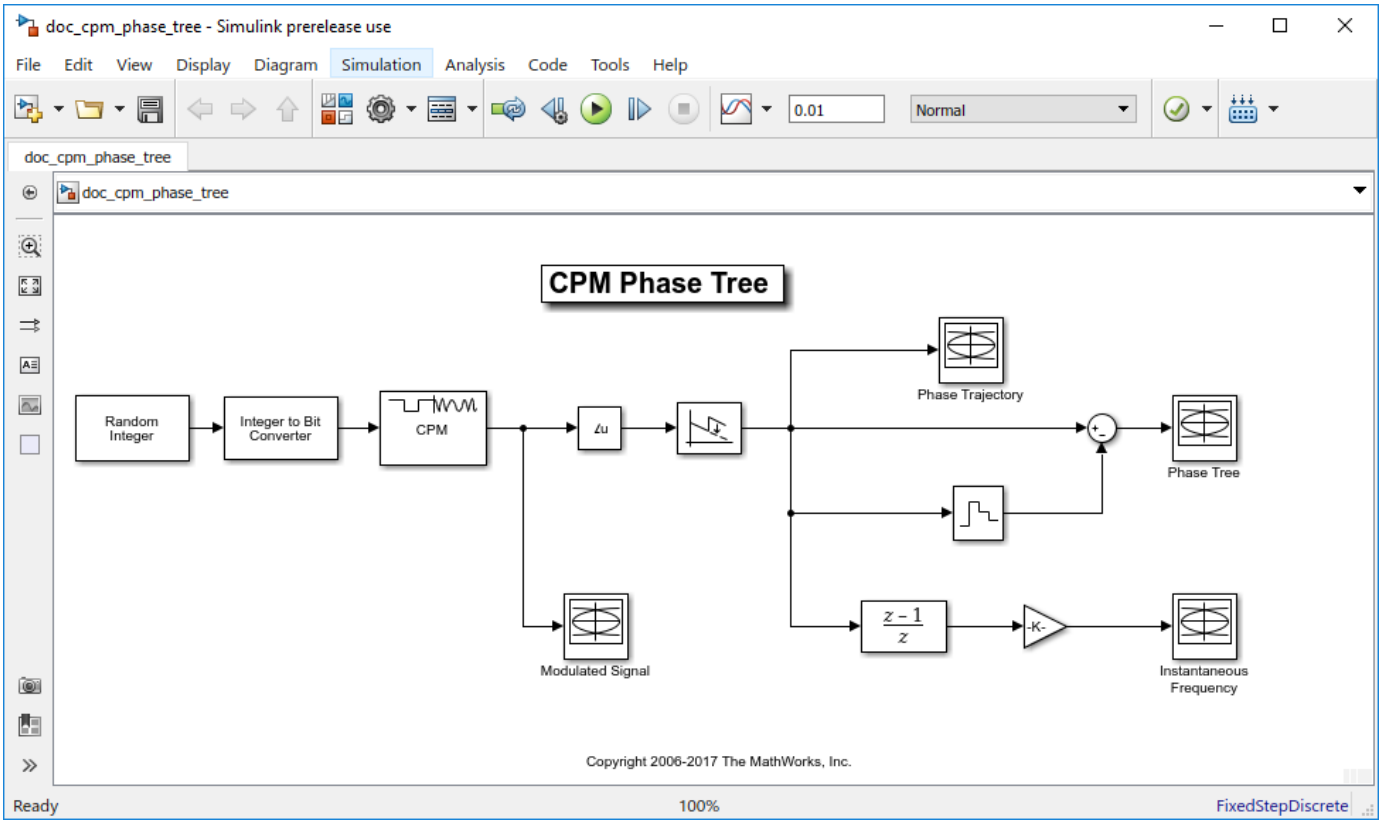
This model shows how to use the Eye Diagram block to view the phase trajectory, phase tree, and instantaneous frequency of a CPM modulated signal.

Structure of the Example

This example, `doc_cpm_phase_tree`, uses various Communications Toolbox, DSP System Toolbox, and Simulink blocks to model a baseband CPM signal.

In particular, the example model includes these blocks:

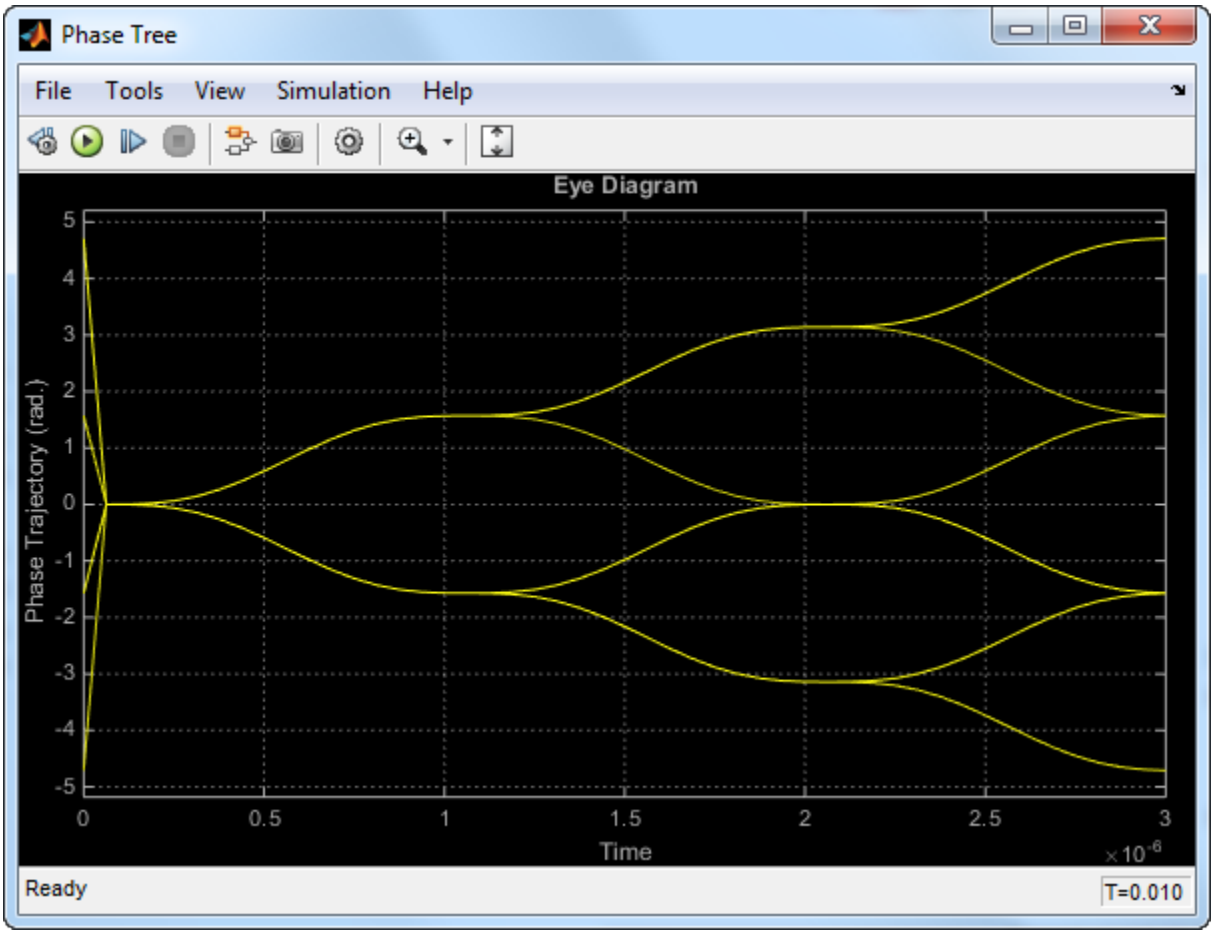
- Random Integer Generator block
- Integer to Bit Converter block
- CPM Modulator Baseband block
- Complex to Magnitude-Angle block
- Phase Unwrap block
- Zero-Order Hold block
- Discrete Transfer Fcn block
- Gain block
- Multiple copies of the Eye Diagram Scope block



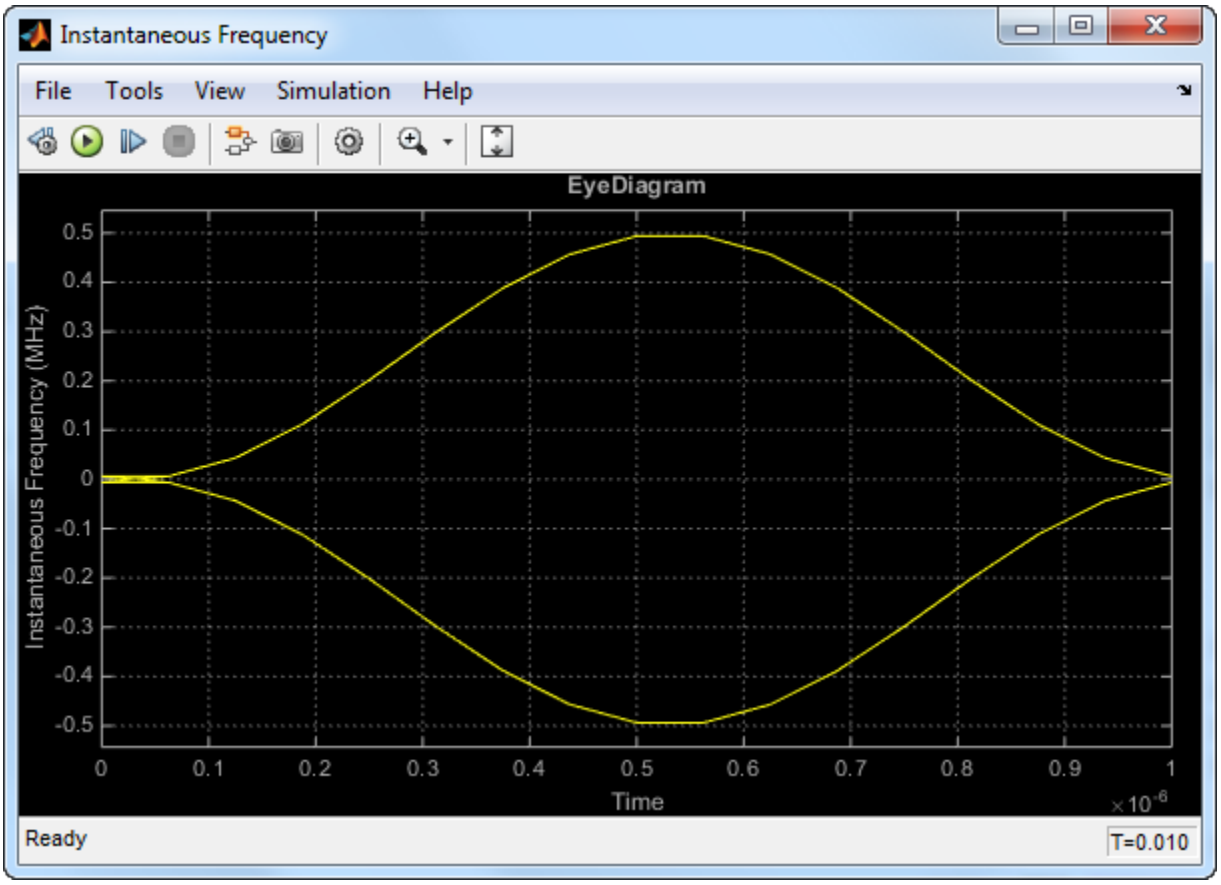
Results and Displays

When you run the example, several Eye Diagram blocks show how the CPM signal changes over time:

- The Modulated Signal block displays the in-phase and quadrature signals. Double-click the block to open the scope. The modulated signal is easy to see in the eye diagram only when the **Modulation index** parameter in the CPM Modulator Baseband block is set to 0.5. If you set the **Modulation index** to another value, for example 2/3, the features of the modulated signal are difficult to decipher for this more complex modulation. Unwrapping the phase and plotting it is another way to illustrate these more complex CPM modulated signals.
- The Phase Trajectory block displays the CPM phase. Double-click the block to open the scope. The Phase Trajectory block reveals that the signal phase is also difficult to view because it drifts with the data input to the modulator.
- The Phase Tree block displays the phase tree of the signal. The CPM phase is processed by a few simple blocks to make the CPM pulse shaping easier to view. This processing holds the phase at the beginning of the symbol interval and subtracts it from the signal. This resets the phase to zero every three symbols. The resulting plot shows the many phase trajectories that can be taken by the signal from any given symbol epoch.



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



Exploring the Example

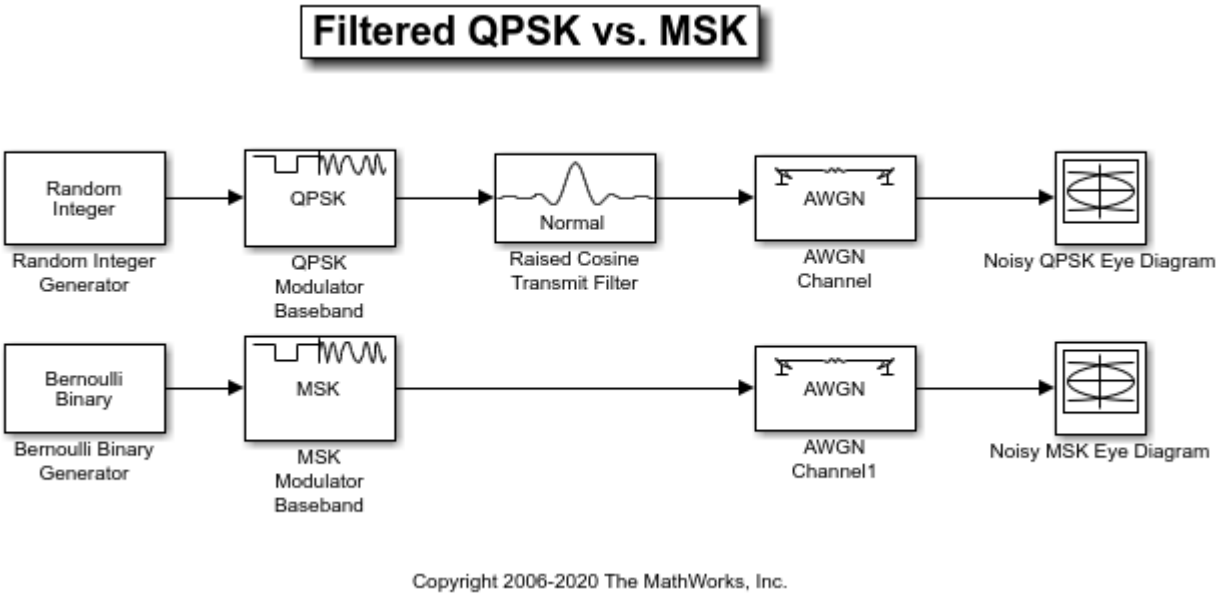
To learn more about the example, try changing the following parameters in the CPM Modulator Baseband block:

- Change **Pulse length** to a value between 1 and 6.
- Change **Frequency pulse shape** to one of the other settings, such as `Rectangular` or `Gaussian`.

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

Compare Filtered QPSK and MSK Signals in Simulink

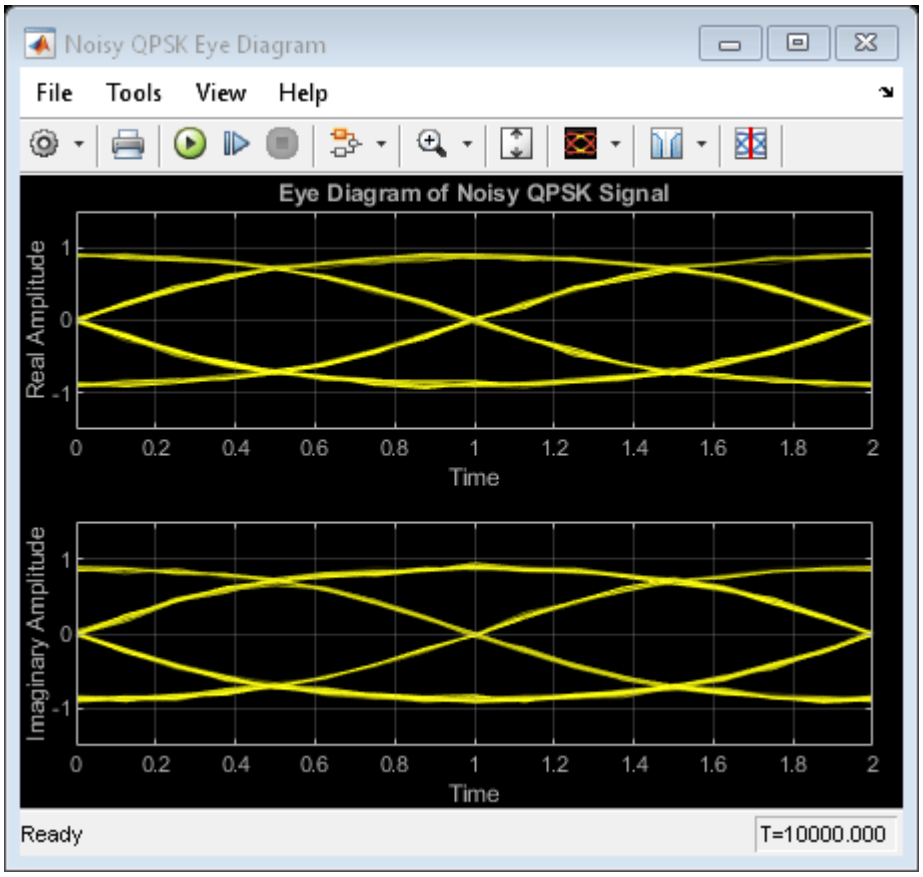
This model compares filtered quadrature phase shift keying (QPSK) and minimum shift keying (MSK) modulation schemes.

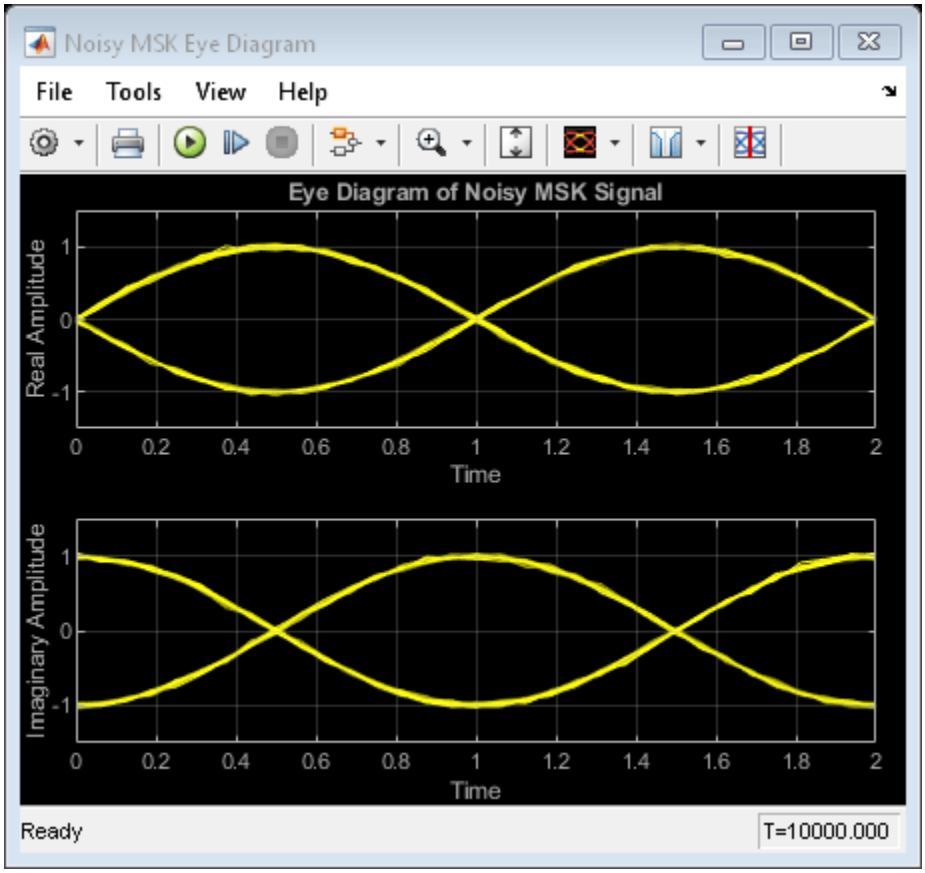


The model generates the filtered QPSK signal using random integer data from the Random Integer Generator block, which gets modulated by the QPSK Modulator Baseband block, and then filtered by the Raised Cosine Transmit Filter block. The model generates the MSK signal using random binary data from the Bernoulli Binary Generator block, which gets modulated by the MSK Modulator Baseband block. Noise is added to both the filtered QPSK and MSK signals by using AWGN Channel blocks. The Eye Diagram blocks are used to visualize eye diagrams of both signals.

For filtered QPSK modulation, the values of both the in-phase and quadrature components of the signal are permitted to change at any symbol interval. For MSK modulation, the symbol interval is half that for QPSK, but the in-phase and quadrature components change values in alternate symbol epochs.

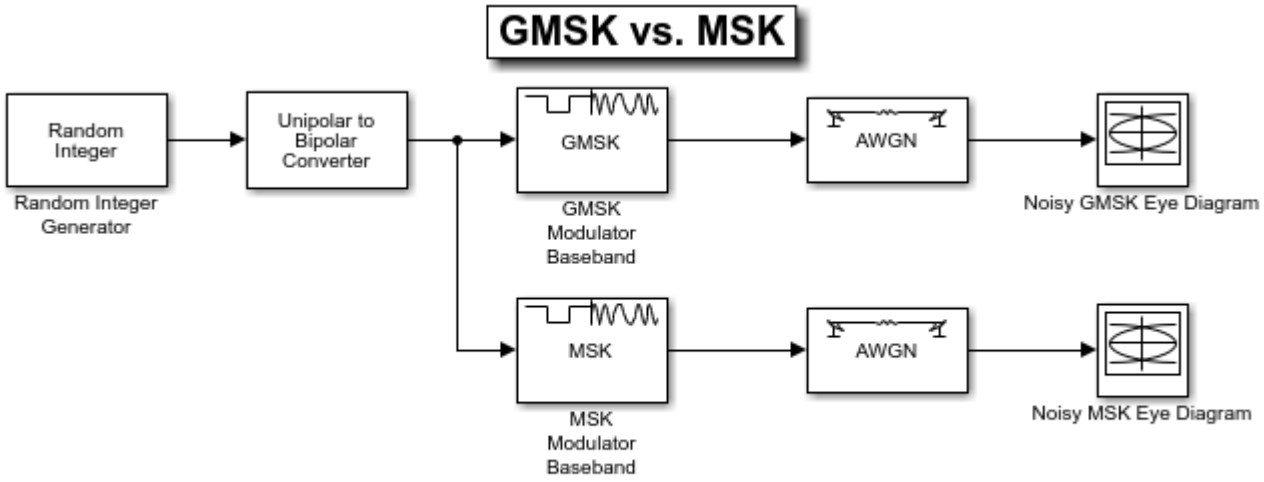
Compare eye diagram plots of a QPSK modulated signal and an MSK modulated signal. For QPSK the ideal sampling period is 1/2 sample, with sampling time for both in-phase and quadrature signal components at 0.5, 1.5, 2.5, ... For MSK, the ideal sample period is 1 sample, with sampling time at 0.5, 1.5, 2.5, ... for the in-phase signal component and 1, 2, 3, ... for the quadrature signal component.


Noisy QPSK Eye Diagram
File
Tools
View
Help
Eye Diagram of Noisy QPSK Signal
Real Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Imaginary Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Ready
T=10000.000


Noisy MSK Eye Diagram
File
Tools
View
Help
Eye Diagram of Noisy MSK Signal
Real Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Imaginary Amplitude
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Time
Ready
T=10000.000

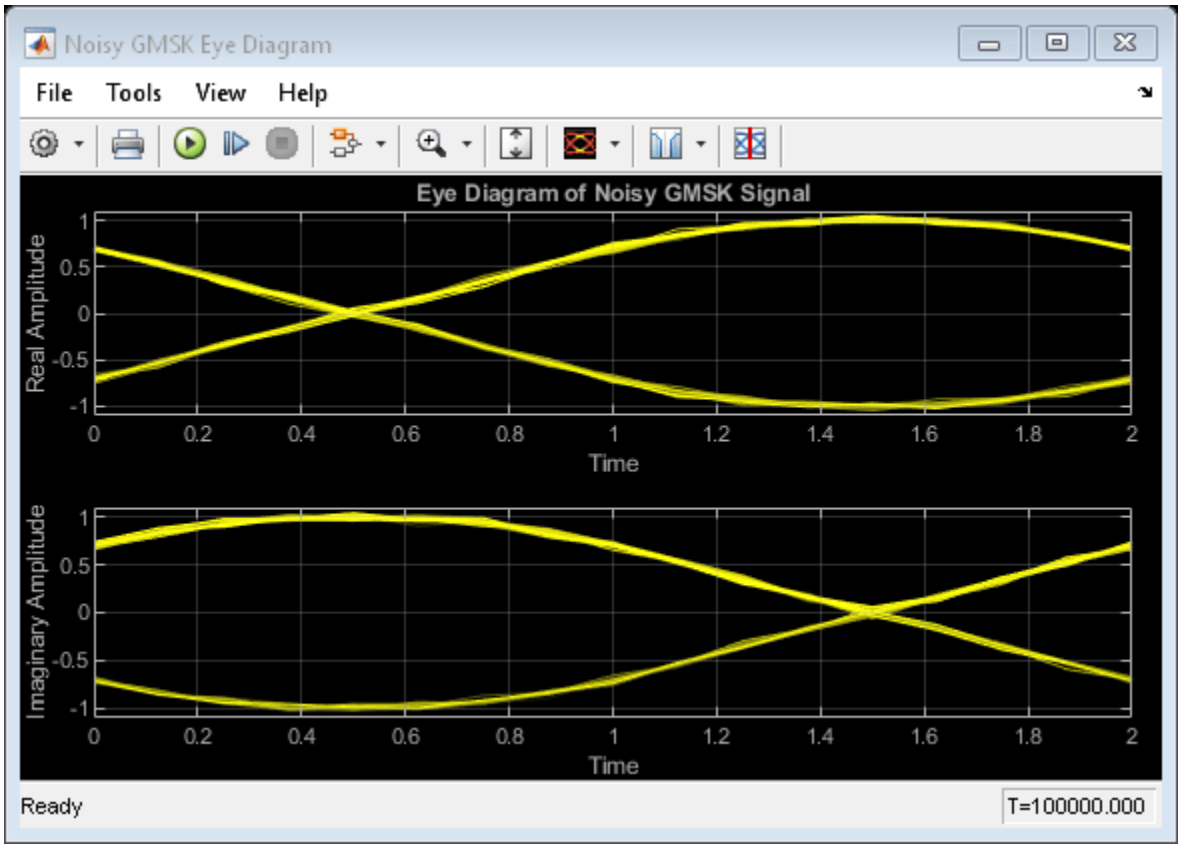
Compare GMSK and MSK Signals in Simulink

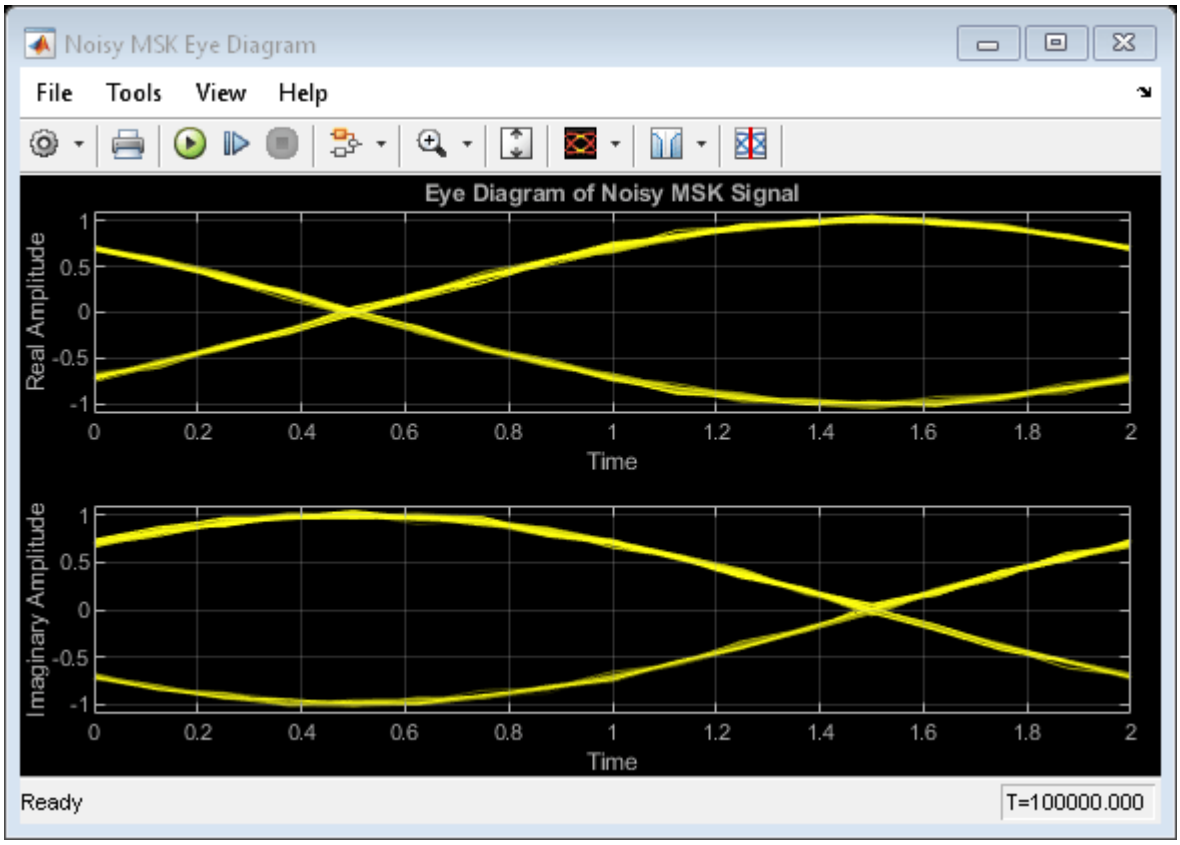
This model compares Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes.



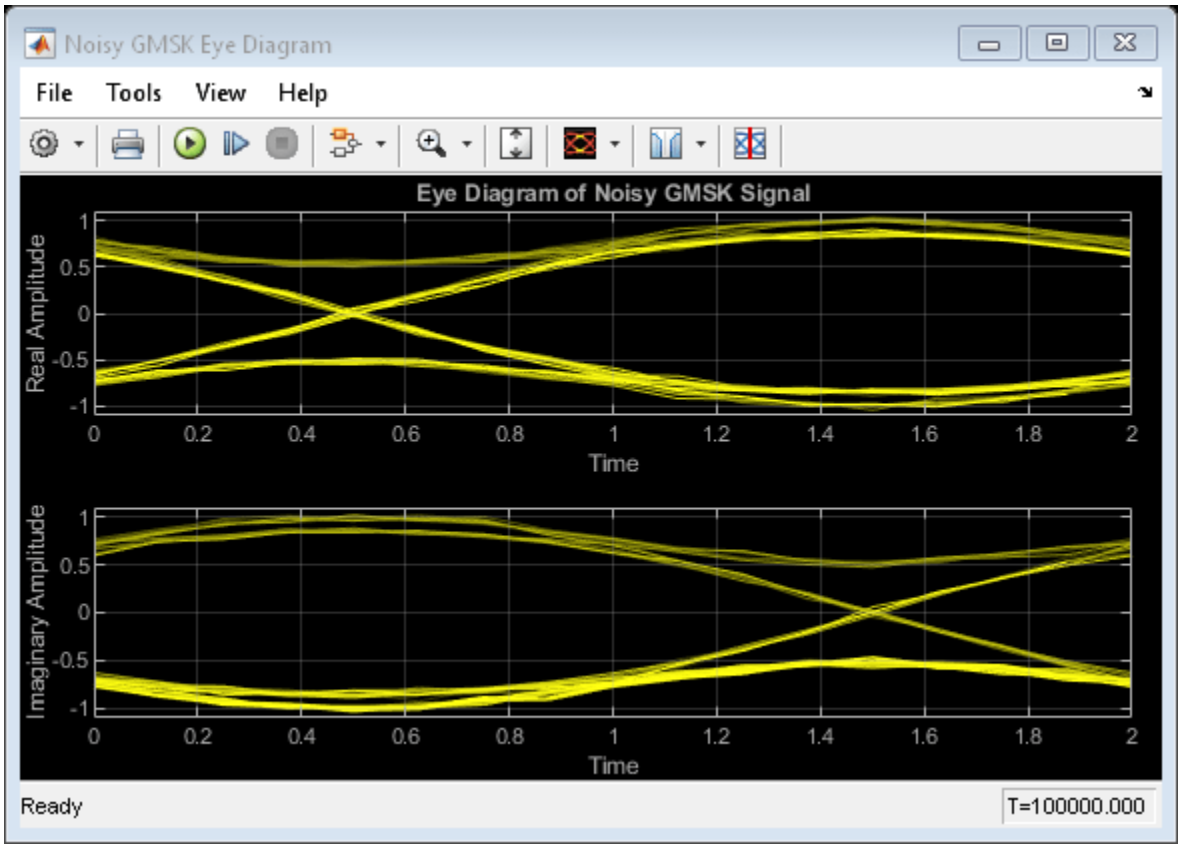
The Random Integer Generator block provides a source of uniformly distributed random integers in the range [0, M-1], where M is the constellation size of the GMSK or MSK signal. The Unipolar to Bipolar Converter block maps a unipolar input signal to a bipolar output consisting of integers between -(M-1) and +(M-1). The bipolar data is routed to separate paths. The top path applies GMSK modulation by using the GMSK Modulator Baseband block. The bottom path applies MSK modulation by using MSK Modulator Baseband block. Noise is added to both the GMSK and MSK signals by using AWGN Channel blocks. The Eye Diagram blocks are used to visualize eye diagrams of both signals.

The eye diagrams show the similarity between the GMSK and MSK signals when you set the initial pulse length of the GMSK Modulator Baseband block to 1.

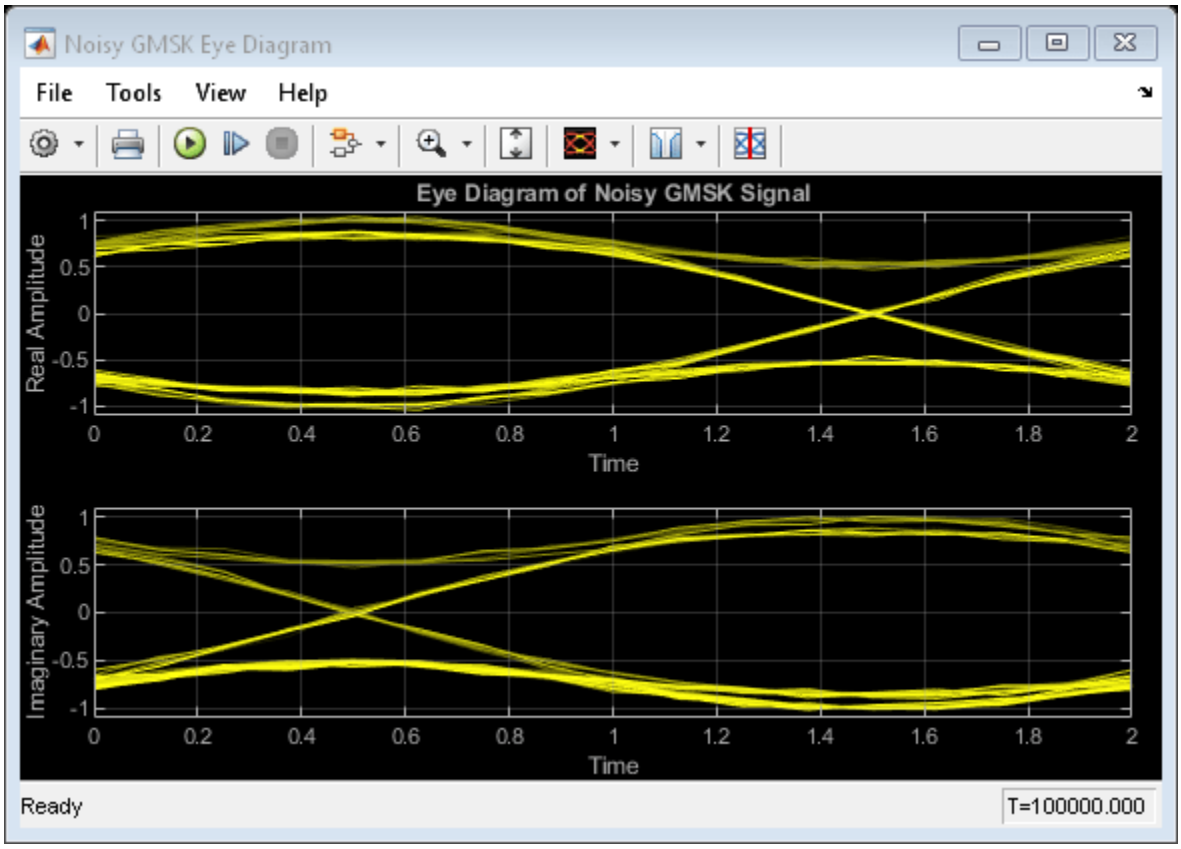

Noisy GMSK Eye Diagram
File Tools View Help
Eye Diagram of Noisy GMSK Signal
Real Amplitude
Time
Imaginary Amplitude
Time
Ready
T=100000.000



Set the initial pulse length in the GMSK modulator to 5 to view the difference that a partial response modulation has on the eye diagram. The increased pulse length results in an increase in the number of paths, showing that the CPM waveform depends on values of the previous symbols as well as the present symbol. Plot the eye diagram of the GMSK signal.



If you change the initial pulse length to an even number, such as 4, you should set initial phase offset of the GMSK modulator to pi/4 and the offset argument of the eye diagram 0 for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use scopes that enable you to view the phase of the signal, as described in the “CPM Phase Tree” on page 17-16 example.


Noisy GMSK Eye Diagram
File
Tools
View
Help
Eye Diagram of Noisy GMSK Signal
Real Amplitude
Time
Imaginary Amplitude
Time
Ready
T=100000.000

Gray Coded 8-PSK

In this section...

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

Structure of the Example

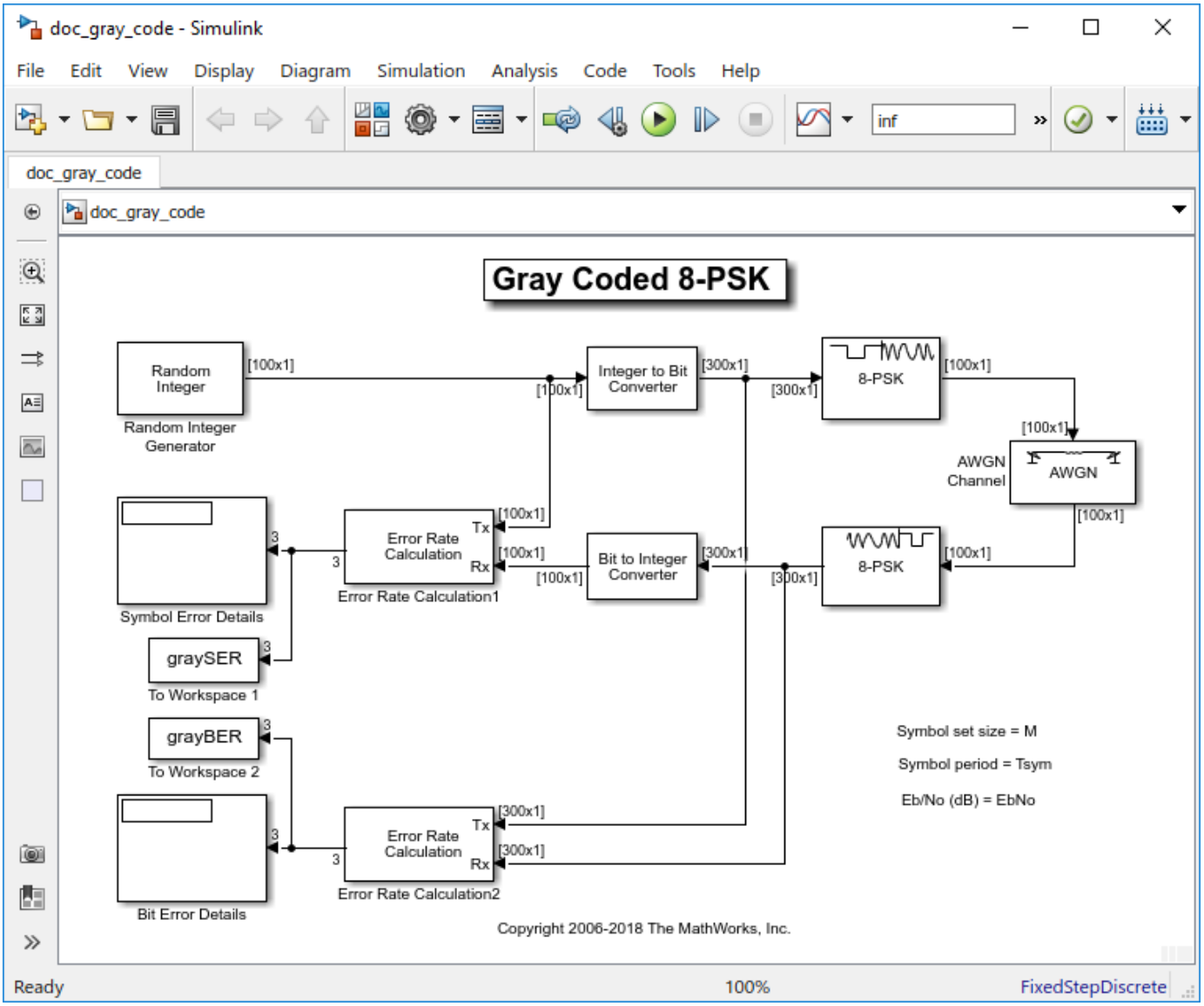
The example model includes these blocks:

- The Random Integer Generator block serves as the source, producing a sequence of integers.
- The Integer to Bit Converter block converts each integer into a corresponding binary representation.
- The AWGN Channel block adds white Gaussian noise to the modulated data.
- The M-PSK Demodulator Baseband block demodulates the corrupted data.
- The Bit to Integer Converter block converts each binary representation to a corresponding integer.
- One copy of the Error Rate Calculation block (labeled `Error Rate Calculation1` in this model) compares the demodulated integer data with the original source data, yielding symbol error statistics. The output of the Error Rate Calculation block is a three-element vector containing the calculated error rate, the number of errors observed, and the amount of data processed.
- Another copy of the Error Rate Calculation library block (labeled `Error Rate Calculation2` in this model) compares the demodulated binary data with the binary representations of the source data, yielding bit error statistics.

Gray-Coded M-PSK Modulation

In this model, the M-PSK Modulator Baseband block:

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



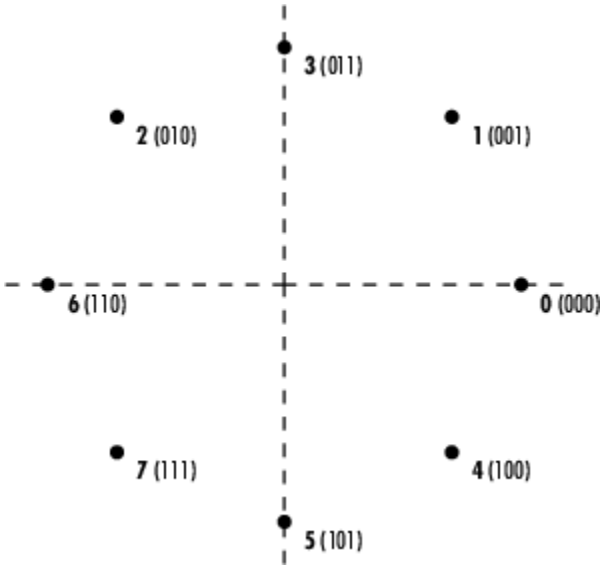
The table indicates which binary representations in the input correspond to which phasors in the output. The second column of the table is an intermediate representation that the block uses in its computations.

Modulator Input	Gray-Coded Ordering	Modulator Output
`000`	0	$\exp(0) = 1$
`001`	1	$\exp(j\pi/4)$
`010`	3	$\exp(j3\pi/4)$
`011`	2	$\exp(j2\pi/4) = \exp(j\pi/2)$
`100`	7	$\exp(j7\pi/4)$
`101`	6	$\exp(j6\pi/4) = \exp(j3\pi/2)$
`110`	4	$\exp(j4\pi/4) = \exp(j\pi)$

Modulator Input	Gray-Coded Ordering	Modulator Output
111	5	exp($j5\pi/4$)

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

Modulator Output	Modulator Input
exp(0)	`000`
exp($j\pi/4$)	`001`
exp($j2\pi/4$) = exp($j\pi/2$)	`011`
exp($j3\pi/4$)	`010`
exp($j4\pi/4$) = exp($j\pi$)	`110`
exp($j5\pi/4$)	`111`
exp($j6\pi/4$) = exp($j3\pi/2$)	`101`
exp($j7\pi/4$)	`100`



Exploring the Example

You can analyze the data that the example produces to compare theoretical performance with simulation performance.

The theoretical symbol error probability of MPSK is

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

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

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

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

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

Simulation Results

To test the Gray code modulation scheme in this model, simulate the graycode model for a range of E_b/N_0 values. If you want to study bit error rates but not symbol error rates, then you can use the `bertool` graphical user interface as described in “Use Bit Error Rate Analysis App” on page 23-12.

The rest of this section studies both the bit and symbol error rates and hence does not use `bertool`.

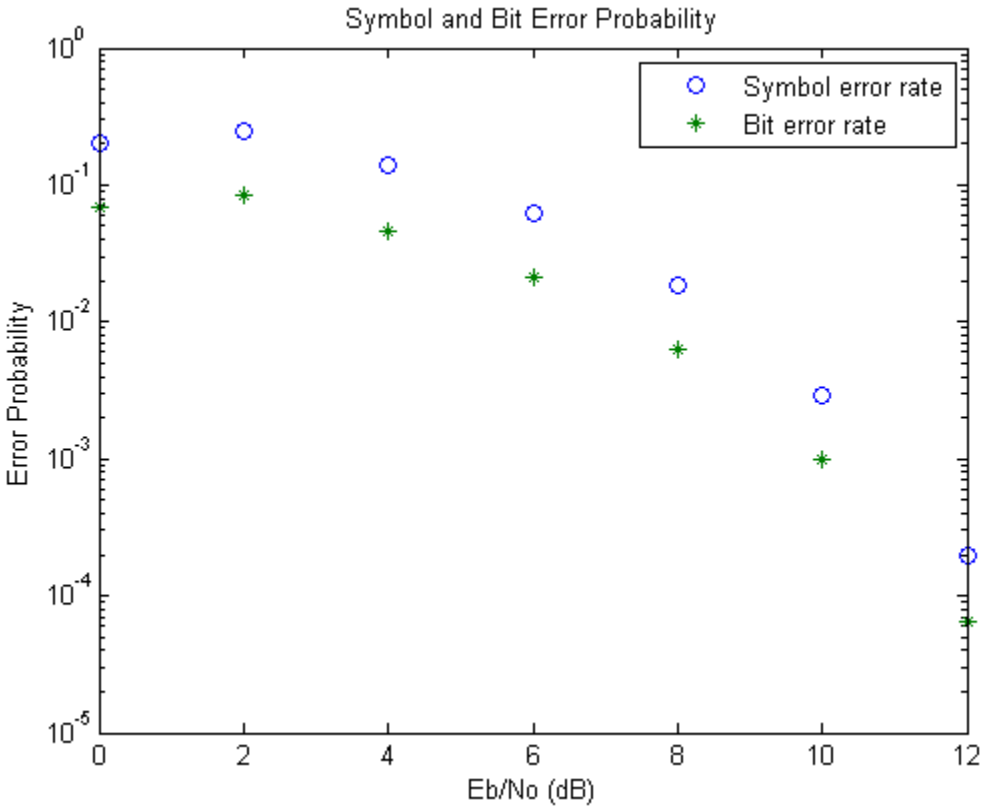
Because increasing the value of E_b/N_0 lowers the number of errors produced, the length of each simulation must be increased to ensure that the statistics of the errors remain stable.

Using the `sim` command to run a Simulink simulation from the MATLAB command window, the following code generates data for symbol error rate and bit error rate curves. It considers E_b/N_0 values in the range 0 dB to 12 dB, in steps of 2 dB.

```
M       = 8;
Tsym    = 0.2;
BERVec  = [];
SERVec  = [];
EbNoVec = [0:2:12];
for n   = 1:length(EbNoVec);
    EbNo = EbNoVec(n);
    sim('doc_gray_code')  ;
    SERVec(n,:) = graySER;
    BERVec(n,:) = grayBER;
end;
```

After simulating for the full set of E_b/N_0 values, you can plot the results using these commands:

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



Comparison with Pure Binary Coding and Theory

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

Soft Decision GMSK Demodulator

In this section...

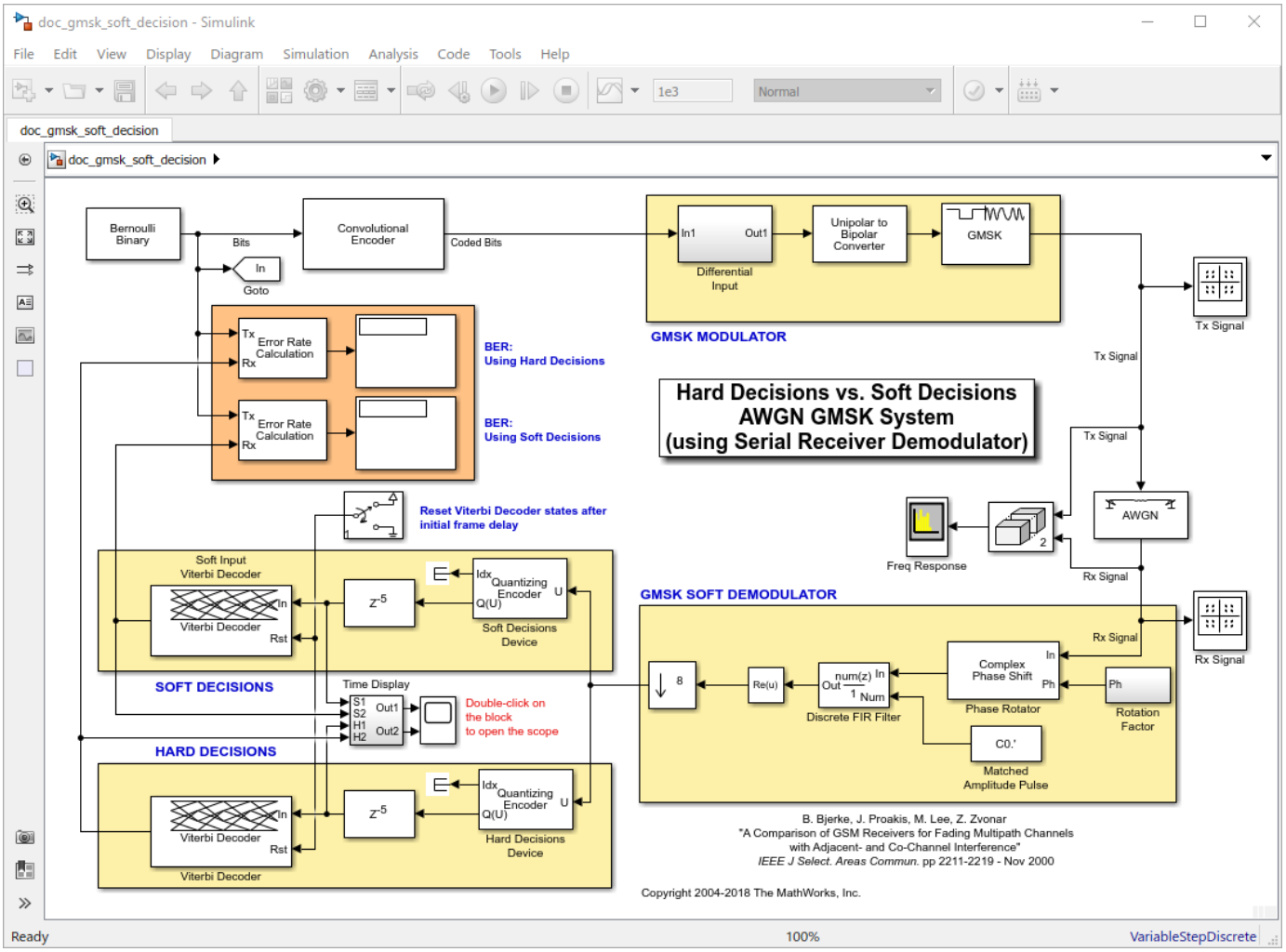
This model shows a system that includes convolutional coding and GMSK modulation. The receiver in this model includes two parallel paths, one that uses soft decisions and another that uses hard decisions. The model uses the bit error rates for the two paths to illustrate that the soft decision receiver performs better. This is to be expected, because soft decisions enable the system to retain more information from the demodulation operation to use in the decoding operation.

Structure of the Example

The example model, `doc_gmsk_soft_decision`, transmits and receives a coded GMSK signal.

The key components are:

- A Bernoulli Binary Generator block, which generates binary numbers.
- A Convolutional Encoder block, which encodes the binary numbers using a rate 1/2 convolutional code.
- A GMSK modulator section, which computes the logical difference between successive bits and modulates the result using the GMSK Modulator Baseband block.
- A **GMSK soft demodulator** section that implements the detector design proposed in [1], called a serial receiver. This section of the model produces a noisy bipolar signal. The section labeled **Soft Decisions** uses an eight-region partition in the Quantizing Encoder block to prepare for 3-bit soft-decision decoding using the Viterbi Decoder block. The section labeled **Hard Decisions** uses a two-region partition to prepare for hard-decision Viterbi decoding. Using a two-region partition here is equivalent to having the demodulator make hard decisions. In each decoding section, a Delay block aligns codeword boundaries with frame boundaries so that the Viterbi Decoder block can decode properly. This is necessary because the combined delay of other blocks in the system is not an integer multiple of the length of a codeword.
- A pair of Error Rate Calculation blocks, as well as Display blocks that show the BER for the system with each type of decision.



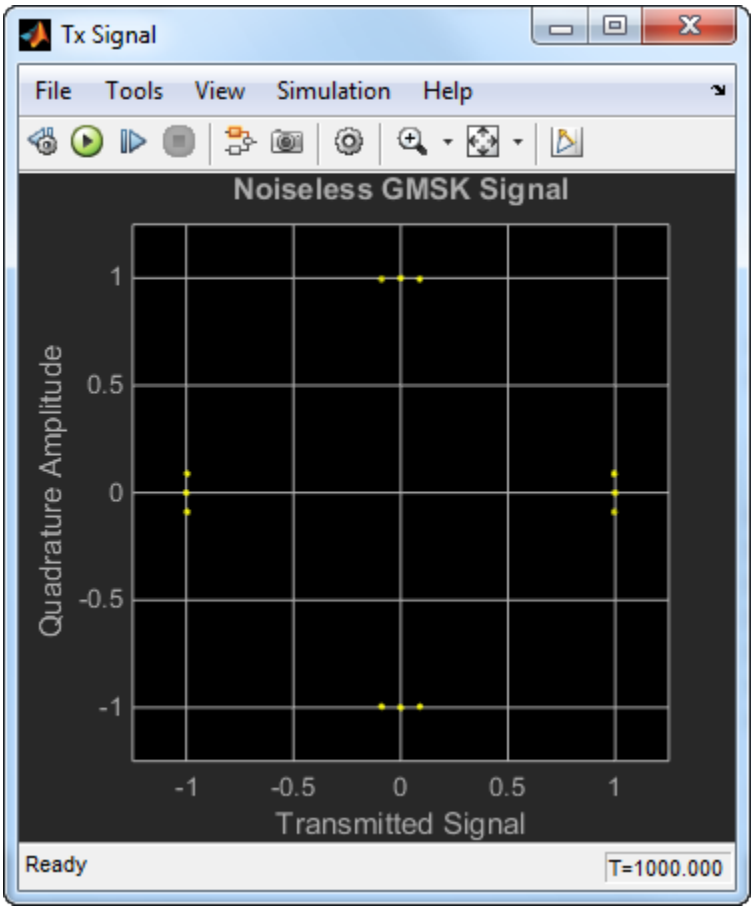
The Serial GMSK Receiver

The serial GMSK receiver is based on the fact that GMSK can be represented as a combination of amplitude pulses [2] - [3], and can, therefore, be demodulated with a matched filter. The GMSK waveform used in this model has a BT product of 0.3 and a frequency pulse length of 4 symbols. As such, it can be represented by eight different amplitude pulses, which are shown in Figure 2 of [3]. The matched filter in this model uses only the largest pulse of the eight, because of its simplicity of implementation. That same simplicity, however, yields BER performance that is inferior to the more traditional Viterbi-based demodulator.

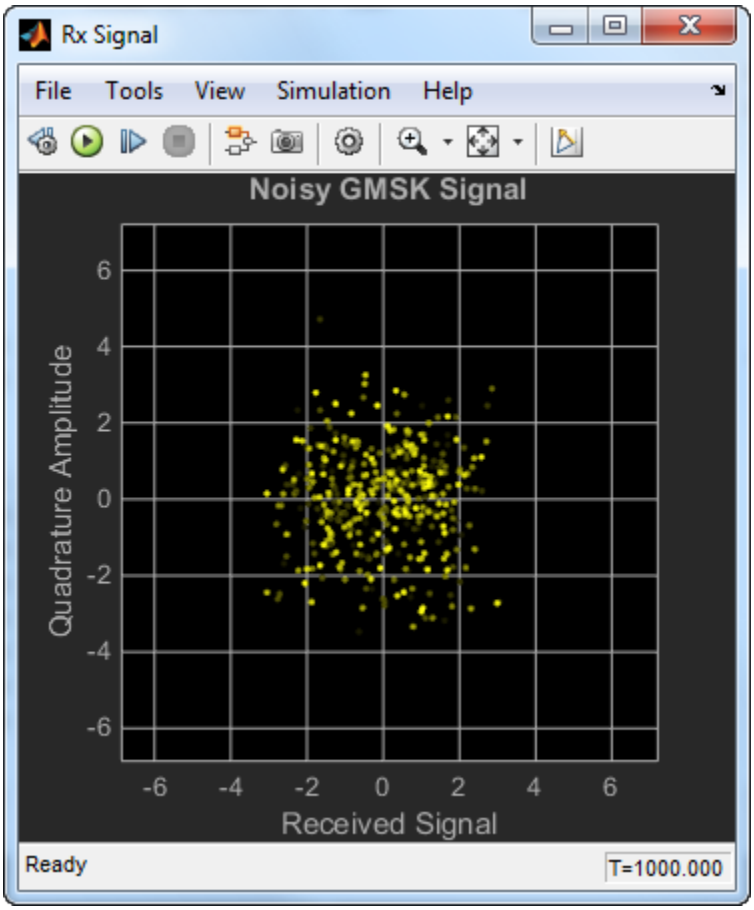
Results and Displays

The example model includes these visualizations to illustrate its performance:

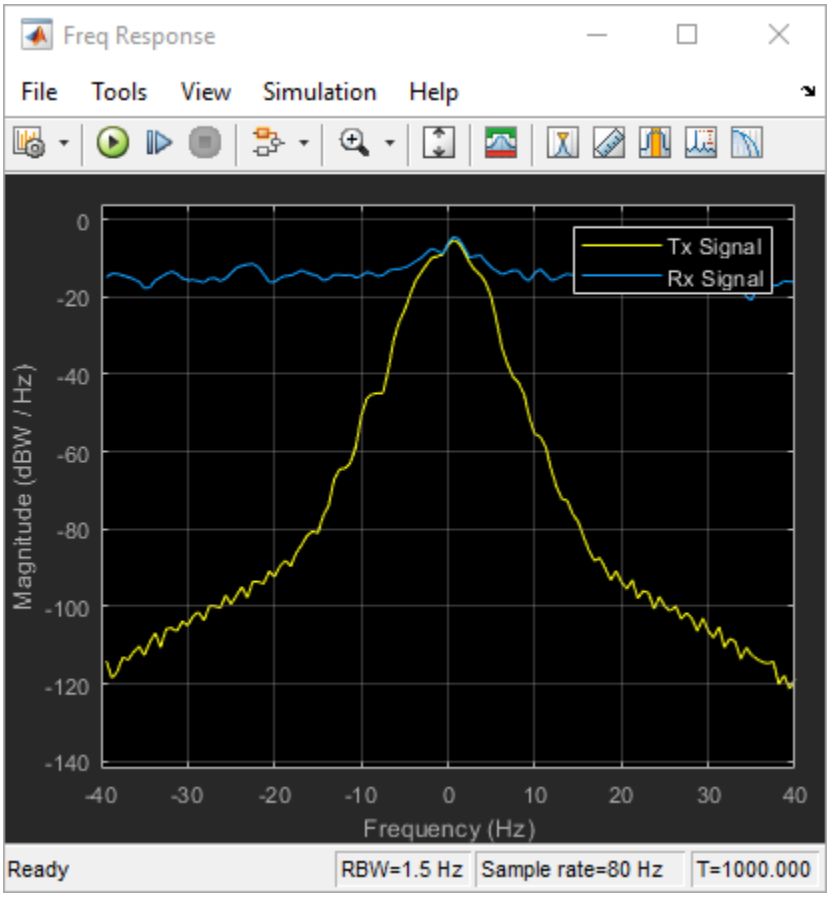
- The Display blocks illustrate that the soft decision receiver performs better (that is, has a smaller BER) than the hard decision receiver.
- The Tx Signal window shows the scatter plot of the signal before the AWGN channel.



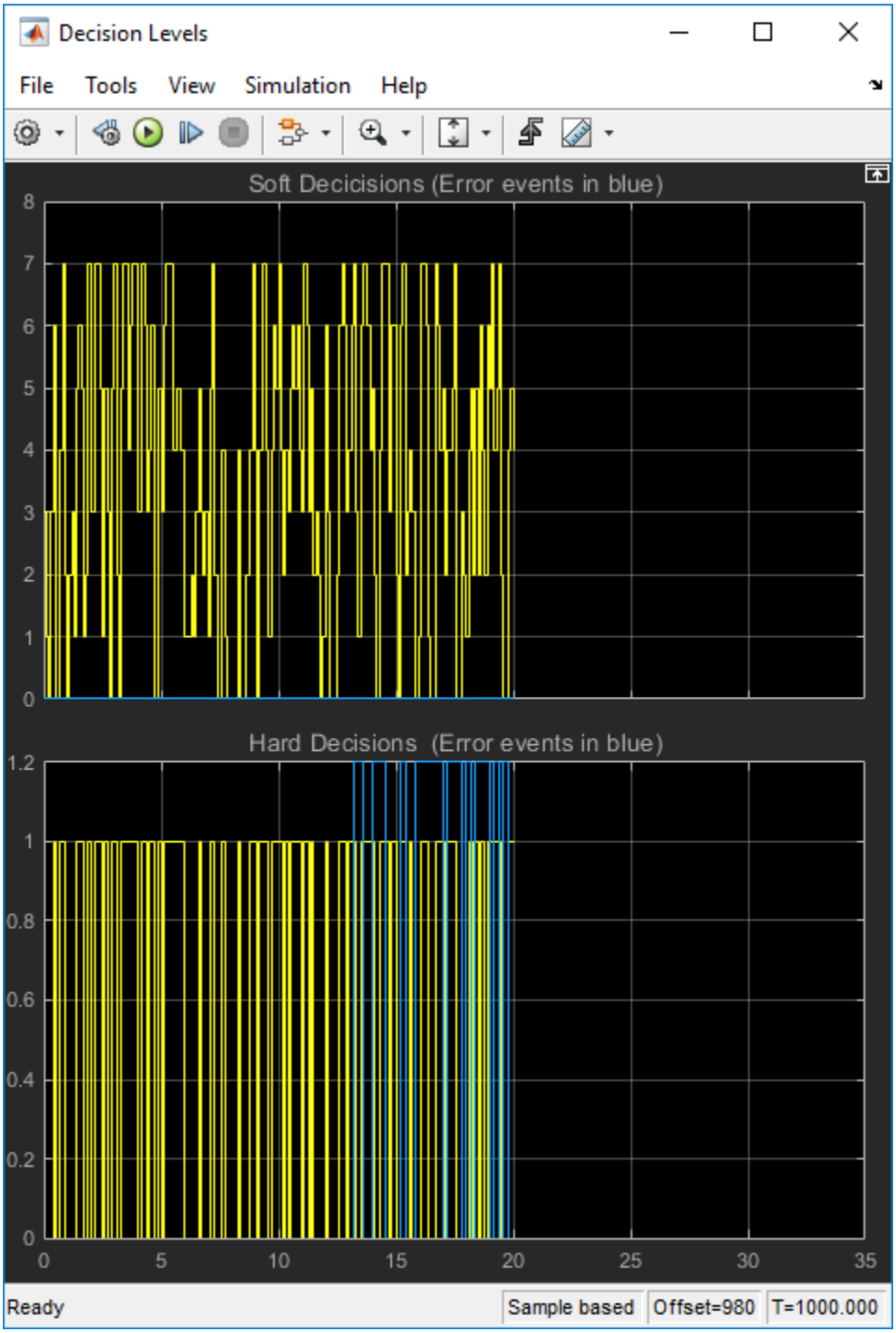
- The Rx Signal window shows the scatter plot of the signal after the AWGN channel.



- The Freq Response window shows the frequency response of the GMSK signal before and after the AWGN channel.



- The Decision Levels window shows, in yellow, the various soft decision levels in the top plot and the binary hard decisions in the bottom plot. This window also indicates, in blue, when errors occur.



References

[1] Bjerke, B., J. Proakis, M. Lee, and Z. Zvonar, "A Comparison of GSM Receivers for Fading Multipath Channels with Adjacent- and Co-Channel Interference," IEEE J. Select. Areas Commun., Nov. 2000, pp. 2211-2219.

[2] Laurent, Pierre, "Exact and Approximate Construction of Digital Phase Modulations by Superposition of Amplitude Modulated Pulses (AMP)," IEEE Trans. Comm., Vol. COM-34, No. 2, Feb. 1986, pp. 150-160.

[3] Jung, Peter, "Laurent's Representation of Binary Digital Continuous Phase Modulated Signals with Modulation index 1/2 Revisited", IEEE Trans. Comm., Vol. COM-42, No. 2/3/4, Feb./Mar./Apr. 1994, pp. 221-224.

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.

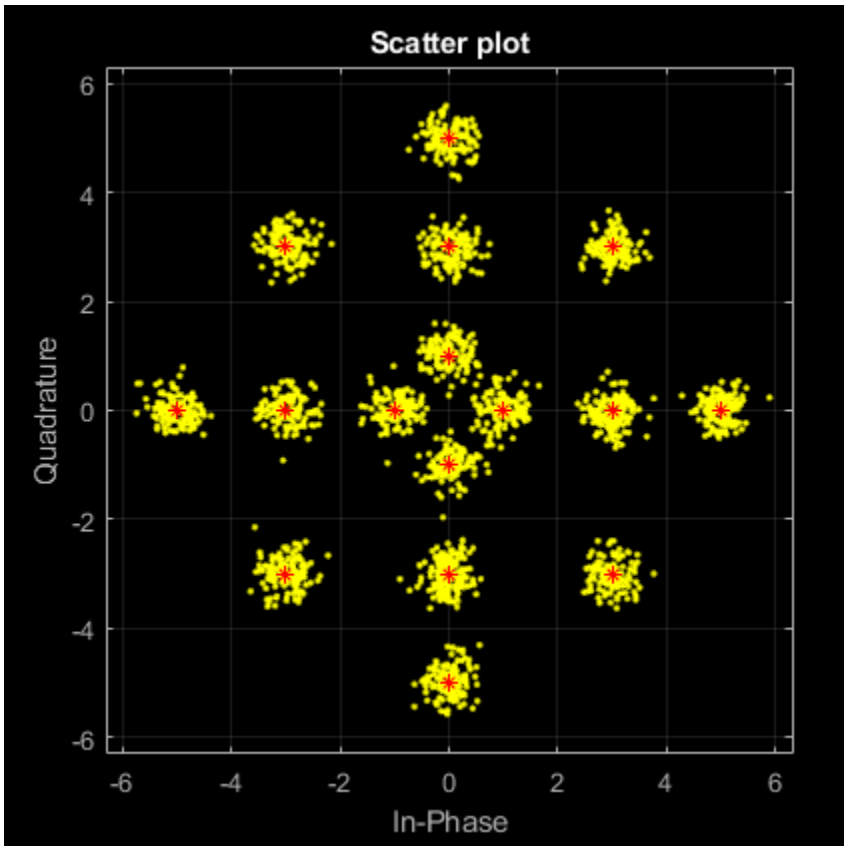
```
modData = genqammod(data,c);
```

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
```

MSK

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

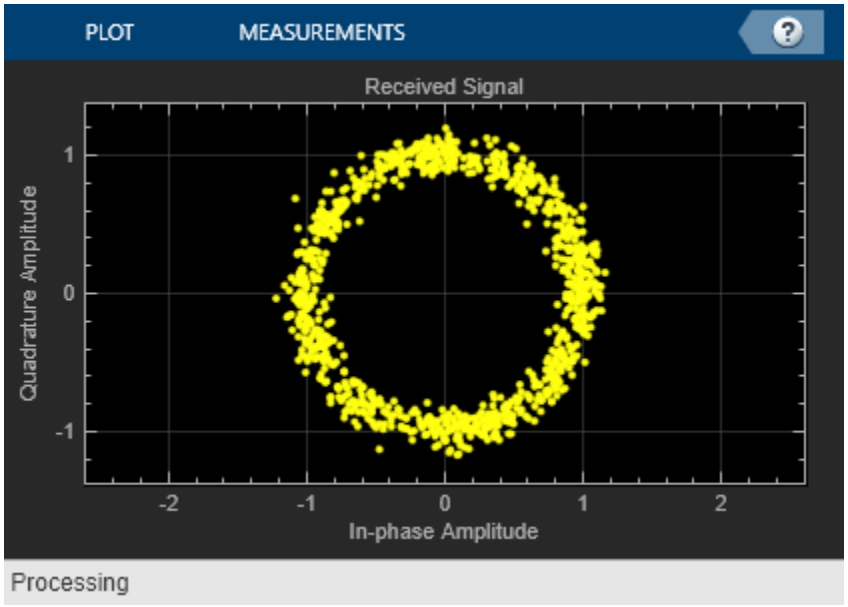
The simulation modulates data using MSK modulation. The modulated symbols pass through the channel model, which applies timing delay, carrier frequency and phase shift, and additive white Gaussian noise. The receiver performs timing phase and carrier frequency and phase recovery. Finally, the signal symbols are demodulated and the bit error rate is calculated. The `plotResultsMSKSignalRecoveryEx` script generates scatter plots in this order to show these effects:

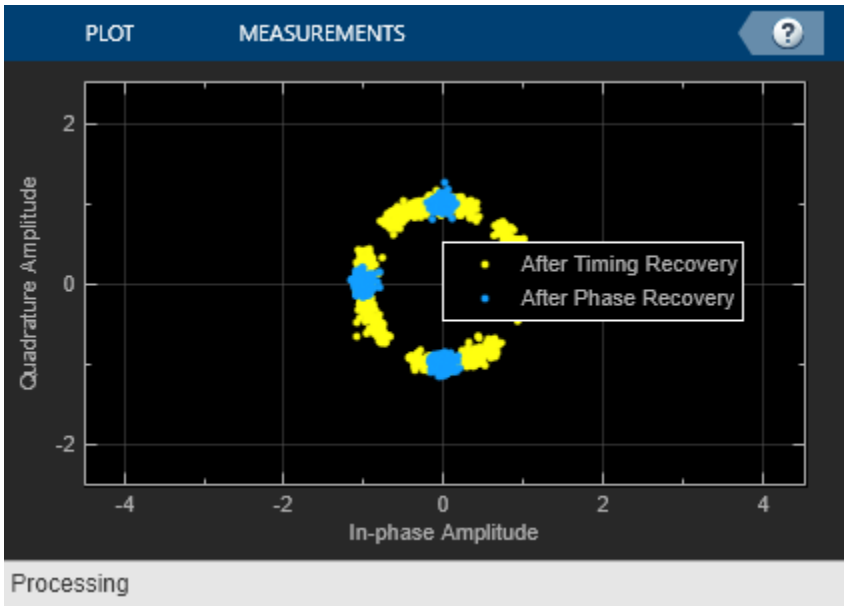
1. Channel impairments
2. Timing synchronization
3. Carrier synchronization

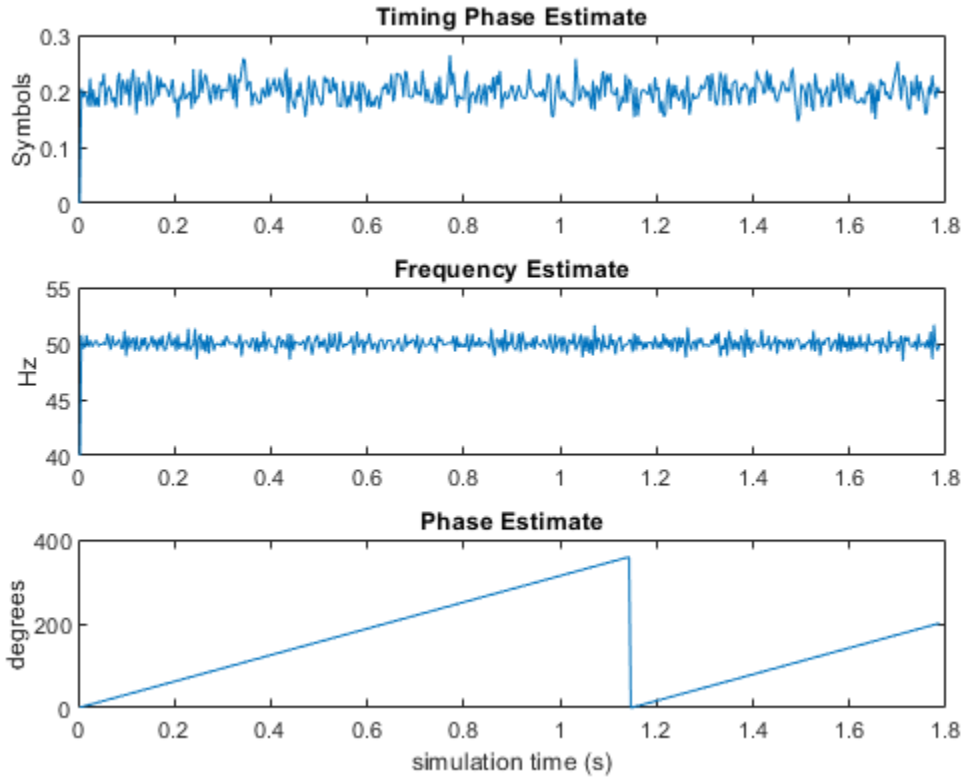
At the end of the simulation, the example displays the timing phase, frequency, and phase estimates as a function of simulation time.

```
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
```


PLOT
MEASUREMENTS
Received Signal
Quadrature Amplitude
In-phase Amplitude
1
0
-1
-2
-1
0
1
2
Processing


PLOT
MEASUREMENTS
Quadrature Amplitude
In-phase Amplitude
After Timing Recovery
After Phase Recovery
2
0
-2
-4
-2
0
2
4
Processing



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`

MSK Signal Recovery

In this section...

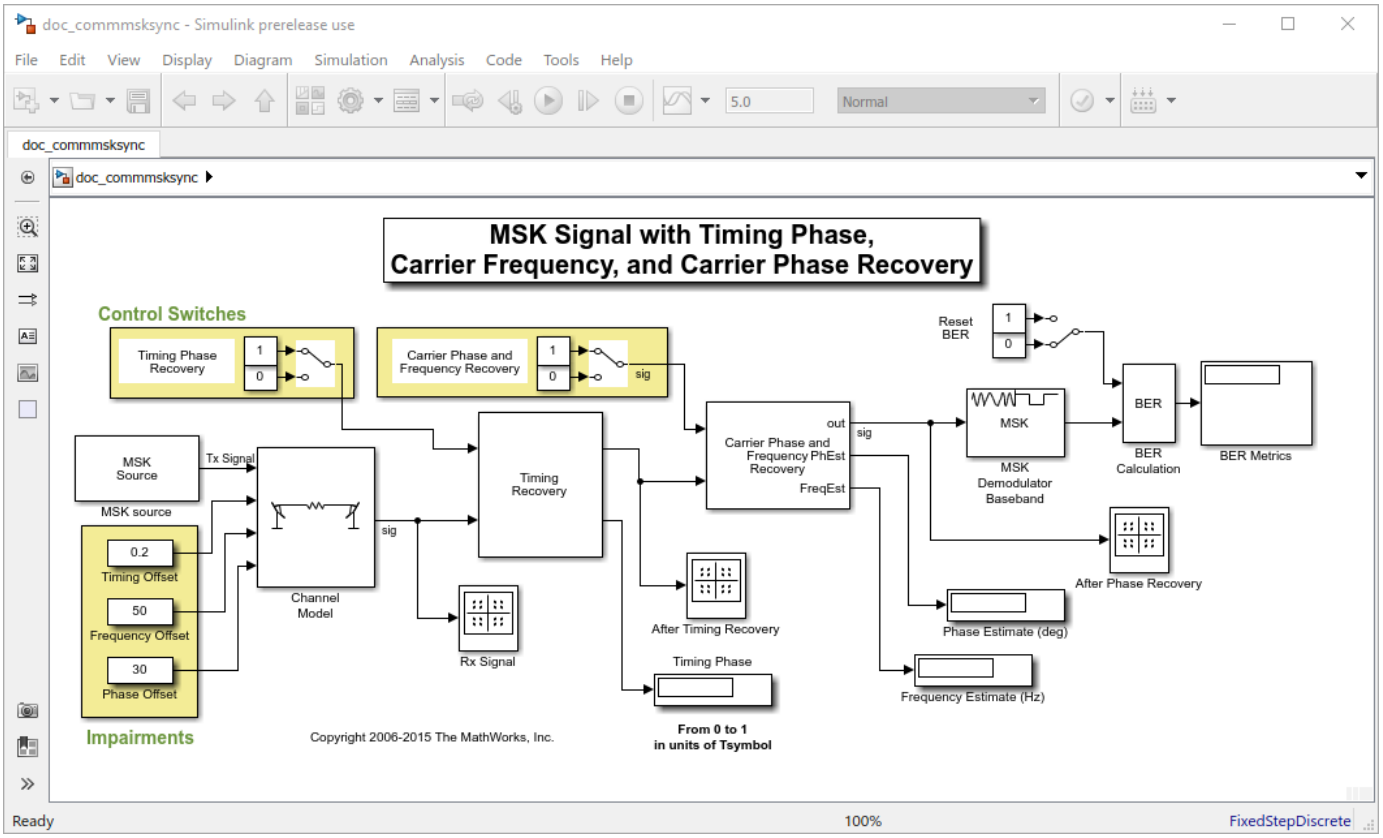
This model shows how channel impairments such as timing phase offset, carrier frequency offset, and phase offset for a minimum shift keying (MSK) signal are modeled. The model uses blocks from the Synchronization library to recover the signal. To open the model, type `doc_commmsksync` at the MATLAB command line.

Exploring the Model

The example models an MSK transmitted signal undergoing channel impairments, including these components:

1. An MSK signal source that uses the Bernoulli Binary Generator block to output equiprobable symbols and modulates the symbols using an MSK Modulator Baseband block
2. A channel model that incorporates independently variable offsets in the timing phase, frequency, and phase. The channel model also includes the AWGN Channel block
3. Signal recovery, consisting of:
   - Timing recovery using the MSK-Type Signal Timing Recovery block
   - Carrier frequency and phase recovery using the Carrier Synchronizer block
4. An MSK Demodulator Baseband block
5. Blocks that compute and display the system's bit error rate (BER)

When you load the model, it also initializes some parameters that several blocks share.

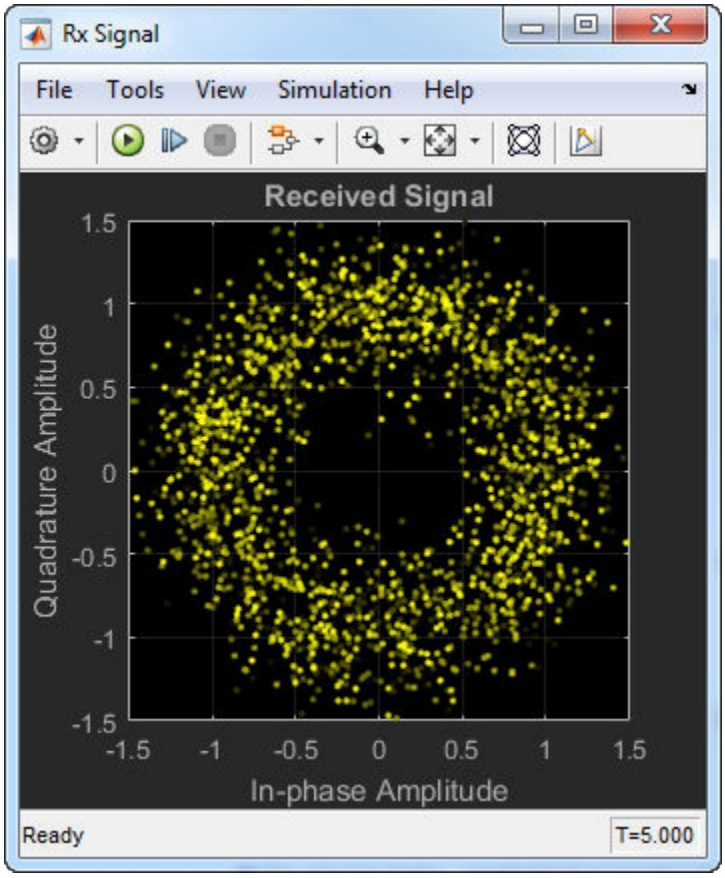


Results and Displays

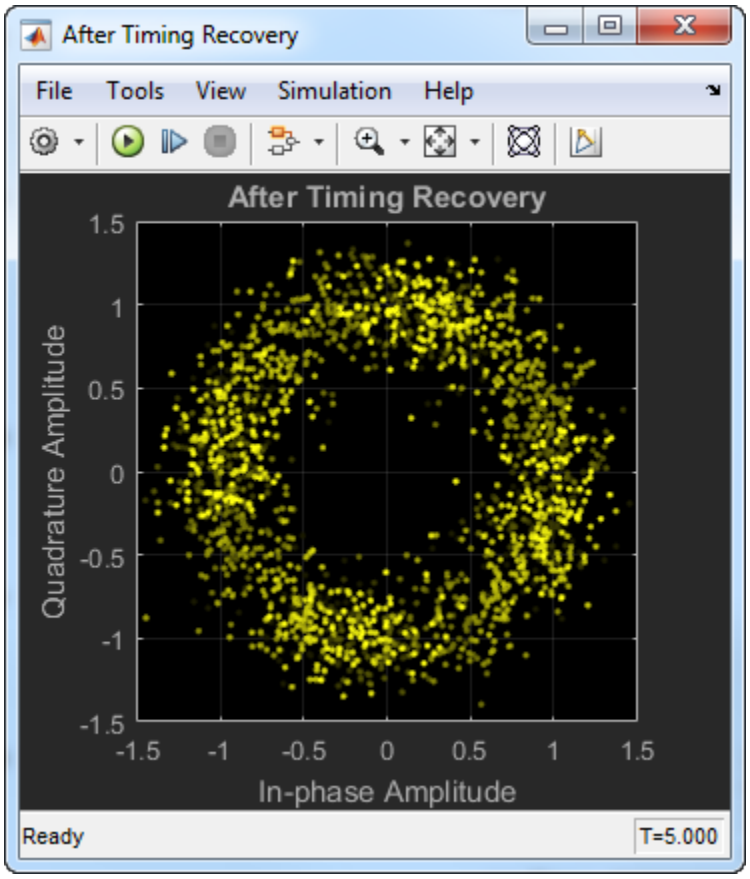
When you run the simulation, the displays show the estimated values for the impairments as well as the BER metrics. Because the Carrier Synchronizer block performs both frequency and phase correction, the display of estimated phase offset may fluctuate rapidly. The display labeled BER Metrics shows a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed.

You can view the MSK signal via the Constellation Diagram blocks at the different stages. This provides a compelling visual rendition of the recovery algorithms in action, especially as you turn the algorithms on and off using the two control switches.

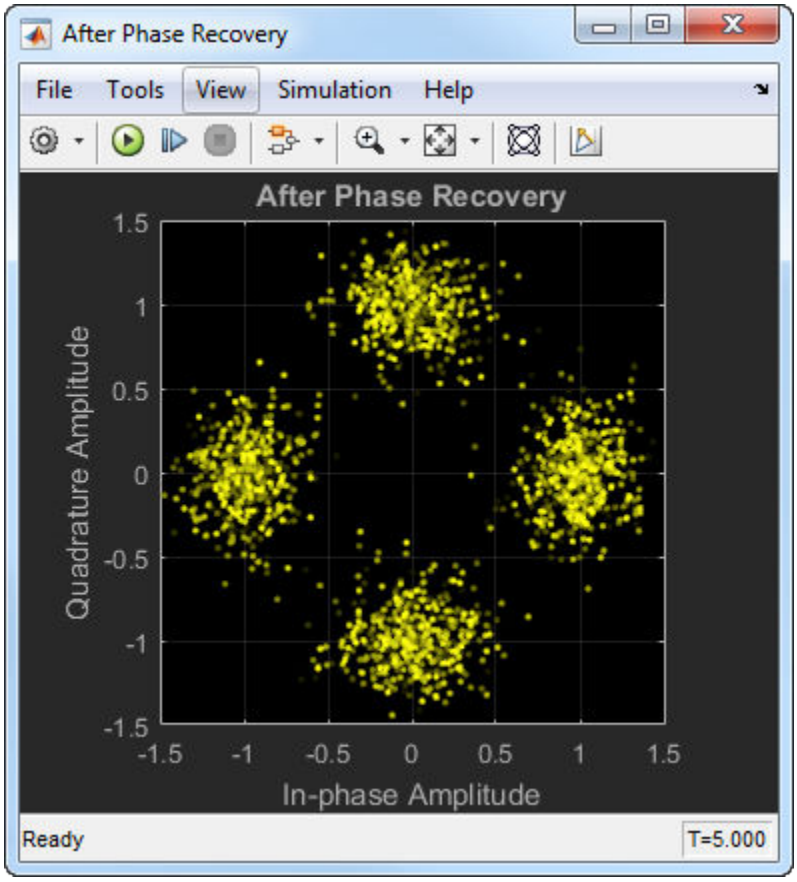
Scatter plot of received signal:



Scatter plot of signal after timing recovery:



Scatter plot of signal after carrier frequency and phase recovery:



You can also reset the BER computation after the signal has reached a steady state.

Experimenting with the Example

The example is designed so that you can vary the impairments independently while the simulation is running. You can also use the toggle switches to turn the recovery schemes on and off while the simulation is running, and then see the effects on the scatter plots.

Further items to investigate include:

- Set the frequency offset to `0` and observe the displayed signal constellations and estimated phase offset.
- Observe that the Carrier Synchronizer block is set for a QPSK constellation with a phase offset of 0°.
- To see how the timing offset is tracked, replace the Constant block with a Sine Wave block. Vary the offset between 0 and 1 over the duration of the simulation.
- Vary the error update gain of the MSK-Type Signal Timing Recovery block to assess its ability to track constant and time-varying offsets. To access the block, open the *Timing Recovery* subsystem and then open the *Timing Recovery Algorithm* subsystem.

Reed-Solomon Coding

Reed-Solomon Coding

See Also

Reed-Solomon Coding with Erasures, Punctures, and Shortening in Simulink

This model shows how to configure Reed-Solomon (RS) codes to perform block coding with erasures, punctures, and shortening.

RS decoders can correct both errors and erasures. The erasures can be generated by a receiver that identifies the most unreliable symbols in a given codeword. When a receiver erases a symbol, it replaces the symbol with a zero and passes a flag to the decoder indicating that the symbol is an erasure, not a valid code symbol.

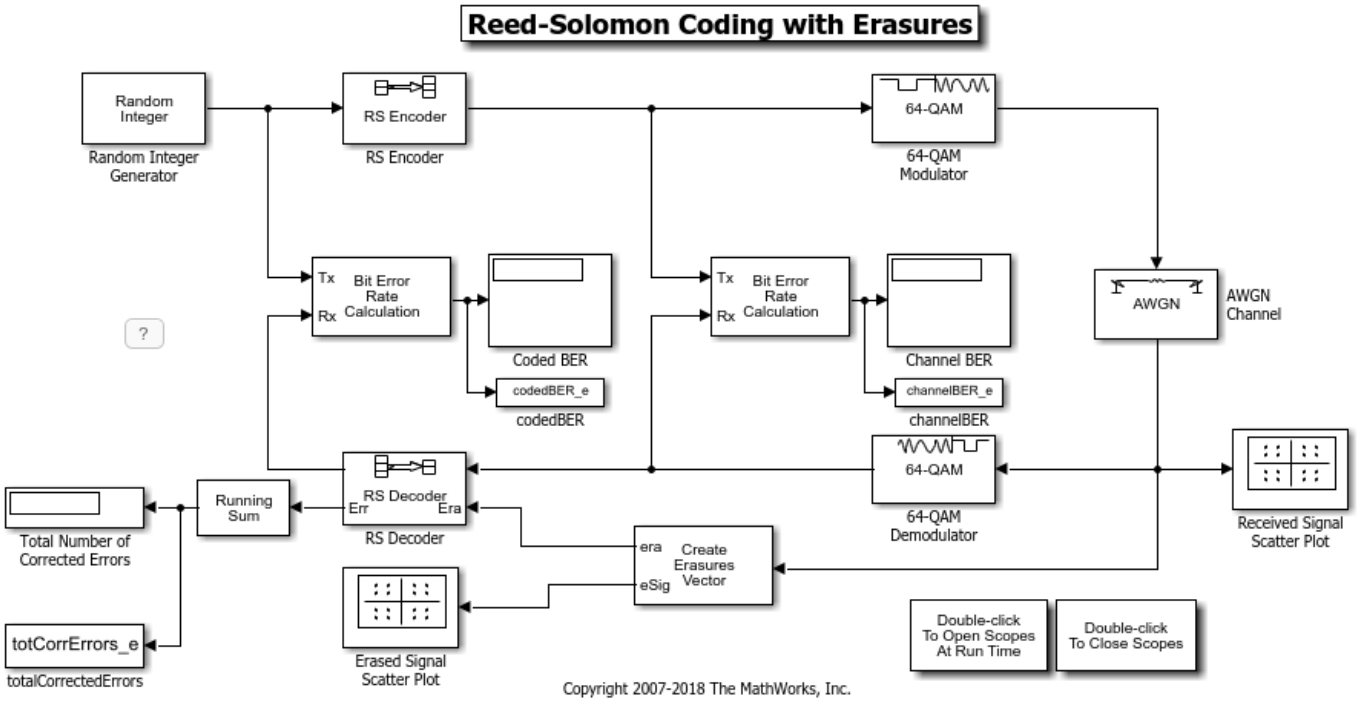
In addition, an encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures in exactly the same way when it decodes.

Puncturing has the added benefit of making the code rate a bit 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 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.

Decoding with Receiver Generated Erasures

This example shows a (63,53) RS code operating in concert with a 64-QAM modulation scheme. Since the code can correct (63-53)/2 = 5 errors, it can alternatively correct (63-53) = 10 erasures. For each demodulated codeword, the receiver determines the six least reliable symbols by finding the symbols within a decision region that are nearest to a decision boundary. It then erases those symbols. The RSCodingErasuresExample model is shown here.

```
model_e = 'RSCodingErasuresExample';
open_system(model_e);
close_system([model_e,'/Received Signal Scatter Plot'])
close_system([model_e,'/Erased Signal Scatter Plot'])
```



Simulation and Visualization with Erasures Only

Define system simulation parameters:

```
RS_TsUncoded = 1; % Sample time (s)
RS_n = 63; % Codeword length
RS_k = 53; % Message length
RS_MQAM = 64; % QAM order
RS_numBitsPerSymbol = ... % 6 bits per symbol
    log2(RS_MQAM);
RS_sigPower = 42; % Assume points at +/-1, +/-3, +/-5, +/-7
RS_numErasures = 6; % Number of erasures
RS_EbNoUncoded = 15; % In dB
```

The system is simulated at an uncoded Eb/N0 of 15 dB. However, the coded Eb/N0 is reduced because of the redundant symbols added by the RS Encoder. Also, the period of each frame in the model remains constant at 53 seconds, corresponding to a sample time of 1 second at the output of the Random Integer Generator. Moreover, the symbol time at the output of the RS Encoder is reduced by a factor of the code rate, because 63 symbols are output over the frame time of 53 seconds. The AWGN Channel block accounts for this by using these parameters:

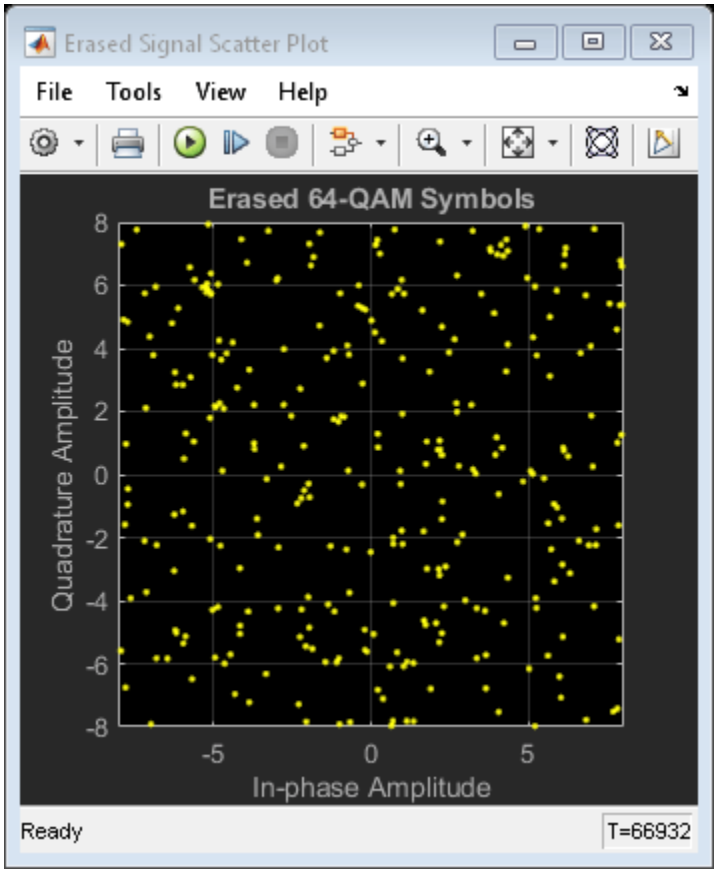
```
RS_EbNoCoded = RS_EbNoUncoded + 10*log10(RS_k/RS_n);
RS_TsymCoded = RS_TsUncoded * (RS_k/RS_n);
```

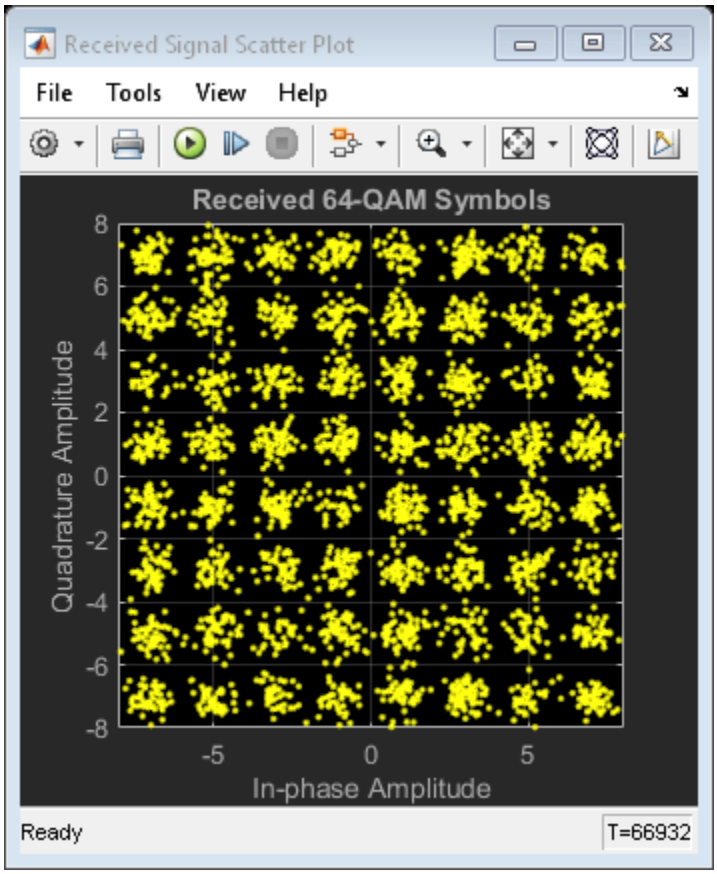
The receiver determines which symbols to erase by finding the 64-QAM symbols, per codeword, that are closest to a decision boundary. It deletes the six least reliable code symbols, which still allows the RS Decoder to correct (10-6)/2 = 2 errors per codeword.

Run the simulation and show the received symbols and those symbols that were erased.

```
sim(model_e)
open_system([model_e,'/Received Signal Scatter Plot'])
```

```
open_system([model_e,'/Erased Signal Scatter Plot'])
%
```





BER Performance with Erasures Only

Examine the BER performance at the output of the decoder. We set the stop time of the simulation to inf, then simulate until 100 bit errors are collected out of the RS Decoder. Display the total number of corrected errors, 64-QAM BER, and RS BER.

```
fprintf('Total number of corrected errors with\n')
fprintf('erasures: %d\n',totCorrErrors_e(1))
fprintf('64-QAM BER with\n')
fprintf('erasures: %s\n',channelBER_e(1))
fprintf('RS BER with\n')
fprintf('erasures: %s\n',codedBER_e(1))
```

```
Total number of corrected errors with
erasures: 6905
64-QAM BER with
erasures: 1.702521e-03
RS BER with
erasures: 2.589668e-06
```

Simulation with Erasures and Punctures

In addition to decoding receiver-generated erasures, the RS Decoder can correct encoder-generated punctures. The decoding algorithm is identical for the two cases, but the per-codeword sum of the punctures and erasures cannot exceed twice the error-correcting capability of the code. Consider the following model that performs decoding for both erasures and punctures.

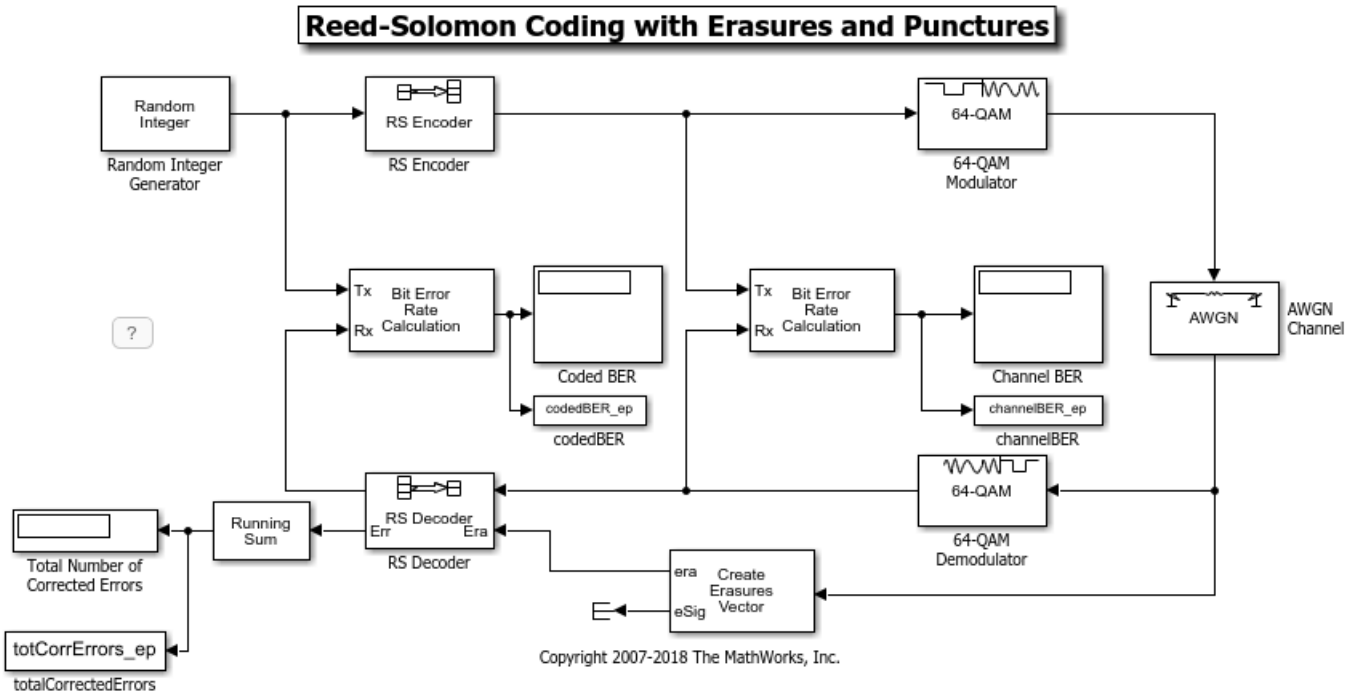
The same puncture vector is specified in both the encoder and decoder blocks. This example punctures two symbols from each codeword. Vector values of "1" indicate nonpunctured symbols, while values of "0" indicate punctured symbols. In the erasures vector, however, values of "1" indicate erased symbols, while values of "0" indicate nonerased symbols.

Several of the parameters for the AWGN Channel block are now slightly different, because the length of the codeword is now different from the previous example. The block accounts for the size difference with the following code:

```
RS_numPuncs = 2;
RS_EbNoCoded = RS_EbNoUncoded + 10*log10(RS_k / (RS_n - RS_numPuncs));
RS_TsymCoded = RS_TsUncoded * (RS_k / (RS_n - RS_numPuncs));
```

We simulate the model, RSCodingErasuresPunctExample.mdl, collecting 1000 errors out of the RS Decoder block. Due to puncturing, the signal dimensions out of the encoder are 61-by-1, rather than 63-by-1 in the model with no puncturing. The Create Erasures Vector subsystem must also account for the size differences as it creates a 61-by-1 erasures vector. The RSCodingErasuresPunctExample model is shown here.

```
model_ep = 'RSCodingErasuresPunctExample';
open_system(model_ep);
```




```
sim(model_ep)
```

BER Performance with Erasures and Punctures

Compare the BERs for erasures decoding with and without puncturing.

The BER out of the 64-QAM Demodulator is slightly better in the punctured case, because the Eb/N0 into the demodulator is slightly higher. However, the BER out of the RS Decoder is much 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. Display the total number of corrected errors, 64-QAM BER, and RS BER for the RS codes with erasures and punctures.

```
fprintf('Total number of corrected errors with\n')
fprintf('              erasures: %d\n',totCorrErrors_e(1))
fprintf('erasures and punctures: %d\n',totCorrErrors_ep(1))
fprintf('64-QAM BER with\n')
fprintf('              erasures: %s\n',channelBER_e(1))
fprintf('erasures and punctures: %s\n',channelBER_ep(1))
fprintf('RS BER with\n')
fprintf('              erasures: %s\n',codedBER_e(1))
fprintf('erasures and punctures: %s\n',codedBER_ep(1))
```

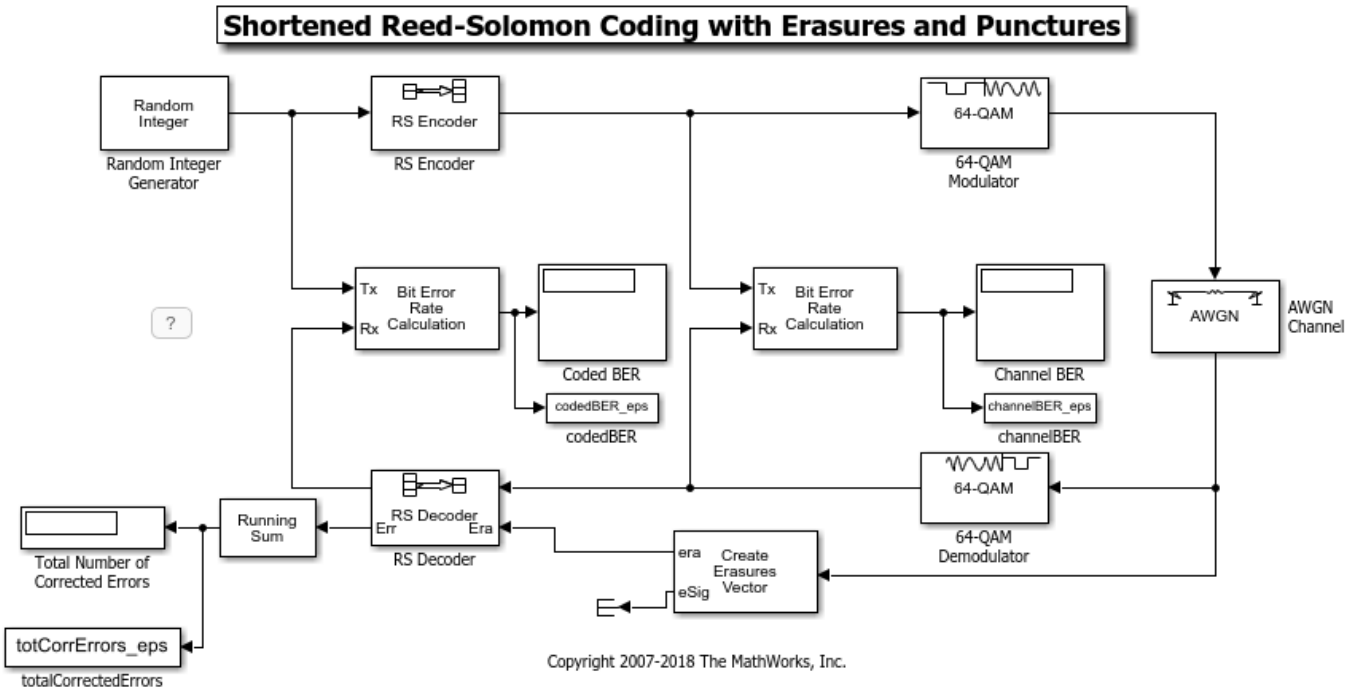
```
Total number of corrected errors with
              erasures: 6905
erasures and punctures: 1960
64-QAM BER with
              erasures: 1.702521e-03
erasures and punctures: 1.475314e-03
RS BER with
              erasures: 2.589668e-06
erasures and punctures: 5.627692e-05
```

Specifying a Shortened Code

Shortening a block code removes symbols from its message portion, where puncturing removes symbols from its parity portion. You can incorporate both techniques with the RS encoder and decoder blocks.

For example, to shorten a (63,53) code to a (53,43) code, you can simply enter 63, 53 and 43 for n, k, and s respectively, in the encoder and decoder block masks. The RSCodingErasuresPunctShortExample model is shown here.

```
model_eps = 'RSCodingErasuresPunctShortExample';
open_system(model_eps);
```



Simulation with Erasures, Punctures, and Shortening

Because shortening alters the code rate much like puncturing does, the AWGN parameters must be changed again. The AWGN Channel block accounts for this with the following code:

```
RS_EbNoCoded = RS_EbNoUncoded + ...
    10*log10(RS_s  / (RS_n - RS_k + RS_s - RS_numPuncs));
RS_TsymCoded = RS_TsUncoded * RS_s / (RS_n - RS_k + RS_s - RS_numPuncs);
```

We simulate the model, once again collecting 1000 errors out of the RS Decoder block. Note that the signal dimensions out of the RS Encoder are 26x1, due to 35 symbols of shortening and 2 symbols of puncturing. Once again, the Create Erasures Vector subsystem must also account for the size difference caused by the shortened code.

```
sim(model_eps)
```

BER Performance with Erasures, Punctures, and Shortening

Compare the BER performance for decoding with erasures only, with erasures and punctures, and with erasures, punctures, and shortening.

The BER out of the 64-QAM Demodulator is worse with shortening than it is without shortening. This is because the code rate of the shortened code is much lower than the code rate of the non-shortened code and therefore the coded Eb/N0 into the demodulator is worse with shortening. A shortened code has the same error correcting capability as non-shortened code for the same Eb/N0, but the reduction in Eb/N0 manifests in the form of a higher BER out of the RS Decoder with shortening than without. Compare the total number of corrected errors, 64-QAM BER, and RS BER for the RS codes with erasures, punctures, and shortening.

```
fprintf('Total number of corrected errors\n')
fprintf( ...
```

```
    '                             erasures: %d\n',totCorrErrors_e(1))
fprintf( ...
    '             erasures and punctures: %d\n',totCorrErrors_ep(1))
fprintf( ...
    'erasures, punctures, and shortening: %d\n',totCorrErrors_eps(1))
fprintf('64-QAM BER with\n')
fprintf('                               erasures: %s\n',channelBER_e(1))
fprintf('             erasures and punctures: %s\n',channelBER_ep(1))
fprintf('erasures, punctures, and shortening: %s\n',channelBER_eps(1))
fprintf('RS BER with\n')
fprintf('                               erasures: %s\n',codedBER_e(1))
fprintf('             erasures and punctures: %s\n',codedBER_ep(1))
fprintf('erasures, punctures, and shortening: %s\n',codedBER_eps(1))
```

```
Total number of corrected errors
                           erasures: 6905
             erasures and punctures: 1960
erasures, punctures, and shortening: 3287
64-QAM BER with
                           erasures: 1.702521e-03
             erasures and punctures: 1.475314e-03
erasures, punctures, and shortening: 3.590801e-03
RS BER with
                           erasures: 2.589668e-06
             erasures and punctures: 5.627692e-05
erasures, punctures, and shortening: 9.748840e-05
```

Further Exploration

You can experiment with these systems by running them over a loop of Eb/N0 values and generating a BER curve for them. You can then compare their performance against a theoretical 64-QAM/RS system without erasures, punctures, or shortening. Use BERTool to generate the theoretical BER curves.

```
close_system(model_e,0);
close_system(model_ep,0);
close_system(model_eps,0);
```

Representation of Polynomials in Communications Toolbox

You can specify polynomials as a character vector or string scalar by using a variety of syntaxes. Communications Toolbox functions that support character vector and string scalar polynomials convert these various syntaxes into the appropriate form, which varies depending on the function. For example, the `comm.BCHEncoder` function expresses polynomials as a binary row vector with powers in descending order.

When specifying a character vector or string scalar to represent a polynomial:

- Ascending or descending order is valid.
- Spaces are ignored.
- The caret symbol, `^`, which indicates the presence of an exponent, is optional. If omitted, the function assumes that the integer that follows the variable is an exponent.
- Braces, `{}`, denote an exponent. For example, you can represent x^2 as `x{2}`.
- Text appearing before the polynomial expression (with or without an equals sign) is ignored.
- Punctuation that follows square brackets is ignored.
- Exponents must be uniformly positive or uniformly negative. Mixed-sign exponents are not allowed. For example, `'x^2 + x + 1'` and `'1 + z^-6 + z^-8'` are valid, but `'1 + z^6 + z^-8'` is not valid.

This list shows some examples of how to express the polynomial $x^{14} + 4x^5 + x^3 + 2x + 1$ in code. Use single quotes for character vectors (as shown) or double quotes for string scalars.

- `'1+2x+x^3+4x^5+x^14'`
- `'1+2m+m3+4m5+m14'`
- `'q14 + 4q5 + q3 + 2q + 1'`
- `'g(x) = 1+2x+x3+4x5+x14'`
- `'g(z) 1+2z+z3+4z5+z14'`
- `'p(x) = x{14} + 4x{5} + x{3} + 2{x} + 1'`
- `'[D14 + 4D5 + D3 + 2D + 1]'`

See Also

Functions
`gfadd` | `bchgenpoly` | `poly2trellis`

Objects
`comm.PNSequence` | `comm.BCHEncoder`

Blocks
Gold Sequence Generator | BCH Encoder

Related Examples

- “Working with Galois Fields” on page 20-2
- “Sequence Generators” on page 9-10

- “Error Detection and Correction” on page 16-14

Estimate BER of QPSK in AWGN with Reed-Solomon Coding

Transmit Reed-Solomon encoded data using QPSK over an AWGN channel. Demodulate and decode the received signal and collect error statistics. Compute theoretical bit error rate (BER) for coded and noncoded data. Plot the BER results to compare performance.

Define the example parameters.

```
rng(1993);     % Seed random number generator for repeatable results
M = 4;         % 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 and account for RS coding gain. Initialize the error statistics matrix.

```
ebnoVec = (3:0.5:8)';
ebnoVecCodingGain = ebnoVec + 10*log10(K/N); % Account for RS coding gain
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.

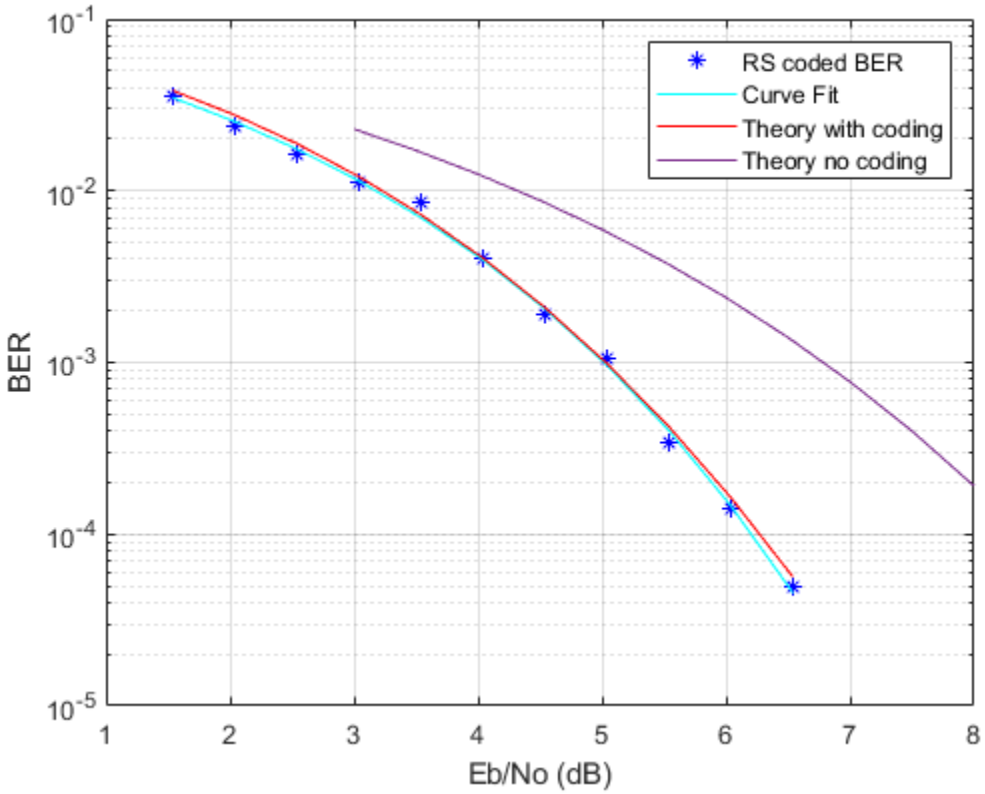
```
for i = 1:length(ebnoVec)
    awgnChannel.EbNo = ebnoVecCodingGain(i);
    reset(errorRate)
    while errorStats(i,2) < 100 && errorStats(i,3) < 1e7
        data = randi([0 1],1500,1);                % Generate binary data
        encData = rsEncoder(data);                 % RS encode
        modData = 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 QPSK performance with and without coding using the `bercoding` and `berawgn` functions.

```
berCurveFit = berfit(ebnoVecCodingGain,errorStats(:,1));
berwCoding = bercoding(ebnoVec,'RS','hard',N,K,'psk',M,'nondiff');
berNoCoding = berawgn(ebnoVec,'psk',M,'nondiff');
```

Plot the RS coded BER data, curve fit of the BER data, theoretical performance with RS coding, and theoretical performance without RS coding. The (7,5) RS code improves the E_b/N_0 required to achieve a 10^{-2} bit error rate by approximately 1.2 dB.

```
semilogy(ebnoVecCodingGain,errorStats(:,1),'b*', ...
ebnoVecCodingGain,berCurveFit,'c-',ebnoVecCodingGain,berwCoding,'r',ebnoVec,berNoCoding)
ylabel('BER')
xlabel('Eb/No (dB)')
legend('RS coded BER','Curve Fit','Theory with coding','Theory no coding')
grid
```



Transmit and Receive Shortened Reed-Solomon Codes

Transmit and receive standard and shortened RS-encoded, 64-QAM-modulated data through an AWGN channel. Compare the performance of the standard and shortened codes.

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. Specify the modulation order, `M`.

```
N = 63;  % Codeword length
K = 51;  % Message length
S = 39;  % Shortened message length
M = 64;  % Modulation order
```

Specify the simulation parameters, where `numErrors` is the number of errors per Eb/No point, and `numBits` is the maximum number of bits per Eb/No point. Specify the range of Eb/No values to be simulated. Initialize the BER arrays.

```
numErrors = 200;
numBits = 1e7;
ebnoVec = (8:13)';
[ber0,ber1] = deal(zeros(size(ebnoVec)));
```

Create an error rate object to collect error statistics.

```
errorRate = comm.ErrorRate;
```

Create a Reed-Solomon encoder and decoder pair for an RS(63,51) code. Calculate the code rate.

```
rsEncoder = comm.RSEncoder(N,K,'BitInput',true);
rsDecoder = comm.RSDecoder(N,K,'BitInput',true);
rate = K/N;
```

Execute the main processing loop.

```
for k = 1:length(ebnoVec)

    % Convert the coded Eb/No to an SNR. Initialize the error statistics
    % vector.
    snrdB = ebnoVec(k) + 10*log10(rate) + 10*log10(log2(M));
    errorStats = zeros(3,1);

    while errorStats(2) < numErrors && errorStats(3) < numBits

        % Generate binary data.
        txData = randi([0 1],K*log2(M),1);

        % Encode the data.
        encData = rsEncoder(txData);

        % Apply 64-QAM modulation.
        txSig = qammod(encData,M, ...
            'UnitAveragePower',true,'InputType','bit');

        % Pass the signal through an AWGN channel.
        rxSig = awgn(txSig,snrdB);

        % Demodulated the noisy signal.
```

```
        demodSig = qamdemod(rxSig,M, ...
            'UnitAveragePower',true,'OutputType','bit');

        % Decode the data.
        rxData = rsDecoder(demodSig);

        % Compute the error statistics.
        errorStats = errorRate(txData,rxData);
    end

    % Save the BER data, and reset the errorRate counter.
    ber0(k) = errorStats(1);
    reset(errorRate)
end
```

Create a Reed-Solomon generator polynomial for an RS(63,51) code.

```
gp = rsgenpoly(N,K,[],0);
```

Create a Reed-Solomon encoder and decoder pair using shortened message length `S` and generator polynomial `gp`. Calculate the rate of the shortened code.

```
rsEncoder = comm.RSEncoder(N,K,gp,S,'BitInput',true);
rsDecoder = comm.RSDecoder(N,K,gp,S,'BitInput',true);
rate = S/(N-(K-S));
```

Execute the main processing loop using the shortened Reed-Solomon code.

```
for k = 1:length(ebnoVec)

    % Convert the coded Eb/No to an SNR. Initialize the error statistics
    % vector.
    snrdB = ebnoVec(k) + 10*log10(rate) + 10*log10(log2(M));
    errorStats = zeros(3,1);

    while errorStats(2) < numErrors && errorStats(3) < numBits

        % Generate binary data.
        txData = randi([0 1],S*log2(M),1);

        % Encode the data.
        encData = rsEncoder(txData);

        % Apply 64-QAM modulation.
        txSig = qammod(encData,M, ...
            'UnitAveragePower',true,'InputType','bit');

        % Pass the signal through an AWGN channel.
        rxSig = awgn(txSig,snrdB);

        % Demodulated the noisy signal.
        demodSig = qamdemod(rxSig,M, ...
            'UnitAveragePower',true,'OutputType','bit');

        % Decode the data.
        rxData = rsDecoder(demodSig);

        % Compute the error statistics.
```

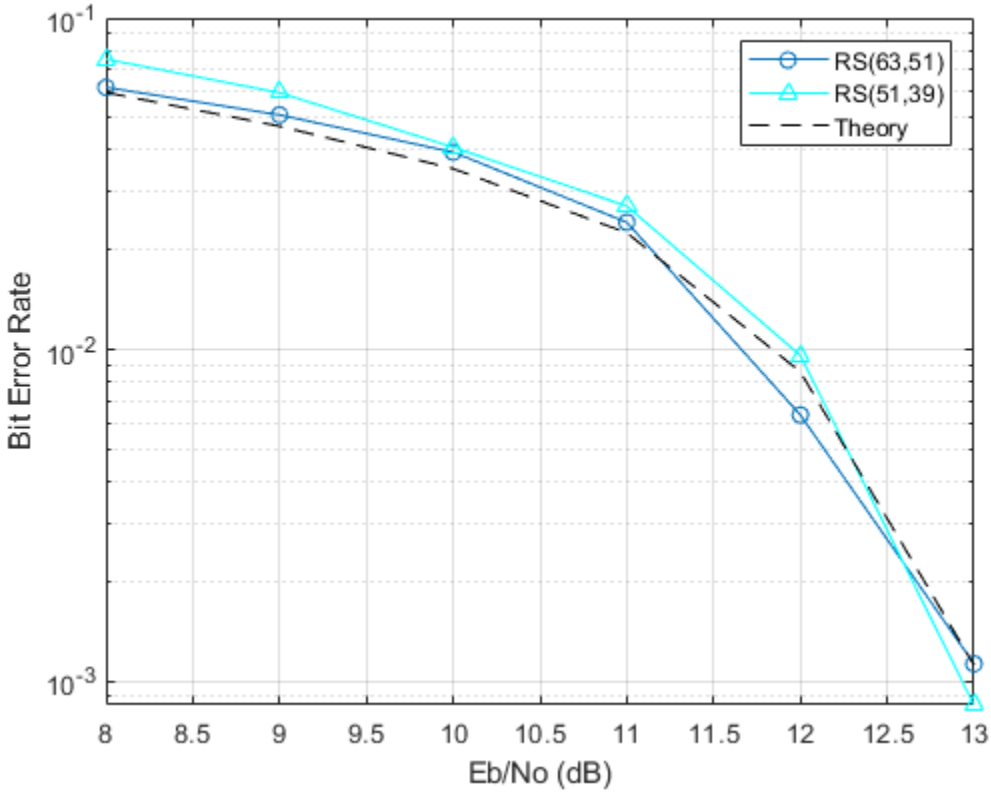
```
        errorStats = errorRate(txData,rxData);
    end

    % Save the BER data, and reset the errorRate counter.
    ber1(k) = errorStats(1);
    reset(errorRate)
end
```

Calculate the approximate BER for an RS (63,51) code.

```
berapprox = bercoding(ebnoVec,'RS','hard',N,K,'qam',64);
```

Compare the BER curves for the RS(63,51) and RS(51,39) codes. Plot the theoretically approximated BER curve. Observe that shortening the code does not affect performance.

```
semilogy(ebnoVec,ber0,'o-',ebnoVec,ber1,'c^-',ebnoVec,berapprox,'k--')
legend('RS(63,51)','RS(51,39)','Theory')
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
grid
```



Galois Fields

Working with Galois Fields

This example shows how to work with Galois fields. This example also shows the effects of using with Hamming codes and Galois field theory for error-control coding.

A Galois field is an algebraic field with a finite number of members. A Galois field that has 2^m members is denoted by $\mathrm{GF}(2^m)$, where m is an integer in the range [1, 16].

Create Galois Field Arrays

Create Galois field arrays using the `gf` function. For example, create the element 3 in the Galois field $\mathrm{GF}(2^2)$.

```
A = gf(3,2)
```

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

Array elements =

   3
```

Use Galois Field Arrays

You can now use `A` as if it is a built-in MATLAB® data type. For example, add two different elements in a Galois field.

```
A = gf(3,2);
B = gf(1,2);
C = A+B
```

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

Array elements =

   2
```

Demonstrate Arithmetic in Galois Fields

The rules for arithmetic operations are different for Galois field elements compared to integers. For example, in $\mathrm{GF}(2^2)$, 3 + 1 = 2 . This table shows some of the differences between Galois field arithmetic and integer arithmetic for integers 0 through 3.

```
+__0__1__2__3

0| 0 1 2 3

1| 1 2 3 4

2| 2 3 4 5

3| 3 4 5 6
```

Define such a table in MATLAB®.

```
A = ones(4,1)*(0:3);
B = (0:3)'*ones(1,4);
A+B
```

```
ans = 4×4

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

Similarly, create an addition table for the Galois field GF$\left(2^2\right)$.

```
A = gf(ones(4,1)*(0:3),2);
B = gf((0:3)'*ones(1,4),2);
A+B
```

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

Array elements =

   0   1   2   3
   1   0   3   2
   2   3   0   1
   3   2   1   0
```

Use MATLAB Functions with Galois Arrays

For a list of MATLAB® functions that work with Galois arrays, see “Galois Computations” on the `gf` function reference page. For example, create two different Galois arrays, and then use the `conv` function to multiply the two polynomials.

```
A = gf([1 33],8);
B = gf([1 55],8);

C = conv(A,B)
```

```
C = GF(2^8) array. Primitive polynomial = D^8+D^4+D^3+D^2+1 (285 decimal)

Array elements =

     1    22   153
```

You can use the `roots` function to find the roots of a polynomial. For example, find the roots of polynomial `C`. The results show that the roots match the original values in polynomials `A` and `B`.

```
roots(C)
```

```
ans = GF(2^8) array. Primitive polynomial = D^8+D^4+D^3+D^2+1 (285 decimal)

Array elements =

   33
   55
```

Use Hamming Codes and Galois Theory

This section shows how to use a simple Hamming code and Galois field theory for error-control coding. An error-control code adds redundancy to information bits. For example, a (7,4) Hamming code maps 4 bits of information to 7-bit codewords by multiplying the 4 information bits by a 4-by-7 generation matrix in Galois field GF(2). Use the `hammgen` function to obtain this matrix.

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

```
paritymat = 3×7

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


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
```

The output `paritymat` is the parity-check matrix, and the output `genmat` is the generator matrix. To encode the information bits `[0 1 0 0]`, multiply the bits by the generator matrix `genmat` in Galois field GF(2).

```
A = gf([0 1 0 0],1)
```

```
A = GF(2) array.

Array elements =

   0   1   0   0
```

```
code = A*genmat
```

```
code = GF(2) array.

Array elements =

   0   1   1   0   1   0   0
```

For this example, suppose that somewhere along transmission, an error is introduced in this codeword. The Hamming code used in this example can correct up to 1 bit error. Insert an error in the transmission by changing the first bit from `0` to `1`.

```
code(1) = 1
```

```
code = GF(2) array.

Array elements =

   1   1   1   0   1   0   0
```

Use the parity-check matrix to determine where the error occurred, by multiplying the erroneous codeword by the parity-check matrix.

```
paritymat*code'
```

```
ans = GF(2) array.

Array elements =

   1
   0
   0
```

Find the error, by inspecting the parity-check matrix, `paritymat`. The column in `paritymat` that matches `[1 0 0]'` is the location of the error. In this example, the first column is `[1 0 0]'`, so the first element of the vector `code` contains the error.

```
paritymat
```

```
paritymat = 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

Functions
`gf` | `hammgen`

More About

- “Error Detection and Correction” on page 16-14

ElGamal Public Key Cryptosystem

Use the Galois field array function, `gf`, to implement an ElGamal public key cryptosystem.

Key Generation

Define the polynomial degree, `m`.

```
m = 15;
q = 2^m;
```

Find a primitive polynomial and group generator. Set the random number generator seed to produce a repeatable result.

```
poly = primpoly(m,'nodisplay');

primeFactors = unique(factor(2^m-1));
rng(123456);
while 1
    g = gf(randi([1,q-1]),m,poly);
    isPrimitive = true;
    for i = 1:length(primeFactors)
        if g^((q-1)/primeFactors(i)) == gf(1,m,poly)
            isPrimitive = false;
            break;
        end
    end
    if isPrimitive
        break;
    end
end
```

Construct private and public keys.

```
privateKey = 12;
publicKey = {g,g^privateKey,poly};
```

Encryption

Create and display the original message.

```
text = ['The Fourier transform decomposes a function of time (a signal)' newline ...
    'into the frequencies that make it up, in a way similar to how a' newline ...
    'musical chord can be expressed as the amplitude (or loudness) of' newline ...
    'its constituent notes.'];
disp(text);
```

```
The Fourier transform decomposes a function of time (a signal)
into the frequencies that make it up, in a way similar to how a
musical chord can be expressed as the amplitude (or loudness) of
its constituent notes.
```

Convert the message to binary and group them every m bits. The message uses ASCII characters. Since the ASCII table has 128 characters, seven bits per character is sufficient.

```
bitsPerChar = 7;
binMsg = int2bit(int8(text'),bitsPerChar);
numPaddedBits = m - mod(numel(binMsg),m);
```

```
if numPaddedBits == m
    numPaddedBits = 0;
end
binMsg = [binMsg; zeros(numPaddedBits,1)];
textToEncrypt = bit2int(binMsg,m);
```

Encrypt the original message.

```
cipherText = gf(zeros(length(textToEncrypt),2),m,poly);

for i = 1:length(textToEncrypt)
    k = randi([1 2^m-2]);
    cipherText(i,:) = [publicKey{1}^k, ...
        gf(textToEncrypt(i),m,poly)*publicKey{2}^k];
end
```

Display the encrypted message.

```
tmp = cipherText.x;
disp(de2char(tmp(:,2),bitsPerChar,m));
```

```
vTchba~*TzEC> *o_c;a5
vS>Do7]{B#wDc0`YCDVxV]:FC0|o|    ],7
S)d/uW
]A=]T    sKX+q3K&q ex_>f=C_gZ"*slX=N&$~*[Xh+R[<5:(NpO6
8+@/)9sBm    &is#Z<DN`Qo~?GaOFIzYA~a+Lygzv?l
```

Decryption

Decrypt the encrypted original message.

```
decipherText = gf(zeros(size(cipherText,1),1),m,poly);
for i = 1:size(cipherText,1)
    decipherText(i) = cipherText(i,2) * cipherText(i,1)^(-privateKey);
end
```

Display the decrypted message.

```
disp(de2char(decipherText.x,bitsPerChar,m));
```

```
The Fourier transform decomposes a function of time (a signal)
into the frequencies that make it up, in a way similar to how a
musical chord can be expressed as the amplitude (or loudness) of
its constituent notes.
```

Supporting Function

`de2char` converts the bits to char messages.

```
function text = de2char(msg,bitsPerChar,m)
binDecipherText = int2bit(msg,m);
text = char(bit2int(binDecipherText(1:end-mod(numel(binDecipherText), ...
    bitsPerChar)),bitsPerChar))';
end
```

See Also

Functions

`gf` | `reshape`

More About

- “Error Detection and Correction” on page 16-14

Error Detection and Correction

High Rate Convolutional Codes for Turbo Coding

Concatenated convolutional codes offer high reliability and have gained in prominence and usage as turbo codes. The `comm.TurboEncoder` and `comm.TurboDecoder` System objects support rate 1/n convolutional codes only. This example shows the parallel concatenation of two rate 2/3 convolutional codes to achieve an effective rate 1/3 turbo code by using `comm.ConvolutionalEncoder` and `comm.APPDecoder` System objects.

System parameters

```
blkLength = 1024;          % Block length
EbNo = 0:5;                % Eb/No values to loop over
numIter = 3;               % Number of decoding iterations
maxNumBlks = 1e2;          % maximum number of blocks per Eb/No value
```

Convolutional Encoder/Decoder Parameters

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13]);
k = log2(trellis.numInputSymbols);      % number of input bits
n = log2(trellis.numOutputSymbols);     % number of output bits
intrIndices = randperm(blkLength/k)';   % Random interleaving
decAlg = 'True App';                    % Decoding algorithm
modOrder = 2;                           % PSK-modulation order
```

Initialize System Objects

Initialize Systems object for convolutional encoding, APP Decoding, BPSK modulation and demodulation, AGWN channel, and error rate computation. The demodulation output soft bits using a log-likelihood ratio method.

```
cEnc1 = comm.ConvolutionalEncoder('TrellisStructure',...
        trellis,'TerminationMethod','Truncated');
cEnc2 = comm.ConvolutionalEncoder('TrellisStructure',...
        trellis,'TerminationMethod','Truncated');
cAPPDec1 = comm.APPDecoder('TrellisStructure',trellis,...
            'TerminationMethod','Truncated','Algorithm',decAlg);
cAPPDec2 = comm.APPDecoder('TrellisStructure',trellis,...
            'TerminationMethod','Truncated','Algorithm',decAlg);

bpskMod = comm.BPSKModulator;
bpskDemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
    'VarianceSource','Input port');

awgnChan = comm.AWGNChannel('NoiseMethod','Variance', ...
    'VarianceSource','Input port');

bitError = comm.ErrorRate; % BER measurement
```

Frame Processing Loop

Loop through a range of E_b/N_0 values to generate results for BER performance. The `helperTurboEnc` and `helperTurboDec` helper functions on page 21-0 perform the turbo encoding and decoding.

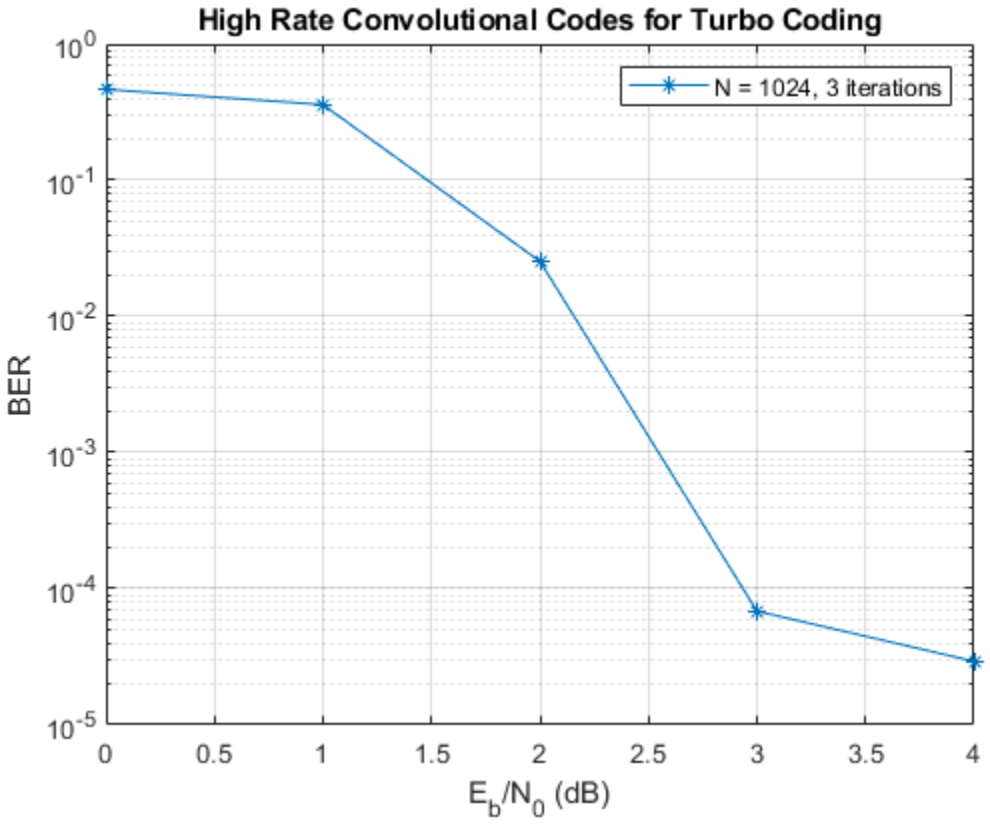
```
ber = zeros(length(EbNo),1);
bitsPerSymbol = log2(modOrder);
turboEncRate = k/(2*n);
```

```
for ebNoIdx = 1:length(EbNo)
    % Calculate the noise variance from EbNo
    EsNo = EbNo(ebNoIdx) + 10*log10(bitsPerSymbol);
    SNRdB = EsNo + 10*log10(turboEncRate); % Account for code rate
    noiseVar = 10^(-SNRdB/10);

    for  numBlks = 1:maxNumBlks
        % Generate binary data
        data = randi([0 1],blkLength,1);

        % Turbo encode the data
        [encodedData,outIndices] = helperTurboEnc(data,cEnc1,cEnc2, ...
            trellis,blkLength,intrIndices);

        % Modulate the encoded data
        modSignal = bpskMod(encodedData);

        % Pass the modulated signal through an AWGN channel
        receivedSignal = awgnChan(modSignal,noiseVar);

        % Demodulate the noisy signal using LLR to output soft bits
        demodSignal = bpskDemod(receivedSignal,noiseVar);

        % Turbo decode the demodulated data
        receivedBits = helperTurboDec(-demodSignal,cAPPDec1,cAPPDec2, ...
            trellis,blkLength,intrIndices,outIndices,numIter);

        % Calculate the error statistics
        errorStats = bitError(data,receivedBits);
    end

    ber(ebNoIdx) = errorStats(1);
    reset(bitError);
end
```

Display Results

While the practical wireless systems, such as LTE and CCSDS, specify base rate-1/n convolutional codes for turbo codes, the results show use of higher rate convolutional codes as turbo codes is viable.

```
figure;
semilogy(EbNo, ber, '*-');
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('High Rate Convolutional Codes for Turbo Coding');
legend(['N = ' num2str(blkLength) ', ' num2str(numIter) ' iterations']);
```



Helper Functions

```
function [yEnc,outIndices] = helperTurboEnc(data,hCEnc1,hCEnc2,trellis,blkLength,intrIndices)
% Turbo encoding using two parallel convolutional encoders.
% No tail bits handling and assumes no output stream puncturing.

    % Trellis parameters
    k = log2(trellis.numInputSymbols);
    n = log2(trellis.numOutputSymbols);
    cLen = blkLength*n/k;

    punctrVec = [0;0;0;0;0;0];      % assumes all streams are output
    N = length(find(punctrVec==0));

    % Encode random data bits
    y1 = step(hCEnc1, data);
    y2 = step(hCEnc2, reshape(intrlv(reshape(data, k, [])',intrIndices)', [], 1));
    y1D = reshape(y1(1:cLen), n, []);
    y2D = reshape(y2(1:cLen), n, []);
    yDTemp = [y1D; y2D];
    y = yDTemp(:);

    % Generate output indices vector using puncturing vector
    idx = 0 : 2*n : (blkLength - 1)*2*(n/k);
    punctrVecIdx = find(punctrVec==0);
    dIdx = repmat(idx, N, 1) + punctrVecIdx;
    outIndices = dIdx(:);
```

```
    yEnc = y(outIndices);
end

function yDec = helperTurboDec(yEnc,cAPPDec1,cAPPDec2,trellis,blkLength,intrIndices,inIndices,nu
% Turbo decoding using two a-posteriori probability (APP) decoders

    % Trellis parameters
    k = log2(trellis.numInputSymbols);
    n = log2(trellis.numOutputSymbols);
    rCodLen = 2*(n/k)*blkLength;
    typeyEnc = class(yEnc);

    % Re-order encoded bits according to outIndices
    x = zeros(rCodLen, 1);
    x(inIndices) = yEnc;

    % Generate output of first encoder
    yD = reshape(x(1:rCodLen), 2*n, []);
    lc1D = yD(1:n, :);
    Lc1_in = lc1D(:);

    % Generate output of second encoder
    lc2D   = yD(n+1:2*n, :);
    Lc2_in = lc2D(:);

    % Initialize unencoded data input
    Lu1_in = zeros(blkLength, 1, typeyEnc);

    % Turbo Decode
    out1 = zeros(blkLength/k, k, typeyEnc);
    for iterIdx = 1 : numIter
        [Lu1_out, ~] = step(cAPPDec1, Lu1_in, Lc1_in);
        tmp = Lu1_out(1:blkLength);
        Lu2_in = reshape(tmp, k, [])';
        [Lu2_out, ~] = step(cAPPDec2, ...
            reshape(Lu2_in(intrIndices, :)', [], 1), Lc2_in);
        out1(intrIndices, :) = reshape(Lu2_out(1:blkLength), k, [])';
        Lu1_in = reshape(out1', [], 1);
    end
    % Calculate llr and decoded bits for the final iteration
    llr = reshape(out1', [], 1) + Lu1_out(1:blkLength);
    yDec = cast((llr>=0), typeyEnc);
end
```

Punctured Convolutional Coding

This example shows how to use the convolutional encoder and Viterbi decoder System objects to simulate a punctured coding system. The complexity of a Viterbi decoder increases rapidly with the code rate. Puncturing is a technique that allows the encoding and decoding of higher rate codes using standard rate 1/2 encoders and decoders.

Introduction

This example showcases the simulation of a communication system consisting of a random binary source, a convolutional encoder, a BPSK modulator, an additive white Gaussian noise (AWGN) channel, and a Viterbi decoder. The example shows how to run simulations to obtain bit error rate (BER) curves and compares these curves to a theoretical bound.

Initialization

Convolutional Encoding with Puncturing

Create a rate 1/2, constraint length 7 `comm.ConvolutionalEncoder` System object. This encoder takes one-bit symbols as inputs and generates 2-bit symbols as outputs. If you assume 3-bit message words as inputs, then the encoder will generate 6-bit codeword outputs.

```
convEncoder = comm.ConvolutionalEncoder(poly2trellis(7, [171 133]));
```

Specify a puncture pattern to create a rate 3/4 code from the previous rate 1/2 code using the puncture pattern vector [1;1;0;1;1;0]. The ones in the puncture pattern vector indicate that bits in positions 1, 2, 4, and 5 are transmitted, while the zeros indicate that bits in positions 3 and 6 are punctured or removed from the transmitted signal. The effect of puncturing is that now, for every 3 bits of input, the punctured code generates 4 bits of output (as opposed to the 6 bits produced before puncturing). This results in a rate 3/4 code. In the example at hand, the length of the puncture pattern vector must be an integer multiple of 6 since 3-bit inputs get converted into 6-bit outputs by the rate 1/2 convolutional encoder.

To set the desired puncture pattern in the convolutional encoder System object, `hConvEnc` , set the `PuncturePatternSource` property to `Property` and the `PuncturePattern` property to `[1;1;0;1;1;0]` .

```
convEncoder.PuncturePatternSource = 'Property';
convEncoder.PuncturePattern = [1;1;0;1;1;0];
```

Modulator and Channel

Create a `comm.BPSKModulator` System object to transmit the encoded data using binary phase shift keying modulation over a channel.

```
bpskMod = comm.BPSKModulator;
```

Create an `comm.AWGNChannel` System object. Set the `NoiseMethod` property of the channel to `Signal to noise ratio (Eb/No)` to specify the noise level using the energy per bit to noise power spectral density ratio (Eb/No). When running simulations, test the coding system for different values of Eb/No ratio by changing the `EbNo` property of the channel object. The output of the BPSK modulator generates unit power signals; set the `SignalPower` property to 1 Watt. The system at hand is at the symbol rate; set the `SamplesPerSymbol` property to 1.

```
channel = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)',...
  'SignalPower', 1, 'SamplesPerSymbol', 1);
```

Viterbi Decoding with Depuncturing

Configure a `comm.ViterbiDecoder` System object so it decodes the punctured code specified for the convolutional encoder. This example assumes unquantized inputs to the Viterbi decoder, so set the `InputFormat` property to `Unquantized` .

```
vitDecoder = comm.ViterbiDecoder(poly2trellis(7, [171 133]), ...
  'InputFormat', 'Unquantized');
```

In general, the puncture pattern vectors you use for the convolutional encoder and Viterbi decoder must be the same. To specify the puncture pattern, set the `PuncturePatternSource` property of the Viterbi decoder System object, `hVitDec` , to `Property` . Set the `PuncturePattern` property to the same puncture pattern vector you use for the convolutional encoder.

Because the punctured bits are not transmitted, there is no information to indicate their values. As a result, the decoding process ignores them.

```
vitDecoder.PuncturePatternSource =  'Property';
vitDecoder.PuncturePattern = convEncoder.PuncturePattern;
```

For a rate 1/2 code with no puncturing, you normally set the traceback depth of a Viterbi decoder to a value close to 40. Decoding punctured codes requires a higher value, in order to give the decoder enough data to resolve the ambiguities that the punctures introduce. This example uses a traceback depth of 96. Set this value using the `TraceBackDepth` property of the Viterbi decoder object, `hVitDec` .

```
vitDecoder.TracebackDepth = 96;
```

Calculating the Error Rate

Create an `comm.ErrorRate` calculator System object to compare decoded bits to the original transmitted bits. The output of the error rate calculator object is a three-element vector containing the calculated bit error rate (BER), the number of errors observed, and the number of bits processed. The Viterbi decoder creates a delay in the output decoded bit stream equal to the traceback length. To account for this delay set the `ReceiveDelay` property of the error rate calculator System object to 96.

```
errorCalc = comm.ErrorRate('ReceiveDelay', vitDecoder.TracebackDepth);
```

Stream Processing Loop

Analyze the BER performance of the punctured coding system for different noise levels.

Uncoded and Coded Eb/No Ratio Values

Typically, you measure system performance according to the value of the energy per bit to noise power spectral density ratio (Eb/No) available at the input of the channel encoder. The reason for this is that this quantity is directly controlled by the systems engineer. Analyze the performance of the coding system for Eb/No values between 2 and 5 dB.

```
EbNoEncoderInput = 2:0.5:5; % in dB
```

The signal going into the AWGN channel is the encoded signal. Convert the Eb/No values so that they correspond to the energy ratio at the encoder output. If you input three bits to the encoder and obtain four bit outputs, then the energy relation is given by the 3/4 rate as follows:

```
EbNoEncoderOutput = EbNoEncoderInput + 10*log10(3/4);
```

Simulation loop

To obtain BER performance results, transmit frames of 3000 bits through the communications system. For each Eb/No value, stop simulations upon reaching a specific number of errors or transmissions. To improve the accuracy of the results, increase the target number of errors or the maximum number of transmissions.

```
frameLength = 3000;         % this value must be an integer multiple of 3
targetErrors = 300;
maxNumTransmissions = 5e6;
```

Loop through the encoded Eb/No values (the simulation will take a few seconds to complete).

```
BERVec = zeros(3,length(EbNoEncoderOutput)); % Allocate memory to store results
for n=1:length(EbNoEncoderOutput)
  reset(errorCalc)
  reset(convEncoder)
  reset(vitDecoder)
  channel.EbNo = EbNoEncoderOutput(n); % Set the channel EbNo value for simulation
  while (BERVec(2,n) < targetErrors) && (BERVec(3,n) < maxNumTransmissions)
    % Generate binary frames of size specified by the frameLength variable
    data = randi([0 1], frameLength, 1);
    % Convolutionally encode the data
    encData = convEncoder(data);
    % Modulate the encoded data
    modData = bpskMod(encData);
    % Pass the modulated signal through an AWGN channel
    channelOutput = channel(modData);
    % Pass the real part of the channel complex outputs as the unquantized
    % input to the Viterbi decoder.
    decData = vitDecoder(real(channelOutput));
    % Compute and accumulate errors
    BERVec(:,n) = errorCalc(data, decData);
  end
end
```

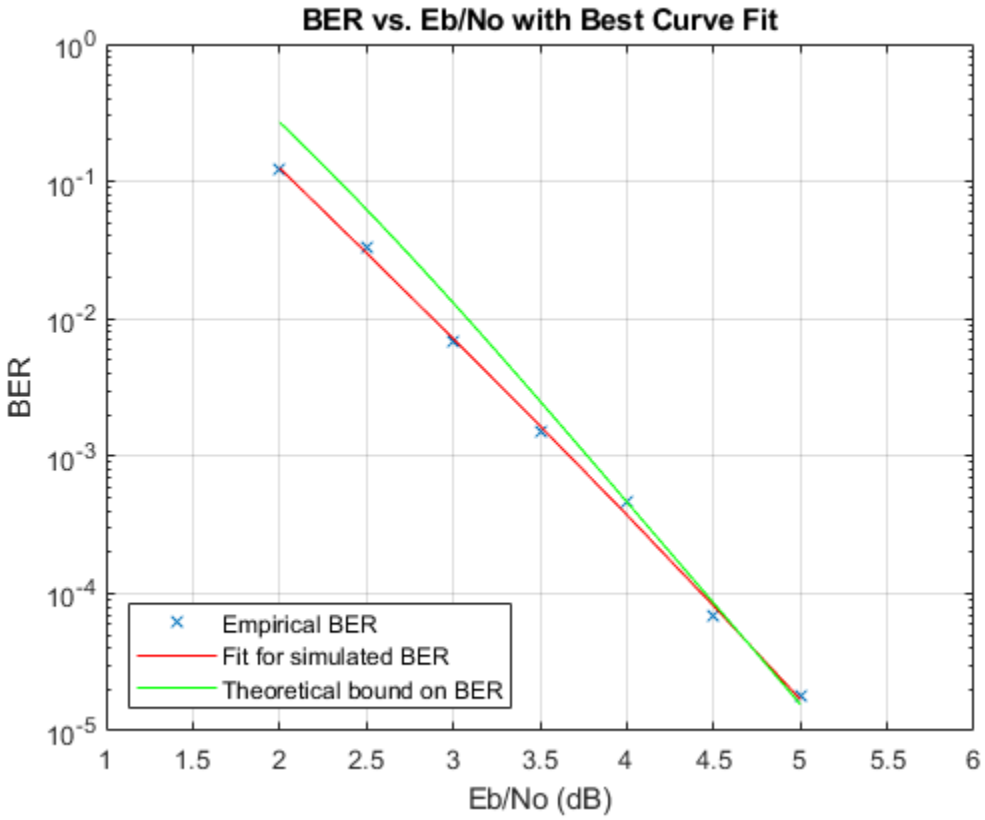
Compare Results to Theoretical Curves

We compare the simulation results using an approximation of the bit error probability bound for a punctured code as per [1]. The following commands compute an approximation of this bound using the first seven terms of the summation for Eb/No values in `2:0.5:5`. The values used for nerr are found in Table 2 of [2].

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

Plot results. If the target number of errors or maximum number of transmissions you specify for the simulation are too small, the curve fitting algorithm might fail.

```
berfit(EbNoEncoderInput,BERVec(1,:)); % Curve-fitted simulation results
hold on;
semilogy((2:.02:5),bound,'g'); % Theoretical results
legend('Empirical BER','Fit for simulated BER', 'Theoretical bound on BER')
axis([1 6 10^-5 1])
```



In some cases, at lower bit error rates, simulation results appear to indicate error rates slightly above the bound. This results from simulation variance (if fewer than 500 bit errors are observed) or from the finite traceback depth in the decoder.

Summary

We utilized several System objects to simulate a communications system with convolutional coding and puncturing. We simulated the system to obtain BER performance versus different Eb/No ratio values. The BER results were compared to theoretical bounds.

Selected Bibliography

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

2 Begin, G., Haccoun, D., and Paquin, C., "Further results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding," IEEE Transactions on Communications, Vol. 38, No. 11, November, 1990, p. 1923

Punctured Convolutional Encoding

In this section...

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

The example is somewhat similar to the one that appears in “Soft-Decision Decoding” on page 16-48, which shows convolutional coding without puncturing.

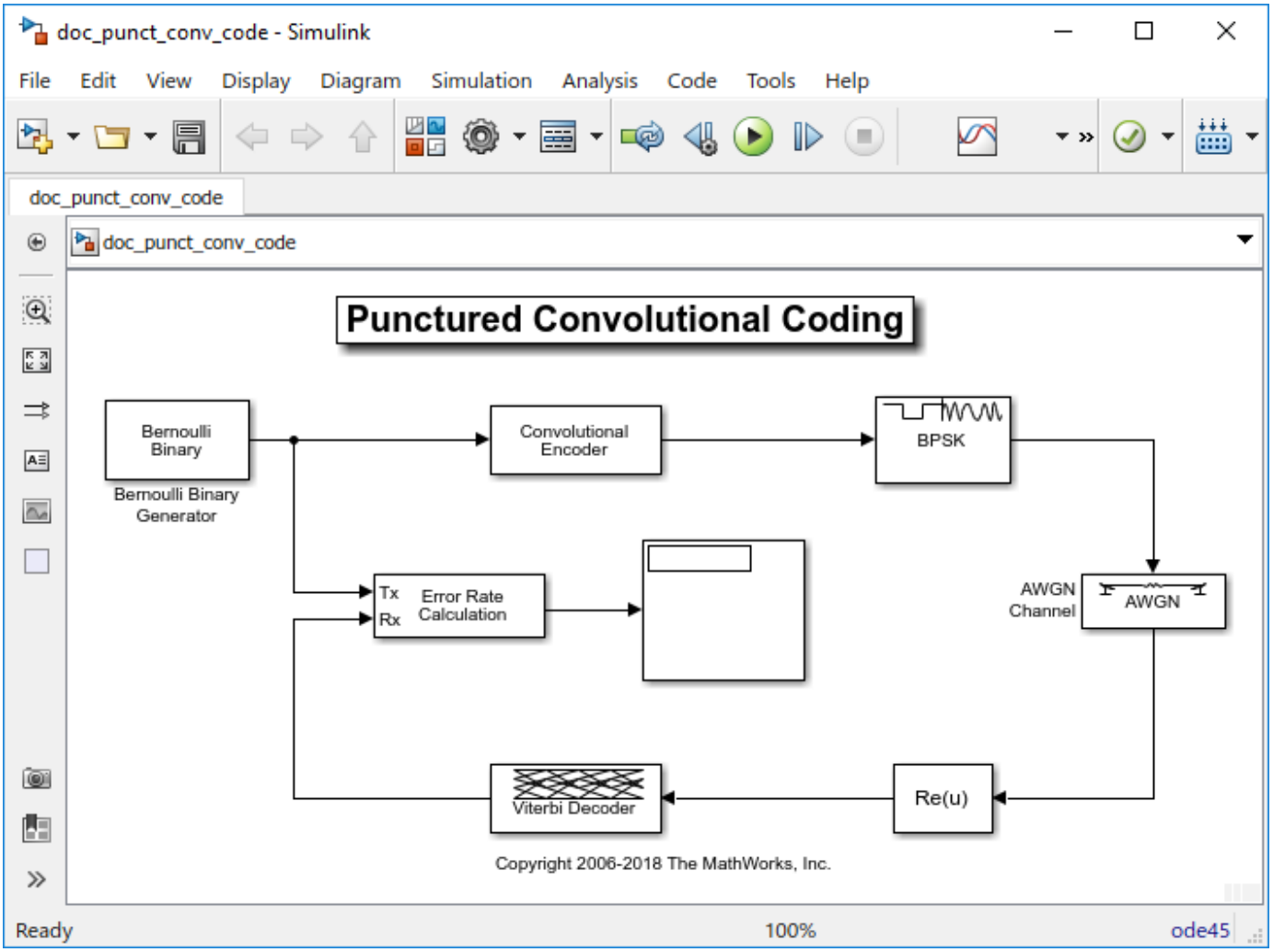
Structure of the Example

This example contains these blocks.

- Bernoulli Binary Generator: Create a sequence of random bits to use as a message.
- Convolutional Encoder: Encode the message using the convolutional encoder.
- BPSK Modulator Baseband: Modulate the encoded message.
- AWGN Channel: Pass the modulated signal through a noisy channel.
- Error Rate Calculation: Compute the number of discrepancies between the original and recovered messages.

Open the example, `doc_punct_conv_code`, by entering the following at the MATLAB command prompt.

```
doc_punct_conv_code
```



Generating Random Data

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

Convolutional Encoding with Puncturing

The Convolutional Encoder block encodes the data from the Bernoulli Binary Generator. This example uses the same code as described in “Soft-Decision Decoding” on page 16-48.

The puncture pattern is specified by the **Puncture vector** parameter in the mask. The puncture vector is a binary column vector. A 1 indicates that the bit in the corresponding position of the input vector is sent to the output vector, while a 0 indicates that the bit is removed.

For example, to create a rate 3/4 code from the rate 1/2, constraint length 7 convolutional code, the optimal puncture vector is [1 1 0 1 1 0].' (where the .' after the vector indicates the transpose). Bits in positions 1, 2, 4, and 5 are transmitted, while bits in positions 3 and 6 are removed. Now, for every 3

bits of input, the punctured code generates 4 bits of output (as opposed to the 6 bits produced before puncturing). This makes the rate 3/4.

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

Transmitting Data

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

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

Demodulating

In this simulation, the Viterbi Decoder block is set to accept unquantized inputs. As a result, the simulation passes the channel output through a Simulink Complex to Real-Imag block that extracts the real part of the complex samples.

Viterbi Decoding of Punctured Codes

The Viterbi Decoder block is configured to decode the same rate 1/2 code specified in the Convolutional Encoder block.

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

Since the punctured bits are not transmitted, there is no information to indicate their values. As a result they are ignored in the decoding process.

The **Puncture vector** parameter indicates the locations of the punctures or the bits to ignore in the decoding process. Each 1 in the puncture vector indicates a transmitted bit while each 0 indicates a puncture or the bit to ignore in the input to the decoder.

In general, the two **Puncture vector** parameters in the Convolutional Encoder and Viterbi Decoder must be the same.

Calculating the Error Rate

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

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

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

Evaluating Results

Generating a bit error rate curve requires multiple simulations. You can perform multiple simulations using the `sim` command. Follow these steps:

- In the model window, remove the Display block and the line connected to its port.
- In the AWGN Channel block, set the **Es/No** parameter to the variable name `EsNodB`.
- In the Error Rate Calculation block, set **Output data** to `Workspace` and then set **Variable name** to `BER_Data`.
- Save the model in your working directory under a different name, such as `my_punct_conv_code.slx`.
- Execute the following code, which runs the simulation multiple times and gathers results.

```
CodeRate = 0.75;
EbNoVec = [2:.5:5];
EsNoVec = EbNoVec + 10*log10(CodeRate);
BERVec = zeros(length(EsNoVec),3);
for n=1:length(EsNoVec),
    EsNodB = EsNoVec(n);
    sim('my_commpunctcnvcod');
    BERVec(n,:) = BER_Data;
end
```

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

$$P_b \le \frac{1}{2(n-1)} \sum_{d=d_{free}}^{\infty} \omega_d \mathrm{erfc}\left(\sqrt{\mathrm{rd}(\mathrm{E_b/N_0})}\right)$$

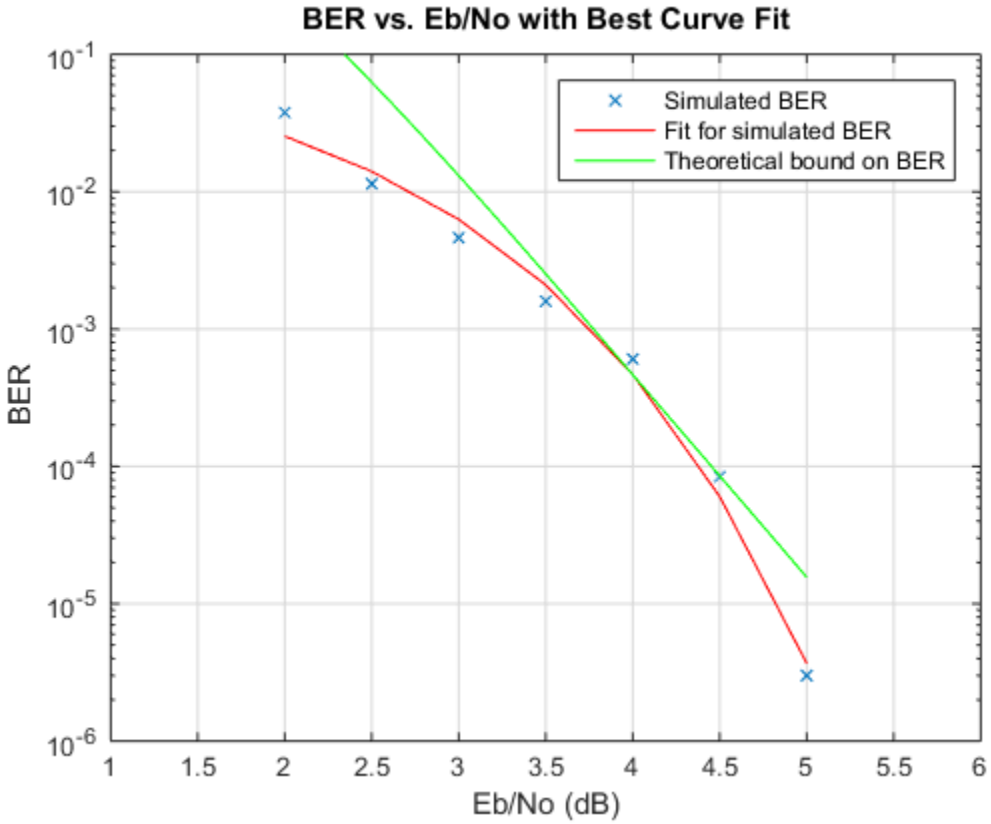
In this expression, `erfc` denotes the complementary error function, r is the code rate, and both d_{free} and ω_d are dependent on the particular code. For the rate 3/4 code of this example, $d_{free} = 5$, $\omega_5 = 42$, $\omega_6 = 201$, $\omega_7 = 1492$, and so on. See reference [1] for more details.

The following commands compute an approximation of this bound using the first seven terms of the summation (the values used for `nerr` are found in Table 2 of reference [2]:

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

To plot the simulation and theoretical results in the same figure, use the commands below.

```
berfit(EbNoVec',BERVec(:,1)); % Curve-fitted simulation results
hold on;
semilogy(EbNo_dB,bound,'g'); % Theoretical results
legend('Simulated BER','Fit for simulated BER',...
    'Theoretical bound on BER')
```



In some cases, at the lower bit error rates, you might notice simulation results that appear to indicate error rates slightly above the bound. This can result from simulation variance (if fewer than 500 bit errors are observed) or from the finite traceback depth in the decoder.

References

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

[2] Begin, G., Haccoun, D., and Paquin, C., "Further results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding," IEEE Transactions on Communications, Vol. 38, No. 11, November, 1990, p. 1923.

Rate 2/3 Convolutional Code in AWGN

This example generates a bit error rate versus Eb/No curve for a link that uses 16-QAM modulation and a rate 2/3 convolutional code in AWGN.

Set the modulation order, and compute the number of bits per symbol.

```
M = 16;
k = log2(M);
```

Create a trellis for a rate 2/3 convolutional code. Set the traceback and code rate parameters.

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13]);
traceBack = 16;
codeRate = 2/3;
```

Create a convolutional encoder and its equivalent Viterbi decoder.

```
convEncoder = comm.ConvolutionalEncoder('TrellisStructure',trellis);
vitDecoder = comm.ViterbiDecoder('TrellisStructure',trellis, ...
    'InputFormat','Hard','TracebackDepth',traceBack);
```

Create an error rate object. Set the receiver delay to twice the traceback depth, which is the delay through the decoder.

```
errorRate = comm.ErrorRate('ReceiveDelay',2*traceBack);
```

Set the range of Eb/No values to be simulated. Initialize the bit error rate statistics matrix.

```
ebnoVec = 0:2:10;
errorStats = zeros(length(ebnoVec),3);
```

Simulate the link by following these steps:

- Generate binary data.
- Encode the data with a rate 2/3 convolutional code.
- 16-QAM modulate the encoded data, configure bit inputs and unit average power.
- Pass the signal through an AWGN channel.
- 16-QAM demodulate the received signal configure bit outputs and unit average power.
- Decode the demodulated signal by using a Viterbi decoder.
- Collect the error statistics.

```
for m = 1:length(ebnoVec)
    snr = ebnoVec(m) + 10*log10(k*codeRate);

    while errorStats(m,2) <= 100 && errorStats(m,3) <= 1e7
        dataIn = randi([0 1],10000,1);
        dataEnc = convEncoder(dataIn);
        txSig = qammod(dataEnc,M, ...
            'InputType','bit','UnitAveragePower',true);
        rxSig = awgn(txSig,snr);
        demodSig = qamdemod(rxSig,M, ...
            'OutputType','bit','UnitAveragePower',true);
        dataOut = vitDecoder(demodSig);
        errorStats(m,:) = errorRate(dataIn,dataOut);
```

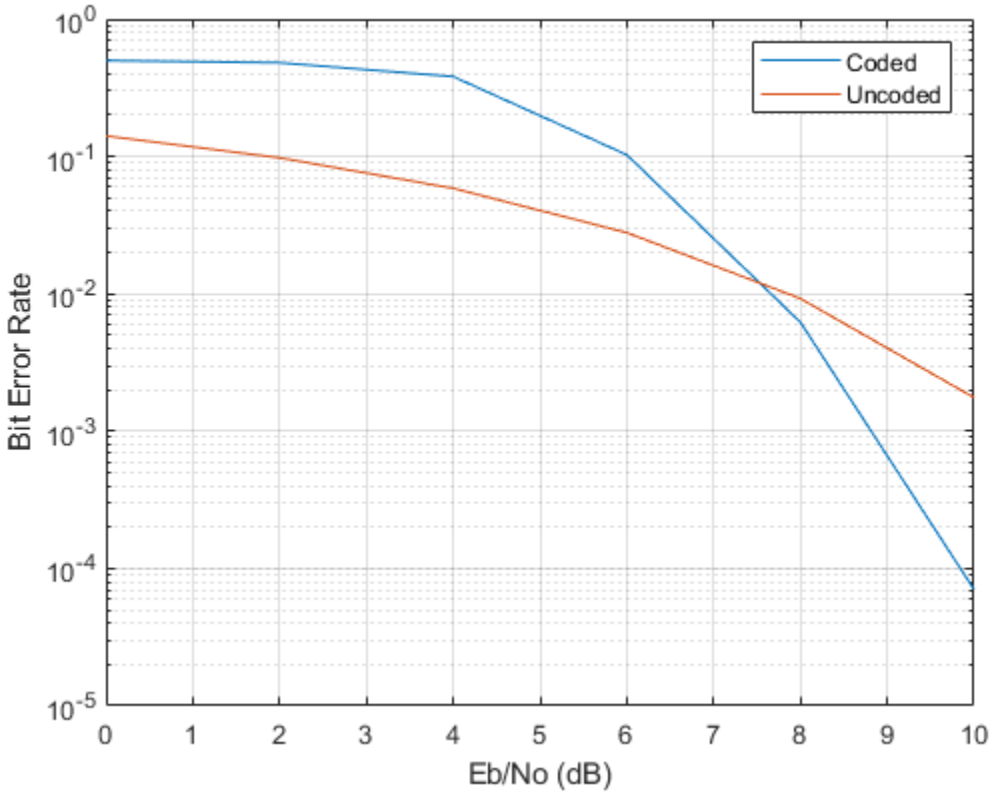
```
    end
    reset(errorRate)
end
```

Compute the theoretical BER vs. Eb/No curve for the case without forward error correction coding.

```
berUncoded = berawgn(ebnoVec','qam',M);
```

Plot the BER vs. Eb/No curve for the simulated coded data and the theoretical uncoded data. At higher Eb/No values, the error correcting code provides performance benefits.

```
semilogy(ebnoVec,[errorStats(:,1) berUncoded])
grid
legend('Coded','Uncoded')
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



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

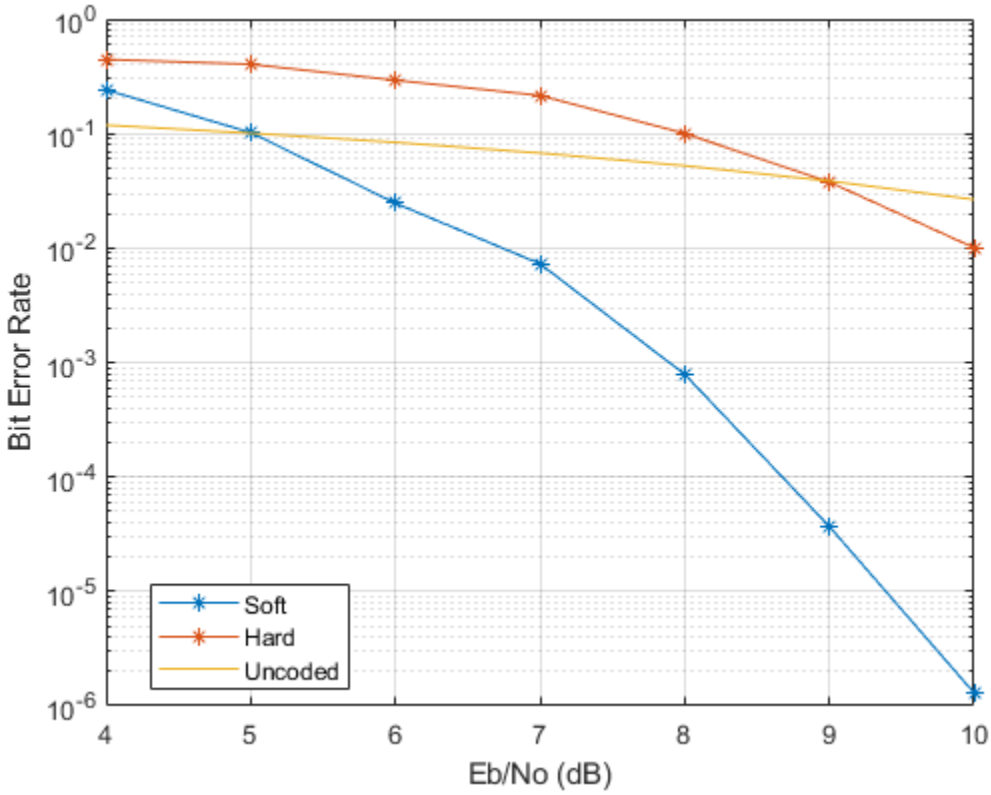
The `while` loop continues to process data until either 100 errors are encountered or 10^7 bits are transmitted.

```
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.

Creation, Validation, and Testing of User Defined Trellis Structure

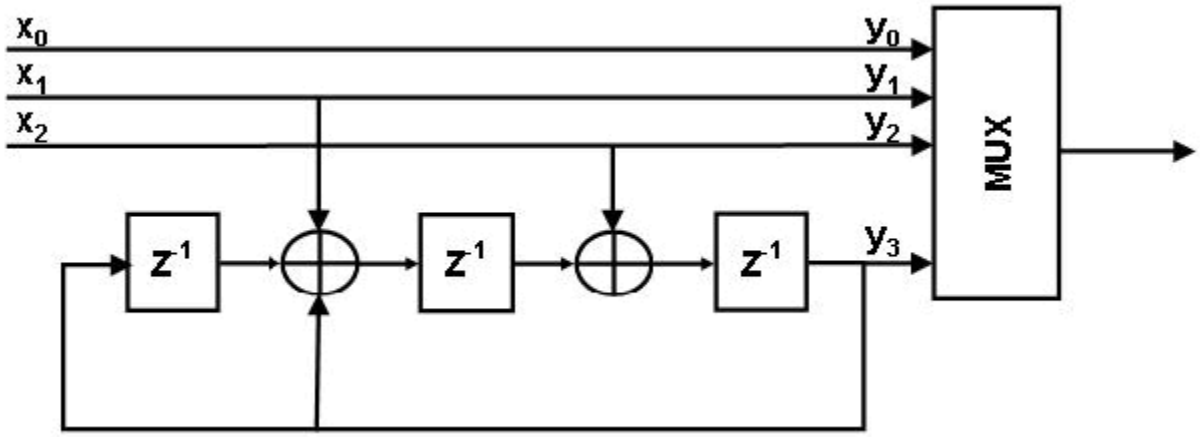
Create User Defined Trellis Structure

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” on page 21-24 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*.

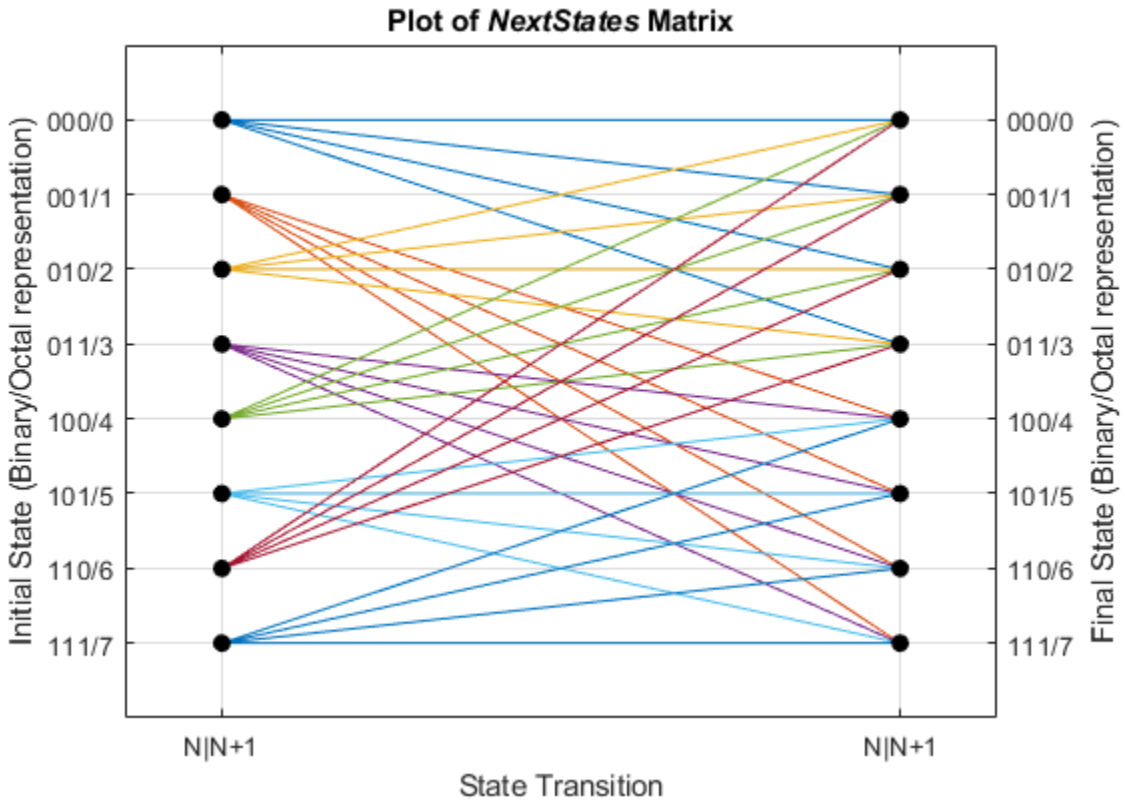
```
myTrellis.nextStates = [0  1  2  3  0  1  2  3; ...
                        6  7  4  5  6  7  4  5; ...
                        1  0  3  2  1  0  3  2; ...
                        7  6  5  4  7  6  5  4; ...
                        2  3  0  1  2  3  0  1; ...
                        4  5  6  7  4  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.

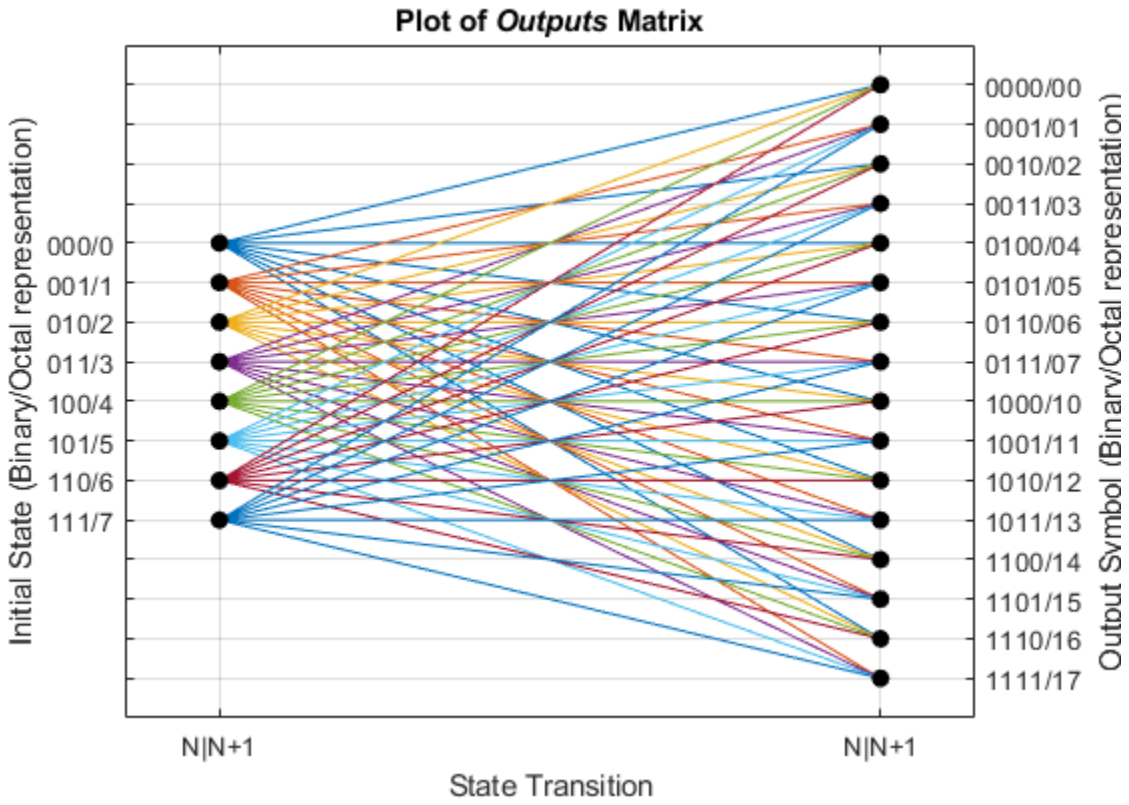
```
myTrellis.outputs = outputs
```

```
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
```

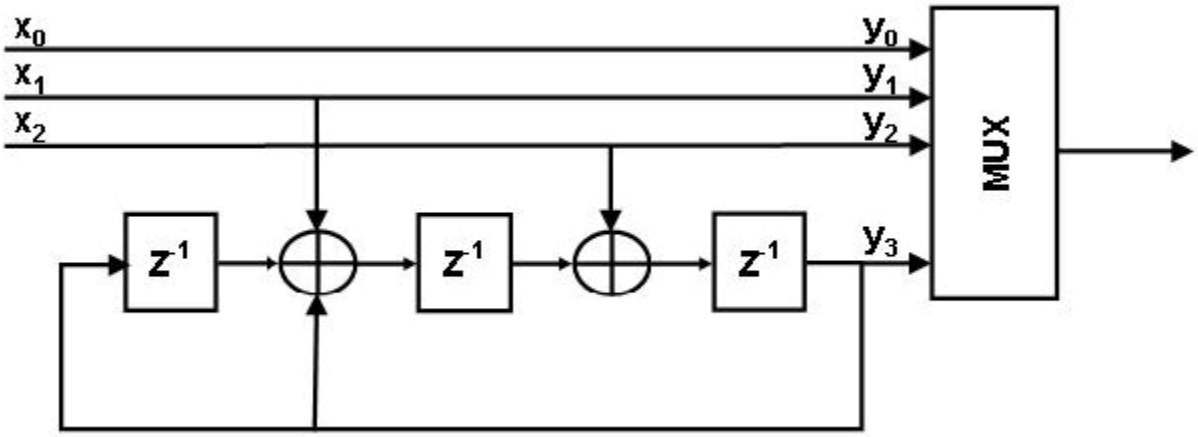
A return value of 1 confirms the trellis structure is valid.

Convolutional Encoder with Uncoded Bits and Feedback

The model serves as a unit test bench for the convolutional code implemented. The model shows how to define and use a trellis that describes a convolutional code. The particular code in this example cannot be described by a set of generator and feedback connection polynomials. The code's trellis cannot be created by the `poly2trellis` because that function expects generator and feedback connection polynomials as input arguments.

Structure of the Convolutional Code

This figure shows the convolutional code.



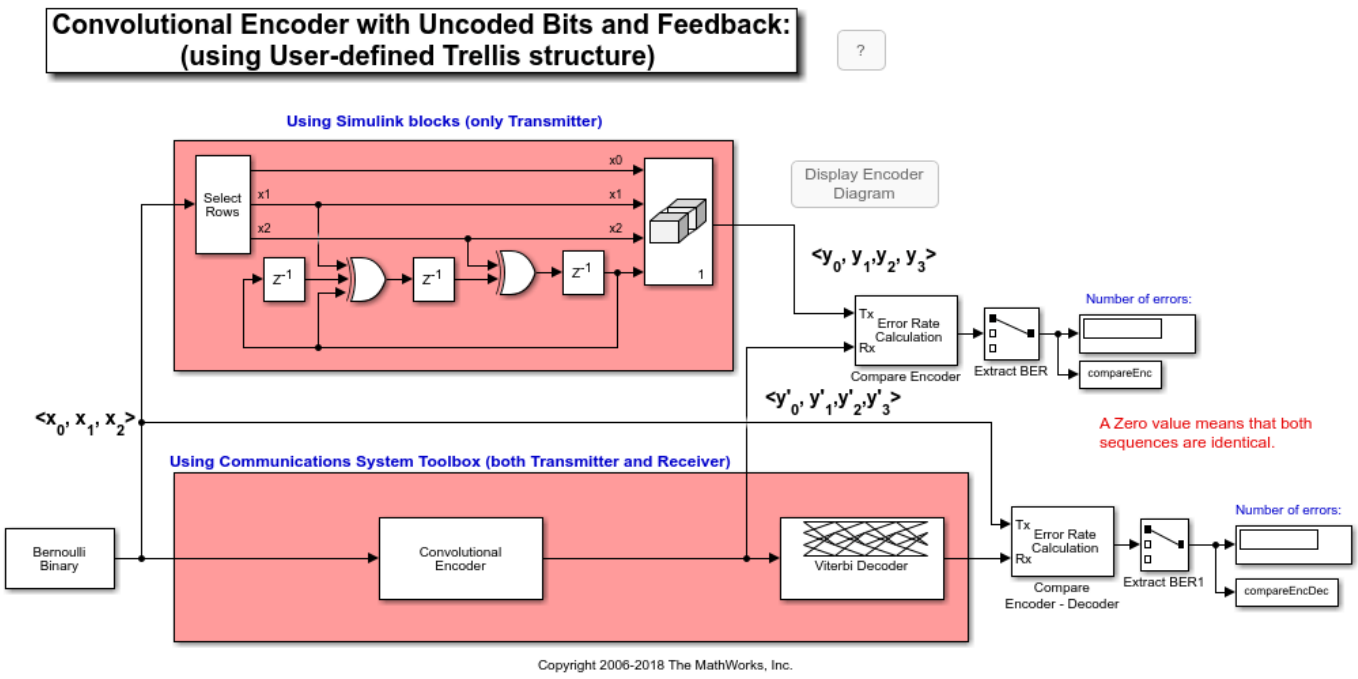
Structure of the Example

The major components in this example include:

- A transmit path that builds a representation of the convolutional encoder using discrete low-level delay and sum (XOR) blocks. This representation looks very similar to the figure showing the structure of the convolutional code.
- A transmit-receive path that builds a representation of the same convolutional encoder using the Convolutional Encoder block. In this case, the description of the encoder is within the block's Trellis structure parameter. This portion of the model also includes the Viterbi Decoder block, which decodes the convolutional code.
- Both paths compute the number of bit errors.

Open Example Model and Explore Its Contents

Open the example model `slex_commcnvencoder`.



Results and Displays

When you run the simulation, the block labeled `Compare Encoder` checks that the two representations of the encoder yield the same result. The block labeled `Compare Encoder - Decoder` checks that the encoder and decoder work properly as a pair. Each Display block in the model shows an error rate of zero, as expected.

```
Error rate for Compare Encoder signal: 0.000
Error rate for Compare Encoder-Decoder signal: 0.000
```

See Also

Channel Modeling and RF Impairments

AWGN Channel

In this section...

Section Overview

An AWGN channel adds white Gaussian noise to the signal that passes through it. You can create an AWGN channel in a model using the `comm.AWGNChannel` System object, the AWGN Channel block, or the `awgn` function.

The following examples use an AWGN Channel: “QPSK Transmitter and Receiver” on page 8-255 and “General QAM Modulation in AWGN Channel” on page 17-40.

AWGN Channel Noise Level

Typical quantities used to describe the relative power of noise in an AWGN channel include

- Signal-to-noise ratio (SNR) per sample. SNR is the actual input parameter to the `awgn` function.
- Ratio of bit energy to noise power spectral density (EbN0). This quantity is used by `BER Analyzer` Tool and performance evaluation functions in this toolbox.
- Ratio of symbol energy to noise power spectral density (EsN0)

Relationship Between EsN0 and EbN0

The relationship between EsN0 and EbN0, both expressed in dB, is as follows:

$$E_s/N_0 \text{ (dB)} = E_b/N_0 \text{ (dB)} + 10\log_{10}(k)$$

where k is the number of information bits per symbol.

In a communications system, k might be influenced by the size of the modulation alphabet or the code rate of an error-control code. For example, in a system using a rate-1/2 code and 8-PSK modulation, the number of information bits per symbol (k) is the product of the code rate and the number of coded bits per modulated symbol. Specifically, $(1/2)\log_2(8) = 3/2$. In such a system, three information bits correspond to six coded bits, which in turn correspond to two 8-PSK symbols.

Relationship Between EsN0 and SNR

The relationship between EsN0 and SNR, both expressed in dB, is as follows:

$$E_s/N_0 \text{ (dB)} = 10\log_{10}\left(T_{sym}/T_{samp}\right) + SNR \text{ (dB)} \quad \text{for complex input signals}$$

$$E_s/N_0 \text{ (dB)} = 10\log_{10}\left(0.5T_{sym}/T_{samp}\right) + SNR \text{ (dB)} \quad \text{for real input signals}$$

where T_{sym} is the symbol period of the signal and T_{samp} is the sampling period of the signal.

For a complex baseband signal oversampled by a factor of 4, the EsN0 exceeds the corresponding SNR by $10\log_{10}(4)$.

Derivation for Complex Input Signals

You can derive the relationship between EsN0 and SNR for complex input signals as follows:

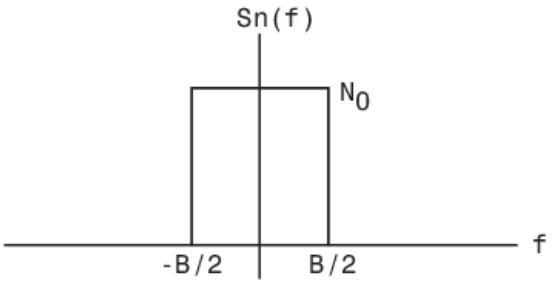
$$\begin{aligned} E_s/N_0 \text{ (dB)} &= 10\log_{10}\left((S \cdot T_{sym})/(N/B_n)\right) \\ &= 10\log_{10}\left((T_{sym}F_s) \cdot (S/N)\right) \\ &= 10\log_{10}\left(T_{sym}/T_{samp}\right) + SNR \text{ (dB)} \end{aligned}$$

where

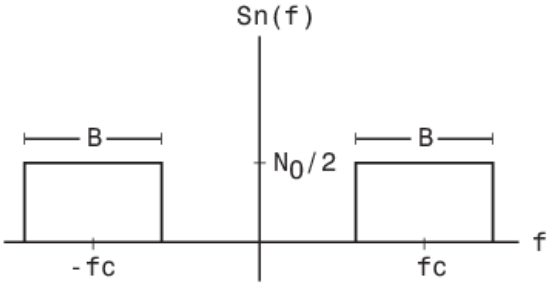
- S = Input signal power, in watts
- N = Noise power, in watts
- B_n = Noise bandwidth, in Hertz = $F_s = 1/T_{samp}$.
- F_s = Sampling frequency, in Hertz

Behavior for Real and Complex Input Signals

These figures illustrate the difference between the real and complex cases by showing the noise power spectral densities of a real bandpass white noise process and its complex lowpass equivalent.



Complex Lowpass Noise Power Spectral Density



Real Bandpass Noise Power Spectral Density

See Also

Objects
`comm.AWGNChannel`

Blocks
AWGN Channel

Functions
awgn

More About

- “QPSK Transmitter and Receiver” on page 8-255
- “General QAM Modulation in AWGN Channel” on page 17-40

Configure Eb/No for AWGN Channels with Coding

This example shows how to set the bit energy to noise density ratio (Eb/No) for communication links employing channel coding.

Specify the codeword and message length for a Reed-Solomon code. Specify the modulation order.

```
N = 15;       % R-S codeword length in symbols
K = 9;        % R-S message length in symbols
M = 16;       % Modulation order
```

Construct a (15,9) Reed-Solomon encoder and a 16-PSK modulator. Specify the objects so that they accept bit inputs.

```
rsEncoder = comm.RSEncoder('CodewordLength',N,'MessageLength',K, ...
    'BitInput',true);
pskModulator = comm.PSKModulator('ModulationOrder',M,'BitInput',true);
```

Create the corresponding Reed-Solomon decoder and 16-PSK demodulator objects.

```
rsDecoder = comm.RSDecoder('CodewordLength',N,'MessageLength',K, ...
    'BitInput',true);
pskDemodulator = comm.PSKDemodulator('ModulationOrder',M,'BitOutput',true);
```

Calculate the Reed-Solomon code rate based on the ratio of message symbols to the codeword length. Determine the bits per symbol for the PSK modulator.

```
codeRate = K/N;
bitsPerSymbol = log2(M);
```

Specify the uncoded Eb/No in dB. Convert the uncoded Eb/No to the corresponding coded Eb/No using the code rate.

```
UncodedEbNo = 6;
CodedEbNo = UncodedEbNo + 10*log10(codeRate);
```

Construct an AWGN channel taking into account the number of bits per symbol. Set the `EbNo` property of `channel` to the coded Eb/No.

```
channel = comm.AWGNChannel('BitsPerSymbol',bitsPerSymbol);
channel.EbNo = CodedEbNo;
```

Set the total number of errors and bits for the simulation. For accuracy, the simulation should run until a sufficient number of bit errors are encountered. The number of total bits is used to ensure that the simulation does not run too long.

```
totalErrors = 100;
totalBits = 1e6;
```

Construct an error rate calculator System object™ and initialize the error rate vector.

```
errorRate = comm.ErrorRate;
errorVec = zeros(3,1);
```

Run the simulation to determine the BER.

```
while errorVec(2) < totalErrors && errorVec(3) < totalBits
    % Generate random bits
```

```
    dataIn = randi([0,1],360,1);
    % Use the RS (15,9) encoder to add error correction capability
    dataEnc = rsEncoder(dataIn);
    % Apply 16-PSK modulation
    txSig = pskModulator(dataIn);
    % Pass the modulated data through the AWGN channel
    rxSig = channel(txSig);
    % Demodulate the received signal
    demodData = pskDemodulator(rxSig);
    % Decode the demodulated data with the RS (15,9) decoder
    dataOut = rsDecoder(demodData);
    % Collect error statistics
    errorVec = errorRate(dataIn,demodData);
end
```

Display the resultant bit error rate.

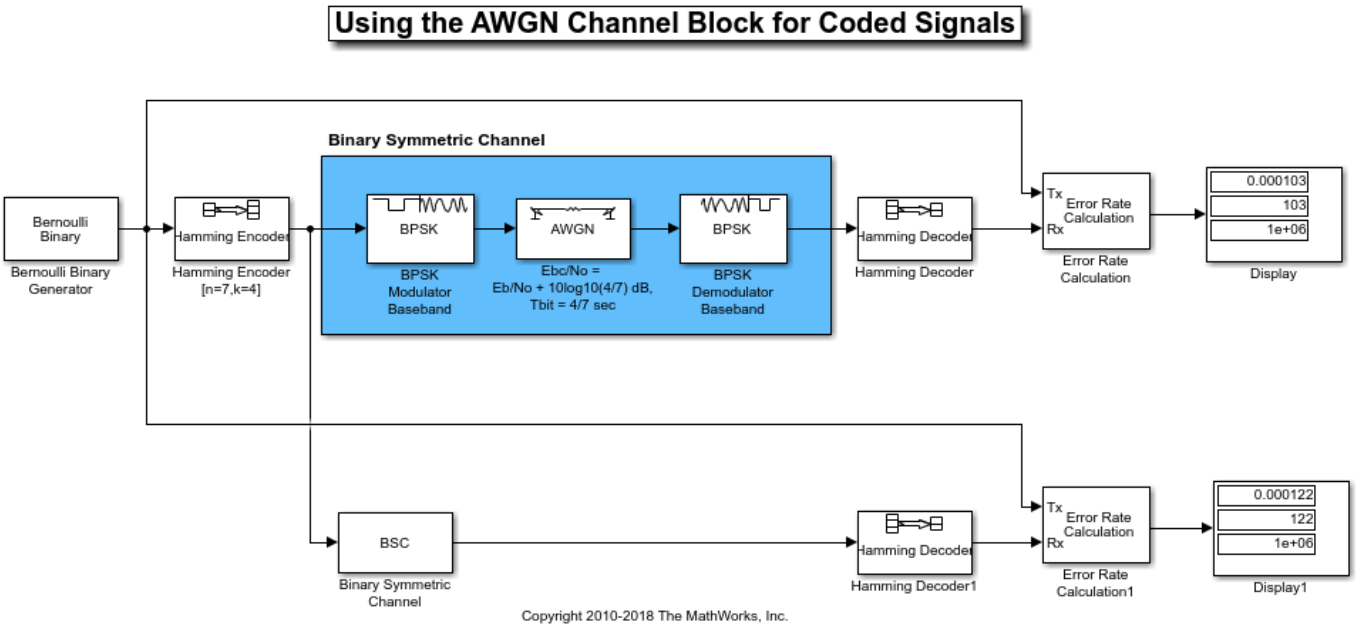
```
ber = errorVec(1)
```

```
ber = 0.0935
```

Using AWGN Channel Block for Coded Signals

Two links perform error control coding on a signal that has passed through an impairment channel. Both links are set for an uncoded Eb/No of 8 dB.

In the top link, the AWGN channel block is set to provide a coded Eb/No of 8 + 10log10(4/7) dB, where 4/7 is the code rate. This accounts for the fact that the coded Eb/No is always lower (by a factor of the code rate) than the uncoded Eb/No. The blue shaded portion of the top link is simply a binary symmetric channel, which is modeled more compactly in the bottom link. The channel error probability of the top link is Q(sqrt(2*Ebc/No)), where Q() is the standard Q function and Ebc/No is the coded Eb/No (in absolute terms, not in dB).



For this example, it is important to note that the bit period at the input of the AWGN Channel block is 4/7 sec. It is 1 sec at the input of the Hamming Encoder block, but that block decreases the bit time by a factor of the code rate.

If you allow the model to run for 1e6 bits, you'll note that the BERs are virtually identical. The difference lies in the stochastic nature of the two random number generators.

You can also check these BER results against expected analytical results by typing this command at the MATLAB® command prompt.

BER = bercoding(8,'block','hard',7,4,3)

This expression finds the upper bound of the BER of a linear, rate 4/7 block code with a minimum distance of 3, and hard decision decoding.

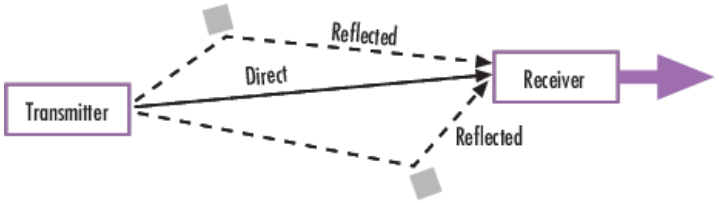
Fading Channels

In this section...

Overview of Fading Channels

Using Communications Toolbox you can implement fading channels using objects or blocks. Rayleigh and Rician fading channels are useful models of real-world phenomena in wireless communications. These phenomena include multipath scattering effects, time dispersion, and Doppler shifts that arise from relative motion between the transmitter and receiver. This section gives a brief overview of fading channels and describes how to implement them using the toolbox.

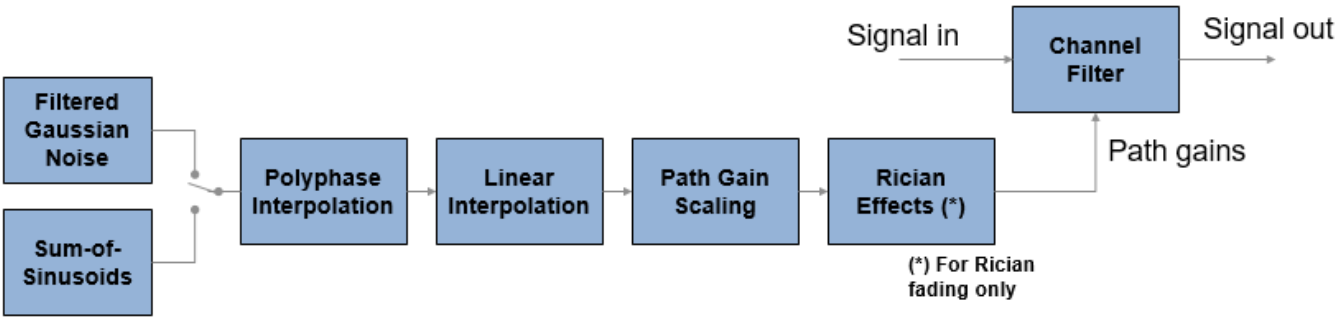
This figure depicts direct and major reflected paths between a stationary radio transmitter and a moving receiver. The shaded shapes represent reflectors such as buildings.



The major paths result in the arrival of delayed versions of the signal at the receiver. In addition, the radio signal undergoes scattering on a *local* scale for each major path. Such local scattering typically results from reflections off objects near the mobile. These irresolvable components combine at the receiver and cause a phenomenon known as *multipath fading*. Due to this phenomenon, each major path behaves as a discrete fading path. Typically, the fading process is characterized by a Rayleigh distribution for a non line-of-sight path and a Rician distribution for a line-of-sight path.

The relative motion between the transmitter and receiver causes Doppler shifts. Local scattering typically comes from many angles around the mobile. This scenario causes a range of Doppler shifts, known as the *Doppler spectrum*. The *maximum* Doppler shift corresponds to the local scattering components whose direction exactly opposes the trajectory of the mobile.

The channel filter applies path gains to the input signal, `Signal in`. The path gains are configured based on settings chosen in the fading channel object or block.



These blocks and objects enable you to model SISO or MIMO fading channels.

Tool	SISO	MIMO

MATLAB	`comm.RayleighChannel`
	
	`comm.RicianChannel`
		`comm.MIMOChannel`

Simulink	SISO Fading Channel	MIMO Fading Channel

Implement Fading Channel Using an Object

A baseband channel model for multipath propagation scenarios that you implement using objects includes:

- N discrete fading paths. Each path has its own delay and average power gain. A channel for which $N = 1$ is called a *frequency-flat fading channel*. A channel for which $N > 1$ is experienced as a *frequency-selective fading channel* by a signal of sufficiently wide bandwidth.
- A Rayleigh or Rician model for each path.
- Default channel path modeling using a Jakes Doppler spectrum, with a maximum Doppler shift that can be specified. Other types of Doppler spectra allowed (identical or different for all paths) include: flat, restricted Jakes, asymmetrical Jakes, Gaussian, bi-Gaussian, rounded, and bell.

  If the maximum Doppler shift is set to 0 or omitted during the construction of a channel object, then the object models the channel as static. For this configuration, the fading does not evolve with time and the Doppler spectrum specified has no effect on the fading process.

Some additional information about typical values for delays and gains is in Choosing Realistic Channel Property Values on page 22-22.

Implement Fading Channel Using a Block

The Channels block library includes MIMO and SISO fading blocks that can simulate real-world phenomena in mobile communications. These phenomena include multipath scattering effects, in addition to Doppler shifts that arise from relative motion between the transmitter and receiver.

Tip To model a channel that involves both fading and additive white Gaussian noise, use a fading channel block followed by an AWGN Channel block.

The MIMO Fading Channel and SISO Fading Channel blocks can be set to model Rayleigh or Rician fading distributions of the channel. Based on the type of signal path, choose the fading distribution to use.

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

You can use a single instance of a fading channel block configured for Rayleigh fading distribution to model multiple major reflected paths simultaneously.

Choosing appropriate block parameters for your situation is important. For more information, see Choosing Realistic Channel Property Values on page 22-22, and the MIMO Fading Channel and SISO Fading Channel block reference pages.

Visualize a Fading Channel

You can view the characteristics of a fading channel using channel visualization tools. For more information, see Channel Visualization on page 25-27.

Compensate for Fading Response

A communications system involving a fading channel usually requires components that compensate for the fading response. Typical approaches to compensate for fading include:

- Differential modulation or a one-tap equalizer helps compensate for a frequency-flat fading channel. For information about implementing differential modulation, see the M-DPSK Modulator Baseband block reference page.
- An equalizer with multiple taps helps compensate for a frequency-selective fading channel. See Equalization on page 14-2 for more information.

The “Adaptive Equalization with Filtering and Fading Channel” on page 14-30 example illustrates why compensating for a fading channel is necessary.

Methodology for Simulating Multipath Fading Channels

The Rayleigh and Rician multipath fading channel simulators in Communications Toolbox use the band-limited discrete multipath channel model of section 9.1.3.5.2 in [1]. This implementation assumes that the delay power profile and the Doppler spectrum of the channel are separable [1]. The multipath fading channel is therefore modeled as a linear finite impulse-response (FIR) filter. Let $\{s_i\}$ denote the set of samples at the input to the channel. Then the samples $\{y_i\}$ at the output of the channel are related to $\{s_i\}$ through:

$$y_i = \sum_{n=-N_1}^{N_2} s_{i-n} g_n$$

where $\{g_n\}$ is the set of tap weights given by:

$$g_n = \sum_{k=1}^{K} a_k \mathrm{sinc}\left[\frac{\tau_k}{T_s} - n\right], \; -N_1 \le n \le N_2$$

In the equations:

- T_s is the input sample period to the channel.
- $\{\tau_k\}$, where $1 \le k \le K$, is the set of path delays. K is the total number of paths in the multipath fading channel.
- $\{a_k\}$, where $1 \le k \le K$, is the set of complex path gains of the multipath fading channel. These path gains are uncorrelated with each other.
- N_1 and N_2 are chosen so that $|g_n|$ is small when n is less than $-N_1$ or greater than N_2.

Two techniques, filtered Gaussian noise and sum-of-sinusoids, are used to generate the set of complex path gains, a_k.

Each path gain process a_k is generated by the following steps:

Filtered Gaussian Noise Technique

1. A complex uncorrelated (white) Gaussian process with zero mean and unit variance is generated in discrete time.
2. The complex Gaussian process is filtered by a Doppler filter with frequency response $H(f) = \sqrt{S(f)}$, where $S(f)$ denotes the desired Doppler power spectrum.
3. The filtered complex Gaussian process is interpolated so that its sample period is consistent with the sample period of the input signal. A combination of linear and polyphase interpolation is used.

Sum-of-sinusoids Technique

1. Mutually uncorrelated Rayleigh fading waveforms are generated using the method described in [2], where $i = 1$ corresponds to the in-phase component and $i = 2$ corresponds to the quadrature component.

$$z_k(t) = \mu_k^{(1)}(t) + j\mu_k^{(2)}(t), \quad k = 1, 2, \ldots, K$$

$$\mu_k^{(i)}(t) = \sqrt{\frac{2}{N_k}} \sum_{n=1}^{N_k} \cos\left(2\pi f_{k,n}^{(i)} t + \theta_{k,n}^{(i)}\right), \quad i = 1, 2$$

   Where

   - N_k specifies the number of sinusoids used to model a single path.
   - $f_{k,n}^{(i)}$ is the discrete Doppler frequency and is calculated for each sinusoid component within a single path.
   - $\theta_{k,n}^{(i)}$ is the phase of the nth component of $\mu_k^{(i)}$ and is an i.i.d. random variable having a uniform distribution over the interval $(0, 2\pi]$.
   - t is the fading process time.

   When modeling a Jakes Doppler spectrum, the discrete Doppler frequencies, $f_{k,n}^{(i)}$, with maximum shift $f_{\max}$ are given by

$$f_{k,n}^{(i)} = f_{\max}\cos\left(\alpha_{k,n}^{(i)}\right)$$

$$= f_{\max}\cos\left[\frac{\pi}{2N_k}\left(n - \frac{1}{2}\right) + \alpha_{k,0}^{(i)}\right]$$

where

$$\alpha_{k,0}^{(i)} \triangleq (-1)^{i-1}\frac{\pi}{4N_k}\cdot\frac{k}{K+2}, \quad i = 1,\ 2 \text{ and } k = 1, 2, \ldots, K$$

2 To advance the fading process in time, an initial time parameter, t_{init}, is introduced. The fading waveforms become

$$\mu_k^{(i)}(t) = \sqrt{\frac{2}{N_k}}\sum_{n=1}^{N_k}\cos\left(2\pi f_{k,n}^{(i)}(t + t_{init}) + \theta_{k,n}^{(i)}\right), \quad i = 1, 2$$

When $t_{\text{init}} = 0$, the fading process starts at time zero. A positive value of t_{init} advances the fading process relative to time zero while maintaining its continuity.

3 Channel fading samples are generated using the GMEDS_1 [2] algorithm.

Calculate Complex Coefficients

The complex process resulting from either technique, z_k, is scaled to obtain the correct average path gain. In the case of a Rayleigh channel, the fading process is obtained as:

$$a_k = \sqrt{\Omega_k}z_k$$

where

$$\Omega_k = E\left[|a_k|^2\right]$$

In the case of a Rician channel, the fading process is obtained as:

$$a_k = \sqrt{\Omega_k}\left[\frac{z_k}{\sqrt{K_{r,k}+1}} + \sqrt{\frac{K_{r,k}}{K_{r,k}+1}}e^{j\left(2\pi f_{d,LOS,k}t + \theta_{LOS,k}\right)}\right]$$

where $K_{r,k}$ is the Rician K-factor of the kth path, $f_{d,LOS,k}$ is the Doppler shift of the line-of-sight component of the kth path (in Hz), and $\theta_{LOS,k}$ is the initial phase of the line-of-sight component of the kth path (in rad).

At the input to the band-limited multipath channel model, the transmitted symbols must be oversampled by a factor at least equal to the bandwidth expansion factor introduced by pulse shaping. For example, if sinc pulse shaping is used, for which the bandwidth of the pulse-shaped signal is equal to the symbol rate, then the bandwidth expansion factor is 1, and at least one sample per symbol is required at the input to the channel. If a raised cosine (RC) filter with a factor more than 1 is used, for which the bandwidth of the pulse-shaped signal is equal to twice the symbol rate, then the bandwidth expansion factor is 2, and at least two samples per symbol are required at the input to the channel.

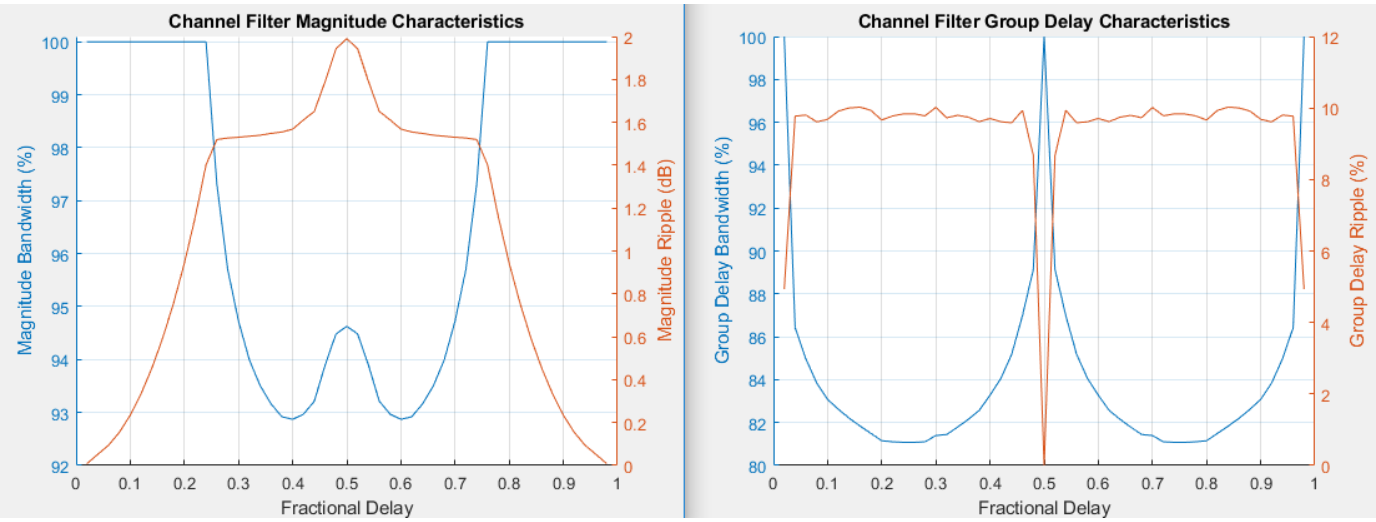
Channel Filter Model Characteristics

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.



References

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

[2] 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.

Specify Fading Channels

Communications Toolbox models a fading channel as a linear FIR filter. Filtering a signal using a fading channel involves these steps:

1. Create a channel object that describes the channel that you want to use. A channel object is a type of MATLAB variable that contains information about the channel, such as the maximum Doppler shift.
2. Adjust properties of the channel object, if necessary, to tailor it to your needs. For example, you can change the path delays or average path gains.
3. Apply the channel object to your signal using calling the object.

This section describes how to define, inspect, and manipulate channel objects. The topics are:

Creating Channel Objects

To create a fading channel object suitable for your modeling situation, select one of these System objects.

Function	Object	Situation Modeled
`comm.RayleighChannel`	Rayleigh fading channel object	One or more major reflected paths
`comm.RicianChannel`	Rician fading channel object	One direct line-of-sight path, possibly combined with one or more major reflected paths

For example, this command creates a channel object representing a Rayleigh fading channel that acts on a signal sampled at 100,000 Hz. The maximum Doppler shift of the channel is 130 Hz.

```
rayChan1 = comm.RayleighChannel('SampleRate',1e5,...
              'MaximumDopplerShift',130); % Rayleigh fading channel object
```

To learn how to call the `rayChan1` fading channel object to filter the transmitted signal through the channel, see Using Fading Channels on page 22-24.

Duplicate and Copy Objects

You can also create another object by duplicating an existing object and then adjust the properties of the new object, if necessary. To duplicate an object always use the `clone` function such as:

```
rayChan2 = clone(rayChan1); % Copy rayChan1 to create an independent rayChan2.
```

instead of `rayChan2 = rayChan1`. The `clone` command creates a copy of `rayChan1` that is independent of `rayChan1`. By contrast, the command `rayChan2 = rayChan1` creates `rayChan2` as merely a reference to `rayChan1`, so that `rayChan1` and `rayChan2` always have indistinguishable content.

Displaying and Changing Object Properties

A channel object has numerous properties that record information about the channel model, about the state of a channel that has already filtered a signal, and about the channel operation on a future signal.

You can view the properties in these ways:

- To view all properties of a channel object, enter the object name in the Command Window.
- You can view a property of a channel object or assign the value to a variable by entering the object name followed by a dot (period), followed by the name of the property.

You can change the writable properties of a channel object in these ways:

- To change the default value of a channel object property, enter the desired value in the object creation syntax.
- To change the value of a writeable property of a channel object, issue an assignment statement that uses dot notation on the channel object. More specifically, dot notation means an expression that consists of the object name, followed by a dot, followed by the name of the property.

Display Rayleigh Channel Object Properties

Create a Rayleigh channel object. Display the object to show the properties initialized by default and the ones specified when creating the object. Entering `rayChan` displays all properties of the channel object. Some of the properties values were assigned when the object was created, while other properties have default values. For more information about specific channel properties, see the reference page for the `comm.RayleighChannel` object.

```
rayChan = comm.RayleighChannel('SampleRate',1e5,'MaximumDopplerShift',130);
rayChan % View all the properties
```

```
rayChan =
  comm.RayleighChannel with properties:

            SampleRate: 100000
            PathDelays: 0
      AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 130
        DopplerSpectrum: [1x1 struct]
      ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
g = rayChan.AveragePathGains % Retrieve the AveragePathGains property of rayChan
```

```
g = 0
```

Adjust Rician Channel Object Properties

A Rician fading channel object has an additional property that does not appear for a Rayleigh fading channel object, namely, a scalar `KFactor` property. For more information about Rician channel properties, see the reference page for the `comm.RicianChannel` object.

Change Rician Channel Object Properties

Create a Rician channel object. The default setting for the `Visualization` property is `'Off'`. Changing the default setting to `'Impulse response'` generates an impulse response plot of the output signal when the object is called. The output displays a subset of all the properties of the channel object. Select `all properties` to see the complete set of properties for `ricChan`.

```
ricChan= comm.RicianChannel; % Create object
ricChan.Visualization = 'Impulse response' % Enables the impulse response channel visualization
```

```
ricChan =
  comm.RicianChannel with properties:

               SampleRate: 1
               PathDelays: 0
         AveragePathGains: 0
        NormalizePathGains: true
                   KFactor: 3
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 1.0000e-03
```

```
          DopplerSpectrum: [1x1 struct]
        ChannelFiltering: true
      PathGainsOutputPort: false

  Show all properties
```

Relationships Among Channel Object Properties

Some properties of a channel object are related to each other such that when one property's value changes, another property's value must change in some corresponding way to keep the channel object consistent. For example, if you change the vector length of `PathDelays`, then the value of `AveragePathGains` must change so that its vector length equals that of the new value of `PathDelays`. This is because the length of each of the two vectors equals the number of discrete paths of the channel. For details about linked properties and an example, see `comm.RayleighChannel` or `comm.RicianChannel`.

Specify Doppler Spectrum of Fading Channel

The Doppler spectrum of a channel object is specified through its `DopplerSpectrum` property. The value of this property must be either:

- A Doppler spectrum structure. In this case, the same Doppler spectrum applies to each path of the channel object.
- A cell array of Doppler spectrum structures of the same length as the `PathDelays` vector property. In this case, the Doppler spectrum of each path is given by the corresponding Doppler spectrum structure in the vector.
  - When the vector length of the `PathDelays` property is increased, the length of `DopplerSpectrum` is automatically increased to match the length of `PathDelays`, by appending Jakes Doppler spectrum structures.
  - If the length of the `PathDelays` vector property is decreased, the length of `DopplerSpectrum` is automatically decreased to match the length of `PathDelays`, by removing the last Doppler spectrum structures.

A Doppler spectrum structure contains the properties used to characterize the Doppler spectrum, but the maximum Doppler shift is a property of the channel object. This section describes how to create and manipulate Doppler spectrum structures, and how to assign them to the `DopplerSpectrum` property of channel objects.

Create a Doppler Spectrum Structure

To create Doppler spectrum structures, use the `doppler` function. The sole purpose of the `doppler` function is to create Doppler spectrum structures used to specify the value of the `DopplerSpectrum` property of channel objects. Select from the following:

- `doppler('Jakes')`
- `doppler('Flat')`
- `doppler('Rounded', ...)`
- `doppler('Bell', ...)`
- `doppler('Asymmetric Jakes', ...)`

- `doppler('Restricted Jakes', ...)`
- `doppler('Gaussian', ...)`
- `doppler('BiGaussian', ...)`

For example, a Gaussian spectrum with a normalized (by the maximum Doppler shift of the channel) standard deviation of 0.1, can be created as:

```
dopp1 = doppler('Gaussian',0.1);
```

Note When creating a Doppler spectrum structure, consider the following dependencies:

- If you assign a single Doppler spectrum structure to `DopplerSpectrum`, all paths have the same specified Doppler spectrum.
- If the `FadingTechnique` property is `'Sum of sinusoids'`, `DopplerSpectrum` must be `doppler('Jakes')`;
- If you assign a row cell array of different Doppler spectrum structures to `DopplerSpectrum`, each path has the Doppler spectrum specified by the corresponding structure in the cell array. In this case, the length of `DopplerSpectrum` must be equal to the length of `PathDelays`.
- To generate C code, specify `DopplerSpectrum` to a single Doppler spectrum structure.

View and Change Doppler Spectrum Structure Properties

Create a Doppler spectrum structure by specifying the type of Doppler spectrum as `'Rounded'`, then modify settings of the polynomial type

A rounded Doppler spectrum structure with default properties is created and displayed, and the third element of the `Polynomial` field is modified.

```
doppRound = doppler('Rounded')
```

```
doppRound = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [1 -1.7200 0.7850]
```

Adjust the third coefficient of the polynomial.

```
doppRound.Polynomial(3) = 0.825
```

```
doppRound = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [1 -1.7200 0.8250]
```

Be aware that it is possible to modify a Doppler spectrum structure to an invalid configuration. Validation of the Doppler spectrum structure settings is performed when the structure is used by a fading channel object. The `doppRound` spectrum structure defined above is valid.

```
ricianCh = comm.RicianChannel('DopplerSpectrum',doppRound)
```

```
ricianCh =
  comm.RicianChannel with properties:
```

```
               SampleRate: 1
               PathDelays: 0
         AveragePathGains: 0
       NormalizePathGains: true
                  KFactor: 3
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 1.0000e-03
           DopplerSpectrum: [1x1 struct]
          ChannelFiltering: true
       PathGainsOutputPort: false

  Show all properties
```

Use Doppler Spectrum Structures Within Channel Objects

The `DopplerSpectrum` property of a channel object can be changed by assigning to it a Doppler spectrum structure or a vector of Doppler spectrum structures.

Create Rayleigh Channel with Flat Doppler Spectrum

This example shows how to change the default Jakes Doppler spectrum of a configured Rayleigh channel object to a flat Doppler spectrum.

Create a Rayleigh channel object

Set the sample rate to 9600 Hz and the maximum Doppler shift to 100 Hz.

```
rayChan = comm.RayleighChannel( ...
    'SampleRate',9600,'MaximumDopplerShift',100)
```

```
rayChan =
  comm.RayleighChannel with properties:

            SampleRate: 9600
            PathDelays: 0
      AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
rayChan.DopplerSpectrum
```

```
ans = struct with fields:
    SpectrumType: 'Jakes'
```

Modify the Doppler spectrum

Create a flat Doppler spectrum structure, and then assign it in the `rayChan` object.

```
doppFlat = doppler('Flat')
```

```
doppFlat = struct with fields:
    SpectrumType: 'Flat'
```

```
rayChan.DopplerSpectrum = doppFlat
```

```
rayChan =
  comm.RayleighChannel with properties:

            SampleRate: 9600
            PathDelays: 0
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
rayChan.DopplerSpectrum
```

```
ans = struct with fields:
    SpectrumType: 'Flat'
```

Create Rician Channel with Gaussian Doppler Spectrum

This example shows how to change the default Jakes Doppler spectrum of a configured Rician channel object to a Gaussian Doppler spectrum with normalized standard deviation of 0.3. Then, display the `DopplerSpectrum` property and change the normalized standard deviation of the Doppler spectrum to 1.1.

Create a Rician channel object

Set the sample rate to 9600 Hz, the maximum Doppler shift to 100 Hz, and K factor to 2.

```
ricChan = comm.RicianChannel( ...
    'SampleRate',9600,'MaximumDopplerShift',100,'KFactor',2)
```

```
ricChan =
  comm.RicianChannel with properties:

               SampleRate: 9600
               PathDelays: 0
          AveragePathGains: 0
        NormalizePathGains: true
                   KFactor: 2
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 100
           DopplerSpectrum: [1x1 struct]
          ChannelFiltering: true
       PathGainsOutputPort: false

  Show all properties
```

```
ricChan.DopplerSpectrum

ans = struct with fields:
    SpectrumType: 'Jakes'
```

Modify the Doppler spectrum

Create a Gaussian Doppler spectrum structure with normalized standard deviation of 0.3 and assign it in the `ricChan` object.

```
doppGaus = doppler('Gaussian',0.3)

doppGaus = struct with fields:
                  SpectrumType: 'Gaussian'
    NormalizedStandardDeviation: 0.3000


ricChan.DopplerSpectrum = doppGaus

ricChan =
  comm.RicianChannel with properties:

               SampleRate: 9600
               PathDelays: 0
          AveragePathGains: 0
        NormalizePathGains: true
                  KFactor: 2
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 100
           DopplerSpectrum: [1x1 struct]
          ChannelFiltering: true
       PathGainsOutputPort: false

  Show all properties


ricChan.DopplerSpectrum

ans = struct with fields:
                  SpectrumType: 'Gaussian'
    NormalizedStandardDeviation: 0.3000


ricChan.DopplerSpectrum.NormalizedStandardDeviation = 1.1;
ricChan.DopplerSpectrum

ans = struct with fields:
                  SpectrumType: 'Gaussian'
    NormalizedStandardDeviation: 1.1000
```

Create Rayleigh Channel Using Independent Doppler Spectrum

This example shows how to change the default Jakes Doppler spectrum of a configured three-path Rayleigh channel object to a cell array of different Doppler spectra, and then change the properties of the Doppler spectrum of the third path.

Create a Rayleigh channel object

Set the sample rate to 9600 Hz, the maximum Doppler shift to 100 Hz, and specify path delays of `0`, `1e-4`, and `2.1e-4 seconds`.

```
rayChan = comm.RayleighChannel( ...
    'SampleRate',9600, ...
    'MaximumDopplerShift',100, ...
    'PathDelays',[0 1e-4 2.1e-4])
```

```
rayChan =
  comm.RayleighChannel with properties:

             SampleRate: 9600
             PathDelays: [0 1.0000e-04 2.1000e-04]
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
rayChan.DopplerSpectrum
```

```
ans = struct with fields:
    SpectrumType: 'Jakes'
```

Modify the Doppler spectrum

Specify the `DopplerSpectrum` property as a cell array with an independent Doppler spectrum for each path.

```
rayChan.DopplerSpectrum = {doppler('Flat') doppler('Flat') doppler('Rounded')}
```

```
rayChan =
  comm.RayleighChannel with properties:

             SampleRate: 9600
             PathDelays: [0 1.0000e-04 2.1000e-04]
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: {[1x1 struct]  [1x1 struct]  [1x1 struct]}
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
rayChan.DopplerSpectrum{:}
```

```
ans = struct with fields:
    SpectrumType: 'Flat'
```

```
ans = struct with fields:
    SpectrumType: 'Flat'


ans = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [1 -1.7200 0.7850]
```

Change the `Polynomial` property for the third path.

```
rayChan.DopplerSpectrum{3}.Polynomial = [1 -1.21 0.7]
```

```
rayChan =
  comm.RayleighChannel with properties:

             SampleRate: 9600
             PathDelays: [0 1.0000e-04 2.1000e-04]
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: {[1x1 struct]  [1x1 struct]  [1x1 struct]}
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

```
rayChan.DopplerSpectrum{:}
```

```
ans = struct with fields:
    SpectrumType: 'Flat'


ans = struct with fields:
    SpectrumType: 'Flat'


ans = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [1 -1.2100 0.7000]
```

Configure Channel Objects

Before you filter a signal using a channel object, make sure that the properties of the channel have suitable values for the situation you want to model. This section offers some guidelines to help you choose realistic values that are appropriate for your modeling needs. The topics are

- “Choose Realistic Channel Property Values” on page 22-22
- “Configure Channel Objects Based on Simulation Needs” on page 22-24

The syntaxes for viewing and changing values of properties of channel objects are described in Specifying a Fading Channel on page 22-13.

Choose Realistic Channel Property Values

Here are some tips for choosing property values that describe realistic channels:

Path Delays

- By convention, the first delay is typically set to zero. The first delay corresponds to the first arriving path.
- For indoor environments, path delays after the first are typically between 1e-9 seconds and 1e-7 seconds.
- For outdoor environments, path delays after the first are typically between 1e-7 seconds and 1e-5 seconds. Large delays in this range might correspond, for example, to an area surrounded by mountains.
- The ability of a signal to resolve discrete paths is related to its bandwidth. If the difference between the largest and smallest path delays is less than about 1% of the symbol period, then the signal experiences the channel as if it had only one discrete path.

Average Path Gains

- The average path gains in the channel object indicate the average power gain of each fading path. In practice, an average path gain value is a large negative dB value. However, computer models typically use average path gains in the range of [-20, 0] dB.
- The dB values in a vector of average path gains often decay roughly linearly as a function of delay, but the specific delay profile depends on the propagation environment.
- To ensure that the expected total power of the combined path gains is 1, you can normalize path gains via the `NormalizePathGains` property of the channel object.

Maximum Doppler Shifts

- Doppler shifts are specified in terms of the relative speed between a transmitter and a receiver. The maximum Doppler shift in Hertz, $f_d = vf / c$. In the formula, v is the relative speed in m/s, f is the transmission carrier frequency in Hertz, and c is the speed of light (3×10^8 m/s). The relative speed is a magnitude with no directional information.
- Apply the maximum Doppler shift formula assuming a transmission carrier frequency of 900 MHz, a car moving at freeway speed, and a walking pedestrian. A signal transmitted from a car moving at freeway speed to a stationary receiver would experience a maximum Doppler shift of approximately 80 Hz. A signal transmitted from a mobile held by a walking pedestrian to a stationary receiver would experience a maximum Doppler shift of approximately 4 Hz.
- A maximum Doppler shift of 0 corresponds to a static channel that comes from a Rayleigh or Rician distribution.

Doppler Spectrum

- The Doppler spectrum used for the channel paths must be outputs of the form returned from the `doppler` function.
- Options for the spectrum type are specified by the `specType` input to the `doppler` function.

K-Factor for Rician Fading Channels

- The Rician K-factor specifies the ratio of specular-to-diffuse power for a direct line-of-sight path. The ratio is expressed linearly, not in dB.
- For Rician fading, the K-factor is typically in the range [1, 10].
- A K-factor of 0 corresponds to Rayleigh fading.

Line-of-Sight (LOS) Path Doppler Shift for Rician Fading Channels

- The Rician LOS path Doppler shift, also known as direct path Doppler shift, specifies the relative motion of the LOS path between a transmitter and a receiver.
- The Rician LOS path Doppler shift in Hertz, $f_{\mathrm{d\_los}} = (u \cdot w) \times f / c$. In the formula, $(u \cdot w)$ is the dot product of vectors u and w, u is the normalized LOS path from the transmitter to the receiver, w is the velocity of the receiver relative to transmitter, f is the transmission carrier frequency in Hertz, and c is the speed of light (3×10^8 m/s).
- Apply this formula for a transmission carrier frequency of 900 MHz at a specified relative velocity. For a signal from a transmitter at the coordinate origin that reaches a receiver at the coordinate [100 100 0], where the relative velocity between the transmitter and receiver w = [3 -6 0.1]. The LOS path Doppler shift is 4.25 Hz.

Doppler Spectrum Parameters

- See the `doppler` reference page for the respective Doppler spectrum structures for descriptions of the parameters and their significance.

Configure Channel Objects Based on Simulation Needs

Tips for configuring a channel object to customize the filtering process:

- If your data is partitioned into a series of vectors (that you process within a loop, for example), you can call the channel object multiple times (in each iteration within a loop). The state information of the channel is updated and saved after each invocation. The channel output is irrelevant to how the data is partitioned (vector length).
- If want the channel output to be repeatable, choose the seed option for the `RandomStream` property of the channel object. To repeat the output, call the `reset` object function to reset both the internal filters and the internal random number generator.
- If you want to model discontinuously transmitted data, set the `FadingTechnique` property to `'Sum of sinusoids'` and the `InitialTimeSource` property to `'Input port'` for the channel object. When calling the object, specify the start time of each data vector/frame to be processed by the channel via an input.
- If you want to normalize the fading process so that the expected value of the path gains' total power is 1 (the channel does not contribute additional power gain or loss), set the `NormalizePathGains` property of the channel object to `true`.

Use Fading Channels

After you have created a channel object as described in Specifying a Fading Channel on page 22-13, you can call the object to pass a signal through the channel. Provide the signal as an input argument to the channel object. At the end of the filtering operation, the channel object retains its state so that you can find out the final path gains or the total number of samples that the channel has processed by calling the info object function with the object as the input argument.

For an example that illustrates the basic syntax and state retention, see Power of a Faded Signal on page 22-26.

To visualize the characteristics of the channel, set the `Visualization` property to `'Impulse response'`, `'Frequency response'`, or `'Doppler spectrum'`. For more information, see Channel Visualization on page 25-27.

Visualize Three-Path Rayleigh Channel

This example shows you how to visualize impulse response of a channel.

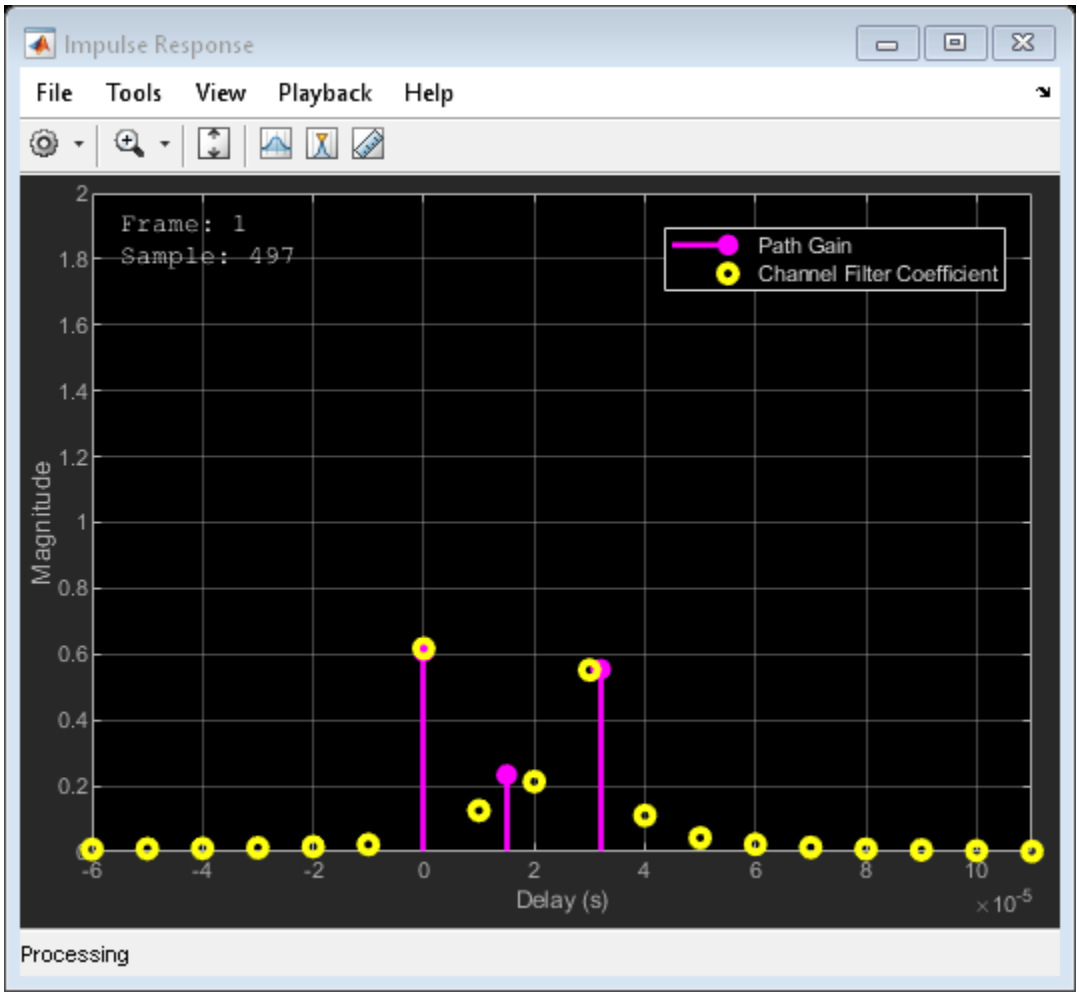
Create a channel object

While creating the channel object, use name-value pairs to set the `Visualization` property to `'Impulse response'`.

```
rayChan = comm.RayleighChannel('SampleRate',100000,'MaximumDopplerShift',130,...
    'PathDelays',[0 1.5e-5 3.2e-5],'AveragePathGains',[0, -3, -3],...
    'Visualization','Impulse response');
```

Generate a bit stream and create a modulator object. Modulate the bit stream and pass the modulated DBPSK signal through the channel by calling the channel object.

```
tx = randi([0 1],500,1);
dbspkMod = comm.DBPSKModulator;
dpskSig = dbspkMod(tx);
y = rayChan(dpskSig);
```



The impulse response is plotted when the object is called.

Rayleigh Fading Channel

These examples use fading channels:

Power of Faded Signal

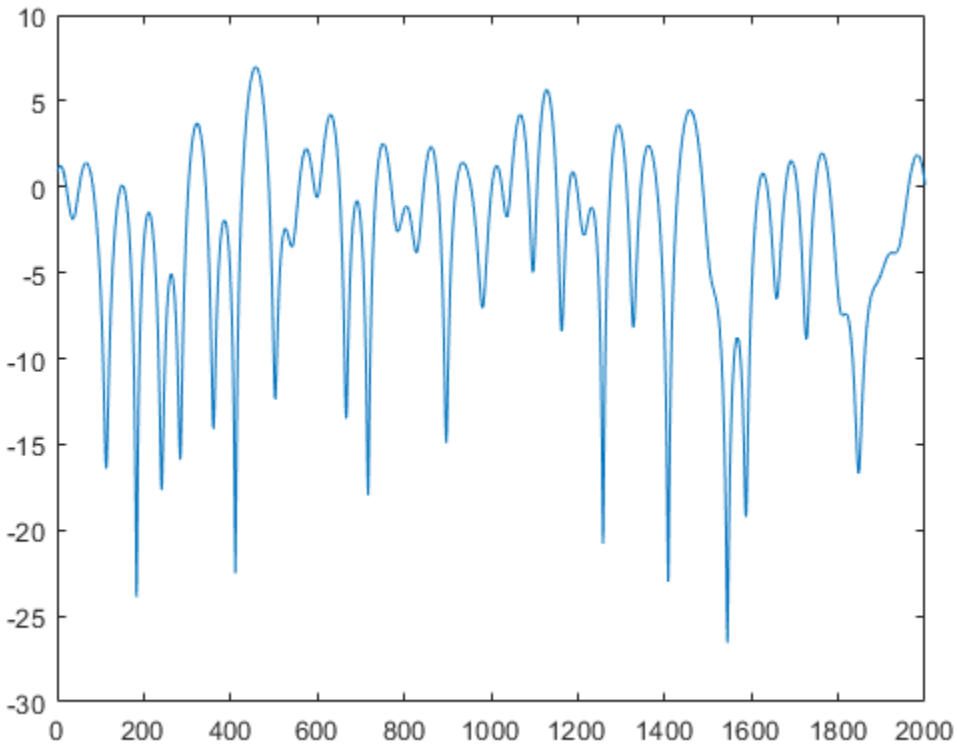
This example plots the power of a faded signal versus sample number. It illustrates the syntax of creating and calling a `comm.RayleighChannel` fading channel object and the state retention of the channel object.

```
rayChan = comm.RayleighChannel('SampleRate',10000,'MaximumDopplerShift',100);
sig = j*ones(2000,1); % Signal
out = rayChan(sig); % Pass signal through channel.
rayChan % Display all properties of the channel object.
```

```
rayChan =
  comm.RayleighChannel with properties:

             SampleRate: 10000
             PathDelays: 0
       AveragePathGains: 0
     NormalizePathGains: true
    MaximumDopplerShift: 100
        DopplerSpectrum: [1x1 struct]
       ChannelFiltering: true
    PathGainsOutputPort: false

  Show all properties
```

Plot power of faded signal, versus sample number.

```
plot(20*log10(abs(out)))
```



DBPSK Empirical Versus Theoretical Performance in Fading Conditions

This example creates a frequency-flat Rayleigh fading channel object and calls it to process a DBPSK signal consisting of a single vector. Bit error rate is computed for different values of the signal-to-noise ratio. The fading channel filter is applied before AWGN. This is the recommended sequence to use when you combine fading with AWGN.

Create Rayleigh fading channel, modulator, and demodulator objects

```
chan = comm.RayleighChannel('SampleRate',1e4,'MaximumDopplerShift',100);
```

Create DBPSK modulator and demodulator objects with the modulation order set to 2. Generate DBPSK modulated data and pass it through the channel.

```
M = 2; % DBPSK modulation order
tx = randi([0 M-1],50000,1); % Generate a random bit stream

mod = comm.DBPSKModulator;
demod = comm.DBPSKDemodulator;

dpskSig = mod(tx);
fadedSig = chan(dpskSig); % Apply the channel effects
```

Create an AWGN channel object and an error rate calculator object.

```
awgnChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)');
errorCalc = comm.ErrorRate;
```

Compute error rate for different values of SNR.

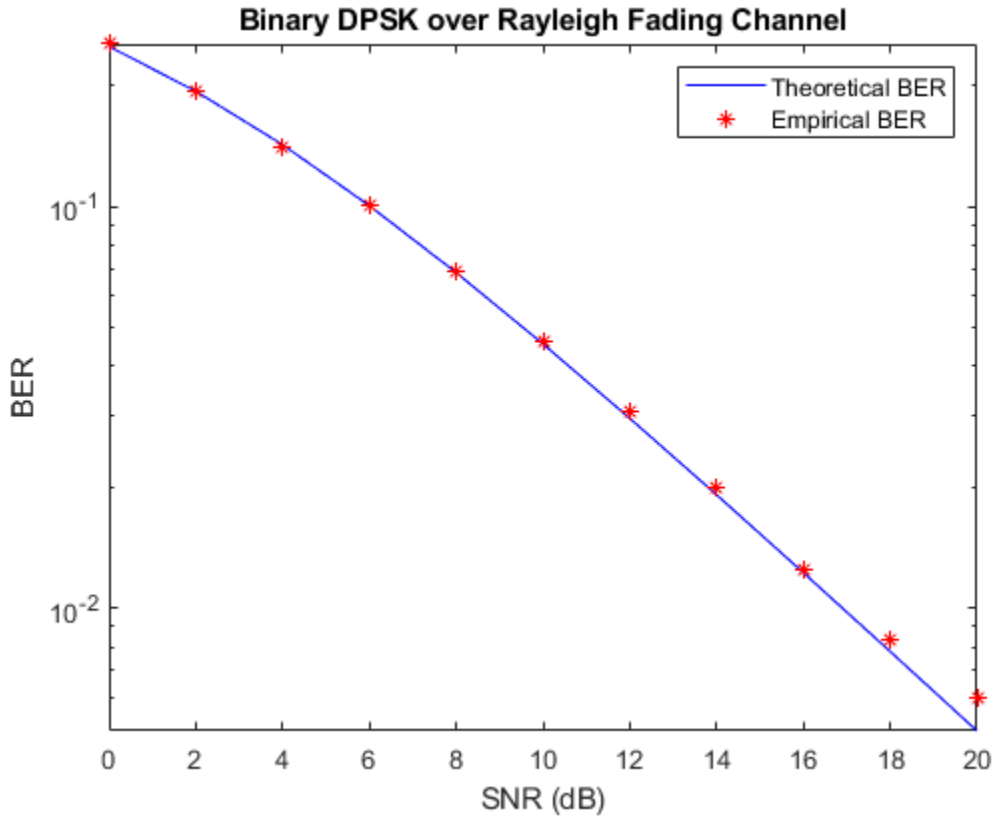
```
SNR = 0:2:20; % Range of SNR values, in dB.
numSNR = length(SNR);
berVec = zeros(3, numSNR); % Preallocate a vector for BER results

for n = 1:numSNR
   awgnChan.SNR = SNR(n);
   rxSig = awgnChan(fadedSig); % Add Gaussian noise
   rx = demod(rxSig);  % Demodulate
   reset(errorCalc)

   berVec(:,n) = errorCalc(tx,rx); % Compute error rate.
end
BER = berVec(1,:);
```

Compute theoretical performance results, for comparison.

```
BERtheory = berfading(SNR,'dpsk',M,1);
```

Plot BER results.

```
semilogy(SNR,BERtheory,'b-',SNR,BER,'r*');
legend('Theoretical BER','Empirical BER');
xlabel('SNR (dB)'); ylabel('BER');
title('Binary DPSK over Rayleigh Fading Channel');
```



Work with Channel Filter Delays

The value of a channel object's ChannelFilterDelay property is the number of samples by which the output of the channel lags the input. If you compare the input and output data sets directly, you must take the delay into account by using appropriate truncating or padding operations.

The example illustrates a way to account for the delay before computing a bit error rate.

Create DBPSK modulator and demodulator objects with the modulation order set to 2. Generate DBPSK modulated data and pass it through the channel.

```
bitRate = 50000;
M = 2; % DQPSK modulation order

mod = comm.DBPSKModulator;
demod = comm.DBPSKDemodulator;
```

Create Rayleigh fading channel object.

```
rayChan = comm.RayleighChannel('SampleRate',bitRate,'MaximumDopplerShift',4,...
    'PathDelays',[0 0.5/bitRate],'AveragePathGains',[0 -10]);
chInfo = info(rayChan);
delay = chInfo.ChannelFilterDelay;
```

Generate random bit stream data. Modulate the data, pass it through the fading channel, and demodulate it.

```
tx = randi([0 M-1],50000,1);

dpskSig = mod(tx);
fadedSig = rayChan(dpskSig);
rx = demod(fadedSig);
```

Compute bit error rate, taking delay into account.

```
errorCalc = comm.ErrorRate('ReceiveDelay', delay);
berVec = step(errorCalc,tx,rx);
ber = berVec(1)
```

```
ber = 0.0147
```

```
num = berVec(2)
```

```
num = 737
```

Channel Filtering Using For Loop

This example filters input data through a Rayleigh fading channel within a `for` loop. It uses the small data sets from successive iterations to create an animated effect. The Rayleigh fading channel has two discrete major paths. For information on filtering data through a channel multiple times while maintaining continuity from one invocation to the next, see “Configure Channel Objects Based on Simulation Needs” on page 22-24.

Set up parameters. Specify a bit rate of `50e3` Hz, and a loop iteration count of `125`. Create a QPSK modulator and Rayleigh fading channel objects.

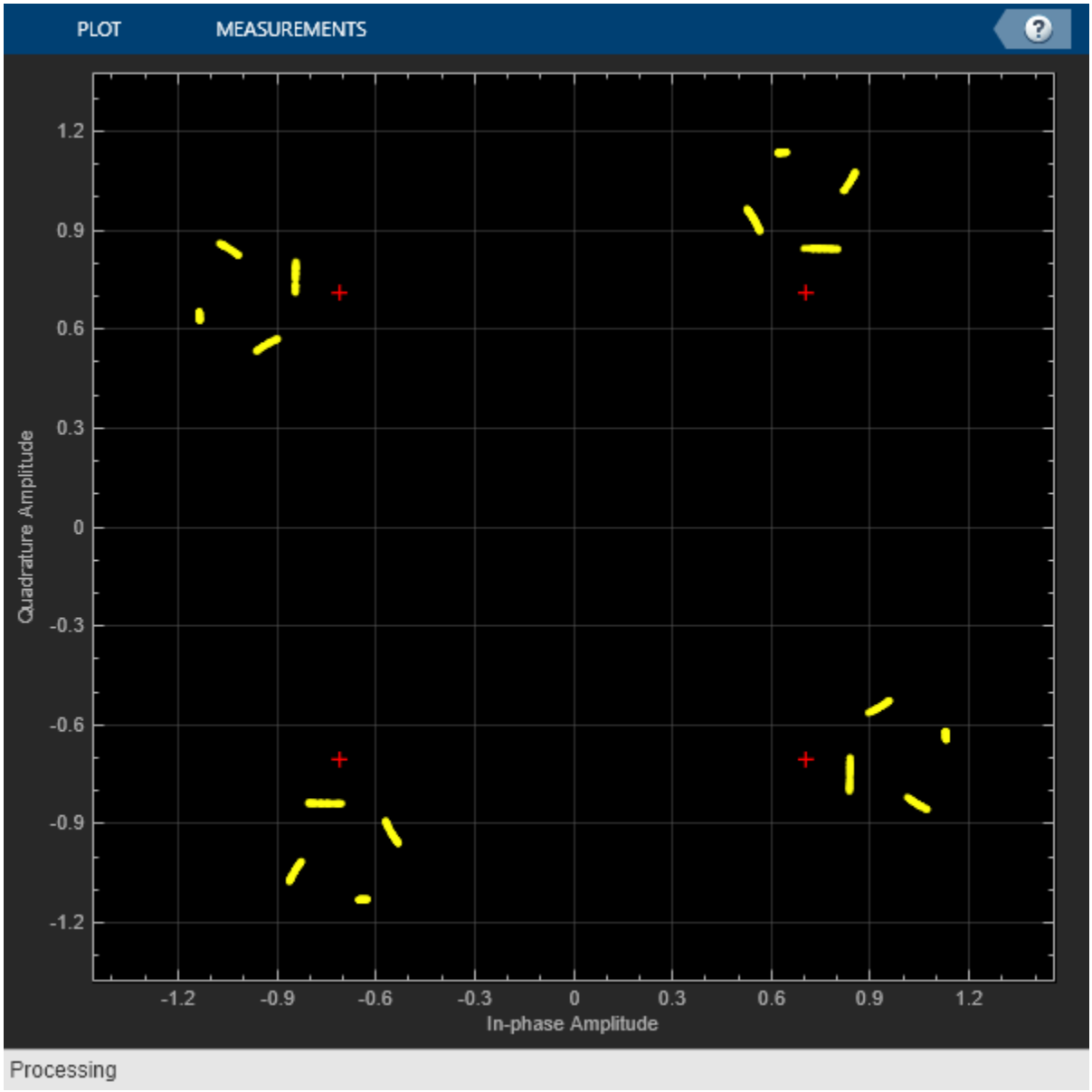
```
bitRate = 50000;    % Data rate is 50 kb/s
numTrials = 125;    % Number of iterations of loop

M = 4; % QPSK modulation order
qpskMod = comm.QPSKModulator;

rayChan = comm.RayleighChannel('SampleRate',bitRate,'MaximumDopplerShift',4,'PathDelays',[0 2e-5
```

Initialize a scatter plot.

```
scatterPlot = comm.ConstellationDiagram;
```

Apply channel in a loop, maintaining continuity. Plot only the current data in each iteration.

```
for n = 1:numTrials
   tx = randi([0 M-1],500,1); % Generate random bit stream
   pskSig = qpskMod(tx); % PSK modulate signal
   fadedSig = rayChan(pskSig); % Apply channel effects

   % Plot the new data from this iteration.
   update(scatterPlot,fadedSig);
end
```



Rician Fading Channel

Quasi-Static Channel Modeling

Typically, a path gain in a fading channel changes insignificantly over a period of `1/(100fd)` seconds, where `fd` is the maximum Doppler shift. Because this period corresponds to a very large number of bits in many modern wireless data applications, assessing performance over a statistically significant range of channel fading requires simulating a prohibitively large amount of data. This example illustrates the quasi-static channel modeling approach to gathering a statistically significant number of errors. Quasi-static channel modeling provides a more tractable approach, which you can implement using these steps:

1 Generate a random channel realization using a maximum Doppler shift of 0.

2. Process some large number of bits.
3. Compute error statistics.
4. Repeat these steps many times to produce a distribution of the performance metric.

Create modulator and demodulator system objects with a modulation order of 4.

```
M = 4;
dpskMod = comm.DPSKModulator('ModulationOrder',M);
dpskDemod = comm.DPSKDemodulator('ModulationOrder',M);
numBits = 10000; % Each trial uses 10000 bits
numTrials = 20;  % Number of BER computations
```

Typically, `numTrials` would be a large number to get an accurate estimate of outage probabilities or packet error rate. Use `20` here just to make the example run more quickly.

Create a Rician channel object and set the maximum Doppler shift to zero.

```
ricianChan = comm.RicianChannel('KFactor',3,'MaximumDopplerShift',0);
```

Within a `for` loop, generate a random bit stream, DPSK modulate the signal, filter the modulated signal through a Rician fading and AWGN channels, and demodulate the faded signal. For the symbol error rate computation on each packet, ignore the first sample because of DPSK initial condition.

```
nErrors = zeros(1,numTrials);
for n = 1:numTrials
   tx = randi([0 M-1],numBits,1);            % Generate random bit stream
   dpskSig = dpskMod(tx);                    % DPSK modulate signal
   fadedSig = ricianChan(dpskSig);           % Apply channel effects
   rxSig = awgn(fadedSig,15,'measured');     % Add Gaussian noise
   rx = dpskDemod(rxSig);                    % Demodulate
   nErrors(n) = symerr(tx(2:end),rx(2:end)); % Symbol error computation, beginning with sample in
end
```

After the `for` loop ends, display the list of symbol error counts in the vector `nErrors` and a computation of the packet error rate. Run to run results vary due to randomness in the example.

```
nErrors
```

```
nErrors = 1×20

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

```
per = mean(nErrors > 0) % Proportion of packets that had errors
```

```
per = 0.0500
```

More About the Quasi-Static Technique

As an example to show how the quasi-static channel modeling approach can save computation, consider a wireless local area network (LAN) in which the carrier frequency is 2.4 GHz, mobile speed is 1 m/s, and bit rate is 10 Mb/s. The following expression shows that the channel changes insignificantly over 12,500 bits:

$$\left(\frac{1}{100 f_d}\ \text{s}\right)(10\ \text{Mb/s}) = \left(\frac{c}{100 v f}\ \text{s}\right)(10\ \text{Mb/s})$$

$$= \frac{3 \times 10^8 \text{m/s}}{100(1\ \text{m/s})(2.4\ \text{GHz})}(10\ \text{Mb/s})$$

$$= 12,500\ \text{b}$$

A traditional Monte Carlo approach for computing the error rate of this system would entail simulating thousands of times, totalling tens of millions of bits. By contrast, a quasi-static channel modeling approach would simulate a few packets at each of about 100 locations to arrive at a spatial distribution of error rates. From this distribution one could determine, for example, how reliable the communication link is for a random location within the indoor space. If each simulation contains 5,000 bits, 100 simulations would process half a million bits in total. This is substantially fewer bits compared to the traditional Monte Carlo approach.

Additional Examples Using Fading Channels

- “Multipath Fading Channel in Simulink” on page 8-182
- “Defense Communications: US MIL-STD-188-110B Baseband End-to-End Link” on page 8-405
- “WCDMA End-to-End Physical Layer” on page 8-410

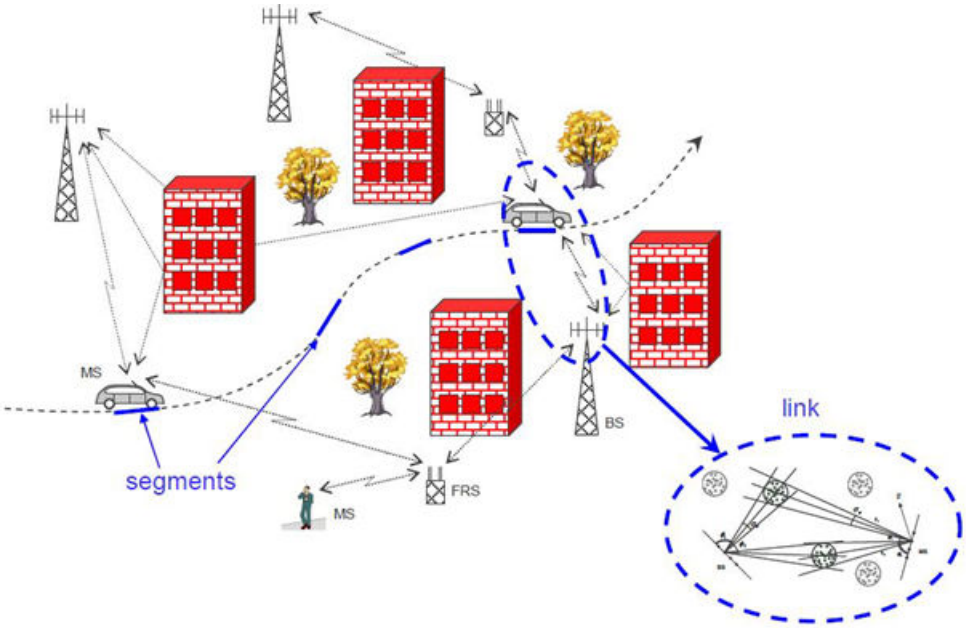
Using Channel Visualization

Communications Toolbox channel objects include a `Visualization` property that enables you to visualize the characteristics of a fading channel when calling the channel object. For more information, see Channel Visualization on page 25-27.

WINNER II Channel

Using WINNER II channel models, you can model and simulate spatially defined channels for multiuser MIMO wireless systems. In the model you can specify an arbitrary number of base stations (BS) and mobile stations (MS) together with their geometry and location information.

Figure 3.1 from [1] (shown here), depicts a system level simulation including multiple base stations and multiple mobile terminals. Within the figure the dashed blue line surrounding a car and cell tower, highlights a link level simulation for the link between one mobile terminal and a base station. The short blue lines, along the path of the car, represent channel segments where large scale parameters are fixed. The system level simulation consists of multiple links. Each link is modeled using the clustered delay line (CDL) method. The inset shows a CDL method model of one link in the scenario.



The channel model enables you to simulate line-of-sight (LOS) and non-LOS propagation conditions. The model also enables you to apply multiple indoor and outdoor propagation scenarios. You can perform channel filtering in a streaming fashion with WINNER-generated channel coefficients.

The channel model supports:

- RF frequencies up to 6 GHz
- Signal bandwidths up to 100 MHz
- LOS and non-LOS propagation
- 12 indoor and outdoor propagation scenarios
- Arbitrarily large antenna arrays (for massive MIMO applications)
- Isotropic, dipole, and user-defined antenna element patterns
- A variety of antenna array types (such as linear, circular, and user-defined)

To use this functionality, download and install the WINNER II Channel Model for Communications Toolbox add-on.

The add-on includes the `comm.WINNER2Channel` System object and provides the capability currently available in the open source download in [1]. The functionality in the download includes these functions:

- `winner2.AntennaArray` — Construct antenna array
- `winner2.dipole` — Calculate field pattern of half wavelength dipole
- `winner2.layoutparset` — WINNER II layout parameter configuration
- `winner2.wim` — Generate channel coefficients using WINNER II channel model
- `winner2.wimparset` — WINNER II model parameter configuration

The add-on extends the open source download by adding the capability to generate channel coefficients for use in channel filtering. For more information, see Mapping WINNER II Public Download to WINNER2Channel on page 22-37.

These examples demonstrate some of the WINNER II fading channel features.

- Simultaneous Simulation of Multiple Fading Channels with WINNER II Channel Model on page 8-117
- 802.11ac Multi-User MIMO Precoding with WINNER II Channel Model on page 8-123

examples demonstrate some of the WINNER II fading channel features.

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.dipole` | `winner2.layoutparset` | `winner2.wim` | `winner2.wimparset`

More About

- WINNER II Channel Model Video
- Mapping WINNER II Public Download to WINNER2Channel on page 22-37
- Simultaneous Simulation of Multiple Fading Channels with WINNER II Channel Model on page 8-117
- 802.11ac Multi-User MIMO Precoding with WINNER II Channel Model on page 8-123

Mapping of WINNER II Open Source Download to WINNER II Channel Model for Communications Toolbox

The WINNER II Channel Model for Communications Toolbox is composed of the `comm.WINNER2Channel` System object and functions. The functions in the WINNER II Channel Model for Communications Toolbox map to functions in the WINNER II open source download.

Function in WINNER II Open Source Download	Corresponding Function in WINNER II Channel Model for Communications Toolbox
`AntennaArray`	`winner2.AntennaArray`
`AntennaResponse`	`winner2.internal.calcAntennaResponse`
`arrayparset`	`winner2.arrayparset`
`dipole`	`winner2.dipole`
`layoutparset`	`winner2.layoutparset`
`layout2link`	`winner2.internal.layout2Link`
`wim`	`winner2.wim`
`wimparset`	`winner2.wimparset`

The following table shows the behavioral changes between the WINNER II open source download and the WINNER II Channel Model for Communications Toolbox.

Behavioral Condition	WINNER II Public Download Behavior	WINNER II Channel Model for Communications Toolbox Behavior
Default value of the `SampleDensity` field in the structure returned by the `winner2.wimparset` function	2	2e6
Default velocity returned for each MS by `winner2.layoutparset`.	10 m/s	1.43 m/s Corresponding to approximate typical walking speed, V_{MS}. $V_{MS} = C / F_{center} \times 25$ = (2.99792458 e8/5.25e9) × 25 ~ 1.43 m/s
The length of the third dimension of the channel coefficients output of the `winner2.wim` function	Equals the maximum number of paths or maximum number of paths plus four for all links with zero padding, `NaN` padding, or a combination of zero and `NaN` padding	Equals the number of paths for the specific link
The number of paths shown in the channel coefficients and path delay outputs of the `winner2.wim` function	Mismatched for many cases when there is more than one link	Matched for each link

Behavioral Condition	WINNER II Public Download Behavior	WINNER II Channel Model for Communications Toolbox Behavior
Strongest cluster segregation when the `IntraClusterDsUsed` field is set to `'yes'`	The two strongest clusters are divided into three subclusters only for the links that have the maximum number of paths. If the second and third strongest paths have the same power, only the single strongest cluster is divided into three subclusters.	For each link, the two strongest clusters are divided into three subclusters.
Updating of the `Phi_LOS` field in the third structure output of the `winner2.wim` function	Updated when the `IntraClusterDsUsed` field is set to `'no'`	Updated regardless of the setting of the `IntraClusterDsUsed` field
Padding of the path delay output of the `winner2.wim` function, when the rows (number of links) have fewer than the maximum number of paths	Zero padded when the `IntraClusterDsUsed` field is set to `'yes'`	`NaN` padded
The channel coefficients calculation specified by [1], Equation 4.14 and Table 4-2, when the `IntraClusterDsUsed` and `PolarisedArrays` fields are set to `'yes'`	Incorrect	Correct
Path loss calculation for A1 NLOS links	Incorrect when the `PathLossOption` field is set to `'CR_heavy'` or `'CR_light'`	Correct
The `Phi_LOS` field per step 10 on page 40 of [1] for the input initial condition and output final condition of `winner2.wim`, should be of size N_L-by-2 to log the phases for both VV and HH polarization each link. N_L is the number of links.	Incorrect, `Phi_LOS` is of size N_L-by-1 for VV polarization only. The phase for HH polarization is not included. This causes issues for a link with LOS path.	Correct, `Phi_LOS` is of size N_L-by-2 to log the phases for both VV and HH polarization each link.

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

`comm.WINNER2Channel`

More About

- “WINNER II Channel” on page 22-35

- Simultaneous Simulation of Multiple Fading Channels with WINNER II Channel Model on page 8-117
- 802.11ac Multi-User MIMO Precoding with WINNER II Channel Model on page 8-123

Measurements

Bit Error Rate Analysis Techniques

In this section...

This topic describes how to compute error statistics for various communications systems.

Computation of Theoretical Error Statistics

The `biterr` function, discussed in the “Compute SERs and BERs Using Simulated Data” on page 23-6 section,can help you gather empirical error statistics, but validating your results by comparing them to the theoretical error statistics is good practice. For certain types of communications systems, closed-form expressions exist for the computation of the bit error rate (BER) or an approximate bound on the BER. The functions listed in this table compute the closed-form expressions for the BER or a bound on it for the specified types of communications systems.

Type of Communications System	Function
Uncoded AWGN channel	`berawgn`
Uncoded Rayleigh and Rician fading channel	`berfading`
Coded AWGN channel	`bercoding`
Uncoded AWGN channel with imperfect synchronization	`bersync`

The analytical expressions used in these functions are discussed in “Analytical Expressions and Notations Used in BER Analysis” on page 23-45. The reference pages of these functions also list references to one or more books containing the closed-form expressions implemented by the function.

Theoretical Performance Results

- “Plot Theoretical Error Rates” on page 23-2
- “Compare Theoretical and Empirical Error Rates” on page 23-3

Plot Theoretical Error Rates

This example uses the `bercoding` function to compute upper bounds on BERs for convolutional coding with a soft-decision decoder.

```
coderate = 1/4; % Code rate
```

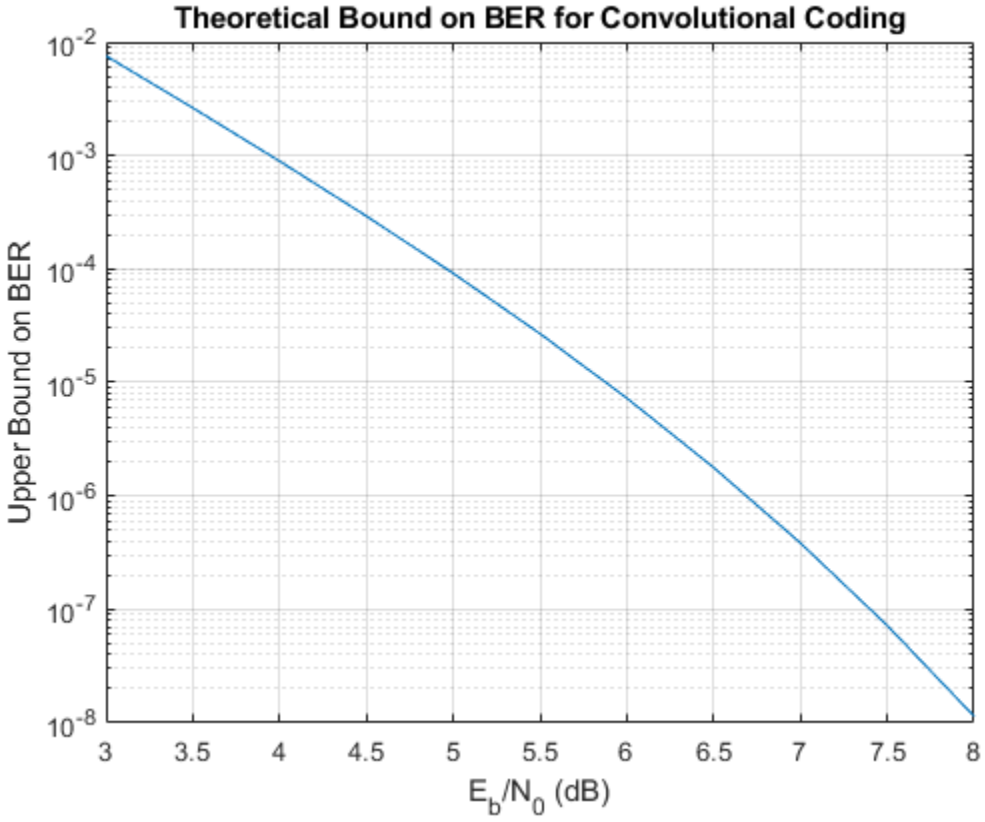
Create a structure, `dspec`, with information about the distance spectrum. Define the energy per bit to noise power spectral density ratio (E_b/N_0) sweep range and generate the theoretical bound results.

```
dspec.dfree = 10; % Minimum free distance of code
dspec.weight = [1 0 4 0 12 0 32 0 80 0 192 0 448 0 1024 ...
    0 2304 0 5120 0]; % Distance spectrum of code
```

```
EbNo = 3:0.5:8;
berbound = bercoding(EbNo,'conv','soft',coderate,dspec);
```

Plot the theoretical bound results.

```
semilogy(EbNo,berbound)
xlabel('E_b/N_0 (dB)');
ylabel('Upper Bound on BER');
title('Theoretical Bound on BER for Convolutional Coding');
grid on;
```



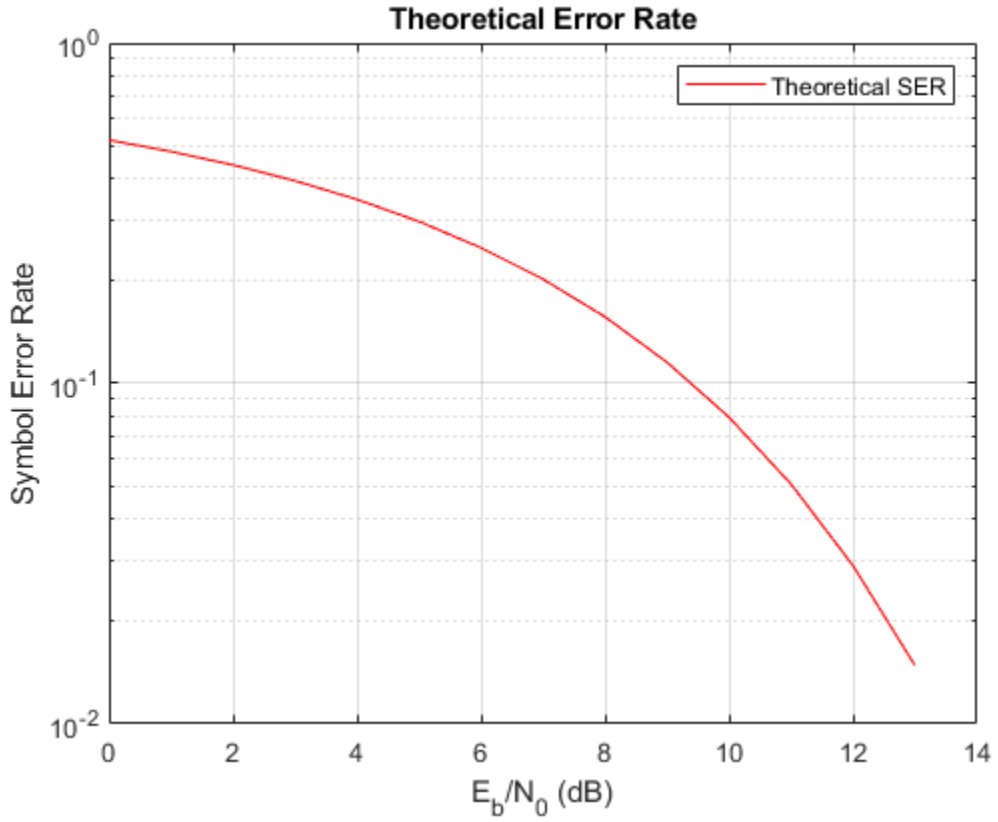
Compare Theoretical and Empirical Error Rates

Using the `berawgn` function, compute the theoretical symbol error rates (SERs) for pulse amplitude modulation (PAM) over a range of E_b/N_0 values. Simulate 8 PAM with an AWGN channel, and compute the empirical SERs. Compare the theoretical and then empirical SERs by plotting them on the same set of axes.

Compute and plot the theoretical SER using `berawgn`.

```
rng('default') % Set random number seed for repeatability
M = 8;
EbNo = 0:13;
[ber,ser] = berawgn(EbNo,'pam',M);

semilogy(EbNo,ser,'r');
```

```
legend('Theoretical SER');
title('Theoretical Error Rate');
xlabel('E_b/N_0 (dB)');
ylabel('Symbol Error Rate');
grid on;
```



Compute the empirical SER by simulating an 8 PAM communications system link. Define simulation parameters and preallocate variables needed for the results. As described in [1], because $N_0 = 2 \times (N_{\text{Variance}})^2$, add 3 dB to the E_b/N_0 value when converting E_b/N_0 values to SNR values.

```
n = 10000; % Number of symbols to process
k = log2(M); % Number of bits per symbol
snr = EbNo+3+10*log10(k); % In dB
ynoisy = zeros(n,length(snr));
z = zeros(n,length(snr));
errVec = zeros(3,length(EbNo));
```

Create an error rate calculator System object to compare decoded symbols to the original transmitted symbols.

```
errcalc = comm.ErrorRate;
```

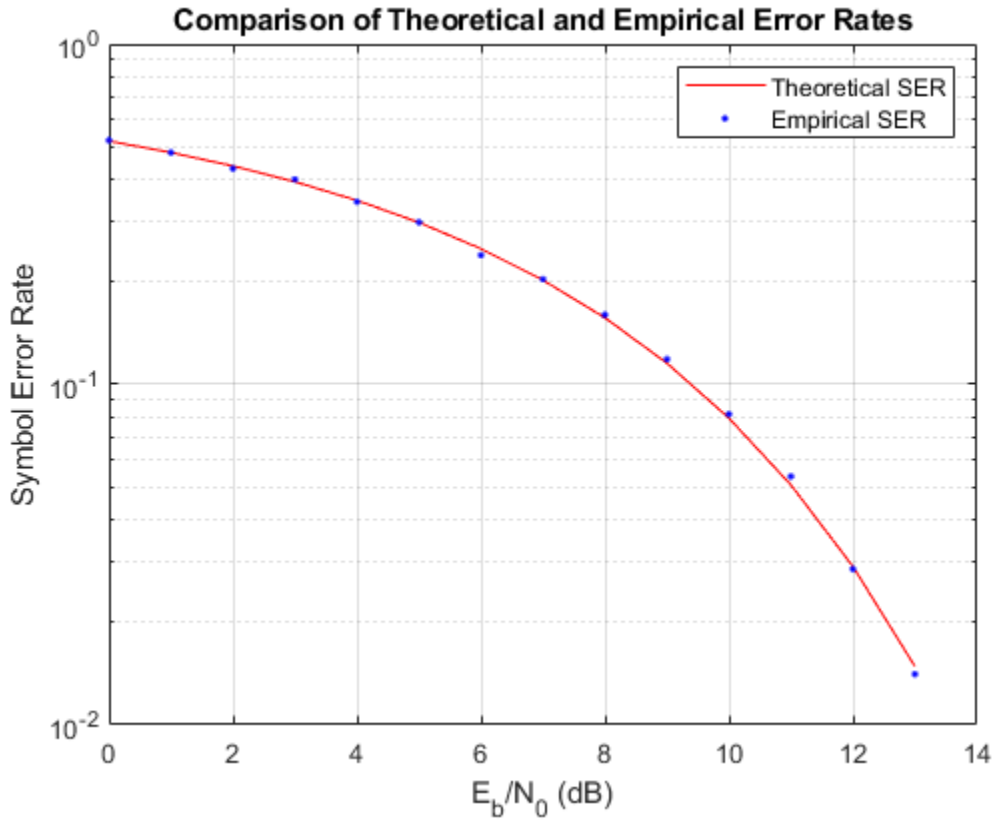
Generate a random data message and apply PAM. Normalize the channel to the signal power. Loop the simulation to generate error rates over the range of SNR values.

```
x = randi([0 M-1],n,1); % Create message signal
y = pammod(x,M); % Modulate
```

```
signalpower = (real(y)'*real(y))/length(real(y));

for jj = 1:length(snr)
    reset(errcalc)
    ynoisy(:,jj) = awgn(real(y),snr(jj),'measured'); % Add AWGN
    z(:,jj) = pamdemod(complex(ynoisy(:,jj)),M); % Demodulate
    errVec(:,jj) = errcalc(x,z(:,jj)); % Compute SER from simulation
end
```

Compare the theoretical and empirical results.

```
hold on;
semilogy(EbNo,errVec(1,:),'b.');
legend('Theoretical SER','Empirical SER');
title('Comparison of Theoretical and Empirical Error Rates');
hold off;
```



Performance Results via Simulation

Section Overview

This section describes how to compare the data messages that enter and leave a communications system simulation and how to compute error statistics using the Monte Carlo technique. Simulations

can measure system performance by using the data messages before transmission and after reception to compute the BER or SER for a communications system. To explore physical layer components used to model and simulate communications systems, see “PHY Components”.

Curve fitting can be useful when you have a small or imperfect data set but want to plot a smooth curve for presentation purposes. To explore the use of curve fitting when computing performance results via simulation, see the “Curve Fitting for Error Rate Plots” on page 23-8 section.

Compute SERs and BERs Using Simulated Data

The example shows how to compute SERs and BERs using the `biterr` and `symerr` functions, respectively. The `symerr` function compares two sets of data and computes the number of symbol errors and the SER. The `biterr` function compares two sets of data and computes the number of bit errors and the BER. An error is a discrepancy between corresponding points in the two sets of data.

The two sets of data typically represent messages entering a transmitter and recovered messages leaving a receiver. You can also compare data entering and leaving other parts of your communications system (for example, data entering an encoder and data leaving a decoder).

If your communications system uses several bits to represent one symbol, counting symbol errors is different from counting bit errors. In either the symbol- or bit-counting case, the error rate is the number of errors divided by the total number of transmitted symbols or bits, respectively.

Typically, simulating enough data to produce at least 100 errors provides accurate error rate results. If the error rate is very small (for example, 10^{-6} or less), using the semianalytic technique might compute the result more quickly than using a simulation-only approach. For more information, see the “Performance Results via Semianalytic Technique” on page 23-8 section.

Compute Error Rates

Use the `symerr` function to compute the SERs for a noisy linear block code. Apply no digital modulation, so that each symbol contains a single bit. When each symbol is a single bit, the symbol errors and bit errors are the same.

After artificially adding noise to the encoded message, compare the resulting noisy code to the original code. Then, decode and compare the decoded message to the original message.

```
m = 3; % Set parameters for Hamming code
n = 2^m-1;
k = n-m;
msg = randi([0 1],k*200,1); % Specify 200 messages of k bits each
code = encode(msg,n,k,'hamming');
codenoisy = bsc(code,0.95); % Add noise
newmsg = decode(codenoisy,n,k,'hamming'); % Decode and correct errors
```

Compute the SERs

```
[~,noisyVec] = symerr(code,codenoisy);
[~,decodedVec] = symerr(msg,newmsg);
```

The error rate decreases after decoding because the Hamming decoder correct errors based on the error-correcting capability of the decoder configuration. Because random number generators produce the message and noise is added, results vary from run to run. Display the SERs.

```
disp(['SER in the received code: ',num2str(noisyVec(1))])
```

```
SER in the received code: 0.94571
```

```
disp(['SER after decoding: ',num2str(decodedVec(1))])
```

```
SER after decoding: 0.9675
```

Comparing SER and BER

These commands show the difference between symbol errors and bit errors in various situations.

Create two three-element decimal vectors and show the binary representation. The vector `a` contains three 2-bit symbols, and the vector `b` contains three 3-bit symbols.

```
a = [1 2 3]'; b = [1 4 4]';
de2bi(a)
```

```
ans = 3×2

     1     0
     0     1
     1     1
```

```
de2bi(b)
```

```
ans = 3×3

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

Compare the binary values of the two vectors and compute the number of errors and the error rate by using the `biterr` and `symerr` functions.

```
format rat % Display fractions instead of decimals
[snum,srate] = symerr(a,b)
```

```
snum =
       2


srate =
       2/3
```

`snum` is 2 because the second and third entries have bit differences. `srate` is `2/3` because the total number of symbols is 3.

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

```
bnum =
       5


brate =
       5/9
```

`bnum` is 5 because the second entries differ in two bits, and the third entries differ in three bits. `brate` is `5/9` because the total number of bits is 9. By definition, the total number of bits is the

number of entries in `a` for symbol error computations or `b` for bit error computations times the maximum number of bits among all entries of `a` and `b`, respectively.

Performance Results via Semianalytic Technique

The technique described in the “Performance Results via Simulation” on page 23-5 section can work for a large variety of communications systems but can be prohibitively time-consuming for small error rates (for example, 10^{-6} or less). The semianalytic technique is an alternative way to compute error rates. The semianalytic technique can produce results faster than a nonanalytic method that uses simulated data.

For more information on implementing the semianalytic technique using a combination of simulation and analysis to determine the error rate of a communications system, see the `semianalytic` function.

Error Rate Plots

Section Overview

Error rate plots can be useful when examining the performance of a communications system and are often included in publications. This section discusses and demonstrates tools you can use to create error rate plots, modify them to suit your needs, and perform curve fitting on the error rate data and the plots.

Creation of Error Rate Plots Using semilogy Function

In many error rate plots, the horizontal axis indicates E_b/N_0 values in dB, and the vertical axis indicates the error rate using a logarithmic (base 10) scale. For examples that create such a plot using the `semilogy` function, see “Compare Theoretical and Empirical Error Rates” on page 23-3 and “Plot Theoretical Error Rates” on page 23-2.

Curve Fitting for Error Rate Plots

Curve fitting can be useful when you have a small or imperfect data set but want to plot a smooth curve for presentation purposes. The `berfit` function includes curve-fitting capabilities that help your analysis when the empirical data describes error rates at different E_b/N_0 values. This function enables you to:

- Customize various relevant aspects of the curve-fitting process, such as a list of selections for the type of closed-form function used to generate the fit.
- Plot empirical data along with a curve that `berfit` fits to the data.
- Interpolate points on the fitted curve between E_b/N_0 values in your empirical data set to smooth the plot.
- Collect relevant information about the fit, such as the numerical values of points along the fitted curve and the coefficients of the fit expression.

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.

Use Curve Fitting on Error Rate Plot

This example simulates a simple differential binary phase shift keying (DBPSK) communications system and plots error rate data for a series of E_b/N_0 values. It uses the `berfit` and `berconfint` functions to fit a curve to a set of empirical error rates.

Initialize Simulation Parameters

Specify the input signal message length, modulation order, range of E_b/N_0 values to simulate, and the minimum number of errors that must occur before the simulation computes an error rate for a given E_b/N_0 value. Preallocate variables for final results and interim results.

Typically, for statistically accurate error rate results, the minimum number of errors must be on the order of 100. This simulation uses a small number of errors to shorten the run time and to illustrate how curve fitting can smooth a set of results.

```
siglen = 100000; % Number of bits in each trial
M = 2; % DBPSK is binary
EbN0vec = 0:5; % Vector of EbN0 values
minnumerr = 5; % Compute BER after only 5 errors occur
numEbN0 = length(EbN0vec); % Number of EbN0 values

ber = zeros(1,numEbN0); % Final BER values
berVec = zeros(3,numEbN0); % Updated BER values
intv = cell(1,numEbN0); % Cell array of confidence intervals
```

Create an error rate calculator System object™.

```
errorCalc = comm.ErrorRate;
```

Loop the Simulation

Simulate the DBPSK-modulated communications system and compute the BER using a `for` loop to vary the E_b/N_0 value. The inner `while` loop ensures that a minimum number of bit errors occur for each E_b/N_0 value. Error rate statistics are saved for each E_b/N_0 value and used later in this example when curve fitting and plotting.

```
for jj = 1:numEbN0
    EbN0 = EbN0vec(jj);
    snr = EbN0; % For binary modulation SNR = EbN0
    reset(errorCalc)

    while (berVec(2,jj) < minnumerr)
        msg = randi([0,M-1],siglen,1); % Generate message sequence
        txsig = dpskmod(msg,M); % Modulate
        rxsig = awgn(txsig,snr,'measured'); % Add noise
        decodmsg = dpskdemod(rxsig,M); % Demodulate
        berVec(:,jj) = errorCalc(msg,decodmsg); % Calculate BER
    end
```

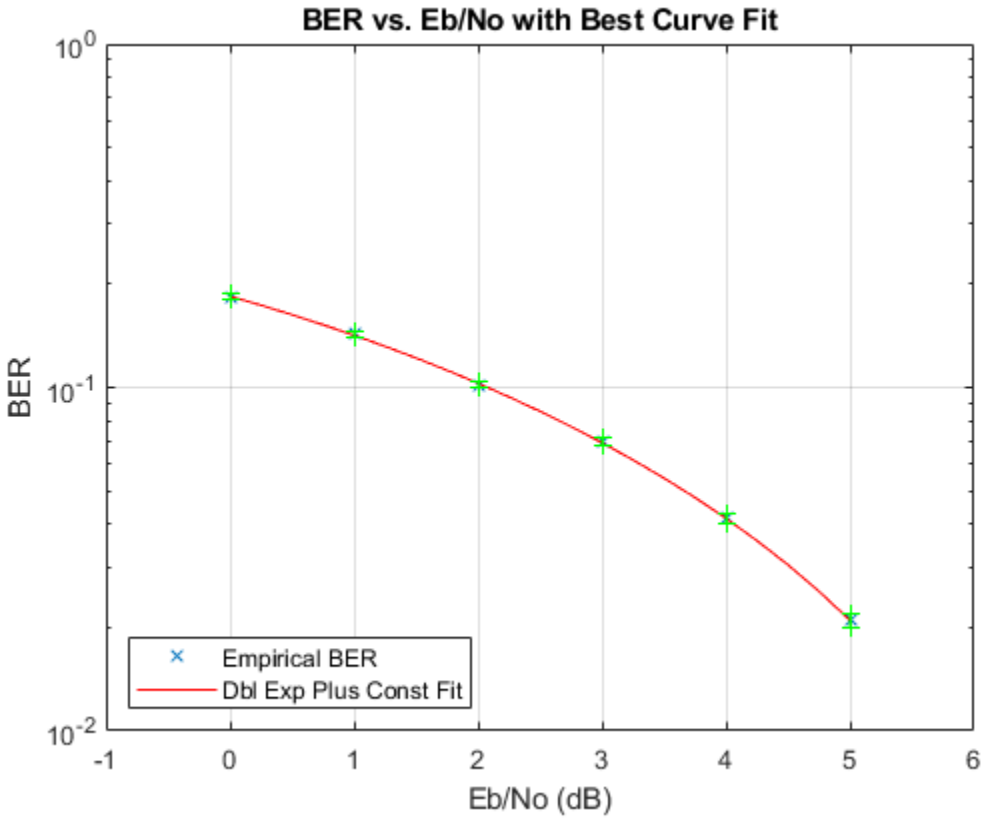
Use the `berconfint` function to compute the error rate at a 98% confidence interval for the E_b/N_0 values.

```
    [ber(jj),intv1] = berconfint(berVec(2,jj),berVec(3,jj),0.98);
    intv{jj} = intv1;
    disp(['EbN0 = ' num2str(EbN0) ' dB, ' num2str(berVec(2,jj)) ...
        ' errors, BER = ' num2str(ber(jj))])
end
```

```
EbN0 = 0 dB, 18392 errors, BER = 0.18392
EbN0 = 1 dB, 14307 errors, BER = 0.14307
EbN0 = 2 dB, 10190 errors, BER = 0.1019
EbN0 = 3 dB, 6940 errors, BER = 0.0694
EbN0 = 4 dB, 4151 errors, BER = 0.04151
EbN0 = 5 dB, 2098 errors, BER = 0.02098
```

Use the `berfit` function to plot the best fitted curve, interpolating between BER points to get a smooth plot. Add confidence intervals to the plot.

```
fitEbN0 = EbN0vec(1):0.25:EbN0vec(end); % Interpolation values
berfit(EbN0vec,ber,fitEbN0);
hold on;
for jj=1:numEbN0
    semilogy([EbN0vec(jj) EbN0vec(jj)],intv{jj},'g-+');
end
hold off;
```



See Also

Apps
Bit Error Rate Analysis

Functions
`berawgn` | `bercoding` | `berconfint` | `berfading` | `berfit` | `bersync`

Related Examples

- “Use Bit Error Rate Analysis App” on page 23-12
- “Analytical Expressions and Notations Used in BER Analysis” on page 23-45

Use Bit Error Rate Analysis App

The **Bit Error Rate Analysis** app calculates BER as a function of the energy per bit to noise power spectral density ratio (E_b/N_0) and enables you to analyze BER performance of communications systems.

Note The **Bit Error Rate Analysis** app is designed for analyzing BERs. For example, if your simulation computes a symbol error rate (SER), convert the SER to a BER before comparing the simulation results with theoretical results in the app.

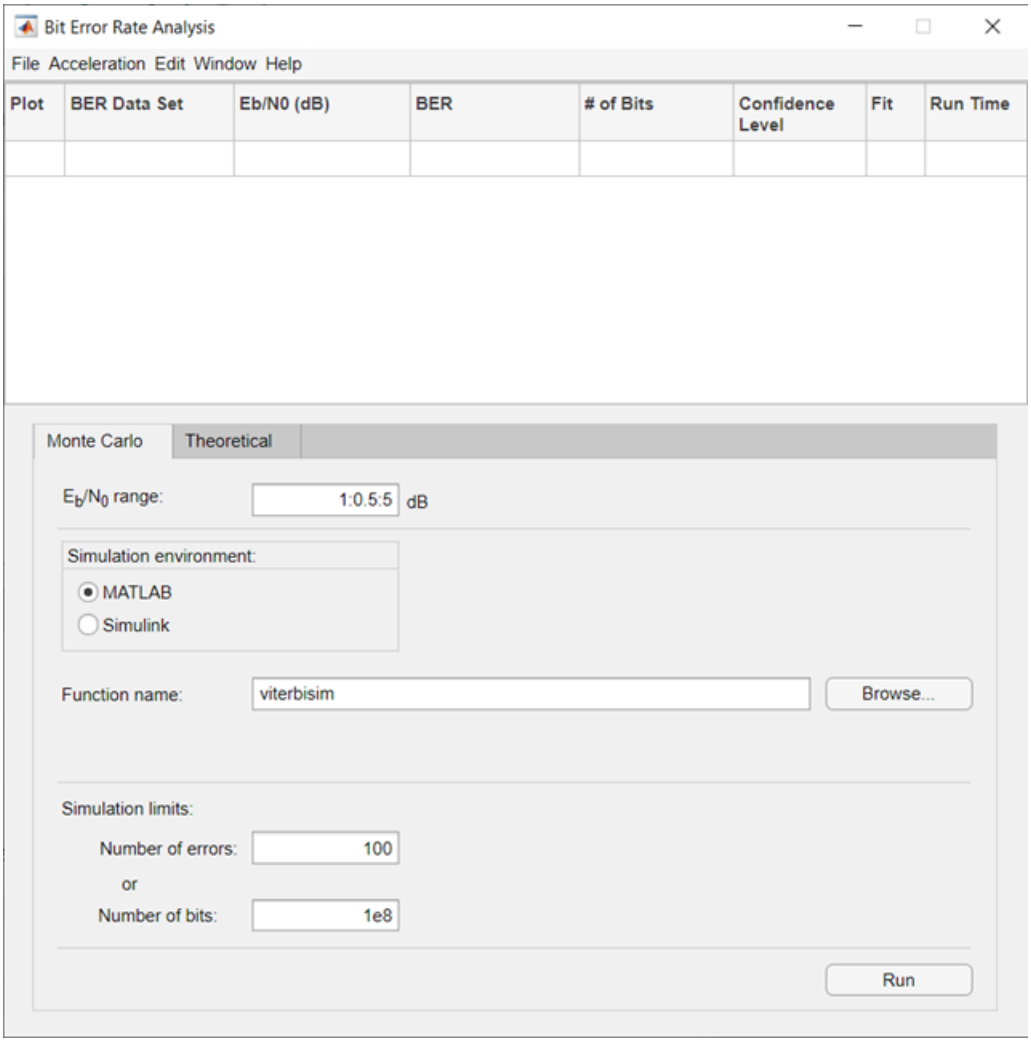
This topic describes the **Bit Error Rate Analysis** app and provides examples that show how to use the app.

In this section...
“Open Bit Error Rate Analysis App” on page 23-12
“Bit Error Rate Analysis App Environment” on page 23-13
“Compute Theoretical BERs Using Bit Error Analysis App” on page 23-15
“Run MATLAB Simulations in Monte Carlo Tab” on page 23-19
“Requirements for Using MATLAB Functions with Bit Error Rate Analysis App” on page 23-25
“Compute Error Rate Simulation Sweeps Using Bit Error Rate Analysis App” on page 23-28
“Run Simulink Simulations in Monte Carlo Tab” on page 23-33
“Requirements for Using Simulink Models with Bit Error Rate Analysis App” on page 23-38
“Manage BER Data” on page 23-39

Open Bit Error Rate Analysis App

You can open the **Bit Error Rate Analysis** app by using either of these options.

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click **Bit Error Rate Analysis**.
- MATLAB command prompt: Use the `bertool` function. If the app is already open, another instance of the app opens.



Bit Error Rate Analysis App Environment

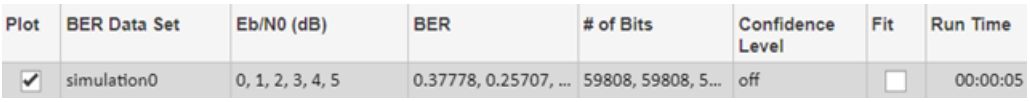
Components of Bit Error Rate Analysis App

The app consists of these three main components: an upper pane, a lower pane, and a separate BER Figure window.

- The upper pane of the app is a data set viewer. The data set viewer lists sets of BER data from the current app session along with high level settings and options for showing the data. By default, this data set viewer is empty.

  Sets of BER data, generated during the active **Bit Error Rate Analysis** app session or imported into the session, appear in the data viewer. This figure shows the `simulation0` BER data set

loaded in the data viewer pane.



Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	simulation0	0, 1, 2, 3, 4, 5	0.37778, 0.25707, ...	59808, 59808, 5...	off	☐	00:00:05

- The lower pane of the app has tabs labeled **Theoretical** and **Monte Carlo**. The tabs correspond to the different methods you can use to generate BER data with the app.

  Note For direct comparisons between theoretical results and simulation results generated when using the **Bit Error Rate Analysis** app, be sure that your MATLAB function or Simulink model simulation run from the **Monte Carlo** tab exactly matches the system defined by the parameters in the **Theoretical** tab.

  For more information, see the "Compute Theoretical BERs Using Bit Error Analysis App" on page 23-15, "Run MATLAB Simulations in Monte Carlo Tab" on page 23-19, and "Run Simulink Simulations in Monte Carlo Tab" on page 23-33 sections.

- A separate BER Figure window displays the BER data sets that have **Plot** selected in the data viewer. The BER Figure window does not open until the **Bit Error Rate Analysis** app has at least one data set to display.

Interaction Between Bit Error Rate Analysis App Components

The components of the app act as one integrated tool.

- If you select a data set in the data viewer, the app reconfigures the tabs to reflect the parameters associated with that data set and highlights the corresponding data in the BER Figure window. This feature is useful if the data viewer displays multiple data sets and if you want to recall the meaning and origin of each data set.
- If you select data plotted in the BER Figure window, the app reflects the parameters associated with that data in the app panes and highlights the corresponding data set in the data viewer.

  Note You cannot click a data point while the app is generating Monte Carlo simulation results. Before selecting data for more information, you must wait until the app generates all of the data points.

- If you configure the **Theoretical** tab in a way that is already reflected in an existing data set, the app highlights that data set in the data viewer. This feature prevents the app from duplicating its computations and entries in the data viewer but still enables the app to show results that you requested.
- If you close the BER Figure window, you can reopen the figure window by selecting **BER Figure** from the **Window** menu in the app.
- If you select options in the data viewer that affect the BER plot, the BER Figure window automatically reflects your selections. Such options relate to data set names, confidence intervals, curve fitting, and the presence or absence of specific data sets in the BER plot.

Note

- If you want to observe the addition of theoretical data to a plot with Monte Carlo simulation data displayed but do not yet have any data sets in the **Bit Error Rate Analysis** app, you can follow

the workflow described in the “Use Theoretical Tab in Bit Error Rate Analysis App” on page 23-16 section.

- If you save the BER Figure window using the **File** menu, the resulting file contains the contents of the window, but not the **Bit Error Rate Analysis** app data that led to the plot. To save an entire **Bit Error Rate Analysis** app session, see the “Save Bit Error Rate Analysis app Session” on page 23-43 section.

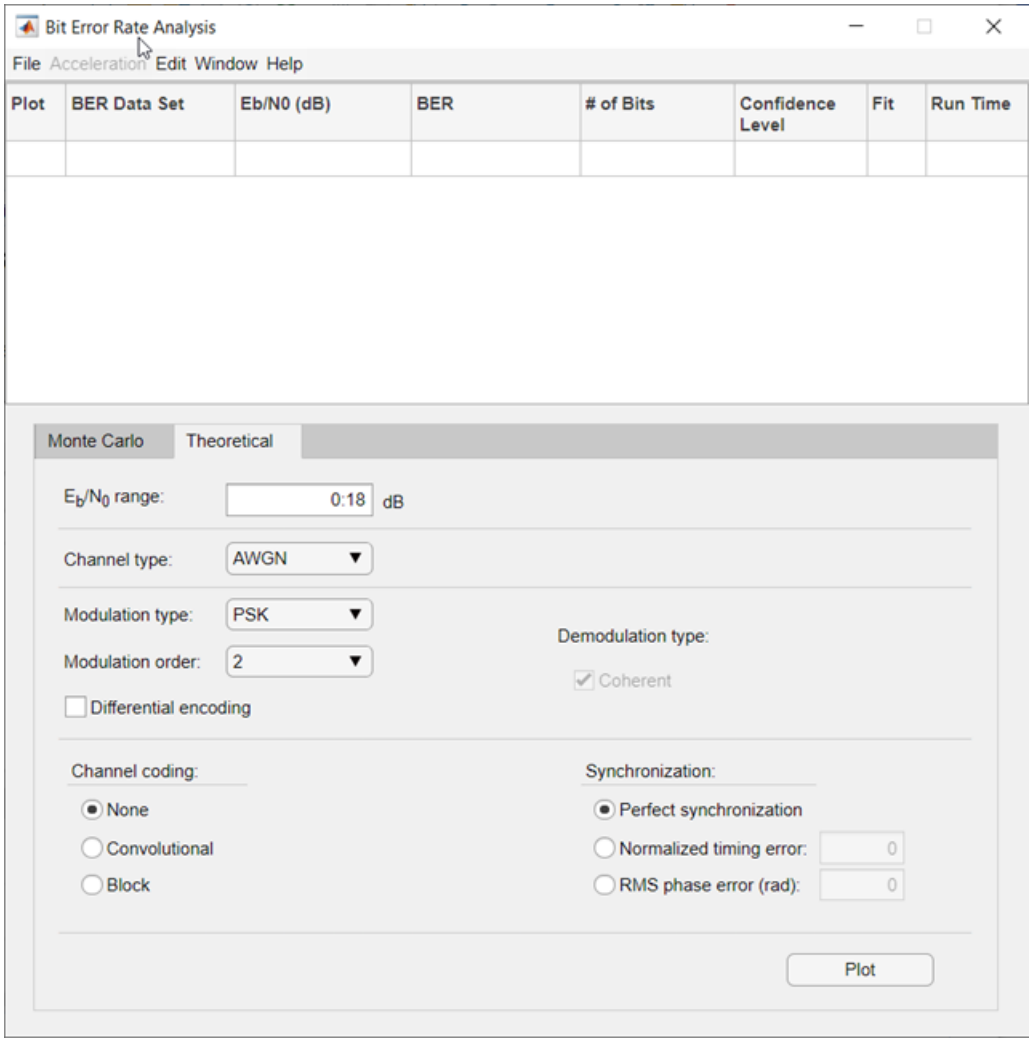
Compute Theoretical BERs Using Bit Error Analysis App

Section Overview

You can use the **Bit Error Rate Analysis** app to generate and analyze theoretical BER data. Theoretical data can be useful for comparison with your simulation results. However, closed-form BER expressions exist for only certain kinds of communications systems. For more information, see “Analytical Expressions and Notations Used in BER Analysis” on page 23-45.

To access app capabilities related to theoretical BER data, follow these steps.

1. Open the **Bit Error Rate Analysis** app, and select the **Theoretical** tab.



2 Set the parameters to reflect the communications system performance that you want to analyze.

3 Click **Plot**.

For an example that shows how to generate and analyze theoretical BER data using the **Bit Error Rate Analysis** app, see the “Use Theoretical Tab in Bit Error Rate Analysis App” on page 23-16 section.

For information about the combinations of parameters available on the **Theoretical** tab and the underlying functions that perform BER computations, see the “Available Sets of Theoretical BER Data” on page 23-18 section.

Use Theoretical Tab in Bit Error Rate Analysis App

This example shows how to use the app to generate and plot theoretical BER data. In particular, the example compares the performance of different modulation orders for QAM in a communications system that includes an AWGN channel.

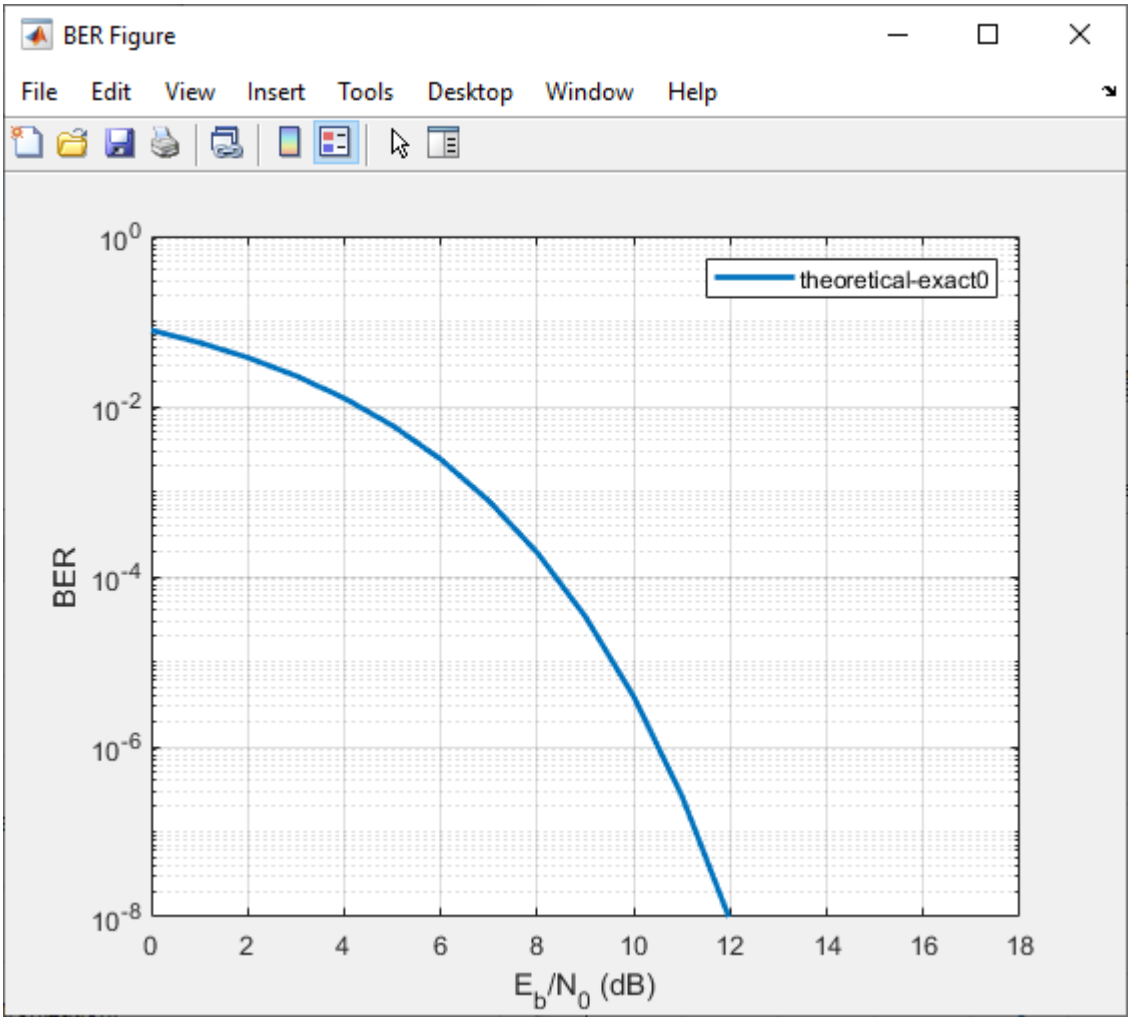
Run Theoretical BER Example

1 Open the **Bit Error Rate Analysis** app, and select the **Theoretical** tab.

2 Set these parameters to the values specified in this table.

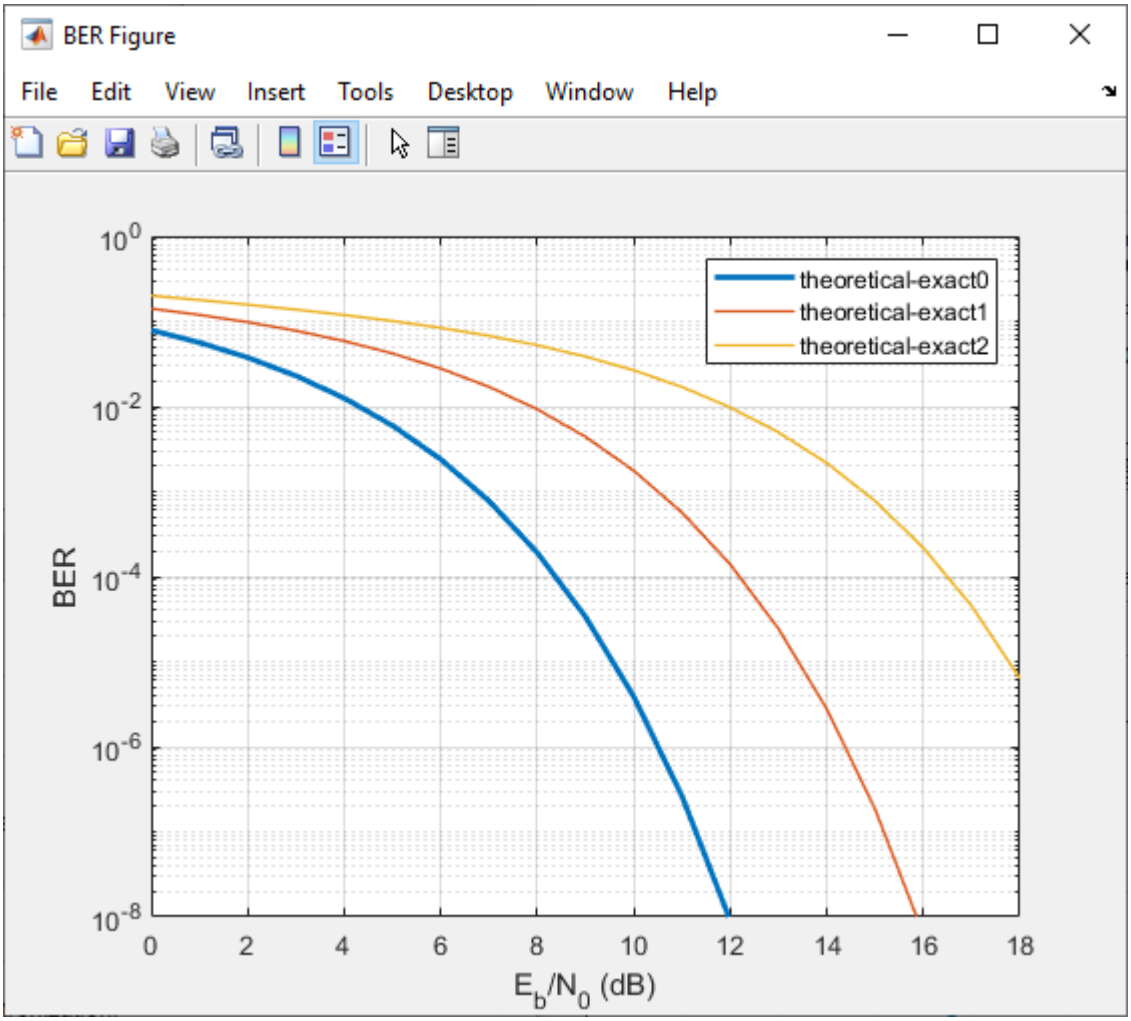
Parameter	Value
E_b / N_0 range	`0:18` (default)
Channel type	`AWGN` (default)
Modulation type	`QAM`
Modulation order	`4`

3 Click **Plot**. The app creates an entry in the data viewer and plots the data in the BER Figure window. Although the specified E_b/N_0 range is 0:18, the plot includes only BER values that exceed 10^{-8}.



4 Change the **Modulation order** parameter to `16`, and click **Plot**. The app creates another entry in the data viewer and plots the new data in the same BER Figure window (not pictured).

5 Change the **Modulation order** parameter to `64`, and click **Plot**. The app creates another entry in the data viewer and plots the new data in the same BER Figure window.



6 Click one of the curves to view the modulation order for that curve. The app responds to this action by adjusting the parameters in the **Theoretical** tab to reflect the values that correspond to that curve.

7 Remove the curve corresponding to 64-QAM from the plot (but not from the data viewer), by clearing **Plot** for the last entry in the data viewer. To restore the curve for 64-QAM to the plot, in the data viewer, select **Plot** for that curve.

Available Sets of Theoretical BER Data

The **Bit Error Rate Analysis** app can generate a large set of theoretical BERs. Parameters in the **Theoretical** tab enable you to configure the channel type, modulation type and order, error detection and correction channel coding, and synchronization error used when the app computes the theoretical BER. The app adjusts the combination of selectable parameter values based on your choices so that the configuration is always valid or uses a dialog box to inform you of valid parameter values.

The app computes the theoretical BER for these modulation types, assuming Gray ordered binary transmission data. The app uses these BER functions to perform underlying computations and limits the modulation order to practical limits.

- `berawgn` — For AWGN channel systems with no coding and perfect synchronization
- `berfading` — For fading channel systems with no coding and perfect synchronization
- `bercoding` — For systems with channel coding
- `bersync` — For systems with BPSK modulation, no coding, and imperfect synchronization
- `berconfint` — For error probability estimate and confidence interval of Monte Carlo simulation
- `berfit` — For fitting curves to nonsmooth empirical BER data

To compute the BER for higher modulation orders than permitted in the app, use the BER functions. For more information about specific combinations of parameters, see the reference pages for the BER functions listed in the **Bit Error Rate Calculation and Estimation** function group of the "Test and Measurement" category.

Run MATLAB Simulations in Monte Carlo Tab

Section Overview

Using the **Monte Carlo** tab with the **Simulation environment** parameter set to **MATLAB**, you can use the **Bit Error Rate Analysis** app in conjunction with your own MATLAB communications system simulation functions to generate and analyze BER data. The app calls the simulation specified by the **Function name** parameter for each specified E_b/N_0 value, collects the BER data from the simulation, and creates a plot. The app also enables you to adjust the E_b/N_0 range and the stopping criteria for the simulation.

To make your own simulation functions compatible with the app, see the "Prepare MATLAB Function for Use in Bit Error Rate Analysis App" example on the **Bit Error Rate Analysis** app reference page.

Use MATLAB Function with Bit Error Rate Analysis App

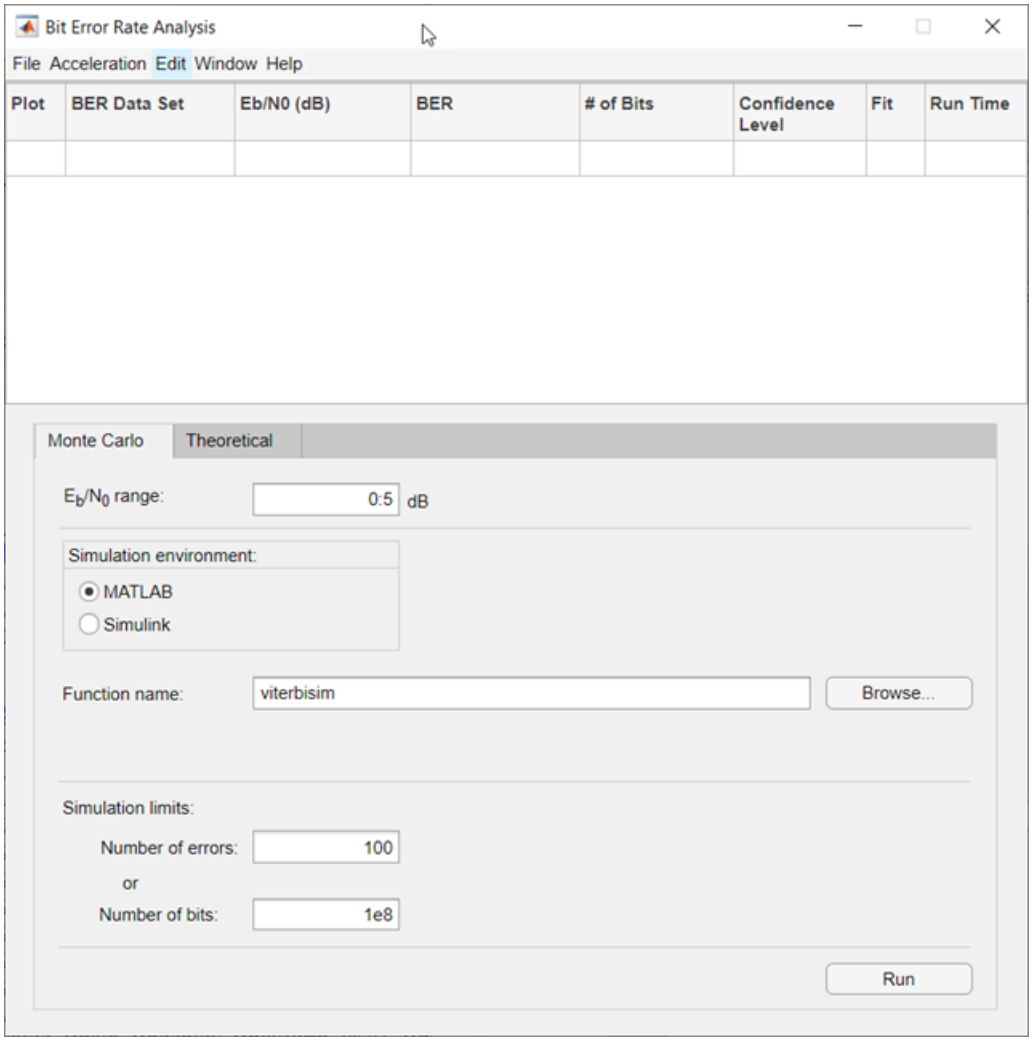
This example shows how the **Bit Error Rate Analysis** app can run the `viterbisim` MATLAB simulation.

To run this example, follow these steps.

1. Open the **Bit Error Rate Analysis** app, and select the **Monte Carlo** tab.
2. Set these parameters to the specified values shown in this table.

Parameter	Value
E_b / N_0 range	`0:5`
Simulation environment	**MATLAB** (default)
Function name	`viterbisim` (default)
Number of Errors	`100` (default)

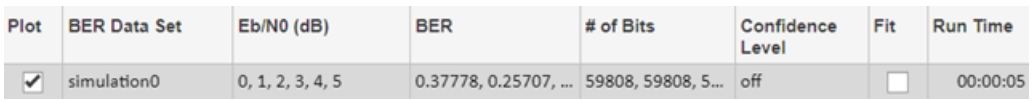
Parameter	Value
Number of bits	1e8 (default)



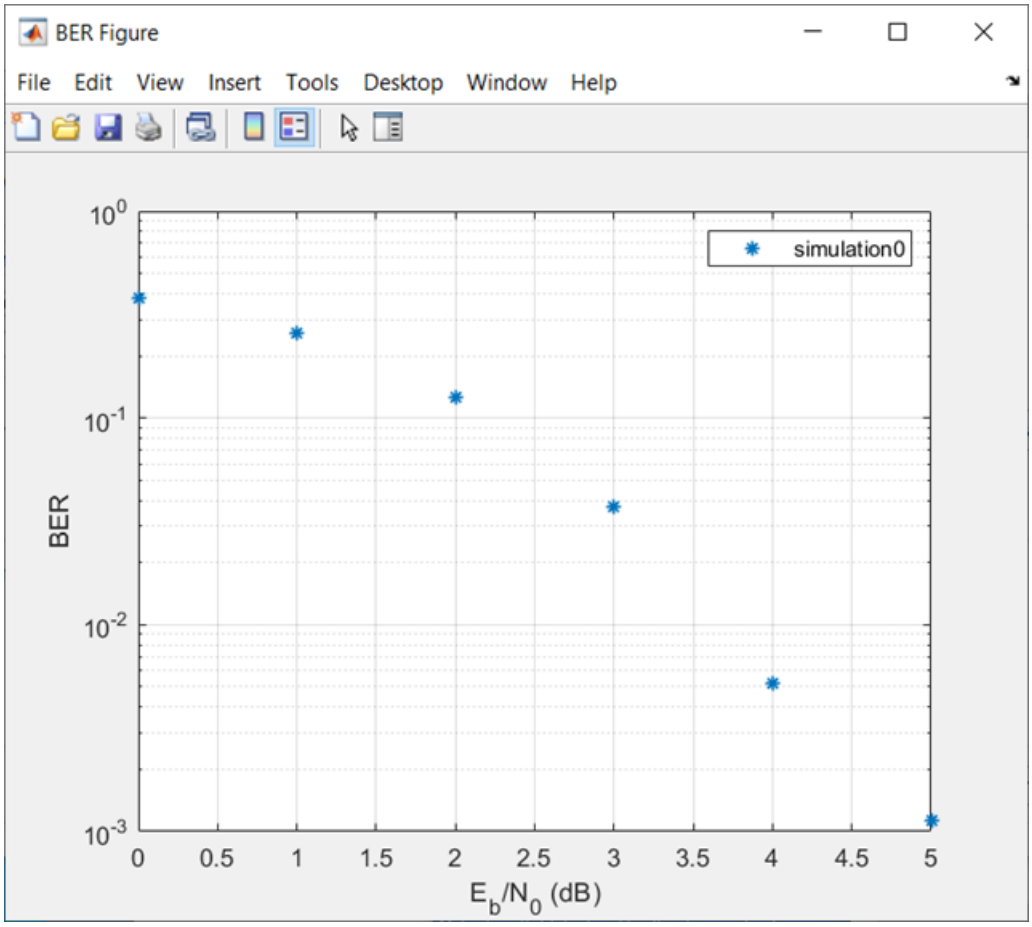
3 Click **Run**. The app runs the simulation function once for each specified E_b/N_0 value and gathers BER data.

Note While the **Bit Error Rate Analysis** app runs the configured simulation, it cannot process certain other tasks, including plotting data from the other tabs of the user interface. However, you can stop the simulation by clicking **Stop** in the Monte Carlo Simulation dialog box.

After computing the BER for each of the specified E_b/N_0 values, the app creates a listing in the data viewer.



The app also plots the data in the BER Figure window.

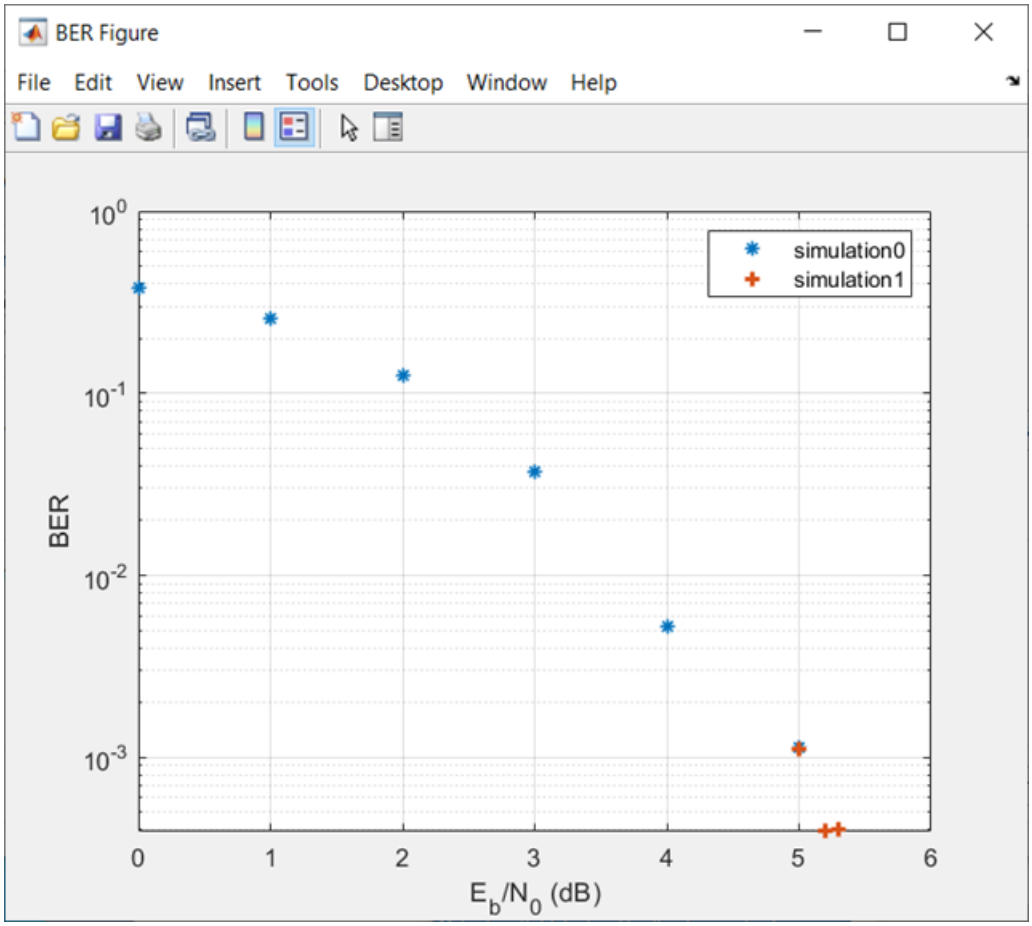


4 Adjust the **E_b/N_0 range** parameter to `[5 5.2 5.3]` and the **Number of bits** parameter to `1e5`. Click **Run** to produce a new set of results.

The app runs the simulation function using the new E_b/N_0 values and computes new BER data. The app then creates another listing in the data viewer.

Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	simulation0	0, 1, 2, 3, 4, 5	0.37778, 0.25707, ...	59808, 59808, 5...	off	☐	00:00:05
☑	simulation1	5, 5.2, 5.3	0.001117, 0.00039...	189808, 379808,...	off	N/A	00:00:00

The app also plots the new data set in the BER Figure window, adjusting the horizontal axis to accommodate the new E_b/N_0 values.

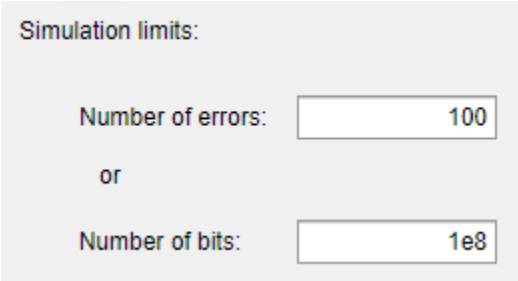


The BER values for the 5 dB E_b/N_0 setting differ between the two sets of data because the number of bits processed by the two simulations was different. If you want the computed BER to converge to a stable value, set the number of bits high enough to ensure that at least 100 bit errors occur. For more information about the criteria used by the **Bit Error Rate Analysis** app to terminate simulations, see the “Assign Function Stopping Criteria” on page 23-22 section.

Assign Function Stopping Criteria

When you create a MATLAB simulation function for use with the **Bit Error Rate Analysis** app, control the simulation run duration by setting the target number of errors and maximum number of bits. The simulation stops the current E_b/N_0 when either limit is reached. For more information about this requirement, see the “Requirements for Using MATLAB Functions with Bit Error Rate Analysis App” on page 23-25 section.

After you create your function, set the target number of errors and maximum number of bits on the **Monte Carlo** tab of the app.



Typically, a **Number of errors** parameter value of at least `100` produces an accurate error rate. The **Number of bits** value prevents the simulation from running too long. Depending on the E_b/N_0 value and other aspects of the communications system modeled (such as modulation characteristics and channel conditions), reaching 100 bit errors might not be realistic. However, if fewer than 100 errors occur because the **Number of bits** parameter value is too small, the returned error rate might be misleading. You can use confidence intervals to gauge the accuracy of the error rates that your simulation produces. As you increase the confidence level, the accuracy of the computed error rate decreases.

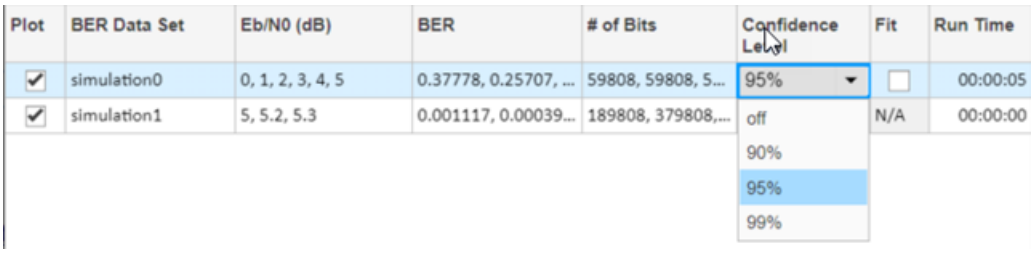
As an example, follow the procedure described in the "Use MATLAB Function with Bit Error Rate Analysis App" on page 23-19 section and set the **Confidence Level** parameter value to `95` for each of the two data sets. The confidence intervals for the second data set are larger than those for the first data set because the BER values associated with the second data set are based on only a small number of observed errors.

Note As long as your function is set up to detect and react to the **Stop** button in the **Bit Error Rate Analysis** app, you can use the button to prematurely stop a series of simulations. For more information, see "Assign Function Stopping Criteria" on page 23-22.

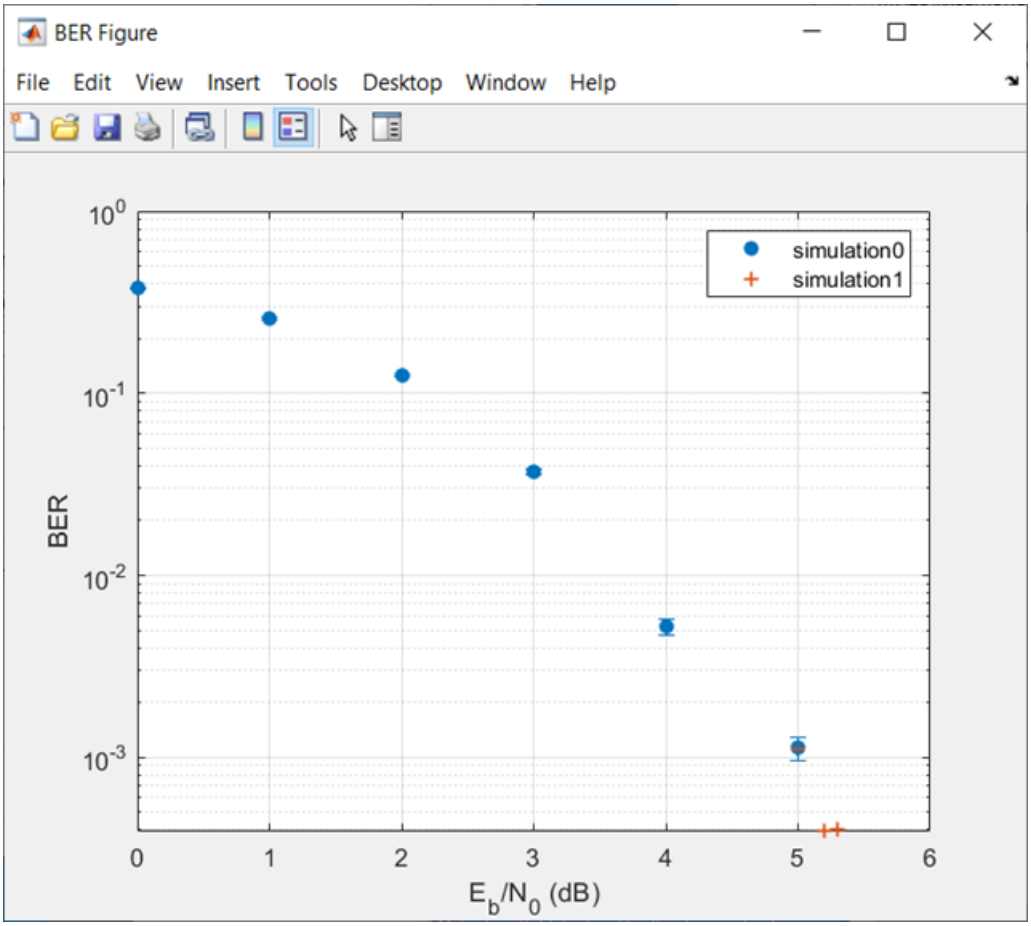
Plot Confidence Intervals

After you run a simulation with the **Bit Error Rate Analysis** app, the resulting data set in the data viewer has an active menu in the **Confidence Level** column. By default the **Confidence Level** value is `off`, meaning the simulation data in the BER Figure window does not show confidence intervals.

To show confidence intervals in the BER Figure window, set **Confidence Level** to `90%`, `95%`, or `99%`.



The plot in the BER Figure window automatically responds the **Confidence Level** value change. This figure shows a sample plot.

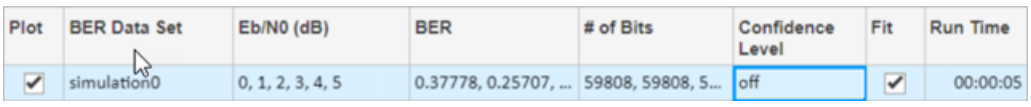


For an example that plots confidence intervals for a Simulink simulation, see the “Use Simulink Model with Bit Error Rate Analysis App” on page 23-34 section.

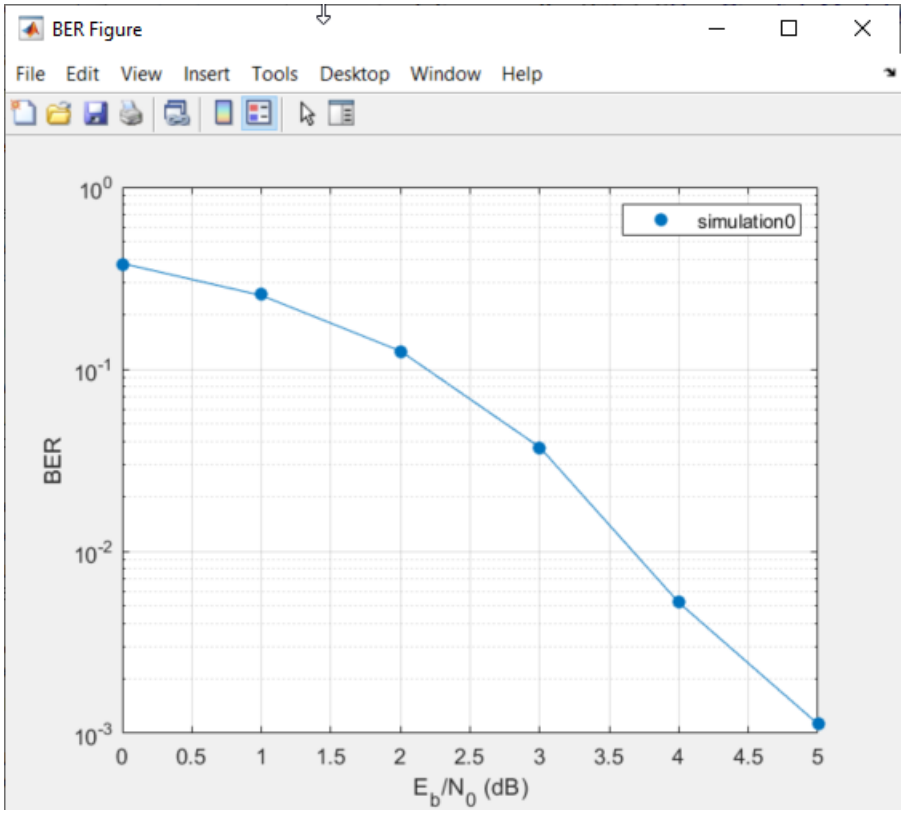
To find confidence intervals for levels not listed in the **Confidence Level** menu, use the `berconfint` function.

Curve Fit BER Points

After you run a simulation with the **Bit Error Rate Analysis** app, the BER Figure window plots individual BER data points. To fit a curve to a data set that contains at least four points, select **Fit** for that data in the data viewer.



The plot in the BER Figure window automatically responds to this selection. This plot shows a curve fit to a set of BER results.



For greater flexibility in the process of fitting a curve to BER data, use the `berfit` function.

Requirements for Using MATLAB Functions with Bit Error Rate Analysis App

When you create a MATLAB function for use with the **Bit Error Rate Analysis** app, ensure the function interacts properly with the user interface. This section describes the inputs, outputs, and basic operation of a function that is compatible with the app.

Input Arguments

The **Bit Error Rate Analysis** app evaluates your entries in fields of the user interface and passes data to the function as these input arguments (in sequential order).

1. One value from the **E_b/N_0 range** vector each time the **Bit Error Rate Analysis** app runs the simulation function
2. **Number of errors** value
3. **Number of bits** value

Output Arguments

Your simulation function must compute and return these output arguments (in sequential order). The **Bit Error Rate Analysis** app uses these output arguments when reporting and plotting results.

1. Bit error rate of the simulation

2 Number of bits processed when computing the BER

Simulation Function Operation

Your simulation function must perform these tasks:

- Simulate the communications system for the E_b/N_0 value specified in the first input argument.
- Stop simulating when the number of errors or number of processed bits equals or exceeds the corresponding threshold specified in the second or third input argument, respectively.
- Detect whether you click **Stop** in the **Bit Error Rate Analysis** app to stop the simulation in that case.

Template for Simulation Function

Use this template when adapting your code to work with the **Bit Error Rate Analysis** app. You can open the template in an editor by entering `edit bertooltemplate` at the MATLAB command prompt. If you develop a simulation function without using the template, be sure your function satisfies the requirements described in the “Requirements for Using MATLAB Functions with Bit Error Rate Analysis App” on page 23-25 section.

Note To use this template, you must insert your own simulation code in the places marked `INSERT YOUR CODE HERE`. For a complete example based on this template, see the “Prepare MATLAB Function for Use in Bit Error Rate Analysis App” example on the **Bit Error Rate Analysis** app reference page.

```
function [ber,numBits] = bertooltemplateTemp(EbNo,maxNumErrs,maxNumBits,varargin)
%BERTOOLTEMPLATE Template for a BERTool (Bit Error Rate Analysis app) simulation function.
%   This file is a template for a BERTool-compatible simulation function.
%   To use the template, insert your own code in the places marked "INSERT
%   YOUR CODE HERE" and save the result as a file on your MATLAB path. Then
%   use the Monte Carlo pane of BERTool to execute the script.
%
%   [BER, NUMBITS] = YOURFUNCTION(EBNO, MAXNUMERRS, MAXNUMBITS) simulates
%   the error rate performance of a communications system. EBNO is a vector
%   of Eb/No values, MAXNUMERRS is the maximum number of errors to collect
%   before stopping the simulation, and MAXNUMBITS is the maximum number of
%   bits to run before stopping the simulation. BER is the computed bit error
%   rate, and NUMBITS is the actual number of bits run. Simulation can be
%   interrupted only after an Eb/No point is simulated.
%
%   [BER, NUMBITS] = YOURFUNCTION(EBNO, MAXNUMERRS, MAXNUMBITS, BERTOOL)
%   also provides BERTOOL, which is the handle for the BERTool app and can
%   be used to check the app status to interrupt the simulation of an Eb/No
%   point.
%
%   For more information about this template and an example that uses it,
%   see the Communications Toolbox documentation.
%
%   See also BERTOOL and VITERBISIM.

% Copyright 2020 The MathWorks, Inc.

% Initialize variables related to exit criteria.
totErr  = 0; % Number of errors observed
numBits = 0; % Number of bits processed

% --- Set up the simulation parameters. ---
% --- INSERT YOUR CODE HERE.

% Simulate until either the number of errors exceeds maxNumErrs
% or the number of bits processed exceeds maxNumBits.
while((totErr < maxNumErrs) && (numBits < maxNumBits))

    % Check if the user clicked the Stop button of BERTool.
    if isBERToolSimulationStopped(varargin{:})
      break
```

```
    end

    % --- Proceed with the simulation.
    % --- Be sure to update totErr and numBits.
    % --- INSERT YOUR CODE HERE.

end % End of loop

% Compute the BER.
ber = totErr/numBits;
```

About Template for Simulation Function

The simulation function template either satisfies the requirements listed in the "Requirements for Using MATLAB Functions with Bit Error Rate Analysis App" on page 23-25 section or indicates where you need to insert code. In particular, the template:

- Has appropriate input and output arguments
- Includes a placeholder for code that simulates a system for the given E_b/N_0 value
- Uses a loop structure to stop simulating when either the number of errors exceeds `maxNumErrs` or the number of bits exceeds `maxNumBits`, whichever occurs first

  Note Although the `while` statement of the loop describes the exit criteria, your own code inserted into the section marked `Proceed with simulation` must compute the number of errors and the number of bits. If you do not perform these computations in your own code, clicking **Stop** in the Monte Carlo Simulation dialog box is the only way to terminate the loop.

- Detects when the user clicks **Stop** in the Monte Carlo Simulation dialog box in each iteration of the loop

Use Simulation Function Template

Follow these steps to update the simulation function template with your own simulation code.

1. Place the code for setup tasks in the template section marked `Set up parameters`. For example, initialize variables such as those containing the modulation alphabet size, filter coefficients, a convolutional coding trellis, or the states of a convolutional interleaver.
2. Place the code for these core simulation tasks in the template section marked `Proceed with simulation`. Determine the core simulation tasks, assuming that all setup work has already been performed. For example, core simulation tasks include filtering, error-control coding, modulation and demodulation, and channel modeling.
3. Also in the template section marked `Proceed with simulation`, include code that updates the values of the `totErr` and `numBits` variables. The `totErr` value represents the number of errors observed so far. The `numBits` value represents the number of bits processed so far. The computations to update these variables depend on how your core simulation tasks work.

   Note Updating the numbers of errors and bits is important for ensuring that the loop terminates.

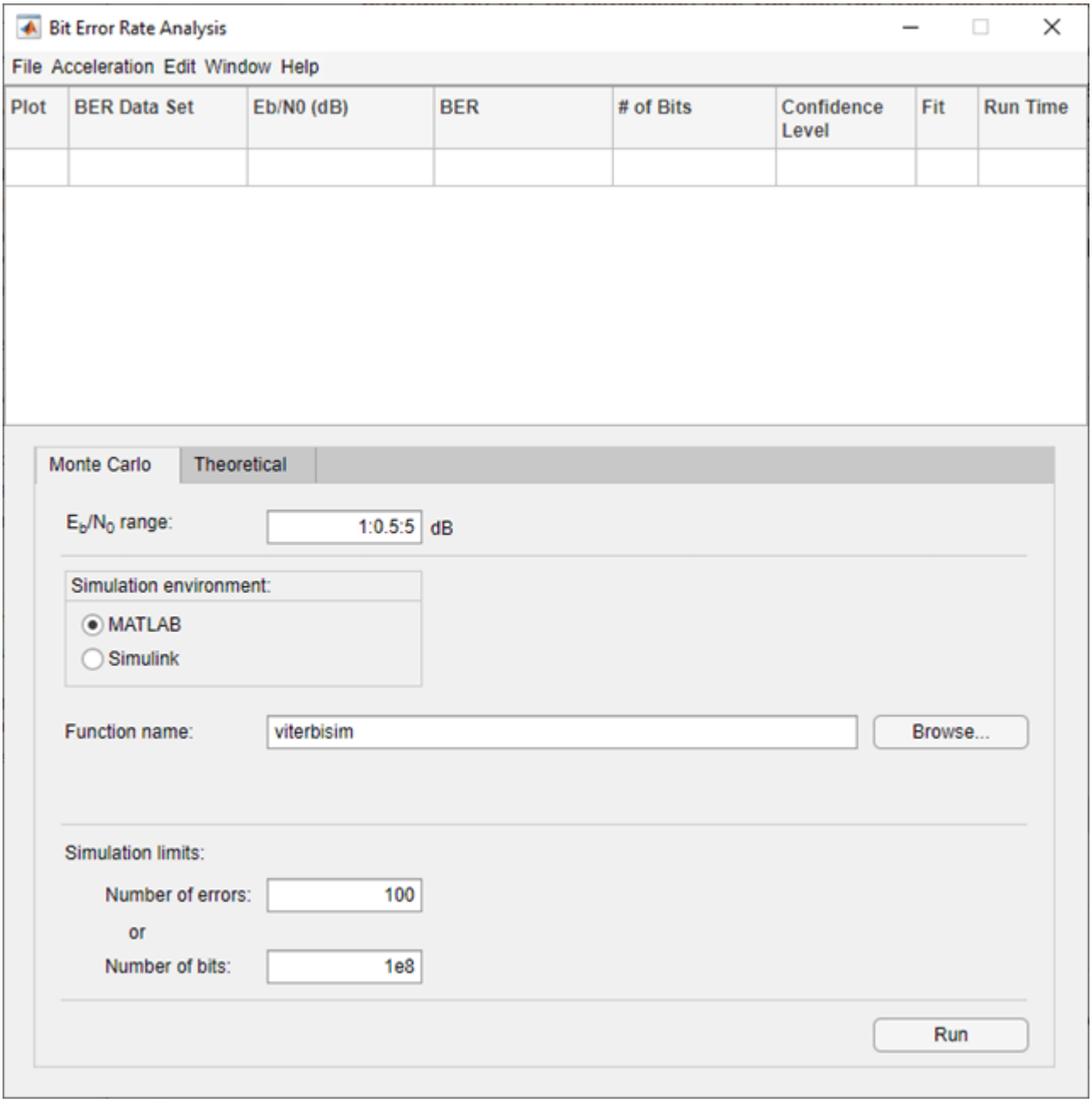
4. Omit from your simulation code any setup code that initializes `EbNo`, `maxNumErrs`, or `maxNumBits` variables, because the app passes these quantities to the function as input arguments after evaluating the data entered on the **Monte Carlo** tab.
5. Adjust your code or the code of the template as necessary to use consistent variable names and meanings. For example, if your original code uses a variable called `ebn0` and the function declaration (first line) for the template uses the variable name `EbNo`, you must change one of the

names so that they match. As another example, if your original code uses SNR instead of E_b/N_0 values, you must convert values appropriately.

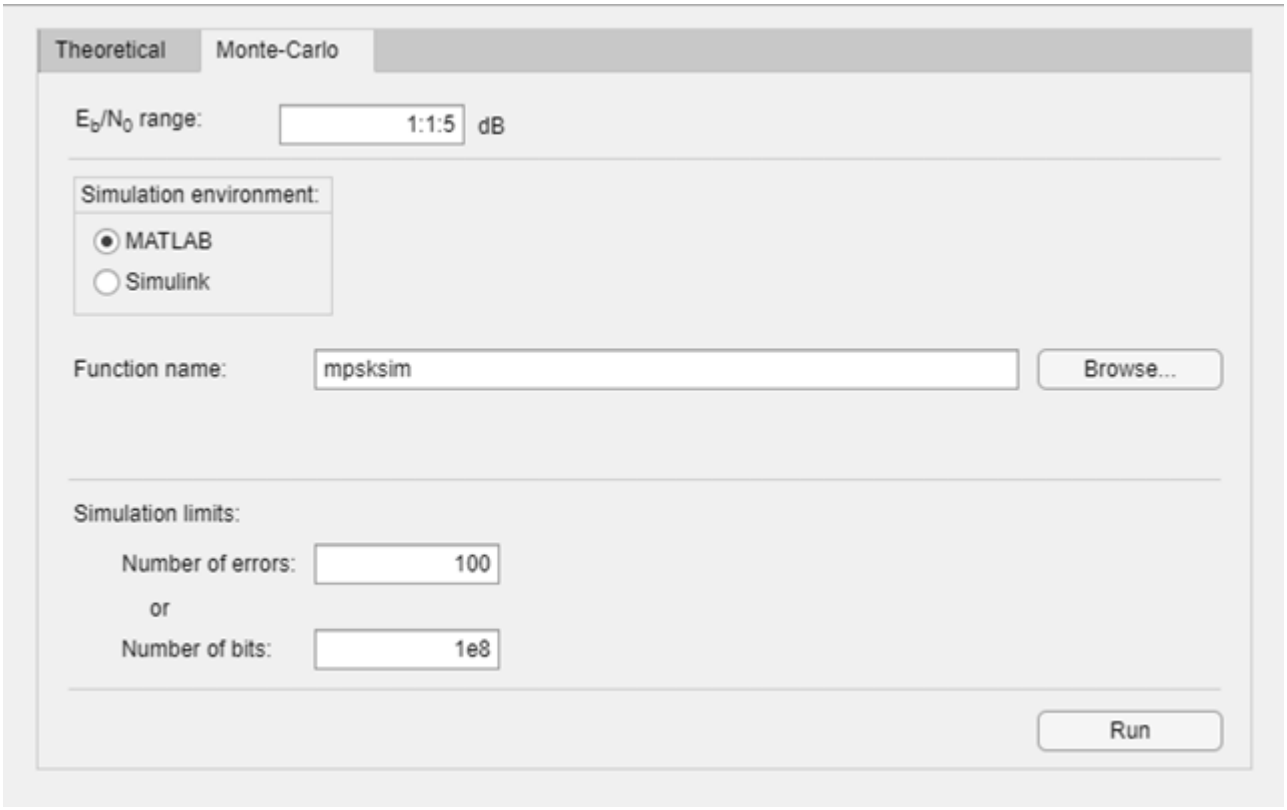
Compute Error Rate Simulation Sweeps Using Bit Error Rate Analysis App

Use the Bit Error Rate Analysis app to compute the BER as a function of E_b/N_0. The app analyzes performance with either Monte Carlo simulations of MATLAB® functions and Simulink® models or theoretical closed-form expressions for selected types of communications systems. The code in the mpsksim.m function provides an M-PSK simulation that you can run from the **Monte Carlo** tab of the app.

Open the **Bit Error Rate Analysis** app from the Apps tab or by running the `bertool` function in the MATLAB® command window.



On the **Monte Carlo** tab, set the E_b/N_0 **range** parameter to `1:1:5` and the **Function name** parameter to `mpsksim`.



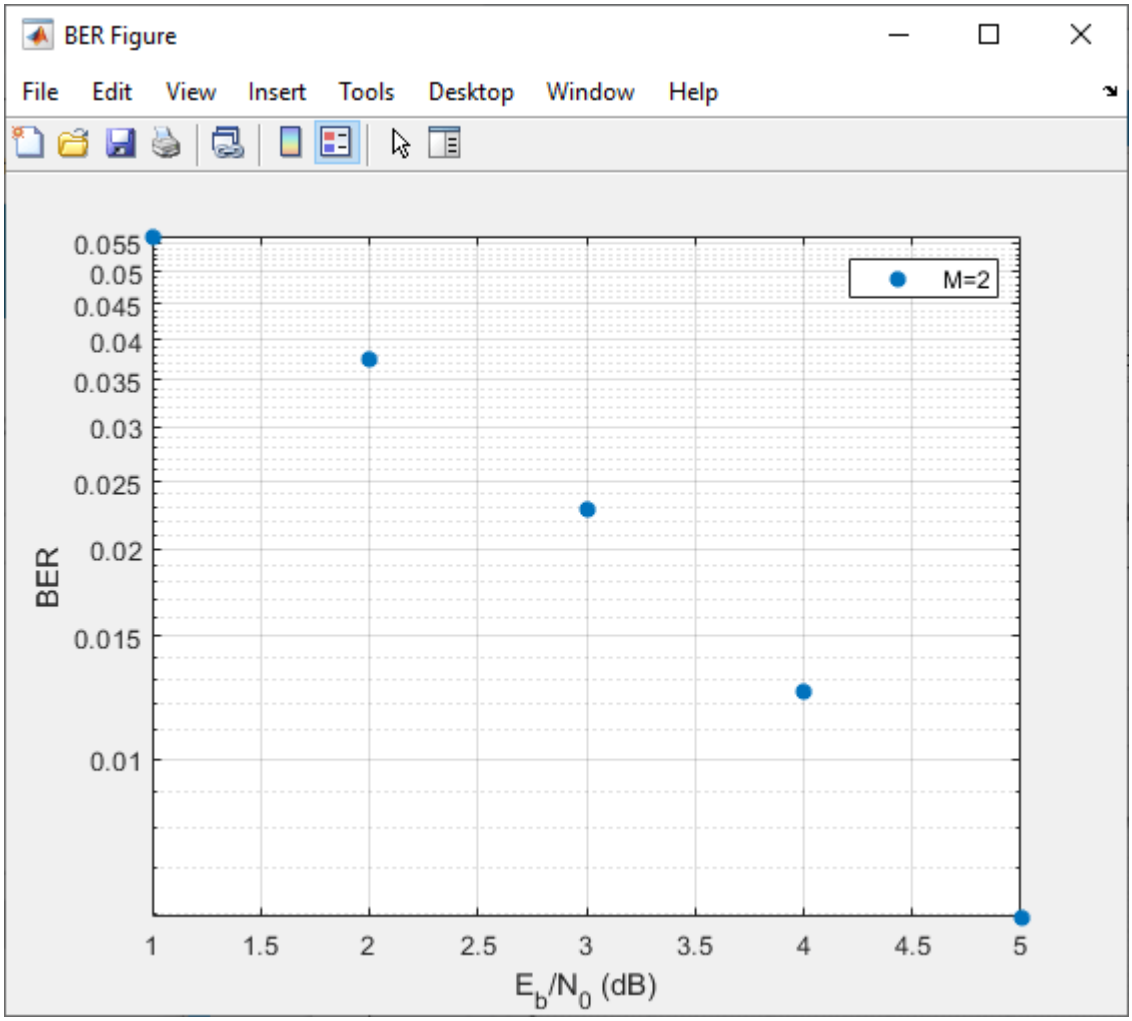
Open the `mpsksim` function for editing, set `M=2`, and save the changed file.

Run the `mpsksim.m` function as configured by clicking **Run** on the **Monte Carlo** tab in the app.

After the app simulates the set of E_b/N_0 points, update the name of the BER data set results by selecting `simulation0` in the **BER Data Set** field and typing `M=2` to rename the set of results. The legend on the BER figure updates the label to `M=2`.

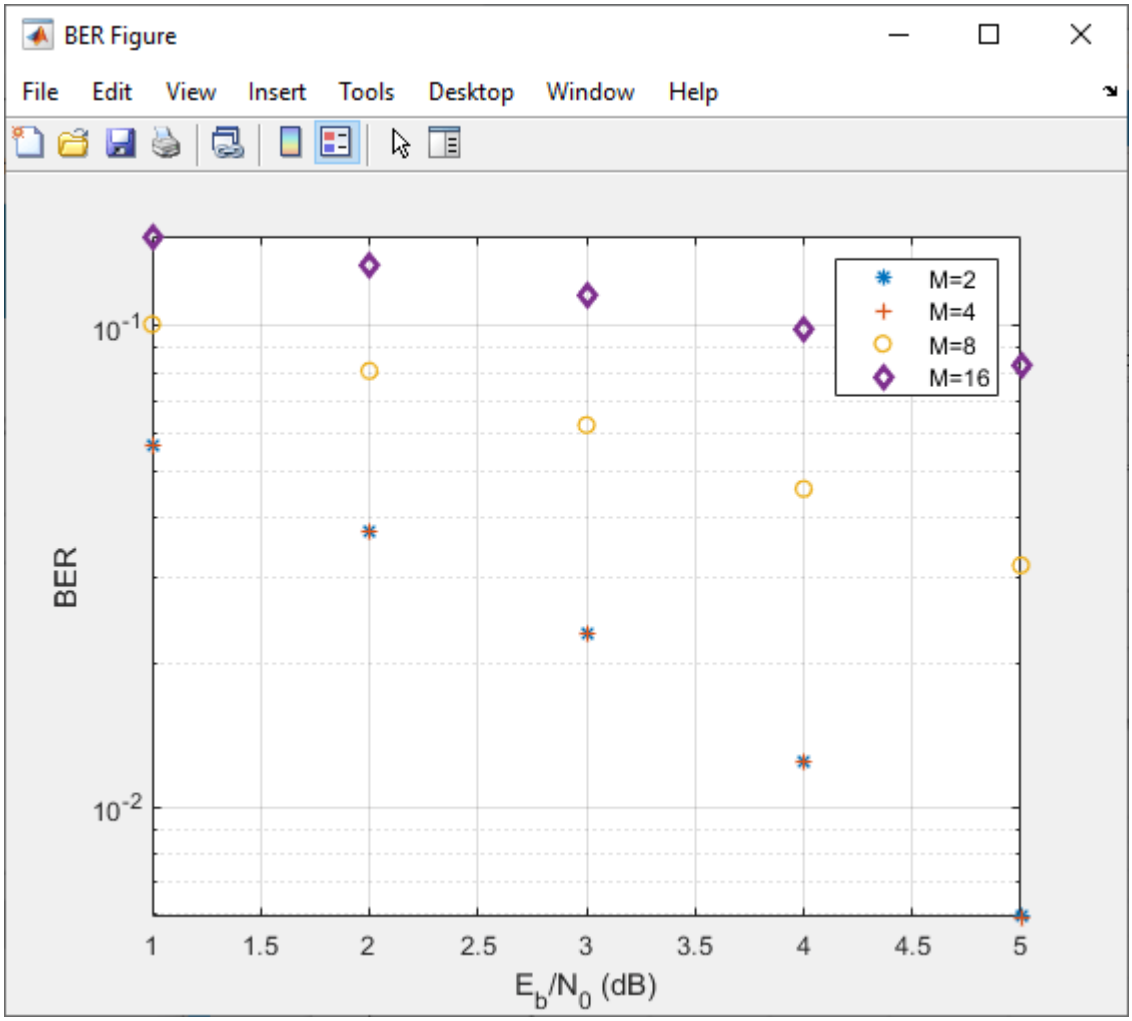
Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	simulation0	1, 2, 3, 4, 5	0.056284, 0.03751...	100002000, 100...	off	☐	00:00:44

Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	M=2	1, 2, 3, 4, 5	0.056284, 0.03751...	100002000, 100...	off	☐	00:00:44



Update the value for `M` in the `mpsksim` function, repeating this process for `M` = 4, 8, and 16. For example, these figures of the **Bit Error Rate Analysis** app and BER Figure window show results for varying `M` values.

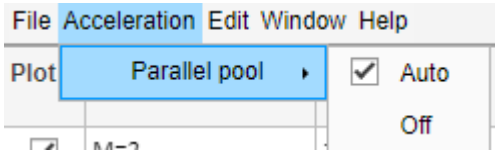
Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	M=2	1, 2, 3, 4, 5	0.056284, 0.03751...	100002000, 100...	off	☐	00:00:44
☑	M=4	1, 2, 3, 4, 5	0.056284, 0.03751...	100002000, 100...	off	☐	00:00:28
☑	M=8	1, 2, 3, 4, 5	0.10078, 0.080668...	100008000, 100...	off	☐	00:00:20
☑	M=16	1, 2, 3, 4, 5	0.15352, 0.13377, ...	100008000, 100...	off	☐	00:00:16



Parallel SNR Sweep Using Bit Error Rate Analysis App

The default configuration for the Monte Carlo processing of the **Bit Error Rate Analysis** app automatically uses parallel pool processing to process individual E_b/N_0 points when you have the Parallel Computing Toolbox™ software but for the processing of your simulation code:

- Any `parfor` function loops in your simulation code execute as standard `for` loops.
- Any `parfeval` (Parallel Computing Toolbox) function calls in your simulation code execute serially.
- Any `spmd` (Parallel Computing Toolbox) statement calls in your simulation code execute serially.



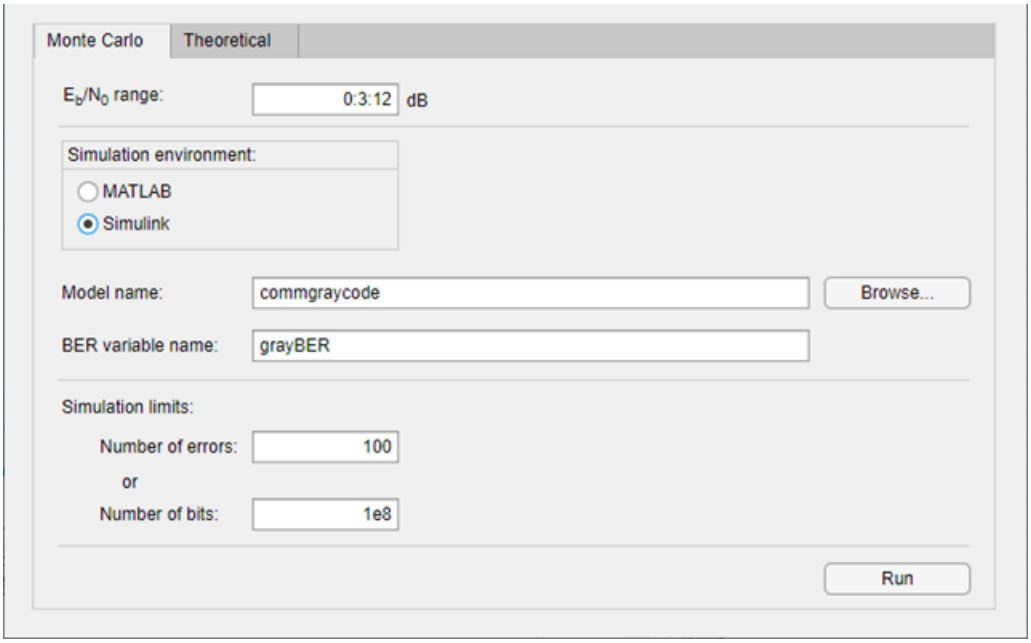
Run Simulink Simulations in Monte Carlo Tab

Section Overview

You can use the **Bit Error Rate Analysis** app in conjunction with Simulink models to generate and analyze BER data. The Simulink model simulates the performance of the communications system that you want to study, while the **Bit Error Rate Analysis** app manages a series of simulations using the model and collects the BER data.

Note To use Simulink models within the **Bit Error Rate Analysis** app, you must have the Simulink software.

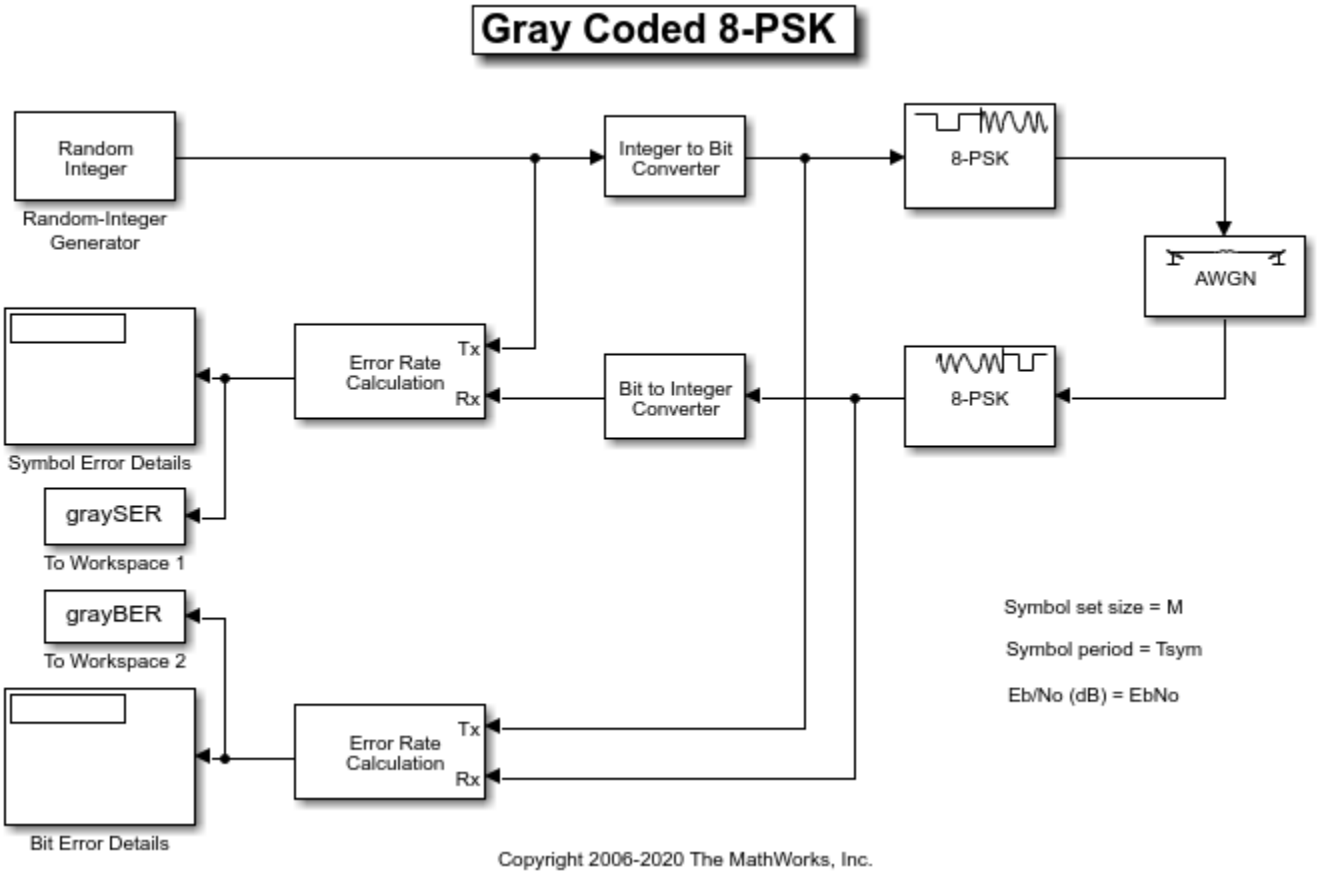
To access the capabilities of the **Bit Error Rate Analysis** app related to Simulink models, open the **Monte Carlo** tab, and then set the **Simulation environment** parameter to **Simulink**. If using parallel processing, the output must be saved to a workspace variable so that the parallel running engine can collect the results. For example, save the output of the Error Rate Calculation block to a workspace variable by using a Signal To Workspace block configured to save the output to the name specified for the **BER variable name**.



For details about confidence intervals and curve fitting for simulation data, see the “Plot Confidence Intervals” on page 23-23 and “Curve Fit BER Points” on page 23-24 sections, respectively.

Use Simulink Model with Bit Error Rate Analysis App

This example shows how the **Bit Error Rate Analysis** app can manage a series of simulations of a Simulink model and how you can vary the plot. This figure shows the `commgraycode` model.



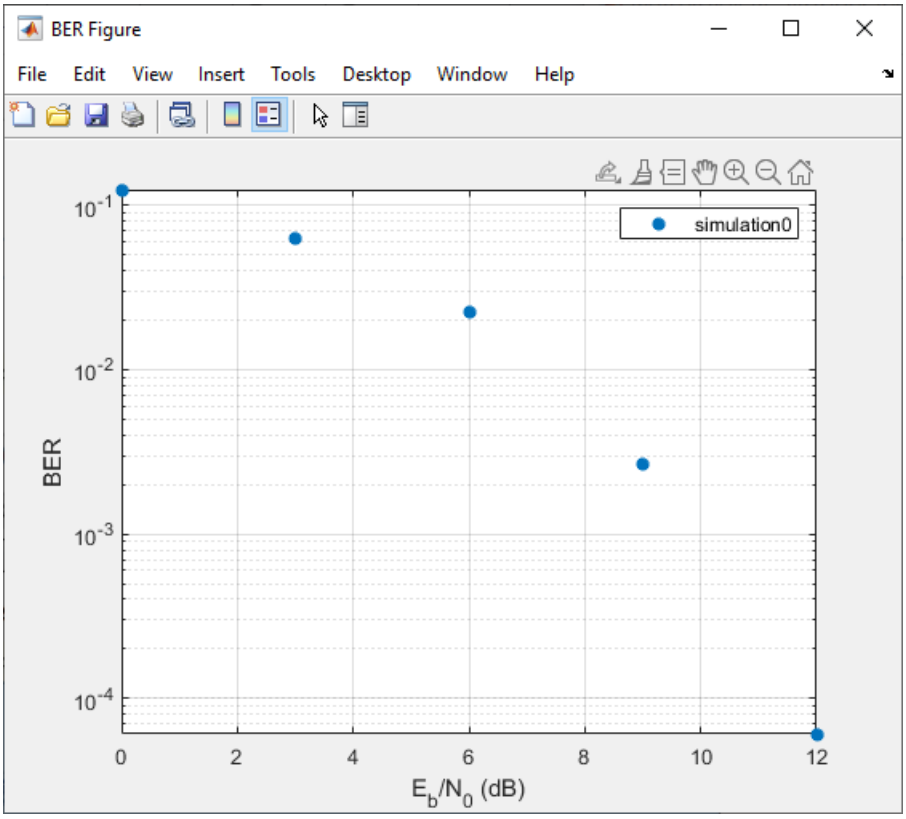
To run this example, follow these steps.

1. Open the **Bit Error Rate Analysis** app. On the **Monte Carlo** tab, enter the Simulink model name and a BER variable name. The default value for the **Model name** parameter is `commgraycode`. the default value for the **BER variable name** parameter is `grayBER`.
2. Click **Run**.

   The **Bit Error Rate Analysis** app loads the model into memory. The model initializes several variables in the MATLAB workspace. The app runs the simulation model once for each E_b/N_0 value, gathers the BER results, and creates a listing for the BER results in the data viewer.

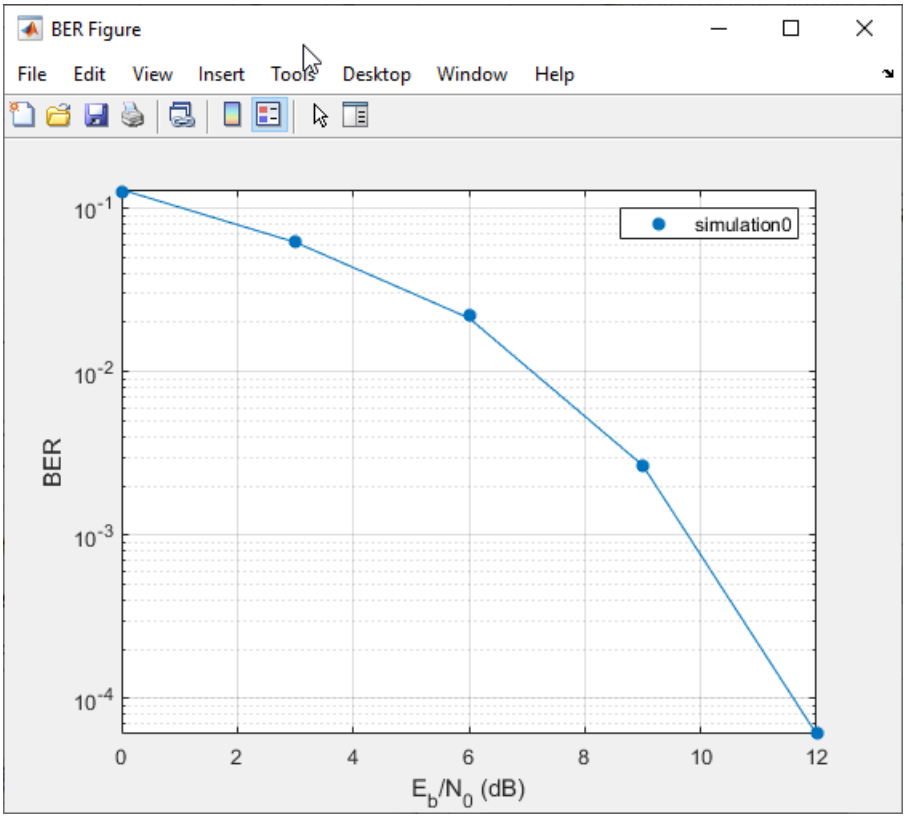
Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits	Confidence Level	Fit	Run Time
☑	simulation0	0, 3, 6, 9, 12	0.12444, 0.062222...	900, 1800, 4500,...	off	☐	00:01:08

   The **Bit Error Rate Analysis** app plots the data in the BER Figure window.



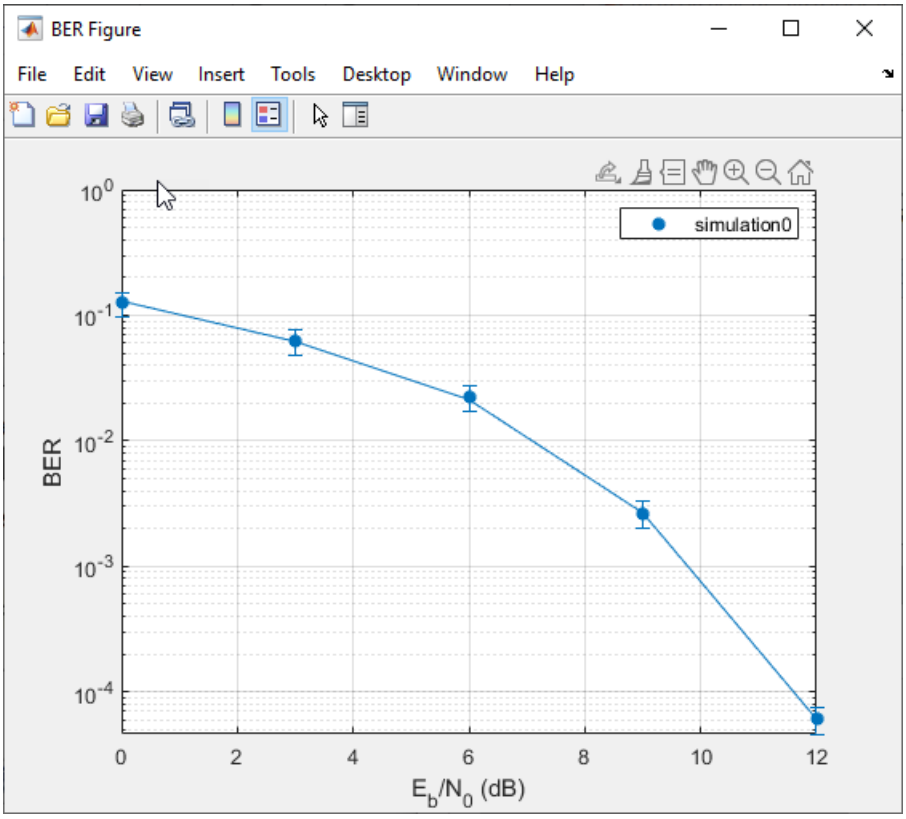
3 To fit a curve to the series of points in the BER Figure window, select **Fit** for the `simulation0` data in the data viewer.

The **Bit Error Rate Analysis** app plots the curve.



4 To indicate a 99% confidence interval around each point in the simulation data, set **Confidence Level** to 99% in the data viewer.

The **Bit Error Rate Analysis** app displays error bars to represent the confidence intervals.

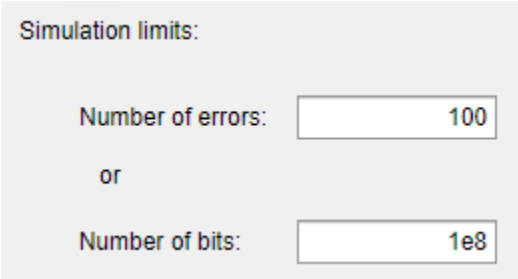


For another example that uses the **Bit Error Rate Analysis** app to manage a series of Simulink simulations, see the “Prepare Simulink Model for Use with Bit Error Rate Analysis App” example on the **Bit Error Rate Analysis** app reference page.

Assign Model Stopping Criteria

When you create a Simulink model for use with the **Bit Error Rate Analysis** app, you must set it up so that the simulation ends when it either detects a target number of errors or processes a maximum number of bits, whichever occurs first. For more information about this requirement, see the “Requirements for Using Simulink Models with Bit Error Rate Analysis App” on page 23-38 section.

After creating your Simulink model, set the target number of errors and the maximum number of bits on the **Monte Carlo** tab of the **Bit Error Rate Analysis** app.



Typically, a **Number of errors** parameter value of at least `100` produces an accurate error rate. The **Number of bits** value prevents the simulation from running too long, especially at large E_b/N_0

values. However, if the **Number of bits** value is so small that the simulation collects very few errors, the error rate might not be accurate. You can use confidence intervals to gauge the accuracy of the error rates that your simulation produces. Larger confidence intervals result in less accurate computed error rates.

You can also click **Stop** in the Monte Carlo Simulation dialog box to stop a series of simulations prematurely.

Requirements for Using Simulink Models with Bit Error Rate Analysis App

When you create a Simulink model for use with the **Bit Error Rate Analysis** app, ensure the model interacts properly with the user interface. This section describes the inputs, outputs, and basic operation of a model that is compatible with the app.

Input Variables

- The channel block must use the `EbNo` variable rather than a hard-coded value for E_b/N_0. For example, to model an AWGN channel, use the AWGN Channel block with the **Mode** parameter set to `Signal to noise ratio (Eb/No)` and the **Eb/No (dB)** parameter set to `EbNo`.
- The simulation must stop either when the error count reaches the value of the `maxNumErrs` variable or when the number of processed bits reaches the value of the `maxNumBits` variable, whichever occurs first. You can configure the Error Rate Calculation block in your model to use these criteria to stop the simulation.

Output Variables

- The simulation must send the final error rate data to the MATLAB workspace as a variable whose name you enter in the **BER variable name** parameter in the **Bit Error Rate Analysis** app. The output error statistics variable must be a three-element vector that lists the BER, the number of bit errors, and the number of processed bits.
- The three-element vector format for the output error statistics is supported by the Error Rate Calculation block.

Simulation Model Operation

- To avoid using an undefined variable name in blocks of the Simulink model, initialize these variables in the MATLAB workspace by using the preload callback function of the model or by assigning them at the MATLAB command prompt.

  ```
  EbNo = 0;
  maxNumErrs = 100;
  maxNumBits = 1e8;
  ```

  Tip Using the preload function callback of the model to initialize the runtime variables enables to you reopen the model in a future MATLAB session with runtime variables preconfigured to run in the app.

  The **Bit Error Rate Analysis** app provides the actual values based on values in the **Monte Carlo** tab, so the initial values in the model or workspace are not important.
- The app assumes the E_b/N_0 is used in the channel modeling. If your model uses the AWGN Channel block, and the **Mode** parameter is not set to `Signal to noise ratio (Eb/No)`, adapt

the block to use the E_b/N_0 mode instead. For more information, see the AWGN Channel block reference page.

- To compute the error rate, use the Error Rate Calculation block with these parameter settings: select **Stop simulation**, set **Target number of errors** to `maxNumErrs`, and set **Maximum number of symbols** to `maxNumBits`.
- If your model computes an SER instead of a BER, use the Integer to Bit Converter block to convert symbols to bits.
- To send data from the Error Rate Calculation block to the MATLAB workspace, set the **Output data** parameter to `Port`, attach a To Workspace block to the Error Rate Calculation block, and set the **Limit data points to last** parameter of the To Workspaceblock to `1`. The **Variable name** parameter in the To Workspace block must match the value you enter in the **BER variable name** parameter of the **Bit Error Rate Analysis** app.

  Tip More than one To Workspace block exists. Select the To Workspace block from the DSP System Toolbox / Sinks sublibrary.
- Frame-based simulations often run faster than sample-based simulations for the same number of bits processed. With a frame-based simulation, because the simulation processes a full frame of data each frame, the number of errors or number of processed bits might exceed the values you enter in the **Bit Error Rate Analysis** app.
- If your model uses a callback function to initialize variables in the MATLAB workspace upon loading the model, before you click **Run** in the **Bit Error Rate Analysis** app, make sure that one of these conditions is met:
  - The model is in memory (whether in a window or not), and the variables are intact.
  - The model is not currently in memory. In this case, the **Bit Error Rate Analysis** app loads the model into memory and runs the callback functions.
- If you clear or overwrite the variables set in the model, clear the model from memory by calling the `bdclose` function at the MATLAB command prompt.

  ```
  bdclose all
  ```

  When you click **Run** in the **Monte Carlo** tab, the app reloads the model.

Manage BER Data

Managing Data in Data Viewer

The data viewer gives you flexibility to rename and delete data sets and to reorder columns in the data viewer.

- To rename a data set in the data viewer, double-click its name in the **BER Data Set** column and type a new name.

Plot	BER Data Set	Eb/N0 (dB)	BER	# of Bits
☑	simulation0	0 1 2 3 4 5 6 7 8 9	0.0825 0.0595 0.0...	2000 2000
☑	theoretical-exact1	0 1 2 3 4 5 6 7 8 9	0.07865 0.056282 ...	N/A

- To delete a data set from the data viewer, select the data set, then select **Edit > Delete**.

Note If the data set originated from the **Theoretical** tab, the **Bit Error Rate Analysis** app deletes the data without asking for confirmation. You cannot undo this operation.

Export Bit Error Rate Analysis app Data Set

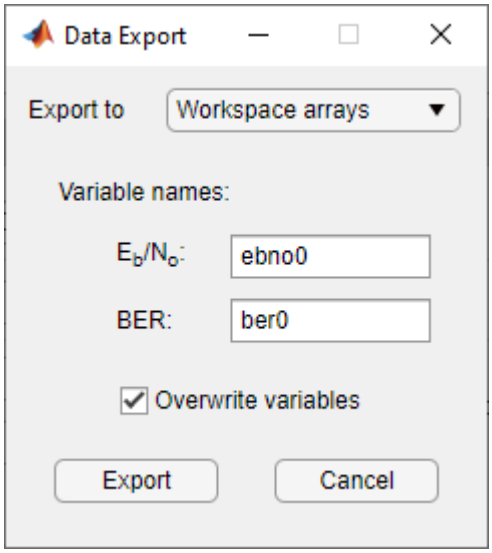
The **Bit Error Rate Analysis** app enables you to export individual data sets to the MATLAB workspace or to MAT-files. Exporting data enables you to process the data outside the **Bit Error Rate Analysis** app. For example, to create customized plots using data from the **Bit Error Rate Analysis** app, export the app data set to the MATLAB workspace and use any of the plotting commands in MATLAB. To reimport a structure later, see the “Import Bit Error Rate Analysis app Data Set” on page 23-43 section.

To export an individual data set, follow these steps.

1. In the data viewer, select the data set you want to export.
2. Select **File > Export Data**. Set **Export to** to indicate the format and destination of the data.

   - `Workspace arrays` — Export the selected data set to a pair of arrays in the MATLAB workspace. Use this option if you want to access the data in the MATLAB workspace (outside the app) and if you do not need to import the data into the **Bit Error Rate Analysis** app later.

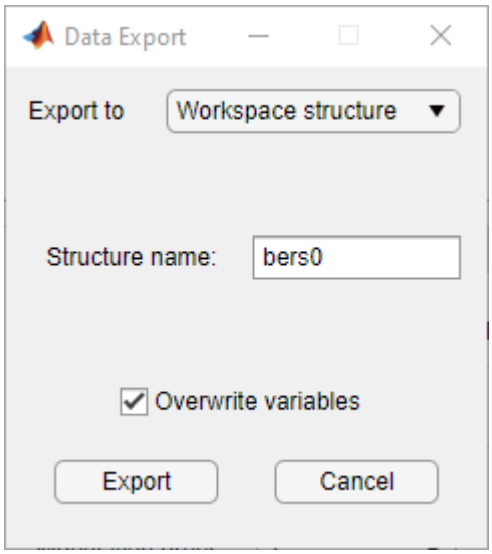
     Under **Variable names**, set **E_b/N_0** and **BER** parameters to specify the variable names for the E_b/N_0 values and BER values, respectively.



     If you want the **Bit Error Rate Analysis** app to use your chosen variable names even if variables by those names already exist in the workspace, select **Overwrite variables**.

- `Workspace structure` — Export the selected data set to a structure in the MATLAB workspace. If you export data using this option, you can import the data structure into the **Bit Error Rate Analysis** app later.

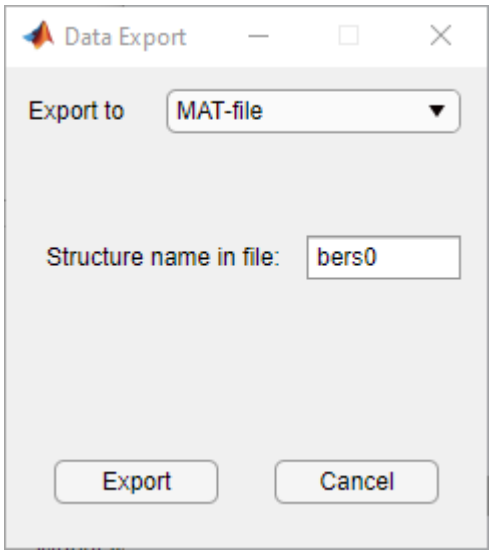
  Set the **Structure name** parameter to specify a workspace structure name.



  If you want the **Bit Error Rate Analysis** app to use your chosen variable name even if a variable with that name already exist in the workspace, select **Overwrite variables**.

- `MAT-file` — Export the selected data set to a structure in a MAT-file. If you export data using this option, you can import a MAT-file data structure into the **Bit Error Rate Analysis** app later.

  Set the **Structure name in file** parameter to specify a MAT-file name. The structure name in the file will also use this name.



3 Click **OK**. If you set **Export to** to `MAT-file`, the **Bit Error Rate Analysis** app prompts you for the path to the MAT-file that you want to create.

Examine an Exported Structure

This section describes the contents of the structure that the **Bit Error Rate Analysis** app exports to the workspace or to a MAT-file. This table describes the fields of the exported data structure. When you want to manipulate exported data, the fields that are most relevant are `paramsEvaled` and `data`.

Field	Description
`params`	The parameter values in the **Bit Error Rate Analysis** app, some of which might be invisible and hence irrelevant for computations
`paramsEvaled`	The parameter values evaluated and used by the **Bit Error Rate Analysis** app when computing the data set
`data`	The E_b/N_0, BER, and number of bits processed
`dataView`	Information about the appearance in the data viewer, which is used by the **Bit Error Rate Analysis** app when reimporting the data
`cellEditabilities`	Indication whether the data viewer has an active **Confidence Level** or **Fit** entry, which is used by the **Bit Error Rate Analysis** app when reimporting the data

Parameter Fields

The `params` and `paramsEvaled` fields are similar to each other, except that `params` describes the exact state of the user interface, whereas `paramsEvaled` indicates the values that are actually used for computations. For example, in a theoretical system with an AWGN channel, `params` records but `paramsEvaled` omits a diversity order parameter. The diversity order is not used in the computations because it is relevant for only systems with Rayleigh channels. As another example, if you type `[0:3]+1` in the user interface as the range of E_b/N_0 values, `params` indicates `[0:3]+1`, whereas `paramsEvaled` indicates `1 2 3 4`.

The length and exact contents of `paramsEvaled` depend on the data set because only relevant information appears. If the meaning of the contents of `paramsEvaled` is not clear upon inspection, one way to learn more is to reimport the data set into the **Bit Error Rate Analysis** app and inspect the parameter values that appear in the user interface.

Data Field

If your exported workspace variable is called `ber0`, the field `ber0.data` is a cell array that contains the numerical results in these vectors:

- `ber0.data{1}` lists the E_b/N_0 values.
- `ber0.data{2}` lists the BER values corresponding to each of the E_b/N_0 values.
- `ber0.data{3}` indicates, for simulation results, how many bits the **Bit Error Rate Analysis** app processed when computing each of the corresponding BER values.

Save Bit Error Rate Analysis app Session

The **Bit Error Rate Analysis** app enables you to save an entire session. This feature is useful if your session contains multiple data sets that you want to return to in a later session. To reimport a saved session, see the “Open Previous Bit Error Rate Analysis app Session” on page 23-44 section.

To save an entire **Bit Error Rate Analysis** app session, follow these steps.

1. Select **File > Save Session**.
2. When the **Bit Error Rate Analysis** app prompts you, enter the path to the file that you want to create.

The **Bit Error Rate Analysis** app saves the data in a MAT file or a binary file that records all data sets currently in the data viewer along with the user interface parameters associated with the data sets.

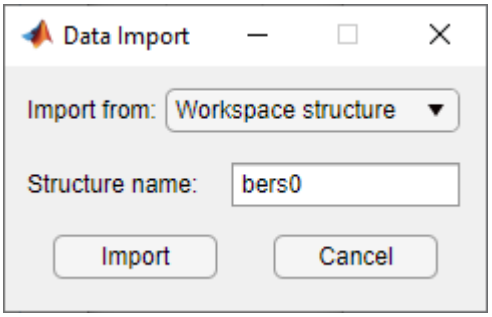
Note If your **Bit Error Rate Analysis** app session requires particular workspace variables, save those separately in a MAT-file using the `save` command in MATLAB.

Import Bit Error Rate Analysis app Data Set

The **Bit Error Rate Analysis** app enables you to reimport individual data sets that you previously exported to a structure. For more information about exporting data sets from the **Bit Error Rate Analysis** app, see the “Export Bit Error Rate Analysis app Data Set” on page 23-40 section.

To import an individual data set that you previously exported from the **Bit Error Rate Analysis** app to a structure, follow these steps.

1. Select **File > Import Data**.



2. Set the **Import from** parameter to either `Workspace structure` or `MAT-file`. If you select `Workspace structure`, type the name of the workspace variable in the **Structure name** parameter.
3. Click **OK**. If you set **Import from** to `MAT-file`, the **Bit Error Rate Analysis** app prompts you to select the file that contains the structure you want to import.

After you dismiss the Data Import dialog box (and the file selection dialog box, in the case of a MAT-file), the data viewer shows the newly imported data set and the BER Figure window the corresponding plot.

Open Previous Bit Error Rate Analysis app Session

The **Bit Error Rate Analysis** app enables you to open previous saved sessions. For more information about exporting data sets from the **Bit Error Rate Analysis** app, see the “Save Bit Error Rate Analysis app Session” on page 23-43 section.

To replace the data sets in the data viewer with data sets from a previous **Bit Error Rate Analysis** app session, follow these steps.

1. Select **File > Open Session**.

   Note If the **Bit Error Rate Analysis** app already contains data sets, your are asked whether you want to save the current session. If you answer no and continue with the loading process, the **Bit Error Rate Analysis** app discards the current session upon opening a new session from the file.

2. When the **Bit Error Rate Analysis** app prompts you, enter the path to the file you want to open. It must be a file that you previously created using the **Save Session** option in the **Bit Error Rate Analysis** app.

After the **Bit Error Rate Analysis** app reads the session file, the data viewer shows the data sets from the file.

If the **Bit Error Rate Analysis** app session requires particular workspace variables that you saved separately in a MAT-file, you can retrieve them by using the `load` function at the MATLAB command prompt. For example, to load the **Bit Error Rate Analysis** app session named `ber_analysis_filename.mat` enter this command.

```
load ber_analysis_filename.mat
```

See Also

Apps
Bit Error Rate Analysis

Functions
`berawgn` | `bercoding` | `berconfint` | `berfading` | `berfit` | `bersync`

Related Examples

- “Bit Error Rate Analysis Techniques” on page 23-2
- “Analytical Expressions and Notations Used in BER Analysis” on page 23-45

Analytical Expressions and Notations Used in BER Analysis

This topic covers the analytical expressions and notations for the theoretical analysis used in the BER functions (`berawgn`, `bercoding`, `berconfint`, `berfadingberfit`, `bersync`), **Bit Error Rate Analysis** app, and “Bit Error Rate Analysis Techniques” on page 23-2 topic.

Common Notation

This table defines the notations used in the analytical expressions in this topic.

Description	Notation
Size of modulation constellation	M
Number of bits per symbol	$k = \log_2 M$
Energy per bit-to-noise power-spectral-density ratio	$\frac{E_b}{N_0}$
Energy per symbol-to-noise power-spectral-density ratio	$\frac{E_s}{N_0} = k\frac{E_b}{N_0}$
Bit error rate (BER)	P_b
Symbol error rate (SER)	P_s
Real part	$\mathrm{Re}[\cdot]$
Floor, largest integer smaller than the value contained in braces	$\lfloor \cdot \rfloor$

This table describes the terms used for mathematical expressions in this topic.

Function	Mathematical Expression
Q function	$Q(x) = \frac{1}{\sqrt{2\pi}}\int_x^\infty \exp(-t^2/2)dt$
Marcum Q function	$Q(a,b) = \int_b^\infty t\exp\left(-\frac{t^2+a^2}{2}\right)I_0(at)dt$
Modified Bessel function of the first kind of order ν	$I_\nu(z) = \sum_{k=0}^{\infty}\frac{(z/2)^{\nu+2k}}{k!\Gamma(\nu+k+1)}$ where $\Gamma(x) = \int_0^\infty e^{-t}t^{x-1}dt$ is the gamma function.

Function	Mathematical Expression
Confluent hypergeometric function	$${}_1F_1(a,c;x) = \sum_{k=0}^{\infty} \frac{(a)_k}{(c)_k} \frac{x^k}{k!}$$ where the Pochhammer symbol, $(\lambda)_k$, is defined as $(\lambda)_0 = 1$, $(\lambda)_k = \lambda(\lambda+1)(\lambda+2)\cdots(\lambda+k-1)$.

This table defines the acronyms used in this topic.

Acronym	Definition
M-PSK	*M*-ary phase-shift keying
DE-M-PSK	Differentially encoded *M*-ary phase-shift keying
BPSK	Binary phase-shift keying
DE-BPSK	Differentially encoded binary phase-shift keying
QPSK	Quaternary phase-shift keying
DE-QPSK	Differentially encoded quadrature phase-shift keying
OQPSK	Offset quadrature phase-shift keying
DE-OQPSK	Differentially encoded offset quadrature phase-shift keying
M-DPSK	*M*-ary differential phase-shift keying
M-PAM	*M*-ary pulse amplitude modulation
M-QAM	*M*-ary quadrature amplitude modulation
M-FSK	*M*-ary frequency-shift keying
MSK	Minimum shift keying
M-CPFSK	*M*-ary continuous-phase frequency-shift keying

Analytical Expressions Used in berawgn Function and Bit Error Rate Analysis App

These sections cover the main analytical expressions used in the `berawgn` function and **Bit Error Rate Analysis** app.

M-PSK

From equation 8.22 in [2],

$$P_s = \frac{1}{\pi}\int_0^{(M-1)\pi/M} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[\pi/M]}{\sin^2\theta}\right)d\theta$$

This expression is similar, but not strictly equal, to the exact BER (from [4] and equation 8.29 from [2]):

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')P_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i,

$$P_i = \frac{1}{2\pi}\int_0^{\pi(1-(2i-1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[(2i-1)\pi/M]}{\sin^2\theta}\right)d\theta$$
$$-\frac{1}{2\pi}\int_0^{\pi(1-(2i+1)/M)} \exp\left(-\frac{kE_b}{N_0}\frac{\sin^2[(2i+1)\pi/M]}{\sin^2\theta}\right)d\theta$$

For M-PSK with $M = 2$, specifically BPSK, this equation 5.2-57 from [1] applies:

$$P_s = P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

For M-PSK with $M = 4$, specifically QPSK, these equations 5.2-59 and 5.2-62 from [1] apply:

$$P_s = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1 - \frac{1}{2}Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$
$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

DE-M-PSK

For DE-M-PSK with $M = 2$, specifically DE-BPSK, this equation 8.36 from [2] applies:

$$P_s = P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right) - 2Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

For DE-M-PSK with $M = 4$, specifically DE-QPSK, this equation 8.38 from [2] applies:

$$P_s = 4Q\left(\sqrt{\frac{2E_b}{N_0}}\right) - 8Q^2\left(\sqrt{\frac{2E_b}{N_0}}\right) + 8Q^3\left(\sqrt{\frac{2E_b}{N_0}}\right) - 4Q^4\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

From equation 5 in [3],

$$P_b = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\left[1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right]$$

OQPSK

For OQPSK, use the same BER and SER computations as for QPSK in [2].

DE-OQPSK

For OQPSK, use the same BER and SER computations as for DE-QPSK in [3].

M-DPSK

For M-DPSK, this equation 8.84 from [2] applies:

$$P_s = \frac{\sin(\pi/M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{\exp(-(kE_b/N_0)(1-\cos(\pi/M)\cos\theta))}{1-\cos(\pi/M)\cos\theta}d\theta$$

This expression is similar, but not strictly equal, to the exact BER (from [4]):

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')A_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i,

$$A_i = F\left((2i+1)\frac{\pi}{M}\right) - F\left((2i-1)\frac{\pi}{M}\right)$$

$$F(\psi) = -\frac{\sin\psi}{4\pi}\int_{-\pi/2}^{\pi/2}\frac{\exp(-kE_b/N_0(1-\cos\psi\cos t))}{1-\cos\psi\cos t}dt$$

For M-DPSK with $M = 2$, this equation 8.85 from [2] applies:

$$P_b = \frac{1}{2}\exp\left(-\frac{E_b}{N_0}\right)$$

M-PAM

From equations 8.3 and 8.7 in [2] and equation 5.2-46 in [1],

$$P_s = 2\left(\frac{M-1}{M}\right)Q\left(\sqrt{\frac{6}{M^2-1}\frac{kE_b}{N_0}}\right)$$

From [5],

$$P_b = \frac{2}{M\log_2 M} \times$$
$$\sum_{k=1}^{\log_2 M} \sum_{i=0}^{(1-2^{-k})M-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{M} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{M} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1) \sqrt{\frac{6\log_2 M}{M^2-1} \frac{E_b}{N_0}} \right) \right\}$$

M-QAM

For square M-QAM, $k = \log_2 M$ is even, so equation 8.10 from [2] and equations 5.2-78 and 5.2-79 from [1] apply:

$$P_s = 4\frac{\sqrt{M}-1}{\sqrt{M}} Q\left(\sqrt{\frac{3}{M-1} \frac{kE_b}{N_0}} \right) - 4\left(\frac{\sqrt{M}-1}{\sqrt{M}} \right)^2 Q^2\left(\sqrt{\frac{3}{M-1} \frac{kE_b}{N_0}} \right)$$

From [5],

$$P_b = \frac{2}{\sqrt{M}\log_2 \sqrt{M}}$$
$$\times \sum_{k=1}^{\log_2 \sqrt{M}} \sum_{i=0}^{(1-2^{-k})\sqrt{M}-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{\sqrt{M}} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1) \sqrt{\frac{6\log_2 M}{2(M-1)} \frac{E_b}{N_0}} \right) \right\}$$

For rectangular (non-square) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$, $I = 2^{\frac{k-1}{2}}$, and $J = 2^{\frac{k+1}{2}}$. So that,

$$P_s = \frac{4IJ - 2I - 2J}{M}$$
$$\times Q\left(\sqrt{\frac{6\log_2(IJ)}{(I^2+J^2-2)} \frac{E_b}{N_0}} \right) - \frac{4}{M}(1 + IJ - I - J) Q^2\left(\sqrt{\frac{6\log_2(IJ)}{(I^2+J^2-2)} \frac{E_b}{N_0}} \right)$$

From [5],

$$P_b = \frac{1}{\log_2(IJ)} \left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l) \right)$$

where

$$P_I(k) = \frac{2}{I} \sum_{i=0}^{(1-2^{-k})I-1} \left\{ (-1)^{\left\lfloor \frac{i2^{k-1}}{I} \right\rfloor} \left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{I} + \frac{1}{2} \right\rfloor \right) Q\left((2i+1) \sqrt{\frac{6\log_2(IJ)}{I^2+J^2-2} \frac{E_b}{N_0}} \right) \right\}$$

and

$$P_J(k) = \frac{2}{J} \sum_{j=0}^{(1-2^{-l})J-1} \left\{ (-1)^{\left\lfloor \frac{j2^{l-1}}{J} \right\rfloor} \left(2^{l-1} - \left\lfloor \frac{j2^{l-1}}{J} + \frac{1}{2} \right\rfloor \right) Q\left((2j+1) \sqrt{\frac{6\log_2(IJ)}{I^2+J^2-2} \frac{E_b}{N_0}} \right) \right\}$$

Orthogonal M-FSK with Coherent Detection

From equation 8.40 in [2] and equation 5.2-21 in [1],

$$P_s = 1 - \int_{-\infty}^{\infty} \left[Q\left(-q - \sqrt{\frac{2kE_b}{N_0}} \right) \right]^{M-1} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{q^2}{2} \right) dq$$

$$P_b = \frac{2^{k-1}}{2^k - 1} P_s$$

Nonorthogonal 2-FSK with Coherent Detection

For $M = 2$, equation 5.2-21 in [1] and equation 8.44 in [2] apply:

$$P_s = P_b = Q\left(\sqrt{\frac{E_b(1 - \mathrm{Re}[\rho])}{N_0}} \right)$$

ρ is the complex correlation coefficient, such that:

$$\rho = \frac{1}{2E_b} \int_0^{T_b} \tilde{s}_1(t) \tilde{s}_2^*(t) dt$$

where $\tilde{s}_1(t)$ and $\tilde{s}_2(t)$ are complex lowpass signals, and

$$E_b = \frac{1}{2} \int_0^{T_b} |\tilde{s}_1(t)|^2 dt = \frac{1}{2} \int_0^{T_b} |\tilde{s}_2(t)|^2 dt$$

For example, with

$$\tilde{s}_1(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_1 t}, \ \tilde{s}_2(t) = \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_2 t}$$

then

$$\rho = \frac{1}{2E_b} \int_0^{T_b} \sqrt{\frac{2E_b}{T_b}} e^{j2\pi f_1 t} \sqrt{\frac{2E_b}{T_b}} e^{-j2\pi f_2 t} dt = \frac{1}{T_b} \int_0^{T_b} e^{j2\pi (f_1 - f_2) t} dt$$

$$= \frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} e^{j\pi \Delta f t}$$

where $\Delta f = f_1 - f_2$.

From equation 8.44 in [2],

$$\mathrm{Re}[\rho] = \mathrm{Re}\left[\frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} e^{j\pi \Delta f t} \right] = \frac{\sin(\pi \Delta f T_b)}{\pi \Delta f T_b} \cos(\pi \Delta f T_b) = \frac{\sin(2\pi \Delta f T_b)}{2\pi \Delta f T_b}$$

$$\Rightarrow P_b = Q\left(\sqrt{\frac{E_b(1 - \sin(2\pi \Delta f T_b)/(2\pi \Delta f T_b))}{N_0}} \right)$$

where $h = \Delta f T_b$.

Orthogonal M-FSK with Noncoherent Detection

From equation 5.4-46 in [1] and equation 8.66 in [2],

$$P_s = \sum_{m=1}^{M-1} (-1)^{m+1} \binom{M-1}{m} \frac{1}{m+1} \exp\left[-\frac{m}{m+1}\frac{kE_b}{N_0}\right]$$

$$P_b = \frac{1}{2}\frac{M}{M-1}P_s$$

Nonorthogonal 2-FSK with Noncoherent Detection

For $M = 2$, this equation 5.4-53 from [1] and this equation 8.69 from [2] apply:

$$P_s = P_b = Q(\sqrt{a}, \sqrt{b}) - \frac{1}{2}\exp\left(-\frac{a+b}{2}\right) I_0(\sqrt{ab})$$

where

$$a = \frac{E_b}{2N_0}\left(1 - \sqrt{1 - |\rho|^2}\right),\ b = \frac{E_b}{2N_0}\left(1 + \sqrt{1 - |\rho|^2}\right)$$

Precoded MSK with Coherent Detection

Use the same BER and SER computations as for BPSK.

Differentially Encoded MSK with Coherent Detection

Use the same BER and SER computations as for DE-BPSK.

MSK with Noncoherent Detection (Optimum Block-by-Block)

The upper bound on error rate from equations 10.166 and 10.164 in [6]) is

$$P_s = P_b$$

$$\leq \frac{1}{2}\left[1 - Q(\sqrt{b_1}, \sqrt{a_1}) + Q(\sqrt{a_1}, \sqrt{b_1})\right] + \frac{1}{4}\left[1 - Q(\sqrt{b_4}, \sqrt{a_4}) + Q(\sqrt{a_4}, \sqrt{b_4})\right] + \frac{1}{2}e^{-\frac{E_b}{N_0}}$$

where

$$a_1 = \frac{E_b}{N_0}\left(1 - \sqrt{\frac{3 - 4/\pi^2}{4}}\right),\ b_1 = \frac{E_b}{N_0}\left(1 + \sqrt{\frac{3 - 4/\pi^2}{4}}\right)$$

$$a_4 = \frac{E_b}{N_0}\left(1 - \sqrt{1 - 4/\pi^2}\right),\ \ b_4 = \frac{E_b}{N_0}\left(1 + \sqrt{1 - 4/\pi^2}\right)$$

CPFSK Coherent Detection (Optimum Block-by-Block)

The lower bound on error rate (from equation 5.3-17 in [1]) is

$$P_s > K_{\delta_{\min}} Q\left(\sqrt{\frac{E_b}{N_0}\delta_{\min}^2}\right)$$

The upper bound on error rate is

$$\delta_{\min}^2 > \min_{1 \leq i \leq M-1} \{2i(1 - \operatorname{sinc}(2ih))\}$$

where h is the modulation index, and $K_{\delta_{\min}}$ is the number of paths with the minimum distance.

$$P_b \cong \frac{P_s}{k}$$

Analytical Expressions Used in berfading Function and Bit Error Rate Analysis App

This section covers the main analytical expressions used in the `berfading` function and the **Bit Error Rate Analysis** app.

Notation

This table describes the additional notations used in analytical expressions in this section.

Description	Notation
Power of the fading amplitude r	$\Omega = E\left[r^2\right]$, where $E[\cdot]$ denotes statistical expectation
Number of diversity branches	L
Signal to Noise Ratio (SNR) per symbol per branch	$\bar{\gamma}_l = \left(\Omega_l \frac{E_s}{N_0}\right)/L = \left(\Omega_l \frac{kE_b}{N_0}\right)/L$ For identically-distributed diversity branches, $\bar{\gamma} = \left(\Omega \frac{kE_b}{N_0}\right)/L$

Description	Notation
Moment generating functions for each diversity branch	For Rayleigh fading channels: $M_{\gamma_l}(s) = \frac{1}{1 - s\bar{\gamma}_l}$ For Rician fading channels: $M_{\gamma_l}(s) = \frac{1+K}{1+K-s\bar{\gamma}_l} e\left[\frac{Ks\bar{\gamma}_l}{(1+K)-s\bar{\gamma}_l}\right]$ K is the ratio of the energy in the specular component to the energy in the diffuse component (linear scale). For identically-distributed diversity branches, $M_{\gamma_l}(s) = M_{\gamma}(s)$ for all l.

This table defines the additional acronyms used in this section.

Acronym	Definition
MRC	Maximal-ratio combining
EGC	Equal-gain combining

M-PSK with MRC

From equation 9.15 in [2],

$$P_s = \frac{1}{\pi} \int_0^{(M-1)\pi/M} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{\sin^2(\pi/M)}{\sin^2\theta}\right) d\theta$$

From [4] and [2],

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2} (w_i')\bar{P}_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i,

$$\bar{P}_i = \frac{1}{2\pi} \int_0^{\pi(1-(2i-1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i-1)\pi}{M}\right) d\theta$$
$$-\frac{1}{2\pi} \int_0^{\pi(1-(2i+1)/M)} \prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\sin^2\frac{(2i+1)\pi}{M}\right) d\theta$$

For the special case of Rayleigh fading with $M = 2$ (from equations C-18 and C-21 and Table C-1 in [6]),

$$P_b = \frac{1}{2}\left[1 - \mu \sum_{i=0}^{L-1} \binom{2i}{i}\left(\frac{1-\mu^2}{4}\right)^i\right]$$

where

$$\mu = \sqrt{\frac{\bar{\gamma}}{\bar{\gamma}+1}}$$

If $L = 1$, then:

$$P_b = \frac{1}{2}\left[1 - \sqrt{\frac{\bar{\gamma}}{\bar{\gamma}+1}}\right]$$

DE-M-PSK with MRC

For $M = 2$ (from equations 8.37 and 9.8-9.11 in [2]),

$$P_s = P_b = \frac{2}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta - \frac{2}{\pi}\int_0^{\pi/4}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1}{\sin^2\theta}\right)d\theta$$

M-PAM with MRC

From equation 9.19 in [2],

$$P_s = \frac{2(M-1)}{M\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(M^2-1)}{\sin^2\theta}\right)d\theta$$

From [5] and [2],

$$P_b = \frac{2}{\pi M \log_2 M} \times \sum_{k=1}^{\log_2 M}\sum_{i=0}^{(1-2^{-k})M-1}\left\{(-1)^{\left\lfloor \frac{i2^{k-1}}{M}\right\rfloor}\left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{M} + \frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(M^2-1)}{\sin^2\theta}\right)d\theta\right\}$$

M-QAM with MRC

For square M-QAM, $k = \log_2 M$ is even (equation 9.21 in [2]),

$$P_s = \frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(2(M-1))}{\sin^2\theta}\right)d\theta - \frac{4}{\pi}\left(1 - \frac{1}{\sqrt{M}}\right)^2\int_0^{\pi/4}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(2(M-1))}{\sin^2\theta}\right)d\theta$$

From [5] and [2]:

$$P_b = \frac{2}{\pi\sqrt{M}\log_2\sqrt{M}} \times \sum_{k=1}^{\log_2\sqrt{M}}\sum_{i=0}^{(1-2^{-k})\sqrt{M}-1}\left\{(-1)^{\left\lfloor \frac{i2^{k-1}}{\sqrt{M}}\right\rfloor}\left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{\sqrt{M}} + \frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(2(M-1))}{\sin^2\theta}\right)d\theta\right\}$$

For rectangular (nonsquare) M-QAM, $k = \log_2 M$ is odd, $M = I \times J$, $I = 2^{\frac{k-1}{2}}$, $J = 2^{\frac{k+1}{2}}$, $\bar{\gamma}_l = \Omega_l \log_2(IJ)\frac{E_b}{N_0}$,

$$P_s = \frac{4IJ - 2I - 2J}{M\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta$$
$$-\frac{4}{M\pi}(1 + IJ - I - J)\int_0^{\pi/4}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta$$

From [5] and [2],

$$P_b = \frac{1}{\log_2(IJ)}\left(\sum_{k=1}^{\log_2 I} P_I(k) + \sum_{l=1}^{\log_2 J} P_J(l)\right)$$
$$P_I(k) = \frac{2}{I\pi}\sum_{i=0}^{(1-2^{-k})I-1}\left\{(-1)^{\left\lfloor \frac{i2^{k-1}}{I}\right\rfloor}\left(2^{k-1} - \left\lfloor \frac{i2^{k-1}}{I} + \frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2i+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$
$$P_J(k) = \frac{2}{J\pi}\sum_{j=0}^{(1-2^{-l})J-1}\left\{(-1)^{\left\lfloor \frac{j2^{l-1}}{J}\right\rfloor}\left(2^{l-1} - \left\lfloor \frac{j2^{l-1}}{J} + \frac{1}{2}\right\rfloor\right)\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(2j+1)^2 3/(I^2+J^2-2)}{\sin^2\theta}\right)d\theta\right\}$$

M-DPSK with Postdetection EGC

From equation 8.165 in [2],

$$P_s = \frac{\sin(\pi/M)}{2\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{[1-\cos(\pi/M)\cos\theta]}\prod_{l=1}^{L} M_{\gamma_l}(-[1-\cos(\pi/M)\cos\theta])d\theta$$

From [4] and [2],

$$P_b = \frac{1}{k}\left(\sum_{i=1}^{M/2}(w_i')\bar{A}_i\right)$$

where $w_i' = w_i + w_{M-i}$, $w_{M/2}' = w_{M/2}$, w_i is the Hamming weight of bits assigned to symbol i,

$$\bar{A}_i = \bar{F}\left((2i+1)\frac{\pi}{M}\right) - \bar{F}\left((2i-1)\frac{\pi}{M}\right)$$
$$\bar{F}(\psi) = -\frac{\sin\psi}{4\pi}\int_{-\pi/2}^{\pi/2}\frac{1}{(1-\cos\psi\cos t)}\prod_{l=1}^{L} M_{\gamma_l}(-(1-\cos\psi\cos t))dt$$

For the special case of Rayleigh fading with $M = 2$ and $L = 1$ (equation 8.173 from [2]),

$$P_b = \frac{1}{2(1+\bar{\gamma})}$$

Orthogonal 2-FSK, Coherent Detection with MRC

From equation 9.11 in [2],

$$P_s = P_b = \frac{1}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{1/2}{\sin^2\theta}\right)d\theta$$

For the special case of Rayleigh fading (equations 14.4-15 and 14.4-21 in [1]),

$$P_s = P_b = \frac{1}{2^L}\left(1-\sqrt{\frac{\bar{\gamma}}{2+\bar{\gamma}}}\right)^L\sum_{k=0}^{L-1}\binom{L-1+k}{k}\frac{1}{2^k}\left(1+\sqrt{\frac{\bar{\gamma}}{2+\bar{\gamma}}}\right)^k$$

Nonorthogonal 2-FSK, Coherent Detection with MRC

From equations 9.11 and 8.44 in [2],

$$P_s = P_b = \frac{1}{\pi}\int_0^{\pi/2}\prod_{l=1}^{L} M_{\gamma_l}\left(-\frac{(1-\mathrm{Re}[\rho])/2}{\sin^2\theta}\right)d\theta$$

For the special case of Rayleigh fading with $L = 1$ (equations 20 in [8] and 8.130 in [2]),

$$P_s = P_b = \frac{1}{2}\left[1-\sqrt{\frac{\bar{\gamma}(1-\mathrm{Re}[\rho])}{2+\bar{\gamma}(1-\mathrm{Re}[\rho])}}\right]$$

Orthogonal M-FSK, Noncoherent Detection with EGC

For Rayleigh fading, from equation 14.4-47 in [1],

$$P_s = 1-\int_0^{\infty}\frac{1}{(1+\bar{\gamma})^L(L-1)!}U^{L-1}e^{-\frac{U}{1+\bar{\gamma}}}\left(1-e^{-U}\sum_{k=0}^{L-1}\frac{U^k}{k!}\right)^{M-1}dU$$

$$P_b = \frac{1}{2}\frac{M}{M-1}P_s$$

For Rician fading from equation 41 in [8],

$$P_s = \sum_{r=1}^{M-1}\frac{(-1)^{r+1}e^{-LK\bar{\gamma}_r/(1+\bar{\gamma}_r)}}{(r(1+\bar{\gamma}_r)+1)^L}\binom{M-1}{r}\sum_{n=0}^{r(L-1)}\beta_{nr}\frac{\Gamma(L+n)}{\Gamma(L)}\left[\frac{1+\bar{\gamma}_r}{r+1+r\bar{\gamma}_r}\right]^n {}_1F_1\left(L+n, L; \frac{LK\bar{\gamma}_r/(1+\bar{\gamma}_r)}{r(1+\bar{\gamma}_r)+1}\right)$$

$$P_b = \frac{1}{2}\frac{M}{M-1}P_s$$

where

$$\bar{\gamma}_r = \frac{1}{1+K}\bar{\gamma}$$

$$\beta_{nr} = \sum_{i=n-(L-1)}^{n}\frac{\beta_{i(r-1)}}{(n-i)!}I_{[0,\ (r-1)(L-1)]}(i)$$

$$\beta_{00} = \beta_{0r} = 1$$

$$\beta_{n1} = 1/n!$$

$$\beta_{1r} = r$$

and $I_{[a,b]}(i) = 1$ if $a \le i \le b$ and 0 otherwise.

Nonorthogonal 2-FSK, Noncoherent Detection with No Diversity

From equation 8.163 in [2],

$$P_s = P_b = \frac{1}{4\pi}\int_{-\pi}^{\pi}\frac{1-\varsigma^2}{1+2\varsigma\sin\theta+\varsigma^2}M_\gamma\left(-\frac{1}{4}(1+\sqrt{1-\rho^2})(1+2\varsigma\sin\theta+\varsigma^2)\right)d\theta$$

where

$$\varsigma = \sqrt{\frac{1-\sqrt{1-\rho^2}}{1+\sqrt{1-\rho^2}}}$$

Analytical Expressions Used in bercoding Function and Bit Error Rate Analysis App

This section covers the main analytical expressions used in the `bercoding` function and the **Bit Error Rate Analysis** app.

- “Common Notation” on page 23-57
- “Block Coding” on page 23-57
- “Convolutional Coding” on page 23-59

Common Notation

This table describes the additional notations used in analytical expressions in this section.

Description	Notation
Energy-per-information bit-to-noise power-spectral-density ratio	$\gamma_b = \frac{E_b}{N_0}$
Message length	K
Code length	N
Code rate	$R_c = \frac{K}{N}$

Block Coding

This section describes the specific notation for block coding expressions, where $d_{\min}$ is the minimum distance of the code.

Soft Decision

For BPSK, QPSK, OQPSK, 2-PAM, 4-QAM, and precoded MSK, equation 8.1-52 in [1]) applies,

$$P_b \le \frac{1}{2}(2^K-1)Q(\sqrt{2\gamma_b R_c d_{\min}})$$

For DE-BPSK, DE-QPSK, DE-OQPSK, and DE-MSK,

$$P_b \le \frac{1}{2}(2^K-1)[2Q(\sqrt{2\gamma_b R_c d_{\min}})[1-Q(\sqrt{2\gamma_b R_c d_{\min}})]]$$

For BFSK coherent detection, equations 8.1-50 and 8.1-58 in [1] apply,

$$P_b \le \frac{1}{2}(2^K - 1)Q(\sqrt{\gamma_b R_c d_{\min}})$$

For BFSK noncoherent square-law detection, equations 8.1-65 and 8.1-64 in [1] apply,

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp\left(-\frac{1}{2}\gamma_b R_c d_{\min}\right)\sum_{i=0}^{d_{\min}-1}\left(\frac{1}{2}\gamma_b R_c d_{\min}\right)^i\frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

For DPSK,

$$P_b \le \frac{1}{2}\frac{2^K - 1}{2^{2d_{\min}-1}}\exp(-\gamma_b R_c d_{\min})\sum_{i=0}^{d_{\min}-1}(\gamma_b R_c d_{\min})^i\frac{1}{i!}\sum_{r=0}^{d_{\min}-1-i}\binom{2d_{\min}-1}{r}$$

Hard Decision

For general linear block code, equations 4.3 and 4.4 in [9], and 12.136 in [6] apply,

$$P_b \le \frac{1}{N}\sum_{m=t+1}^{N}(m+t)\binom{N}{m}p^m(1-p)^{N-m}$$

$$t = \left\lfloor\frac{1}{2}(d_{\min}-1)\right\rfloor$$

For Hamming code, equations 4.11 and 4.12 in [9] and 6.72 and 6.73 in [7] apply

$$P_b \approx \frac{1}{N}\sum_{m=2}^{N} m\binom{N}{m}p^m(1-p)^{N-m} = p - p(1-p)^{N-1}$$

For rate (24,12) extended Golay code, equations 4.17 in [9] and 12.139 in [6] apply:

$$P_b \le \frac{1}{24}\sum_{m=4}^{24}\beta_m\binom{24}{m}p^m(1-p)^{24-m}$$

where β_m is the average number of channel symbol errors that remain in corrected N-tuple format when the channel caused m symbol errors (see table 4.2 in [9]).

For Reed-Solomon code with $N = Q - 1 = 2^q - 1$,

$$P_b \approx \frac{2^{q-1}}{2^q - 1}\frac{1}{N}\sum_{m=t+1}^{N} m\binom{N}{m}(P_s)^m(1-P_s)^{N-m}$$

For FSK, equations 4.25 and 4.27 in [9], 8.1-115 and 8.1-116 in [1], 8.7 and 8.8 in [7], and 12.142 and 12.143 in [6] apply,

$$P_b \approx \frac{1}{q}\frac{1}{N}\sum_{m=t+1}^{N} m\binom{N}{m}(P_s)^m(1-P_s)^{N-m}$$

otherwise, if $\log_2 Q/\log_2 M = q/k = h$, where h is an integer (equation 1 in [10]) applies,

$$P_s = 1 - (1-s)^h$$

where s is the SER in an uncoded AWGN channel.

For example, for BPSK, $M = 2$ and $P_s = 1 - (1 - s)^q$, otherwise P_s is given by table 1 and equation 2 in [10].

Convolutional Coding

This section describes the specific notation for convolutional coding expressions, where d_{free} is the free distance of the code, and a_d is the number of paths of distance d from the all-zero path that merges with the all-zero path for the first time.

Soft Decision

From equations 8.2-26, 8.2-24, and 8.2-25 in [1] and 13.28 and 13.27 in [6] apply,

$$P_b < \sum_{d = d_{free}}^{\infty} a_d f(d) P_2(d)$$

The transfer function is given by

$$T(D, N) = \sum_{d = d_{free}}^{\infty} a_d D^d N^{f(d)}$$

$$\left.\frac{dT(D, N)}{dN}\right|_{N = 1} = \sum_{d = d_{free}}^{\infty} a_d f(d) D^d$$

where $f(d)$ is the exponent of N as a function of d.

This equation gives the results for BPSK, QPSK, OQPSK, 2-PAM, 4-QAM, precoded MSK, DE-BPSK, DE-QPSK, DE-OQPSK, DE-MSK, DPSK, and BFSK:

$$P_2(d) = P_b\left|_{\frac{E_b}{N_0} = \gamma_b R_c d}\right.$$

where P_b is the BER in the corresponding uncoded AWGN channel. For example, for BPSK (equation 8.2-20 in [1]),

$$P_2(d) = Q(\sqrt{2\gamma_b R_c d})$$

Hard Decision

From equations 8.2-33, 8.2-28, and 8.2-29 in [1] and 13.28, 13.24, and 13.25 in [6] apply,

$$P_b < \sum_{d = d_{free}}^{\infty} a_d f(d) P_2(d)$$

When d is odd,

$$P_2(d) = \sum_{k = (d + 1)/2}^{d} \binom{d}{k} p^k (1 - p)^{d - k}$$

and when d is even,

$$P_2(d) = \sum_{k = d/2 + 1}^{d} \binom{d}{k} p^k (1-p)^{d-k} + \frac{1}{2}\binom{d}{d/2} p^{d/2}(1-p)^{d/2}$$

where p is the bit error rate (BER) in an uncoded AWGN channel.

Analytical Expressions Used in bersync Function and Bit Error Rate Analysis App

- "Timing Synchronization Error" on page 23-60
- "Timing Synchronization Error" on page 23-60

This section covers the main analytical expressions used in the `bersync` function and the **Bit Error Rate Analysis** app.

Timing Synchronization Error

To compute the BER for a communications system with a timing synchronization error, the `bersync` function uses this formula from [13]:

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

where σ is the timing error, and R is the linear E_b/N_0 value.

Timing Synchronization Error

To compute the BER for a communications system with a carrier synchronization error, the `bersync` function uses this formula from [13]:

$$\frac{1}{\pi\sigma}\int_{0}^{\infty}\exp\left(-\frac{\phi^2}{2\sigma^2}\right)\int_{\sqrt{2R}\cos\phi}^{\infty}\exp\left(-\frac{y^2}{2}\right)dy\,d\phi$$

where σ is the phase error R is the linear E_b/N_0 value.

See Also

Apps
Bit Error Rate Analysis

Functions
`berawgn` | `bercoding` | `berconfint` | `berfading` | `berfit` | `bersync`

Related Examples

- "Bit Error Rate Analysis Techniques" on page 23-2
- "Use Bit Error Rate Analysis App" on page 23-12

Error Vector Magnitude (EVM)

Error Vector Magnitude (EVM) is a measurement of modulator or demodulator performance in the presence of impairments. Essentially, EVM is the vector difference at a given time between the ideal (transmitted) signal and the measured (received) signal. If used correctly, these measurements can help in identifying sources of signal degradation, such as: phase noise, I-Q imbalance, amplitude non-linearity and filter distortion

These types of measurements are useful for determining system performance in communications applications. For example, determining if an EDGE system conforms to the 3GPP radio transmission standards requires accurate RMS, EVM, Peak EVM, and 95th percentile for the EVM measurements.

Users can create the EVM object in two ways: using a default object or by defining parameter-value pairs. As defined by the 3GPP standard, the unit of measure for RMS, Maximum, and Percentile EVM measurements is a percentile (%). For more information, see the EVM Measurement or `comm.EVM` help page.

Measuring Modulator Accuracy

Overview

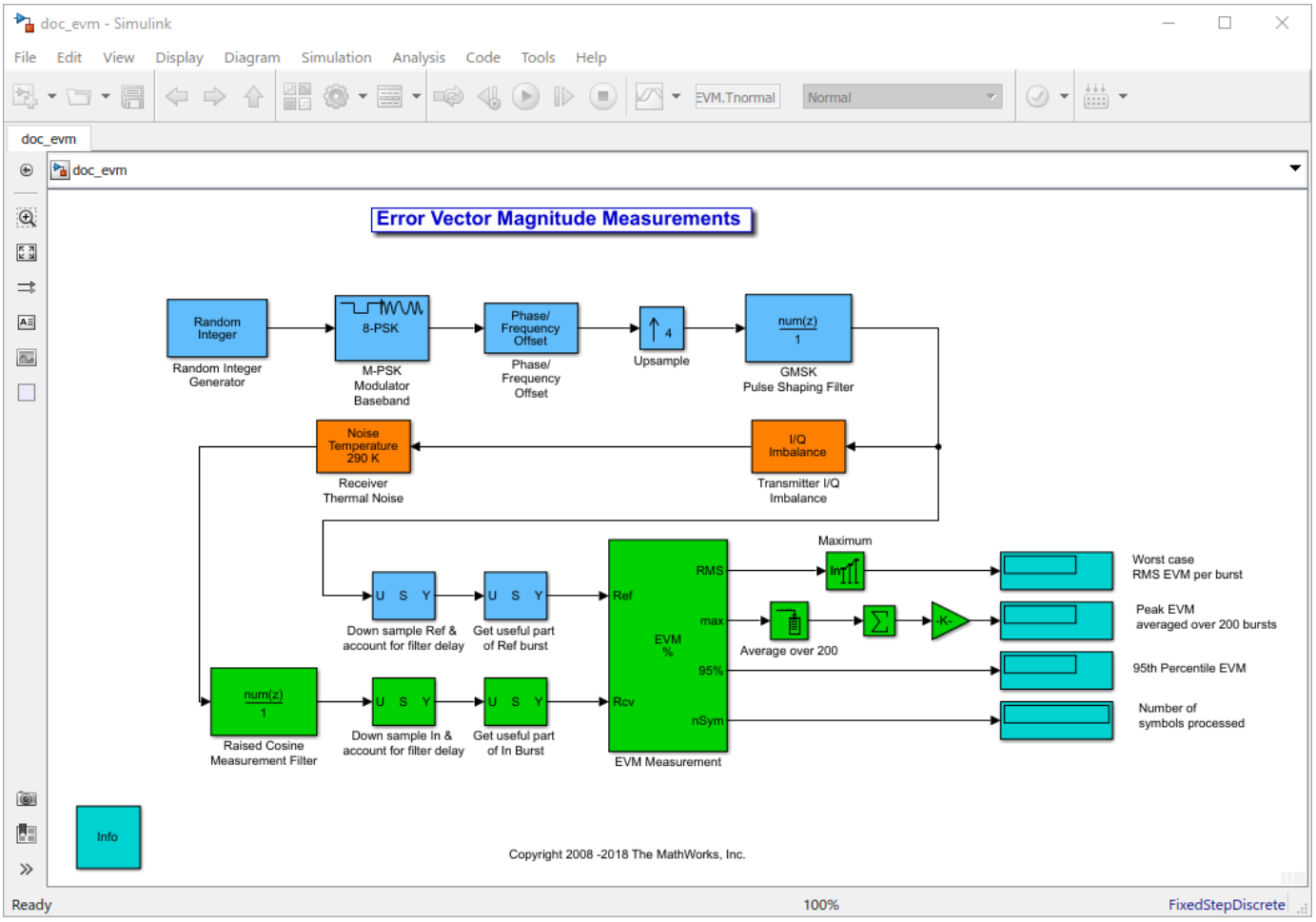
The Communications Toolbox provides two blocks you can use for measuring modulator accuracy: EVM Measurement and MER Measurement.

This example tests an EDGE transmitter for system design impairments using EVM measurements. In this example, the EVM Measurements block compares an ideal reference signal to a measured signal, and then computes RMS EVM, maximum EVM, and percentile EVM values. According to the EDGE standard [1], the error vector magnitude of the received signal, calculated relative to the transmitted waveform, should not exceed the following values:

EDGE Standard Measurement Specifications [2]

Measurement	Mobile Station		Base Transceiver Station	
	Normal	Extreme	Normal	Extreme
RMS	9%	10%	7%	8%
Peak EVM	30%	30%	22%	22%
95th Percentile EVM	15%	15%	11%	11%

This example uses this model.



You can open this model by typing `doc_evm` at the MATLAB command line.

Structure

The model essentially contains three parts:

- Transmitter
- Receiver impairments
- EVM calculation

The following sections of the tutorial contain descriptions for each part of the model.

Transmitter

The following blocks comprise the transmitter:

- Random Integer Generator
- M-PSK Modulator Baseband
- Phase/Frequency Offset
- Upsample

- Discrete FIR Filter
- I/Q Imbalance

The Random Integer Generator block simulates random data generation. The EDGE standard specifies that the transmitter performs measurements during the useful part of the burst – excluding tail bits – over at least 200 bursts. In this mode, the transmitter produces 435 symbols per burst (9 additional symbols account for filter delays). The Phase Offset block provides continuous 3π/8 phase rotation to the signal. For synchronization purposes, the Upsample block oversamples the signal by a factor of 4.

The Discrete FIR Filter block provides a GMSK pulse linearization, the main component in a Laurent decomposition of the GMSK modulation [3]. A helper function computes the filter coefficients and uses a direct-form FIR digital filter to create the pulse shaping effect. The filter normalization provides unity gain at the main tap.

The I/Q Imbalance block simulates transmitter impairments. This block adds rotation to the signal, simulating a defect in the transmitter under test. The **I/Q amplitude imbalance** is `0.5` dB, and **I/Q phase imbalance** is 1°.

Receiver Impairments

In this model, the Receiver Thermal Noise block represents receiver impairments. This model assumes 290 K of thermal noise, representing imperfections of the hardware under test.

EVM Calculation

The EVM calculation relies upon the following blocks:

- Discrete FIR Filter
- Selector
- EVM Measurement
- Display

The EVM measurement block computes the vector difference between an ideal reference signal and an impaired signal. The output of the FIR filter provides the `Reference` input for the EVM block. The output of the Noise Temperature block provides the impaired signal at the `Input` port of the EVM block.

While the block has different normalization options available, the EDGE standard requires normalizing by the `Average reference signal power`. For illustration purposes in this example, the EVM block outputs RMS, maximum, and percentile measurement values.

Experimenting with the Model

1. Run the model by clicking the play button in the Simulink model window.
2. Examine the output of the EVM block and compare the measurements to the limits in the EDGE Standard Measurement Specifications table.

   In this example, the EVM Measurement block computes the following:

   - Worst case RMS EVM per burst: 9.77%
   - Peak EVM: 18.95%
   - 95th Percentile EVM:14.76%

As a result, this simulated EDGE transmitter passes the EVM test for a Mobile Station under extreme conditions.

3. Double-click the I/Q Imbalance block.
4. Enter 2 into **I/Q Imbalance (dB)** and click **OK**.
5. Click the Play button in the Simulink model window.
6. Examine the output of the EVM block. Then, compare the measurements to the limits in the EDGE Standard Measurement Specifications table.

   In this example, the EVM Measurement block computes the following results:

   - Worst case RMS EVM per burst: 15.15%
   - Peak EVM: 29.73%
   - 95th Percentile EVM: 22.55%.

   These EVM values are clearly unacceptable according to the EDGE standard. You can experiment with the other I/Q imbalance values, examine the impact on calculations, and compare them to the values provided in the table.

References

[1] 3GPP TS 45.004, “Radio Access Networks; Modulation,” Release 7, v7.2.0, 2008-02.

[2] 3GPP TS 45.005, “Radio Access Network; Radio transmission and reception,” Release 8, v8.1.0, 2008-05.

[3] Laurent, Pierre. “Exact and approximate construction of digital phase modulation by superposition of amplitude modulated pulses (AMP).” *IEEE Transactions on Communications*. Vol. COM-34, #2, Feb. 1986, pp. 150-160.

Modulation Error Ratio (MER)

Communications Toolbox can perform Modulation Error Ratio (MER) measurements. MER is a measure of the signal-to-noise ratio (SNR) in a digital modulation applications. These types of measurements are useful for determining system performance in communications applications. For example, determining if an EDGE system conforms to the 3GPP radio transmission standards requires accurate RMS, EVM, Peak EVM, and 95th percentile for the EVM measurements.

As defined by the DVB standard, the unit of measure for MER is decibels (dB). For consistency, the unit of measure for Minimum MER and Percentile MER measurements is also in decibels. For more information, see the `comm.MER` help page.

Adjacent Channel Power Ratio (ACPR)

Adjacent channel power ratio (ACPR) calculations (also known as adjacent channel leakage ratio (ACLR)), characterize spectral regrowth in a communications system component, such as a modulator or an analog front end. Amplifier nonlinearity causes spectral regrowth. ACPR calculations determine the likelihood that a given system causes interference with an adjacent channel.

Many transmission standards, such as IS-95, CDMA, WCDMA, 802.11, and Bluetooth, contain a definition for ACPR measurements. Most standards define ACPR measurements as the ratio of the average power in the main channel and any adjacent channels. The offset frequencies and measurement bandwidths (BWs) you use when obtaining measurements depends on which specific industry standard you are using. For instance, measurements for CDMA amplifiers involve two offsets (from the carrier frequency) of 885 kHz and 1.98 MHz, and a measurement BW of 30 KHz.

For more information, see the `comm.ACPR` help page.

Obtain ACPR Measurements

Communications Toolbox contains the `comm.ACPR` System object. In this tutorial, you obtain ACPR measurements using a WCDMA communications signal, according to the 3GPP™ TS 125.104 standard.

This example uses baseband WCDMA sample signals at the input and output of a nonlinear amplifier. The WCDMASignal.mat file contains sample data for use with the tutorial. This file divides the data into 25 signal snapshots of 7e3 samples each and stores them in the columns of data matrices, `dataBeforeAmplifier` and `dataAfterAmplifier`.

The WCDMA specification requires that you obtain all measurements using a 3.84 MHz sampling frequency.

Create comm.ACPR System Object and Set Up Measurements

1. Define the sample rate, load the WCDMA file, and get the data by entering the following at the MATLAB command line:

```
% System sampling frequency, 3.84 MHz chip rate, 8 samples per chip
SampleRate = 3.84e6*8;
load WCDMASignal.mat
% Use the first signal snapshot
txSignalBeforeAmplifier = dataBeforeAmplifier(:,1);
txSignalAfterAmplifier = dataAfterAmplifier(:,1);
```

2. Create the comm.ACPR System object and specify the sampling frequency.

```
hACPR = comm.ACPR('SampleRate',SampleRate)
```

The System object presents the following information:

```
                NormalizedFrequency: false
                         SampleRate: 30720000
                MainChannelFrequency: 0
           MainMeasurementBandwidth: 50000
              AdjacentChannelOffset: [-100000 100000]
      AdjacentMeasurementBandwidth: 50000
           MeasurementFilterSource: 'None'
```

```
                SpectralEstimation: 'Auto'
                          FFTLength: 'Next power of 2'
                            MaxHold: false
                         PowerUnits: 'dBm'
        MainChannelPowerOutputPort: false
    AdjacentChannelPowerOutputPort: false
```

3 Specify the *main channel* center frequency and measurement bandwidth.

Specify the main channel center frequency using the `MainChannelFrequency` property. Then, specify the main channel measurement bandwidth using the `MainMeasurementBandwidth` property.

For the baseband data you are using, the main channel center frequency is at 0 Hz. The WCDMA standard specifies that you obtain main channel power using a 3.84-MHz measurement bandwidth. Specify these by typing the following.

```
hACPR.MainChannelFrequency = 0;
hACPR.MainMeasurementBandwidth = 3.84e6;
```

4 Specify *adjacent channel* offsets and measurement bandwidths.

The WCDMA standard specifies ACPR limits for four adjacent channels, located at 5, -5, 10, -10 MHz away from the main channel center frequency. In all cases, you obtain adjacent channel power using a 3.84-MHz bandwidth. Specify the adjacent channel offsets and measurement bandwidths using the `AdjacentChannelOffset` and `AdjacentMeasurementBandwidth` properties.

```
hACPR.AdjacentChannelOffset = [-10 -5 5 10]*1e6;
hACPR.AdjacentMeasurementBandwidth = 3.84e6;
```

Notice that if the measurement bandwidths for all the adjacent channels are equal, you specify a scalar value. If measurement bandwidths are different, you specify a vector of measurement bandwidths with a length equal to the length of the offset vector.

5 Set the `MainChannelPowerOutputPort` and `AdjacentChannelPowerOutputPort` properties to `true` by entering the following at the MATLAB command line:

```
hACPR.MainChannelPowerOutputPort = true
hACPR.AdjacentChannelPowerOutputPort = true
```

6 Create a `comm.ACPR` System object to measure the amplifier output.

```
hACPRoutput = clone(hACPR);
```

Obtain the ACPR Measurements

You obtain ACPR measurements by calling the `step` method of `comm.ACPR`. You can also obtain the power measurements for the main and adjacent channels. The `PowerUnits` property specifies the unit of measure. The property value defaults to dBm (power ratio referenced to one milliwatt (mW)).

1 Obtain the ACPR measurements at the amplifier input:

```
[ACPR mainChannelPower adjChannelPower] = hACPR(txSignalBeforeAmplifier);
```

The `comm.ACPR` System object produces the following output measurement data:

```
ACPR =

  -68.6668  -54.9002  -55.0653  -68.4604
```

```
mainChannelPower =

   29.5190


adjChannelPower =

  -39.1477  -25.3812  -25.5463  -38.9414
```

2 Obtain the ACPR measurements at the amplifier output:

```
[ACPR mainChannelPower adjChannelPower] = hACPRoutput(txSignalAfterAmplifier)
```

The `comm.ACPR` System object produces the following input measurement data:

```
ACPR =

  -42.1625  -27.0912  -26.8785  -42.4915

mainChannelPower =

   40.6725

adjChannelPower =

   -1.4899   13.5813   13.7941   -1.8190
```

Notice the increase in ACPR values at the output of the amplifier. This increase reflects distortion due to amplifier nonlinearity. The WCDMA standard specifies that ACPR values be below -45 dB at +/- 5 MHz offsets, and below -50 dB at +/- 10 MHz offsets. In this example, the signal at the amplifier input meets the specifications while the signal at the amplifier output does not.

Specifying a Measurement Filter

The WCDMA standard specifies that you obtain ACPR measurements using a root-raised-cosine filter. It also states that you measure *both* the main channel power and adjacent channel powers using a matched root-raised-cosine (RRC) filter with rolloff factor 0.22. You specify the measurement filter using the `MeasurementFilter` property. This property value defaults to an all-pass filter with unity gain.

The filter must be an FIR filter, and its response must center at 0 Hz. The ACPR object automatically shifts and applies the filter at each of the specified main and adjacent channel bands. (The power measurement still falls within the bands specified by the `MainMeasurementBandwidth`, and `AdjacentMeasurementBandwidth` properties.)

The WCDMASignal.mat file contains data that was obtained using a 96 tap filter with a rolloff factor of 0.22.

1 Create the filter (using `rcosdesign`, from the Signal Processing Toolbox software) and obtain measurements by entering the following at the MATLAB command line:

```
% Scale for 0 dB passband gain
measFilt = rcosdesign(0.22,16,8)/sqrt(8);
```

2 Set the filter you created in the previous step as the measurement filter for the ACPR object.

```
release(hACPR);
hACPR.MeasurementFilterSource = 'Property';
hACPR.MeasurementFilter = measFilt;
```

3 Implement the same filter at the amplifier output by cloning the `comm.ACPR` System object.

```
hACPRoutput = clone(hACPR)
```

4 Obtain the ACPR power measurements at the amplifier input.

```
ACPR = hACPR(txSignalBeforeAmplifier)
```

The `comm.ACPR` System object produces the following measurement data:

```
ACPR =
  -71.4648  -55.5514  -55.9476  -71.3909
```

5 Obtain the ACPR power measurements at the amplifier output.

```
ACPRoutput = hACPRoutput(txSignalAfterAmplifier)
```

The `comm.ACPR` System object produces the following measurement data:

```
ACPR =
   -42.2364  -27.2242  -27.0748  -42.5810
```

Control the Power Spectral Estimator

By default, the ACPR object measures power uses a Welch power spectral estimator with a Hamming window and zero percent overlap. The object uses a rectangle approximation of the integral for the power spectral density estimates in the measurement bandwidth of interest. If you set SpectralEstimatorOption to 'User defined' several properties become available, providing you control of the resolution, variance, and dynamic range of the spectral estimates.

1 Enable the SegmentLength, OverlapPercentage, and WindowOption properties by entering the following at the MATLAB command line:

```
release(hACPRoutput)
hACPRoutput.SpectralEstimation = 'Specify window parameters'
```

This change allows you to customize the spectral estimates for obtaining power measurements. For example, you can set the spectral estimator segment length to 1024 and increase the overlap percentage to 50%, reducing the consequent variance increase. You can also choose a window with larger side lobe attenuation (compared to the default Hamming window).

2 Create a spectral estimator with a 'Chebyshev' window and a side lobe attenuation of 200 dB.

```
hACPRoutput.SegmentLength = 1024;
hACPRoutput.OverlapPercentage = 50;
% Choosing a Chebyshev window enables a SidelobeAtten property
% you can use to set the side lobe attenuation of the window.
hACPRoutput.Window = 'Chebyshev';
hACPRoutput.SidelobeAttenuation = 200;
```

3 Run the object to obtain the ACPR power measurements at the amplifier output.

```
ACPRoutput = hACPRoutput(txSignalAfterAmplifier)
```

The ACPR object produces the following measurement data:

```
ACPR =
  -44.9399  -30.7136  -30.7670  -44.4450
```

Measure Power Using the Max-Hold Option.

Some communications standards specify using max-hold spectrum power measurements when computing ACPR values. Such calculations compare the current power spectral density vector estimation to the previous max-hold accumulated power spectral density vector estimation. When obtaining max-hold measurements, the object obtains the power spectral density vector estimation using the current input data. It obtains the previous max-hold accumulated power spectral density vector from the previous call to the object. The object uses the maximum values at each frequency bin for calculating average power measurements. A call to the reset method clears the max-hold spectrum.

1 Accumulate max-hold spectra for 25 amplifier output data snapshots and get ACPR measurements by typing the following at the MATLAB command line:

```
for idx = 1:24
    hACPRoutput(dataAfterAmplifier(:,idx));
end
ACPRoutput = hACPRoutput(dataAfterAmplifier(:,25))
```

The ACPR object produces the following output data:

```
ACPR =

  -43.1123  -26.6964  -27.0009  -42.4803
```

Plotting the Signal Spectrum

Use the MATLAB software to plot the power spectral density of the WCDMA signals at the input and output of the nonlinear amplifier. The plot allows you to visualize the spectral regrowth effects intrinsic to amplifier nonlinearity. Notice how the measurements reflect the spectral regrowth. (Note: the following code is just for visualizing signal spectra; it has nothing to do with obtaining the ACPR measurements).

```
win = hamming(1024);
[PSD1,F] = pwelch(txSignalBeforeAmplifier,win,50,1024,SampleRate,'centered');
[PSD2,F] = pwelch(txSignalAfterAmplifier,win,50,1024,SampleRate,'centered');
plot(F,10*log10(PSD1))
hold on
grid on
plot(F,10*log10(PSD2),'g')
legend('Amplifier input', 'Amplifier output')
```

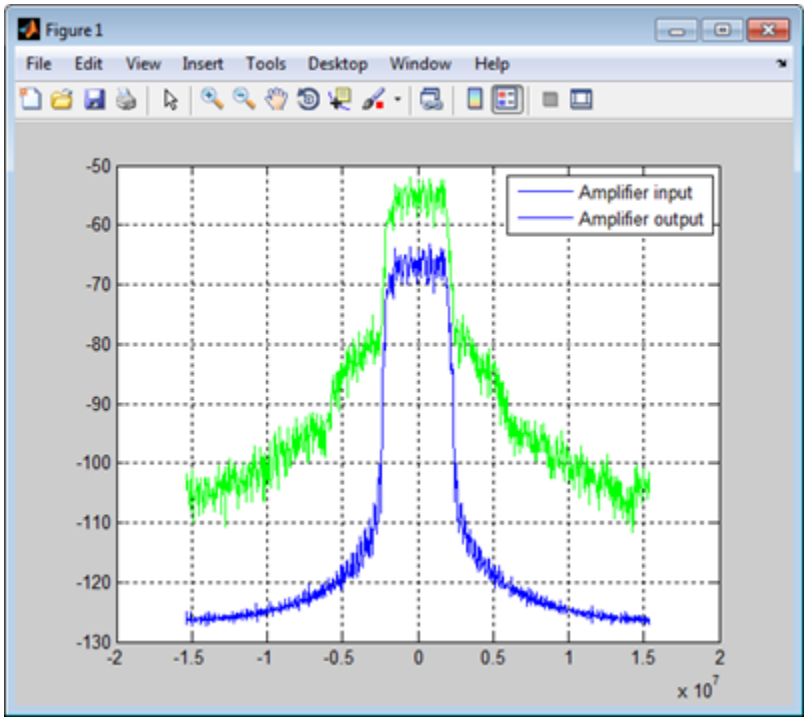


Figure 1
File Edit View Insert Tools Desktop Window Help
Amplifier input
Amplifier output
-50
-60
-70
-80
-90
-100
-110
-120
-130
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
x 10^7

Complementary Cumulative Distribution Function CCDF

Using the `comm.CCDF` System object you can measure the probability that the instantaneous power of a signal is above its specified level average power.

Selected Bibliography for Measurements

List of references for further reading about measurements used in analysis of communications systems bit error rate performance.

References

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

[2] Simon, M. K., and M. S. Alouini. *Digital Communication over Fading Channels: A Unified Approach to Performance Analysis*. 1st ed. New York: John Wiley & Sons, 2000.

[3] Simon, M. K. "On the bit-error probability of differentially encoded QPSK and offset QPSK in the presence of carrier synchronization," *IEEE Transactions on Communications*, vol. 54, no. 5 (May 2006): 806-812, https://doi.org/10.1109/TCOMM.2006.874002.

[4] Lee, P. J. "Computation of the Bit Error Rate of Coherent M-ary PSK with Gray Code Bit Mapping," *IEEE Transactions on Communications*, vol. 34, no. 5 (May 1986): 488-491, https://doi: 10.1109/TCOM.1986.1096558.

[5] Cho, K., and D. Yoon. "On the general BER expression of one- and two-dimensional amplitude modulations," *IEEE Transactions on Communications*, vol. 50, no. 7 (July 2002): 1074-1080, https://doi.org/10.1109/TCOMM.2002.800818.

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

[7] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. 2nd ed. Upper Saddle River, N.J: Prentice-Hall PTR, 2001.

[8] Lindsey, W. C. "Error probabilities for Rician fading multichannel reception of binary and N-ary signals," *IEEE Trans. Inform. Theory*, vol. IT-10, (1964): 339-350.

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

[10] Gulliver, T. A. "Matching Q-ary Reed-Solomon codes with M-ary modulation," *IEEE Transactions on Communications*, vol. 45, no. 11, pp. 1349-1353, Nov. 1997, doi: 10.1109/26.649739.

[11] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan. *Simulation of Communication Systems*. Second edition. Boston, MA: Springer US, 2000.

[12] Frenger, P., P. Orten, and T. Ottosson. “Convolutional Codes with Optimum Distance Spectrum.” *IEEE Communications Letters* 3, no. 11 (November 1999): 317–19. https://doi.org/10.1109/4234.803468.

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

See Also

Functions

`berawgn` | `bercoding` | `berconfint` | `berfading` | `berfit` | `bersync`

More About

- “Analytical Expressions and Notations Used in BER Analysis” on page 23-45
- “Bit Error Rate Analysis Techniques” on page 23-2

Filtering Section

Filtering

In this section...

The Communications Toolbox software includes several functions, objects, and blocks that can help you design and use filters. Other filtering capabilities are in the Signal Processing Toolbox and the DSP System Toolbox. The sections of this chapter are as follows:

For an example involving raised cosine filters, type `showdemo rcosdemo`.

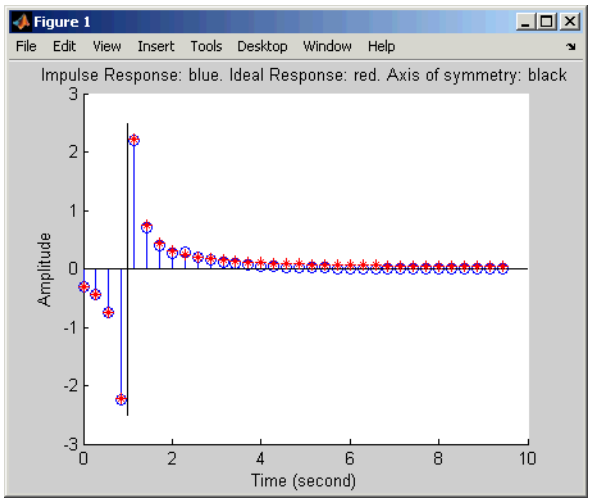
Filter Features

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

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

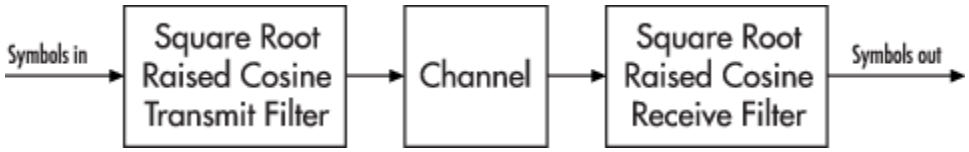
where θ represents the phase of the filter and ω represents the frequency in radians per second. This delay is set so that the impulse response before time zero is negligible and can safely be ignored by the function.

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



Filtering tasks that blocks in the Communications Toolbox support include:

- “Filter Using Simulink Raised Cosine Filter Blocks” on page 24-12. Raised cosine filters are very commonly used for pulse shaping and matched filtering. The following block diagram illustrates a typical use of raised cosine filters.



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

Additional filtering capabilities exist in the Filter Designs and Multirate Filters libraries of the DSP System Toolbox product.

For more background information about filters and pulse shaping, see the works listed in the “Selected Bibliography Filtering” on page 24-3.

Selected Bibliography Filtering

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

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

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

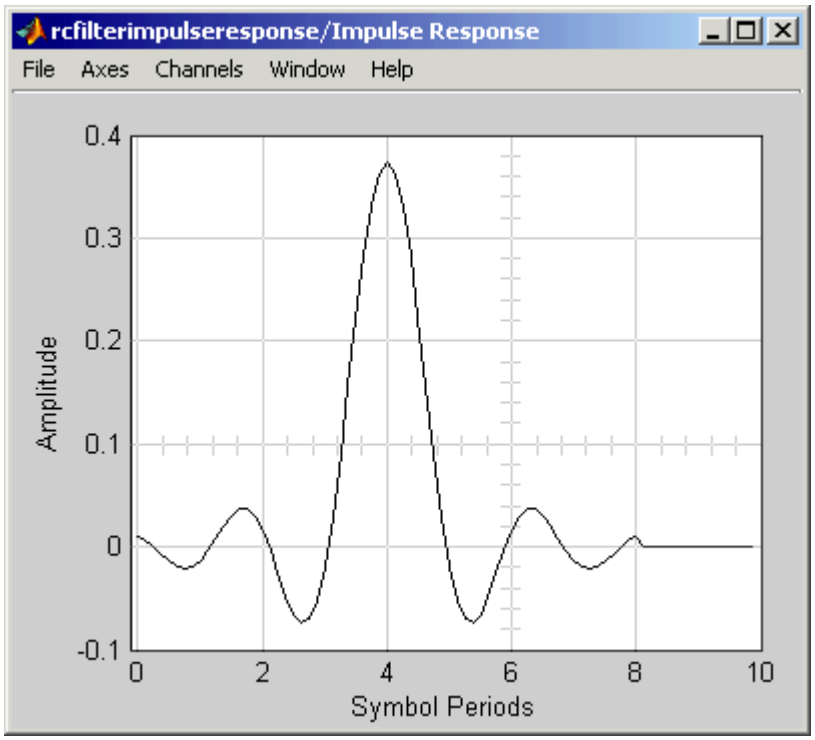
[4] Rappaport, Theodore S., *Wireless Communications: Principles and Practice*, Upper Saddle River, NJ, Prentice Hall, 1996.

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

Group Delay

The raised cosine filter blocks in the `commfilt2` library implement realizable filters by delaying the peak response. This delay, known as the filter's *group delay*, is the length of time between the filter's initial response and its peak response. The filter blocks in this library have a **Filter span in symbols** parameter, which is twice the group delay in symbols.

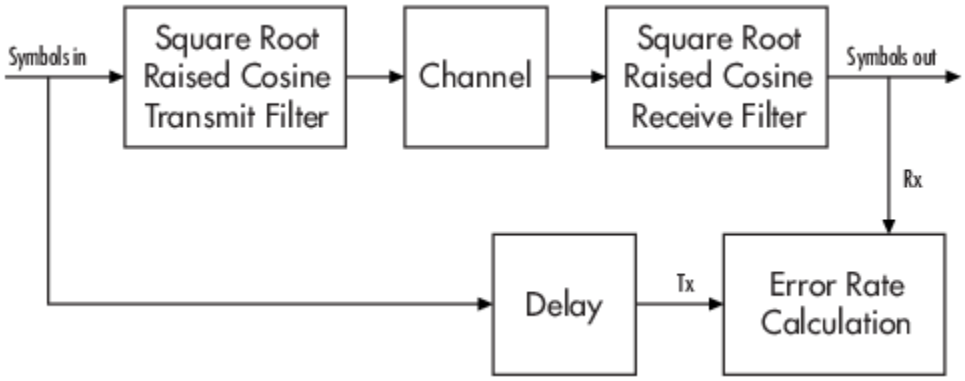
For example, the square-root raised cosine filter whose impulse response shown in the following figure uses a **Filter span in symbols** parameter of 8 in the filter block. In the figure, the initial impulse response is small and the peak impulse response occurs at the fourth symbol.



Implications of Delay for Simulations

A filter block's group delay has implications for other parts of your model. For example, suppose you compare the symbol streams marked Symbols In and Symbols Out in the schematics shown on the "Filtering" on page 24-2 page by plotting or computing an error rate. Use one of these methods to make sure you are comparing symbols that truly correspond to each other:

- Use the Delay block to delay the Symbols In signal, thus aligning it with the Symbols Out signal. Set the **Delay** parameter equal to the filter's group delay (or the sum of both values, if your model uses a pair of square root raised cosine filter blocks). The following figure illustrates this usage.



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

For more information about how to manage delays in a model, see “Delays” on page 10-6.

Pulse Shaping Using a Raised Cosine Filter

Filter a 16-QAM signal using a pair of square root raised cosine matched filters. Plot the eye diagram and scatter plot of the signal. After passing the signal through an AWGN channel, calculate the number of bit errors.

Set the simulation parameters.

```
M = 16; % Modulation order
k = log2(M); % Bits/symbol
n = 20000; % Transmitted bits
nSamp = 4; % Samples per symbol
EbNo = 10; % Eb/No (dB)
```

Set the filter parameters.

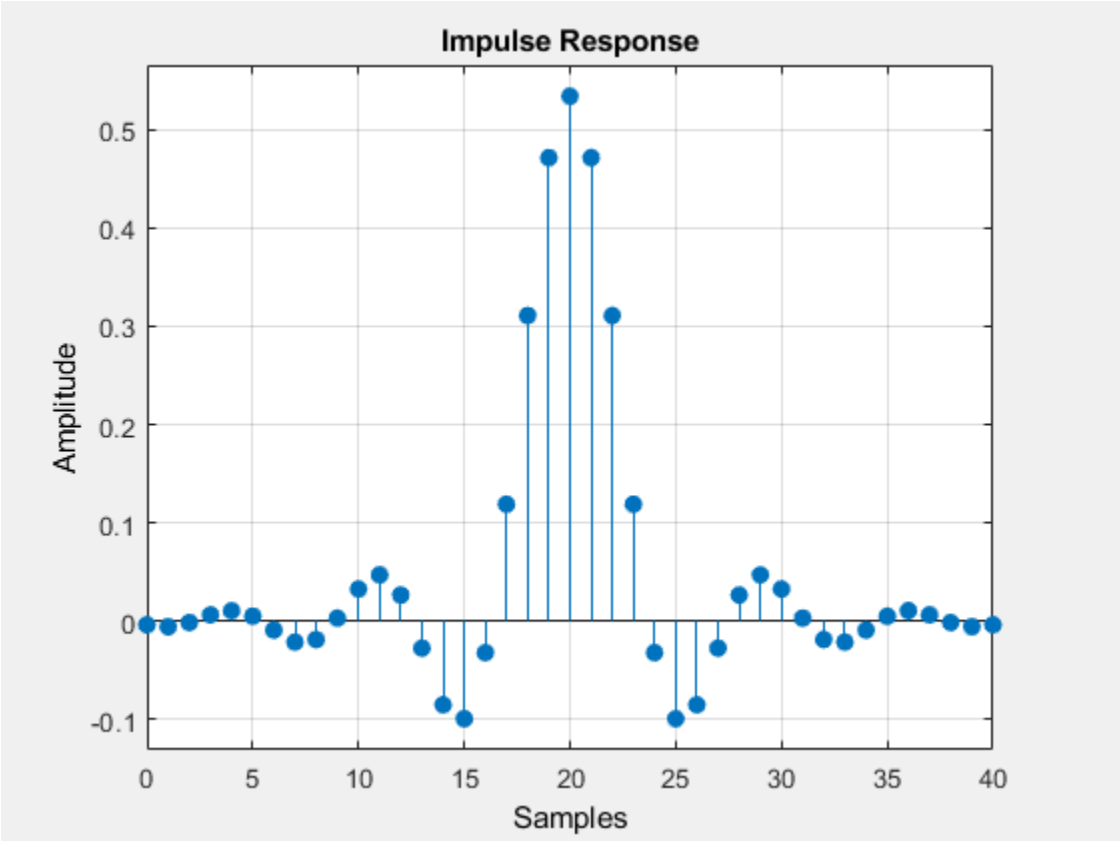
```
span = 10; % Filter span in symbols
rolloff = 0.25; % Rolloff factor
```

Create the raised cosine transmit and receive filters using the previously defined parameters.

```
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',rolloff, ...
    'FilterSpanInSymbols',span,'OutputSamplesPerSymbol',nSamp);

rxfilter = comm.RaisedCosineReceiveFilter('RolloffFactor',rolloff, ...
    'FilterSpanInSymbols',span,'InputSamplesPerSymbol',nSamp, ...
    'DecimationFactor',nSamp);
```

Plot the impulse response of `hTxFilter`.

```
fvtool(txfilter,'impulse')
```



Calculate the delay through the matched filters. The group delay is half of the filter span through one filter and is, therefore, equal to the filter span for both filters. Multiply by the number of bits per symbol to get the delay in bits.

```
filtDelay = k*span;
```

Create an error rate counter System object. Set the `ReceiveDelay` property to account for the delay through the matched filters.

```
errorRate = comm.ErrorRate('ReceiveDelay',filtDelay);
```

Generate binary data.

```
x = randi([0 1],n,1);
```

Modulate the data.

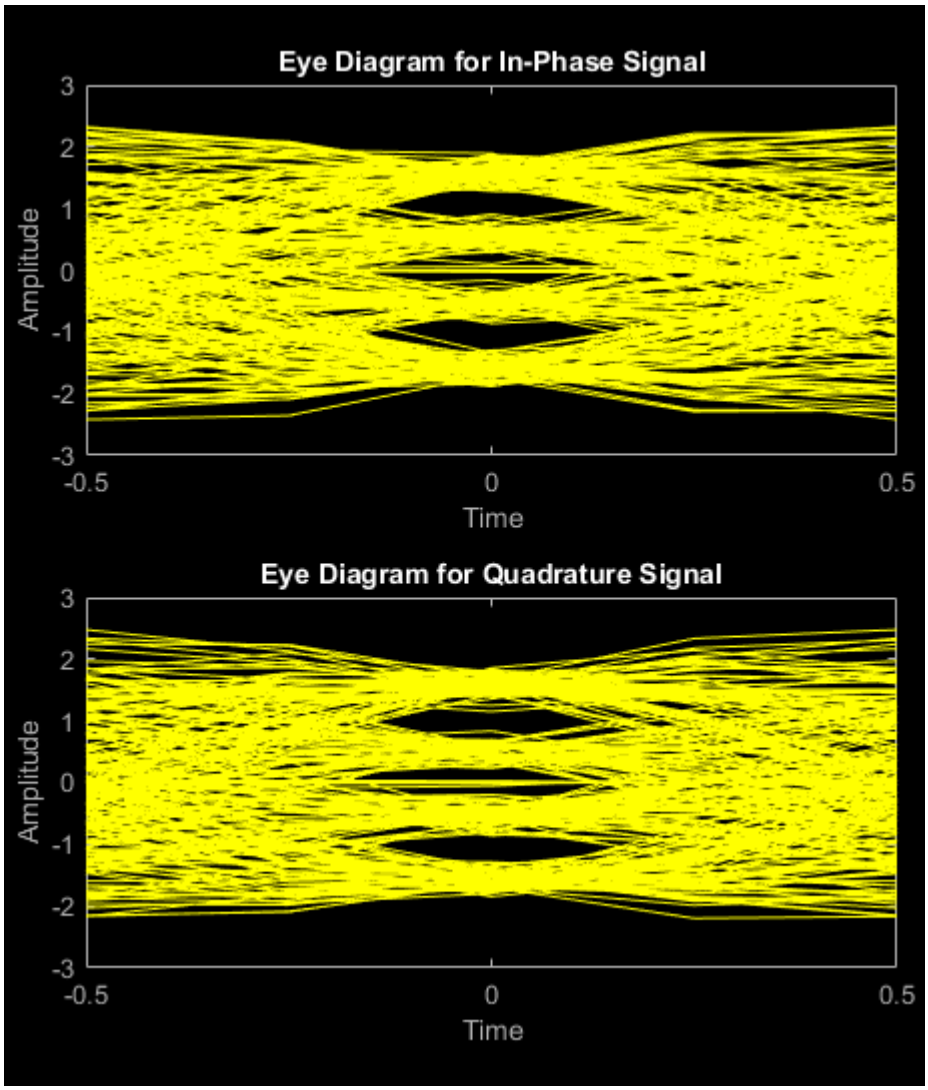
```
modSig = qammod(x,M,'InputType','bit');
```

Filter the modulated signal.

```
txSig = txfilter(modSig);
```

Plot the eye diagram of the first 1000 samples.

```
eyediagram(txSig(1:1000),nSamp)
```

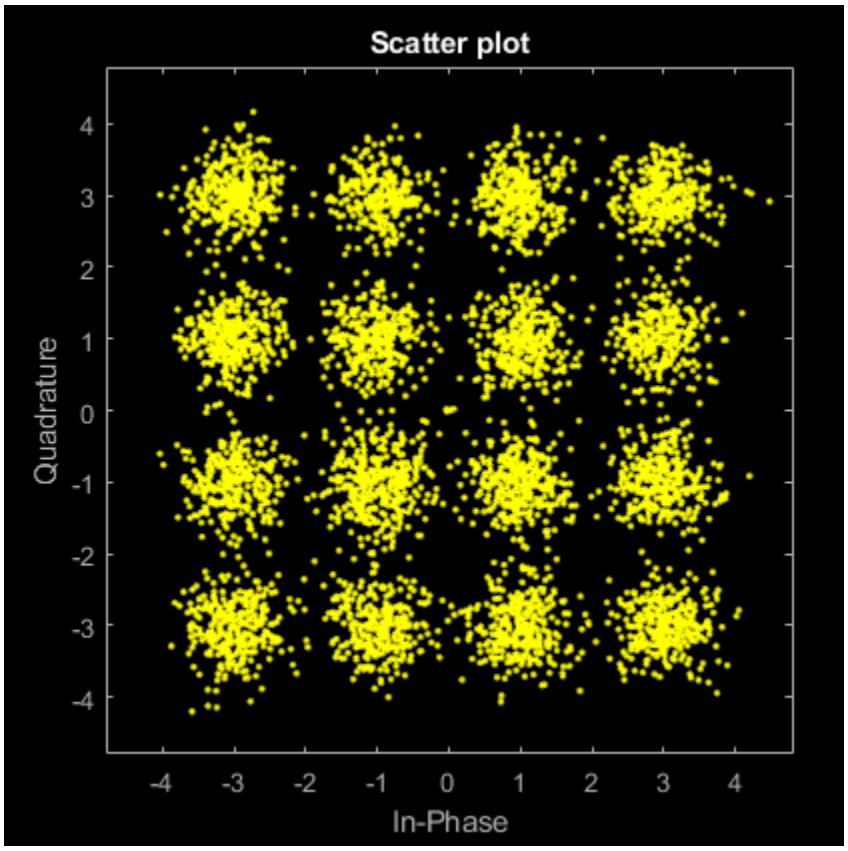


Calculate the signal-to-noise ratio (SNR) in dB given `EbNo`. Pass the transmitted signal through the AWGN channel using the `awgn` function.

```
SNR = EbNo + 10*log10(k) - 10*log10(nSamp);
noisySig = awgn(txSig,SNR,'measured');
```

Filter the noisy signal and display its scatter plot.

```
rxSig = rxfilter(noisySig);
scatterplot(rxSig)
```



Demodulate the filtered signal and calculate the error statistics. The delay through the filters is accounted for by the `ReceiveDelay` property in `errorRate` .

```
z = qamdemod(rxSig,M,'OutputType','bit');

errStat = errorRate(x,z);
fprintf('\nBER = %5.2e\nBit Errors = %d\nBits Transmitted = %d\n',...
    errStat)
```

```
BER = 1.85e-03
Bit Errors = 37
Bits Transmitted = 19960
```

Design Raised Cosine Filters Using MATLAB Functions

In this section...

Section Overview

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

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

The function returns the FIR coefficients as output.

Example Designing a Square-Root Raised Cosine Filter

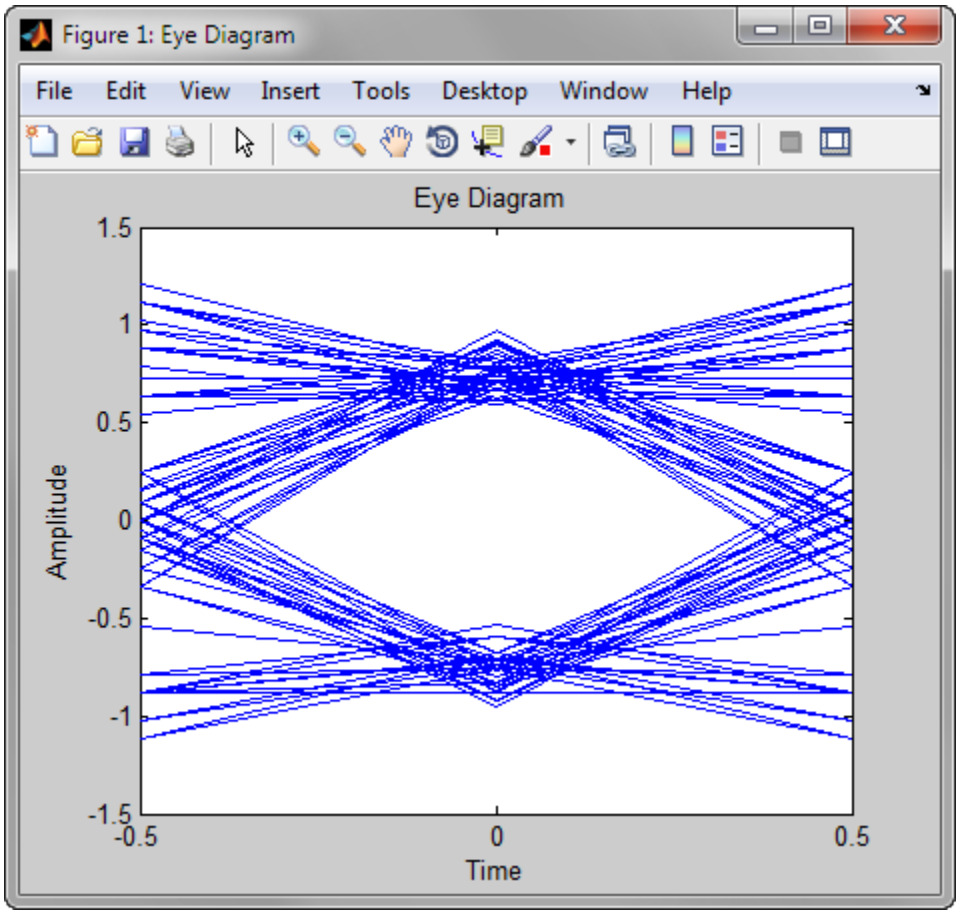
For example, the command below designs a square-root raised cosine FIR filter with a rolloff of 0.25, a filter span of 6 symbols, and an oversampling factor of 2.

```
sps = 2;
num = rcosdesign(0.25, 6, sps)

num =
  Columns 1 through 7
   -0.0265    0.0462    0.0375   -0.1205   -0.0454    0.4399    0.7558
  Columns 8 through 13
    0.4399   -0.0454   -0.1205    0.0375    0.0462   -0.0265
```

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

You can use the `upfirdn` function to filter data with a raised cosine filter generated by `rcosdesign`. The following code illustrates this usage:

```
d = 2*randi([0 1], 100, 1)-1;
f = upfirdn(d, num, sps);
eyediagram(f(7:200),sps)
```



The eye diagram shows an imperfect eye because `num` characterizes a square-root filter.

Filter Using Simulink Raised Cosine Filter Blocks

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

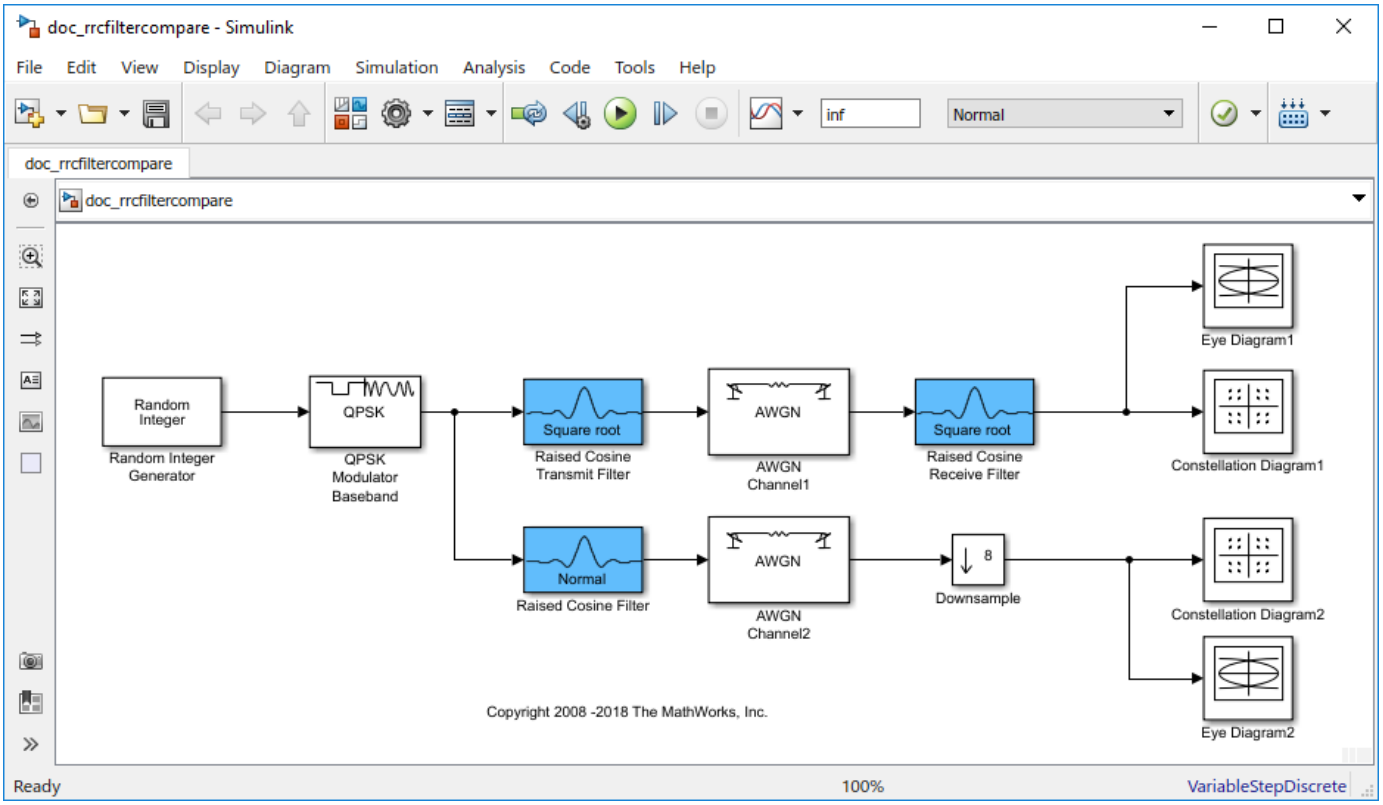
The Raised Cosine Transmit Filter and Raised Cosine Receive Filter blocks are tailored for use at the transmitter and receiver, respectively. The transmit filter outputs an upsampled (interpolated) signal, while the receive filter expects its input signal to be upsampled. The receive filter lets you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

Both raised cosine filter blocks introduce a propagation delay, as described in “Group Delay” on page 24-4.

Combining Two Square-Root Raised Cosine Filters

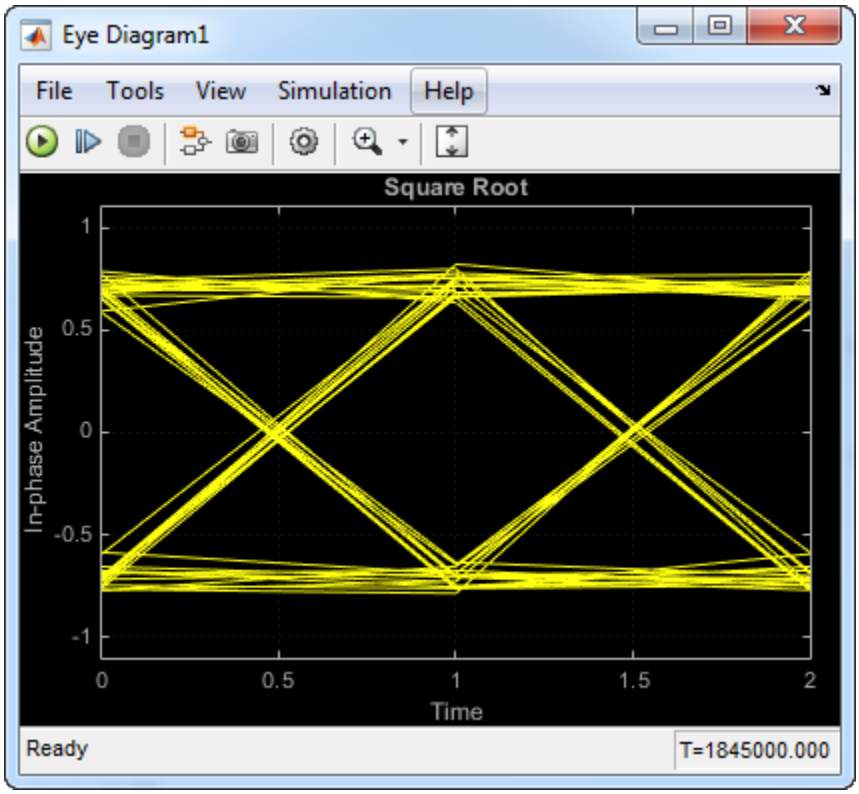
This model shows how to split the filtering equally between the transmitter's filter and the receiver's filter by using a pair of square root raised cosine filters.

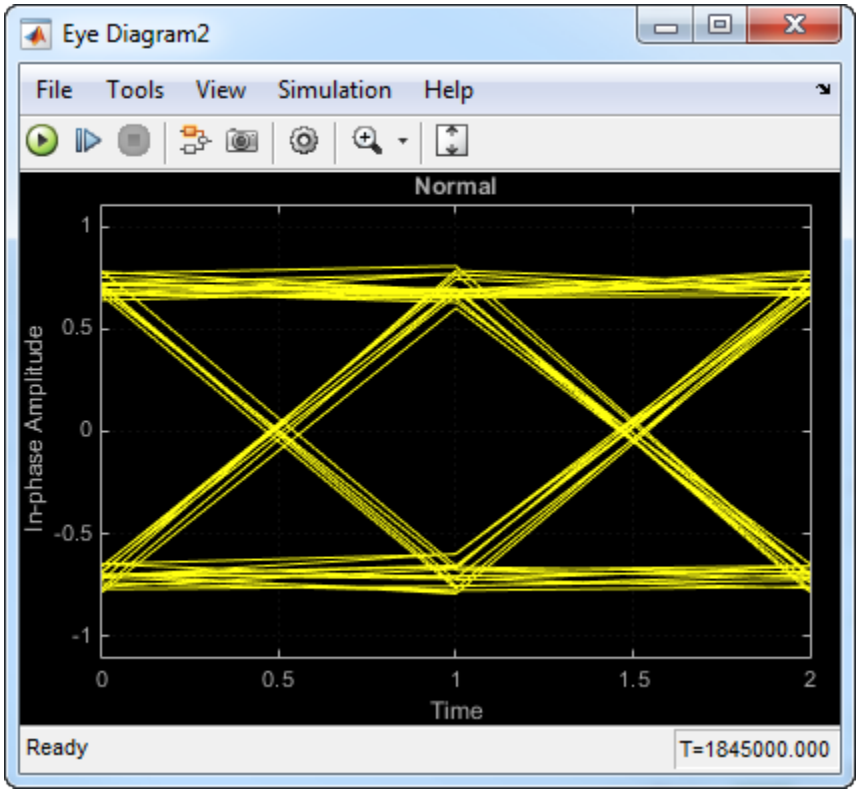
The use of two matched square root raised cosine filters is equivalent to a single normal raised cosine filter. To see this illustrated, type `doc_rrcfiltercompare` at the MATLAB command line to open the model.

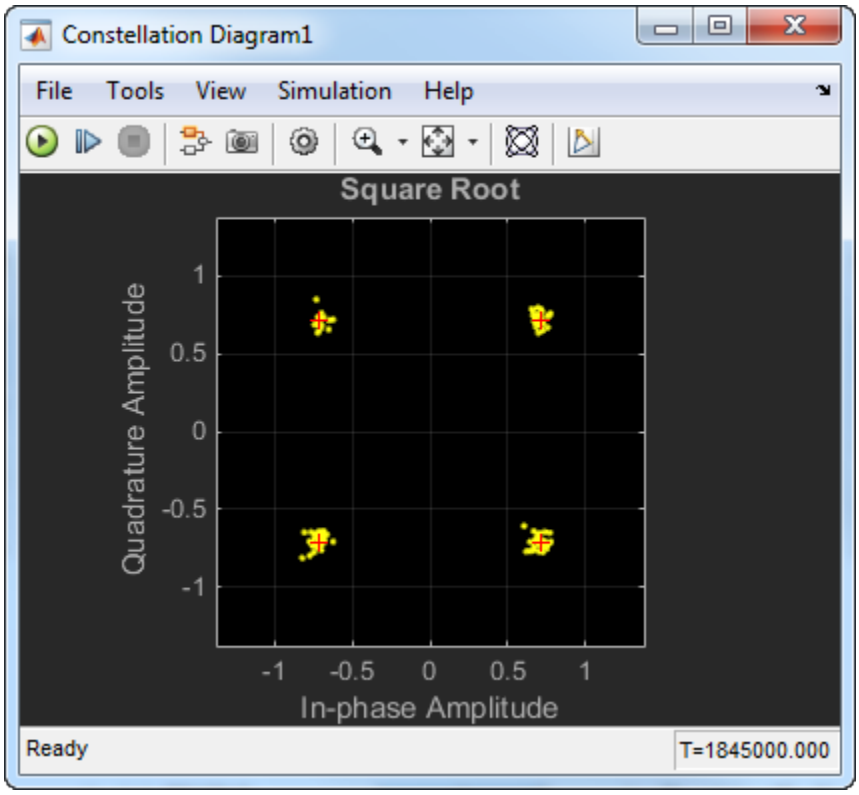


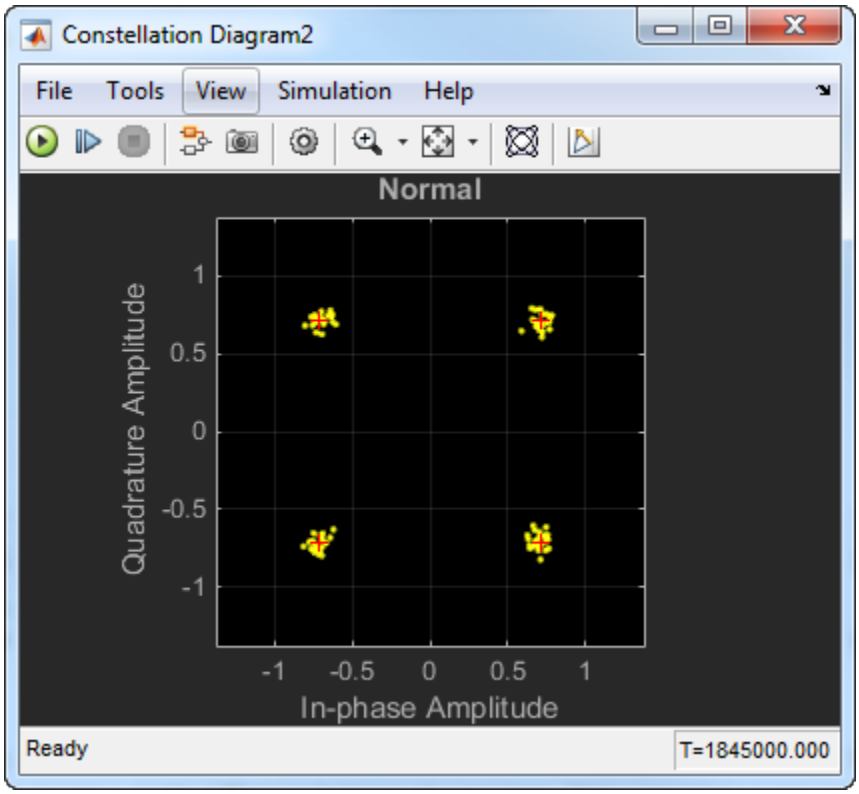
The filters share the same span and use the same number samples per symbol but the filters on the upper path have a square root shape while the filter on the lower path has the normal shape.

Run the model and observe the eye and constellation diagrams. The performance is nearly identical for the two methods. Note that the limited impulse response of practical square root raised cosine filters causes a slight difference between the response of two cascaded square root raised cosine filters and the response of one raised cosine filter.



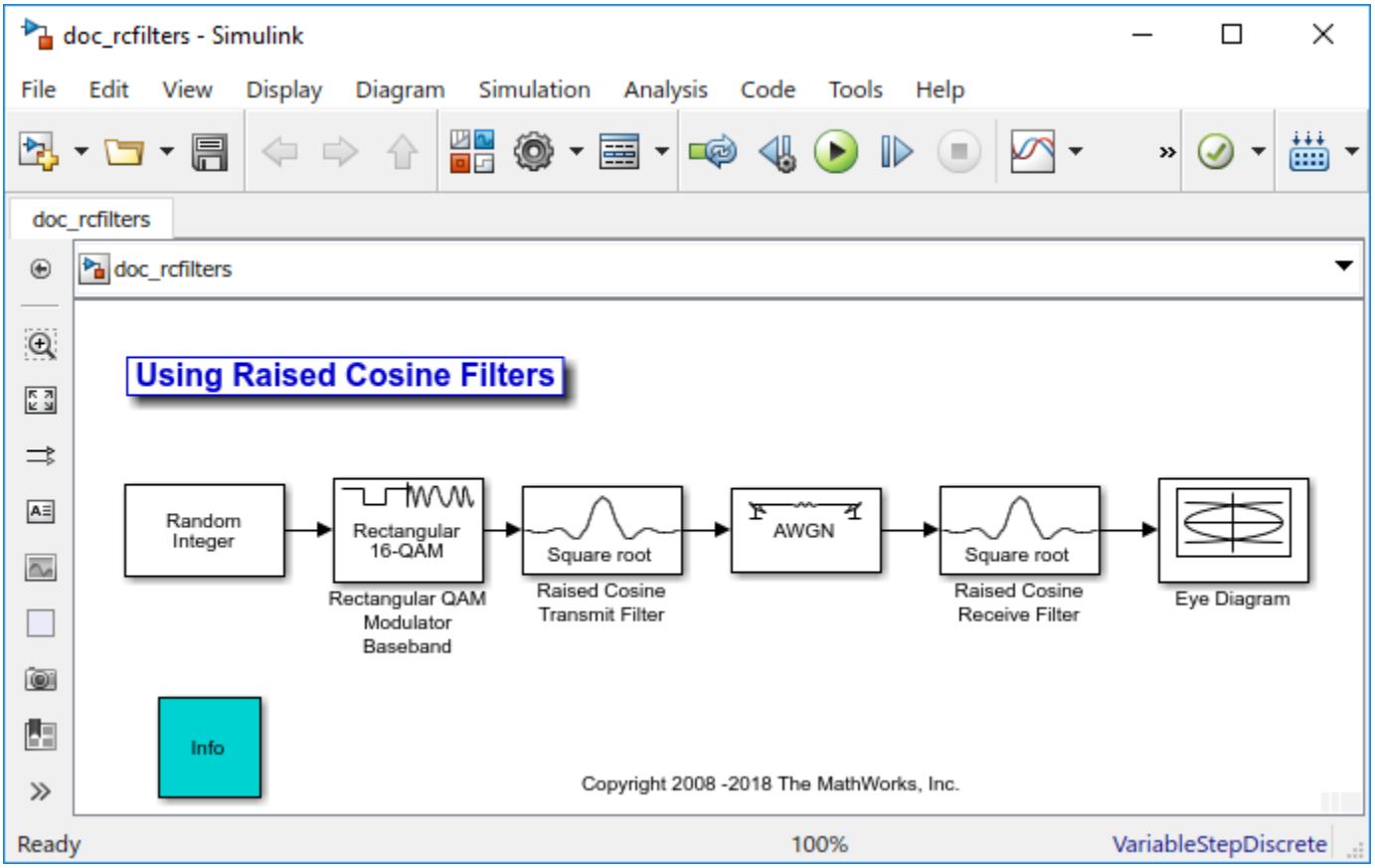

Eye Diagram2
File Tools View Simulation Help
Normal
In-phase Amplitude
1
0.5
0
-0.5
-1
0
0.5
1
1.5
2
Time
Ready
T=1845000.000


Constellation Diagram1
File Tools View Simulation Help
Square Root
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=1845000.000


Constellation Diagram2
File
Tools
View
Simulation
Help
Normal
Quadrature Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
In-phase Amplitude
Ready
T=1845000.000

Design Raised Cosine Filters in Simulink

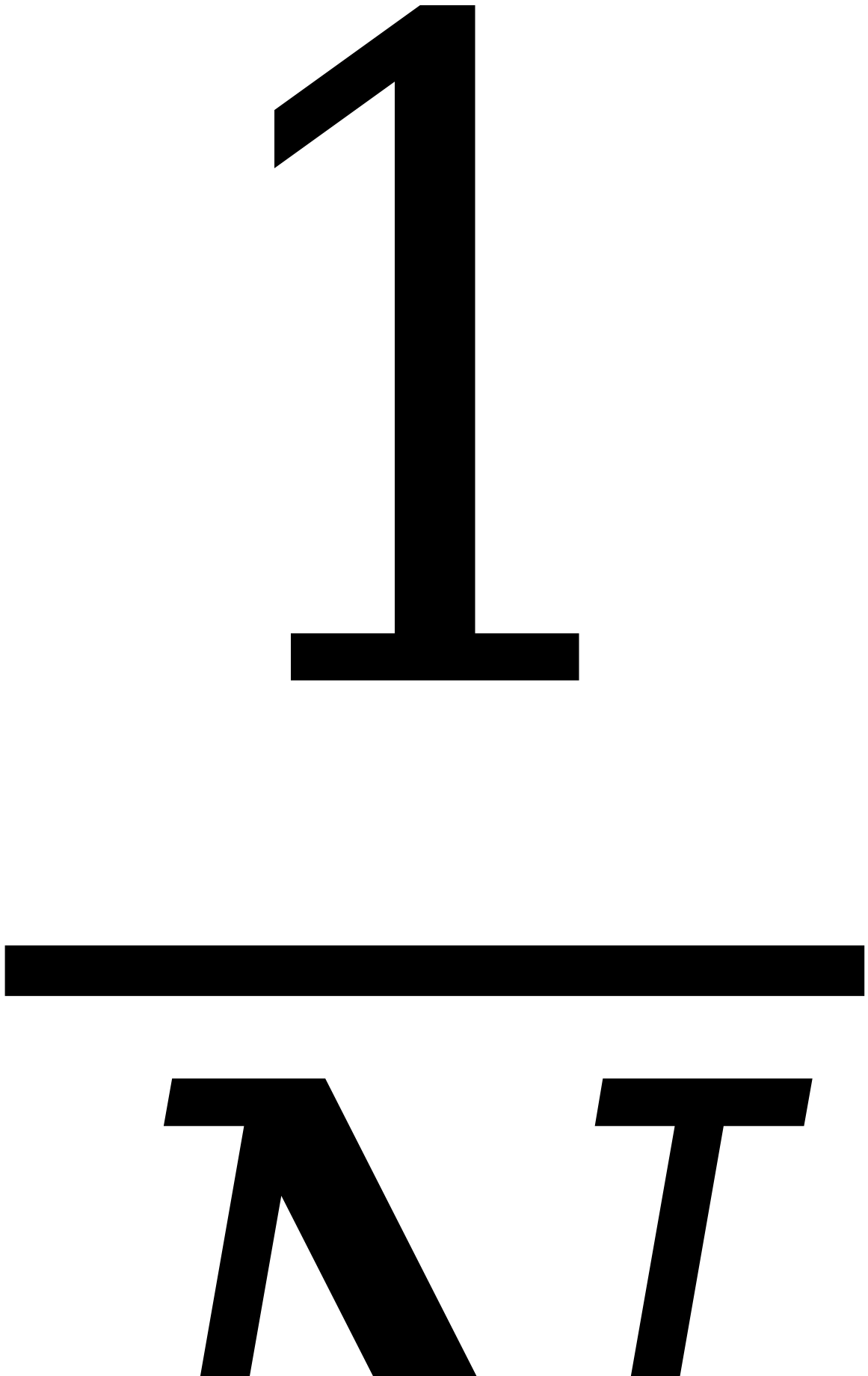
This example illustrates a typical setup in which a transmitter uses a square root raised cosine filter to perform pulse shaping and the corresponding receiver uses a square root raised cosine filter as a matched filter. The example plots an eye diagram from the filtered received signal.



To open the model, enter `doc_rcfilters` at the MATLAB command line. The following is a summary of the block parameters used in the model:

- Random Integer Generator, in the Random Data Sources sublibrary of the Comm Sources library:
  - **M-ary number** is set to `16`.
  - **Sample time** is set to `1/100`.
  - **Frame-based outputs** is selected.
  - **Samples per frame** is set to `100`.
- Rectangular QAM Modulator Baseband, in the AM sublibrary of the Digital Baseband sublibrary of Modulation:
  - **Normalization method** is set to `Peak Power`.
  - **Peak power** is set to `1`.
- Raised Cosine Transmit Filter, in the Comm Filters library:

  - **Filter span in symbols** is set to `8`.
  - **Rolloff factor** is set to `0.2`
- AWGN Channel, in the Channels library:
  - **Mode** is set to `Signal to noise ratio (SNR)`.
  - **SNR** is set to `40`.
  - **Input signal power** is set to `0.0694`. The power gain of a square-root raised cosine transmit filter is



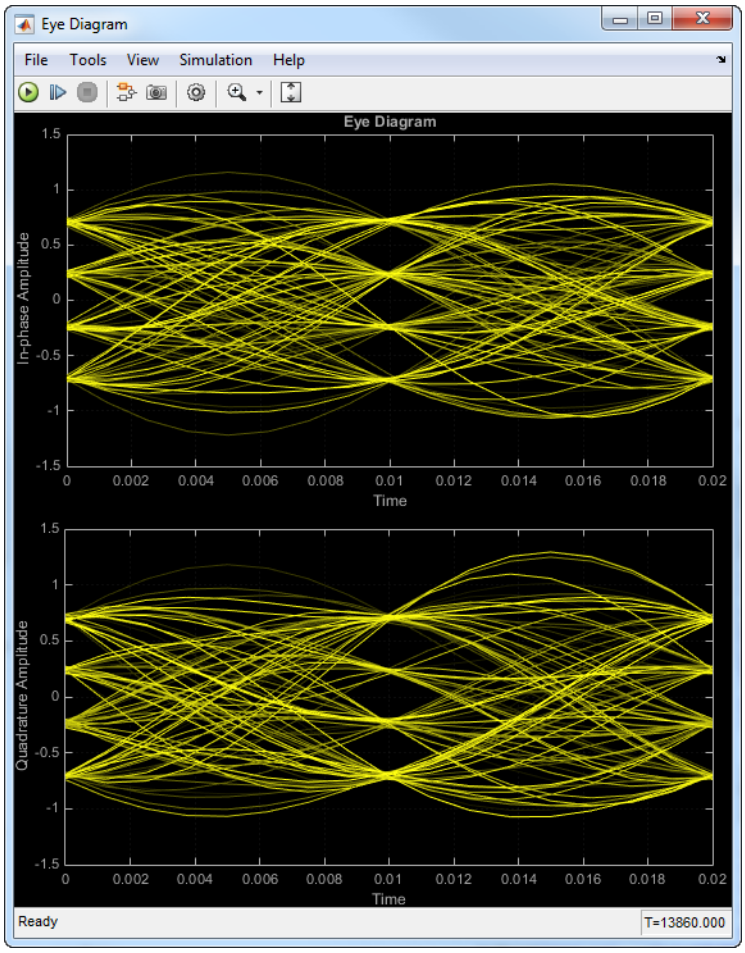
, where *N* represents the upsampling factor of the filter. The input signal power of filter is `0.5556`. Because the **Peak power** of the 16-QAM Rectangular modulator is set to 1 watt, it translates to an average power of 0.5556. Therefore, the output signal power of filter is

$$\frac{0.5556}{8} = 0.0694$$

.

- Raised Cosine Receive Filter, in the Comm Filters library:
  - **Filter span in symbols** is set to `8`.
  - **Rolloff factor** is set to `0.2`.
- Eye Diagram Scope, in the Comm Sinks library:
  - **Symbols per trace** is set to `2`.
  - **Traces to display** is set to `100`.

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



The large signal-to-noise ratio in this example produces an eye diagram with large eye openings. If you decrease the **SNR** parameter in the AWGN Channel block, the eyes in the diagram will close more.

Reduce ISI Using Raised Cosine Filtering

Employ raised cosine filtering to reduce inter-symbol interference (ISI) that results from a nonlinear amplifier.

Initialize a simulation variable for modulation order.

```
M = 16; % Modulation order
```

Create square root raised cosine filter objects.

```
txfilter = comm.RaisedCosineTransmitFilter;
rxfilter = comm.RaisedCosineReceiveFilter;
```

Create a memoryless nonlinearity System object to introduce nonlinear behavior to the modulated signal. Using name-value pairs, set the `Method` property to `Saleh model` to emulate a high power amplifier.

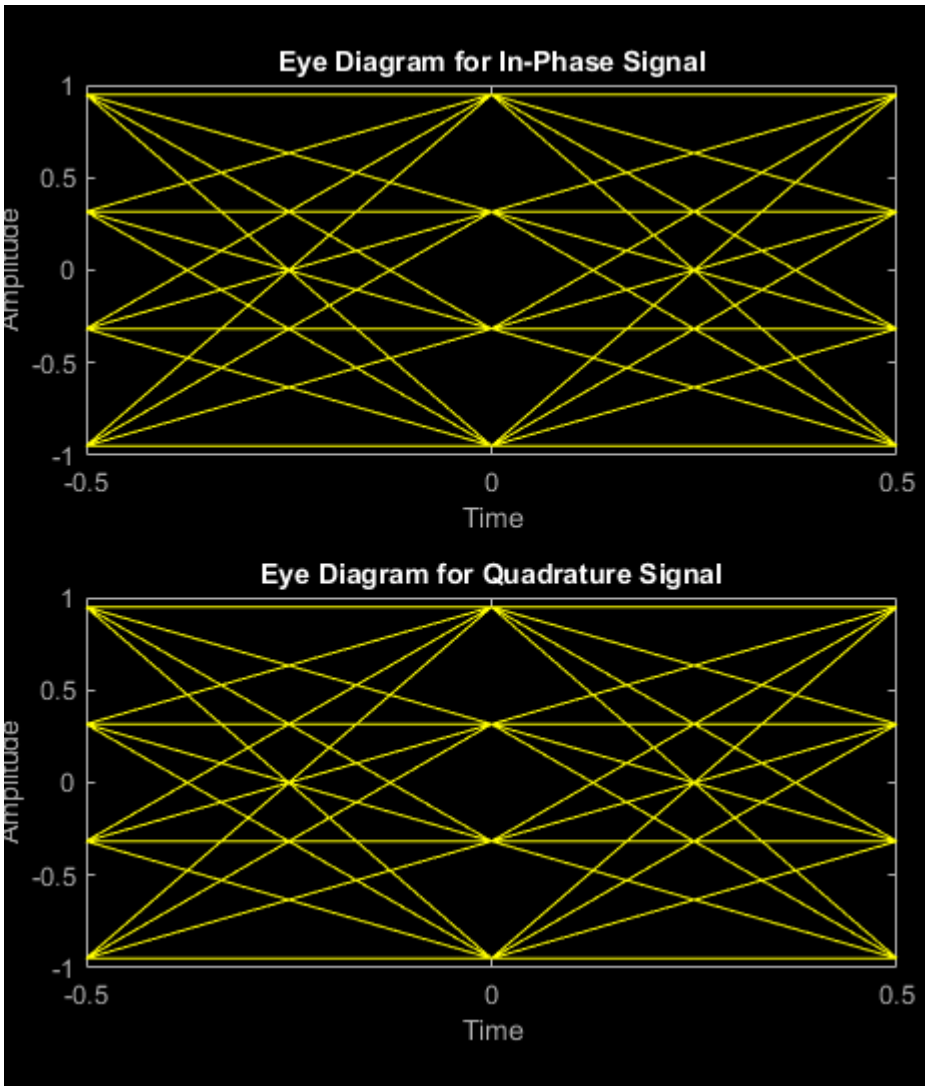
```
hpa = comm.MemorylessNonlinearity('Method','Saleh model', ...
    'InputScaling',-10,'OutputScaling',0);
```

Generate random integers and apply 16-QAM modulation.

```
x = randi([0 M-1],1000,1);
modSig = qammod(x,M,'UnitAveragePower',true);
```

Plot the eye diagram of the modulated signal. At time 0, there are three distinct "eyes" for 16-QAM modulation.

```
eyediagram(modSig,2)
```

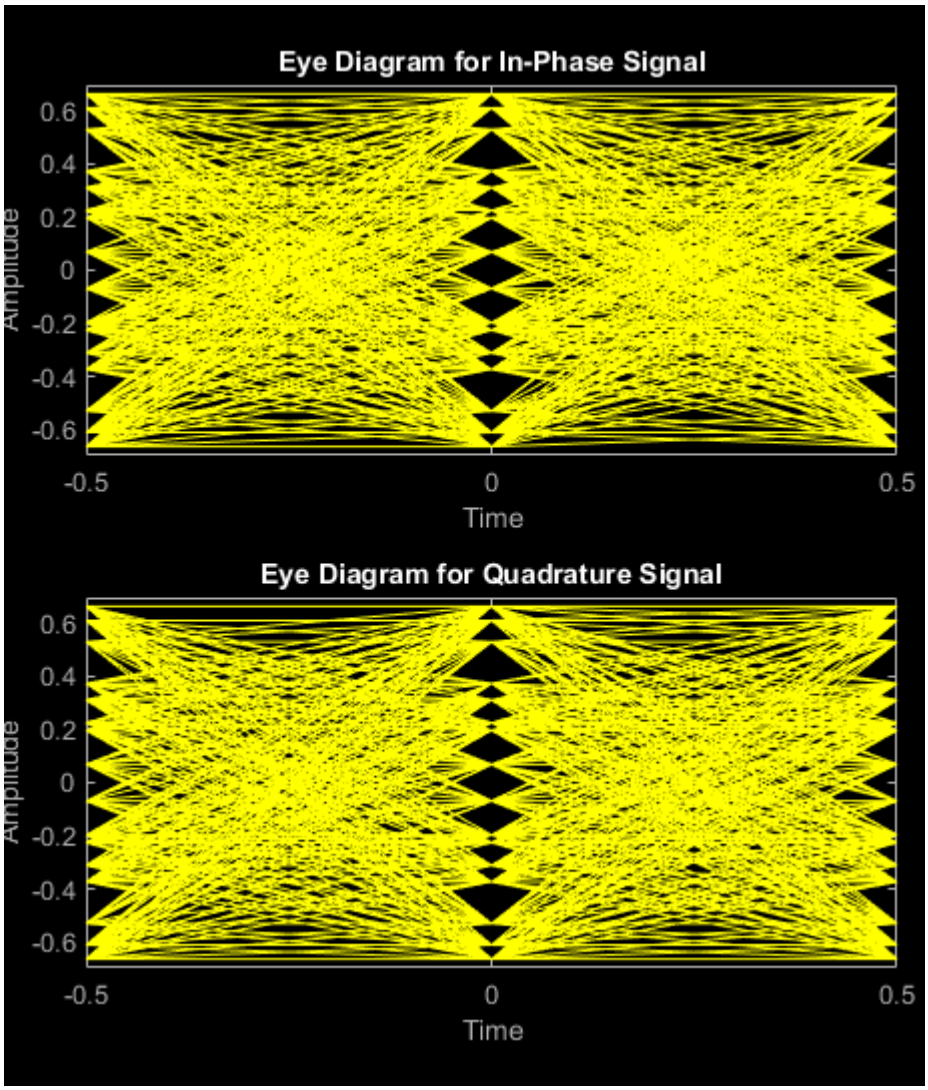


Amplify the modulated signal using `hpa`.

```
txSigNoFilt = hpa(modSig);
```

Plot the eye diagram of the amplified signal without RRC filtering. At time 0, there are multiple eyes. This is a result of inter-symbol interference from the nonlinear amplifier.

```
eyediagram(txSigNoFilt,2)
```



Filter the modulated signal using the RRC transmit filter.

```
filteredSig = txfilter(modSig);
```

Release `hpa` and amplify the filtered signal. The `release` function is needed because the input signal dimensions change due to filter interpolation.

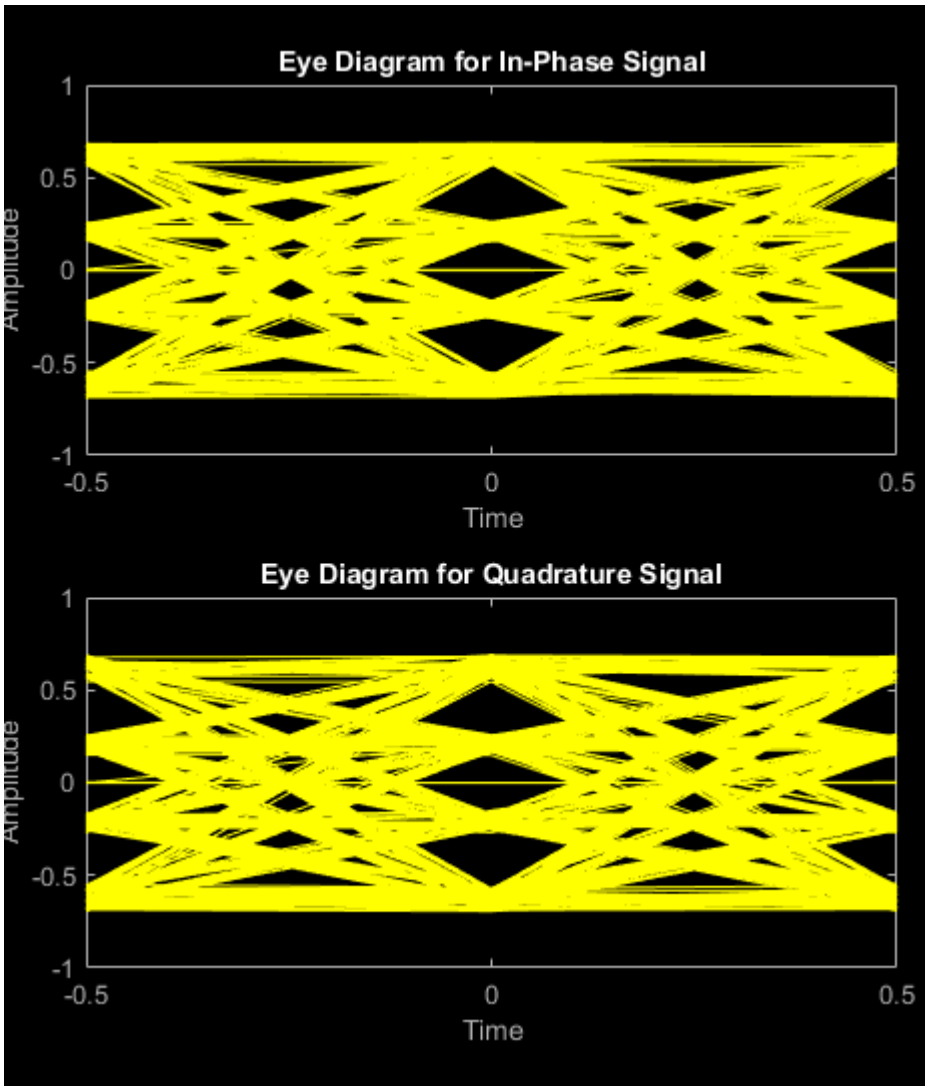
```
release(hpa)
txSig = hpa(filteredSig);
```

Filter `txSig` using the RRC matched receive filter.

```
rxSig = rxfilter(txSig);
```

Plot the eye diagram of the signal after the application of the receive filter. There are once again three distinct eyes as the matched RRC filters mitigate ISI.

```
eyediagram(rxSig,2)
```


Eye Diagram for In-Phase Signal
Amplitude
1
0.5
0
-0.5
-1
-0.5
0
0.5
Time
Eye Diagram for Quadrature Signal
Amplitude
1
0.5
0
-0.5
-1
-0.5
0
0.5
Time

Find Delay for Encoded and Filtered Signal

Determine the delay for a convolutionally encoded and filtered link. Use the delay to accurately determine the number of bit errors.

Create a QPSK modulator and demodulator pair. Specify the objects to operate on bits.

```
qpskmod = comm.QPSKModulator('BitInput',true);
qpskdemod = comm.QPSKDemodulator('BitOutput',true);
```

Create a raised cosine transmit and receive filter pair.

```
txfilt = comm.RaisedCosineTransmitFilter;
rxfilt = comm.RaisedCosineReceiveFilter;
```

Create a convolutional encoder and Viterbi decoder pair.

```
convEnc = comm.ConvolutionalEncoder;
vitDec = comm.ViterbiDecoder('InputFormat','Hard');
```

Generate random binary data. Convolutionally encode the data.

```
txData = randi([0 1],1000,1);
encData = convEnc(txData);
```

Modulate the encoded data. Pass the modulated data through the raised cosine transmit filter.

```
modSig = qpskmod(encData);
txSig = txfilt(modSig);
```

Pass the filtered signal through an AWGN channel.

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

Filter and then demodulate the received signal.

```
filtSig = rxfilt(rxSig);
demodSig = qpskdemod(filtSig);
```

Decode the demodulated data.

```
rxData = vitDec(demodSig);
```

Find the delay between the transmitted and received binary data by using the `finddelay` function.

```
td = finddelay(txData,rxData)
```

```
td = 44
```

Confirm that the computed delay matches the expected delay, which is equal to the sum of the group delay of the matched filters and the traceback depth of the Viterbi decoder.

```
tdexpected = (txfilt.FilterSpanInSymbols + rxfilt.FilterSpanInSymbols)/2 + ...
    vitDec.TracebackDepth;
isequal(td,tdexpected)
```

```
ans = logical
   1
```

Calculate the number of bit errors by discarding the last `td` bits from the transmitted sequence and discarding the first `td` bits from the received sequence.

```
numErrors = biterr(txData(1:end-td),rxData(td+1:end))
```

```
numErrors = 0
```

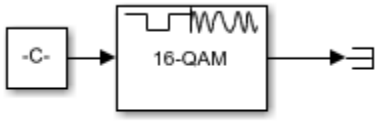
Visual Analysis

View Constellation of Modulator Block

This example shows how to visualize constellations by clicking the **View Constellation** button on the mask of linear modulator block. These linear modulator blocks provide the capability to visualize a signal constellation from the block mask.

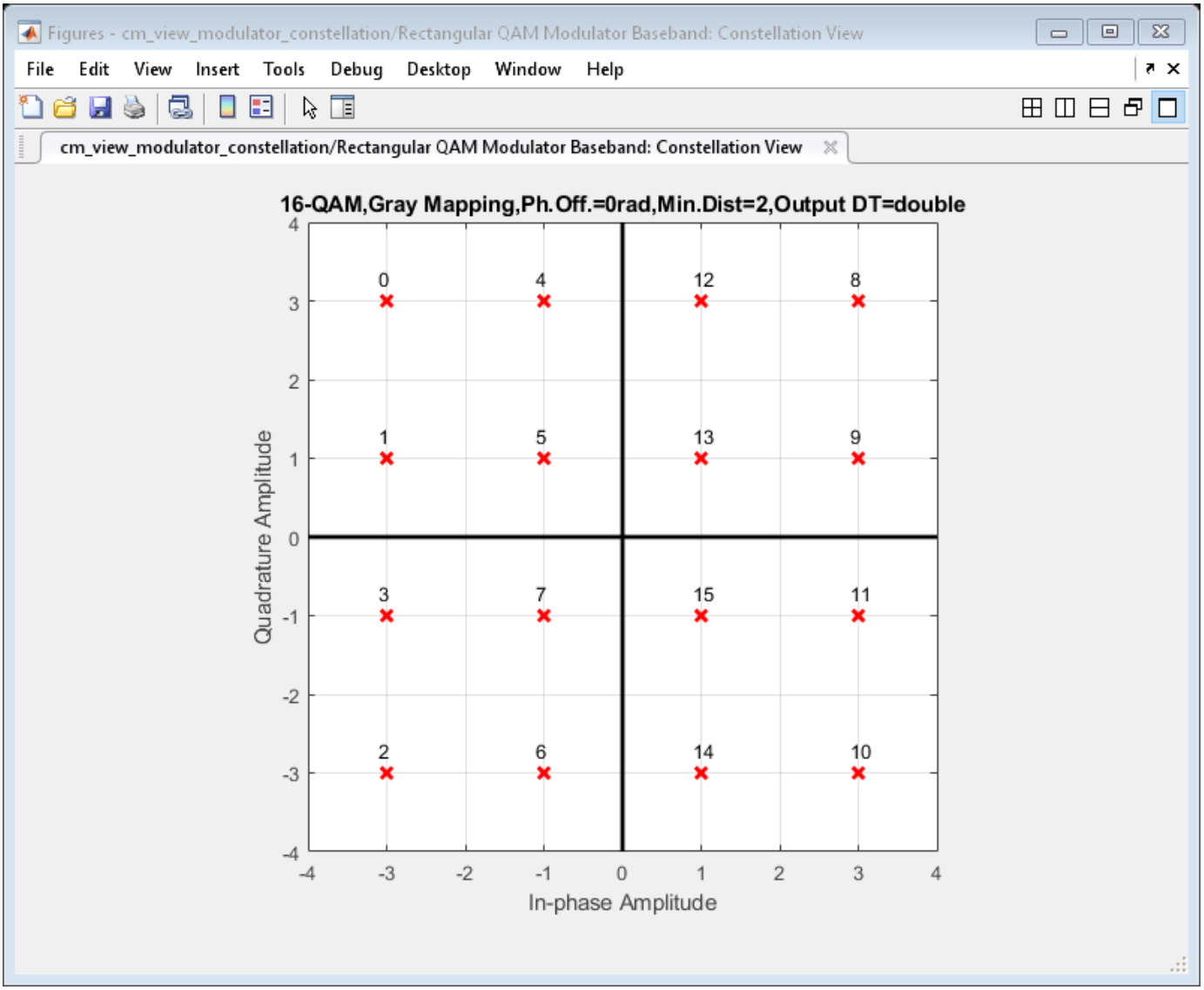
- BPSK Modulator Baseband
- QPSK Modulator Baseband
- M-PSK Modulator Baseband
- M-PAM Modulator Baseband
- Rectangular QAM Modulator Baseband
- General QAM Modulator Baseband

For these linear modulator blocks, clicking **View Constellation** on the block mask plots the signal constellation using the applied block settings. Use the `cm_view_modulator_constellation` model to create constellation figures by clicking **View Constellation**. This model uses the Rectangular QAM Modulator Baseband block with the modulation order set to the workspace variable `M`. The value for `M` is specified in the `PreLoadFcn` callback function. To view the callback, select MODELING, SETUP, Model Settings, and then Model Properties. In the Model Properties window, select Callbacks, and then `PreLoadFcn`.



Modulator Configuration and Signal Constellation

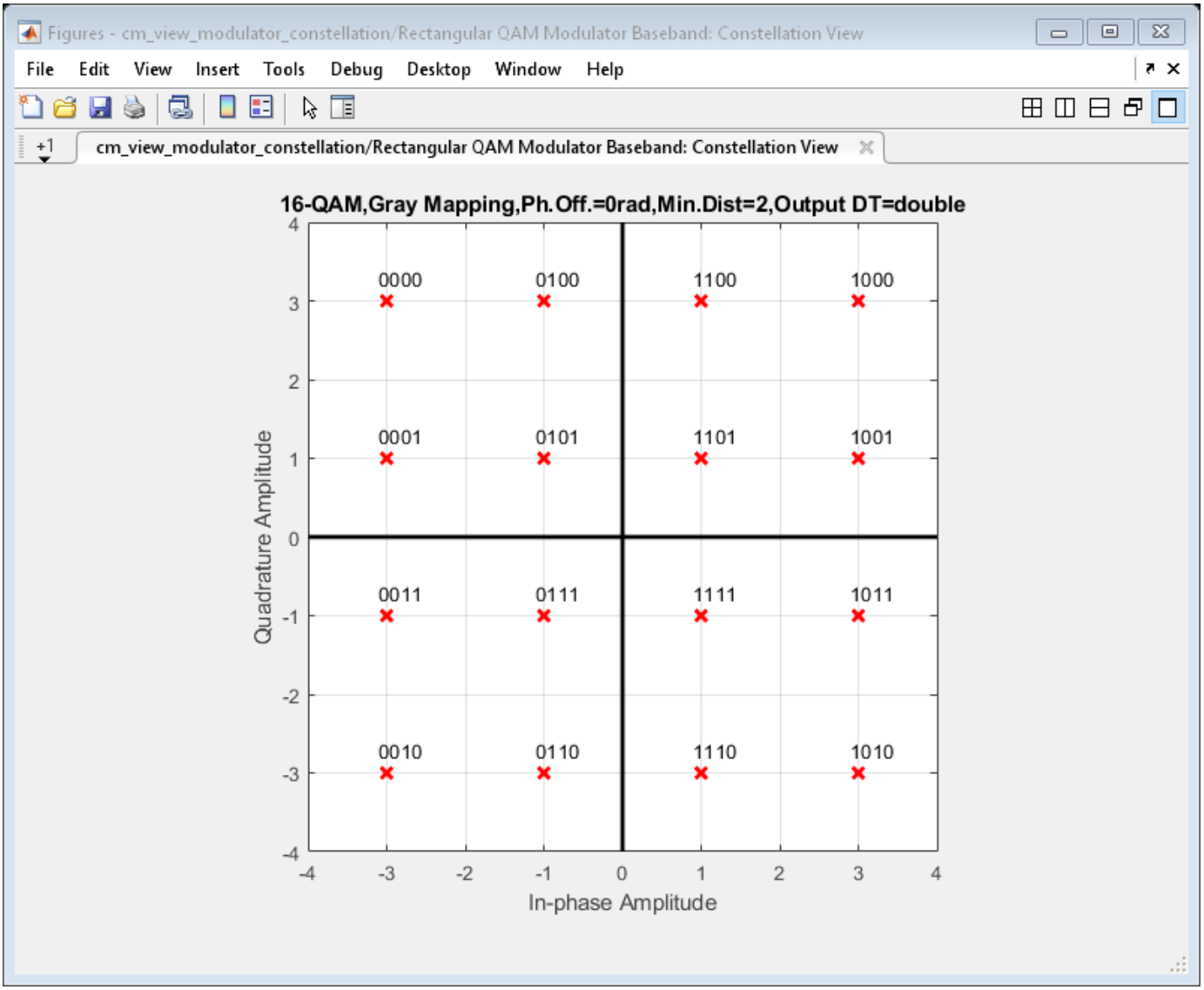
When you click **View Constellation** on the modulator block mask, the constellation diagram opens in a MATLAB® `figure` window. The title of the plot indicates the values of significant parameters.



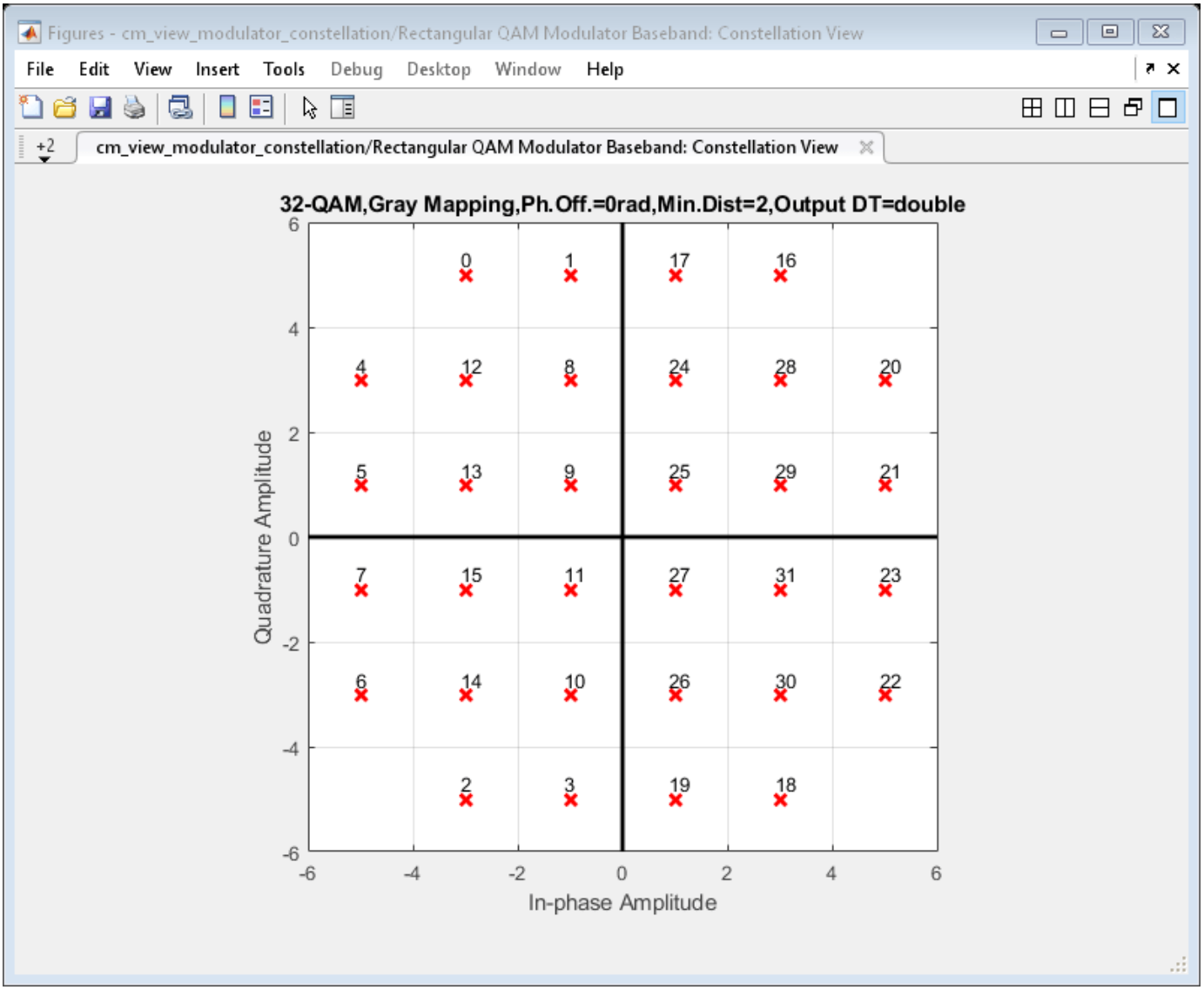
This constellation plot figure shows the default signal constellation for the Rectangular QAM Modulator Baseband block:

- 16-QAM modulation scheme
- Gray constellation mapping
- 0-degree phase offset
- Minimum distance of 2 between two constellation points
- Double precision data type signal
- Integer symbol representation

From the block mask, set the **Input type** parameter to `Bit`, and the **Constellation ordering** parameter to `Binary`. To view the constellation for the updated configuration, click **Apply** before clicking **View Constellation**. The updated plot indicates binary constellation mapping and displays the bit representation for the symbols.



Since the modulation order setting of the Rectangular QAM Modulator Baseband block is `M`, the value can be updated by using the variable `M` defined in the MATLAB workspace. Set `M = 32` in the MATLAB workspace. The modulation order setting updates the model workspace in Simulink®. Click **View Constellation** to show the 32-QAM constellation.



To capture a figure for future use, save the figure before closing the model. When you close the Simulink model, all of the constellation figures closes as well.

See Also

Blocks

BPSK Modulator Baseband | QPSK Modulator Baseband | M-PSK Modulator Baseband | M-PAM Modulator Baseband | Rectangular QAM Modulator Baseband | General QAM Modulator Baseband

Related Examples

- “Plot Signal Constellations” on page 25-6
- “Eye Diagram Analysis” on page 25-10

Plot Signal Constellations

In this section...

Create 16-PSK Constellation Diagram

This example shows how to plot a PSK constellation having 16 points.

Set the parameters for 16-PSK modulation with no phase offset and binary symbol mapping.

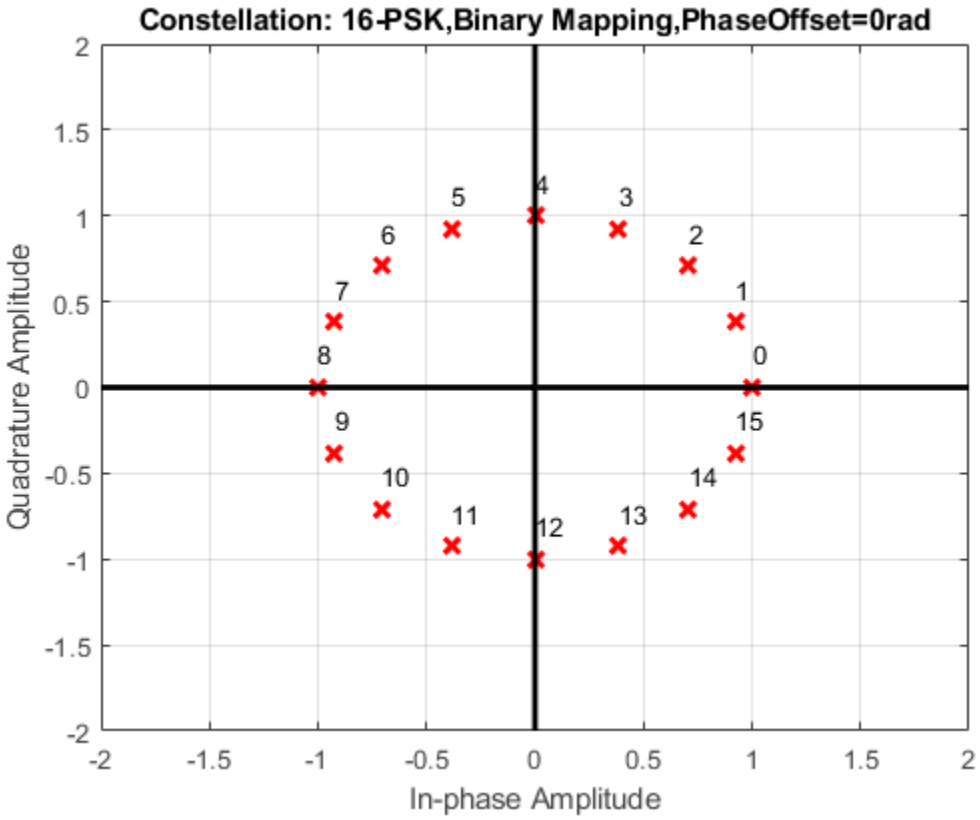
```
M = 16;              % Modulation alphabet size
phOffset = 0;        % Phase offset
symMap = 'binary';   % Symbol mapping (either 'binary' or 'gray')
```

Construct the modulator object.

```
pskModulator = comm.PSKModulator(M,phOffset,'SymbolMapping',symMap);
```

Plot the constellation.

```
constellation(pskModulator)
```



Create 32-QAM Constellation Diagram

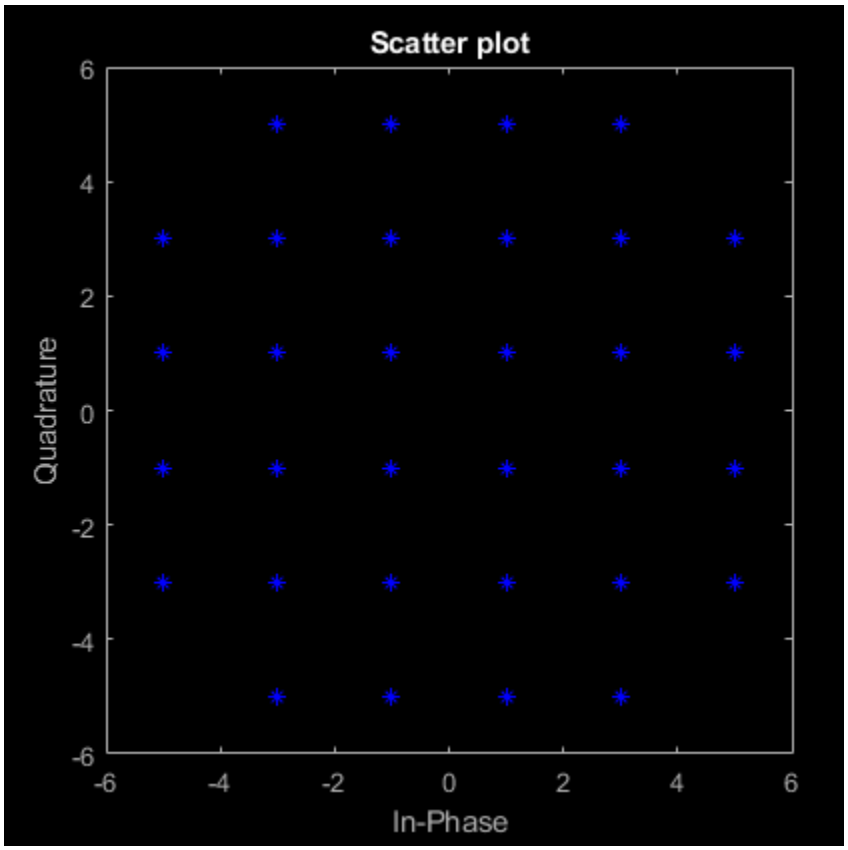
This example shows how to plot a QAM constellation having 32 points.

Use the `qammod` function to generate the 32-QAM symbols with binary symbol ordering.

```
M = 32;
data = 0:M-1;
sym = qammod(data,M,'bin');
```

Plot the constellation. Label the order of the constellation symbols.

```
scatterplot(sym,1,0,'b*');
for k = 1:M
    text(real(sym(k))-0.4,imag(sym(k))+0.4,num2str(data(k)));
end
axis([-6 6 -6 6])
```



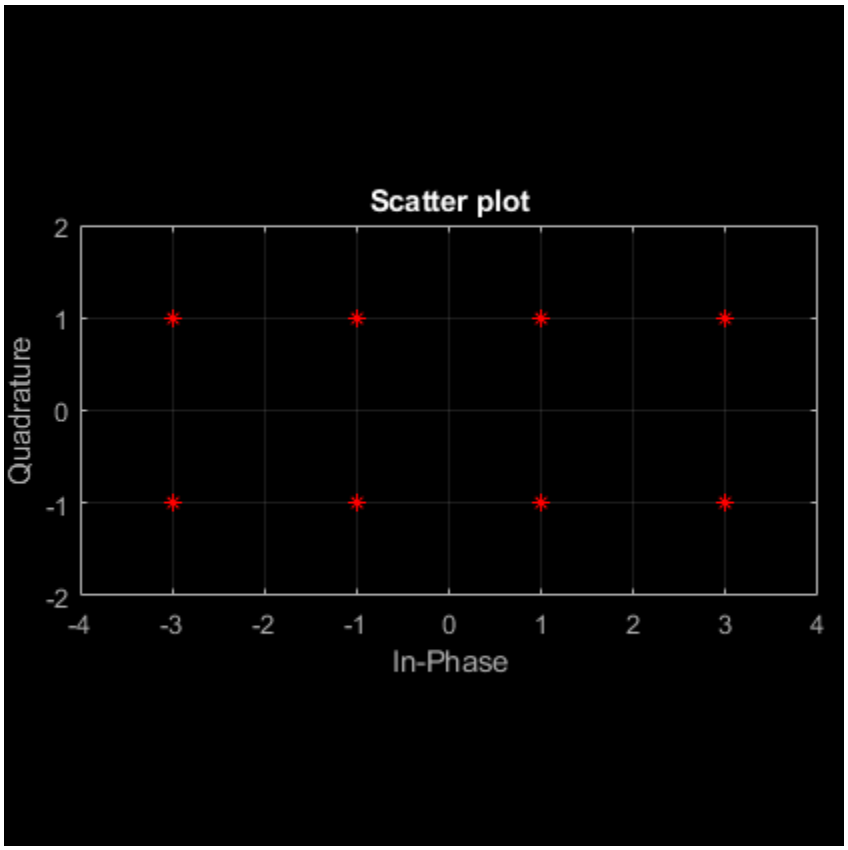
Create 8-QAM Gray Coded Constellation Diagram

Use the qammod function to generate the 8-QAM symbols with Gray symbol ordering. Note that Gray coding is the default symbol mapping for the `qammod` function.

```
M = 8;
data = 0:M-1;
sym = qammod(data,M);
```

Plot the constellation. Label the order of the constellation symbols.

```
scatterplot(sym,1,0,'r*');
grid on
for k = 1:M
    text(real(sym(k))-0.4,imag(sym(k))+0.4,num2str(data(k)));
end
axis([-4 4 -2 2])
```



Plot a Triangular Constellation for QAM

This example shows how to plot a customized QAM reference constellation.

Describe the constellation.

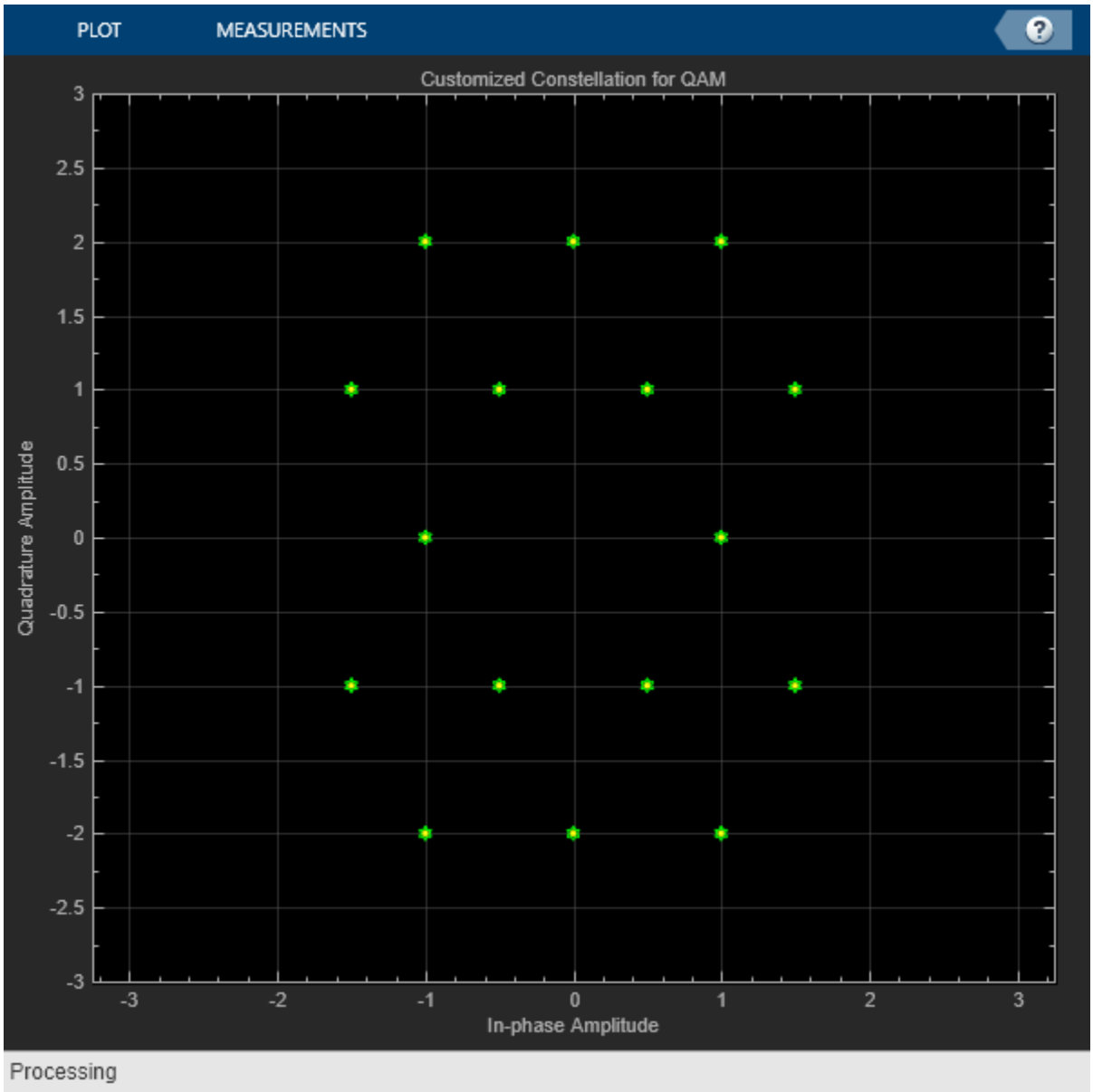
```
inphase = [1/2 -1/2 1 0 3/2 -3/2 1 -1];
quadr = [1 1 0 2 1 1 2 2];
inphase = [inphase; -inphase];
inphase = inphase(:);
quadr = [quadr; -quadr];
quadr = quadr(:);
refConst = inphase + 1i*quadr;
```

Construct a constellation diagram System object using name-value pairs to specify the title, the axes limits, the reference marker type, and the reference marker color.

```
constDiagram = comm.ConstellationDiagram('Title','Customized Constellation for QAM', ...
    'XLimits',[-3 3],'YLimits',[-3 3], ...
    'ReferenceConstellation',refConst, ...
    'ReferenceMarker','*','ReferenceColor',[0 1 0]);
```

Plot the customized constellation.

```
constDiagram(refConst)
```



See Also

“View Constellation of Modulator Block” on page 25-2

Eye Diagram Analysis

In digital communications, an eye diagram provides a visual indication of how noise might impact system performance.

Use the Eye Diagram Scope block to examine the eye diagram of signals.

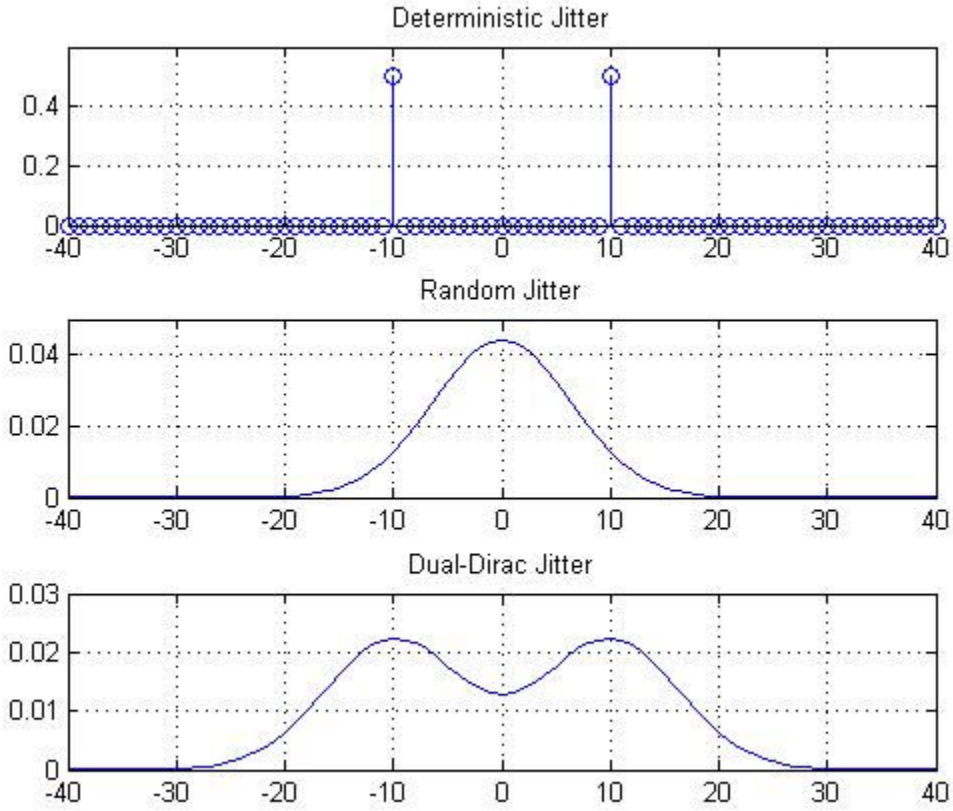
You can obtain the following measurements on an eye diagram:

- Amplitude Measurements
  - Eye Amplitude
  - Eye Crossing Amplitude
  - Eye Crossing Percentage
  - Eye Height
  - Eye Level
  - Eye SNR
  - Quality Factor
  - Vertical Eye Opening
- Time Measurements
  - Deterministic Jitter
  - Eye Crossing Time
  - Eye Delay
  - Eye Fall Time
  - Eye Rise Time
  - Eye Width
  - Horizontal Eye Opening
  - Peak-to-Peak Jitter
  - Random Jitter
  - RMS Jitter
  - Total Jitter

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.

The deterministic jitter, horizontal eye opening, quality factor, random jitter, and vertical eye opening measurements utilize a dual-Dirac algorithm. *Jitter* is the deviation of a signal's timing event from its intended (ideal) occurrence in time [1]. 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). The DJ PDF comprises two delta functions, one at μ_L and one at μ_R. The RJ PDF is assumed to be Gaussian with zero mean and variance σ.

The *Total Jitter (TJ) PDF* is the convolution of these two PDFs, which is composed of two Gaussian curves with variance σ and mean values μ_L and μ_R.



The dual-Dirac model is described in [5] in more detail. The amplitude of the two Dirac functions may not be the same. In such a case, the analyze method estimates these amplitudes, ρ_L and ρ_R.

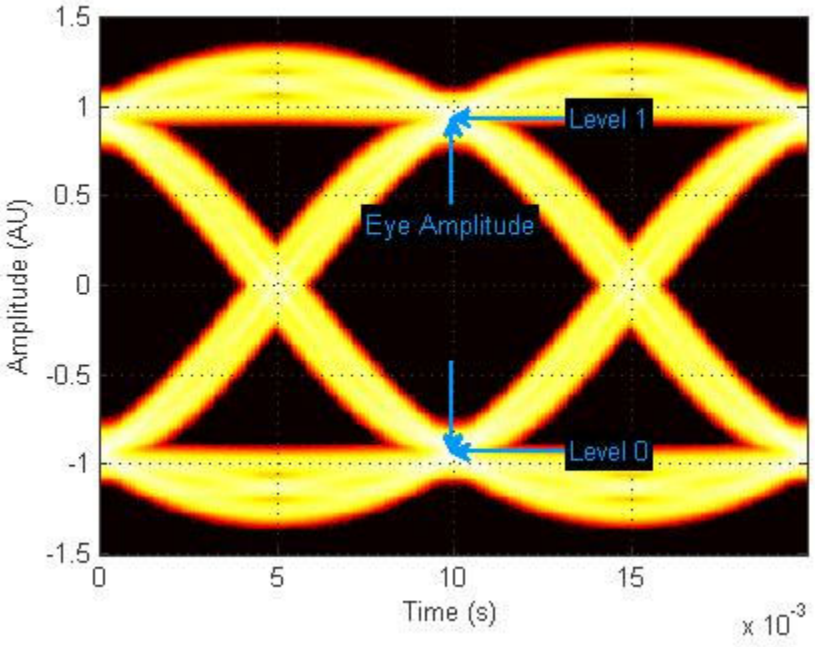
Amplitude Measurements

You can use the vertical histogram to obtain a variety of amplitude measurements. For complex signals, measurements are done on both in-phase and the quadrature components, unless otherwise specified.

Note For amplitude measurements, at least one bin per vertical histogram must reach 10 hits before the measurement is taken, ensuring higher accuracy.

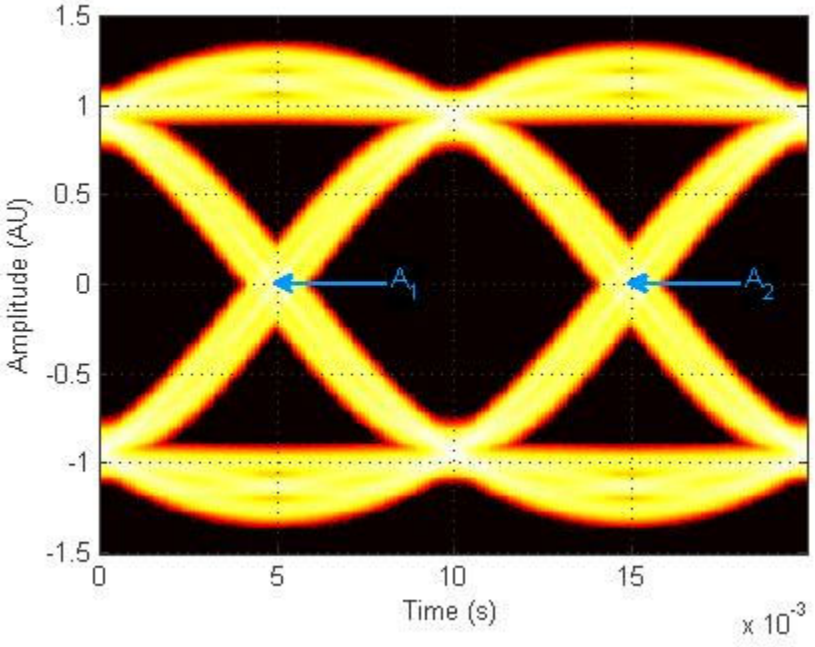
Eye Amplitude (EyeAmplitude)

Eye Amplitude, measured in Amplitude Units (AU), is defined as the distance between two neighboring eye levels. For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye amplitude is the difference of these two values.

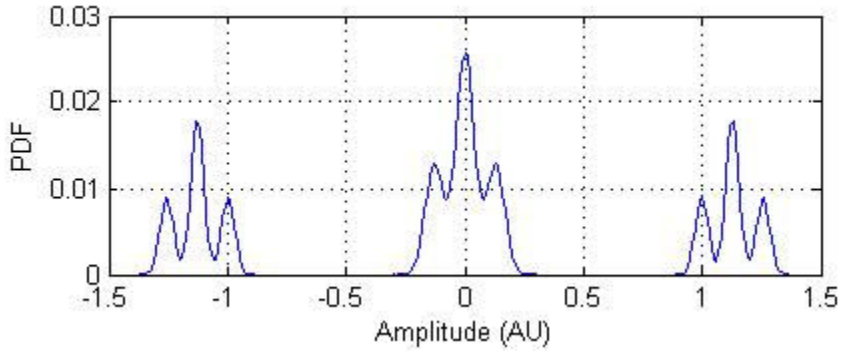


Eye Crossing Amplitude (EyeCrossingLevel)

Eye crossing amplitudes are the amplitude levels at which the eye crossings occur, measured in Amplitude Units (AU). The analyze method calculates this value using the mean value of the vertical histogram at the crossing times [3].



The next figure shows the vertical histogram at the first eye crossing time.



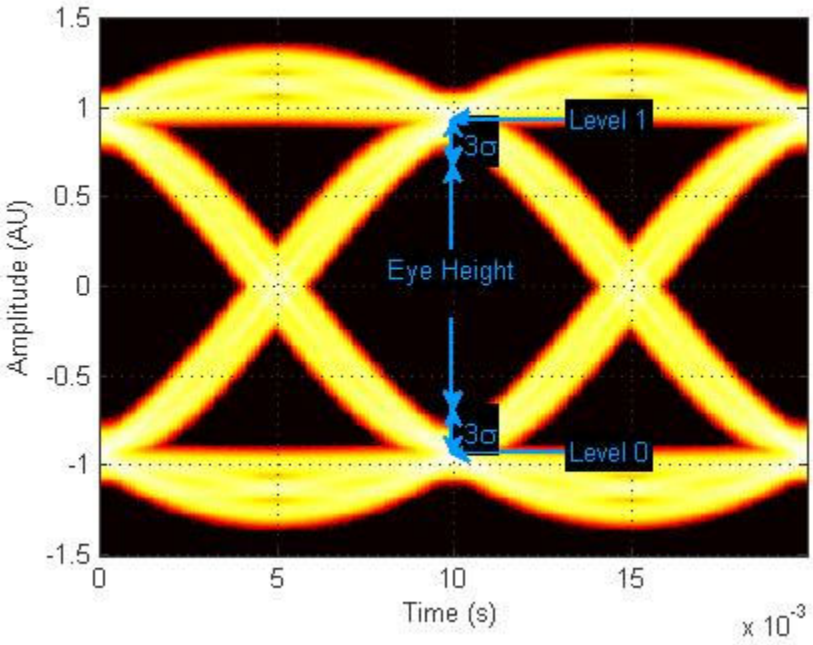
Eye Crossing Percentage (EyeOpeningVer)

Eye Crossing Percentage is the location of the eye crossing levels as a percentage of the eye amplitude.

Eye Height (EyeHeight)

Eye Height, measured in Amplitude Units (AU), is defined as the 3σ distance between two neighboring eye levels.

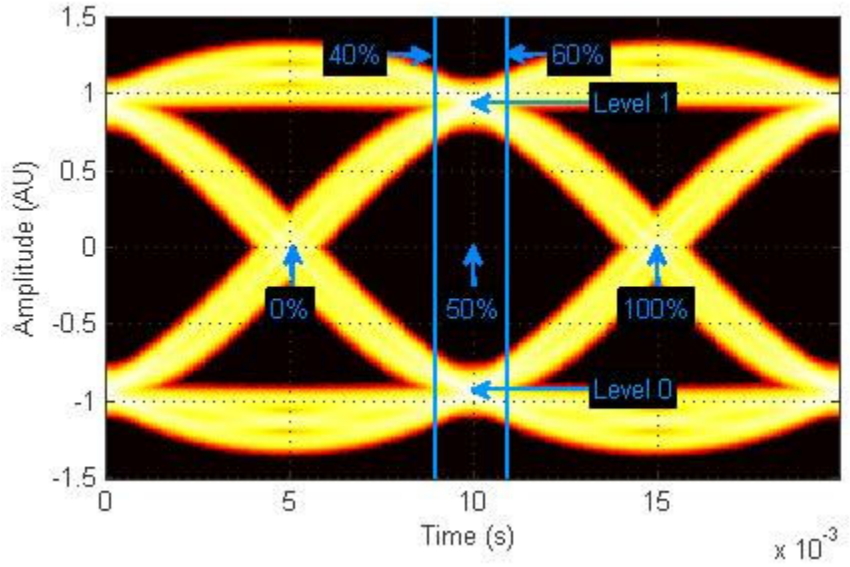
For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye height is the difference of the two 3σ points. The 3σ point is defined as the point that is three standard deviations away from the mean value of a PDF.



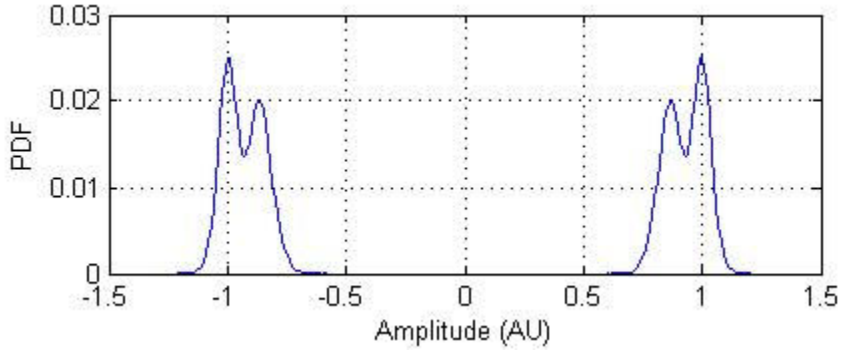
Eye Level (EyeLevel)

Eye Level is the amplitude level used to represent data bits, measured in Amplitude Units (AU).

For an ideal NRZ signal, there are two eye levels: +A and –A. The analyze method calculates eye levels by estimating the mean value of the vertical histogram in a window around the EyeDelay, which is also the 50% point between eye crossing times [3]. The width of this window is determined by the EyeLevelBoundary property of the eye measurement setup object.



The analyze method calculates the mean value of all the vertical histograms within the eye level boundaries. The mean vertical histograms show two distinct PDFs, one for each eye level.



Eye SNR (EyeSNR)

Eye signal-to-noise ratio is defined as the ratio of the eye amplitude to the sum of the standard deviations of the two eye levels. It can be expressed as:

$$\text{SNR} = \frac{L_1 - L_0}{\sigma_1 + \sigma_0}$$

where L_1 and L_0 represent eye level 1 and 0, respectively, and σ_1 and σ_2 are the standard deviation of eye level 1 and 0, respectively.

For an NRZ signal, eye level 1 corresponds to the high level, and the eye level 0 corresponds to low level.

Quality Factor (QualityFactor)

The analyze method calculates *Quality Factor* the same way as the eye SNR. However, instead of using the mean and standard deviation values of the vertical histogram for L_1 and σ_1, the analyze method uses the mean and standard deviation values estimated using the dual-Dirac method. For more detail, see dual-Dirac section in [2].

Vertical Eye Opening (EyeOpeningVer)

Vertical Eye Opening is defined as the vertical distance between two points on the vertical histogram at EyeDelay that corresponds to the BER value defined by the BERThreshold property of the eye measurement setup object. The analyze method calculates this measurement taking into account the random and deterministic components using a dual-Dirac model [5] (see the Dual Dirac Section). A typical BER value for the eye opening measurements is 10^{-12}, which approximately corresponds to the 7σ point assuming a Gaussian distribution.

Time Measurements

You can use the horizontal histogram of an eye diagram to obtain a variety of timing measurements.

Note For time measurements, at least one bin per horizontal histogram must reach 10 hits before the measurement is taken.

Deterministic Jitter (JitterDeterministic)

Deterministic Jitter is the deterministic component of the jitter. You calculate it using the tail mean value, which is estimated using the dual-Dirac method as follows [5]:

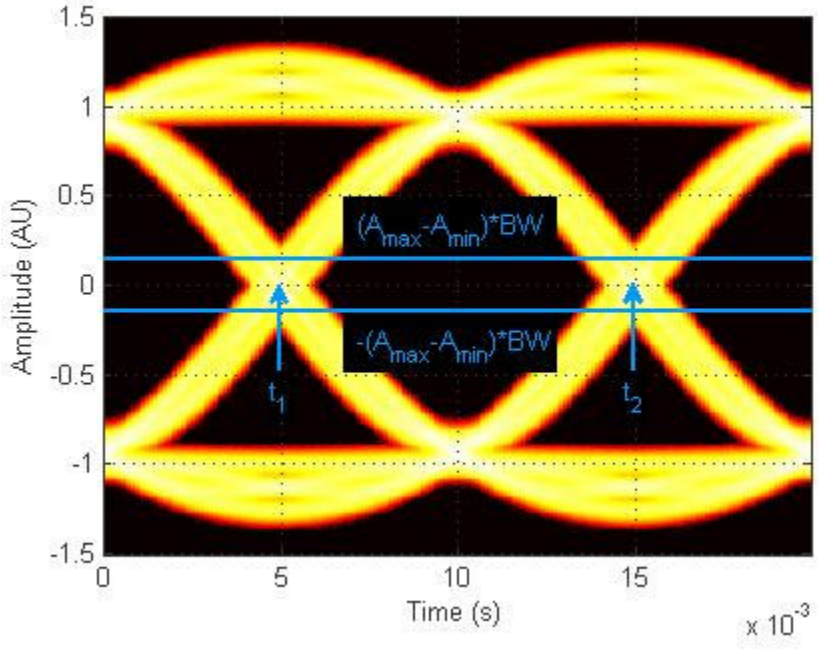
$$DJ = \mu_L - \mu_R$$

where μ_L and μ_R are the mean values returned by the dual-Dirac algorithm.

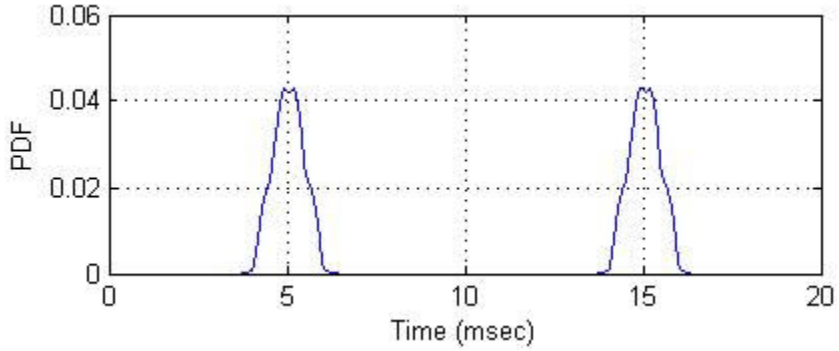
Eye Crossing Time (EyeCrossingTime)

Eye crossing times are calculated as the mean of the horizontal histogram for each crossing point, around the reference amplitude level. This value is measured in seconds. The mean value of all the horizontal PDFs is calculated in a region defined by the CrossingBandWith property of the eye measurement setup object.

The region is from $-A_{total} * BW$ to $+A_{total} * BW$, where A_{total} is the total amplitude range of the eye diagram (i.e., $A_{total} = A_{max} - A_{min}$) and BW is the crossing band width.



Because this example assumes two symbols per trace, the average PDF in this region indicate there are two crossing points.

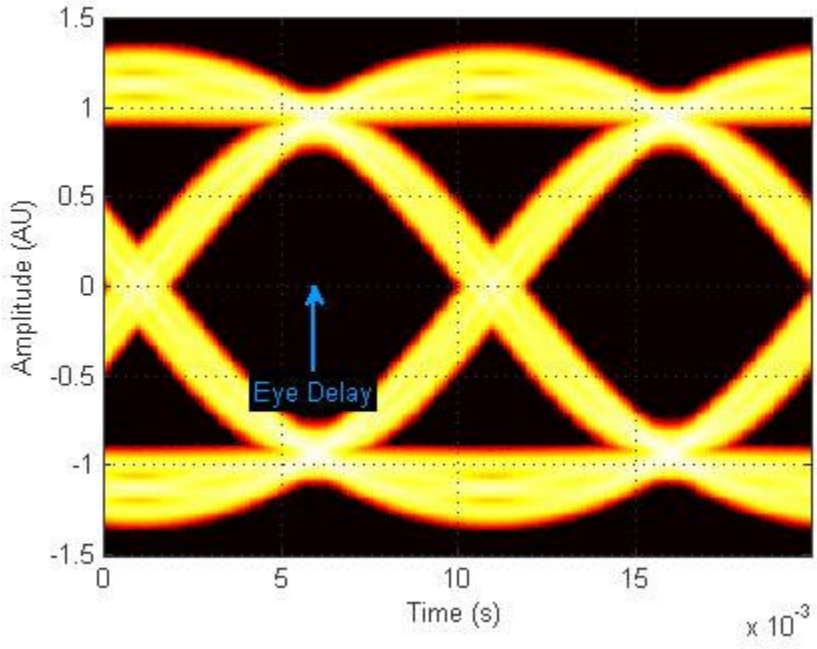


Note When an eye crossing time measurement falls within the [-0.5/Fs, 0) seconds interval, the time measurement wraps to the end of the eye diagram, i.e., the measurement wraps by 2*Ts seconds (where Ts is the symbol time). For a complex signal case, the analyze method issues a warning if the crossing time measurement of the in-phase branch wraps while that of the quadrature branch does not (or vice versa).

To avoid the 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.

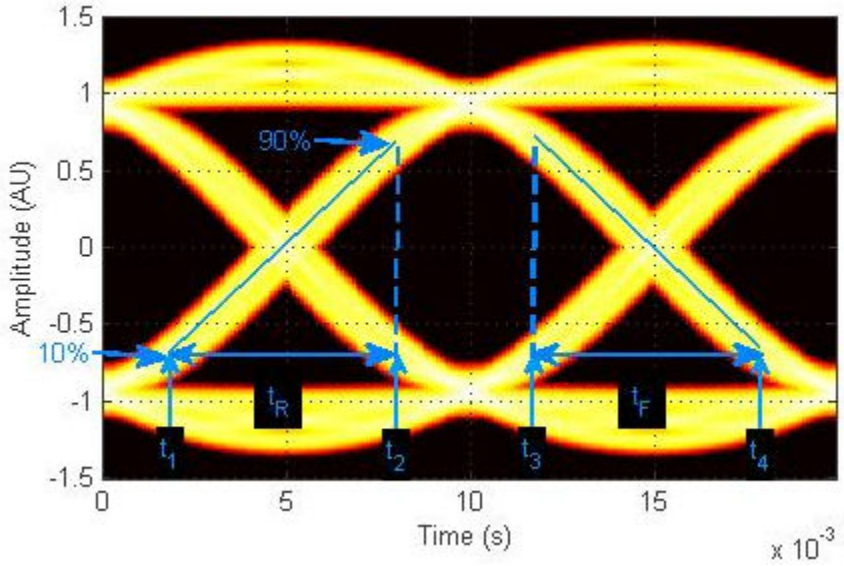
Eye Delay (EyeDelay)

Eye Delay is the distance from the midpoint of the eye to the time origin, measured in seconds. The analyze method calculates this distance using the crossing time. For a symmetric signal, EyeDelay is also the best sampling point.



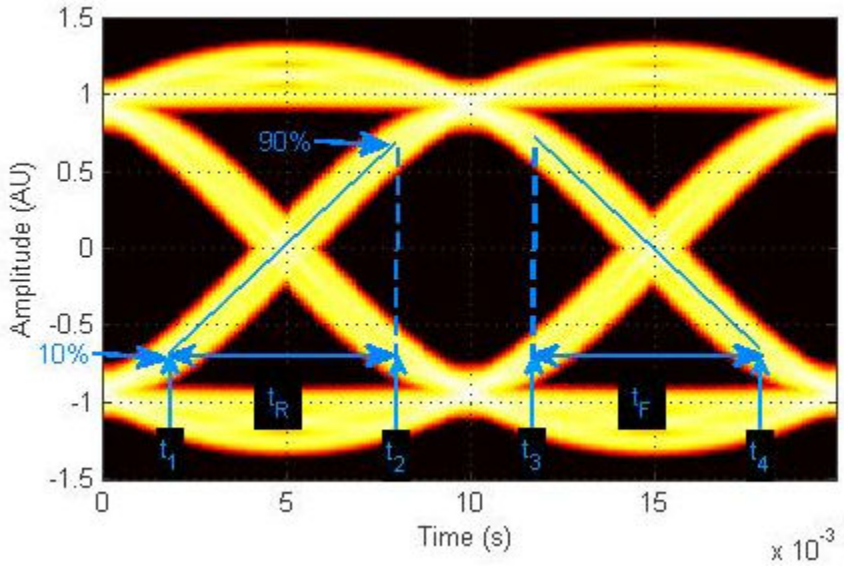
Eye Fall Time (EyeFallTime)

Eye Fall Time is the mean time between the high and low threshold values defined by the AmplitudeThreshold property of the eye measurement setup object. The fall time is calculated from 10% to 90% of the eye amplitude.



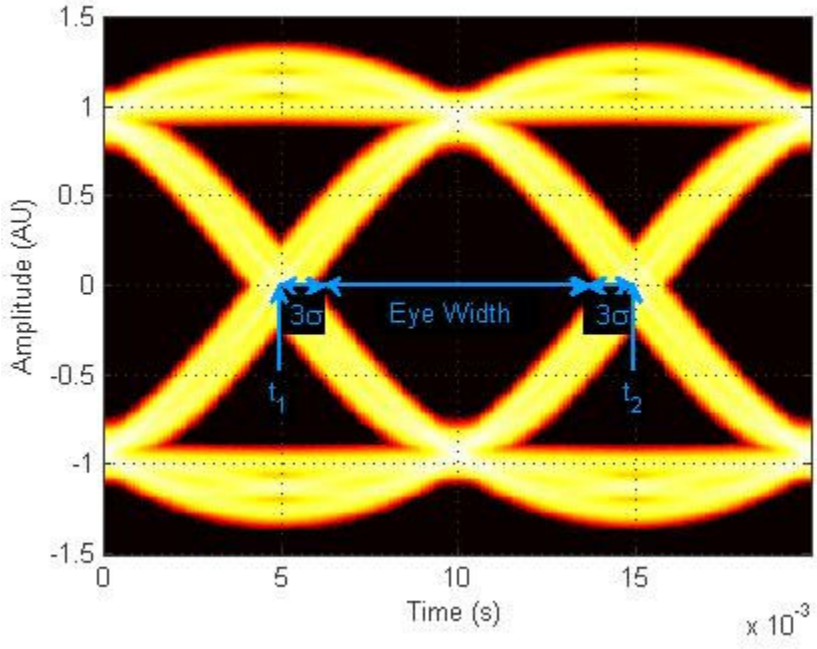
Eye Rise Time (EyeRiseTime)

Eye Rise Time is the mean time between the low and high threshold values defined by the AmplitudeThreshold property of the eye measurement setup object. The rise time is calculated from 10% to 90% of the eye amplitude.



Eye Width (EyeWidth)

Eye Width is the horizontal distance between two points that are three standard deviations (3σ) from the mean eye crossing times, towards the center of the eye. The value for *Eye Width* measurements is seconds.



Horizontal Eye Opening (EyeOpeningHor)

Horizontal Eye Opening is the horizontal distance between two points on the horizontal histogram that correspond to the *BER* value defined by the *BERThreshold* property of the eye measurement setup object. The measurement is take at the amplitude value defined by the `ReferenceAmplitude` property of the eye measurement setup object. It is calculated taking into account the random and deterministic components using a dual-Dirac model [5] (see the Dual Dirac Section).

A typical *BER* value for the eye opening measurements is 10^{-12}, which approximately corresponds to the 7σ point assuming a Gaussian distribution.

Peak-to-Peak Jitter (JitterP2P)

Peak-To-Peak Jitter is the difference between the extreme data points of the histogram.

Random Jitter (JitterRandom)

Random Jitter is defined as the Gaussian unbounded component of the jitter. The analyze method calculates it using the tail standard deviation estimated using the dual-Dirac method as follows [5]:

$RJ = (Q_L + Q_R) * \sigma$

where

$$Q_L = \sqrt{2} * erfc^{-1}\left(\frac{2 * BER}{\rho_L}\right)$$

and

$$Q_R = \sqrt{2} * erfc^{-1}\left(\frac{2 * BER}{\rho_R}\right)$$

BER is the bit error ratio at which the random jitter is calculated. It is defined with the *BERThreshold* property of the eye measurement setup object.

RMS Jitter (JitterRMS)

RMS Jitter is the standard deviation of the jitter calculated from the horizontal histogram.

Total Jitter (JitterTotal)

Total Jitter is the sum of the random jitter and the deterministic jitter [5].

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`

Scatter Plots and Constellation Diagrams

A scatter plot or constellation diagram is used to visualize the constellation of a digitally modulated signal.

To produce a scatter plot from a signal, use the `scatterplot` function or use the `comm.ConstellationDiagram` System object. A scatter plot or constellation diagram can be useful when comparing system performance to a published standard, such as 3GPP or DVB.

You create the `comm.ConstellationDiagram` object with a default object or by defining name-value pairs.

View Signals Using Constellation Diagrams

This example shows how to use constellation diagrams to view QPSK transmitted and received signals which are pulse shaped with a raised cosine filter.

Create a QPSK modulator.

```
qpsk = comm.QPSKModulator;
```

Create a raised cosine transmit filter with samples per symbol, `sps`, equal to 16.

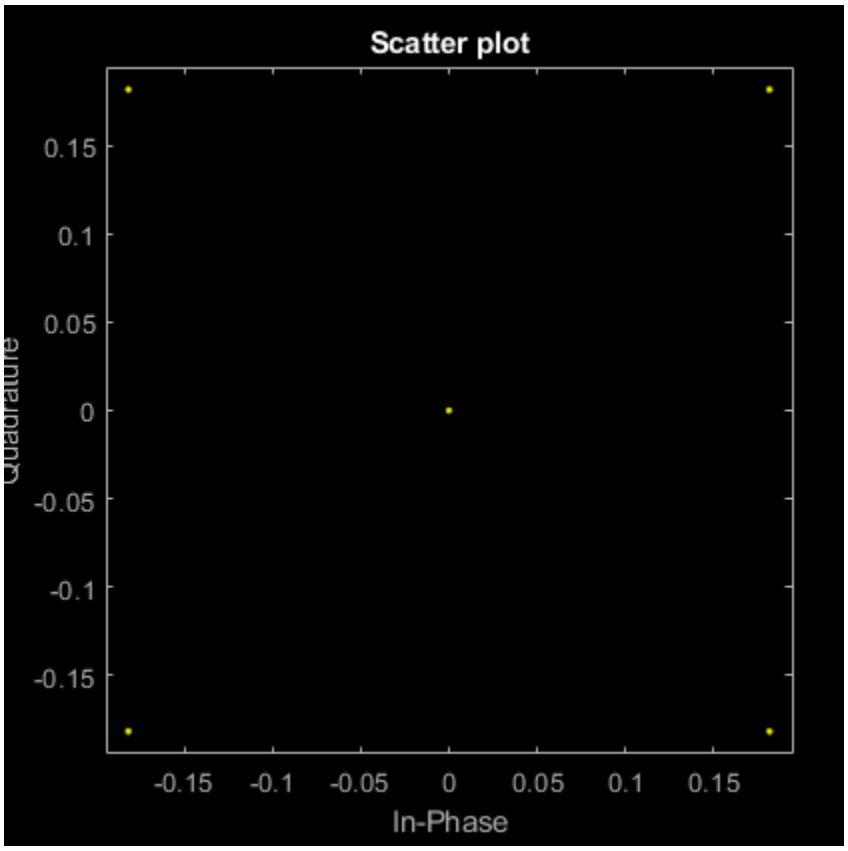
```
sps = 16;
txfilter = comm.RaisedCosineTransmitFilter('Shape','Normal', ...
    'RolloffFactor',0.22, ...
    'FilterSpanInSymbols',20, ...
    'OutputSamplesPerSymbol',sps);
```

Generate data symbols, apply QPSK modulation, and pass the modulated data through the raised cosine transmit filter.

```
data = randi([0 3],200,1);
modData = qpsk(data);
txSig = txfilter(modData);
```

You can display the constellation diagram of the transmitted signal using `scatterplot`. Since the signal is oversampled at the filter output, you need to decimate by the number of samples per symbol so that the scatter plot does not show the transition path between constellation points. If the signal had a timing offset, you could provide that as an input parameter to display the signal constellation with the timing offset corrected.

```
scatterplot(txSig,sps)
```



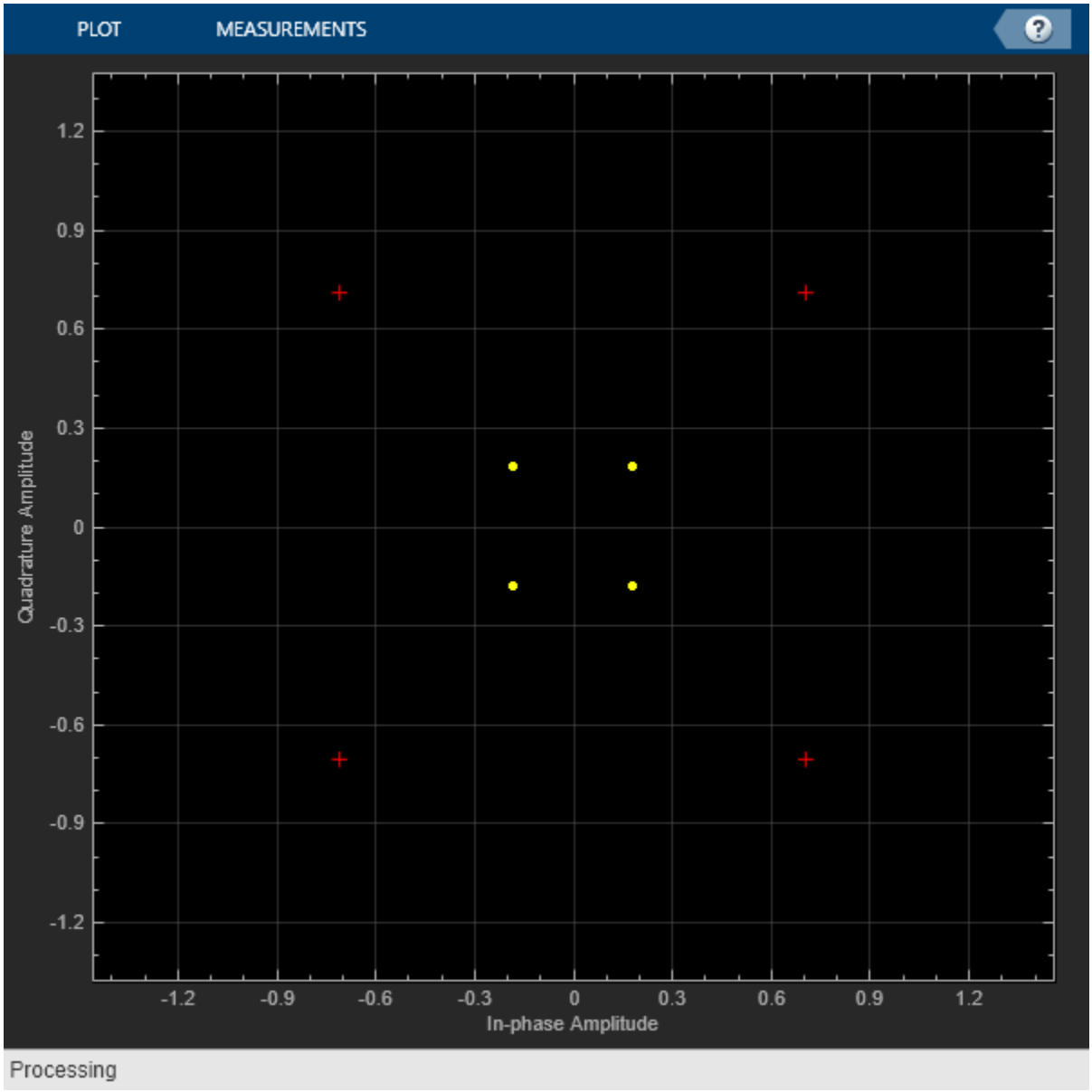
Alternately, you can use `comm.ConstellationDiagram`, specifying the number of samples per symbol, and if needed the timing offset. Also, using `comm.ConstellationDiagram` the reference constellation can be shown.

Create a constellation diagram and set the `SamplesPerSymbol` property to the oversampling factor of the signal. Specify the constellation diagram so that it only displays the last 100 samples. This hides the zero values output by the RRC filter for the first `FilterSpanInSymbols` samples.

```
constDiagram = comm.ConstellationDiagram('SamplesPerSymbol',sps, ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',100);
```

Display the constellation diagram of the transmitted signal.

```
constDiagram(txSig)
```



To match the signal to its reference constellation, normalize the filter by setting its gain to the square root of the `OutputSamplesPerSymbol` property. This was previously specified as `sps`. The filter gain is nontunable so the object must be released prior to changing this value.

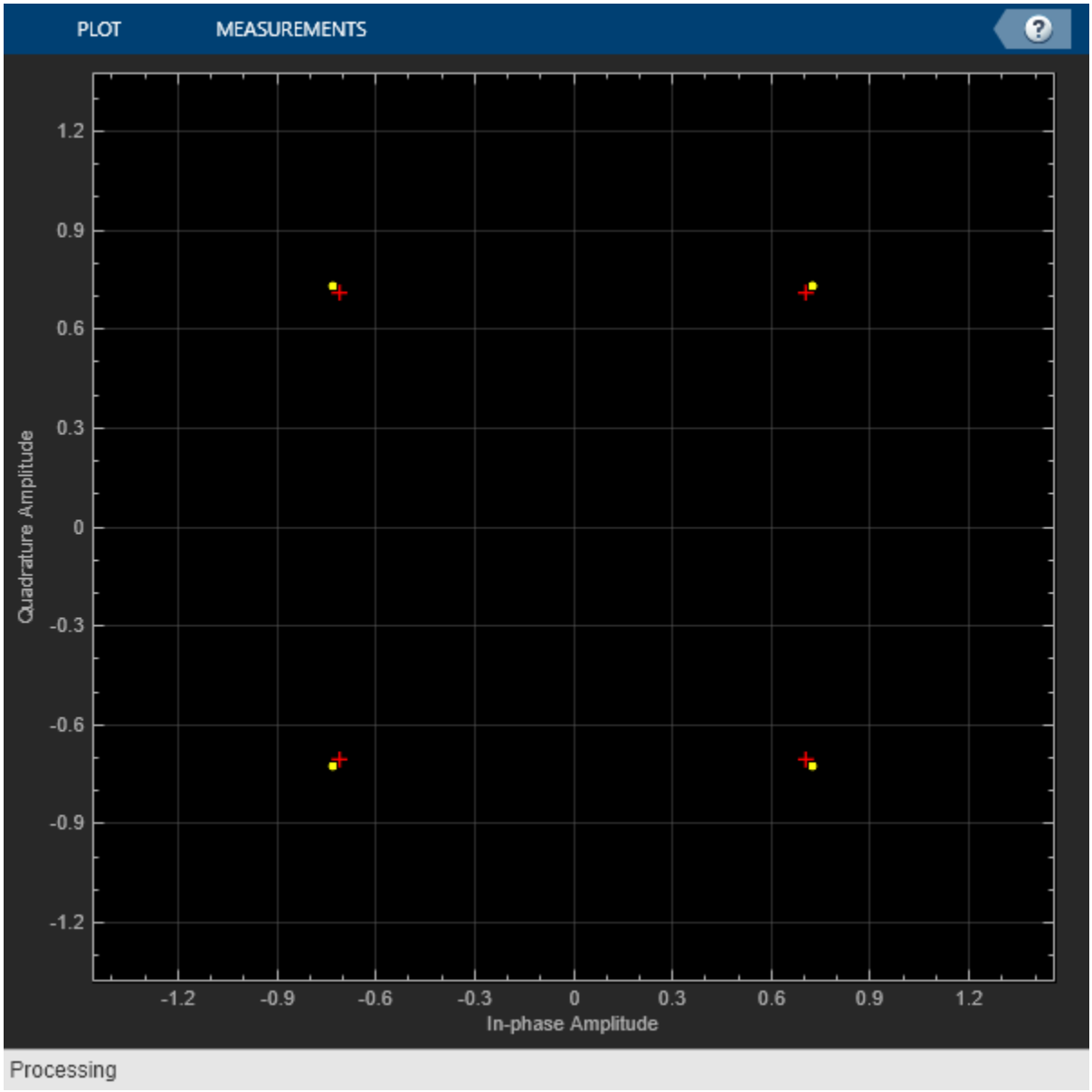
```
release(txfilter)
txfilter.Gain = sqrt(sps);
```

Pass the modulated signal through the normalized filter.

```
txSig = txfilter(modData);
```

Display the constellation diagram of the normalized signal. The data points and reference constellation nearly overlap.

```
constDiagram(txSig)
```



To view the transmitted signal more clearly, hide the reference constellation by setting the `ShowReferenceConstellation` property to `false`.

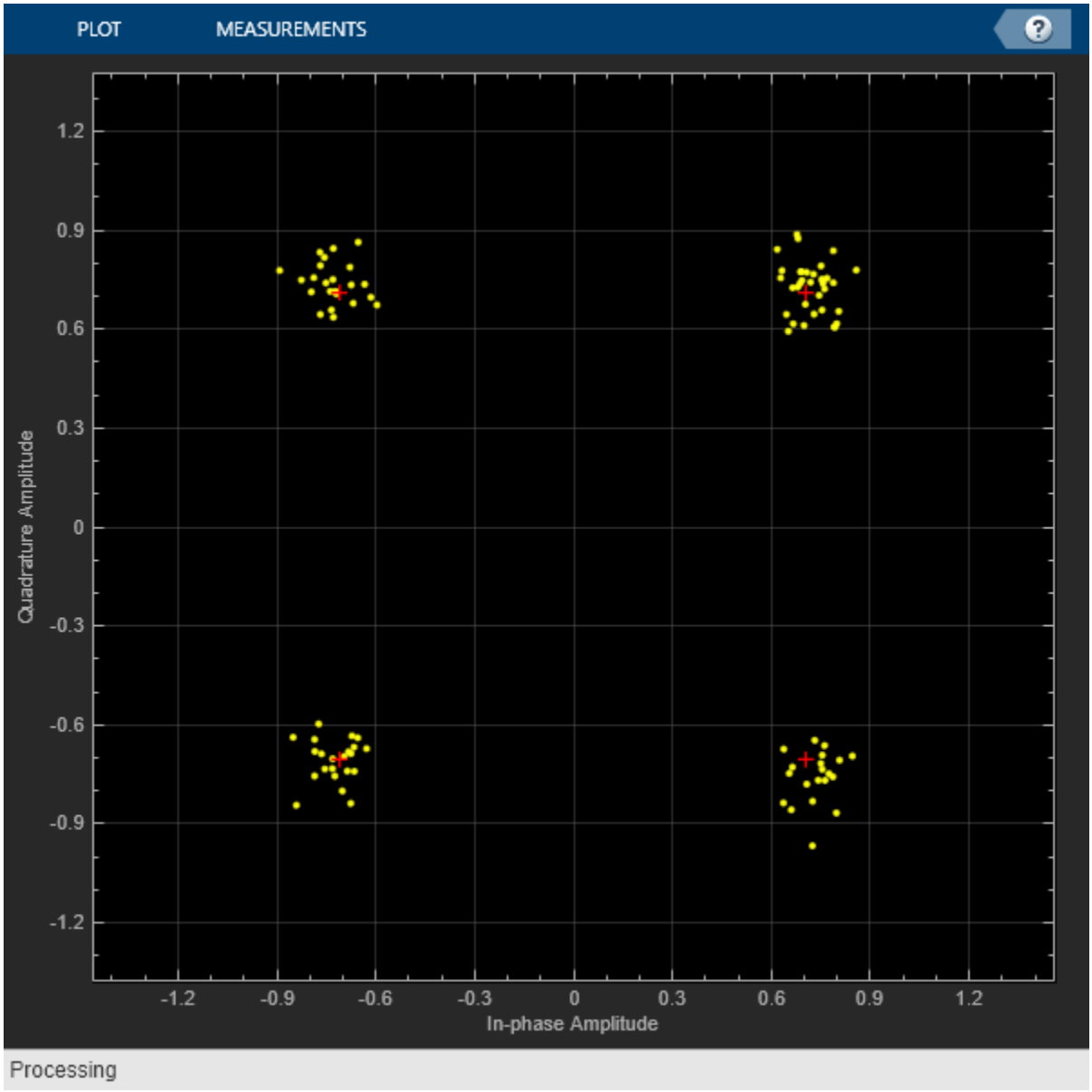
```
constDiagram.ShowReferenceConstellation = false;
```

Create a noisy signal by passing `txSig` through an AWGN channel.

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

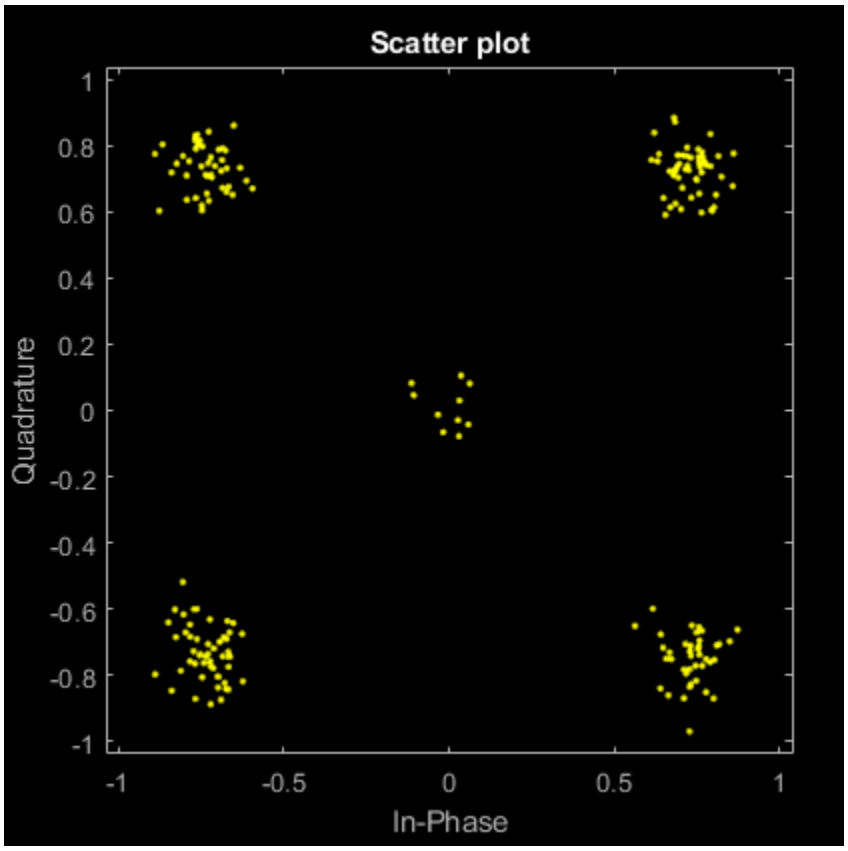
Show the reference constellation and plot the received signal constellation.

```
constDiagram.ShowReferenceConstellation = true;
constDiagram(rxSig)
```



You can also use `scatterplot` to view this noisy signal but there is no built in option to add the reference constellation using `scatterplot`.

```
scatterplot(rxSig,sps)
```



See Also

“View Constellation of Modulator Block” on page 25-2

Channel Visualization

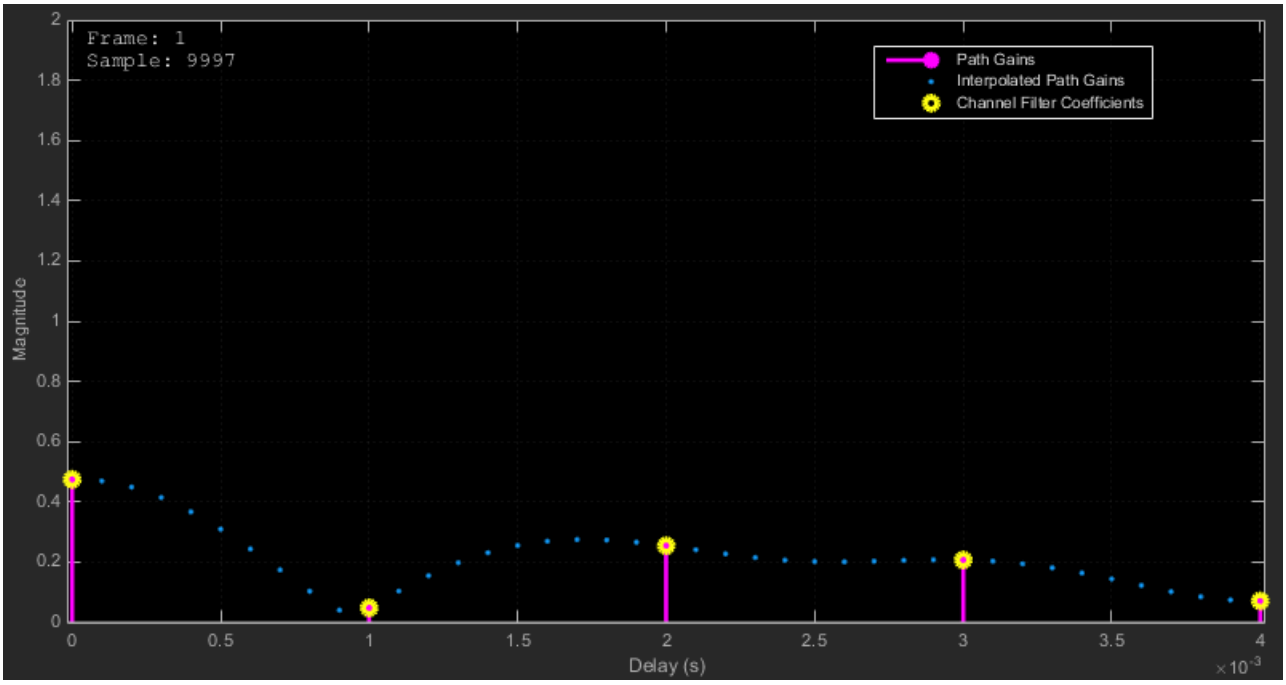
Communications Toolbox software provides a plotting function that helps you visualize the characteristics of a fading channel using a GUI. See Fading Channels on page 22-8 for a description of fading channels, objects, and blocks.

Select the desired visualization setting to plot the Impulse Response on page 25-27, Frequency Response on page 25-29, or Doppler Spectrum on page 25-30 of the channel.

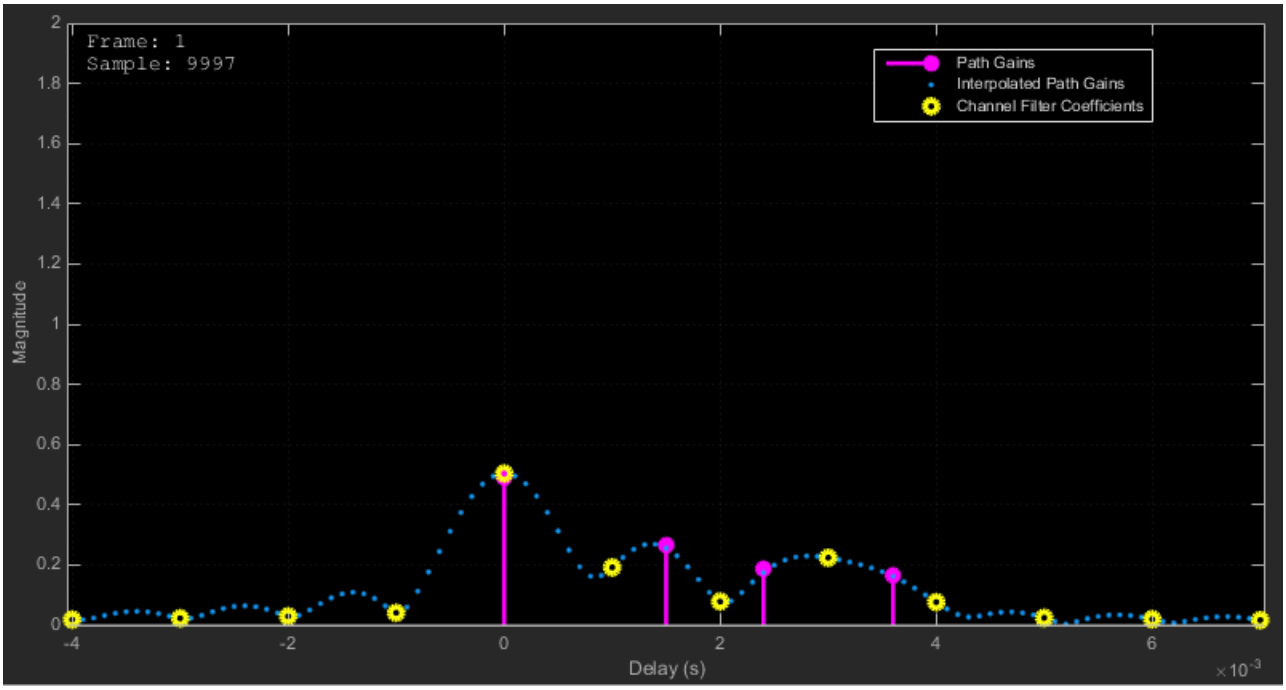
Impulse Response

The Impulse Response plot displays the path gains, the channel filter coefficients, and the interpolated path gains of the channel. The path gains shown in magenta occur at time instances that correspond to the specified path delays. These might not be aligned with the input sampling time. The channel filter coefficients shown in yellow are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. When the path gains align with the sampling time, they overlap the filter coefficients. Sinc interpolation is used to generate the blue points that appear between the channel filter coefficients. 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 points are not displayed. For all impulse response plots, the frame and sample numbers appear in the upper left corner of the display. The Impulse Response plot shares the same toolbar and menus as the System object it was based on, `dsp.ArrayPlot`.

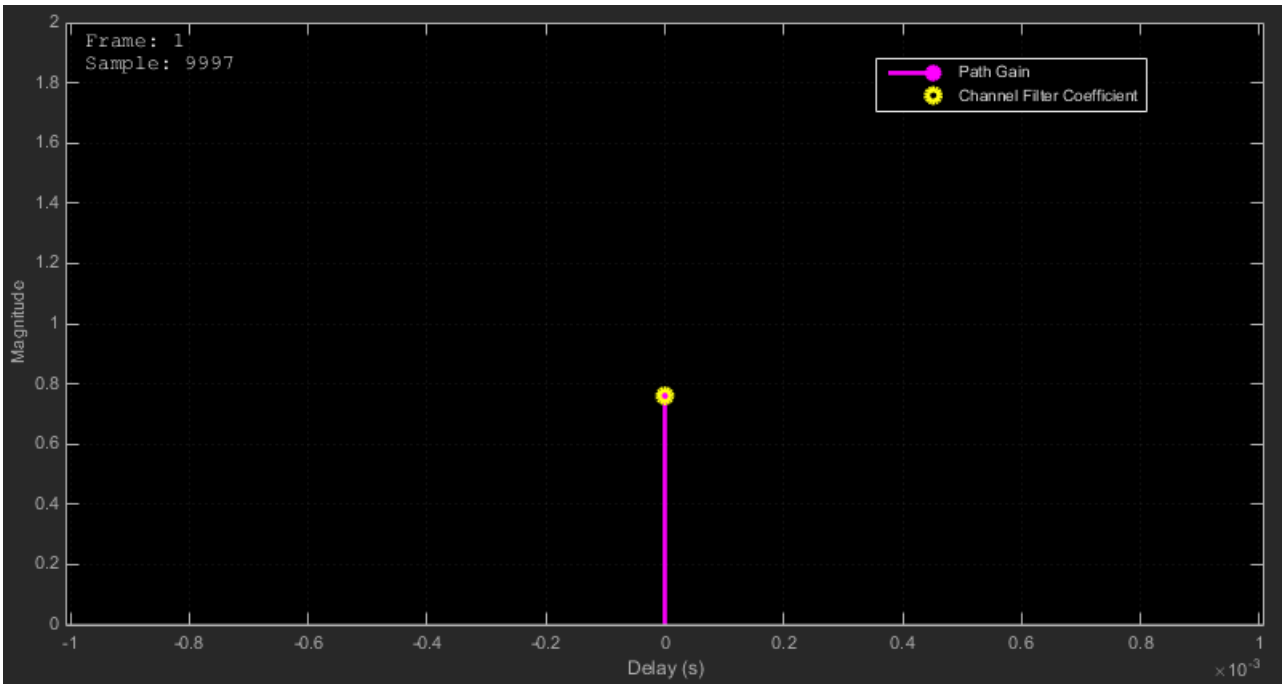
The figure shows the impulse response of a channel in which the path gains align with the sample time. The path gains and channel filter coefficients overlap.



The next plot shows when the specified path gains are not aligned with the sample rate. Observe that the path gains and the channel filter coefficients do not overlap and that the filter coefficients are equally distributed.



The impulse response for a frequency-flat channel is shown next. Because the channel is represented by a single coefficient, no interpolation is done, and the interpolated path gains do not appear.



Note

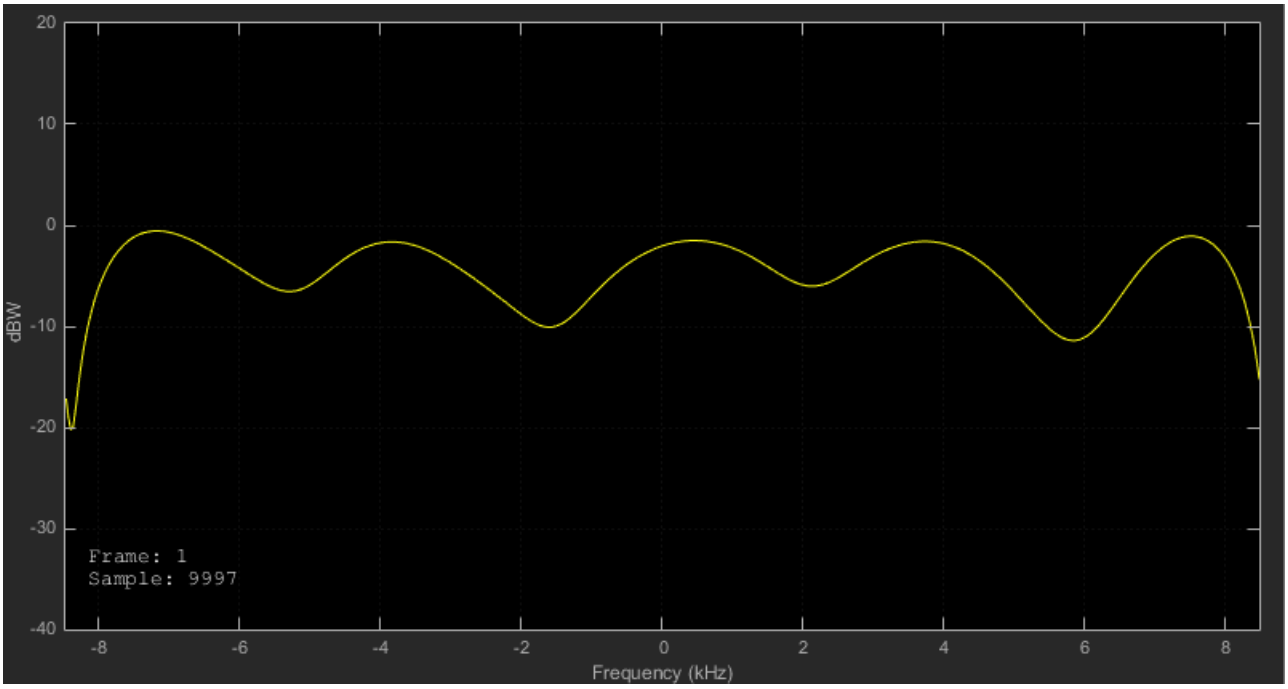
- The displayed and specified path gain locations can differ by as much as 5% of the input sample time.

- For MIMO, when the antenna selection property is set to any value other than `Off` and the specified transmit-receive pair is not selected for the current frame transmission, nothing will be displayed. Antenna selection is not applicable for SISO.
- The visualization display speed is controlled by the combination of the samples to display property and the **Playback > Reduce Updates to Improve Performance** menu item. Reducing the percentage of samples to display and enabling reduced updates speeds up the rendering of the impulse response.
- After the Impulse Response plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is set to `Off`.

Frequency Response

The Frequency Response plot displays the channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. For the MIMO case, this transform is performed for the specified transmit-receive antenna pair. The Frequency Response plot shares the same toolbar and menus as the System object it was based on, `dsp.SpectrumAnalyzer`. The default settings use a rectangular window. The window length is set based on the channel model configuration. Use the **View > Spectrum Settings** menu to change property values from their default settings.

The frequency response plot for a frequency-selective channel is shown.



Note

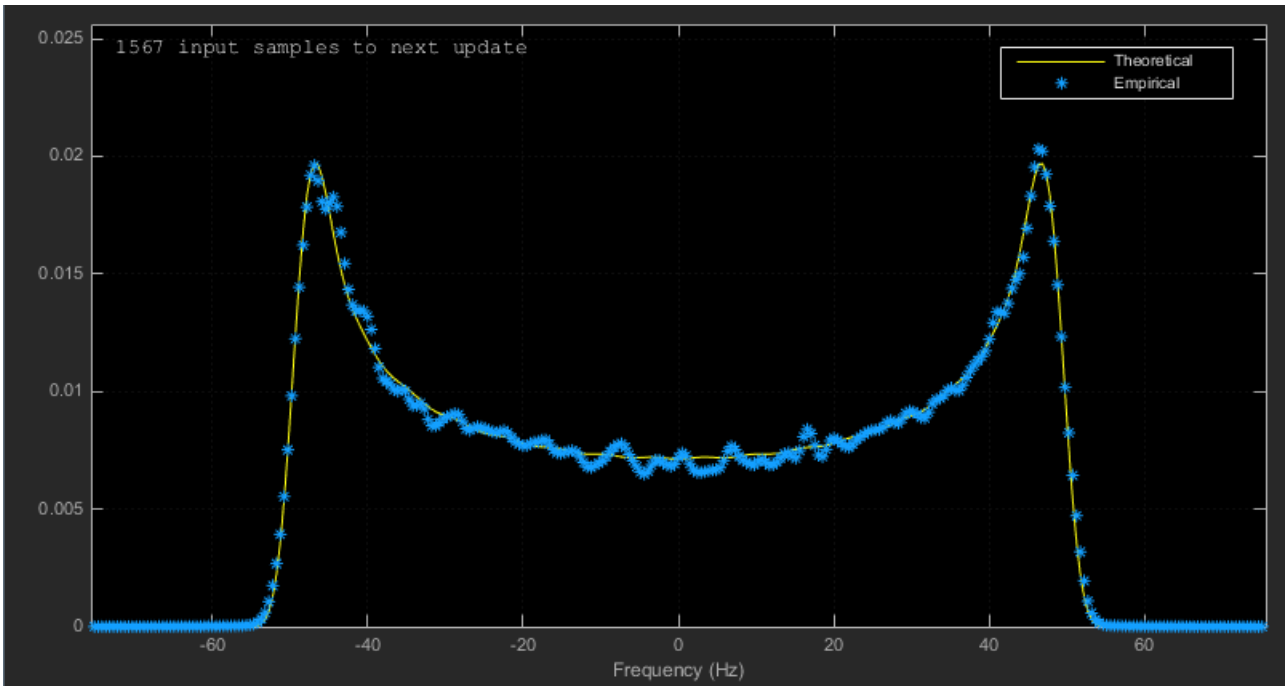
- The visualization display speed is controlled by the combination of the samples to display property and the **Playback > Reduce Plot Rate to Improve Performance** menu item. Reducing the

percentage of samples to display and enabling reduced updates speeds up the rendering of the frequency response.

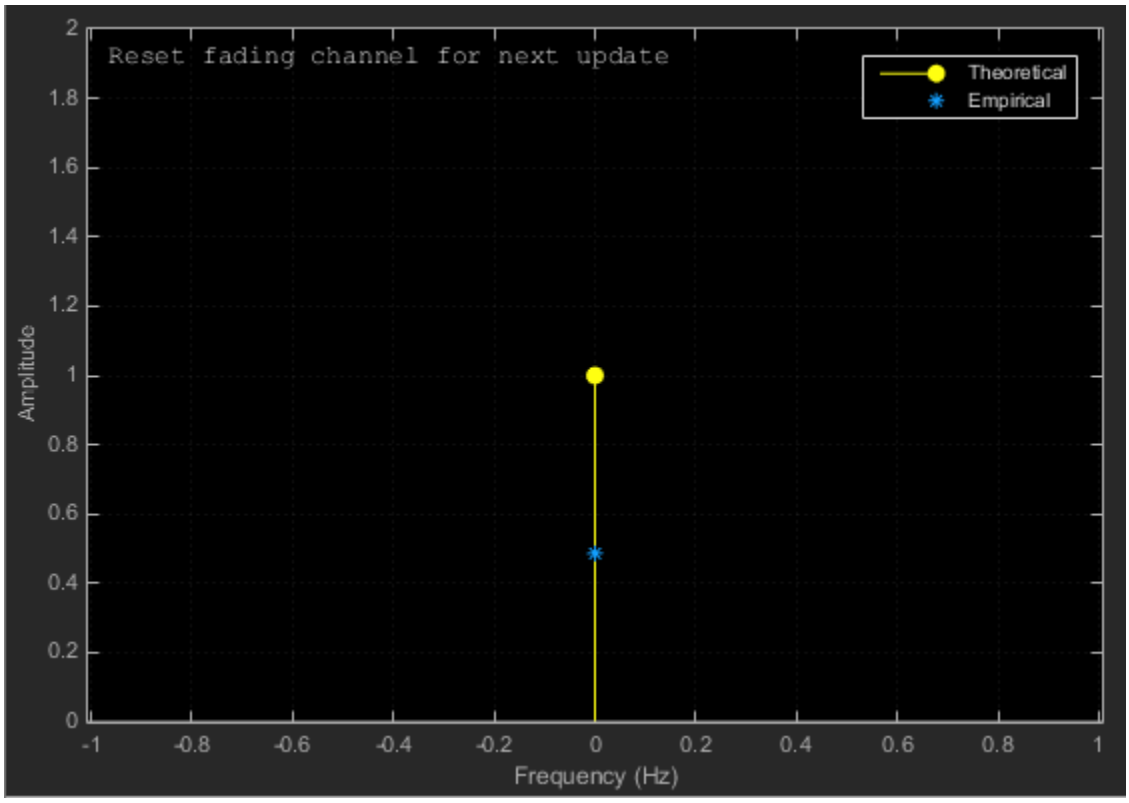
- After the Frequency Response plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is set to `Off`.

Doppler Spectrum

The Doppler Spectrum plot displays both the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a yellow line for nonstatic channels and as a yellow point for static channels. The empirical data is shown in blue. When the internal buffer is completely filled with filtered Gaussian samples, the empirical plot is updated. 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. The samples needed is a function of the sample rate and the maximum Doppler shift. The Doppler Spectrum plot shares the same toolbar and menus as the System object it was based on, `dsp.ArrayPlot`.



For static channels, the text `Reset fading channel for next update` is displayed.



Note

- After the Doppler Spectrum plots are manually closed, the channel model executes at its normal speed.
- Code generation is available only when the visualization property is `Off`.

See Also

Blocks
MIMO Fading Channel | SISO Fading Channel

Objects
`comm.MIMOChannel`

Visualize 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);
```

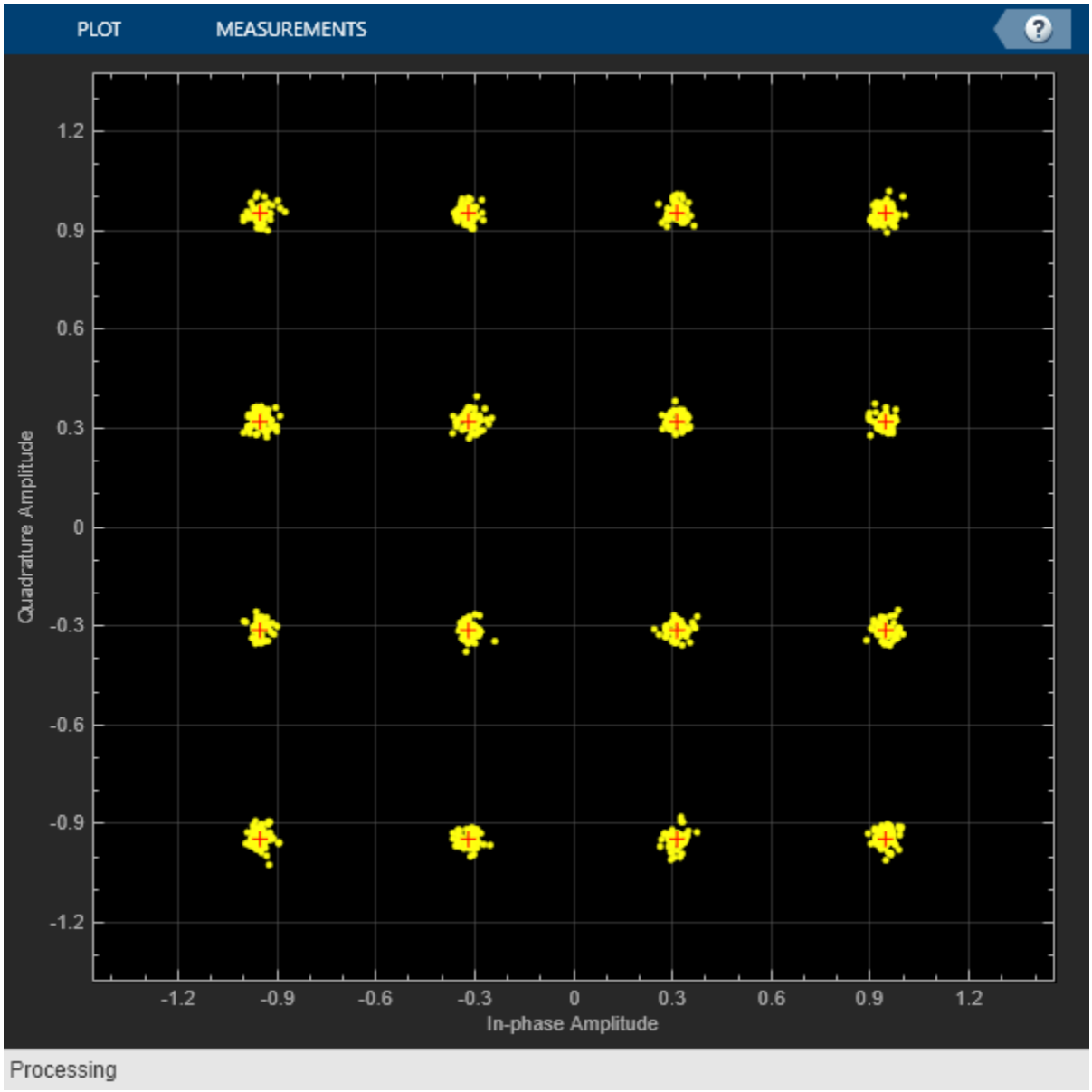
Create constellation diagram and time scope objects to visualize the impairment effects.

```
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 = 3.1768
```

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 = 30.1071
```

The estimate is quite close to the specified SNR of 30 dB.

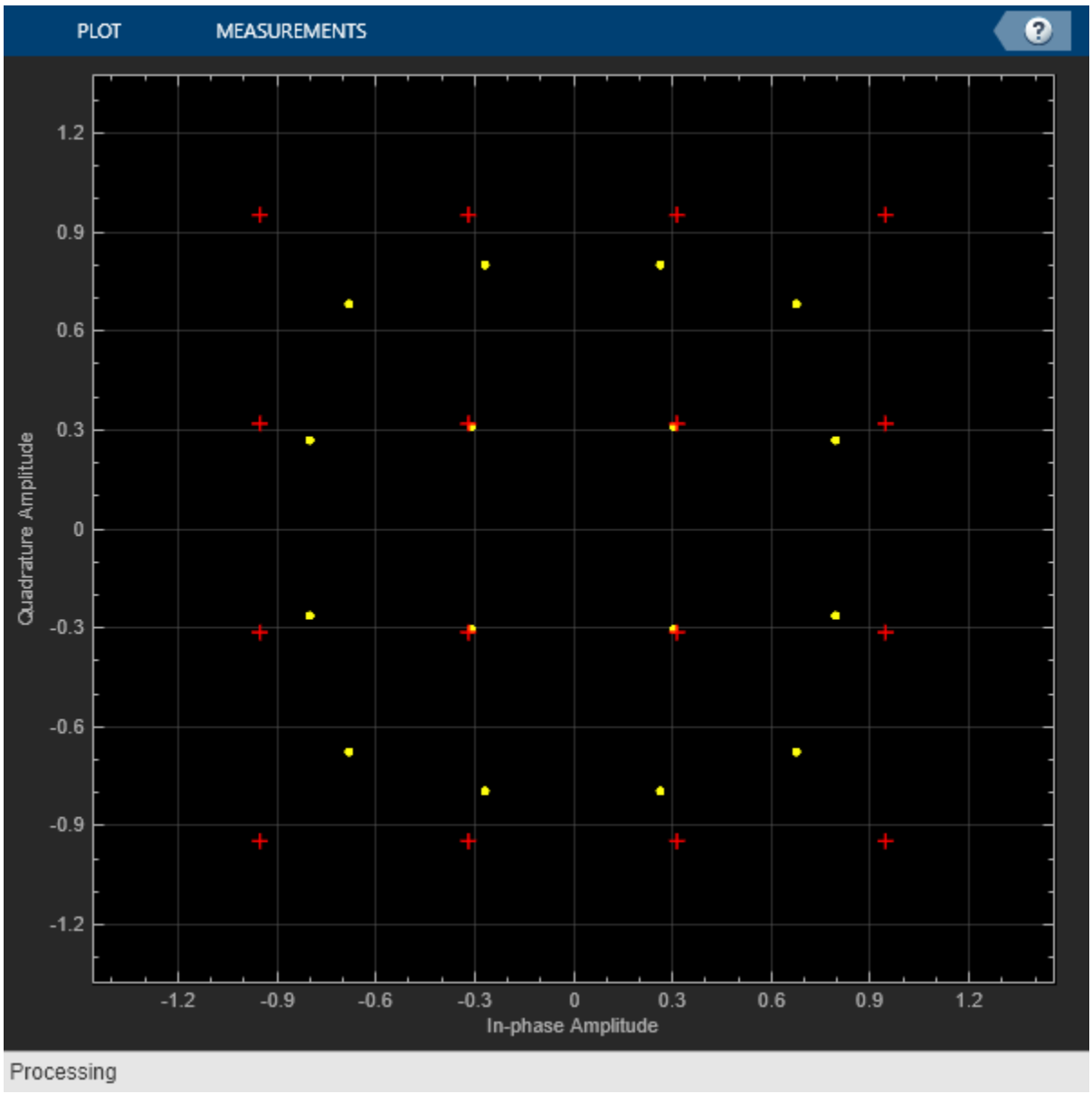
Amplifier Distortion

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.

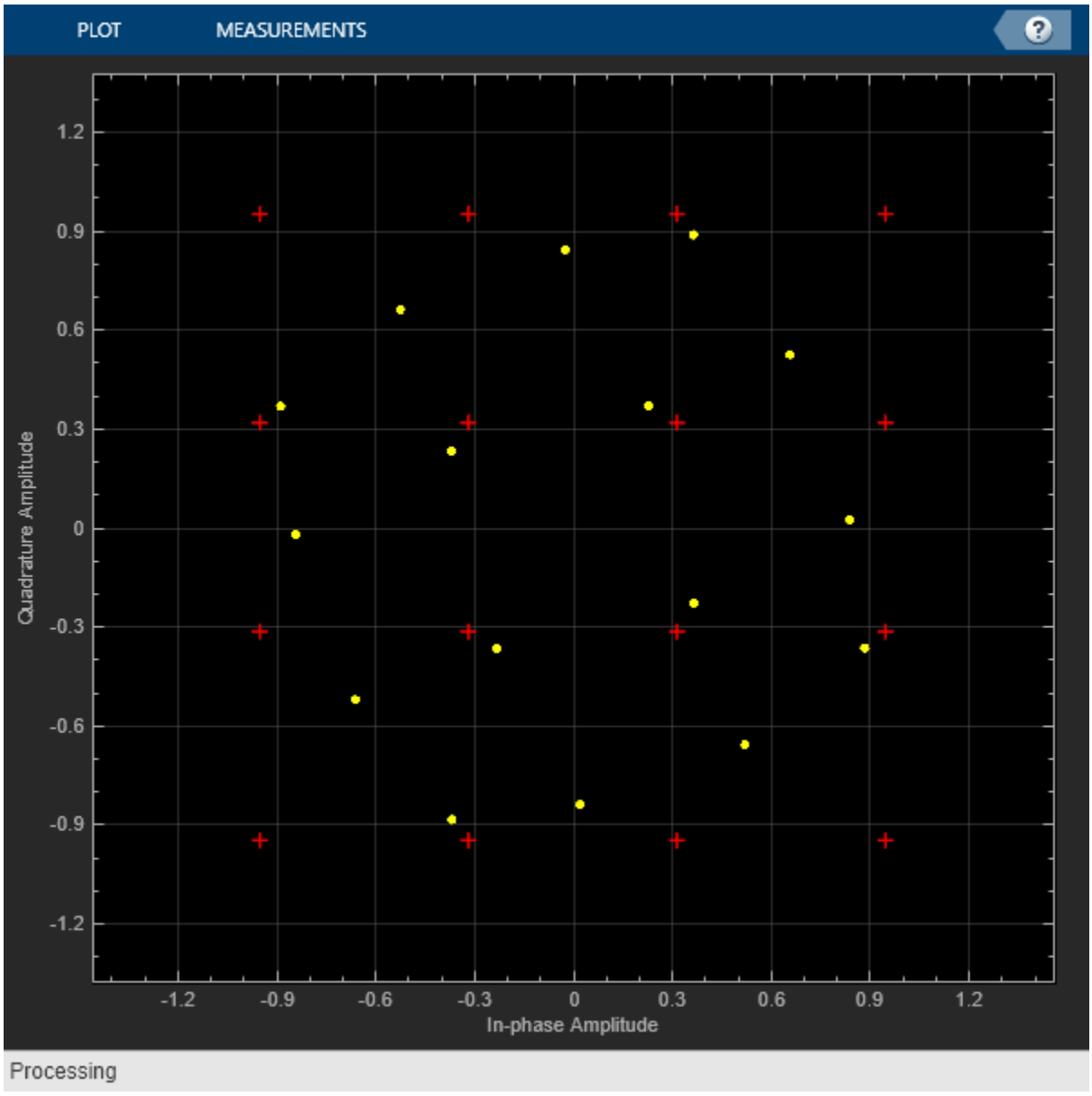
```
txSig = amp(modSig);
constDiagram(txSig)
```



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)
```

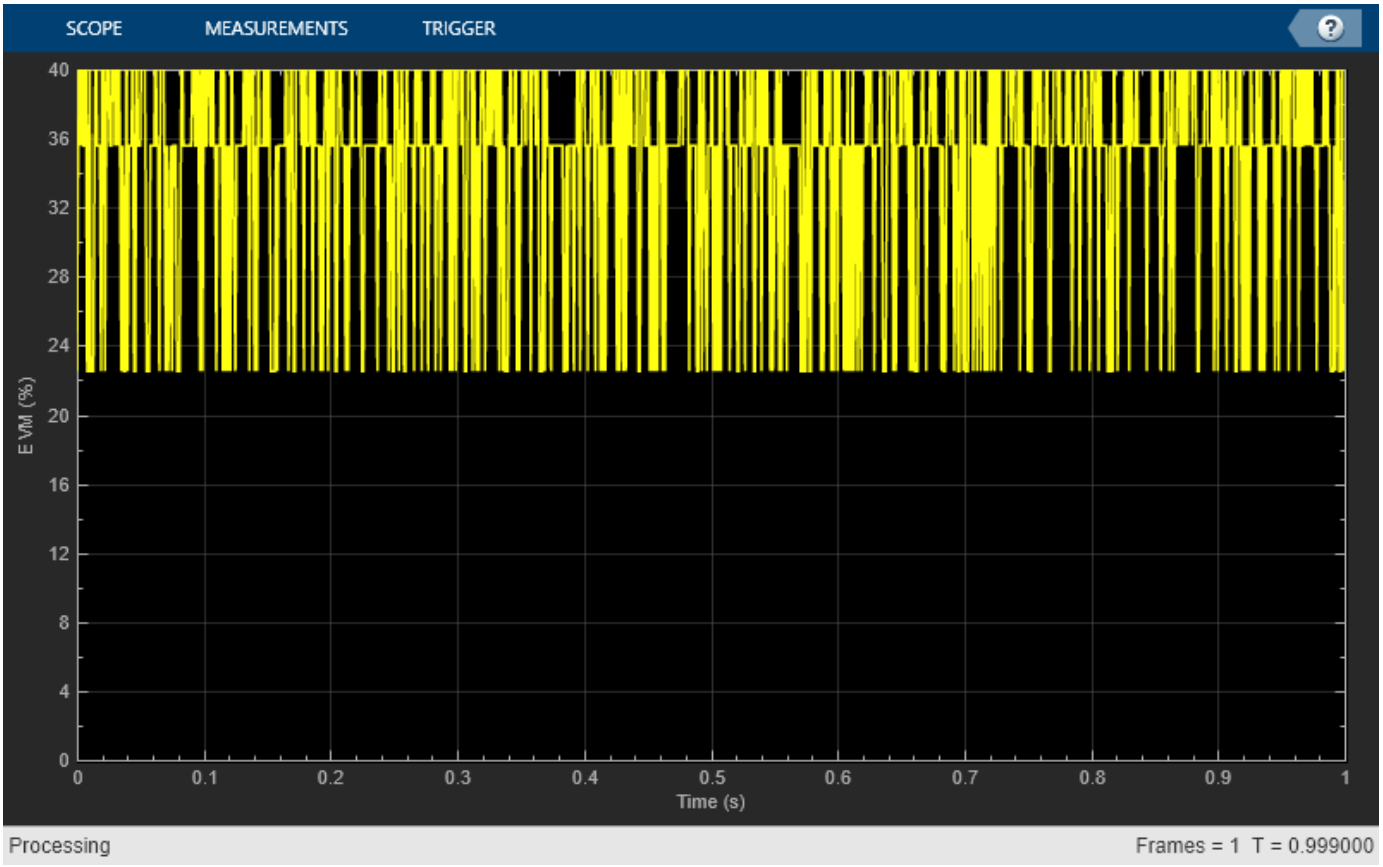


The constellation has rotated due to the AM/PM conversion. To compute the time-varying EVM, release the EVM object and set the `AveragingDimensions` property to 2. To estimate the EVM against an input signal, omit the `ReferenceSignalSource` property definition. This method produces more accurate results.

```
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 = 35.5919
```

Compute the MER.

```
mer = comm.MER;
snrEst = mer(modSig,txSig)
```

```
snrEst = 8.1392
```

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.

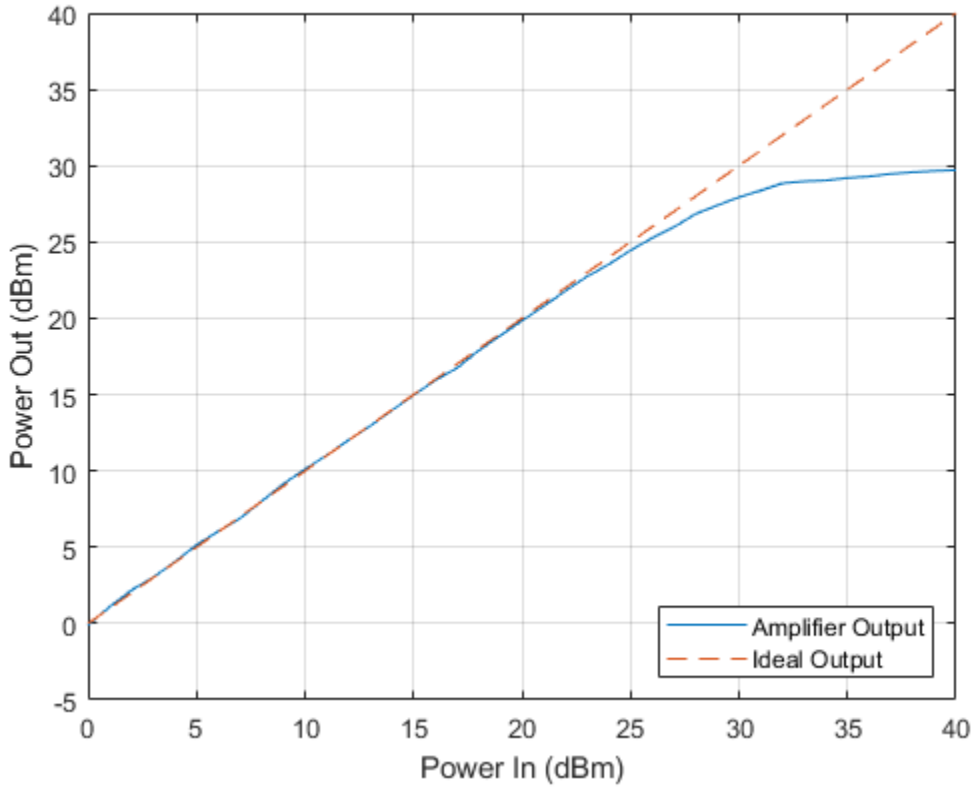
```
powerIn = 0:40;
pin = 10.^((powerIn-30)/10);
powerOut = zeros(length(powerIn),1);
```

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.

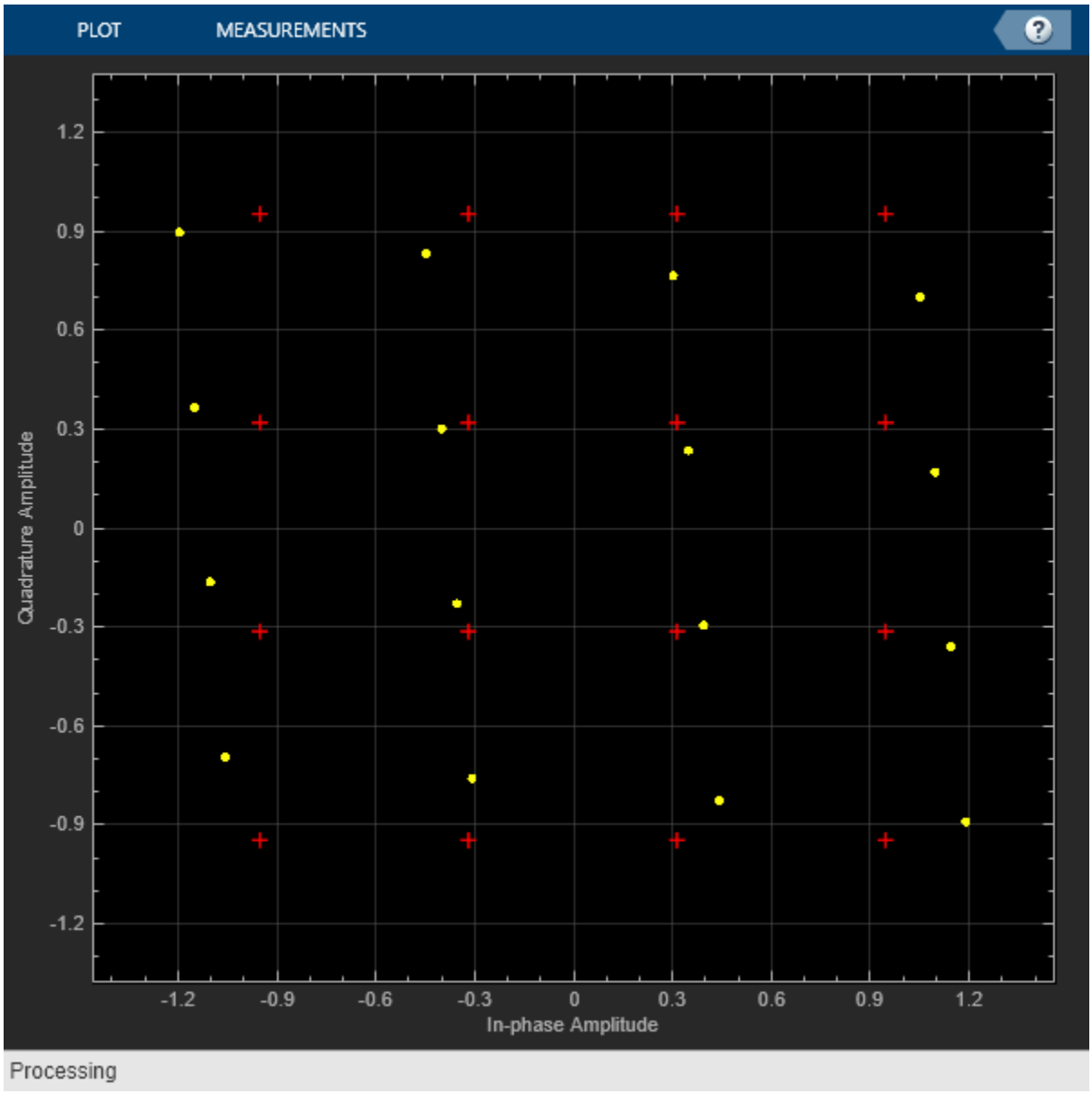
I/Q Imbalance

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.

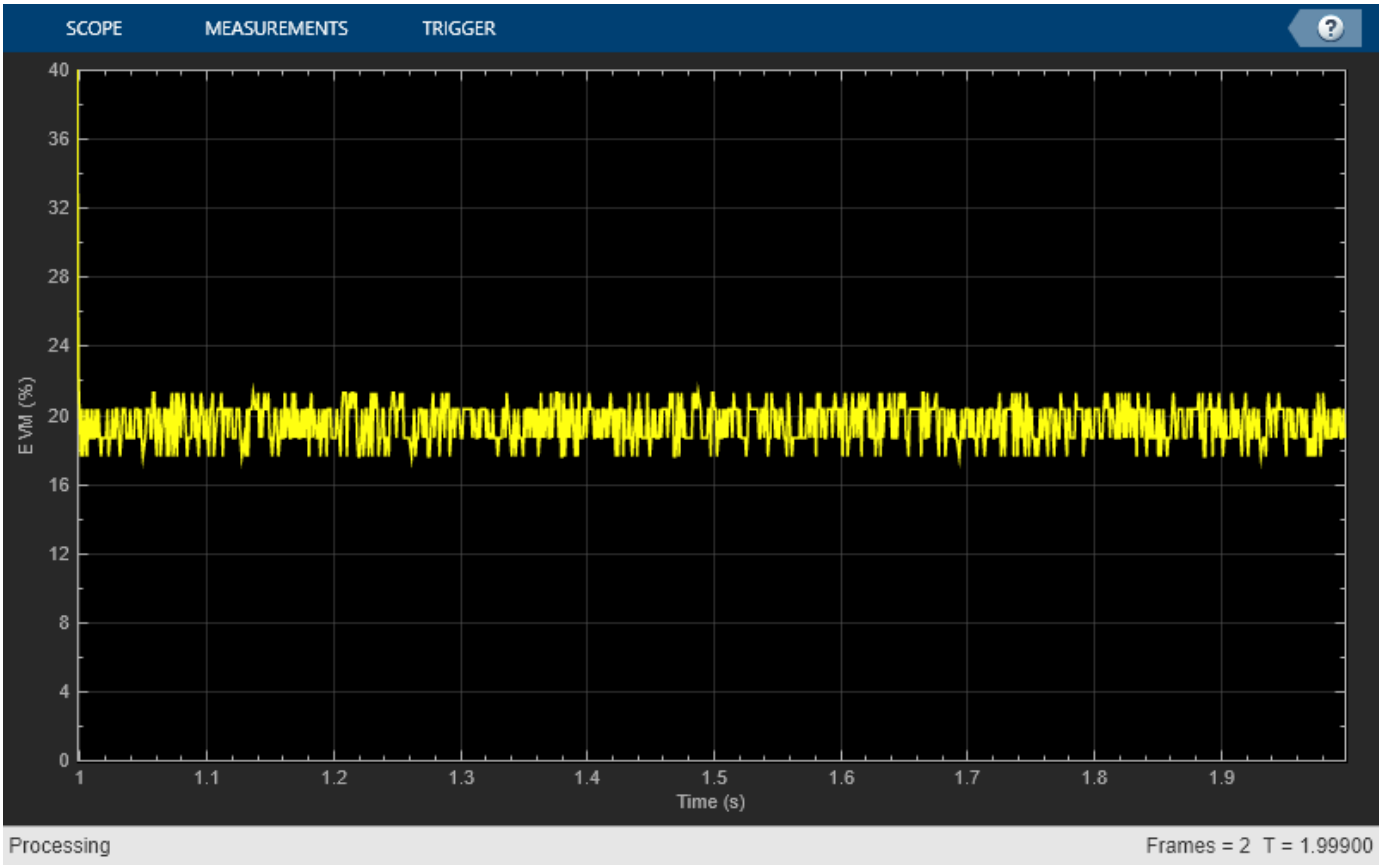
```
constDiagram(rxSig)
```



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.

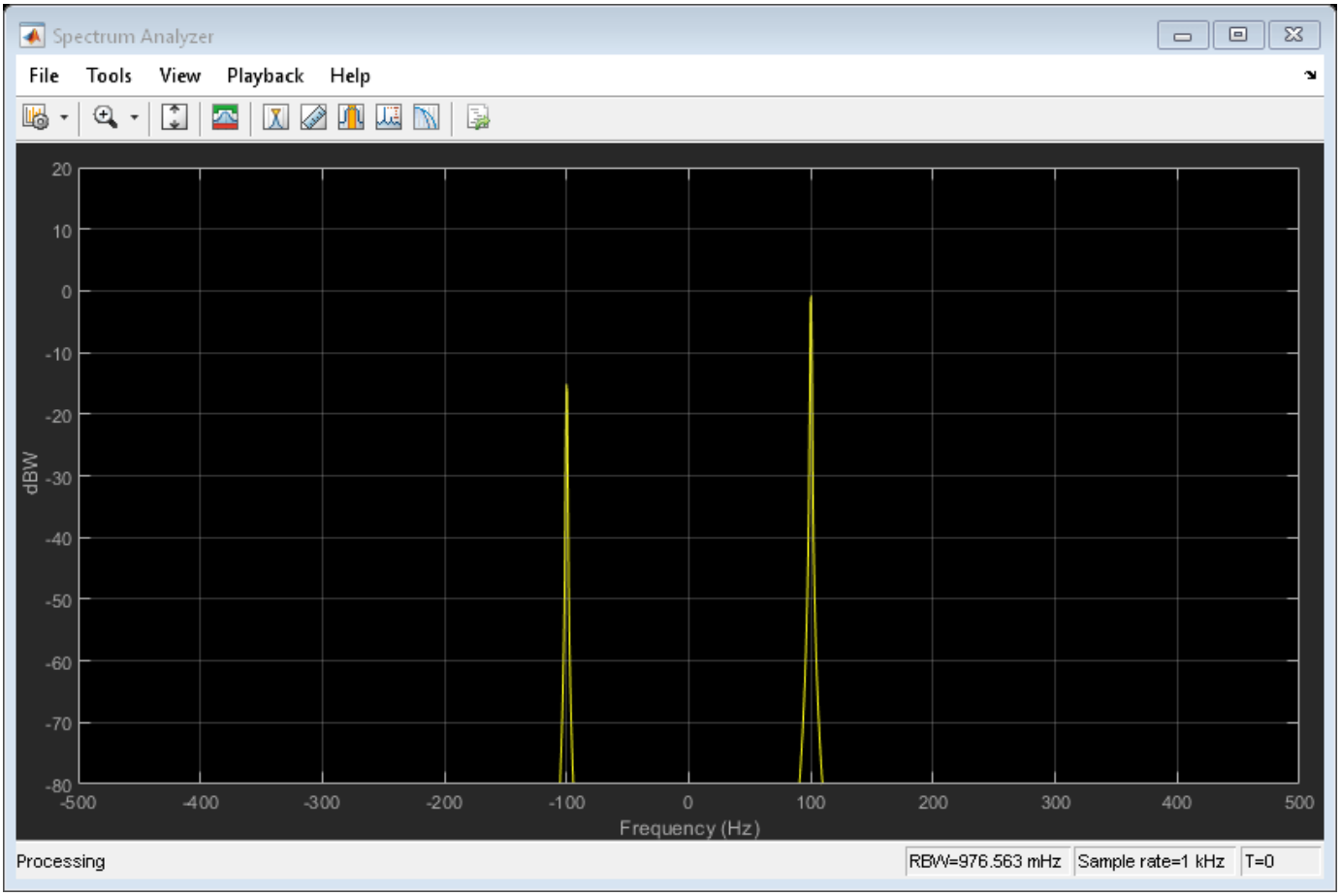
```
sinewave = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
    'SamplesPerFrame',1e4,'ComplexOutput',true);

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)
```

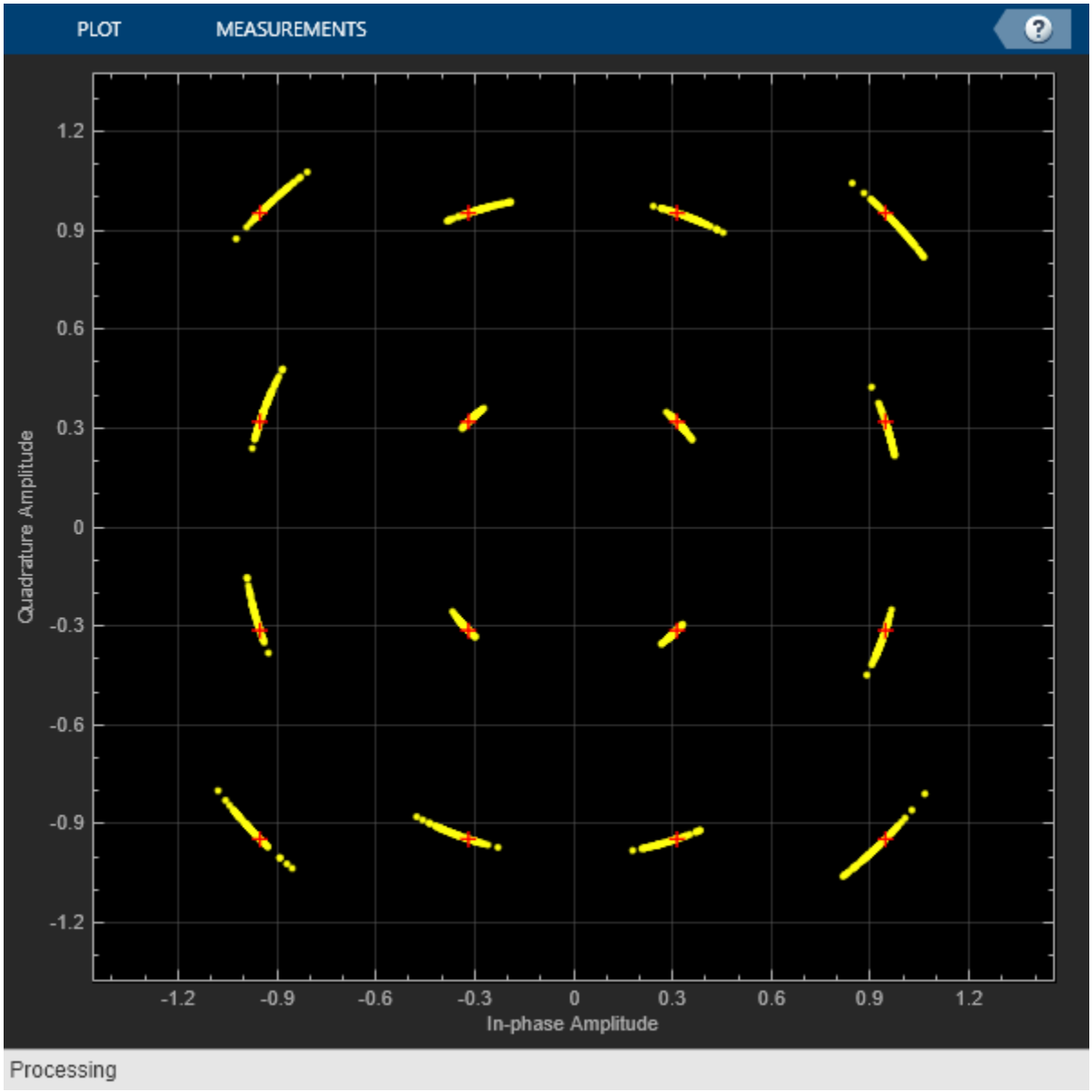


The I/Q imbalance introduces a second tone at -100 Hz, which is the inverse of the input tone.

Phase Noise

Apply phase noise to the transmitted signal. Plot the resulting constellation diagram.

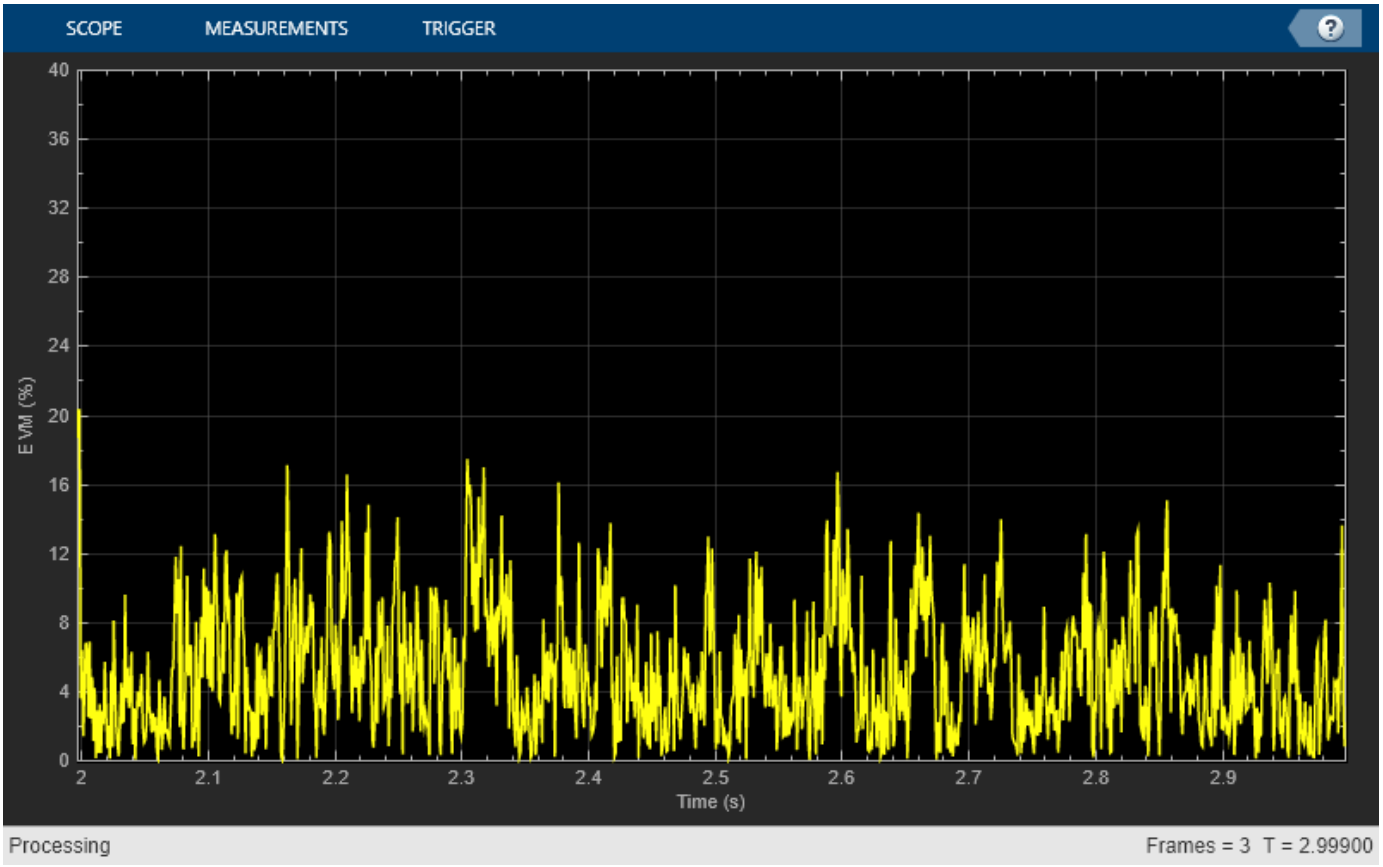
```
pnoise = comm.PhaseNoise('Level',-50,'FrequencyOffset',20,'SampleRate',fs);
pnoiseSig = pnoise(modSig);
constDiagram(pnoiseSig)
```



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 = 6.1989
```

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.

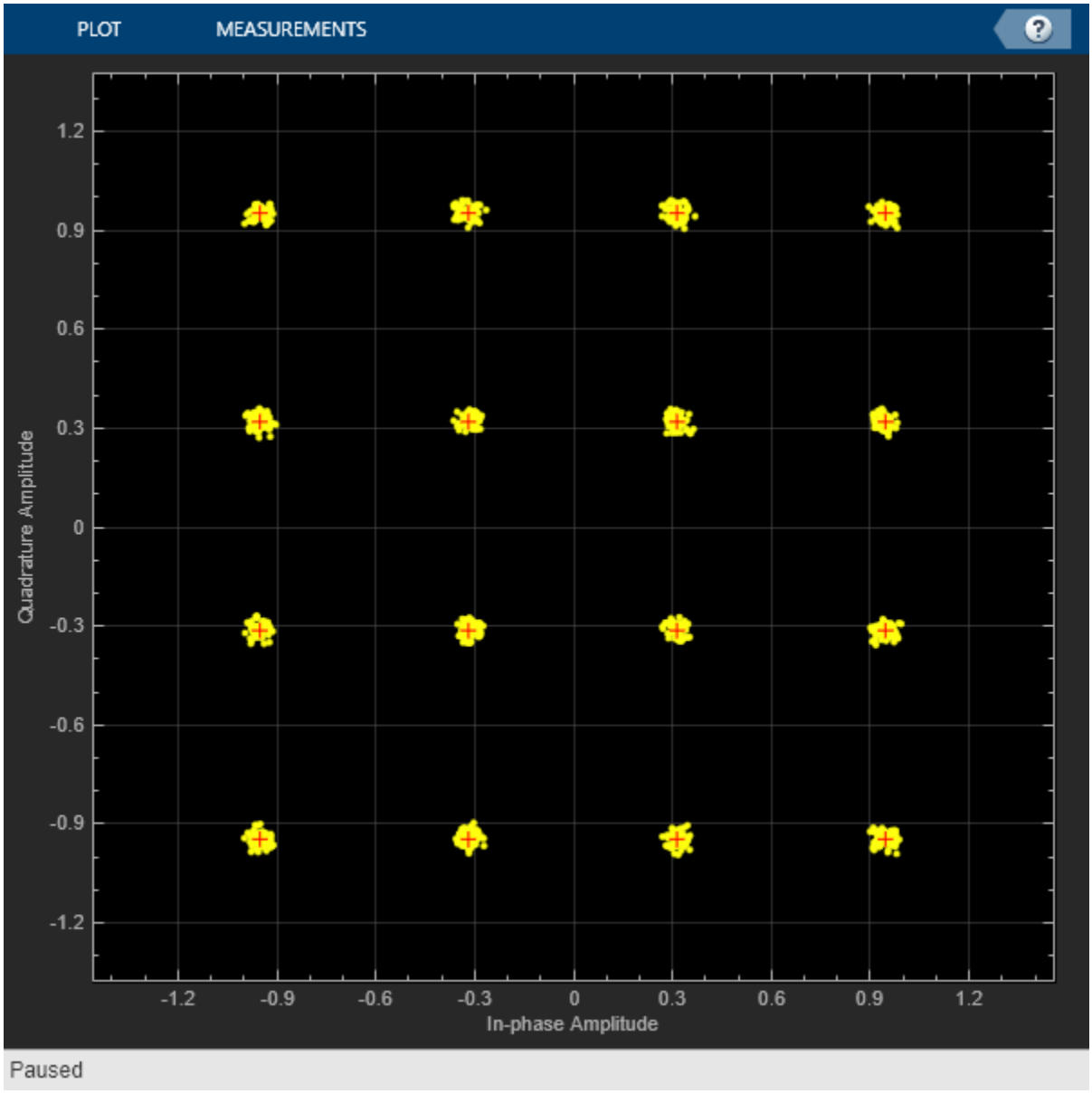
```
rxSig = rxSig(fltDelay+1:end);
```

To accommodate the change in the number of received signal samples, create new constellation diagram and time scope objects.

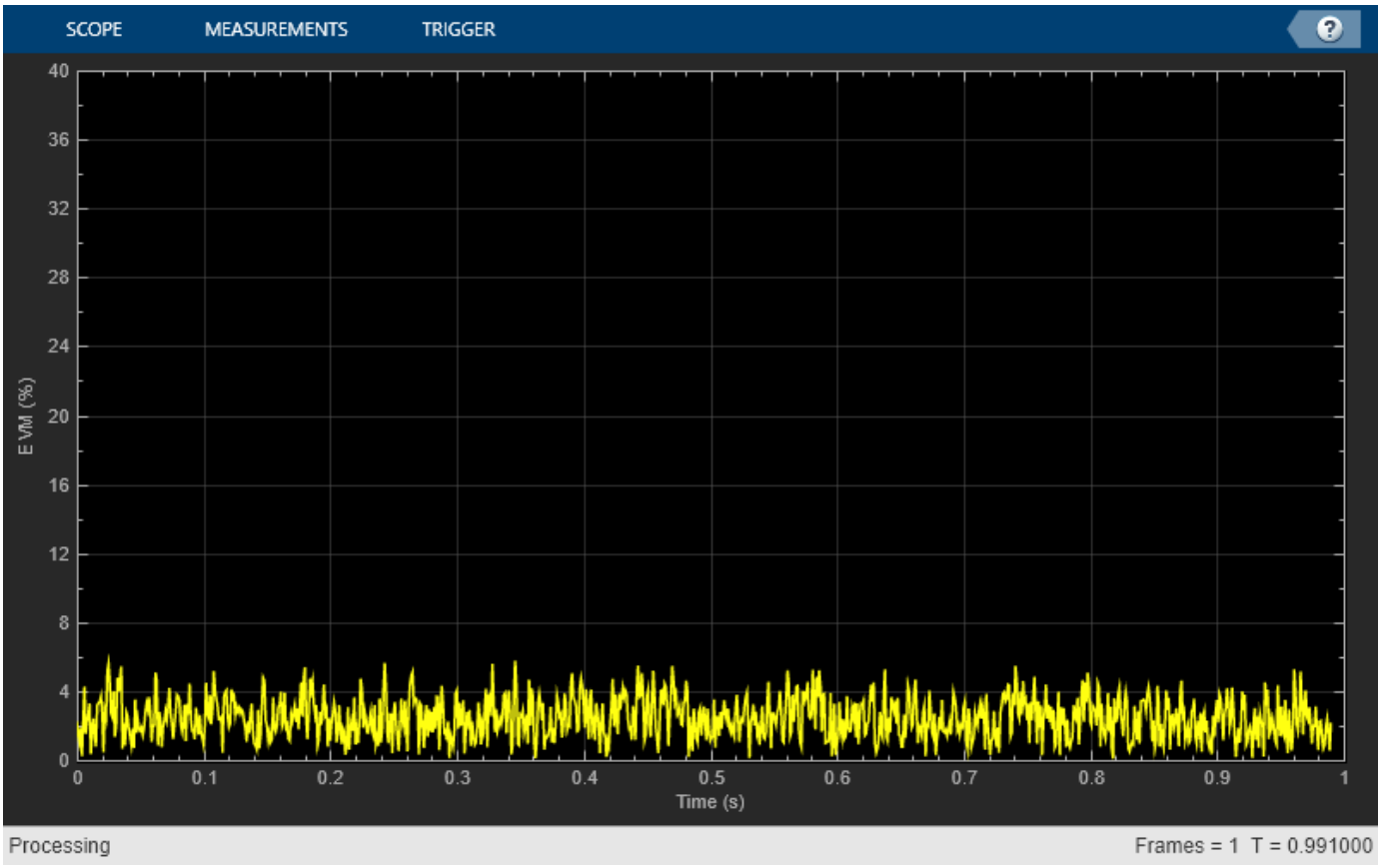
```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = timescope('YLimits',[0 40],'SampleRate',fs,'TimeSpanSource','property','TimeSpan',1,
    'ShowGrid',true,'YLabel','EVM (%)');
```

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 = 2.7199
```

Determine the equivalent SNR.

```
mer = comm.MER;
snrEst = mer(modSig(1:end-fltDelay),rxSig)
```

```
snrEst = 31.4603
```

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.

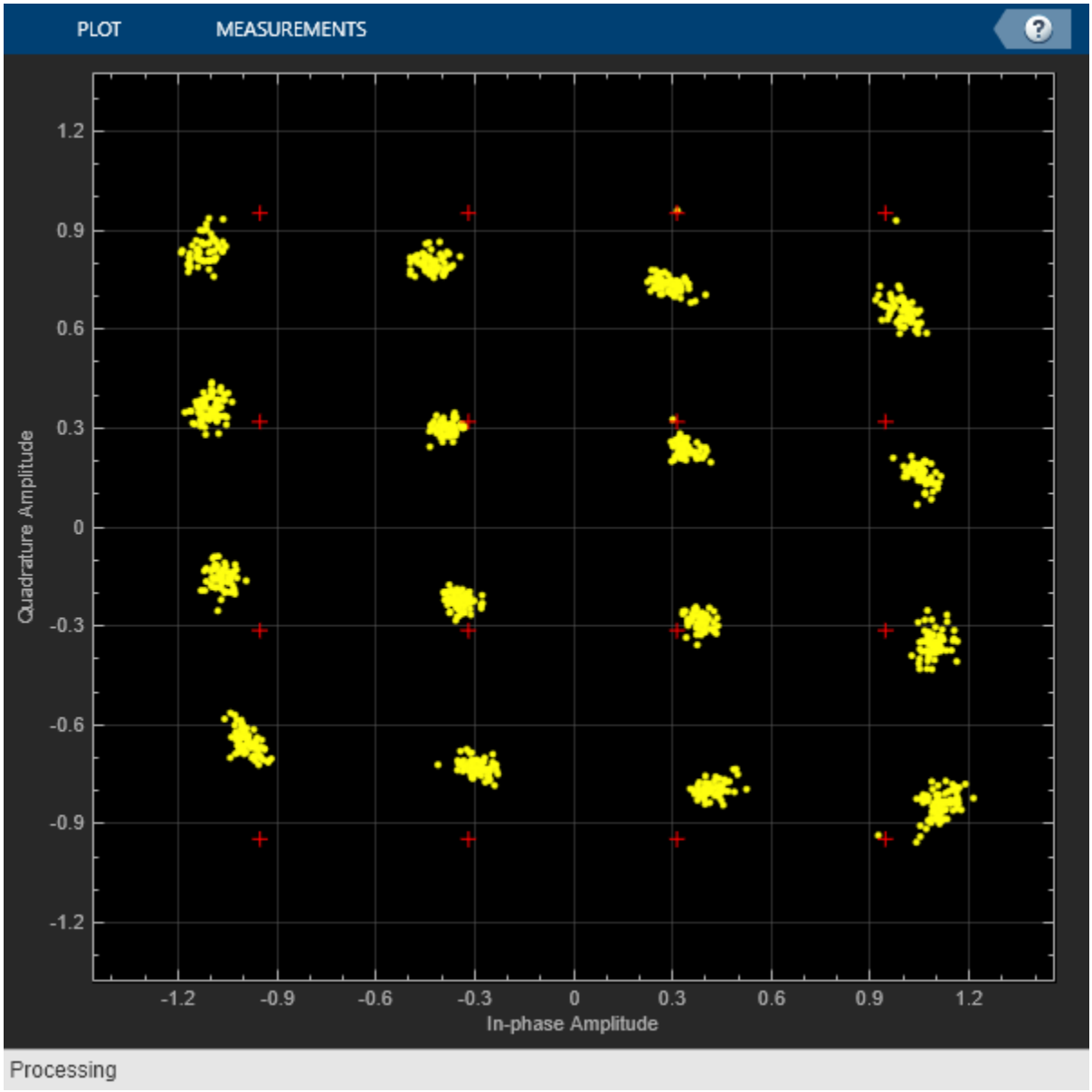
```
amp = comm.MemorylessNonlinearity('IIP3',45,'AMPMConversion',0);
pnoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',fs);
```

Filter and then amplify the modulated signal.

```
txfiltOut = txfilter(modSig);
txSig = amp(txfiltOut);
```

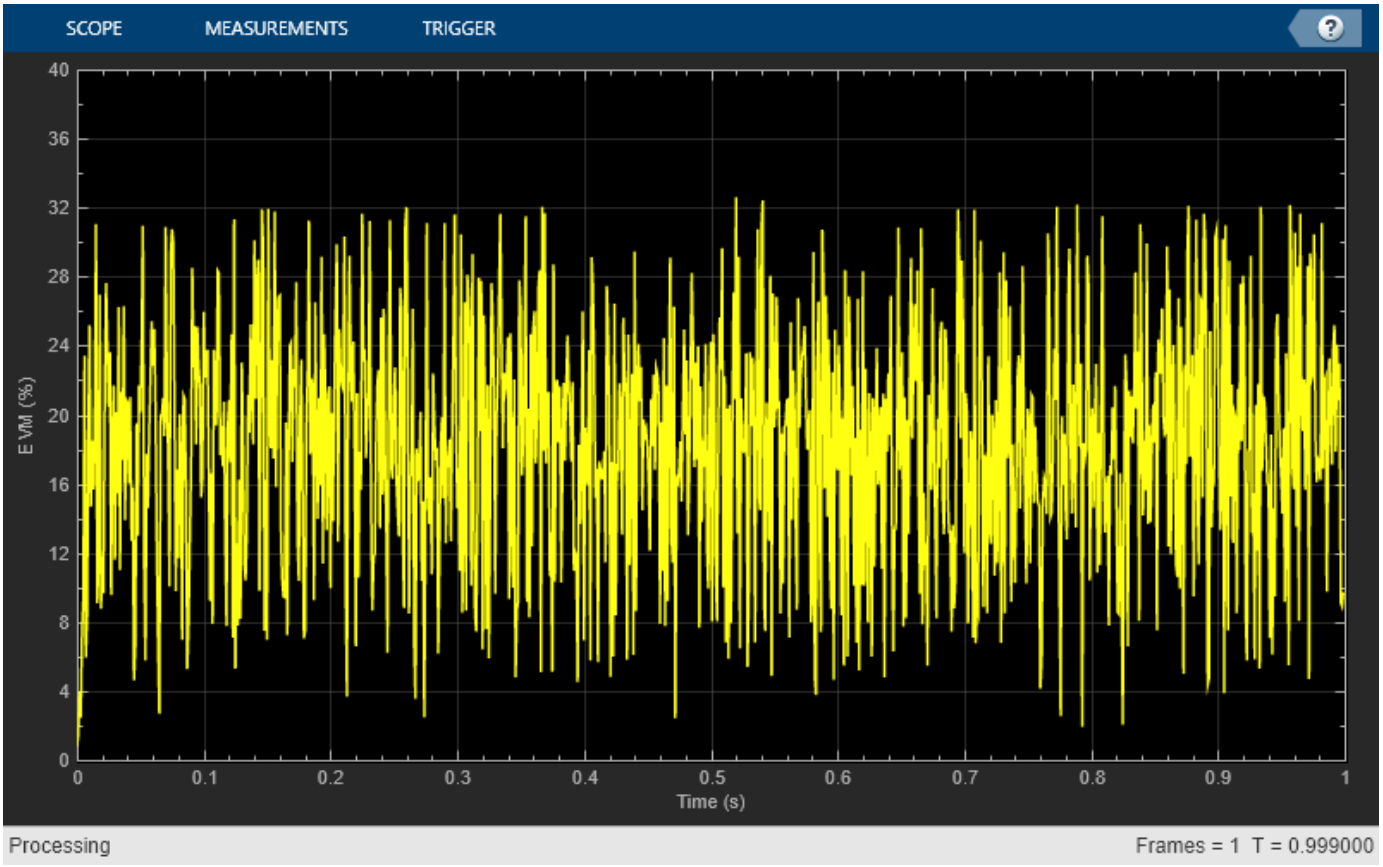
Add phase noise. Pass the impaired signal through the AWGN channel. Plot the constellation diagram.

```
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 = 19.4992
```

Estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(rxfiltOut)
```

```
snrEst = 14.1996
```

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.

See Also

Fading Channels on page 22-8 | “Impact of RF Effects on Communication System Performance” on page 8-32

C Code Generation

Generate C Code from MATLAB Code

MATLAB Coder generates highly optimized ANSI C and C++ code from functions and System objects in Communications Toolbox . You can deploy this code in a wide variety of applications. The workflow described in this topic uses DSP System Toolbox features but the same workflow applies for Communications Toolbox.

This example generates C code from the “Construct a Sinusoidal Signal Using High Energy FFT Coefficients” example and builds an executable from the generated code.

Here is the MATLAB code for this example:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
'PhaseOffset',10,'SampleRate',44100,'Frequency',1000);
ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    FFTCoeff = ft(Input);
    FFTCoeffMagSq = abs(FFTCoeff).^2;

    EnergyFreqDomain = (1/L)*sum(FFTCoeffMagSq);
    [FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');

    CumFFTCoeffs = cumsum(FFTCoeffSorted);
    EnergyPercent = (CumFFTCoeffs/EnergyFreqDomain)*100;
    Vec = find(EnergyPercent > 99.99);
    FFTCoeffsModified = zeros(L,1);
    FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
    ReconstrSignal = ift(FFTCoeffsModified);
end
max(abs(Input-ReconstrSignal))
plot(Input,'*');
hold on;
plot(ReconstrSignal,'o');
hold off;
```

You can run the generated executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have MATLAB installed. You can generate code using the MATLAB Coder app or the `codegen` function. This example shows you the workflow using the `codegen` function. For more information on the app workflow, see “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).

Set Up the Compiler

The first step is to set up a supported C compiler. MATLAB Coder automatically locates and uses a supported installed compiler. You can change the default compiler using `mex -setup`. For more details, see “Change Default Compiler”. For a current list of supported compilers, see Supported and Compatible Compilers.

Break Out the Computational Part of the Algorithm into a MATLAB Function

To generate C code, the entry point must be a function. You do not have to generate code for the entire MATLAB application. If you have specific portions that are computationally intensive, generate code from these portions in order to speed up your algorithm. The harness or the driver that calls this MATLAB function does not need to generate code. The harness runs in MATLAB and can contain visualization and other verification tools that are not actually part of the system under test. For

example, in the “Construct a Sinusoidal Signal Using High Energy FFT Coefficients” example, the `plot` functions plot the input signal and the reconstructed signal. `plot` is not supported for code generation and must stay in the harness. To generate code from the harness that contains the visualization tools, rewrite the harness as a function and declare the visualization functions as extrinsic functions using `coder.extrinsic`. To run the generated code that contains the extrinsic functions, you must have MATLAB installed on your machine.

The MATLAB code in the `for` loop that reconstructs the original signal using high-energy FFT coefficients is the computationally intensive portion of this algorithm. Speed up the `for` loop by moving this computational part into a function of its own, `GenerateSignalWithHighEnergyFFTCoeffs.m`.

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
end
max(abs(Input-ReconstrSignal))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignal,'*')
hold off

function [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input)

ft = dsp.FFT('FFTImplementation','FFTW');
ift = dsp.IFFT('FFTImplementation','FFTW','ConjugateSymmetricInput',true);

FFTCoeff = ft(Input);
FFTCoeffMagSq = abs(FFTCoeff).^2;
L = size(Input,1);
EnergyF = (1/L)*sum(FFTCoeffMagSq);
[FFTCoeffSorted, ind] = sort(((1/L)*FFTCoeffMagSq),1,'descend');

CumFFTCoeffs = cumsum(FFTCoeffSorted);
EnergyPercent = (CumFFTCoeffs/EnergyF)*100;
Vec = find(EnergyPercent > 99.99);
FFTCoeffsModified = zeros(L,1);
FFTCoeffsModified(ind(1:Vec(1))) = FFTCoeff(ind(1:Vec(1)));
numCoeff = Vec(1);
ReconstrSignal = ift(FFTCoeffsModified);
end
```

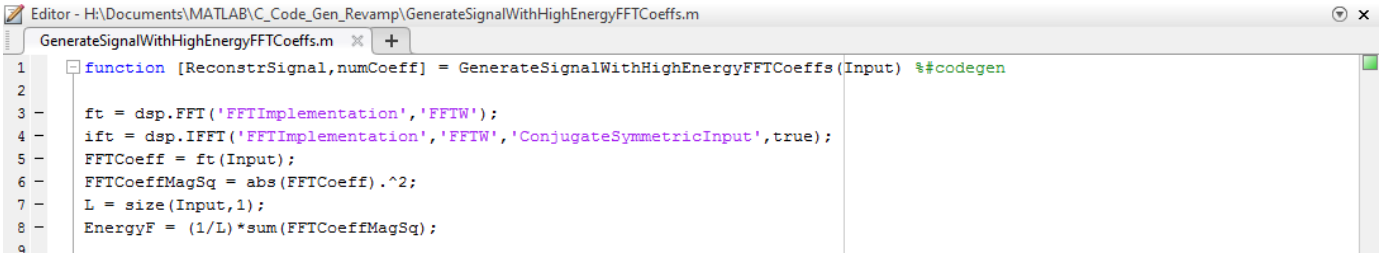
Make Code Suitable for Code Generation

Before you generate code, you must prepare your MATLAB code for code generation.

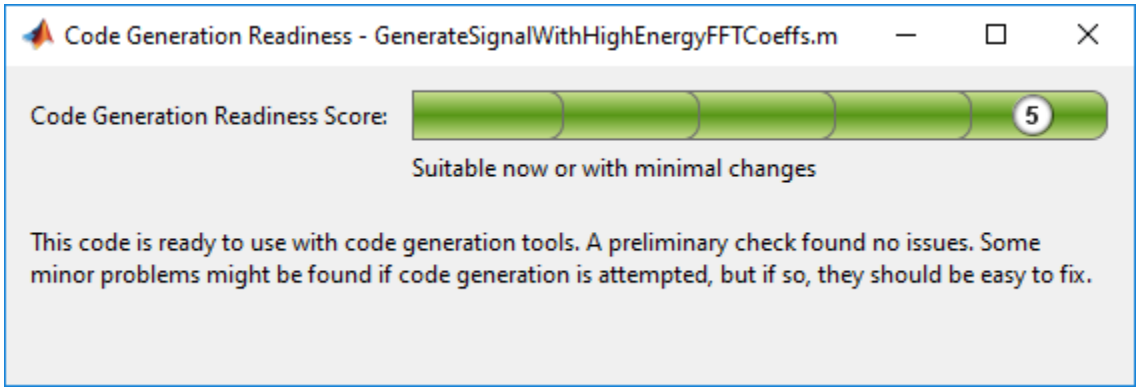
Check Issues at Design Time

The first step is to eliminate unsupported constructs and check for any code generation issues. For a list of Communications Toolbox features supported by MATLAB Coder, see Functions and System Objects Supported for C Code Generation. For a list of supported language constructs, see “MATLAB Language Features Supported for C/C++ Code Generation” (MATLAB Coder).

The code analyzer detects coding issues at design time as you enter the code. To enable the code analyzer, you must add the `%#codegen` pragma to your MATLAB file.



The code generation readiness tool screens MATLAB code for features that are not supported for code generation. One of the ways to access this tool is by right-clicking on the MATLAB file in its current folder. Running the code generation tool on `GenerateSignalWithHighEnergyFFTCoeffs.m` finds no issues.



Check Issues at Code Generation Time

Before you generate C code, ensure that the MATLAB code successfully generates a MEX function. The `codegen` command used to generate the MEX function detects any errors that prevent the code for being suitable for code generation.

Run `codegen` on `GenerateSignalWithHighEnergyFFTCoeffs.m` function.

```
codegen -args {Input} GenerateSignalWithHighEnergyFFTCoeffs
```

The following message appears in the MATLAB command prompt:

```
??? The left-hand side has been constrained to be non-complex, but the right-hand side
is complex. To correct this problem, make the right-hand side real using the function
REAL, or change the initial assignment to the left-hand side variable to be a complex
value using the COMPLEX function.

Error in ==> GenerateSignalWithHighEnergy Line: 24 Column: 1
Code generation failed: View Error Report
Error using codegen
```

This message is referring to the variable `FFTCoeffsModified`. The coder is expecting this variable to be initialized as a complex variable. To resolve this issue, initialize the `FFTCoeffsModified` variable as complex.

```
FFTCoeffsModified = zeros(L,1)+0i;
```

Rerun the `codegen` function and you can see that a MEX file is generated successfully in the current folder with a `.mex` extension.

```
codegen -args {Input} GenerateSignalWithHighEnergyFFTCoeffs
```

Check Issues at Run Time

Run the generated MEX function to see if there are any run-time issues reported. To do so, replace

```
[ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
```

with

```
[ReconstrSignalMex,numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input);
```

inside the harness.

The harness now looks like:

```
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignalMex,numCoeffMex] = GenerateSignalWithHighEnergyFFTCoeffs_mex(Input,L);
end
max(abs(Input-ReconstrSignalMex))
figure(1);
plot(Input)
hold on;
plot(ReconstrSignalMex,'*')
hold off
```

The code runs successfully, indicating that there are no run-time errors.

Compare the MEX Function with the Simulation

Notice that the harness runs much faster with the MEX function compared to the regular function. The reason for generating the MEX function is not only to detect code generation and run-time issues, but also to speed up specific parts of your algorithm. For an example, see “Signal Processing Algorithm Acceleration in MATLAB”.

You must also check that the numeric output results from the MEX and the regular function match. Compare the reconstructed signal generated by the `GenerateSignalWithHighEnergyFFTCoeffs.m` function and its MEX counterpart `GenerateSignalWithHighEnergyFFTCoeffs_mex`.

```
max(abs(ReconstrSignal-ReconstrSignalMex))

ans =

     2.2204e-16
```

The results match very closely, confirming that the code generation is successful.

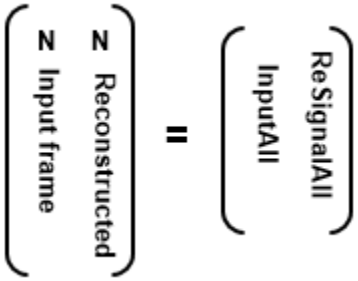
Generate a Standalone Executable

If your goal is to run the generated code inside the MATLAB environment, your build target can just be a MEX function. If deployment of code to another application is the goal, then generate a standalone executable from the entire application. To do so, the harness must be a function that calls the subfunction `GenerateSignalWithHighEnergyFFTCoeffs`. Rewrite the harness as a function.

```
function reconstructSignalTestbench()
L = 1020;
```

```
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
rng(1);
numIter = 1000;
for Iter = 1:numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeff] = GenerateSignalWithHighEnergyFFTCoeffs(Input,L);
end
```

Log all 1000 frames of the input and reconstructed signal and the number of FFT coefficients used to reconstruct each frame of the signal. Write all this data to a binary file named `data.bin` using the `dsp.BinaryFileWriter` System object. This example logs the number of coefficients, which are scalar values, as the first element of each frame of the input signal and the reconstructed signal. The data to be written has a frame size of $M = L + 1$ and has a format that looks like this figure.



N is the number of FFT coefficients that represent 99.99% of the signal energy of the current input frame. The meta data of the binary file specifies this information. Release the binary file writer and close the binary file at the end.

The updated harness function, `reconstructSignalTestbench`, is shown here:

```
function reconstructSignalTestbench()
L = 1020;
Sineobject = dsp.SineWave('SamplesPerFrame',L,...
    'SampleRate',44100,'Frequency',1000);
header = struct('FirstElemInBothCols','Number of Coefficients',...
    'FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfw = dsp.BinaryFileWriter('data.bin','HeaderStructure',header);
numIter = 1000;

M = L+1;
ReSignalAll = zeros(M*numIter,1);
InputAll = zeros(M*numIter,1);
rng(1);

for Iter = 1 : numIter
    Sinewave1 = Sineobject();
    Input = Sinewave1 + 0.01*randn(size(Sinewave1));
    [ReconstrSignal,numCoeffs] = GenerateSignalWithHighEnergyFFTCoeffs(Input);
    InputAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;Input];
    ReSignalAll(((Iter-1)*M)+1:Iter*M) = [numCoeffs;ReconstrSignal];
end

bfw([InputAll ReSignalAll]);
release(bfw);
```

The next step in generating a C executable is to create a `coder.config` object for an executable and provide a `main.c` function to this object.

```
cfg =  coder.config('exe');
cfg.CustomSource = 'reconstructSignalTestbench_Main.c';
```

Here is how the `reconstructSignalTestbench_Main.c` function looks for this example.

```
/*
** reconstructSignalTestbench_main.c
*
* Copyright 2017 The MathWorks, Inc.
*/
#include <stdio.h>
#include <stdlib.h>

#include "reconstructSignalTestbench_initialize.h"
#include "reconstructSignalTestbench.h"
#include "reconstructSignalTestbench_terminate.h"

int main()
{
    reconstructSignalTestbench_initialize();
    reconstructSignalTestbench();
    reconstructSignalTestbench_terminate();

    return 0;
}
```

For additional details on creating the main function, see “Generating Standalone C/C++ Executables from MATLAB Code” (MATLAB Coder).

Set the `CustomInclude` property of the configuration object to specify the location of the main file. In this example, the location is the current folder.

```
cfg.CustomInclude = ['"',pwd,'"'];
```

Generate the C executable by running the following command in the MATLAB command prompt:

```
codegen -config cfg -report reconstructSignalTestbench
```

MATLAB Coder compiles and links the main function with the C code that it generates from the `reconstructSignalTestbench.m`.

If you are using Windows, you can see that `reconstructSignalTestbench.exe` is generated in the current folder. If you are using Linux, the generated executable does not have the `.exe` extension.

Read and Verify the Binary File Data

Running the executable creates a binary file, `data.bin`, in the current directory and writes the input, reconstructed signal, and the number of FFT coefficients used to reconstruct the signal.

```
!reconstructSignalTestbench
```

You can read this data from the binary file using the `dsp.BinaryFileReader` object. To verify that the data is written correctly, read data from the binary file in MATLAB and compare the output with variables `InputAll` and `ReSignalAll`.

The header prototype must have a structure similar to the header structure written to the file. Read the data as two channels.

```
M = 1021;
numIter = 1000;
headerPro = struct('FirstElemInBothCols','Number of Coefficients',...
    'FirstColumn','Input','SecondColumn','ReconstructedSignal');
bfr = dsp.BinaryFileReader('data.bin','HeaderStructure',...
```

```
headerPro,'SamplesPerFrame',M*numIter,'NumChannels',2);
Data = bfr();
```

Compare the first channel with `InputAll` and the second channel with `ReSignalAll`.

```
isequal(InputAll,Data(:,1))

ans =

  logical

   1

isequal(ReSignalAll,Data(:,2))

ans =

  logical

   1
```

The results match exactly, indicating a successful write operation.

Relocate Code to Another Development Environment

Once you generate code from your MATLAB algorithm, you can relocate the code to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB. You can package the files into a compressed file using the `packNGo` function at the command line or the **Package** option in the MATLAB Coder app. For an example that illustrates both the workflows, see “Package Code for Other Development Environments” (MATLAB Coder). For more information on the `packNGo` option, see `packNGo` in “RTW.BuildInfo Methods” (MATLAB Coder). You can relocate and unpack the compressed zip file using a standard zip utility. For an example on how to package the executable generated in this example, see “Relocate Code Generated from MATLAB Code to Another Development Environment”.

See Also

Functions

`codegen`

More About

- “Relocate Code Generated from MATLAB Code to Another Development Environment”
- “Generate C Code from Simulink Model”
- “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder)
- “Generate C Code at the Command Line” (MATLAB Coder)
- “Code Generation Workflow” (MATLAB Coder)

External Websites

- Supported and Compatible Compilers

Generate C Code from Simulink Model

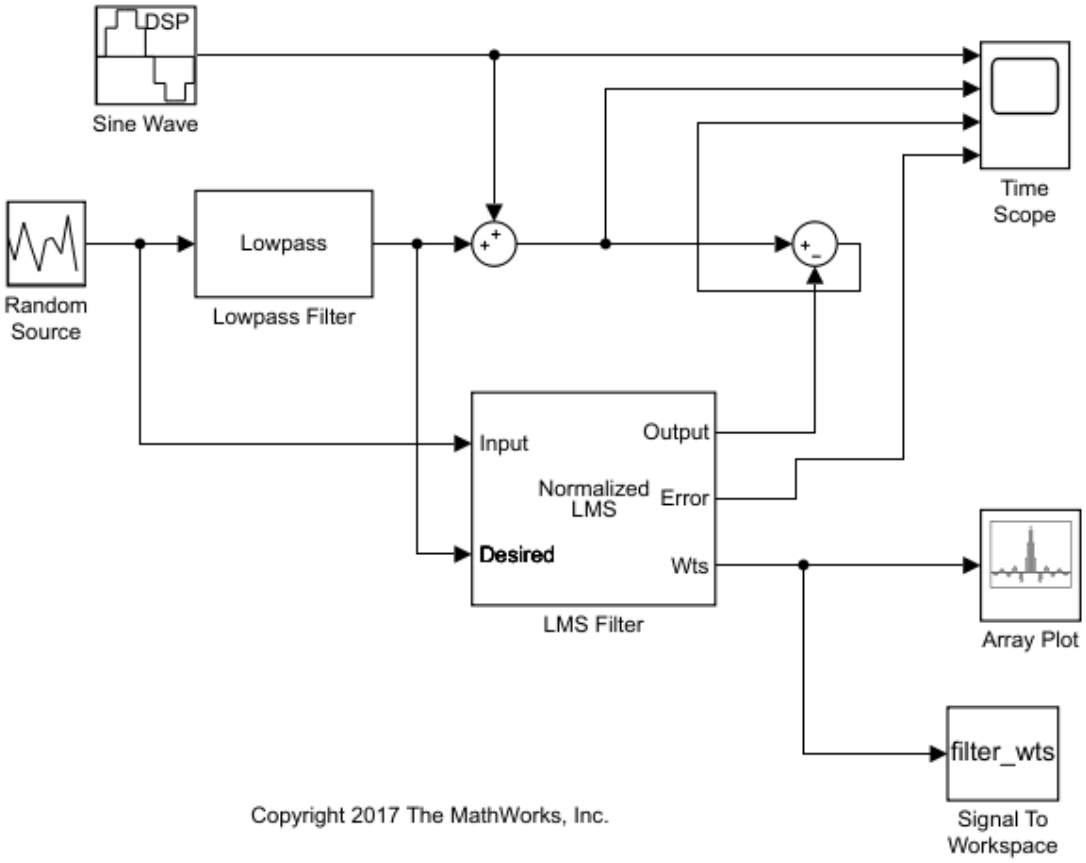
Simulink Coder generates standalone C and C++ code from Simulink models for deployment in a wide variety of applications. The workflow described in this topic uses DSP System Toolbox features but the same workflow applies for Communications Toolbox. For a list of Communications Toolbox features supported by Simulink Coder, see Blocks Supported for C Code Generation.

This example generates C code from the `ex_codegen_dsp` model and builds an executable from the generated code. You can run the executable inside the MATLAB environment. In addition, you can package and relocate the code to another development environment that does not have the MATLAB and Simulink products installed.

Open the Model

The `ex_codegen_dsp` model implements a simple adaptive filter to remove noise from a signal while simultaneously identifying a filter that characterizes the noise frequency content. To open this model, enter the following command in MATLAB command prompt:

```
open_system('ex_codegen_dsp')
```



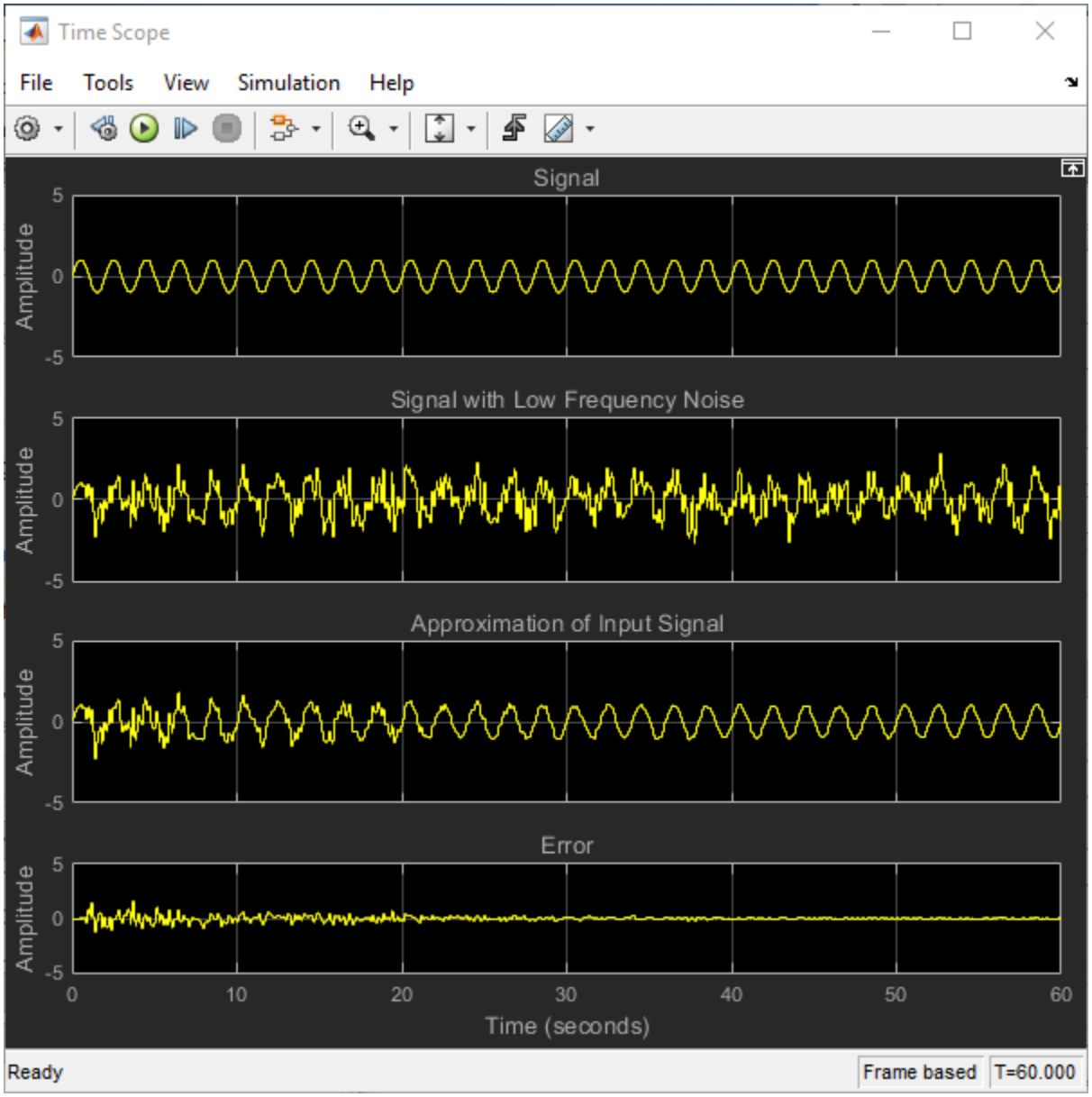
You can alternatively create the model using the **DSP System** template. For more information, see “Configure the Simulink Environment for Signal Processing Models”.

Configure Model for Code Generation

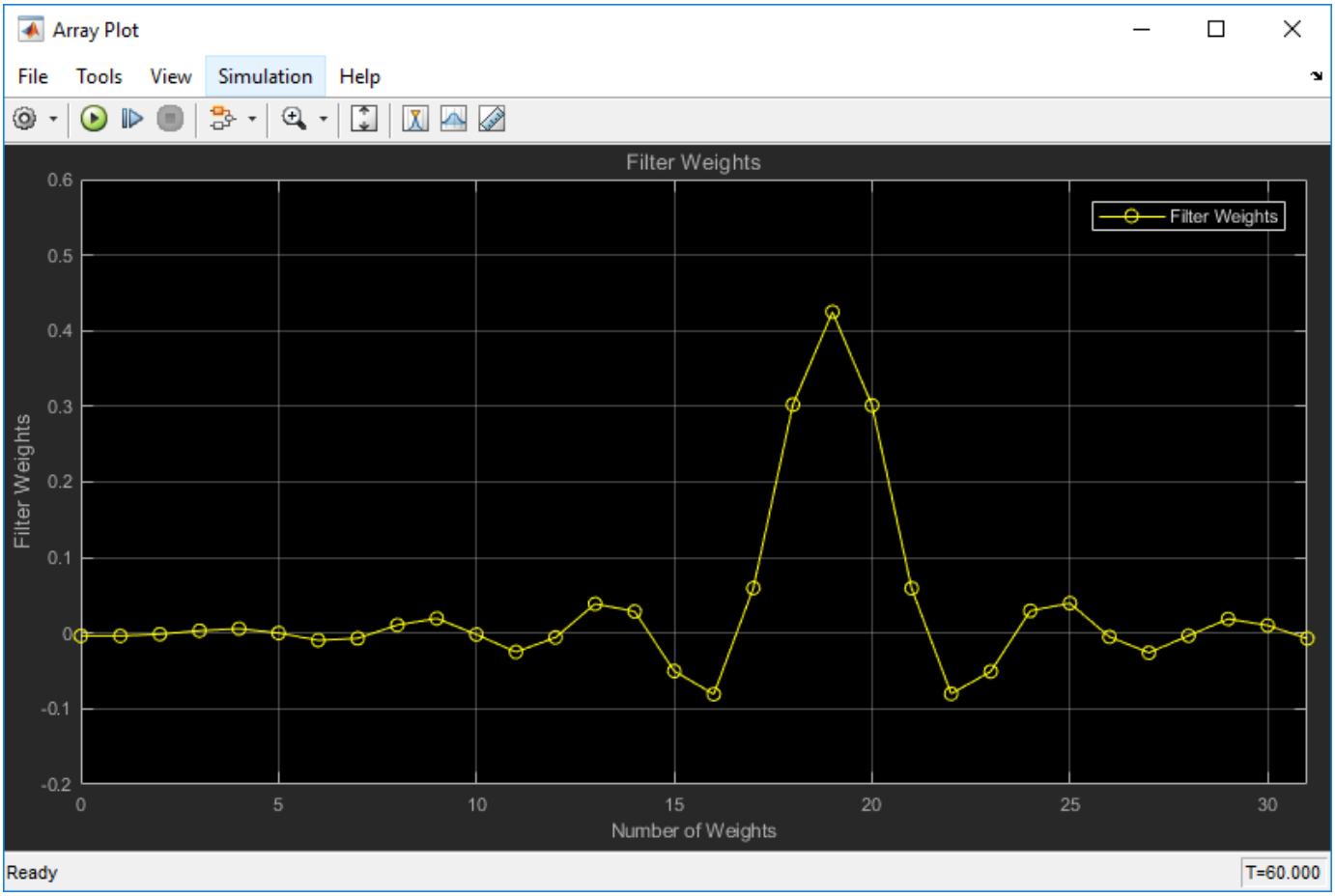
Prepare the model for code generation by specifying code generation settings in the **Configuration Parameters** dialog box. Choose the appropriate solver and code generation target, and check the model configuration for execution efficiency. For more details on each of these steps, see “Generate C Code for a Model” (Simulink Coder).

Simulate the Model

Simulate the model. The Time Scope shows the input and filtered signal characteristics.



The Array Plot shows the last 32 filter weights for which the LMS filter has effectively adapted and filtered out the noise from the signal.



These coefficients can also be accessed using the following command:

```
filter_wts(:,:,1201)
```

Generate Code from the Model

Before you generate code from the model, you must first ensure that you have write permission in your current folder.

To generate code, you must make the following changes:

1. In the **Modeling** tab of the model toolstrip, click **Model Settings**. The **Configuration Parameters** dialog opens. Navigate to the **Code Generation** tab, select the **Generate code only** parameter, and click **Apply**.
2. In the Apps gallery, click **Simulink Coder**. The **C Code** tab appears. Click the **Generate Code** icon ( ).

After the model finishes generating code, the **Code Generation Report** appears, allowing you to inspect the generated code. Note that the build process creates a new subfolder called `ex_codegen_dsp_grt_rtw` in your current MATLAB working folder. This subfolder contains all the files created by the code generation process, including those that contain the generated C source

code. For more information on viewing the generated code, see “Generate C Code for a Model” (Simulink Coder).

Build and Run the Generated Code

Set Up the C/C++ Compiler

To build an executable, you must set up a supported C compiler. For a list of compilers supported in the current release, see Supported and Compatible Compilers.

To set up your compiler, run the following command in the MATLAB command prompt:

```
mex –setup
```

Build the Generated Code

After your compiler is setup, you can build and run the compiled code. The ex_codegen_dsp model is currently configured to generate code only. To build the generated code, you must first make the following changes:

1. In the **Modeling** tab of the model toolstrip, click **Model Settings**. The **Configuration Parameters** dialog opens. Navigate to the **Code Generation** tab, clear the **Generate code only** parameter, and click **Apply**.
2. In the **C Code** tab of the model toolstrip, click the **Build** icon ().

The code generator builds the executable and generates the **Code Generation Report**. The code generator places the executable in the working folder. On Windows, the executable is `ex_codegen_dsp.exe`. On Linux, the executable is `ex_codegen_dsp`.

Run the Generated Code

To run the generated code, enter the following command in the MATLAB command prompt:

```
!ex_codegen_dsp
```

Running the generated code creates a MAT-file that contains the same variables as those generated by simulating the model. The variables in the MAT-file are named with a prefix of `rt_`. After you run the generated code, you can load the variables from the MAT-file by typing the following command at the MATLAB prompt:

```
load ex_codegen_dsp.mat
```

You can now compare the variables from the generated code with the variables from the model simulation. To access the last set of coefficients from the generated code, enter the following in the MATLAB prompt:

```
rt_filter_wts(:,:,1201)
```

Note that the coefficients in `filter_wts(:,:,1201)` and `rt_filter_wts(:,:,1201)` match.

For more details on building and running the executable, see “Generate C Code for a Model” (Simulink Coder).

Relocate Code to Another Development Environment

Once you generate code from your Simulink model, you can relocate the code to another development environment using the pack-and-go utility. Use this utility when the development environment does not have the MATLAB and Simulink products.

The pack-and-go utility uses the tools for customizing the build process after code generation and a `packNGo` function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

You can package the code by either using the user interface or by using the command-line interface. The command-line interface provides more control over the details of code packaging. For more information on each of these methods, see “Relocate Code to Another Development Environment” (Simulink Coder).

For an example on how to package the C code and executable generated from this example, see “Relocate Code Generated from a Simulink Model to Another Development Environment”.

See Also

More About

- “Generate C Code for a Model” (Simulink Coder)
- “Relocate Code Generated from a Simulink Model to Another Development Environment”
- “Relocate Code to Another Development Environment” (Simulink Coder)
- “Generate C Code from MATLAB Code”
- “How To Run a Generated Executable Outside MATLAB”

External Websites

- Supported and Compatible Compilers

HDL Code Generation

Find Blocks That Support HDL Code Generation

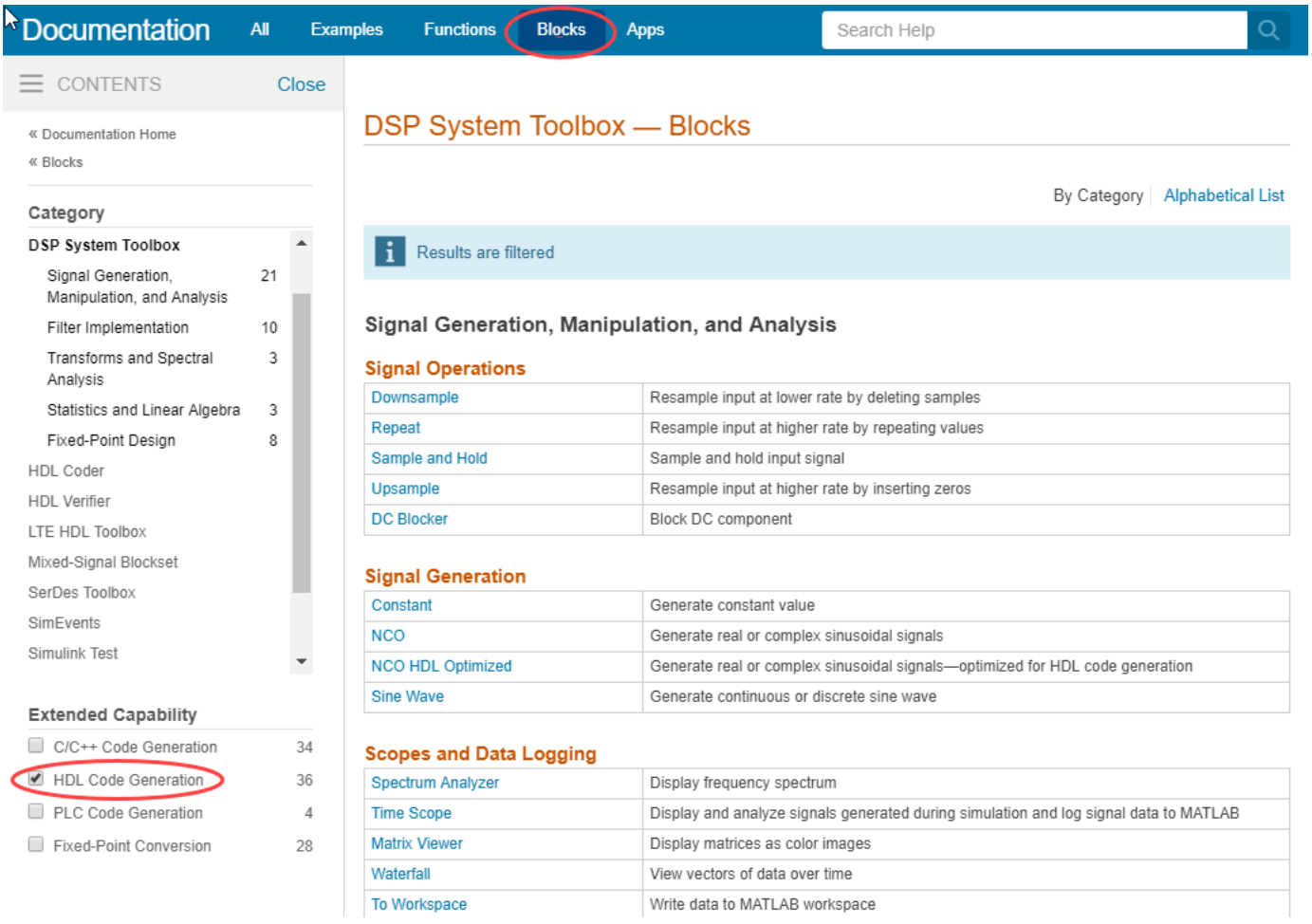
Blocks

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the **HDL Coder**, **Communications Toolbox HDL Support**, **DSP System Toolbox HDL Support** block libraries, and others.

To create a library of HDL-supported blocks from all your installed products, enter `hdllib` at the MATLAB command line. This command requires an HDL Coder™ license.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click **Blocks** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.



You can also use Communications Toolbox blocks with blocks from Wireless HDL Toolbox. Wireless HDL Toolbox provides sample-based algorithms in Simulink for the design and implementation of 5G NR, LTE, and other wireless communications subsystems on FPGAs and ASICs.

System Objects

To find System objects supported for HDL code generation, see Predefined System Objects (HDL Coder).

Wireless Communications Design for FPGAs and ASICs

In this section...

Deploying algorithmic models to FPGA hardware makes it possible to do over-the-air testing and verification. However, designing wireless communications systems for hardware requires design tradeoffs between hardware resources and throughput. You can speed up hardware design and deployment by using HDL-optimized blocks that have hardware-suitable interfaces and architectures, reference applications that implement portions of the LTE and 5G NR physical layer, and automatic HDL code generation. You can also use hardware support packages to assist with deploying and verifying your design on real hardware.

MathWorks® HDL products, such as Wireless HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from LTE Toolbox™ or 5G Toolbox™, and design a hardware implementation of that algorithm that is suitable for FPGAs and ASICs.

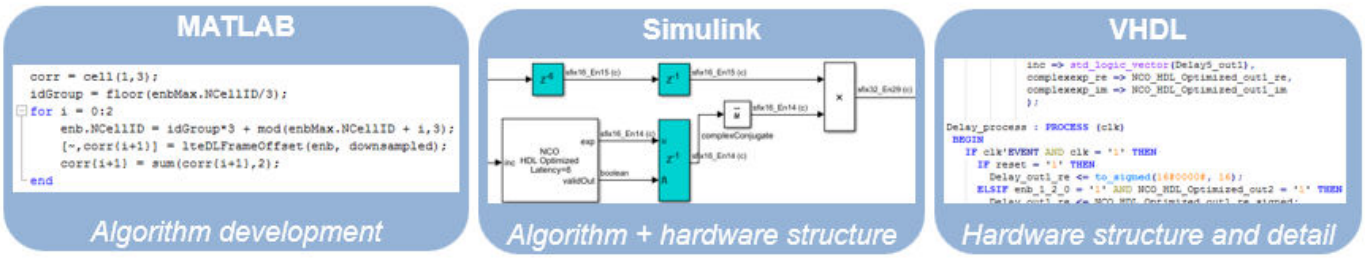
From Mathematical Algorithm to Hardware Implementation

Wireless communications design often starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

Hardware engineers typically receive a mathematical specification from an algorithm team, and reimplement the algorithm for hardware. Hardware designs require tradeoffs of resource usage for clock speed and overall throughput. Usually this tradeoff means operating on streaming data, and using some logic to control the storage and flow of data. Hardware engineers usually work in hardware description languages (HDLs), like VHDL and Verilog, that provide cycle-based modeling and parallelism.

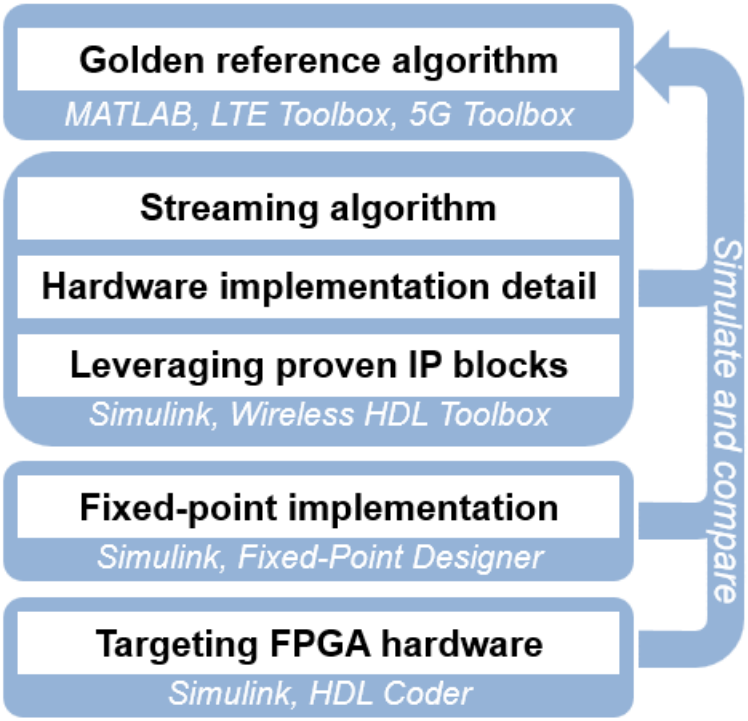
To bridge this gap between mathematical algorithm and hardware implementation, use the MATLAB algorithm model as a starting point for hardware implementation. Make incremental changes to the design to make it suitable for hardware, and progress towards a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardware-compatible implementation in Simulink, and then the generated VHDL code.

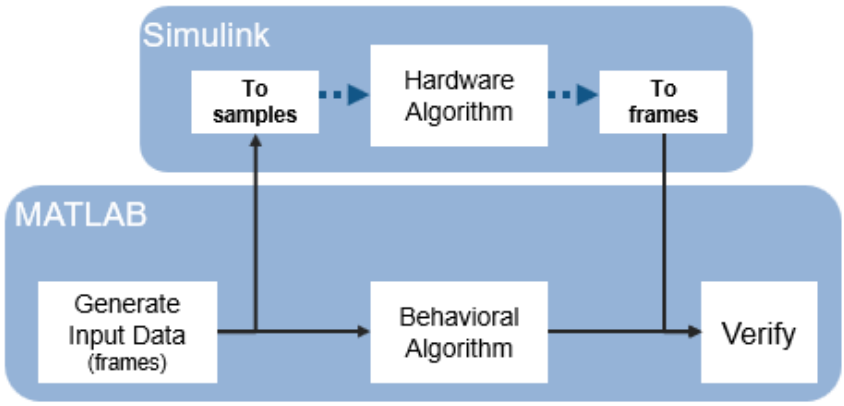


While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing of the data stream. To aid in fixed-point type choices, it clearly visualizes data type propagation in the design. It also allows for easy pipelining of mathematical operations to improve maximum clock frequency in hardware.

While you create your hardware-ready design, use the MATLAB algorithm as a "golden reference" to verify that each version of the design still meets the mathematical requirements. The workflow shown in the diagram uses MATLAB and Simulink as collaboration and communication tools between the algorithm and hardware design teams.



For instance, when designing for LTE or 5G wireless standards, you can use LTE Toolbox and 5G Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from Wireless HDL Toolbox and blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation. You can reuse test and data generation infrastructure from MATLAB by importing data from MATLAB to your Simulink model and returning the output of the model to MATLAB to verify it against the "golden reference".



HDL-Optimized Blocks

Library blocks from Wireless HDL Toolbox implement encoders, decoders, modulators, demodulators, and sequence generators for use in an LTE, 5G, or general wireless communications system. These blocks use a standard streaming data interface for hardware. This interface makes it easy to connect parts of the algorithm together, and includes control signals that manage the flow of data and mark frame boundaries. These blocks support automatic HDL code generation with HDL Coder. You can also use blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation.

The blocks provide hardware-suitable architectures that optimize resource use, such as including adder and multiplier pipelining to fit well into FPGA DSP slices. They also support automatic and configurable fixed-point data types. Using predefined blocks also allows you to try different parameter configurations without changing the rest of the design.

For lists of blocks that support HDL code generation, see Wireless HDL Toolbox Block List (HDL Code Generation), Communications Toolbox Block List (HDL Code Generation), and DSP System Toolbox Block List (HDL Code Generation).

Reference Applications

Wireless HDL Toolbox provides reference applications that contain hardware-ready implementations of large parts of the LTE and 5G NR physical layer. These designs are verified against the "golden reference" functions provided by LTE Toolbox and 5G Toolbox. They have also been tested on FPGA boards to confirm that they encode and decode over-the-air waveforms and use a reasonable amount of hardware resources. They are designed to be modular, scalable, and extensible so you can insert additional physical channels. The receiver design was tested using waveforms captured off-the-air.

The suite of reference applications includes:

- LTE and 5G NR primary and secondary synchronization signal (PSS/SSS) generation and detection
- LTE downlink shared control channel detector and master information block (MIB) generation and recovery
- LTE first system information block (SIB1) decoder
- Hardware-software interface models for MIB and SIB1 bit parsing and channel estimation data indexing

- LTE waveform generation for multiple-antenna transmission
- Support for FDD and TDD for LTE transmitter and receiver applications

These reference applications can be used as-is to deliver packet information to your unique application and to generate synthesizable VHDL or Verilog with HDL Coder. They also serve as examples to illustrate recommended practices for implementing communications algorithms on FPGA or ASIC hardware.

Generate HDL Code and Prototype on FPGA

Wireless HDL Toolbox provides blocks that support HDL code generation. To generate HDL code from designs that use these blocks, you must have an HDL Coder license. HDL Coder produces device-independent code with signal names that correspond to the Simulink model. HDL Coder also provides a tool to drive the FPGA synthesis and targeting process, and enables you to generate scripts and test benches for use with third-party HDL simulators.

To assist with the setup and targeting of programmable logic on a prototype board, and to verify your wireless communications system design on hardware, download a hardware support package such as Communications Toolbox Support Package for Xilinx® Zynq®-Based Radio.

See Also

External Websites

- Wireless HDL Toolbox
- HDL Coder

Simulation Acceleration

Simulation Acceleration Using GPUs

In this section...

GPU-Based System objects

GPU-based System objects look and behave much like the other System objects in the Communications Toolbox product. The important difference is that the algorithm is executed on a Graphics Processing Unit (GPU) rather than on a CPU. Using the GPU can accelerate your simulation.

System objects for the Communications Toolbox product are located in the `comm` package and are constructed as:

```
H = comm.<object name>
```

For example, a Viterbi Decoder System object is constructed as:

```
H = comm.ViterbiDecoder
```

In cases where a corresponding GPU-based implementation of a System object exists, they are located in the `comm.gpu` package and constructed as:

```
H = comm.gpu.<object name>
```

For example, a GPU-based Viterbi Decoder System object is constructed as:

```
H = comm.gpu.ViterbiDecoder
```

To see a list of available GPU-based implementations enter `help comm` at the MATLAB command line and click **GPU Implementations**.

General Guidelines for Using GPUs

Graphics Processing Units (GPUs) excel at processing large quantities of data and performing computations with high compute intensity. Processing large quantities of data is one way to maximize the throughput of your GPU in a simulation. The amount of the data that the GPU processes at any one time depends on the size of the data passed to the input of a GPU System object. Therefore, one way to maximize this data size is by processing multiple frames of data.

You can use a single GPU System object to process multiple data frames simultaneously or in parallel. This differs from the way many of the standard, or non-GPU, System objects are implemented. For

GPU System objects, the number of frames the objects process in a single call to the object function is either implied by one of the object properties or explicitly stated using the `NumFrames` property on the objects.

Transmit and decode using BPSK modulation and turbo coding

This example shows how to transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, it shows how to 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))
```

Process Multiple Data Frames Using a GPU

This example shows how to simultaneously process two data frames using an LDPC Decoder System object. The `ParityCheckMatrix` property determines the frame size. The number of frames that the object processes is determined by the frame size and the input data vector length.

```
numframes = 2;

ldpcEnc = comm.LDPCEncoder;
ldpcGPUDec = comm.gpu.LDPCDecoder;
ldpcDec = comm.LDPCDecoder;


msg = randi([0 1], 32400,2);

for ii=1:numframes,
    encout(:,ii) = ldpcEnc(msg(:,ii));
end

%single ended to bipolar (for LLRs)
encout = 1-2*encout;

%Decode on the CPU
for ii=1:numframes;
    cout(:,ii) = ldpcDec(encout(:,ii));
end

%Multiframe decode on the GPU
gout = ldpcGPUDec(encout(:));

%check equality
isequal(gout,cout(:))
```

Process Multiple Data Frames Using NumFrames Property

This example shows how to process multiple data frames using the `NumFrames` property of the GPU-based Viterbi Decoder System object. For a Viterbi Decoder, the frame size of your system cannot be inferred from an object property. Therefore, the `NumFrames` property defines the number of frames present in the input data.

```
numframes = 10;

convEncoder = comm.ConvolutionalEncoder('TerminationMethod', 'Terminated');
vitDecoder = comm.ViterbiDecoder('TerminationMethod', 'Terminated');

%Create a GPU Viterbi Decoder, using NumFrames property.
vitGPUDecoder = comm.gpu.ViterbiDecoder('TerminationMethod', 'Terminated', ...
                                'NumFrames', numframes );

msg = randi([0 1], 200, numframes);

for ii=1:numframes,
    convEncOut(:,ii) = 1-2*convEncoder(msg(:,ii));
end
```

```
%Decode on the CPU
for ii=1:numframes;
    cVitOut(:,ii) = vitDecoder(convEncOut(:,ii));
end

%Decode on the GPU
gVitOut = vitGPUDecoder(convEncOut(:));

isequal(gVitOut,cVitOut(:))
```

gpuArray and Regular MATLAB Numerical Arrays

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).

Passing MATLAB arrays to a GPU System object requires transferring the initial data from a CPU to the GPU. Then, the GPU System object performs calculations and transfers the output data back to the CPU. This process introduces latency. When data in the form of a gpuArray is passed to a GPU System object, the object does not incur the latency from data transfer. Therefore, a GPU System object runs faster when you supply a gpuArray as the input.

In general, you should try to minimize the amount of data transfer between the CPU and the GPU in your simulation.

Pass gpuArray as an Input

This example shows how to pass a gpuArray to the input of the PSK modulator, reducing latency.

```
pskGPUModulator = comm.gpu.PSKModulator;
x = randi([0 7], 1000, 1, 'single');
gx = gpuArray(x);

o = pskGPUModulator(x);
class(o)

release(pskGPUModulator); %allow input types to change

go = pskGPUModulator(gx);
class(go)
```

System Block Support for GPU System Objects

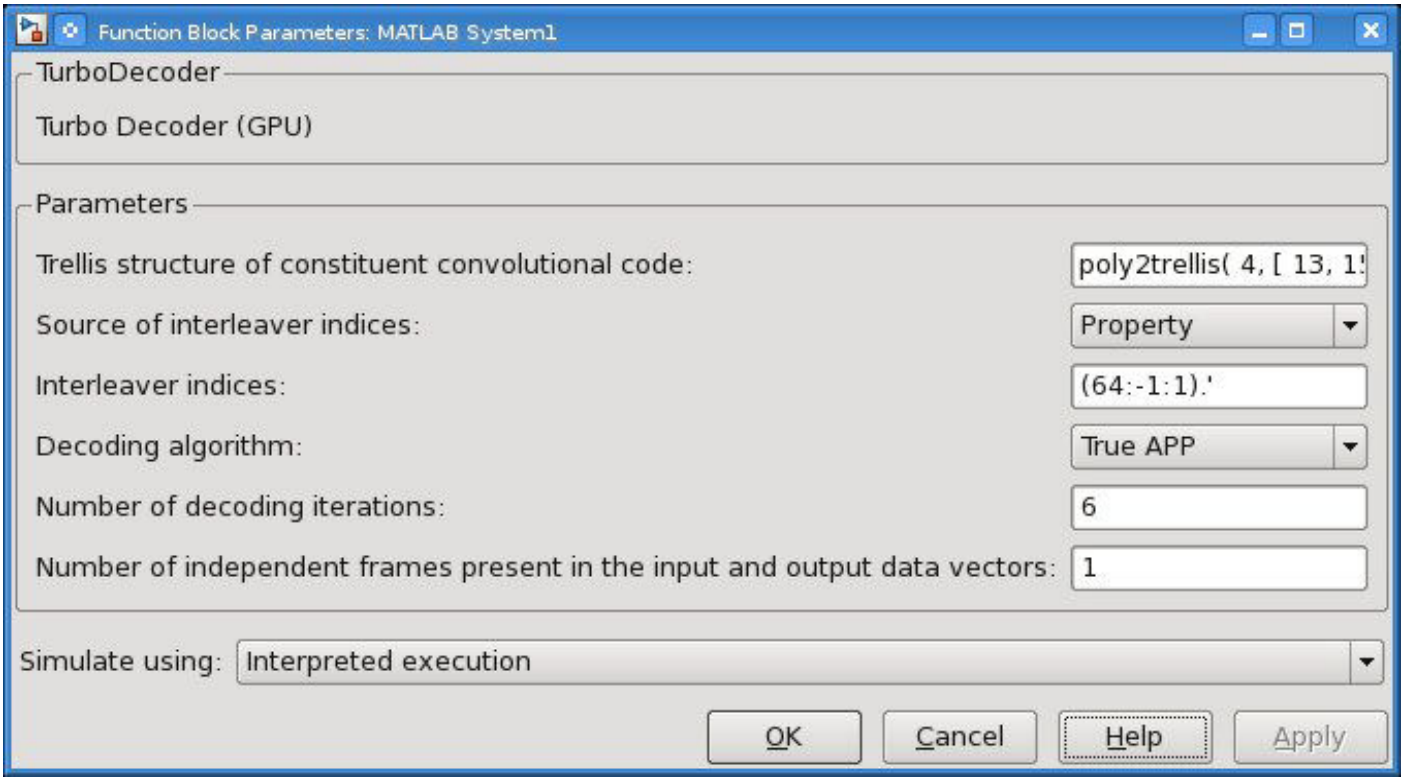
- “GPU System Objects Supported in System Block” on page 28-6
- “System Block Limitations for GPU System Objects” on page 28-6

GPU System Objects Supported in System Block

- `comm.gpu.AWGNChannel`
- `comm.gpu.BlockDeinterleaver`
- `comm.gpu.BlockInterleaver`
- `comm.gpu.ConvolutionalDeinterleaver`
- `comm.gpu.ConvolutionalEncoder`
- `comm.gpu.ConvolutionalInterleaver`
- `comm.gpu.PSKDemodulator`
- `comm.gpu.PSKModulator`
- `comm.gpu.TurboDecoder`
- `comm.gpu.ViterbiDecoder`

System Block Limitations for GPU System Objects

The GPU System objects must be simulated using `Interpreted Execution`. You must select this option explicitly on the block mask; the default value is `Code generation`.



See Also

More About

- “GPU Capabilities and Performance” (Parallel Computing Toolbox)

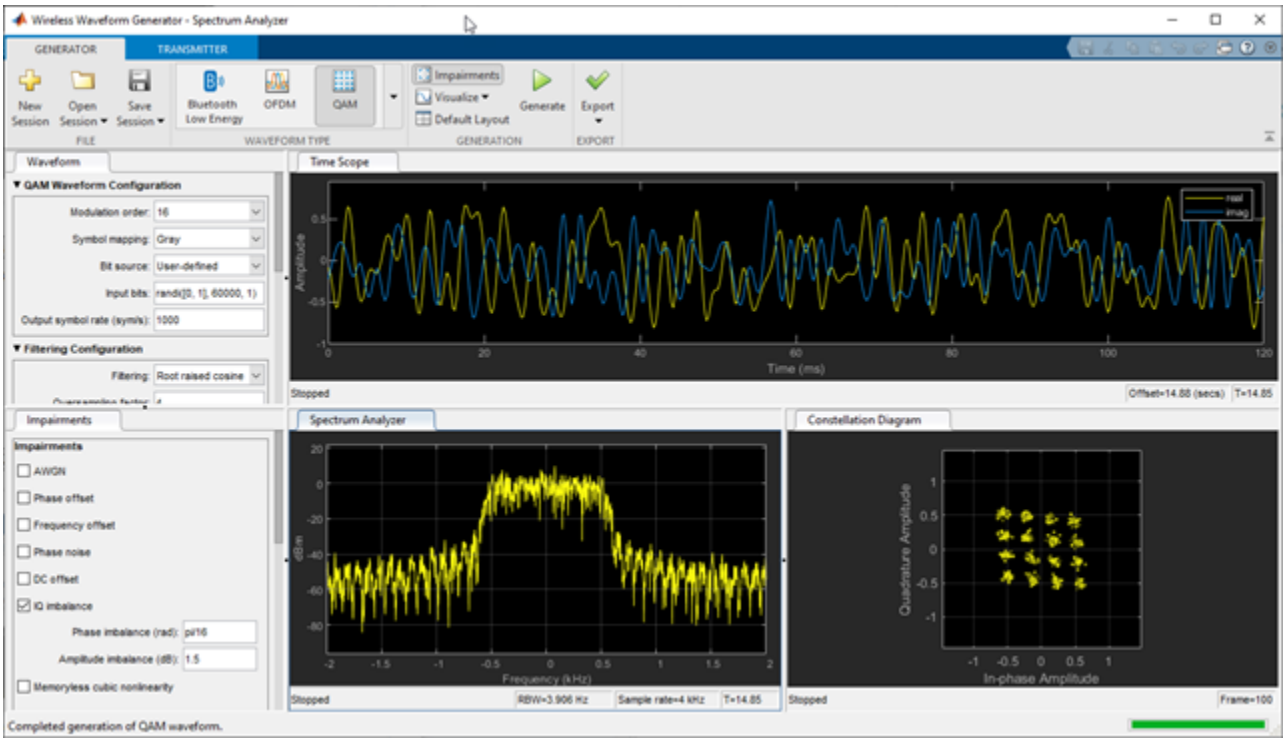
Wireless Waveform Generator App

Use Wireless Waveform Generator App

The **Wireless Waveform Generator** app is an interactive tool for creating, impairing, visualizing, and exporting modulated waveforms.

Start the app. On the **Apps** tab in the MATLAB toolstrip, under **Signal Processing and Communications**, click the app icon 

. You can also start the app by entering `wirelessWaveformGenerator` at the MATLAB command prompt.



These are the typical workflows when using the **Wireless Waveform Generator** app.

- Generate a waveform.
  - Select the desired waveform type from the options available in the **Waveform Type** section. Adjust the configuration parameters in the **Waveform** pane. For more information, see “Waveform Type” on page 29-3.
  - Select **Impairments** in the **Generation** section to open the **Impairments** pane. Adjust the configuration parameters in the **Impairments** pane. For more information, see “Add Impairments” on page 29-3.
  - To perform signal generation, click **Generate**. After generation, the waveform is displayed. You can adjust the waveform, filtering, and impairment configurations and then regenerate the waveform.
  - The default visualization plots vary based on the waveform type selected. Additional “Visualization Options” on page 29-4 can be opened by selecting them from **Visualize** in the **Generation** section.

- After generating a waveform you can export it by selecting the desired options from **Export** in the **Export** section. For more information, see “Export Waveform” on page 29-4.
- You can save the current session, open a previously saved session, or open a new session by selecting the desired option in the **File** section. For more information, see “Waveform Generator Session” on page 29-5.

Waveform Type

To generate the various available waveforms, the **Wireless Waveform Generator** app uses Communications Toolbox features. The supported waveform types include:

- OFDM — The app uses the `comm.OFDMModulator` System object to generate this type of waveform.
- QAM — The app uses the `qammod` function to generate this type of waveform.
- PSK — The app uses the `pskmod` function to generate this type of waveform.
- Sinewave — The app uses the `dsp.SineWave` System object to generate this type of waveform.
- 5G — If you have the 5G Toolbox, you can also generate 5G NR waveforms using features in the “5G Toolbox”. For more information, see the **5G Waveform Generator** app reference page.
- LTE — If you have the LTE Toolbox you can also generate LTE modulated waveforms using features in the “LTE Toolbox”. For more information, see the **LTE Waveform Generator** app reference page.
- WLAN — If you have the WLAN Toolbox™ you can also generate 802.11™ modulated waveforms using features in the “WLAN Toolbox”. For more information, see the **WLAN Waveform Generator** app reference page.
- Bluetooth — You can download and install the Communications Toolbox Library for the Bluetooth Protocol add-on to generate waveforms using features described in “Bluetooth”.

By default, generated waveforms have no filtering applied. To apply filtering to the waveform, select the desired filter option from the **Filtering** parameter on the **Waveform** pane. The available filter options vary based on the waveform type you select.

Add Impairments

You can add these impairments to the waveform that you generate.

- AWGN — The app uses the `awgn` function to impair the waveform.
- Phase offset — The app impairs the waveform by applying the specified phase offset as $y = xe^{j\varphi}$, where φ is the phase offset in radians.
- Frequency offset — The app uses the `comm.PhaseFrequencyOffset` System object to impair the waveform.
- Phase noise — The app uses the `comm.PhaseNoise` System object to impair the waveform.
- DC offset — The app impairs the waveform by applying the specified DC offset as $y = x + dcOff$, where $dcOff$ is the complex DC offset in Volts.
- IQ imbalance — The app uses the `iqimbal` function to impair the waveform.
- Memoryless cubic nonlinearity — The app uses the `comm.MemorylessNonlinearity` System object to impair the waveform.

Visualization Options

You can use these plot types to visualize waveforms that you generate.

- Spectrum Analyzer — The app plots the waveform in the frequency domain.
- OFDM Grid — For OFDM waveforms, the app plots the resource allocation of data and control channels.
- Time scope — The app plots the inphase and quadrature (IQ) waveform samples in the time domain.
- Constellation diagram — The app plots the constellation points of the modulation symbols.

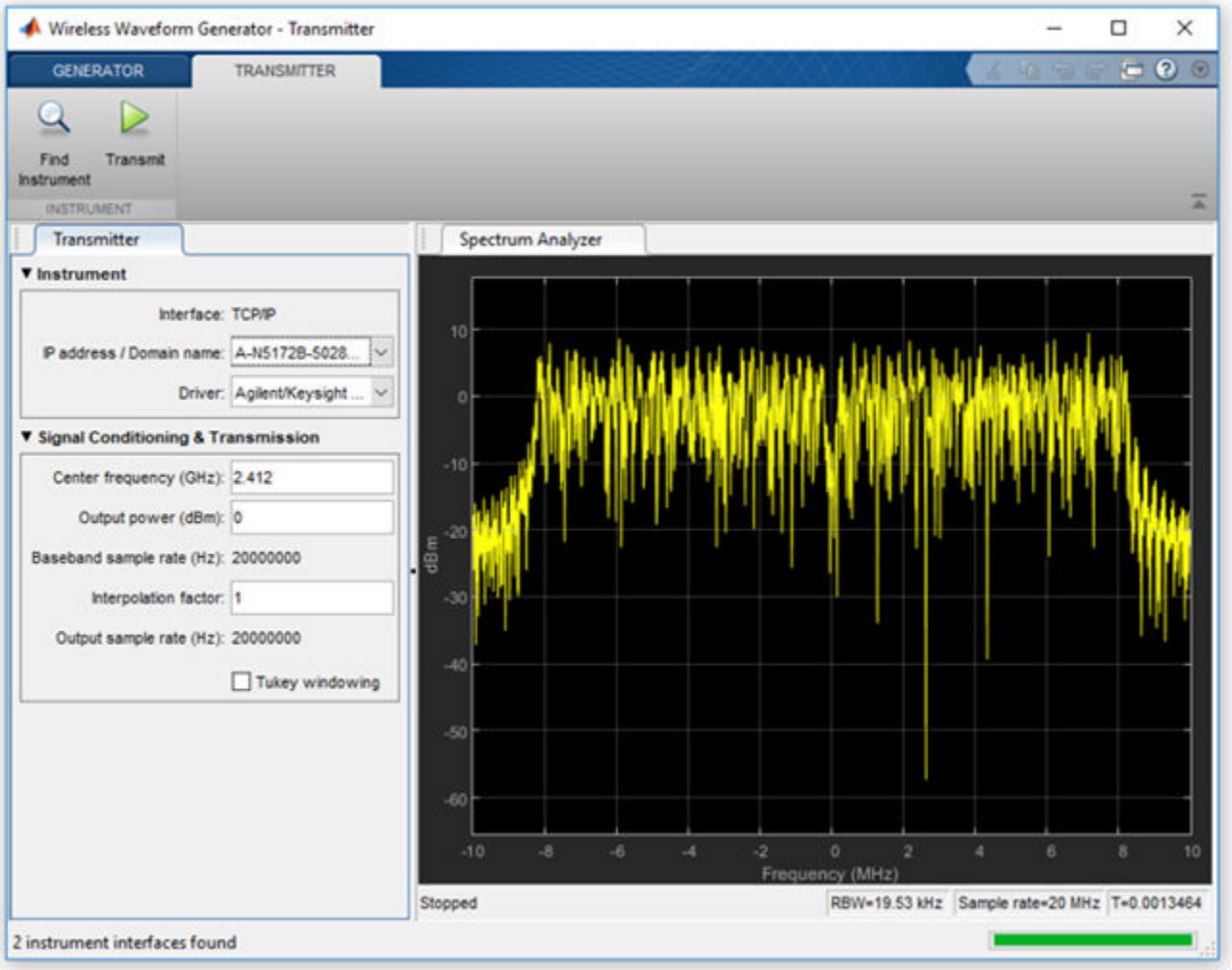
Export Waveform

You can export the waveform to a runnable MATLAB script or Simulink block., to your workspace, or to a signal file.

- Use the exported script to generate your waveform without the app from the command line.
- Use the exported block as a waveform source in a Simulink model. For more information, see Waveform From Wireless Waveform Generator App.
- Waveforms exported to the workspace are saved as a structure containing these fields:
  - `type` — This field is a character vector indicating the waveform type.
  - `config` — This field is a structure or object containing fields that specify the configured waveform type.
  - `Fs` — This field is the signal sample rate in Hertz.
  - `waveform` — The field is the complex waveform samples output as an N_S-by-1 column vector or an N_S-by-N_T matrix. N_S is the number of time-domain samples, and N_T is the number of transmit antennas.
- Waveforms exported to a signal file can be saved as a *.mat* or *.bb* file.
  - MAT-files are binary MATLAB files that store workspace variables. For more information, see “MAT-File Versions”.
  - The app uses the `comm.BasebandFileWriter` System object to save *.bb* files.

Transmit Using Lab Test Instrument

Generate a waveform that you can transmit using a connected lab test instrument. The **Wireless Waveform Generator** app can transmit using instruments supported by the `rfsiggen` function.



Use of the transmit feature in the **Wireless Waveform Generator** app requires “Instrument Control Toolbox”.

Waveform Generator Session

You can save the current session, open a previously saved session, or open a new session by selecting the desired option in the **FILE** section. When you save a waveform generator session, the session configuration is saved as a *.mat* file. For more information, see “MAT-File Versions”.

See Also

Apps
Wireless Waveform Generator

Blocks
Waveform From Wireless Waveform Generator App

Related Examples

- “Generate Wireless Waveform in Simulink Using App-Generated Block” on page 29-7

Generate Wireless Waveform in Simulink Using App-Generated Block

This example shows how to configure and use the block that is generated using the **Export to Simulink** capability that is available in the **Wireless Waveform Generator** app.

Introduction

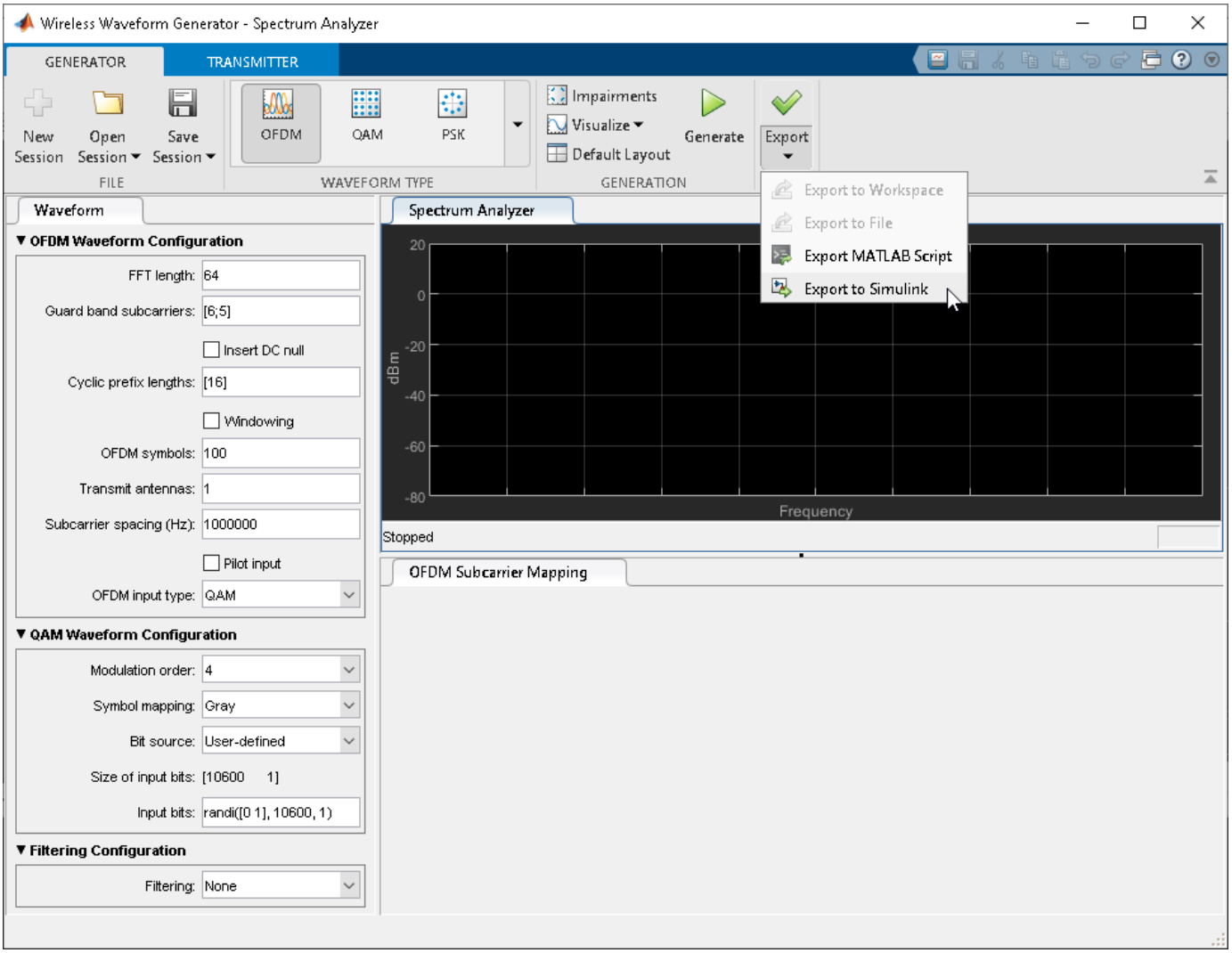
The **Wireless Waveform Generator** app is an interactive tool for creating, impairing, visualizing, and exporting waveforms. You can export the waveform to your workspace or to a `.mat` or `.bb` file. You can also export the waveform generation parameters to a runnable MATLAB® script or a Simulink® block. You can use the exported Simulink block to reproduce your waveform in Simulink. This example shows how to use the **Export to Simulink** capability of the app and how to configure the exported block to generate waveforms in Simulink.

Although this example focuses on exporting an OFDM waveform, the same process applies for all of the supported waveform types.

Export Wireless Waveform Configuration to Simulink

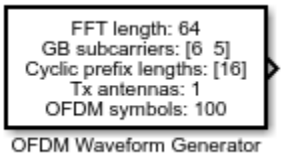
Open the **Wireless Waveform Generator** app by clicking the app icon on the **Apps** tab, under **Signal Processing and Communications**. Alternatively, enter `wirelessWaveformGenerator` at the MATLAB command prompt.

In the **Waveform Type** section, select an OFDM waveform by clicking **OFDM**. In the left-most pane of the app, adjust any configuration parameters for the selected waveform. Then export the configuration by clicking **Export** in the app toolstrip and selecting **Export to Simulink**.



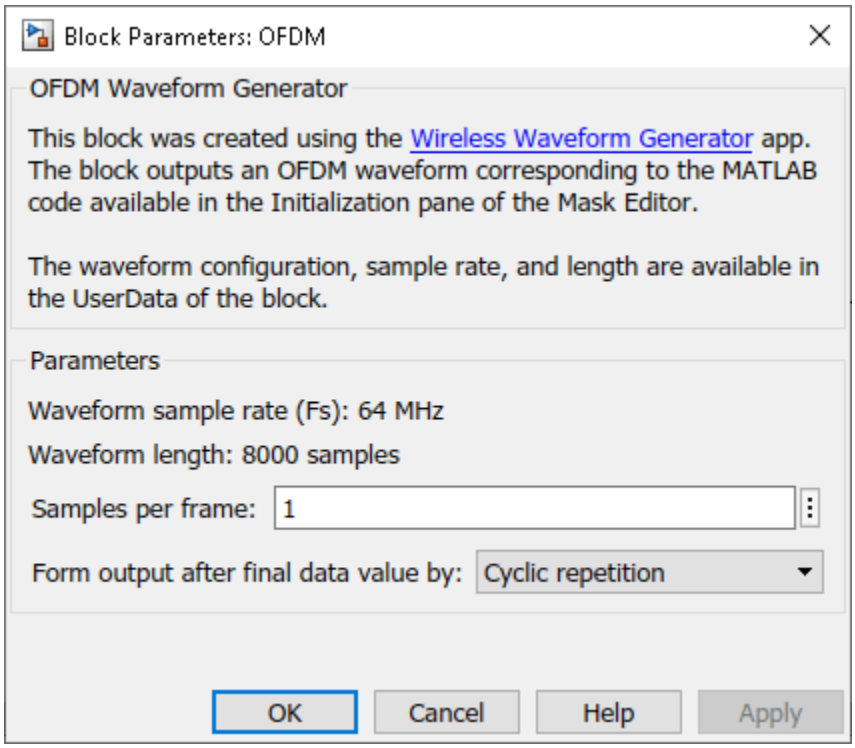
The **Export to Simulink** option creates a Simulink block, which outputs the selected waveform when you run the Simulink model. The block is exported to a new model if no open models exist.

```
modelName = 'WWGExport2SimulinkBlock';
open_system(modelName);
```



The **Form output after final data value by** block parameter specifies the output after all of the specified signal samples are generated. The value options for this parameter are `Cyclic repetition` and `Setting to zero`. The `Cyclic repetition` option repeats the signal from the beginning after it reaches the last sample in the signal. The `Setting to zero` option generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

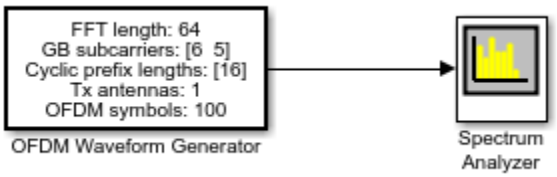
The **Waveform sample rate (Fs)** and **Waveform length** block parameters are derived from the waveform configuration that is available in the **Initialization** tab of the Mask Editor dialog box. For further information about the block parameters, see Waveform From Wireless Waveform Generator App. This figure shows the parameters of the exported block.




```
close_system(modelName);
```

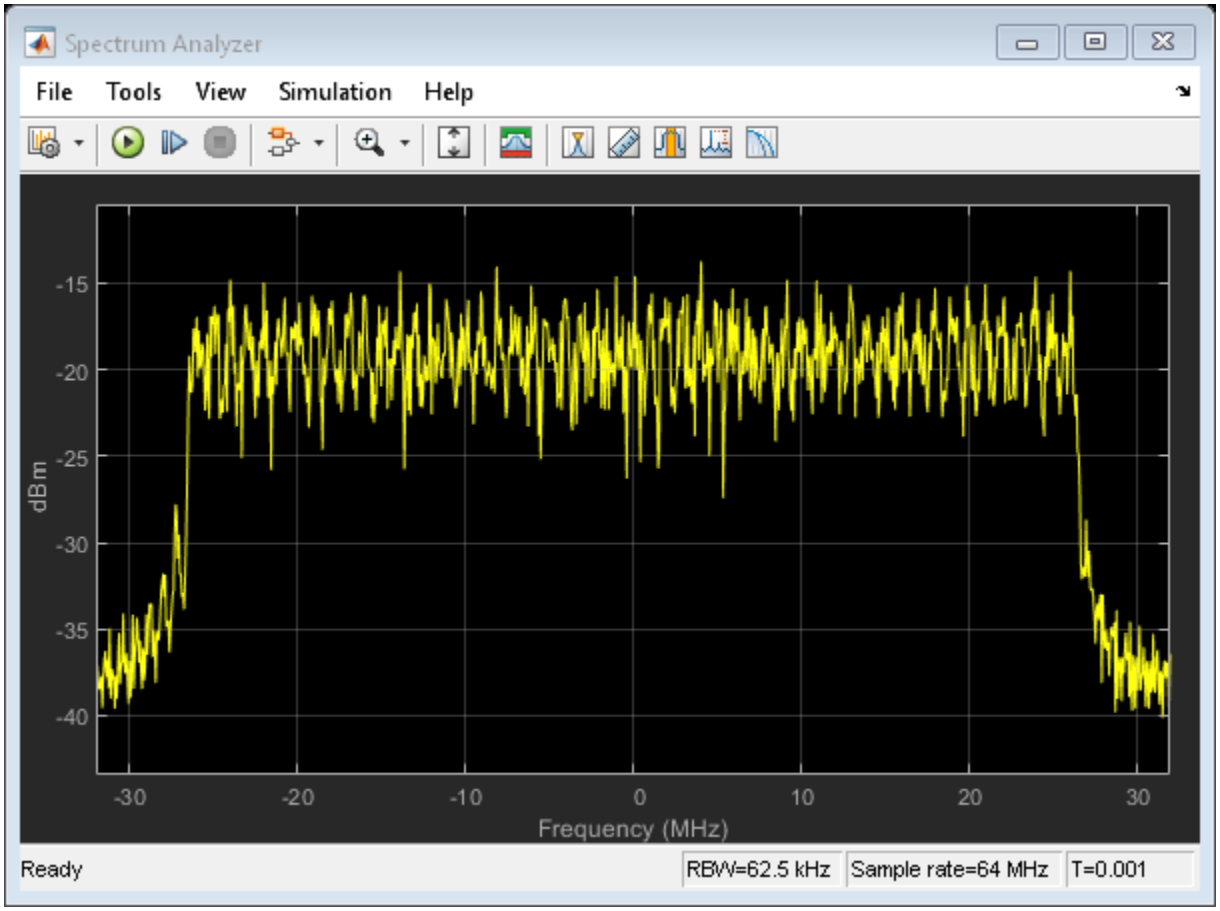
Connect a Spectrum Analyzer block to the exported block.

```
modelName = 'WWGExport2SimulinkModel';
open_system(modelName);
```



Simulate the model to visualize the waveform using the current configuration.

```
sim(modelName);
```



The Spectrum Analyzer block inherits the **Waveform sample rate (Fs)** parameter, which is 64 MHz.

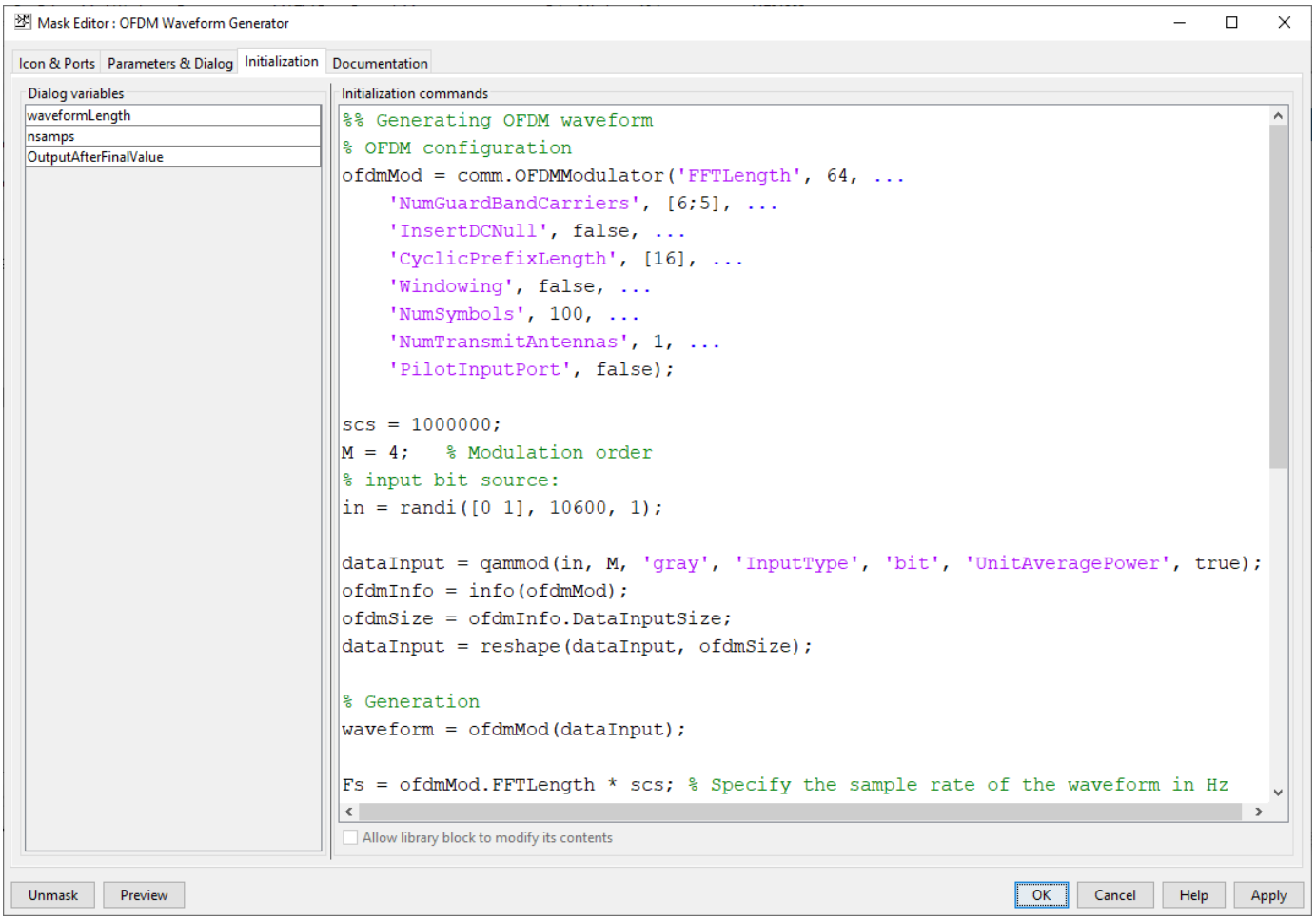
```
close_system(modelName);
```

Modify Wireless Waveform Configuration

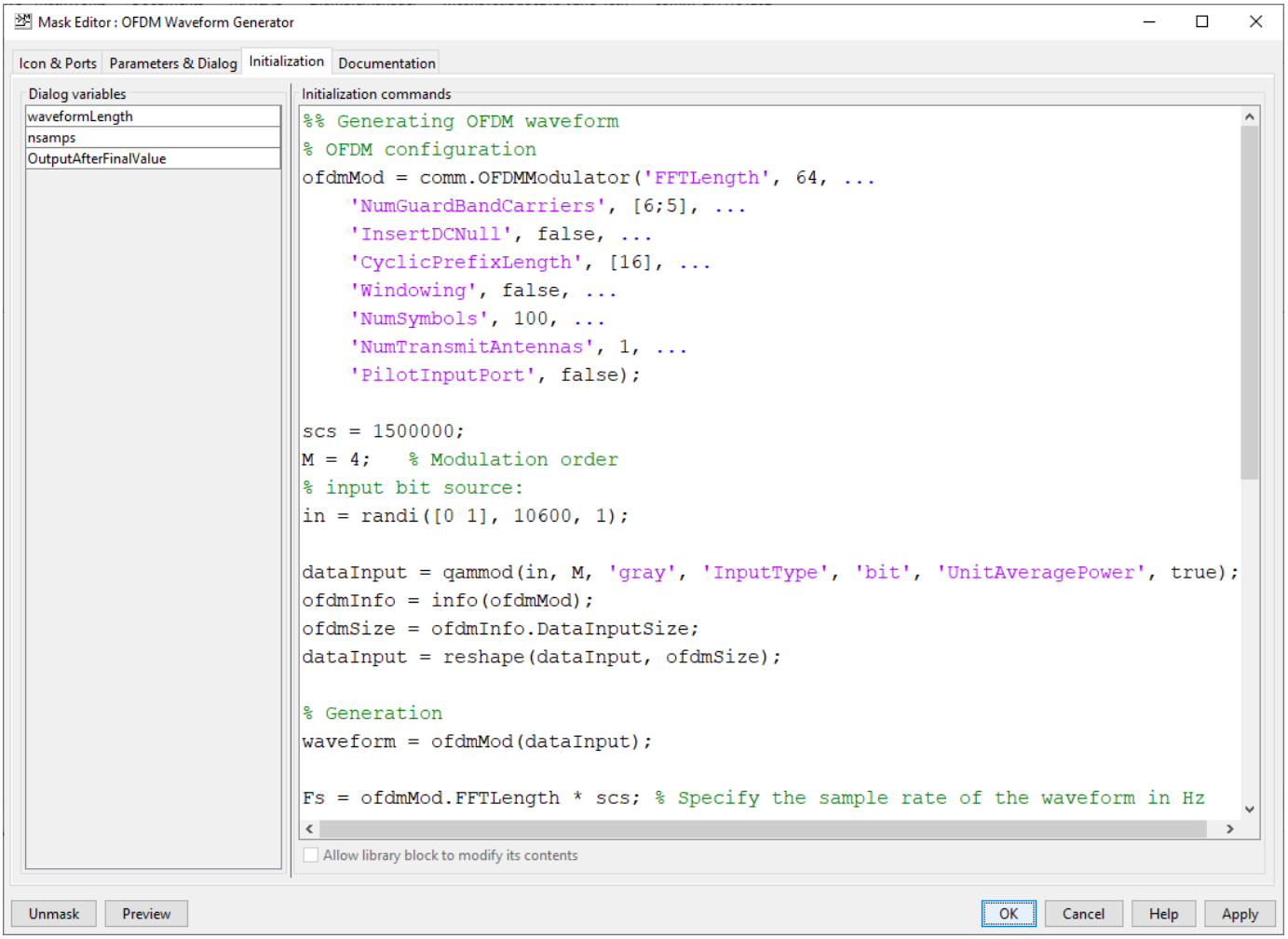
When you run the Simulink model, the exported block outputs the waveform generated in the **Initialization** tab of the Mask Editor dialog box for the block. The MATLAB code that initializes the waveform in this tab corresponds to the configuration that you selected in the **Wireless Waveform Generator** app before exporting the block. To modify the configuration of the waveform, choose one of these options:

- Open the **Wireless Waveform Generator** app, select the configuration of your choice, and export a new block. This option provides interaction with an app interface instead of MATLAB code, parameter range validation during the parameterization process, and visualization of the waveform before running the Simulink model.
- Update the configuration parameters that are available in the **Initialization** tab of the Mask Editor dialog box of the exported block. This option requires modifying the MATLAB code available in this tab so that the parameter range validation occurs only when you apply the changes. This option does not provide visualization of the waveform before running the Simulink model. Modifying the waveform parameters using this option is not recommended if you are not familiar with the MATLAB code that generates the selected waveform.

If you choose to modify the configuration by using the **Initialization** tab, you can open it by clicking the exported block, pressing **Ctrl+M** to open the Mask Editor dialog, and clicking the **Initialization** tab.

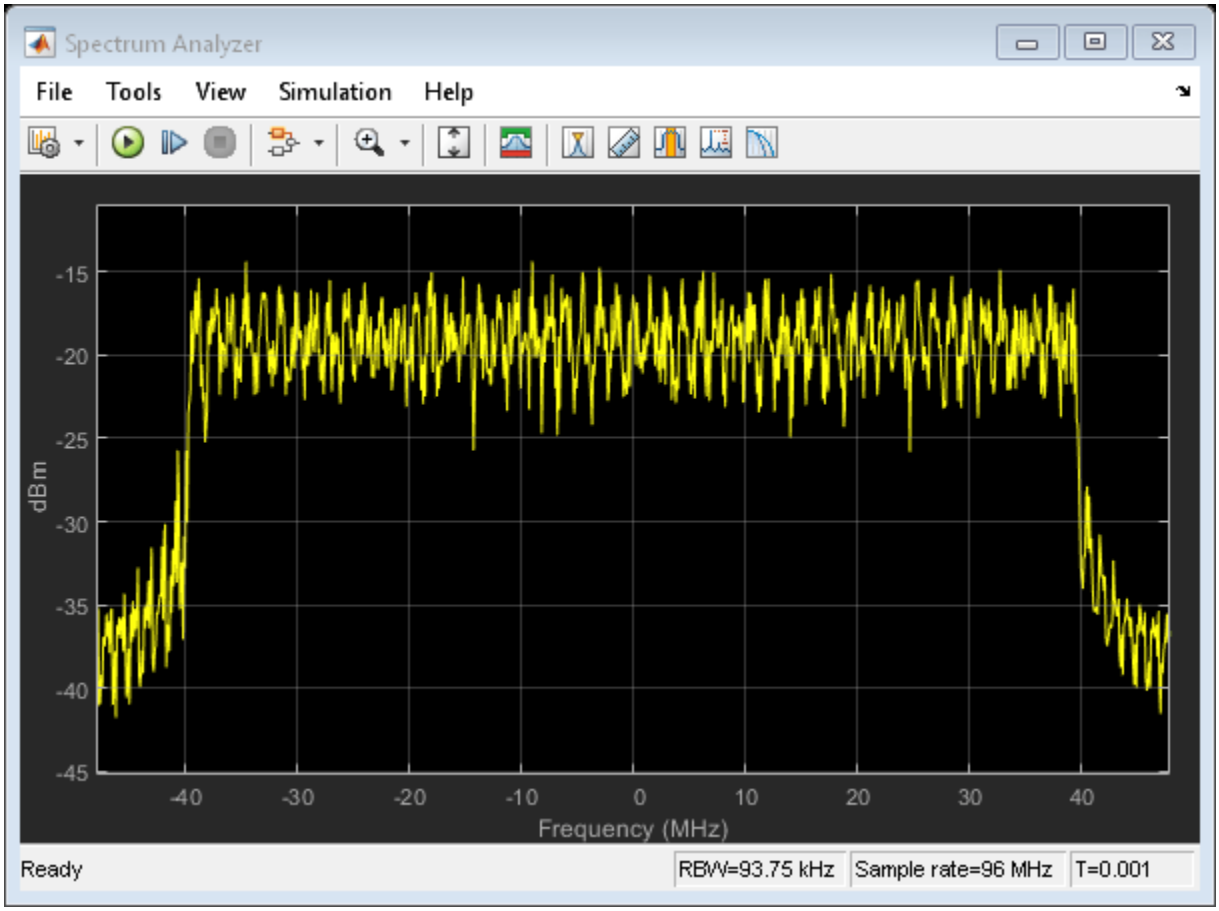


Use the MATLAB code that is available in the **Initialization** tab to update the parameters of your choice. For example, set the subcarrier spacing, `scs`, to 1,500,000 Hz.



Click **OK** to apply the changes and close the Mask Editor dialog box. Simulate the model to visualize the updated waveform.

```
modelName = 'WWGExport2SimulinkModelSCSModified';
sim(modelName);
```



The Spectrum Analyzer block now shows a sample rate of 96 MHz, which is 1.5 times the previous sample rate, as expected.

Share Wireless Waveform Configuration with Other Blocks in the Model

To access read-only block parameters and waveform configuration parameters, use the `UserData` common block property, which is a structure with these fields.

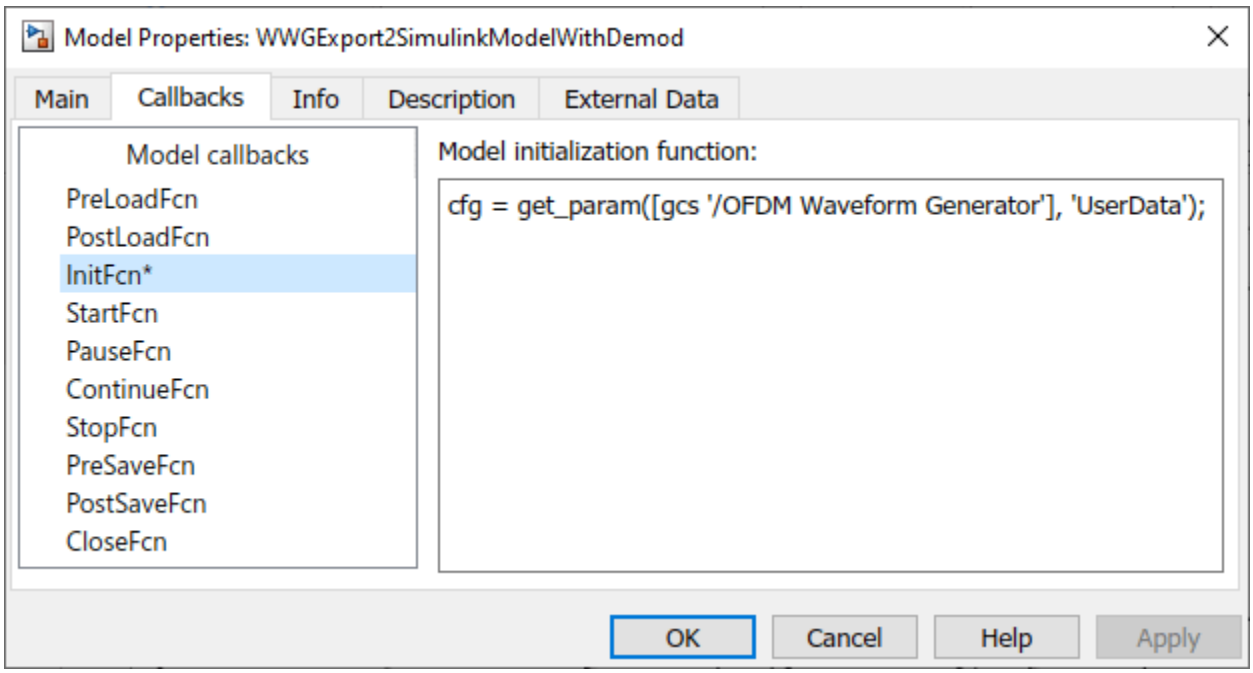
- `WaveformConfig`: Waveform configuration
- `WaveformLength`: Waveform length
- `Fs`: Waveform sample rate

You can access the user data of the exported block by using the `get_param` function.

```
get_param([gcs '/OFDM Waveform Generator'],'UserData')
```

```
ans =

  struct with fields:

    WaveformConfig: [1x1 comm.OFDMModulator]
    WaveformLength: 8000
                Fs: 96000000
```

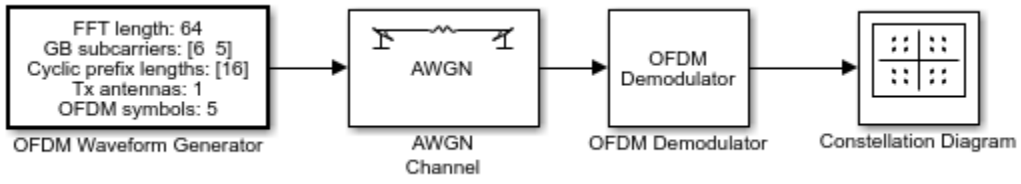
Store the structure available in the user data in a base workspace variable by using the `InitFcn` in the callback. The `InitFcn` callback is executed during a model update and simulation. To use this callback, click the **MODELING** tab, then click the **Model Settings** dropdown, and click the **Model Properties** option. In the **Callbacks** pane, select the `InitFcn` callback. Assign the user data to a new base workspace variable (for example, `cfg`).



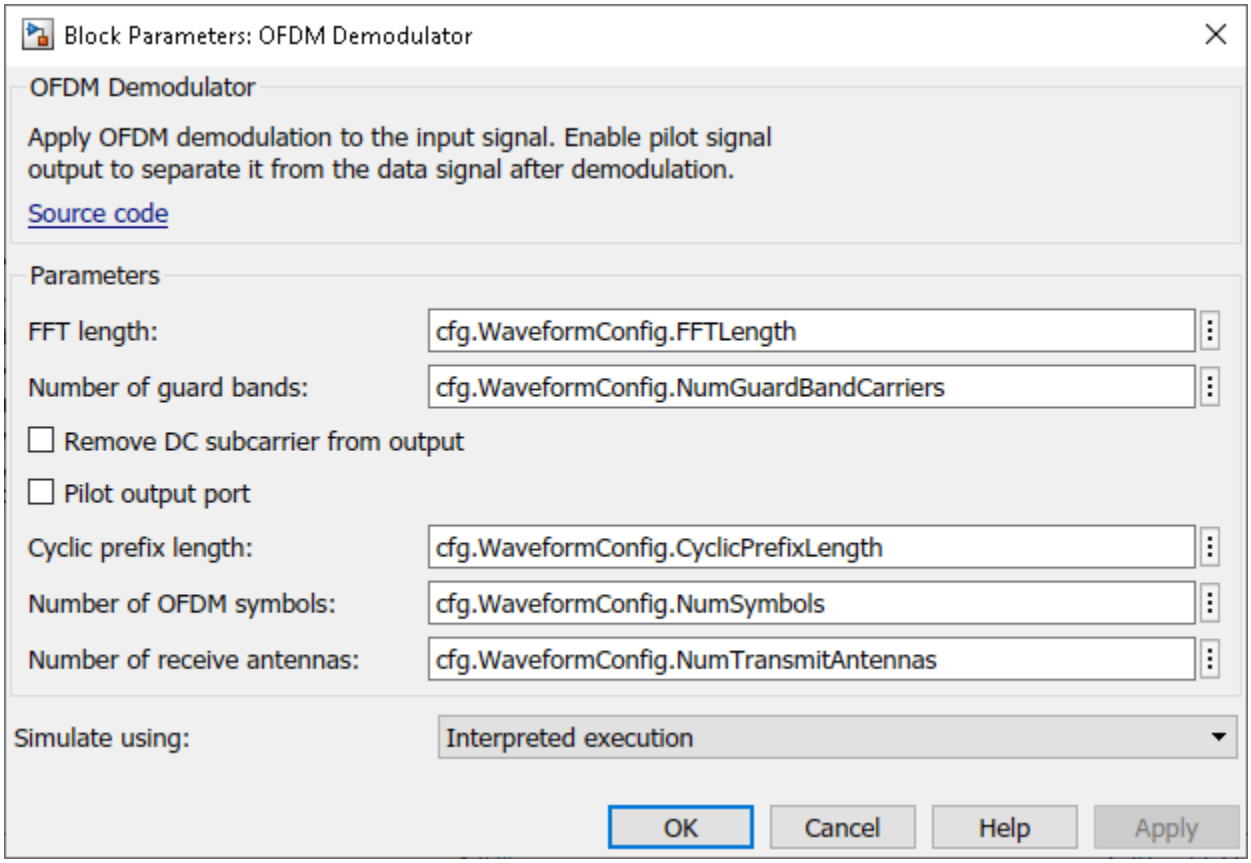
The parameters that are available in the user data of the exported block are updated every time you apply configuration changes in the **Initialization** tab.

To demodulate the OFDM waveform, add an OFDM Demodulator block to the model. Connect an AWGN Channel block between the OFDM Waveform Generator and OFDM Demodulator blocks to add white Gaussian noise to the input signal. Also add a Constellation Diagram block to plot the demodulated symbols.

```
modelName = 'WWGExport2SimulinkModelWithDemod';
open_system(modelName);
```

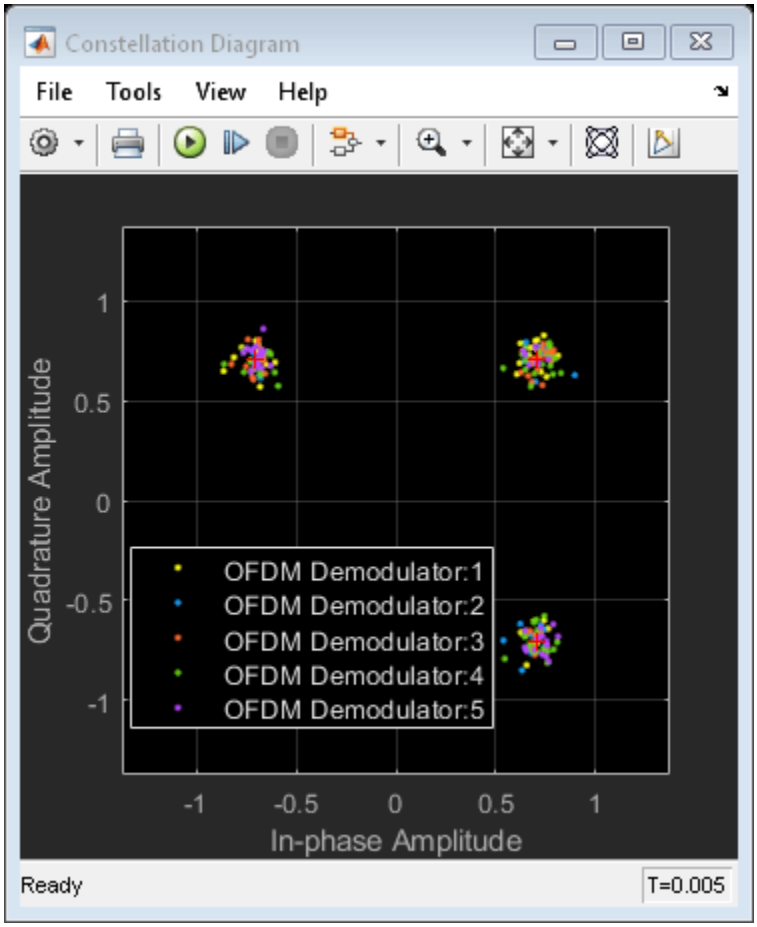


The parameters that are required to configure the OFDM Demodulator block must match the parameters that are used to configure the exported block, (otherwise, demodulation fails). To access the configuration parameters of the exported block, use the variable `cfg`. This figure shows the parameters of the OFDM Demodulator block.



Because the OFDM Demodulator block requires the entire OFDM waveform for demodulation, set the **Samples per frame** parameter in the exported block to `cfg.WaveformLength`. Simulate the model.

```
sim(modelName);
```

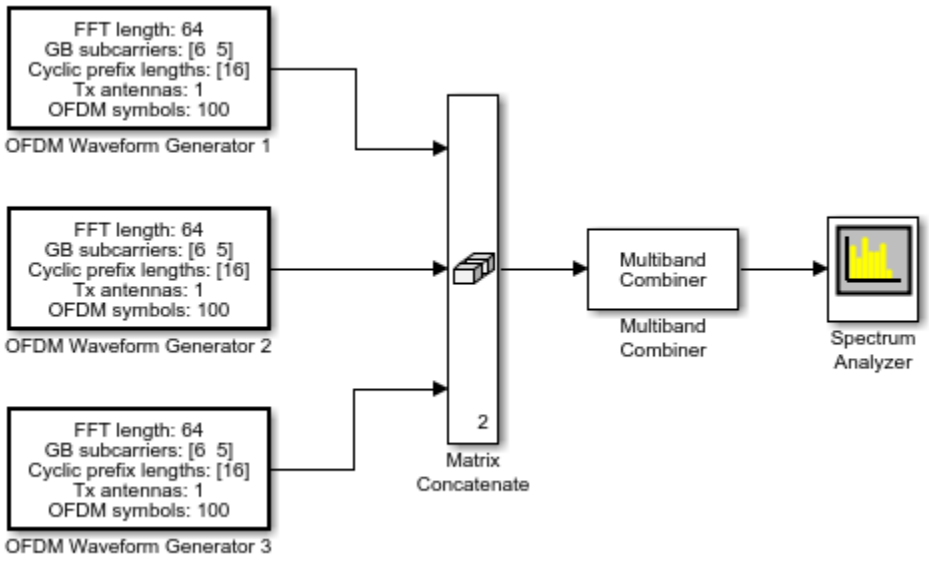


After demodulating the OFDM waveform by using the OFDM Demodulator block, the Constellation Diagram block displays the resulting QAM symbols.

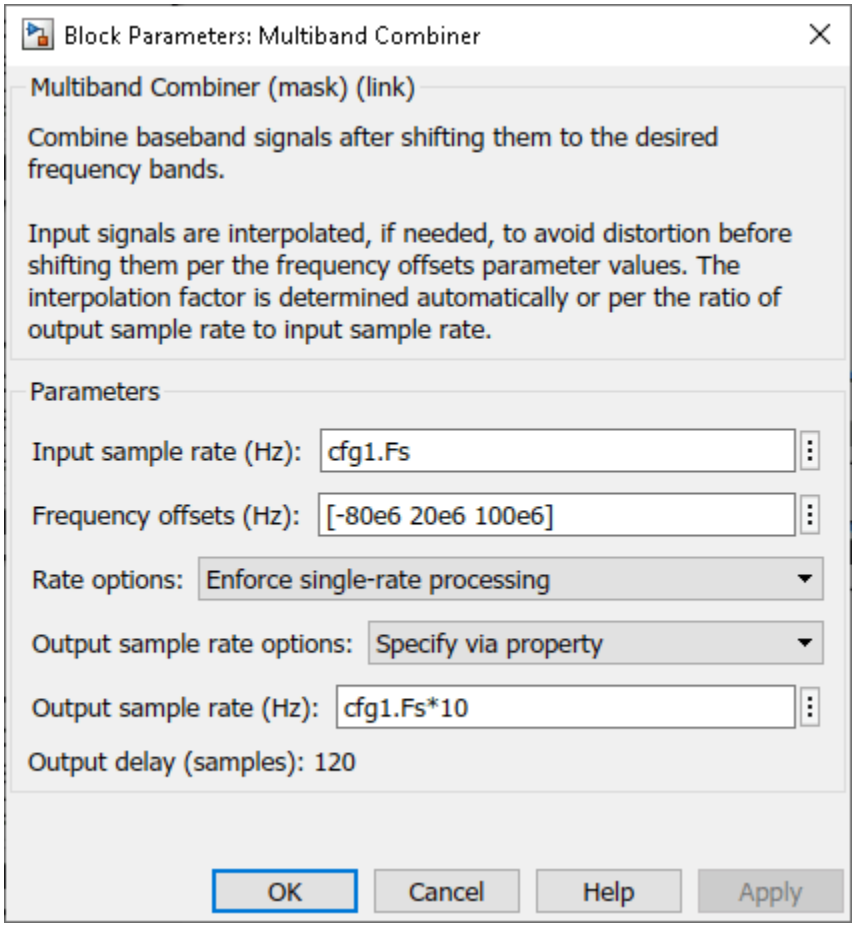
Generate Multicarrier Waveforms

For multicarrier generation, the sampling rates for all of the waveforms must be the same. To shift the waveforms to a carrier offset and aggregate them, you can use the Multiband Combiner block.

```
modelName = 'WWGExport2SimulinkMulticarrier';
open_system(modelName);
```

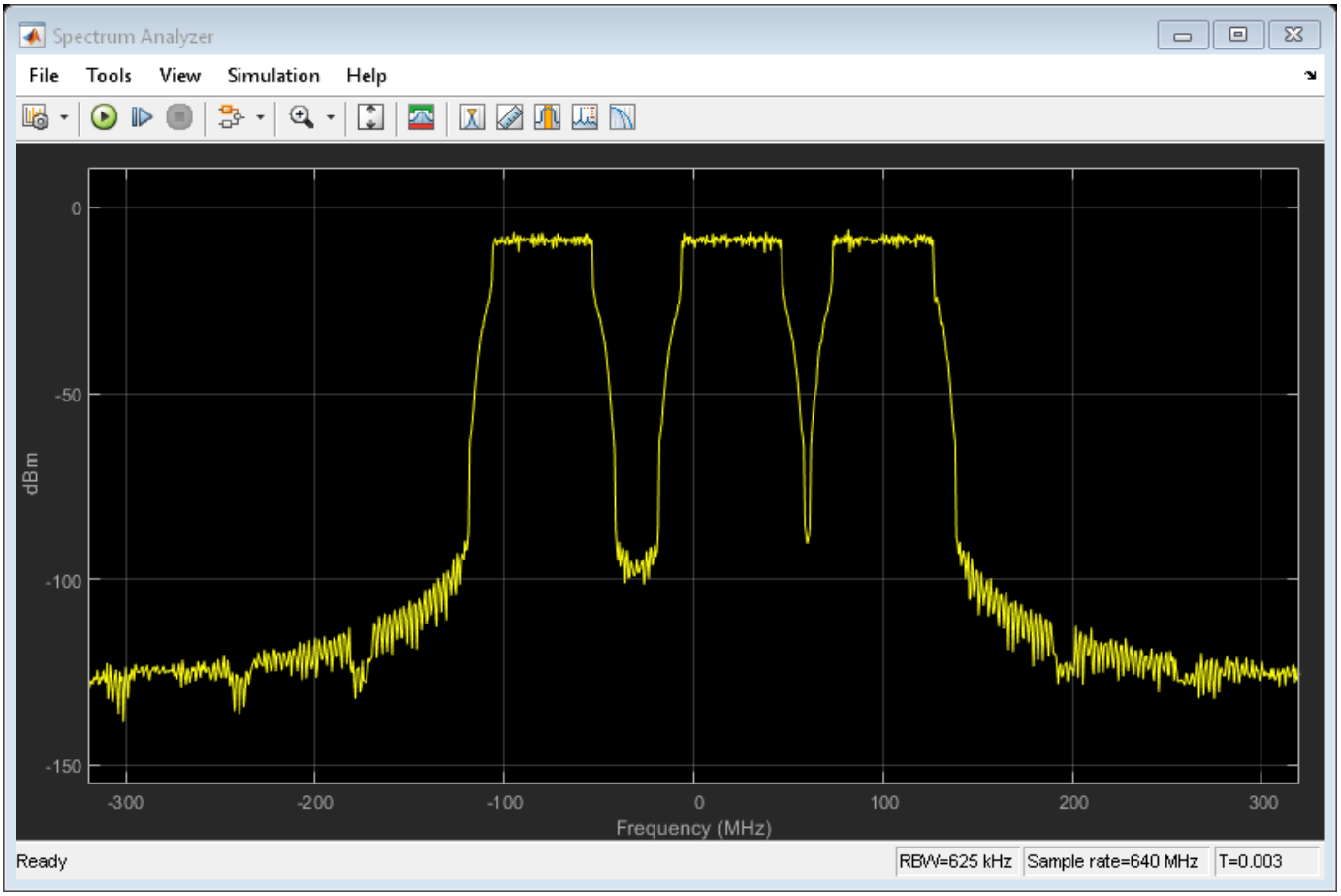


To shift the waveforms in frequency, you might have to increase the sampling rates. The Multiband Combiner block provides the option to oversample the input waveforms before shifting and combining them. This figure shows the parameters of the Multiband Combiner block.



Simulate the model to visualize the waveforms that are centered at -80, 20, and 100 MHz.

```
sim(modelName);
```



See Also

Apps
Wireless Waveform Generator

Blocks
Waveform From Wireless Waveform Generator App

More About

- “Use Wireless Waveform Generator App” on page 29-2

RF Propagation

Troubleshooting Site Viewer

In this section...

Internet Connection Failure

When you create a Site Viewer, a check is made to make sure that you have an internet connection to retrieve the default basemap and terrain data.

If Site Viewer cannot connect to the Internet the following warning messages are displayed:

- *Warning: Unable to access the Internet, showing Dark Water instead of Satellites. See Access Basemaps and Terrain in Site Viewer.*
- *Warning: Unable to access terrain data. See Access Basemaps and Terrain in Site Viewer.*

If Site Viewer cannot connect to the Internet, then terrain data is not used and the `Dark Water` basemap is selected.

Graphics Environment

Site Viewer can fail to open because of two reasons:

- In MATLAB, OpenGL® is set to software graphics. An error message is displayed in the command window, notifying you to upgrade the graphics hardware driver or select hardware graphics using OpenGL.

  For more information, see `opengl`, and “Resolving Low-Level Graphics Issues”.
- When starting Site Viewer, JavaScript® for WebGL™ support fails. An error message is displayed in the command window, notifying you to update the graphics hardware driver.

  For more information, see “Resolving Low-Level Graphics Issues”

See Also

Functions
`opengl`

Objects
`siteviewer`

More About

- “Access Basemaps and Terrain in Site Viewer” on page 30-3
- “System Requirements for Graphics”
- “Resolving Low-Level Graphics Issues”

Access Basemaps and Terrain in Site Viewer

In this section...

Access and Download Basemaps

To download MATLAB basemaps:

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Add-Ons**.
2. In the Add-On Explorer, scroll to the **MathWorks Features** section, and click **show all** to find the basemap packages. You can also search for the basemap add-ons by name (listed in the following table) or click **Features** in **Filter by Type**.
3. Select the basemap data packages. For more information about basemaps, see `geobubble`.

Basemap Name	Basemap Data Package Name
`'bluegreen'`	MATLAB Basemap Data - bluegreen
`'grayland'`	MATLAB Basemap Data - grayland
`'colorterrain'`	MATLAB Basemap Data - colorterrain
`'grayterrain'`	MATLAB Basemap Data - grayterrain
`'landcover'`	MATLAB Basemap Data - landcover

In addition, Site Viewer also supports external basemaps that you can select from the basemap picker. The following options are available:

- `Satellite`
- `Topographic`
- `Streets`
- `Streets-Light`
- `Streets-Dark`
- `OpenStreetMap`

You need an active internet connection for all Site viewer basemaps except `Dark Water`. Use `addCustomBasemap` to access basemaps from user-specified URLs.

Note If basemap does not render correctly in Site Viewer (for example only the ocean is visible), check if the basemap server supports CORS (cross-origin resource sharing). Site Viewer does not support basemaps that do not support CORS.

Access Terrain

To access terrain data for Site Viewer, you need an active internet connection. Use `addCustomTerrain` to access terrain data from DTED-files without an active internet connection.

See Also

Objects
`siteviewer`

More About

- “Troubleshooting Site Viewer” on page 30-2
- “System Requirements for Graphics”
- “Resolving Low-Level Graphics Issues”

Access TIREM Software

The Terrain Integrated Rough Earth Model™ (TIREM™) is a propagation model for computing the path loss for irregular terrain and seawater scenarios. TIREM is developed, trademarked, and licensed by Alion Science. To use TIREM, you need to acquire it from `Alion Science`.

TIREM is designed to calculate the reference basic median propagation loss (path loss) based on the terrain profile along the great circle path between two antennas, for example, using digital terrain elevation data (DTED). You can use TIREM model to calculate the point-to-point path loss between sites over irregular terrain. The model combines physics with empirical data to provide path loss estimates. The TIREM propagation model can predict path loss at frequencies between 1 MHz and 1 THz.

Use `tiremSetup` to enable TIREM access from within MATLAB. The TIREM library folder contains the `tirem3` shared library. The full library name is platform-dependent:

Platform	Shared Library Name
`Windows`	`libtirem3.dll` or `tirem3.dll`
`Linux`	`libtirem3.so`
`Mac`	`libtirem3.dylib`

See Also

Functions
`tiremSetup`

Objects
`TIREM`

Related Examples

- “Planning Radar Network Coverage over Terrain” (Antenna Toolbox)

Choose a Propagation Model

Introduction

Propagation models allow you to predict the propagation and attenuation of radio signals as the signals travel through the environment. You can simulate different models by using the `propagationModel` function. Additionally, you can determine the range and path loss of radio signals in these simulated models by using the `range` and `pathloss` functions.

The following sections describe various propagation and ray tracing models. The tables in each section list the models that are supported by the `propagationModel` function and compare, for each model, the supported frequency ranges, model combinations, and limitations.

Atmospheric

Atmospheric propagation models predict path loss between sites as a function of distance. These models assume line-of-sight (LOS) conditions and disregard the curvature of the Earth, terrain, and other obstacles.

Model	Description	Frequency	Combinations	Limitations
freespace (`FreeSpace`)	Ideal propagation model with clear line of sight between transmitter and receiver	No enforced range	Can be combined with rain, fog, and gas	Assumes line of sight
rain (`Rain`)	Propagation of a radio wave signal and its path loss in rain. For more information, see [3].	1 GHz to 1000 GHz	Can be combined with any other propagation model	Assumes line of sight
gas (`Gas`)	Propagation of radio wave signal and its path loss due to oxygen and water vapor. For more information, see [5].	1GHz to 1000 GHz	Can be combined with any other propagation model	Assumes line of sight
fog (`Fog`)	Propagation of the radio wave signal and its path loss in cloud and fog. For more information, see [2].	10GHz to 1000 GHz	Can be combined with any other propagation model	Assumes line of sight

Empirical

Like atmospheric propagation models, empirical models predict path loss as a function of distance. Unlike atmospheric models, the close-in empirical model supports non-line-of-sight (NLOS) conditions.

Model	Description	Frequency	Combinations	Limitations
close-in (`CloseIn`)	Propagation of signals in urban macro cell scenarios. For more information, see [1].	No enforced range	Can be combined with rain, fog, and gas	—

Terrain

Terrain propagation models assume that propagation occurs between two points over a slice of terrain. Use these models to calculate the point-to-point path loss between sites over irregular terrain, including buildings.

Terrain models calculate path loss from free-space loss, terrain and obstacle diffraction, ground reflection, atmospheric refraction, and tropospheric scatter. They provide path loss estimates by combining physics with empirical data.

Model	Description	Frequency	Combinations	Limitations
longley-rice (`LongleyRice`)	Also known as Irregular Terrain Model (ITM). For more information, see [4].	20 MHz to 20 GHz	Can be combined with rain, fog, and gas	Antenna height minimum is 0.5 m and maximum is 3000 m
tirem (`TIREM`)	Terrain Integrated Rough Earth Model	1 MHz to 1000 GHz	Can be combined with rain, fog, and gas	• Requires access to external TIREM library • Antenna height maximum is 30000 m

Ray Tracing

Ray tracing models, represented by `RayTracing` objects, compute propagation paths using 3-D environment geometry [7][8]. They determine the path loss and phase shift of each ray using electromagnetic analysis, including tracing the horizontal and vertical polarizations of a signal through the propagation path. The path loss includes free-space loss and reflection losses. For each reflection, the model calculates losses on the horizontal and vertical polarizations by using the Fresnel equation, the incident angle, and the relative permittivity and conductivity of the surface material [5][6] at the specified frequency.

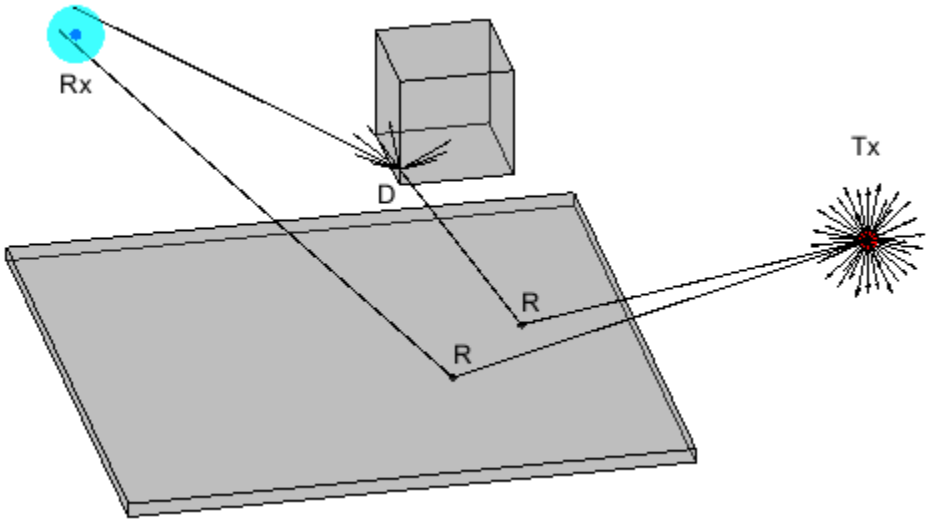
While the other supported models compute single propagation paths, ray tracing models compute multiple propagation paths.

These models support both 3-D outdoor and indoor environments.

Ray Tracing Method	Description	Frequency	Combinations	Limitations
shooting and bouncing rays (SBR)	• Supports calculation of approximate propagation paths for up to ten path reflections. The locations of receiver sites calculated by the SBR method are not exact. The accuracy of the calculated propagation paths decreases as the length of the paths increases. • Computational complexity increases linearly with the number of reflections. As a result, the SBR method is generally faster than the image method.	100 MHz to 100 GHz	Can be combined with rain, fog, and gas	Does not include effects from diffraction, refraction, and scattering
image	• Supports up to two path reflections and calculates exact propagation paths. • Computational complexity increases exponentially with the number of reflections.	100 MHz to 100 GHz	Can be combined with rain, fog, and gas	Does not include effects from diffraction, refraction, and scattering

SBR Method

This figure illustrates the SBR method for calculating propagation paths from a transmitter, *Tx*, to a receiver, *Rx*.



The SBR method launches many rays from a geodesic sphere centered at *Tx*. The geodesic sphere enables the model to launch rays that are approximately uniformly spaced.

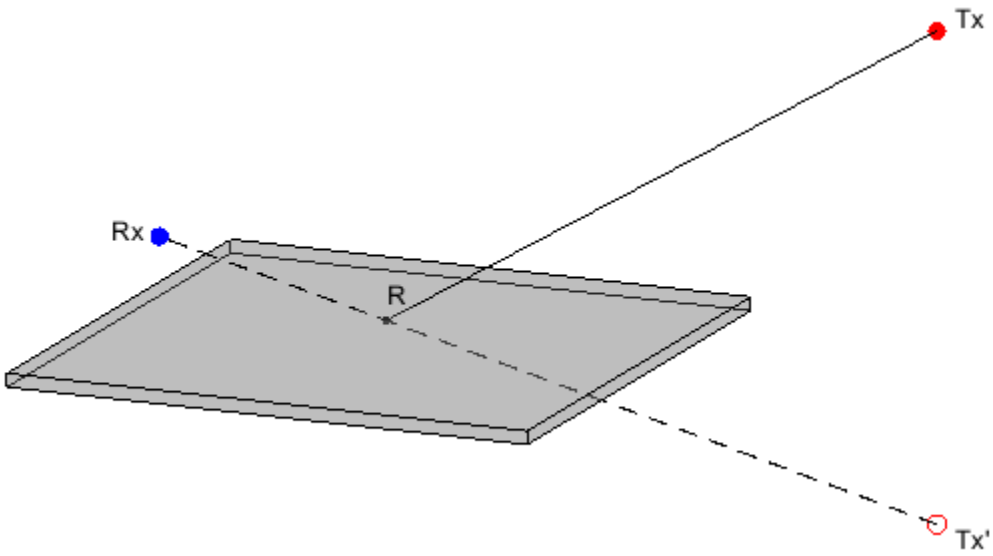
Then, the method traces every ray from *Tx* and can model different types of interactions between the rays and surrounding objects, such as reflections, diffractions, refractions, and scattering. Note that the implementation considers only reflections.

- When a ray hits a flat surface, shown as *R*, the ray reflects based on the law of reflection.
- When a ray hits an edge, shown as *D*, the ray spawns many diffracted rays based on the law of diffraction [9][10]. Each diffracted ray has the same angle with the diffracting edge as the incident ray. The diffraction point then becomes a new launching point and the SBR method traces the diffracted rays in the same way as the rays launched from *Tx*. A continuum of diffracted rays form a cone around the diffracting edge, which is commonly known as a Keller cone [10]. The current implementation of the SBR method does not consider edge diffractions.

For each launched ray, the method surrounds *Rx* with a sphere, called a reception sphere, with a radius that is proportional to the angular separation of the launched rays and the distance the ray travels. If the ray intersects the sphere, then the model considers the ray a valid path from *Tx* to *Rx*.

Image Method

This figure illustrates the image method for calculating the propagation path of a single reflection ray for the same transmitter and receiver as the SBR method. The image method locates the image of *Tx* with respect to a planar reflection surface, *Tx*'. Then, the method connects *Tx*' and *Rx* with a line segment. If the line segment intersects the planar reflection surface, shown as *R* in the figure, then a valid path from *Tx* to *Rx* exists. The method determines paths with multiple reflections by recursively extending these steps.



References

[1] Sun, Shu, Theodore S. Rappaport, Timothy A. Thomas, Amitava Ghosh, Huan C. Nguyen, Istvan Z. Kovacs, Ignacio Rodriguez, Ozge Koymen, and Andrzej Partyka. "Investigation of Prediction Accuracy, Sensitivity, and Parameter Stability of Large-Scale Propagation Path Loss Models for 5G Wireless Communications." *IEEE Transactions on Vehicular Technology* 65, no. 5 (May 2016): 2843–60. https://doi.org/10.1109/TVT.2016.2543139.

[2] International Telecommunications Union Radiocommunication Sector. *Attenuation due to clouds and fog*. Recommendation P.840-6. ITU-R, approved September 30, 2013. https://www.itu.int/rec/R-REC-P.840-6-201309-S/en.

[3] International Telecommunications Union Radiocommunication Sector. *Specific attenuation model for rain for use in prediction methods*. Recommendation P.838-3. ITU-R, approved March 8, 2005. https://www.itu.int/rec/R-REC-P.838-3-200503-I/en.

[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. National Telecommunications and Information Administration, April 1, 1982.

[5] International Telecommunications Union Radiocommunication Sector. *Effects of building materials and structures on radiowave propagation above about 100MHz.* Recommendation P.2040-1. ITU-R, approved July 29, 2015. https://www.itu.int/rec/R-REC-P.2040-1-201507-I/en.

[6] International Telecommunications Union Radiocommunication Sector. *Electrical characteristics of the surface of the Earth*. Recommendation P.527-5. ITU-R, approved August 14, 2019. https://www.itu.int/rec/R-REC-P.527-5-201908-I/en.

[7] Yun, Zhengqing, and Magdy F. Iskander. "Ray Tracing for Radio Propagation Modeling: Principles and Applications." *IEEE Access* 3 (2015): 1089–1100. https://doi.org/10.1109/ACCESS.2015.2453991.

[8] Schaubach, K.R., N.J. Davis, and T.S. Rappaport. "A Ray Tracing Method for Predicting Path Loss and Delay Spread in Microcellular Environments." In *[1992 Proceedings] Vehicular*

Technology Society 42nd VTS Conference - Frontiers of Technology, 932–35. Denver, CO, USA: IEEE, 1992. https://doi.org/10.1109/VETEC.1992.245274.

[9] International Telecommunications Union Radiocommunication Sector. *Propagation by diffraction*. Recommendation P.526-15. ITU-R, approved October 21, 2019. https://www.itu.int/rec/R-REC-P.526-15-201910-I/en.

[10] Keller, Joseph B. "Geometrical Theory of Diffraction." *Journal of the Optical Society of America* 52, no. 2 (February 1, 1962): 116. https://doi.org/10.1364/JOSA.52.000116.

See Also

Functions

`propagationModel` | `raytrace`

Related Examples

- "Ray Tracing for Wireless Communications" on page 30-12
- "Visualize Antenna Coverage Map and Communication Links" on page 2-12
- "Urban Link and Coverage Analysis Using Ray Tracing" on page 2-21

Ray Tracing for Wireless Communications

Introduction

Wireless communication systems use radio waves to transmit signals. Propagation modeling enables you to estimate the strength of signals based on system parameters such as frequency, antenna height, terrain properties, and building properties.

Theoretical and empirical models estimate path loss based on range, and are valid for only those environments that resemble the modeling environment. As a result, they usually do not provide accurate temporal or spatial information. Unlike these models, ray tracing models are specific to the 3-D environment, and are therefore appropriate for scenarios such as urban environments.

For propagation modeling, a ray is an individual radio signal that [1]:

- Travels in a straight line through a homogeneous medium.
- Obeys the laws of reflection, refraction, and diffraction.
- Carries energy. Propagation models treat rays like tubes, where the energy density on the cross section becomes smaller as the ray interacts with the environment.

For a given 3-D environment, ray tracing models use numeric simulations to:

- Predict the paths of rays from transmitters to receivers. The models can find many rays from a transmitter to a receiver. The models derive the angle of departure, angle of arrival, and time of arrival from the paths.
- Estimate the path loss and phase change for each ray. Total path loss is the sum of interaction losses, free space loss, and, optionally, atmospheric loss.

A ray interacts with the environment in several ways [1].

Interaction	Description
Line-of-sight (LOS)	The ray travels directly from the transmitter to the receiver.
Reflection	The ray reflects off a surface according to the law of reflection.
Refraction (transmission)	The ray refracts as it moves into a new medium, according to the law of refraction.
Diffraction	The ray diffracts off a surface according to the law of diffraction. One ray can spawn many diffracted rays.
Scattering	The ray interacts with a rough surface such as the ocean or a building facade.

Use these functions to create ray tracing models, predict propagation paths, and calculate path losses and phase shifts.

- `propagationModel` — Create a ray tracing model as a `RayTracing` object. Specify options such as the ray tracing method, the maximum number of reflections, and surface materials. You can use ray tracing models as input when conducing RF analysis, such as when generating coverage maps

by using the `coverage` function or when calculating total received power by using the `sigstrength` function.

- `raytrace` — Display propagation paths (rays) on a map or return propagation paths as `comm.Ray` objects. Each object represents the full path from the transmitter to the receiver, and contains information such as the path loss, phase shift, and types of surface interactions.
- `raypl` — Calculate the path loss and phase shift for a propagation path based on surface materials and antenna polarization types.

For examples that show ray tracing in indoor and urban environments, see “Indoor MIMO-OFDM Communication Link Using Ray Tracing” on page 8-9 and “Urban Link and Coverage Analysis Using Ray Tracing” on page 2-21, respectively.

Ray Tracing Methods

The ray tracing model used by the `propagationModel` and `raytrace` functions finds LOS and non-line-of-sight (NLOS) paths.

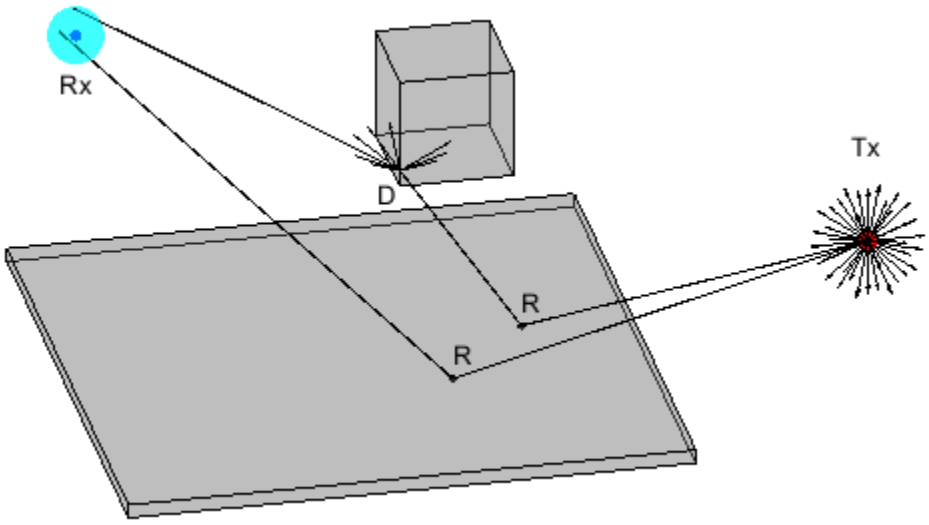
- The model finds LOS paths by shooting a ray from the transmitter toward the receiver. If the ray does not interact with a surface before reaching the receiver, then an LOS path exists.
- The model finds NLOS paths by using either the shooting and bouncing rays (SBR) method [2] or the image method. You can specify the method by using the `propagationModel` function.

Choose a method based on the types of interactions you want to model, the computation speed, and the accuracy.

Method	Interaction Types	Computation Speed	Computation Accuracy
SBR	Includes effects from reflection and does not include effects from diffraction, refraction, or scattering. Supports calculation of approximate propagation paths for up to ten path reflections.	Computational complexity increases linearly with the number of reflections. As a result, the SBR method is generally faster than the image method.	The locations of receiver sites calculated by the SBR method are not exact. The accuracy of the calculated propagation paths decreases as the length of the paths increases.
Image	Includes effects from reflection and does not include effects from diffraction, refraction, or scattering. Supports up to two path reflections.	Computational complexity increases exponentially with the number of reflections.	Calculates exact propagation paths.

SBR Method

This figure illustrates the SBR method for calculating propagation paths from a transmitter, *Tx*, to a receiver, *Rx*.



The SBR method launches many rays from a geodesic sphere centered at *Tx*. The geodesic sphere enables the model to launch rays that are approximately uniformly spaced.

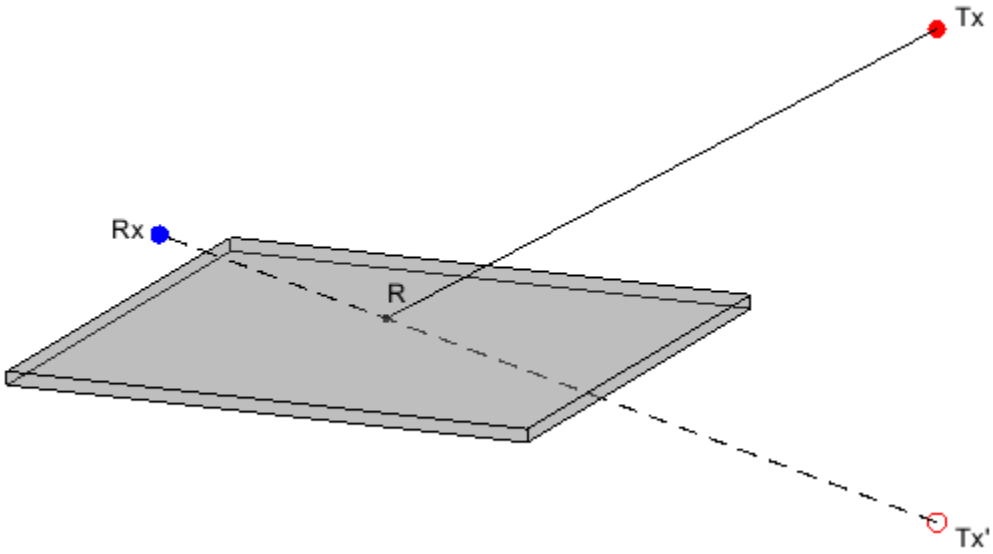
Then, the method traces every ray from *Tx* and can model different types of interactions between the rays and surrounding objects, such as reflections, diffractions, refractions, and scattering. Note that the implementation considers only reflections.

- When a ray hits a flat surface, shown as *R*, the ray reflects based on the law of reflection.
- When a ray hits an edge, shown as *D*, the ray spawns many diffracted rays based on the law of diffraction [3][4]. Each diffracted ray has the same angle with the diffracting edge as the incident ray. The diffraction point then becomes a new launching point and the SBR method traces the diffracted rays in the same way as the rays launched from *Tx*. A continuum of diffracted rays form a cone around the diffracting edge, which is commonly known as a Keller cone [4]. The current implementation of the SBR method does not consider diffraction.

For each launched ray, the method surrounds *Rx* with a sphere, called a reception sphere, with a radius that is proportional to the angular separation of the launched rays and the distance the ray travels. If the ray intersects the sphere, then the model considers the ray a valid path from *Tx* to *Rx*.

Image Method

This figure illustrates the image method for calculating the propagation path of a single reflection ray for the same transmitter and receiver as the SBR method. The image method locates the image of *Tx* with respect to a planar reflection surface, *Tx*'. Then, the method connects *Tx*' and *Rx* with a line segment. If the line segment intersects the planar reflection surface, shown as *R* in the figure, then a valid path from *Tx* to *Rx* exists. The method determines paths with multiple reflections by recursively extending these steps.



Propagation Loss

The ray tracing model used by the `propagationModel`, `raytrace`, and `raypl` functions calculates reflection losses by tracking the horizontal and vertical polarizations of signals through the propagation path. Total power loss is the sum of free space loss, and reflection loss.

Effect of Surface Materials

When a ray interacts with a surface, the surface material impacts the reflection losses.

The ray tracing model incorporates building and surface materials into the propagation loss calculations by using the complex relative permittivity of the surface, ε_r. The ITU-R P.2040-1 [5] and ITU-R P.527 [6] recommendations include methods, equations, and values used to calculate ε_r for a range of frequencies.

The equations for ε_r are:

$$\varepsilon_r = \varepsilon_r' + j\varepsilon_r''$$

$$\varepsilon_r'' = \frac{\sigma}{2\pi\varepsilon_0 f},$$

where:

- ε_r' is the real relative permittivity.
- σ is the conductivity in S/m.
- ε_0 is the permittivity of free space (electric constant).
- f is the frequency in Hz.

For building materials, the ray tracing model calculates ε_r' and σ as:

$$\varepsilon_r' = af^b$$

$\sigma = cf^d$,

where a, b, c, and d are constants determined by the surface material. For readability, the table shows the frequency range in GHz.

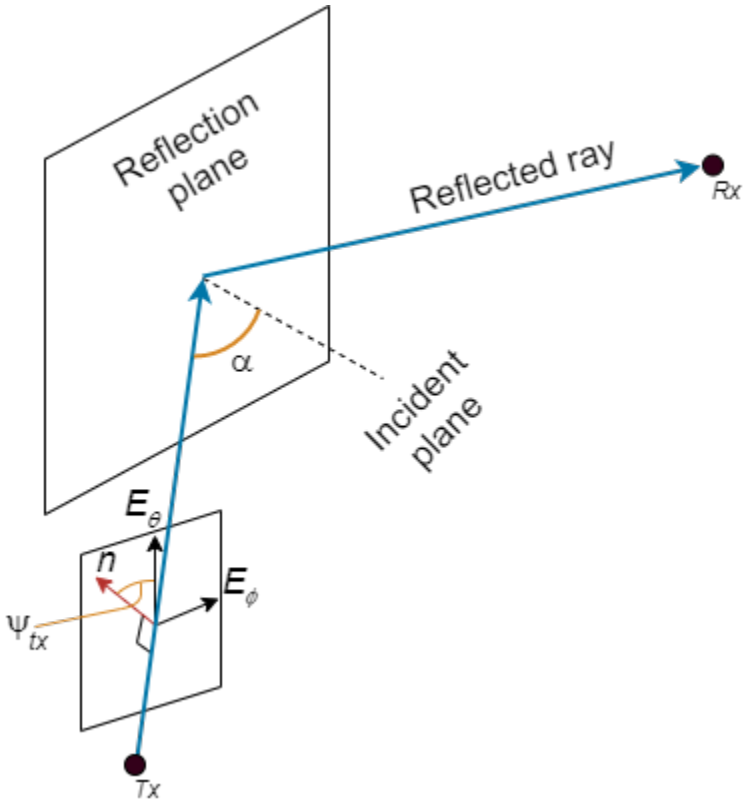
Material Class	Real Part of Relative Permittivity		Conductivity (S/m)		Frequency Range (GHz)
	a	***b***	***c***	***d***	
Vacuum (~ air)	1	0	0	0	[0.001, 100]
Concrete	5.31	0	0.0326	0.8095	[1, 100]
Brick	3.75	0	0.038	0	[1, 10]
Plasterboard	2.94	0	0.0116	0.7076	[1, 100]
Wood	1.99	0	0.0047	1.0718	[0.001, 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]
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.					

For earth surfaces such as water, sea water, dry or wet ice, dry or wet soil, and vegetation, the ray tracing model calculates ε_r using the methods and equations presented in ITU-R P.527 [6].

Reflection Loss

The ray tracing model computes reflection loss by using the reflection matrix computations described in IEEE document 802.11-09/0334r8 [7].

This image shows a reflection path from a transmitter site *Tx* to a receiver site *Rx*.



For a first order signal reflection, the equation for reflection path loss, PL_R, is:

$$PL_R = JV_{rx}' H_{ref1} JV_{tx},$$

where:

- JV_{rx} and JV_{tx} are 2-by-1 polarization vectors for the receiver and transmitter, specified as Jones vectors.
- H_{ref1} is a reflection matrix.

The equation for the reflection matrix H_{ref1} is:

$$H_{ref1} = \begin{pmatrix} \cos(\Psi_{rx}) & \sin(\Psi_{rx}) \\ -\sin(\Psi_{rx}) & \cos(\Psi_{rx}) \end{pmatrix} \times \begin{pmatrix} R_\perp(\alpha) & 0 \\ 0 & R_\parallel(\alpha) \end{pmatrix} \times \begin{pmatrix} \cos(\Psi_{tx}) & \sin(\Psi_{tx}) \\ -\sin(\Psi_{tx}) & \cos(\Psi_{tx}) \end{pmatrix},$$

where:

- The third and first terms are geometric coupling matrices. The third term recalculates the polarization vector from the basis of the transmitter coordinates to the basis of the incident plane. The first term recalculates the polarization vector from the basis of the incident plane to the basis of the receiver coordinates. Ψ_{rx} and Ψ_{tx} are the angles between the vertical electromagnetic field vector E_θ and a normal to the incident plane, n, at the receiver and transmitter, respectively.
- The second term is a polarization matrix, where $R_\parallel$ and $R_\square$ are the reflection coefficients for the horizontal and vertical polarizations, respectively.

The model accounts for the geometric coupling between horizontal and vertical polarizations only when both the transmitter and receiver antennas are polarized.

Ray tracing models calculate $R_{\parallel}$ and $R_{\perp}$ by using the Fresnel equation:

$$R_{\parallel}(\alpha) = \frac{\cos(\alpha) - \sqrt{\varepsilon_r - \sin^2(\alpha)}}{\cos(\alpha) + \sqrt{\varepsilon_r - \sin^2(\alpha)}}$$

$$R_{\perp}(\alpha) = \frac{\cos(\alpha) - \sqrt{(\varepsilon_r - \sin^2(\alpha))/\varepsilon_r{}^2}}{\cos(\alpha) + \sqrt{(\varepsilon_r - \sin^2(\alpha))/\varepsilon_r{}^2}},$$

where:

- α is the incident angle of the propagation vector.
- ε_r is the complex relative permittivity of the material.

The model computes higher order reflections by using an additional geometric coupling matrix and polarization matrix for each reflection.

References

[1] Yun, Zhengqing, and Magdy F. Iskander. “Ray Tracing for Radio Propagation Modeling: Principles and Applications.” *IEEE Access* 3 (2015): 1089–1100. https://doi.org/10.1109/ACCESS.2015.2453991.

[2] Schaubach, K.R., N.J. Davis, and T.S. Rappaport. “A Ray Tracing Method for Predicting Path Loss and Delay Spread in Microcellular Environments.” In *[1992 Proceedings] Vehicular Technology Society 42nd VTS Conference - Frontiers of Technology*, 932–35. Denver, CO, USA: IEEE, 1992. https://doi.org/10.1109/VETEC.1992.245274.

[3] International Telecommunications Union Radiocommunication Sector. *Propagation by diffraction*. Recommendation P.526-15. ITU-R, approved October 21, 2019. https://www.itu.int/rec/R-REC-P.526-15-201910-I/en.

[4] Keller, Joseph B. “Geometrical Theory of Diffraction.” *Journal of the Optical Society of America* 52, no. 2 (February 1, 1962): 116. https://doi.org/10.1364/JOSA.52.000116.

[5] International Telecommunications Union Radiocommunication Sector. *Effects of building materials and structures on radiowave propagation above about 100MHz.* Recommendation P.2040-1. ITU-R, approved July 29, 2015. https://www.itu.int/rec/R-REC-P.2040-1-201507-I/en.

[6] International Telecommunications Union Radiocommunication Sector. *Electrical characteristics of the surface of the Earth*. Recommendation P.527-5. ITU-R, approved August 14, 2019. https://www.itu.int/rec/R-REC-P.527-5-201908-I/en.

[7] Maltsev, A., et al. "Channel models for 60 GHz WLAN systems." IEEE Document 802.11-09/0334r8, May 2010.

[8] McNamara, D. A., C. W. I. Pistorius, and J. A. G. Malherbe. *Introduction to the Uniform Geometrical Theory of Diffraction*. Boston: Artech House, 1990.

See Also

Functions
`propagationModel` | `raytrace` | `raypl` | `buildingMaterialPermittivity` | `earthSurfacePermittivity`

Objects
`RayTracing` | `comm.Ray`

Related Examples

- “Indoor MIMO-OFDM Communication Link Using Ray Tracing” on page 8-9
- “Urban Link and Coverage Analysis Using Ray Tracing” on page 2-21
- “Three-Dimensional Indoor Positioning with 802.11az Fingerprinting and Deep Learning” (WLAN Toolbox)
- “CDL Channel Model Customization with Ray Tracing” (5G Toolbox)

Guidance for Discouraged Features

Source blocks output frames of contiguous time samples but do not use frame attribute

Source blocks output frames of contiguous time samples but do not use the frame attribute. Frame processing is still supported. Starting in R2020a:

- The Bernoulli Binary Generator and Random Integer Generator blocks now enable you to use the Upgrade Advisor. Use the Upgrade Advisor to update existing models that include the Bernoulli Binary Generator or Random Integer Generator block. You can update to the block version introduced in R2015b or keep the block version available in releases before to R2015b.
- Simulink no longer enables you to use versions of the Poisson Integer Generator, Barker Code Generator, Gold Sequence Generator, Hadamard Code Generator, Kasami Sequence Generator, OVSF Code Generator, PN Sequence Generator, or Walsh Code Generator blocks available in releases before to R2015b. Existing models automatically update to load the block version introduced in R2015b. For more information on block forwarding, see “Maintain Compatibility of Library Blocks Using Forwarding Tables” (Simulink).

Compatibility Considerations

The behavior of the random number generator for the Bernoulli Binary Generator and Poisson Integer Generator block has changed. The statistics have been improved.

For the Bernoulli Binary Generator, Poisson Integer Generator, and the Random Integer Generator blocks, the following changes were made:

- Removed **Frame-based outputs** and **Interpret vector parameters as 1-D** parameters. Blocks always output a sample-based 2-D vector.
- Introduced **Source of initial seed** parameter.

  To use the default MATLAB random number generator, leave the **Source of initial seed** parameter set to `Auto`. To set an initial seed, set **Source of initial seed** to `Parameter` and then set the **Initial seed** value.
- Behavior of the random number generator is changed. The statistics are improved.

For the Poisson Integer Generator block, the **Lambda** parameter is now **Poisson parameter (lambda)**. For the Random Integer Generator block, the **M-ary number** parameter is now **Set size**.

The **Frame-based outputs** parameter was removed for these blocks:

- Barker Code Generator
- Gold Sequence Generator
- Hadamard Code Generator
- Kasami Sequence Generator
- OVSF Code Generator
- PN Sequence Generator
- Walsh Code Generator

They always output sample-based 2-D vectors. These blocks can be upgraded using the Upgrade Advisor.

AGC object and block have simplified interfaces, better dynamic range, and faster convergence times

The AGC System object and block are improved to incorporate a simplified interface, tolerate a significantly larger input signal power range, and converge more quickly.

Compatibility Considerations

The algorithm and some properties changed in release R2015b. The properties and behavior of the previous releases can be accessed by setting the hidden `LegacyMode` property to `true`. By default, `LegacyMode` is `false`. The properties associated with the two legacy mode states are summarized.

Property	**`LegacyMode = true`**	**`LegacyMode = false`**
`AdaptationStepSize`		X
`ReferenceLevel`		X
`AveragingLength`		X
`MaximumGain`	X	X
`DetectorMethod`	X	
`LoopMethod`	X	
`UpdatePeriod`	X	
`StepSize`	X	
`GainOutputPort`	X	

Note For Simulink models in which the output gain port is enabled, the legacy mode is automatically enabled. This is required because the port is not available from the updated AGC block.