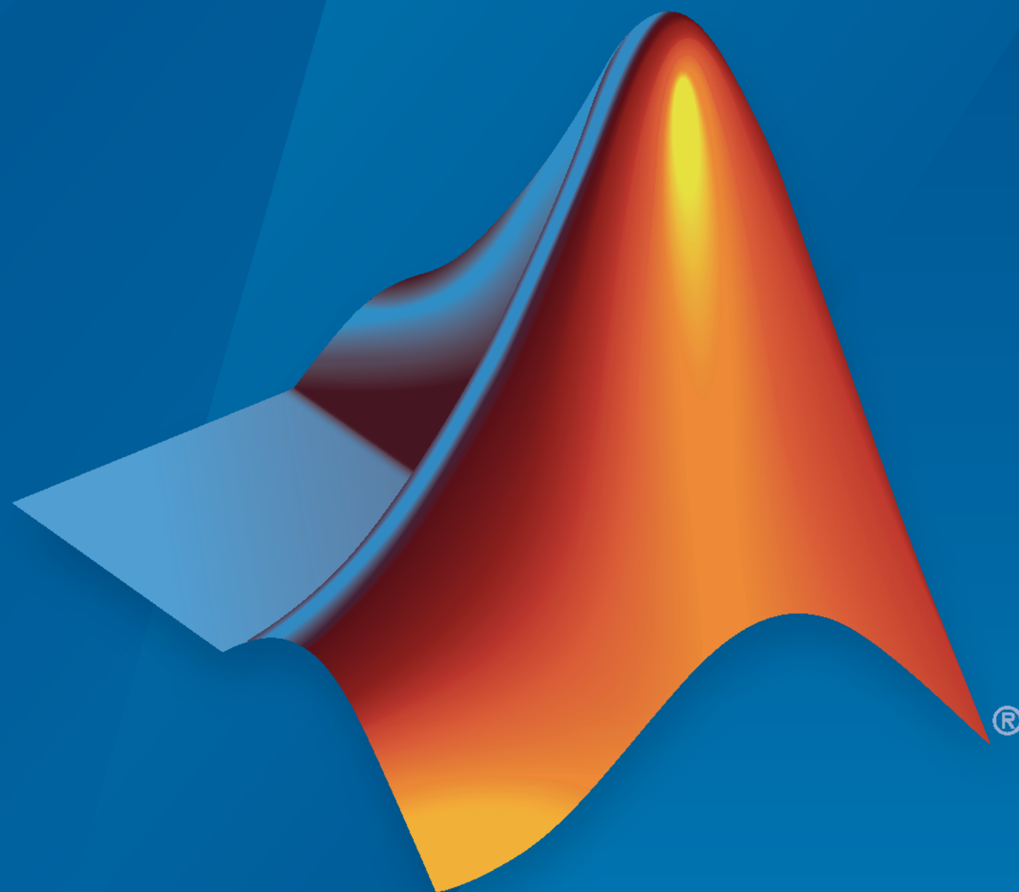
WLAN Toolbox™

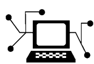
User's Guide



MATLAB®

R2023a

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

WLAN Toolbox™ User's Guide

© COPYRIGHT 2015–2023 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

October 2015	Online only	New for Version 1.0 (Release 2015b)
March 2016	Online only	Revised for Version 1.1 (Release 2016a)
September 2016	Online only	Revised for Version 1.2 (Release 2016b)
March 2017	Online only	Revised for Version 1.3 (Release 2017a)
September 2017	Online only	Revised for Version 1.4 (Release 2017b)
March 2018	Online only	Revised for Version 1.5 (Release 2018a)
September 2018	Online only	Revised for Version 2.0 (Release 2018b)
March 2019	Online only	Revised for Version 2.1 (Release 2019a)
September 2019	Online only	Revised for Version 2.2 (Release 2019b)
March 2020	Online only	Revised for Version 3.0 (Release 2020a)
September 2020	Online only	Revised for Version 3.1 (Release 2020b)
March 2021	Online only	Revised for Version 3.2 (Release 2021a)
September 2021	Online only	Revised for Version 3.3 (Release 2021b)
March 2022	Online only	Revised for Version 3.4 (Release 2022a)
September 2022	Online only	Revised for Version 3.5 (Release 2022b)
March 2023	Online only	Revised for Version 3.6 (Release 2023a)

Contents

1 PHY Modeling

PHY Simulation of Bluetooth BR/EDR, LE, and WLAN Coexistence 1-2

802.11be Waveform Generation 1-18

802.11az Waveform Generation 1-48

Build VHT PPDU 1-53

Generate VHT Multi-User Waveform 1-56

Basic VHT Data Recovery 1-60

802.11ax Waveform Generation 1-64

Basic WLAN Link Modeling 1-86

802.11ac Multi-User MIMO Precoding 1-93

2 MAC Modeling

802.11 MAC Frame Generation 2-2

802.11 MAC Frame Decoding 2-12

802.11ac Waveform Generation with MAC Frames 2-18

802.11 OFDM Beacon Frame Generation 2-25

3 Signal Transmission

802.11be Transmitter Measurements 3-2

802.11ba WUR Waveform Generation and Analysis 3-14

802.11ad Waveform Generation with Beamforming 3-21

802.11ac Transmit Beamforming 3-26

802.11ah Waveform Generation 3-36

802.11n Link in Simulink 3-45

4 Signal Reception

Recover and Analyze Packets in 802.11 Waveform 4-2

Recovery Procedure for an 802.11ax Packet 4-13

Recovery Procedure for an 802.11ac Packet 4-30

Joint Sampling Clock and Carrier Frequency Offset Tracking 4-41

Transmit and Recover L-SIG, VHT-SIG-A, VHT-SIG-B in Fading Channel 4-50

End-to-End VHT Simulation with Frequency Correction 4-53

5 Propagation Channel Models

802.11ad Packet Error Rate Single Carrier PHY Simulation with TGay Channel 5-2

802.11ac Packet Error Rate Simulation for 8x8 TGac Channel 5-10

802.11n Packet Error Rate Simulation for 2x2 TGn Channel 5-16

802.11ah Packet Error Rate Simulation for 2x2 TGah Channel 5-22

Delay Profile and Fluorescent Lighting Effects 5-28

6 End-to-End Simulation

Detect Human Presence Using WLAN Signals and Deep Learning 6-2

SNR Definition in End-to-End Simulations 6-13

802.11be Packet Error Rate Simulation for Uplink Trigger-Based Format 6-17

802.11be Downlink Multi-User MIMO and OFDMA Throughput Simulation 6-26

802.11be Packet Error Rate Simulation for an EHT MU Single-User Packet Format 6-37

802.11ax Packet Error Rate Simulation for Single-User Format 6-44

802.11ax Downlink OFDMA and Multi-User MIMO Throughput Simulation 6-51

802.11ax Packet Error Rate Simulation for Uplink Trigger-Based Format 6-62

802.11ax Compressed Beamforming Packet Error Rate Simulation 6-71

802.11ax Feedback Status Misdetection Simulation for Uplink Trigger-Based Feedback NDP 6-80

Three-Dimensional Indoor Positioning with 802.11az Fingerprinting and Deep Learning 6-87

802.11az Positioning Using Super-Resolution Time of Arrival Estimation 6-105

Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation 6-119

Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation 6-133

802.11ad Packet Error Rate Simulation for Control PHY 6-144

802.11ad Single Carrier Link with RF Beamforming in Simulink 6-150

802.11p Packet Error Rate Simulation for a Vehicular Channel 6-158

802.11 Dynamic Rate Control Simulation 6-164

7 System-Level Simulation

Spatial Reuse with BSS Coloring in 802.11ax Network Simulation 7-2

802.11ax Downlink OFDMA Multinode System-Level Simulation 7-12

Noncollaborative Bluetooth LE Coexistence with WLAN Signal Interference 7-21

Get Started with WLAN System-Level Simulation in MATLAB 7-31

802.11ax Multinode System-Level Simulation of Residential Scenario . . . 7-40

802.11 MAC and Application Throughput Measurement . . . 7-49

802.11ax System-Level Simulation with Physical Layer Abstraction . . . 7-66

802.11ax PHY-Focused System-Level Simulation . . . 7-77

Physical Layer Abstraction for System-Level Simulation . . . 7-90

Generate and Visualize FTP Application Traffic Pattern . . . 7-105

Simulate an 802.11ax Network with Uplink and Downlink Application Traffic . . . 7-111

WLAN System-Level Simulation Statistics . . . 7-114
- Default Statistics . . . **7-114**
- Additional Statistics . . . **7-116**

Simulate an 802.11ax Network with Full MAC and Abstracted PHY . . . 7-118

Create, Configure, and Simulate an 802.11ax Mesh Network . . . 7-121

Simulate a Multiband 802.11ax Network . . . 7-124

Simulate an 802.11ax Hybrid Mesh Network . . . 7-127

8 Test and Measurement

Modeling and Testing an 802.11ax RF Receiver with 5G Interference . . . 8-2

Modeling and Testing an 802.11ax RF Transmitter . . . 8-21

802.11ac Receiver Minimum Input Sensitivity Test . . . 8-39

802.11ac Transmitter Measurements . . . 8-46

Generate Wireless Waveform in Simulink Using App-Generated Block . . . 8-62

802.11ad Transmitter Spectral Emission Mask Testing . . . 8-74

802.11p Spectral Emission Mask Testing . . . 8-79

9 Code Generation and Deployment

What is C Code Generation from MATLAB? **9-2**
Using MATLAB Coder **9-2**
C/C++ Compiler Setup **9-2**
Functions and System Objects That Support Code Generation **9-3**

Code Generation of WLAN Toolbox Features **9-4**

10 Software-Defined Radio

Image Transmission and Reception Using 802.11 Waveform and SDR **10-2**

OFDM Wi-Fi Scanner Using SDR Preamble Detection **10-20**

OFDM Beacon Receiver Using Software-Defined Radio **10-33**

PHY Modeling

PHY Simulation of Bluetooth BR/EDR, LE, and WLAN Coexistence

This example shows you how to model homogenous and heterogeneous coexistence between Bluetooth® basic rate/enhanced data rate (BR/EDR), low energy (LE) , and wireless local area waveforms (WLAN) by using Bluetooth® Toolbox.

Using this example, you can:

- Perform Bluetooth BR/EDR and LE end-to-end simulation in the presence of Bluetooth BR/EDR, LE, or WLAN interference.
- Perform adaptive frequency hopping (AFH) by classifying the channels as "good" or "bad" based on the packet error rate (PER).
- Compute the bit error rate (BER) and signal-to-interference-plus noise ratio (SINR).
- Visualize the spectrum and spectrogram of the Bluetooth BR/EDR or LE waveform in the presence of interference.

Bluetooth and WLAN Coexistence

Bluetooth operates in the unlicensed 2.4 GHz industrial, scientific, and medical (ISM) band from 2.4 to 2.4835 GHz, which is also used by other technologies such as Zigbee and WLAN. Multiple homogenous and heterogeneous networks operating in this band are likely to coexist in a physical scenario. To mitigate interference, Bluetooth and WLAN implement AFH and carrier-sense multiple access with collision avoidance (CSMA/CA), respectively. AFH enables Bluetooth devices to improve their robustness to interference and avoid interfering with other devices in the 2.4 GHz ISM band. The basic principle is to classify interference channels as bad channels and discard them from the list of available channels. This classification mechanism of AFH enables a Bluetooth device to use 79 channels or fewer in BR/EDR mode and 40 channels or fewer in LE mode. The Bluetooth Core Specification [2 on page 1-16] allows a minimum of 20 channels in BR/EDR mode and 2 channels in LE mode. For more information about coexistence between Bluetooth and WLAN, see “Bluetooth-WLAN Coexistence” (Bluetooth Toolbox).

Coexistence mechanisms can be classified into two categories: collaborative or noncollaborative, depending on whether the involved networks operate independently of one another or coordinate their use of the spectrum. In noncollaborative coexistence, each network treats other networks as interference and performs interference mitigation techniques. In collaborative coexistence, all the networks collaborate and coordinate their use of the spectrum. This example illustrates a noncollaborative coexistence mechanism between homogenous and heterogeneous networks.

This example uses these terminologies:

- AWN - Affected wireless node, can be one of these:

- Bluetooth: BR, EDR 2Mbps, EDR 3Mbps, LE 1Mbps, LE 2Mbps, LE 500Kbps, and LE 125Kbps

- IWN - Interfering wireless node, can be one of these:

- Bluetooth:

BR, EDR 2Mbps, EDR 3Mbps, LE 1Mbps, LE 2Mbps, LE 500Kbps, and LE 125Kbps

- WLAN:

802.11b with 22 MHz bandwidth

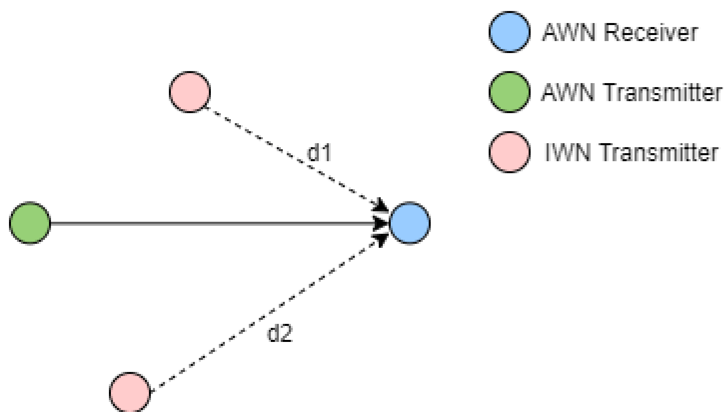
802.11g with 20 MHz bandwidth

802.11n with 20 MHz and 40 MHz bandwidths

802.11ax with 20 MHz and 40 MHz bandwidths

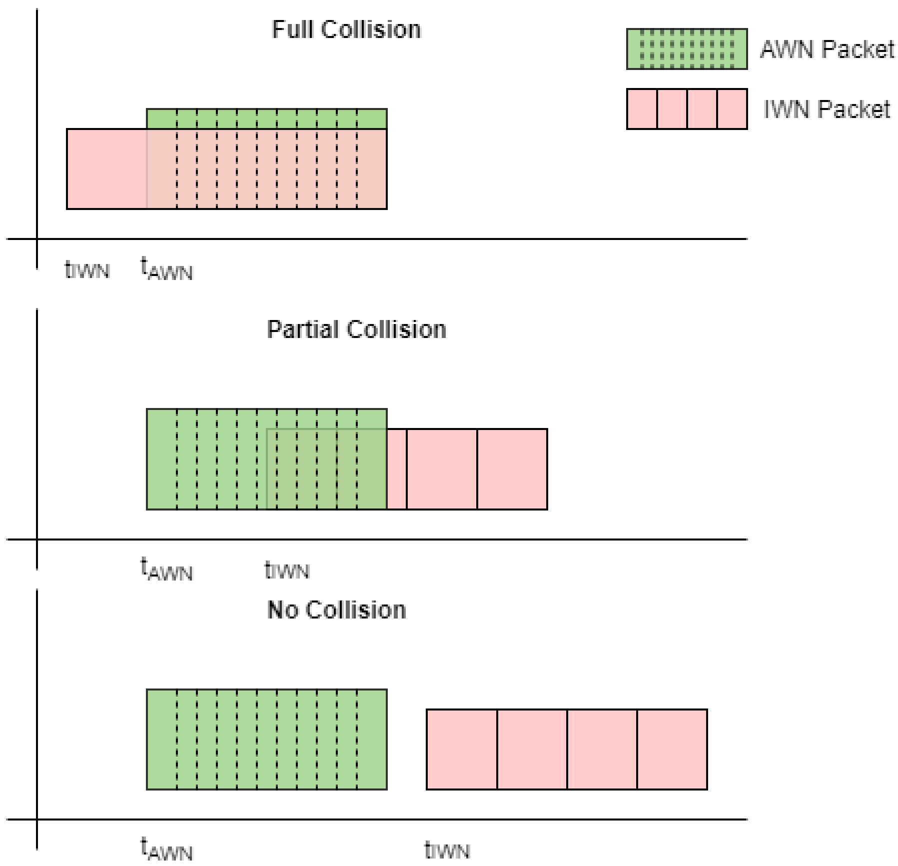
Impact of Interference in Space, Time, and Frequency Domains

Space: As the distance between AWN and IWN nodes increases, the impact of interference in space domain decreases. In this figure, if d1 and d2 increase, the impact of the IWN transmitter interference on the AWN receiver decreases.

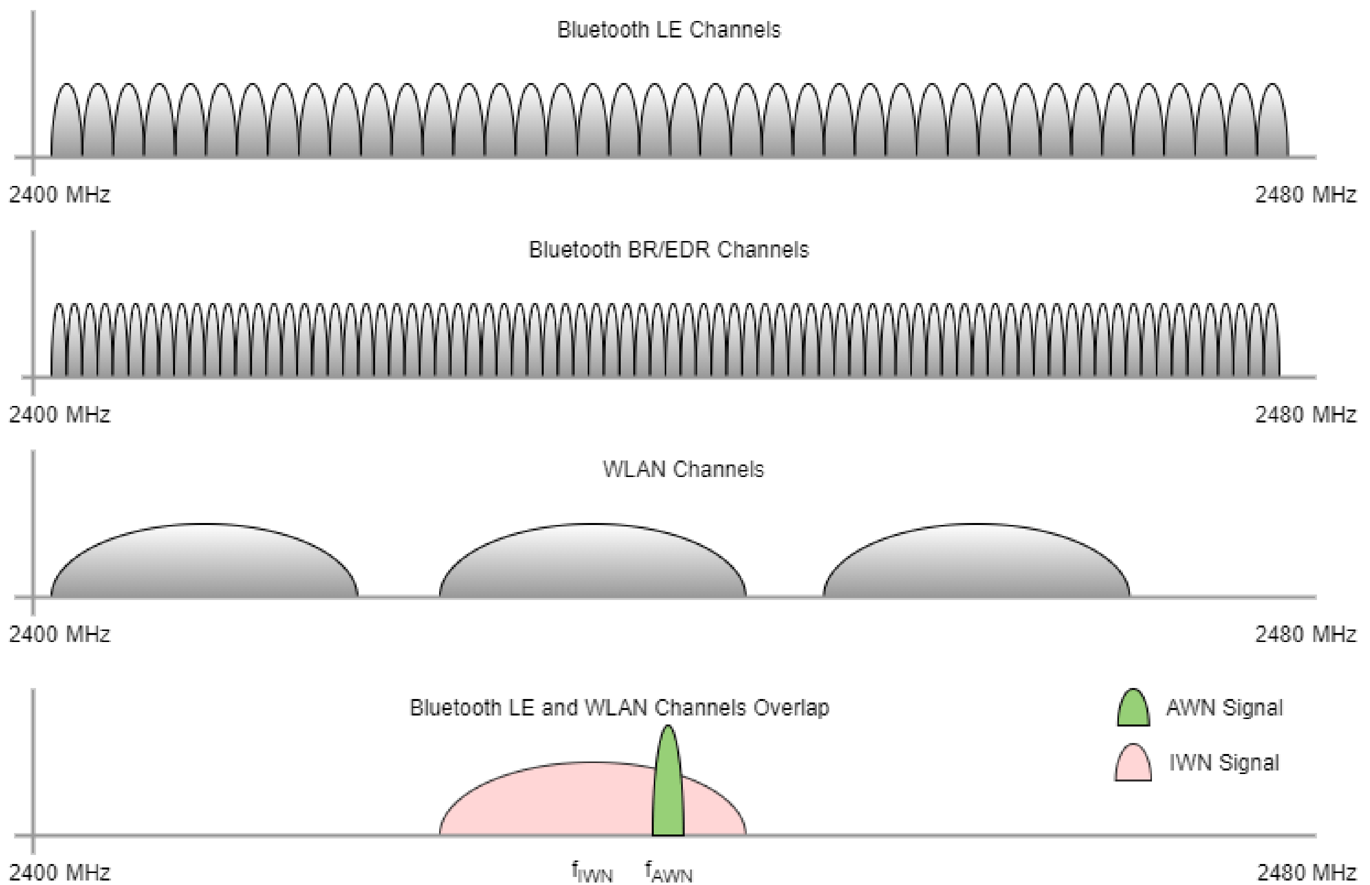


Time: Depending on the packet transmission timings, three possible collision probabilities arise in the time domain: full collision, partial collision, or no collision.

- Full - IWN packet completely interferes with the AWN packet.
- Partial - IWN packet partially interferes with the AWN packet with the given probability.
- No - IWN packet does not interfere with the AWN packet.



Frequency: As the channel separation between the AWN and IWN nodes increases, the impact of interference in frequency domain decreases. In this figure, if difference between f_{IWN} and f_{AWN} increase, the impact of the IWN transmitter interference on the AWN receiver decreases.



Simulation Parameters

Specify the AWN parameters such as the signal type, transmitter position, receiver position, transmitter power, and packet type.

Specify frequency hopping as one of these values.

- Off (default) - To run simulation at a fixed frequency, use this value. If you use this value, the example configures `awnFrequency`.
- On - To run simulation with AFH, use this value. If you use this value, the example does not configure `awnFrequency`.

```
awnSignalType = Bluetooth LE 1M ;
awnTxPosition = [0,0,0] ;      % In meters
awnRxPosition = [10,0,0] ;     % In meters
awnTxPower = 30 ;              % In dBm
awnPacket = Disabled ;
awnFrequencyHopping = Off ;
awnFrequency = 2440 *1e6; % In Hz
```

Configure a single or multiple IWNs and their respective parameters such as the signal type, transmitter position, fixed frequency of operation, and transmitter power. Create and configure multiple IWN nodes by using the IWN structure with different indices.

Specify the collision probability in the range [0,1]. Any value between 0 and 1 simulates partial collision. To simulate full collision, set this value to 1. To disable interference and simulate with no collisions, set this value to 0.

Add different types of WLAN signals as interference using WLAN Toolbox™. If you do not have WLAN Toolbox™, use `WLANBasebandFile` to add 802.11ax signal.

```
iwn(1).SignalType = WLANBasebandFile ;
iwn(1).TxPosition = [20,0,0];       % In meters
iwn(1).Frequency = 2437e6;          % In Hz
iwn(1).TxPower = 30;                % In dBm
iwn(1).CollisionProbability = 1;    % Probability of collision in time, must be between [0,1]

iwn(2).SignalType = Bluetooth LE 1M ;
iwn(2).TxPosition = [25,0,0];       % In meters
iwn(2).Frequency = 2420e6;          % In Hz
iwn(2).TxPower = 30;                % In dBm
iwn(2).CollisionProbability = 0.2; % Probability of collision in time, must be between [0,1]
```

Specify the environment, bit energy to noise density ratio (Eb/No), sample rate, and number of packets.

```
environment = Outdoor ;
EbNo = 10;          % In dB
sampleRate = 80e6; % In Hz
numPackets = 500;
```

Set the seed for the random number generator.

```
rng default;
```

Configure the waveform transmission and reception parameters of the AWN.

```
phyFactor = 1+strcmp(awnSignalType,"LE2M");
sps = sampleRate/(1e6*phyFactor);                    % Samples per symbol
if sps > 8                                           % Decimation factor for the receiver fil
    decimationFactor = gcd(sps,8);
else
    decimationFactor = 1;
end

if any(strcmp(awnSignalType,["LE1M","LE2M","LE500K","LE125K"]))
    payloadLength = 100;                             % Length of the payload in bytes
    accessAddress = "01234567";                      % Access address
    accessAddBits = int2bit(hex2dec(accessAddress),32,false);

    % Derive channel index based on the AWN frequency
    channelIndexArray = [37 0:10 38 11:36 39];
    awnBandwidth = 2e6;
    channelIndex = channelIndexArray((awnFrequency-2402e6)/awnBandwidth+1);
```

```
    % Configure the receiver parameters in a structure
    rxCfg = struct(Mode=awnSignalType,SamplesPerSymbol=sps/decimationFactor,ChannelIndex=channel
        DFPacketType=awnPacket,AccessAddress=accessAddBits);
    rxCfg.CoarseFreqCompensator = comm.CoarseFrequencyCompensator(Modulation="OQPSK", ...
                                         SampleRate=sampleRate/decimationFactor, ...
                                         SamplesPerSymbol=2*rxCfg.SamplesPerSymbol, ...
                                         FrequencyResolution=100);
    rxCfg.PreambleDetector = comm.PreambleDetector(Detections="First");
else
    % Create and configure Bluetooth waveform generation parameters
    awnWaveformConfig = bluetoothWaveformConfig(Mode=awnSignalType,PacketType=awnPacket, ...
                        SamplesPerSymbol=sps);
    if strcmp(awnPacket,"DM1")
        awnWaveformConfig.PayloadLength = 17;            % Maximum length of DM1 packets in byte
    end
    payloadLength = getPayloadLength(awnWaveformConfig); % Length of the payload

    % Get the receiver configuration parameters
    rxCfg = getPhyConfigProperties(awnWaveformConfig);
    rxCfg.SamplesPerSymbol = sps/decimationFactor;
end
```

Estimate the AWN path loss.

```
% Estimate distance between AWN transmitter and AWN receiver
distanceAWNTxRx = sqrt(sum((awnTxPosition-awnRxPosition).^2));
[awnPathloss,pathlossdB] = helperBluetoothEstimatePathLoss(environment,distanceAWNTxRx);
```

Create and configure the IWN by using the `helperIWNConfig` object. Generate IWN waveforms by using the `generateIWNWaveform` method. Add the path loss based on the environment and node positions by using the `applyPathloss` method.

```
iwnConfig = helperIWNConfig(IWN=iwn,SampleRate=sampleRate,Environment=environment);
iwnWaveform = generateIWNWaveform(iwnConfig);
[iwnWaveformPL,iwnPathloss] = applyPathloss(iwnConfig,iwnWaveform,awnRxPosition);
```

Use the `bluetoothFrequencyHop` and `bleChannelSelection` objects to select a channel index for the transmission and reception of Bluetooth BR/EDR and LE waveforms, respectively.

```
if strcmp(awnFrequencyHopping,"On")
    if any(strcmp(awnSignalType,["LE1M","LE2M","LE500K","LE125K"]))
        frequencyHop = bleChannelSelection;   % Bluetooth LE channel index System object™
        numBTChannels = 37;                   % Number of Bluetooth LE channels
        minChannels = 2;                      % Minimum number of channels to classify
    else
        frequencyHop = bluetoothFrequencyHop; % Bluetooth BR/EDR channel index object
        frequencyHop.SequenceType = "Connection Adaptive";
        numBTChannels = 79;                   % Number of Bluetooth BR/EDR channels
        minChannels = 20;                     % Minimum number of channels to classify
        inputClock = 0;
        numSlots = 1*(any(strcmp(awnPacket,["ID","NULL","POLL","FHS","HV1","HV2", ...
                    "HV3","DV","EV3","DM1","DH1","AUX1","2-DH1","3-DH1","2-EV3","3-EV3"])))...
                    +(3*any(strcmp(awnPacket,["EV4","EV5","DM3","DH3","2-EV5","3-EV5","2-DH3", .
                    "3-DH3"])))+ (5*any(strcmp(awnPacket,["DM5","DH5","2-DH5","3-DH5"])));
        slotValue = numSlots*2;
        clockTicks = slotValue*2;             % Clock ticks (one slot is two clock ticks)
    end
end
```

Design a receiver filter to capture the AWN waveform.

```
if any(strcmp(awnSignalType,["EDR2M","EDR3M"]))
    rolloff = 0.4;
    span = 8;
    filterCoeff = rcosdesign(rolloff,span,sps,"sqrt");
else
    N = 200;                                   % Order
    Fc = 1.5e6/(1+strcmp(awnSignalType,"BR")); % Cutoff frequency
    flag = "scale";                            % Sampling flag
    alpha = 3;                                 % Window parameter

    % Create the window vector for the design algorithm
    win = gausswin(N+1,alpha);

    % Calculate the coefficients using the FIR1 function
    filterCoeff = fir1(N,Fc/(sampleRate/2),"low",win,flag);
end
firdec = dsp.FIRDecimator(decimationFactor,filterCoeff);
```

Compute the signal-to-noise ratio (SNR).

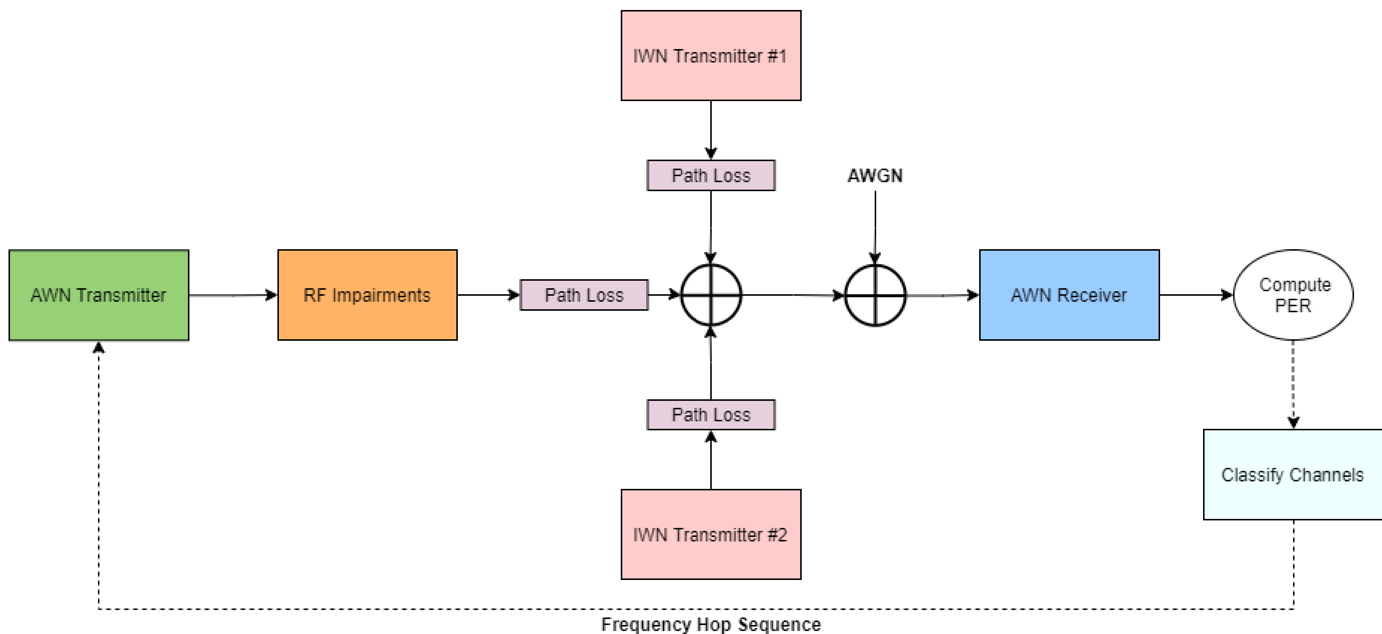
```
codeRate = 1*any(strcmp(awnSignalType,["LE1M","LE2M"]))+1/2*strcmp(awnSignalType,"LE500K")+1/8*s
           any(strcmp(awnSignalType,["BR","EDR2M","EDR3M"]))*(1-2/3*strcmp(awnPacket,"HV1")-...
           1/3*any(strcmp(awnPacket,["FHS","DM1","DM3","DM5","HV2","DV","EV4"]))); % Code rate
bitsPerSymbol = 1+ strcmp(awnSignalType,"EDR2M") + 2*(strcmp(awnSignalType,"EDR3M")); % Number o
snr = EbNo + 10*log10(codeRate) + 10*log10(bitsPerSymbol) - 10*log10(sps);
```

Create and configure the spectrum analyzer to visualize the spectrum and spectrogram of the Bluetooth BR/EDR or LE waveform in the presence of interference.

```
spectrumAnalyzer = dsp.SpectrumAnalyzer(...
    Name="Bluetooth Coexistence Modeling", ...
    ViewType="Spectrum and spectrogram", ...
    TimeResolutionSource="Property", ...
    TimeResolution=0.0005, ...
    SampleRate=sampleRate, ...
    TimeSpanSource="Property", ...
    TimeSpan=0.05, ...
    FrequencyResolutionMethod="WindowLength", ...
    WindowLength=512, ...
    AxesLayout="Horizontal", ...
    YLimits=[-100 20], ...
    ColorLimits=[-100 20]);
```

Coexistence Simulation

This diagram summarizes the example workflow.



Perform these steps to simulate the coexistence scenario.

1. Generate AWN (Bluetooth BR/EDR or LE) waveforms.
2. Distort each AWN waveform with these RF impairments: timing offset, carrier frequency offset, and DC offset.
3. Hop the waveform (frequency shift based on a center frequency of 2440 MHz) using the channel index derived from AFH.
4. Scale the hopped waveform with the transmitter power and path loss.
5. Generate and add IWN waveforms (Bluetooth BR/EDR, LE, or WLAN) based on the collision probabilities.
6. Add additive white Gaussian noise (AWGN).
7. Filter the noisy waveform.
8. Recover the bits from the filtered waveform by performing timing synchronization, carrier frequency offset correction, and DC offset correction.
9. Compute the PER, BER, and SINR.
10. If frequency hopping is on, classify the channels.

```
% Classify the channels for every |numPacketsToClassify| packets. If the PER of the
% channel is greater than |thresholdPER|, then map the corresponding channel
% as bad.
numPacketsToClassify = 50;
thresholdPER = 0.2;

% Create an instance of the error rate
errorRate = comm.ErrorRate;

% Initialize variables to perform the simulation
numErrors = 0;
numPktLost = 0;
countPER = 0;
```

```
countPreviousPER = 0;
midFrequency = 2440e6;
if strcmp(awnFrequencyHopping,"On")
    errorsBasic = deal(zeros(numBTChannels,3));
    errorsBasic(:,1) = (0:numBTChannels-1);
end

% Number of interfering nodes that collide with AWN
collisionCount = 0;
for index = 1:iwnConfig.NumIWNNodes
    collisionCount = collisionCount + (iwn(index).CollisionProbability > 0);
end

if strcmp(awnFrequencyHopping,"On") && collisionCount ~= 0
    sinr = zeros(numPackets,1);
end

% Loop to simulate multiple packets
for inum = 1:numPackets

    % Generate AWN waveform
    if any(strcmp(awnSignalType,["LE1M","LE2M","LE500K","LE125K"]))
        if strcmp(awnFrequencyHopping,"On")
            channelIndex = frequencyHop();
            channelFrequencies = [2404:2:2424 2428:2:2478 2402 2426 2480]*1e6;
            awnFrequency = channelFrequencies(channelIndex+1);
        end
        txBits = randi([0 1],payloadLength*8,1,"int8");
        awnWaveform = bleWaveformGenerator(int8(txBits),Mode=awnSignalType,ChannelIndex=channelI
            SamplesPerSymbol=sps,AccessAddress=accessAddBits,DFPacketType=awnPacket);
    else
        if strcmp(awnFrequencyHopping,"On")
            inputClock = inputClock + clockTicks;

            % Frequency hopping
            channelIndex = nextHop(frequencyHop,inputClock)
            awnFrequency = (2402+channelIndex)*1e6;

            % Generate whiten initialization vector from clock
            clockBinary = int2bit(inputClock,28,false).';
            awnWaveformConfig.WhitenInitialization = [clockBinary(2:7)'; 1];
        end
        txBits = randi([0 1],payloadLength*8,1);
        awnWaveform = bluetoothWaveformGenerator(txBits,awnWaveformConfig);
    end

    % Add timing offset
    timingOffset = randsrc(1,1,1:0.1:100);
    timingOffsetWaveform = helperBLEDelaySignal(awnWaveform,timingOffset);

    % Add frequency offset
    freqOffsetImp = randsrc(1,1,-10e3:100:10e3);
    freqOffsetWaveform = helperBLEFrequencyOffset(timingOffsetWaveform,sampleRate,freqOffsetImp)

    % Add DC offset
    dcValue = (5/100)*max(freqOffsetWaveform);
    dcWaveform = freqOffsetWaveform + dcValue;
```

```
    % Shift the waveform by making 2440 MHz as the mid frequency
    freqOffset = awnFrequency-midFrequency;
    hopWaveform = helperBLEFrequencyOffset(dcWaveform,sampleRate,freqOffset);

    % Scale the waveform as per the transmitter power and path loss
    soiAmplitudeLinear = 10^((awnTxPower-30)/20)/awnPathloss;
    attenAWNWaveform = soiAmplitudeLinear*hopWaveform;

    % Add IWN waveforms to AWN waveform
    addIWN2AWN = addInterference(iwnConfig,attenAWNWaveform,iwnWaveformPL,timingOffset);

    % Frequency shift the waveform by |-freqOffset|
    freqShiftWaveform = helperBLEFrequencyOffset(addIWN2AWN,sampleRate,-freqOffset);

    % Add AWGN
    soiPower = 20*log10(soiAmplitudeLinear);
    noisePower = soiPower - snr;
    splusibyn = 10*log10(var(freqShiftWaveform))-noisePower;
    noisyWaveform = awgn(freqShiftWaveform,splusibyn,"measured");

    % Apply filter
    if rem(length(noisyWaveform),sps)
        remainder = sps-rem(length(noisyWaveform),sps);
        noisyWaveform = [noisyWaveform;zeros(remainder,1)]; %#ok<AGROW>
    end
    delay = floor(length(firdec.Numerator)/(2*decimationFactor));
    noisyWaveformPadded = [noisyWaveform;zeros(delay*decimationFactor,1)];
    filteredWaveform = firdec(noisyWaveformPadded);
    release(firdec)
    filteredWaveform = filteredWaveform(1+delay:end)*sqrt(decimationFactor);

    % Recover the data bits
    if any(strcmp(awnSignalType,["LE1M","LE2M","LE500K","LE125K"]))
        rxCfg.ChannelIndex = channelIndex;
        [rxBits,accAddress] = helperBLEPracticalReceiver(filteredWaveform,rxCfg);
        if isempty(rxBits) || ~isequal(accessAddBits,accAddress)
            pktStatus = [];
        end
    else
        % Get PHY properties
        rxCfg.WhitenInitialization = awnWaveformConfig.WhitenInitialization;
        [rxBits,~,pktStatus]...
                            = helperBluetoothPracticalReceiver(filteredWaveform,rxCfg);
    end
```

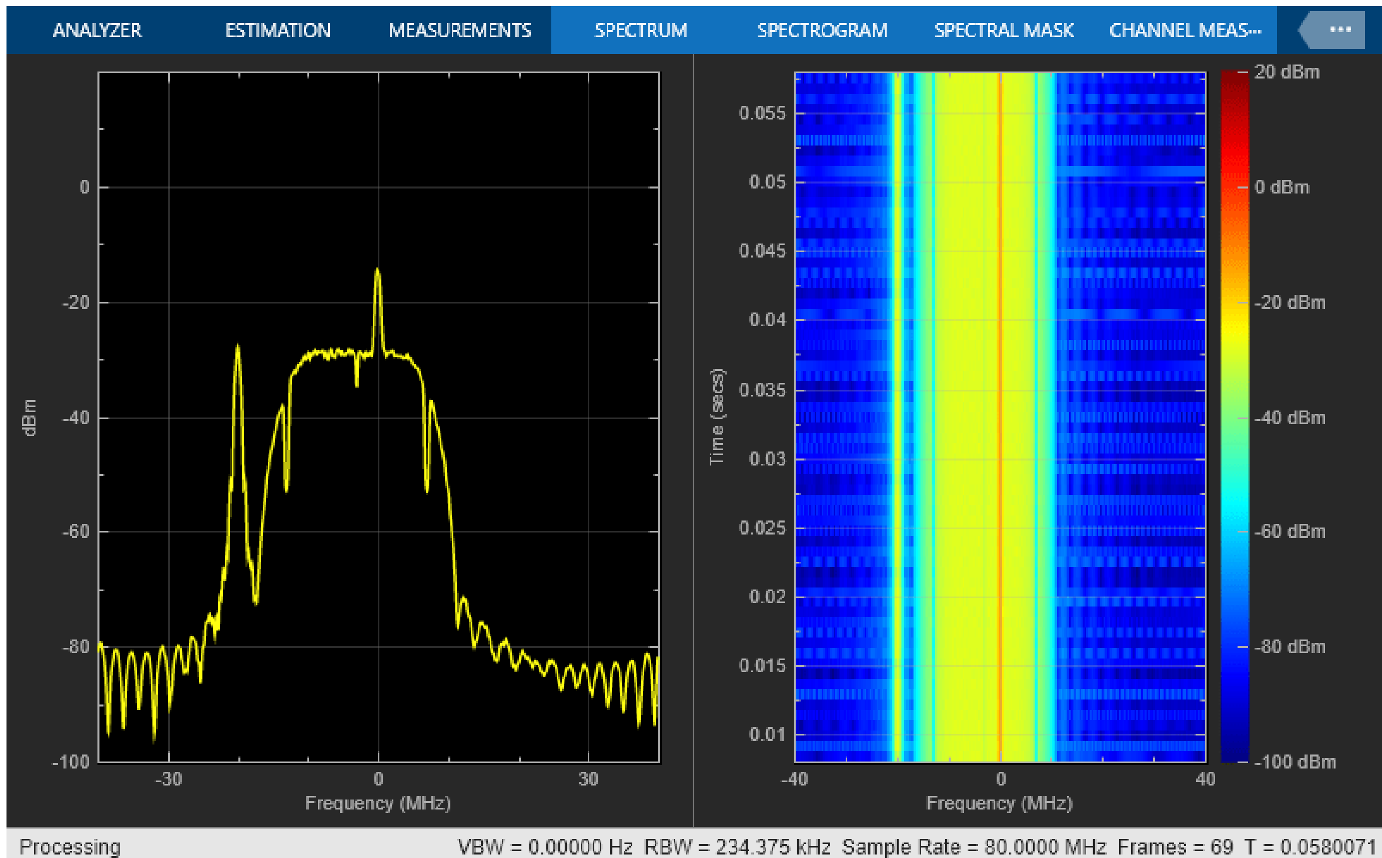
Simulation Results

Compute the BER and PER for each packet. If frequency hopping is on,

- Perform channel classification for every `numPacketsToClassify` based on the PER.
- Compute the SINR for each packet.
- Visualize the Bluetooth BR/EDR or LE waveform with the interference.

```
    % Compute BER and PER
    lengthTx = length(txBits);
    lengthRx = length(rxBits);
    lengthMinimum = min(lengthTx,lengthRx)-1;
```

```
        countPreviousPER = countPER;
        if lengthTx && lengthRx
            vectorBER = errorRate(txBits(1:lengthMinimum),rxBits(1:lengthMinimum));
            currentErrors = vectorBER(2)-numErrors;    % Number of errors in current packet
            if currentErrors || (lengthTx ~= lengthRx) % Check if current packet is in error or not
                countPER  = countPER+1;                % Increment the PER count
            end
            numErrors = vectorBER(2);
        elseif ~isempty(pktStatus)
            countPER  = countPER+~pktStatus;           % Increment the PER count
        else
            numPktLost = numPktLost+1;
        end

        % Perform frequency hopping
        if strcmp(awnFrequencyHopping,"On")
            chIdx = channelIndex+1;
            if countPreviousPER ~= countPER
                errorsBasic(chIdx,3) = errorsBasic(chIdx,3)+1;
            end

            % Classify the channels
            if any(inum == (1:floor(numPackets/numPacketsToClassify))*numPacketsToClassify)
                channelMap = errorsBasic(:,3)/numPacketsToClassify > thresholdPER;
                if nnz(channelMap) == 0
                    continue;
                end
                badChannels = find(channelMap)-1;
                if length(frequencyHop.UsedChannels)-length(badChannels) < minChannels
                    errorsBasic(badChannels+1,3) = 0;
                    usedChannels = 0:36;
                else
                    errorsBasic(badChannels+1,3) = 0;
                    usedChannels = setdiff(frequencyHop.UsedChannels,badChannels);
                end
                frequencyHop.UsedChannels = usedChannels;
            end
        end

        % Visualize the spectrum and spectrogram. Compute SINR.
        if strcmp(awnFrequencyHopping,"On") && collisionCount ~= 0
            sinr(inum) = helperBluetoothSINREstimate(snr,awnTxPower,awnFrequency,pathlossdB,iwnConfig
            spectrumAnalyzer(addIWN2AWN)
        elseif (strcmp(awnFrequencyHopping,"Off") && inum < 70) || (strcmp(awnFrequencyHopping,"On")
            if inum == 1
                sinr = helperBluetoothSINREstimate(snr,awnTxPower,awnFrequency,pathlossdB,iwnConfig,i
            end
            spectrumAnalyzer(addIWN2AWN)
        end
    end
```




```
% Compute BER and PER
if ~any(strcmp(awnPacket,["ID","NULL","POLL"]))
    if numPackets ~= numPktLost
        per = countPER/(numPackets-numPktLost);
        ber = vectorBER(1);
        fprintf('Mode %s, Simulated for Eb/No = %d dB, Obtained BER: %d, Obtained PER: %d\n',awnS
    else
        fprintf('No Bluetooth packets were detected.\n');
    end
else
    if numPackets ~= numPktLost
        per = countPER/(numPackets-numPktLost);
        fprintf('Mode %s, Simulated for Eb/No = %d dB, Obtained PER: %d\n',awnSignalType,EbNo,per
    else
        fprintf('No Bluetooth packets were detected.\n');
    end
end
```

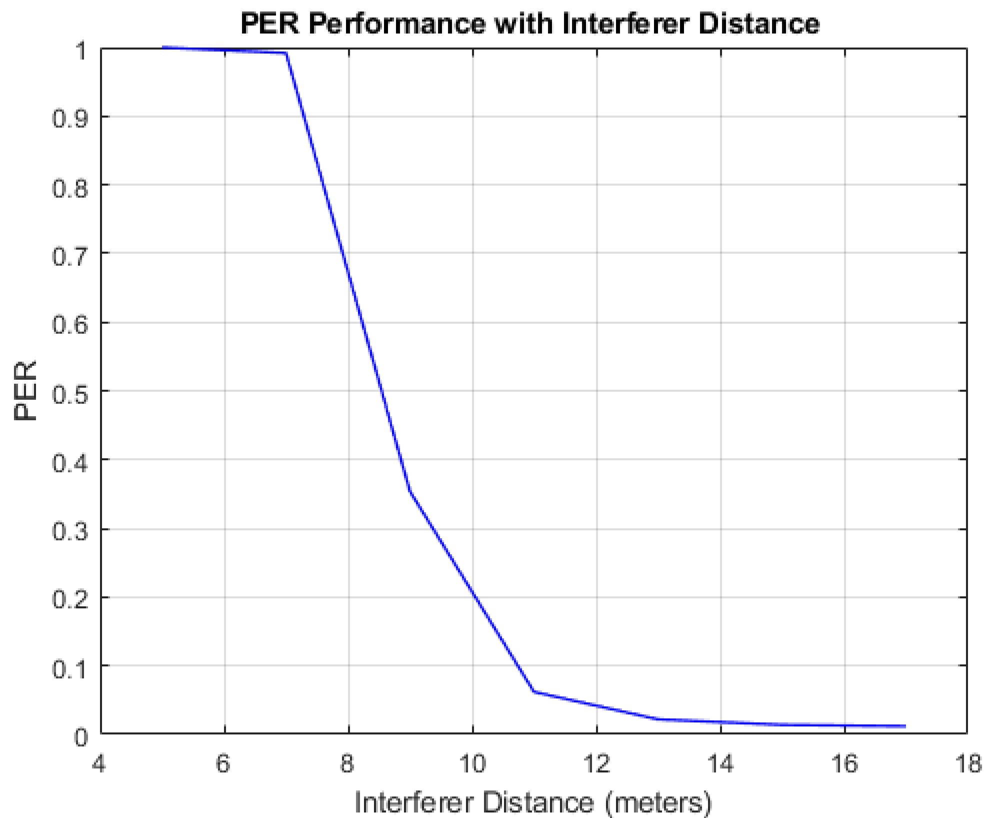
```
Mode LE1M, Simulated for Eb/No = 10 dB, Obtained BER: 4.205257e-04, Obtained PER: 8.980000e-01
```

This example simulates an end-to-end link for Bluetooth BR/EDR and LE with Bluetooth BR/EDR, LE, or WLAN waveforms as interference. You can implement AFH to mitigate interference by classifying channels as good or bad based on the PER value.

Further Exploration

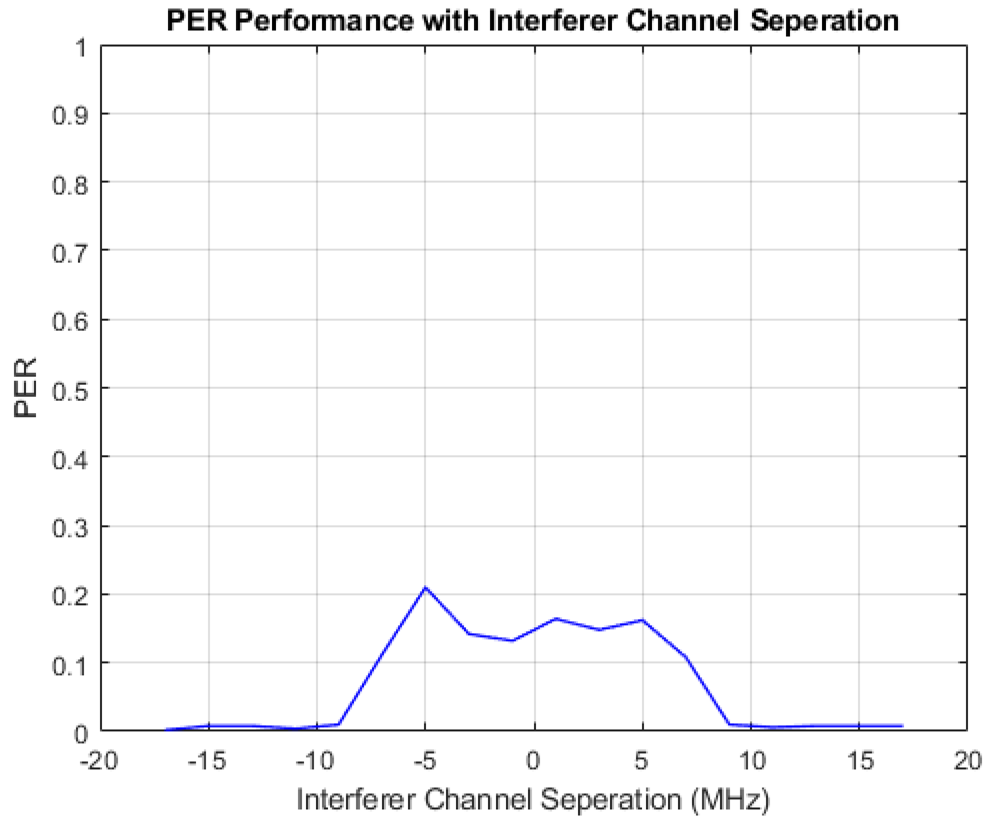
To observe the PER performance with interferer distance, you can run the simulation for different IWN transmitter positions. This plot shows the impact of interferer distance on PER given:

- Bluetooth LE1M AWN, 10 meters distance between the AWN transmitter and receiver, an AWN frequency of 2440 MHz, and an Eb/No value of 10 dB.
- 802.11 g with 20 MHz bandwidth IWN, an IWN frequency of 2437 MHz (co-channel interference), and a collision probability of 1.



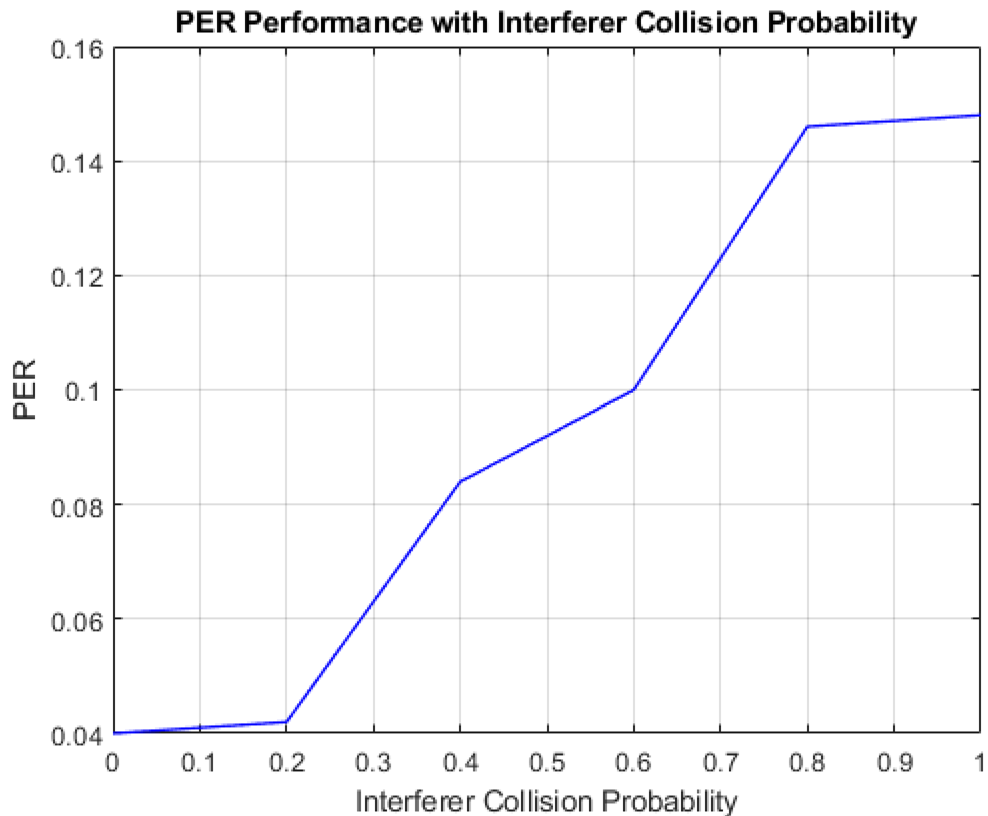
To observe the PER performance with interferer channel separation, you can run the simulation for different IWN frequencies (when frequency hopping is off). This plot shows the impact of interferer channel separation on PER given:

- Bluetooth LE1M AWN, 10 m distance between the AWN transmitter and receiver, and an Eb/No value of 10 dB.
- 802.11g with 20 MHz bandwidth IWN, an IWN frequency of 2437 MHz, a collision probability of 1, and 10 m distance between the AWN and IWN transmitters.



To observe the PER performance with collision probability, you can run the simulation for different collision probabilities. This plot shows the impact of collision probability on PER by considering:

- Bluetooth LE1M AWN, an AWN frequency of 2440 MHz, 10 m between the AWN transmitter and receiver, and an Eb/No value of 10 dB.
- 802.11g with 20 MHz bandwidth IWN, an IWN frequency of 2437 MHz, and 10 m between the AWN and IWN transmitters.



Appendix

The example uses these helper functions:

- helperIWNConfig: Interference wireless node configuration parameters
- helperBLEDelaySignal: Introduce time delay in the signal
- helperBLEFrequencyOffset: Apply frequency offset to the input signal
- helperBLEPracticalReceiver: Demodulate and decode the received signal
- helperBluetoothPracticalReceiver: Detect, synchronize, and decode the received Bluetooth BR/EDR waveform
- helperBluetoothEstimatePathLoss: Estimate the path loss between the node and locator
- helperBluetoothSINREstimate: Estimate SINR

Selected Bibliography

1 Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 15, 2021. https://www.bluetooth.com.

2 Bluetooth Special Interest Group (SIG). "Core System Package [Low Energy Controller Volume]". Bluetooth Core Specification. Version 5.3, Volume https://www.bluetooth.com.

See Also

Related Examples

- “Noncollaborative Bluetooth LE Coexistence with WLAN Signal Interference” on page 7-21

802.11be Waveform Generation

This example shows how to parameterize and generate IEEE® 802.11be™ extremely high throughput (EHT) multi-user waveforms.

Introduction

802.11be

The physical layer of 802.11be (Wi-Fi 7) [1 on page 1-47] is an extension of 802.11ax (Wi-Fi 6) [2 on page 1-47], focusing on WLAN indoor and outdoor operation with a maximum throughput of 30 Gbps [1 on page 1-47]. EHT devices coexist with legacy 802.11-compliant devices operating in the 2.4 GHz, 5 GHz, and 6 GHz unlicensed spectra. The standard introduces these key new features in the EHT format [1 on page 1-47].

- 320 MHz channel bandwidth
- Multiple Resource Units (MRUs) per stations (STAs)
- 4096-QAM
- Non-OFDMA preamble puncturing

IEEE P802.11be/D2.0 [1 on page 1-47] specifies two packet formats.

- EHT multi-user (EHT MU)
- EHT trigger-based (EHT TB)

The EHT MU packet can be configured for OFDMA transmission, non-OFDMA transmission, or a combination of the two. With this flexibility, an EHT MU packet can transmit to a single user over the whole band, multiple users over different parts of the band (OFDMA), or multiple users over the same part of the band (MU-MIMO). This example shows how to generate an EHT MU packet for each of these different waveforms and demonstrates some of the key features of the draft standard [1 on page 1-47].

The EHT TB packet allows for OFDMA or MU-MIMO transmission in the uplink. The access point (AP) controls EHT TB transmissions. A trigger frame sent to all STAs participating in the transmission contains all the parameters required for the transmission. Each STA transmits an EHT TB packet simultaneously when triggered by the AP. This example shows how to generate an EHT TB transmission in response to a trigger frame for four users in an OFDMA/MU-MIMO transmission.

Subcarriers and Resource Allocation

As in HE transmissions, the channel bandwidth of an EHT transmission is divided into RUs. An RU is a group of subcarriers assigned to one or more users. An RU is defined by the size (the number of subcarriers) and an index. The RU index specifies the location of the RU within the channel. For example, an 80 MHz transmission contains four possible 242-tone RUs, one in each 20 MHz subchannel. RU# 242-1 (size 242, index 1) is the RU occupying the lowest absolute frequency within the 80 MHz, and RU# 242-4 (size 242, index 4) is the RU occupying the highest absolute frequency. The draft standard defines possible sizes and locations of RUs in section 36.3.2 of [1 on page 1-47].

In EHT, the RU sizes in tones are 26, 52, 106, 242, 484, 996, 2x996, or 4x996, where 242, 484, 996, 2x996, and 4x996 correspond to the entire 20 MHz, 40 MHz, 80 MHz, 160 MHz, and 320 MHz channel bandwidth, respectively. An EHT AP can assign 26-, 52-, or 106-tone RUs to a single user, or

242-, 484-, 996-, 2x996-, or 4x996-tone RUs to a single user or multiple users in MU-MIMO configuration.

Unlike in HE, more than one RU can be assigned to a STA. RUs can be aggregated to form an MRU. RUs are classified as small-size RUs or large-size RUs.

Small-size RUs are 26-, 52-, and 106-tone RUs. Small-size RUs can only be combined with other small-size RUs to form a small-size MRU. Small-size MRUs are used for both uplink and downlink transmission in OFDMA format. These combinations of small-size MRUs can be assigned to a single STA:

- 52+26-tone MRU
- 106+26-tone MRU

Large-size RUs are 242-, 484-, 996-, 2x996-, and 4x996-tone RUs. Large-size RUs can only be combined with other large-size RUs to form a large-size MRU. The large-size MRUs are defined for both uplink and downlink transmission in non-OFDMA and OFDMA format. These combinations of large-size MRUs can be assigned to a STA as defined in section 36.3.2.2.3 of [1 on page 1-47]:

- 484+242-tone MRU
- 996+484-tone MRU
- 996+484+242-tone MRU (can only be assigned to non-OFDMA STAs)
- 2x996+484-tone MRU
- 3x996-tone MRU (2x996 + 996 tones)
- 3x996+484-tone MRU (2x996 + 996 + 484 tones)

The index and size of an MRU is the combined indices and sizes of the RUs that form it. For example, take a 20 MHz transmission that contains a 106+26-tone RU and a 106-tone RU. RU# 106+26 (size 106 and 26, index 1 and 5) is an MRU occupying the lowest absolute frequency within the 20 MHz, and RU# 106 (size 106, index 2) is the RU occupying the highest absolute frequency. The draft standard defines possible sizes and locations of RUs in section 36.3.2 of [1 on page 1-47]. See also Appendix A on page 1-43.

EHT Multi-User (EHT MU) Non-OFDMA Packet Format

A full-band non-OFDMA packet is an EHT MU packet that contains the transmission of a single user or multiple users in an MU-MIMO configuration over the full channel bandwidth.

Single-User Packet Generation

An EHT MU single-user (SU) packet is a full-band, non-OFDMA transmission to a single user. Use EHT MU configuration object `wlanEHTMUConfig` to configure the transmission parameters of a full-band EHT MU packet for a single user. Create a full-band 320 MHz single-user MIMO configuration and set the transmission parameters of the user.

```
cfgSU = wlanEHTMUConfig('CBW320');
numTx = 2; % Number of transmit antennas
cfgSU.NumTransmitAntennas = numTx;
cfgSU.User{1}.APEPLength = 8000; % A-MPDU length pre-EOF padding, in bytes
cfgSU.User{1}.MCS = 12;          % Modulation and coding scheme
cfgSU.User{1}.NumSpaceTimeStreams = numTx;
```

Get the required PSDU length for the specified transmission configuration and create a random PSDU of this length for transmission.

```
psdu = randi([0 1],psduLength(cfgSU)*8,1,'int8');
```

Generate the EHT MU single-user packet by using the `wlanWaveformGenerator` function.

```
tx = wlanWaveformGenerator(psdu,cfgSU); %#ok<*NASGU> % Create packet
```

MU-MIMO Packet Generation

Configure transmission parameters for a full-band non-OFDMA EHT MU packet for multiple users by using the EHT MU configuration object. Use the `NumUsers` property of `wlanEHTMUConfig` object to specify the number of users in an MU-MIMO configuration.

Create a 20 MHz (MU-MIMO) EHT MU configuration for two users and set parameters common to all users.

```
numTx = 2;
% Set common transmission parameters for all users
cfgMUMIMO = wlanEHTMUConfig('CBW20','NumUsers',2);
cfgMUMIMO.NumTransmitAntennas = numTx;
cfgMUMIMO.GuardInterval = 3.2;
cfgMUMIMO.EHTLTFType = 4;
```

The `ruInfo` object function provides details of the RUs in the configuration. In this case a single RU contains two users.

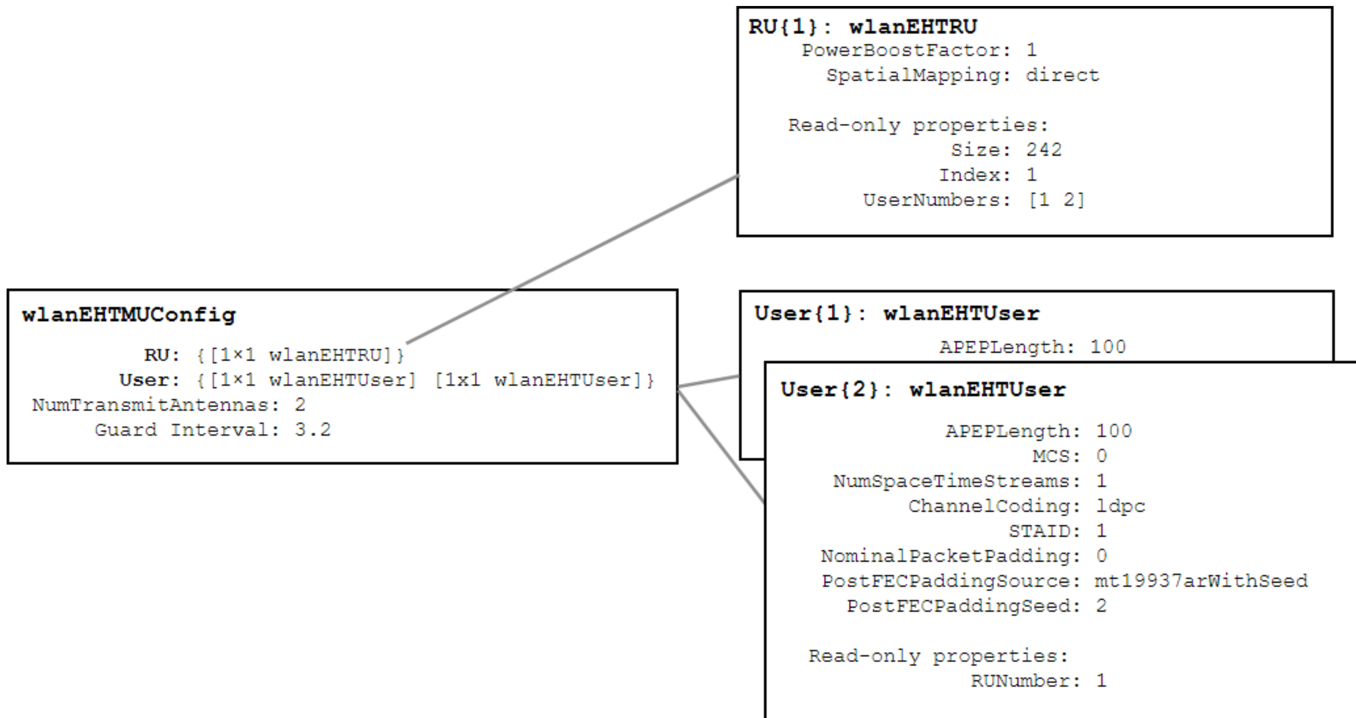
```
allocInfo = ruInfo(cfgMUMIMO);
disp('Allocation info:')
```

```
Allocation info:
```

```
disp(allocInfo)
```

```
                    NumUsers: 2
                      NumRUs: 1
                   RUIndices: {[1]}
                     RUSizes: {[242]}
               NumUsersPerRU: 2
    NumSpaceTimeStreamsPerRU: 2
       PowerBoostFactorPerRU: 1
                   RUNumbers: 1
```

The properties of `cfgMUMIMO` describe the transmission configuration. The `cfgMUMIMO.RU` and `cfgMUMIMO.User` properties of `cfgMUMIMO` are cell arrays. Each element of the cell arrays contains an object which configures an RU or a user. When you create the `cfgMUMIMO` object, the elements of `cfgMUMIMO.RU` and `cfgMUMIMO.User` create the desired number of RUs and users. Each element of `cfgMUMIMO.RU` is a `wlanEHTRU` object describing the configuration of an RU. Similarly, each element of `cfgMU.User` is a `wlanEHTUser` object describing the configuration of a user. This figure shows the object hierarchy.



This example creates a single RU in non-OFDMA format for a channel bandwidth of 20 MHz, so `cfgMUMIMO.RU` is a cell array with one element. The index and size of each RU are configured according to the channel bandwidth specified when `cfgMUMIMO` was created.

```
disp('RU configuration:')
```

```
RU configuration:
```

```
disp(cfgMUMIMO.RU{1})
```

```
  wlanEHTRU with properties:

    PowerBoostFactor: 1
      SpatialMapping: direct

   Read-only properties:
                Size: 242
               Index: 1
         UserNumbers: [1 2]
```

After creating the object, you can configure each RU to create the desired transmission configuration by setting the properties of the appropriate RU object. For example, the spatial mapping and power boost factor can be configured per RU.

Create a random spatial mapping array and configure RU index 1.

```
ruIndex = 1;
ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfgMUMIMO,ruIndex);
numST = ofdmInfo.NumTones; % Number of occupied subcarriers
numSTS = allocInfo.NumSpaceTimeStreamsPerRU(ruIndex);
```

```
cfgMUMIMO.RU{ruIndex}.SpatialMapping = 'Custom';
cfgMUMIMO.RU{ruIndex}.SpatialMappingMatrix = rand(numST,numSTS,numTx);
```

In this section, the `NumUsers` property specifies two users in the transmission, so `cfgMUMIMO.User` contains two elements. The transmission properties of users can be configured by modifying individual user objects, for example the modulation and coding scheme (MCS), APEP length, and channel coding scheme.

Set transmission properties of user 1.

```
cfgMUMIMO.User{1}.APEPLength = 500; % A-MPDU length pre-EOF padding in bytes
cfgMUMIMO.User{1}.MCS = 12;
cfgMUMIMO.User{1}.ChannelCoding = 'LDPC';
cfgMUMIMO.User{1}.NumSpaceTimeStreams = 1;
```

Set transmission properties of user 2.

```
cfgMUMIMO.User{2}.APEPLength = 700; % A-MPDU length pre-EOF padding in bytes
cfgMUMIMO.User{2}.MCS = 7;
cfgMUMIMO.User{2}.ChannelCoding = 'BCC';
cfgMUMIMO.User{2}.NumSpaceTimeStreams = 1;
```

The read-only `RUNumber` property indicates which RU is used to transmit this user.

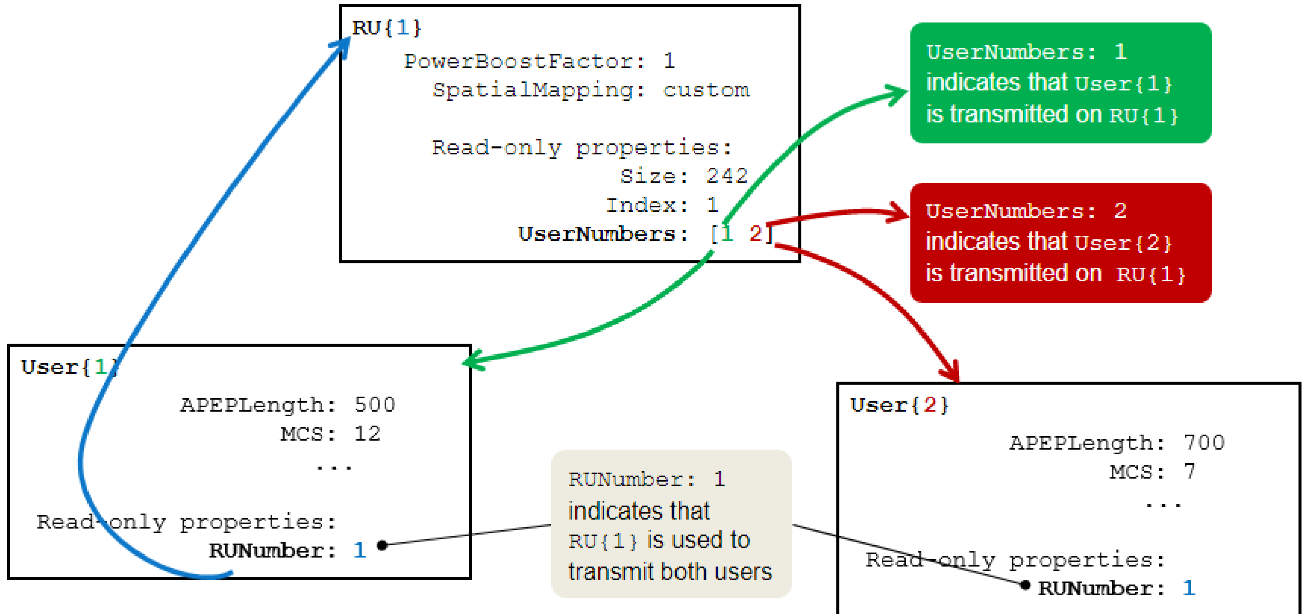
```
disp('First user configuration:')
```

```
First user configuration:
```

```
disp(cfgMUMIMO.User{1})
```

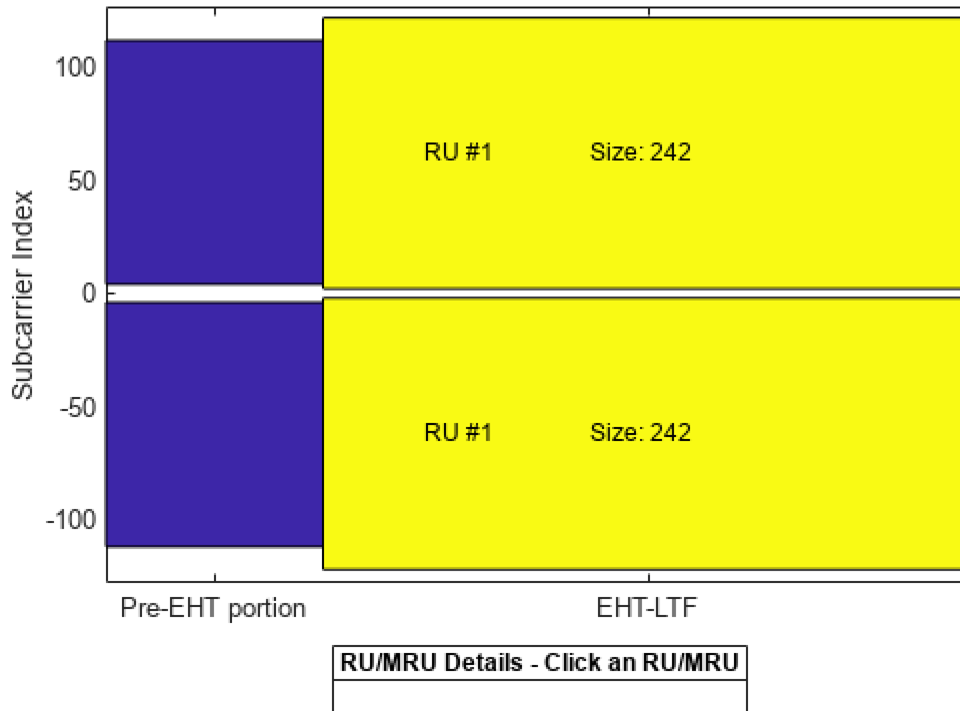
```
  wlanEHTUser with properties:

             APEPLength: 500
                    MCS: 12
    NumSpaceTimeStreams: 1
          ChannelCoding: ldpc
                  STAID: 0
   NominalPacketPadding: 0
   PostFECPaddingSource: mt19937arwithseed
     PostFECPaddingSeed: 1

   Read-only properties:
               RUNumber: 1
```

In this example the non-OFDMA configuration consists of a single RU and two users. The `UserNumbers` property of an RU object indicates which users (elements of the `cfgMUMIMO.User` cell array) are transmitted on that RU. Similarly, the `RUNumber` property of each `User` object indicates which RU (element of the `cfgMUMIMO.RU` cell array) is used to transmit the user. For more information, see the “802.11ax Waveform Generation” on page 1-64 example.



The `showAllocation` object function visualizes the occupied RUs or MRUs and subcarriers for the specified configuration. The colored blocks illustrate the occupied subcarriers in the pre-EHT and EHT portions of the packet. White indicates unoccupied subcarriers. The pre-EHT portion illustrates the occupied subcarriers in the fields preceding the EHT-STF. The EHT portion illustrates the occupied subcarriers in the EHT-STF, EHT-LTF, and EHT-Data field and therefore shows the RU/MRU allocation. To display information about an RU or MRU, click on the RU or MRU. The RU number corresponds to the *i*th `RU` element of the `cfgMUMIMO.RU` property. The `showAllocation` object function also shows the size of each RU or MRU, and the users assigned to each. The user number corresponds to the *i*th `User` element of the `cfgMUMIMO`.

```
showAllocation(cfgMUMIMO);
```



To generate a non-OFDMA EHT MU waveform, create a random PSDU for each user. Use a cell array to store the PSDU for each user as the PSDU lengths differ. The `psduLength` object function returns a vector with the required PSDU per user for the specified configuration.

```
psduLen = psduLength(cfgMUMIMO);
psdu = cell(1,allocInfo.NumUsers);
for i = 1:allocInfo.NumUsers
    psdu{i} = randi([0 1],psduLen(i)*8,1,'int8');
end
```

Generate a 20 MHz non-OFDMA (MU-MIMO) packet with two users.

```
tx = wlanWaveformGenerator(psdu,cfgMUMIMO);
```

EHT DUP Mode Packet Generation

An EHT DUP mode transmission is a non-OFDMA transmission to a single user where the data payload is duplicated in frequency. EHT DUP mode is only applicable for 80 MHz, 160 MHz, and 320 MHz transmissions. Configure the transmission of an EHT DUP using the EHT MU configuration object by setting the `EHTDUPMode` property of `wlanEHTMUConfig` to true. Create a 320 MHz EHT DUP mode configuration and set the transmission parameters of the user. Note the MCS of the user is automatically set to 14.

```
cfgDUPMode = wlanEHTMUConfig('CBW320','EHTDUPMode',true);
cfgDUPMode.User{1}.APEPLength = 200;
disp(cfgDUPMode.User{1}.MCS)
```

```
    14
```

Get the required PSDU length for the specified transmission configuration and create a random PSDU of this length for transmission.

```
psdu = randi([0 1],psduLength(cfgDUPMode)*8,1,'int8');
```

Generate the EHT DUP mode packet by using the `wlanWaveformGenerator` function.

```
tx = wlanWaveformGenerator(psdu,cfgDUPMode); %#ok<*NASGU> % Create packet
```

Punctured Packet Generation with Large MRUs

The draft standard allows for punctured 20 MHz subchannels in a non-OFDMA or OFDMA 80 MHz, 160 MHz, or 320 MHz transmission to allow a legacy system to operate in the punctured subchannel. The preamble portion of the punctured 20 MHz subchannel is not transmitted and the corresponding RU is not allocated to any user (no data transmitted). This method is also described as channel bonding.

In a non-OFDMA format, large-size MRUs are obtained by puncturing any one of the 242-, 484-, or 996-tone RUs within an 80 MHz, 160 MHz, or 320 MHz channel bandwidth. As an example, a 996+484+242-tone MRU is obtained by puncturing any one of eight 242-tone RUs in the 160 MHz channel, see section 36.3.12.11.3 of [1 on page 1-47]. The preamble is not transmitted in the punctured subchannel within a large MRU and the corresponding RU is not allocated to any user. The non-OFDMA packet format supports single-user and MU-MIMO packet transmission in large MRUs. The EHT-USIG field of an EHT MU packet carries the punctured sub-channel indication for a non-OFDMA transmission.

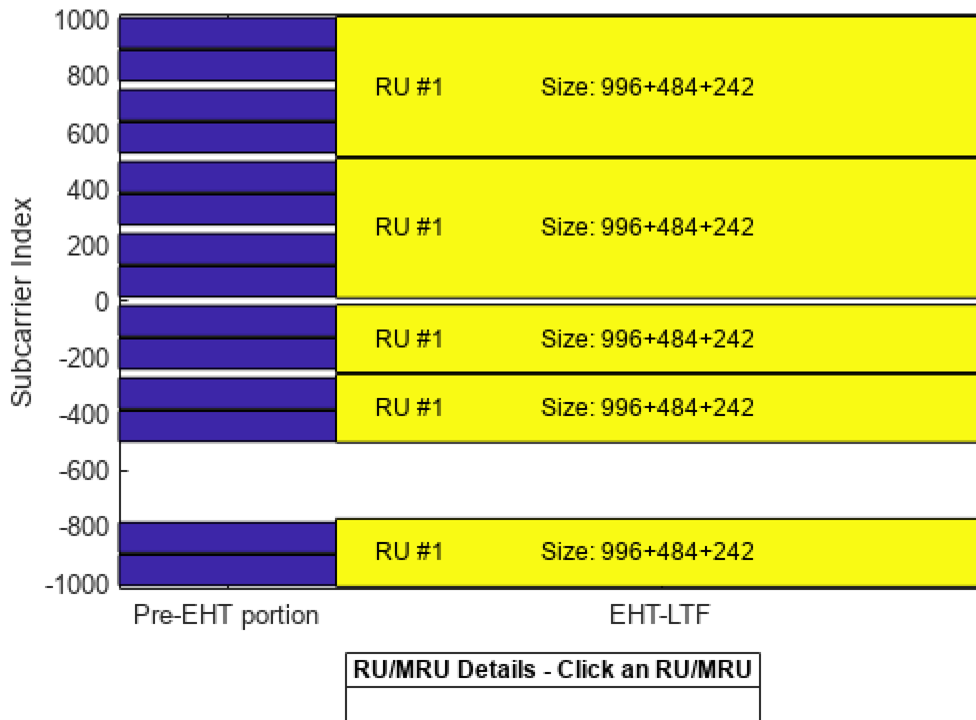
Use the EHT MU configuration object to configure the non-OFDMA transmission properties of an EHT MU packet transmission on a large MRU. Appendix B on page 1-45 lists possible puncturing patterns for 80 MHz, 160 MHz, and 320 MHz channel bandwidths and the corresponding value of the `PuncturedChannelFieldValue` property when creating the `wlanEHTMUConfig` object.

Create a configuration for a 160 MHz EHT MU packet transmission on a large 996+484+242-tone MRU. Puncture the second 20 MHz subchannel in a 160 MHz channel as shown in Appendix B on page 1-45 by setting the `PuncturedChannelFieldValue` property to `2`.

```
cfgPunc = wlanEHTMUConfig('CBW160','PuncturedChannelFieldValue',2);
tx = wlanWaveformGenerator([1 0 1 0],cfgPunc);
```

View the punctured RU allocation.

```
showAllocation(cfgPunc);
```



Create a configuration for two users in MU-MIMO configuration in a 320 MHz channel. The EHT MU packet transmission is on a large 3x996+484-tone MRU, see section 36.3.2.2.3 of [1 on page 1-47]. Create the `wlanEHTMUConfig` object, setting the `PuncturedChannelFieldValue` property to `3` to puncture the third 40 MHz subchannel in a 320 MHz channel as shown in Appendix B on page 1-45.

```
cfg = wlanEHTMUConfig('CBW320','NumUsers',2,'PuncturedChannelFieldValue',3);
cfg.NumTransmitAntennas = 8;
```

Set transmission properties of user 1.

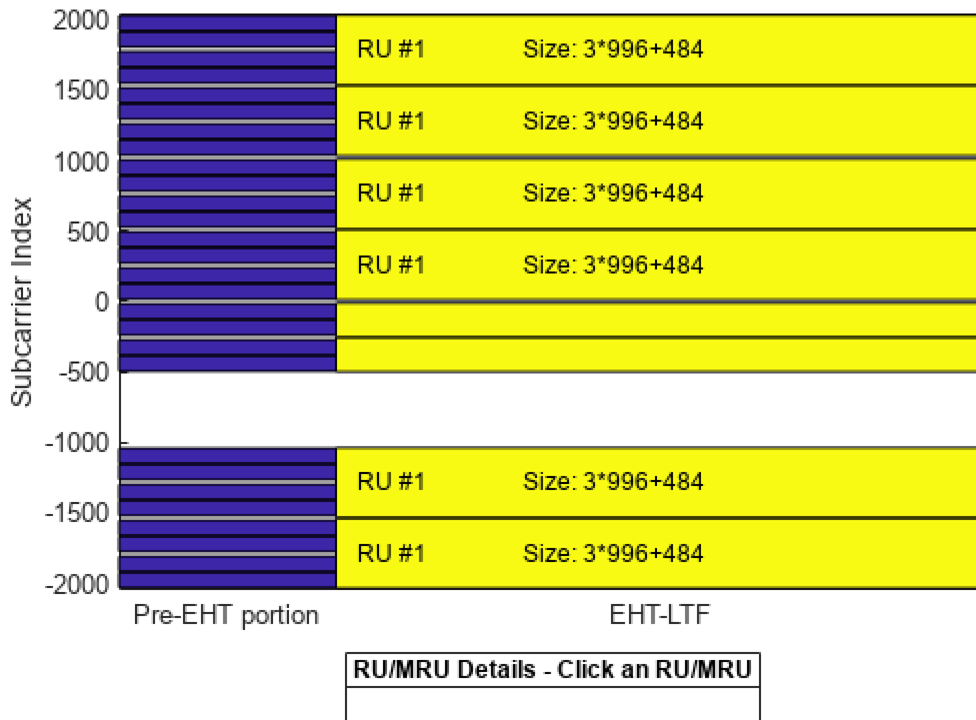
```
cfg.User{1}.NumSpaceTimeStreams = 4;
cfg.User{1}.APEPLength = 500000;
cfg.User{1}.MCS = 12;
```

Set transmission properties of user 2.

```
cfg.User{2}.NumSpaceTimeStreams = 4;
cfg.User{2}.APEPLength = 400000;
cfg.User{2}.MCS = 13;
```

View the punctured RU allocation.

```
showAllocation(cfg);
```



Create a random PSDU for each user. Use a cell array to store the PSDU for each user as the PSDU lengths differ. The `psduLength` object function returns a vector with the required PSDU per user for the specified configuration.

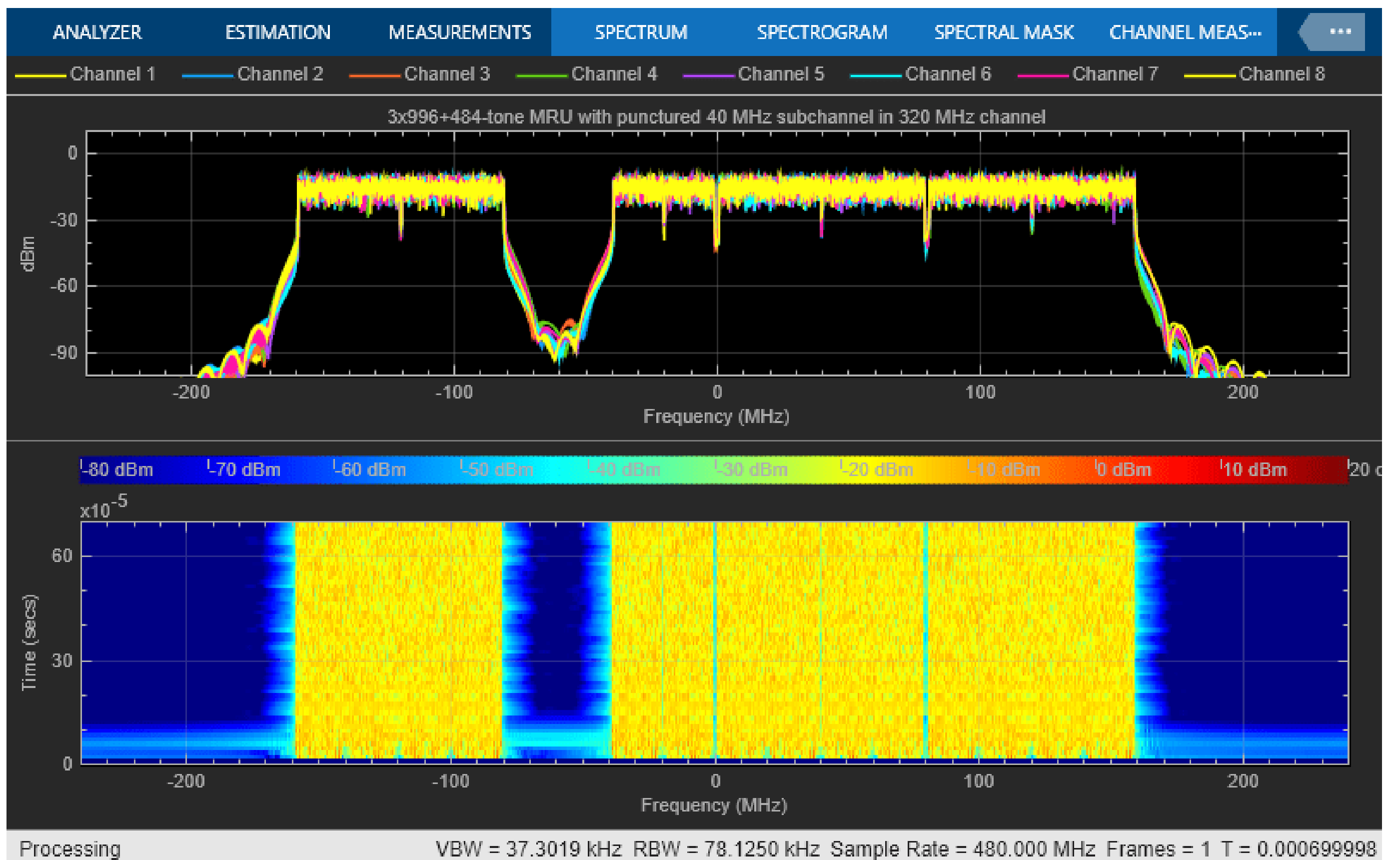
```
psduLen = psduLength(cfg);
allocInfo = ruInfo(cfg); % Get allocation information
psdu = cell(1,allocInfo.NumUsers);
for i=1:allocInfo.NumUsers
    psdu{i} = randi([0 1],psduLen(i)*8,1);
end
```

Create an oversampled waveform using the `wlanWaveformGenerator` function. This function generates an oversampled waveform by using a larger inverse fast Fourier transform (IFFT) size than required for the nominal baseband rate.

```
osf = 1.5;
tx = wlanWaveformGenerator(psdu,cfg,'OversamplingFactor',osf);
```

Generate the spectrum and spectrogram of the generated signal. The spectrogram shows the punctured 40 MHz subchannel in the 320 MHz channel.

```
sa = plotSpectrumAndSpectrogram(tx,cfg,osf,'3x996+484-tone MRU with punctured 40 MHz subchannel :
```



EHT Multi-User (EHT MU) OFDMA Packet Format

The allocation index defines the assignment of RUs/MRUs in an OFDMA transmission. Appendix A on page 1-43 lists the allocation indices for all RU/MRU assignments. For each 20 MHz subchannel, a 9-bit index describes the number and size of RUs/MRUs and the number of users transmitted on each RU/MRU. The length of allocation Index must be 1, 2, 4, 8, or 16 to define the assignment for each 20 MHz subchannel in a 20 MHz, 40 MHz, 80 MHz, 160 MHz, or 320 MHz channel bandwidth, respectively. The allocation index also determines which content channel the EHT-SIG field uses to signal a user.

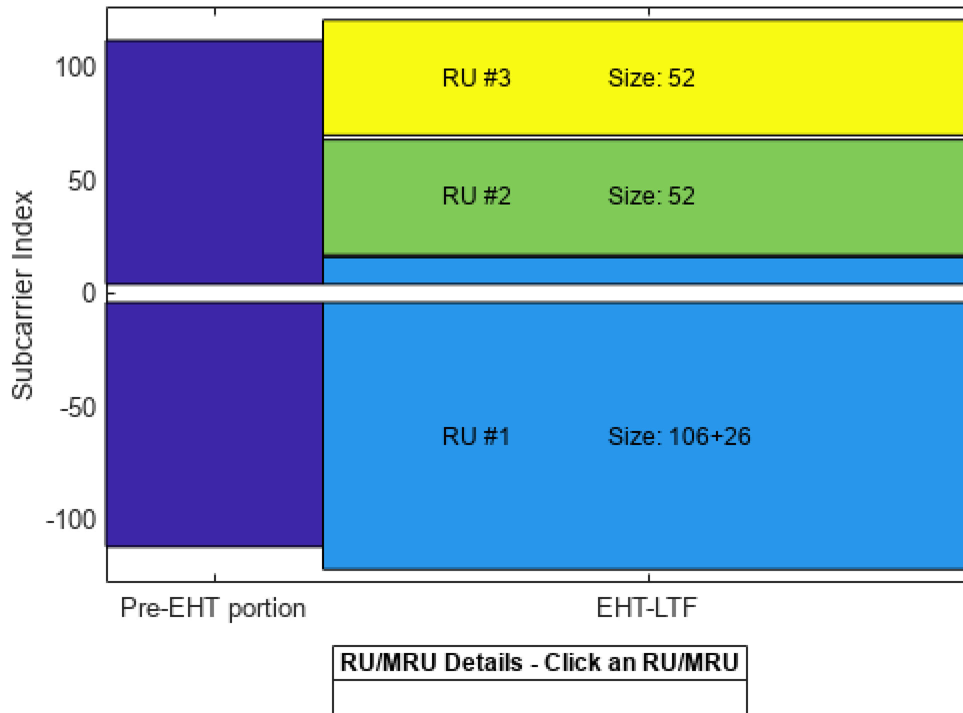
Small MRU Packet Generation

Configure transmission parameters for an EHT MU packet format in a OFDMA configuration with an EHT MU configuration object. Create an OFDMA configuration for a 20 MHz EHT MU packet with two 52-tone RUs, one 106+26-tone MRU, and one user per RU, as defined by allocation index 47.

```
cfgOFDMA = wlanEHTMUConfig(47);
```

Visualize the RU allocation with the `showAllocation` object function.

```
showAllocation(cfgOFDMA)
```



The `ruInfo` object function provides details of the RUs in the configuration. You can see three RUs with a user in each RU.

```
allocInfo = ruInfo(cfgOFDMA);
disp('Allocation info:')
```

```
Allocation info:
```

```
disp(allocInfo)
```

```
                   NumUsers: 3
                     NumRUs: 3
                  RUIndices: {[1 5]  [3]  [4]}
                    RUSizes: {[106 26]  [52]  [52]}
              NumUsersPerRU: [1 1 1]
    NumSpaceTimeStreamsPerRU: [1 1 1]
       PowerBoostFactorPerRU: [1 1 1]
                  RUNumbers: [1 2 3]
```

Set common transmission parameters for all users.

```
numTx = 3;
cfgOFDMA.NumTransmitAntennas = numTx;
cfgOFDMA.EHTLTFType = 2;
cfgOFDMA.GuardInterval = 1.6
```

```
cfgOFDMA = 
  wlanEHTMUConfig with properties:
```

```
                    RU: {[1x1 wlanEHTRU]  [1x1 wlanEHTRU]  [1x1 wlanEHTRU]}
                  User: {[1x1 wlanEHTUser]  [1x1 wlanEHTUser]  [1x1 wlanEHTUser]}
   NumTransmitAntennas: 3
         GuardInterval: 1.6000
            EHTLTFType: 2
  NumExtraEHTLTFSymbols: 0
             EHTSIGMCS: 0
      UplinkIndication: 0
              BSSColor: 0
          SpatialReuse: 0
          TXOPDuration: []

   Read-only properties:
      ChannelBandwidth: 'CBW20'
       AllocationIndex: 47
```

Configure RU 1 and user 1.

```
cfgOFDMA.RU{1}.SpatialMapping = 'Direct';
cfgOFDMA.User{1}.APEPLength = 1e3;
cfgOFDMA.User{1}.MCS = 12;
cfgOFDMA.User{1}.NumSpaceTimeStreams = numTx;
cfgOFDMA.User{1}.ChannelCoding = 'LDPC';
```

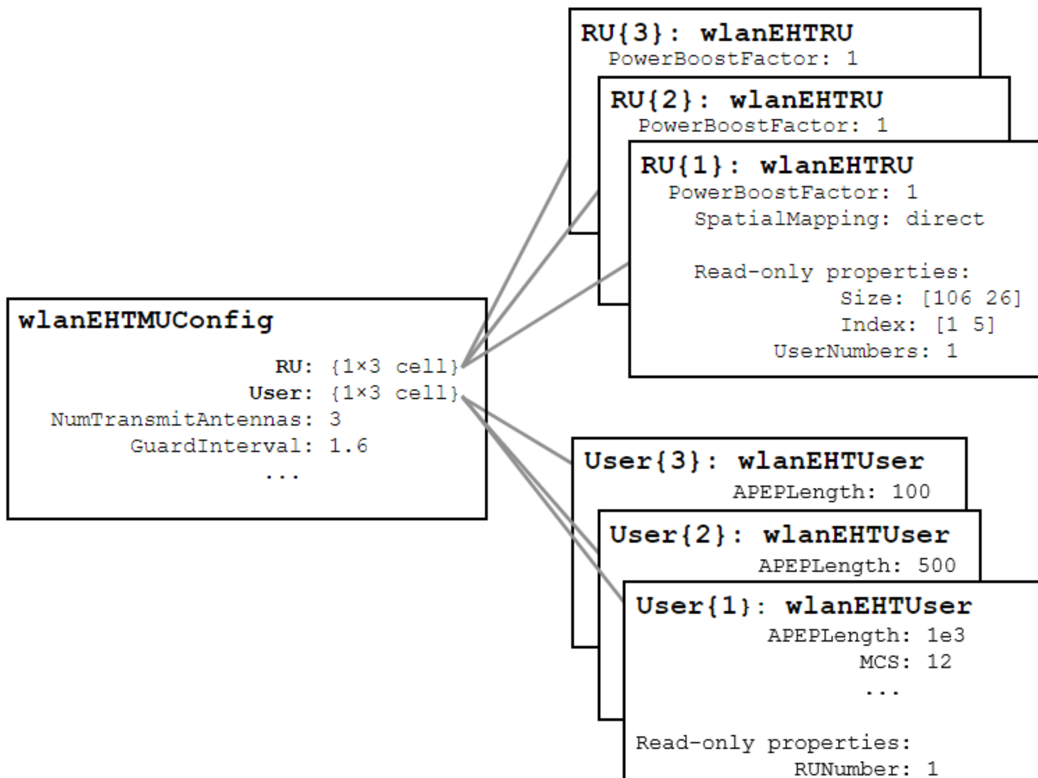
Configure RU 2 and user 2.

```
cfgOFDMA.RU{2}.SpatialMapping = 'Fourier';
cfgOFDMA.User{2}.APEPLength = 500;
cfgOFDMA.User{2}.MCS = 5;
cfgOFDMA.User{2}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{2}.ChannelCoding = 'LDPC';
```

Configure RU 3 and user 3.

```
cfgOFDMA.RU{3}.SpatialMapping = 'Fourier';
cfgOFDMA.User{3}.APEPLength = 100;
cfgOFDMA.User{3}.MCS = 4;
cfgOFDMA.User{3}.NumSpaceTimeStreams = 1;
cfgOFDMA.User{3}.ChannelCoding = 'BCC';
```

Allocation index 47 specifies three RUs and three users, therefore `cfgOFDMA.RU` and `cfgOFDMA.User` are both cell arrays with three elements. This figure shows the object hierarchy.



The `Size` and `Index` properties of `cfgOFDMA.RU{1}` show the size and index of the RUs making up the 106+26-tone MRU.

```
disp('First RU configuration:')
```

```
First RU configuration:
```

```
disp(cfgOFDMA.RU{1})
```

```
  wlanEHTRU with properties:

    PowerBoostFactor: 1
      SpatialMapping: direct

   Read-only properties:
                Size: [106 26]
               Index: [1 5]
         UserNumbers: 1
```

Configure the transmission properties of users by modifying individual user object properties. The read-only `RUNumber` property indicates which RU is used to transmit this user.

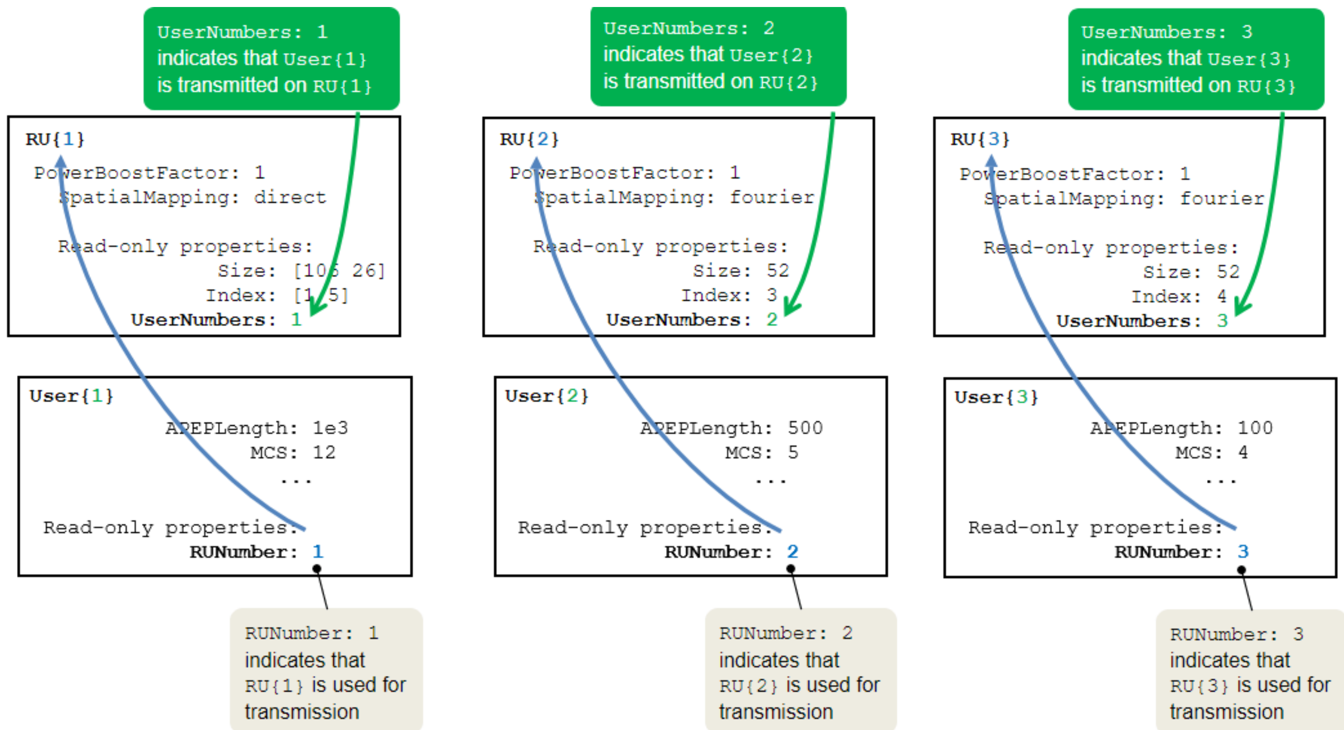
```
disp('First user configuration:')
```

```
First user configuration:
```

```
disp(cfgOFDMA.User{1})
```

```
  wlanEHTUser with properties:

                 APEPLength: 1000
                        MCS: 12
       NumSpaceTimeStreams: 3
             ChannelCoding: ldpc
                     STAID: 0
      NominalPacketPadding: 0
      PostFECPaddingSource: mt19937arwithseed
        PostFECPaddingSeed: 1

   Read-only properties:
                  RUNumber: 1
```

The `UserNumbers` property of an `RU` object indicates which users (elements of the `cfgOFDMA.User` cell array) are transmitted on that RU. Similarly, the `RUNumber` property of each `User` object, indicates which RU (element of the `cfgMUMIMO.RU` cell array) is used to transmit the user. For more information see the “802.11ax Waveform Generation” on page 1-64 example.



To generate an EHT MU waveform, create a random PSDU for each user. Store the PSDU for each user in a cell array as the PSDU lengths differ. The `psduLength` object function returns a vector with the required PSDU per user given the configuration.

```
psduLen = psduLength(cfgOFDMA);
psdu = cell(1,allocInfo.NumUsers);
for i=1:allocInfo.NumUsers
    psdu{i} = randi([0 1],psduLen(i)*8,1);
end
```

Generate the waveform.

```
tx = wlanWaveformGenerator(psdu,cfgOFDMA);
```

Large MRU Packet Generation

An EHT MU transmission with a channel bandwidth greater than 20 MHz uses two content channels to signal the common and user configuration information. These content channels are duplicated over each 40 MHz subchannel within an 80 MHz segment. When an OFDMA system contains an RU with size greater than 242, the users assigned to the RU can be signaled on either of the two EHT SIG content channels as defined in section 36.3.12.8 of [1 on page 1-47].

As an example, consider this 160 MHz configuration, which serves ten users with three RUs including a 996-[ ]-484 MRU:

- A large 996+484-tone MRU (RU #1) with eight users (user #1-8)
- A 242-tone RU (RU #2) with one user (user #9)
- A 242-tone RU (RU #3) with one user (user #10)

Configure a 160 MHz OFDMA transmission with eight allocation indices, one for each 20 MHz subchannel. To configure the example scenario, use these allocation indices.

`[X1 X2 X3 X4 64 64 X5 X6]`

- `X1 to X6` configure the 996+484-tone MRU, with users #1-8.
- Allocation index 64 configures a 242-tone RU with one user.

The selection of `X1` to `X6` configures the appropriate number of users through an allocation index in the 20 MHz subchannel and determines which EHT-SIG content channel signals the users on the 996+484-tone RU. A 996+484-tone RU spans six 20 MHz subchannels. The two 242-tone RUs each indicated by allocation index 64 span two 20 MHz subchannels within MRU 996-[ ]-484. The 996+484-tone MRU requires six allocation indices indicated by `X1` to `X6`. The ten users can be signaled on two EHT-SIG content channels in different combinations. Table 1 shows seven of the numerous possible combinations.

	Lower 80 MHz segment		Upper 80 MHz segment	
Combination	Content Channel 1	Content Channel 2	Content Channel 1	Content Channel 2
A	Users 1-8	No users	User 9	User 10
B	No users	Users 1-8	User 9	User 10
C	Users 1, 2, 5, 6	Users 3, 4, 7, 8	User 9	User 10
D	Users 1, 2, 3	Users 4, 5, 6	Users 9, 7	Users 10, 8
E	Users 1, 2, 3	Users 4, 5, 6	Users 9, 7, 8	User 10
F	Users 1, 2, 3	Users 4, 5, 6	User 9	Users 10, 7, 8
G	No users	No users	Users 9, 1, 2, 3, 4	Users 10, 5, 6, 7, 8

Table 1

An allocation index within the range 144-151 specifies 1-8 users on a 996+484-tone MRU. To signal no users on a content channel, use allocation index 29 or 30 when the 20 MHz subchannel

corresponding to the index overlaps with a 448-tone or 996-tone RU, respectively. Therefore, the combinations in Table 1 require eight allocation indices as shown in Table 2. Other than allocation index 64, the remaining six allocation indices in each row of Table 2 are X1 to X6. Table 2 shows the number of users on each content channel in the two 80 MHz segments for various combinations of RU allocation index.

Combination	RU Allocation Index	Segment 1		Segment 2	
		Content Channel 1	Content Channel 2	Content Channel 1	Content Channel 2
A	[151 30 30 30 64 64 29 29]	8	0	1	1
B	[30 151 30 30 64 64 29 29]	0	8	1	1
C	[145 145 145 145 64 64 29 29]	4	4	1	1
D	[146 146 30 30 64 64 144 144]	3	3	2	2
E	[146 146 30 30 64 64 145 29]	3	3	3	1
F	[146 146 30 30 64 64 29 145]	3	3	1	3
G	[30 30 30 30 64 64 147 147]	0	0	5	5

Table 2

To configure 'Combination C', use the allocation indices [145 145 145 145 64 64 29 29].

Configure the transmission parameters for an EHT MU packet format for the OFDMA configuration with an EHT MU configuration object.

```
cfgLMRU = wlanEHTMUConfig([145 145 145 145 64 64 29 29]); % 1 MRU (996+484) with 8 users and two
cfgLMRU.NumTransmitAntennas = 8;
```

Set transmission parameters of the users.

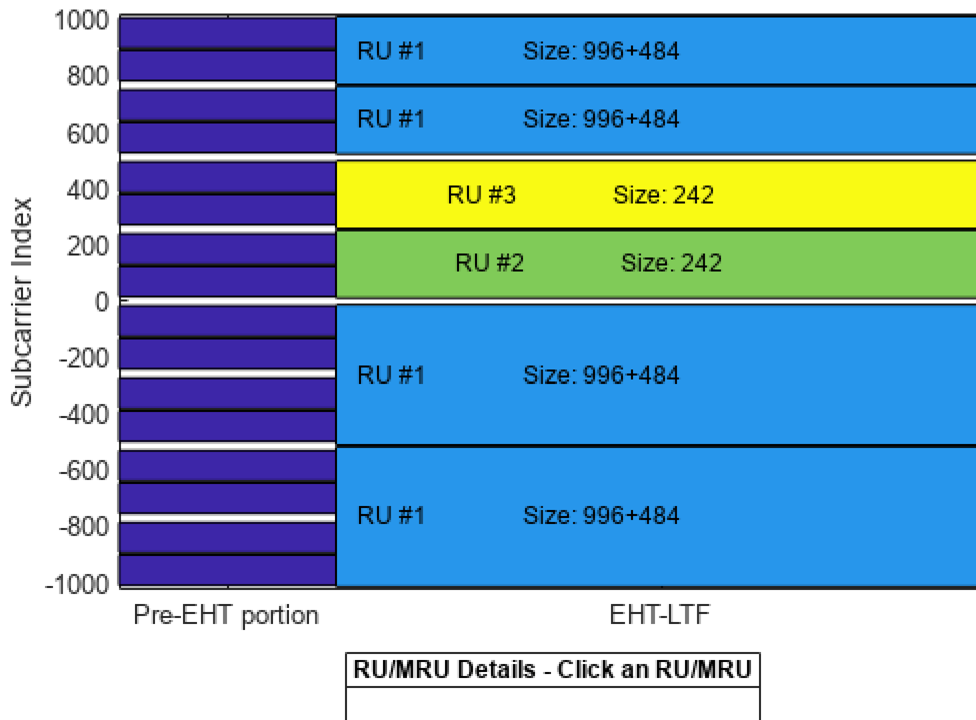
```
userSTS = [1 1 1 1 1 1 1 1 8 8]; % Number of space time streams per user
allocInfo = ruInfo(cfgLMRU); % Get details of the RU/MRU in the configuration
for u=1:allocInfo.NumUsers
    cfgLMRU.User{u}.NumSpaceTimeStreams = userSTS(u);
    cfgLMRU.User{u}.MCS = 12;
    cfgLMRU.User{u}.APEPLength = randi([100 500],1,1);
end
```

Create packet with a repeated bit sequence as the PSDU.

```
tx = wlanWaveformGenerator([1 0 1 0],cfgLMRU);
```

Visualize the RU allocation.

```
showAllocation(cfgLMRU)
```



Preamble Puncturing

Create a configuration for an OFDMA 160 MHz transmission with punctured subchannels. Use the 20 MHz subchannel allocation index 26 to puncture a 20 MHz subchannel. Puncture the second 20 MHz subchannel in the lower 80 MHz segment. Puncture the second and third 20 MHz subchannels in the upper 80 MHz segment.

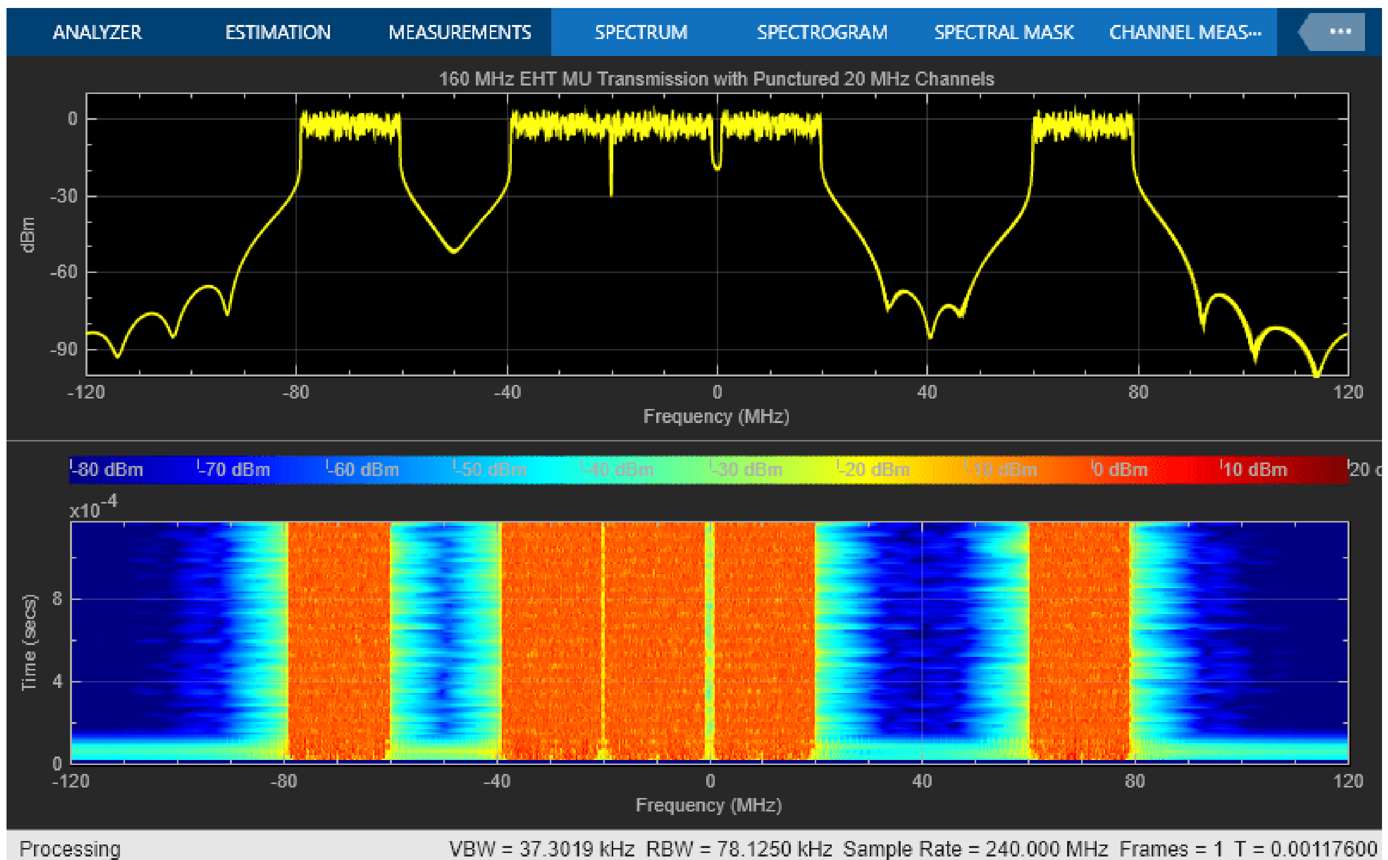
```
cfgPuncture = wlanEHTMUConfig([64 26 64 64 64 26 26 64]);
for u=1:numel(cfgPuncture.User)
    cfgPuncture.User{u}.APEPLength = 1000;
end
```

Generate on oversampled waveform using a larger IFFT size than required for the nominal baseband rate.

```
osf = 1.5;
txPuncture = wlanWaveformGenerator([1 0 1 0],cfgPuncture,'OversamplingFactor',osf);
```

View the punctured 20 MHz subchannels in the generated waveform by using the spectrum analyzer.

```
sa = plotSpectrumAndSpectrogram(txPuncture,cfgPuncture,osf,'160 MHz EHT MU Transmission with Punc
```



Unassigned RUs

The draft standard allows for RUs to be unassigned (no data transmitted) without the preamble portion of the corresponding 20 MHz being punctured. The standard specifies two methods for creating unassigned RUs: using an unassigned 242-tone allocation index, or signaling a user with STA-ID 2046.

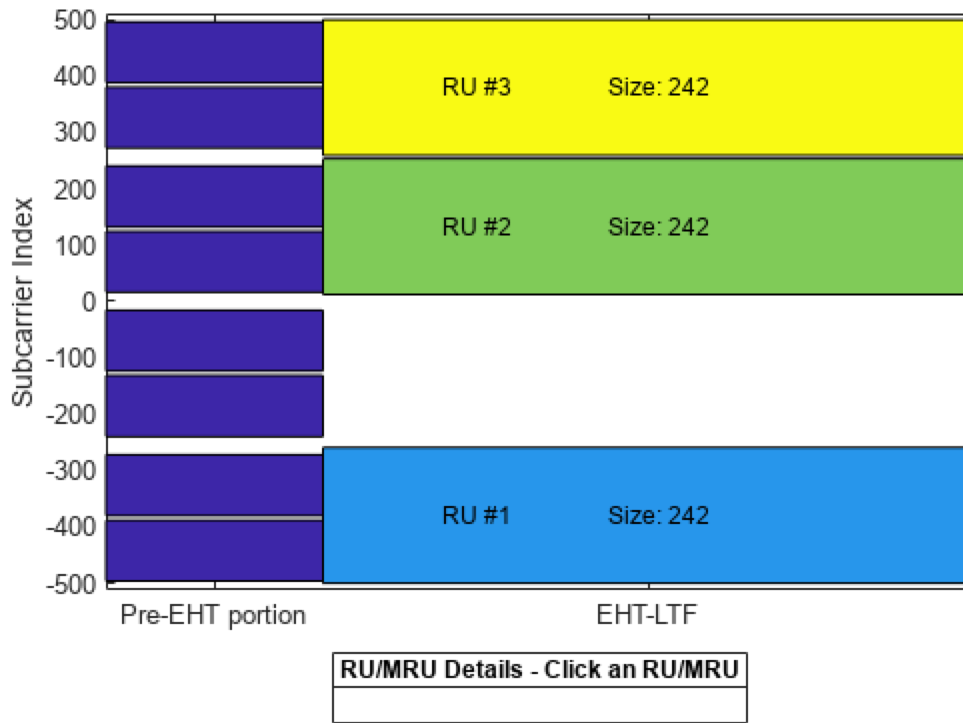
Unassigned 242-tone Allocation Index

Configure a 160 MHz transmission with no data transmission in a 20 MHz subchannel without preamble puncturing by using allocation index 27.

```
cfgUnassigned = wlanEHTMUConfig([64 27 64 64]);
for u=1:numel(cfgUnassigned.User)
    cfgUnassigned.User{u}.APEPLength = 1000;
end
```

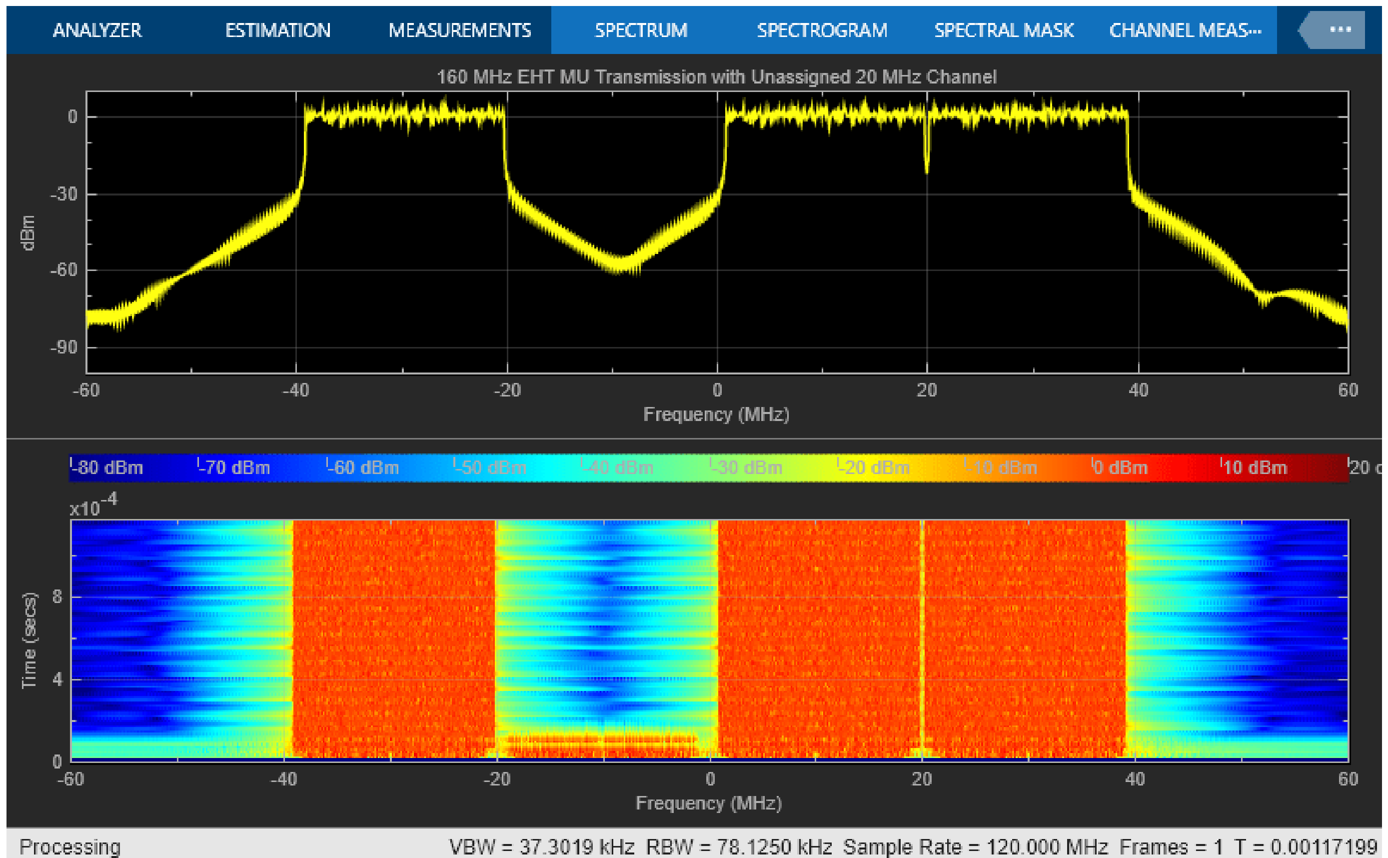
Visualize the RU allocation, which shows that the preamble is not punctured.

```
showAllocation(cfgUnassigned);
```



Generate an oversampled waveform and view the spectrogram. The preamble is visible in all 20 MHz subchannels at the start of the waveform.

```
osf = 1.5;
tx = wlanWaveformGenerator([1 0 0 1],cfgUnassigned,'OversamplingFactor',osf);
sa = plotSpectrumAndSpectrogram(tx,cfgUnassigned,osf,'160 MHz EHT MU Transmission with Unassigned
```



Signal User with STA-ID 2046

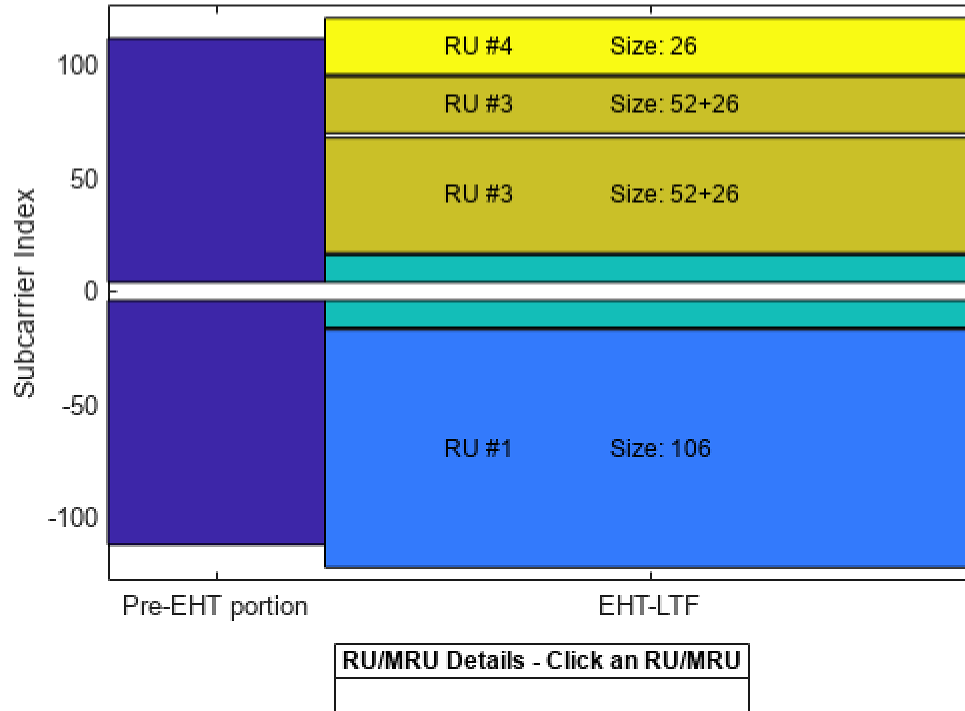
The draft standard states an RU is unassigned (no data transmitted) if the STA-ID of the EHT-SIG User field is 2046.

Create a 20 MHz OFDMA configuration with four RUs with a user in each RU using allocation index 52

```
cfgUnassigned = wlanEHTMUConfig(52);
```

Visualize the RU allocation.

```
showAllocation(cfgUnassigned)
```

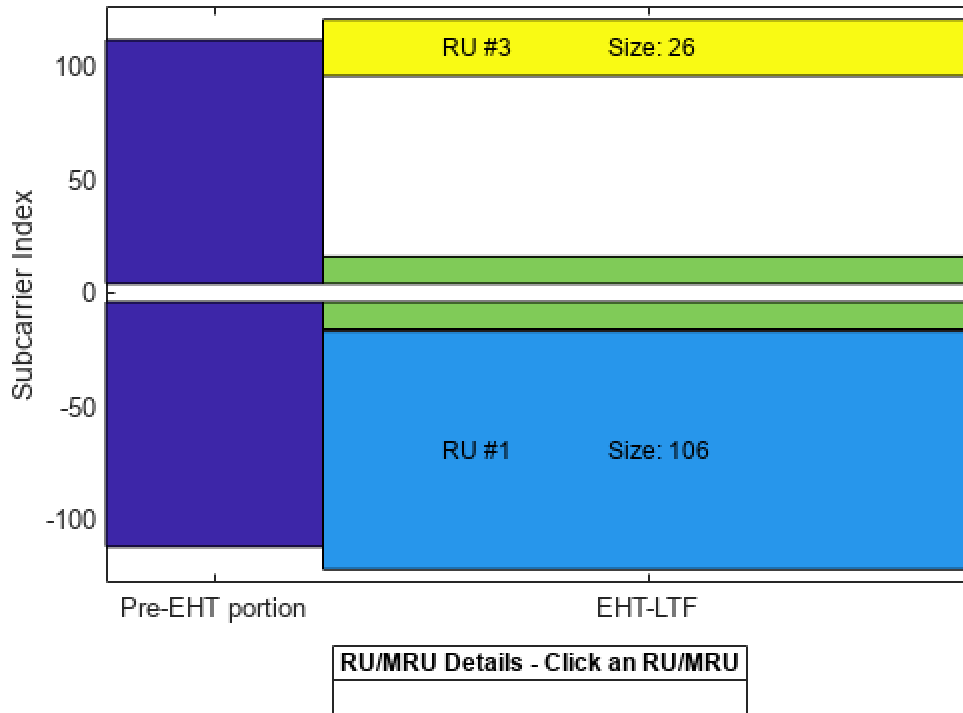


Set the `STAID` property of the third user to 2046 so the 52+26-tone MRU is unassigned.

```
cfgUnassigned.User{3}.STAID = 2046; % No data transmitted to 3rd user on MRU 52+26
```

Visualize the RU allocation, which shows that the third RU is now unassigned but the preamble is not punctured.

```
showAllocation(cfgUnassigned)
```



View details of the active RUs in the configuration by using the `ruInfo` object function. This shows three users and three RUs as one user and RU is unassigned.

```
allocInfo = ruInfo(cfgUnassigned);
disp('Allocation info:')
```

```
Allocation info:
```

```
disp(allocInfo)
```

```
                  NumUsers: 4
                    NumRUs: 3
                 RUIndices: {[1]  [5]  [9]}
                   RUSizes: {[106]  [26]  [26]}
             NumUsersPerRU: [1 1 1]
    NumSpaceTimeStreamsPerRU: [1 1 1]
       PowerBoostFactorPerRU: [1 1 1]
                 RUNumbers: [1 2 4]
```

EHT Trigger-Based (EHT TB) Packet Format

The EHT trigger-based (TB) format allows for OFDMA or MU-MIMO transmission in the uplink. Each station (STA) transmits a TB packet simultaneously, when triggered by the access point (AP). An AP controls the TB transmission and provides the parameters to all STAs participating in the TB transmission through a trigger frame. This example shows a TB transmission in response to a trigger frame for four users in an OFDMA/MU-MIMO system. All four STAs will transmit simultaneously to an AP.

The 80 MHz allocation range 104 to 111 corresponds to a large size MRU in a OFDMA/MU-MIMO configuration. As an example, the allocation 104 and 50 corresponds to three RU/MRUs, where:

- A 242-[]-484-tone large size MRU (RU #1) has two users (user #1-2)
- A 106-tone RU (RU #2) has one user (user #3)
- A 106+26-tone small size MRU (RU #3) has one user (user #4)

Use a `wlanEHTMUConfig` object to obtain the allocation information.

```
% Generate an OFDMA/MU-MIMO allocation
cfgMU = wlanEHTMUConfig([104 50 104 29]);
allocationInfo = ruInfo(cfgMU);
```

In a TB transmission several parameters are the same for all users in the transmission. Some of these are specified below:

```
% These parameters are the same for all users in a TB transmission
channelBandwidth = cfgMU.ChannelBandwidth; % Bandwidth of TB waveform
lsigLength = 286;        % L-SIG length
preFECPaddingFactor = 1; % Pre-FEC padding factor
numEHTLTFSymbols = 2;    % Number of EHT-LTF symbols
ldpcExtraSymbol = true;  % LDPC extra symbol
```

Use a `wlanEHTTBConfig` object to configure a TB transmission for a single user within the system. In this example, an array of four objects is created to describe the transmission of four users.

```
% Create a trigger configuration for each user
numUsers = allocationInfo.NumUsers;
cfgTriggerUser = repmat(wlanEHTTBConfig,1,numUsers);
```

The non-default system-wide properties are set for each user.

```
for userIdx = 1:numUsers
   cfgTriggerUser(userIdx).ChannelBandwidth = channelBandwidth;
   cfgTriggerUser(userIdx).LSIGLength = lsigLength;
   cfgTriggerUser(userIdx).PreFECPaddingFactor = preFECPaddingFactor;
   cfgTriggerUser(userIdx).NumEHTLTFSymbols = numEHTLTFSymbols;
end
```

Next set the per-user properties. When multiple users are transmitting in the same RU, in an MU-MIMO configuration, each user must transmit on different space-time stream indices. Set properties `StartingSpaceTimeStream` and `NumSpaceTimeStreamSteams` for each user. In this example user 1 and 2 are in an MU-MIMO configuration, therefore `StartingSpaceTimeStream` for user two is set to `2`, as user one transmit on a single space-time stream with `StartingSpaceTimeStream = 1`.

```
% These parameters are for the first user - Large MRU, MU-MIMO user 1
cfgTriggerUser(1).RUSize = allocationInfo.RUSizes{1};
cfgTriggerUser(1).RUIndex = allocationInfo.RUIndices{1};
cfgTriggerUser(1).MCS = 4;                          % Modulation and coding scheme
cfgTriggerUser(1).NumTransmitAntennas = 2;          % Number of transmit antennas
cfgTriggerUser(1).NumSpaceTimeStreams = 1;          % Number of space-time streams
cfgTriggerUser(1).StartingSpaceTimeStream = 1;      % The starting index of the space-time strea
cfgTriggerUser(1).ChannelCoding = 'ldpc';           % Channel coding
cfgTriggerUser(1).LDPCExtraSymbol = ldpcExtraSymbol; % LDPC extra symbols
cfgTriggerUser(1).SpatialMapping = 'fourier';       % Spatial mapping

% These parameters are for the second user - Large MRU, MU-MIMO user 2
```

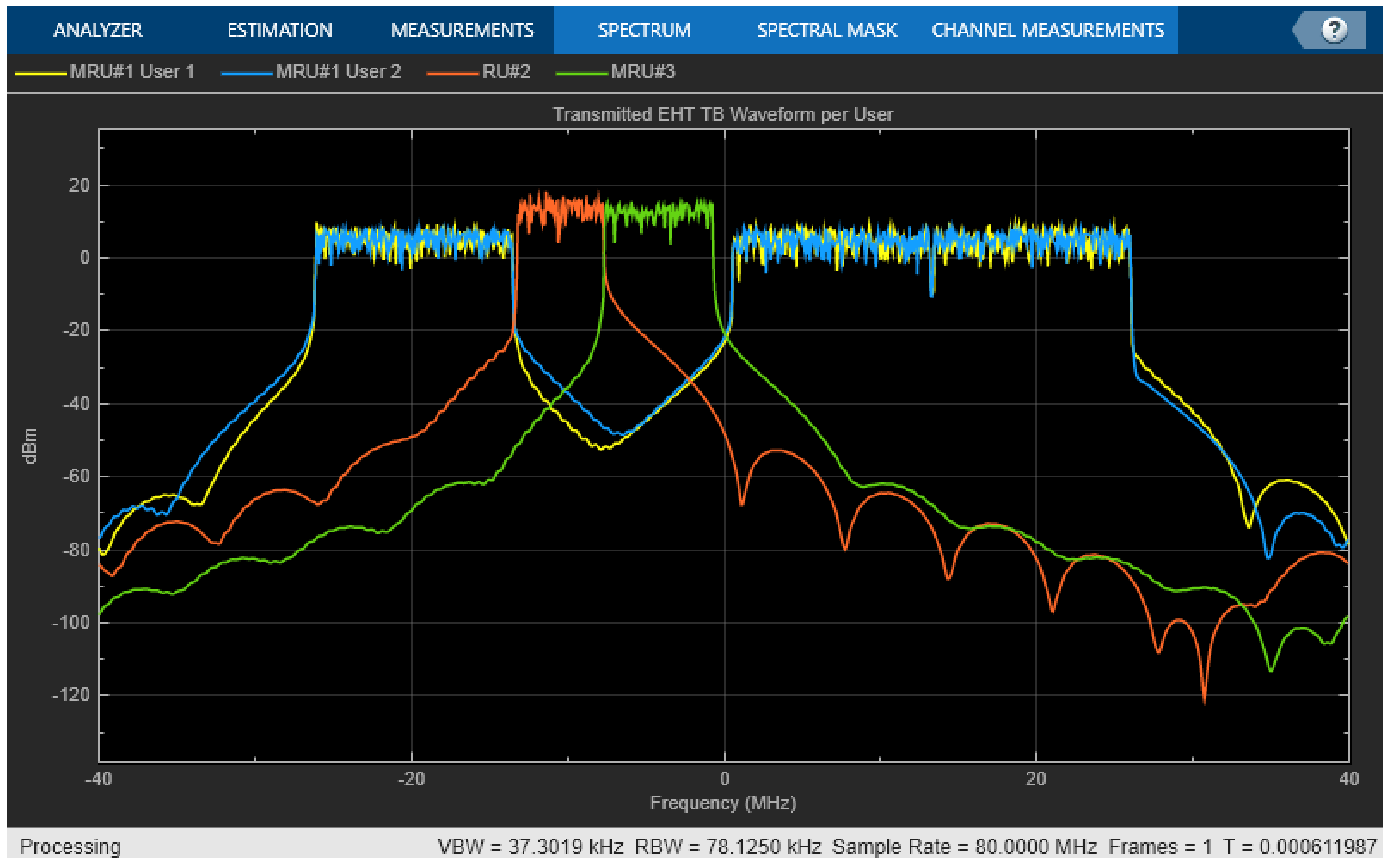
```
cfgTriggerUser(2).RUSize = allocationInfo.RUSizes{1};
cfgTriggerUser(2).RUIndex = allocationInfo.RUIndices{1};
cfgTriggerUser(2).MCS = 2;                          % Modulation and coding scheme
cfgTriggerUser(2).NumTransmitAntennas = 2;          % Number of transmit antennas
cfgTriggerUser(2).NumSpaceTimeStreams = 1;          % Number of space-time streams
cfgTriggerUser(2).StartingSpaceTimeStream = 2;      % The starting index of the space-time strea
cfgTriggerUser(2).ChannelCoding = 'ldpc';           % Channel coding
cfgTriggerUser(2).LDPCExtraSymbol = ldpcExtraSymbol; % LDPC extra symbols
cfgTriggerUser(2).SpatialMapping = 'fourier';       % Spatial mapping

% These parameters are for the third user - RU#2
cfgTriggerUser(3).RUSize = allocationInfo.RUSizes{2};
cfgTriggerUser(3).RUIndex = allocationInfo.RUIndices{2};
cfgTriggerUser(3).MCS = 3;                   % Modulation and coding scheme
cfgTriggerUser(3).NumTransmitAntennas = 2;   % Number of transmit antennas
cfgTriggerUser(3).NumSpaceTimeStreams = 1;   % Number of space-time streams
cfgTriggerUser(3).ChannelCoding = 'bcc';     % Channel coding
cfgTriggerUser(3).SpatialMapping = 'fourier'; % Spatial mapping

% These parameters are for the fourth user - Small MRU
cfgTriggerUser(4).RUSize = allocationInfo.RUSizes{3};
cfgTriggerUser(4).RUIndex = allocationInfo.RUIndices{3};
cfgTriggerUser(4).MCS = 1;                          % Modulation and coding scheme
cfgTriggerUser(4).NumTransmitAntennas = 2;          % Number of transmit antennas
cfgTriggerUser(4).NumSpaceTimeStreams = 1;          % Number of space-time streams
cfgTriggerUser(4).ChannelCoding = 'ldpc';           % Channel coding
cfgTriggerUser(4).LDPCExtraSymbol = ldpcExtraSymbol; % LDPC extra symbols
cfgTriggerUser(4).SpatialMapping = 'fourier';       % Spatial mapping
```

A packet containing random data is now transmitted by each user with `wlanWaveformGenerator`. The waveform transmitted by each user is stored for analysis.

```
osf = 1.5; % Over sampling factor
trigInd = wlanFieldIndices(cfgTriggerUser(1),'OversamplingFactor',osf); % Get the indices of each
txTrigStore = zeros(trigInd.EHTData(2),numUsers);
for userIdx = 1:numUsers
    % Generate waveform for a user
    cfgTrigger =cfgTriggerUser(userIdx);
    txPSDU = randi([0 1],psduLength(cfgTrigger)*8,1);
    txTrig = wlanWaveformGenerator(txPSDU,cfgTrigger,'OversamplingFactor',osf);

    % Store the transmitted STA waveform for analysis
    txTrigStore(:,userIdx) = sum(txTrig,2);
end
```

The spectrum of the transmitted oversampled waveform from each STA shows the different portions of the spectrum used, and the overlap in the MU-MIMO RU.

```
fs = wlanSampleRate(cfgTriggerUser(1));
ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfgTriggerUser(1));
rbw = fs/ofdmInfo.FFTLength; % Resolution bandwidth
spectrumScope = spectrumAnalyzer(SampleRate=fs,...
            RBWSource='property',RBW=rbw,...
            AveragingMethod='exponential',ForgettingFactor=0.25,...
            ChannelNames={'MRU#1 User 1','MRU#1 User 2','RU#2','MRU#3'},...
            ShowLegend=true,Title='Transmitted EHT TB Waveform per User');
spectrumScope(txTrigStore);
```




```
function spectrumScope = plotSpectrumAndSpectrogram(tx,cfg,osf,titleStr)
    % Plot the specrum and spectrogram for a given waveform and
    % configuration
    fs = wlanSampleRate(cfg.ChannelBandwidth,'OversamplingFactor',osf); % Get baseband sample ra
    ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfg,1,'OversamplingFactor',osf);
    fftsize = ofdmInfo.FFTLength; % Use the data field fft size
    rbw = fs/fftsize; % Resolution bandwidth
    spectrumScope = spectrumAnalyzer(SampleRate=fs,...
    RBWSource='property',RBW=rbw,...
    AveragingMethod='exponential',ForgettingFactor=0.25,...
    ReducePlotRate=false,YLimits=[-100,10],...
    Title=titleStr);
    spectrumScope.ViewType = 'spectrum-and-spectrogram';
    spectrumScope.TimeSpanSource = 'property';
    spectrumScope.TimeSpan = length(tx)/fs;
    spectrumScope(tx)
end
```

Appendix A

The RU allocation table for allocations <= 20 MHz is shown below, with annotated descriptions.

Allocation Index	20 MHz Subchannel Resource Unit (RU) Assignment								
0	26	26	26	26	26	26	26	26	26
1	26	26	26	26	26	26	26	52	
2	26	26	26	26	26	52		26	26
3	26	26	26	26	26	52		52	
4	26	26	52		26	26	26	26	26
5	26	26	52		26	26	26	52	
6	26	26	52		26	52		26	26
7	26	26	52		26	52		52	
8	52		26	26	26	26	26	26	26
9	52		26	26	26	26	26	52	
10	52		26	26	26	52		26	26
11	52		26	26	26	52		52	
12	52		52		26	26	26	26	26
13	52		52		26	26	26	52	
14	52		52		26	52		26	26
15	52		52		26	52			
16	26	26	26	26	26	106			
17	26	26	52		26	106			
18	52		26	26	26	106			
19	52		52		26	106			
20	106				26	26	26	26	26
21	106				26	26	26	52	
22	106				26	52		26	26
23	106				26	52		52	
24	52		52		-	52		52	
25	106				26	106			
26	Punctured 242-tone RU								
27	Unassigned 242-tone RU								
28	242-tone RU with no users signaled on the corresponding EHT-SIG content channel								
29	484-tone RU with no users signaled on the corresponding EHT-SIG content channel								
30	996-tone RU with no users signaled on the corresponding EHT-SIG content channel								
31	Validate								

26 tone RU assigned to 1 user as part of a 20 MHz subchannel assignment of 9 26-tone RUs

No users assigned to this RU; no data field transmitted on these subcarriers (empty 26-tone RU with no user assigned)

If selected, this 20 MHz subchannel is unused; the subchannel is punctured

If selected no user data is transmitted in the 242-tone RU, but the corresponding 20 MHz preamble is transmitted

Signifies a 242-tone RU with zeros users signaled on the corresponding EHT-SIG content channel

Must be used with other allocation indices. Signifies a 484-tone or 996-tone RU with zero users signaled on the corresponding EHT-SIG content channel

Reserved entry

RU assigned to 1 user

RU allocation and EHT-SIG user signaling for allocations >= 20 MHz, with annotated descriptions.

Allocation Index	20 MHz Subchannel Resource Unit (RU) Assignment								
32	26	26	26	26	26	52+26			26
33	26	26	52		26	52+26			26
34	52		26	26	26	52+26			26
35	52		52		26	52+26			26
36	26	52+26			26	26	26	26	26
37	26	52+26			26	26	26	52	
38	26	52+26			26	52		26	26
39	26	52+26			26	52		52	
40	26	26	26	26	106+26				
41	26	26	52		106+26				
42	52		26	26	106+26				
43	52		52		106+26				
44	106+26					26	26	26	26
45	106+26					26	26	52	
46	106+26					52		26	26
47	106+26					52		52	
48	106+26					106			
49	106+26					52+26			26
50	106				106+26				
51	26	52+26			106+26				
52	106				26	52+26			26
53	26	52+26			26	106			
54	26	52+26			26	52+26			26
55	52		52+26			52		52	
56-63	Validate								

52+26-tone small MRU is assigned to 1 user

Reserved entries

RU assigned to 1 user

RU allocation and EHT-SIG user signaling for 242-, 484-, 996-, and 2x996-tone RUs.

Allocation Index	RU Allocation & Number of Users on the Corresponding EHT-SIG Content Channel for RU Size >= 242
64-71 (63 + N)	20 MHz (N users), or 242-tone RU with N users signaled in the corresponding EHT-SIG content channel
72-79 (71 + N)	40 MHz (N users), or 484-tone RU with N users signaled in the corresponding EHT-SIG content channel
80-87 (79 + N)	80 MHz (N users), or 996-tone RU with N users signaled in the corresponding EHT-SIG content channel
88-95 (87 + N)	160 MHz (N users), or 2x996-tone RU with N users signaled in the corresponding EHT-SIG content channel

>=242-tone RU allocation

MRU allocation and EHT-SIG user signaling for allocation >=80 MHz.

Allocation Index	RU Allocation & Number of Users on the Corresponding EHT-SIG Content Channel for RU Size > 484
96-103 (95 + N)	MRU of [ ]-242-484 484+242-tone MRU in 80 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
104-111 (103 + N)	MRU of 242-[ ]-484 484+242-tone MRU in 80 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
112-119 (111 + N)	MRU of 484-[ ]-242 484+242-tone MRU in 80 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
120-127 (119 + N)	MRU of 484-242-[ ] 484+242-tone MRU in 80 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
128-135 (127 + N)	MRU of [ ]-484-996 996+484-tone MRU in 160 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
136-143 (135 + N)	MRU of 484-[ ]-996 996+484-tone MRU in 160 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
144-151 (143 + N)	MRU of 996-[ ]-484 996+484-tone MRU in 160 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
152-159 (151 + N)	MRU of 996-484-[ ] 996+484-tone MRU in 160 MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
160-167 (159 + N)	MRU of [ ]-996-996-996 3x996-tone MRU with N users signaled in the corresponding EHT-SIG content channel
168-175 (167 + N)	MRU of 996-[ ]-996-996 3x996-tone MRU with N users signaled in the corresponding EHT-SIG content channel
176-183 (175 + N)	MRU of 996-996-[ ]-996 3x996-tone MRU with N users signaled in the corresponding EHT-SIG content channel
184-191 (183 + N)	MRU of 996-996-996-[ ] 3x996-tone MRU with N users signaled in the corresponding EHT-SIG content channel
192-199 (191 + N)	MRU of [ ]-484-996-996-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
200-207 (199 + N)	MRU of 484-[ ]-996-996-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
208-215 (207 + N)	MRU of 996-[ ]-484-996-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
216-223 (215 + N)	MRU of 996-484-[ ]-996-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
224-231 (223 + N)	MRU of 996-996-[ ]-484-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
232-239 (231 + N)	MRU of 996-996-484-[ ]-996 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
240-247 (239 + N)	MRU of 996-996-996-[ ]-484 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
248-255 (247 + N)	MRU of 996-996-996-484-[ ] 3x996+484-tone MRU with N users signaled in the corresponding EHT-SIG content channel
256-263 (255 + N)	MRU of [ ]-484-996-996 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
264-271 (263 + N)	MRU of 484-[ ]-996-996 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
272-279 (271 + N)	MRU of 996-[ ]-484-996 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
280-287 (279 + N)	MRU of 996-484-[ ]-996 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
288-295 (287 + N)	MRU of 996-996-[ ]-484 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
296-303 (295 + N)	MRU of 996-996-484-[ ] 2x996+484-tone MRU in 240MHz sub-block with N users signaled in the corresponding EHT-SIG content channel
304-511	Disregard

Appendix B

Non-OFDMA punctured channel indication for a 80 MHz PPDU, with annotated descriptions.

Puncturing subchannel bandwidth	Puncturing pattern	Punctured channel field value
No puncturing	[1 1 1 1]	0
20 MHz	[0 1 1 1]	1
	[1 0 1 1]	2
	[1 1 0 1]	3
	[1 1 1 0]	4

A value of 0 indicates that the corresponding 20 MHz subchannel is punctured. A value of 1 indicates a non-punctured 20 MHz subchannel

Non-OFDMA punctured channel indication for a 160 MHz PPDU, with annotated descriptions.

Puncturing subchannel bandwidth	Puncturing pattern	Punctured channel field value
No puncturing	[1 1 1 1 1 1 1 1]	0
20 MHz	[0 1 1 1 1 1 1 1]	1
	[1 0 1 1 1 1 1 1]	2
	[1 1 0 1 1 1 1 1]	3
	[1 1 1 0 1 1 1 1]	4
	[1 1 1 1 0 1 1 1]	5
	[1 1 1 1 1 0 1 1]	6
	[1 1 1 1 1 1 0 1]	7
	[1 1 1 1 1 1 1 0]	8
40 MHz	[0 0 1 1 1 1 1 1]	9
	[1 1 0 0 1 1 1 1]	10
	[1 1 1 1 0 0 1 1]	11
	[1 1 1 1 1 1 0 0]	12

A value of 0 indicates that the corresponding 20 MHz subchannel is punctured. A value of 1 indicates a non-punctured 20 MHz subchannel

Non-OFDMA punctured channel indication for a 320 MHz PPDU.

Puncturing subchannel bandwidth	Puncturing pattern	Punctured channel field value
No puncturing	[1 1 1 1 1 1 1 1]	0
40 MHz	[0 1 1 1 1 1 1 1]	1
	[1 0 1 1 1 1 1 1]	2
	[1 1 0 1 1 1 1 1]	3
	[1 1 1 0 1 1 1 1]	4
	[1 1 1 1 0 1 1 1]	5
	[1 1 1 1 1 0 1 1]	6
	[1 1 1 1 1 1 0 1]	7
	[1 1 1 1 1 1 1 0]	8
80 MHz	[0 0 1 1 1 1 1 1]	9
	[1 1 0 0 1 1 1 1]	10
	[1 1 1 1 0 0 1 1]	11
	[1 1 1 1 1 1 0 0]	12
40 MHz and 80 MHz	[0 0 0 1 1 1 1 1]	13
	[0 0 1 0 1 1 1 1]	14
	[0 0 1 1 0 1 1 1]	15
	[0 0 1 1 1 0 1 1]	16
	[0 0 1 1 1 1 0 1]	17
	[0 0 1 1 1 1 1 0]	18
	[0 1 1 1 1 1 0 0]	19
	[1 0 1 1 1 1 0 0]	20
	[1 1 0 1 1 1 0 0]	21
	[1 1 1 0 1 1 0 0]	22
	[1 1 1 1 0 1 0 0]	23
	[1 1 1 1 1 0 0 0]	24

A value of 0 indicates that the corresponding 40 MHz subchannel is punctured. A value of 1 indicates a non-punctured 40 MHz subchannel

Related Examples

- "802.11be Transmitter Measurements" on page 3-2. Measure transmitter modulation accuracy, spectral mask, and spectral flatness of an EHT MU waveform.
- "802.11be Packet Error Rate Simulation for an EHT MU Single-User Packet Format" on page 6-37 Measure the packet error rate of an EHT MU single user packet format.
- "802.11be Packet Error Rate Simulation for Uplink Trigger-Based Format" on page 6-17 Measure the packet error rate of an EHT TB transmission.

References

1. IEEE P802.11be™/D2.0 Draft 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 8: Enhancements for extremely high throughput (EHT).
2. IEEE Std 802.11ax(TM)-2021. 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 1: Enhancements for High Efficiency WLAN.

802.11az Waveform Generation

This example shows how to parameterize and generate IEEE® 802.11az™ high-efficiency (HE) ranging null data packet (NDP) waveforms and highlights some of the key features of the standard.

Introduction

The 802.11az standard [1 on page 1-51], commonly referred to as next-generation positioning (NGP), enables a station to identify its position relative to other stations. This standard supports two HE ranging physical layer (PHY) protocol data unit (PPDU) formats:

- HE ranging NDP
- HE trigger-based (TB) ranging NDP

The HE ranging NDP and HE TB ranging NDP are the respective analogues of the HE sounding NDP and HE TB NDP feedback PPDU formats, as defined in the 802.11ax™ standard. For more information on these HE PPDU formats, see [2 on page 1-51].

The HE ranging NDP supports the positioning of one or more users with an optional secure HE long training field (HE-LTF) sequence. The single-user HE ranging waveform contains HE-LTF symbols for a single user, which also support an optional secure HE-LTF sequence. The multi-user HE ranging waveform permits only secure HE-LTF symbols for multiple users. Single-user and multi-user waveforms can contain multiple repetitions of the HE-LTF symbols. This feature can help improve distance estimation accuracy.

Because the 802.11az standard uses the same underlying PHY technologies as the 802.11ax standard, the processing chains are very similar. This example shows how to generate 802.11az HE ranging NDP waveforms with secure and nonsecure HE-LTF sequences.

HE Ranging NDP Without Secure HE-LTF

The HE Ranging NDP contains HE-LTF symbols for a single user and uses the regular HE-LTF sequence defined in [2 on page 1-51]. The number of HE-LTF symbols is the product of the number of HE-LTF repetitions and the number of HE-LTF symbols per repetition. The number of HE-LTF symbols depends on the number of space-time streams as specified in Table 21-13 of [3 on page 1-51]. The construction of HE-LTF symbols in an HE Ranging NDP follows the steps defined in section 27.3.10.10 of [2 on page 1-51] for all repeated HE-LTF symbols in an HE-LTF.

Single-User HE Ranging NDP Generation

Configure a transmission with two antennas, two space-time streams, and two HE-LTF repetitions.

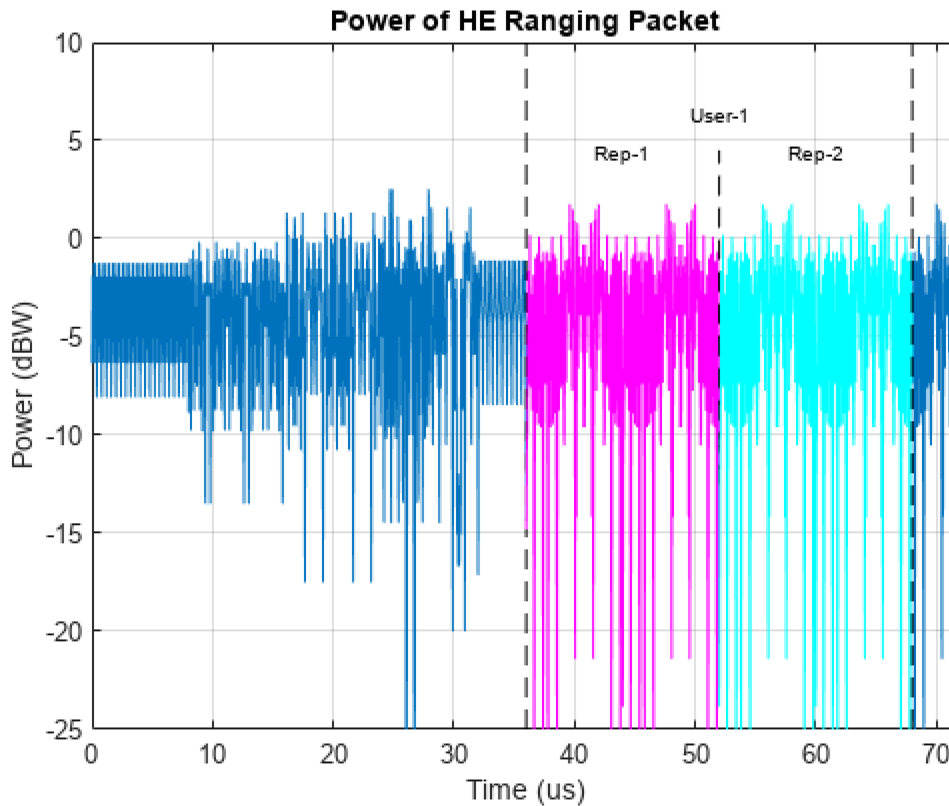
```
cfg = heRangingConfig('NumTransmitAntennas',2);
cfg.User{1}.NumSpaceTimeStreams = 2;
cfg.User{1}.NumHELTFRepetition = 2;
```

Generate the HE ranging NDP waveform for the specified configuration.

```
tx = heRangingWaveformGenerator(cfg);
```

Plot the transmission power on the first antenna.

```
heRangingWavGenPlot(tx,cfg);
```



HE Ranging NDP With Secure HE-LTF

To generate an HE ranging NDP with secure HE-LTF symbols, as defined in [2 on page 1-51], specify either of these transmission parameter combinations.

- A single-user `heRangingConfig` object with its `SecureHELTF` property set to `1` (`true`)
- A multi-user `heRangingConfig` object

The secure HE-LTF comprises a randomized LTF sequence as defined in Section 27.3.17c of [1 on page 1-51]. To specify this sequence for a chosen user, set the `SecureHELTFSequence` property of the corresponding `User` property of the `heRangingConfig` object. If the number of bits in the `SecureHELTFSequence` property is less than the required number of bits for the given user configuration, the object cyclically extends the secure sequence. If the number of bits in `SecureHELTFSequence` is more than the required number of bits for the given user configuration, the object uses only the required number of bits. The object extracts the required bits from the specified hexadecimal sequence. When the transmission contains a secure HE-LTF sequence, the sequence must use a zero-power guard interval for the HE-LTF symbols. The packet extension (PE) starts with a zero-power guard interval.

Single-User HE Ranging NDP with Secure HE-LTF Generation

Configure a transmission with two antennas, two space-time streams, three HE-LTF repetitions, and secure HE-LTF symbols.

```
cfg = heRangingConfig('NumTransmitAntennas',2,'SecureHELTF',true);
cfg.User{1}.NumSpaceTimeStreams = 2;
```

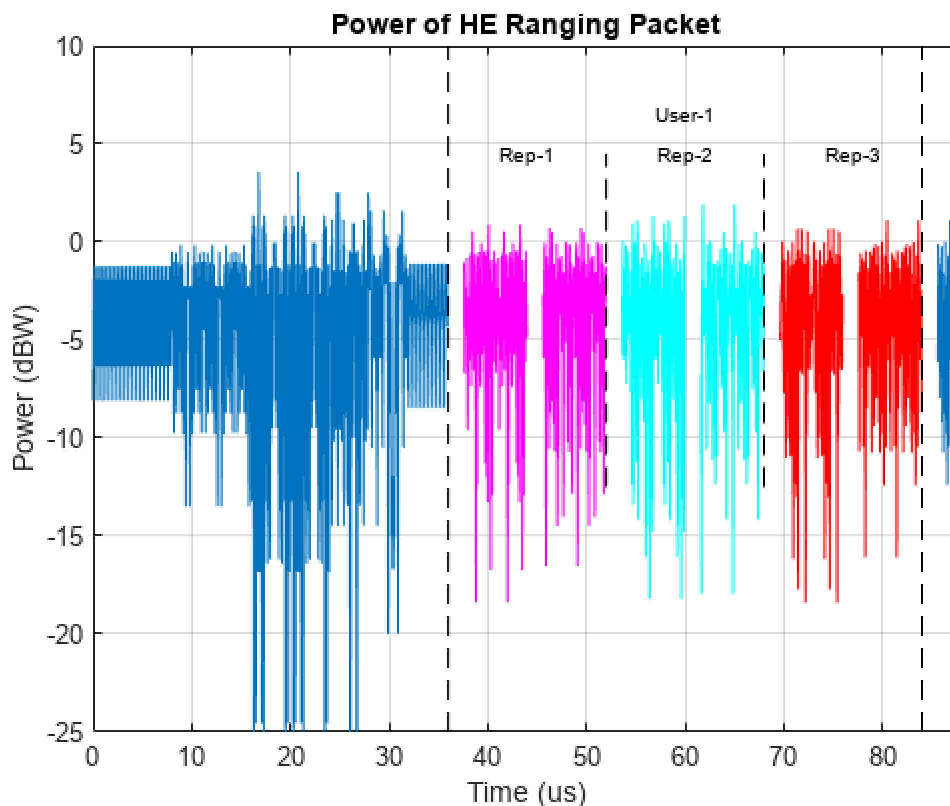
```
cfg.User{1}.NumHELTFRepetition = 3;
cfg.User{1}.SecureHELTFSequence = '12345678ABCDEF1234';
```

Generate the HE ranging NDP waveform for the specified configuration.

```
tx = heRangingWaveformGenerator(cfg);
```

Plot the transmission power on the first antenna.

```
heRangingWavGenPlot(tx,cfg);
```



Multi-User HE Ranging NDP Generation

A multi-user HE ranging NDP waveform contains secure HE-LTF symbols for multiple users. The transmission concatenates HE-LTF symbols for each user up to a maximum of 64 consecutive symbols. This example demonstrates waveform generation with a secure HE-LTF sequence shown for two users. Configure transmission parameters, specifying two users and the number of space-time streams and HE-LTF repetitions for each user.

```
cfg = heRangingConfig(2);
cfg.User{1}.NumSpaceTimeStreams = 1;
cfg.User{1}.NumHELTFRepetition = 2;
cfg.User{2}.NumSpaceTimeStreams = 1;
cfg.User{2}.NumHELTFRepetition = 3;
```

Determine the number of secure HE-LTF bits required to generate the secure HE-LTF symbols for each user by using the `numSecureHELTFBits` object function.

```
numNibbles = numSecureHELTFBits(cfg)/4; % 4 bits per nibble
```

Set the secure HE-LTF sequences for each user.

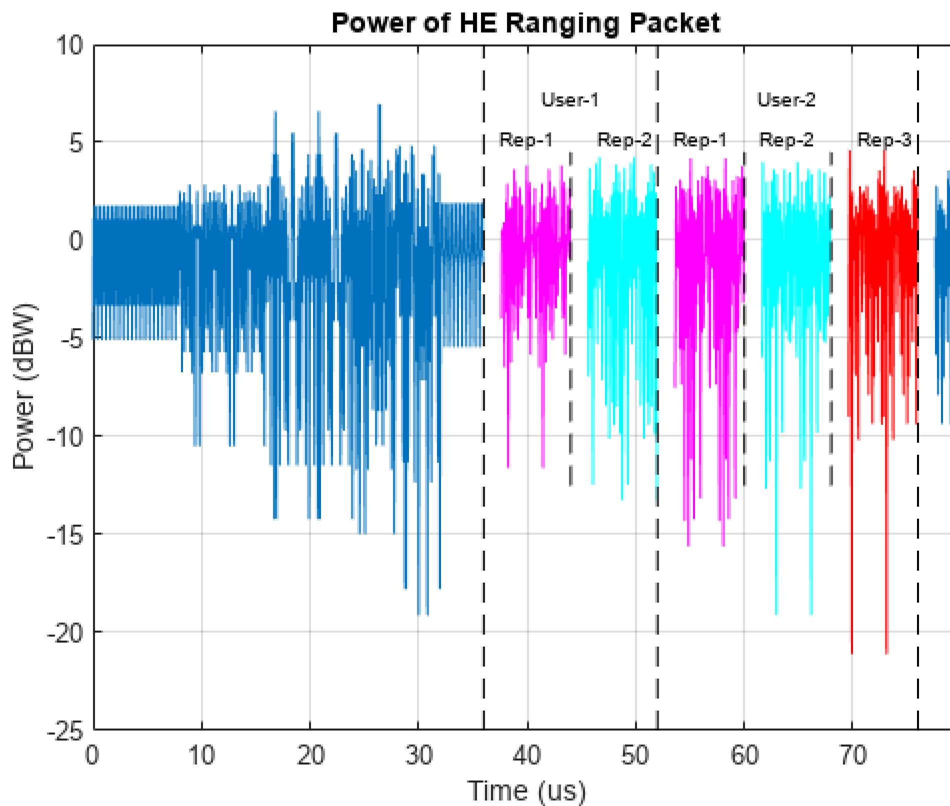
```
secureSeqUser1 = 'a12c67f8b90dc56e78a2b3f1';
cfg.User{1}.SecureHELTFSequence = secureSeqUser1(1:numNibbles(1));
secureSeqUser2 = 'b3a49c5e6c1a2d35ed47c2d915f';
cfg.User{2}.SecureHELTFSequence = secureSeqUser2(1:numNibbles(2));
```

Generate the HE ranging NDP waveform for the specified configuration.

```
tx = heRangingWaveformGenerator(cfg);
```

Plot the transmission power on the first antenna.

```
heRangingWavGenPlot(tx,cfg);
```



References

1 IEEE P802.11az™/D2.0 Draft Standard for Information technology - Telecommunications and information exchange between systems Local and metropolitan area networks - Specific requirements - Amendment 3: Enhancements for positioning.

2 IEEE 802.11ax™-2021 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 6: Enhancements for High Efficiency WLAN.

3 IEEE Std 802.11™-2020 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.

Build VHT PPDU

This example shows how to build VHT PPDUs by using the WLAN waveform generator function or by building each field individually, then concatenating.

You can create a VHT, HT, or non-HT PPDU waveform by generating and concatenating waveforms for individual PPDU fields.

This list shows which functions you can use to build a PPDU for each PHY format.

- Very high throughput (VHT) — `wlanLSTF`, `wlanLLTF`, `wlanLSIG`, `wlanVHTSTF`, `wlanVHTLTF`, `wlanVHTSIGA`, `wlanVHTSIGB`, and `wlanVHTData`
- High throughput (HT) — `wlanLSTF`, `wlanLLTF`, `wlanLSIG`, `wlanHTSTF`, `wlanHTLTF`, `wlanHTSIG`, and `wlanHTData`
- Non-high-throughput (non-HT) — `wlanLSTF`, `wlanLLTF`, `wlanLSIG`, and `wlanNonHTData`

Generate VHT Waveform Using Waveform Generator Function

Create a VHT configuration object.

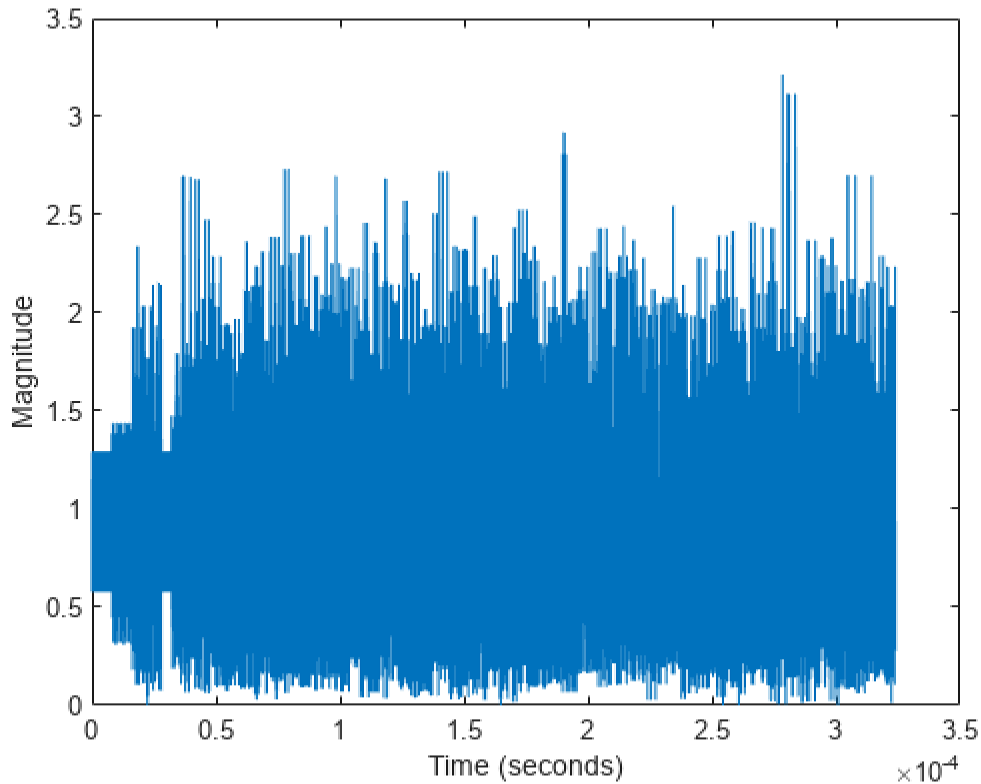
```
cfgVHT = wlanVHTConfig;
```

Generate the VHT PPDU. The length of the input data sequence in bits must be eight times the length of the PSDU, which is expressed in bytes. Turn off windowing.

```
bits = randi([0 1],cfgVHT.PSDULength*8,1);
y = wlanWaveformGenerator(bits,cfgVHT,WindowTransitionTime=0);
```

Plot the magnitude of the waveform.

```
fsVHT = wlanSampleRate(cfgVHT.ChannelBandwidth);
time = (0:length(y)-1)/fsVHT;
plot(time,abs(y))
xlabel ('Time (seconds)')
ylabel('Magnitude')
```



Build VHT Waveform From Individual PPDU Fields

Create L-STF, L-LTF, L-SIG, VHT-SIG-A, VHT-STF, VHT-LTF, and VHT-SIG-B preamble fields.

```
lstf = wlanLSTF(cfgVHT);
lltf = wlanLLTF(cfgVHT);
lsig = wlanLSIG(cfgVHT);
vhtSigA = wlanVHTSIGA(cfgVHT);
vhtstf = wlanVHTSTF(cfgVHT);
vhtltf = wlanVHTLTF(cfgVHT);
vhtSigB = wlanVHTSIGB(cfgVHT);
```

Generate the VHT-Data field using the input data `bits`.

```
vhtData = wlanVHTData(bits,cfgVHT);
```

Concatenate the individual fields to create a single PPDU.

```
z = [lstf; lltf; lsig; vhtSigA; vhtstf; vhtltf; vhtSigB; vhtData];
```

Verify that the PPDUs created by the two methods are identical.

```
isequal(y,z)
```

```
ans = logical
   1
```

See Also

Related Examples

- “Waveform Generation”

Generate VHT Multi-User Waveform

This example shows how to generate a VHT multi-user waveform from individual components. It also shows how to generate the same waveform by using the `wlanWaveformGenerator` function.

Create a VHT configuration object, specifying three users and three transmit antennas.

```
vht = wlanVHTConfig('NumUsers',3,'NumTransmitAntennas',3);
```

Set the number of space-time streams to the vector `[1 1 1]`, which indicates that each user is assigned one space-time stream. Set the user positions to `[0 1 2]`. Set the group ID to 5. Group ID values from 1 to 62 apply for multiuser operation.

```
vht.NumSpaceTimeStreams = [1 1 1];
vht.UserPositions = [0 1 2];
vht.GroupID = 5;
```

Set a different MCS value for each user.

```
vht.MCS = [0 2 4];
```

Set the APEP length to 2000, 1400, and 1800 bytes. Each element corresponds to the number of bytes assigned to each user.

```
vht.APEPLength = [2000 1400 1800]
```

```
vht = 
  wlanVHTConfig with properties:

       ChannelBandwidth: 'CBW80'
               NumUsers: 3
          UserPositions: [0 1 2]
    NumTransmitAntennas: 3
    NumSpaceTimeStreams: [1 1 1]
         SpatialMapping: 'Direct'
                    MCS: [0 2 4]
          ChannelCoding: 'BCC'
             APEPLength: [2000 1400 1800]
          GuardInterval: 'Long'
                GroupID: 5

   Read-only properties:
             PSDULength: [2000 6008 12019]
```

Display the PSDU lengths for the three users. The PSDU length is a function of both the APEP length and the MCS value.

```
vht.PSDULength
```

```
ans = 1×3

        2000        6008       12019
```

Display the field indices for the VHT waveform.

```
ind = wlanFieldIndices(vht)
```

```
ind = struct with fields:
       LSTF: [1 640]
       LLTF: [641 1280]
       LSIG: [1281 1600]
    VHTSIGA: [1601 2240]
     VHTSTF: [2241 2560]
     VHTLTF: [2561 3840]
    VHTSIGB: [3841 4160]
    VHTData: [4161 48000]
```

Create the individual fields that comprise the VHT waveform.

```
lstf = wlanLSTF(vht);
lltf = wlanLLTF(vht);
lsig = wlanLSIG(vht);
[vhtsigA,sigAbits] = wlanVHTSIGA(vht);
vhtstf = wlanVHTSTF(vht);
vhtltf = wlanVHTLTF(vht);
[vhtsigB,sigBbits] = wlanVHTSIGB(vht);
```

Extract the first two VHT-SIG-A information bits and convert them to their decimal equivalent.

```
bw = bit2int(double(sigAbits(1:2)),2,false)
```

```
bw = 2
```

The value, 2, corresponds to an 80 MHz bandwidth (see `wlanVHTSIGA`).

Extract VHT-SIG-A information bits 5 through 10, and convert them to their decimal equivalent.

```
groupid = bit2int(double(sigAbits(5:10)),6,false)
```

```
groupid = 5
```

The extracted group ID, 5, matches the corresponding property in the VHT configuration object.

Extract the packet length from the VHT-SIG-B information bits. For multiuser operation with an 80 MHz bandwidth, the first 19 bits contain the APEP length information. Convert the field lengths to their decimal equivalents. Multiply them by 4 because the length of the VHT-SIG-B field is expressed in units of 4 bytes.

```
pktLen = bit2int(double(sigBbits(1:19,:)),19,false)'*4
```

```
pktLen = 3×1

        2000
        1400
        1800
```

Confirm that the extracted APEP length matches the value set in the configuration object.

```
isequal(pktLen',vht.APEPLength)
```

```
ans = logical
   1
```

Extract the MCS values from the VHT-SIG-B information bits. The MCS component is specified by bits 20 to 23.

```
mcs = bit2int(double(sigBbits(20:23,:)),4,false)'
```

```
mcs = 3×1

     0
     2
     4
```

The values correspond to those set in the VHT configuration object.

Create three data sequences, one for each user.

```
d1 = randi([0 1],vht.PSDULength(1)*8,1);
d2 = randi([0 1],vht.PSDULength(2)*8,1);
d3 = randi([0 1],vht.PSDULength(3)*8,1);
```

Generate a VHT data field using these data sequences.

```
vhtdata = wlanVHTData({d1 d2 d3},vht);
```

Generate a multiuser VHT waveform with windowing is disabled. Extract the data field from the waveform.

```
wv = wlanWaveformGenerator({d1 d2 d3},vht,'WindowTransitionTime',0);
wvdata = wv(ind.VHTData(1):ind.VHTData(2),:);
```

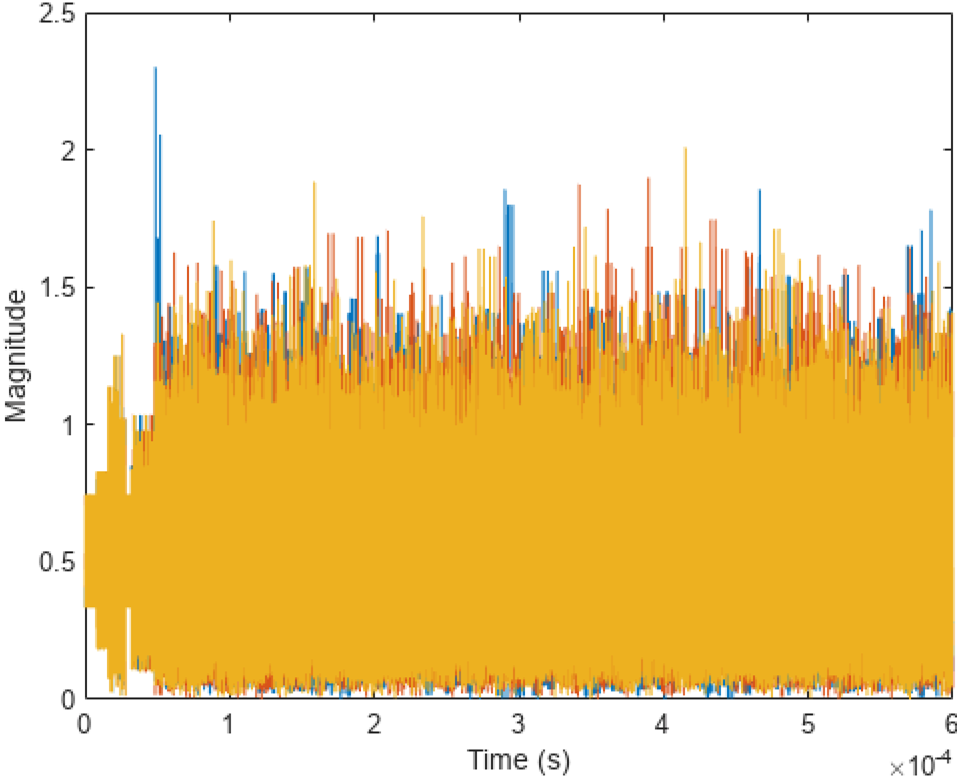
Confirm that the two generation approaches produce identical results.

```
isequal(vhtdata,wvdata)
```

```
ans = logical
   1
```

Visualize the waveform by plotting its magnitude.

```
t = ((1:length(wv))'-1)/80e6;
plot(t,abs(wv))
xlabel('Time (s)')
ylabel('Magnitude')
```


Magnitude
Time (s)
2.5
2
1.5
1
0.5
0
0
1
2
3
4
5
6
×10-4

Basic VHT Data Recovery

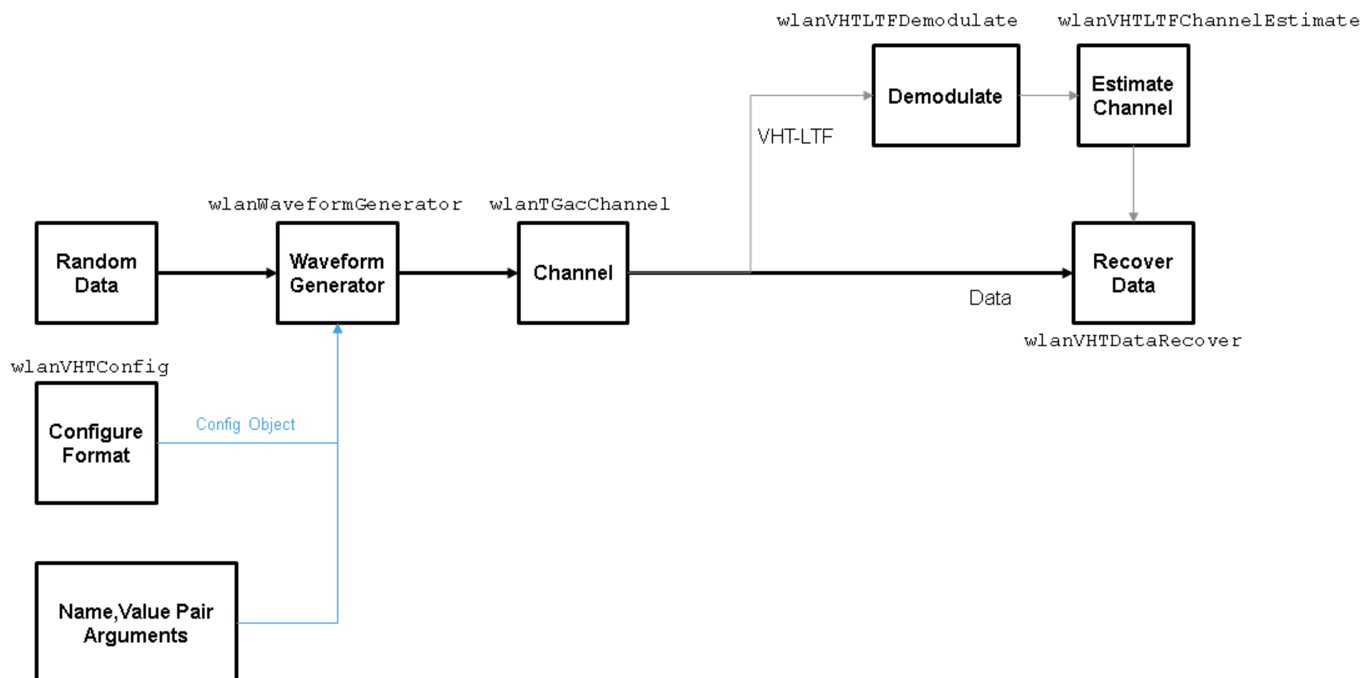
This example shows how to perform basic VHT data recovery. It also shows how to recover VHT data when the received signal has a carrier frequency offset. Similar procedures can be used to recover data with the HT and non-HT formats.

Basic Data Recovery

WLAN Toolbox™ provides functions to generate and recover IEEE® 802.11ac™ standards-compliant waveforms. The data recovery process comprises these steps.

1. Generate a VHT waveform
2. Pass the waveform through a channel
3. Extract the VHT-LTF and demodulate
4. Estimate the channel by using the demodulated VHT-LTF
5. Extract the data field
6. Recover the data by using the channel and noise variance estimates

The block diagram shows these steps, along with their corresponding commands.



Create VHT configuration object.

```
cfg = wlanVHTConfig;
```

Create a VHT transmit waveform by using the VHT configuration object. Set the data sequence to `[1;0;1;1]`. The waveform generator function repeats the data sequence to generate the specified number of packets.

```
txSig = wlanWaveformGenerator([1;0;1;1],cfg);
```

Pass the received signal through an AWGN channel.

```
rxSig = awgn(txSig,10);
```

Determine the field indices of the waveform.

```
ind = wlanFieldIndices(cfg);
```

Extract the VHT-LTF from the received signal.

```
rxVHTLTF = rxSig(ind.VHTLTF(1):ind.VHTLTF(2),:);
```

Demodulate the VHT-LTF. Estimate the channel response by using the demodulated signal.

```
demodVHTLTF = wlanVHTLTFDemodulate(rxVHTLTF,cfg);
chEst = wlanVHTLTFChannelEstimate(demodVHTLTF,cfg);
```

Extract the VHT data field.

```
rxData = rxSig(ind.VHTData(1):ind.VHTData(2),:);
```

Recover the information bits by using the channel and noise variance estimates. Confirm that the first 8 bits match two repetitions of the input data sequence of `[1;0;1;1]`.

```
rxBits = wlanVHTDataRecover(rxData,chEst,0.1,cfg);
```

```
rxBits(1:8)
```

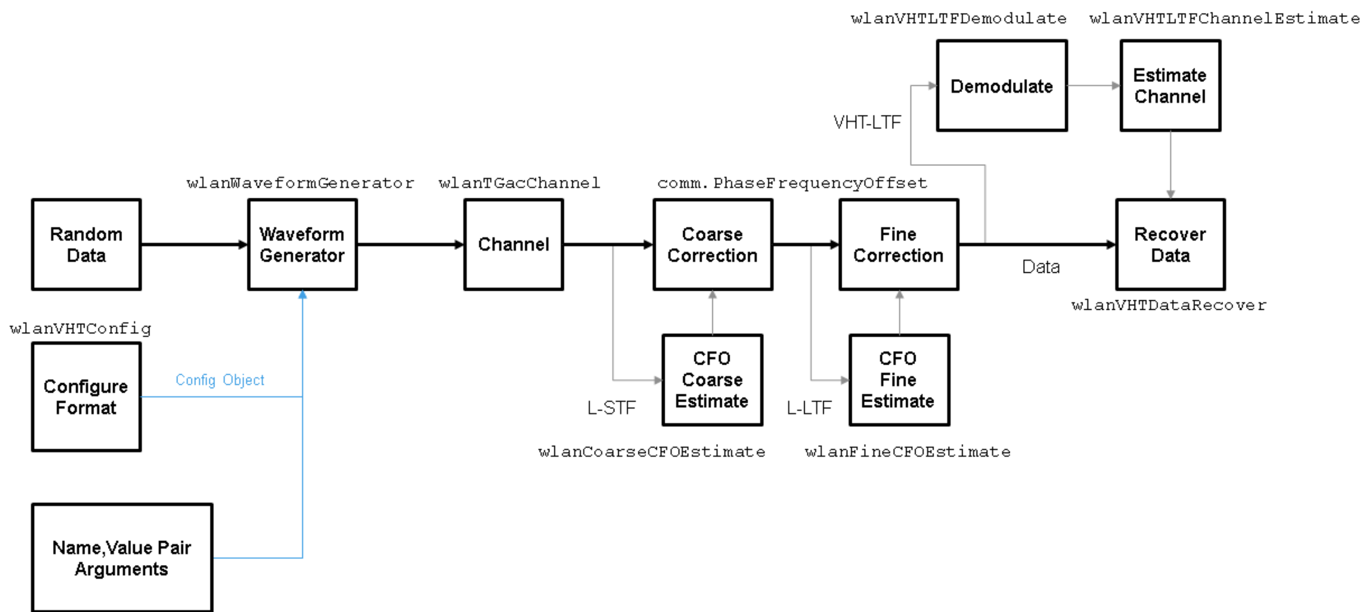
```
ans = 8x1 int8 column vector

   1
   0
   1
   1
   1
   0
   1
   1
```

Data Recovery with Frequency Correction

Data recovery when a carrier frequency offset is present is accomplished by these steps.

1. Generate a VHT waveform
2. Pass the waveform through a channel
3. Extract the L-STF and perform a coarse frequency offset estimate
4. Correct for the offset by using the coarse estimate
5. Extract the L-LTF and perform a fine frequency offset estimate
6. Correct for the offset by using the fine estimate
7. Extract the VHT-LTF and demodulate
8. Estimate the channel by using the demodulated VHT-LTF
9. Extract the data field
10. Recover the data by using the channel and noise variance estimates

The block diagram shows these steps, along with their corresponding commands.



Set the channel bandwidth and sample rate.

```
cbw = 'CBW160';
fs = 160e6;
```

Create a VHT configuration object that supports a 2x2 MIMO transmission.

```
cfg = wlanVHTConfig('ChannelBandwidth',cbw, ...
    'NumTransmitAntennas',2,'NumSpaceTimeStreams',2);
```

Generate a VHT waveform containing a random PSDU.

```
txPSDU = randi([0 1],cfg.PSDULength*8,1);
txSig = wlanWaveformGenerator(txPSDU,cfg);
```

Create a 2x2 TGac channel.

```
tgacChan = wlanTGacChannel('SampleRate',fs,'ChannelBandwidth',cbw, ...
    'NumTransmitAntennas',2,'NumReceiveAntennas',2);
```

Create a phase and frequency offset object.

```
pfOffset = comm.PhaseFrequencyOffset('SampleRate',fs,'FrequencyOffsetSource','Input port');
```

Pass the transmitted waveform through the noisy TGac channel.

```
rxSigNoNoise = tgacChan(txSig);
rxSig = awgn(rxSigNoNoise,15);
```

Introduce a frequency offset of 500 Hz to the received signal.

```
rxSigFreqOffset = pfOffset(rxSig,500);
```

Find the start and stop indices for all component fields of the PPDU.

```
ind = wlanFieldIndices(cfg);
```

Extract the L-STF. Estimate and correct for the carrier frequency offset.

```
rxLSTF = rxSigFreqOffset(ind.LSTF(1):ind.LSTF(2),:);

foffset1 = wlanCoarseCFOEstimate(rxLSTF,cbw);
rxSig1 = pfOffset(rxSigFreqOffset,-foffset1);
```

Extract the L-LTF from the corrected signal. Estimate and correct for the residual frequency offset.

```
rxLLTF = rxSig1(ind.LLTF(1):ind.LLTF(2),:);

foffset2 = wlanFineCFOEstimate(rxLLTF,cbw);
rxSig2 = pfOffset(rxSig1,-foffset2);
```

Extract and demodulate the VHT-LTF. Estimate the channel coefficients.

```
rxVHTLTF = rxSig2(ind.VHTLTF(1):ind.VHTLTF(2),:);
demodVHTLTF = wlanVHTLTFDemodulate(rxVHTLTF,cfg);
chEst = wlanVHTLTFChannelEstimate(demodVHTLTF,cfg);
```

Extract the VHT data field from the received and frequency-corrected PPDU. Recover the data field.

```
rxData = rxSig2(ind.VHTData(1):ind.VHTData(2),:);
rxPSDU = wlanVHTDataRecover(rxData,chEst,0.03,cfg);
```

Calculate the number of bit errors in the received packet.

```
numErr = biterr(txPSDU,rxPSDU)
```

```
numErr = 2
```

802.11ax Waveform Generation

This example shows how to parameterize and generate different IEEE® 802.11ax™ (Wi-Fi 6) high efficiency (HE) format packets.

Introduction

IEEE Std 802.11ax™-2021 [1] specifies four high efficiency (HE) packet formats:

1. Single-user
2. Extended-range single-user
3. Multi-user
4. Trigger-based

This example shows how packets can be generated for these different formats, and demonstrates some of the key features of the draft standard [1].

HE Single-User Format

An HE single-user (SU) packet is a full-band transmission to a single user. The transmit parameters for the HE SU format are configured using a `wlanHESUConfig` object. The `wlanHESUConfig` object can be configured to operate in extended-range mode. To enable or disable this mode, set the `ExtendedRange` property to `true` or `false`. In this example we create a configuration for an HE SU transmission and configure transmission properties.

```
cfgSU = wlanHESUConfig;
cfgSU.ExtendedRange = false;      % Do not use extended-range format
cfgSU.ChannelBandwidth = 'CBW20'; % Channel bandwidth
cfgSU.APEPLength = 1000;          % Payload length in bytes
cfgSU.MCS = 0;                    % Modulation and coding scheme
cfgSU.ChannelCoding = 'LDPC';     % Channel coding
cfgSU.NumSpaceTimeStreams = 1;    % Number of space-time streams
cfgSU.NumTransmitAntennas = 1;    % Number of transmit antennas
```

A single-user packet can be generated with the waveform generator, `wlanWaveformGenerator`. The `getPSDULength()` method returns the required PSDU length given the transmission configuration. This length is used to create a random PSDU for transmission.

```
psdu = randi([0 1],getPSDULength(cfgSU)*8,1,'int8'); % Random PSDU
txSUWaveform = wlanWaveformGenerator(psdu,cfgSU);    % Create packet
```

HE Extended-Range Single-User Format

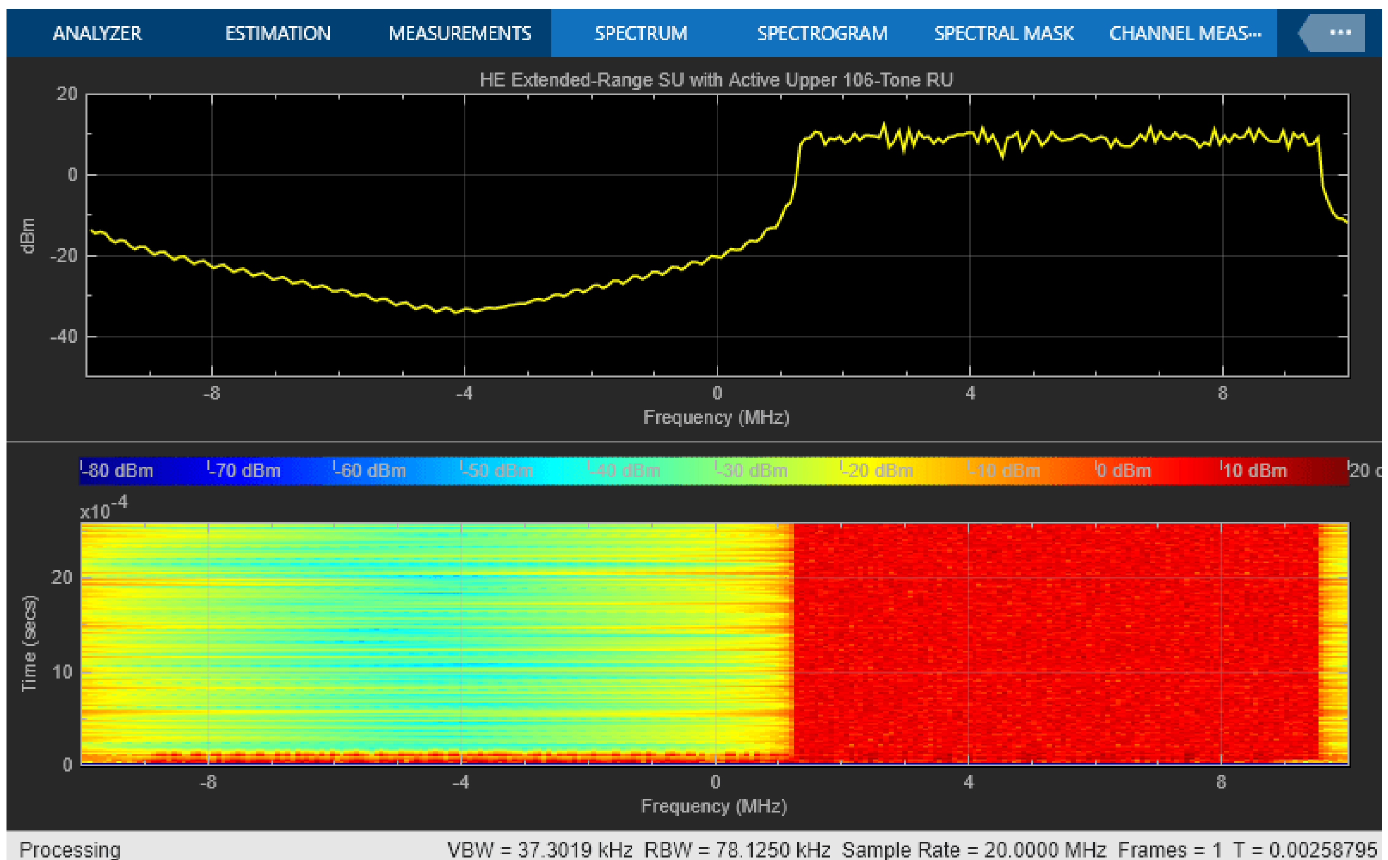
An extended-range single-user packet has the same fields as the standard single-user format, but the powers of some fields are boosted, and some fields are repeated to improve performance at low SNRs. An extended-range packet can be configured using a `wlanHESUConfig` object with `ChannelBandwidth` set to `'CBW20'` and `ExtendedRange` set to `true`. An extended-range packet has an option to only transmit in the upper 106-tone resource unit (RU) within the 20 MHz channel, or over the entire bandwidth. This can be configured with the `Upper106ToneRU` property:

```
cfgExtSU = cfgSU;
cfgExtSU.ExtendedRange = true;  % Enable extended-range format
cfgExtSU.Upper106ToneRU = true; % Use only upper 106-tone RU
```

```
% Generate a packet
psdu = randi([0 1],getPSDULength(cfgExtSU)*8,1,'int8'); % Random PSDU
txExtSUWaveform = wlanWaveformGenerator(psdu,cfgExtSU);   % Create packet
```

Look at the spectrum and spectrogram of the generated signal. In the spectrogram you can see that the packet headers use the available bandwidth, however, the data portion only occupies the upper half of the channel.

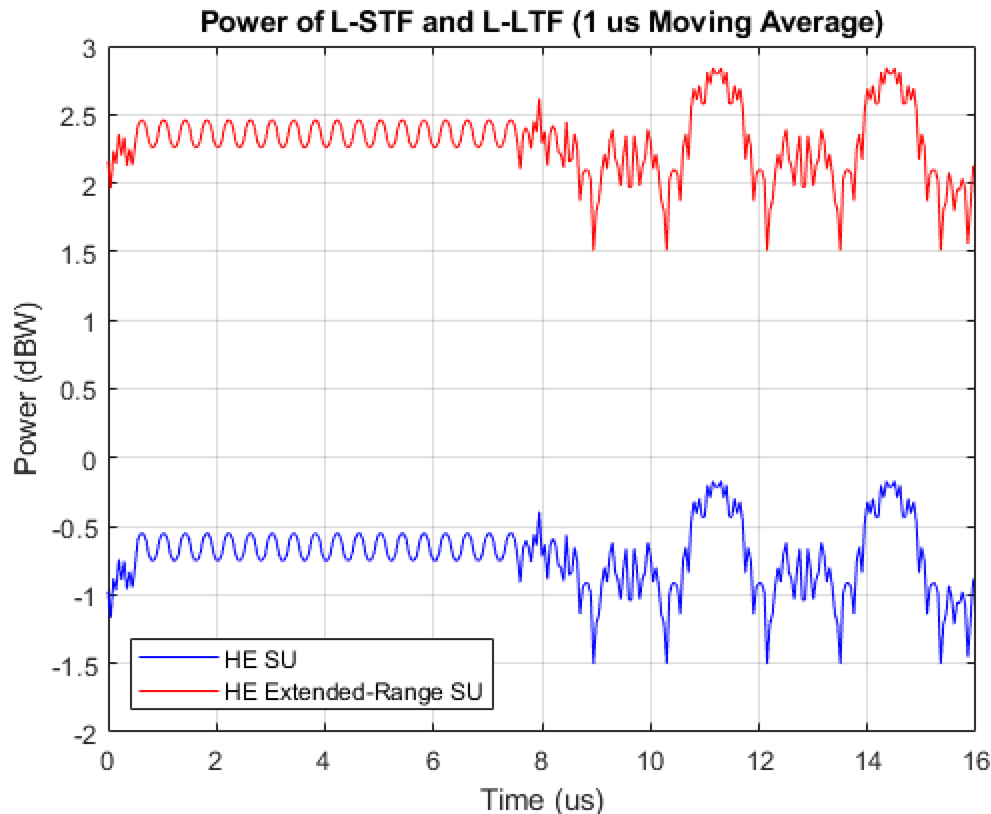
```
fs = wlanSampleRate(cfgExtSU); % Get baseband sample rate
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgExtSU);
fftsize = ofdmInfo.FFTLength; % Use the data field fft size
rbw = fs/fftsize; % Resoluton bandwidth
spectrumScope = spectrumAnalyzer(SampleRate=fs,...
    RBWSource='property',RBW=rbw,...
    AveragingMethod='exponential',ForgettingFactor=0.25,...
    YLimits=[-50,20],...
    Title='HE Extended-Range SU with Active Upper 106-Tone RU');
spectrumScope.ViewType = 'Spectrum and Spectrogram';
spectrumScope.TimeSpanSource = 'Property';
spectrumScope.TimeSpan = length(txExtSUWaveform)/fs;
spectrumScope(txExtSUWaveform)
```



If you compare the power of the L-STF and L-LTF fields you can see that the extended-range transmission is boosted by 3 dB.

```
figure;
ind = wlanFieldIndices(cfgExtSU);
```

```
t = (0:(double(ind.LLTF(2))-1))/fs*1e6;
plot(t,20*log10(movmean(abs(txSUWaveform(1:ind.LLTF(2))),20)),'-b')
hold on;
plot(t,20*log10(movmean(abs(txExtSUWaveform(1:ind.LLTF(2))),20)),'-r')
grid on;
title('Power of L-STF and L-LTF (1 us Moving Average)');
xlabel('Time (us)');
ylabel('Power (dBW)');
legend('HE SU','HE Extended-Range SU','Location','SouthWest');
```



HE Multi-User Format - OFDMA

The HE multi-user (HE MU) format can be configured for an OFDMA transmission, a MU-MIMO transmission, or a combination of the two. This flexibility allows an HE MU packet to transmit to a single user over the whole band, multiple users over different parts of the band (OFDMA), or multiple users over the same part of the band (MU-MIMO).

For an OFDMA transmission, the channel bandwidth is divided into resource units (RUs). An RU is a group of subcarriers assigned to one or more users. An RU is defined by a size (the number of subcarriers) and an index. The RU index specifies the location of the RU within the channel. For example, in an 80 MHz transmission there are four possible 242-tone RUs, one in each 20 MHz subchannel. RU# 242-1 (size 242, index 1) is the RU occupying the lowest absolute frequency within the 80 MHz, and RU# 242-4 (size 242, index 4) is the RU occupying the highest absolute frequency. The draft standard defines possible sizes and locations of RUs in Section 27.3.2.2 of [1].

The assignment of RUs in a transmission is defined by the allocation index. The allocation index is defined in Table 27-26 of [1]. For each 20 MHz subchannel, an 8-bit index describes the number and

size of RUs, and the number of users transmitted on each RU. The allocation index also determines which content channel is used to signal a user in HE-SIG-B. The allocation indices within Table 27-26, and the corresponding RU assignments, are provided in the table returned by the function `heRUAllocationTable`. The first 10 allocations within the table are shown below. For each allocation index, the 8-bit allocation index, the number of users, number of RUs, RU indices, RU sizes, and number of users per RU are displayed. A note is also provided about allocations which are reserved, or serve a special purpose. The allocation table can also be viewed in the Appendix.

```
allocationTable = heRUAllocationTable;
disp('First 10 entries in the allocation table: ')
disp(allocationTable(1:10,:));
```

```
First 10 entries in the allocation table:
    Allocation    BitAllocation    NumUsers    NumRUs          RUIndices                RUS
    __________    _____________    ________    ______    _____________________    ______________

        0          "00000000"          9          9      {[1 2 3 4 5 6 7 8 9]}    {[26 26 26 26
        1          "00000001"          8          8      {[  1 2 3 4 5 6 7 4]}    {[   26 26 26
        2          "00000010"          8          8      {[  1 2 3 4 5 3 8 9]}    {[   26 26 26
        3          "00000011"          7          7      {[    1 2 3 4 5 3 4]}    {[      26 26
        4          "00000100"          8          8      {[  1 2 2 5 6 7 8 9]}    {[   26 26 52
        5          "00000101"          7          7      {[    1 2 2 5 6 7 4]}    {[      26 26
        6          "00000110"          7          7      {[    1 2 2 5 3 8 9]}    {[      26 26
        7          "00000111"          6          6      {[      1 2 2 5 3 4]}    {[         26
        8          "00001000"          8          8      {[  1 3 4 5 6 7 8 9]}    {[   52 26 26
        9          "00001001"          7          7      {[    1 3 4 5 6 7 4]}    {[      52 26
```

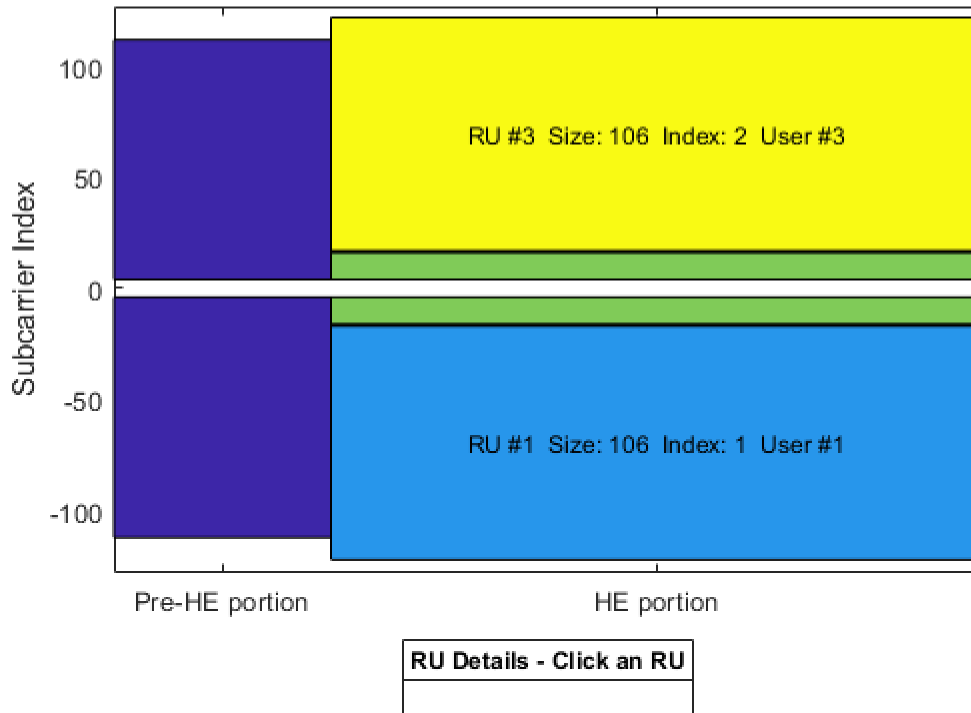
A `wlanHEMUConfig` object is used to configure the transmission of an HE MU packet. The allocation index for each 20 MHz subchannel must be provided when creating an HE MU configuration object, `wlanHEMUConfig`. An integer between 0 and 223, corresponding to the 8-bit number in Table 27-26 of [1], must be provided for each 20 MHz subchannel.

The allocation index can be provided as a decimal or 8-bit binary sequence. In this example, a 20 MHz HE MU configuration is created with 8-bit allocation index "10000000". This is equivalent to the decimal allocation index 128. This configuration specifies 3 RUs, each with one user.

```
allocationIndex = "10000000"; % 3 RUs, 1 user per RU
cfgMU = wlanHEMUConfig(allocationIndex);
```

The `showAllocation` method visualizes the occupied RUs and subcarriers for the specified configuration. The colored blocks illustrate the occupied subcarriers in the pre-HE and HE portions of the packet. White indicates subcarriers are unoccupied. The pre-HE portion illustrates the occupied subcarriers in the fields preceding HE-STF. The HE portion illustrates the occupied subcarriers in the HE-STF, HE-LTF and HE-Data field and therefore shows the RU allocation. Clicking on an RU will display information about the RU. The RU number corresponds to the i-th RU element of the `cfgMU.RU` property. The size and index are the details of the RU. The RU index is the i-th possible RU of the corresponding RU size within the channel bandwidth, for example Index 2 is the 2nd possible 106-tone RU within the 20 MHz channel bandwidth. The user number corresponds to the i-th User element of the `cfgMU.User` property, and the user field in HE-SIG-B. Note the middle RU (RU #2) is split across the DC subcarriers.

```
showAllocation(cfgMU);
axAlloc = gca; % Get axis handle for subsequent plotting
```

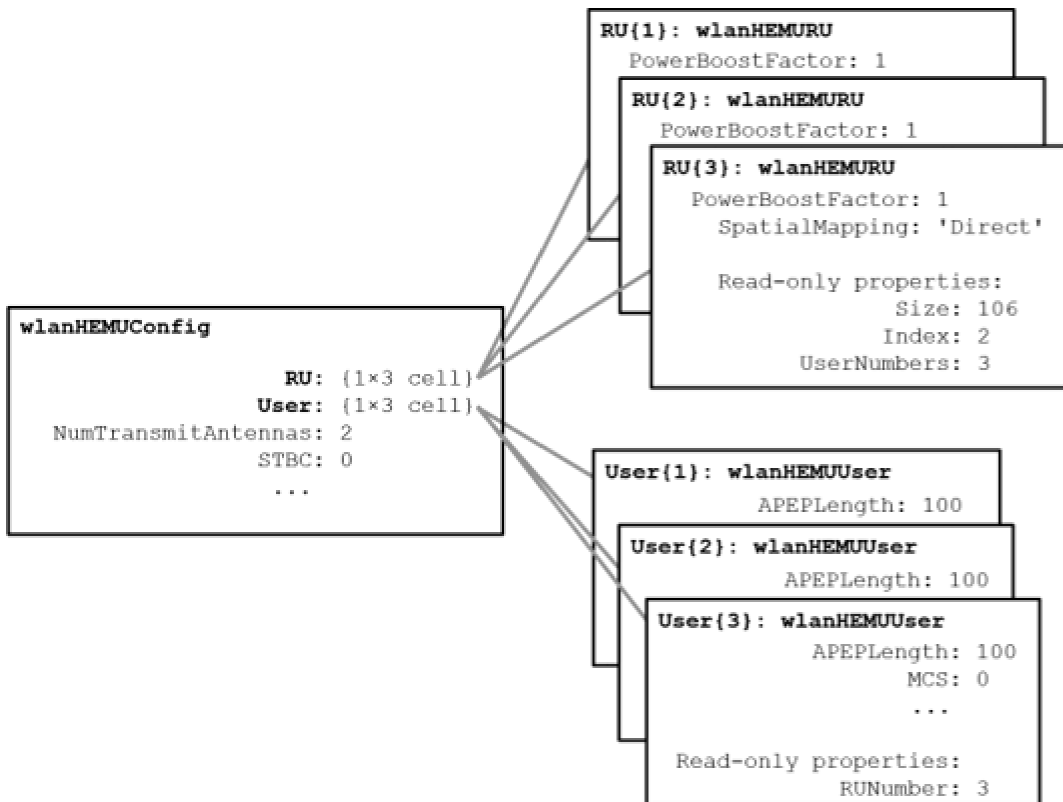


The `ruInfo` method provides details of the RUs in the configuration. In this case we can see three users and three RUs.

```
allocInfo = ruInfo(cfgMU);
disp('Allocation info:')
disp(allocInfo)
```

```
Allocation info:
                    NumUsers: 3
                      NumRUs: 3
                   RUIndices: [1 5 2]
                     RUSizes: [106 26 106]
               NumUsersPerRU: [1 1 1]
    NumSpaceTimeStreamsPerRU: [1 1 1]
       PowerBoostFactorPerRU: [1 1 1]
                   RUNumbers: [1 2 3]
```

The properties of `cfgMU` describe the transmission configuration. The `cfgMU.RU` and `cfgMU.User` properties of `cfgMU` are cell arrays. Each element of the cell arrays contains an object which configures an RU or a User. When the `cfgMU` object is created, the elements of `cfgMU.RU` and `cfgMU.User` are configured to create the desired number of RUs and users. Each element of `cfgMU.RU` is a `wlanHEMURU` object describing the configuration of an RU. Similarly, each element of `cfgMU.User` is a `wlanHEMUUser` object describing the configuration of a User. This object hierarchy is shown below:



In this example, three RUs are specified by the allocation index 128, therefore `cfgMU.RU` is a cell array with three elements. The index and size of each RU are configured according to the allocation index used to create `cfgMU`. After the object is created, each RU can be configured to create the desired transmission configuration, by setting the properties of the appropriate RU object. For example, the spatial mapping and power boost factor can be configured per RU. The `Size` and `Index` properties of each RU are fixed once the object is created, and therefore are read-only properties. Similarly, the `UserNumbers` property is read-only and indicates which user is transmitted on the RU. For this configuration the first RU is size 106, index 1.

```
disp('First RU configuration:')
disp(cfgMU.RU{1})
```

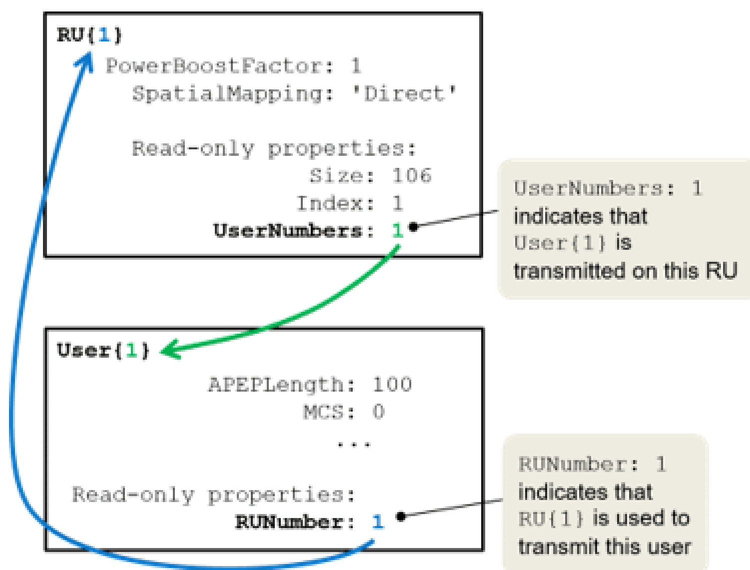
```
First RU configuration:
  wlanHEMURU with properties:

    PowerBoostFactor: 1
      SpatialMapping: 'Direct'

   Read-only properties:
                Size: 106
               Index: 1
         UserNumbers: 1
```

In this example, the allocation index specifies three users in the transmission, therefore, `cfgMU.User` contains three elements. The transmission properties of users can be configured by modifying individual user objects, for example the MCS, APEP length and channel coding scheme. The read-only `RUNumber` property indicates which RU is used to transmit this user.

```
disp('First user configuration:')
disp(cfgMU.User{1})
```

```
First user configuration:
  wlanHEMUUser with properties:

               APEPLength: 100
                      MCS: 0
      NumSpaceTimeStreams: 1
                      DCM: 0
            ChannelCoding: 'LDPC'
                    STAID: 0
     NominalPacketPadding: 0
     PostFECPaddingSource: 'mt19937ar with seed'
       PostFECPaddingSeed: 1

   Read-only properties:
                 RUNumber: 1
```

The number of users per RU, and mapping of users to RUs is determined by the allocation index. The `UserNumbers` property of an RU object indicates which users (elements of the `cfgMU.User` cell array) are transmitted on that RU. Similarly, the `RUNumber` property of each User object, indicates which RU (element of the `cfgMU.RU` cell array) is used to transmit the user:

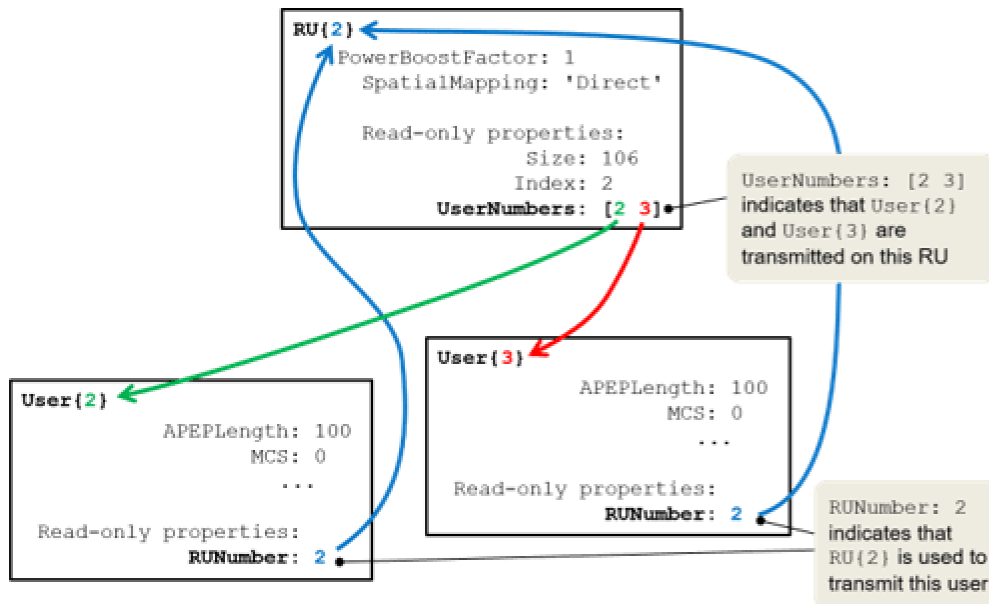


This allows the properties of an RU associated with a User to be accessed easily:

```
ruNum = cfgMU.User{2}.RUNumber; % Get the RU number associated with user 2
disp(cfgMU.RU{ruNum}.SpatialMapping); % Display the spatial mapping
```

```
Direct
```

When an RU serves multiple users, in a MU-MIMO configuration, the `UserNumbers` property can index multiple users:



Once the cfgMU object is created, transmission parameters can be set as demonstrated below.

```
% Configure RU 1 and user 1
cfgMU.RU{1}.SpatialMapping = 'Direct';
cfgMU.User{1}.APEPLength = 1e3;
cfgMU.User{1}.MCS = 2;
cfgMU.User{1}.NumSpaceTimeStreams = 4;
cfgMU.User{1}.ChannelCoding = 'LDPC';

% Configure RU 2 and user 2
cfgMU.RU{2}.SpatialMapping = 'Fourier';
cfgMU.User{2}.APEPLength = 500;
cfgMU.User{2}.MCS = 3;
cfgMU.User{2}.NumSpaceTimeStreams = 2;
cfgMU.User{2}.ChannelCoding = 'LDPC';

% Configure RU 3 and user 3
cfgMU.RU{3}.SpatialMapping = 'Fourier';
cfgMU.User{3}.APEPLength = 100;
cfgMU.User{3}.MCS = 4;
cfgMU.User{3}.DCM = true;
cfgMU.User{3}.NumSpaceTimeStreams = 1;
cfgMU.User{3}.ChannelCoding = 'BCC';
```

Some transmission parameters are common for all users in the HE MU transmission.

```
% Configure common parameters for all users
cfgMU.NumTransmitAntennas = 4;
cfgMU.SIGBMCS = 2;
```

To generate the HE MU waveform, we first create a random PSDU for each user. A cell array is used to store the PSDU for each user as the PSDU lengths differ. The `getPSDULength()` method returns a vector with the required PSDU per user given the configuration. The waveform generator is then used to create a packet.

```
psduLength = getPSDULength(cfgMU);
psdu = cell(1,allocInfo.NumUsers);
for i = 1:allocInfo.NumUsers
    psdu{i} = randi([0 1],psduLength(i)*8,1,'int8'); % Generate random PSDU
end

% Create MU packet
txMUWaveform = wlanWaveformGenerator(psdu,cfgMU);
```

To configure an OFDMA transmission with a channel bandwidth greater than 20 MHz, an allocation index must be provided for each 20 MHz subchannel. For example, to configure an 80 MHz OFDMA transmission, four allocation indices are required. In this example four 242-tone RUs are configured. The allocation index `192` specifies one 242-tone RU with a single user in a 20 MHz subchannel, therefore the allocation indices `[192 192 192 192]` are used to create four of these RUs, over 80 MHz:

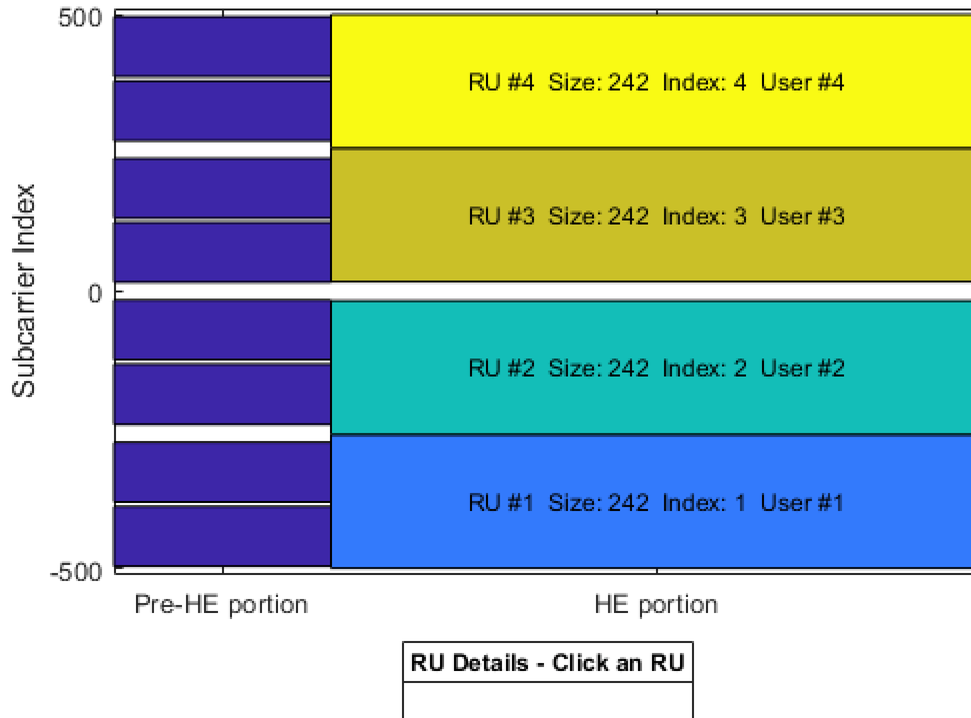
```
% Display 192 allocation index properties in the table (the 193rd row)
disp('Allocation #192 table entry:')
disp(allocationTable(193,:))

% Create 80 MHz MU configuration, with four 242-tone RUs
cfgMU80MHz = wlanHEMUConfig([192 192 192 192]);
```

```
Allocation #192 table entry:
    Allocation    BitAllocation    NumUsers    NumRUs    RUIndices    RUSizes    NumUsersPerRU
    __________    _____________    ________    ______    _________    _______    _____________

       192         "11000000"          1          1        {[1]}      {[242]}        {[1]}
```

When multiple 20 MHz subchannels are specified, the `ChannelBandwidth` property is set to the appropriate value. For this configuration it is set to `'CBW80'` as four 20 MHz subchannels are specified. This is also visible in the allocation plot.

```
disp('Channel bandwidth for HE MU allocation:')
disp(cfgMU80MHz.ChannelBandwidth)
showAllocation(cfgMU80MHz,axAlloc)
```

```
Channel bandwidth for HE MU allocation:
CBW80
```



HE Multi-User Format - MU-MIMO

An HE MU packet can also transmit an RU to multiple users using MU-MIMO. For a full band MU-MIMO allocation, the allocation indices between 192 and 199 configure a full-band 20 MHz allocation (242-tone RU). The index within this range determines how many users are configured. The allocation details can be viewed in the allocation table. Note the `NumUsers` column in the table grows with index but the `NumRUs` is always 1. The allocation table can also be viewed in the Appendix.

```
disp('Allocation #192-199 table entries:')
disp(allocationTable(193:200,:)) % Indices 192-199 (rows 193 to 200)
```

```
Allocation #192-199 table entries:
    Allocation     BitAllocation     NumUsers    NumRUs    RUIndices    RUSizes    NumUsersPerRU
    __________    _____________    ________    ______    _________    _______    _____________

       192         "11000000"          1          1         {[1]}      {[242]}        {[1]}    
       193         "11000001"          2          1         {[1]}      {[242]}        {[2]}    
       194         "11000010"          3          1         {[1]}      {[242]}        {[3]}    
       195         "11000011"          4          1         {[1]}      {[242]}        {[4]}    
       196         "11000100"          5          1         {[1]}      {[242]}        {[5]}    
       197         "11000101"          6          1         {[1]}      {[242]}        {[6]}    
       198         "11000110"          7          1         {[1]}      {[242]}        {[7]}    
       199         "11000111"          8          1         {[1]}      {[242]}        {[8]}    
```

The allocation index 193 transmits a 20 MHz 242-tone RU to two users. In this example, we will create a transmission with a random spatial mapping matrix which maps a single space-time stream for each user, onto two transmit antennas.

```
% Configure 2 users in a 20 MHz channel
cfgMUMIMO = wlanHEMUConfig(193);

% Set the transmission properties of each user
cfgMUMIMO.User{1}.APEPLength = 100; % Bytes
cfgMUMIMO.User{1}.MCS = 2;
cfgMUMIMO.User{1}.ChannelCoding = 'LDPC';
cfgMUMIMO.User{1}.NumSpaceTimeStreams = 1;

cfgMUMIMO.User{2}.APEPLength = 1000; % Bytes
cfgMUMIMO.User{2}.MCS = 6;
cfgMUMIMO.User{2}.ChannelCoding = 'LDPC';
cfgMUMIMO.User{2}.NumSpaceTimeStreams = 1;

% Get the number of occupied subcarriers in the RU
ruIndex = 1; % Get the info for the first (and only) RU
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgMUMIMO,ruIndex);
numST = ofdmInfo.NumTones; % Number of occupied subcarriers

% Set the number of transmit antennas and generate a random spatial mapping
% matrix
numTx = 2;
allocInfo = ruInfo(cfgMUMIMO);
numSTS = allocInfo.NumSpaceTimeStreamsPerRU(ruIndex);
cfgMUMIMO.NumTransmitAntennas = numTx;
cfgMUMIMO.RU{ruIndex}.SpatialMapping = 'Custom';
cfgMUMIMO.RU{ruIndex}.SpatialMappingMatrix = rand(numST,numSTS,numTx);

% Create packet with a repeated bit sequence as the PSDU
txMUMIMOWaveform = wlanWaveformGenerator([1 0 1 0],cfgMUMIMO);
```

A full band MU-MIMO transmission with a channel bandwidth greater than 20 MHz is created by providing a single RU allocation index within the range 200-223 when creating the `wlanHEMUConfig` object. For these allocations HE-SIG-B compression is used.

The allocation indices between 200 and 207 configure a full-band MU-MIMO 40 MHz allocation (484-tone RU). The index within this range determines how many users are configured. The allocation details can be viewed in the allocation table. Note the `NumUsers` column in the table grows with index but the `NumRUs` is always 1.

```
disp('Allocation #200-207 table entries:')
disp(allocationTable(201:208,:)) % Indices 200-207 (rows 201 to 208)
```

```
Allocation #200-207 table entries:
    Allocation    BitAllocation    NumUsers    NumRUs    RUIndices    RUSizes    NumUsersPerRU
    __________    _____________    ________    ______    _________    _______    _____________

       200         "11001000"         1          1         {[1]}      {[484]}        {[1]}
       201         "11001001"         2          1         {[1]}      {[484]}        {[2]}
       202         "11001010"         3          1         {[1]}      {[484]}        {[3]}
       203         "11001011"         4          1         {[1]}      {[484]}        {[4]}
       204         "11001100"         5          1         {[1]}      {[484]}        {[5]}
       205         "11001101"         6          1         {[1]}      {[484]}        {[6]}
       206         "11001110"         7          1         {[1]}      {[484]}        {[7]}
```

```
      207         "11001111"         8          1         {[1]}      {[484]}        {[8]}
```

Similarly, the allocation indices between 208 and 215 configure a full-band MU-MIMO 80 MHz allocation (996-tone RU), and the allocation indices between 216 and 223 configure a full-band MU-MIMO 160 MHz allocation (2x996-tone RU).

As an example, the allocation index `203` specifies a 484-tone RU with 4 users:

```
cfg484MU = wlanHEMUConfig(203);
showAllocation(cfg484MU,axAlloc)
```

HE Multi-User Format - OFDMA with RU Sizes Greater Than 242 Subcarriers

For an HE MU transmission with a channel bandwidth greater than 20 MHz, two HE-SIG-B content channels are used to signal user configurations. These content channels are duplicated over each 40 MHz subchannel for larger channel bandwidths, as described in Section 27.3.11.8.5 of [1]. When an RU size greater than 242 is specified as part of an OFDMA system, the users assigned to the RU can be signaled on either of the two HE-SIG-B content channels. The allocation index provided when creating an `wlanHEMUConfig` object controls which content channel each user is signaled on. The allocation table in the Appendix shows the relevant allocation indices.

As an example, consider the following 80 MHz configuration which serves 7 users:

- One 484-tone RU (RU #1) with four users (users #1-4)
- One 242-tone RU (RU #2) with one user (user #5)

- Two 106-tone RUs (RU #3 and #4), each with one user (users #6 and #7)

To configure an 80 MHz OFDMA transmission, four allocation indices are required, one for each 20 MHz subchannel. To configure the above scenario the allocation indices below are used:

```
[X Y 192 96]
```

- X and Y configure the 484-tone RU, with users #1-4. The possible values of X and Y are discussed below.
- 192 configures a 242-tone RU with one user, user #5.
- 96 signals two 106-tone RUs, each with one user, users #6 and #7.

The selection of X and Y configures the appropriate number of users in the 242-tone RU, and determines which HE-SIG-B content channel is used to signal the users. A 484-tone RU spans two 20 MHz subchannels, therefore two allocation indices are required. All seven users from the four RUs will be signaled on the HE-SIG-B content channels, but for now we will only consider the signaling of users on the 484-tone RU. For the 484-tone RU, the four users can be signaled on the two HE-SIG-B content channels in different combinations as shown in Table 1.

Combination	Content Channel 1	Content Channel 2
A	User 1	Users 2-4
B	Users 1-2	Users 3-4
C	Users 1-3	User 4
D	Users 1-4	No users
E	No users	Users 1-4

Table 1

An allocation index within the range 200-207 specifies 1-8 users on a 484-tone RU. To signal no users on a content channel, the allocation index 114 or 115 can be used, for a 484-tone or 996-tone RU. Therefore, the combinations in Table 1 can be defined using two allocation indices as shown in Table 2. The two allocation indices in each row of Table 2 are X and Y.

Combination	484-Tone RU Allocation Index	Number of Users Per Content Channel	
		Content Channel 1	Content Channel 2
A	[200 202]	1	3
B	[201 201]	2	2
C	[202 200]	3	1
D	[203 114]	4	0
E	[114 203]	0	4

Table 2

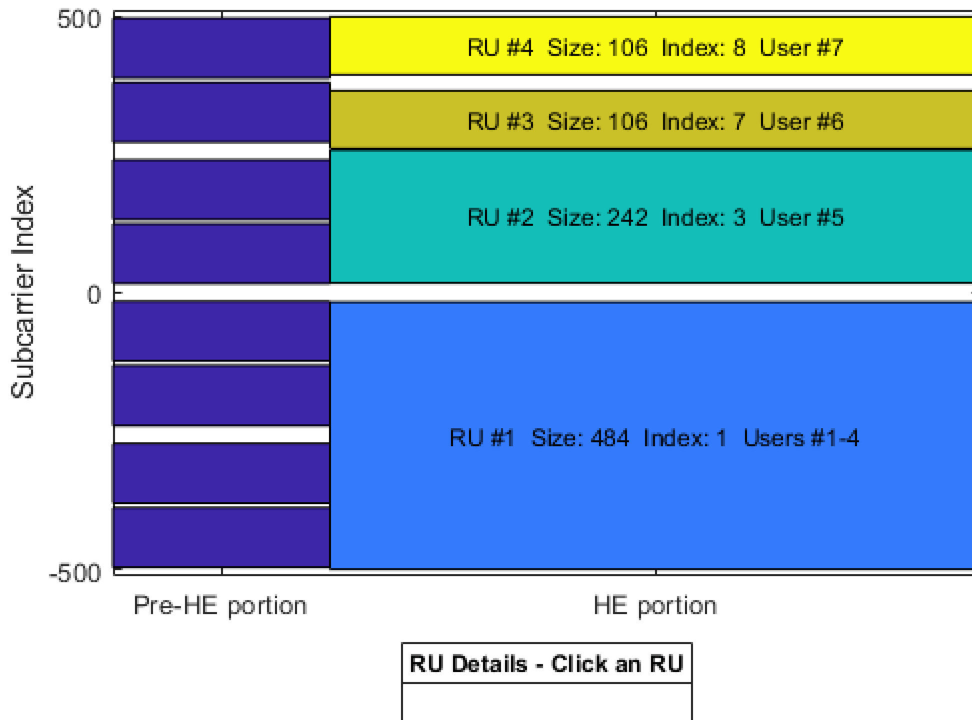
Therefore, to configure 'Combination E' the following 80 MHz allocation indices are used:

```
[114 203 192 96]
```

- 114 and 203 configure the 484-tone RU, with users #1-4.
- 192 configures a 242-tone RU with one user, user #5.

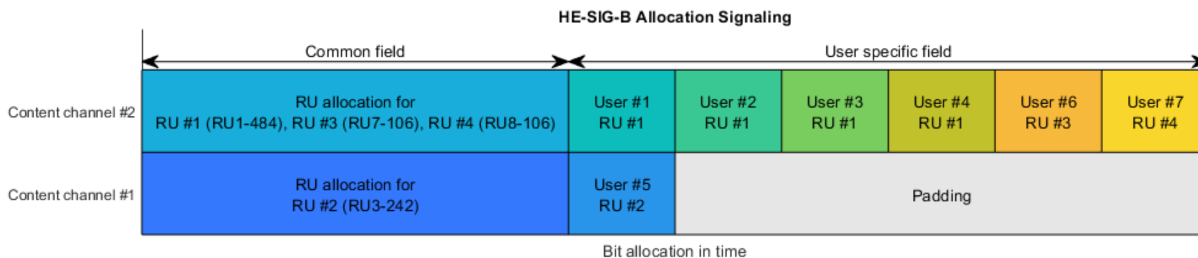
- 96 signals two 106-tone RUs, each with one user, users #6 and #7.

```
cfg484OFDMA = wlanHEMUConfig([114 203 192 96]);
showAllocation(cfg484OFDMA,axAlloc);
```



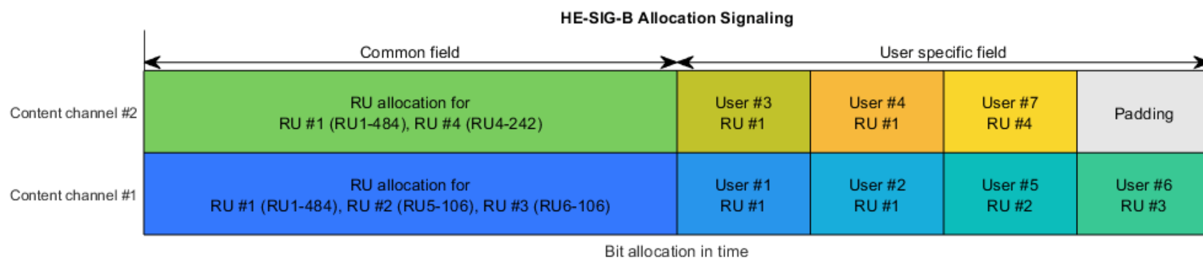
To view the HE-SIG-B allocation signaling, use the `hePlotHESIGBAllocationMapping` function. This shows the user fields signaled on each HE-SIG-B content channel, and which RU and user in the `wlanHEMUConfig` object, each user field signals. In this case we can see the users on RU #1, 3 and 4 are all signaled on content channel 2, and the user of RU #2 is signaled on content channel 1. The second content channel signals six users, while the first content channel only signals one user. Therefore, the first content channel will be padded up to the length of the second for transmission. In the diagram, the RU allocation information is provided in the form index-size, e.g. RU8-106 is the 8th 106-tone RU.

```
figure;
hePlotHESIGBAllocationMapping(cfg484OFDMA);
axSIGB = gca; % Get axis handle for subsequent plotting
```



To balance the user field signaling in HE-SIG-B, we can use 'Combination B' in Table 2 when creating the allocation index for the 484-tone RU. This results in two users being signaled on each content channel of HE-SIG-B, creating a better balance of user fields, and potentially fewer HE-SIG-B symbols in the transmission.

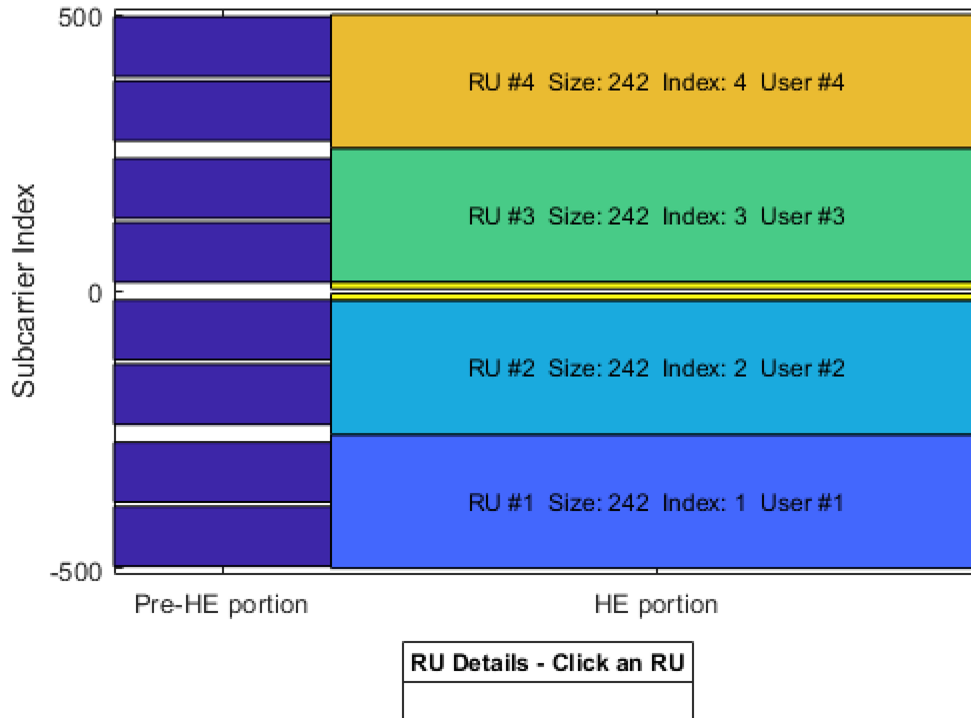
```
cfg484OFDMABalanced = wlanHEMUConfig([201 201 96 192]);
hePlotHESIGBAllocationMapping(cfg484OFDMABalanced,axSIGB);
```



HE Multi-User Format - Central 26-Tone RU

In an 80 MHz transmission, when a full band RU is not used, the central 26-tone RU can be optionally active. The central 26-tone RU is enabled using a name-value pair when creating the `wlanHEMUConfig` object.

```
% Create a configuration with no central 26-tone RU
cfgNoCentral = wlanHEMUConfig([192 192 192 192],'LowerCenter26ToneRU',false);
showAllocation(cfgNoCentral,axAlloc);

% Create a configuration with a central 26-tone RU
cfgCentral = wlanHEMUConfig([192 192 192 192],'LowerCenter26ToneRU',true);
showAllocation(cfgCentral,axAlloc);
```



Similarly, for a 160 MHz transmission, the central 26-tone RU in each 80 MHz segment can be optionally used. Each central 26-tone RU can be enabled using name-value pairs when creating the `wlanHEMUConfig` object. In this example only the upper central 26-tone RU is created. Four 242-tone RUs, each with one user are specified with the allocation index `[200 114 114 200 200 114 114 200]`.

```
cfgCentral160MHz = wlanHEMUConfig([200 114 114 200 200 114 114 200],'UpperCenter26ToneRU',true);
disp(cfgCentral160MHz)
```

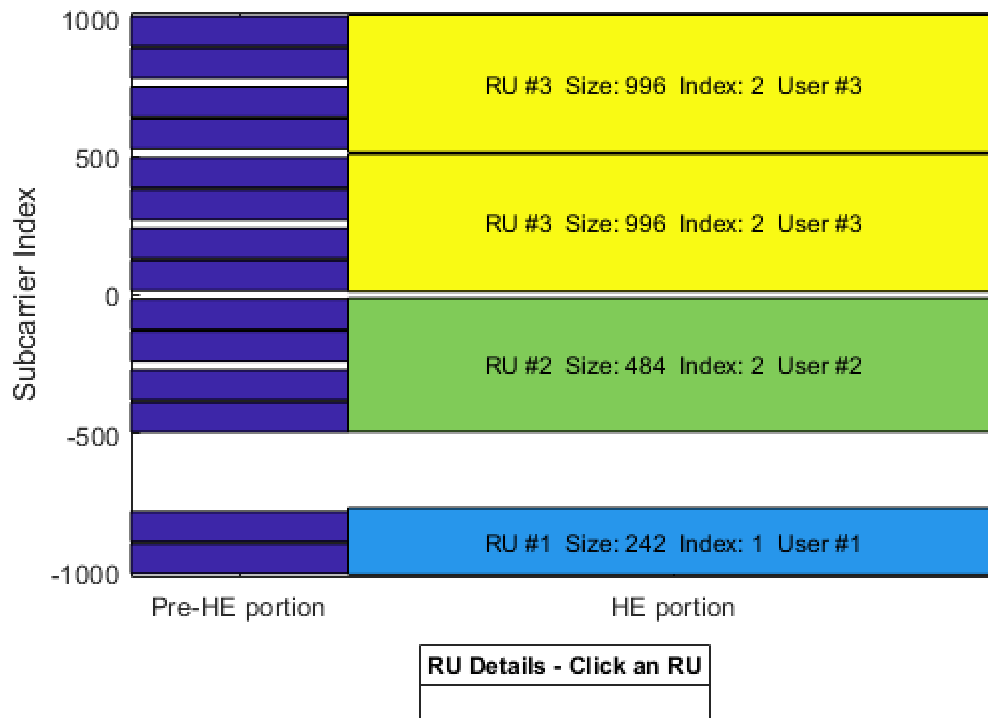
```
  wlanHEMUConfig with properties:

                     RU: {1x5 cell}
                   User: {1x5 cell}
      PrimarySubchannel: 1
    NumTransmitAntennas: 1
                   STBC: 0
          GuardInterval: 3.2000
              HELTFType: 4
                SIGBMCS: 0
                SIGBDCM: 0
        UplinkIndication: 0
               BSSColor: 0
           SpatialReuse: 0
           TXOPDuration: 127
            HighDoppler: 0

   Read-only properties:
```

```
       ChannelBandwidth: 'CBW160'
        AllocationIndex: [200 114 114 200 200 114 114 200]
    LowerCenter26ToneRU: 0
    UpperCenter26ToneRU: 1
```

HE Multi-User Format - Preamble Puncturing

In an 80 MHz or 160 MHz transmission, 20 MHz subchannels can be punctured to allow a legacy system to operate in the punctured channel. This method is also described as channel bonding. To null a 20 MHz subchannel the 20 MHz subchannel allocation index `113` can be used. The punctured 20 MHz subchannel can be viewed with the `showAllocation` method.

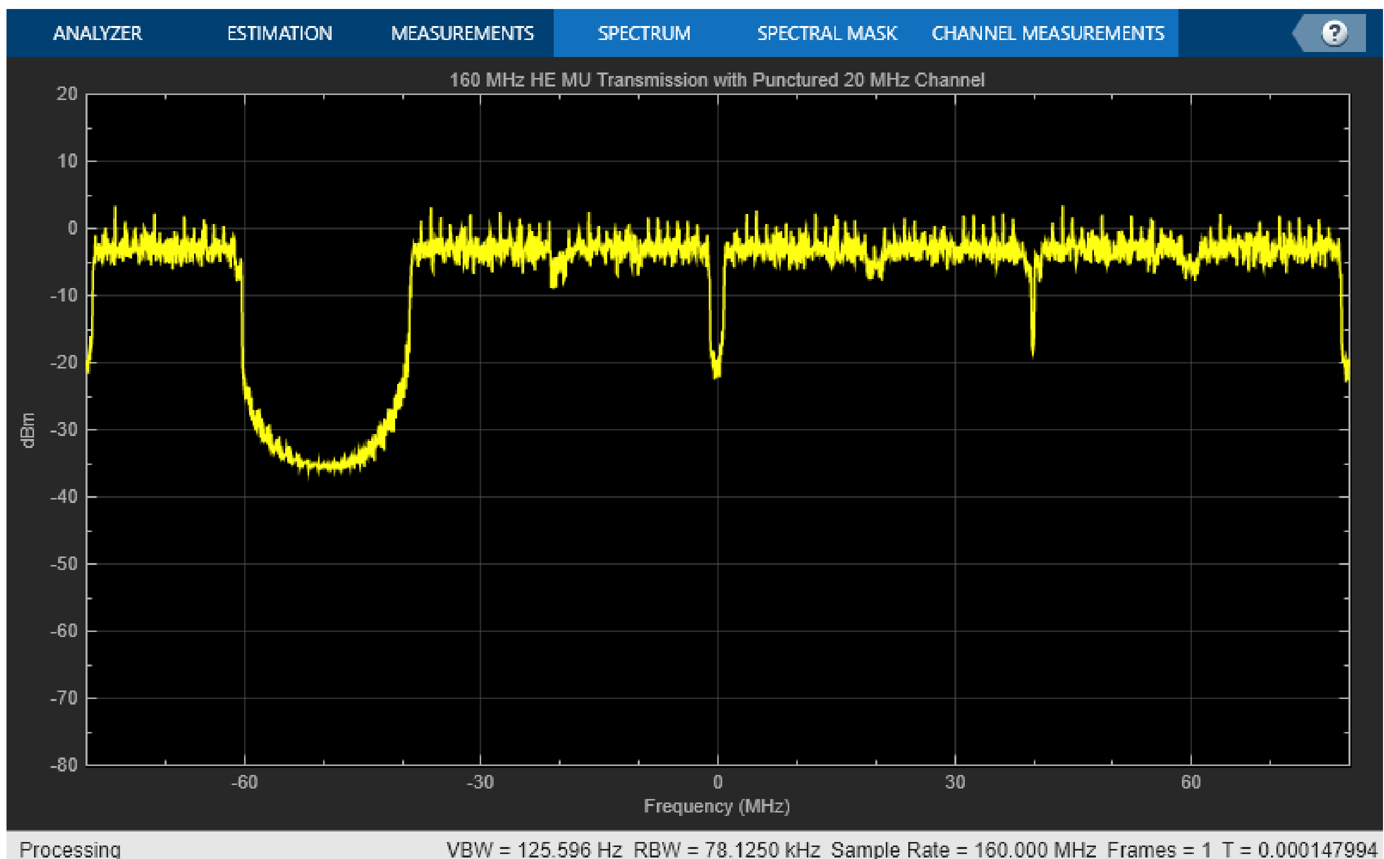
```
% Null second lowest 20 MHz subchannel in a 160 MHz configuration
cfgNull = wlanHEMUConfig([192 113 114 200 208 115 115 115]);

% Plot the allocation
showAllocation(cfgNull,axAlloc);
```



The punctured 20 MHz can also be viewed with the generated waveform and the spectrum analyzer.

```
% Set the transmission properties of each user in all RUs
cfgNull.User{1}.APEPLength = 100;
cfgNull.User{1}.MCS = 2;
cfgNull.User{1}.ChannelCoding = 'LDPC';
cfgNull.User{1}.NumSpaceTimeStreams = 1;

cfgNull.User{2}.APEPLength = 1000;
```

```
cfgNull.User{2}.MCS = 6;
cfgNull.User{2}.ChannelCoding = 'LDPC';
cfgNull.User{2}.NumSpaceTimeStreams = 1;

cfgNull.User{3}.APEPLength = 100;
cfgNull.User{3}.MCS = 1;
cfgNull.User{3}.ChannelCoding = 'LDPC';
cfgNull.User{3}.NumSpaceTimeStreams = 1;

% Create packet
txNullWaveform = wlanWaveformGenerator([1 0 1 0],cfgNull);

% Visualize signal spectrum
fs = wlanSampleRate(cfgNull);
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgNull,1);
fftsize = ofdmInfo.FFTLength;
spectrumScope = spectrumAnalyzer(SampleRate=fs,...
            AveragingMethod='exponential',ForgettingFactor=0.99,...
            RBWSource='property',RBW=fs/fftsize,...
            Title='160 MHz HE MU Transmission with Punctured 20 MHz Channel');
spectrumScope(txNullWaveform);
```



Trigger-Based MU Format

The HE trigger-based (TB) format allows for OFDMA or MU-MIMO transmission in the uplink. Each station (STA) transmits a TB packet simultaneously, when triggered by the access point (AP). A TB transmission is controlled entirely by the AP. All the parameters required for the transmission are

provided in a trigger frame to all STAs participating in the TB transmission. In this example a TB transmission in response to a trigger frame for three users in an OFDMA/MU-MIMO system is configured; three STAs will transmit simultaneously to an AP.

The 20 MHz allocation `97` is used which corresponds to two RUs, one of which serves two users in MU-MIMO.

```
disp('Allocation #97 table entry:')
disp(allocationTable(98,:)) % Index 97 (row 98)
```

```
Allocation #97 table entry:
    Allocation    BitAllocation    NumUsers    NumRUs    RUIndices      RUSizes       NumUsersPerI
    __________    _____________    ________    ______    _________    ___________    ____________

        97         "01100001"         3          2       {[1 2]}     {[106 106]}       {[1 2]}
```

The allocation information is obtained by creating a MU configuration with `wlanHEMUConfig`.

```
% Generate an OFDMA allocation
cfgMU = wlanHEMUConfig(97);
allocationInfo = ruInfo(cfgMU);
```

In a TB transmission several parameters are the same for all users in the transmission. Some of these are specified below:

```
% These parameters are the same for all users in the OFDMA system
trgMethod = 'TriggerFrame'; % Method used to trigger an HE TB PPDU
channelBandwidth = cfgMU.ChannelBandwidth; % Bandwidth of OFDMA system
lsigLength = 142;          % L-SIG length
preFECPaddingFactor = 2;   % Pre-FEC padding factor
ldpcExtraSymbol = false;   % LDPC extra symbol
numHELTFSymbols = 2;       % Number of HE-LTF symbols
```

A TB transmission for a single user within the system is configured with a `wlanHETBConfig` object. In this example, a cell array of three objects is created to describe the transmission of the three users.

```
% Create a trigger configuration for each user
numUsers = allocationInfo.NumUsers;
cfgTriggerUser = repmat({wlanHETBConfig},1,numUsers);
```

The non-default system-wide properties are set for each user.

```
for userIdx = 1:numUsers
    cfgTriggerUser{userIdx}.TriggerMethod = trgMethod;
    cfgTriggerUser{userIdx}.ChannelBandwidth = channelBandwidth;
    cfgTriggerUser{userIdx}.LSIGLength = lsigLength;
    cfgTriggerUser{userIdx}.PreFECPaddingFactor = preFECPaddingFactor;
    cfgTriggerUser{userIdx}.LDPCExtraSymbol = ldpcExtraSymbol;
    cfgTriggerUser{userIdx}.NumHELTFSymbols = numHELTFSymbols;
end
```

Next the per-user properties are set. When multiple users are transmitting in the same RU, in a MU-MIMO configuration, each user must transmit on different space-time stream indices. The properties `StartingSpaceTimeStream` and `NumSpaceTimeStreamSteams` must be set for each user to make sure different space-time streams are used. In this example user 1 and 2 are in a MU-MIMO

configuration, therefore `StartingSpaceTimeStream` for user two is set to `2`, as user one is configured to transmit 1 space-time stream with `StartingSpaceTimeStream = 1`.

```
% These parameters are for the first user - RU#1 MU-MIMO user 1
cfgTriggerUser{1}.RUSize = allocationInfo.RUSizes(1);
cfgTriggerUser{1}.RUIndex = allocationInfo.RUIndices(1);
cfgTriggerUser{1}.MCS = 4;                        % Modulation and coding scheme
cfgTriggerUser{1}.NumSpaceTimeStreams = 1;     % Number of space-time streams
cfgTriggerUser{1}.NumTransmitAntennas = 1;     % Number of transmit antennas
cfgTriggerUser{1}.StartingSpaceTimeStream = 1; % The starting index of the space-time streams
cfgTriggerUser{1}.ChannelCoding = 'LDPC';      % Channel coding

% These parameters are for the second user - RU#1 MU-MIMO user 2
cfgTriggerUser{2}.RUSize = allocationInfo.RUSizes(1);
cfgTriggerUser{2}.RUIndex = allocationInfo.RUIndices(1);
cfgTriggerUser{2}.MCS = 3;                        % Modulation and coding scheme
cfgTriggerUser{2}.NumSpaceTimeStreams = 1;     % Number of space-time streams
cfgTriggerUser{2}.StartingSpaceTimeStream = 2; % The starting index of the space-time streams
cfgTriggerUser{2}.NumTransmitAntennas = 1;     % Number of transmit antennas
cfgTriggerUser{2}.ChannelCoding = 'LDPC';      % Channel coding

% These parameters are for the third user - RU#2
cfgTriggerUser{3}.RUSize = allocationInfo.RUSizes(2);
cfgTriggerUser{3}.RUIndex = allocationInfo.RUIndices(2);
cfgTriggerUser{3}.MCS = 4;                        % Modulation and coding scheme
cfgTriggerUser{3}.NumSpaceTimeStreams = 2;     % Number of space-time streams
cfgTriggerUser{3}.StartingSpaceTimeStream = 1; % The starting index of the space-time streams
cfgTriggerUser{3}.NumTransmitAntennas = 2;     % Number of transmit antennas
cfgTriggerUser{3}.ChannelCoding = 'BCC';       % Channel coding
```

A packet containing random data is now transmitted by each user with `wlanWaveformGenerator`. The waveform transmitted by each user is stored for analysis.

```
trigInd = wlanFieldIndices(cfgTriggerUser{1}); % Get the indices of each field
txTrigStore = zeros(trigInd.HEData(2),numUsers);
for userIdx = 1:numUsers
    % Generate waveform for a user
    cfgTrigger = cfgTriggerUser{userIdx};
    txPSDU = randi([0 1],getPSDULength(cfgTrigger)*8,1);
    txTrig = wlanWaveformGenerator(txPSDU,cfgTrigger);

    % Store the transmitted STA waveform for analysis
    txTrigStore(:,userIdx) = sum(txTrig,2);
end
```

The spectrum of the transmitted waveform from each STA shows the different portions of the spectrum used, and the overlap in the MU-MIMO RU.

```
fs = wlanSampleRate(cfgTriggerUser{1});
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgTriggerUser{1});
spectrumScope = spectrumAnalyzer(SampleRate=fs,...
            Method='welch',...
            AveragingMethod='exponential',ForgettingFactor=0,...
            ChannelNames={'RU#1 User 1','RU#1 User 2','RU#2'},...
            ShowLegend=true,Title='Transmitted HE TB Waveform per User');
spectrumScope(txTrigStore);
```

Appendix

The RU allocation table for allocations <= 20 MHz is shown below, with annotated descriptions.

Allocation Index	20 MHz Subchannel Resource Unit (RU) Assignment								
0	26	26	26	26	26	26	26	26	26
1	26	26	26	26	26	26	26	52	
2	26	26	26	26	26	52		26	26
3	26	26	26	26	26	52		52	
4	26	26	52		26	26	26	26	26
5	26	26	52		26	26	26	52	
6	26	26	52		26	52		26	26
7	26	26	52		26	52		52	
8	52		26	26	26	26	26	26	26
9	52		26	26	26	26	26	52	
10	52		26	26	26	52		26	26
11	52		26	26	26	52		52	
12	52		52		26	26	26	26	26
13	52		52		26	26	26	52	
14	52		52		26	52		26	26
15	52		52		26	52		52	
16-23 (15 + N)	52		52		-	106 (N users)			
24-31 (23 + N)	106 (N users)				-	52		52	
32-39 (31 + N)	26	26	26	26	26	106 (N users)			
40-47 (39 + N)	26	26	52		26	106 (N users)			
48-55 (47 + N)	52		26	26	26	106 (N users)			
56-63 (55 + N)	52		52		26	106 (N users)			
64-71 (63 + N)	106 (N users)				26	26	26	26	26
72-79 (71 + N)	106 (N users)				26	26	26	52	
80-87 (79 + N)	106 (N users)				26	52		26	26
88-95 (87 + N)	106 (N users)				26	52		52	
96-99 (95 + M)	106				-	106 (M users)			
100-103 (99 + M)	106 (2 users)				-	106 (M users)			
104-107 (103 + M)	106 (3 users)				-	106 (M users)			
108-111 (107 + M)	106 (4 users)				-	106 (M users)			
112	52		52		-	52		52	
113	Empty 242-tone RU - No user assigned								
116-127	Reserved								
128-135 (127 + N)	106				26	106 (N users)			
136-143 (135 + N)	106 (2 users)				26	106 (N users)			
144-151 (143 + N)	106 (3 users)				26	106 (N users)			
152-159 (151 + N)	106 (4 users)				26	106 (N users)			
160-167 (159 + N)	106 (5 users)				26	106 (N users)			
168-175 (167 + N)	106 (6 users)				26	106 (N users)			
176-183 (175 + N)	106 (7 users)				26	106 (N users)			
184-191 (183 + N)	106 (8 users)				26	106 (N users)			
192-199 (191 + N)	242 (N users)								

26 tone RU assigned to 1 user as part of a 20 MHz subchannel assignment of 9 26-tone RUs

No users assigned to this RU; no data field transmitted on these subcarriers

The number of users (N) assigned to this 106-tone RU depends on the allocation index and must be 1-8.

The number of users (M) assigned to this 106-tone RU depends on the allocation index and must be 1-4.

The number of users assigned to the upper 106-tone RU depends on the allocation index, but 2 users are always assigned to the lower 106-tone RU

If selected, this 20 MHz subchannel is unused; the subchannel is punctured

RU assigned to 1 user

RU assigned to 1-4/8 users, depending on the allocation index

RU assigned to specified number of users, irrespective of the allocation index

The RU allocation and HE-SIG-B user signaling for allocations > 20 MHz is shown in the table below, with annotated descriptions.

Allocation Index	RU Allocation & Number of Users on the Corresponding HE-SIG-B Content Channel for RU Size > 242
114	484-tone RU with no users signaled on the corresponding HE-SIG-B content channel
115	996-tone RU with no users signaled on the corresponding HE-SIG-B content channel
200-207 (199 + N)	Full band 40 MHz (N users), or 484-tone RU with N users signaled in the corresponding HE-SIG-B content channel
208-215 (207 + N)	Full band 80 MHz (N users), or 996-tone RU with N users signaled in the corresponding HE-SIG-B content channel
216-223 (215 + N)	Full band 160 MHz (N users)
224-255	Reserved

>242-tone RU allocation

Must be used with other allocation indices. Signifies a 484-tone RU with zero users signaled on the corresponding HE-SIG-B content channel

A single allocation index between 200-207 configures a full-band 40 MHz 484-tone RU with N users. N must be 1-8.

Selected Bibliography

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

Basic WLAN Link Modeling

This example shows how to create a basic WLAN link model using WLAN Toolbox™. An IEEE® 802.11™ [1] VHT packet is created, passed through a TGac channel. The received signal is equalized and decoded in order to recover the transmitted bits.

Introduction

This example shows how a simple transmitter-channel-receiver simulation may be created using functions from WLAN Toolbox. A VHT transmit and receive link is implemented as shown in the figure below. A VHT packet is transmitted through a TGac channel, demodulated and the equalized symbols are recovered. The equalized symbols are decoded to recover the transmitted bits.

Waveform Generation

An 802.11ac VHT transmission is simulated in this example. The transmit parameters for the VHT format of the 802.11™ standard are configured using a VHT configuration object. The `wlanVHTConfig` creates a VHT configuration object. In this example the object is configured for a 20 MHz channel bandwidth, MCS 5 and single transmit antenna.

```
% Create a format configuration object for a SISO VHT transmission
cfgVHT = wlanVHTConfig;
cfgVHT.NumTransmitAntennas = 1;    % Transmit antennas
cfgVHT.NumSpaceTimeStreams = 1;    % Space-time streams
cfgVHT.APEPLength = 4096;          % APEP length in bytes
cfgVHT.MCS = 5;                    % Single spatial stream, 64-QAM
cfgVHT.ChannelBandwidth = 'CBW20'; % Transmitted signal bandwidth
Rs = wlanSampleRate(cfgVHT);       % Sampling rate
```

A single VHT packet is generated consisting of training, signal and data fields:

- Non-HT Short Training Field (L-STF)
- Non-HT Long Training Field (L-LTF)
- Non-HT Signal (L-SIG) field
- VHT Signal A (VHT-SIG-A) field
- VHT Short Training Field (VHT-STF)
- VHT Long Training Field (VHT-LTF)
- VHT Signal B (VHT-SIG-B) field
- Data field

These fields are generated separately using functions from WLAN Toolbox and are concatenated to produce a VHT transmit packet.

The first field in the PPDU is the L-STF and is used for the start of packet detection and automatic gain control (AGC) setting. It is also used for initial frequency offset estimation and coarse timing

synchronization. The `wlanLSTF` function generates the L-STF field in the time-domain using some of the parameters included in configuration object `cfgVHT`.

```
lstf = wlanLSTF(cfgVHT);
```

The L-LTF is used for fine time synchronization, channel estimation and fine frequency offset estimation. The `wlanLLTF` function generates the L-LTF in the time-domain.

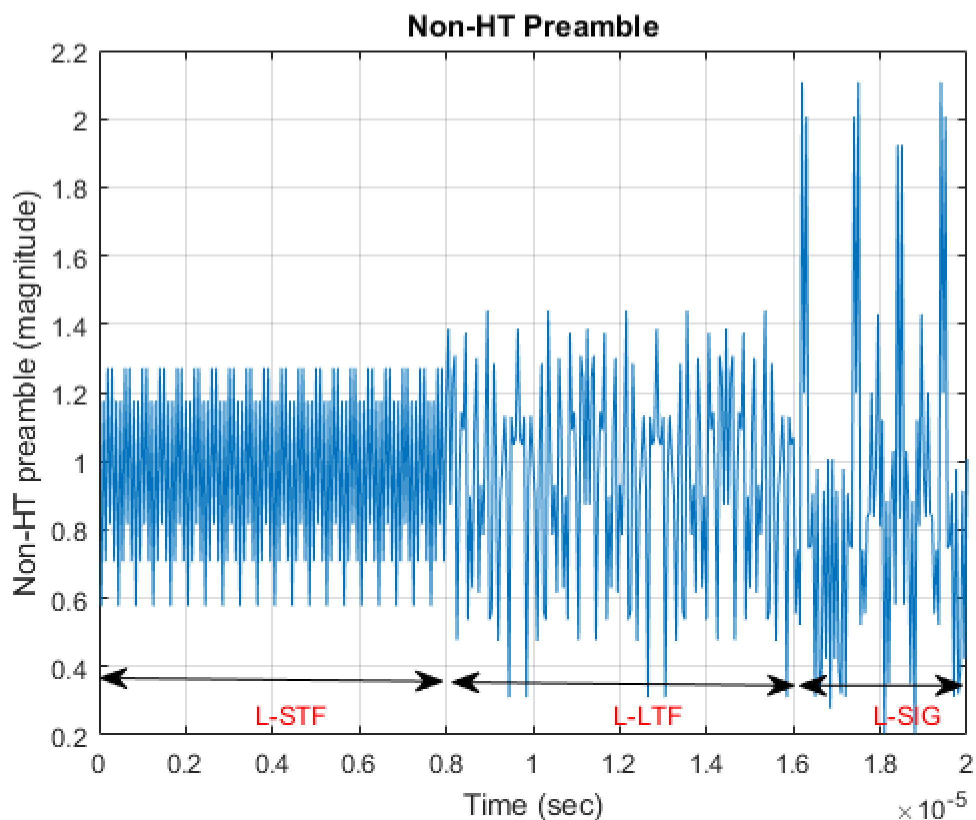
```
lltf = wlanLLTF(cfgVHT);
```

The L-SIG field carries packet configuration such as data rate, modulation and code rate for non-HT format. The `wlanLSIG` function generates the L-SIG field in the time-domain.

```
lsig = wlanLSIG(cfgVHT);
```

The figure below shows the L-STF, L-LTF and L-SIG fields. These fields are common to the VHT, HT-Mixed and non-HT OFDM transmission formats.

```
nonHTfield = [lstf;lltf;lsig]; % Combine the non-HT preamble fields
```



The VHT specific signal and training fields are generated after the non-HT preamble fields. The purpose of the VHT-SIG-A field is to provide information to allow the receiver to decode the data payload. The VHT-SIG-A is composed of two symbols VHT-SIG-A1 and VHT-SIG-A2. The `wlanVHTSIGA` function generates the VHT-SIG-A field in the time-domain.

```
vhtsiga = wlanVHTSIGA(cfgVHT);
```

The purpose of the VHT-STF is to improve the gain control estimation in a MIMO transmission and help the receiver detect the repeating pattern similar to the L-STF field. The `wlanVHTSTF` function generates the VHT-STF field in the time-domain.

```
vhtstf = wlanVHTSTF(cfgVHT);
```

The VHT-LTF provides a mean for the receiver to estimate the channel between the transmitter and the receiver. Depending on the number of space time streams, it consists of 1,2,4,6 or 8 VHT-LTF symbols. The `wlanVHTLTF` function generates the VHT-LTF in the time-domain.

```
vhtltf = wlanVHTLTF(cfgVHT);
```

The VHT-SIG-B field is used to set the data rate and the length of the data field payload of the transmitted packet. The `wlanVHTSIGB` function generates the VHT-SIG-B field in the time-domain.

```
vhtsigb = wlanVHTSIGB(cfgVHT);
```

Construct the preamble with the generated signal and training fields for the VHT format.

```
preamble = [lstf;lltf;lsig;vhtsiga;vhtstf;vhtltf;vhtsigb];
```

The `wlanVHTData` function generates the time-domain VHT data field. The VHT format configuration `cfgVHT` specifies the parameters for generating the data field from the PSDU bits. The `cfgVHT.PSDULength` property gives the number of bytes to be transmitted in the VHT data field. This property is used to generate the random PSDU bits `txPSDU`.

```
rng(0) % Initialize the random number generator
txPSDU = randi([0 1],cfgVHT.PSDULength*8,1); % Generate PSDU data in bits
data = wlanVHTData(txPSDU,cfgVHT);

% A VHT waveform is constructed by prepending the non-HT and VHT
% preamble fields with data
txWaveform = [preamble;data]; % Transmit VHT PPDU
```

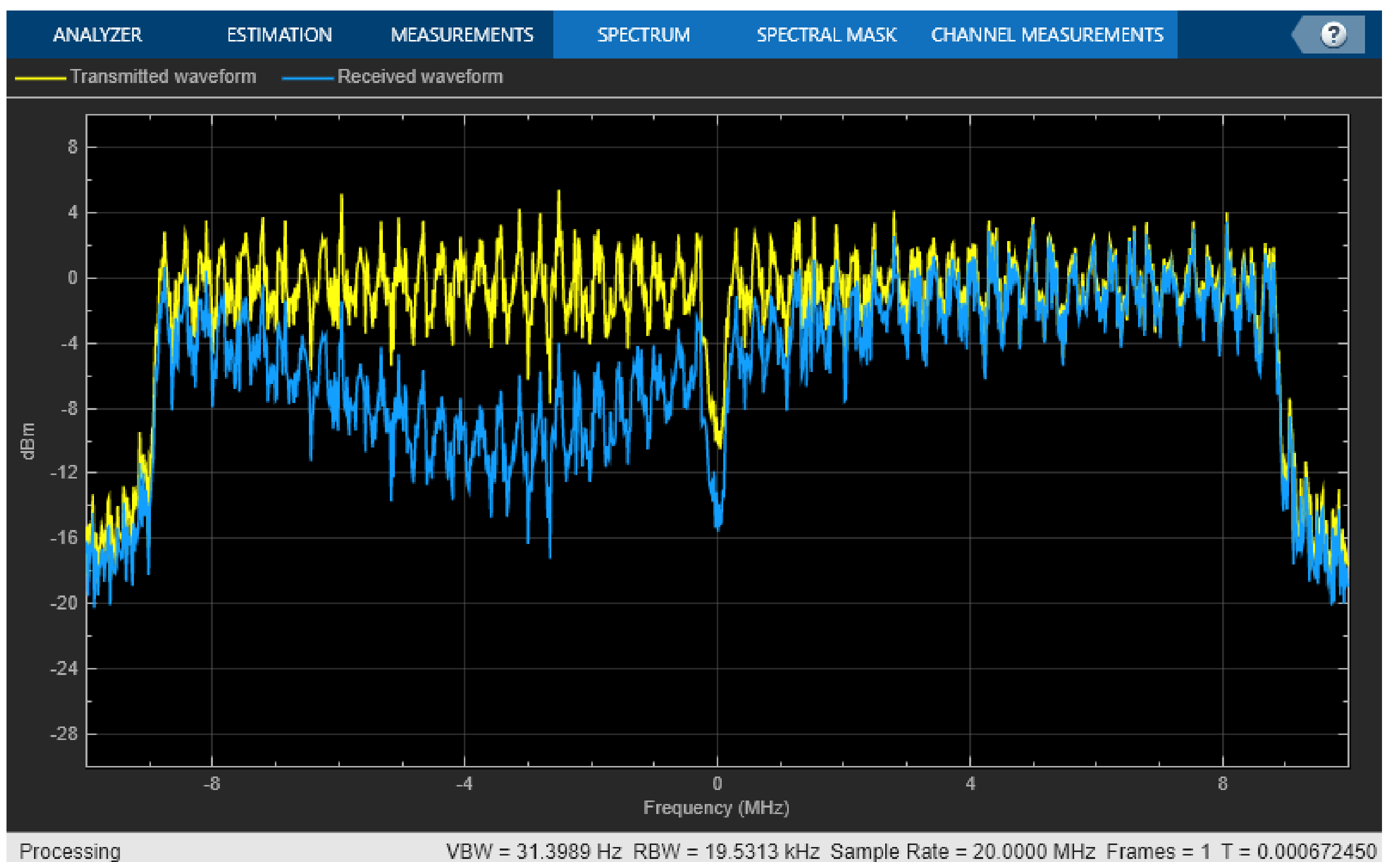
Alternatively the waveform for a given format configuration can also be generated using a single function call `wlanWaveformGenerator` function. This function can produce one or more VHT packets. By default OFDM windowing is applied to the generated waveform. For more information on OFDM windowing, see the reference page for the `wlanWaveformGenerator` function.

Channel Impairments

This section simulates the effects of over-the-air transmission. The transmitted signal is impaired by the channel and AWGN. The level of the AWGN is given in dBs. In this example the TGac channel model [2] is used with delay profile Model-B. For this delay profile when the distance between transmitter and receiver is greater than or equal to 5 meters, the model is in Non-Line-of-Sight (N-LOS) configuration. This is described further in the help for `wlanTGacChannel`.

```
% Parameterize the channel
tgacChannel = wlanTGacChannel;
tgacChannel.DelayProfile = 'Model-B';
tgacChannel.NumTransmitAntennas = cfgVHT.NumTransmitAntennas;
tgacChannel.NumReceiveAntennas = 1;
tgacChannel.LargeScaleFadingEffect = 'None';
tgacChannel.ChannelBandwidth = 'CBW20';
tgacChannel.TransmitReceiveDistance = 5;
tgacChannel.SampleRate = Rs;
tgacChannel.RandomStream = 'mt19937ar with seed';
```

```
tgacChannel.Seed = 10;

% Pass signal through the channel. Append zeroes to compensate for channel
% filter delay
txWaveform = [txWaveform;zeros(10,1)];
chanOut = tgacChannel(txWaveform);

snr = 40; % In dBs
rxWaveform = awgn(chanOut,snr,0);

% Display the spectrum of the transmitted and received signals. The
% received signal spectrum is affected by the channel
spectrumScope  = spectrumAnalyzer(SampleRate=Rs, ...
            AveragingMethod='exponential',ForgettingFactor=0.99, ...
            YLimits=[-30 10],ShowLegend=true, ...
            ChannelNames={'Transmitted waveform','Received waveform'});
spectrumScope([txWaveform rxWaveform]);
```



Channel Estimation and Equalization

In this section the time-domain VHT-LTF is extracted from the received waveform. The waveform is assumed to be synchronized to the start of the packet by taking the channel filter delay into account. The VHT-LTF is demodulated and is used to estimate the channel. The received signal is then equalized using the channel estimate obtained from the VHT-LTF.

In this example the received signal is synchronized to the start of the packet by compensating for a known channel filter delay. For more information on how to automatically detect and synchronize to the received signal see the following examples:

- "802.11n Packet Error Rate Simulation for 2x2 TGn Channel" on page 5-16
- "802.11ac Packet Error Rate Simulation for 8x8 TGac Channel" on page 5-10

```
chInfo = info(tgacChannel); % Get characteristic information
% Channel filter delay, measured in samples
chDelay  = chInfo.ChannelFilterDelay;
rxWaveform = rxWaveform(chDelay+1:end,:);
```

After synchronization the receiver has to extract the relevant fields from the received packet. The `wlanFieldIndices` function is used to return the start and end time-domain sample indices of all fields relative to the first sample in a packet. These indices are used to extract the required fields for further processing.

```
indField = wlanFieldIndices(cfgVHT);
```

To extract the VHT-LTF from the received signal the start and end indices are used to generate a vector of indices.

```
indVHTLTF = indField.VHTLTF(1):indField.VHTLTF(2);
```

The VHT-LTF is used to estimate the channel between all space-time streams and receive antennas. The VHT-LTF is extracted from the received waveform and is demodulated using the `wlanVHTLTFDemodulate` function.

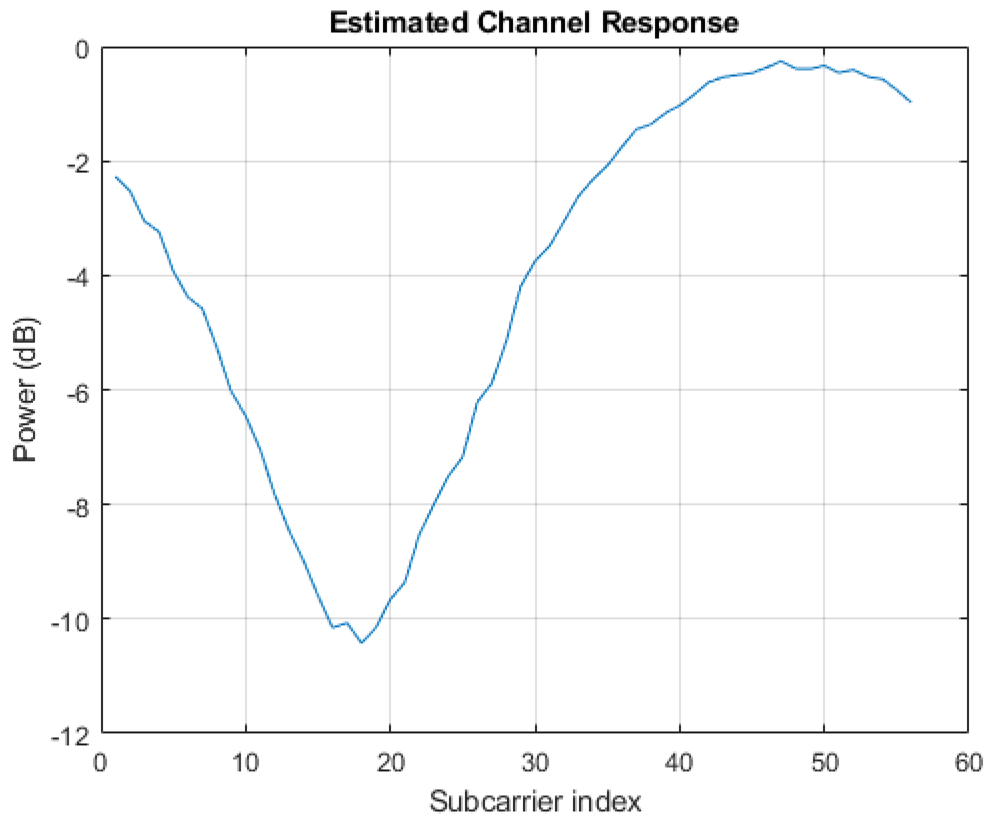
```
demodVHTLTF = wlanVHTLTFDemodulate(rxWaveform(indVHTLTF,:),cfgVHT);
```

The channel estimate includes the effect of the applied spatial mapping and cyclic shifts at the transmitter for a multi antenna configuration. The `wlanVHTLTFChannelEstimate` function returns the estimated channel between all space-time streams and receive antennas.

```
chanEstVHTLTF = wlanVHTLTFChannelEstimate(demodVHTLTF,cfgVHT);
```

The transmit signal encounters a deep fade as shown in the channel frequency response in the figure below. The effect of channel fades can also be seen in the spectrum plot shown previously.

```
figure
plot(20*log10(abs(chanEstVHTLTF)));
grid on;
title('Estimated Channel Response');
xlabel('Subcarrier index');
ylabel('Power (dB)');
```



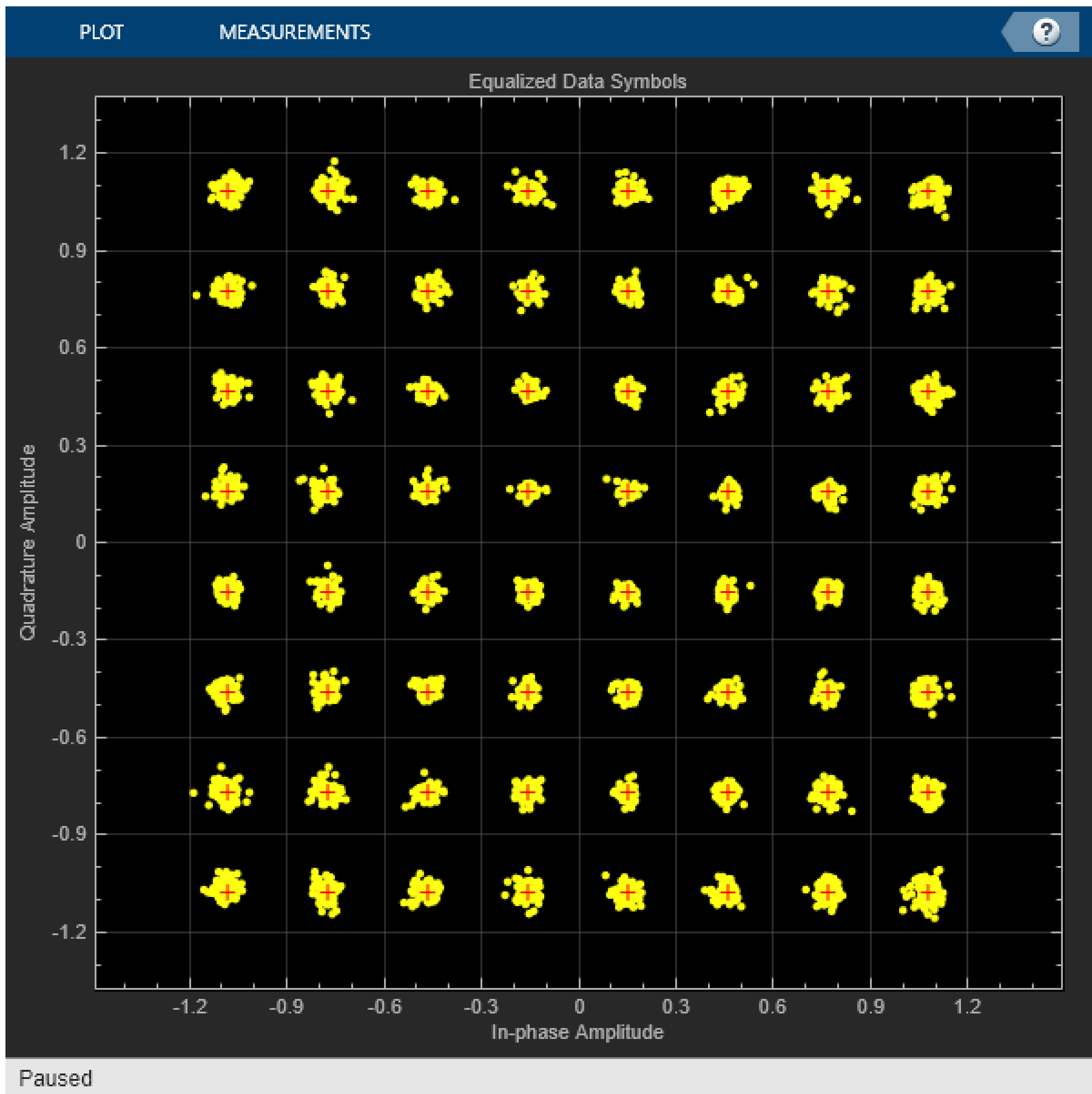
To extract the data field from the received signal the start and end indices for the data field are used to generate a vector of indices.

```
indData = indField.VHTData(1):indField.VHTData(2);

% Get channel estimates at pilot subcarriers
ofdmInfo = wlanVHTOFDMInfo("VHT-Data",cfgVHT.ChannelBandwidth,cfgVHT.GuardInterval);
chanEstPilots = chanEstVHTLTF(ofdmInfo.PilotIndices);

% Estimate the noise power in VHT data field
nVar = vhtNoiseEstimate(rxWaveform(indData,:),chanEstPilots,cfgVHT);

% Recover the bits and equalized symbols in the VHT Data field using the
% channel estimates from VHT-LTF
[rxPSDU,~,eqSym] = wlanVHTDataRecover(rxWaveform(indData,:),chanEstVHTLTF,nVar,cfgVHT);

% Compare transmit and receive PSDU bits
numErr = biterr(txPSDU,rxPSDU);
```

The following plot shows the constellation of the equalized symbols at the output of the `wlanVHTDataRecover` function compared against the reference constellation. Increasing the channel noise should begin to spread the distinct constellation points.

```
% Plot equalized symbols
constellationDiagram = comm.ConstellationDiagram;
constellationDiagram.ReferenceConstellation = wlanReferenceSymbols(cfgVHT);
% Compare received and reference constellation
```

```
constellationDiagram(reshape(eqSym,[],1));
constellationDiagram.Title = 'Equalized Data Symbols';
```



Selected Bibliography

1 IEEE Std 802.11™-2020. 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.

2 Breit, G., H. Sampath, S. Vermani, et al. TGac Channel Model Addendum. Version 12. IEEE 802.11-09/0308r12, March 2010.

802.11ac Multi-User MIMO Precoding

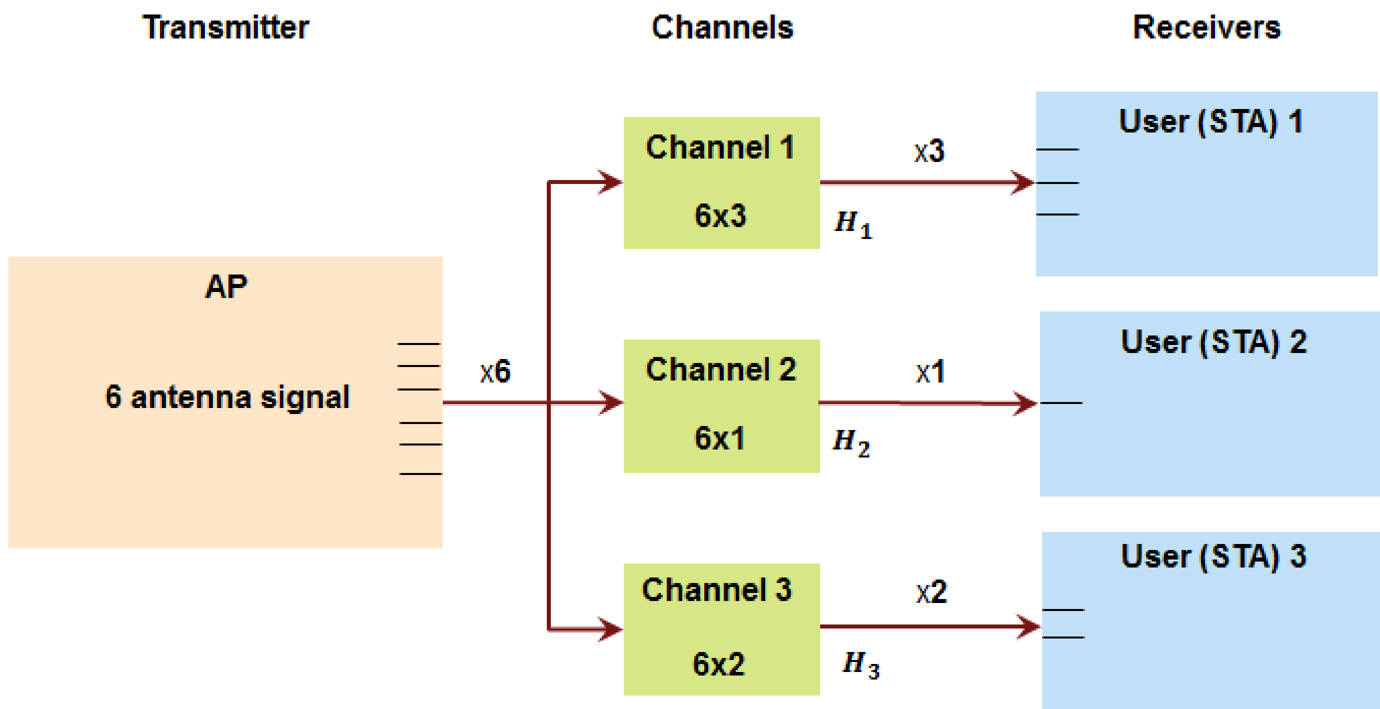
This example shows the transmit and receive processing for a 802.11ac™ multi-user downlink transmission over a fading channel. The example uses linear precoding techniques based on a singular-value-decomposition (SVD) of the channel.

Introduction

802.11ac supports downlink (access-point to station) multi-user 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” on page 3-26 example, is used to determine the CSI for each user in a multi-user 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 multi-user transmit waveform, passes it through a channel per user 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 channels and determines the precoding matrix under the assumption of perfect feedback.

Simulation Parameters and Configuration

For 802.11ac, a maximum of eight spatial streams is allowed. A 6x6 MIMO configuration for three users is used in this example, where the first user has three streams, second has one, and the third has two streams allocated to it. Different rate parameters and payload sizes for up to four users are specified as vector parameters. These are indexed appropriately in the transmission configuration based on the number of active users.

```
s = rng(21);                              % Set RNG seed for repeatability

% Transmission parameters
chanBW      = 'CBW80';                    % Channel bandwidth
numUsers    = 3;                          % Number of active users
numSTSAll   = [3 1 2 2];                  % Number of streams for 4 users
userPos     = [0 1 2 3];                  % User positions for maximum 4 users
mcsVec      = [4 6 2 2];                  % MCS for maximum 4 users
apepVec     = [15120 8192 5400 6000];     % Payload, in bytes, for 4 users
chCodingVec = {'BCC', 'LDPC', 'LDPC', 'BCC'};  % Channel coding for 4 users

% Channel and receiver parameters
chanMdl       = 'Model-A';                % TGac fading channel model
precodingType = 'ZF';                     % Precoding type; ZF or MMSE
snr           = 38;                       % SNR in dB
eqMethod      = 'ZF';                     % Equalization method

% Create the multi-user VHT format configuration object, appropriately
% indexing into the vector values for the active users
if (numUsers==1)
    groupID = 0;
else
    groupID = 2;
end
numSTSVec = numSTSAll(1:numUsers);
numTx = sum(numSTSVec);
cfgVHTMU = wlanVHTConfig('ChannelBandwidth', chanBW,...
    'NumUsers', numUsers, ...
    'NumTransmitAntennas', numTx, ...
    'GroupID', groupID, ...
    'NumSpaceTimeStreams', numSTSVec,...
    'UserPositions', userPos(1:numUsers), ...
    'MCS', mcsVec(1:numUsers), ...
    'APEPLength', apepVec(1:numUsers), ...
    'ChannelCoding', chCodingVec(1:numUsers));
```

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 multi-user 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);
```

Transmission Channel

The TGac multi-user channel consists of independent single-user MIMO channels between the access point and spatially separated stations [3]. In this example, the same delay profile Model-A channel is applied for each of the users, even though individual users can experience different conditions. The flat-fading channel allows a simpler receiver without front-end synchronization. It is also assumed that each user's number of receive antennas are equal to the number of space-time streams allocated to them.

Cell arrays are used in the example to store per-user elements which allow for a flexible number of users. Here, as an example, each instance of the TGac channel per user is stored as an element of a cell array.

```
% Create three independent channels
TGAC      = cell(numUsers, 1);
chanSeeds = [1111 2222 3333 4444];  % chosen for a maximum of 4 users
uIndex    = [10 5 2 1];             % chosen for a maximum of 4 users
chanDelay = zeros(numUsers, 1);
for uIdx = 1:numUsers
    TGAC{uIdx} = wlanTGacChannel(...
        'ChannelBandwidth', cfgVHTMU.ChannelBandwidth,...
        'DelayProfile', chanMdl, ...
        'UserIndex', uIndex(uIdx), ...
        'NumTransmitAntennas', numTx, ...
        'NumReceiveAntennas', numSTSVec(uIdx), ...
        'RandomStream', 'mt19937ar with seed', ...
        'Seed', chanSeeds(uIdx),...
        'SampleRate', wlanSampleRate(cfgVHTMU), ...
        'TransmitReceiveDistance',5);
    chanInfo = info(TGAC{uIdx});
    chanDelay(uIdx) = chanInfo.ChannelFilterDelay;
end
```

The channels for each individual user use different seeds for random number generation. A different user index is specified to allow for random angle offsets to be applied to the arrival (AoA) and departure (AoD) angles for the clusters. The channel filtering delay is stored to allow for its compensation at the receiver. In practice, symbol timing estimation would be used.

```
% Append zeroes to allow for channel filter delay
txNDPSig = [txNDPSig; zeros(10, numTx)];

% Sound the independent channels per user for all transmit streams
```

```
rxNDPSig = cell(numUsers, 1);
for uIdx = 1:numUsers
    rxNDPChan = TGAC{uIdx}(txNDPSig);

    % Add WGN per receiver
    rxNDPSig{uIdx} = awgn(rxNDPChan, snr);
end
```

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);
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 multi-user 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, the data is transmitted over the fading channel.

```
% Generate the multi-user VHT waveform
txSig = wlanWaveformGenerator(psduDataBits, cfgVHTMU);

% Transmit through per-user fading channel
rxSig = cell(numUsers, 1);
for uIdx = 1:numUsers
    % Append zeroes to allow for channel filter delay
    rxSig{uIdx} = TGAC{uIdx}([txSig; zeros(10, numTx)]);
end
```

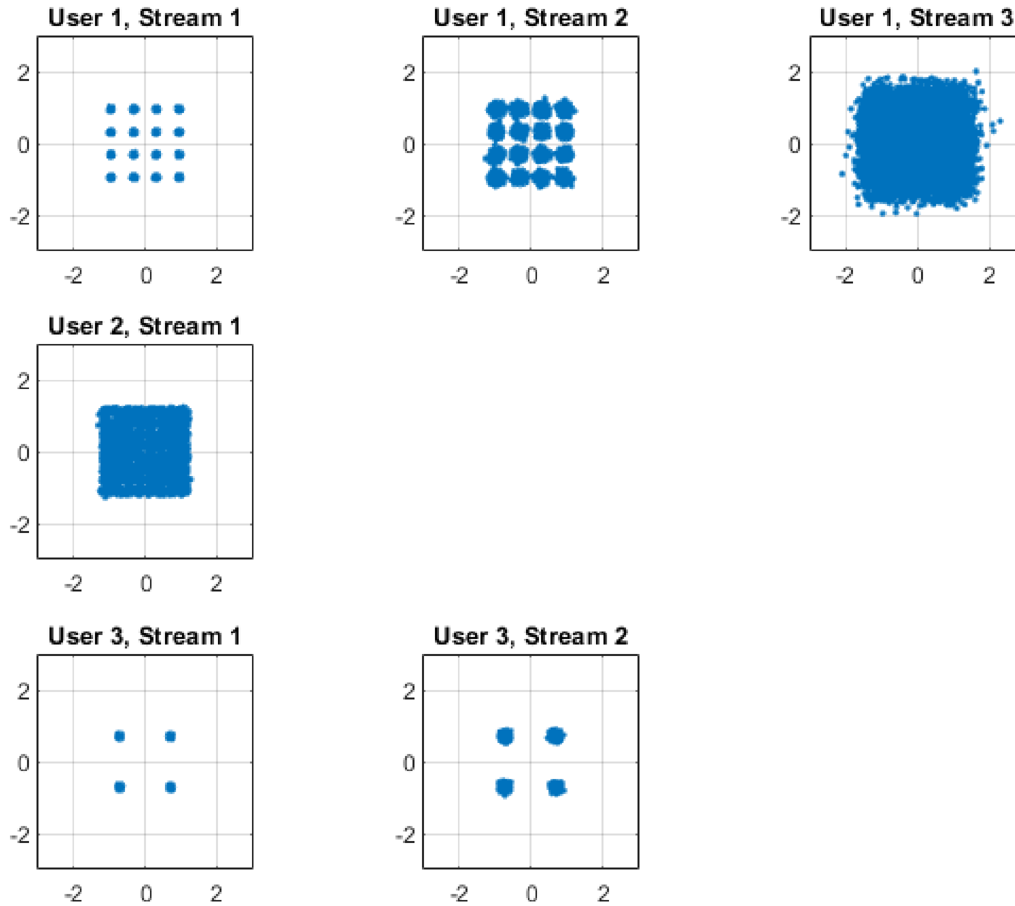
Data Recovery Per User

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
    % Add WGN per receiver
    rxNSig = awgn(rxSig{uIdx}, snr);
    rxNSig = rxNSig(chanDelay(uIdx)+1:end, :);

    % User space-time streams
    stsU = numSTSVec(uIdx);

    % Perform channel estimation based on VHT-LTF
    rxVHTLTF  = rxNSig(ind.VHTLTF(1):ind.VHTLTF(2),:);
    demodVHTLTF = wlanVHTLTFDemodulate(rxVHTLTF, chanBW, numSTSVec);
    [chanEst, chanEstSSPilots] = wlanVHTLTFChannelEstimate(demodVHTLTF, chanBW, numSTSVec);

    % Extract VHT Data samples from the waveform
```

```
        rxVHTData = rxNSig(ind.VHTData(1):ind.VHTData(2),:);

        % Estimate the noise power in VHT data field
        nVar = vhtNoiseEstimate(rxVHTData,chanEstSSPilots,cfgVHTMU);

        % Recover information bits in VHT Data field
        [rxDataBits{uIdx}, ~, eqsym] = wlanVHTDataRecover(rxVHTData, ...
            chanEst, nVar, cfgVHTMU, uIdx, 'EqualizationMethod', eqMethod, ...
            'PilotPhaseTracking', 'None', 'LDPCDecodingMethod', 'norm-min-sum');

        % 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: 0.00013228
Bit Error Rate for User 2: 0
Bit Error Rate for User 3: 0
```

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 multi-user transmit configuration, independent per-user channel modeling, and the individual receive processing using the channel inversion precoding techniques.

Further exploration includes modifications to the transmission and channel parameters, alternate precoding techniques, more realistic receivers and feedback mechanism incorporating delays and quantization.

Selected Bibliography

1. IEEE Std 802.11™-2020. 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.
2. Perahia, E., R. Stacey, "Next Generation Wireless LANS: 802.11n and 802.11ac", Cambridge University Press, 2013.
3. Breit, G., H. Sampath, S. Vermani, et al., "TGac Channel Model Addendum", Version 12. IEEE 802.11-09/0308r12, March 2010.

2

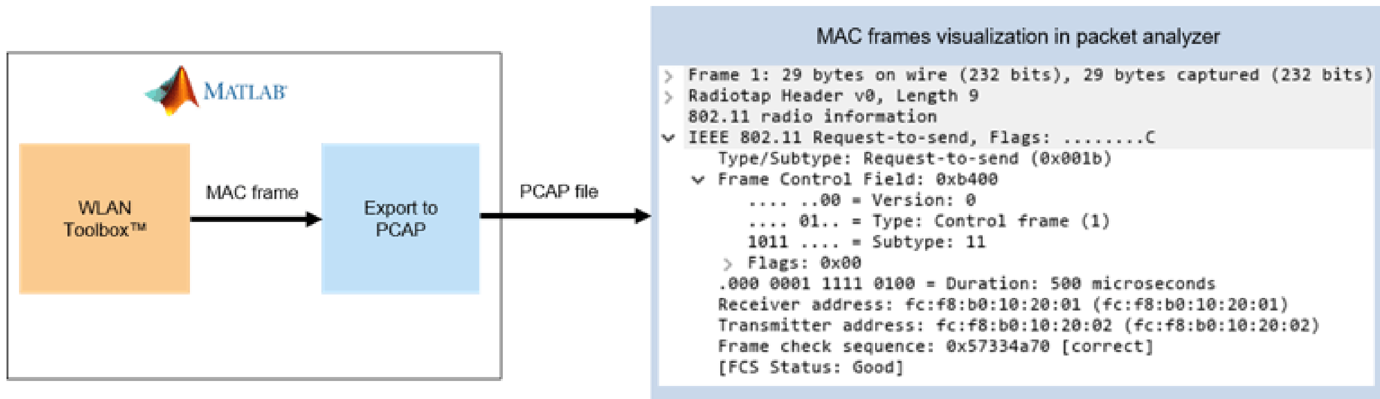
MAC Modeling

802.11 MAC Frame Generation

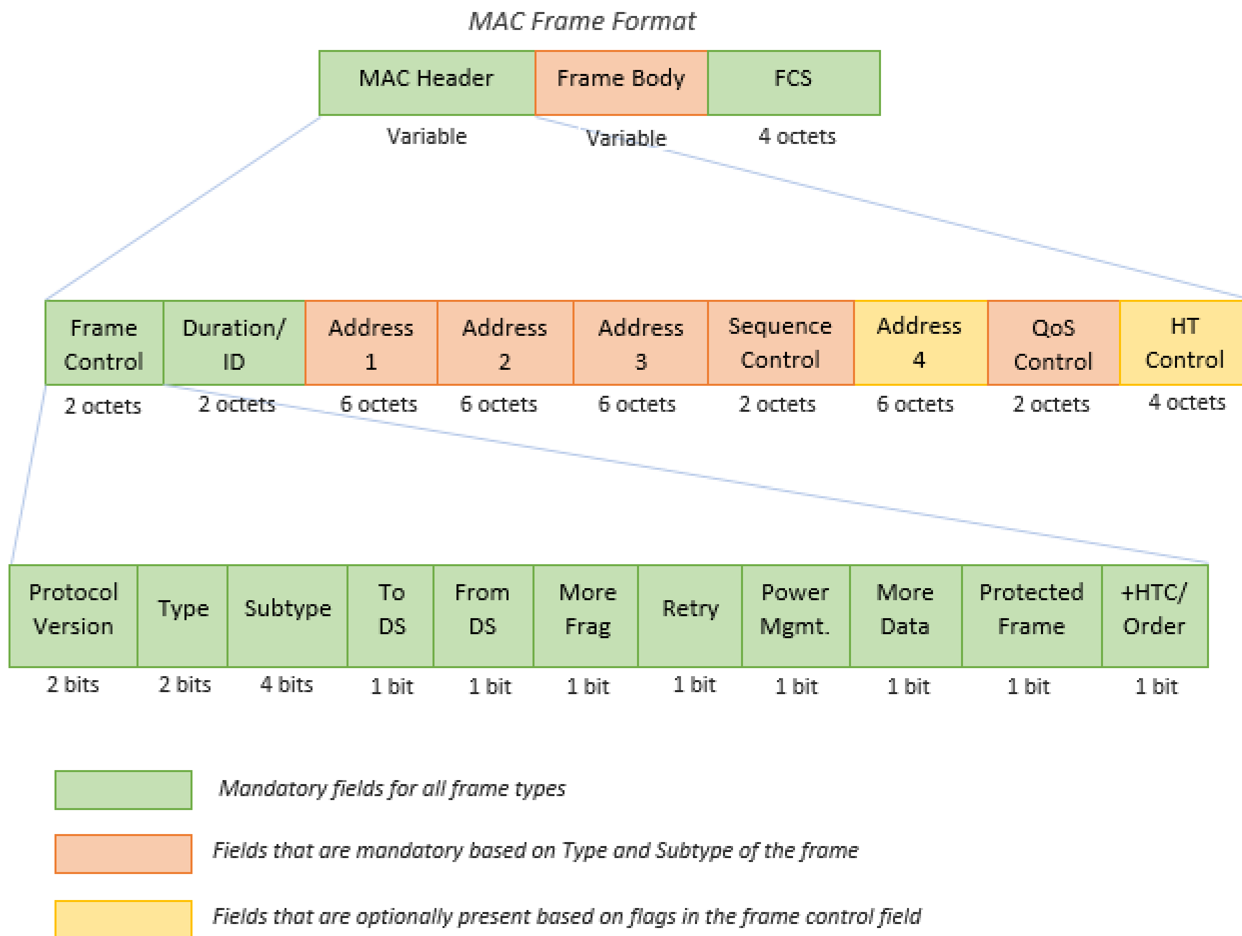
This example shows how to generate IEEE® 802.11™ MAC frames.

Introduction

This example shows how to generate WLAN MAC frames, specified in section 9 of [1] and [2], and export these frames to a packet capture (PCAP) file for analysis with third-party packet analysis tools.



The general MAC frame format consists of a header, frame-body, and frame check sequence (FCS). The header holds information about the frame. The frame-body carries data that needs to be transmitted. The transmitter calculates the FCS over the header and frame-body. The receiver uses the FCS to confirm that the header and frame-body are properly formed. This diagram shows the structure of a general MAC frame.



For more information, see the “WLAN MAC Frame Structure” topic.

You can use the `wlanMACFrame` function to generate MAC frames. This function accepts a MAC frame configuration object `wlanMACFrameConfig` as an input. This object configures the fields in the MAC header. Set the `FrameType` property to the desired *Subtype description* in Table 9-1 of [1] to set the appropriate *Type* and *Subtype* fields in the MAC header. The `wlanMACFrame` function supports the generation of these MPDUs.

- **Management Frames:** Beacon
- **Data Frames:** Data, Null, QoS Data, QoS Null
- **Control Frames:** RTS, CTS, Ack, Block Ack

In addition to these MPDUs, `wlanMACFrame` also supports generation of A-MPDUs containing MPDUs of type `QoS Data`.

Control Frame Generation

To generate an RTS frame, create a MAC frame configuration object with the `FrameType` set to `'RTS'`.

```
rtsCfg = wlanMACFrameConfig('FrameType', 'RTS');
disp(rtsCfg);
```

```
  wlanMACFrameConfig with properties:

          FrameType: 'RTS'
    PowerManagement: 0
           MoreData: 0
           Duration: 0
           Address1: 'FFFFFFFFFFFF'
           Address2: '00123456789B'

   Read-only properties:
            Decoded: 0
```

Configure the frame header fields.

```
% Duration
rtsCfg.Duration = 500;
% Receiver address
rtsCfg.Address1 = 'FCF8B0102001';
% Transmitter address
rtsCfg.Address2 = 'FCF8B0102002';
```

Generate an RTS frame using the configuration.

```
% Generate octets for an RTS frame
rtsFrame = wlanMACFrame(rtsCfg);
```

By default, the output of `wlanMACFrame` is a sequence of hexadecimal octets. If you want to generate the MAC frame as a sequence of bits, set the `OutputFormat` parameter to `bits`.

```
% Generate bits for an RTS frame
rtsFrameBits = wlanMACFrame(rtsCfg, 'OutputFormat', 'bits');
```

Data Frame Generation

To generate a QoS Data frame, create a MAC frame configuration object with the `FrameType` set to `'QoS Data'`.

```
qosDataCfg = wlanMACFrameConfig('FrameType', 'QoS Data');
disp(qosDataCfg);
```

```
  wlanMACFrameConfig with properties:

          FrameType: 'QoS Data'
        FrameFormat: 'Non-HT'
               ToDS: 0
             FromDS: 1
     Retransmission: 0
    PowerManagement: 0
           MoreData: 0
           Duration: 0
           Address1: 'FFFFFFFFFFFF'
           Address2: '00123456789B'
           Address3: '00123456789B'
     SequenceNumber: 0
                TID: 0
          AckPolicy: 'No Ack'
    MSDUAggregation: 0
               EOSP: 0
```

```
        IsMeshFrame: 0

   Read-only properties:
            Decoded: 0
```

Configure the frame header fields.

```
% From DS flag
qosDataCfg.FromDS = 1;
% To DS flag
qosDataCfg.ToDS = 0;
% Acknowledgment Policy
qosDataCfg.AckPolicy = 'Normal Ack';
% Receiver address
qosDataCfg.Address1 = 'FCF8B0102001';
% Transmitter address
qosDataCfg.Address2 = 'FCF8B0102002';
```

The QoS Data frame is used to transmit a payload from higher-layer. A 20-byte payload containing a repeating sequence of hexadecimal value '11' is used in this example.

```
payload = repmat('11', 1, 20);
```

Generate a QoS Data frame using payload and configuration.

```
% Generate octets for a QoS Data frame
qosDataFrame = wlanMACFrame(payload, qosDataCfg);
```

By default, the output of `wlanMACFrame` is a sequence of hexadecimal octets. If you want to generate the MAC frame as a sequence of bits, set the `OutputFormat` parameter to `bits`.

```
% Generate bits for a QoS Data frame
qosDataFrameBits = wlanMACFrame(payload, qosDataCfg, 'OutputFormat', 'bits');
```

The output MAC frame is an MPDU with a single MSDU. For more information about A-MSDU and A-MPDU generation, see “802.11ac Waveform Generation with MAC Frames” on page 2-18.

Management Frame Generation

To generate a Beacon frame, create a MAC frame configuration object with the `FrameType` set to `'Beacon'`.

```
beaconCfg = wlanMACFrameConfig('FrameType', 'Beacon');
disp(beaconCfg);
```

```
  wlanMACFrameConfig with properties:

           FrameType: 'Beacon'
                ToDS: 0
              FromDS: 1
      Retransmission: 0
     PowerManagement: 0
            MoreData: 0
            Duration: 0
            Address1: 'FFFFFFFFFFFF'
            Address2: '00123456789B'
            Address3: '00123456789B'
```

```
      SequenceNumber: 0
    ManagementConfig: [1x1 wlanMACManagementConfig]

   Read-only properties:
             Decoded: 0
```

Beacon frame-body consists of information fields and information elements as explained in section 9.3.3.2 of [1]. You can configure these information fields and elements using `wlanMACManagementConfig`.

```
% Create a management frame-body configuration object
frameBodyCfg = wlanMACManagementConfig;
disp(frameBodyCfg);
```

```
  wlanMACManagementConfig with properties:

                  FrameType: 'Beacon'
                  Timestamp: 0
             BeaconInterval: 100
              ESSCapability: 1
             IBSSCapability: 0
                    Privacy: 0
              ShortPreamble: 0
         SpectrumManagement: 0
                 QoSSupport: 1
          ShortSlotTimeUsed: 0
                APSDSupport: 0
           RadioMeasurement: 0
     DelayedBlockAckSupport: 0
   ImmediateBlockAckSupport: 0
                       SSID: 'default SSID'
                 BasicRates: {'6 Mbps'  '12 Mbps'  '24 Mbps'}
            AdditionalRates: {}

   Read-only properties:
        InformationElements: {511x2 cell}
```

Configure the information fields and elements in the frame-body configuration. You can add information elements using `addIE(elementID, information)` method as shown below. Section 9.4 in [1] lists the information fields and information elements.

```
% Beacon Interval
frameBodyCfg.BeaconInterval = 100;
% Timestamp
frameBodyCfg.Timestamp = 123456;
% SSID
frameBodyCfg.SSID = 'TEST_BEACON';
% Add DS Parameter IE (element ID - 3) with channel number 11 (0x0b)
frameBodyCfg = frameBodyCfg.addIE(3, '0b');
```

Assign the updated frame-body configuration object to the `ManagementConfig` property in the MAC frame configuration.

```
% Update management frame-body configuration
beaconCfg.ManagementConfig = frameBodyCfg;
```

Generate the Beacon frame with the updated frame configuration.

```
% Generate octets for a Beacon frame
beaconFrame = wlanMACFrame(beaconCfg);
```

By default, the output of `wlanMACFrame` is a sequence of hexadecimal octets. If you want to generate the MAC frame as a sequence of bits, set the `OutputFormat` parameter to `bits`.

```
% Generate bits for a Beacon frame
beaconFrameBits = wlanMACFrame(beaconCfg, 'OutputFormat', 'bits');
```

Export WLAN MAC Frames to PCAP or PCAPNG File

The packet capture (PCAP) or packet capture next generation (PCAPNG) file (.pcap or .pcapng, respectively) is a widely used packet capture file format to perform packet analysis.

To capture the packet characteristics, export the generated MAC frames to a PCAP or PCAPNG file by using the `wlanPCAPWriter` object. You can visualize and analyze the PCAP or PCAPNG file by using a third-party packet analyzer tool such as Wireshark.

Specify the name and extension of the PCAP file. To export the MAC frames to a PCAPNG file, set the file extension to 'pcapng'.

```
fileName = 'macFrames';
fileExtension = 'pcap';
```

If a file with the `fileName` name already exists in the current directory, delete the existing file.

```
if isfile(strcat(fileName, '.', fileExtension))
    delete(strcat(fileName, '.', fileExtension));
end
```

Set the packet arrival time in POSIX® microseconds.

```
timestamp = 124800;
```

Create a WLAN PCAP file writer object with the specified file name and extension by using the `wlanPCAPWriter` object.

```
pcap = wlanPCAPWriter('FileName', fileName, 'FileExtension', fileExtension);
```

Specify the MAC frames to be exported to the PCAP file.

```
frames = {rtsFrame, qosDataFrame, beaconFrame};
```

Write the MAC frames to the PCAP file.

```
for idx = 1:numel(frames)
    write(pcap, frames{idx}, timestamp);
end
```

Delete the PCAP file writer object.

```
delete(pcap);
```

Visualization of the Generated MAC Frames

You can open the PCAP files containing the generated MAC frames in a packet analyzer. The frames decoded by Wireshark match the standard compliant MAC frames generated using the WLAN Toolbox. This figure shows the analysis of the captured MAC frames in Wireshark.

- **RTS frame**

```
IEEE 802.11 Request-to-send, Flags: ........C
    Type/Subtype: Request-to-send (0x001b)
    Frame Control Field: 0xb400
        .... ..00 = Version: 0
        .... 01.. = Type: Control frame (1)
        1011 .... = Subtype: 11
        Flags: 0x00
            .... ..00 = DS status: Not leaving DS or network is operating in AD-HOC mode (To DS: 0 From DS: 0) (0x0)
            .... .0.. = More Fragments: This is the last fragment
            .... 0... = Retry: Frame is not being retransmitted
            ...0 .... = PWR MGT: STA will stay up
            ..0. .... = More Data: No data buffered
            .0.. .... = Protected flag: Data is not protected
            0... .... = Order flag: Not strictly ordered
    .000 0001 1111 0100 = Duration: 500 microseconds
    Receiver address: fc:f8:b0:10:20:01 (fc:f8:b0:10:20:01)
    Transmitter address: fc:f8:b0:10:20:02 (fc:f8:b0:10:20:02)
    Frame check sequence: 0x57334a70 [correct]
    [FCS Status: Good]
```

- **QoS Data frame**

```
IEEE 802.11 QoS Data, Flags: ......F.C
    Type/Subtype: QoS Data (0x0028)
    Frame Control Field: 0x8802
        .... ..00 = Version: 0
        .... 10.. = Type: Data frame (2)
        1000 .... = Subtype: 8
        Flags: 0x02
            .... ..10 = DS status: Frame from DS to a STA via AP(To DS: 0 From DS: 1) (0x2)
            .... .0.. = More Fragments: This is the last fragment
            .... 0... = Retry: Frame is not being retransmitted
            ...0 .... = PWR MGT: STA will stay up
            ..0. .... = More Data: No data buffered
            .0.. .... = Protected flag: Data is not protected
            0... .... = Order flag: Not strictly ordered
    .000 0000 0000 0000 = Duration: 0 microseconds
    Receiver address: fc:f8:b0:10:20:01 (fc:f8:b0:10:20:01)
    Destination address: fc:f8:b0:10:20:01 (fc:f8:b0:10:20:01)
    Transmitter address: fc:f8:b0:10:20:02 (fc:f8:b0:10:20:02)
    Source address: CamilleB_56:78:9b (00:12:34:56:78:9b)
    BSS Id: fc:f8:b0:10:20:02 (fc:f8:b0:10:20:02)
    STA address: fc:f8:b0:10:20:01 (fc:f8:b0:10:20:01)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0000 .... = Sequence number: 0
    Frame check sequence: 0xc85aa49b [correct]
    [FCS Status: Good]
    Qos Control: 0x0000
        .... .... .... 0000 = TID: 0
        [.... .... .... .000 = Priority: Best Effort (Best Effort) (0)]
        .... .... ...0 .... = EOSP: Service period
        .... .... .00. .... = Ack Policy: Normal Ack (0x0)
        .... .... 0... .... = Payload Type: MSDU
        0000 0000 .... .... = QAP PS Buffer State: 0x00
Logical-Link Control
Data (16 bytes)
    Data: 11111111111111111111111111111111
    [Length: 16]
```

- **Beacon frame**

```
IEEE 802.11 Beacon frame, Flags: ......F.C
  Type/Subtype: Beacon frame (0x0008)
  Frame Control Field: 0x8002
    .... ..00 = Version: 0
    .... 00.. = Type: Management frame (0)
    1000 .... = Subtype: 8
    Flags: 0x02
  .000 0000 0000 0000 = Duration: 0 microseconds
  Receiver address: Broadcast (ff:ff:ff:ff:ff:ff)
  Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
  Transmitter address: CamilleB_56:78:9b (00:12:34:56:78:9b)
  Source address: CamilleB_56:78:9b (00:12:34:56:78:9b)
  BSS Id: CamilleB_56:78:9b (00:12:34:56:78:9b)
  STA address: Broadcast (ff:ff:ff:ff:ff:ff)
  .... .... .... 0000 = Fragment number: 0
  0000 0000 0000 .... = Sequence number: 0
  Frame check sequence: 0x5e56cf45 [correct]
  [FCS Status: Good]
IEEE 802.11 wireless LAN
  Fixed parameters (12 bytes)
    Timestamp: 0x000000000001e240
    Beacon Interval: 0.102400 [Seconds]
    Capabilities Information: 0x0201
      .... .... .... ...1 = ESS capabilities: Transmitter is an AP
      .... .... .... ..0. = IBSS status: Transmitter belongs to a BSS
      .... ..1. .... 00.. = CFP participation capabilities: QAP (HC) does not use CFP for delivery of unicast data type frames (0x80)
      .... .... ...0 .... = Privacy: AP/STA cannot support WEP
      .... .... ..0. .... = Short Preamble: Not Allowed
      .... .... .0.. .... = PBCC: Not Allowed
      .... .... 0... .... = Channel Agility: Not in use
      .... ...0 .... .... = Spectrum Management: Not Implemented
      .... .0.. .... .... = Short Slot Time: Not in use
      .... 0... .... .... = Automatic Power Save Delivery: Not Implemented
      ...0 .... .... .... = Radio Measurement: Not Implemented
      ..0. .... .... .... = DSSS-OFDM: Not Allowed
      .0.. .... .... .... = Delayed Block Ack: Not Implemented
      0... .... .... .... = Immediate Block Ack: Not Implemented
  Tagged parameters (21 bytes)
    Tag: SSID parameter set: TEST_BEACON
      Tag Number: SSID parameter set (0)
      Tag length: 11
      SSID: TEST_BEACON
    Tag: Supported Rates 6(B), 12(B), 24(B), [Mbit/sec]
      Tag Number: Supported Rates (1)
      Tag length: 3
      Supported Rates: 6(B) (0x8c)
      Supported Rates: 12(B) (0x98)
      Supported Rates: 24(B) (0xb0)
    Tag: DS Parameter set: Current Channel: 11
      Tag Number: DS Parameter set (3)
      Tag length: 1
      Current Channel: 11
```

Conclusion and Further Exploration

This example shows how to generate MAC frames for the IEEE 802.11 standard. You can use a packet analyzer to view the generated MAC frames. To transmit the generated MAC frames over the air, see “802.11 OFDM Beacon Frame Generation” on page 2-25 and “802.11ac Waveform Generation with MAC Frames” on page 2-18.

References

[1] IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016). “Part 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.

[2] IEEE Std 802.11ax-2021 (Amendment to IEEE Std 802.11-2020). “Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 1: Enhancements

for High Efficiency WLAN." IEEE Standard for Information technology — Telecommunications and information exchange between systems. Local and metropolitan area networks — Specific requirements.

[3] Wireshark · Go Deep. https://www.wireshark.org/. Accessed 30 June 2020

[4] Group, The Tcpdump. Tcpdump/Libpcap Public Repository. https://www.tcpdump.org. Accessed 30 June 2020

See Also

More About

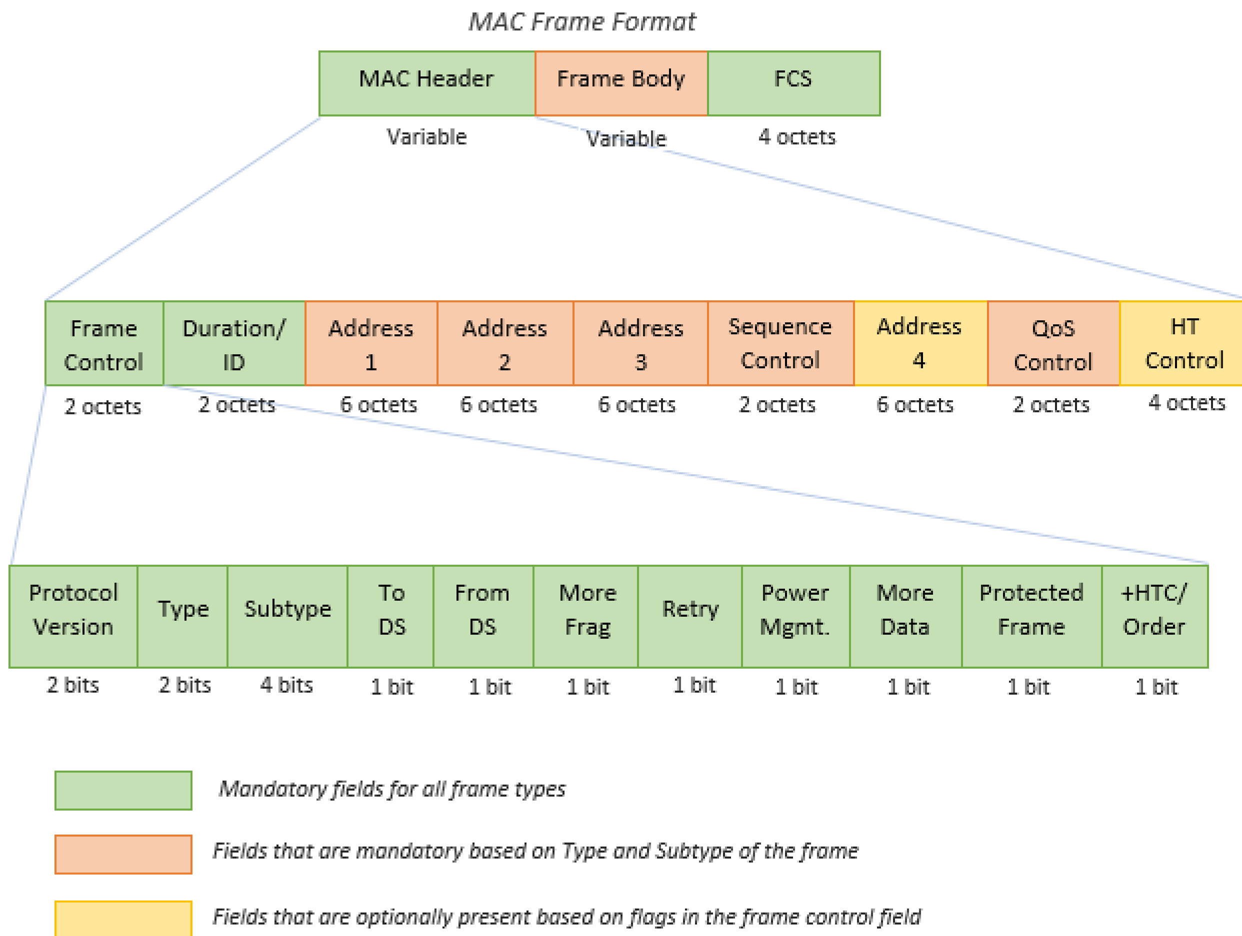
- "WLAN MAC Frame Structure"
- "802.11ac Waveform Generation with MAC Frames" on page 2-18
- "802.11 MAC Frame Decoding" on page 2-12

802.11 MAC Frame Decoding

This example shows how to decode IEEE® 802.11™ MAC frames.

Background

The general MAC frame format consists of a header, frame-body, and frame check sequence (FCS). The header holds information about the frame. The frame-body carries data that needs to be transmitted. The transmitter calculates the FCS over the header and frame-body. The receiver uses the FCS to confirm that the header and frame-body are properly received. The following diagram shows the structure of a general MAC frame.



For more information, see the “WLAN MAC Frame Structure” topic.

Introduction

This example shows how WLAN MAC frames specified in Section 9.3 of [1] or [2] can be decoded. It also shows how aggregated MAC frames specified in Section 9.7 of [1] or [2] can be deaggregated.

WLAN Toolbox™ supports MPDU decoding for the following MAC frames:

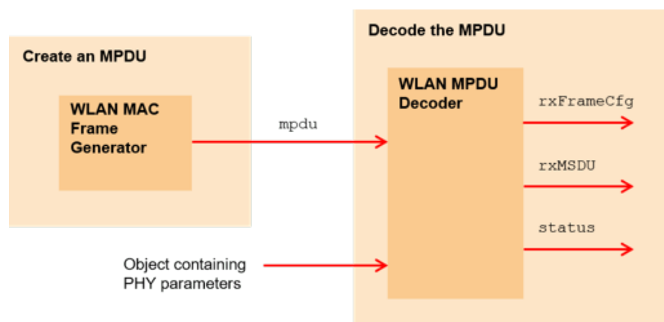
- **Management Frames:** Beacon
- **Data Frames:** Data, Null, QoS Data, QoS Null
- **Control Frames:** RTS, CTS, Ack, Block Ack

In addition to MPDU decoding, WLAN Toolbox also supports deaggregation of an A-MPDU.

MPDU Decoding

An MPDU can be a data, control or management frame type. `wlanMPDUDecode` can be used to decode an MPDU. This function processes the given MPDU and a physical layer configuration object to output the decoded MAC parameters.

To illustrate MPDU decoding, a valid MPDU is created using `wlanMACFrame`. The created MPDU is passed to the `wlanMPDUDecode` function and the outputs are observed.



Create an MPDU

A QoS Data frame is created for this example using `wlanMACFrame`. The following inputs are required to form a Non-HT format QoS Data frame containing a 40-octet payload:

1. `txFrameCfg` : A MAC frame configuration object of type `wlanMACFrameConfig`.
2. `txMSDU` : A 40-octet payload (MSDU) to be included in the QoS Data frame.

```
% Create a MAC frame configuration object
txFrameCfg = wlanMACFrameConfig('FrameType', 'QoS Data', ...
                                'FrameFormat', 'Non-HT');

% 40-octet payload for each 'QoS Data' frame
txMSDU = randi([0, 255], 40, 1);

% Physical layer configuration
phyCfg = wlanNonHTConfig;

% Create the MPDU
mpdu = wlanMACFrame(txMSDU, txFrameCfg);
```

Decode the MPDU

`wlanMPDUDecode` consumes an MPDU, a PHY configuration object of type `wlanNonHTConfig`, `wlanHTConfig`, `wlanVHTConfig`, or `wlanHESUConfig` and optionally a (Name, Value) pair for `DataFormat` specifying the input format of the MPDU. Since the MPDU generated using `wlanMACFrame` is in terms of octets, `DataFormat` is set to `octets`. `wlanMPDUDecode` decodes the MPDU and outputs the following information:

1. `rxFrameCfg` : A MAC frame configuration object of type `wlanMACFrameConfig`, containing the decoded MAC parameters.
2. `rxMSDU` : A cell array, where each element is an n-by-2 character array representing the decoded MSDU. Multiple MSDUs are returned when the MPDU contains an aggregated MSDU (A-MSDU) as the payload.
3. `status` : An enumeration of type "status", which indicates whether the MPDU decoding was successful.

```
% Decode the MPDU.
[rxFrameCfg, rxMSDU, status] = wlanMPDUDecode(mpdu, phyCfg, ...
                                            'DataFormat', 'octets');

% Check if the MPDU is decoded successfully
disp(['Status of the MPDU decoding: ' char(status)])

% Observe the outputs, if the MPDU is decoded successfully
if strcmp(status, 'Success')
    disp(['Type of the decoded MPDU: ' rxFrameCfg.FrameType])
    disp(['Number of MSDUs in the MPDU: ' num2str(numel(rxMSDU))])
    for i = 1:numel(rxMSDU)
        disp(['Size of MSDU-' num2str(i) ': ' num2str(size(rxMSDU{i}, 1)) ' octets'])
    end
end
```

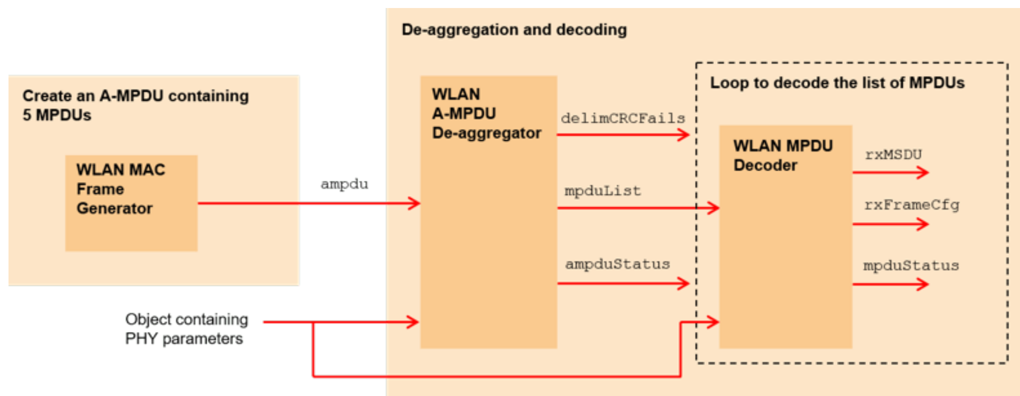
```
Status of the MPDU decoding: Success
Type of the decoded MPDU: QoS Data
Number of MSDUs in the MPDU: 1
Size of MSDU-1: 40 octets
```

A-MPDU Deaggregation

An A-MPDU is an aggregation of multiple MPDUs. The type of MPDUs in an A-MPDU are restricted as specified in Section 9.7.3 of [1].

`wlanAMPDUDeaggregate` can be used to deaggregate an A-MPDU. This function processes the given A-MPDU and the corresponding physical layer configuration object to output the deaggregated list of MPDUs. `wlanAMPDUDeaggregate` is capable of decoding HT (High Throughput), VHT (Very High Throughput), HE-SU (High Efficiency Single User) and HE-EXT-SU (High Efficiency Extended Range Single User) format A-MPDUs as specified in [1] and [2].

To illustrate the A-MPDU deaggregation, a valid A-MPDU containing five MPDUs is created using `wlanMACFrame`. The created A-MPDU is passed to the `wlanAMPDUDeaggregate` function and the outputs are observed.



Create an A-MPDU

The following inputs are required to form an HE-SU format A-MPDU containing five MPDUs (QoS Data frames), each MPDU containing a 40-octet payload:

1 `txFrameCfg` : A MAC frame configuration object of type `wlanMACFrameConfig`.
2 `txMSDUList` : A five element cell array containing payload (MSDU) for five MPDUs. Since `MSDUAggregation` is set to false in the `txFrameCfg`, a separate MPDU is created for each MSDU.
3 `phyCfg` : A physical layer configuration object of type `wlanHESUConfig`.

```
% Create a MAC frame configuration object
txFrameCfg = wlanMACFrameConfig('FrameType', 'QoS Data', ...
                                'FrameFormat', 'HE-SU', ...
                                'MPDUAggregation', true, ...
                                'MSDUAggregation', false);

% 40-octet payload for each 'QoS Data' frame
txMSDUList = repmat({randi([0, 255], 40, 1)}, 1, 5);

% Physical layer configuration
phyCfg = wlanHESUConfig('MCS', 3);

% Create the A-MPDU containing 5 MPDUs
ampdu = wlanMACFrame(txMSDUList, txFrameCfg, phyCfg);
```

Deaggregate the A-MPDU

`wlanAMPDUDeaggregate` consumes an A-MPDU, a PHY configuration object of type `wlanHTConfig`, `wlanVHTConfig`, or `wlanHESUConfig` and optionally a (Name, Value) pair for `DataFormat` specifying the input format of the A-MPDU. It finds and validates the MPDU delimiters, extracts the MPDUs and outputs the following information that can be used for further processing the MPDUs:

1 `mpduList` : A cell array containing the list of MPDUs extracted from the A-MPDU.
2 `delimCRCFails` : A logical row vector representing delimiter CRC validity for the corresponding index in `mpduList`. A value of true represents that the MPDU present in `mpduList` at the corresponding index may not be properly extracted.
3 `ampduStatus` : An enumeration of type “status”, which indicates whether the A-MPDU deaggregation was successful.

```
% Deaggregate the A-MPDU
[mpduList, delimCRCFails, ampduStatus] = wlanAMPDUDeaggregate(ampdu, phyCfg, ...
                                                    'DataFormat', 'octets');

% Observe the outputs
disp(['Status of A-MPDU deaggregation: ' char(ampduStatus)])
disp(['Number of MPDUs extracted from the A-MPDU: ' num2str(numel(mpduList))])
disp(['Number of MPDUs with delimiter CRC fails: ' num2str(nnz(delimCRCFails))])
```

```
Status of A-MPDU deaggregation: Success
Number of MPDUs extracted from the A-MPDU: 5
Number of MPDUs with delimiter CRC fails: 0
```

Decode the list of MPDUs

The `mpduList` contains the list of MPDUs extracted from the A-MPDU. Each of the MPDUs present in the list can be decoded separately. However, if the `delimCRCFails` contains any `true` values, the MPDU present in `mpduList` at the corresponding index can be considered invalid as it may not be properly extracted because of the delimiter CRC failure.

```
% Decode the list of MPDUs
if strcmp(ampduStatus, 'Success')
    % Number of MPDUs in the list
    numMPDUs = numel(mpduList);

    for i = 1:numMPDUs
        % Decode the MPDU only if the corresponding delimiter CRC is valid
        if ~delimCRCFails(i)
            [rxFrameCfg, rxMSDU, mpduStatus] = wlanMPDUDecode(mpduList{i}, phyCfg, ...
                                                    'DataFormat', 'octets');
            disp(['MPDU-' num2str(i) ' decoding status: ' char(mpduStatus)])
            disp(['MPDU-' num2str(i) ' type: ' rxFrameCfg.FrameType])
            disp(['MPDU-' num2str(i) ' payload size: ' num2str(size(rxMSDU{1}, 1)) ' octets'])
            disp(' ')
        end
    end
end
```

```
MPDU-1 decoding status: Success
MPDU-1 type: QoS Data
MPDU-1 payload size: 40 octets

MPDU-2 decoding status: Success
MPDU-2 type: QoS Data
MPDU-2 payload size: 40 octets

MPDU-3 decoding status: Success
MPDU-3 type: QoS Data
MPDU-3 payload size: 40 octets

MPDU-4 decoding status: Success
MPDU-4 type: QoS Data
MPDU-4 payload size: 40 octets

MPDU-5 decoding status: Success
MPDU-5 type: QoS Data
MPDU-5 payload size: 40 octets
```

Conclusion and Further Exploration

This example demonstrated how to deaggregate and decode IEEE 802.11 MAC frames. You can also explore "OFDM Beacon Receiver Using Software-Defined Radio" on page 10-33 and "Recovery Procedure for an 802.11ac Packet" on page 4-30 examples for decoding the MAC frames retrieved from the captured waveforms.

Selected Bibliography

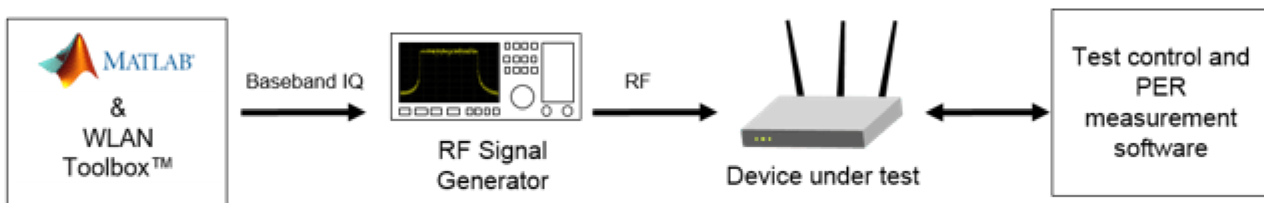
1. IEEE Std 802.11™-2020. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.
2. IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

802.11ac Waveform Generation with MAC Frames

This example shows how to generate an IEEE® 802.11ac™ transmission containing MAC frames suitable for performing radio packet error rate (PER) receiver tests.

Introduction

WLAN Toolbox™ can be used to generate standard compliant waveforms for performing receiver tests. A basic WLAN receiver test scenario is shown in the diagram below.



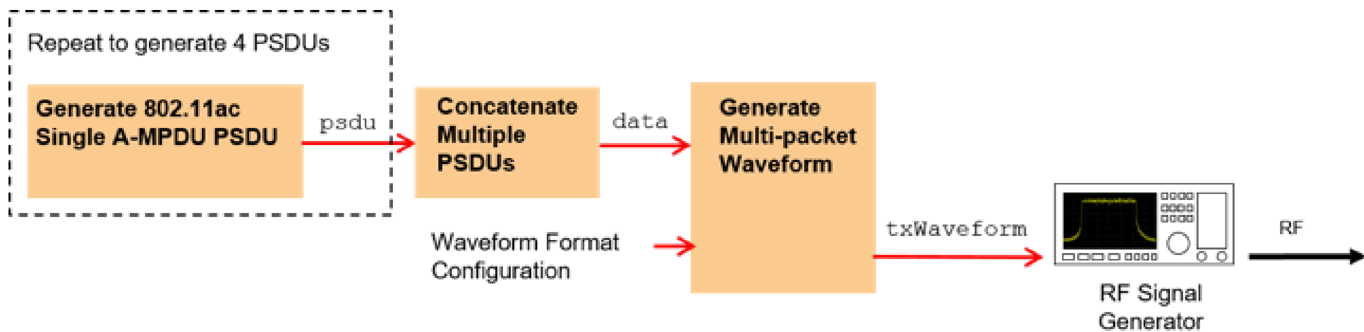
The device under test (DUT) is stimulated with RF test vectors, usually through a wired link. The packet error rate (PER) is a metric used to test the performance of a receiver at a given receive signal power in the presence of noise, interference, or other impairments. The PER is defined as the number of incorrectly decoded packets divided by the total number of transmitted packets.

The frame check sequence (FCS) within a MAC frame is used to determine whether a MAC frame has been decoded correctly by the receiver, and therefore whether the packet has been received in error. The general MAC frame for IEEE 802.11ac contains the following fields:

- MAC header
- Frame body
- FCS

The data to transmit from a higher layer is contained within the frame body of the MAC frame. The transmitter uses a cyclic redundancy check over the MAC header and frame body field to generate the FCS value. The receiver calculates the CRC and compares this to the received FCS field to determine if an error has occurred during transmission.

In this example an IEEE 802.11ac waveform consisting of multiple VHT format packets is generated. The `wlanWaveformGenerator` function can be used to generate a waveform containing one or more packets. The `wlanWaveformGenerator` function consumes physical layer service data units (PSDUs) for each packet and performs the appropriate physical layer processing to create the waveform. A PSDU containing a MAC header and valid FCS can be generated using the `wlanMACFrame` function. In this example a multi-packet baseband waveform containing MAC packets is synthesized. This waveform may be downloaded to a signal generator for RF transmission and used for receiver PER testing. Source code is provided to download and play the waveform using a Keysight Technologies™ N5172B signal generator. The example processing is illustrated in the following diagram:



802.11ac VHT Format Configuration

The format-specific configuration of a VHT waveform synthesized with the `wlanWaveformGenerator` function is described by the VHT format configuration object, `wlanVHTConfig`. The properties of the object contain the configuration. In this example an object is configured for a 160 MHz bandwidth, 1 transmit antenna, 1 space-time stream and QPSK rate 1/2 (MCS 1).

```
vhtCfg = wlanVHTConfig;                % Create packet configuration
vhtCfg.ChannelBandwidth = 'CBW160'; % 160 MHz channel bandwidth
vhtCfg.NumTransmitAntennas = 1;     % 1 transmit antenna
vhtCfg.NumSpaceTimeStreams = 1;     % 1 space-time stream
vhtCfg.MCS = 1;                     % Modulation: QPSK Rate: 1/2
```

Waveform Generation Configuration

The `wlanWaveformGenerator` function can be configured to generate one or more packets and add an idle time between each packet. The function can be configured to generate an oversampled or nominal rate waveform. In this example four oversampled packets with a 20 microsecond idle period will be created.

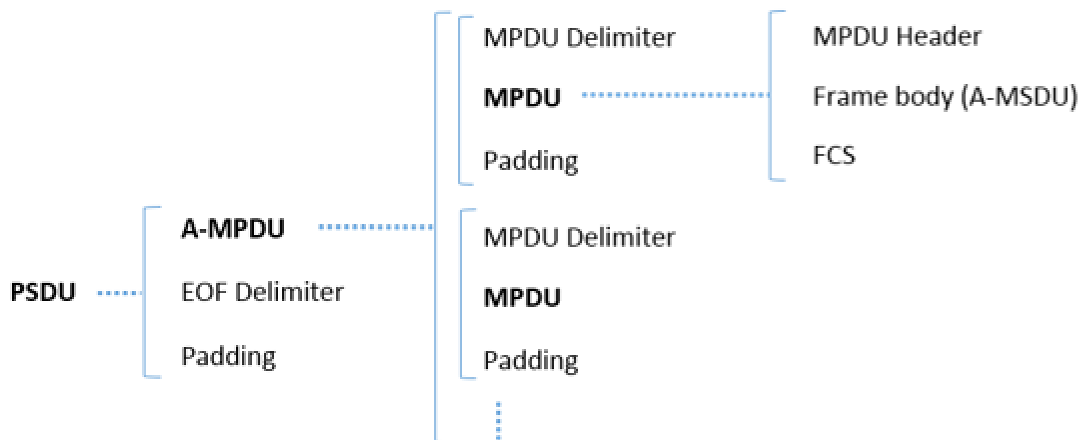
```
numPackets = 4;   % Generate 4 packets
idleTime = 20e-6; % 20 microseconds idle period after packet
oversamplingFactor = 1.5; % Oversample waveform 1.5x nominal baseband rate
```

The PSDU transmitted in each packet is scrambled using a random seed for each packet. This is accomplished by specifying a vector of scrambler initialization seeds. The valid range of the seed is between 1 and 127 inclusive.

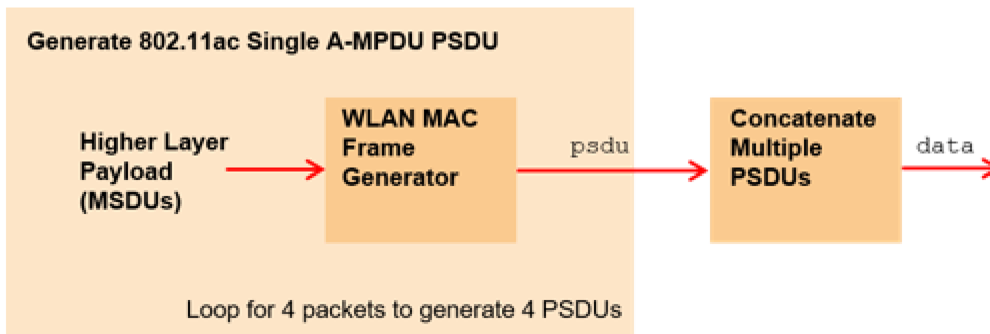
```
% Initialize the scrambler with a random integer for each packet
scramblerInitialization = randi([1 127],numPackets,1);
```

Create a PSDU for Each Packet

For an IEEE 802.11ac data transmission the MAC frame is termed a MAC protocol data unit (MPDU), the MAC header is termed the MPDU header, and the frame body is an aggregated MAC service data unit (A-MSDU). One or more MPDUs are delimited, padded and aggregated to create an aggregated MPDU (A-MPDU). The A-MPDU is delimited and padded to form the physical layer service data unit (PSDU) which is coded and modulated to create the transmitted packet. This process of encapsulation is shown in the following diagram:



In this example a PSDU is created containing a single MPDU for each packet. The MPDU consists of an MPDU header, A-MSDU frame containing concatenated A-MSDU subframes with random data and valid FCS. The `wlanMACFrame` function creates an A-MPDU with EOF delimiters and padding, i.e. the PSDU, as specified in [1 on page 2-24]. It also returns the length of the A-MPDU, termed as the APEP Length, which is used to set the `APEPLength` property of the VHT configuration object. A PSDU is generated for each packet and is concatenated into a vector `data` for transmission with the `wlanWaveformGenerator` function. The processing to create the concatenated PSDU bits `data` is shown in the diagram below:




```
% Create frame configuration
macCfg = wlanMACFrameConfig('FrameType', 'QoS Data');
macCfg.FrameFormat = 'VHT';     % Frame format
macCfg.MSDUAggregation = true;  % Form A-MSDUs internally
bitsPerByte = 8;                % Number of bits in 1 byte
data = [];

for i=1:numPackets
    % Get MSDU lengths to create a random payload for forming an A-MPDU of
    % 4048 octets (pre-EOF padding)
    msduLengths = wlanMSDULengths(4048, macCfg, vhtCfg);
    msdu = cell(numel(msduLengths), 1);

    % Create MSDUs with the obtained lengths
    for j = 1:numel(msduLengths)
        msdu{j} = randi([0 255], 1, msduLengths(j));
    end

    % Generate PSDU bits containing A-MPDU with EOF delimiters and padding
```

```
    [psdu, apepLength] = wlanMACFrame(msdu, macCfg, vhtCfg, 'OutputFormat', 'bits');

    % Set the APEP length in the VHT configuration
    vhtCfg.APEPLength = apepLength;

    % Concatenate packet PSDUs for waveform generation
    data = [data; psdu]; %#ok<AGROW>
end
```

Generate a Baseband Waveform

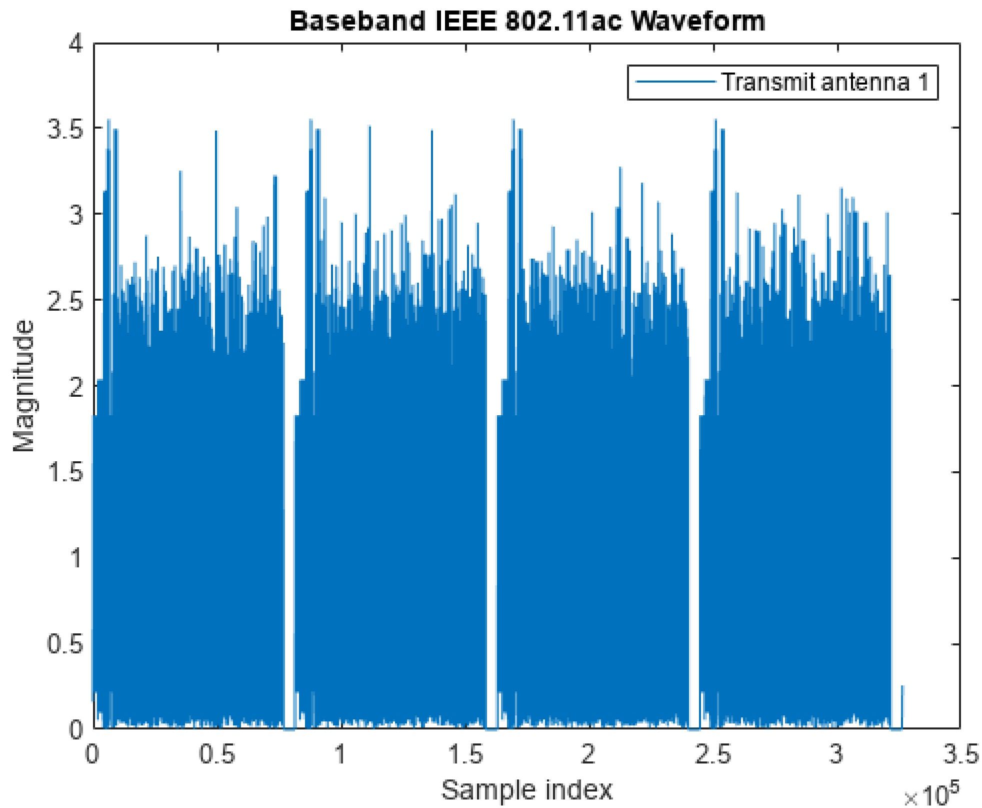
The concatenated PSDU bits for all packets, `data`, are passed as an argument to the `wlanWaveformGenerator` function along with the VHT packet configuration object `vhtCfg`. This configures the waveform generator to synthesize an 802.11ac VHT waveform. To generate 802.11n™ HT or other format waveforms, use a different format configuration object, for example `wlanHTConfig` or `wlanNonHTConfig`. The waveform generator is additionally configured using name-value pairs to generate multiple oversampled packets with a specified idle time between packets, and initial scrambler states.

```
% Generate baseband VHT packets
txWaveform = wlanWaveformGenerator(data,vhtCfg, ...
    'NumPackets',numPackets,'IdleTime',idleTime, ...
    'ScramblerInitialization',scramblerInitialization, ...
    'OversamplingFactor',oversamplingFactor);

fs = wlanSampleRate(vhtCfg,'OversamplingFactor',oversamplingFactor);
disp(['Baseband sampling rate: ' num2str(fs/1e6) ' Msps']);
```

```
Baseband sampling rate: 240 Msps
```

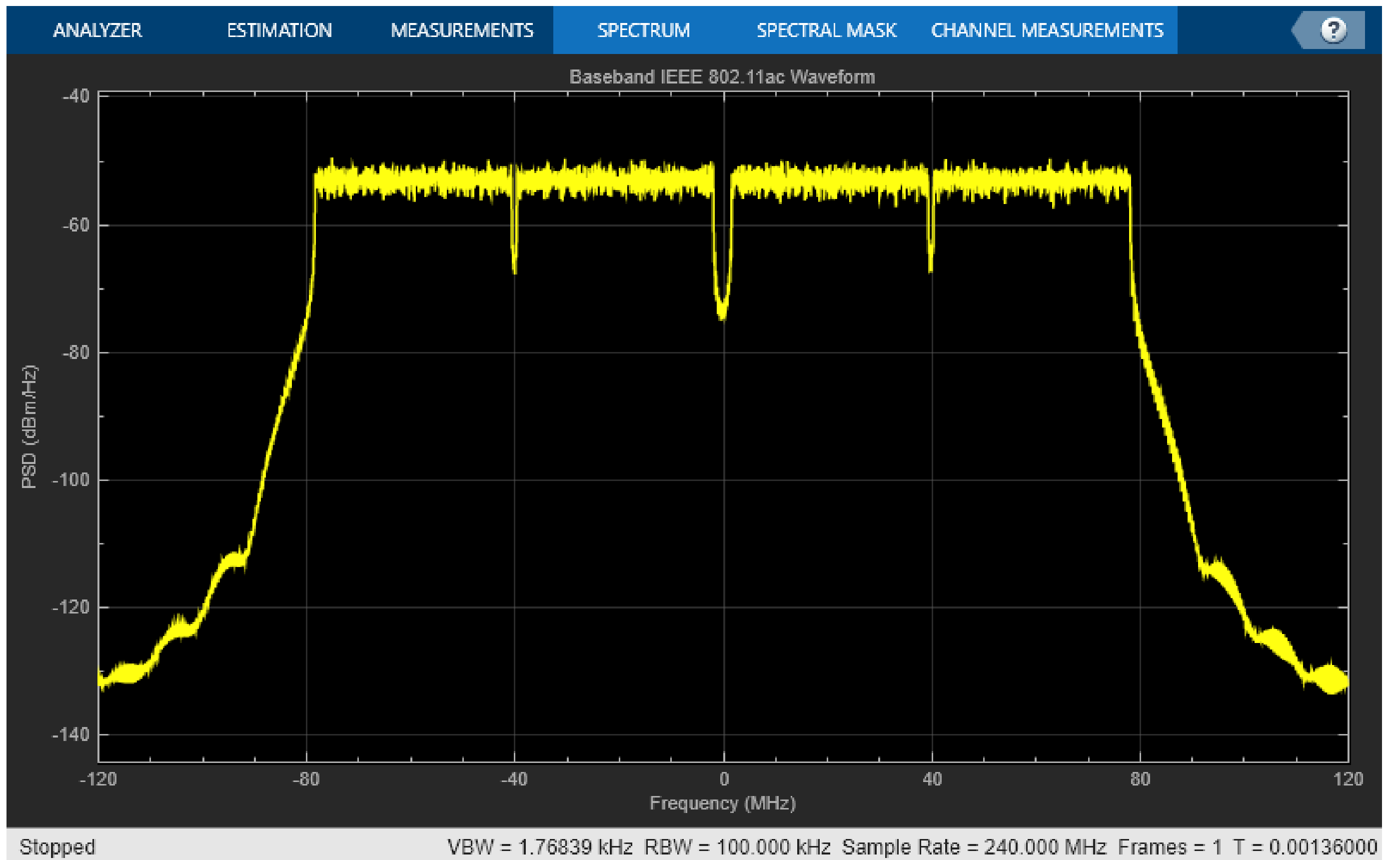
The magnitude of the baseband waveform is displayed below. Note the number of packets and idle time configured.

```
figure;
plot(abs(txWaveform));
xlabel('Sample index');
ylabel('Magnitude');
title('Baseband IEEE 802.11ac Waveform');
legend('Transmit antenna 1');
```



View the frequency spectrum of the generated time domain waveform by using the “DSP System Toolbox” `spectrumAnalyzer`. As expected, the 160 MHz signal bandwidth is clearly visible.

```
spectrumScope = spectrumAnalyzer;
spectrumScope.SampleRate = fs;
spectrumScope.SpectrumType = 'power-density';
spectrumScope.RBWSource = 'property';
spectrumScope.RBW = 100e3;
spectrumScope.AveragingMethod = 'exponential';
spectrumAnalyze.ForgettingFactor = 0.99;
spectrumScope.YLabel = 'PSD';
spectrumScope.Title = 'Baseband IEEE 802.11ac Waveform';
spectrumScope(txWaveform);
release(spectrumScope)
```



Generate Over-the-Air Signal Using an RF Signal Generator

The baseband waveform created by WLAN Toolbox can now be downloaded to a signal generator to perform receiver tests. Use “Instrument Control Toolbox” to generate an RF signal with a center frequency of 5.25 GHz RF using the Keysight Technologies N5172B signal generator.

```
% Control whether to download the waveform to the waveform generator
playOverTheAir = false;

% Download the baseband IQ waveform to the instrument. Generate the RF
% signal at a center frequency of 5.25 GHz and output power of -10 dBm.
if playOverTheAir
    fc = 5.25e9; %#ok<UNRCH> % Center frequency
    power = -10;      % Output power
    loopCount = Inf; % Number time to loop

    % Configure the signal generator, download the waveform and loop
    rf = rfsiggen();
    rf.Resource = 'TCPIP0::192.168.0.1::inst0::INSTR';
    rf.Driver = 'AgRfSigGen';
    connect(rf);                    % Connect to the instrument
    download(rf,txWaveform.',fs); % Download the waveform to the instrument
    start(rf,fc,power,loopCount); % Start transmitting waveform

    % When you have finished transmitting, stop the waveform output
    stop(rf);
```

```
    disconnect(rf);
end
```

Selected Bibliography

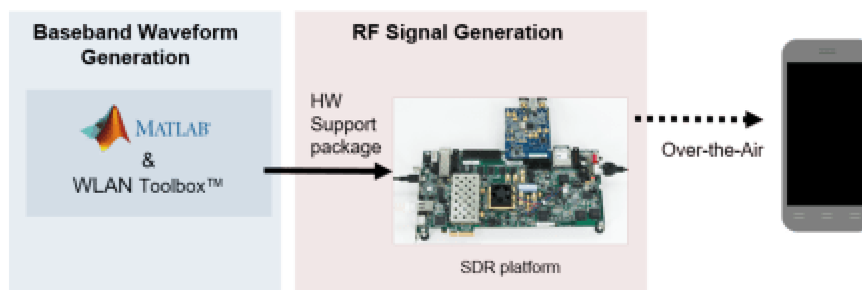
1 IEEE Std 802.11™-2016 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

802.11 OFDM Beacon Frame Generation

This example shows how to generate packets containing medium access control (MAC) beacon frames suitable for baseband simulation or over-the-air transmission using a software-defined radio (SDR) platform.

Introduction

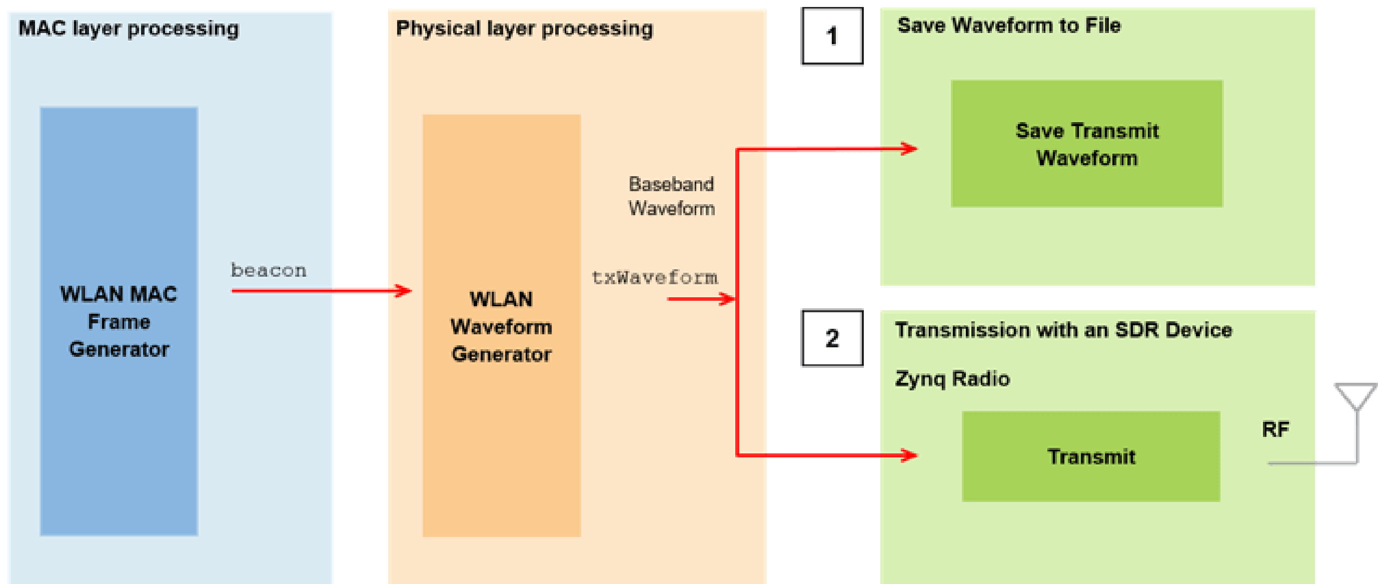
In this example, you create an IEEE® 802.11™ beacon frame as described in section 9.3.3.3 of [1 on page 2-28]. You can view the beacon packet transmitted using SDR by using A Wi-Fi device, as shown in this figure.



A beacon frame is a type of management frame that identifies a basic service set (BSS) formed by some 802.11 devices. The beacon frame consists of a MAC header, a beacon frame body and a valid frame check sequence (FCS). The beacon frame body contains information fields that allow stations to associate with the network.

You can store the generated waveform in a baseband file format.

You can also transmit the generated waveform over the air. Upconvert the beacon packet for RF transmission using Xilinx® Zynq-Based Radio SDR hardware. The radio hardware can transmit the waveform over the air.



Transmitting the beacon over the air requires the Xilinx Zynq-based radio support package, which you can install with the Add-On Explorer. For more information about SDR platforms, see Hardware Support: Communications Toolbox.

Example Setup

The beacon packet can be written to a baseband file and transmitted using an SDR platform. To transmit the beacon using the SDR platform set `useSDR` to true. To write to a baseband file set `saveToFile` to true.

```
useSDR = false;
saveToFile = false;
```

Create IEEE 802.11 Beacon Frame

A station (STA) periodically transmits beacon packets as specified by the target beacon transmission time (TBTT) in the Beacon Interval field. The beacon interval represents the number of time units (TUs) between TBTTs, where 1 TU represents 1024 microseconds. A beacon interval of 100 TUs results in a 102.4 millisecond time interval between successive beacons. You can generate a beacon frame by using the `wlanMACFrame` function with medium access control (MAC) frame configuration object `wlanMACFrameConfig` and MAC frame-body configuration object `wlanMACManagementConfig`.

Specify the network service set identifier (SSID), beacon interval, operating band, and channel number.

```
ssid = "TEST_BEACON";
beaconInterval = 100;
band = 5;
chNum = 52;
```

Create a MAC frame-body configuration object, setting the SSID and Beacon Interval field value.

```
frameBodyConfig = wlanMACManagementConfig( ...
    BeaconInterval=beaconInterval, ...
    SSID=ssid);
```

Add the DS Parameter information element (IE) to the frame body by using the `addIE` object function.

```
dsElementID = 3;
dsInformation = dec2hex(chNum,2);
frameBodyConfig = frameBodyConfig.addIE(dsElementID,dsInformation);
```

Create beacon frame configuration object.

```
beaconFrameConfig = wlanMACFrameConfig(FrameType="Beacon", ...
    ManagementConfig=frameBodyConfig);
```

Generate beacon frame bits.

```
[mpduBits,mpduLength] = wlanMACFrame(beaconFrameConfig,OutputFormat="bits");
```

Calculate center frequency for the specified operating band and channel number.

```
fc = wlanChannelFrequency(chNum,band);
```

Create IEEE 802.11 Beacon Packet

Configure a non-HT beacon packet with the relevant PSDU length, specifying a channel bandwidth of 20 MHz, one transmit antenna, and BPSK modulation with a coding rate of 1/2 (corresponding to MCS index 0) by using the `wlanNonHTConfig` object.

```
cfgNonHT = wlanNonHTConfig(PSDULength=mpduLength);
```

Generate an oversampled beacon packet by using the `wlanWaveformGenerator` function, specifying an idle time.

```
osf = 2;
tbtt = beaconInterval*1024e-6;
txWaveform = wlanWaveformGenerator(mpduBits,cfgNonHT,...
    OversamplingFactor=osf,Idletime=tbtt);
```

Get the waveform sample rate.

```
Rs = wlanSampleRate(cfgNonHT,OversamplingFactor=osf);
```

Save Waveform to File

Save the waveform in a baseband file using the `comm.BasebandFileWriter` object.

```
if saveToFile
    bbw = comm.BasebandFileWriter("nonHTBeaconPacket.bb",Rs,fc); %#ok<UNRCH>
    bbw(txWaveform);
    release(bbw);
end
```

Transmission with an SDR Device

In this section, you transmit the beacon packet over the air using an SDR device.

```
if useSDR
    % The SDR platform must support transmitRepeat. Valid platforms are
    % 'AD936x' and 'FMCOMMS5'.
    sdrPlatform = 'AD936x'; %#ok<UNRCH>
    tx = sdrtx(sdrPlatform);
    tx.BasebandSampleRate = Rs;
    % Set the center frequency to the corresponding channel number
    tx.CenterFrequency = fc;
end
```

To impair the signal or reduce transmission quality of the waveform, you can adjust the transmitter gain `tx.Gain`. This parameter, expressed in dB, drives the power amplifier in the radio. You can use these suggested values, or select different values appropriate for your antenna configuration.

- For increased gain, set the `tx.Gain` parameter to `0`.
- For default gain, set the `tx.Gain` parameter to `-10`.
- For reduced gain, set the `tx.Gain` parameter to `-20`.

Transfer the baseband waveform to the SDR platform by using the `transmitRepeat` function, and then store the signal samples in hardware memory. The example then repeatedly transmits this waveform over-the-air until the release method of the transmit object is called. Messages are displayed in the command window to confirm that transmission has started successfully.

```
if useSDR
    % Set transmit gain
    tx.Gain = 0; %#ok<UNRCH>
    % Transmit over the air
    transmitRepeat(tx,txWaveform);
end
```

Conclusion and Further Exploration

This example shows how to generate a beacon packet for the IEEE 802.11 standard and view the beacon packet transmitted using SDR hardware by using a Wi-Fi device. You can use the stored baseband beacon packet to recover the transmitted information using the “OFDM Beacon Receiver Using Software-Defined Radio” on page 10-33 example.

Related Examples

- “OFDM Beacon Receiver Using Software-Defined Radio” on page 10-33
- “Recover and Analyze Packets in 802.11 Waveform” on page 4-2
- “802.11 MAC Frame Generation” on page 2-2

References

1. IEEE Std 802.11™-2020 IEEE(Revision of IEEE Std 802.11-2016). “Part 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.

3

Signal Transmission

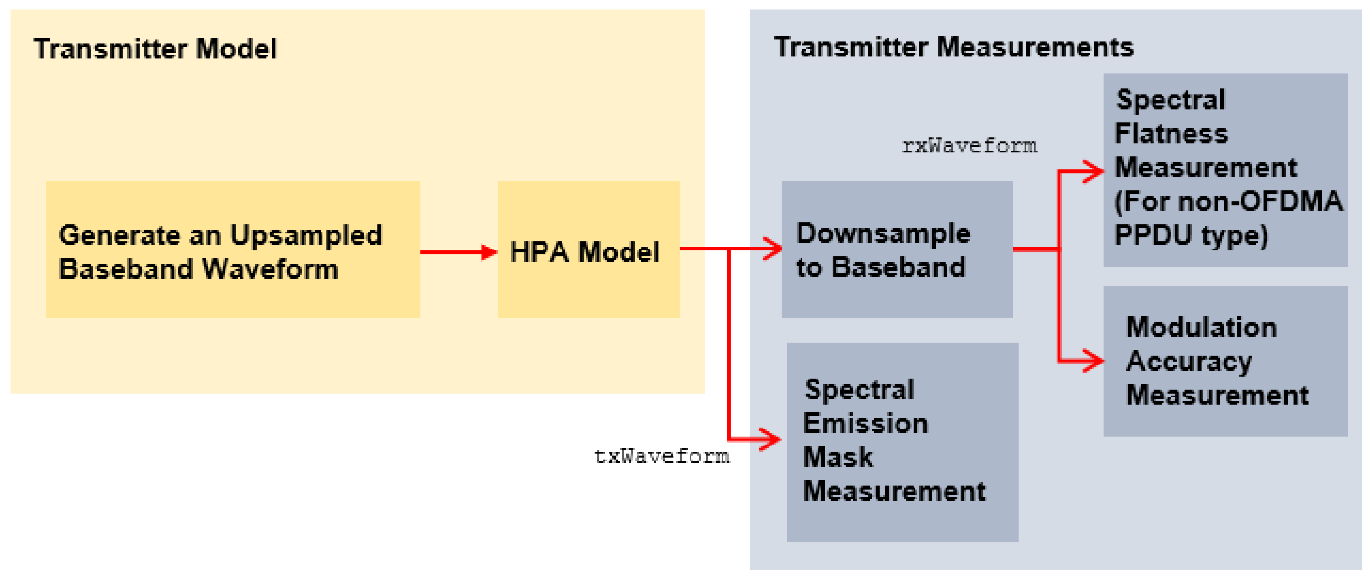
802.11be Transmitter Measurements

This example demonstrates how to measure transmitter modulation accuracy, spectral mask, and spectral flatness for IEEE® 802.11be™ waveforms.

Introduction

In this example, you generate an oversampled IEEE 802.11be extremely high throughput (EHT) multi-user (MU) waveform, as defined in IEEE P802.11be/D2.0 [1 on page 3-12]. You can configure the example to generate orthogonal frequency-division multiple-access (OFDMA) waveforms, which transmit data for multiple users over different parts of the band, or non-OFDMA waveforms, which transmit data for a single user over the whole band. You can also introduce in-band distortion and spectral regrowth by using a high-power amplifier (HPA) model. You then perform the transmitter modulation accuracy, required spectral mask, and required spectral flatness measurement on the waveform for the measurement configuration specified in Section 36.3.19 of [1 on page 3-12].

For each user, the example decodes the EHT Data field and measures the error vector magnitude (EVM) to determine the modulation accuracy after downsampling the waveform to baseband sampling rate. Additionally, for the non-OFDMA PPDU type, the example measures the spectral flatness of the recovered waveform. This diagram shows the example workflow.



Simulation Setup

Configure the example to generate two EHT MU packets with a 10 microsecond idle period between each packet.

```
numPackets = 2 ;
idleTime   = 10 ;
```

802.11be Waveform Configuration and Generation

The draft standard defines the EHT MU format for the transmission of non-OFDMA and OFDMA PPDU types. This example supports generation of non-OFDMA and OFDMA EHT MU PPDU types. For more information about the parameterization and generation of IEEE 802.11be EHT MU waveforms, see the "802.11be Waveform Generation" on page 1-18 example.

```
savedState = rng(0); % Set random state
ppduType = OFDMA ; % Set ppduType to OFDMA or Non-OFDMA
```

Configure transmission parameters of an EHT MU packet by using EHT MU configuration object `wlanEHTMUConfig` For an OFDMA PPDU type, create an OFDMA configuration for a 20 MHz EHT MU packet with allocation index 47 as defined in Table 36-34 of [1 on page 3-12]. This allocation has two 52-tone resource units (RUs) and one 106+26-tone multiple resource unit (MRU). This configuration specifies the transmission of a single user per RU. Set the transmission parameters for each user.

```
if strcmp(ppduType,"OFDMA")
    allocationIndex = 47; % Allocation index 47 specifies three RUs and three users
    mcs = [12 5 4]; % Modulation and coding scheme per user
    spatialMapping = ["Direct" "Direct" "Direct"]; % Set spatial mapping property per RU
    apepLength = [1000 500 400]; % A-MPDU length pre-EOF padding in bytes per user
    numSTSs = [1 1 1]; % Number of space-time streams per user
    channelCoding = ["LDPC" "LDPC" "BCC"]; % Set channel coding property per user
    numTx = 1; % Number of transmit antennas

    cfgEHT = wlanEHTMUConfig(allocationIndex);
    chanBW = cfgEHT.ChannelBandwidth;
    numUsers = numel(cfgEHT.User);
    cfgEHT.NumTransmitAntennas = numTx;
    for i = 1:numUsers
        cfgEHT.RU{i}.SpatialMapping = spatialMapping(i);
        cfgEHT.User{i}.APEPLength = apepLength(i);
        cfgEHT.User{i}.MCS = mcs(i);
        cfgEHT.User{i}.NumSpaceTimeStreams = numSTSs(i);
        cfgEHT.User{i}.ChannelCoding = channelCoding(i);
    end
end
```

For the non-OFDMA PPDU type, create a full-band 320 MHz single-user (SU) MIMO configuration and set the transmission parameters of the user.

```
if strcmp(ppduType,"Non-OFDMA")
    chanBW = "CBW320"; % Channel bandwidth
    mcs = 12; % Modulation and coding scheme
    apepLength = 8000; % A-MPDU length pre-EOF padding in bytes
    numTx = 1; % Number of transmit antennas

    cfgEHT = wlanEHTMUConfig(chanBW); % EHT MU configuration object
    numUsers = numel(cfgEHT.User);
    cfgEHT.NumTransmitAntennas = numTx;
    cfgEHT.RU{1}.SpatialMapping = "Direct";
    cfgEHT.User{1}.APEPLength = apepLength;
    cfgEHT.User{1}.MCS = mcs;
    cfgEHT.User{1}.NumSpaceTimeStreams = numTx;
    cfgEHT.User{1}.ChannelCoding = "LDPC";
end
```

To model the effect of an HPA on the waveform and view the out-of-band spectral emissions, the waveform must be oversampled. Generate the waveform using a larger IFFT than required for the nominal baseband rate, resulting in an oversampled waveform.

```
osf = 4; % Oversampling factor
```

Create random bits for all packets.

```
psduLen = psduLength(cfgEHT).*8;
data = cell(1,numUsers);
for i=1:numUsers
    data{i} = randi([0 1],psduLen(i)*numPackets,1);
end
```

Generate the EHT MU waveform for the specified bits and configuration by using the `wlanWaveformGenerator` function, specifying the desired oversampling factor, number of packets, and idle time between each packet.

```
txWaveform = wlanWaveformGenerator(data,cfgEHT, ...
NumPackets=numPackets,IdleTime=idleTime*1e-6, ...
OversamplingFactor=osf);
```

Get the baseband sampling rate of the waveform.

```
fs = wlanSampleRate(chanBW);
disp(['Baseband sampling rate: ' num2str(fs/1e6) ' Msps']);
```

```
Baseband sampling rate: 20 Msps
```

Prepend zeros to the waveform to allow for early timing synchronization.

```
txWaveform = [zeros(round(idleTime*1e-6*fs),numTx); txWaveform];
```

Add Impairments

HPA Modeling

The HPA introduces nonlinear behavior in the form of in-band distortion and spectral regrowth. This example simulates the power amplifiers by using the Rapp model [3 on page 3-12], which introduces AM/AM distortion.

Model the amplifier by using the `dsp.FIRInterpolator` object and configure reduced distortion by specifying a backoff, `hpaBackoff`, such that the amplifier operates below its saturation point. You can increase the backoff to reduce EVM for higher MCS values.

```
pSaturation = 25; % Saturation power of a power amplifier in dBm
hpaBackoff = 16;  % Power amplifier backoff in dB
nonLinearity = comm.MemorylessNonlinearity;
nonLinearity.Method = "Rapp model";
nonLinearity.Smoothness = 3; % p parameter
nonLinearity.LinearGain = -hpaBackoff;
nonLinearity.OutputSaturationLevel = db2mag(pSaturation-30);
txWaveform = nonLinearity(txWaveform);
```

Thermal Noise

Add thermal noise to each transmit antenna by using the `comm.ThermalNoise` object with a noise figure of 6 dB [4 on page 3-12].

```
thNoise = comm.ThermalNoise(NoiseMethod="Noise Figure",SampleRate=fs*osf,NoiseFigure=6);

for i = 1:cfgEHT.NumTransmitAntennas
    txWaveform(:,i) = thNoise(txWaveform(:,i));
end
```

EVM and Spectral Flatness Measurements

Downsampling and Filtering

Resample the oversampled waveform down to baseband for physical layer processing and EVM and spectral flatness measurements, applying a low-pass anti-aliasing filter before downsampling. The impact of the low-pass filter is visible in the spectral flatness measurement. Set the parameters for the anti-aliasing filter so that all active subcarriers are within the filter passband.

Design resampling filter.

```
aStop = 40; % Stopband attenuation
ofdmInfo = wlanEHTOFDMInfo("EHT-Data",cfgEHT,1); % OFDM parameters for the first RU
SCS = fs/ofdmInfo.FFTLength; % Subcarrier spacing
txbw = max(abs(ofdmInfo.ActiveFrequencyIndices))*2*SCS; % Occupied bandwidth
[L,M] = rat(1/osf);
maxLM = max([L M]);
R = (fs-txbw)/fs;
TW = 2*R/maxLM; % Transition width
```

Resample the waveform to baseband.

```
firdec = designMultirateFIR(L,M,TW,aStop,SystemObject=true);
rxWaveform = firdec(txWaveform);
```

Receiver Processing

In this section you detect, synchronize, and extract each packet in `rxWaveform`, and then measure the EVM and spectral flatness. Perform the spectral flatness measurement for non-OFDMA PPDU type. For each packet, perform these steps.

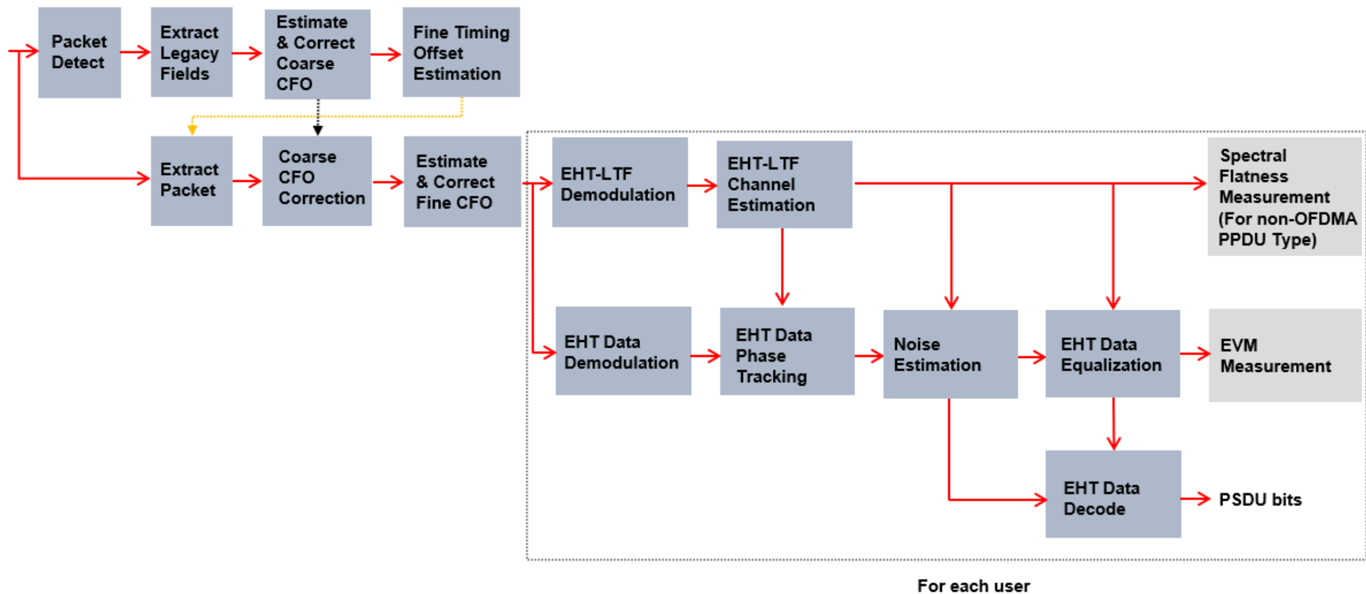
- Detect the start of the packet
- Extract the legacy fields
- Estimate and correct coarse carrier frequency offset (CFO)
- Perform fine symbol timing estimate by using the frequency-corrected legacy fields
- Extract the packet from the waveform by using the fine symbol timing offset
- Correct the extracted packet with the coarse CFO estimate
- Extract the L-LTF, then estimate and correct the fine CFO

For each packet and each user, perform these steps.

- Extract the EHT-LTF and perform channel estimation for each of the transmit streams
- Measure the spectral flatness by using the channel estimate for non-OFDMA PPDU type
- Extract and OFDM demodulate the EHT Data field
- Perform noise estimation by using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers

- Phase-correct and equalize the EHT Data field by using the channel and noise estimates
- For each data-carrying subcarrier in each spatial stream, find the closest constellation point and measure the EVM
- Recover the PSDU by decoding the equalized symbols

This diagram shows the processing chain.



This example performs two different EVM measurements.

1. RMS EVM per user per packet, which comprises averaging the EVM over subcarriers, OFDM symbols, and spatial streams.
2. RMS EVM per subcarrier per spatial stream per user for a packet. Because this configuration maps spatial streams directly to antennas, this measurement can help detect frequency-dependent impairments, which tend to affect individual RF chains differently. This measurement averages the EVM over OFDM symbols only.

Get indices for accessing each field within the time-domain packet.

```
ind = wlanFieldIndices(cfgEHT);
```

Define the minimum detectable length of data, in samples.

```
minPktLen = double(ind.LSTF(2)-ind.LSTF(1))+1;
```

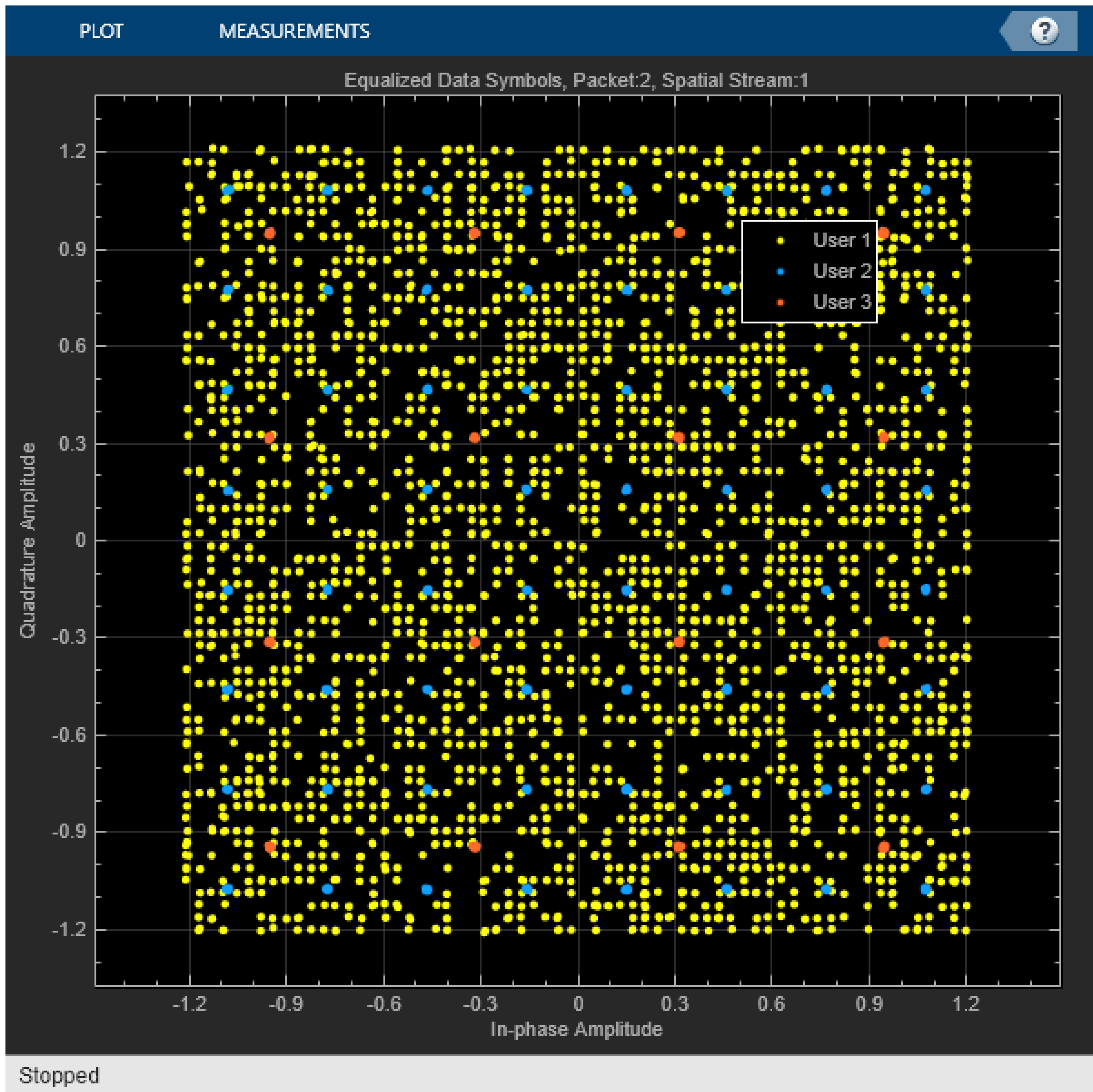
Detect and process packets within the received waveform by using a `while` loop, which performs these steps.

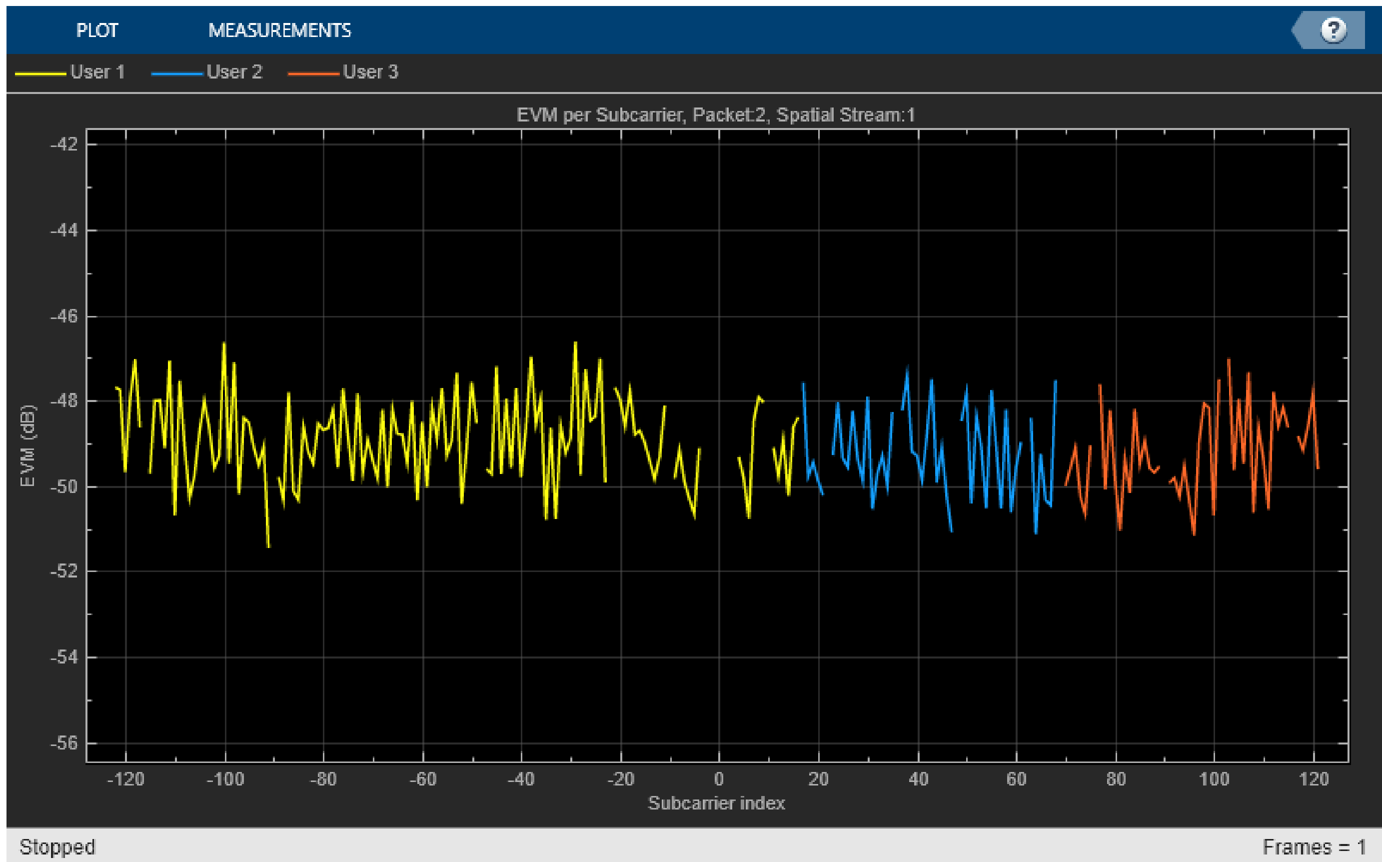
1. Detect a packet by indexing into `rxWaveform` with the sample offset, `searchOffset`
2. Detect and process the first packet within `rxWaveform`
3. Detect and process the next packet by incrementing the sample index offset
4. Repeat until no further packets are detected

```
rxWaveformLength = size(rxWaveform,1);
pktLength = double(ind.EHTData(2));
```

```
rmsEVM = zeros(numPackets,numUsers);
eqSym = cell(1,numUsers);
evmPerSC = cell(1,numUsers);
decodeSuccess = false(numPackets,numUsers);
passSF = false(numPackets,1);
pktOffsetStore = zeros(numPackets,1);

rng(savedState); % Restore random state
pktNum = 0;
searchOffset = 0; % Start at first sample (no offset)
while (searchOffset+minPktLen)<=rxWaveformLength

    % Detect packet and determine coarse packet offset
    pktOffset = wlanPacketDetect(rxWaveform,cfgEHT.ChannelBandwidth,searchOffset);
    % Packet offset from start of the waveform
    pktOffset = searchOffset+pktOffset;
    % Skip packet if L-STF is empty
    if isempty(pktOffset) || (pktOffset<0) || ...
            ((pktOffset+ind.LSIG(2))>rxWaveformLength)
        break;
    end

    % Extract L-STF and perform coarse frequency offset correction
    nonht = rxWaveform(pktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
    coarsefreqOff = wlanCoarseCFOEstimate(nonht,cfgEHT.ChannelBandwidth);
    nonht = frequencyOffset(nonht,fs,-coarsefreqOff);

    % Extract the legacy fields and determine fine packet offset
    lltfOffset = wlanSymbolTimingEstimate(nonht,cfgEHT.ChannelBandwidth);
    pktOffset = pktOffset+lltfOffset; % Determine packet offset

    % If offset is outside the bounds of the waveform, then skip samples
    % and continue searching within remainder of the waveform
    if (pktOffset<0) || ((pktOffset+pktLength)>rxWaveformLength)
        searchOffset = pktOffset+double(ind.LSTF(2))+1;
        continue;
    end

    % Timing synchronization complete; extract the detected packet
    rxPacket = rxWaveform(pktOffset+(1:pktLength),:);
    pktNum = pktNum+1;

    % Apply coarse frequency correction to the extracted packet
    rxPacket = frequencyOffset(rxPacket,fs,-coarsefreqOff);

    % Perform fine frequency offset correction on the extracted packet
    lltf = rxPacket(ind.LLTF(1):ind.LLTF(2),:); % Extract L-LTF
    fineFreqOff = wlanFineCFOEstimate(lltf,cfgEHT.ChannelBandwidth);
    rxPacket = frequencyOffset(rxPacket,fs,-fineFreqOff);

    % Extract EHT-LTF samples, demodulate, and perform channel estimation
    ehtLTF = rxPacket(ind.EHTLTF(1):ind.EHTLTF(2),:);
    for i = 1:numUsers
        ehtLTFDemod = wlanEHTDemodulate(ehtLTF,'EHT-LTF',cfgEHT,i);

        % Estimate channel
        [chanEst,pilotEst] = wlanEHTLTFChannelEstimate(ehtLTFDemod,cfgEHT,i);
```

```
        % Measure spectral flatness for non-OFDMA PPDU type
        if strcmp(ppduType,"Non-OFDMA")
            [passSF(pktNum),deviation,testsc] = wlanSpectralFlatness(chanEst,'EHT',cfgEHT.Channe
            % Plot deviation against limits
            ehtPlotTxSpectralFlatness(deviation,testsc,pktNum);
        end

        % Data demodulate
        rxData = rxPacket(ind.EHTData(1):ind.EHTData(2),:);
        demodSym = wlanEHTDemodulate(rxData,'EHT-Data',cfgEHT,i);

        % Perform pilot phase tracking
        demodSym = wlanEHTTrackPilotError(demodSym,chanEst,cfgEHT,'EHT-Data',i);

        % Estimate noise power in EHT fields
        ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfgEHT,i); % OFDM parameters
        nVarEst = ehtNoiseEstimate(demodSym(ofdmInfo.PilotIndices,:,:),pilotEst,cfgEHT,i);

        % Extract data subcarriers from demodulated symbols and channel
        % estimate
        demodDataSym = demodSym(ofdmInfo.DataIndices,:,:);
        chanEstData = chanEst(ofdmInfo.DataIndices,:,:);

        % Equalize
        [eqSym{i},csi] = ehtEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgEHT,i);

        % Set up EVM measurements
        [EVMPerPkt,EVMPerSC] = evmSetup(cfgEHT,i);

        % Compute RMS EVM over all spatial streams for the packet
        rmsEVM(pktNum,i) = EVMPerPkt(eqSym{i});

        % Compute RMS EVM per subcarrier and spatial stream for the packet
        evmPerSC{i} = EVMPerSC(eqSym{i}); % Nst-by-1-by-Nss

        % Recover data field bits
        rxPSDU = wlanEHTDataBitRecover(eqSym{i},nVarEst,csi,cfgEHT,i);

        if isequal(rxPSDU,data{i}((1:psduLen(i))+(pktNum-1)*psduLen(i)))
            decodeSuccess(pktNum,i) = true;
        end

    end

    % Plot equalized constellation and RMS EVM per subcarrier
    ehtTxEVMConstellationPlots(eqSym,evmPerSC,cfgEHT,pktNum);

    % Store the offset of each packet within the waveform
    pktOffsetStore(pktNum) = pktOffset;

    % Increment waveform offset and search remaining waveform for a packet
    searchOffset = pktOffset+pktLength+minPktLen;
end
```


PLOT
MEASUREMENTS
Equalized Data Symbols, Packet:2, Spatial Stream:1
User 1
User 2
User 3
Quadrature Amplitude
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
In-phase Amplitude
Stopped



Set the unit of EVM to decibel or percentage.

```
evmUnit = Decibel ;
```

Display tables for decode status and measurement summary.

```
ehtMeasurementSummary(cfgEHT,rmsEVM,decodeSuccess,pktOffsetStore,passSF,evmUnit);
```

```
Decode Status

    Packet Number    Start Index     User 1       User 2       User 3
    _____________    ___________    _________    _________    _________

          1               74        "Success"    "Success"    "Success"
          2             9074        "Success"    "Success"    "Success"

Measurement Summary

    Packet Number    User 1 EVM (dB)    User 2 EVM (dB)    User 3 EVM (dB)    Packet EVM (dB)
    _____________    _______________    _______________    _______________    _______________

          1              -49.438            -49.408            -49.607            -49.484
          2               -48.78            -49.123            -49.103            -49.001

Average EVM for 3 users:
  User 1: -49.10dB
```

```
User 2: -49.26dB
User 3: -49.35dB
All users: -49.24dB
```

Spectral Mask Measurement

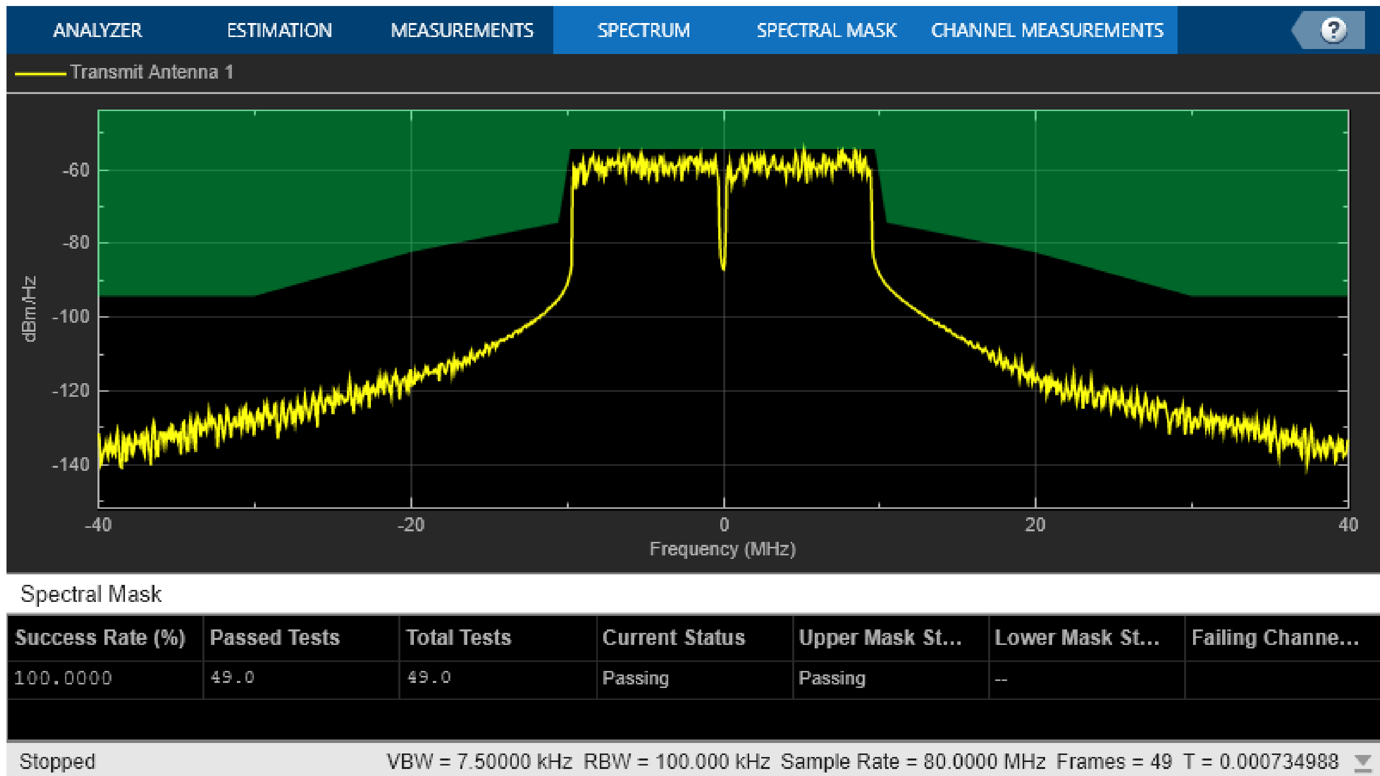
In this section, you measure the spectral mask of the filtered and impaired waveform after HPA modeling. The transmitter spectral mask test [5 on page 3-12] uses a time-gated spectral measurement of the EHT Data field. The example extracts the EHT Data field of each packet from the oversampled waveform by using the start indices of each packet within the baseband waveform. Any delay introduced in the baseband processing chain used to determine the packet indices must be accounted for when gating the EHT Data field within `txWaveform`. Concatenate the extracted EHT Data fields in preparation for measurement.

```
startIdx = osf*(ind.EHTData(1)-1)+1; % Upsampled start of EHT Data
endIdx = osf*ind.EHTData(2);         % Upsampled end of EHT Data
delay = grpdelay(firdec,1);          % Group delay of downsampling filter
idx = zeros(endIdx-startIdx+1,pktNum);
for i = 1:pktNum
    % Start of packet in txWaveform
    pktOffset = round(osf*pktOffsetStore(i))-delay;
    % Indices of EHT Data in txWaveform
    idx(:,i) = (pktOffset+(startIdx:endIdx));
end
gatedEHTTData = txWaveform(idx(:),:);
```

The 802.11be standard specifies the spectral mask relative to the peak power spectral density (PSD). This plot overlays the required mask with the measured PSD.

```
if pktNum>0
    ehtSpectralMaskTest(gatedEHTTData,fs,osf);
end
```

```
Spectral mask passed
```



Conclusion and Further Exploration

This example shows how to measure and plot these properties of an IEEE 802.11be waveform.

- Spectral flatness for non-OFDMA PPDU type
- RMS EVM per subcarrier
- Equalized constellation
- Spectral mask

The HPA model introduces significant in-band distortion and spectral regrowth, which is visible in the EVM results, noisy constellation, and out-of-band emissions in the spectral mask plot. Try increasing the HPA backoff and observe the improved EVM, constellation, and lower out-of-band emissions. The downsampling (to bring the waveform to baseband for processing) stage includes filtering. The filter response affects the spectral flatness measurement. The ripple in the spectral flatness measurement is due to downsampling to baseband. Try using a different filter or changing the stop-band attenuation and observe the impact on the spectral flatness. For meaningful EVM results generate at least 20 EHT MU packets as specified in Section 36.3.19.4.4 of [1 on page 3-12].

References

1 IEEE P802.11be™/D2.0 Draft 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 8: Enhancements for extremely high throughput (EHT).

2 IEEE P802.11ax™-2021. Draft 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 1: Enhancements for High Efficiency WLAN.

3 Loc and Cheong. IEEE P802.11 Wireless LANs. TGac Functional Requirements and Evaluation Methodology Rev. 16. 2011-01-19.

4 Perahia, E., and R. Stacey. Next Generation Wireless LANs: 802.11n and 802.11ac. 2nd Edition. United Kingdom: Cambridge University Press, 2013.

5 Archambault, Jerry, and Shravan Surineni. "IEEE 802.11 spectral measurements using vector signal analyzers." RF Design 27.6 (2004): 38-49.

802.11ba WUR Waveform Generation and Analysis

This example shows how to generate IEEE® 802.11ba™ wake-up radio (WUR) packet waveforms. The example also demonstrates how to measure the transmit spectrum mask and spectral flatness.

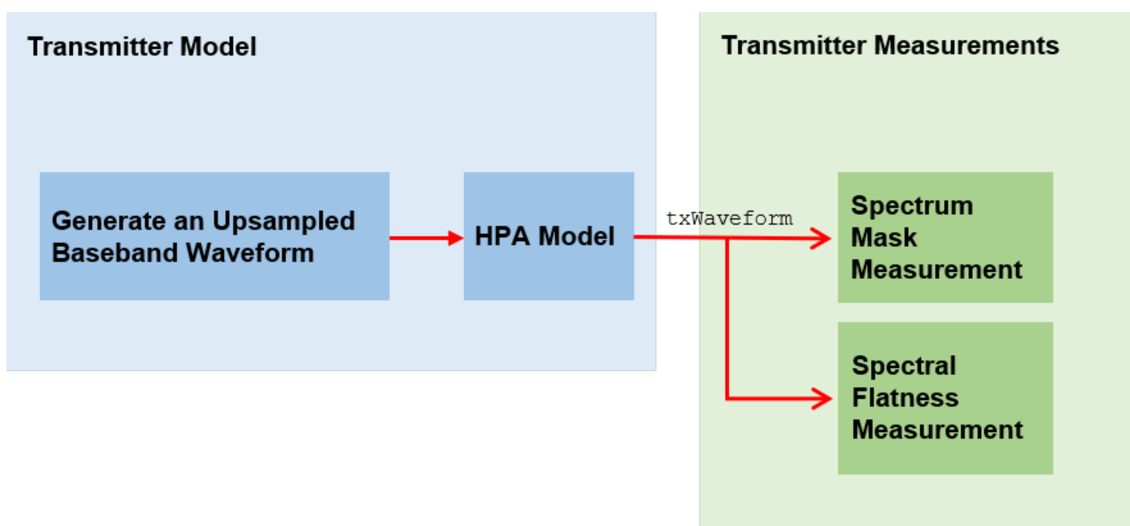
Introduction

The 802.11ba standard [1], referred to as wake-up radio (WUR), defines a mechanism to enable IEEE 802.11 stations (STAs) to operate at extremely low power consumption and to react to incoming traffic with low latency through a wake-up signal. This standard defines two types of WUR PPDU format.

- WUR Basic PPDU with 20 MHz channel bandwidth (one subchannel)
- WUR FDMA PPDU with 40 MHz (two subchannels) or 80 MHz (four subchannels) contiguous channel bandwidths

Each 20 MHz subchannel can transmit data for a single user. For each 20 MHz subchannel, the WUR physical layer (PHY) provides support for data rates of 62.5 kb/s and 250 kb/s, indicated as low data rate (LDR) and high data rate (HDR), respectively. Additionally, Annex AC of [1] provides three examples of symbol sequences and cyclic shift durations (CSDs) at specified data rates.

This example generates 5 WUR packets with a 20 MHz channel bandwidth and 10 microsecond gaps between each of the packets. The example oversamples the waveform using a larger inverse fast Fourier transform (IFFT) length than required for the nominal baseband rate and does not perform spectral filtering. The example then models the effects of a high-power amplifier (HPA), which introduces in-band distortion and spectral regrowth. After HPA modelling, the example extracts the non-WUR portion of the WUR PHY preamble, comprising the L-STF, L-LTF, L-SIG, BPSK-Mark1, and BPSK-Mark2 fields, and the WUR portion, comprising the WUR-Sync and WUR-Data fields, from the generated waveform. The example then performs two transmit spectrum mask measurements for the non-WUR portion of WUR PHY preamble and WUR portion, respectively, and measures the spectral flatness for the WUR portion. This figure shows the workflow contained in this example.



Simulation Setup

Configure the example to generate 5 WUR packets with a 10 microsecond idle period between each packet.

```
numPackets = 5;
idleTime = 10; % In microseconds
```

802.11ba WUR Waveform Configuration and Generation

Create a WUR configuration object for a 20 MHz transmission.

```
numSubchannels = 1; % Number of subchannels
cfgWUR = wlanWURConfig(numSubchannels); % Create WUR packet configuration
cfgWUR.NumTransmitAntennas = 1; % Number of transmit antennas
fs = wlanSampleRate(cfgWUR); % Get the nominal baseband sample rate
osf = 2; % Oversampling factor
```

Parameterize the transmission by setting WUR configuration object properties for each user. The waveform generator function uses only the first `cfgWUR.NumUsers` elements of the `cfgWUR` cell array to generate the corresponding WUR packets.

```
psdu = cell(1,cfgWUR.NumUsers);
psduLength = [5,10,15,20]; % Bytes, 1 to 22 inclusive for each 20 MHz subchannel
dataRate = {'LDR','HDR','LDR','HDR'};
symDesign = {'Example1','Example2','Example1','Example2'};
rng(0); % Set random state
for i = 1:cfgWUR.NumUsers
    cfgWUR.Subchannel{i}.PSDULength = psduLength(i);
    psdu{i} = randi([0 1],psduLength(i)*8,1,'int8');
    cfgWUR.Subchannel{i}.DataRate = dataRate{i};
    cfgWUR.Subchannel{i}.SymbolDesign = symDesign{i};
end
```

Generate the WUR waveform for the specified bits and configuration by using the `wlanWaveformGenerator` function, setting the number of packets, idle time between each packet, and oversampling factor.

```
txWaveform = wlanWaveformGenerator(psdu,cfgWUR,'NumPackets',numPackets,'IdleTime',idleTime*1e-6,
```

Add Impairments

The HPA introduces nonlinear behaviour in the form of in-band distortion and spectral regrowth. This example simulates the power amplifiers by using the Rapp model, which introduces AM/AM distortion [2]. Model the amplifier by using the `comm.MemorylessNonlinearity` object, and configure reduced distortion by specifying a back-off, `hpaBackoff`, such that the amplifier operates below its saturation point.

```
pSaturation = 25; % Saturation power of a power amplifier in dBm
hpaBackoff = 15; % Power amplifier backoff in dB
nonLinearity = comm.MemorylessNonlinearity;
nonLinearity.Method = 'Rapp model';
nonLinearity.Smoothness = 3; % p parameter
nonLinearity.LinearGain = -hpaBackoff;
nonLinearity.OutputSaturationLevel = db2mag(pSaturation-30);
```

Apply the HPA model to the waveform.

```
txWaveform = nonLinearity(txWaveform);
```

Add thermal noise to each transmit antenna by using the `comm.ThermalNoise` object with a noise figure of 6 dB [3].

```
thNoise = comm.ThermalNoise('NoiseMethod','Noise Figure','SampleRate',fs*osf,'NoiseFigure',6);

for i = 1:cfgWUR.NumTransmitAntennas
    txWaveform(:,i) = thNoise(txWaveform(:,i));
end
```

Transmit Spectrum Mask and Spectral Flatness Measurement

Get indices for accessing WUR fields within the time-domain packet.

```
ind = wlanFieldIndices(cfgWUR,'OversamplingFactor',osf);
% Get the number of samples in the idle time.
numIdle = osf*fs*idleTime*1e-6;
% Define the length of a WUR packet, in samples.
pktLength = double(max(ind.WURData(:,2)))+numIdle;
% Define the length of non-WUR portion of the WUR PHY preamble, composed of the
% L-STF, L-LTF, L-SIG, BPSK-Mark1 and BPSK-Mark2 fields, in samples.
idxPreamble = zeros(ind.BPSKMark2(2),numPackets,'uint32');
% Define the length of the WUR-Sync and WUR-Data fields, in samples.
idxWUR = zeros(max(ind.WURData(:,2))-ind.WURSync(1,1)+1,numPackets,'uint32');
```

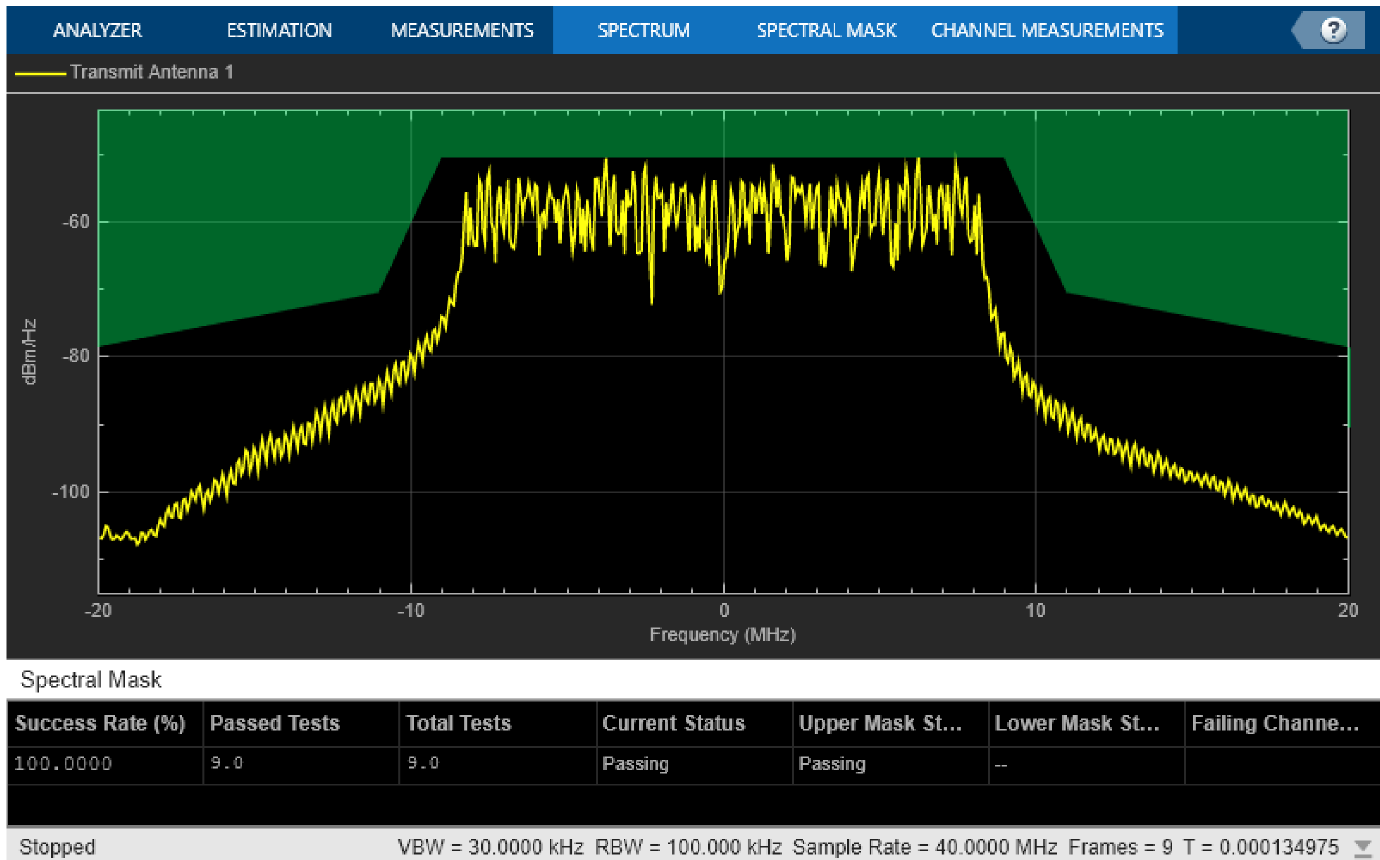
Extract the non-WUR portion of the WUR PHY preamble and WUR portion of each packet within the waveform.

```
pktOffset = 0; % Start at first sample (no offset)
for i = 1:numPackets
    % Indices of WUR preamble fields
    idxPreamble(:,i) = pktOffset+(1:ind.BPSKMark2(2));
    % Indices of WUR-Sync and WUR-Data fields
    idxWUR(:,i) = pktOffset+(ind.WURSync(1,1):max(ind.WURData(:,2)));
    % Packet offset to generate the end index of each packet
    pktOffset = i*pktLength;
end
preamFields = txWaveform(idxPreamble(:),:);
wurFields = txWaveform(idxWUR(:),:);
```

Measure the transmit spectrum mask of the non-WUR portion of the WUR PHY preamble, as specified in Section 30.3.12.1 of [1].

```
if numPackets>0
    helperSpectralMaskTest(preamFields,fs,osf);
end
```

```
Spectrum mask passed
```



Based on the values of limits (in dBr) relative to maximum spectral density of the signal and corresponding frequency offsets (in MHz) provided in Section 30.3.12.1 of [1], define the transmit spectrum masks of the WUR-Sync and WUR-Data fields for WUR basic and FDMA PPDUs are.

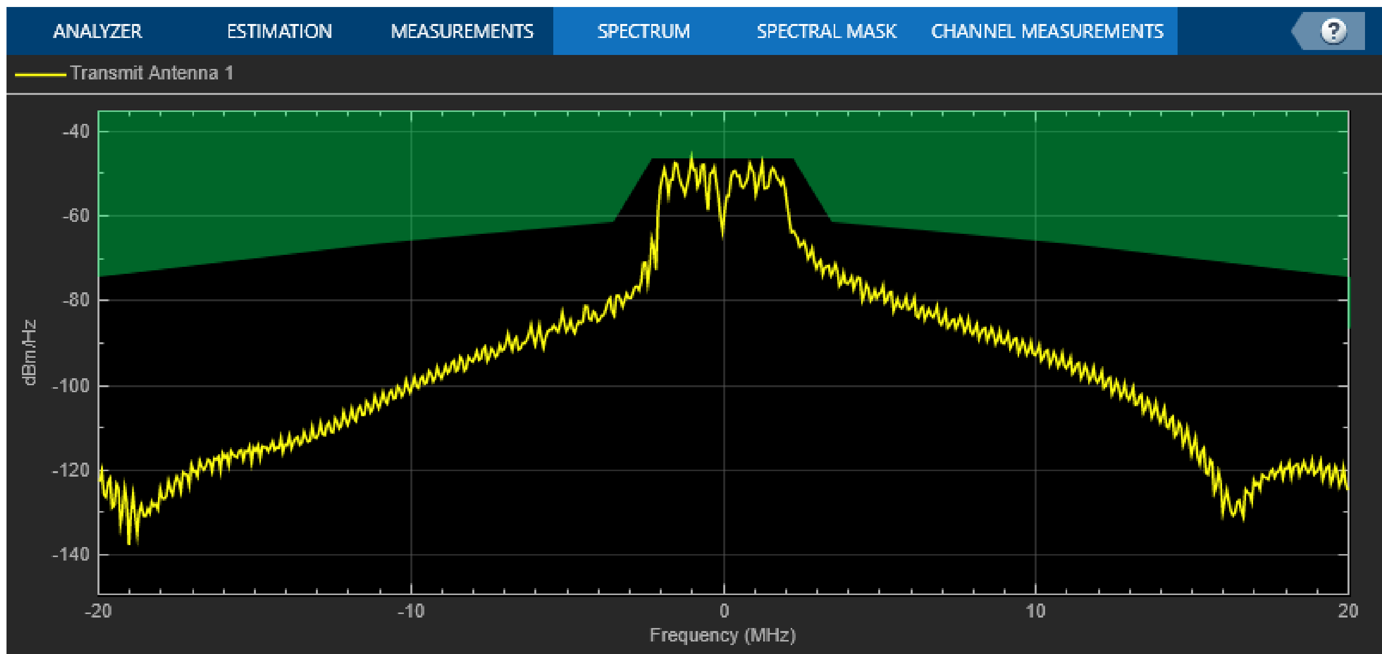
```
switch numSubchannels
    case 1 % WUR Basic PPDU
        dBrLimits = [-40 -40 -28 -20 -15 0 ...
            0 -15 -20 -28 -40 -40];
        fLimits = [-Inf -30 -20 -11 -3.5 -2.25 ...
            2.25 3.5 11 20 30 Inf];
    case 2 % 40 MHz WUR FDMA PPDU
        dBrLimits = [-40 -40 -28 -20 -15 0 0 -15 -20 ...
            -20 -15 0 0 -15 -20 -28 -40 -40];
        fLimits = [-Inf -60 -40 -21 -13.5 -12.25 -7.75 -6.5 -1 ...
            1 6.5 7.75 12.25 13.5 21 40 60 Inf];
    otherwise % 80 MHz WUR FDMA PPDU
        dBrLimits = [-40 -40 -28 -20 -15 0 0 -15 -20 -20 -15 0 0 -15 -20 ...
            -20 -15 0 0 -15 -20 -20 -15 0 0 -15 -20 -28 -40 -40];
        fLimits = [-Inf -120 -80 -41 -33.5 -32.25 -27.75 -26.5 -21 -19 -13.5 -12.25 -7.75 -6.5 -
            1 6.5 7.75 12.25 13.5 19 21 26.5 27.75 32.25 33.5 41 80 120 Inf];
end
```

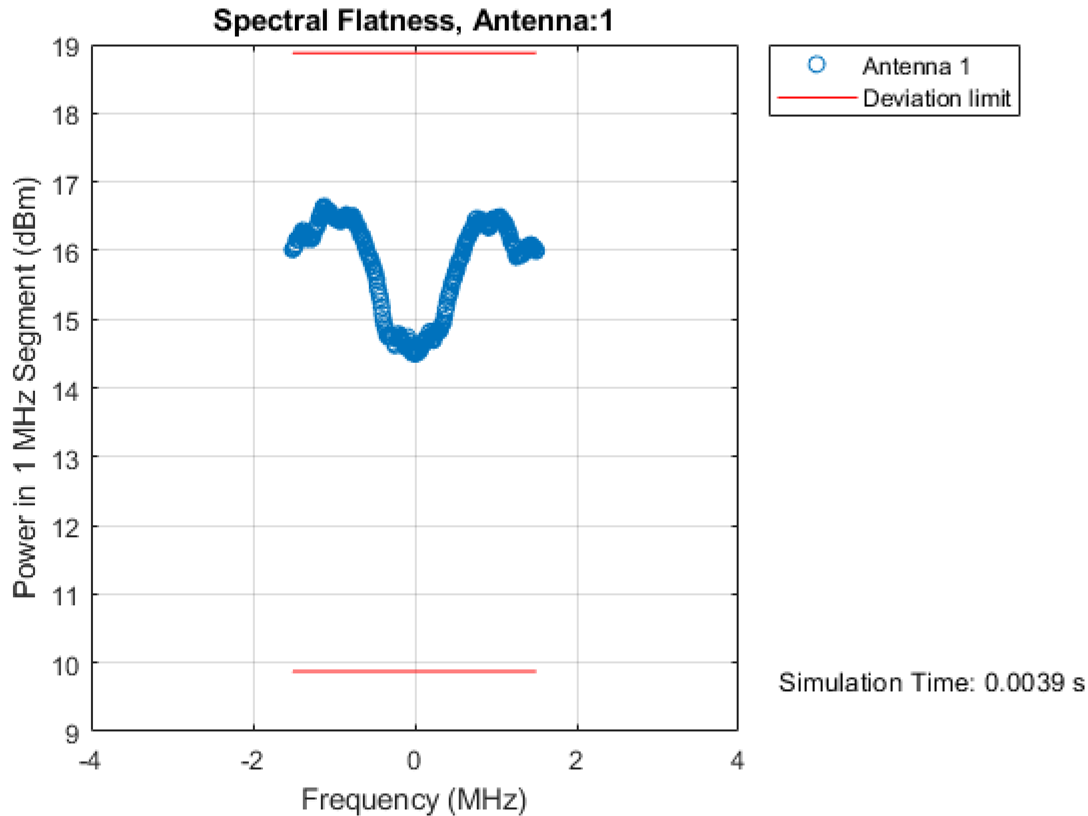
Measure the transmit spectrum mask and spectral flatness of the WUR-Sync and WUR-Data fields, as specified in Section 30.3.12.1 and Section 30.3.12.2 of [1]. The `wurTxSpectralFlatnessMeasurement` function measures the spectral flatness by comparing the power in any contiguous 1 MHz segment within the center 4 MHz of each 20 MHz channel to the

total transmitted power in the center 4 MHz and measures the transmitted power in the manner described in Section 5.4.3.2.1 of [4] for equipment with continuous and non-continuous transmissions.

```
if numPackets>0
    helperSpectralMaskTest(wurFields,fs,osf,dBrLimits,fLimits);
    isPassed = wurTxSpectralFlatnessMeasurement(wurFields,fs,osf);
    if isPassed
        fprintf('    Spectral flatness passed\n');
    else
        fprintf('    Spectral flatness failed\n');
    end
end
```

```
Spectrum mask passed
Spectral flatness passed
```





Conclusion and Further Exploration

This example shows how to generate WUR waveforms as specified in the IEEE 802.11ba standard, and measure these quantities.

- Transmitter spectrum mask
- Spectral flatness

The HPA model introduces significant in-band distortion and spectral regrowth, which is visible in the spectral mask plot. Try increasing the value of the HPA backoff and observe lower out-of-band emissions. The various patterns of ripple and deviation limits in the spectral flatness measurement is due to the different Multi-Carrier On-Off Keying (MC-OOK) On symbol designs, such as MC-OOK On symbol sequences and CSD values applied. The spectral flatness test will fail for a 20 MHz subchannel if the subchannel in the generated waveform is configured with symbol design and data rate set to `'Example3'` and `'HDR'`, respectively. Try using a different MC-OOK On symbol design and observe the impact on the spectral flatness.

References

1 IEEE Std 802.11ba™-2021 - 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 3: Wake-Up Radio Operation.

2 R. Porat, et al. TGax Evaluation Methodology, IEEE 11/14-0571r12. 2016-01-21.

3. Perahia, E., and R. Stacey. Next Generation Wireless LANs: 802.11n and 802.11ac. 2nd Edition. United Kingdom: Cambridge University Press, 2013. Archambault, Jerry, and Shravan Surineni. "IEEE 802.11 spectral measurements using vector signal analyzers." RF Design 27.6 (2004): 38-49.
4. ETSI EN 300 328 V2.1.1. Wideband transmission systems; Data transmission equipment operating in the 2,4 GHz ISM band and using wide band modulation techniques; Harmonised Standard covering the essential of article 3.2 of Directive 2014/53EU. 2016-11.

Copyright 2021-2022 The MathWorks, Inc.

802.11ad Waveform Generation with Beamforming

This example shows how to beamform an IEEE® 802.11ad™ DMG waveform with a phased array using WLAN Toolbox™ and Phased Array System Toolbox™.

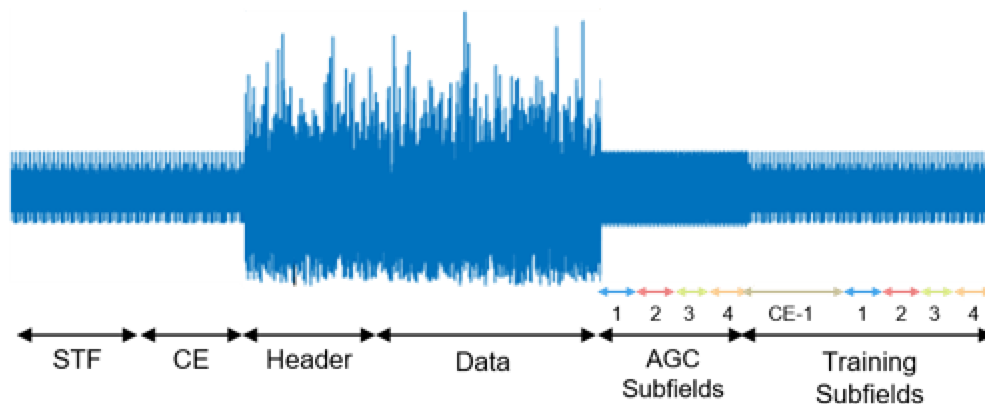
Introduction

IEEE 802.11ad [1 on page 3-25] defines the directional multi-gigabit (DMG) transmission format operating at 60 GHz. To overcome the large path loss experienced at 60 GHz, the IEEE 802.11ad standard is designed to support directional beamforming. By using phased antenna arrays you can apply an antenna weight vector (AWV) to focus the antenna pattern in the desired direction. Each packet is transmitted on all array elements, but the AWV applies a phase shift to each element to steer the transmission. The quality of a communication link can be improved by appending optional training fields to DMG packets, and testing different AWVs at the transmitter or receiver. This process is called beam refinement.

A DMG packet consists of the following fields:

1. STF - The short training field, which is used for synchronization.
2. CE - Channel estimation field, which is used for channel estimation.
3. Header - The signaling field, which the receiver decodes to determine transmission parameters.
4. Data - The data field, which carries the user data payload.
5. AGC Subfields - Optional automatic gain control (AGC) subfields, used for beam refinement.
6. Training Subfields - Optional training subfields, used for beam refinement.

The STF and CE fields form the preamble. The preamble, header, and data fields of a DMG packet are transmitted with the same AWV. For transmitter beam refinement training, up to 64 training (TRN) subfields can be appended to the packet. Each TRN subfield is transmitted using a different AWV. This allows the performance of up to 64 different AWVs to be measured, and the AWV for the preamble, header, and data fields to be refined for subsequent transmissions. CE subfields are periodically transmitted, one for every four TRN subfields, among the TRN subfields. Each CE subfield is transmitted using the same AWV as the preamble. To allow the receiver to reconfigure AGC before receiving the TRN subfields, the TRN subfields are preceded by AGC subfields. For each TRN subfield, an AGC subfield is transmitted using the same AWV applied to the individual TRN subfield. This allows a gain to be set at the receiver, suitable to measuring all TRN subfields. The diagram below shows the packet structure with four AGC and TRN subfields numbered and highlighted. Therefore, four AWVs are tested as part of beam refinement. The same AWVs are applied to AGC and TRN subfields with the same number.



This example simulates transmitter training by applying different AWVs to each of the training subfields to steer the transmission in multiple directions. The strength of each training subfield is evaluated at a receiver by evaluating the far-field plane wave to determine which transmission AWV is optimal. This simulation does not include a channel or path loss.

This example requires “WLAN Toolbox” and “Phased Array System Toolbox”.

Waveform Specification

The waveform is configured for a DMG packet transmission with single-carrier modulation (SC) physical layer, a 100-byte physical layer service data unit (PSDU), and four transmitter training subfields. The four training subfields allow four AWVs to be tested for beam refinement. Using the function `wlanDMGConfig`, create a DMG configuration object. A DMG configuration object specifies transmission parameters.

```
dmg = wlanDMGConfig;
dmg.MCS = 1;              % Single-carrier modulation
dmg.TrainingLength = 4;   % Use 4 training subfields
dmg.PacketType = 'TRN-T'; % Transmitter training
dmg.PSDULength = 100;     % Bytes
```

Beamforming Specification

The transmitter antenna pattern is configured as a 16-element uniform linear array with half-wavelength spacing. Using the objects `phased.ULA` (Phased Array System Toolbox) and `phased.SteeringVector` (Phased Array System Toolbox), create the phased array and the AWVs. The location of the receiver for evaluating the transmission is specified as an offset from the boresight of the transmitter.

```
receiverAz = 6; % Degrees off the transmitter's boresight
```

A uniform linear phased array with 16 elements is created to steer the transmission.

```
N = 16;                          % Number of elements
c = physconst('LightSpeed'); % Propagation speed (m/s)
fc = 60.48e9;                    % Center frequency (Hz)
lambda = c/fc;                   % Wavelength (m)
d = lambda/2;                    % Antenna element spacing (m)
TxArray = phased.ULA('NumElements',N,'ElementSpacing',d);
```

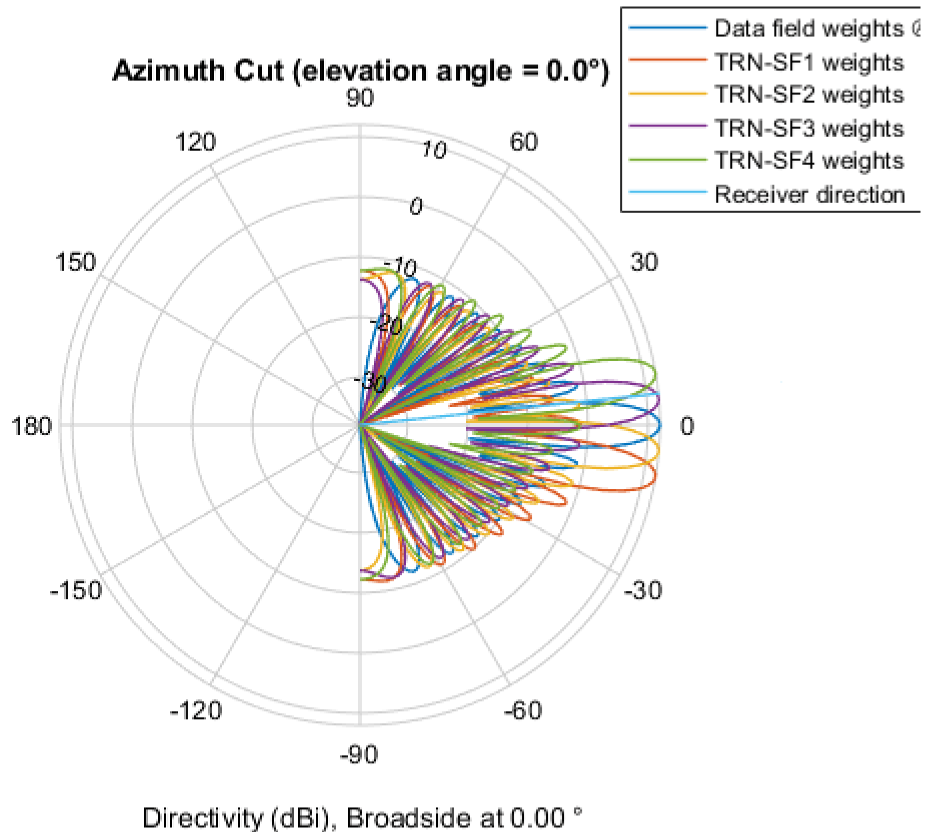
The AWVs are created using a `phased.SteeringVector` (Phased Array System Toolbox) object. Five steering angles are specified to create five AWVs, one for the preamble and data fields, and one

for each of the four the training subfields. The preamble and data fields are transmitted at boresight. The four training subfields are transmitted at angles around boresight.

```
% Create a directional steering vector object
SteeringVector = phased.SteeringVector('SensorArray',TxArray);

% The directional angle for the preamble and data is 0 degrees azimuth, no
% elevation, therefore at boresight. [Azimuth; Elevation]
preambleDataAngle = [0; 0];

% Each of the four training fields uses a different set of weights to steer
% to a slightly different direction. [Azimuth; Elevation]
trnAngle = [[-10; 0] [-5; 0] [5; 0] [10; 0]];

% Generate the weights for all of the angles
weights = SteeringVector(fc,[preambleDataAngle trnAngle]);

% Each row of the AWV is a weight to apply to a different antenna element
preambleDataAWV = conj(weights(:,1)); % AWV used for preamble, data and CE fields
trnAWV = conj(weights(:,2:end));      % AWV used for each TRN subfield
```

Using the plotArrayResponse helper function, the array response shows the direction of the receiver is most aligned with the direction of training subfield TRN-SF3.

```
plotArrayResponse(TxArray,receiverAz,fc,weights);
```



Generate Baseband Waveform

Use the configured DMG object and a PSDU filled with random data as inputs to the waveform generator, `wlanWaveformGenerator`. The waveform generator modulates PSDU bits according to a format configuration.

```
% Create a PSDU of random bits
s = rng(0); % Set random seed for repeatable results
psdu = randi([0 1],dmg.PSDULength*8,1);

% Generate packet
tx = wlanWaveformGenerator(psdu,dmg);
```

Apply Weight Vectors to Each Field

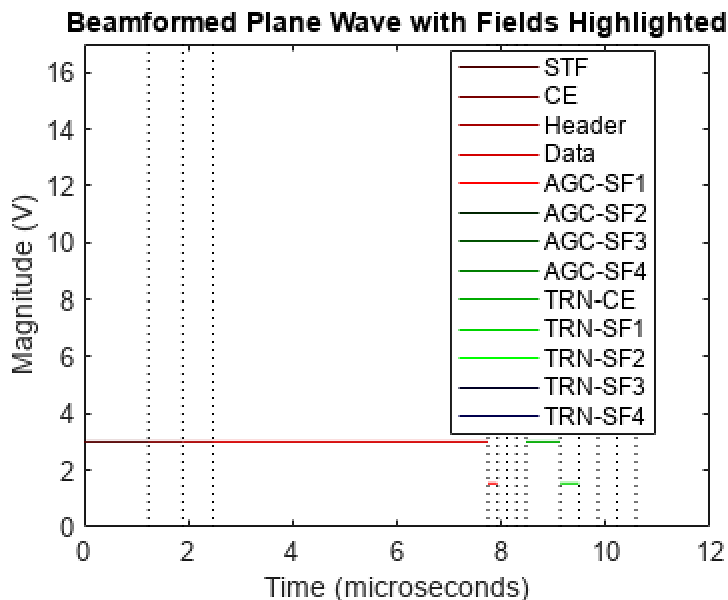
A `phased.Radiator` (Phased Array System Toolbox) object is created to apply the AWVs to the waveform, combine the radiated signal from each element to form a plane wave, and determine the plane wave at the angle of interest, `receiverAz`. Each portion of the DMG waveform `tx` is passed through the Radiator with a specified set of AWVs, and the angle at which to evaluate the plane wave.

```
Radiator = phased.Radiator;
Radiator.Sensor = TxArray;        % Use the uniform linear array
Radiator.WeightsInputPort = true; % Provide AWV as argument
Radiator.OperatingFrequency = fc; % Frequency in Hertz
Radiator.CombineRadiatedSignals = true; % Create plane wave

% The plane wave is evaluated at a direction relative to the radiator
steerAngle = [receiverAz; 0]; % [Azimuth; Elevation]

% The beamformed waveform is evaluated as a plane wave at the receiver
planeWave = zeros(size(tx));

% Get indices for fields
ind = wlanFieldIndices(dmg);

% Get the plane wave while applying the AWV to the preamble, header, and data
idx = (1:ind.DMGData(2));
planeWave(idx) = Radiator(tx(idx),steerAngle,preambleDataAWV);

% Get the plane wave while applying the AWV to the AGC and TRN subfields
for i = 1:dmg.TrainingLength
    % AGC subfields
    agcsfIdx = ind.DMGAGCSubfields(i,1):ind.DMGAGCSubfields(i,2);
    planeWave(agcsfIdx) = Radiator(tx(agcsfIdx),steerAngle,trnAWV(:,i));
    % TRN subfields
    trnsfIdx = ind.DMGTRNSubfields(i,1):ind.DMGTRNSubfields(i,2);
    planeWave(trnsfIdx) = Radiator(tx(trnsfIdx),steerAngle,trnAWV(:,i));
end

% Get the plane wave while applying the AWV to the TRN-CE
for i = 1:dmg.TrainingLength/4
    trnceIdx = ind.DMGTRNCE(i,1):ind.DMGTRNCE(i,2);
    planeWave(trnceIdx) = Radiator(tx(trnceIdx),steerAngle,preambleDataAWV);
end
```

Evaluate the Beamformed Waveform

The helper function plotDMGWaveform plots the magnitude of the beamformed plane wave. When evaluating the magnitude of the beamformed plane wave we can see that the fields beamformed in the direction of the receiver are stronger than other fields.

```
plotDMGWaveform(planeWave,dmg,'Beamformed Plane Wave with Fields Highlighted');
```




```
rng(s); % Restore random state
```

Conclusion

This example showed how to generate an IEEE 802.11ad DMG waveform and apply AWVs to different portions of the waveform. The example uses WLAN Toolbox to generate a standard-compliant waveform, and Phased Array System Toolbox to apply the AWVs and evaluate the magnitude of the resultant plane wave in the direction of a receiver.

Selected Bibliography

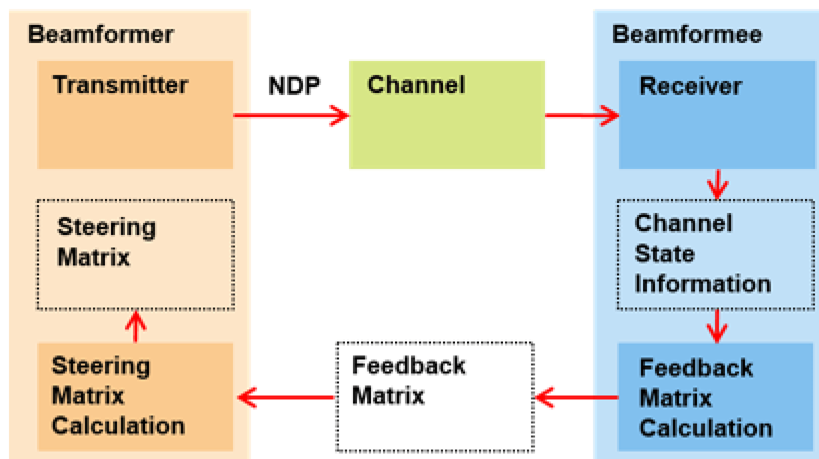
1 IEEE Std 802.11™-2020 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

802.11ac Transmit Beamforming

This example shows how to improve the performance of an IEEE® 802.11ac™ link by beamforming the transmission when channel state information is available at the transmitter.

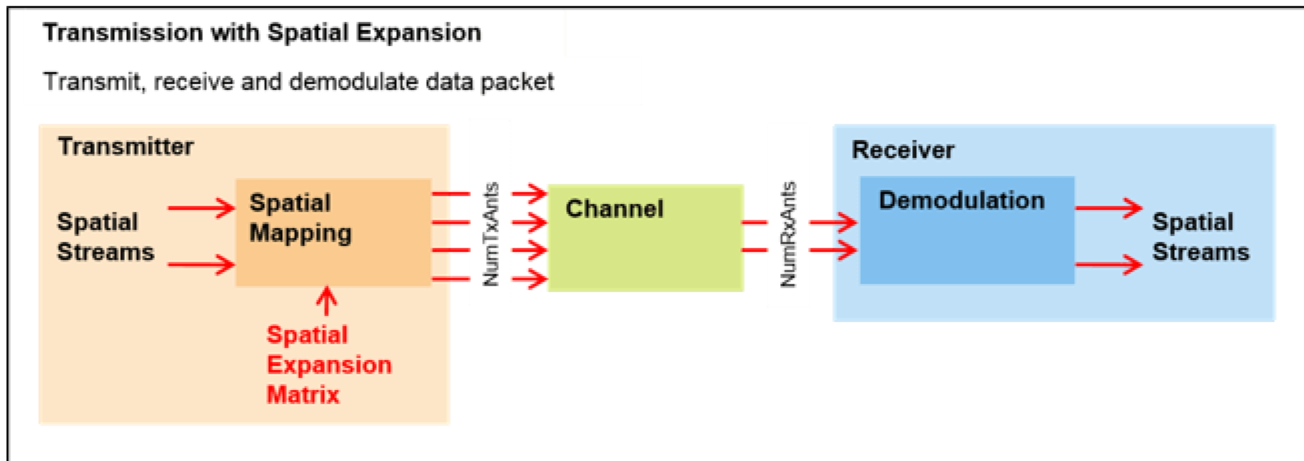
Introduction

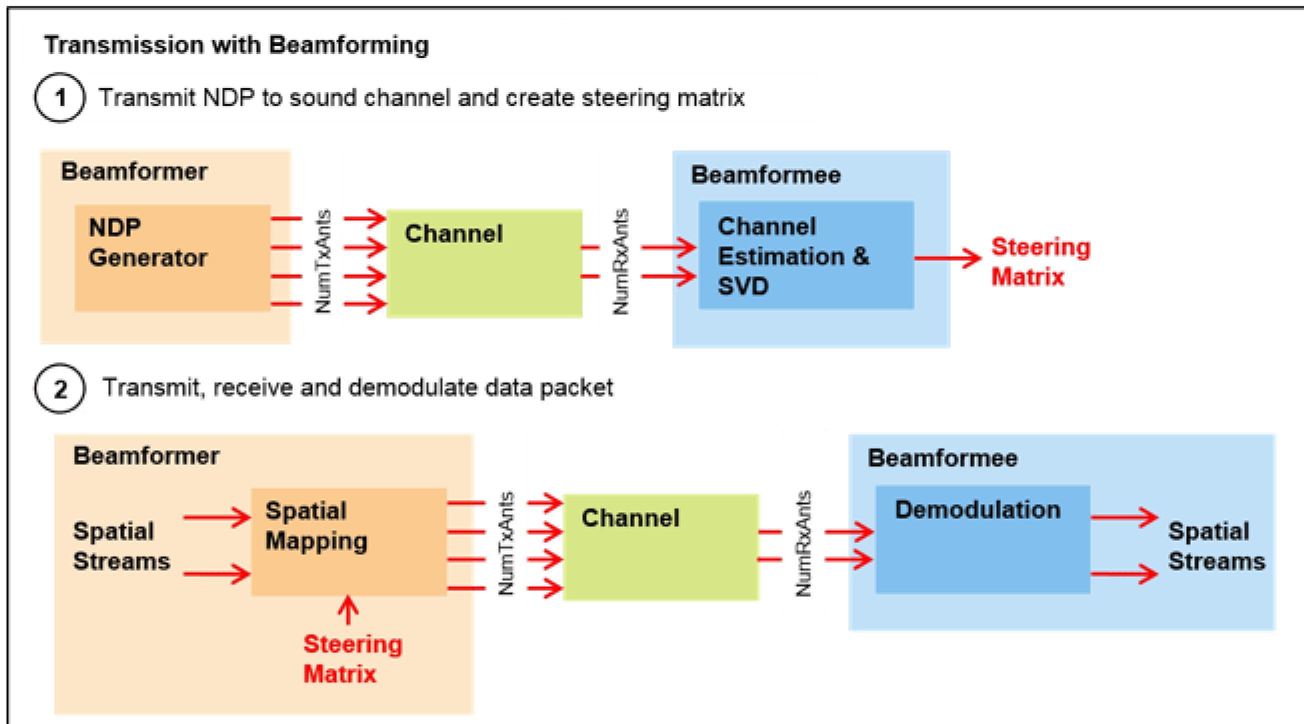
Transmit beamforming focuses energy towards a receiver to improve the SNR of a link. In this scheme the transmitter is called a beamformer and the receiver is called a beamformee. A steering matrix is used by the beamformer to direct the energy to the beamformee. The steering matrix is calculated using channel state information obtained through channel measurements. In IEEE 802.11ac [1 on page 3-35] these measurements are obtained by sounding the channel between beamformer and beamformee. To sound the channel the beamformer sends a null data packet (NDP) to the beamformee. The beamformee uses the channel information provided by sounding to calculate a feedback matrix. This matrix is fed back to the beamformer in a compressed format. The beamformer can then use the feedback matrix to create a steering matrix and beamform transmissions to the beamformee. The process of forming the steering matrix is shown in this diagram.



In IEEE 802.11ac the single user beamformee capability is not mandatory. Therefore, a multi-antenna transmitter may have to use a different scheme to transmit packets to a receiver which cannot act as a beamformee. One such scheme is spatial expansion. Spatial expansion allows space-time streams to be transmitted on a greater number of transmit antennas. Using spatial expansion can provide a small transmit diversity gain in channels with flat fading when compared to directly mapping space-time streams to transmit antennas [2 on page 3-35].

This example demonstrates the benefits of transmit beamforming for a 4x2 MIMO configuration between a transmitter and receiver, with two space-time streams. First, the example demonstrates the operation of a receiver which is not capable of being a beamformee by measuring the signal quality of a transmission made using spatial expansion. To show the benefits of transmit beamforming the example measures the signal quality of a transmission over the same channel realization using transmit beamforming and compares the performance of the two schemes. The stages are shown in the diagram below.





Waveform Configuration

This examples simulates a 4x2 MIMO configuration with 2 space-time streams.

```
NumTxAnts = 4;  % Number of transmit antennas
NumSTS = 2;     % Number of space-time streams
NumRxAnts = 2;  % Number of receive antennas
```

The format specific configuration of a VHT waveform is described using a VHT format configuration object. Configure the waveform with a 20 MHz bandwidth and the MIMO configuration specified above.

```
cfgVHT = wlanVHTConfig;
cfgVHT.ChannelBandwidth = 'CBW20';
cfgVHT.APEPLength = 4000;
```

```
cfgVHT.NumTransmitAntennas = NumTxAnts;
cfgVHT.NumSpaceTimeStreams = NumSTS;
cfgVHT.MCS = 4; % 16-QAM, rate 3/4
```

Channel Configuration

This example uses a TGac channel model with delay profile Model-B. The channel realization is controlled with a seed to allow repeatability.

```
tgacChannel = wlanTGacChannel;
tgacChannel.DelayProfile = 'Model-B';
tgacChannel.ChannelBandwidth = cfgVHT.ChannelBandwidth;
tgacChannel.SampleRate = wlanSampleRate(cfgVHT);
tgacChannel.NumReceiveAntennas = NumRxAnts;
tgacChannel.NumTransmitAntennas = NumTxAnts;
tgacChannel.TransmitReceiveDistance = 100; % Meters
tgacChannel.RandomStream = 'mt19937ar with seed';
tgacChannel.Seed = 70; % Seed to allow repeatability
```

Noise is added to the time domain waveform at the output of the channel with a power, `noisePower`.

```
noisePower = -37; % dBW
```

Setup other objects and variables for simulation.

```
% Indices for extracting fields
ind = wlanFieldIndices(cfgVHT);

% AWGN channel to add noise with a specified noise power. The random
% process controlling noise generation is seeded to allow repeatability.
awgnChannel = comm.AWGNChannel;
awgnChannel.RandomStream = 'mt19937ar with seed';
awgnChannel.Seed = 5;
awgnChannel.NoiseMethod = 'Variance';
awgnChannel.Variance = 10^(noisePower/10);

% Calculate the expected noise variance after OFDM demodulation
noiseVar = vhtBeamformingNoiseVariance(noisePower,cfgVHT);

% Number of spatial streams
Nss = NumSTS/(cfgVHT.STBC+1);

% Get the number of occupied subcarriers in VHT fields
ofdmInfo = wlanVHTOFDMInfo('VHT-Data',cfgVHT);
Nst = ofdmInfo.NumTones;

% Generate a random PSDU which will be transmitted
rng(0); % Set random state for repeatability
psdu = randi([0 1],cfgVHT.PSDULength*8,1);
```

Transmission with Spatial Expansion

First perform a transmission using spatial expansion. This type of transmission may be made by a multi-antenna transmitter to a receiver which is not capable of being a beamformee. The `SpatialMapping` property of the format configuration object allows different spatial mapping schemes to be selected. This example uses the example spatial expansion matrix provided in Section 19.3.11.1.1.2 of [1 on page 3-35]. Therefore, configure a `'Custom'` spatial mapping. Use the custom spatial mapping matrix by assigning the `SpatialMappingMatrix` of the format

configuration object. This matrix describes the mapping of each subcarrier for each space-time stream to all transmit antennas. Therefore the size of the spatial mapping matrix used is `Nst-by-Nsts-by-Nt`. `Nst` is the number of occupied subcarriers, `Nsts` is the number of space-time streams, and `Nt` is the number of transmit antennas. The spatial mapping matrix duplicates some of the space-time streams to form the desired number of transmit streams.

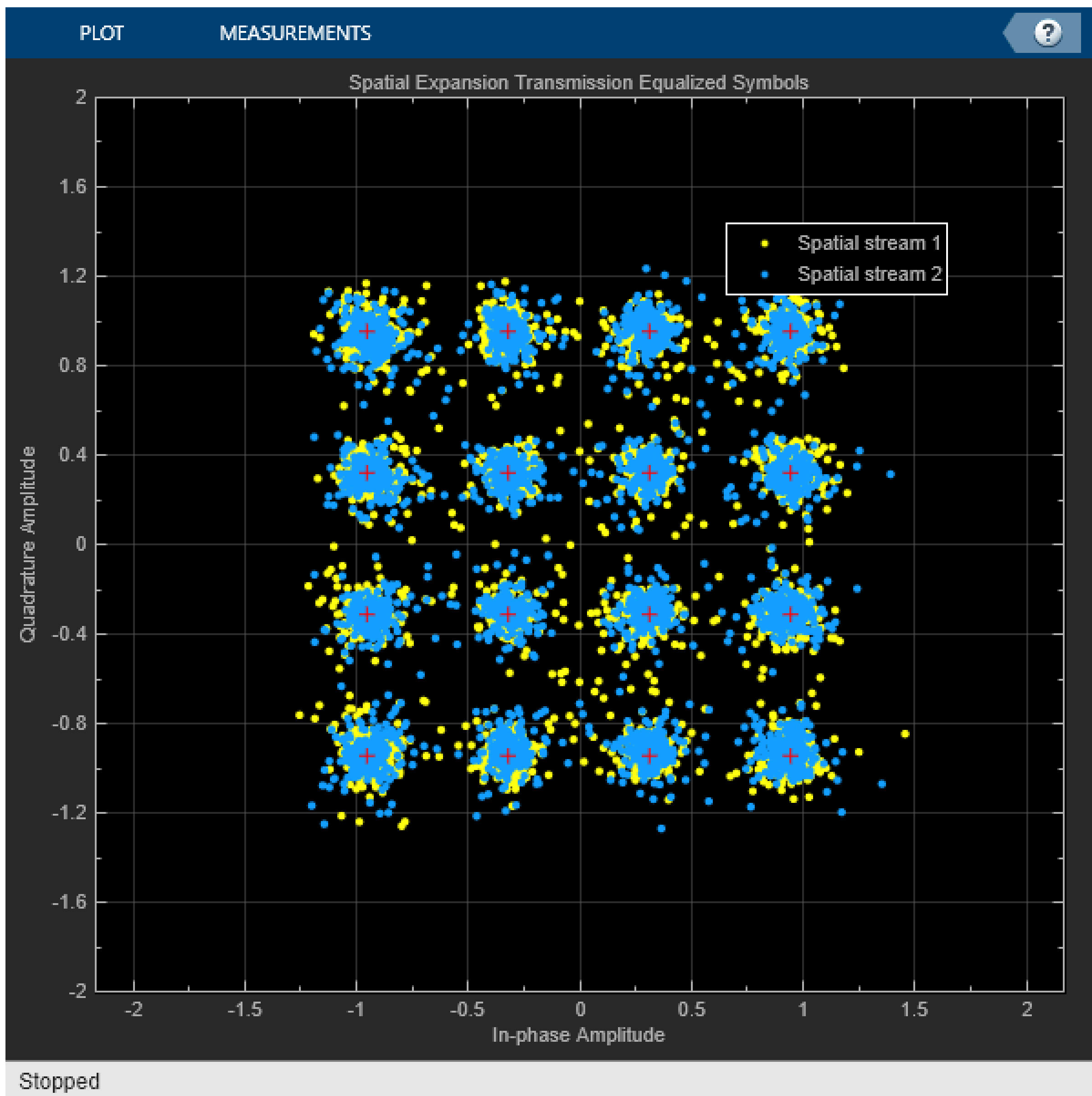
```
% Configure a spatial expansion transmission
vhtSE = cfgVHT;
vhtSE.SpatialMapping = 'Custom'; % Use custom spatial expansion matrix
vhtSE.SpatialMappingMatrix = helperSpatialExpansionMatrix(vhtSE);

% Generate waveform
tx = wlanWaveformGenerator(psdu,vhtSE);

% Pass waveform through a fading channel and add noise. Trailing zeros
% are added to allow for channel filter delay.
rx = tgacChannel([tx; zeros(15,NumTxAnts)]);
% Allow same channel realization to be used subsequently
reset(tgacChannel);
rx = awgnChannel(rx);
% Allow same noise realization to be used subsequently
reset(awgnChannel);

% Estimate symbol timing
tOff = wlanSymbolTimingEstimate(rx(ind.LSTF(1):ind.LSIG(2),:),vhtSE.ChannelBandwidth);

% Channel estimation
vhtltf = rx(tOff+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,vhtSE);
chanEstSE = wlanVHTLTFChannelEstimate(vhtltfDemod,vhtSE);
```

Demodulate and equalize the data field to recover OFDM symbols for each spatial stream.

```
vhtdata = rx(tOff+(ind.VHTData(1):ind.VHTData(2)),:);
[~,~,symSE] = wlanVHTDataRecover(vhtdata,chanEstSE,noiseVar,vhtSE,...
    'PilotPhaseTracking','None');
```

Plot the constellation of each spatial stream.

```
refSym = wlanReferenceSymbols(cfgVHT); % Reference constellation
seConst = vhtBeamformingPlotConstellation(symSE,refSym, ...
    'Spatial Expansion Transmission Equalized Symbols');
```



The variance in the constellation is approximately the same for each spatial stream as the SNRs are approximately the same. This is because the average power in the channel is on average approximately the same per space-time stream:

```
disp('Mean received channel power per space-time stream with spatial expansion: ')
```

```
Mean received channel power per space-time stream with spatial expansion:
```

```
for i = 1:NumSTS
    fprintf('  Space-time stream %d: %2.2f W\n',i, ...
        sum(mean(chanEstSE(:,i,:).*conj(chanEstSE(:,i,:)),1),3))
end
```

```
  Space-time stream 1: 0.73 W
  Space-time stream 2: 0.50 W
```

Transmission with Beamforming

When the receiver is capable of being a beamformee, a beamformed transmission can create a higher SNR compared to spatial expansion. This example demonstrates the advantage of having channel state information available to create and use a steering matrix.

Calculate a beamforming steering matrix, by passing an NDP through the channel. Use `'Direct'` spatial mapping for the NDP transmission with the number of space-time streams is configured to match the number of transmit antennas. This uses the VHT-LTF to sound channels between each of the transmit antennas and receive antennas. Use the calculated beamforming matrix to beamform a transmission through the channel. The same channel realization is used for sounding and data transmission and there is no feedback compression between beamformee and beamformer, therefore the beamforming is regarded as perfect in this example.

```
% Configure a sounding packet
vhtSound = cfgVHT;
vhtSound.APEPLength = 0; % NDP so no data
vhtSound.NumSpaceTimeStreams = NumTxAnts;
vhtSound.SpatialMapping = 'Direct'; % Each TxAnt carries a STS

% Generate sounding waveform
soundingPSDU = [];
tx = wlanWaveformGenerator(soundingPSDU,vhtSound);

% Pass sounding waveform through the channel and add noise. Trailing zeros
% are added to allow for channel filter delay.
rx = tgacChannel([tx; zeros(15,NumTxAnts)]);
% Allow same channel realization to be used subsequently
reset(tgacChannel);
rx = awgnChannel(rx);
% Allow same noise realization to be used subsequently
reset(awgnChannel);

% Estimate symbol timing
tOff = wlanSymbolTimingEstimate(rx(ind.LSTF(1):ind.LSIG(2),:),vhtSound.ChannelBandwidth);
```

Perform channel estimation using the sounding packet to estimate the actual channel response between each transmit and receive antenna.

```
% Channel estimation
vhtLLTFInd = wlanFieldIndices(vhtSound,'VHT-LTF');
vhtltf = rx(tOff+(vhtLLTFInd(1):vhtLLTFInd(2)),:);
vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,vhtSound);
chanEstSound = wlanVHTLTFChannelEstimate(vhtltfDemod,vhtSound);
```

The channel estimated using the `wlanVHTLTFChannelEstimate` function includes cyclic shifts applied at the transmitter to each space-time stream. Remove the cyclic shifts applied at the transmitter from the channel estimate to calculate a beamforming steering matrix.

```
chanEstSound = vhtBeamformingRemoveCSD(chanEstSound, ...
    vhtSound.ChannelBandwidth,vhtSound.NumSpaceTimeStreams);
```

This example calculates the beamforming steering matrix using singular value decomposition (SVD). The SVD of the channel matrix results in two unitary matrices, `U` and `V`, and a diagonal matrix of singular values `S`. The first `NumSTS` columns of `V` per subcarrier are used as the beamforming steering matrix. The SVD is computed using the `svd` function.

```
chanEstPerm = permute(chanEstSound,[3 2 1]); % Permute to Nr-by-Nt-by-Nst
[U,S,V] = pagesvd(chanEstPerm,'econ');
steeringMatrix = permute(V(:,1:NumSTS,:),[3 2 1]); % Permute to Nst-by-Nsts-by-Nt
```

Apply the beamforming steering matrix calculated above as a custom spatial mapping matrix and use it to send data through the same channel.

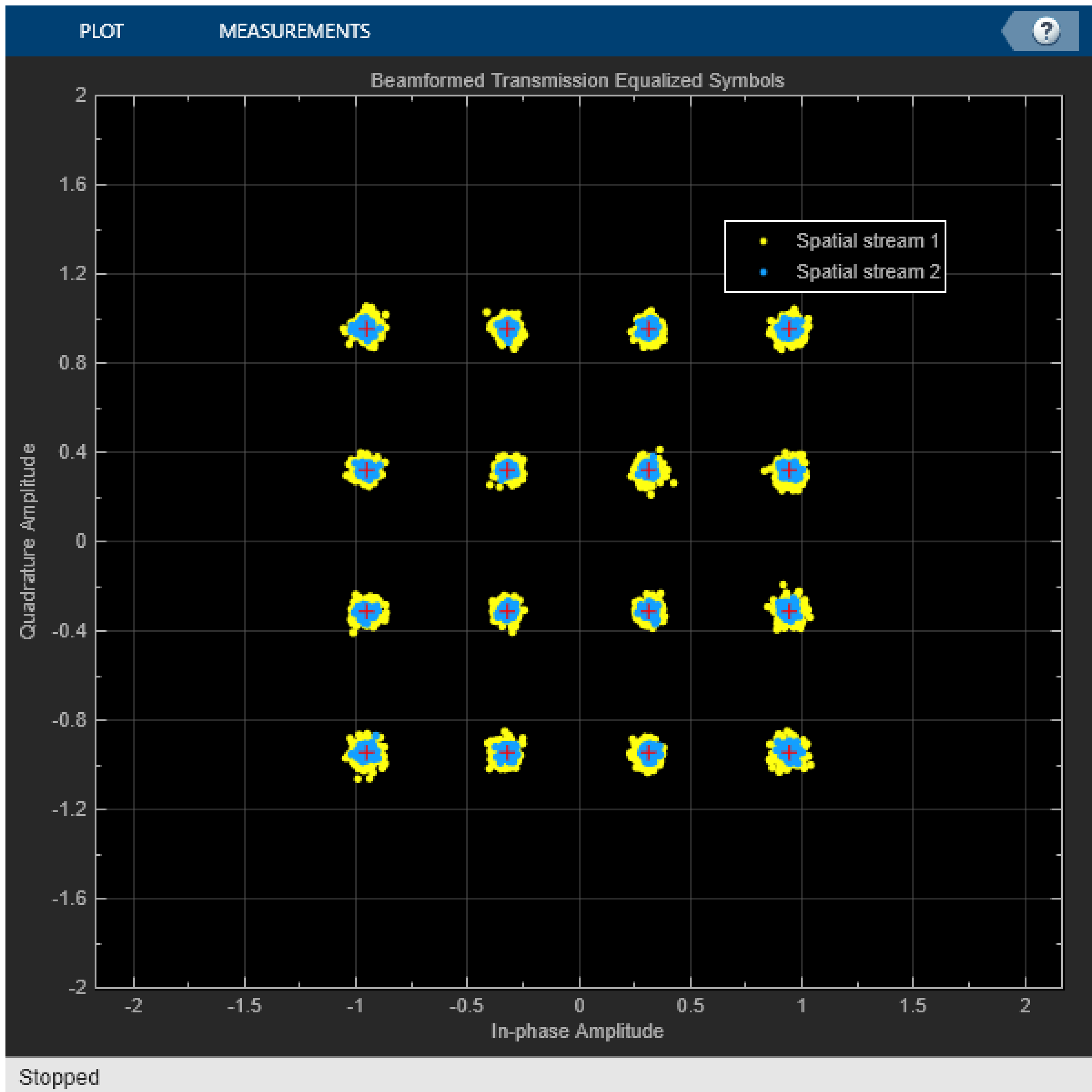
```
% Configure a transmission with beamforming
vhtBF = cfgVHT;
vhtBF.SpatialMapping = 'Custom';
vhtBF.SpatialMappingMatrix = steeringMatrix;

% Generate beamformed data transmission
tx = wlanWaveformGenerator(psdu,vhtBF);

% Pass through the channel and add noise. Trailing zeros
% are added to allow for channel filter delay.
rx = tgacChannel([tx; zeros(15,NumTxAnts)]);
rx = awgnChannel(rx);

% Estimate symbol timing
tOff = wlanSymbolTimingEstimate(rx(ind.LSTF(1):ind.LSIG(2),:),vhtBF.ChannelBandwidth);

% Channel estimation
vhtltf = rx(tOff+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,vhtBF);
chanEstBF = wlanVHTLTFChannelEstimate(vhtltfDemod,vhtBF);
```

Demodulate and equalize the received data field to recover OFDM symbols for each spatial stream.

```
vhtdata = rx(tOff+(ind.VHTData(1):ind.VHTData(2)),:);
[~,~,symBF] = wlanVHTDataRecover(vhtdata,chanEstBF,noiseVar,vhtBF,...
    'PilotPhaseTracking','None','LDPCDecodingMethod','norm-min-sum');
```

Plot the equalized constellation for each spatial stream. The higher-order spatial stream has a larger variance. This is due to the ordered singular values of the channels used in SVD beamforming.

```
bfConst = vhtBeamformingPlotConstellation(symBF,refSym, ...
    'Beamformed Transmission Equalized Symbols');
```



This ordering is also visible in the average power of the received space-time streams. The power of the received first space-time stream is larger than the second space-time stream. This is because the received signal strength is a function of the singular values of the channel which SVD orders in a decreasing fashion.

```
disp('Mean received channel power per space-time stream with SVD transmit beamforming: ')
```

```
Mean received channel power per space-time stream with SVD transmit beamforming:
```

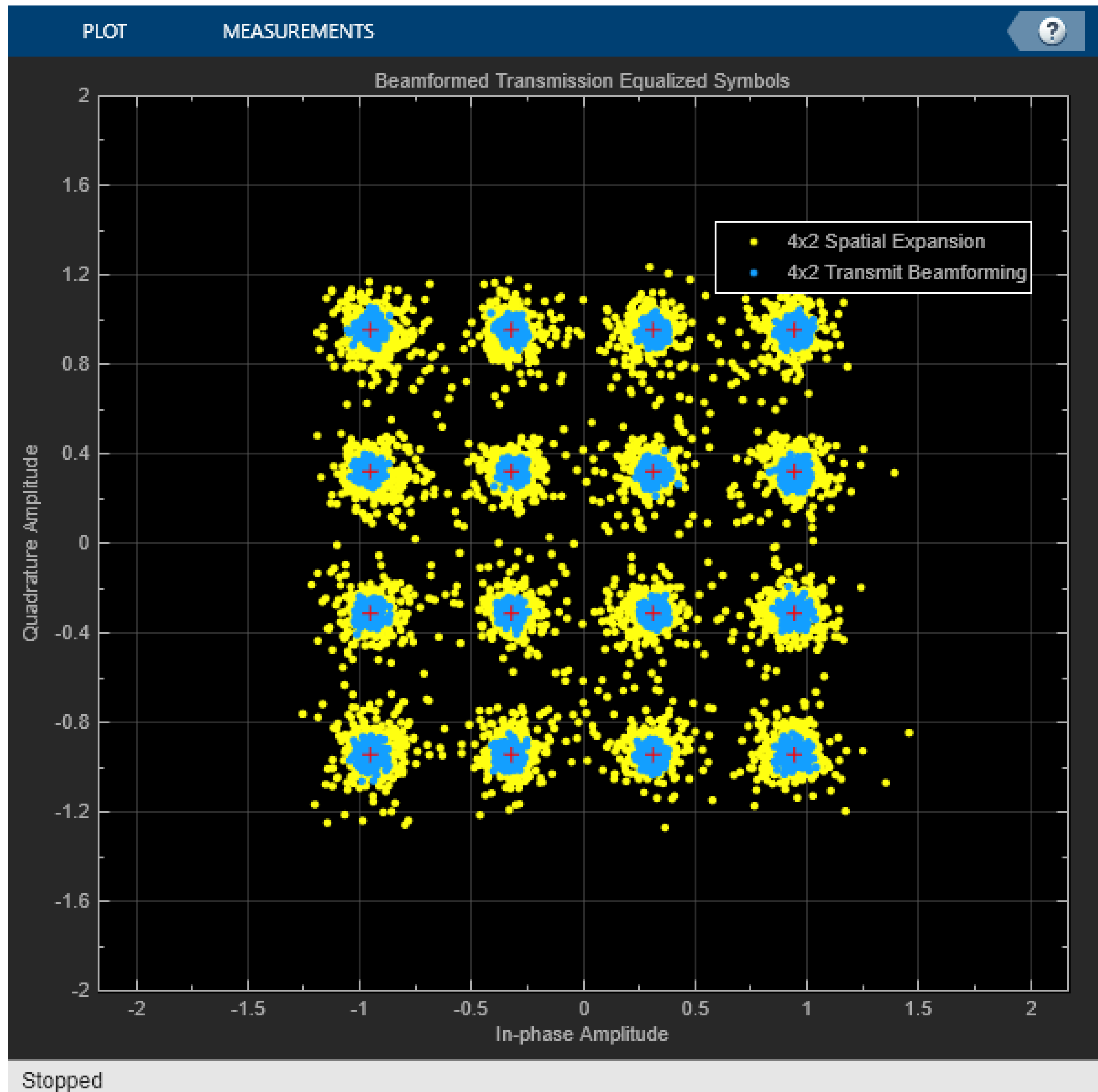
```
for i = 1:NumSTS
    fprintf('  Space-time stream %d: %2.2f W\n',i, ...
        sum(mean(chanEstBF(:,i,:).*conj(chanEstBF(:,i,:)),1),3))
end
```

```
  Space-time stream 1: 2.08 W
  Space-time stream 2: 0.45 W
```

Comparison and Conclusion

Plot the equalized constellation from the spatial expansion and beamformed transmissions for all spatial streams. Note the improved constellation using SVD-based transmit beamforming.

```
str = sprintf('%dx%d',NumTxAnts,NumRxAnts);
compConst = vhtBeamformingPlotConstellation([symSE(:) symBF(:)],refSym, ...
    'Beamformed Transmission Equalized Symbols', ...
    {[str ' Spatial Expansion'],[str ' Transmit Beamforming']});
```



Measure the improvement using the RMS and maximum error vector magnitude (EVM). EVM is a measure of demodulated signal quality.

```
EVM = comm.EVM;
EVM.AveragingDimensions = [1 2]; % Average over all subcarriers and symbols
```

```
EVM.MaximumEVMOutputPort = true;
EVM.ReferenceSignalSource  = 'Estimated from reference constellation';
EVM.ReferenceConstellation = refSym;

[rmsEVMSE,maxEVMSE] = EVM(symSE); % EVM using spatial expansion
[rmsEVMBF,maxEVMBF] = EVM(symBF); % EVM using beamforming

for i = 1:Nss
    fprintf(['Spatial stream %d EVM:\n' ...
        '  Spatial expansion:    %2.1f%% RMS, %2.1f%% max\n' ...
        '  Transmit beamforming: %2.1f%% RMS, %2.1f%% max\n'], ...
        i,rmsEVMSE(i),maxEVMSE(i),rmsEVMBF(i),maxEVMBF(i));
end
```

```
Spatial stream 1 EVM:
  Spatial expansion:    9.2% RMS, 44.8% max
  Transmit beamforming: 2.0% RMS, 8.6% max
Spatial stream 2 EVM:
  Spatial expansion:    9.2% RMS, 52.3% max
  Transmit beamforming: 4.1% RMS, 12.7% max
```

This example demonstrates that if a receiver is capable of being a beamformee, the SNR can potentially be improved when a transmission is beamformed compared to a spatial expansion transmission. The increase in received power when using beamforming can lead to more reliable demodulation or potentially even a higher order modulation and coding scheme to be used for the transmission.

In a realistic operational simulation the performance of beamforming would be degraded due to the delay between channel state information calculation and feedback by the beamformee and feedback quantization. For more information, see [2 on page 3-35].

References

1. IEEE Std 802.11™-2020 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.
2. Perahia, Eldad, and Robert Stacey. Next Generation Wireless LANS: 802.11n and 802.11ac. Cambridge University Press, 2013.

802.11ah Waveform Generation

This example shows how to generate IEEE® 802.11ah™ S1G waveforms and highlights some of the key features of the standard.

Introduction

802.11ah is intended for extended range and low power applications in the unlicensed sub 1 GHz band, including machine to machine communication and the internet of things. 802.11ah uses narrower contiguous channel bandwidths than 802.11n™ and 802.11ac™ to facilitate long range, low power communication at a lower data rate. Valid channel bandwidths are 1, 2, 4, 8, and 16 MHz.

Since 802.11ah uses the same underlying physical layer technologies as 802.11n and 802.11ac, the processing chains are very similar. With the exception of 1 MHz transmissions, in general data is modulated using the same process as in 802.11ac with a 1/10 clock rate.

In this example a number of 802.11ah S1G [1] waveforms are generated to highlight some of the key modes and features of the 802.11ah standard.

802.11ah Modes and PHY Features

The 802.11ah standard defines three modes:

- The 1 MHz mode (S1G_1M) is intended for low data rate applications. This mode features an extended preamble and a new modulation and coding scheme, MCS10, to improve robustness. MCS10 is BPSK 1/2 rate with 2 times repetition. When MCS10 is used the short training field (STF) is boosted by 3 dB to allow for packet detection [2]. In this mode the whole PPDU is beamformed.
- The >= 2 MHz long preamble mode (S1G_LONG) is used for single or multi-user transmissions with a 2, 4, 8, or 16 MHz channel bandwidth. The PPDU is similar to a 802.11ac VHT PPDU, consisting of an omni-directional portion and beam-changeable portion.
- The >= 2 MHz short preamble mode (S1G_SHORT) is used for single-user transmissions with a 2, 4, 8, or 16 MHz channel bandwidth. In this mode the whole PPDU is beamformed.

The 802.11ah standard is intended to operate outdoors as well as indoors. Traveling Pilots have been introduced to compensate for Doppler spread caused by reflections due to vehicular motion. In previous 802.11 standards the pilot locations are fixed to the same subcarrier for the duration of a packet. Tracking the varying channel conditions due to a high Doppler environment is not effective with fixed pilot locations. Traveling pilots change the subcarriers that carry the pilots over time which improves the ability to track changing channel conditions. In this example a waveform is generated for each of the three modes introduced above with configurations for MCS10 and traveling pilot highlighted.

S1G 1MHz Mode

An S1G 1 MHz PPDU consists of five fields, all of which can be beamformed:

1. STF - The short training field, which is used for coarse synchronization
2. LTF1 - The first long training field, which is used for fine synchronization and initial channel estimation
3. SIG - The signaling field, which the receiver decodes to determine transmission parameters

4. LTF2-N - The subsequent long training fields, which is used for MIMO channel estimation
5. Data - The data field, which carries the user data payload

Examples of waveform generation for MCS0 and MCS10 1 MHz transmissions are shown. When MCS10 is used a 3 dB power boost is applied to the short training field. This power boost will be visualized.

The function `wlanWaveformGenerator` returns an S1G configuration object. Create an S1G configuration object for 1 MHz bandwidth, 1 transmit antenna, 1 space-time stream, BPSK rate 1/2 (MCS0), and a 256 byte APEP length.

```
cfg1MHz = wlanS1GConfig;
cfg1MHz.ChannelBandwidth = 'CBW1';
cfg1MHz.NumTransmitAntennas = 1;
cfg1MHz.NumSpaceTimeStreams = 1;
cfg1MHz.MCS = 0;
cfg1MHz.APEPLength = 256;
```

Create a PSDU of random bits using the required length for the specified format configuration.

```
psdu = randi([0 1],cfg1MHz.PSDULength*8,1);
```

Generate an S1G waveform using the configured S1G format object and PSDU as inputs to the waveform generator, `wlanWaveformGenerator`. The waveform generator modulates PSDU bits according to a format configuration. The waveform generator also performs OFDM windowing. In this example windowing is disabled for clearer visualization.

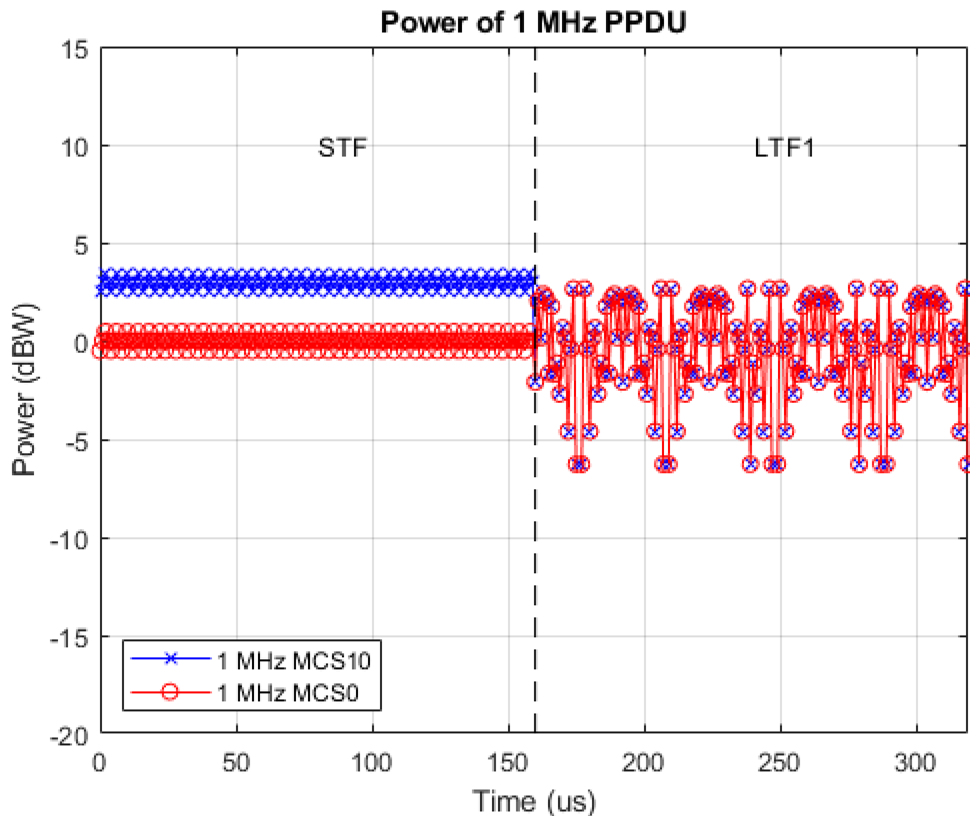
```
% Generate waveform with windowing disabled
txMCS0 = wlanWaveformGenerator(psdu,cfg1MHz,'WindowTransitionTime',0);
```

Change the MCS of `cfg1MHz` to 10 and generate a second waveform to demonstrate the STF power boost.

```
cfg1MHz.MCS = 10;
txMCS10 = wlanWaveformGenerator(psdu,cfg1MHz,'WindowTransitionTime',0);
```

The power is plotted for the first 320 microseconds of both waveforms to capture the duration of the STF and first LTF in the 1 MHz transmission. Note the power boost of the STF when MCS10 is used. The power boost is required to obtain sufficient packet detection sensitivity to support MCS10 [2].

```
t = 320; % Duration to plot in microseconds
sr = wlanSampleRate(cfg1MHz); % Sample rate Hz
tick = (1/sr)*1e6; % Microseconds per sample
hf = figure;
hp(1) = plot(0:tick:t-tick,20*log10(abs(txMCS10(1:t*sr*1e-6,:))),'bx-');
hold on;
hp(2) = plot(0:tick:t-tick,20*log10(abs(txMCS0(1:t*sr*1e-6,:))),'ro-');
xlim([0 t-1]);
ylim([-20 15]);
s1gWavGenPlotFieldOverlay(cfg1MHz,hf);
grid on;
legend(hp,'1 MHz MCS10','1 MHz MCS0','Location','SouthWest');
title('Power of 1 MHz PPDU');
xlabel('Time (us)');
ylabel('Power (dBW)');
```



S1G >=2 MHz Long Preamble Mode

The 802.11ah long preamble supports single and multi-user transmissions. The long preamble PPDU consists of two portions; the omni-directional portion and the beam-changeable portion.

The omni-directional portion is transmitted to all users without beamforming. It consists of three fields:

1. STF - The short training field, which is used for coarse synchronization
2. LTF1 - The first long training field, which is used for fine synchronization and initial channel estimation
3. SIG-A - The signaling A field, which the receiver decodes to determine transmission parameters relevant to all users

The beam-changeable portion can be beamformed to each user. It consists of four fields:

1. D-STF - The beamformed short training field, which is used by the receiver for automatic gain control
2. D-LTF - The beamformed long training fields, which is used for MIMO channel estimation
3. SIG-B - The signaling B field. In a multi-user transmission the SIG-B signals the MCS for each user. In a single-user transmission the MCS is signaled in the SIG-A field of the omni-directional portion of the preamble. Therefore in a single-user transmission the SIG-B symbol transmitted is an exact repetition of the first D-LTF. This repetition allows for improved channel estimation.
4. Data - The data field, which carries the user data payload

To visualize repetition of the first D-LTF an S1G 2 MHz long preamble format configuration object is created using the `wlanS1GConfig` function and configured for one space-time stream and one transmit antenna.

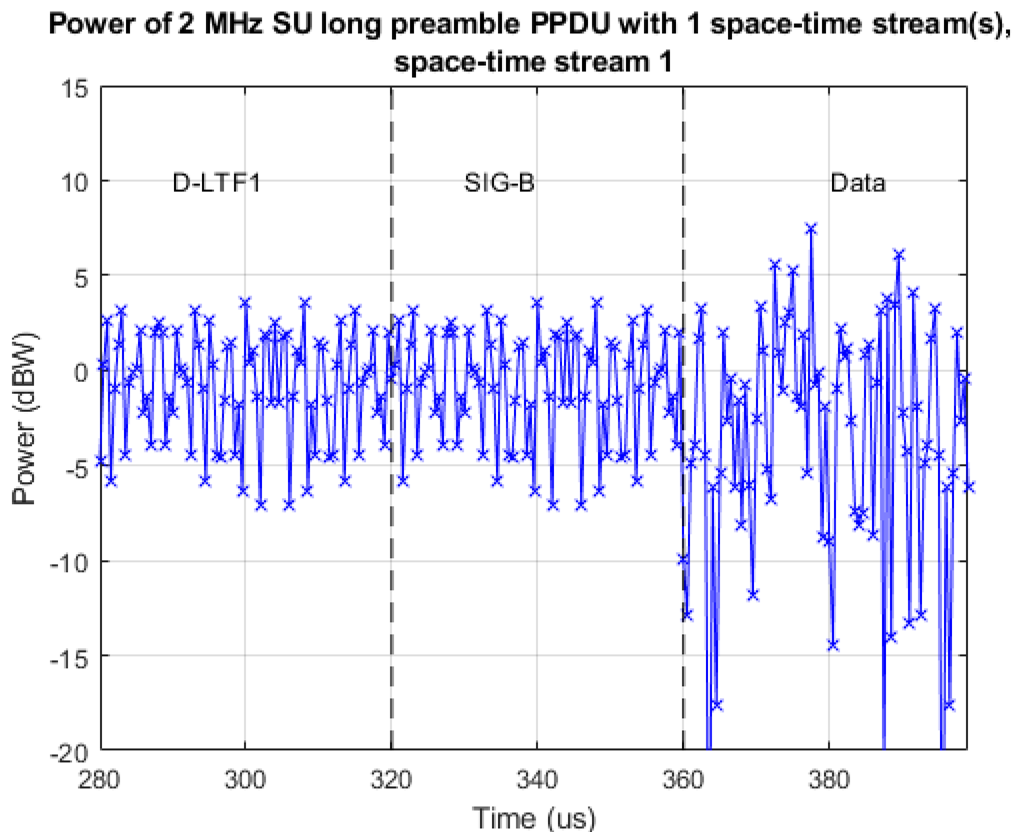
```
cfgSU = wlanS1GConfig;
cfgSU.ChannelBandwidth = 'CBW2';
cfgSU.Preamble = 'Long';
cfgSU.NumUsers = 1;
cfgSU.NumSpaceTimeStreams = 1;
cfgSU.NumTransmitAntennas = 1;
cfgSU.MCS = 1;
cfgSU.APEPLength = 150;
```

The >=2 MHz long preamble waveform with a single space-time stream is generated using the `cfgSU` object.

```
% Generate a PSDU containing random bits
psdu = randi([0 1],cfgSU.PSDULength*8,1);

% Generate a PPDU waveform
txSU = wlanWaveformGenerator(psdu,cfgSU);
```

The D-LTF and SIG-B fields are plotted. Note the repetition of the D-LTF in the SIG-B symbol.

```
s1gWavGenPlotSIGB(cfgSU,txSU);
```



As a comparison a 2 MHz long preamble multi-user waveform will be generated and visualized. First, a format configuration object is created for two users. The user positions, number of space times

streams, MCS and APEP length are configured per user using vectors to parameterize the relevant properties of the `cfgMU` object.

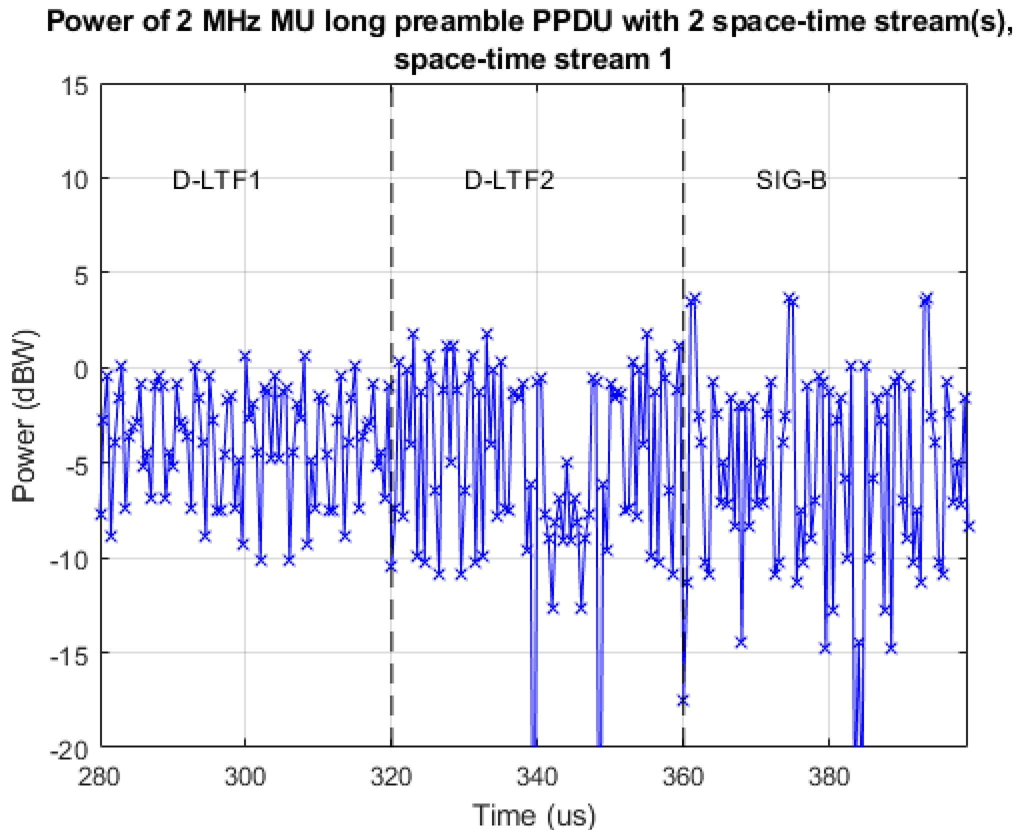
```
cfgMU = wlanS1GConfig;
cfgMU.ChannelBandwidth = 'CBW2';
cfgMU.Preamble = 'Long';
cfgMU.NumUsers = 2;
cfgMU.UserPositions = [0 1];
cfgMU.NumSpaceTimeStreams = [1 1];
cfgMU.NumTransmitAntennas = sum(cfgMU.NumSpaceTimeStreams);
cfgMU.MCS = [1 2];
cfgMU.APEPLength = [150 250];
```

A random PSDU is created for each user and a multi-user waveform is generated. The PSDU length for each user, `cfgMU.PSDULength`, is calculated based on transmission properties by the `cfgMU` object.

```
% Generate cell array containing the PSDUs for all users
psdu = cell(cfgMU.NumUsers,1);
for i = 1:cfgMU.NumUsers
    psdu{i} = randi([0 1],cfgMU.PSDULength(i),1);
end

% Generate waveform
txMU = wlanWaveformGenerator(psdu,cfgMU);
```

The two D-LTF fields and SIG-B fields are plotted for the first space-time stream. Note the SIG-B symbol is no longer a repetition of D-LTF1 as it carries the MCS per user.

```
s1gWavGenPlotSIGB(cfgMU,txMU);
```



S1G >=2 MHz Short Preamble Mode

An S1G >=2 MHz short preamble waveform consists of five fields, all of which can be beamformed:

1. STF - The short training field, which is used for coarse synchronization
2. LTF1 - The first long training field, which is used for fine synchronization and initial channel estimation
3. SIG - The signaling field, which the receiver decodes to determine transmission parameters
4. LTF2-N - The subsequent long training fields, which is used for MIMO channel estimation
5. Data - The data field, which carries the user data payload

In this example S1G 2 MHz short preamble waveforms with and without traveling pilots will be generated.

Traveling pilots are an optional feature for all three S1G modes to allow for outdoor links where Doppler spread is potentially introduced due to moving vehicles. The traveling pilots are boosted 1.5 times compared to fixed pilots to improve channel estimation performance in this environment [3].

Two >= 2 MHz short preamble waveforms are generated; one with fixed pilots and one with traveling pilots. First a S1G 2 MHz short preamble format configuration with fixed pilot locations is created using the `wlanS1GConfig` object.

```
cfgFix = wlanS1GConfig;
cfgFix.ChannelBandwidth = 'CBW2';
cfgFix.Preamble = 'Short';
```

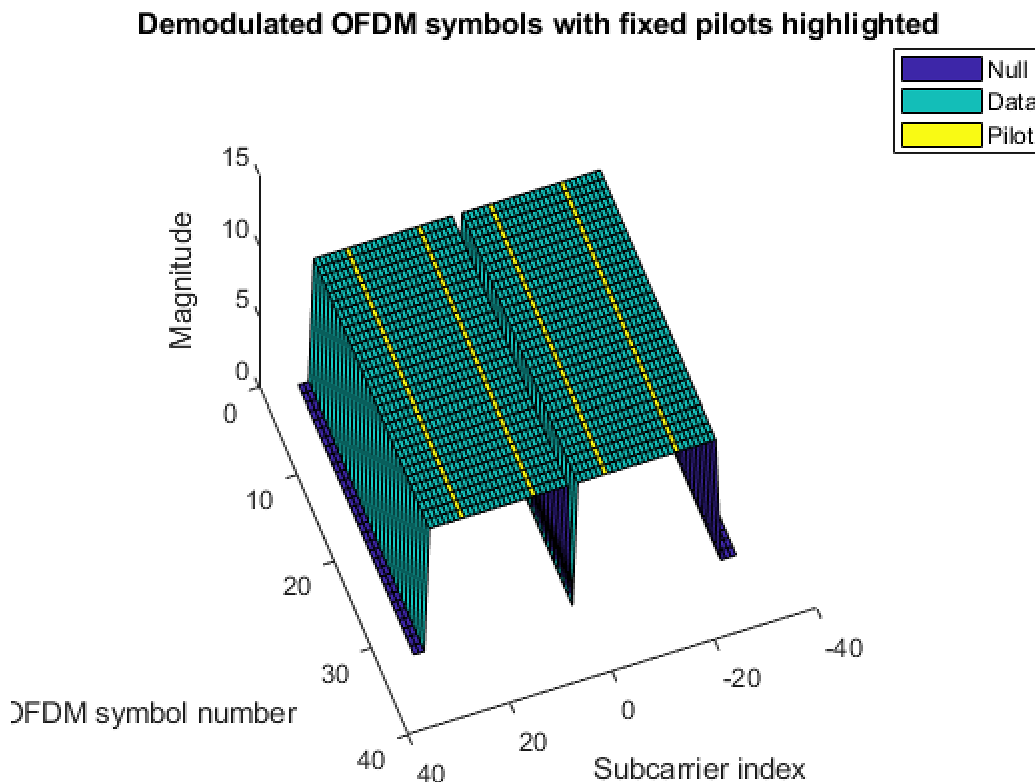
```
cfgFix.NumTransmitAntennas = 1;
cfgFix.NumSpaceTimeStreams = 1;
cfgFix.MCS = 0; % BPSK so same power on all subcarriers for analysis
cfgFix.APEPLength = 100;
cfgFix.TravelingPilots = false; % Fixed pilot subcarriers
```

Generate a fixed pilot waveform using the `cfgFix` object and random PSDU bits. The PSDU bits are created using the required length for the specified format configuration.

```
% Generate a PSDU containing random bits
psdu = randi([0 1],cfgFix.PSDULength*8,1);

% Generate a PPDU waveform
txFix = wlanWaveformGenerator(psdu,cfgFix);
```

Extract the data field from the time domain waveform using the known duration of the preamble. Plot the magnitude of the OFDM symbols and subcarriers. The location of nulls, data carrying subcarriers, and pilot carrying subcarriers are highlighted. The pilot locations remain unchanged for the duration of the packet.

```
s1gWavGenPlotGrid(txFix,cfgFix, ...
    'Demodulated OFDM symbols with fixed pilots highlighted')
```

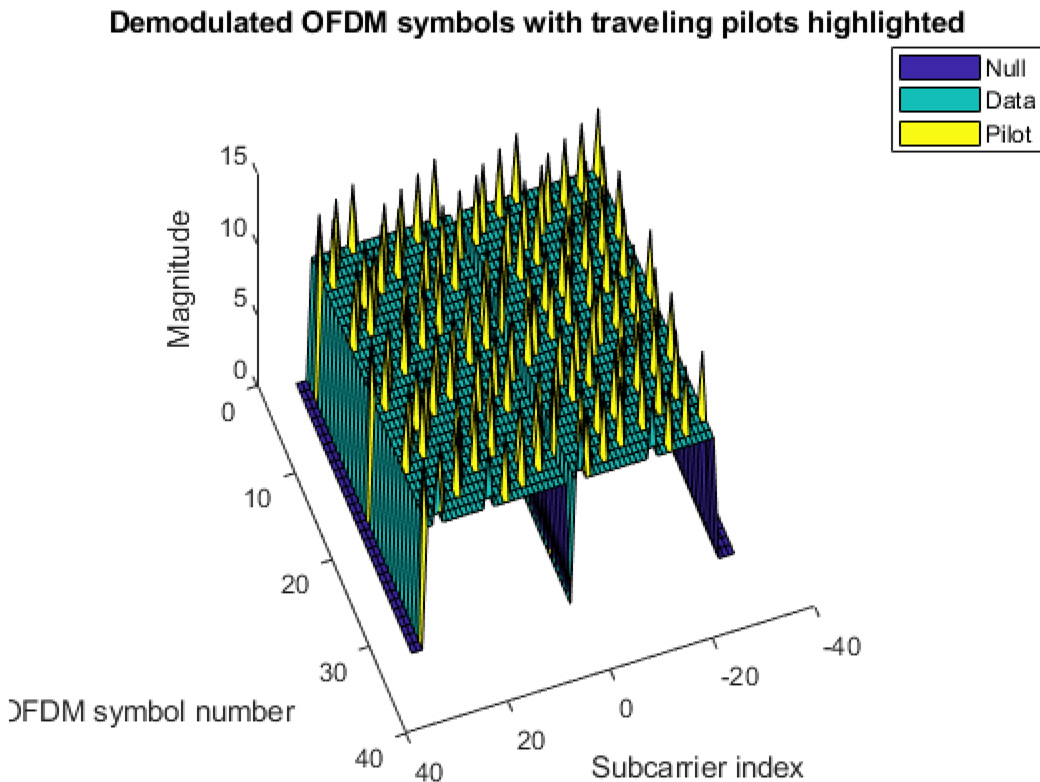


Now a waveform is generated using the same configuration but with traveling pilots. This could be accomplished by changing the `TravelingPilots` property of the existing configuration object and regenerating the waveform, but in this example a separate object is created and used.

```
% Copy the format configuration object and enable traveling pilots
cfgTravel = cfgFix;
cfgTravel.TravelingPilots = true;

% Generate waveform with traveling pilots
txTravel = wlanWaveformGenerator(psdu,cfgTravel);
```

The magnitude of the OFDM symbols and subcarriers is plotted again. The pilot locations now change per OFDM symbol. The magnitude of pilot subcarriers is 1.5 times that of data carrying subcarriers.

```
s1gWavGenPlotGrid(txTravel,cfgTravel, ...
    'Demodulated OFDM symbols with traveling pilots highlighted')
```



Conclusion

This example has demonstrated how to generate waveforms for different 802.11ah S1G modes and highlighted some of the key features of the standard.

Appendix

This example uses the following helper functions:

- s1gWavGenPlotFieldOverlay.m
- s1gWavGenPlotSIGB.m
- s1gWavGenPlotGrid.m

Selected Bibliography

1 IEEE P802.11ah™/D5.0 Draft 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 2: Sub 1 GHz License Exempt Operation.

2 Sameer Vermani et al. "Preamble Format for 1 MHz", IEEE 802.11-11/1482r4, 2012-01-16.

3 Ron Porat et al. "Traveling Pilots", IEEE 8902.11-12/1322r0, 2012-11-12.

802.11n Link in Simulink

This example shows how to simulate an IEEE® 802.11n™ HT link in Simulink® with WLAN Toolbox™.

Introduction

An 802.11n HT [1] link with a fading channel is simulated in this model. Multiple packets are transmitted through a 2-by-2 TGn MIMO channel, demodulated and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the packet error rate. Packet detection, timing synchronization, carrier frequency offset correction and pilot phase tracking are performed by the receiver.

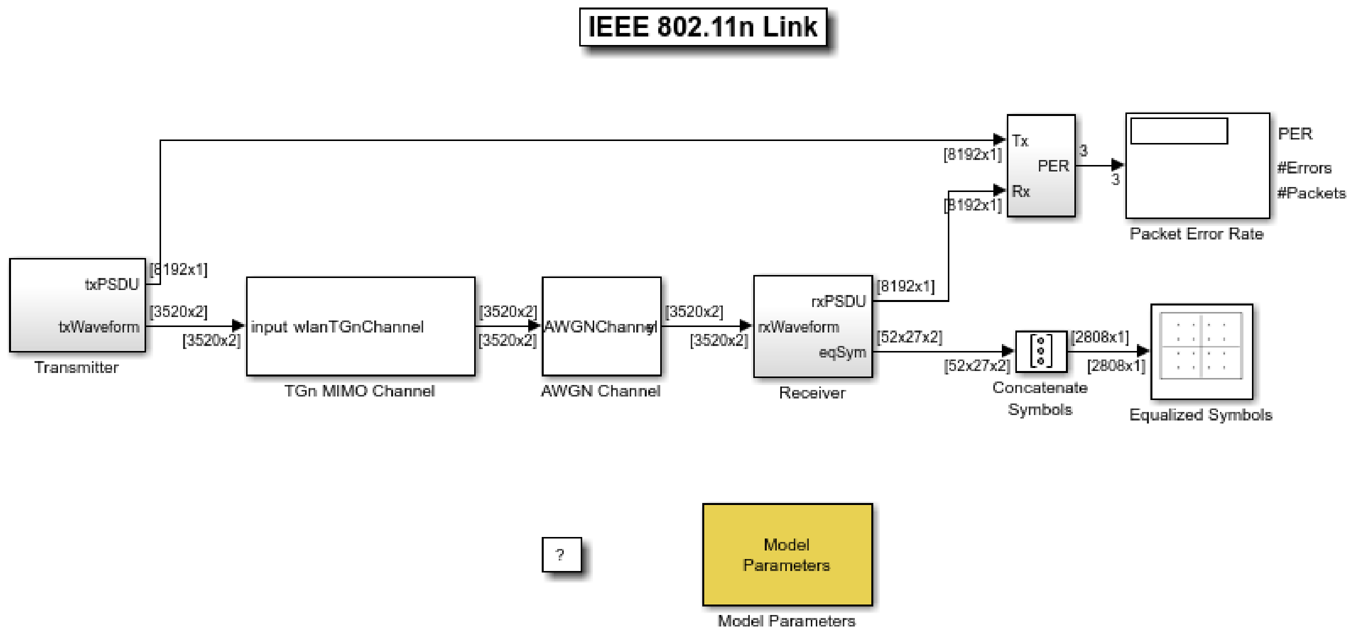
The MATLAB Function block allows MATLAB® functions to be used in a Simulink model. In this example an 802.11n link is modeled in Simulink by using MATLAB Function blocks to call WLAN Toolbox functions. For an equivalent 802.11n simulation in MATLAB see the example “802.11n Packet Error Rate Simulation for 2x2 TGn Channel” on page 5-16.

Structure of the Example

The model has four main parts:

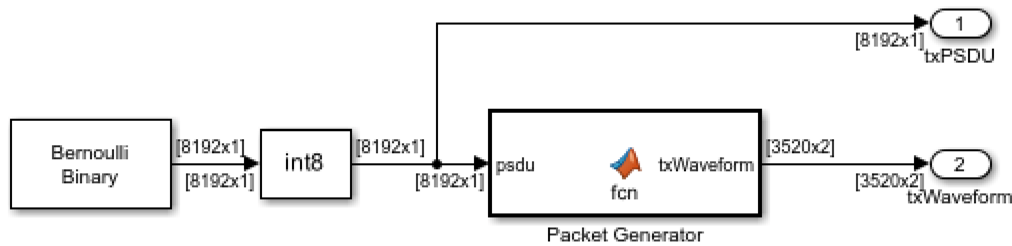
- Transmitter: Generates a random PSDU and creates an 802.11n HT packet.
- Channel: Models a TGn 2x2 MIMO channel with AWGN.
- Receiver: Recovers the transmitted PSDU by performing packet detection, time and frequency synchronization, MIMO channel estimation and PSDU demodulation and decoding.
- Analysis: Compares the transmitted and recovered PSDUs to determine the packet error rate, and displays the equalized symbols.

The following sections describe the transmitter and receiver in more detail.



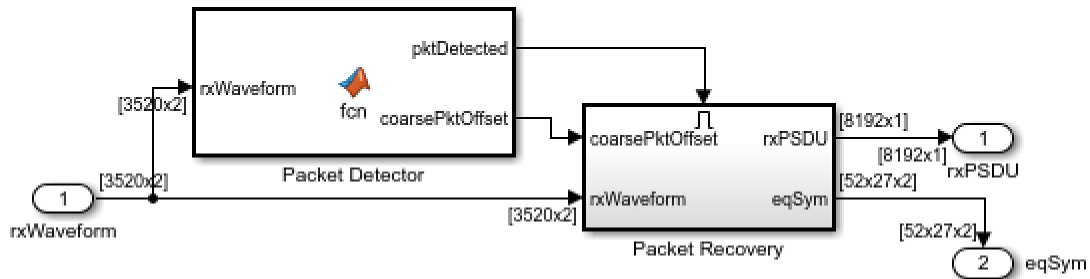
Transmitter

The Transmitter block creates a random PSDU and encodes the bits to create a single packet waveform. The `wlanWaveformGenerator` function is called within the Packet Generator block to generate a waveform for a packet. An idle period is added after each packet to create periodic bursts.



Receiver

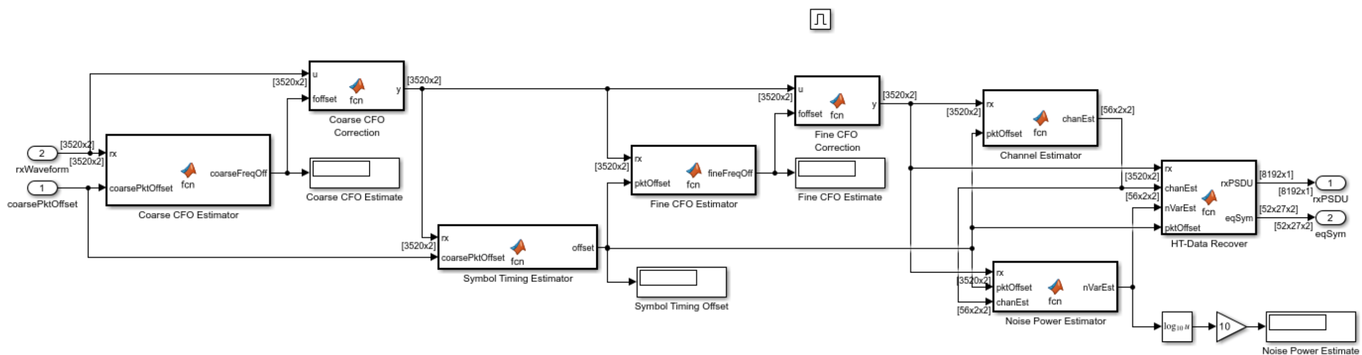
The receiver has two components: packet detection and packet recovery.



The `wlanPacketDetect` function is called within the Packet Detector block. If a packet is detected the Packet Recovery subsystem is enabled to process the detected packet.

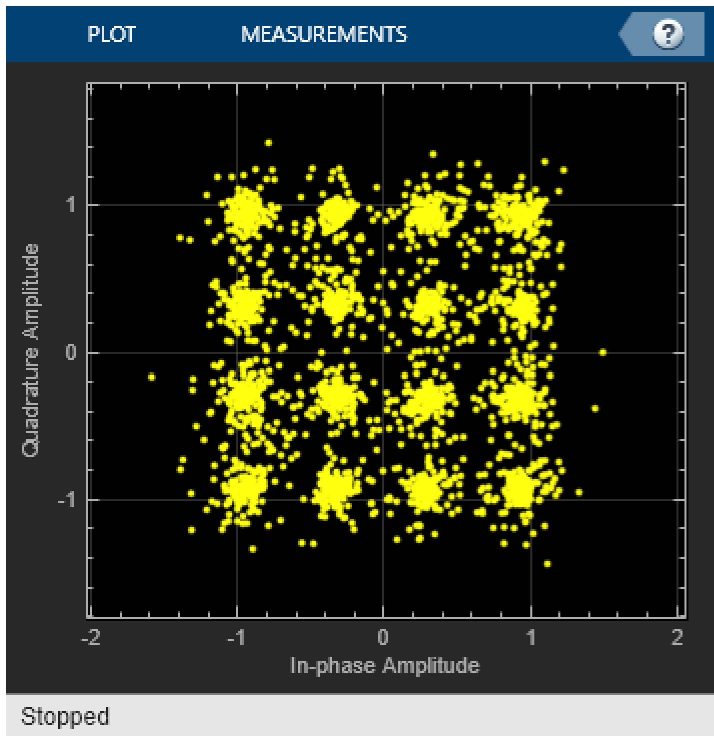
The Packet Recovery subsystem processing consists of the following steps:

1. Coarse carrier frequency offset is estimated and corrected.
2. Fine timing synchronization is established.
3. Fine carrier frequency offset is estimated and corrected.
4. The HT-LTF is extracted from the synchronized received waveform. The HT-LTF is OFDM demodulated and channel estimation is performed.
5. The HT Data field is extracted from the synchronized received waveform.
6. Noise estimation is performed using the demodulated data field pilots.
7. The PSDU is recovered using the extracted field, the channel and noise power estimates.



Results and Displays

When the simulation is run, the packet error rate is displayed. This is updated after each packet is processed. The equalized data symbols are also displayed for each packet processed. By default, 200 packets are simulated.



Exploring the Example

Try changing the signal to noise ratio (SNR) in the AWGN channel block. Decreasing the SNR increases the packet error rate and the noise visible in the equalized symbol constellation. Link parameters such as the modulation and coding scheme (MCS), number of transmit and receive antennas and space-time streams can be changed in the Model Parameters block.

Selected Bibliography

1 IEEE Std 802.11™-2020. 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.

Signal Reception

Recover and Analyze Packets in 802.11 Waveform

This example blindly detects, decodes, and analyzes multiple IEEE 802.11a™, IEEE 802.11n™, IEEE 802.11ac™, and IEEE 802.11ax™ packets in a waveform. The example provides a summary of the detected packets and displays the MAC contents, error vector magnitude (EVM), power, and signaling information for a selected packet.

Introduction

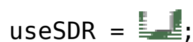
In this example, you detect, decode, and analyze multiple packets within a waveform. This example can decode OFDM non-HT, non-HT duplicate, HT, VHT [1 on page 4-11], HE MU, HE SU, and HE ER SU [2 on page 4-11] packet formats. The receiver does not know any transmission parameters, except for the channel bandwidth. It retrieves these parameters by decoding the preamble fields of the packet.

- The spectrum and time domain samples.
- The signaling field contents.
- The resource unit and user information for an HE waveform.
- The constellation of the equalized data symbols.
- The EVM of the signaling fields.
- The EVM per data subcarrier averaged over spatial streams and symbols.
- The EVM per data symbol averaged over spatial streams and subcarriers.
- The spectral flatness for non-OFDMA packets.
- The MAC frame contents: A-MPDU deaggregation status, Address1, Address2, Frame Check Sequence (FCS) and Frame Type.

Setup Waveform Recovery Parameters

This example analyzes I/Q data containing non-HT, HT-MF, VHT, and HE packets. The `useSDR` variable controls the data source for this example:

- When you select `useSDR`, an SDR captures an off-the-air waveform.
- When you clear `useSDR`, `comm.BasebandFileReader` reads a synthetic waveform stored in a binary file format.

`useSDR =` `;`

Reception with an SDR Device

This example uses SDRs supported by Wireless Testbench™ and SDRs supported by Communication Toolbox™ support packages. This list provides information on which radios can be used by this example along with the required products.

- USRP™ N310/N320/N321/X310/X410 (requires Wireless Testbench™ and Wireless Testbnech Support Package for NI™ USRP Radios)
- ADALM-PLUTO (requires Communications Toolbox Support Package for Analog Devices® ADALM-Pluto Radio)
- USRP™ E310/E312 (requires Communications Toolbox Support Package for USRP™ Embedded Series Radio)

- AD9361/FMCOMMS2/3/4/5 (requires Communications Toolbox Support Package for Xilinx® Zynq®-Based Radio)
- USRP™ N200/N210/USRP2/B200/B210/X300 (requires Communications Toolbox Support Package for USRP™ Radio)

See the relevant documentation for the chosen product and SDR on set up and installation.

Select SDR Device and Capture Antenna(s)

When you select an SDR as the data source, specify the device name from the dropdown along with the capture antenna configuration.

If using an NI USRP hardware with Wireless Testbench, click **Update** to see your saved radio setup configuration name appear at the top of the dropdown list.

```
if useSDR %#ok<*UNRCH>
    deviceOptions    = getDeviceOptions;
    deviceName       = Pluto  Update ;
    antennaOptions   = getAntennaOptions(deviceName);
    antennaSelection = 1 ;
end
```

Specify Capture Parameters and Initiate Capture

After configuring `deviceName` and `antennaSelection`, specify the frequency band, channel number, capture duration, expected channel bandwidth of packets, radio sample rate, and radio gain.

To determine which channels in the 5 GHz band contain traffic from commercial 802.11 hardware, use the "OFDM Beacon Receiver Using Software-Defined Radio" on page 10-33 example for Communication Toolbox Support Package radios or the "OFDM Wi-Fi Scanner Using SDR Preamble Detection" on page 10-20 example for Wireless Testbench radios.

```
if useSDR
    frequencyBand = 5 GHz ; % Frequency band
    channelNumber = 102 ; % WLAN channel number
    captureTime   = milliseconds(100) ; % Signal capture duration, to be specified as a durat
    chanBW        = CBW20 ; % Channel bandwidth of all packets within the wavefor
    sr            = 20000000 ; % Radio sample rate
    gain          = 50 ; % Radio gain

    % Set the center frequency to the corresponding channel number. The
    % center frequency should be centered on the channel bandwidth.
    fc = wlanChannelFrequency(channelNumber,frequencyBand);

    % Create SDR variable (if one does not exist already) and initiate waveform capture
    if ~exist('rx','var')
        rx = [];
    end
```

```
    rx = getSDRObject(deviceName,antennaSelection,fc,sr,gain,rx);
    rxWaveform = capture(rx,captureTime);
end
```

Import a Captured Waveform from a File

If using a precaptured waveform as the data source, this section shows how to load I/Q data from an existing binary file using `comm.BasebandFileReader`. The baseband file format includes the sample rate and the number of channels in the captured waveform. Alternatively, you can load the waveform from a MAT file.

```
% Configure the analysis source
if ~useSDR
    BBR = comm.BasebandFileReader('wlanWaveform.bb'); % Create a baseband file reader object
    chanBW = 'CBW20'; % Channel bandwidth of all packets within the waveform
    bbrInfo = info(BBR);
    BBR.SamplesPerFrame = bbrInfo.NumSamplesInData; % Number of samples in the waveform
    rxWaveform = BBR();  % Load the I&Q sample from a binary file
    sr = BBR.SampleRate; % Sampling rate of the input signal
    release(BBR);
end
```

To recover beacon packets from a MAT-file, see the “OFDM Beacon Receiver Using Software-Defined Radio” on page 10-33 example.

Signal Recovery and Analysis

This section detects, analyzes, and displays a summary of the detected packets. All packets in the waveform must have the specified channel bandwidth. Parse and analyze the packets within a waveform by using the `WaveformAnalyzer` object.

```
analyzer = WaveformAnalyzer;
process(analyzer,rxWaveform,chanBW,sr);
```

Display a summary of the detected packets.

```
detectionSummary(analyzer);
```

Summary of the Detecte

detSummary=*11×9 table*

Number	Format	PHY Status	Power (dBm)	CFO (Hz)	Offset (samples)	MAC Conte
1	"Non-HT"	"Success"	12.7	61431	97	"Beacon"
2	"Non-HT"	"Success"	13.08	-39757	2577	"RTS"
3	"Non-HT"	"Success"	13.01	62250	4017	"CTS"
4	"HE-MU"	"Success"	14.98	-39660	5297	"A-MPDU"
5	"Non-HT"	"Success"	13.04	-39437	18657	"Block Ac
6	"Non-HT"	"Success"	13.07	-29899	20417	"RTS"
7	"Non-HT"	"Success"	13.01	52489	21857	"CTS"
8	"VHT"	"Success"	17.43	62290	23137	"A-MPDU"
9	"Non-HT"	"Success"	14.99	-38861	28337	"RTS"
10	"Non-HT"	"Success"	14.94	42363	29777	"CTS"
11	"HT-MF"	"Success"	15.03	22238	31058	"A-MPDU"

Use the `pktNum` variable to display the MAC and PHY analysis for a selected packet.

```
pktNum = 4 ;
```

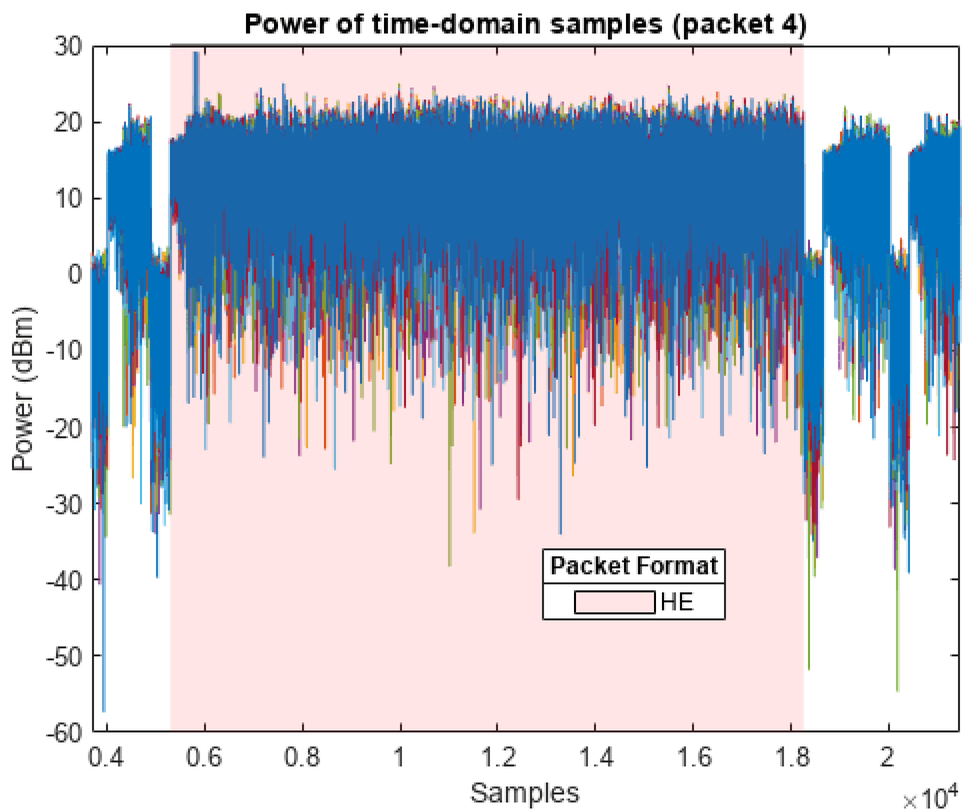
Display the MAC information of the selected packet.

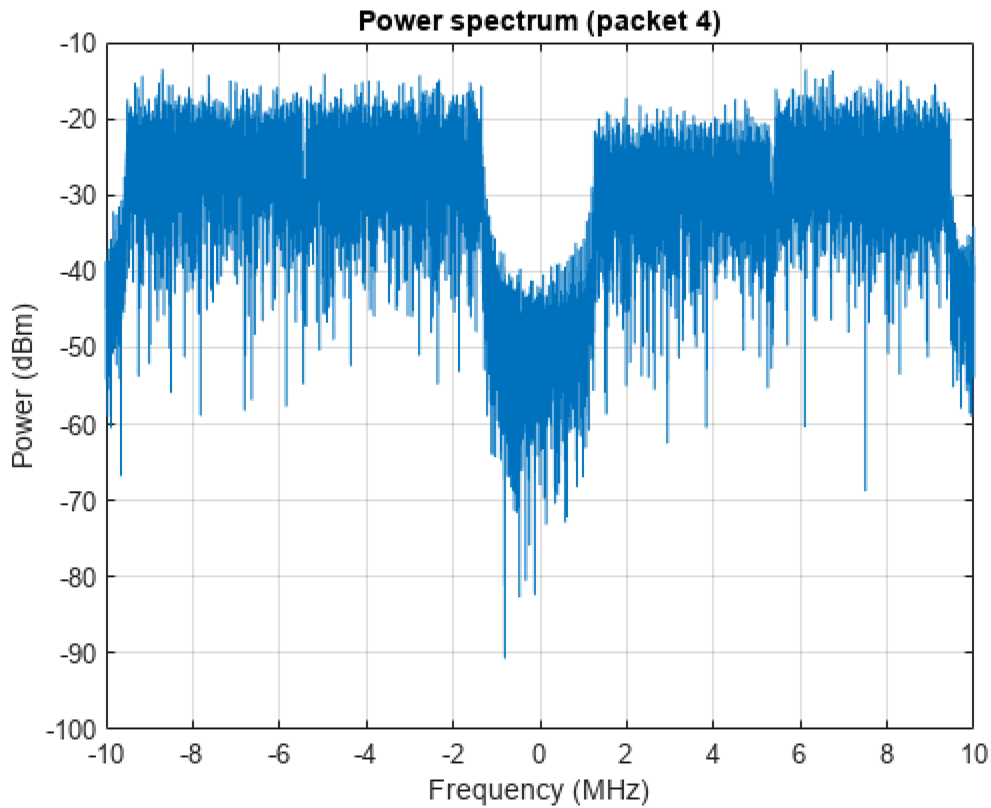
```
macSummary(analyzer,pktNum);
```

```
                    Recovered MPDU Summary of Packet 4

    AMPDU/MPDU Number    STAID      Address1          Address2        AMPDU/MPDU Decode Status
    _________________    _____    ______________    ______________    ________________________

     "AMPDU1_MPDU1"        1      "1342ABC2FF1F"    "00123456789B"           "Success"
     "AMPDU2_MPDU1"        2      "23FFAB1234AC"    "00123456789B"           "Success"
     "AMPDU3_MPDU1"        3      "13EF35781356"    "00123456789B"           "Success"
     "AMPDU4_MPDU1"        4      "159A123AFFFF"    "00123456789B"           "Success"
```

Display the time samples and spectrum of the detected packet.

```
plotWaveform(analyzer,pktNum)
```





Display the packet field information of the selected packet.

```
fieldSummary(analyzer,pktNum);
```

```
                                    Field Summary of Packet 4 (HE-MU)

    Field Name      Modulation      Num Symbols      Parity Check/CRC      Power (dBm)      RMS EVM (dB)
    __________      __________      ___________      ________________      ___________      ____________

     L-STF          BPSK                 2                                    14.59
     L-LTF          BPSK                 2                                    14.61
     L-SIG          BPSK                 1                 Pass               14.96            -27.59
     RL-SIG         BPSK                 1                 Pass               14.86            -27.14
     HE-SIG-A       BPSK                 2                 Pass               15.37            -26.14
     HE-SIG-B       BPSK                 5                 Pass               14.98            -27.56
     HE-STF         BPSK                 1                                    14.95
     HE-LTF         BPSK                 2                                    15.01
     Data                               35                                    14.99            -25.35
```

Display the signaling field information of the selected packet.

```
signalingSummary(analyzer,pktNum);
```

```
                        Signaling Field Summary of Packet 4 (HE-MU)

        Property        Value          Property          Value          Property            Value
```

L-SIG Length	467	Bandwidth	CBW20	Num HE-LTF Symbols	2
L-SIG Rate	0xB	Num HE-SIG-B Symbols	5	LDPC Extra Symbol	True
UL/DL Indication	DL	SIGB Compression	False	STBC	False
SIGB MCS	0	Guard Interval	3.2	Pre-FEC Padding Factor	1
SIGB DCM	False	HE-LTF Type	4	PE Disambiguity	False
BSS Color	0	Doppler	False		
Spatial Reuse	0	TXOP	127		

Display the RU information.

```
ruSummary(analyzer,pktNum);
```

Resource Unit (RU) Information of Packet 4 (HE-MU)

RU Number	RU Size	Subcarrier Index (Start)	Subcarrier Index (End)	Num Users	Nu
"RU1"	52	-121	-70	1	
"RU2"	52	-68	-17	1	
"RU3"	52	17	68	1	
"RU4"	52	70	121	1	

Display the user information.

```
userSummary(analyzer,pktNum);
```

User Information of Packet 4 (HE-MU)

STAID	RU Number	MCS	Modulation	Code Rate	DCM	Channel Coding	Num STS	T
1	"RU1"	0	"BPSK"	"1/2"	0	"LDPC"	1	
2	"RU2"	2	"QPSK"	"3/4"	0	"LDPC"	1	
3	"RU3"	4	"16QAM"	"3/4"	0	"LDPC"	2	
4	"RU4"	6	"64QAM"	"3/4"	0	"LDPC"	1	

Display EVM per spatial streams for all users.

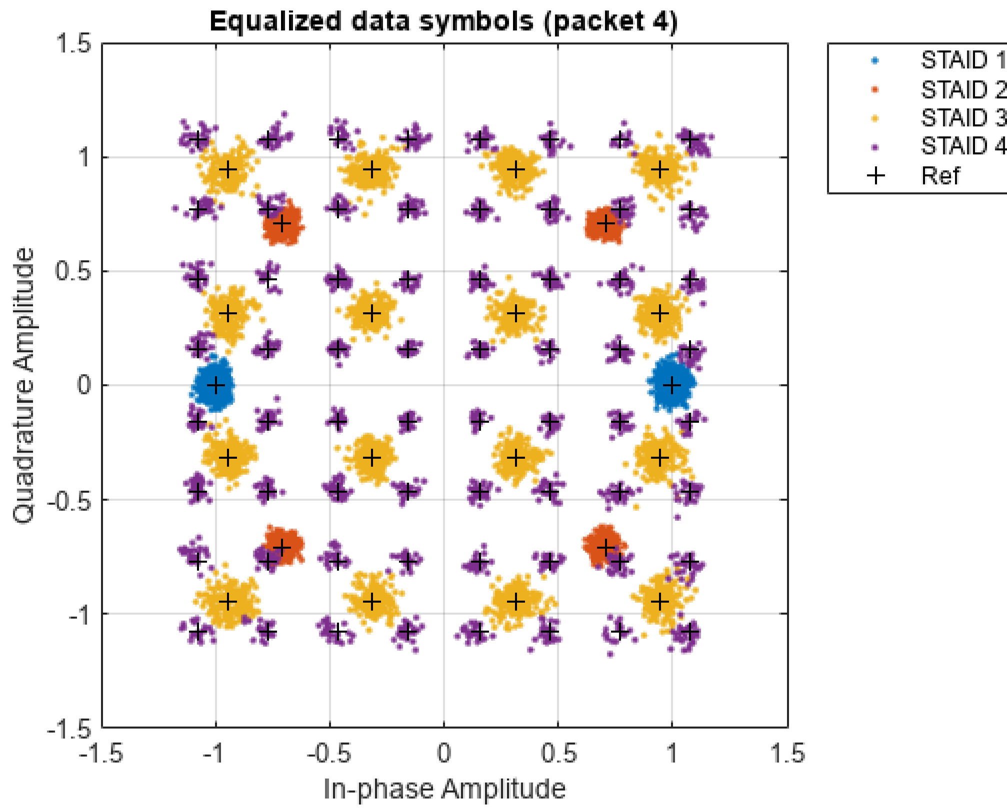
```
userEVM(analyzer,pktNum);
```

User EVM per Spatial Stream of Packet 4 (HE-MU)

STAID	Spatial Stream Index	RMS EVM (dB)	Max EVM (dB)
1	1	-26.391	-17.295
2	1	-27.401	-19.682
3	1	-23.564	-12.353
3	2	-23.122	-14.444
4	1	-27.278	-17.793

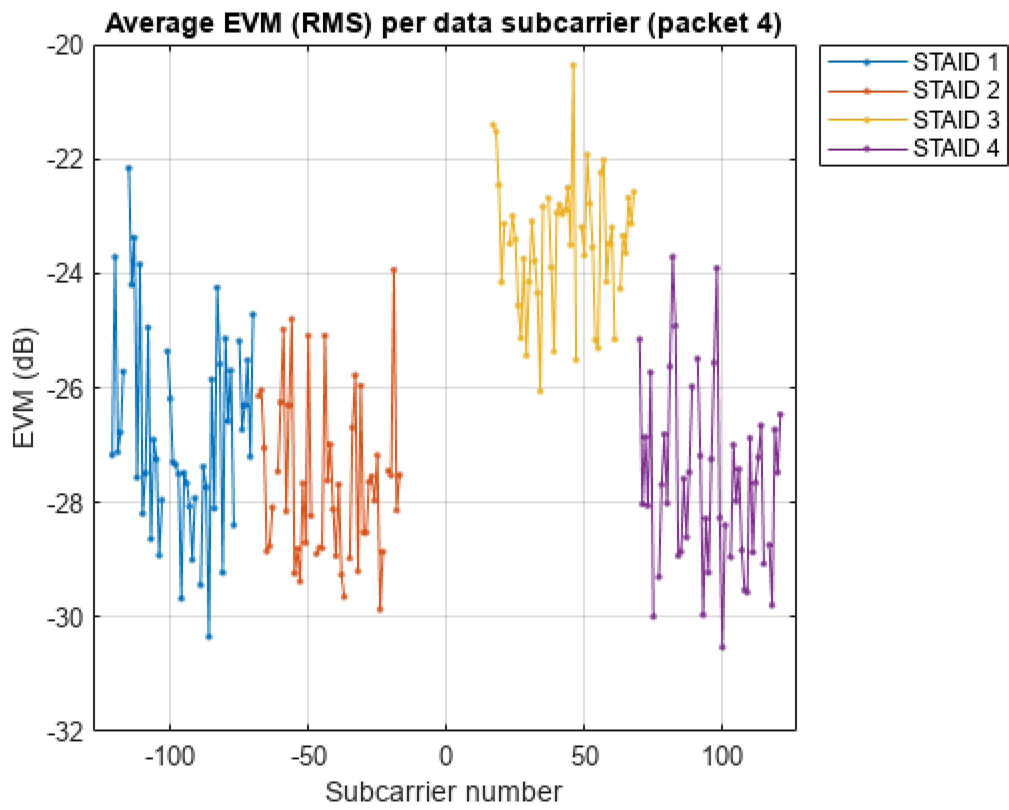
Plot the constellation for all users.

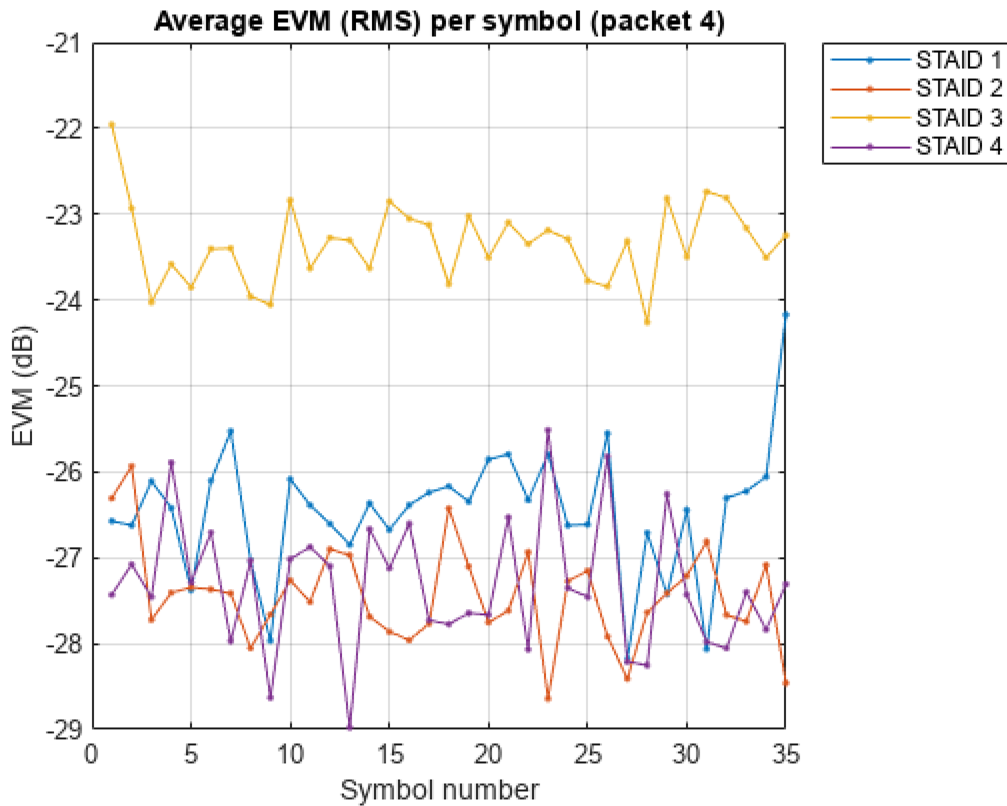
```
plotConstellation(analyzer,pktNum);
```



Plot the EVM.

```
plotEVM(analyzer,pktNum);
```


Average EVM (RMS) per data subcarrier (packet 4)
EVM (dB)
Subcarrier number
STAID 1
STAID 2
STAID 3
STAID 4
-20
-22
-24
-26
-28
-30
-32
-100
-50
0
50
100



Plot the spectral flatness for non-OFDMA packets.

```
plotSpectralFlatness(analyzer,chanBW,pktNum);
```

Further Exploration

The `WaveformAnalyzer` provides properties to control the pilot tracking, equalization, DC blocking, and packet detection algorithms that can be tweaked to improve packet detection and analysis performance.

False packet detections

False packet detections are detected packets that you do not believe are actual packets. Evaluating the time domain waveform of the packet is one way to determine if the detected packet is legitimate. If there is a significant number of false detections present these techniques may help reduce them:

- Enable the `EnergyDetection` property and set the `EnergyDetectionThreshold` property to a suitable value given the noise floor of the capture device. When enabled, `EnergyDetection` only detects packets with a power exceeding `EnergyDetectionThreshold` during the preamble.
- Increase the `LLTFSNRDetectionThreshold` and `PacketDetectionThreshold` properties to discard packets with a low measured SNR during detection.

Missed packet detections

Missed packet detections are packets that you believe are in the waveform but have not been detected.

- One possible reason a packet detection may have been missed is if a false detection occurred earlier in the waveform, but the L-SIG check passed, causing samples to be skipped. To search within possible false detections, enable the `SearchWithinUnsupportedPacket` property.
- Alternatively try decreasing the `PacketDetectionThreshold` property to detect packet with low SNR during detection.

For detail on 802.11ax and 802.11ac signal recovery and processing, see the “Recovery Procedure for an 802.11ax Packet” on page 4-13 and “Recovery Procedure for an 802.11ac Packet” on page 4-30 examples.

References

1. IEEE Std 802.11™ - 2020 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.
2. IEEE 802.11ax™ - 2021 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 6: Enhancements for High Efficiency WLAN.

Local Functions

These functions assist in SDR set up.

```
function options = getDeviceOptions
% GETDEVICEOPTIONS returns a string array with all saved Wireless
% Testbench radio configurations and Communication toolbox support package
% SDRs
spkgSDRs = getSupportPackageSDRs;

% Check for WT configurations if user has WT installed and a valid license
if ~isempty(ver('wt')) && license('test','Wireless_Testbench')
    savedRadioConfigurations = radioConfigurations;
    savedRadioConfigurationNames = [string({savedRadioConfigurations.Name})];
else
    savedRadioConfigurationNames = [];
end

options = [savedRadioConfigurationNames spkgSDRs];
end

function options = getAntennaOptions(deviceName)
%GETANTENNAOPTIONS returns a string array of valid antenna configurations
%for a specified DEVICENAME
% Acquire valid antenna values based on radio
if ~matches(deviceName,getSupportPackageSDRs)
    antennas = hCaptureAntennas(deviceName);
elseif matches(deviceName,["X300","FMCOMMS5"])
    antennas = [1 2 3 4]';
elseif matches(deviceName,["Pluto", "B200", "N200/N210/USRP2"])
    antennas = 1;
else
    antennas = [1 2]';
```

```
    end

    % Generate a string array list of all valid antenna configurations
    options = strings(0,1);
    for a = 1:length(antennas)
        % Generate unique combinations
        AntennaCombinations = nchoosek(antennas,a);
        % Generate list of all unique combinations
        for i = 1:size(AntennaCombinations, 1)
            options = [options;string(mat2str(AntennaCombinations(i,:)))]; %#ok<AGROW>
        end
    end
end

function spkgSDRs = getSupportPackageSDRs
%GETSUPPORTPACKAGESDRs returns a string array of Communication Toolbox
%support package SDRs that are supported for this example
spkgSDRs = ["Pluto" "AD936x" "FMCOMMS5" "E3xx" "B200" "B210" "N200/N210/USRP2" "X300"];
end

function rx = getSDRObject(deviceName,antennas,fc,sr,gain,rx)
%GETSDROBJECT returns a Wireless Testbench basebandReceiver object
%or an hSDRReceiver object with the properties of the object set according
%to the input parameters

% Create SDR object for waveform capture and set object's properties
if matches(deviceName,getSupportPackageSDRs)
    rx = hSDRReceiver(deviceName);
    rx.OutputDataType = 'double';
    rx.ChannelMapping = eval(antennas);
    rx.Gain = gain;
else % Wireless Testbench hardware
    % Keep existing basebandReceiver object for performance purposes
    if ~isa(rx,'basebandReceiver')
        rx = basebandReceiver(deviceName);
    end
    rx.CaptureDataType = 'double';
    rx.Antennas = eval(antennas);
    rx.RadioGain = gain;
end
rx.SampleRate = sr;
rx.CenterFrequency = fc;
end
```

Recovery Procedure for an 802.11ax Packet

This example shows how to detect a packet and decode payload bits in a received IEEE® 802.11ax™ waveform. The receiver recovers the packet format parameters from the preamble fields to decode the data field and the MAC frame.

Introduction

In an 802.11ax packet the transmission parameters are signaled to the receiver using the L-SIG, HE-SIG-A, and HE-SIG-B preamble fields [1]:

- The L-SIG field contains information to allow the receiver to determine the transmission time of the packet.
- The HE-SIG-A field contains common transmission parameters for HE-MU users and all transmission parameters for HE-SU and HE-EXT-SU packets.
- The combination of length information in the L-SIG field, the modulation type, and the number of OFDM symbols in the HE-SIG-A field determines the HE packet format.
- The HE-SIG-B field contains Resource unit (RU) allocation information and the transmission parameters for the users in an HE-MU packet.

In this example we detect and decode an HE-MU packet within a generated waveform. This example can also recover HE-SU and HE-EXT-SU packets. All transmission parameters apart from the channel bandwidth are assumed to be unknown and are therefore retrieved from the decoded L-SIG, HE-SIG-A, and HE-SIG-B preamble fields. The recovered transmission parameters are used to decode the HE-Data field. Additionally, the following analysis is performed:

- The waveform of the detected packet is recovered and displayed.
- The spectrum of the detected packet is recovered and displayed.
- The constellation of the equalized data symbols for all spatial streams is displayed.
- The Error Vector Magnitude (EVM) of each field is measured.
- An A-MPDU is detected and Frame Check Sequence (FCS) is determined for the recovered MAC frame.
- The EVM per data symbol and spatial stream averaged over subcarriers is displayed.
- The EVM per data subcarrier and spatial stream averaged over symbols is displayed.

Waveform Transmission

In this example an 802.11ax HE-MU waveform is synthesized but you can use a captured waveform. You can use MATLAB® to acquire I/Q data from a wide range of instruments with the Instrument Control Toolbox™ and software defined radio platforms.

The synthesized waveform is impaired by a 2x2 TGax indoor fading channel, additive white Gaussian noise, and carrier frequency offset. To generate an HE-MU waveform we configure an HE-MU format configuration object wlanHEMUConfig. Note that the wlanHEMUConfig configuration object is used at the transmitter side only. The receiver will formulate an HE recovery configuration object wlanHERecoveryConfig. The unknown properties of the HE recovery configuration object are set after decoding the information bits in the L-SIG, HE-SIG-A, and HE-SIG-B fields. The helper function heSigRecGenerateWaveform generates the impaired waveform. The following processing steps are performed:

- A random payload of MSDUs is created for the MAC frame, which is encoded into an HE-MU packet.
- The waveform is passed through a TGax indoor fading channel model.
- Carrier frequency offset (CFO) and Additive white Gaussian noise (AWGN) are added to the waveform.

```
% A mixed OFDMA and MU-MIMO configuration is defined for an HE-MU packet.
% The allocation index 17 defines two 52-tone RUs, with one user in each
% RU, and one 106-tone RU. The 106-tone RU has two users in a MU-MIMO
% configuration.
cfgMU = wlanHEMUConfig(17);
cfgMU.NumTransmitAntennas = 2;

% Configure RU 1 and user 1
cfgMU.RU{1}.SpatialMapping = 'Direct';
cfgMU.User{1}.STAID = 1;
cfgMU.User{1}.APEPLength = 1e3;
cfgMU.User{1}.MCS = 5;
cfgMU.User{1}.NumSpaceTimeStreams = 2;
cfgMU.User{1}.ChannelCoding = 'LDPC';

% Configure RU 2 and user 2
cfgMU.RU{2}.SpatialMapping = 'Fourier';
cfgMU.User{2}.STAID = 2;
cfgMU.User{2}.APEPLength = 500;
cfgMU.User{2}.MCS = 4;
cfgMU.User{2}.NumSpaceTimeStreams = 1;
cfgMU.User{2}.ChannelCoding = 'BCC';

% Configure RU 3 and user 1
cfgMU.RU{3}.SpatialMapping = 'Fourier';
cfgMU.User{3}.STAID = 3;
cfgMU.User{3}.APEPLength = 100;
cfgMU.User{3}.MCS = 2;
cfgMU.User{3}.NumSpaceTimeStreams = 1;
cfgMU.User{3}.ChannelCoding = 'BCC';

% Configure RU 3 and user 2
cfgMU.User{4}.STAID = 4;
cfgMU.User{4}.APEPLength = 500;
cfgMU.User{4}.MCS = 3;
cfgMU.User{4}.NumSpaceTimeStreams = 1;
cfgMU.User{4}.ChannelCoding = 'LDPC';

% Specify propagation channel
numRx = 2; % Number of receive antennas
delayProfile = 'Model-D'; % TGax channel delay profile

% Specify impairments
noisePower = -40; % Noise power to apply in dBW
cfo = 62e3; % Carrier frequency offset Hz

% Generate waveform
rx = heSigRecGenerateWaveform(cfgMU,numRx,delayProfile,noisePower,cfo);
```

Packet Recovery

The signal to process is stored in the variable `rx`. The processing steps to recover a packet are:

- An HE recovery configuration object, wlanHERecoveryConfig is created. The object properties are updated as preamble fields are decoded.
- The packet is detected and synchronized.
- The L-LTF is extracted and demodulated. The demodulated L-LTF symbols do not include tone rotation for each 20 MHz segment as described in [2], section 21.3.7.5.
- The demodulated L-LTF symbols are used for channel and noise estimates.
- The time-domain signal containing samples equivalent to four OFDM symbols immediately following the L-LTF are used to determine the HE packet format. The packet format is updated in the wlanHERecoveryConfig object.
- The L-LTF is demodulated. The demodulated L-LTF symbols include tone rotation for each 20 MHz segment as described in [2], section 21.3.7.5. The L-LTF channel estimates (with tone rotation) are used to decode the pre-HE-LTF.
- The L-SIG and RL-SIG fields are extracted. The channel is estimated on an extra four subcarriers per subchannel in the L-SIG and RL-SIG fields. The L-LTF channel estimates are updated to include the channel estimates on the extra subcarriers.
- The information bits in the L-SIG field are recovered to determine the length of the packet in microseconds.
- After HE-SIG-A decoding, the recovery configuration object is updated with common transmission parameters for an HE-MU packet and all transmission parameters for HE-SU and HE-EXT-SU packets.
- For an HE-MU packet format the HE-SIG-B field is decoded. For a non-compressed SIGB waveform the HE-SIG-B common field is decoded first followed by HE-SIG-B user field. For a compressed SIGB waveform only the HE-SIG-B user field is decoded.
- For an HE-MU format without SIGB compression the RU allocation and user transmission parameters are recovered from the HE-SIG-B field. For a compressed SIGB waveform the RU allocation information is inferred from HE-SIG-A field and the user transmission parameters are determined from the HE-SIG-B user field bits.
- The wlanHERecoveryConfig object is created using the recovered transmission parameters for each user after HE-SIG-B decoding.
- The HE-LTF field is extracted and demodulated. The demodulated symbols are used for channel estimation of the subcarriers allocated to the user of interest. The MIMO channel estimates are used to decode the HE-Data field.
- The HE-Data field is extracted and the PSDU bits are recovered using the wlanHERecoveryConfig object for each user.
- Detect A-MPDU within the recovered PSDU and check the FCS for the recovered MAC frame.

Setup Waveform Recovery Parameters

In this example all transmission parameters apart from the channel bandwidth are assumed to be unknown and will be recovered. A recovery configuration object, wlanHERecoveryConfig, is created to store the recovered information in the L-SIG, HE-SIG-A, and HE-SIG-B preamble fields. The transmission properties in wlanHERecoveryConfig are updated after subsequent decoding of the preamble fields. The following code configures objects and variables for processing.

```
chanBW = cfgMU.ChannelBandwidth; % Assume channel bandwidth is known
sr = wlanSampleRate(cfgMU); % Sample rate
```

```
% Specify pilot tracking method for recovering the data field. This can be:
%  'Joint' - use joint common phase error and sample rate offset tracking
%  'CPE' - use only common phase error tracking
% When recovering 26-tone RUs only CPE tracking is used as the joint
% tracking algorithm is susceptible to noise.
pilotTracking = 'Joint';

% Create an HE recovery configuration object and set the channel bandwidth
cfgRx = wlanHERecoveryConfig;
cfgRx.ChannelBandwidth = chanBW;

% The recovery configuration object is used to get the start and end
% indices of the pre-HE-SIG-B field.
ind = wlanFieldIndices(cfgRx);

% Setup plots for the example
[spectrumAnalyzer,timeScope,ConstellationDiagram,EVMPerSubcarrier,EVMPerSymbol] = heSigRecSetupP

% Minimum packet length is 10 OFDM symbols
lstfLength = double(ind.LSTF(2));
minPktLen = lstfLength*5; % Number of samples in L-STF

rxWaveLen = size(rx,1);
```

Front-End Processing

The front-end processing consists of packet detection, coarse carrier frequency offset correction, timing synchronization, and fine carrier frequency offset correction. A `while` loop is used to detect and synchronize a packet within the received waveform. The sample offset `searchOffset` is used to index into `rx` to detect a packet. The first packet within `rx` is detected and processed. If the synchronization fails for the detected packet, the sample index offset `searchOffset` is incremented to move beyond the processed packet in `rx`. This is repeated until a packet has been successfully detected and synchronized.

```
searchOffset = 0; % Offset from start of waveform in samples
while (searchOffset + minPktLen) <= rxWaveLen
    % Packet detection
    pktOffset = wlanPacketDetect(rx,chanBW,searchOffset);

    % Adjust packet offset
    pktOffset = searchOffset + pktOffset;
    if isempty(pktOffset) || (pktOffset + ind.LSIG(2) > rxWaveLen)
        error('** No packet detected **');
    end

    % Coarse frequency offset estimation and correction using L-STF
    rxLSTF = rx(pktOffset+(ind.LSTF(1):ind.LSTF(2)), :);
    coarseFreqOffset = wlanCoarseCFOEstimate(rxLSTF,chanBW);
    rx = frequencyOffset(rx,sr,-coarseFreqOffset);

    % Symbol timing synchronization
    searchBufferLLTF = rx(pktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
    pktOffset = pktOffset+wlanSymbolTimingEstimate(searchBufferLLTF,chanBW);

    % Fine frequency offset estimation and correction using L-STF
    rxLLTF = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
```

```
    fineFreqOffset = wlanFineCFOEstimate(rxLLTF,chanBW);
    rx = frequencyOffset(rx,sr,-fineFreqOffset);

    % Timing synchronization complete: packet detected
    fprintf('Packet detected at index %d\n',pktOffset + 1);

    % Display estimated carrier frequency offset
    cfoCorrection = coarseFreqOffset + fineFreqOffset; % Total CFO
    fprintf('Estimated CFO: %5.1f Hz\n\n',cfoCorrection);

    break; % Front-end processing complete, stop searching for a packet
end

% Scale the waveform based on L-STF power (AGC)
gain = 1./(sqrt(mean(rxLSTF.*conj(rxLSTF))));
rx = rx.*gain;
```

```
Packet detected at index 404
Estimated CFO: 61942.9 Hz
```

Packet Format Detection

The time-domain samples equivalent to four OFDM symbols immediately following the L-LTF are used to determine the HE packet format [1 Figure. 27-63]. The L-LTF is extracted and demodulated. For format detection, the demodulated L-LTF symbols must not include tone rotation for each 20 MHz segment as described in [2], section 21.3.7.5. The demodulated L-LTF is used for channel and noise estimation. The L-LTF channel (without tone rotation) and noise power estimates are used to detect the packet format.

```
rxLLTF = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
lltfDemod = wlanLLTFDemodulate(rxLLTF,chanBW);
lltfChanEst = wlanLLTFChannelEstimate(lltfDemod,chanBW);
noiseVar = wlanLLTFNoiseEstimate(lltfDemod);

disp('Detect packet format...');
rxSIGA = rx(pktOffset+(ind.LSIG(1):ind.HESIGA(2)),:);
pktFormat = wlanFormatDetect(rxSIGA,lltfChanEst,noiseVar,chanBW);
fprintf('  %s packet detected\n\n',pktFormat);

% Set the packet format in the recovery object and update the field indices
cfgRx.PacketFormat = pktFormat;
ind = wlanFieldIndices(cfgRx);
```

```
Detect packet format...
  HE-MU packet detected
```

L-LTF Channel Estimate

Demodulate the L-LTF and perform channel estimation. The demodulated L-LTF symbols include tone rotation for each 20 MHz segment as described in [2], section 21.3.7.5. The L-LTF channel estimates (with tone rotation) are used to equalize and decode the pre-HE-LTF fields.

```
lltfDemod = wlanHEDemodulate(rxLLTF,'L-LTF',chanBW);
lltfChanEst = wlanLLTFChannelEstimate(lltfDemod,chanBW);
```

L-SIG and RL-SIG Decoding

The L-SIG field is used to determine the receive time, or RXTIME, of the packet. The RXTIME is calculated using the length bits of the L-SIG payload. The L-SIG and RL-SIG fields are recovered to perform the channel estimate on the extra subcarriers in the L-SIG and RL-SIG fields. The `lltfChanEst` channel estimates are updated to include the channel estimates on extra subcarriers in the L-SIG and RL-SIG fields. The L-SIG payload is decoded using an estimate of the channel and noise power obtained from the L-LTF field. The L-SIG length property in wlanHERecoveryConfig is updated after L-SIG decoding.

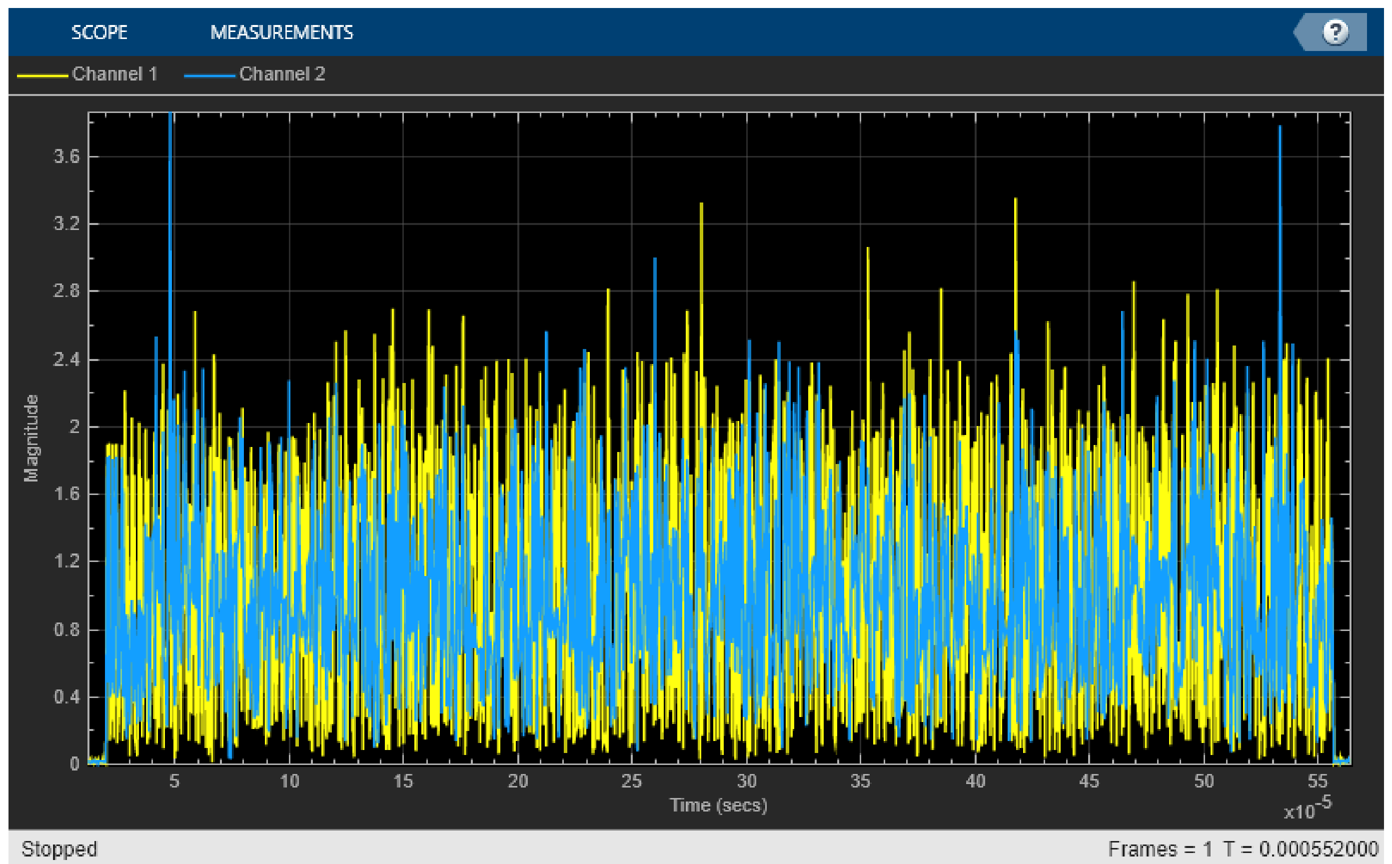
```
disp('Decoding L-SIG... ');
% Extract L-SIG and RL-SIG fields
rxLSIG = rx(pktOffset+(ind.LSIG(1):ind.RLSIG(2)),:);

% OFDM demodulate
helsigDemod = wlanHEDemodulate(rxLSIG,'L-SIG',chanBW);

% Estimate CPE and phase correct symbols
helsigDemod = wlanHETrackPilotError(helsigDemod,lltfChanEst,cfgRx,'L-SIG');

% Estimate channel on extra 4 subcarriers per subchannel and create full
% channel estimate
preheInfo = wlanHEOFDMInfo('L-SIG',chanBW);
preHEChanEst = wlanPreHEChannelEstimate(helsigDemod,lltfChanEst,chanBW);

% Average L-SIG and RL-SIG before equalization
helsigDemod = mean(helsigDemod,2);

% Equalize data carrying subcarriers, merging 20 MHz subchannels
[eqLSIGSym,csi] = preHESymbolEqualize(helsigDemod(preheInfo.DataIndices,:,:), ...
    preHEChanEst(preheInfo.DataIndices,:,:),noiseVar,preheInfo.NumSubchannels);

% Decode L-SIG field
[~,failCheck,lsigInfo] = wlanLSIGBitRecover(eqLSIGSym,noiseVar,csi);

if failCheck
    disp(' ** L-SIG check fail **');
else
    disp(' L-SIG check pass');
end
% Get the length information from the recovered L-SIG bits and update the
% L-SIG length property of the recovery configuration object
lsigLength = lsigInfo.Length;
cfgRx.LSIGLength = lsigLength;

% Measure EVM of L-SIG symbols
EVM = comm.EVM;
EVM.ReferenceSignalSource = 'Estimated from reference constellation';
EVM.Normalization = 'Average constellation power';
EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK');
rmsEVM = EVM(eqLSIGSym);
fprintf(' L-SIG EVM: %2.2fdB\n\n',20*log10(rmsEVM/100));

% Calculate the receive time and corresponding number of samples in the
% packet
RXTime = ceil((lsigLength + 3)/3) * 4 + 20; % In microseconds
numRxSamples = round(RXTime * 1e-6 * sr);   % Number of samples in time
```

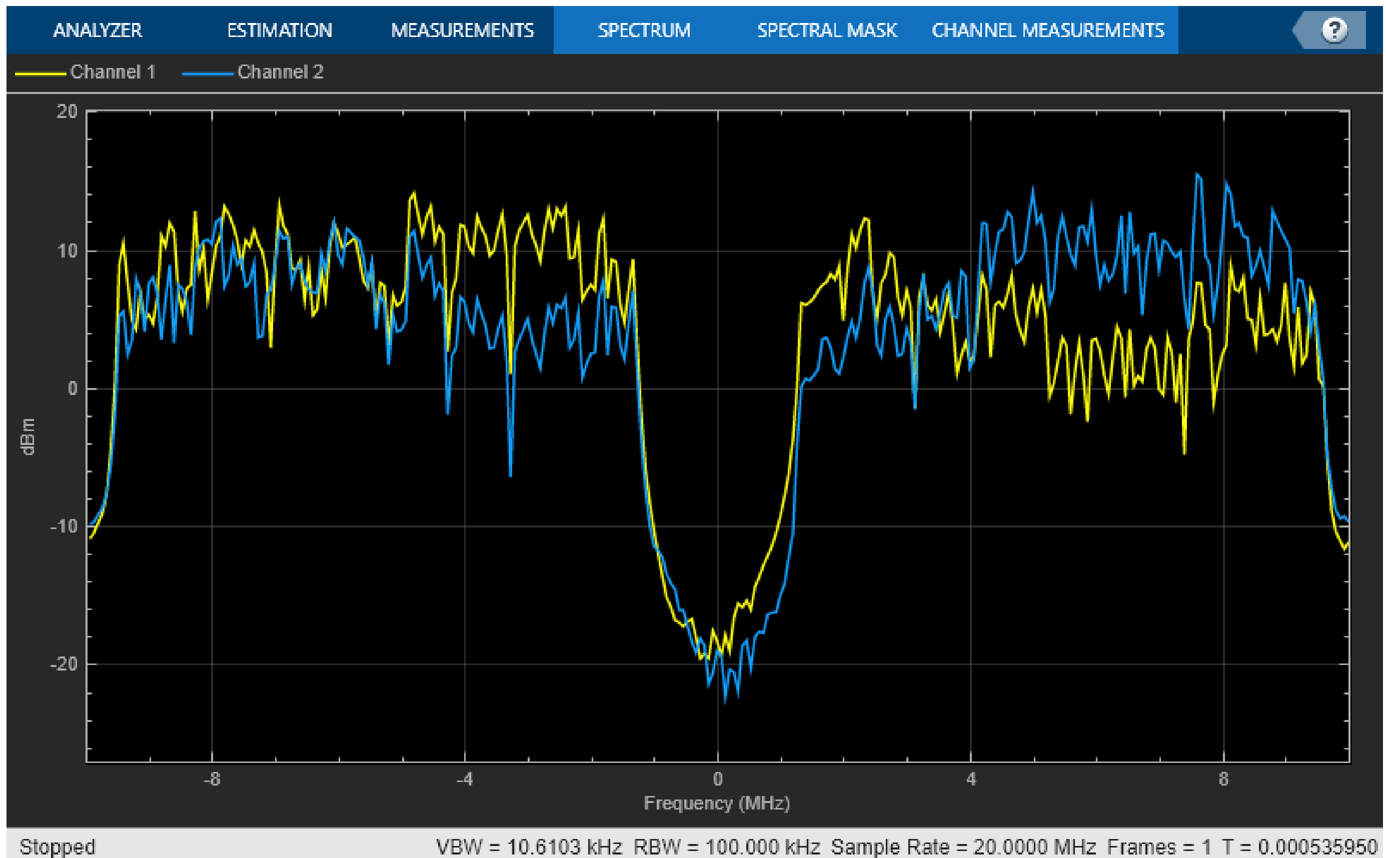
```
fprintf(' RXTIME: %dus\n',RXTime);
fprintf(' Number of samples in the packet: %d\n\n',numRxSamples);
```

```
Decoding L-SIG...
 L-SIG check pass
 L-SIG EVM: -36.91dB

 RXTIME: 536us
 Number of samples in the packet: 10720
```

The waveform and spectrum of the detected packet within `rx` is displayed given the calculated RXTIME and corresponding number of samples.

```
sampleOffset = max((-lstfLength + pktOffset),1); % First index to plot
sampleSpan = numRxSamples + 2*lstfLength; % Number samples to plot
% Plot as much of the packet (and extra samples) as we can
plotIdx = sampleOffset:min(sampleOffset + sampleSpan,rxWaveLen);

% Configure timeScope to display the packet
timeScope.TimeSpan = sampleSpan/sr;
timeScope.TimeDisplayOffset = sampleOffset/sr;
timeScope.YLimits = [0 max(abs(rx(:)))];
timeScope(abs(rx(plotIdx,:)));
release(timeScope);

% Display the spectrum of the detected packet
spectrumAnalyzer(rx(pktOffset + (1:numRxSamples),:));
release(spectrumAnalyzer);
```


SCOPE
MEASUREMENTS
Channel 1
Channel 2
Magnitude
3.6
3.2
2.8
2.4
2
1.6
1.2
0.8
0.4
0
5
10
15
20
25
30
35
40
45
50
55
Time (secs)
x10^-5
Stopped
Frames = 1 T = 0.000552000



HE-SIG-A Decoding

The HE-SIG-A field contains the transmission configuration of an HE packet. An estimate of the channel and noise power obtained from the L-LTF is required to decode the HE-SIG-A field.

```
disp('Decoding HE-SIG-A...')
rxSIGA = rx(pktOffset+(ind.HESIGA(1):ind.HESIGA(2)),:);
sigaDemod = wlanHEDemodulate(rxSIGA,'HE-SIG-A',chanBW);
hesigaDemod = wlanHETrackPilotError(sigaDemod,preHEChanEst,cfgRx,'HE-SIG-A');

% Equalize data carrying subcarriers, merging 20 MHz subchannels
preheInfo = wlanHEOFDMInfo('HE-SIG-A',chanBW);
[eqSIGASym,csi] = preHESymbolEqualize(hesigaDemod(preheInfo.DataIndices,:,:), ...
                                      preHEChanEst(preheInfo.DataIndices,:,:), ...
                                      noiseVar,preheInfo.NumSubchannels);
% Recover HE-SIG-A bits
[sigaBits,failCRC] = wlanHESIGABitRecover(eqSIGASym,noiseVar,csi);

% Perform the CRC on HE-SIG-A bits
if failCRC
    disp(' ** HE-SIG-A CRC fail **');
else
    disp(' HE-SIG-A CRC pass');
end

% Measure EVM of HE-SIG-A symbols
release(EVM);
```

```
if strcmp(pktFormat,'HE-EXT-SU')
    % The second symbol of an HE-SIG-A field for an HE-EXT-SU packet is
    % QBPSK.
    EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK',[0 pi/2 0 0]);
    % Account for scaling of L-LTF for an HE-EXT-SU packet
    rmsEVM = EVM(eqSIGASym*sqrt(2));
else
    EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK');
    rmsEVM = EVM(eqSIGASym);
end
fprintf(' HE-SIG-A EVM: %2.2fdB\n\n',20*log10(mean(rmsEVM)/100));
```

```
Decoding HE-SIG-A...
 HE-SIG-A CRC pass
 HE-SIG-A EVM: -35.17dB
```

Interpret Recovered HE-SIG-A bits

The wlanHERecoveryConfig object is updated after interpreting the recovered HE-SIG-A bits.

```
cfgRx = interpretHESIGABits(cfgRx,sigaBits);
ind = wlanFieldIndices(cfgRx); % Update field indices
```

Display the common transmission configuration obtained from HE-SIG-A field for an HE-MU packet. The properties indicated by -1 are unknown or undefined. The unknown user-related properties are updated after successful decoding of the HE-SIG-B field.

```
disp(cfgRx)
```

```
  wlanHERecoveryConfig with properties:

                     PacketFormat: 'HE-MU'
                 ChannelBandwidth: 'CBW20'
                       LSIGLength: 383
                   SIGBCompression: 0
                          SIGBMCS: 0
                          SIGBDCM: 0
           NumSIGBSymbolsSignaled: 5
                             STBC: 0
                   LDPCExtraSymbol: 0
               PreFECPaddingFactor: 4
                   PEDisambiguity: 0
                    GuardInterval: 3.2000
                        HELTFType: 4
                  NumHELTFSymbols: 2
                 UplinkIndication: 0
                         BSSColor: 0
                     SpatialReuse: 0
                     TXOPDuration: 127
                      HighDoppler: 0
                  AllocationIndex: -1
         NumUsersPerContentChannel: -1
           RUTotalSpaceTimeStreams: -1
                           RUSize: -1
                          RUIndex: -1
                            STAID: -1
                              MCS: -1
```

```
                      DCM: -1
            ChannelCoding: 'Unknown'
              Beamforming: -1
       NumSpaceTimeStreams: -1
SpaceTimeStreamStartingIndex: -1
```

HE-SIG-B Decoding

For an HE-MU packet the HE-SIG-B field contains:

- The RU allocation information for a non-compressed SIGB waveform is inferred from HE-SIG-B Common field [1 Table. 27-24]. For a compressed SIGB waveform the RU allocation information is inferred from the recovered HE-SIG-A bits.
- For a non-compressed SIGB waveform the number of HE-SIG-B symbols are updated in the wlanHERecoveryConfig object. The symbols are only updated if the number of HE-SIG-B symbols indicated in the HE-SIG-A field is set to 15 and all content channels pass the CRC. The number of HE-SIG-B symbols indicated in the HE-SIG-A field are not updated if any HE-SIG-B content channel fails the CRC.
- The user transmission parameters for both SIGB compressed and non-compressed waveforms are inferred from the HE-SIG-B user field [1 Table. 27-27, 27-28].

An estimate of the channel and noise power obtained from the L-LTF is required to decode the HE-SIG-B field.

```
if strcmp(pktFormat,'HE-MU')
    fprintf('Decoding HE-SIG-B...\n');
    if ~cfgRx.SIGBCompression
        fprintf(' Decoding HE-SIG-B common field...\n');
        s = getSIGBLength(cfgRx);
        % Get common field symbols. The start of HE-SIG-B field is known
        rxSym = rx(pktOffset+(double(ind.HESIGA(2))+(1:s.NumSIGBCommonFieldSamples)),:);
        % Decode HE-SIG-B common field
        [status,cfgRx] = heSIGBCommonFieldDecode(rxSym,preHEChanEst,noiseVar,cfgRx);

        % CRC on HE-SIG-B content channels
        if strcmp(status,'Success')
            fprintf('  HE-SIG-B (common field) CRC pass\n');
        elseif strcmp(status,'ContentChannel1CRCFail')
            fprintf('  ** HE-SIG-B CRC fail for content channel-1\n **');
        elseif strcmp(status,'ContentChannel2CRCFail')
            fprintf('  ** HE-SIG-B CRC fail for content channel-2\n **');
        elseif any(strcmp(status,{'UnknownNumUsersContentChannel1','UnknownNumUsersContentChanne
            error('  ** Unknown packet length, discard packet\n **');
        else
            % Discard the packet if all HE-SIG-B content channels fail
            error('  ** HE-SIG-B CRC fail **');
        end
        % Update field indices as the number of HE-SIG-B symbols are
        % updated
        ind = wlanFieldIndices(cfgRx);
    end

    % Get complete HE-SIG-B field samples
    rxSIGB = rx(pktOffset+(ind.HESIGB(1):ind.HESIGB(2)),:);
    fprintf(' Decoding HE-SIG-B user field... \n');
```

```
    % Decode HE-SIG-B user field
    [failCRC,cfgUsers] = heSIGBUserFieldDecode(rxSIGB,preHEChanEst,noiseVar,cfgRx);

    % CRC on HE-SIG-B users
    if ~all(failCRC)
        fprintf('  HE-SIG-B (user field) CRC pass\n\n');
        numUsers = numel(cfgUsers);
    elseif all(failCRC)
        % Discard the packet if all users fail the CRC
        error('  ** HE-SIG-B CRC fail for all users **');
    else
        fprintf('  ** HE-SIG-B CRC fail for at least one user\n **');
        % Only process users with valid CRC
        numUsers = numel(cfgUsers);
    end

else % HE-SU, HE-EXT-SU
    cfgUsers = {cfgRx};
    numUsers = 1;
end
```

```
Decoding HE-SIG-B...
 Decoding HE-SIG-B common field...
  HE-SIG-B (common field) CRC pass
 Decoding HE-SIG-B user field...
  HE-SIG-B (user field) CRC pass
```

HE-Data Decoding

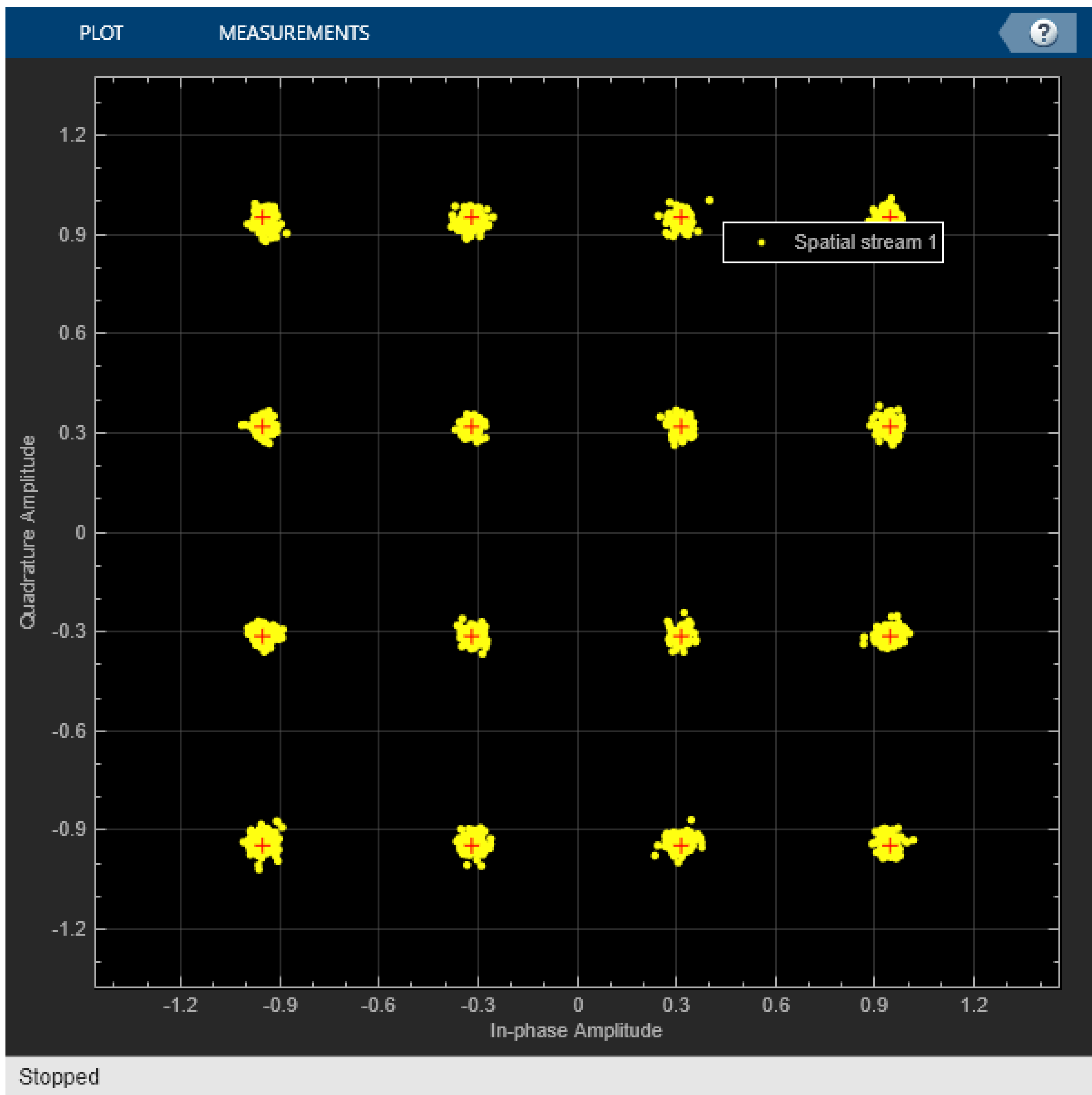
The updated wlanHERecoveryConfig object for each user can then be used to recover the PSDU bits for each user in the HE-Data field.

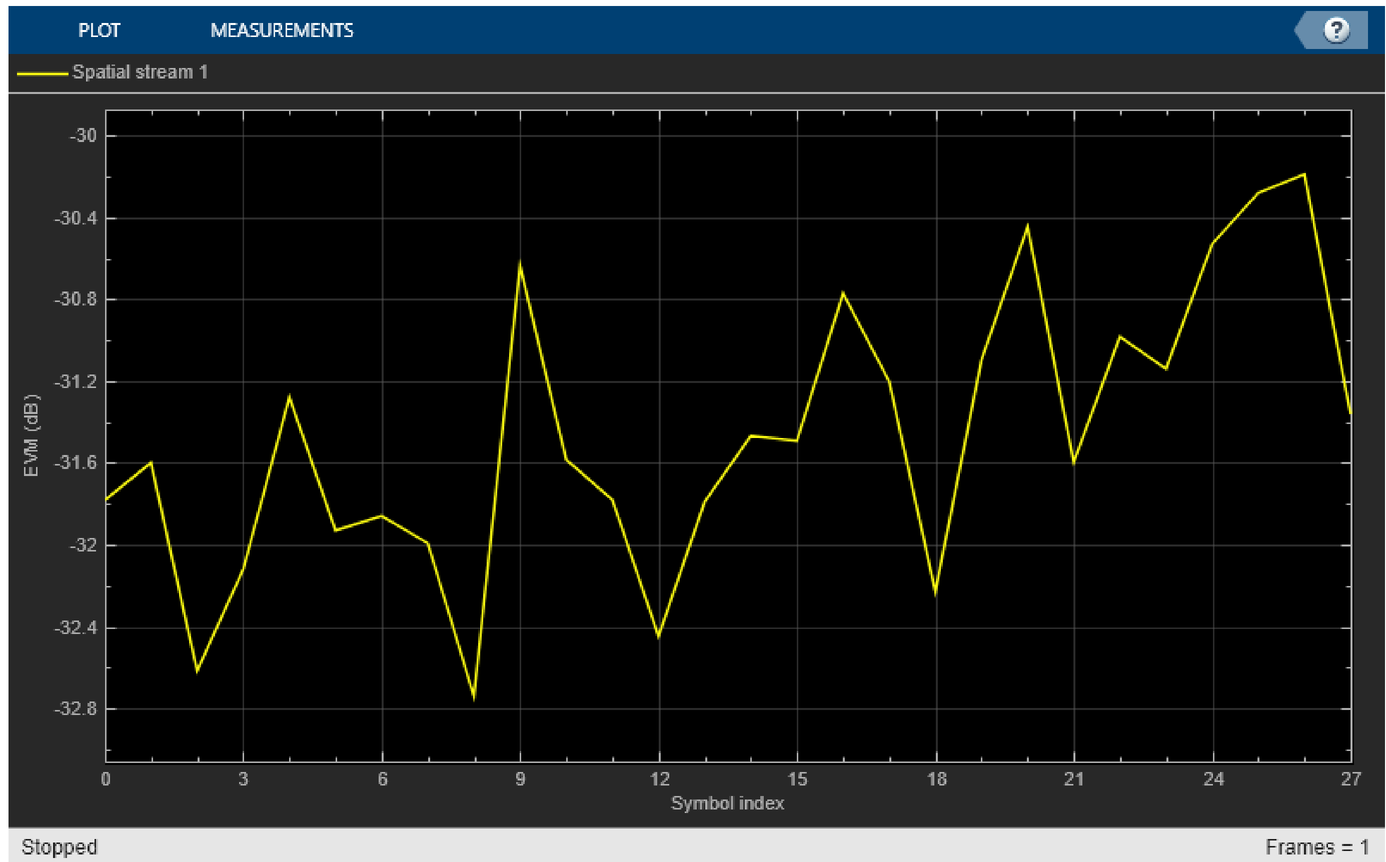
```
cfgDataRec = trackingRecoveryConfig;
cfgDataRec.PilotTracking = pilotTracking;

fprintf('Decoding HE-Data...\n');
for iu = 1:numUsers
    % Get recovery configuration object for each user
    user = cfgUsers{iu};
    if strcmp(pktFormat,'HE-MU')
        fprintf(' Decoding User:%d, STAID:%d, RUSize:%d\n',iu,user.STAID,user.RUSize);
    else
        fprintf(' Decoding RUSize:%d\n',user.RUSize);
    end

    heInfo = wlanHEOFDMInfo('HE-Data',chanBW,user.GuardInterval,[user.RUSize user.RUIndex]);

    % HE-LTF demodulation and channel estimation
    rxHELTF = rx(pktOffset+(ind.HELTF(1):ind.HELTF(2)),:);
    heltfDemod = wlanHEDemodulate(rxHELTF,'HE-LTF',chanBW,user.GuardInterval, ...
        user.HELTFType,[user.RUSize user.RUIndex]);
    [chanEst,pilotEst] = wlanHELTFChannelEstimate(heltfDemod,user);

    % Number of expected data OFDM symbols
    symLen = heInfo.FFTLength+heInfo.CPLength;
    numOFDMSym = double((ind.HEData(2)-ind.HEData(1)+1))/symLen;
```

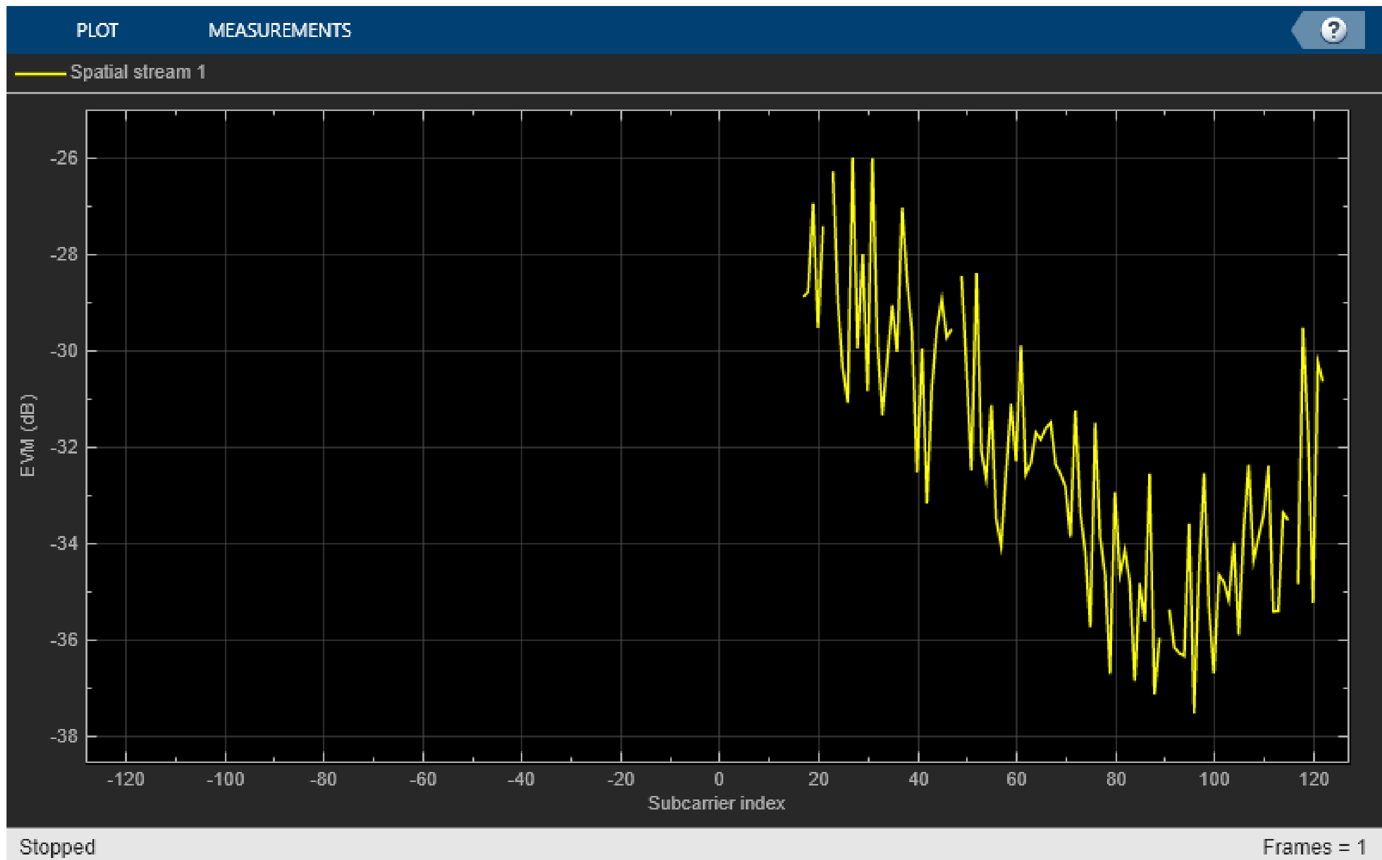
```
% HE-Data demodulation with pilot phase and timing tracking
% Account for extra samples when extracting data field from the packet
% for sample rate offset tracking. Extra samples may be required if the
% receiver clock is significantly faster than the transmitter.
maxSRO = 120; % Parts per million
Ne = ceil(numRxSamples*maxSRO*1e-6); % Number of extra samples
Ne = min(Ne,rxWaveLen-numRxSamples); % Limited to length of waveform
numRxSamplesProcess = numRxSamples+Ne;
rxData = rx(pktOffset+(ind.HEData(1):numRxSamplesProcess),:);
if user.RUSize==26
    % Force CPE only tracking for 26-tone RU as algorithm susceptible
    % to noise
    cfgDataRec.PilotTracking = 'CPE';
else
    cfgDataRec.PilotTracking = pilotTracking;
end
[demodSym,cpe,peg] = heTrackingOFDMDemodulate(rxData,chanEst,numOFDMSym,user,cfgDataRec);

% Estimate noise power in HE fields
demodPilotSym = demodSym(heInfo.PilotIndices,:,:);
nVarEst = heNoiseEstimate(demodPilotSym,pilotEst,user);

% Equalize
[eqSym,csi] = heEqualizeCombine(demodSym,chanEst,nVarEst,user);

% Discard pilot subcarriers
eqSymUser = eqSym(heInfo.DataIndices,:,:);
csiData = csi(heInfo.DataIndices,:);

% Demap and decode bits
rxPSDU = wlanHEDataBitRecover(eqSymUser,nVarEst,csiData,user,'LDPCDecodingMethod','norm-min-s

% Deaggregate the A-MPDU
[mpduList,~,status] = wlanAMPDUDeaggregate(rxPSDU,wlanHESUConfig);
if strcmp(status,'Success')
    fprintf('  A-MPDU deaggregation successful \n');
else
    fprintf('  A-MPDU deaggregation unsuccessful \n');
end

% Decode the list of MPDUs and check the FCS for each MPDU
for i = 1:numel(mpduList)
    [~,~,status] = wlanMPDUDecode(mpduList{i},wlanHESUConfig,'DataFormat','octets');
    if strcmp(status,'Success')
        fprintf('  FCS pass for MPDU:%d\n',i);
    else
        fprintf('  FCS fail for MPDU:%d\n',i);
    end
end

% Plot equalized constellation of the recovered HE data symbols for all
% spatial streams per user
hePlotEQConstellation(eqSymUser,user,ConstellationDiagram,iu,numUsers);

% Measure EVM of HE-Data symbols
release(EVM);
EVM.ReferenceConstellation = wlanReferenceSymbols(user);
rmsEVM = EVM(eqSymUser(:));
```

```
    fprintf('  HE-Data EVM:%2.2fdB\n\n',20*log10(rmsEVM/100));

    % Plot EVM per symbol of the recovered HE data symbols
    hePlotEVMPerSymbol(eqSymUser,user,EVMPerSymbol,iu,numUsers);

    % Plot EVM per subcarrier of the recovered HE data symbols
    hePlotEVMPerSubcarrier(eqSymUser,user,EVMPerSubcarrier,iu,numUsers);
end
```

```
Decoding HE-Data...
 Decoding User:1, STAID:1, RUSize:52
  A-MPDU deaggregation successful
  FCS pass for MPDU:1
  HE-Data EVM:-28.61dB

 Decoding User:2, STAID:2, RUSize:52
  A-MPDU deaggregation successful
  FCS pass for MPDU:1
  HE-Data EVM:-39.94dB

 Decoding User:3, STAID:3, RUSize:106
  A-MPDU deaggregation successful
  FCS pass for MPDU:1
  HE-Data EVM:-28.22dB

 Decoding User:4, STAID:4, RUSize:106
  A-MPDU deaggregation successful
  FCS pass for MPDU:1
  HE-Data EVM:-31.44dB
```


PLOT
MEASUREMENTS
Spatial stream 1
Quadrature Amplitude
In-phase Amplitude
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Stopped


PLOT
MEASUREMENTS
Spatial stream 1
-30
-30.4
-30.8
-31.2
-31.6
-32
-32.4
-32.8
EVM (dB)
0
3
6
9
12
15
18
21
24
27
Symbol index
Stopped
Frames = 1



Selected Bibliography

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

2 IEEE Std 802.11™-2020 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.

Recovery Procedure for an 802.11ac Packet

This example shows how to detect a packet and decode payload bits in a received IEEE® 802.11ac™ VHT waveform. The receiver recovers the packet format parameters from the preamble fields to decode the data.

Introduction

In a single-user 802.11ac packet the transmission parameters are signaled to the receiver using the L-SIG and VHT-SIG-A preamble fields [1]:

- The L-SIG field contains information to allow the receiver to determine the transmission time of a packet.
- The VHT-SIG-A field contains the transmission parameters including the modulation and coding scheme, number of space-time streams and channel coding.

In this example we detect and decode a packet within a generated waveform containing a valid MAC frame with frame check sequence (FCS). All transmission parameters apart from the channel bandwidth are assumed unknown and are therefore retrieved from the decoded L-SIG and VHT-SIG-A preamble fields in each packet. The retrieved transmission configuration is used to decode the VHT-SIG-B and VHT Data fields. Additionally the following analysis is performed:

- The waveform of the detected packet is displayed.
- The spectrum of the detected packet is displayed.
- The constellation of the equalized data symbols per spatial stream is displayed.
- The error vector magnitude (EVM) of each field is measured.

Waveform Transmission

In this example an 802.11ac VHT single-user waveform is generated locally but a captured waveform could be used. MATLAB® can be used to acquire I/Q data from a wide range of instruments using the Instrument Control Toolbox™ and software defined radio platforms.

The locally generated waveform is impaired by a 3x3 TGac fading channel, additive white Gaussian noise, and carrier frequency offset. To generate a waveform locally we configure a VHT packet format configuration object. Note that the VHT packet configuration object is used at the transmitter side only. The receiver will dynamically formulate another VHT configuration object when the packet is decoded. The helper function vhtSigRecGenerateWaveform generates the impaired waveform locally. The processing steps within the helper function are:

- A valid MAC frame is generated and encoded into a VHT waveform.
- The waveform is passed through a TGac fading channel model.
- Carrier frequency offset is added to the waveform.
- Additive white Gaussian noise is added to the waveform.

```
% VHT link parameters
cfgVHTTx = wlanVHTConfig( ...
    'ChannelBandwidth',    'CBW80', ...
    'NumTransmitAntennas', 3, ...
    'NumSpaceTimeStreams', 2, ...
    'SpatialMapping',      'Hadamard', ...
```

```
    'STBC',                true, ...
    'MCS',                 5, ...
    'GuardInterval',       'Long', ...
    'APEPLength',          1052);

% Propagation channel
numRx = 3;                  % Number of receive antennas
delayProfile = 'Model-C';   % TGac channel delay profile

% Impairments
noisePower = -30;  % Noise power to apply in dBW
cfo = 62e3;        % Carrier frequency offset (Hz)

% Generated waveform parameters
numTxPkt = 1;      % Number of transmitted packets
idleTime = 20e-6;  % Idle time before and after each packet

% Generate waveform
rx = vhtSigRecGenerateWaveform(cfgVHTTx, numRx, ...
    delayProfile, noisePower, cfo, numTxPkt, idleTime);
```

Packet Recovery

The signal to process is stored in the variable `rx`. The processing steps to recover a packet are:

- The packet is detected and synchronized.
- The format of the packet is detected.
- The L-SIG field is extracted and its information bits are recovered to determine the length of the packet in microseconds.
- The VHT-SIG-A field is extracted and its information bits are recovered.
- The packet format parameters are retrieved from the decoded L-SIG and VHT-SIG-A bits.
- The VHT-LTF field is extracted to perform MIMO channel estimation for decoding the VHT-SIG-B and VHT Data fields.
- The VHT-SIG-B field is extracted and its information bits recovered.
- The VHT-Data field is extracted and the PSDU and VHT-SIG-B CRC bits recovered using the retrieved packet parameters.

The start and end indices for some preamble fields depend on the channel bandwidth, but are independent of all other transmission parameters. These indices are calculated using a default transmission configuration object with the known bandwidth.

```
cfgVHTRx = wlanVHTConfig('ChannelBandwidth', cfgVHTTx.ChannelBandwidth);
idxLSTF = wlanFieldIndices(cfgVHTRx, 'L-STF');
idxLLTF = wlanFieldIndices(cfgVHTRx, 'L-LTF');
idxLSIG = wlanFieldIndices(cfgVHTRx, 'L-SIG');
idxSIGA = wlanFieldIndices(cfgVHTRx, 'VHT-SIG-A');
```

The following code configures objects and variables for processing.

```
chanBW = cfgVHTTx.ChannelBandwidth;
sr = wlanSampleRate(cfgVHTTx);

% Setup plots for example
[spectrumAnalyzer, timeScope, constellationDiagram] = vhtSigRecSetupPlots(sr);
```

```
% Minimum packet length is 10 OFDM symbols
lstfLen = double(idxLSTF(2)); % Number of samples in L-STF
minPktLen = lstfLen*5;

rxWaveLen = size(rx, 1);
```

Front-End Processing

The front-end processing consists of packet detection, coarse carrier frequency offset correction, timing synchronization and fine carrier frequency offset correction. A `while` loop is used to detect and synchronize a packet within the received waveform. The sample offset `searchOffset` is used to index elements within the array `rx` to detect a packet. The first packet within `rx` is detected and processed. If the synchronization fails for the detected packet, the sample index offset `searchOffset` is incremented to move beyond the processed packet in `rx`. This is repeated until a packet has been successfully detected and synchronized.

```
searchOffset = 0; % Offset from start of waveform in samples
while (searchOffset + minPktLen) <= rxWaveLen
    % Packet detection
    pktOffset = wlanPacketDetect(rx, chanBW, searchOffset);

    % Adjust packet offset
    pktOffset = searchOffset + pktOffset;
    if isempty(pktOffset) || (pktOffset + idxLSIG(2) > rxWaveLen)
        error('** No packet detected **');
    end

    % Coarse frequency offset estimation using L-STF
    LSTF = rx(pktOffset + (idxLSTF(1):idxLSTF(2)), :);
    coarseFreqOffset = wlanCoarseCFOEstimate(LSTF, chanBW);

    % Coarse frequency offset compensation
    rx = frequencyOffset(rx,sr,-coarseFreqOffset);

    % Symbol timing synchronization
    LLTFSearchBuffer = rx(pktOffset+(idxLSTF(1):idxLSIG(2)),:);
    pktOffset = pktOffset+wlanSymbolTimingEstimate(LLTFSearchBuffer,chanBW);
    if (pktOffset + minPktLen) > rxWaveLen
        fprintf('** Not enough samples to recover packet **\n\n');
        break;
    end

    % Timing synchronization complete: packet detected
    fprintf('Packet detected at index %d\n\n', pktOffset + 1);

    % Fine frequency offset estimation using L-LTF
    LLTF = rx(pktOffset + (idxLLTF(1):idxLLTF(2)), :);
    fineFreqOffset = wlanFineCFOEstimate(LLTF, chanBW);

    % Fine frequency offset compensation
    rx = frequencyOffset(rx, sr, -fineFreqOffset);

    % Display estimated carrier frequency offset
    cfoCorrection = coarseFreqOffset + fineFreqOffset; % Total CFO
    fprintf('Estimated CFO: %5.1f Hz\n\n', cfoCorrection);
```

```
    break; % Front-end processing complete, stop searching for a packet
end
```

```
Packet detected at index 1600

Estimated CFO: 61954.3 Hz
```

Format Detection

The format of the packet is detected using the three OFDM symbols immediately following the L-LTF. An estimate of the channel and noise power obtained from the L-LTF is required.

```
% Channel estimation using L-LTF
LLTF = rx(pktOffset + (idxLLTF(1):idxLLTF(2)), :);
demodLLTF = wlanLLTFDemodulate(LLTF, chanBW);
chanEstLLTF = wlanLLTFChannelEstimate(demodLLTF, chanBW);

% Estimate noise power in non-HT fields
noiseVarNonHT = wlanLLTFNoiseEstimate(demodLLTF);


% Detect the format of the packet
fmt = wlanFormatDetect(rx(pktOffset + (idxLSIG(1):idxSIGA(2)), :), ...
    chanEstLLTF, noiseVarNonHT, chanBW);
disp([fmt ' format detected']);
if ~strcmp(fmt,'VHT')
    error('** A format other than VHT has been detected **');
end
```

```
VHT format detected
```

L-SIG Decoding

In a VHT transmission the L-SIG field is used to determine the receive time, or RXTIME, of the packet. RXTIME is calculated using the field bits of the L-SIG payload [1 Eq. 22-105]. The number of samples which contain the packet within `rx` can then be calculated. The L-SIG payload is decoded using an estimate of the channel and noise power obtained from the L-LTF.

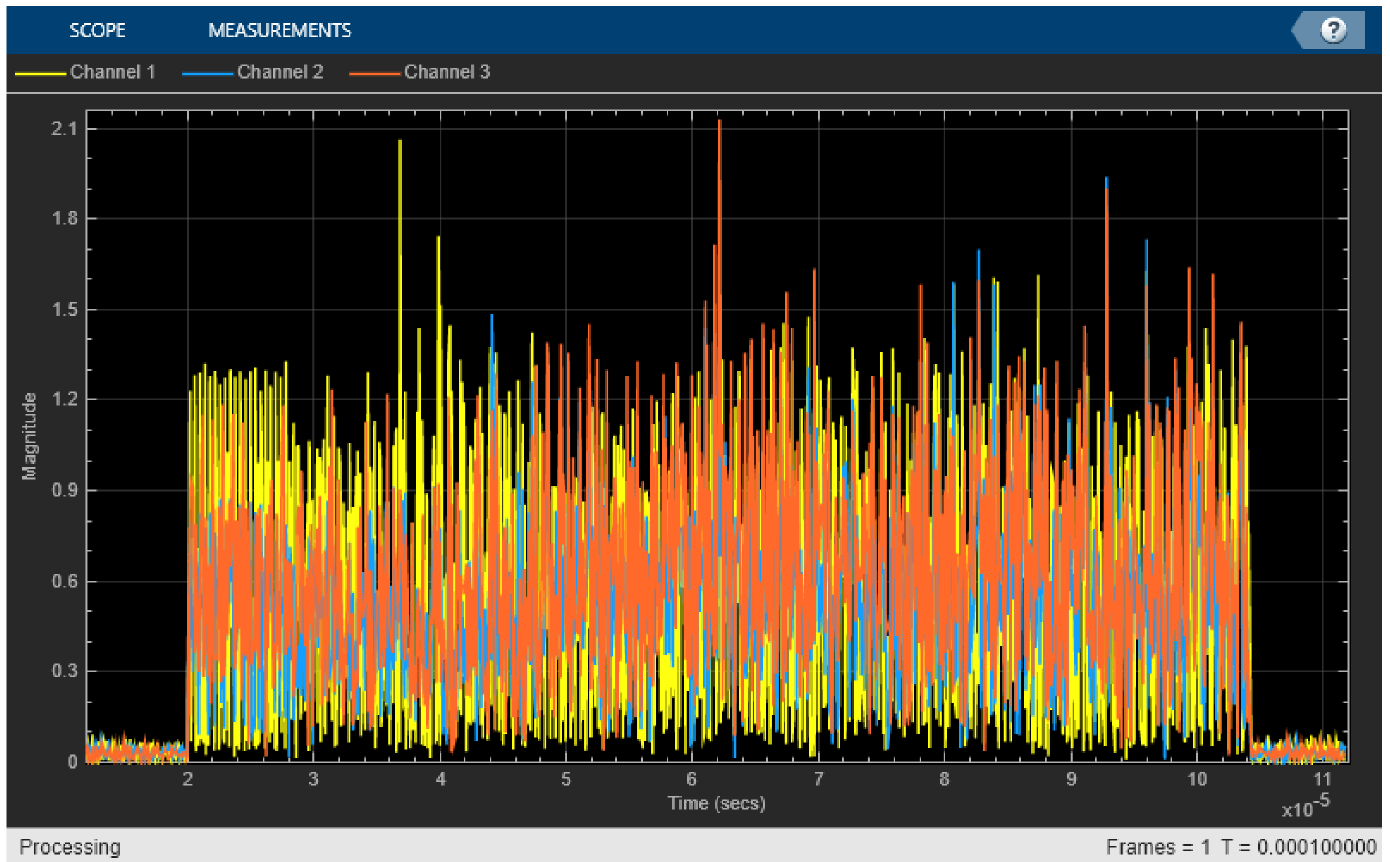
```
% Recover L-SIG field bits
disp('Decoding L-SIG... ');
[rxLSIGBits, failCheck, eqLSIGSym] = wlanLSIGRecover(rx(pktOffset + (idxLSIG(1):idxLSIG(2)), :),
    chanEstLLTF, noiseVarNonHT, chanBW);

if failCheck % Skip L-STF length of samples and continue searching
    disp('** L-SIG check fail **');
else
    disp('L-SIG check pass');
end

% Measure EVM of L-SIG symbol
EVM = comm.EVM;
EVM.ReferenceSignalSource = 'Estimated from reference constellation';
EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK');
rmsEVM = EVM(eqLSIGSym);
fprintf('L-SIG EVM: %2.2f%% RMS\n', rmsEVM);

% Calculate the receive time and corresponding number of samples in the
```

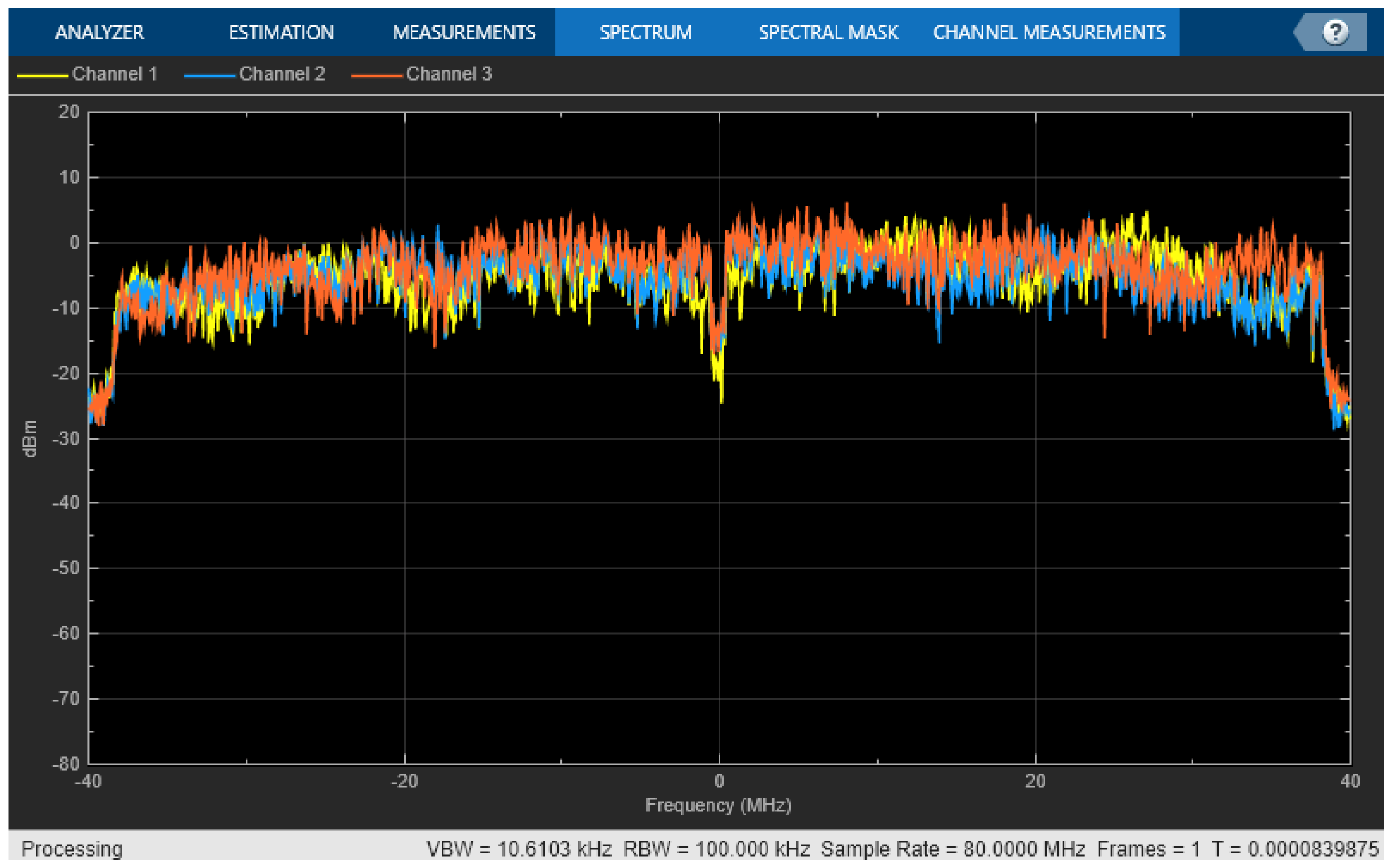
```
% packet
lengthBits = rxLSIGBits(6:17);
RXTime = ceil((bit2int(double(lengthBits),12,false) + 3)/3) * 4 + 20; % us
numRxSamples = RXTime * 1e-6 * sr; % Number of samples in receive time

fprintf('RXTIME: %dus\n', RXTime);
fprintf('Number of samples in packet: %d\n\n', numRxSamples);
```

```
Decoding L-SIG...
L-SIG check pass
L-SIG EVM: 1.83% RMS
RXTIME: 84us
Number of samples in packet: 6720
```

The waveform and spectrum of the detected packet within `rx` are displayed for the calculated RXTIME and corresponding number of samples.

```
sampleOffset = max((-lstfLen + pktOffset), 1); % First index to plot
sampleSpan = numRxSamples + 2*lstfLen;         % Number of samples to plot
% Plot as much of the packet (and extra samples) as we can
plotIdx = sampleOffset:min(sampleOffset + sampleSpan, rxWaveLen);

% Configure timeScope to display the packet
timeScope.TimeSpan = sampleSpan/sr;
timeScope.TimeDisplayOffset = sampleOffset/sr;
timeScope.YLimits = [0 max(abs(rx(:)))];
timeScope(abs(rx(plotIdx ,:)));

% Display the spectrum of the detected packet
spectrumAnalyzer(rx(pktOffset + (1:numRxSamples), :));
```


SCOPE
MEASUREMENTS
Channel 1
Channel 2
Channel 3
Magnitude
2.1
1.8
1.5
1.2
0.9
0.6
0.3
0
2
3
4
5
6
7
8
9
10
11
Time (secs)
x10-5
Processing
Frames = 1 T = 0.000100000



VHT-SIG-A Decoding

The VHT-SIG-A field contains the transmission configuration of the packet. The VHT-SIG-A bits are recovered using the channel and noise power estimates obtained from the L-LTF.

```
% Recover VHT-SIG-A field bits
disp('Decoding VHT-SIG-A... ');
[rxSIGABits, failCRC, eqSIGASym] = wlanVHTSIGARecover(rx(pktOffset + (idxSIGA(1):idxSIGA(2)), :)
    chanEstLLTF, noiseVarNonHT, chanBW);

if failCRC
    disp('** VHT-SIG-A CRC fail **');
else
    disp('VHT-SIG-A CRC pass');
end

% Measure EVM of VHT-SIG-A symbols for BPSK and QBPSK modulation schemes
release(EVM);
EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK');
rmsEVMSym1 = EVM(eqSIGASym(:,1));
release(EVM);
EVM.ReferenceConstellation = wlanReferenceSymbols('QBPSK');
rmsEVMSym2 = EVM(eqSIGASym(:,2));
fprintf('VHT-SIG-A EVM: %2.2f%% RMS\n', mean([rmsEVMSym1 rmsEVMSym2]));
```

```
Decoding VHT-SIG-A...
VHT-SIG-A CRC pass
VHT-SIG-A EVM: 2.06% RMS
```

The helper function helperVHTConfigRecover returns a VHT format configuration object, `cfgVHTRx`, based on recovered VHT-SIG-A and L-SIG bits. Properties that are not required to decode the waveform are set to default values for a `wlanVHTConfig` object and therefore may differ from the value in `cfgVHTTx`. Examples of such properties include `NumTransmitAntennas` and `SpatialMapping`.

```
% Create a VHT format configuration object by retrieving packet parameters
% from the decoded L-SIG and VHT-SIG-A bits
cfgVHTRx = helperVHTConfigRecover(rxLSIGBits, rxSIGABits);

% Display the transmission configuration obtained from VHT-SIG-A
vhtSigRecDisplaySIGAInfo(cfgVHTRx);
```

```
  Decoded VHT-SIG-A contents:
       ChannelBandwidth: 'CBW80'
    NumSpaceTimeStreams: 2
                   STBC: 1
                    MCS: 5
          ChannelCoding: {'BCC'}
          GuardInterval: 'Long'
                GroupID: 63
             PartialAID: 275
            Beamforming: 0
             PSDULength: 1167
```

The information provided by VHT-SIG-A allows the location of subsequent fields within the received waveform to be calculated.

```
% Obtain starting and ending indices for VHT-LTF and VHT-Data fields
% using retrieved packet parameters
idxVHTLTF  = wlanFieldIndices(cfgVHTRx, 'VHT-LTF');
idxVHTSIGB = wlanFieldIndices(cfgVHTRx, 'VHT-SIG-B');
idxVHTData = wlanFieldIndices(cfgVHTRx, 'VHT-Data');

% Warn if waveform does not contain whole packet
if (pktOffset + double(idxVHTData(2))) > rxWaveLen
    fprintf('** Not enough samples to recover entire packet **\n\n');
end
```

VHT-SIG-B Decoding

The primary use of VHT-SIG-B is for signaling user information in a multi-user packet. In a single-user packet the VHT-SIG-B carries the length of the packet which can also be calculated using the L-SIG and VHT-SIG-A (which is demonstrated in the sections above). Despite not being required to decode a single-user packet, the VHT-SIG-B is recovered below and the bits interpreted. The VHT-SIG-B symbols are demodulated using a MIMO channel estimate obtained from the VHT-LTF. Note the CRC for VHT-SIG-B is carried in the VHT Data field.

```
% Estimate MIMO channel using VHT-LTF and retrieved packet parameters
demodVHTLTF = wlanVHTLTFDemodulate(rx(pktOffset + (idxVHTLTF(1):idxVHTLTF(2)), :), cfgVHTRx);
[chanEstVHTLTF, chanEstSSPilots] = wlanVHTLTFChannelEstimate(demodVHTLTF, cfgVHTRx);

% The L-LTF OFDM demodulator normalizes the output by the number of
```

```
% subcarriers. The VHT-SIG-B OFDM demodulator normalizes the output by the
% number of subcarriers and space-time streams. Therefore, estimate the
% noise power in VHT-SIG-B by scaling the L-LTF noise estimate by the ratio
% of the number of subcarriers in both fields, and the number of space-time
% streams.

numSTSTotal = sum(cfgVHTRx.NumSpaceTimeStreams, 1);
scalingFactor = (height(demodVHTLTF)/size(demodLLTF, 1))*numSTSTotal;
noiseVarVHT = noiseVarNonHT*scalingFactor;

% VHT-SIG-B Recover
disp('Decoding VHT-SIG-B...');
[rxSIGBBits, eqSIGBSym] = wlanVHTSIGBRecover(rx(pktOffset + (idxVHTSIGB(1):idxVHTSIGB(2)),:), ..
    chanEstVHTLTF, noiseVarVHT, chanBW);

% Measure EVM of VHT-SIG-B symbol
release(EVM);
EVM.ReferenceConstellation = wlanReferenceSymbols('BPSK');
rmsEVM = EVM(eqSIGBSym);
fprintf('VHT-SIG-B EVM: %2.2f%% RMS\n', rmsEVM);

% Interpret VHT-SIG-B bits to recover the APEP length (rounded up to a
% multiple of four bytes) and generate reference CRC bits
[refSIGBCRC, sigbAPEPLength] = helperInterpretSIGB(rxSIGBBits, chanBW, true);
disp('Decoded VHT-SIG-B contents: ');
fprintf('  APEP Length (rounded up to 4 byte multiple): %d bytes\n\n', sigbAPEPLength);
```

```
Decoding VHT-SIG-B...
VHT-SIG-B EVM: 5.21% RMS
Decoded VHT-SIG-B contents:
  APEP Length (rounded up to 4 byte multiple): 1052 bytes
```

VHT Data Decoding

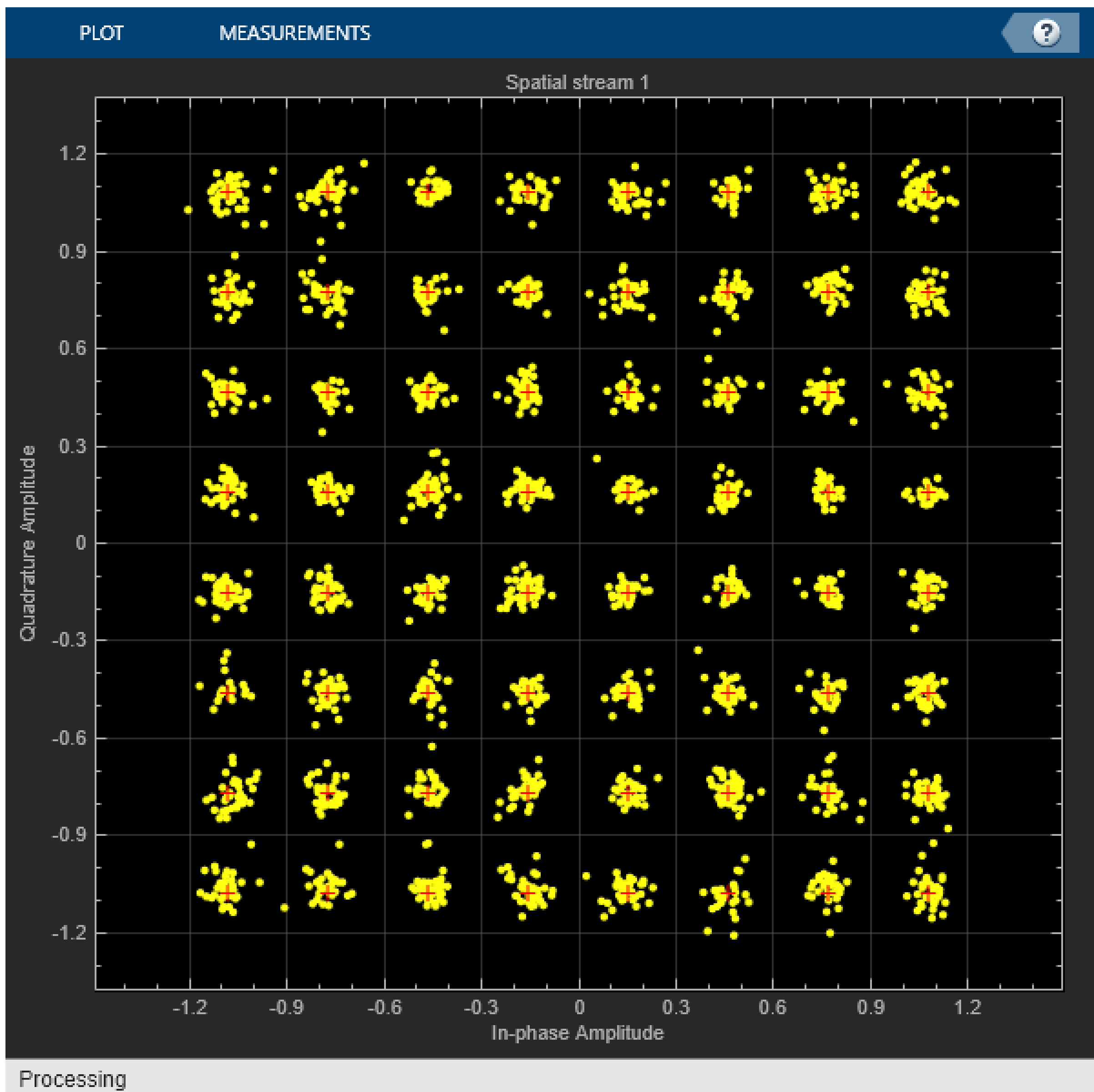
The reconstructed VHT configuration object can then be used to recover the VHT Data field. This includes the VHT-SIG-B CRC bits and PSDU.

The recovered VHT data symbols can then be analyzed as required. In this example the equalized constellation of the recovered VHT data symbols per spatial stream are displayed.

```
% Extract VHT Data samples from the waveform
vhtdata = rx(pktOffset + (idxVHTData(1):idxVHTData(2)), :);

% Estimate the noise power in VHT data field
noiseVarVHT = vhtNoiseEstimate(vhtdata, chanEstSSPilots, cfgVHTRx);

% Recover PSDU bits using retrieved packet parameters and channel
% estimates from VHT-LTF
disp('Decoding VHT Data field...');
[rxPSDU, rxSIGBCRC, eqDataSym] = wlanVHTDataRecover(vhtdata, chanEstVHTLTF, noiseVarVHT, cfgVHTRx
    'LDPCDecodingMethod', 'norm-min-sum');

% Plot equalized constellation for each spatial stream
refConst = wlanReferenceSymbols(cfgVHTRx);
[Nsd, Nsym, Nss] = size(eqDataSym);
eqDataSymPerSS = reshape(eqDataSym, Nsd*Nsym, Nss);
for iss = 1:Nss
```

```
    constellationDiagram{iss}.ReferenceConstellation = refConst;
    constellationDiagram{iss}(eqDataSymPerSS(:, iss));
end

% Measure EVM of VHT-Data symbols
release(EVM);
EVM.ReferenceConstellation = refConst;
rmsEVM = EVM(eqDataSym(:));
fprintf('VHT-Data EVM: %2.2f%% RMS\n', rmsEVM);
```

```
Decoding VHT Data field...
VHT-Data EVM: 4.68% RMS
```



The CRC bits for VHT-SIG-B recovered in VHT Data are then compared to the locally generated reference to determine whether the VHT-SIG-B and VHT data service bits have been recovered successfully.

```
% Test VHT-SIG-B CRC from service bits within VHT Data against
% reference calculated with VHT-SIG-B bits
if ~isequal(refSIGBCRC, rxSIGBCRC)
    disp('** VHT-SIG-B CRC fail **');
else
    disp('VHT-SIG-B CRC pass');
end
```

```
VHT-SIG-B CRC pass
```

The FCS in the MAC frames can be validated using `wlanMPDUDecode`. As a VHT format frame is recovered, the PSDU contains an A-MPDU. The MPDUs are extracted from the A-MPDU using `wlanAMPDUDeaggregate`.

```
mpduList = wlanAMPDUDeaggregate(rxPSDU, cfgVHTRx);
fprintf('Number of MPDUs present in the A-MPDU: %d\n', numel(mpduList));
```

```
Number of MPDUs present in the A-MPDU: 1
```

The `mpduList` contains the de-aggregated list of MPDUs. Each MPDU in the list is passed to `wlanMPDUDecode` which validates the FCS and decodes the MPDU.

```
for i = 1:numel(mpduList)
    [macCfg, payload, decodeStatus] = wlanMPDUDecode(mpduList{i}, cfgVHTRx, ...
                                                     'DataFormat', 'octets');
    if strcmp(decodeStatus, 'FCSFailed')
        fprintf('** FCS failed for MPDU-%d **\n', i);
    else
        fprintf('FCS passed for MPDU-%d\n', i);
    end
end
```

```
FCS passed for MPDU-1
```

Selected Bibliography

1. IEEE Std 802.11™-2020. 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.

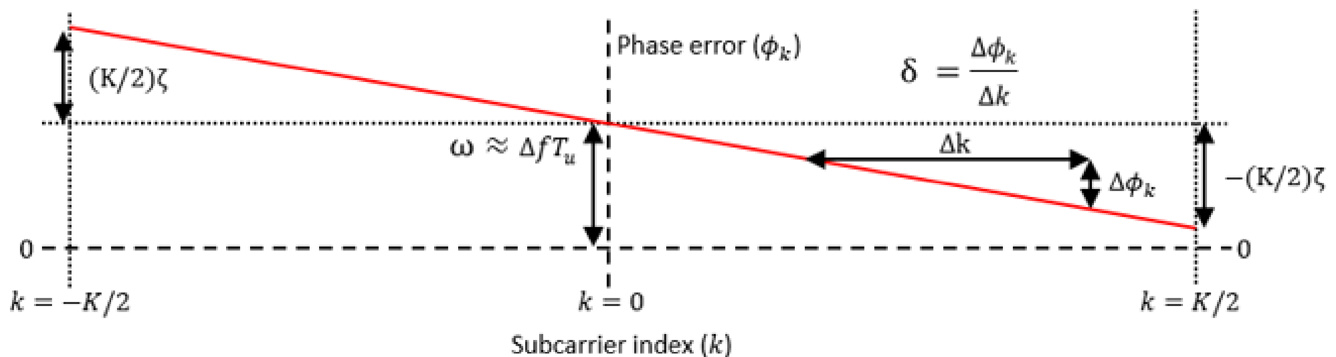
Joint Sampling Clock and Carrier Frequency Offset Tracking

This example demonstrates joint sampling clock and carrier frequency offset tracking in a WLAN receiver.

Introduction

A WLAN radio typically uses a single oscillator to derive clocks for sampling and modulation. The oscillators in the transmitter and receiver radios do not run at the exact same frequency. Due to this mismatch, carrier frequency offset (CFO) exists between the receiver and transmitter and the sampling instants at the receiver shift relative to the transmitter. This shift of sampling instants between receiver and transmitter is described as sampling clock offset (SCO). When the sampling clock at the receiver is running slower than the transmitter, this results in a larger sampling period and a positive sampling clock offset. The inclusion of pilot subcarriers in the IEEE® 802.11™ standard allows for tracking and correction of SCO and CFO impairments.

In OFDM systems SCO manifests itself as a subcarrier- and symbol-dependent phase rotation and inter-carrier interference (ICI) [1 on page 4-49]. When the SCO is large, and a packet is long, subcarriers far away from DC will experience a substantial impairment. CFO manifests itself as ICI and a symbol-dependent phase rotation common to all subcarriers. This figure illustrates the phase rotation on subcarriers from one OFDM symbol to the next due to these impairments. Φ_k is the phase error for subcarrier index k, K is the number of subcarriers, ζ is the SCO, Δf is the carrier frequency offset, T_u is the period of the symbol, δ is the phase error gradient (PEG), and ω is the common phase error (CPE). The PEG and CPE can be used to estimate SCO and residual CFO.



This example shows how to demonstrate an IEEE 802.11ac™ VHT waveform with fixed SCO and CFO impairments [2 on page 4-49], and shows the equalized constellation of the demodulated impaired waveform with and without joint timing and phase tracking to correct for SCO and CFO to demonstrate the effectiveness of tracking.

Generate a Baseband Waveform

Create a VHT configuration object to parameterize the transmission. Use a data payload with only 500 bytes and 16-point quadrature amplitude modulation (16-QAM) to produce a small number of OFDM symbols and make the impairment easy to visualize.

```
cfgVHT = wlanVHTConfig;
cfgVHT.ChannelBandwidth = 'CBW20';
cfgVHT.NumTransmitAntennas = 1;
```

```
cfgVHT.NumSpaceTimeStreams = 1;
cfgVHT.MCS = 4;          % 16-QAM and 3/4 coding rate
cfgVHT.APEPLength = 500; % Bytes
```

Create a random PSDU.

```
s = rng(10); % Seed the random number generator
psdu = randi([0 1],cfgVHT.PSDULength*8,1,'int8');
```

Generate a VHT packet.

```
tx = wlanWaveformGenerator(psdu,cfgVHT);
```

Model Impairments

Model -100 parts per million (PPM) sampling clock offset between the transmitter and receiver using `comm.SampleRateOffset`.

```
sco = -100; % Sampling clock offset in PPM
```

Use `comm.SampleRateOffset` by specifying the desired sample rate offset. Sample rate offset is the relative offset between the sample rate of transmitter and receiver. Sampling clock offset is the relative offset between the sampling period of transmitter and receiver.

```
sro = -sco/(1+sco/1e6); % Sample rate offset in PPM
```

Model sampling clock offset, appending zeros to the waveform to allow for filter delay.

```
samplingClockoffset = comm.SampleRateOffset(sro);
rx = samplingClockoffset([tx; zeros(100,cfgVHT.NumTransmitAntennas)]);
```

Add residual carrier frequency offset to the waveform. This example assumes the same oscillator is used for sampling and modulation, so that the CFO depends on the SCO and carrier frequency.

```
fc = 5.25e9;         % Carrier frequency, Hertz
cfo = (sco*1e-6)*fc; % Carrier frequency offset, Hertz
fs = wlanSampleRate(cfgVHT);          % Baseband sample rate
rx = frequencyOffset(rx,fs,cfo); % Add frequency offset
```

Add noise to the waveform with 30 dBW variance.

```
awgnChannel = comm.AWGNChannel('NoiseMethod','Variance','Variance',10^(-30/10));
rx = awgnChannel(rx);
```

Front-End Synchronization and Receiver Processing

To synchronize the packet, in preparation for recovering the data field, the example performs these processing steps.

1. Detect the packet
2. Perform coarse carrier frequency offset estimation and correction
3. Establish symbol timing synchronization
4. Perform fine carrier frequency offset estimation and correction
5. Demodulate the L-LTF and estimate the noise power
6. Demodulate the VHT-LTF and estimate the channel response

Generate field indices and perform packet detection.

```
ind = wlanFieldIndices(cfgVHT);
tOff = wlanPacketDetect(rx,cfgVHT.ChannelBandwidth);
```

Perform coarse frequency offset correction.

```
lstf = rx(tOff+(ind.LSTF(1):ind.LSTF(2)),:);
coarseCFOEst = wlanCoarseCFOEstimate(lstf,cfgVHT.ChannelBandwidth);
rx = frequencyOffset(rx,fs,-coarseCFOEst);
```

Perform symbol timing synchronization.

```
nonhtPreamble = rx(tOff+(ind.LSTF(1):ind.LSIG(2)),:);
symOff = wlanSymbolTimingEstimate(nonhtPreamble,cfgVHT.ChannelBandwidth);
tOff = tOff+symOff;
```

Perform fine frequency offset correction.

```
lltf = rx(tOff+(ind.LLTF(1):ind.LLTF(2)),:);
fineCFOEst = wlanFineCFOEstimate(lltf,cfgVHT.ChannelBandwidth);
rx = frequencyOffset(rx,fs,-fineCFOEst);
```

Perform channel estimation.

```
vhtltf = rx(tOff+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,cfgVHT);
[chanEst,chanEstSSPilots] = wlanVHTLTFChannelEstimate(vhtltfDemod,cfgVHT);
```

Recovery Without Sampling Clock Offset or Residual CFO Tracking

The coarse and fine frequency offset estimation and correction removes the majority of CFO, but residual CFO remains due to the presence of impairments in the waveform. The receiver must track and correct this offset.

```
disp('Front-end impairment correction:');
```

```
Front-end impairment correction:
```

```
frontEndCFOEst = coarseCFOEst+fineCFOEst;
disp(['  Estimated CFO: ' num2str(frontEndCFOEst,'%.1f') ' Hz']);
```

```
  Estimated CFO: -525209.0 Hz
```

```
residualCFO = cfo-frontEndCFOEst;
disp(['  Residual CFO after initial correction: ' num2str(residualCFO,'%.1f') ' Hz']);
```

```
  Residual CFO after initial correction: 209.0 Hz
```

Use the trackingVHTDataRecover function to recover the VHT data field with optional pilot tracking to correct for timing and phase errors due to SCO and CFO. Control pilot tracking using the trackingRecoveryConfig object.

First, recover the data field is without pilot tracking. Extract the data field from the waveform using the start and end sample indices of the field at the baseband rate. If the receiver sampling rate is higher than the transmitter rate, the receiver requires more samples than the transmitter produces. To allow for this, extract `Ne` additional samples from the waveform and pass to the recovery function. The maximum number of additional samples required depends on the expected SCO, baseband sampling rate, and maximum packet duration.

Create a recovery configuration with pilot tracking disabled.

```
cfgRec = trackingRecoveryConfig;
cfgRec.PilotTracking = 'None';
```

Extract data field with `Ne` additional samples to allow for negative SCO.

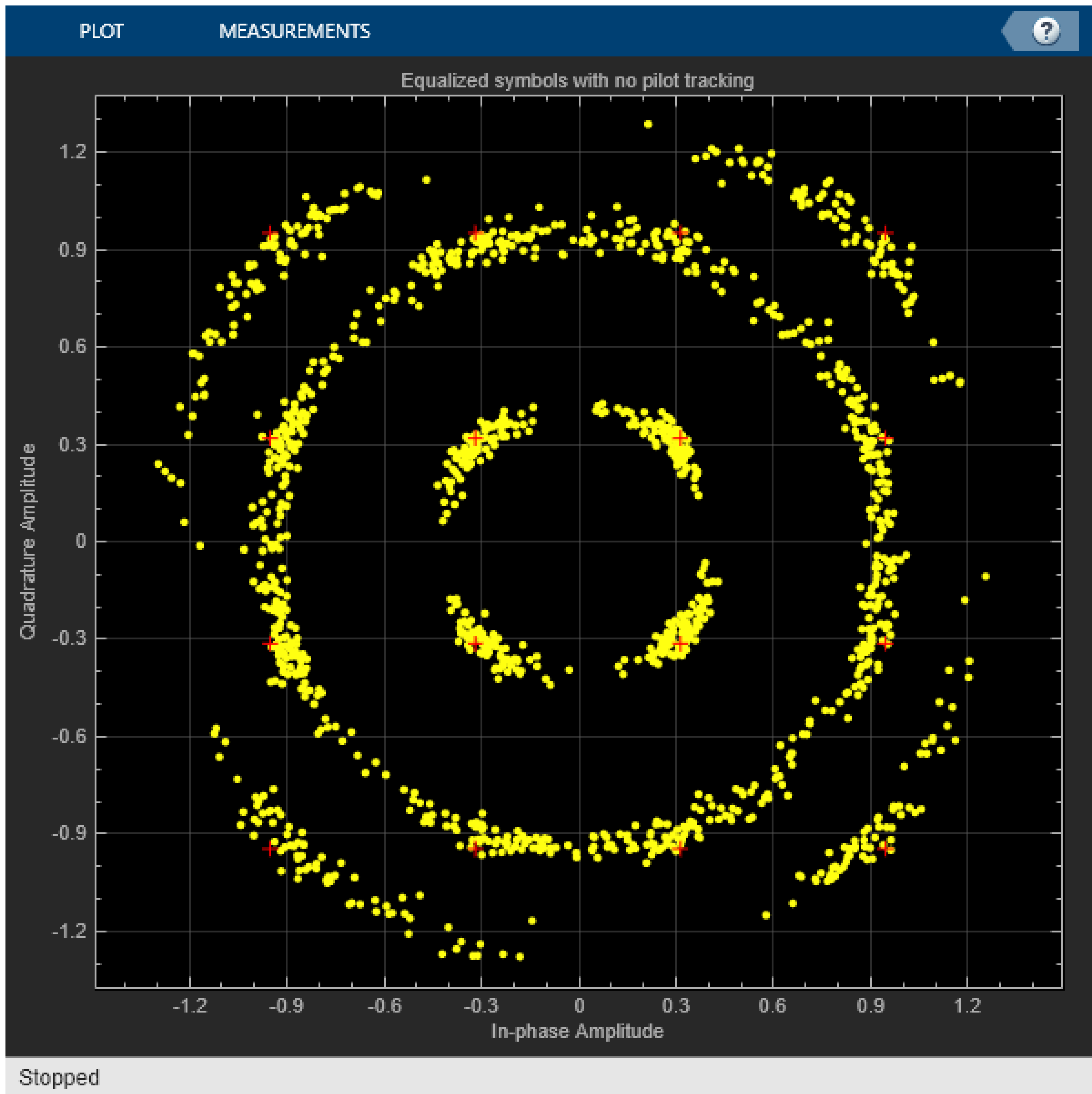
```
maxDuration = 5.484e-3; % Maximum packet duration in seconds
maxSCO = 120;           % PPM
Ne = ceil(fs*maxDuration*maxSCO*1e-6); % Number of extra samples
dataInd = tOff+(ind.VHTData(1):ind.VHTData(2)+Ne);
dataInd = dataInd(dataInd<=length(rx)); % Only use indices within waveform
data = rx(dataInd,:);
```

Perform demodulation and decoding.

```
[rxPSDUNoTrack,~,eqSymNoTrack] = trackingVHTDataRecover(data,chanEst,chanEstSSPilots,cfgVHT,cfgRe
```

Plot the equalized constellation. This plot shows a rotation of all constellation points caused by residual CFO, and a spreading of constellation points due to SCO. Despite the modest AWGN added to the waveform, the impairments cause bit errors within the decoded PSDU.

```
ConstNoTrack = comm.ConstellationDiagram;
ConstNoTrack.Title = 'Equalized symbols with no pilot tracking';
ConstNoTrack.ReferenceConstellation = wlanReferenceSymbols(cfgVHT);
ConstNoTrack(eqSymNoTrack(:));
release(ConstNoTrack)
```




```
[~,berNoTrack] = biterr(rxPSDUNoTrack,psdu);
disp('Bit error rate:');
```

```
Bit error rate:
```

```
disp(['  Without tracking: ' num2str(berNoTrack)]);
```

```
  Without tracking: 0.066964
```

Recovery With Sampling Clock Offset Tracking and Residual CFO Tracking

Now recover the data field with joint timing and phase pilot tracking to correct for SCO and residual CFO.

The tracking algorithm in this example estimates absolute values of δ and ω per OFDM symbol and applies a per subcarrier and symbol phase correction to the demodulated symbols to reverse the phase errors caused by SCO and CFO. The algorithm calculates the phase error between each received pilot subcarrier and the expected value per symbol averaged over `PilotTrackingWindow` OFDM symbols. From this value, the algorithm calculates least-square estimates of δ and ω per symbol, and uses these estimates to apply a phase correction to each symbol and subcarrier [3 on page 4-49,4 on page 4-49] .

Create a recovery configuration with pilot tracking enabled.

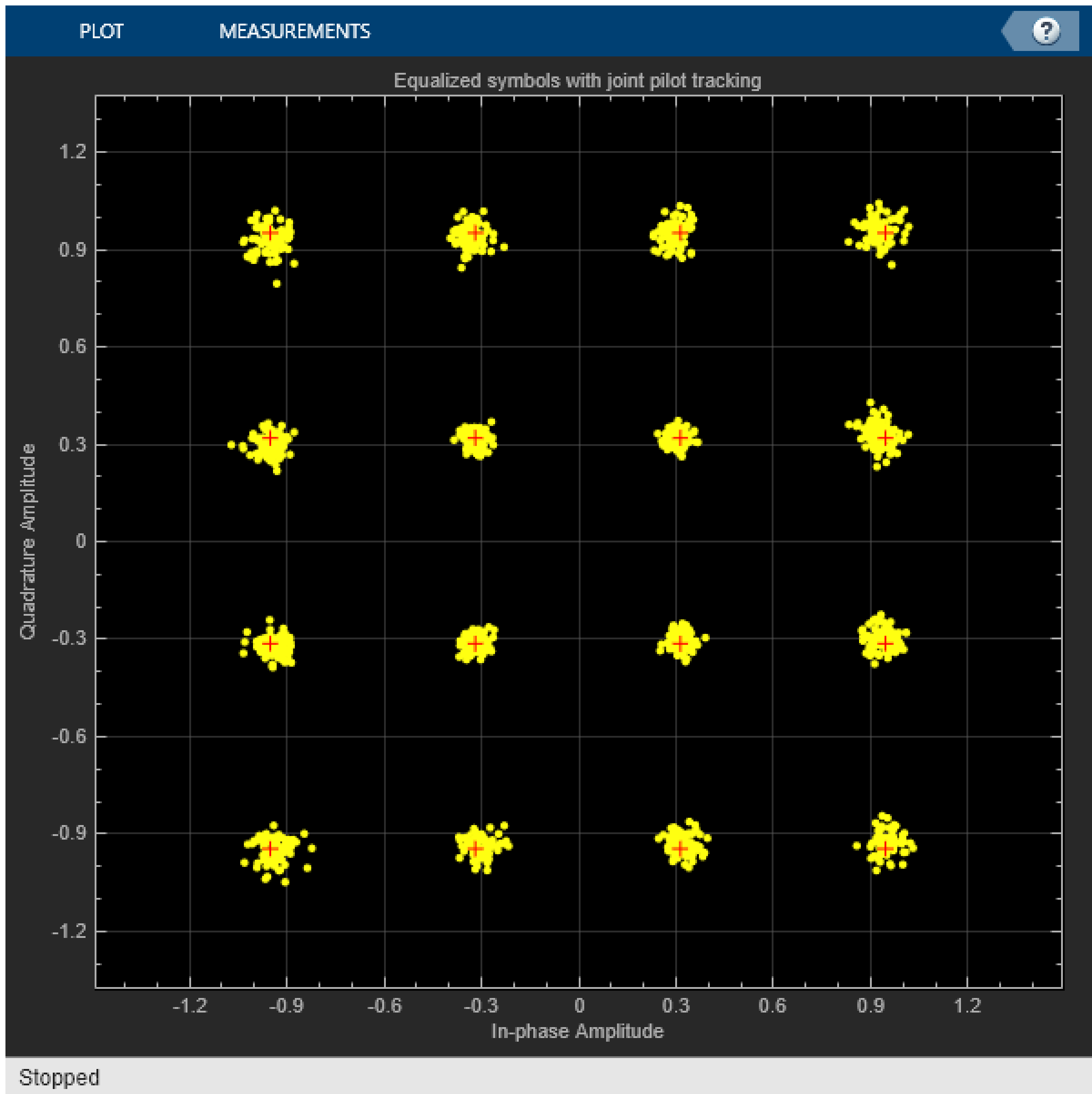
```
cfgRec = trackingRecoveryConfig;
cfgRec.PilotTracking = 'Joint'; % Joint timing and phase tracking
cfgRec.PilotTrackingWindow = 9; % Averaging window in OFDM symbols
```

Perform demodulation and decoding.

```
[rxPSDU,~,eqSymTrack,cpe,peg] = trackingVHTDataRecover(data,chanEst,chanEstSSPilots,cfgVHT,cfgRec
```

Plot the equalized constellation. This shows a clear 16-QAM constellation with no spreading or rotation. The decoded PSDU contains no bit errors.

```
ConstTrack = comm.ConstellationDiagram;
ConstTrack.Title = 'Equalized symbols with joint pilot tracking';
ConstTrack.ReferenceConstellation = wlanReferenceSymbols(cfgVHT);
ConstTrack(eqSymTrack(:));
release(ConstTrack)
```




```
[~,berTrack] = biterr(rxPSDU,psdu);
disp(['  With tracking: ' num2str(berTrack)]);
```

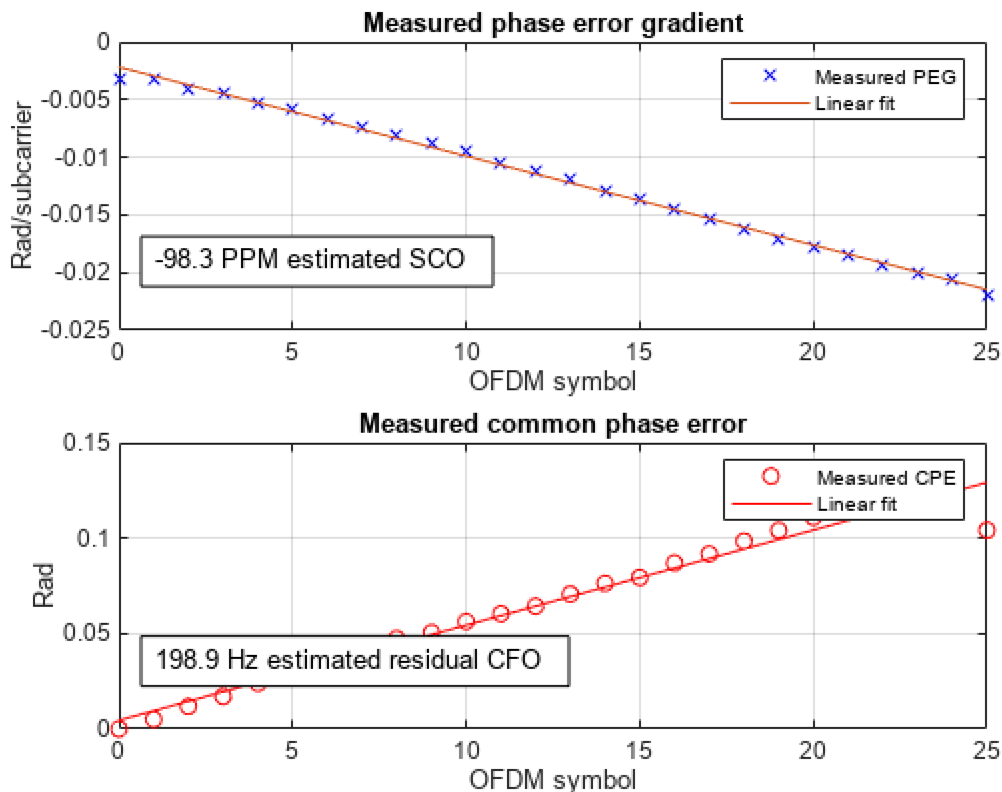
```
  With tracking: 0
```

The trackingVHTDataRecover function returns measurements from which the residual CFO, and SCO can be estimated:

- `cpe` - The common phase error (radians) per symbol
- `peg` - The phase error gradient (radians per subcarrier) per symbol

Estimate the SCO and residual CFO from these measurements using a linear least-square fit of the rate of change. The trackingPlotSCOCFOEstimates function performs these measurements and plot the results.

```
[residualCFOEst,scoEst] = trackingPlotSCOCFOEstimates(cpe,peg,cfgVHT);
```




```
fprintf('Tracked impairments:\n');
```

```
Tracked impairments:
```

```
fprintf('  Estimated residual CFO: %3.1f Hz (%.1f Hz error)\n', ...
    residualCFOEst,residualCFOEst-residualCFO);
```

```
  Estimated residual CFO: 198.9 Hz (-10.1 Hz error)
```

```
fprintf('  Estimated SCO: %3.1f PPM (%.1f PPM error)\n',scoEst,scoEst-sco);
```

```
  Estimated SCO: -98.3 PPM (1.7 PPM error)
```

```
cfoEst = frontEndCFOEst+residualCFOEst; % Initial + tracked CFO estimate
fprintf('Estimated CFO (initial + tracked): %.1f Hz (%.1f Hz error)\n',cfoEst,cfoEst-cfo);
```

```
Estimated CFO (initial + tracked): -525010.1 Hz (-10.1 Hz error)
```

```
rng(s); % Restore the state of the random number generator
```

Conclusion

This example shows how you can track and correct sampling clock and carrier frequency offsets when recovering the data field of a WLAN waveform.

This example uses data field recovery functions with joint pilot tracking for VHT, HT-MF and non-HT formats, and an object to configure the recovery algorithms.

- trackingVHTDataRecover.m
- trackingHTDataRecover.m
- trackingNonHTDataRecover.m
- trackingRecoveryConfig.m

To see an example of pilot tracking for HE format packets see the “Recovery Procedure for an 802.11ax Packet” on page 4-13 example.

References

1. Speth, M., S.A. Fechtel, G. Fock, and H. Meyr. “Optimum Receiver Design for Wireless Broad-Band Systems Using OFDM. I.” *IEEE Transactions on Communications* 47, no. 11 (November 1999): 1668–77. https://doi.org/10.1109/26.803501.
2. IEEE Std 802.11™-2020. 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. Chiueh, Tzi-Dar, Pei-Yun Tsai, Lai. I-Wei, and Tzi-Dar Chiueh. *Baseband Receiver Design for Wireless MIMO-OFDM Communications*. 2nd ed. Hoboken, N.J: J. Wiley & Sons, 2012.
4. Horlin, François, and André Bourdoux. *Digital Compensation for Analog Front-Ends: A New Approach to Wireless Transceiver Design*. Chichester, West Sussex ; Hoboken, NJ: J. Wiley & Sons, 2008.

Transmit and Recover L-SIG, VHT-SIG-A, VHT-SIG-B in Fading Channel

Transmit a VHT waveform through a noisy MIMO channel. Extract the L-SIG, VHT-SIG-A, and VHT-SIG-B fields and verify that they were correctly recovered.

Set the parameters used throughout the example.

```
cbw = 'CBW40';                    % Channel bandwidth
fs = 40e6;                        % Sample rate (Hz)
ntx = 2;                          % Number of transmit antennas
nsts = 2;                         % Number of space-time streams
nrx = 3;                          % Number of receive antennas
```

Create a VHT configuration object that supports a 2x2 MIMO transmission and has an APEP length of 2000.

```
vht = wlanVHTConfig('ChannelBandwidth',cbw,'APEPLength',2000, ...
    'NumTransmitAntennas',ntx,'NumSpaceTimeStreams',nsts, ...
    'SpatialMapping','Direct','STBC',false);
```

Generate a VHT waveform containing a random PSDU.

```
txPSDU = randi([0 1],vht.PSDULength*8,1);
txPPDU = wlanWaveformGenerator(txPSDU,vht);
```

Create a 2x2 TGac channel and an AWGN channel with an SNR=10 dB.

```
tgacChan = wlanTGacChannel('SampleRate',fs,'ChannelBandwidth',cbw, ...
    'NumTransmitAntennas',ntx,'NumReceiveAntennas',nrx, ...
    'LargeScaleFadingEffect','Pathloss and shadowing', ...
    'DelayProfile','Model-C');

chNoise = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)',...
    'SNR',10);
```

Pass the VHT waveforms through a 2x2 TGac channel and add the AWGN channel noise.

```
rxPPDU = chNoise(tgacChan(txPPDU));
```

Add additional white noise corresponding to a receiver with a 9 dB noise figure. The noise variance is equal to $k*T*B*F$, where k is Boltzmann's constant, T is the ambient temperature, B is the channel bandwidth (sample rate), and F is the receiver noise figure.

```
nVar = 10^((-228.6+10*log10(290) + 10*log10(fs) + 9 )/10);
rxNoise = comm.AWGNChannel('NoiseMethod','Variance','Variance',nVar);

rxPPDU = rxNoise(rxPPDU);
```

Find the start and stop indices for all component fields of the PPDU.

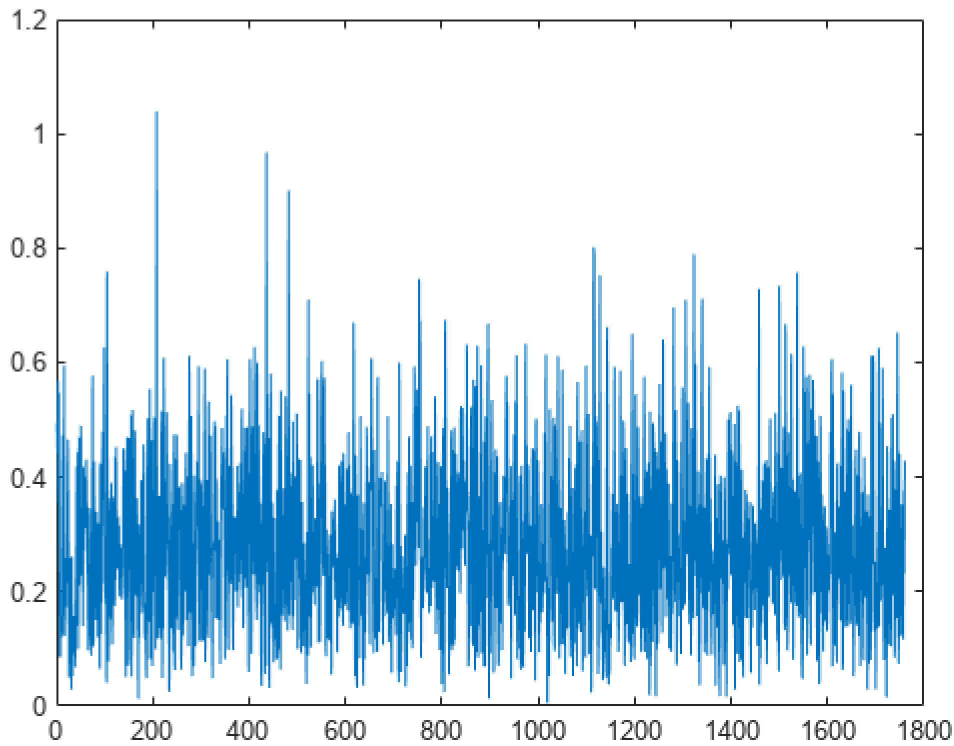
```
ind = wlanFieldIndices(vht)
```

```
ind = struct with fields:
       LSTF: [1 320]
       LLTF: [321 640]
```

```
       LSIG: [641 800]
    VHTSIGA: [801 1120]
     VHTSTF: [1121 1280]
     VHTLTF: [1281 1600]
    VHTSIGB: [1601 1760]
    VHTData: [1761 25600]
```

The preamble is contained in the first 1760 symbols. Plot the preamble.

```
plot(abs(rxPPDU(1:1760)))
```



Extract the L-LTF from the received PPDU using the start and stop indices determined by the `wlanFieldIndices` function. Demodulate the L-LTF and estimate the channel coefficients.

```
rxLLTF = rxPPDU(ind.LLTF(1):ind.LLTF(2),:);
demodLLTF = wlanLLTFDemodulate(rxLLTF,vht);
chEstLLTF = wlanLLTFChannelEstimate(demodLLTF,vht);
```

Extract the L-SIG field from the received PPDU and recover its information bits.

```
rxLSIG = rxPPDU(ind.LSIG(1):ind.LSIG(2),:);
infoLSIG = wlanLSIGRecover(rxLSIG,chEstLLTF,nVar,cbw);
```

Inspect the L-SIG rate information and confirm that the sequence `[1 1 0 1]` is received. This sequence corresponds to a 6 MHz data rate, which is used for all VHT transmissions.

```
rate = infoLSIG(1:4)'
```

```
rate = 1x4 int8 row vector

   0   1   1   1
```

Extract the VHT-SIG-A and confirm that the CRC check passed.

```
rxVHTSIGA = rxPPDU(ind.VHTSIGA(1):ind.VHTSIGA(2),:);
[infoVHTSIGA,failCRC] = wlanVHTSIGARecover(rxVHTSIGA, ...
    chEstLLTF,nVar,cbw);
failCRC
```

```
failCRC = logical
   1
```

Extract and demodulate the VHT-LTF. Use the demodulated signal to estimate the channel coefficients needed to recover the VHT-SIG-B field.

```
rxVHTLTF = rxPPDU(ind.VHTLTF(1):ind.VHTLTF(2),:);
demodVHTLTF = wlanVHTLTFDemodulate(rxVHTLTF,vht);
chEstVHTLTF = wlanVHTLTFChannelEstimate(demodVHTLTF,vht);
```

Extract and recover the VHT-SIG-B.

```
rxVHTSIGB = rxPPDU(ind.VHTSIGB(1):ind.VHTSIGB(2),:);
infoVHTSIGB = wlanVHTSIGBRecover(rxVHTSIGB,chEstVHTLTF,nVar,cbw);
```

Verify that the APEP length, contained in the first 19 bits of the VHT-SIG-B, corresponds to the specified length of 2000 bits.

```
pktLbits = infoVHTSIGB(1:19);
pktLen = bit2int(double(pktLbits),19,false)*4
```

```
pktLen = 1676920
```

End-to-End VHT Simulation with Frequency Correction

This example shows how to generate, transmit, recover and view a VHT MIMO waveform.

Steps in the example:

- Transmit a VHT waveform through a MIMO channel with AWGN
- Perform a two-stage process to estimate and correct for a frequency offset
- Estimate the channel response
- Recover the VHT data field
- Compare the transmitted and received PSDUs to determine if bit errors occurred

Set the parameters used throughout the example.

```
cbw = 'CBW160';                   % Channel bandwidth
fs = 160e6;                       % Sample rate (Hz)
ntx = 2;                          % Number of transmit antennas
nsts = 2;                         % Number of space-time streams
nrx = 2;                          % Number of receive antennas
```

Create a VHT configuration object that supports a 2x2 MIMO transmission and has an APEP length of 2000.

```
vht = wlanVHTConfig('ChannelBandwidth',cbw,'APEPLength',2000, ...
    'NumTransmitAntennas',ntx,'NumSpaceTimeStreams',nsts, ...
    'SpatialMapping','Direct','STBC',false);
```

Generate a VHT waveform containing a random PSDU.

```
txPSDU = randi([0 1],vht.PSDULength*8,1);
txPPDU = wlanWaveformGenerator(txPSDU,vht);
```

Create a 2x2 TGac channel and an AWGN channel.

```
tgacChan = wlanTGacChannel('SampleRate',fs,'ChannelBandwidth',cbw, ...
    'NumTransmitAntennas',ntx,'NumReceiveAntennas',nrx, ...
    'LargeScaleFadingEffect','Pathloss and shadowing', ...
    'DelayProfile','Model-C');
awgnChan = comm.AWGNChannel('NoiseMethod','Variance', ...
    'VarianceSource','Input port');
```

Create a phase/frequency offset object.

```
pfOffset = comm.PhaseFrequencyOffset('SampleRate',fs,'FrequencyOffsetSource','Input port');
```

Calculate the noise variance for a receiver with a 9 dB noise figure. Pass the transmitted waveform through the noisy TGac channel.

```
nVar = 10^((-228.6 + 10*log10(290) + 10*log10(fs) + 9)/10);
rxPPDU = awgnChan(tgacChan(txPPDU), nVar);
```

Introduce a frequency offset of 500 Hz.

```
rxPPDUcfo = pfOffset(rxPPDU,500);
```

Find the start and stop indices for all component fields of the PPDU.

```
ind = wlanFieldIndices(vht);
```

Extract the L-STF. Estimate and correct for the carrier frequency offset.

```
rxLSTF = rxPPDUcfo(ind.LSTF(1):ind.LSTF(2),:);

foffset1 = wlanCoarseCFOEstimate(rxLSTF,cbw);
rxPPDUcorr = pfOffset(rxPPDUcfo,-foffset1);
```

Extract the L-LTF from the corrected signal. Estimate and correct for the residual frequency offset.

```
rxLLTF = rxPPDUcorr(ind.LLTF(1):ind.LLTF(2),:);

foffset2 = wlanFineCFOEstimate(rxLLTF,cbw);
rxPPDU2 = pfOffset(rxPPDUcorr,-foffset2);
```

Extract and demodulate the VHT-LTF. Estimate the channel coefficients.

```
rxVHTLTF = rxPPDU2(ind.VHTLTF(1):ind.VHTLTF(2),:);
dLTF = wlanVHTLTFDemodulate(rxVHTLTF,vht);
chEst = wlanVHTLTFChannelEstimate(dLTF,vht);
```

Extract the VHT data field from the received and frequency-corrected PPDU. Recover the data field.

```
rxVHTData = rxPPDU2(ind.VHTData(1):ind.VHTData(2),:);
rxPSDU = wlanVHTDataRecover(rxVHTData,chEst,nVar,vht);
```

Calculate the number of bit errors in the received packet.

```
numErr = biterr(txPSDU,rxPSDU)
```

```
numErr = 0
```

5

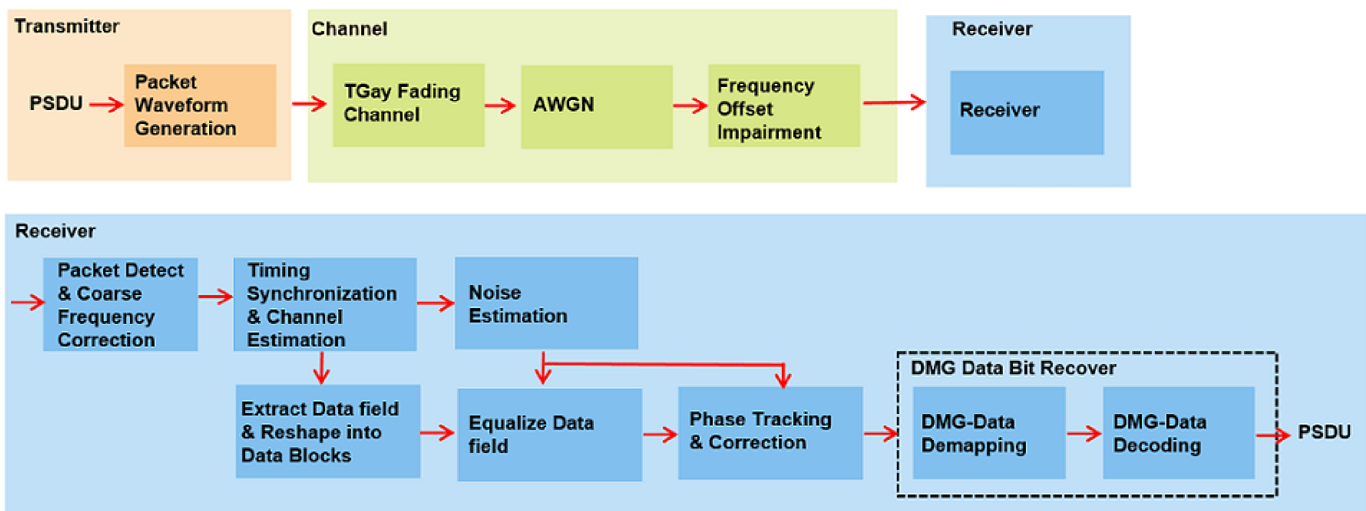
Propagation Channel Models

802.11ad Packet Error Rate Single Carrier PHY Simulation with TGay Channel

This example shows how to measure the packet error rate of an IEEE® 802.11ad™ DMG single carrier (SC) PHY link using an end-to-end simulation.

Introduction

In this example an end-to-end simulation is used to determine the packet error rate for an 802.11ad SC [1] link with a TGay millimeter-wave channel [2] at a selection of SNR points. At each SNR point multiple packets are transmitted through the fading channel, synchronized, demodulated and the PSDUs recovered. Carrier frequency offset and a time delay are also modeled. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate. The processing for each packet is summarized in the following diagram.



This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each SNR point to speed up a simulation. `parfor`, as part of the Parallel Computing Toolbox™, executes processing for each SNR in parallel to reduce the total simulation time.

Waveform Configuration

An 802.11ad DMG SC PHY transmission is simulated in this example. The DMG format configuration object contains the format specific configuration of the transmission. The object is created using the `wlanDMGConfig` function. The properties of the object contain the configuration of the transmitted packet. In this example the object is configured to generate a single carrier waveform of MCS "9". The MCS determines the PHY type used and in this example it must be a string within the range 1-12, or one of {9.1 12.1 12.2 12.3 12.4 12.5 12.6} to simulate the SC PHY.

```
% Create a format configuration object
cfgDMG = wlanDMGConfig;
mcs = "9"; % MCS specified as a string scalar or string vector
cfgDMG.PSDULength = 4096; % PSDULength in bytes
```

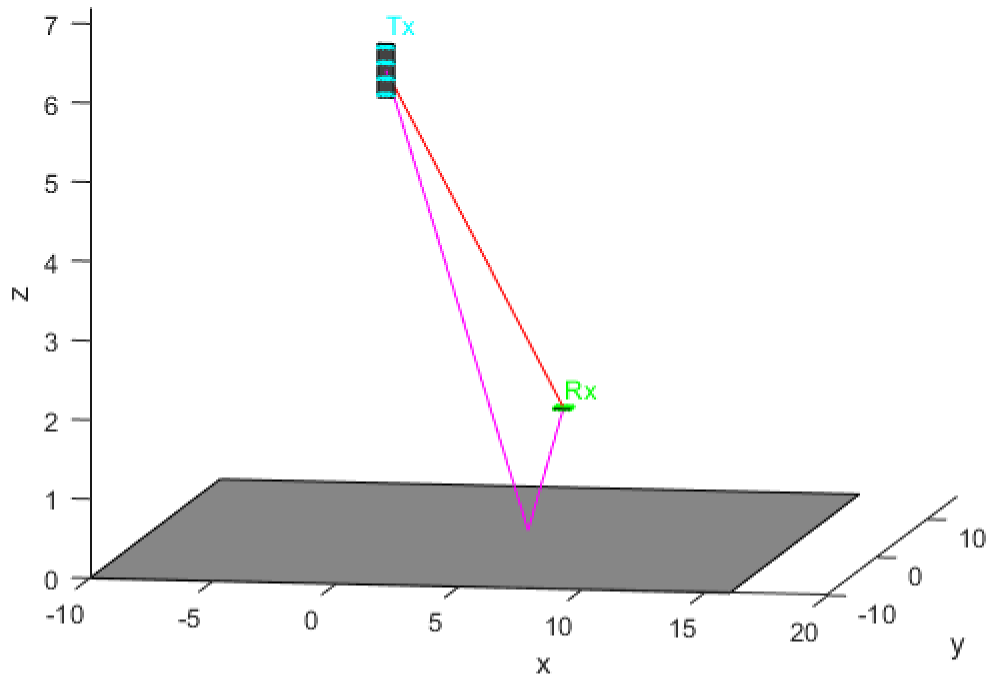
Fading Channel Configuration

In this example we simulate a TGay channel model for the open area hotspot scenario, using the `wlanTGayChannel` object. Both the transmit and receive arrays are a 4x4 uniform rectangular array (URA). Ray-tracing is performed from the transmit to the receive array to derive two deterministic rays: one LOS ray and another NLOS ray with one-order reflection from the ground. Random rays and intra-cluster rays are subsequently generated according to the quasi-deterministic (Q-D) modeling approach and parameters in [2]. Beamforming is performed to set the transmit and receive antenna steering direction along the ray with the maximum power. The fading channel impulse responses are normalized so no power gain or loss is introduced by the channel. Both the channel input and output signals are unpolarized.

```
% Get sampling rate and specify carrier frequency
fs = wlanSampleRate(cfgDMG);
fc = 60e9;

% Create a TGay channel object
tgayChan = wlanTGayChannel;
tgayChan.SampleRate                = fs;
tgayChan.CarrierFrequency          = fc;
tgayChan.Environment               = 'Open area hotspot';
tgayChan.TransmitArray.Size        = [4 4];
tgayChan.TransmitArrayPosition     = [0; 0; 6];     % Meters
tgayChan.TransmitArrayOrientation  = [0; 270; 0];   % Degrees
tgayChan.ReceiveArray.Size         = [4 4];
tgayChan.ReceiveArrayPosition      = [6; 6; 1.5];   % Meters
tgayChan.ReceiveArrayOrientation   = [90; 0; 0];    % Degrees
tgayChan.BeamformingMethod         = 'Maximum power ray';
tgayChan.NormalizeImpulseResponses = true;
tgayChan.NormalizeChannelOutputs   = false;
```

Given the channel object configuration, we display a 3D map to show the environment and antenna array settings. The two deterministic rays from ray-tracing are also shown in the figure.

```
showEnvironment(tgayChan);
```



Channel Impairment

The maximum tolerance for the transmitter center frequency must be within [-20, +20] ppm [1]. In this example, a clock accuracy of 20ppm is considered to derive the CFO. The transmitted signal is delayed by 500 samples and also appended by 100 zero samples at the end to account for delays from TGay channel filtering.

```
ppm = 20;                 % Clock accuracy to drive the CFO (ppm)
freqOffset = ppm*1e-6*fc; % Carrier frequency offset (Hz)
delay = 500;              % Sample to delay the waveform
zeroPadding = 100;        % Add trailing zeros to allow for channel delay
```

Simulation Parameters

For each SNR point (dB) in the cell `snrRanges` a number of packets are generated, passed through a channel and demodulated to determine the packet error rate. The SNR points to test are selected from `snrRanges` based on the MCS simulated.

```
snrRanges = {-2.5:0.5:0, ...       % MCS 1
             -0.5:0.75:3.5, ...    % MCS 2
              1.0:0.75:5, ...      % MCS 3
              1.5:0.75:5.5, ...    % MCS 4
              2.0:0.75:6.0, ...    % MCS 5
              2.5:0.75:6.5, ...    % MCS 6
              3:0.75:7, ...        % MCS 7
              4.5:0.75:8.5,...     % MCS 8
              6:0.5:8.5, ...       % MCS 9
```

```
    7.0:0.5:9.5, ...    % MCS 9.1
    8.5:1:13.5, ...     % MCS 10
    9.5:1.0:14.5, ...   % MCS 11
    10.5:1.25:17.0, ... % MCS 12
    11.5:1.5:19.0, ...  % MCS 12.1
    13.0:1.0:18, ...    % MCS 12.2
    14.0:1.0:19, ...    % MCS 12.3
    17.0:1.5:25, ...    % MCS 12.4
    19.0:2:29, ...      % MCS 12.5
    19.5:2:30};         % MCS 12.6
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For meaningful results we recommend increasing the numbers.

```
maxNumErrors = 10;   % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated.

For each packet the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through a TGay channel model. Different channel realizations are modeled for different packets.
3. AWGN is added to the received waveform.
4. The frequency offset impairment is added to each packet.
5. The packet is detected.
6. Carrier frequency offset is estimated and corrected.
7. Fine timing synchronization is established. The CE field samples are provided for fine timing to allow for packet detection at the start of the STF.
8. The STF and CE fields are extracted from the synchronized received waveform. The noise and channel estimation is performed on the recovered fields respectively.
9. The data field, excluding the first guard interval is extracted and reshaped into blocks. The received symbols in the data field are equalized.
10. The received symbols are tracked and corrected for phase errors caused by any residual carrier frequency offset.
11. The data field is decoded to recover the PSDU bits.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the `for` statement and uncomment the `parfor` statement below.

```
numSNR = numel(snrRanges{1}); % Number of SNR points
if ~isstring(mcs)
```

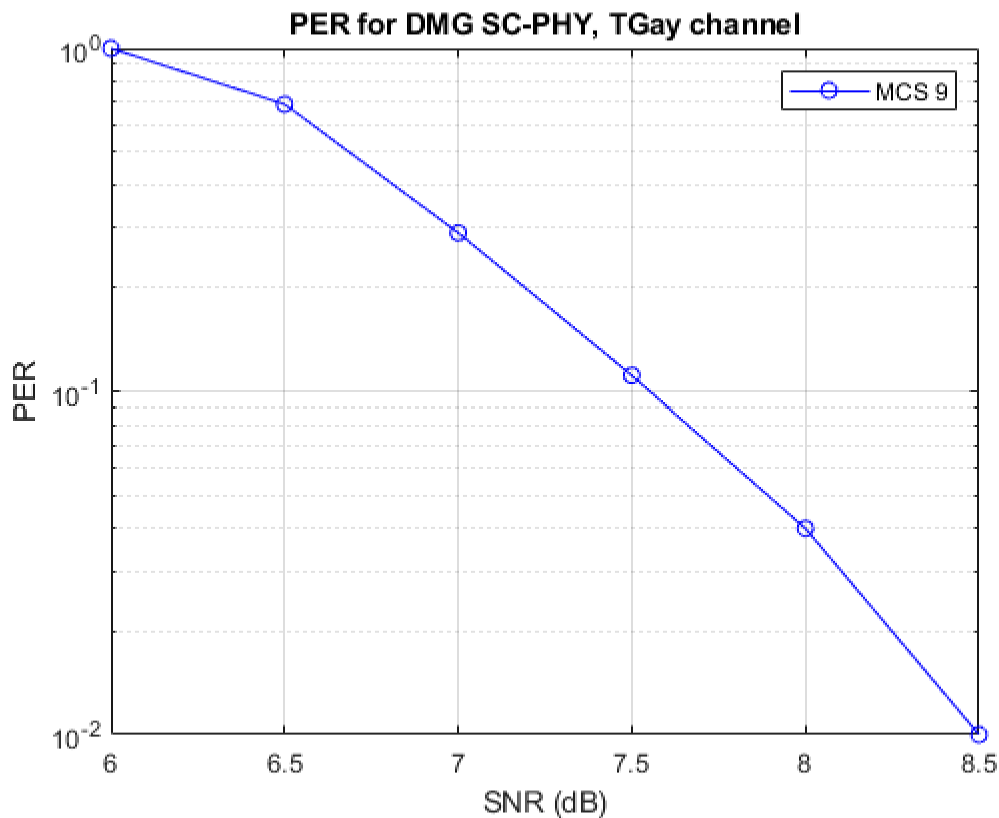
```
    error('MCS must be specified as a string scalar or string vector');
end
numMCS = numel(mcs);          % Number of MCS
packetErrorRate = zeros(numMCS,numSNR);
Ngi = 64; % Fixed GI length defined in the standard (20.6.3.2.5)
validMCS = string(sort([1:12 9.1 12.1:0.1:12.6]));

for imcs = 1:numMCS
    cfgDMG.MCS = mcs(imcs);
    if ~strcmp(phyType(cfgDMG),'SC')
        error('This example only supports DMG SC PHY simulation');
    end
    ind = wlanFieldIndices(cfgDMG);
    snr = snrRanges{mcs(imcs)==validMCS}; % SNR points to simulate from MCS

    % parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
    for isnr = 1:numSNR     % 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',1);
        stream.Substream = isnr;
        RandStream.setGlobalStream(stream);

        % Set simulation parameters
        numPacketErrors = 0;
        numPkt = 1; % Index of the transmitted packet

        while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
            % Generate a packet waveform
            psdu = randi([0 1],cfgDMG.PSDULength*8,1);
            txWaveform = wlanWaveformGenerator(psdu,cfgDMG);

            % Add delay and trailing zeros
            tx = [zeros(delay,1); txWaveform; zeros(zeroPadding,1)];

            % Transmit through a TGay channel. Reset the channel for a
            % different realization per packet.
            reset(tgayChan);
            chanOut = tgayChan(tx);

            % Add noise
            rx = awgn(chanOut,snr(isnr));

            % Add CFO
            rx = frequencyOffset(rx,fs,freqOffset);

            % Packet detection
            threshold = 0.03; % Good for low SNRs
            pktStartOffset = dmgPacketDetect(rx,0,threshold);
            if isempty(pktStartOffset) % If empty no STF detected; packet error
                numPacketErrors = numPacketErrors+1;
                numPkt = numPkt+1;
                continue; % Go to next loop iteration
            end

            % Frequency offset estimation and correction
            stf = rx(pktStartOffset+(ind.DMGSTF(1):ind.DMGSTF(2)));
            fOffsetEst = dmgCFOEstimate(stf);
```

```
        rx = frequencyOffset(rx,fs,-fOffsetEst);

        % Symbol timing and channel estimate
        preamblefield = rx(pktStartOffset+1:pktStartOffset+ind.DMGHeader(2),:);
        [symbolTimingOffset,chanEst] = dmgTimingAndChannelEstimate(preamblefield);
        startOffset = pktStartOffset+symbolTimingOffset;

        % If not enough samples to decode detected data field start,
        % then assume synchronization error and packet error
        if (startOffset+ind.DMGData(2))>size(rx,1)
            numPacketErrors = numPacketErrors+1;
            numPkt = numPkt+1;
            continue; % Go to next loop iteration
        end

        % Noise estimation using the STF as repeating sequence
        stf = rx(pktStartOffset+(ind.DMGSTF(1):ind.DMGSTF(2)));
        nVarEst = dmgSTFNoiseEstimate(stf);

        % Extract data field (ignore first GI)
        rxData = rx(startOffset+((ind.DMGData(1)+Ngi):ind.DMGData(2)));

        % Linear frequency domain equalization
        rxEqDataBlks = dmgSingleCarrierFDE(rxData,chanEst,nVarEst);

        % Unique word phase tracking
        rxEqDataBlks = dmgUniqueWordPhaseTracking(rxEqDataBlks);

        % Discard GI from all blocks
        rxDataSym = rxEqDataBlks(1:end-Ngi,:);

        % Recover the transmitted PSDU from DMG Data field
        dataDecode = wlanDMGDataBitRecover(rxDataSym,nVarEst,cfgDMG);

        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(psdu,dataDecode));
        numPacketErrors = numPacketErrors+packetError;
        numPkt = numPkt+1;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(imcs,isnr) = numPacketErrors/(numPkt-1);
    disp(join(["     MCS:" cfgDMG.MCS ", SNR " ...
        num2str(snr(isnr)) " completed after " ...
        num2str(numPkt-1) " packets, PER: " ...
        num2str(packetErrorRate(imcs,isnr))],""));
  end
end
```

```
MCS:9, SNR 6 completed after 11 packets, PER: 1
MCS:9, SNR 6.5 completed after 16 packets, PER: 0.6875
MCS:9, SNR 7 completed after 38 packets, PER: 0.28947
MCS:9, SNR 7.5 completed after 99 packets, PER: 0.11111
MCS:9, SNR 8 completed after 100 packets, PER: 0.04
MCS:9, SNR 8.5 completed after 100 packets, PER: 0.01
```

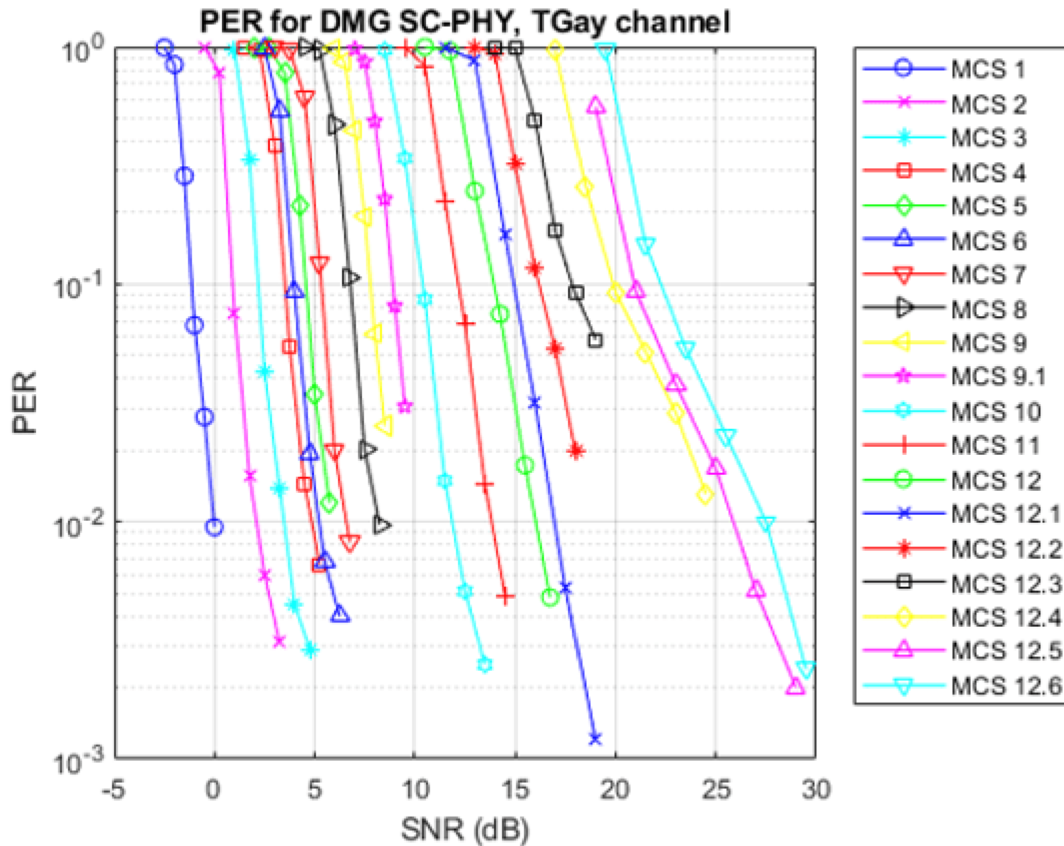
Plot Packet Error Rate vs SNR Results

```
markers = 'ox*sd^v><ph+ox*sd^v';
color = 'bmcrgbrkymcrgbrkymc';
figure;
for imcs = 1:numMCS
    semilogy(snrRanges{mcs(imcs)==validMCS},packetErrorRate(imcs,:).',['-' markers(imcs) color(in
    hold on;
end
grid on;
xlabel('SNR (dB)');
ylabel('PER');
dataStr = arrayfun(@(x)sprintf('MCS %s',x),mcs,'UniformOutput',false);
legend(dataStr);
title('PER for DMG SC-PHY, TGay channel');
```



Further Exploration

The number of packets tested at each SNR point is controlled by two parameters; `maxNumErrors` and `maxNumPackets`. For meaningful results, it is recommended that these values should be larger than those presented in this example. Increasing the number of packets simulated allows the PER under different scenarios to be compared. Try changing the MCS value and compare the packet error rate. As an example, the figure below was created by running the example for all single carrier MCS with `PSDULength`: 8192 bytes, `maxNumErrors`: 1000 and `maxNumPackets`: 10000.



Explore the TGay channel settings by changing the environment ('Street canyon hotspot' or 'Large hotel lobby'), user configurations, polarization type, array configurations, beamforming method and so on. In the street canyon scenario, the object calculates deterministic rays up to 1-order reflection from the ground and walls. In the hotel lobby scenario, the deterministic rays are up to 2-order reflection from the ground, ceiling and/or walls. Due to the additional reflection of rays, the PER results are typically better than the above obtained from the open area scenario.

Selected Bibliography

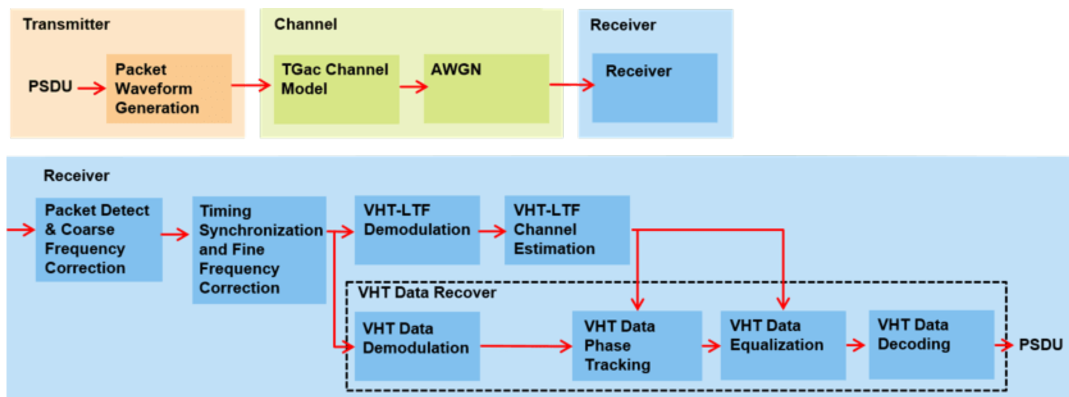
1. IEEE Std 802.11™-2020. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.
2. A. Maltsev and et. al, Channel Models for IEEE 802.11ay, IEEE 802.11-15/1150r9, Mar. 2017.

802.11ac Packet Error Rate Simulation for 8x8 TGac Channel

This example shows how to measure the packet error rate of an IEEE® 802.11ac™ VHT link using an end-to-end simulation with a fading TGac channel model and additive white Gaussian noise.

Introduction

In this example an end-to-end simulation is used to determine the packet error rate for an 802.11ac [1] VHT link with a fading channel at a selection of SNR points. At each SNR point multiple packets are transmitted through a channel, demodulated and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate. Packet detection, timing synchronization, carrier frequency offset correction and phase tracking are performed by the receiver. The processing for each packet is summarized in the following



This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each SNR point to speed up a simulation. The `parfor` function, as part of the Parallel Computing Toolbox™, executes processing for each SNR in parallel to reduce the total simulation time.

Waveform Configuration

An 802.11ac VHT transmission is simulated in this example. The VHT format configuration object, `wlanVHTConfig`, contains the format-specific configuration of the transmission. The properties of the object contain the configuration. In this example the object is configured for a 80 MHz channel bandwidth, 8 transmit antennas, 8 space-time streams, no space time block coding, 256-QAM rate-5/6 (MCS 9), and binary convolutional coding (BCC).

```
% Create a format configuration object for a 8-by-8 VHT transmission
cfgVHT = wlanVHTConfig;
cfgVHT.ChannelBandwidth = 'CBW80'; % 80 MHz channel bandwidth
cfgVHT.NumTransmitAntennas = 8;    % 8 transmit antennas
cfgVHT.NumSpaceTimeStreams = 8;    % 8 space-time streams
cfgVHT.APEPLength = 3000;          % APEP length in bytes
cfgVHT.MCS = 9;                    % 256-QAM rate-5/6
cfgVHT.ChannelCoding = 'BCC';      % Binary convolutional coding
```

Channel Configuration

In this example a TGac N-LOS channel model is used with delay profile Model-D. For Model-D when the distance between transmitter and receiver is greater than or equal to 10 meters, the model is

NLOS. This is described further in `wlanTGacChannel`. An 8x8 MIMO channel is simulated in this example therefore 8 receive antennas are specified.

```
% Create and configure the channel
tgacChannel = wlanTGacChannel;
tgacChannel.DelayProfile = 'Model-D';
tgacChannel.NumReceiveAntennas = 8;
tgacChannel.TransmitReceiveDistance = 10; % Distance in meters for NLOS
tgacChannel.ChannelBandwidth = cfgVHT.ChannelBandwidth;
tgacChannel.NumTransmitAntennas = cfgVHT.NumTransmitAntennas;
tgacChannel.LargeScaleFadingEffect = 'None';
tgacChannel.NormalizeChannelOutputs = false;
```

Simulation Parameters

For each SNR point in the vector `snr` a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 40:5:50;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For meaningful results we recommend increasing the numbers.

```
maxNumErrors = 10;   % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Set the remaining variables for the simulation.

```
% Get the baseband sampling rate
fs = wlanSampleRate(cfgVHT);

% Get the OFDM info
ofdmInfo = wlanVHTOFDMInfo('VHT-Data',cfgVHT);

% Set the sampling rate of the channel
tgacChannel.SampleRate = fs;

% Indices for accessing each field within the time-domain packet
ind = wlanFieldIndices(cfgVHT);
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated.

For each packet the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through a different realization of the TGac channel model.
3. AWGN is added to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.

4 The packet is detected.
5 Coarse carrier frequency offset is estimated and corrected.
6 Fine timing synchronization is established. The L-STF, L-LTF and L-SIG samples are provided for fine timing to allow for packet detection at the start or end of the L-STF.
7 Fine carrier frequency offset is estimated and corrected.
8 The VHT-LTF is extracted from the synchronized received waveform. The VHT-LTF is OFDM demodulated and channel estimation is performed.
9 The VHT Data field is extracted from the synchronized received waveform. The PSDU is recovered using the extracted field and the channel estimate.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the `for` statement and uncomment the `parfor` statement below.

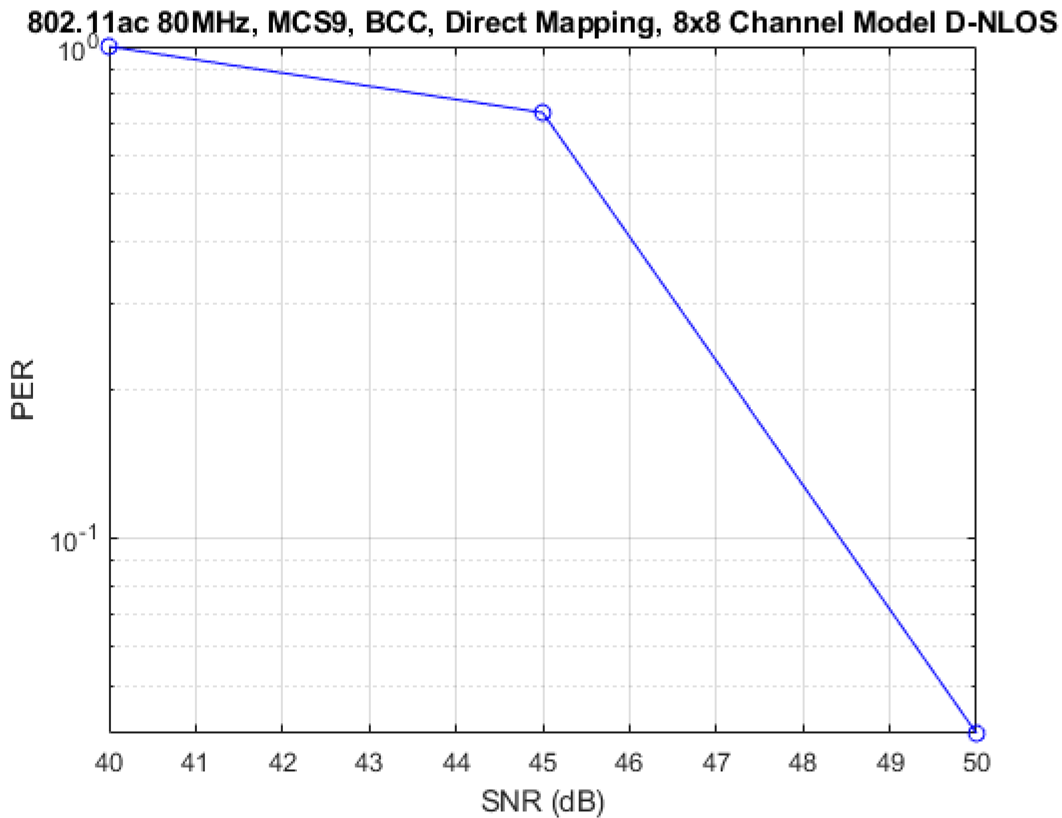
```
S = numel(snr);
packetErrorRate = zeros(S,1);
%parfor i = 1:S % Use 'parfor' to speed up the simulation
for i = 1:S     % Use 'for' to debug the simulation
    % Set random substream index per iteration to ensure that each
    % iteration uses a repeatable set of random numbers
    stream = RandStream('combRecursive','Seed',0);
    stream.Substream = i;
    RandStream.setGlobalStream(stream);

    % Account for noise energy in nulls so the SNR is defined per
    % active subcarrier
    packetSNR = snr(i)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

    % Loop to simulate multiple packets
    numPacketErrors = 0;
    numPkt = 1; % Index of packet transmitted
    while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
        % Generate a packet waveform
        txPSDU = randi([0 1],cfgVHT.PSDULength*8,1); % PSDULength in bytes
        tx = wlanWaveformGenerator(txPSDU,cfgVHT);

        % Add trailing zeros to allow for channel delay
        tx = [tx; zeros(50,cfgVHT.NumTransmitAntennas)]; %#ok<AGROW>

        % Pass the waveform through the fading channel model
        reset(tgacChannel); % Reset channel for different realization
        rx = tgacChannel(tx);

        % Add noise
        rx = awgn(rx,packetSNR);

        % Packet detect and determine coarse packet offset
        coarsePktOffset = wlanPacketDetect(rx,cfgVHT.ChannelBandwidth);
        if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
            numPacketErrors = numPacketErrors+1;
            numPkt = numPkt+1;
            continue; % Go to next loop iteration
        end

        % Extract L-STF and perform coarse frequency offset correction
```

```
        lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
        coarseFreqOff = wlanCoarseCFOEstimate(lstf,cfgVHT.ChannelBandwidth);
        rx = frequencyOffset(rx,fs,-coarseFreqOff);

        % Extract the non-HT fields and determine fine packet offset
        nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
        finePktOffset = wlanSymbolTimingEstimate(nonhtfields,...
                    cfgVHT.ChannelBandwidth);

        % Determine final packet offset
        pktOffset = coarsePktOffset+finePktOffset;

        % If packet detected outwith the range of expected delays from the
        % channel modeling; packet error
        if pktOffset>50
            numPacketErrors = numPacketErrors+1;
            numPkt = numPkt+1;
            continue; % Go to next loop iteration
        end

        % Extract L-LTF and perform fine frequency offset correction
        lltf = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
        fineFreqOff = wlanFineCFOEstimate(lltf,cfgVHT.ChannelBandwidth);
        rx = frequencyOffset(rx,fs,-fineFreqOff);

        % Extract VHT-LTF samples from the waveform, demodulate and perform
        % channel estimation
        vhtltf = rx(pktOffset+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
        vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,cfgVHT);

        % Channel estimate
        [chanEst,chanEstSSPilots] = wlanVHTLTFChannelEstimate(vhtltfDemod,cfgVHT);

        % Extract VHT Data samples from the waveform
        vhtdata = rx(pktOffset+(ind.VHTData(1):ind.VHTData(2)),:);

        % Estimate the noise power in VHT data field
        nVarVHT = vhtNoiseEstimate(vhtdata,chanEstSSPilots,cfgVHT);

        % Recover the transmitted PSDU in VHT Data
        rxPSDU = wlanVHTDataRecover(vhtdata,chanEst,nVarVHT,cfgVHT,...
            'LDPCDecodingMethod','norm-min-sum');

        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(txPSDU,rxPSDU));
        numPacketErrors = numPacketErrors+packetError;
        numPkt = numPkt+1;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(i) = numPacketErrors/(numPkt-1);
    disp(['SNR ' num2str(snr(i)) ' completed after ' ...
        num2str(numPkt-1) ' packets, PER: ' ...
        num2str(packetErrorRate(i))]);
end
```

```
SNR 40 completed after 11 packets, PER: 1
SNR 45 completed after 15 packets, PER: 0.73333
SNR 50 completed after 100 packets, PER: 0.04
```

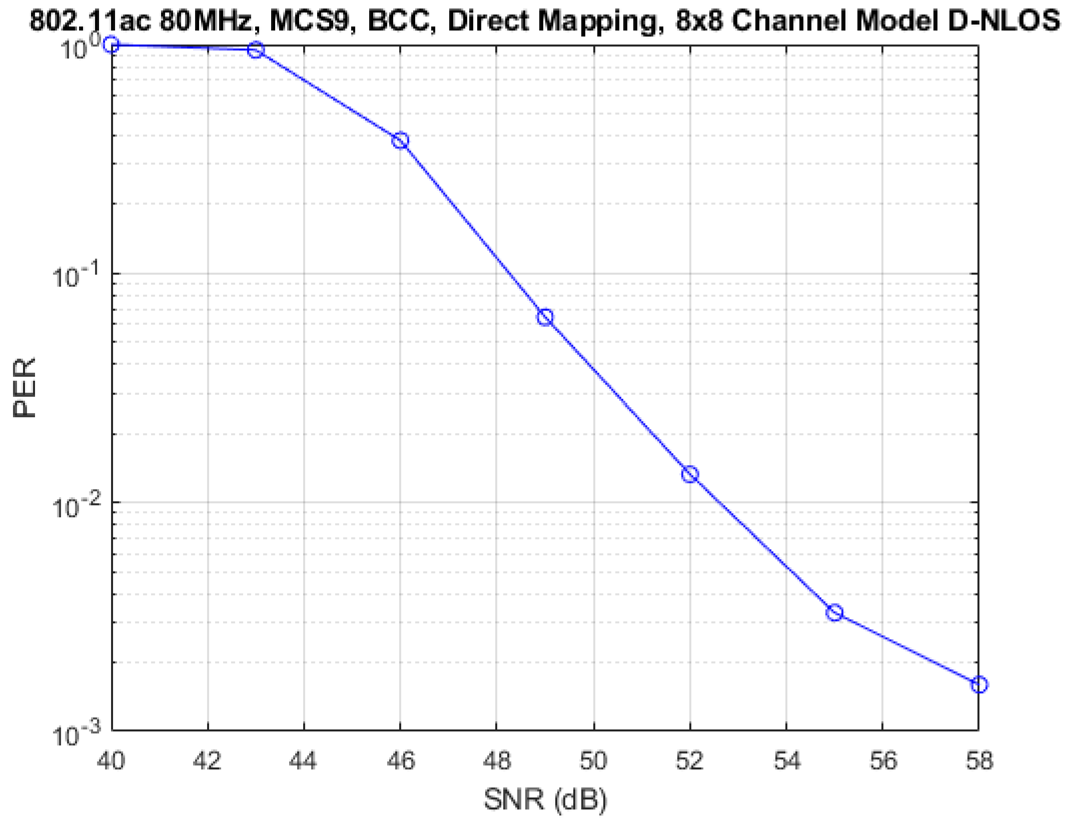
Plot Packet Error Rate vs SNR Results

```
figure
semilogy(snr,packetErrorRate,'-ob');
grid on;
xlabel('SNR (dB)');
ylabel('PER');
title('802.11ac 80MHz, MCS9, BCC, Direct Mapping, 8x8 Channel Model D-NLOS');
```



Further Exploration

The number of packets tested at each SNR point is controlled by two parameters; `maxNumErrors` and `maxNumPackets`. For meaningful results it is recommend that these values should be larger than those presented in this example. Increasing the number of packets simulated allows the PER under different scenarios to be compared. Try changing the transmission and reception configurations and compare the packet error rate. As an example, the figure below was created by running the example for `maxNumErrors`: 1000 and `maxNumPackets`: 10000.



Selected Bibliography

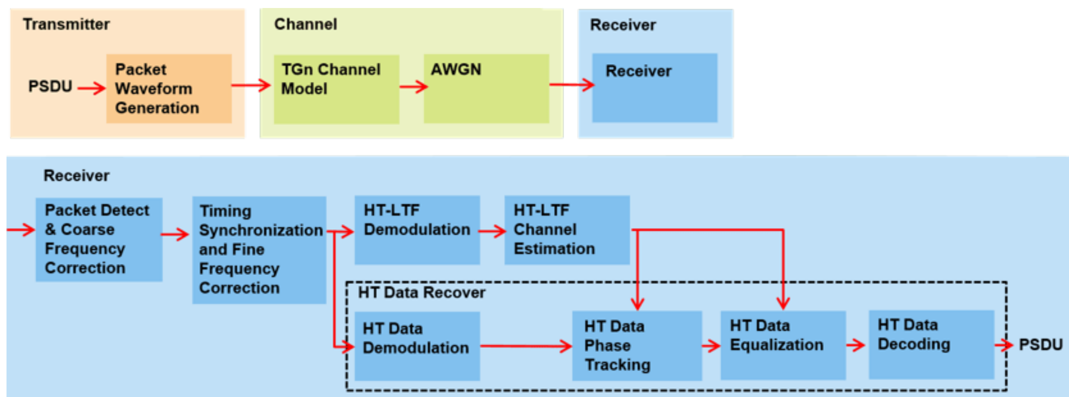
1 IEEE Std 802.11™-2020. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

802.11n Packet Error Rate Simulation for 2x2 TGn Channel

This example shows how to measure the packet error rate of an IEEE® 802.11n™ HT link using an end-to-end simulation with a fading TGn channel model and additive white Gaussian noise.

Introduction

In this example an end-to-end simulation is used to determine the packet error rate for an 802.11n HT [1] link with a fading channel at a selection of SNR points. At each SNR point multiple packets are transmitted through a channel, demodulated and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate. Packet detection, timing synchronization, carrier frequency offset correction and phase tracking are performed by the receiver. The processing for each packet is summarized in the following diagram.



This example also demonstrates how a `parfor` loop can be used instead of a `for` loop when simulating each SNR point to speed up a simulation. The `parfor` function, as part of the Parallel Computing Toolbox™, executes processing for each SNR in parallel to reduce the total simulation time.

Waveform Configuration

An 802.11n HT transmission is simulated in this example. The HT format configuration object, `wlanHTConfig`, contains the format specific configuration of the transmission. The properties of the object contain the configuration. In this example the object is configured for a 20 MHz channel bandwidth, 2 transmit antennas, 2 space time streams and no space time block coding.

```
% Create a format configuration object for a 2-by-2 HT transmission
cfgHT = wlanHTConfig;
cfgHT.ChannelBandwidth = 'CBW20'; % 20 MHz channel bandwidth
cfgHT.NumTransmitAntennas = 2;    % 2 transmit antennas
cfgHT.NumSpaceTimeStreams = 2;    % 2 space-time streams
cfgHT.PSDULength = 1000;          % PSDU length in bytes
cfgHT.MCS = 15;                   % 2 spatial streams, 64-QAM rate-5/6
cfgHT.ChannelCoding = 'BCC';      % BCC channel coding
```

Channel Configuration

In this example a TGn N-LOS channel model is used with delay profile Model-B. For Model-B when the distance between transmitter and receiver is greater than or equal to five meters, the model is NLOS. This is described further in `wlanTGnChannel`.

```
% Create and configure the channel
tgnChannel = wlanTGnChannel;
tgnChannel.DelayProfile = 'Model-B';
tgnChannel.NumTransmitAntennas = cfgHT.NumTransmitAntennas;
tgnChannel.NumReceiveAntennas = 2;
tgnChannel.TransmitReceiveDistance = 10; % Distance in meters for NLOS
tgnChannel.LargeScaleFadingEffect = 'None';
tgnChannel.NormalizeChannelOutputs = false;
```

Simulation Parameters

For each SNR point in the vector `snr` a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 25:10:45;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumPEs` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For meaningful results we recommend increasing the numbers.

```
maxNumPEs = 10; % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Set the remaining variables for the simulation.

```
% Get the baseband sampling rate
fs = wlanSampleRate(cfgHT);

% Get the OFDM info
ofdmInfo = wlanHTOFDMInfo('HT-Data',cfgHT);

% Set the sampling rate of the channel
tgnChannel.SampleRate = fs;

% Indices for accessing each field within the time-domain packet
ind = wlanFieldIndices(cfgHT);
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated.

For each packet the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through a different realization of the TGn channel model.
3. AWGN is added to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.
4. The packet is detected.
5. Coarse carrier frequency offset is estimated and corrected.

6. Fine timing synchronization is established. The L-STF, L-LTF and L-SIG samples are provided for fine timing to allow for packet detection at the start or end of the L-STF.
7. Fine carrier frequency offset is estimated and corrected.
8. The HT-LTF is extracted from the synchronized received waveform. The HT-LTF is OFDM demodulated and channel estimation is performed.
9. The HT Data field is extracted from the synchronized received waveform. The PSDU is recovered using the extracted field and the channel estimate.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

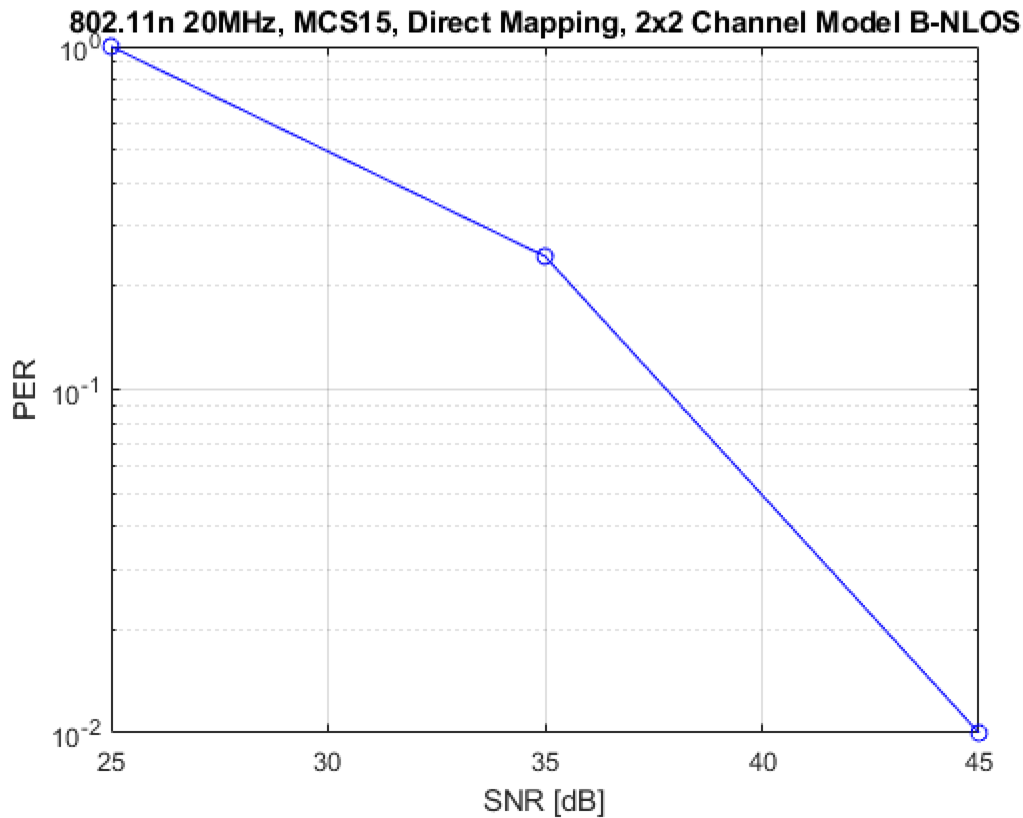
```
S = numel(snr);
packetErrorRate = zeros(S,1);
%parfor i = 1:S % Use 'parfor' to speed up the simulation
for i = 1:S % Use 'for' to debug the simulation
    % Set random substream index per iteration to ensure that each
    % iteration uses a repeatable set of random numbers
    stream = RandStream('combRecursive','Seed',0);
    stream.Substream = i;
    RandStream.setGlobalStream(stream);

    % Account for noise energy in nulls so the SNR is defined per
    % active subcarrier
    packetSNR = snr(i)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

    % Loop to simulate multiple packets
    numPacketErrors = 0;
    n = 1; % Index of packet transmitted
    while numPacketErrors<=maxNumPEs && n<=maxNumPackets
        % Generate a packet waveform
        txPSDU = randi([0 1],cfgHT.PSDULength*8,1); % PSDULength in bytes
        tx = wlanWaveformGenerator(txPSDU,cfgHT);

        % Add trailing zeros to allow for channel filter delay
        tx = [tx; zeros(15,cfgHT.NumTransmitAntennas)]; %#ok<AGROW>

        % Pass the waveform through the TGn channel model
        reset(tgnChannel); % Reset channel for different realization
        rx = tgnChannel(tx);

        % Add noise
        rx = awgn(rx,packetSNR);

        % Packet detect and determine coarse packet offset
        coarsePktOffset = wlanPacketDetect(rx,cfgHT.ChannelBandwidth);
        if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
            numPacketErrors = numPacketErrors+1;
            n = n+1;
            continue; % Go to next loop iteration
        end

        % Extract L-STF and perform coarse frequency offset correction
        lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
        coarseFreqOff = wlanCoarseCFOEstimate(lstf,cfgHT.ChannelBandwidth);
        rx = frequencyOffset(rx,fs,-coarseFreqOff);
```

```
        % Extract the non-HT fields and determine fine packet offset
        nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
        finePktOffset = wlanSymbolTimingEstimate(nonhtfields,...
            cfgHT.ChannelBandwidth);

        % Determine final packet offset
        pktOffset = coarsePktOffset+finePktOffset;

        % If packet detected outwith the range of expected delays from the
        % channel modeling; packet error
        if pktOffset>15
            numPacketErrors = numPacketErrors+1;
            n = n+1;
            continue; % Go to next loop iteration
        end

        % Extract L-LTF and perform fine frequency offset correction
        lltf = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
        fineFreqOff = wlanFineCFOEstimate(lltf,cfgHT.ChannelBandwidth);
        rx = frequencyOffset(rx,fs,-fineFreqOff);

        % Extract HT-LTF samples from the waveform, demodulate and perform
        % channel estimation
        htltf = rx(pktOffset+(ind.HTLTF(1):ind.HTLTF(2)),:);
        htltfDemod = wlanHTLTFDemodulate(htltf,cfgHT);
        chanEst = wlanHTLTFChannelEstimate(htltfDemod,cfgHT);

        % Extract HT Data samples from the waveform
        htdata = rx(pktOffset+(ind.HTData(1):ind.HTData(2)),:);

        % Estimate the noise power in HT data field
        nVarHT = htNoiseEstimate(htdata,chanEst,cfgHT);

        % Recover the transmitted PSDU in HT Data
        rxPSDU = wlanHTDataRecover(htdata,chanEst,nVarHT,cfgHT,...
            "LDPCDecodingMethod","norm-min-sum");

        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(txPSDU,rxPSDU));
        numPacketErrors = numPacketErrors+packetError;
        n = n+1;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(i) = numPacketErrors/(n-1);
    disp(['SNR ' num2str(snr(i))...
          ' completed after '  num2str(n-1) ' packets,'...
          ' PER: ' num2str(packetErrorRate(i))]);
end
```

```
SNR 25 completed after 11 packets, PER: 1
SNR 35 completed after 45 packets, PER: 0.24444
SNR 45 completed after 100 packets, PER: 0.01
```

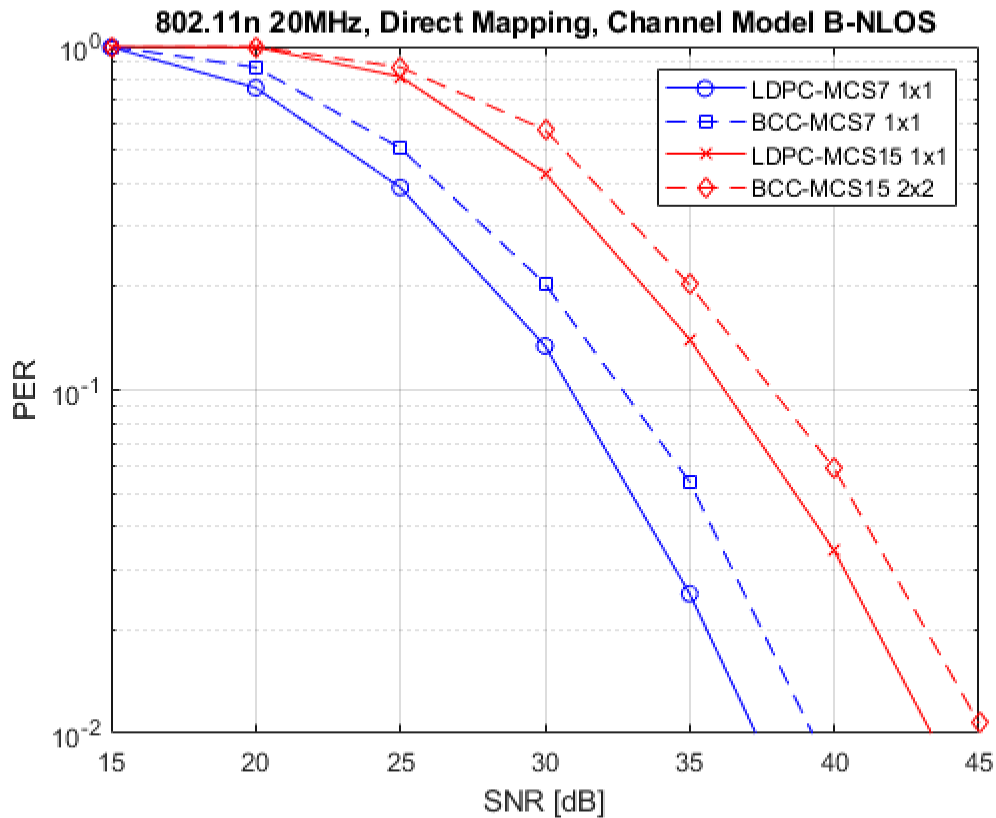
Plot Packet Error Rate vs SNR Results

```
figure;
semilogy(snr,packetErrorRate,'-ob');
grid on;
xlabel('SNR [dB]');
ylabel('PER');
title('802.11n 20MHz, MCS15, Direct Mapping, 2x2 Channel Model B-NLOS');
```



Further Exploration

The number of packets tested at each SNR point is controlled by two parameters; `maxNumPEs` and `maxNumPackets`. For meaningful results it is recommended that these values should be larger than those presented in this example. Increasing the number of packets simulated allows the PER under different scenarios to be compared. Try changing the transmit encoding scheme to LDPC and compare the packet error rate. As an example, the figure below was created by running the example for `maxNumPEs`: 200 and `maxNumPackets`: 10000, with four different configurations; 1x1 and 2x2 with BCC and LDPC encoding.



Appendix

This example uses the following helper functions:

- htNoiseEstimate.m

Selected Bibliography

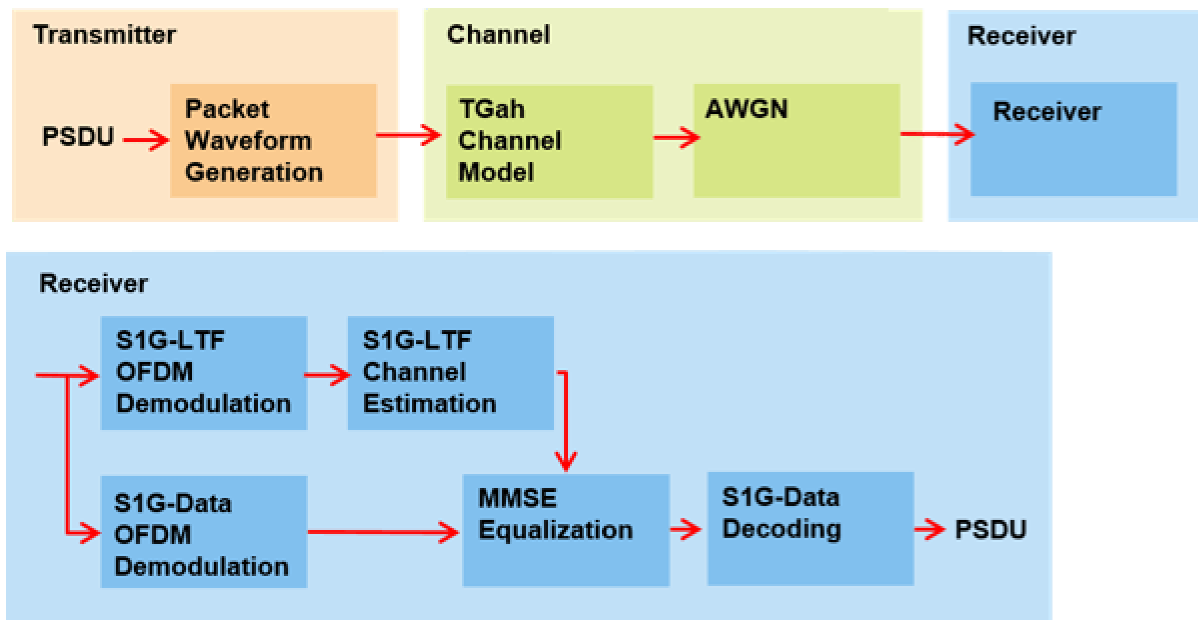
1 IEEE Std 802.11™-2020. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

802.11ah Packet Error Rate Simulation for 2x2 TGah Channel

This example shows how to measure the packet error rate of an IEEE® 802.11ah™ S1G short preamble link using an end-to-end simulation with a fading TGah indoor channel model and additive white Gaussian noise.

Introduction

In this example an end-to-end simulation is used to determine the packet error rate for an 802.11ah [1] S1G short preamble link with a fading channel at a selection of SNR points. For each SNR point multiple packets are transmitted through a channel, demodulated and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate. The processing for each packet is summarized in the following diagram.



This example simulates the S1G-Short format with no impairment correction apart from channel estimation and equalization. The received signal is synchronized to the start of the packet by compensating for a known delay and the default OFDM demodulation symbol sampling offset. No frequency synchronization is performed. For information on how to automatically detect and synchronize to the received signal, see these examples for 802.11n™ and 802.11ac™.

- “802.11n Packet Error Rate Simulation for 2x2 TGn Channel” on page 5-16
- “802.11ac Packet Error Rate Simulation for 8x8 TGac Channel” on page 5-10

This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each SNR point to speed up a simulation. `parfor` as part of the Parallel Computing Toolbox™, executes processing for each SNR in parallel to reduce the total simulation time.

Waveform Configuration

A single-user 802.11ah S1G short preamble transmission is simulated in this example. An S1G format configuration object contains the format specific configuration of the transmission. The object is

created using the `wlanS1GConfig` function. The properties of the object contain the configuration. In this example the object is configured for a 2 MHz channel bandwidth, short preamble, 2 transmit antennas, 2 space-time streams, 256 bytes payload, and 64-QAM rate-5/6 (MCS 7).

```
% Create S1G configuration object for single user S1G short preamble
% transmission with 2 transmit antennas and 2 space-time streams
cfgS1G = wlanS1GConfig;
cfgS1G.ChannelBandwidth = 'CBW2'; % 2 MHz channel bandwidth
cfgS1G.Preamble = 'Short';        % Short preamble
cfgS1G.NumTransmitAntennas = 2;   % 2 transmit antennas
cfgS1G.NumSpaceTimeStreams = 2;   % 2 space-time streams
cfgS1G.APEPLength = 256;          % APEP length in bytes
cfgS1G.MCS = 7;                   % 64-QAM rate-5/6
```

Channel Configuration

In this example a TGah N-LOS indoor channel model is used with delay profile Model-B. For Model-B, when the distance between transmitter and receiver is greater than or equal to 5 meters, the model is NLOS. This is described further in `wlanTGahChannel`. A 2x2 MIMO channel is simulated in this example therefore 2 receive antennas are specified.

```
% Create and configure the TGah channel
tgahChannel = wlanTGahChannel;
tgahChannel.DelayProfile = 'Model-B';
tgahChannel.NumTransmitAntennas = cfgS1G.NumTransmitAntennas;
tgahChannel.NumReceiveAntennas = 2;
tgahChannel.TransmitReceiveDistance = 5; % Distance in meters for NLOS
tgahChannel.ChannelBandwidth = cfgS1G.ChannelBandwidth;
tgahChannel.LargeScaleFadingEffect = 'None';
tgahChannel.NormalizeChannelOutputs = false;
```

Simulation Parameters

For each SNR point in the vector `snr` a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 25:10:45;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For meaningful results we recommend increasing the numbers.

```
maxNumErrors = 1e2;  % The maximum number of packet errors at an SNR point
maxNumPackets = 1e3; % The maximum number of packets at an SNR point
```

Set the remaining variables for the simulation.

```
% Indices for accessing each field within the time-domain packet
fieldInd = wlanFieldIndices(cfgS1G);

% OFDM information
```

```
ofdmInfo = wlanS1GOFDMInfo('S1G-Data',cfgS1G);

% Set the sampling rate of the channel
tgahChannel.SampleRate = wlanSampleRate(cfgS1G);

if ~strcmp(packetFormat(cfgS1G),'S1G-Short')
    error('This example only supports the S1G-Short packet format');
end
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated. At each SNR point:

1. Multiple data packets are transmitted through a 2x2 TGah channel with AWGN.
2. Each packet is time synchronized given a known delay.
3. The S1G-LTF1 and S1G-LTF2N fields are demodulated and channel estimation is performed.
4. The S1G-Data field is extracted from the synchronized received waveform and OFDM demodulated.
5. The data carrying subcarriers are equalized using the channel estimates.
6. The PSDU is recovered using the equalized data subcarriers, noise variance estimate, and channel state information (CSI).
7. The recovered PSDU of each packet is compared to those transmitted to determine the number of packet errors and hence the packet error rate.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

```
packetErrorRate = zeros(numel(snr),1);
%parfor i = 1:numel(snr) % Use 'parfor' to speed up the simulation
for i = 1:numel(snr) % 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 = i;
    RandStream.setGlobalStream(stream);

    % Account for noise energy in nulls so the SNR is defined per
    % active subcarrier
    packetSNR = snr(i)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

    % Loop to simulate multiple packets
    numPacketErrors = 0;
    numPkt = 1; % Index of packet transmitted
    while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
        % Generate a packet for 802.11ah short preamble
        txPSDU = randi([0 1],cfgS1G.PSDULength*8,1);
        txWaveform = wlanWaveformGenerator(txPSDU,cfgS1G);

        % Add trailing zeros to allow for channel delay
        tx = [txWaveform; zeros(50,cfgS1G.NumTransmitAntennas)];

        % Pass through fading indoor TGah channel
        reset(tgahChannel); % Reset channel for different realization
```

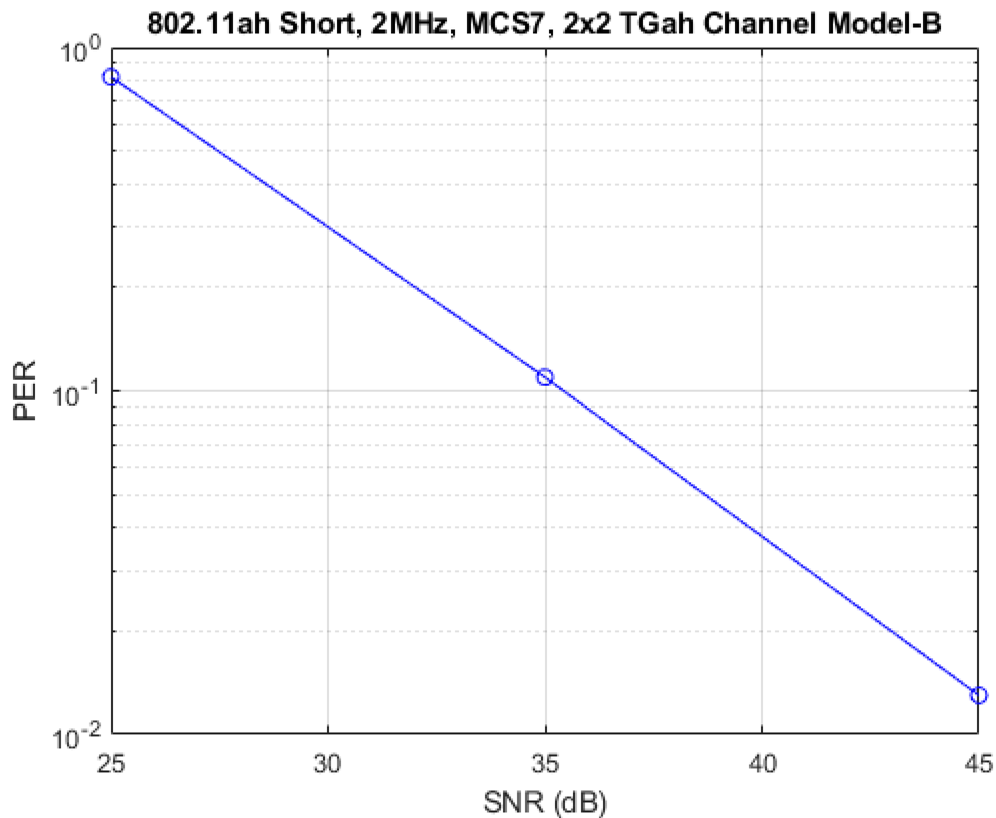
```
        rx = tgahChannel(tx);

        % Add noise
        rx = awgn(rx,packetSNR);

        % Synchronize
        % The received signal is synchronized to the start of the packet by
        % compensating for a known delay and the default OFDM demodulation
        % symbol sampling offset.
        delay = 4;
        rxSync = rx(delay+1:end,:);

        % LTF demodulation and channel estimation
        % Demodulate S1G-LTF1
        rxLTF1 = rxSync(fieldInd.S1GLTF1(1):fieldInd.S1GLTF1(2),:);
        demodLTF1 = wlanS1GDemodulate(rxLTF1,'S1G-LTF1',cfgS1G);

        % If required, demodulate S1G-LTF2N, and perform channel estimation
        if cfgS1G.NumSpaceTimeStreams>1
            % Use S1G-LTF1 and S1G-LTF2N for channel estimation
            rxLTF2N = rxSync(fieldInd.S1GLTF2N(1):fieldInd.S1GLTF2N(2),:);
            demodLTF2N = wlanS1GDemodulate(rxLTF2N,'S1G-LTF2N',cfgS1G);
            chanEst = s1gLTFChannelEstimate([demodLTF1 demodLTF2N],cfgS1G);
        else
            % Use only S1G-LTF1 for channel estimation
            chanEst = s1gLTFChannelEstimate(demodLTF1,cfgS1G);
        end

        % Noise variance estimate from S1G-LTF1 demodulated symbols
        noiseVarEst = wlanLLTFNoiseEstimate(demodLTF1);

        % Extract S1G-Data field
        rxData = rxSync(fieldInd.S1GData(1):fieldInd.S1GData(2),:);

        % OFDM demodulation
        demodSym = wlanS1GDemodulate(rxData,'S1G-Data',cfgS1G);

        % Extract data subcarriers from demodulated symbols and channel
        % estimate
        demodDataSym = demodSym(ofdmInfo.DataIndices,:,:);
        chanEstData = chanEst(ofdmInfo.DataIndices,:,:);

        % MMSE frequency domain equalization
        [eqDataSym,csi] = ofdmEqualize(demodDataSym,chanEstData,noiseVarEst);

        % Recover PSDU bits
        rxPSDU = s1gDataBitRecover(eqDataSym,noiseVarEst,csi,cfgS1G);

        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(txPSDU,rxPSDU));
        numPacketErrors = numPacketErrors+packetError;
        numPkt = numPkt+1;
    end

    % Compute PER for this SNR point
    packetErrorRate(i) = numPacketErrors/(numPkt-1);
    disp(['SNR ' num2str(snr(i))...
          ' completed after ' num2str(numPkt-1) ' packets,'...
```

```
        ' PER: ' num2str(packetErrorRate(i))]);
end
```

```
SNR 25 completed after 123 packets, PER: 0.82114
SNR 35 completed after 922 packets, PER: 0.10954
SNR 45 completed after 1000 packets, PER: 0.013
```

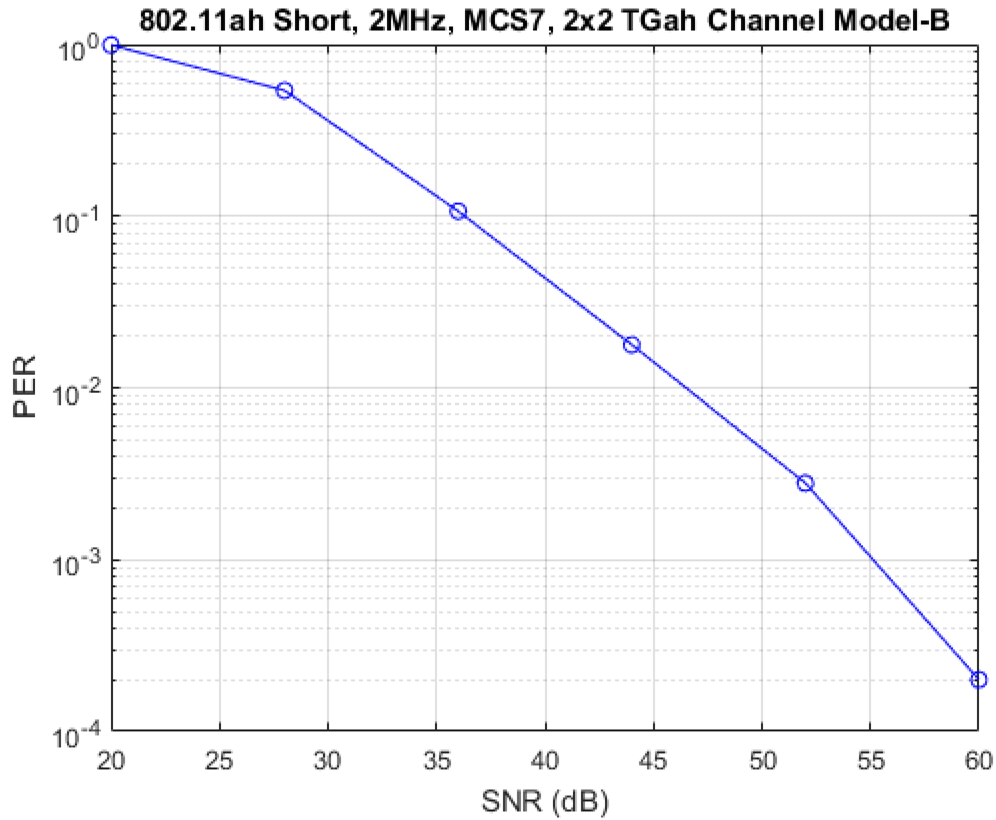
Plot Packet Error Rate vs SNR Results

```
figure;
semilogy(snr,packetErrorRate,'-ob');
grid on;
xlabel('SNR (dB)');
ylabel('PER');
title(['802.11ah ' cfgS1G.Preamble ', ' num2str(cfgS1G.ChannelBandwidth(4:end)) ...
    'MHz, MCS' num2str(cfgS1G.MCS) ', ' ...
    num2str(tgahChannel.NumTransmitAntennas) 'x' num2str(tgahChannel.NumReceiveAntennas) ...
    ' TGah Channel ' num2str(tgahChannel.DelayProfile)]);
```



Further Exploration

The number of packets tested at each SNR point is controlled by two parameters: `maxNumErrors` and `maxNumPackets`. For meaningful results, these values should be larger than those presented in this example. As an example, the figure below was created by running a longer simulation with `maxNumErrors = 1e3` and `maxNumPackets = 1e4`, and the SNR range `snr = 20:8:60`.



Selected Bibliography

1 IEEE P802.11ah™ D5.0 IEEE Draft 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 2: Sub 1 GHz License Exempt Operation.

Delay Profile and Fluorescent Lighting Effects

This example demonstrates the impact of changing the TGac delay profile, and it shows how fluorescent lighting affects the time response of the channel.

Delay Profile Effects

Create VHT configuration object. Set the sample rate to 80 MHz.

```
cfgVHT = wlanVHTConfig;
fs = 80e6;
```

Generate random binary data and create a transmit waveform parameterized by the VHT configuration object.

```
d = randi([0 1],8*cfgVHT.PSDULength,1);
testWaveform = wlanWaveformGenerator(d,cfgVHT);
```

Create a TGac channel object. Set the delay profile to `'Model-A'`, which corresponds to flat fading. Disable large-scale fading effects.

```
tgacChan = wlanTGacChannel('SampleRate',fs, ...
    'ChannelBandwidth',cfgVHT.ChannelBandwidth, ...
    'DelayProfile','Model-A', ...
    'LargeScaleFadingEffect','None');
```

Pass the transmitted waveform through the TGac channel.

```
rxModelA = tgacChan(testWaveform);
```

Set the delay profile to Model-C, which corresponds to a multipath channel with 14 distinct paths and a 30 ns RMS delay spread. The maximum delay spread is 200 ns, which corresponds to a coherence bandwidth of 2.5 MHz.

```
release(tgacChan)
tgacChan.DelayProfile = 'Model-C';
```

Pass the waveform through the model-C channel.

```
rxModelC = tgacChan(testWaveform);
```

Create a spectrum analyzer and use it to visualize the spectrum of the received signals.

```
saScope = spectrumAnalyzer(SampleRate=fs, ...
    ShowLegend=true,ChannelNames={'Model-A','Model-C'},...
    AveragingMethod='exponential',ForgettingFactor=0.99);
saScope([rxModelA rxModelC])
```

As expected, the frequency response of the model-A signal is flat across the 80 MHz bandwidth. Conversely, the model-C frequency response varies because its coherence bandwidth is much smaller than the channel bandwidth.

Fluorescent Effects

Release the TGac channel, and set its delay profile to `'Model-D'`. Disable the fluorescent lighting effect.

```
release(tgacChan)
tgacChan.DelayProfile = 'Model-D';
tgacChan.FluorescentEffect = false;
```

To better illustrate the Doppler effects of fluorescent lighting, change the bandwidth and sample rate of the channel. Generate a test waveform of ones.

```
tgacChan.ChannelBandwidth = 'CBW20';
fs = 20e6;
tgacChan.SampleRate = fs;
testWaveform = ones(5e5,1);
```

To ensure repeatability, set the global random number generator to a fixed value.

```
rng(37)
```

Pass the waveform through the TGac channel.

```
rxSig0 = tgacChan(testWaveform);
```

Enable the fluorescent lighting effect. Reset the random number generator, and pass the waveform through the channel.

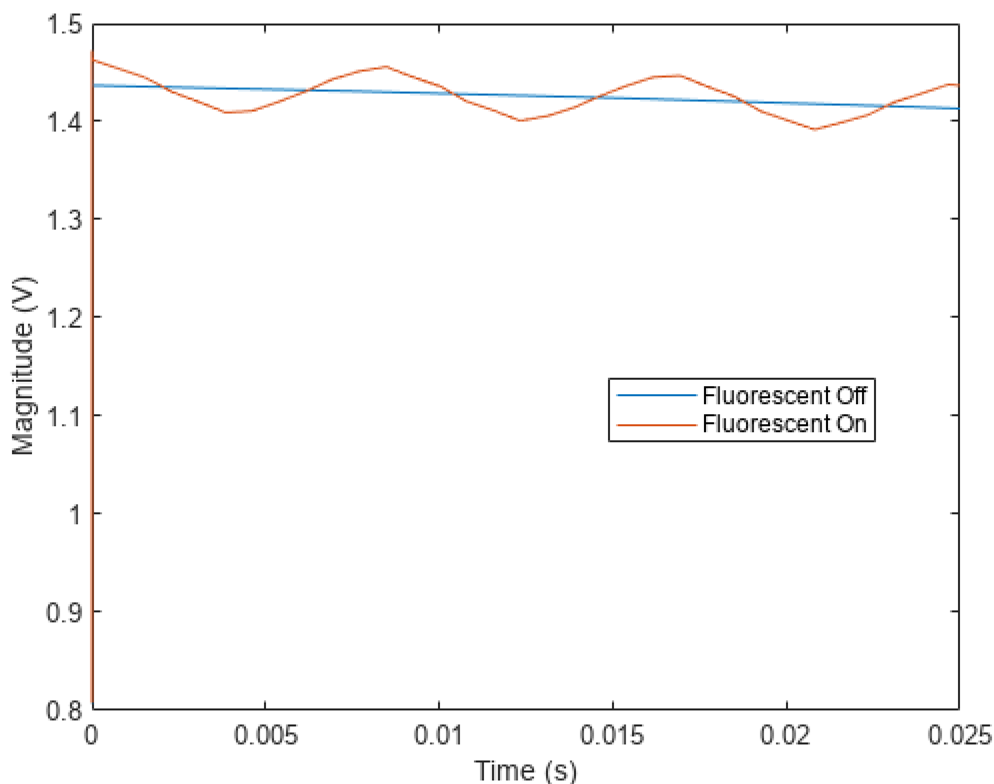
```
release(tgacChan)
tgacChan.FluorescentEffect = true;
rng(37)
rxSig1 = tgacChan(testWaveform);
```

Determine the time axis and channel filter delay.

```
t = ((1:size(rxSig0,1))'-1)/fs;
fDelay = tgacChan.info.ChannelFilterDelay;
```

Plot the magnitude of the received signals while accounting for the channel filter delay.

```
plot(t(fDelay+1:end),[abs(rxSig0(fDelay+1:end)) abs(rxSig1(fDelay+1:end))])
xlabel('Time (s)')
ylabel('Magnitude (V)')
legend('Fluorescent Off','Fluorescent On','location','best')
```



Fluorescent lighting introduces a Doppler component at twice the power line frequency (120 Hz in the U.S.).

Confirm that the peaks are separated by approximately 0.0083 s (inverse of 120 Hz) by measuring distance between the second and third peaks.

```
[~,loc] = findpeaks(abs(rxSig1(1e5:4e5)));
peakTimes = loc/fs;
peakSeparation = diff(peakTimes)
```

```
peakSeparation = 0.0085
```

6

End-to-End Simulation

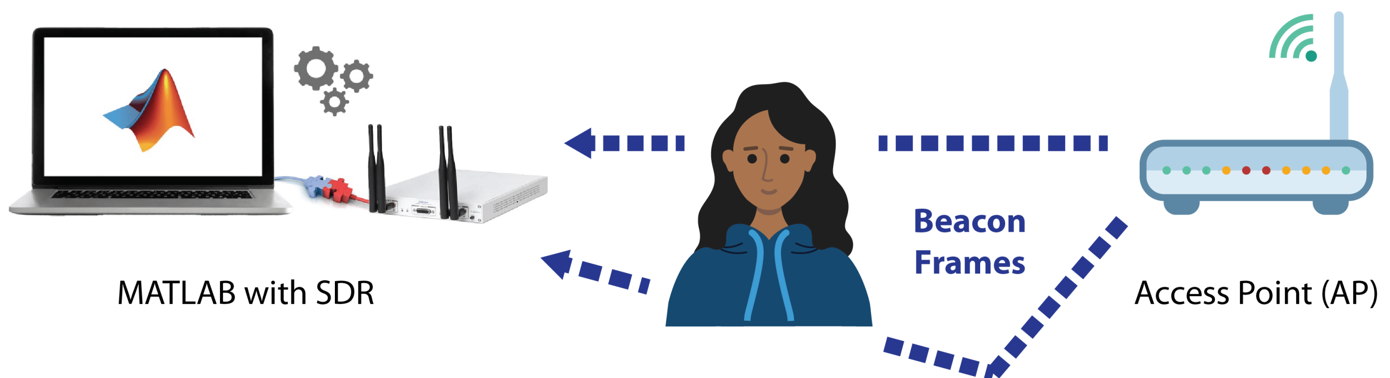
Detect Human Presence Using WLAN Signals and Deep Learning

This example shows how to use a convolutional neural network (CNN) to detect human presence by using the channel state information (CSI) in wireless local area networks. In this example, you can capture beacon frames from real routers with a software defined radio (SDR) to generate your own sensing dataset, or you can use prerecorded data. Then you train a CNN that detects human presence from the live SDR captures or the prerecorded data.

Introduction

WLAN sensing (also known as *Wi-Fi™ sensing*) uses Wi-Fi networks for applications such as gesture recognition, location tracking, monitoring vital signs, and human presence detection. WLAN sensing is currently being standardized by the IEEE® 802.11bf task group [1 on page 6-11]. This technology can sense presence, range, velocity, and location of objects in various environments through subtle changes in the communication channel. CSI is the most common metric for determining the changes in the channel [2 on page 6-11]. However, CSI is not typically exposed to you as a user of commercial Wi-Fi devices.

Using WLAN Toolbox™ with a supported SDR, you can capture WLAN waveforms, detect beacon packets, and extract CSI. From this CSI, you create a collection of periodogram images to use as the training and validation datasets for deep learning. Using Deep Learning Toolbox™, you train a CNN and use the trained network for live detection of human presence. Alternatively, you can use the prerecorded dataset to train the CNN and evaluate the presence detection performance against the test dataset. Beacon packets are useful for WLAN sensing as access points (APs) transmit beacon packets periodically. As a result, the sensing resolution is independent of the traffic load at the AP in the temporal domain.



SDR Setup

This section explains how to set up an SDR for capturing the data that will be used in training and live detection of human presence. If you do not have SDR hardware and plan to use the prerecorded data, you can skip this section.

This example supports these SDRs:

- ADALM-Pluto from the Communications Toolbox Support Package for Analog Devices® ADALM-Pluto Radio

- USRP™ E310/E312 from the Communications Toolbox Support Package for USRP™ Embedded Series Radio
- AD936x/FMCOMMS5 from the Communications Toolbox Support Package for Xilinx® Zynq®-Based Radio
- USRP™ N200/N210/USRP2/N320/N321/B200/B210/X300/X310 from the Communications Toolbox Support Package for USRP™ Radio

1) Ensure that you have installed the appropriate support package for the SDR that you intend to use and that you have configured the hardware accordingly.

2) To use your SDR as the data source for training the neural network and then inferencing with the trained CNN, select the `useSDR` check box.

```
useSDR = ; % You can skip this section if unchecked
```

3) Specify your capture device (`rxsim.DeviceName`), radio gain (`rxsim.RadioGain`), and a channel number (`rxsim.ChannelNumber`) for your operation frequency band (`rxsim.FrequencyBand)`. These settings determine the center frequency from which your SDR captures beacon packets. If you do not know a channel number, use the “OFDM Beacon Receiver Using Software-Defined Radio” on page 10-33 example to determine which channel(s) contain beacon packets.

```
if useSDR
    % User-defined parameters
    rxsim.DeviceName    = Pluto ;
    rxsim.RadioGain     = 15 ; % Can be 'AGC Slow Attack', 'AGC Fast Attack',
    rxsim.ChannelNumber = 36 ; % Valid values for 5 GHz band are integers in r
    rxsim.FrequencyBand = 5; % in GHz

    % Set up SDR receiver object
    rx = hSDRReceiver(rxsim.DeviceName);
    rx.SampleRate = 20e6; % Configured for 20 MHz since this is beacon transmission bandwidth
    rx.Gain = rxsim.RadioGain;
    rx.CenterFrequency = wlanChannelFrequency(rxsim.ChannelNumber,rxsim.FrequencyBand);
    rx.ChannelMapping = 1;
    rx.OutputDataType = 'single';

    rxsim.SDRObj = rx;
end
```

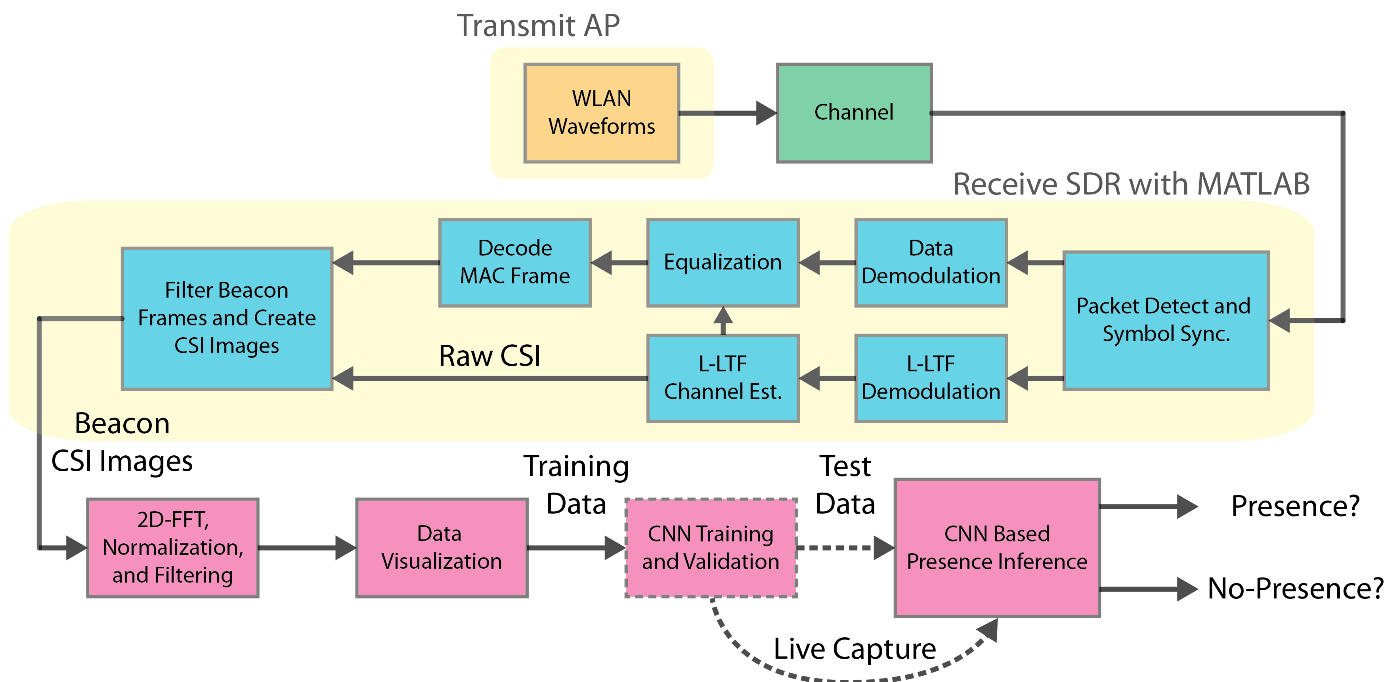
4) Configure these capture parameters:

- The number of beacon packets to be extracted from each capture (`rxsim.NumPacketsPerCapture`).
- The number of captures needed to create the dataset and train the neural network (`rxsim.NumCaptures`).
- The beacon interval (`rxsim.BeaconInterval`). The default beacon interval for a commercial AP is 100 time units (TUs), where 1 TU equals 1024 microseconds. You typically do not need to adjust the beacon interval value unless you have adjusted the interval of your access point (AP).
- Filter the beacons based on their SSID if more than one SSID exists on the same channel (`rxsim.BeaconSSID`).

```
if useSDR %#ok<*UNRCH>
    % User defined parameters
    rxsim.NumCaptures           = 10 ;
    rxsim.NumPacketsPerCapture = 8 ;
    rxsim.BeaconInterval        = 100 ; % in time units (TUs). The default value
    rxsim.BeaconSSID            = Enter text ; % Optional

    % Calculated parameters
    rxsim.CaptureDuration = rxsim.BeaconInterval*milliseconds(1.024)*rxsim.NumPacketsPerCapture -

    % SDR setup is complete
    msgbox("SDR object configuration is complete! Run steps 1 and 2 to capture data.")
    return % Stop execution
end
```

System Description

This example uses the following process to obtain CSI images for human presence detection:



When you use an SDR, the system captures an over-the-air waveform and performs receiver processing in WLAN Toolbox to extract the CSI from beacon packets. It does the same for all the packets in a capture to form a `numSubcarriers`-by-`rxsim.NumPacketsPerCapture` raw CSI array. Repeat this process for multiple captures to create an array of images. When you create the training data, the system assigns "no-presence" or "presence" labels to the captures. It obtains the periodogram representation of the raw CSI image array by using 2D-FFT, applying `[0 1]` range scaling, and removing DC components along the *x* and *y* axes. CSI periodogram images make it easier to focus on the CSI changes caused by human motion by discarding the effects of imperfections in the hardware, software, or environment. This focus also improves the training performance. Next, you use the periodogram images as an input dataset for CNN training and validation. Finally, you use the

trained CNN with live SDR captures to detect presence or test the performance of the CNN by using a subset of the prerecorded dataset. When capturing training data or performing live detection, you can visualize the magnitude spectrum and periodogram representation of the raw CSI.

This process was applied to obtain the prerecorded CSI dataset, which consists of 500 samples, with 250 labeled "no-presence" and 250 labeled "presence". When this dataset was being created, the `rxsim.NumCaptures` parameter was set to 250. An ADALM-Pluto SDR captured over-the-air beacon frames in a meeting room that contained furniture and had floor dimensions of 3m-by-3m. The beacon interval (`rxsim.BeaconInterval`) of the AP was set to 20 TUs. Two different human presence states were recorded: no presence (empty room) and presence (one person walking a random path).

Step 1: Create "no-presence" Data

In this section, you generate the "no-presence" data (`dataNoPresence`) and associated labels vector (`labelNoPresence`).

- If you are using an SDR, make sure no human presence exists in the environment before you create the data. The `captureCSIDataset` function displays the current capture number, status, and timestamp (`timestampNoPresence`).
- If you are using the prerecorded dataset, the `loadCSIDataset` function generates the `dataNoPresence`, `labelNoPresence`, `timestampNoPresence` parameters and visualizes the CSI dataset.

To generate the data, click **Create "no-presence" Data.** Two plots are generated for each capture:

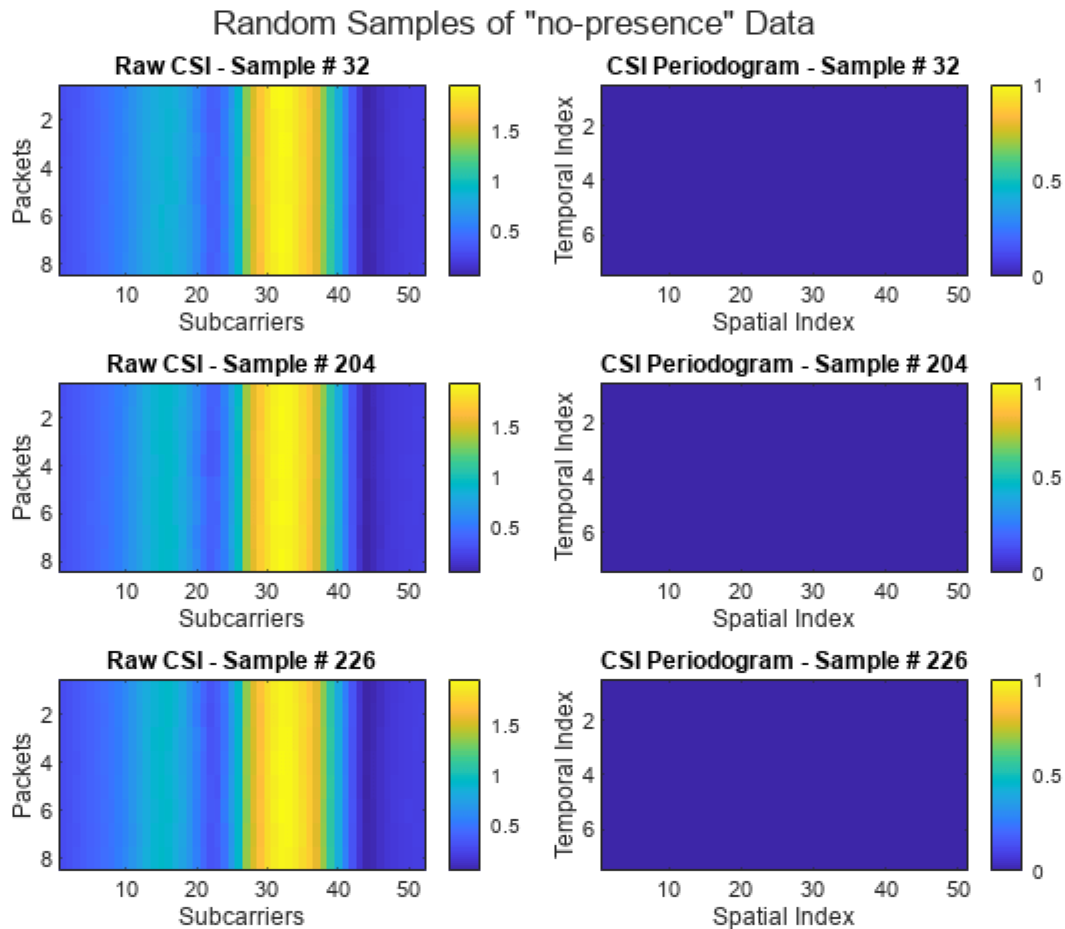
1. Magnitude spectrum of the captured CSI. Although the captured CSI is a complex valued array, this example uses the magnitude response of the CSI for visualization.
2. CSI periodogram image.

In "no-presence" data, the power content of the CSI spreads over both axes of the periodogram representation due to the lack of time-domain disturbance induced by the motion.

Create "no-presence" Data

```
if useSDR % Capture live CSI with SDR
    [dataNoPresence,labelNoPresence,timestampNoPresence] = captureCSIDataset(rxsim,"no-presence"
else % No SDR hardware, load CSI dataset
    [dataNoPresence,labelNoPresence,timestampNoPresence] = loadCSIDataset("dataset-no-presence.ma
end
```

```
Dimensions of the no-presence dataset (numSubcarriers x numPackets x numCaptures): [52    8  250
```



Step 2: Create "presence" Data

In this section, you generate the "presence" data (`dataPresence`) and associated labels vector (`labelPresence`).

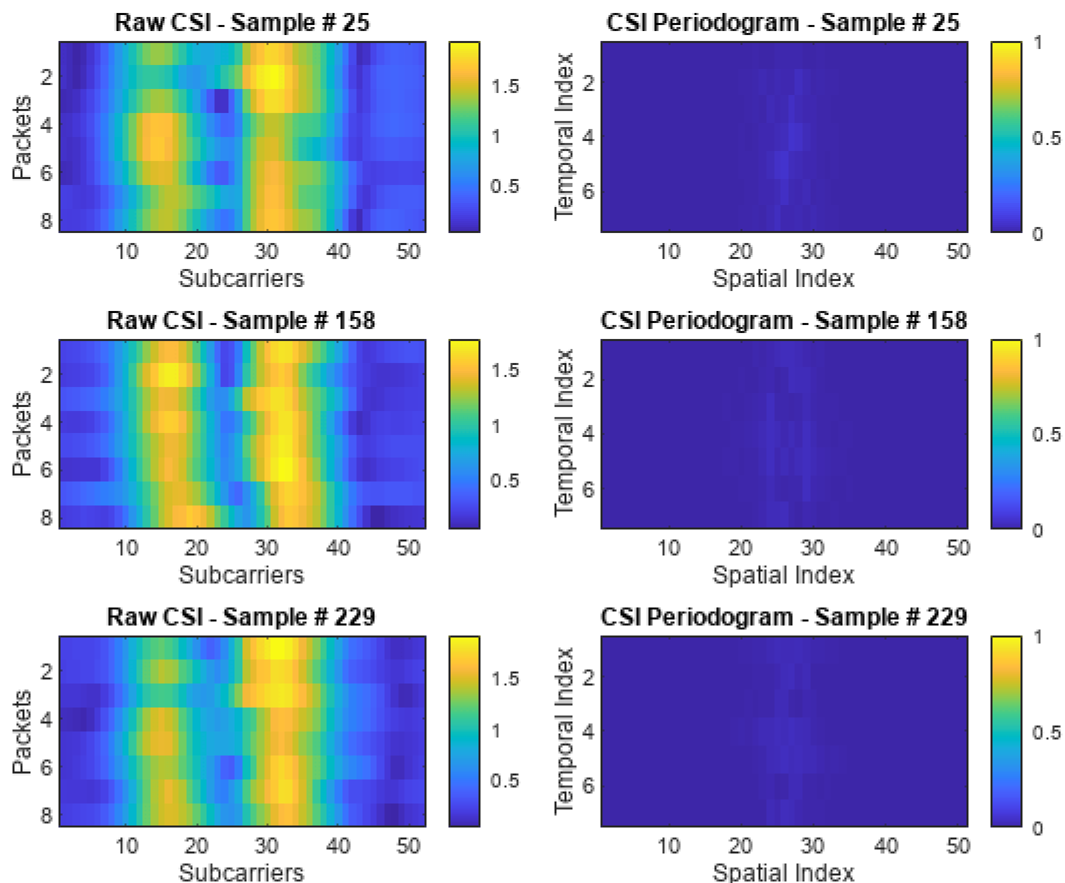
- If you are capturing data with an SDR, make sure constant human motion exists in the environment before you create the data. Otherwise, the quality of the captured data could significantly worsen the performance during training and inference. The `captureCSIDataset` function displays the current capture number, status, timestamp (`timestampPresence`), and CSI visualization.
- If you are using the prerecorded dataset, the `loadCSIDataset` function generates the `dataPresence`, `labelPresence`, `timestampPresence` parameters and visualizes randomly selected samples of the data. The "presence" data in the prerecorded dataset was collected with a person walking constantly in the room for the duration of the capture.

The capture visualization techniques are the same as in step 1. For the "presence" data, the periodogram plot shows focus of the CSI power content in both axes near the low frequency region that corresponds to the center of the image.

Create "presence" Data

```
if useSDR % Capture CSI with SDR
    [dataPresence,labelPresence,timestampPresence] = captureCSIDataset(rxsim,"presence");
else % No SDR hardware, load CSI dataset
    [dataPresence,labelPresence,timestampPresence] = loadCSIDataset("dataset-presence.mat","prese
end
```

```
Dimensions of the presence dataset (numSubcarriers x numPackets x numCaptures): [52    8  250]
```



Step 3: Create and Train CNN

CNNs are one of the most computationally efficient, accurate, and scalable approaches to problems involving deep learning. Their predominant application is to deep learning tasks based on computer vision [3 on page 6-11]. In this section, you define the CNN architecture and divide the data into training (`trainingData`) and validation (`validationData`) datasets to evaluate the training and final network performance.

- If you are using an SDR, you can evaluate the accuracy of the trained CNN using the live capture results. The example does not create a test set if you use an SDR.

- If you are using the precaptured data, you can evaluate the performance of the trained CNN using a test set (`testData`).

To create, train and validate the CNN click **Create CNN and Train**.

Create CNN and Train

```
trainingRatio =  0.8;
numEpochs = 10;
sizeMiniBatch = 8;
```

Use the `trainValTestSplit` function to create array datastore objects `trainingData`. This process is based on the `trainingRatio` parameter. A training dataset (`trainingData`), which comprises 80% of the whole dataset, updates the weights of the CNN to fit the general characteristics of the data. This process is based on the training labels contained in the array datastore object. A validation dataset (`validationData`), which comprises 10% of the whole dataset, ensures that the model is learning the general behavior of the system rather than memorizing the patterns in it. Finally, a test dataset, which also comprises 10% of the whole dataset, evaluates the performance of the trained CNN using unseen samples of the data. If you are using an SDR, the size of `validationData` becomes 20% of the whole dataset, because the testing is based on the human presence status inferences of the trained CNN.

```
[trainingData,validationData,testData,imgInputSize,numClasses] = trainValTestSplit(useSDR,trainin
```

```
CSI image input size: [51   7]
Number of training images: 400
```

The example uses CSI periodogram images with size (`numSubcarriers-1`)-by-(`rxsim.NumPacketsPerCapture-1`) as the training input. Use the “Create Simple Deep Learning Neural Network for Classification” (Deep Learning Toolbox) example to learn more about the individual layers and their roles in the architecture.

```
cnn = [
    imageInputLayer(imgInputSize,'Normalization','none')
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    dropoutLayer(0.1)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

cnn = layerGraph(cnn);
disp(cnn.Layers)
```

```
  16x1 Layer array with layers:

     1   'imageinput'    Image Input             51x7x1 images
     2   'conv_1'        2-D Convolution         8 3x3 convolutions with stride [1  1] and paddi
     3   'batchnorm_1'   Batch Normalization     Batch normalization
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     2-D Max Pooling         2x2 max pooling with stride [2  2] and padding
     6   'conv_2'        2-D Convolution         16 3x3 convolutions with stride [1  1] and padd
     7   'batchnorm_2'   Batch Normalization     Batch normalization
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     2-D Max Pooling         2x2 max pooling with stride [2  2] and padding
    10   'conv_3'        2-D Convolution         32 3x3 convolutions with stride [1  1] and padd
    11   'batchnorm_3'   Batch Normalization     Batch normalization
    12   'relu_3'        ReLU                    ReLU
    13   'dropout'       Dropout                 10% dropout
    14   'fc'            Fully Connected         2 fully connected layer
    15   'softmax'       Softmax                 softmax
    16   'classoutput'   Classification Output   crossentropyex
```

```
% Definition of the training options
options = trainingOptions('adam', ...
    LearnRateSchedule='piecewise', ...
    InitialLearnRate=0.001, ... % learning rate
    MaxEpochs=numEpochs, ...
    MiniBatchSize=sizeMiniBatch, ...
    ValidationData=validationData, ...
    Shuffle='every-epoch', ...
    ExecutionEnvironment='auto',...
    Verbose=true);
```

Train the deep network architecture `cnn` using the training dataset `trainingData` and the training options defined by `options`.

```
% Train the network
trainedCNN = trainNetwork(trainingData,cnn,options);
```

```
Training on single CPU.
|==========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validat
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|==========================================================================================
|       1 |           1 |       00:00:01 |       37.50% |       84.00% |       0.8120 |       0.
|       1 |          50 |       00:00:03 |      100.00% |      100.00% |       0.0085 |       0.
|       2 |         100 |       00:00:05 |      100.00% |      100.00% |       0.0059 |       0.
|       3 |         150 |       00:00:07 |      100.00% |      100.00% |       0.0134 |       0.
|       4 |         200 |       00:00:09 |      100.00% |      100.00% |       0.0029 |       0.
|       5 |         250 |       00:00:11 |      100.00% |      100.00% |       0.0123 |       0.
|       6 |         300 |       00:00:13 |      100.00% |      100.00% |       0.0012 |       0.
|       7 |         350 |       00:00:15 |      100.00% |      100.00% |       0.0008 |       0.
|       8 |         400 |       00:00:17 |      100.00% |      100.00% |       0.0006 |       0.
|       9 |         450 |       00:00:18 |      100.00% |      100.00% |       0.0001 |       0.
|      10 |         500 |       00:00:20 |      100.00% |      100.00% |       0.0002 |       0.
|==========================================================================================
Training finished: Max epochs completed.
```

Step 4: Evaluate Performance

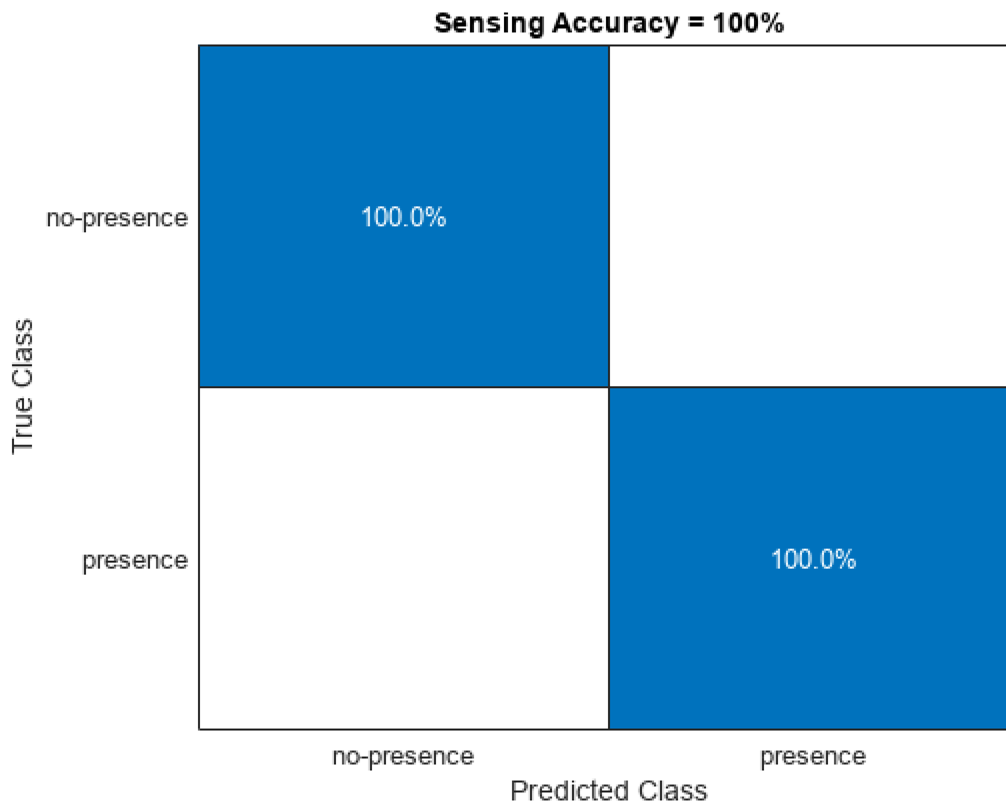
In this section, you evaluate the performance of the neural network using training data or live inference with an SDR.

- If you are using an SDR, this example uses the `livePresenceDetection` function to capture over-the-air beacons and infer human presence ("no-presence" or "presence") using the trained CNN that is for each capture. The function plots the instantaneous raw CSI and CSI periodogram plots, as described in steps 1 and 2, for each capture. The timestamped results for the 10 latest predictions are plotted. By default, `livePresenceDetection` runs for 20 captures. You can set `numCaptures` argument to any integer value. For continuous capturing, you can set `numCaptures` to `Inf`. The function returns results in the 1-by-2 cell array `sensingResults.` The first element is the prediction results and the second element is the timestamps. Note that `sensingResults` will not be created if `livePresenceDetection` is in the continuous capturing mode.
- If you are using the prerecorded data, this example uses the `testPresenceDetection` function to generate a confusion matrix in which the rows correspond to the ground truth values ("no-presence" or "presence") and the columns correspond to the CNN predictions ("no-presence" or "presence"). The diagonal elements represent the percentage of samples that are correctly classified. The off-diagonal elements are the percentage of incorrectly classified observations. The sensing accuracy value is calculated using the `testData` and returned in `sensingResults`.

To run the evaluation, click **Evaluate Performance**.

Evaluate Performance

```
if useSDR % Predict movement from the live SDR captures
    sensingResults = livePresenceDetection(rxsim,trainedCNN,20);
else % Predict movement by using the test set
    sensingResults = testPresenceDetection(testData,trainedCNN);
end
```



References

[1] IEEE P802.11 - TASK GROUP BF (WLAN SENSING), https://www.ieee802.org/11/Reports/tgbf_update.htm.

[2] IEEE 802.11-21/0407r2 - Multi-Band WiFi Fusion for WLAN Sensing, March 2021, https://mentor.ieee.org/802.11/dcn/21/11-21-0407-00-00bf-multi-band-wifi-fusion-for-wlan-sensing.pptx.

[3] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.

See Also

Related Examples

- “Three-Dimensional Indoor Positioning with 802.11az Fingerprinting and Deep Learning” on page 6-87
- “Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation” on page 6-133

- “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” on page 6-119

SNR Definition in End-to-End Simulations

This example shows how WLAN Toolbox™ defines signal-to-noise ratio (SNR) in end-to-end simulations.

SNR Definition

WLAN Toolbox™ end-to-end simulation examples introduce AWGN to the received signal in the time domain, after the fading channel, and before OFDM demodulation.

WLAN Toolbox defines the SNR as the expected SNR per subcarrier (SC) per receive antenna. In IEEE Std 802.11-2020 [1 on page 6-16], the transmitter normalizes the waveform, so that the sum of the baseband power over all transmit antennas is 1 W.

The expected SNR per subcarrier (SC) is denoted by SNR_{SC}.

$$SNR_{\text{SC}} = \frac{S_{\text{SC}}}{N_{\text{SC}}}$$

S_{SC} and N_{SC} are the average signal and noise power per subcarrier per receive antenna, respectively. N_{SC} models the AWGN that is added to the signal.

For a signal x with discrete Fourier transform (DFT) X, Parseval's theorem states

$$\sum_{n=1}^{N_{\text{FFT}}} |x_n|^2 = \frac{1}{N_{\text{FFT}}} \sum_{k=1}^{N_{\text{FFT}}} |X_k|^2.$$

N_{FFT} is the FFT length. Divide the equation by N_{FFT} to get the average signal power

$$S = \frac{1}{N_{\text{FFT}}} \sum_{n=1}^{N_{\text{FFT}}} |x_n|^2 = \frac{1}{N_{\text{FFT}}^2} \sum_{k=1}^{N_{\text{FFT}}} |X_k|^2 = \frac{1}{N_{\text{FFT}}} X_{\text{RMS}}^2.$$

In WLAN, the signal of interest does not use all FFT bins (or subcarriers) because of guard bands. If the signal uses only K_S subcarriers of the FFT, the signal power is

$$S = \frac{K_S}{N_{\text{FFT}}} \left(\frac{1}{N_{\text{FFT}}} X_{\text{RMS}}^2 \right) = \frac{K_S}{N_{\text{FFT}}{}^2} X_{\text{RMS}}^2. \ (1)$$

K_S is the number of nonzero power subcarriers per OFDM symbol.

The signal power per subcarrier is

$S_{SC} = \frac{S}{K_S}.$

The noise power per subcarrier is

$N_{SC} = \frac{N}{N_{FFT}}.$

Because the noise is added in the time domain, the noise occupies all subcarriers, not just the allocated subcarriers. Therefore, the noise power, N, is divided by N_{FFT} and not K_S.

With these definitions, the SNR_{SC} becomes

$$SNR_{SC} = \frac{S_{SC}}{N_{SC}} = \frac{\frac{K_S}{K_S N_{FFT}{}^2} X_{RMS}^2}{\frac{N}{N_{FFT}}} = \frac{X_{RMS}^2}{N_{FFT} N}. \ (2)$$

Rearranging equation 2, the noise power at the input of an OFDM demodulator is

$$N = \frac{X_{RMS}^2}{N_{FFT}\, SNR_{SC}}. \ (3)$$

The awgn function adds noise in the time domain. If you use equation 1 and equation 3, the time-domain SNR becomes

$$SNR = \frac{S}{N} = \frac{\left(\frac{K_S\, X_{RMS}{}^2}{N_{NFFT}{}^2}\right)}{\frac{X_{RMS}^2}{N_{FFT}\, SNR_{SC}}} = \frac{SNR_{SC}}{\left(\frac{N_{NFFT}}{K_S}\right)}.$$

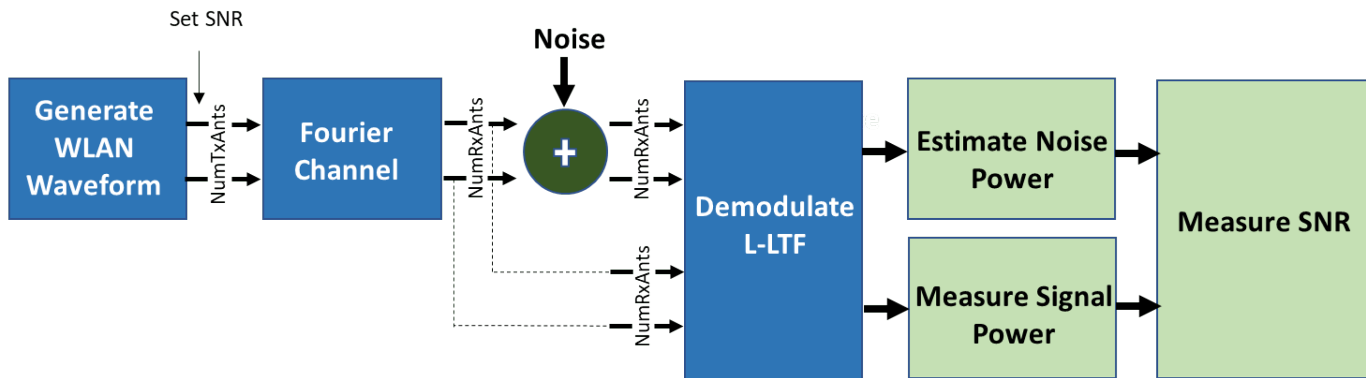
SNR_{SC} is in decibels. The SNR in decibels is

$$SNR = SNR_{SC} - 10\log_{10}\left({}^{N_{FFT}}\!/_{K_S}\right). \ (4)$$

SNR Verification

To verify your target SNR, you need to understand the process that WLAN Toolbox uses to simulate an end-to-end link for the given SNR. For this example, WLAN Toolbox simulates an end-to-end link and measures the SNR of the received packet as shown in the figure below. Multiple packets are transmitted through a Fourier channel. A Fourier matrix in a MIMO system assumes that all receive antennas are coupled with the transmit antennas. The matrix is orthogonal, so the spatial streams are completely separate, resulting in no noise enhancement [2 on page 6-16].

The example uses the demodulated L-LTF field with and without noise to estimate the noise and signal power of each packet, and hence, the SNR. The example displays the simulated SNR, which is averaged over multiple packets.



For each packet, the following steps occur in the process:

1. Generate a non-HT waveform.
2. Set the required SNR according to Equation-4.
3. Pass the waveform through a Fourier channel.
4. Add AWGN to the received waveform to create the desired average SNR per subcarrier after OFDM demodulation.
5. Extract and OFDM-demodulate the L-LTF field before and after adding noise.
6. Estimate the signal power by using the demodulated L-LTF symbols before adding noise.
7. Estimate the noise power by using the demodulated L-LTF symbols after adding noise.
8. Calculate and display the average SNR.

Simulation Setup

For this example specify a target SNR of 10 dB.

```
SNRSC = 10;
rng("default") % Set default random number generator for repeatability
```

Set the number of transmit and receive antennas.

```
nTxAnts = 2;
nRxAnts = 2;
```

Create a configuration object for an non-HT transmission.

```
cfg = wlanNonHTConfig;
cfg.NumTransmitAntennas = nTxAnts; % N transmit antennas
```

SNR Setup

Scale the SNR to account for energy in unused subcarriers according to equation 4.

```
ofdmInfo = wlanNonHTOFDMInfo('NonHT-Data',cfg); % Get the OFDM info
SNRdB = SNRSC-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);
```

Simulate Link

Now simulate the transmission and verify that the measured SNR approximates your target SNR.

```
% Indices for accessing each field within the time-domain packet
ind = wlanFieldIndices(cfg);
```

```
N = 100;
nVar = zeros(1,N);
signalPower = zeros(1,N);

for i=1:N
    % Generate a packet waveform
    txPSDU = randi([0 1],cfg.PSDULength*8,1); % PSDULength in bytes
    tx = wlanWaveformGenerator(txPSDU,cfg);

    % Channel
    h = dftmtx(max(nTxAnts,nRxAnts)); % Fourier channel
    h = h(1:nTxAnts,1:nRxAnts);
    rx = tx*h;

    % Add noise
    rxNoise = awgn(rx,SNRdB);

    % Measure noise power by comparing L-LTF over 2 symbols. The noise
    % power is averaged across receive antennas.
    lltf = rxNoise(ind.LLTF(1):ind.LLTF(2),:);
    lltfDemod = wlanNonHTOFDMDemodulate(lltf,'L-LTF',cfg);
    nVar(i) = wlanLLTFNoiseEstimate(lltfDemod);

    % Measure signal power before noise addition
    lltf = rx(ind.LLTF(1):ind.LLTF(2),:);
    lltfDemod = wlanNonHTOFDMDemodulate(lltf,'L-LTF',cfg);
    signalPower(i) = mean(lltfDemod.*conj(lltfDemod),'all');
end

% Measure SNR
measuredSNR = signalPower./nVar;

% Verify that the measured SNR approximates the target SNR
disp(['Measured SNR ' num2str(10*log10(mean(measuredSNR))) ' dB'])
```

```
Measured SNR 10.0758 dB
```

References

1. IEEE Std 802.11 - 2020. Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks Specific Requirements.
2. Perahia, Eldad, and Robert Stacey. Next Generation Wireless LANS: 802.11n and 802.11ac. Cambridge University Press, 2013.

See Also

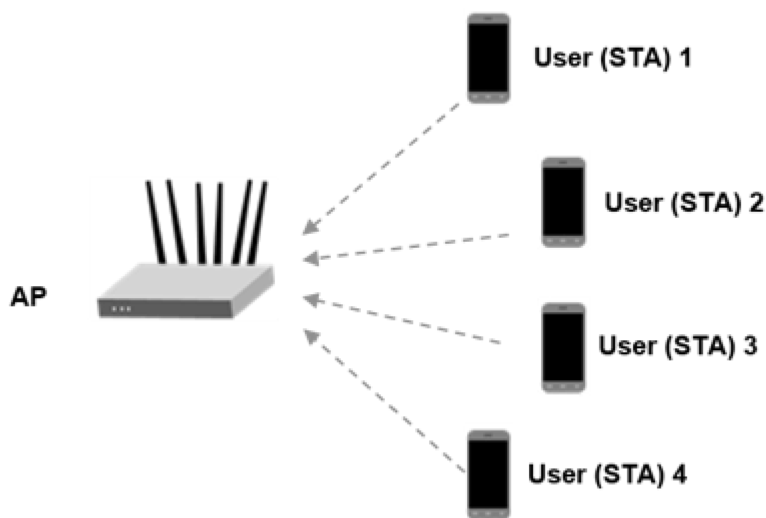
Functions
awgn

802.11be Packet Error Rate Simulation for Uplink Trigger-Based Format

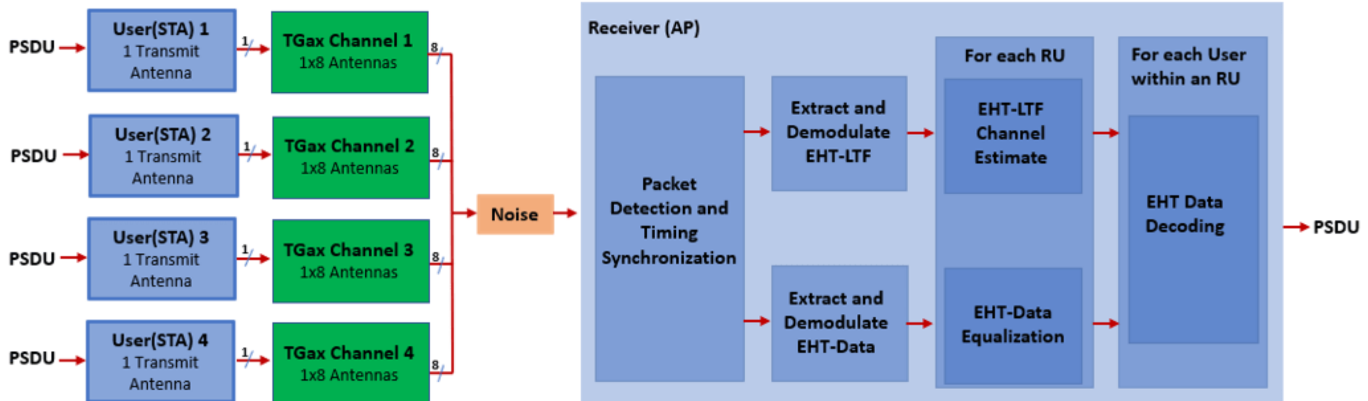
This example shows how to measure the packet error rate of an IEEE® 802.11be™ (Wi-Fi 7) extremely high-throughput (EHT) uplink, trigger-based (TB) format.

Introduction

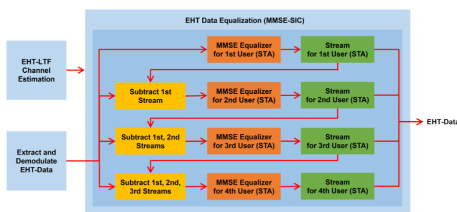
The 802.11be [1] EHT trigger-based (EHT TB) format allows for OFDMA or MU-MIMO transmission in the uplink. The AP controls the EHT TB transmission and provides the required parameters to all stations (STAs) participating in the transmission using a trigger frame. Each STA transmits an EHT TB packet simultaneously, when triggered by the AP as shown in the following diagram.



This example uses an end-to-end simulation to determine the packet error rate of an EHT TB link for four STAs in an MU-MIMO configuration. The transmitter sends multiple packets at each SNR point, with no impairments apart from channel and noise. The receiver demodulates the packets and recovers the PSDUs for each STA. The example compares the recovered PSDUs to the transmitted PSDUs to determine the number of packet errors, and hence, the packet error rate for all users. The receiver performs packet detection, timing synchronization, and symbol equalization. This example does not perform frequency offset correction. The EHT TB processing chain is shown in the following diagram.



The receiver performs a minimum-mean-square-error-based ordered successive interference cancellation (MMSE-SIC) process for data equalization [2]. To avoid error propagation in the cancellation stage, data streams for all STAs are sorted in descending order based on the channel state information and equalized sequentially. This diagram shows the MMSE-SIC equalization procedure.



Equalization Method

In this example, you can specify the equalization method as 'mmse' or 'mmse-sic'. The default equalizer is 'mmse-sic'.

```
equalizationMethod = 'mmse-sic';
```

User Configuration

This example configures the allocation and transmit parameters for multiple uplink STAs using an `ehtTBSystemConfig` object.

```
cfgSys = ehtTBSystemConfig('CBW20','NumUsers',4);
```

In a trigger-based transmission some parameters are the same for all uplink users, while some can differ. The `User` property of `cfgSys` contains a cell array of user configurations. Each element of the cell array is an object which sets the parameters of an individual user. In this example, all users have the same transmission parameters.

```
% These parameters are the same for all users in the MU-MIMO system
cfgSys.EHTLTFType = 4;       % EHT-LTF compression mode
cfgSys.GuardInterval = 3.2; % Guard interval type
numRx = 8;                   % Number of receive(AP) antennas

% The individual parameters for each user are specified below
```

```
allocInfo = ruInfo(cfgSys);
numUsers = allocInfo.NumUsers; % Number of uplink users

for userIdx = 1:numUsers
    cfgSys.User{userIdx}.NumTransmitAntennas = 1;
    cfgSys.User{userIdx}.NumSpaceTimeStreams = 1;
    cfgSys.User{userIdx}.SpatialMapping = 'Direct';
    cfgSys.User{userIdx}.MCS = 7;
    cfgSys.User{userIdx}.APEPLength = 1e3;
    cfgSys.User{userIdx}.ChannelCoding = 'LDPC';
end
```

Use a `wlanEHTTBConfig` object to configure a trigger-based transmission for a single user within the system. The method `userConfig` configures the transmission for all users. This example creates a cell array of four EHT TB objects which describes the transmission of four users.

```
cfgTB = userConfig(cfgSys);
```

Simulation Parameters

For each SNR point (dB) in the `snr` vector, the example generates a number of packets. These are passed through a channel and demodulated to determine the packet error rate.

```
snr = 20:2:24;

% The sample rate and field indices for the EHT TB packet are the same for
% all users. Use the trigger configuration of the first user to get the
% sample rate and field indices of the EHT TB PPDU.
fs = wlanSampleRate(cfgTB{1});    % Same for all users
ind = wlanFieldIndices(cfgTB{1}); % Same for all users
```

Channel Configuration

This example uses a TGax NLOS indoor channel model with delay profile Model-B. Model-B is in non-line of sight (NLOS) when the distance between the transmitter and receiver is greater than or equal to 5 meters. This is described further in `wlanTGaxChannel`. In this example all STAs are at the same distance from the AP.

```
tgaxBase = wlanTGaxChannel;
tgaxBase.SampleRate = fs;
tgaxBase.TransmissionDirection = 'Uplink';
tgaxBase.TransmitReceiveDistance = 10;
chanBW = cfgSys.ChannelBandwidth;
tgaxBase.ChannelBandwidth = chanBW;
tgaxBase.NumReceiveAntennas = numRx;
tgaxBase.NormalizeChannelOutputs = false;
```

The example creates individual channels for each of the four users. Each channel is a clone of `tgaxBase`, but with a different `UserIndex` property, and is stored in a cell array `tgax`. The `UserIndex` property of each individual channel is set to provide a unique channel for each user. This example uses a random channel realization for each packet by randomly varying the `UserIndex` property for each transmitted packet.

```
% A cell array stores the channel objects, one per user
tgax = cell(1,numUsers);
for userIdx = 1:numUsers
    tgax{userIdx} = clone(tgaxBase);
    tgax{userIdx}.NumTransmitAntennas = cfgSys.User{userIdx}.NumTransmitAntennas;
```

```
    tgax{userIdx}.UserIndex = userIdx;
end
```

Processing SNR Points

For each SNR point, the example tests a number of packets and calculates the packet error rate. The pre-EHT preamble of 802.11be is backwards compatible with 802.11ax™. Therefore, this example uses the timing synchronization components of an 802.11ax waveform to synchronize the EHT waveform at the receiver. For each user, the example follows these processing steps to create a waveform at the receiver containing all four users:

1. To create an EHT TB waveform, create and encode a PSDU for each user, based on predefined user parameters.
2. Pass the waveform for each user through an indoor TGax channel model. The randomly varying `UserIndex` property of the channel creates different channel realizations for each user. This results in different spatial correlation properties for each user.
3. Scale and combine the waveforms for all EHT TB users to ensure the same SNR for each user after the addition of noise.
4. Add AWGN to the received waveform. This is to create the desired average SNR per active subcarrier after OFDM demodulation.

The receiver (AP) follows these processing steps:

1. Detect the packet.
2. Perform fine timing synchronization. The L-STF, L-LTF and L-SIG samples provide fine timing to allow for packet detection at the start or end of the L-STF.
3. Extract EHT-LTF and EHT-Data fields for all users after synchronization of the received waveform. OFDM demodulate the EHT-LTF and EHT-Data fields.
4. Perform channel estimation on the demodulated EHT-LTF symbols for each RU.
5. Perform noise estimation using the demodulated data field pilots for each RU.
6. Extract and demodulate the data field and perform equalization for all users within an RU.
7. Recover PSDU bits for each RU and user within the RU by demodulating and decoding the spatial streams for a user.

You can use a parfor loop to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed, comment out the 'for' statement and uncomment the 'parfor' statement below.

```
ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfgTB{1});
numSNR = numel(snr); % Number of SNR points
numPackets = 50; % Number of packets to simulate
packetErrorRate = zeros(numUsers,numSNR);
txPSDU = cell(numUsers);

% parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
for isnr = 1:numSNR
    % Create EHT TB object for receiver processing
    cfgEHTTB = wlanEHTTBConfig;
    cfgEHTTB.ChannelBandwidth = cfgSys.ChannelBandwidth;
    cfgEHTTB.EHTLTFType = cfgSys.EHTLTFType;

    % 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);

% RU allocation information
sysInfo = ruInfo(cfgSys);

% Simulate multiple packets
numPacketErrors = zeros(numUsers,1);
for pktIdx = 1:numPackets

    % Transmit processing
    rxWaveform = 0;
    packetError = zeros(numUsers,1);
    txPSDU = cell(1,numUsers);

    % Generate random channel realization for each packet by varying
    % the UserIndex property of the channel. This assumes all users
    % have the same number of transmit antennas.
    chPermutations = randperm(numUsers);
    for userIdx = 1:numUsers
        % EHT TB config object for each user
        cfgUser = cfgTB{userIdx};

        % Generate a packet with random PSDU
        txPSDU{userIdx} = randi([0 1],psduLength(cfgUser)*8,1,'int8');

        % Generate EHT TB waveform, containing payload for single user
        txTrig = wlanWaveformGenerator(txPSDU{userIdx},cfgUser);

        % Pass waveform through a random TGax Channel
        channelIdx = chPermutations(userIdx);
        reset(tgax{channelIdx}); % New channel realization
        rxTrig = tgax{channelIdx}([txTrig; zeros(15,size(txTrig,2))]);

        % Scale the transmit power of the user within an RU. This is to
        % ensure the same SNR for each user after the addition of noise.
        ruNum = cfgSys.User{userIdx}.RUNumber;
        SF = sqrt(1/sysInfo.NumUsersPerRU(ruNum))*sqrt(sum(cfgUser.RUSize)/(sum(cell2mat(sys

        % Combine uplink users into one waveform
        rxWaveform = rxWaveform+SF*rxTrig;
    end

    % Pass the waveform through AWGN channel. Account for noise energy
    % in nulls so the SNR is defined per active subcarriers.
    packetSNR = snr(isnr)-10*log10(ofdmInfo.FFTLength/(sum(cell2mat(sysInfo.RUSizes))));
    rxWaveform = awgn(rxWaveform,packetSNR);

    % Receive processing
    % Packet detect and determine coarse packet offset
    coarsePktOffset = wlanPacketDetect(rxWaveform,chanBW);
    if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
        numPacketErrors = numPacketErrors+1;
        continue; % Go to next loop iteration
    end
```

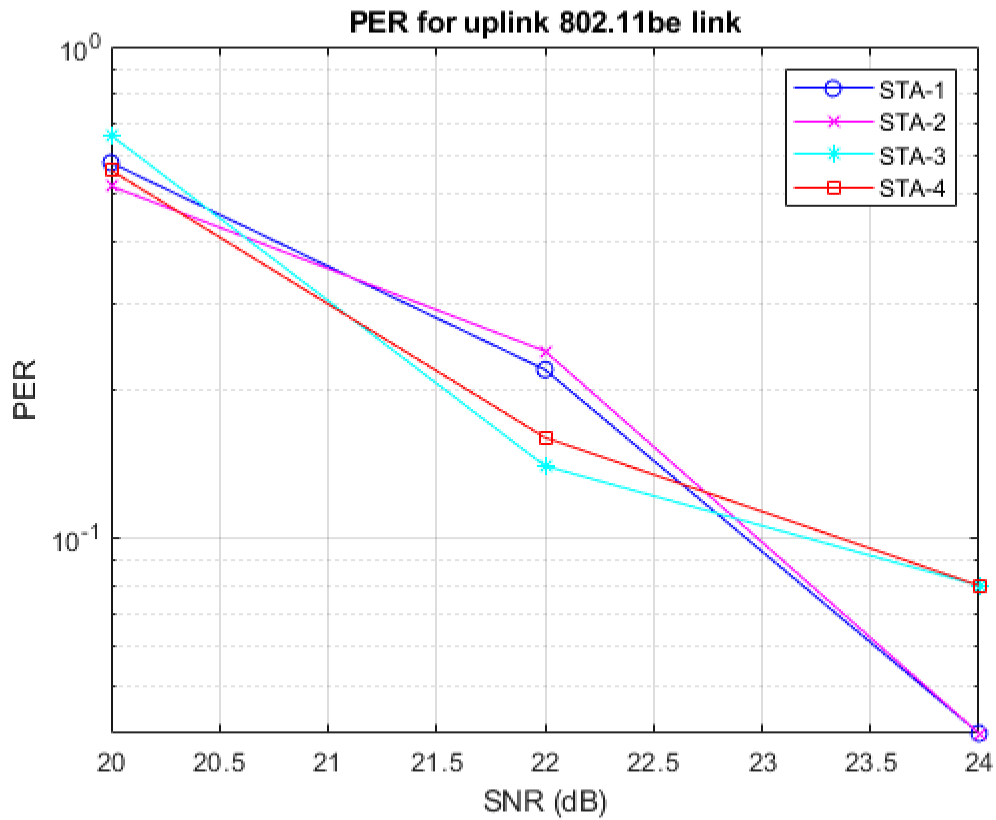
```
% Extract the non-HT fields and determine fine packet offset
nonhtfields = rxWaveform(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

% Determine final packet offset
pktOffset = coarsePktOffset+finePktOffset;

% If packet detected out with the range of expected delays from
% the channel modeling; packet error
if pktOffset>50
    numPacketErrors = numPacketErrors+1;
    continue; % Go to next loop iteration
end

% Extract EHT-LTF and EHT-Data fields for all RUs
rxLTF = rxWaveform(pktOffset+(ind.EHTLTF(1):ind.EHTLTF(2)),:);
rxData = rxWaveform(pktOffset+(ind.EHTData(1):ind.EHTData(2)),:);

for ruIdx = 1:allocInfo.NumRUs
    userNumber = cfgSys.RU{ruIdx}.UserNumbers; % Same for all users
    cfgUserRef = cfgTB{userNumber(1)}; % Reference user configuration

    % Demodulate EHT-LTF and EHT-Data field
    demodLTFRU = wlanEHTDemodulate(rxLTF,'EHT-LTF',cfgUserRef);
    demodDataRU = wlanEHTDemodulate(rxData,'EHT-Data',cfgUserRef);

    % Configure the relevant properties in EHT TB object
    cfgEHTTB.RUSize = allocInfo.RUSizes{ruIdx};
    cfgEHTTB.RUIndex = allocInfo.RUIndices{ruIdx};
    cfgEHTTB.NumSpaceTimeStreams = allocInfo.NumSpaceTimeStreamsPerRU(ruIdx);

    % Channel estimate
    [chanEst,ssPilotEst] = wlanEHTLTFChannelEstimate(demodLTFRU,cfgEHTTB);

    % Get indices of data and pilots within RU (without nulls)
    ruOFDMInfo = wlanEHTOFDMInfo('EHT-Data',cfgUserRef);

    % Estimate noise power in EHT fields of each user
    nVarEst = ehtNoiseEstimate(demodDataRU(ruOFDMInfo.PilotIndices,:,:),ssPilotEst,cfgUse

    % Discard pilot subcarriers
    demodDataSym = demodDataRU(ruOFDMInfo.DataIndices,:,:);
    chanEstData = chanEst(ruOFDMInfo.DataIndices,:,:);

    % Equalize
    if strcmpi(equalizationMethod,'mmse-sic')
        [eqSym,csi] = heSuccessiveEqualize(demodDataSym,chanEstData,nVarEst,cfgSys,ruIdx
    else
        [eqSym,csi] = ehtEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgTB{userNumbe
    end

    for userIdx = 1:allocInfo.NumUsersPerRU(ruIdx)
        % Get TB config object for each user
        userNum = cfgSys.RU{ruIdx}.UserNumbers(userIdx); % User within an RU
        cfgUserTB = cfgTB{userNum};

        % Get space-time stream indices for the current user
        stsIdx = cfgUserTB.StartingSpaceTimeStream-1+(1:cfgUserTB.NumSpaceTimeStreams);
```

```
                % Demap and decode bits
                rxPSDU = wlanEHTDataBitRecover(eqSym(:,:,stsIdx),nVarEst,csi(:,stsIdx),cfgUserTB

                % PER calculation
                packetError(userNum) = any(biterr(txPSDU{userNum},rxPSDU));
            end
        end
        numPacketErrors = numPacketErrors+packetError;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(:,isnr)= numPacketErrors/numPackets;
    disp(['SNR ' num2str(snr(isnr)) ...
          ' completed for ' num2str(numUsers) ' users']);

end
```

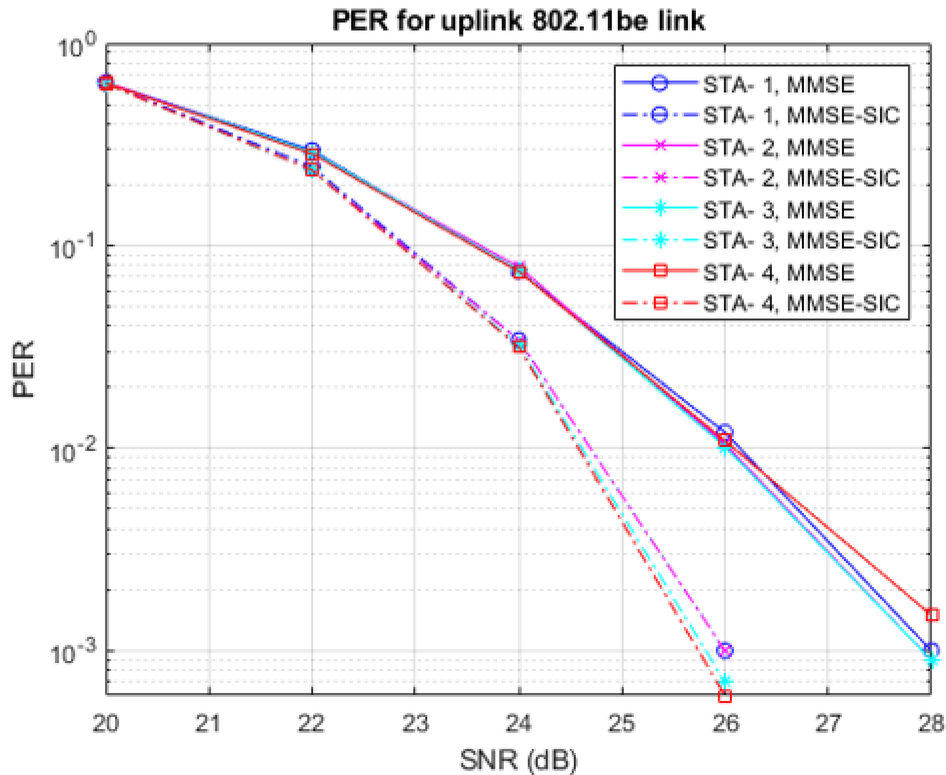
```
SNR 20 completed for 4 users
SNR 22 completed for 4 users
SNR 24 completed for 4 users
```

Plot Packet Error Rate vs SNR

```
markers = 'ox*sd^v><ph+ox*sd^v';
color = 'bmcrgbrkymcrgbrkymc';
figure;

for nSTA = 1:numUsers
    semilogy(snr,packetErrorRate(nSTA,:).',['-' markers(nSTA) color(nSTA)]);
    hold on;
end

grid on;
xlabel('SNR (dB)');
ylabel('PER');
dataStr = arrayfun(@(x)sprintf('STA-%d',x),1:numUsers,'UniformOutput',false);
legend(dataStr);
title('PER for uplink 802.11be link');
```



The `numPackets` controls the number of packets at each SNR point. For meaningful results, this value should be larger than those presented in this example. The figure below was created by running a longer simulation with `numPackets`:1e4 and `snr`:20:2:28, which shows the packet error rate of both MMSE equalizer and MMSE-SIC equalizer.



Selected Bibliography

1 IEEE P802.11be™/D2.0. 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 8: Enhancements for extremely high throughput (EHT).

2 M. Debbah, B. Muquet, M. de Courville, M. Muck, S. Simoens, and P. Loubaton. A MMSE successive interference cancellation scheme for a new adjustable hybrid spread OFDM system. IEEE 51st Vehicular Technology Conference Proceedings, pp. 745-749, vol. 2, 2000.

See Also

Related Examples

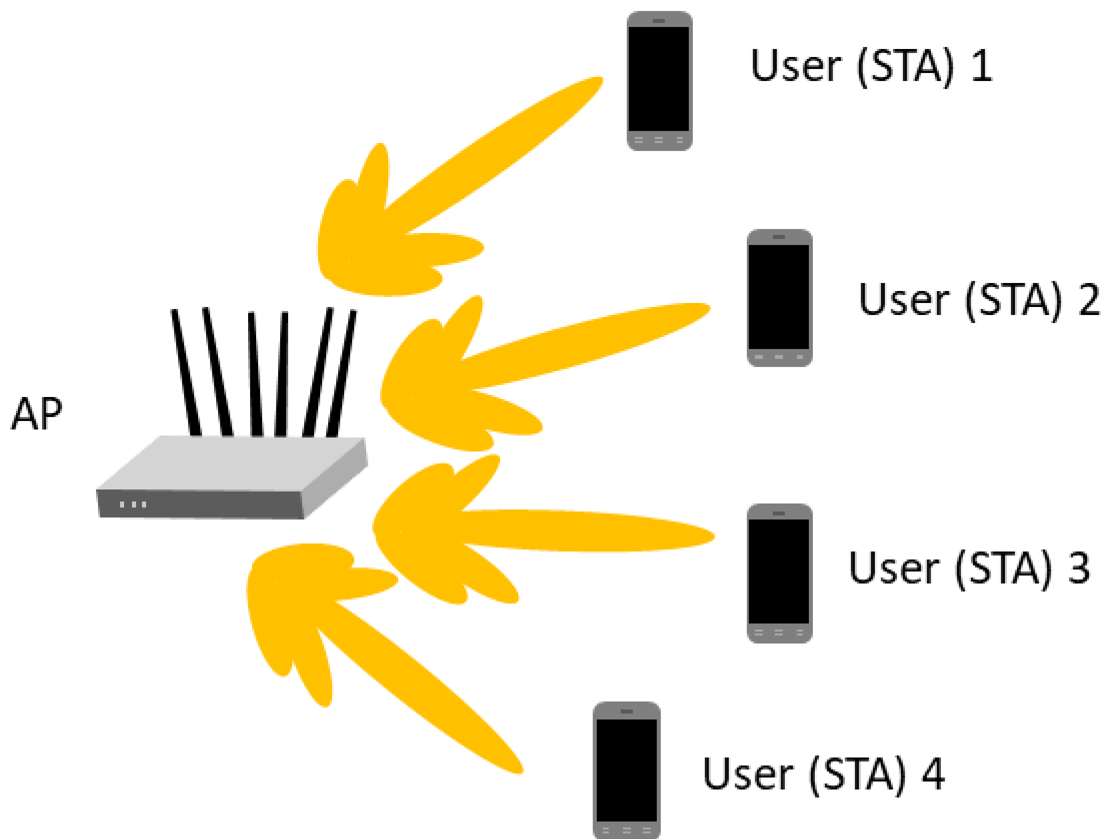
- “802.11be Packet Error Rate Simulation for an EHT MU Single-User Packet Format” on page 6-37
- “802.11be Waveform Generation” on page 1-18

802.11be Downlink Multi-User MIMO and OFDMA Throughput Simulation

This example shows transmit and receive processing for an IEEE® 802.11be™ (Wi-Fi 7) extremely high throughput (EHT) multi-user (MU) downlink transmission over a TGax indoor fading channel. It simulates two PPDU types: MU-MIMO and OFDMA.

Introduction

This example simulates an access point (AP) transmitting to four stations (STAs) simultaneously using the EHT MU packet format as specified in [1 on page 6-36].



You can configure the EHT MU format for MU-MIMO transmission or OFDMA transmission. This flexibility allows an EHT MU packet to transmit to a single user over the whole band, or multiple users over different parts of the band (OFDMA). In this example, two transmission PPDU types are simulated for the multi-user downlink scenario:

1. MU-MIMO transmission - all four users share the full band.
2. OFDMA transmission - two users share a large-size multiple resource unit (MRU), and the remaining two users are assigned a single resource unit (RU) each.

For a detailed overview of 802.11be PPDU types, see the “802.11be Waveform Generation” on page 1-18 example.

For each PPDU type, the AP transmits a burst of 10 packets, and each STA demodulates and decodes the data intended for it. An evolving TGax indoor MIMO channel with AWGN is modeled between the AP and each STA. The raw AP throughput is calculated by counting the number of packets transmitted successfully to all STAs. The simulation repeats for different path losses. In this example, all the transmissions are beamformed. Therefore, before simulating the data transmission, the channel between the AP and each station is sounded under perfect conditions to obtain the channel state information (CSI).

Transmission Configuration

Use a `wlanEHTMUConfig` object to configure the transmission of an EHT MU packet. Specify two transmission configuration objects to define the two different AP transmissions:

1. `cfgMUMIMO` is an 80 MHz MU-MIMO configuration that consists of a single 996-tone RU with four users. Each user has one space-time stream.
2. `cfgOFDMA` is an 80 MHz OFDMA configuration that consists of a 484+282-tone MRU with two users, a 106-tone and a 106+26-tone RU, each with one user. Each user has two space-time streams.

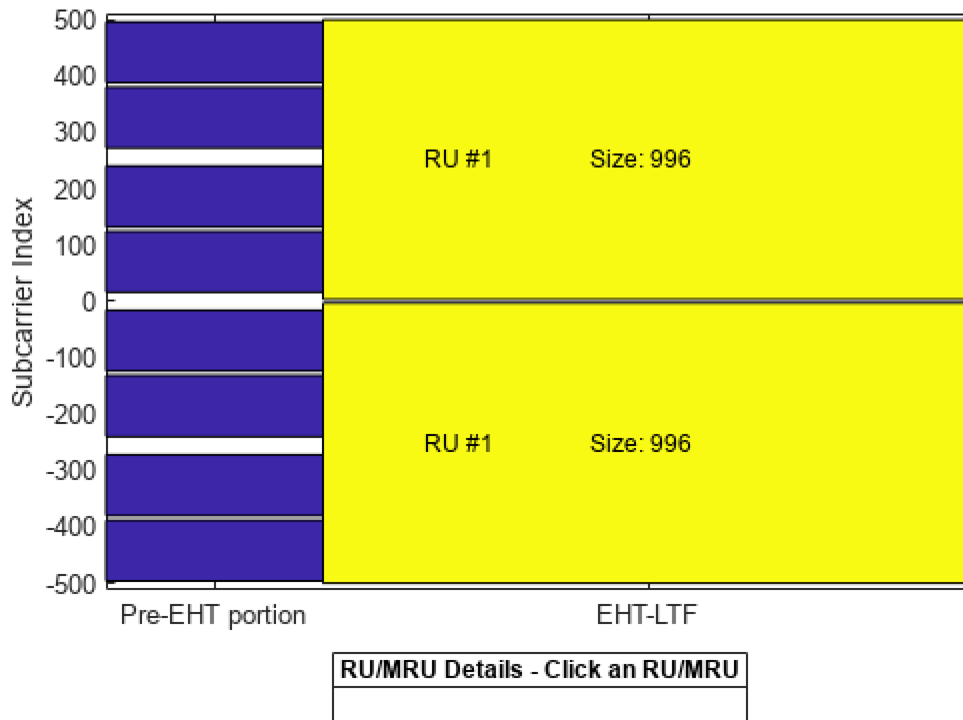
The other transmission parameters, such as the guard interval, EHT-LTF type, APEPLength, and MCS, are the same for all users in all configurations.

First, define an 80 MHz bandwidth and a single 996-tone RU with four users for the MU-MIMO configuration. For a description of multiple users in an MU-MIMO configuration over the full channel bandwidth, see the “802.11be Waveform Generation” on page 1-18 example.

```
% MU-MIMO configuration - 4 users on one 996-tone RU
cfgMUMIMO = wlanEHTMUConfig('CBW80','NumUsers',4);
```

The allocation plot shows a single RU assigned to all four users.

```
showAllocation(cfgMUMIMO);
```



Configure the transmission parameters for each user.

```
numTx = 6; % Number of transmit antennas
guardInterval = 3.2; % Guard interval in microseconds
ltfType = 4; % EHT-LTF type
apepLength = 1e3; % APEP length in bytes
mcs = 4; % MCS

% Configure common parameters for all users
cfgMUMIMO.NumTransmitAntennas = numTx;
cfgMUMIMO.GuardInterval = guardInterval;
cfgMUMIMO.EHTLTFType = ltfType;

% Configure per user parameters
% STA #1
cfgMUMIMO.User{1}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{1}.MCS = mcs;
cfgMUMIMO.User{1}.APEPLength = apepLength;
% STA #2
cfgMUMIMO.User{2}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{2}.MCS = mcs;
cfgMUMIMO.User{2}.APEPLength = apepLength;
% STA #3
cfgMUMIMO.User{3}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{3}.MCS = mcs;
cfgMUMIMO.User{3}.APEPLength = apepLength;
% STA #4
```

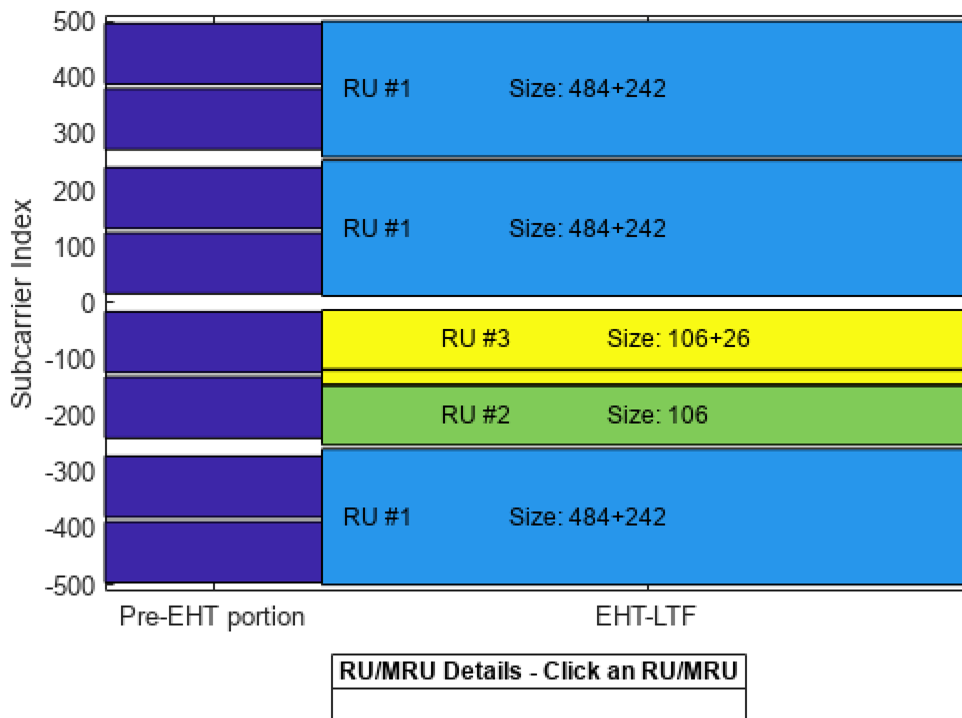
```
cfgMUMIMO.User{4}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{4}.MCS = mcs;
cfgMUMIMO.User{4}.APEPLength = apepLength;
```

Next, define the OFDMA configuration. The allocation index `[105 50 29 29]` defines a 484+242-tone MRU with two users as well as a 106+26-tone and 106-tone RU, each serving a single user.

```
% OFDMA configuration - 4 users, two on a 484+242-tone MRU, one on a
% 106+26-tone RU, and one on 106-tone RU
cfgOFDMA = wlanEHTMUConfig([105 50 29 29]);
```

The allocation plot shows the MRU with two users and two RUs, each with a single user. When comparing this allocation plot to the full band MU-MIMO plot, you can see that the total number of subcarriers used (484+242+106+26+106 = 964 subcarriers) is less than the MU-MIMO allocation (996 subcarriers). The smaller number of subcarriers allows guards between MRU and single RU users.

```
showAllocation(cfgOFDMA);
```



Configure the transmission parameters for each user.

```
% Configure common parameters for all users
cfgOFDMA.NumTransmitAntennas = numTx;
cfgOFDMA.GuardInterval = guardInterval;
cfgOFDMA.EHTLTFType = ltfType;

% Configure per user parameters
```

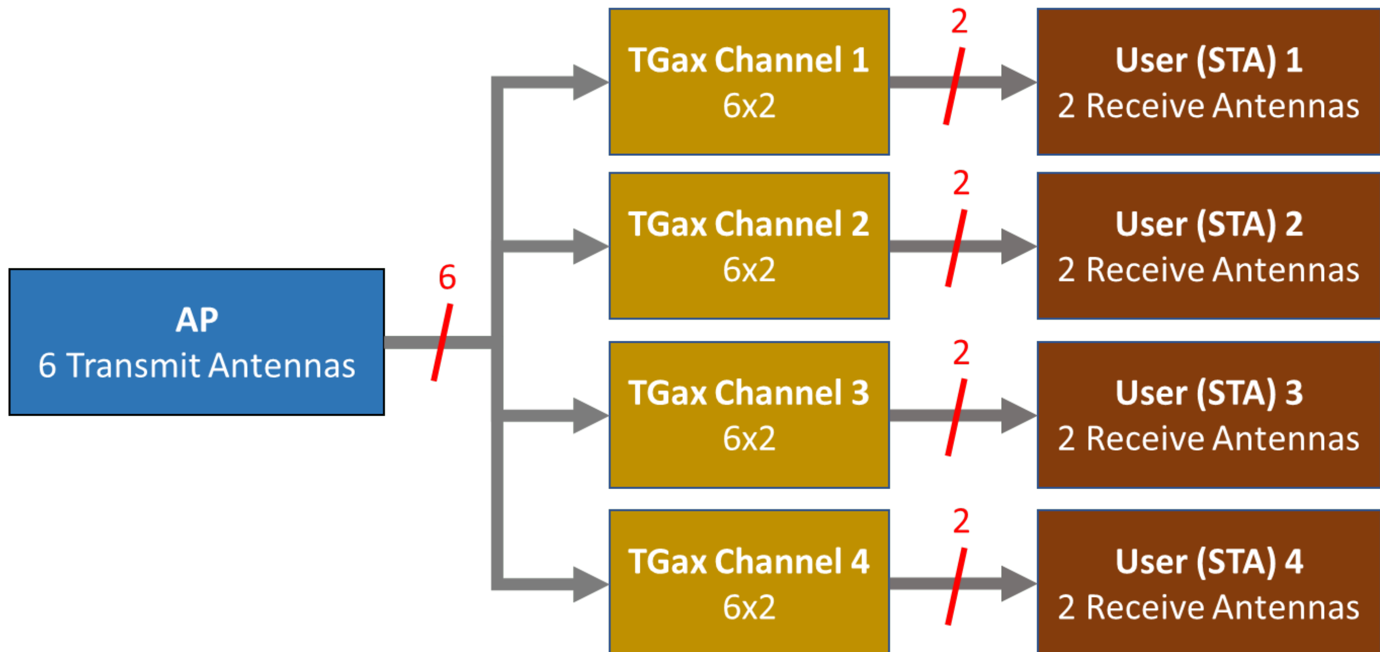
```
% STA #1 (RU #1)
cfgOFDMA.User{1}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{1}.MCS = mcs;
cfgOFDMA.User{1}.APEPLength = apepLength;
% STA #2 (RU #1)
cfgOFDMA.User{2}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{2}.MCS = mcs;
cfgOFDMA.User{2}.APEPLength = apepLength;
% STA #3 (RU #2)
cfgOFDMA.User{3}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{3}.MCS = mcs;
cfgOFDMA.User{3}.APEPLength = apepLength;
% STA #4 (RU #3)
cfgOFDMA.User{4}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{4}.MCS = mcs;
cfgOFDMA.User{4}.APEPLength = apepLength;
```

Channel Model Configuration

A TGax indoor channel model is used in this example. An individual channel is used to simulate the link between the AP and each user. Create a TGax channel object, `tgaxBase`, with properties relevant for all users. In this example, the delay profile (Model-B) and the number of receive antennas are common for all users. When the distance between transmitter and receiver is greater than or equal to 5 meters, Model-B is considered non-line of sight. This is described further in `wlanTGaxChannel`. Use a fixed seed for the channel to allow repeatability.

```
% Create channel configuration common for all users
tgaxBase = wlanTGaxChannel;
tgaxBase.DelayProfile = 'Model-B';     % Delay profile
tgaxBase.NumTransmitAntennas = numTx;  % Number of transmit antennas
tgaxBase.NumReceiveAntennas = 2;       % Each user has two receive antennas
tgaxBase.TransmitReceiveDistance = 5;  % Non-line of sight distance
tgaxBase.ChannelBandwidth = cfgMUMIMO.ChannelBandwidth;
tgaxBase.SampleRate = wlanSampleRate(cfgMUMIMO);
% Set a fixed seed for the channel
tgaxBase.RandomStream = 'mt19937ar with seed';
tgaxBase.Seed = 5;
```

Next, create a channel for each user. The channel for each user is a clone of `tgaxBase`, with a unique `UserIndex` property, stored in a cell array `tgax`. Set the `UserIndex` property of each individual channel to provide a unique channel for each user. The resultant channels are shown below.




```
% A cell array stores the channel objects, one per user
numUsers = numel(cfgMUMIMO.User); % Number of users simulated in this example
tgax = cell(1,numUsers);

% Generate per-user channels
for userIdx = 1:numUsers
    tgax{userIdx} = clone(tgaxBase);
    tgax{userIdx}.UserIndex = userIdx; % Set unique user index
end
```

Beamforming Feedback

Transmit beamforming for both MU-MIMO and OFDMA relies on knowledge of the channel state between transmitter and receiver at the beamformer. To set up the channel for beamforming in this example, each STA provides feedback of the per-subcarrier channel state using channel sounding. The AP transmits a null data packet (NDP), and each STA uses this packet to determine the channel state. They then feed the channel state back to the AP. The same process is used for the “802.11ax Downlink OFDMA and Multi-User MIMO Throughput Simulation” on page 6-51 example. The `ehtUserBeamformingFeedback` helper function detects the NDP and uses channel estimation to determine the CSI. Singular value decomposition (SVD) is then used to calculate the beamforming feedback.

```
% Create an NDP with the correct number of space-time streams to generate
% enough LTF symbols
cfgNDP = wlanEHTMUConfig(tgaxBase.ChannelBandwidth);
cfgNDP.GuardInterval = guardInterval;
cfgNDP.EHTLTFType = ltfType;
cfgNDP.NumTransmitAntennas = cfgMUMIMO.NumTransmitAntennas;
cfgNDP.User{1}.APEPLength = 0; % No data in an NDP
cfgNDP.User{1}.NumSpaceTimeStreams = cfgMUMIMO.NumTransmitAntennas;

% Generate NDP packet - with an empty PSDU as no data
txNDP = wlanWaveformGenerator([],cfgNDP);
```

```
% For each user STA, pass the NDP packet through the channel and calculate
% the feedback channel state matrix by SVD.
staFeedback = cell(1,numUsers);
for userIdx = 1:numel(tgax)
    % Received waveform at user STA with 50 sample padding. No noise.
    rx = tgax{userIdx}([txNDP; zeros(50,size(txNDP,2))]);

    % Get the full-band beamforming feedback for a user
    staFeedback{userIdx} = ehtUserBeamformingFeedback(rx,cfgNDP);
end
```

Simulation Settings

This example simulates different path losses. To set up for path loss simulation, apply the same path loss and noise floor to all users. For each path loss specify 10 packets to pass through the channel. Separate the packets by 20 microseconds.

```
cfgSim = struct;
cfgSim.NumPackets = 10;       % Number of packets to simulate for each path loss
cfgSim.Pathloss = (97:2:103); % Path loss to simulate in dB
cfgSim.TransmitPower = 30;    % AP transmit power in dBm
cfgSim.NoiseFloor = -89.9;    % STA noise floor in dBm
cfgSim.IdleTime = 20;         % Idle time between packets in µs
```

MU-MIMO Simulation

For this exercise, you first simulate the MU-MIMO configuration. To calculate the beamforming matrix for an RU given the CSI feedback for all users in the MU-MIMO allocation, use the `ehtMUCalculateSteeringMatrix` helper function. Use a zero forcing solution to calculate the steering matrix within the helper function.

```
% Calculate the steering matrix to apply to the RU given the feedback
ruIdx = 1; % Index of the one and only RU
steeringMatrix = ehtMUCalculateSteeringMatrix(staFeedback,cfgMUMIMO,cfgNDP,ruIdx);

% Apply the steering matrix to the RU
cfgMUMIMO.RU{1}.SpatialMapping = 'Custom';
cfgMUMIMO.RU{1}.SpatialMappingMatrix = steeringMatrix;
```

The `ehtMUSimulateScenario` helper function performs the simulation. The pre-EHT preamble of 802.11be is backwards compatible with 802.11ac. Therefore, in this example, the front-end synchronization components for a VHT waveform are used to synchronize the EHT waveform at each STA. For each packet and path loss simulated, the following process occurs:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through an evolving TGax channel model and AWGN is added to the received waveform. The channel state is maintained between packets.
3. The packet is detected.
4. Coarse carrier frequency offset is estimated and corrected.
5. Fine timing synchronization is established.
6. Fine carrier frequency offset is estimated and corrected.
7. The EHT-LTF is extracted from the synchronized received waveform. The EHT-LTF is OFDM demodulated and channel estimation is performed.

8. The EHT-Data field is extracted from the synchronized received waveform and OFDM demodulated.
9. Common pilot phase tracking is performed to track any residual carrier frequency offset.
10. The phase corrected OFDM symbols are equalized with the channel estimate.
11. Noise estimation is performed using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers.
12. The equalized symbols are demodulated and decoded to recover the PSDU.
13. The recovered PSDU is compared to the transmitted PSDU to determine if the packet has been recovered successfully.

Run the simulation for the MU-MIMO configuration.

```
disp('Simulating MU-MIMO...');
```

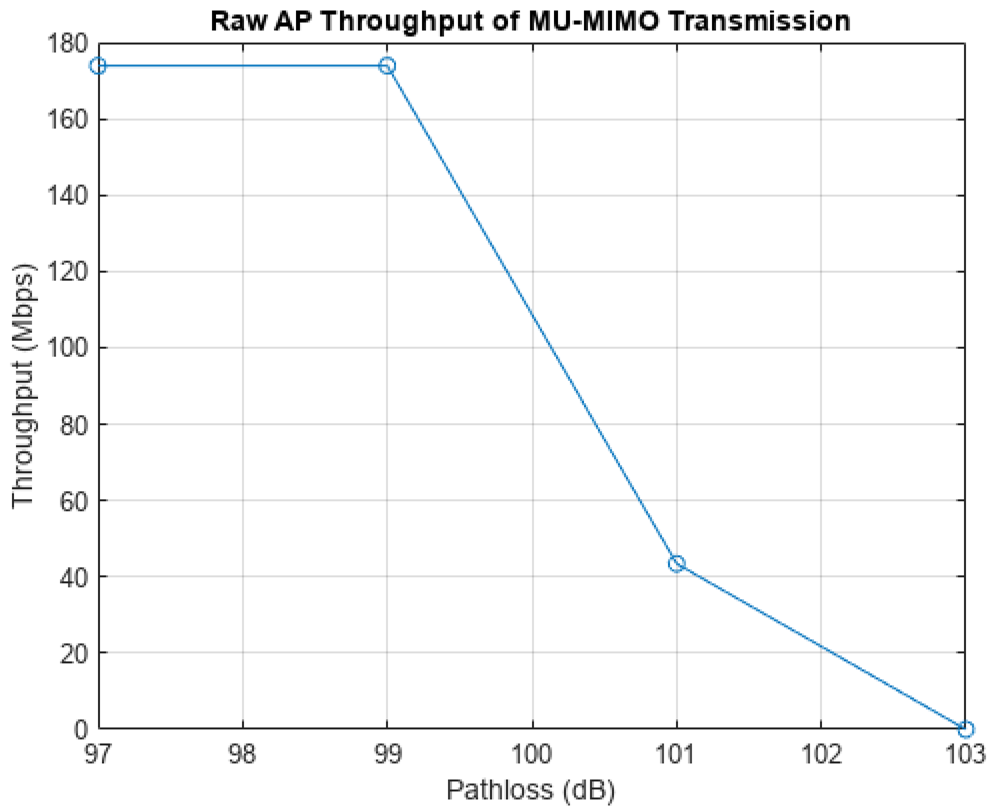
```
Simulating MU-MIMO...
```

```
throughputMUMIMO = ehtMUSimulateScenario(cfgMUMIMO,tgax,cfgSim);
```

```
 Pathloss 97.0 dB, AP throughput 173.9 Mbps
 Pathloss 99.0 dB, AP throughput 173.9 Mbps
 Pathloss 101.0 dB, AP throughput 43.5 Mbps
 Pathloss 103.0 dB, AP throughput 0.0 Mbps
```

Plot the raw AP throughput of the MU-MIMO simulation. The throughput accounts for the packet duration including the idle time, the APEP length of each user, and the number of successfully decoded packets. The results show for this channel realization at high SNRs (low path loss) that the throughput provided by the MU-MIMO configuration is considerable. However, the throughput drops dramatically when the noise dominates (low SNRs).

```
% Sum throughput for all STAs and plot for all configurations
figure;
plot(cfgSim.Pathloss,sum(throughputMUMIMO,2),'-o');
grid on;
xlabel('Pathloss (dB)');
ylabel('Throughput (Mbps)');
title('Raw AP Throughput of MU-MIMO Transmission');
```



OFDMA Simulation

Now simulate the OFDMA configuration and transmit beamforming.

Calculate the steering matrix for each RU by using the feedback from the STAs. To calculate the beamforming matrix for an RU given the CSI feedback, use the `ehtMUCalculateSteeringMatrix` helper function..

```
% For each RU, calculate the steering matrix to apply
for ruIdx = 1:numel(cfgOFDMA.RU)
    % Calculate the steering matrix to apply to the RU given the feedback
    steeringMatrix = ehtMUCalculateSteeringMatrix(staFeedback,cfgOFDMA,cfgNDP,ruIdx);

    % Apply the steering matrix to each RU
    cfgOFDMA.RU{ruIdx}.SpatialMapping = 'Custom';
    cfgOFDMA.RU{ruIdx}.SpatialMappingMatrix = steeringMatrix;
end
```

Run the simulation for the OFDMA configuration.

```
disp('Simulating OFDMA...');
```

```
Simulating OFDMA...
```

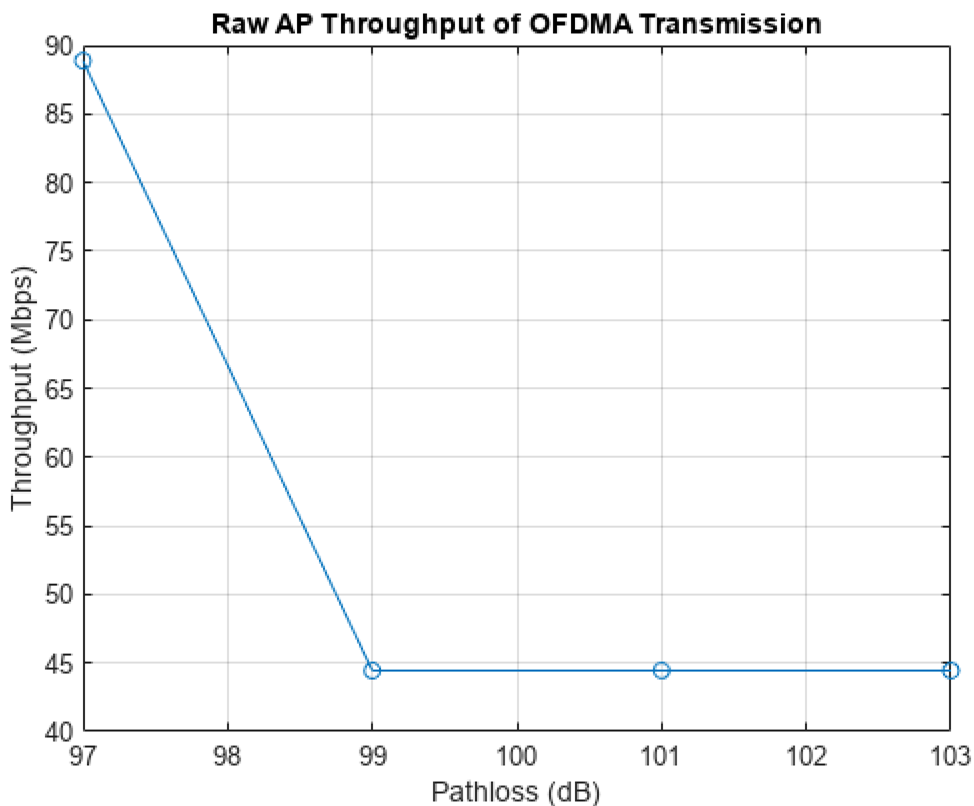
```
throughputOFDMA = ehtMUSimulateScenario(cfgOFDMA,tgax,cfgSim);
```

```
 Pathloss 97.0 dB, AP throughput 88.9 Mbps
 Pathloss 99.0 dB, AP throughput 44.4 Mbps
```

```
Pathloss 101.0 dB, AP throughput 44.4 Mbps
Pathloss 103.0 dB, AP throughput 44.4 Mbps
```

Plot the raw AP throughput of the OFDMA simulation. The results show that, as the path loss increases, several packets fail to be decoded for some users and the overall raw throughput drops.

```
% Sum throughput for all STAs and plot for all configurations
figure;
plot(cfgSim.Pathloss,sum(throughputOFDMA,2),'-o');
grid on;
xlabel('Pathloss (dB)');
ylabel('Throughput (Mbps)');
title('Raw AP Throughput of OFDMA Transmission');
```



Conclusion

This example shows how to simulate the downlink throughput for MU-MIMO and OFDMA configurations of 802.11be waveforms. Because throughput depends on many factors that differ between the two cases, such as RU size and the number of space-time streams, comparing the downlink throughput of MU-MIMO and OFDMA configurations is challenging.

In this example, for the OFDMA configuration, the RU sizes of User 3 and User 4 are much smaller than those of User 1 and User 2. The APEP length is the same for all users. Therefore, the OFDMA packet duration is limited by the smallest RU size, resulting in a long packet duration and the throughput is limited.

References

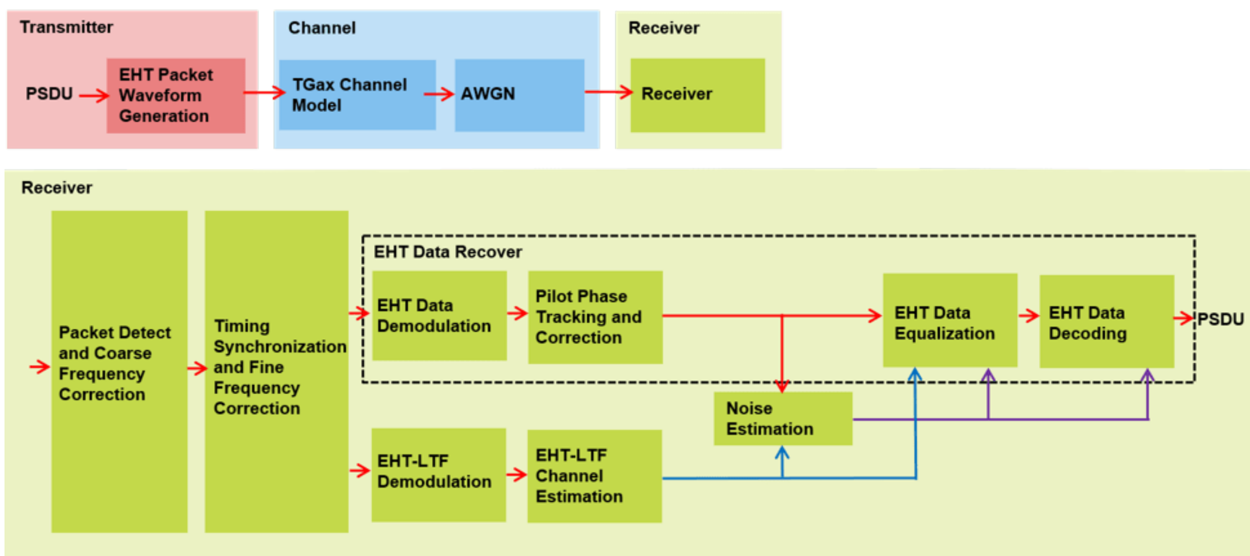
1 IEEE Std 802.11be™/D2.0 Draft Standard for Information technology - Telecommunications and information exchange between systems Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 8: Enhancements for Extremely High Throughput (EHT).

802.11be Packet Error Rate Simulation for an EHT MU Single-User Packet Format

This example shows how to measure the packet error rate of an IEEE® 802.11be™ Extremely High Throughput multi-user (EHT MU) packet format link with a single user.

Introduction

This example determines the packet error rate for an 802.11be [1] single-user (SU) link by using an end-to-end simulation for a selection of signal-to-noise ratio (SNR) points. At each SNR point, the example simulates the transmission of multiple packets through a noisy TGax indoor channel, then demodulates the received packets and recovers the PSDUs. The example then compares the transmitted and received packets to determine the packet error rate. This diagram shows the processing steps for each packet.



Waveform Configuration

An EHT MU SU packet is a full-band transmission to a single user. Configure the transmission parameters for an SU packet format by using the `wlanEHTMUConfig` object. The properties of the object contain the physical layer (PHY) configuration.

Create a configuration object for an EHT MU transmission, setting a channel bandwidth of 20 MHz, an APEP length of 1000 bytes, two transmit antennas, two space-time streams, and a modulation and coding scheme (MCS) value of 13, which specifies 4096-point quadrature amplitude modulation (4096-QAM) and a coding rate of 5/6. If you specify `mcs` as a vector, the example performs the simulation for each MCS index value.

```
chanBW = 'CBW20';                           % Channel bandwidth
cfgEHT = wlanEHTMUConfig(chanBW);
cfgEHT.User{1}.APEPLength = 1e3;            % APEP length (bytes)
numTx = 2;                                  % Number of transmit antennas
numRx = 2;                                  % Number of receive antennas
cfgEHT.NumTransmitAntennas = numTx;
```

```
cfgEHT.User{1}.NumSpaceTimeStreams = numTx; % Number of space-time streams
mcs = 13;                                   % MCS index
```

Channel Configuration

This example uses a TGax non-line-of-sight (NLOS) indoor channel model with delay profile Model-B. Model-B is considered NLOS when the distance between transmitter and receiver is greater than or equal to 5 meters. For more information about the TGax channel model, see `wlanTGaxChannel`.

```
% Create and configure a 2x2 MIMO channel.
tgaxChannel = wlanTGaxChannel;
tgaxChannel.DelayProfile = 'Model-B';
tgaxChannel.NumTransmitAntennas = cfgEHT.NumTransmitAntennas;
tgaxChannel.NumReceiveAntennas = numRx;
tgaxChannel.TransmitReceiveDistance = 5; % Distance in meters for NLOS
tgaxChannel.ChannelBandwidth = chanBW;
tgaxChannel.LargeScaleFadingEffect = 'None';
fs = wlanSampleRate(chanBW);
tgaxChannel.SampleRate = fs;
```

Simulation Parameters

For each SNR point in `snrRange`, the example generates the specified number of packets, passes the packets through a channel, then demodulates the received signal to determine the packet error rate. Set the SNR values in the `snrRange` parameter to simulate the transition from all packets being decoded in error to all packets being decoded successfully as the SNR value increases for MCS 13. If you specify `snrRange` as a matrix, each row represents the SNR points for the corresponding MCS index, defined in `mcs`.

```
snrRange = 37:5:57; % Set the range of SNR values
```

These parameters control the number of packets tested for each SNR point.

1. `maxNumErrors`: the maximum number of packet errors simulated for each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets`: the maximum number of packets simulated for each SNR point, which limits the length of the simulation if the simulation does not reach the packet error limit.

The default parameter values lead to a very short simulation. For meaningful results, increase these values.

```
maxNumErrors = 10;
maxNumPackets = 100;
```

Processing SNR Points

This section measures the packet error rate for each SNR point by performing these processing steps for the specified number of packets.

1. Create a PSDU and encode to generate a single-packet waveform.
2. Pass the waveform through an indoor TGax channel model, using different channel realizations for each packet.
3. Add AWGN to the received waveform to create the desired average SNR per subcarrier after OFDM demodulation. The configuration accounts for the normalization within the channel by the number of receive antennas and the noise energy in unused subcarriers. The example removes the unused subcarriers during OFDM demodulation.

4. Detect the packet
5. Estimate and correct coarse carrier frequency offset (CFO)
6. Perform fine timing synchronization by using L-STF, L-LTF, and L-SIG samples. This synchronization enables packet detection at the start or end of the L-STF.
7. Estimate and correct fine CFO
8. Extract the EHT-LTF from the synchronized received waveform
9. OFDM demodulate the EHT-LTF and perform channel estimation
10. Extract the data field from the synchronized received waveform and perform OFDM demodulation
11. Track any residual CFO by performing common phase error pilot tracking
12. Perform noise estimation by using the demodulated data field pilots and single-stream channel estimation at pilot subcarriers
13. Equalize the phase corrected OFDM symbols by using channel estimation
14. Recover the PSDU by demodulating and decoding the equalized symbols

This example also demonstrates how to speed up simulations by using a `parfor` loop instead of a `for` loop when simulating each SNR point. The `parfor` function executes processing for each SNR in parallel to reduce the total simulation time. Use a `parfor` loop to parallelize processing of the SNR points. To use parallel computing for increased speed, comment out the `for` statement and uncomment the `parfor` statement in this code.

```
numSNR = size(snrRange,2); % Number of SNR points
numMCS = numel(mcs); % Number of MCS
packetErrorRate = zeros(numMCS,numSNR);

for imcs = 1:numel(mcs)
    cfgEHT.User{1}.MCS = mcs(imcs);
    ofdmInfo = wlanEHTOFDMInfo('EHT-Data',cfgEHT);
    % SNR points to simulate from MCS
    snr = snrRange(imcs,:);
    ind = wlanFieldIndices(cfgEHT);

    %parfor isnr = 1:numSNR % Use parfor to speed up the simulation
    for isnr = 1:numSNR % 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',99);
        stream.Substream = isnr;
        RandStream.setGlobalStream(stream);

        % Define the SNR per active subcarrier to account for noise energy
        % in nulls
        snrValue = snr(isnr)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

        % Loop to simulate multiple packets
        numPacketErrors = 0;
        numPkt = 1; % Index of packet transmitted
        while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
            % Generate waveform
            txPSDU = randi([0 1],psduLength(cfgEHT)*8,1); % PSDULength (bytes)
            tx = wlanWaveformGenerator(txPSDU,cfgEHT);
```

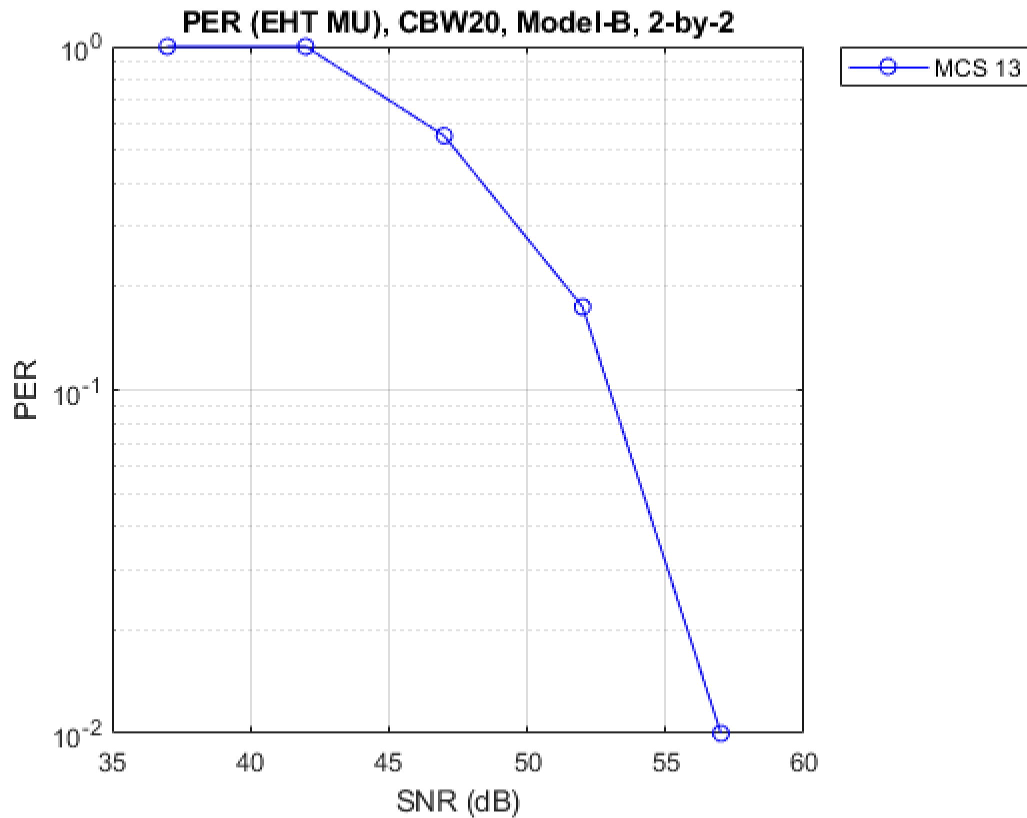
```
% Add trailing zeros to allow for channel delay
txPad = [tx; zeros(50,cfgEHT.NumTransmitAntennas)];

% Pass through fading indoor TGax channel
reset(tgaxChannel); % Reset channel for different realization
rx = tgaxChannel(txPad);

% Pass waveform through an AWGN channel
rx = awgn(rx,snrValue);

% Detect packet and determine coarse packet offset
coarsePktOffset = wlanPacketDetect(rx,chanBW);
if isempty(coarsePktOffset) % If empty no L-STF detected, packet error
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-STF and perform coarse frequency offset correction
lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
coarseFreqOff = wlanCoarseCFOEstimate(lstf,chanBW);
rx = frequencyOffset(rx,fs,-coarseFreqOff);

% Extract the non-HT fields and determine fine packet offset
nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

% Determine final packet offset
pktOffset = coarsePktOffset+finePktOffset;

% If packet detected outwith range of expected delays from
% the channel modeling, packet error
if pktOffset>50
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-LTF and perform fine frequency offset correction
rxLLTF = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
fineFreqOff = wlanFineCFOEstimate(rxLLTF,chanBW);
rx = frequencyOffset(rx,fs,-fineFreqOff);

% EHT-LTF demodulation and channel estimation
rxHELTF = rx(pktOffset+(ind.EHTLTF(1):ind.EHTLTF(2)),:);
heltfDemod = wlanEHTDemodulate(rxHELTF,'EHT-LTF',cfgEHT);
[chanEst,pilotEst] = wlanEHTLTFChannelEstimate(heltfDemod,cfgEHT);

% Demodulate the Data field
rxData = rx(pktOffset+(ind.EHTData(1):ind.EHTData(2)),:);
demodSym = wlanEHTDemodulate(rxData,'EHT-Data',cfgEHT);

% Perform pilot phase tracking
demodSym = wlanEHTTrackPilotError(demodSym,chanEst,cfgEHT,'EHT-Data');

% Estimate noise power in EHT fields
nVarEst = ehtNoiseEstimate(demodSym(ofdmInfo.PilotIndices,:,:),pilotEst,cfgEHT);
```

```
            % Extract data subcarriers from demodulated symbols and channel
            % estimate
            demodDataSym = demodSym(ofdmInfo.DataIndices,:,:);
            chanEstData = chanEst(ofdmInfo.DataIndices,:,:);

            % Equalization
            [eqSym,csi] = ehtEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgEHT);

            % Recover data field bits
            rxPSDU = wlanEHTDataBitRecover(eqSym,nVarEst,csi,cfgEHT);

            % Determine if any bits are in error
            packetError = any(biterr(txPSDU,rxPSDU));
            numPacketErrors = numPacketErrors+packetError;
            numPkt = numPkt+1;
        end

        % Calculate PER at SNR point
        packetErrorRate(imcs,isnr) = numPacketErrors/(numPkt-1);
        disp(['MCS ' num2str(mcs(imcs)) ','...
              ' SNR ' num2str(snr(isnr)) ...
              ' completed after ' num2str(numPkt-1) ' packets,'...
              ' PER:' num2str(packetErrorRate(imcs,isnr))]);
    end
end
```

```
MCS 13, SNR 37 completed after 11 packets, PER:1
MCS 13, SNR 42 completed after 11 packets, PER:1
MCS 13, SNR 47 completed after 20 packets, PER:0.55
MCS 13, SNR 52 completed after 63 packets, PER:0.1746
MCS 13, SNR 57 completed after 100 packets, PER:0.01
```

Plot Packet Error Rate vs SNR

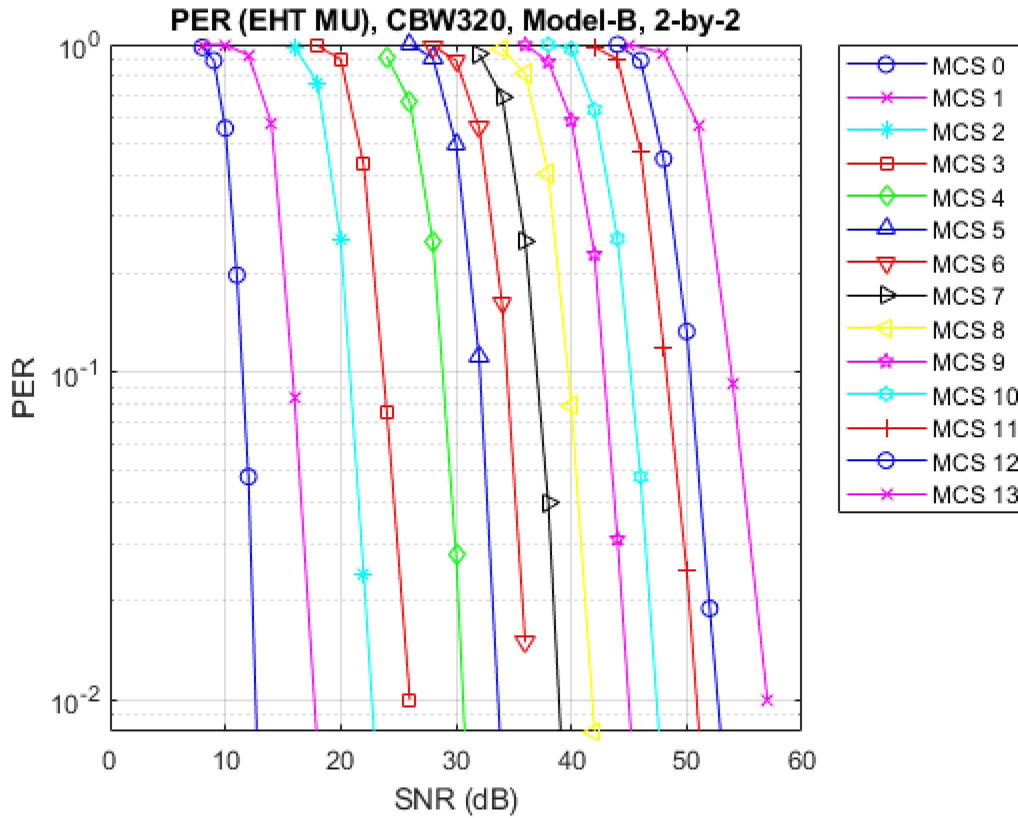
```
markers = 'ox*sd^v><ph+ox*sd^v><ph+';
color = 'bmcrgbrkymcrbmcrgbrkymcr';
figure;
for imcs = 1:numMCS
    semilogy(snrRange(imcs,:),packetErrorRate(imcs,:).',['-' markers(imcs) color(imcs)]);
    hold on;
end
grid on;
xlabel('SNR (dB)');
ylabel('PER');
dataStr = arrayfun(@(x)sprintf('MCS %d',x),mcs,'UniformOutput',false);
legend(dataStr,'Location','NorthEastOutside');
title(['PER (EHT MU), ' num2str(cfgEHT.ChannelBandwidth) ', Model-B, ' num2str(numTx) '-by-' num2
```



Further Exploration

The `maxNumErrors` and `maxNumPackets` parameters control the number of packets tested for each SNR point. For meaningful results, increase these values. For example, this figure shows results for a channel bandwidth of 320 MHz, an APEP length of 16000 bytes, MCS values of 0-13, a `maxNumErrors` value of 100, and a `maxNumPackets` value of 1000. The corresponding SNR values for MCS between 0 and 13 are:

```
snrRange = [...
     8:1:13; ... % MCS 0
     8:2:18; ... % MCS 1
     16:2:26; ... % MCS 2
     18:2:28; ... % MCS 3
     24:2:34; ... % MCS 4
     26:2:36; ... % MCS 5
     28:2:38; ... % MCS 6
     32:2:42; ... % MCS 7
     34:2:44; ... % MCS 8
     36:2:46; ... % MCS 9
     38:2:48; ... % MCS 10
     42:2:52; ... % MCS 11
     44:2:54; ... % MCS 12
     45:3:60]; ...% MCS 13
```



Selected Bibliography

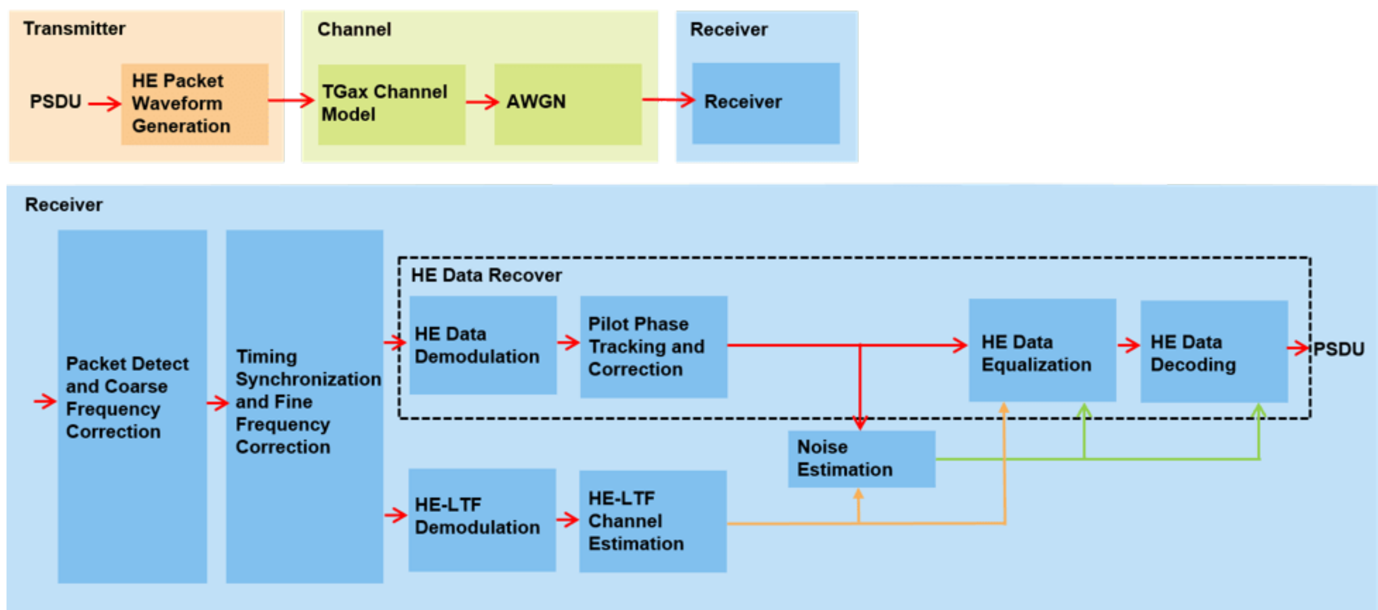
1 IEEE Std 802.11be™/D2.0 Draft Standard for Information technology - Telecommunications and information exchange between systems Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 8: Enhancements for Extremely High Throughput (EHT).

802.11ax Packet Error Rate Simulation for Single-User Format

This example shows how to measure the packet error rate of an IEEE® 802.11ax™ (Wi-Fi 6) high efficiency (HE) single user format link.

Introduction

In this example, an end-to-end simulation is used to determine the packet error rate for an 802.11ax [1] single user format link for a selection of SNR points. At each SNR point, multiple packets are transmitted through a noisy TGax indoor channel, demodulated and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the packet error rate. The processing for each packet is summarized in the following diagram.



Waveform Configuration

An HE single user (SU) packet is a full-band transmission to a single user. The transmit parameters for the HE SU format are configured using an `wlanHESUConfig` object. The properties of the object contain the configuration. In this example, the object is configured for a 20 MHz channel bandwidth, 2 transmit antennas, 2 space-time streams, no space time block coding and 16-QAM rate-1/2 (MCS 3).

```
cfgHE = wlanHESUConfig;
cfgHE.ChannelBandwidth = 'CBW20';  % Channel bandwidth
cfgHE.NumSpaceTimeStreams = 2;     % Number of space-time streams
cfgHE.NumTransmitAntennas = 2;     % Number of transmit antennas
cfgHE.APEPLength = 1e3;            % Payload length in bytes
cfgHE.ExtendedRange = false;       % Do not use extended range format
cfgHE.Upper106ToneRU = false;      % Do not use upper 106 tone RU
cfgHE.PreHESpatialMapping = false; % Spatial mapping of pre-HE fields
cfgHE.GuardInterval = 0.8;         % Guard interval duration
cfgHE.HELTFType = 4;               % HE-LTF compression mode
cfgHE.ChannelCoding = 'LDPC';      % Channel coding
cfgHE.MCS = 3;                     % Modulation and coding scheme
```

Channel Configuration

In this example, a TGax NLOS indoor channel model is used with delay profile Model-B. Model-B is considered NLOS when the distance between transmitter and receiver is greater than or equal to 5 meters. This is described further in `wlanTGaxChannel`. A 2x2 MIMO channel is simulated in this example.

```
% Create and configure the TGax channel
chanBW = cfgHE.ChannelBandwidth;
tgaxChannel = wlanTGaxChannel;
tgaxChannel.DelayProfile = 'Model-B';
tgaxChannel.NumTransmitAntennas = cfgHE.NumTransmitAntennas;
tgaxChannel.NumReceiveAntennas = 2;
tgaxChannel.TransmitReceiveDistance = 5; % Distance in meters for NLOS
tgaxChannel.ChannelBandwidth = chanBW;
tgaxChannel.LargeScaleFadingEffect = 'None';
tgaxChannel.NormalizeChannelOutputs = false;
fs = wlanSampleRate(cfgHE);
tgaxChannel.SampleRate = fs;
```

Simulation Parameters

For each SNR point (dB) in the `snr` vector a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 10:5:35;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For statistically meaningful results we recommend increasing these numbers.

```
maxNumErrors = 10;   % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated. The pre-HE preamble of 802.11ax is backwards compatible with 802.11ac™, therefore in this example the front-end synchronization components for a VHT waveform are used to synchronize the HE waveform at the receiver. For each packet the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through an indoor TGax channel model. Different channel realizations are modeled for different packets.
3. AWGN is added to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.
4. The packet is detected.
5. Coarse carrier frequency offset is estimated and corrected.

6. Fine timing synchronization is established. The L-STF, L-LTF and L-SIG samples are provided for fine timing to allow for packet detection at the start or end of the L-STF.
7. Fine carrier frequency offset is estimated and corrected.
8. The HE-LTF is extracted from the synchronized received waveform. The HE-LTF is OFDM demodulated and channel estimation is performed.
9. The data field is extracted from the synchronized received waveform and OFDM demodulated.
10. Common phase error pilot tracking is performed to track any residual carrier frequency offset.
11. Noise estimation is performed using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers.
12. The phase corrected OFDM symbols are equalized with the channel estimate.
13. The equalized symbols are demodulated and decoded to recover the PSDU.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

```
numSNR = numel(snr); % Number of SNR points
packetErrorRate = zeros(1,numSNR);

% Get occupied subcarrier indices and OFDM parameters
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgHE);

% Indices to extract fields from the PPDU
ind = wlanFieldIndices(cfgHE);

%parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
for isnr = 1:numSNR
    % Set random substream index per iteration to ensure that each
    % iteration uses a repeatable set of random numbers
    stream = RandStream('combRecursive','Seed',99);
    stream.Substream = isnr;
    RandStream.setGlobalStream(stream);

    % Account for noise energy in nulls so the SNR is defined per
    % active subcarrier
    packetSNR = snr(isnr)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

    % Loop to simulate multiple packets
    numPacketErrors = 0;
    numPkt = 1; % Index of packet transmitted
    while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
        % Generate a packet with random PSDU
        psduLength = getPSDULength(cfgHE); % PSDU length in bytes
        txPSDU = randi([0 1],psduLength*8,1);
        tx = wlanWaveformGenerator(txPSDU,cfgHE);

        % Add trailing zeros to allow for channel delay
        txPad = [tx; zeros(50,cfgHE.NumTransmitAntennas)];

        % Pass through a fading indoor TGax channel
        reset(tgaxChannel); % Reset channel for different realization
        rx = tgaxChannel(txPad);

        % Pass the waveform through AWGN channel
```

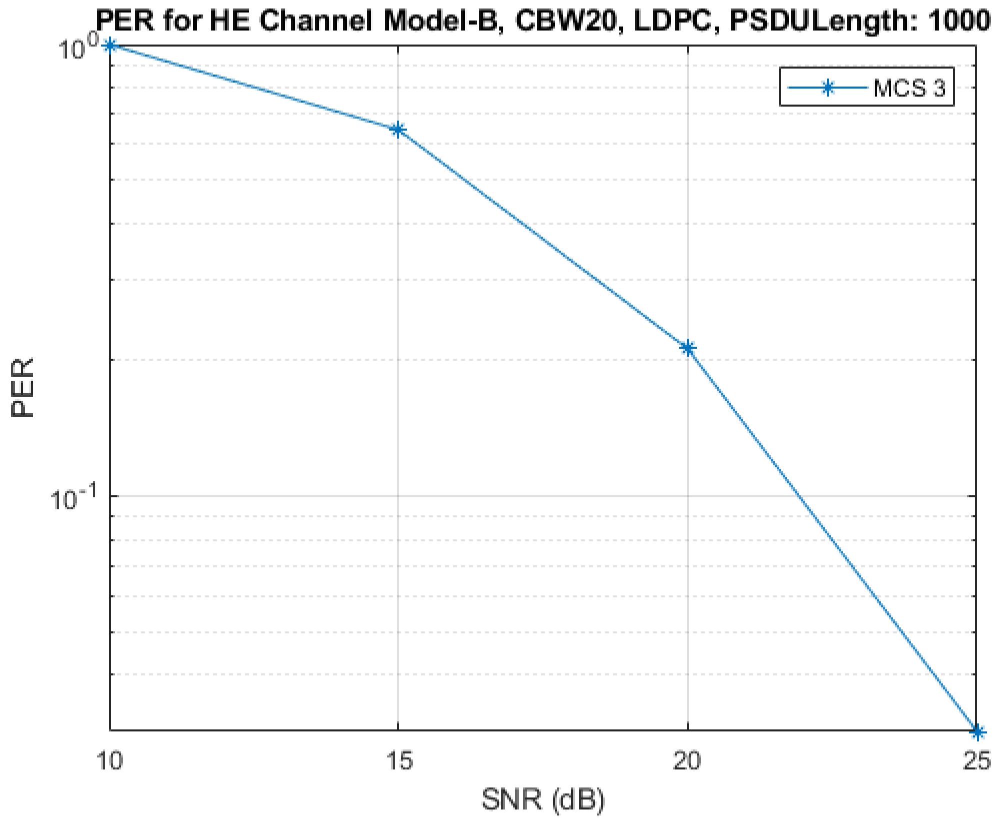
```
rx = awgn(rx,packetSNR);

% Packet detect and determine coarse packet offset
coarsePktOffset = wlanPacketDetect(rx,chanBW);
if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-STF and perform coarse frequency offset correction
lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
coarseFreqOff = wlanCoarseCFOEstimate(lstf,chanBW);
rx = frequencyOffset(rx,fs,-coarseFreqOff);

% Extract the non-HT fields and determine fine packet offset
nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

% Determine final packet offset
pktOffset = coarsePktOffset+finePktOffset;

% If packet detected outwith the range of expected delays from
% the channel modeling; packet error
if pktOffset>50
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-LTF and perform fine frequency offset correction
rxLLTF = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
fineFreqOff = wlanFineCFOEstimate(rxLLTF,chanBW);
rx = frequencyOffset(rx,fs,-fineFreqOff);

% HE-LTF demodulation and channel estimation
rxHELTF = rx(pktOffset+(ind.HELTF(1):ind.HELTF(2)),:);
heltfDemod = wlanHEDemodulate(rxHELTF,'HE-LTF',cfgHE);
[chanEst,pilotEst] = wlanHELTFChannelEstimate(heltfDemod,cfgHE);

% Data demodulate
rxData = rx(pktOffset+(ind.HEData(1):ind.HEData(2)),:);
demodSym = wlanHEDemodulate(rxData,'HE-Data',cfgHE);

% Pilot phase tracking
demodSym = wlanHETrackPilotError(demodSym,chanEst,cfgHE,'HE-Data');

% Estimate noise power in HE fields
nVarEst = heNoiseEstimate(demodSym(ofdmInfo.PilotIndices,:,:),pilotEst,cfgHE);

% Extract data subcarriers from demodulated symbols and channel
% estimate
demodDataSym = demodSym(ofdmInfo.DataIndices,:,:);
chanEstData = chanEst(ofdmInfo.DataIndices,:,:);

% Equalization and STBC combining
[eqDataSym,csi] = heEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgHE);
```

```
        % Recover data
        rxPSDU = wlanHEDataBitRecover(eqDataSym,nVarEst,csi,cfgHE,'LDPCDecodingMethod','norm-min

        % Determine if any bits are in error, i.e. a packet error
        packetError = ~isequal(txPSDU,rxPSDU);
        numPacketErrors = numPacketErrors+packetError;
        numPkt = numPkt+1;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(isnr) = numPacketErrors/(numPkt-1);
    disp(['MCS ' num2str(cfgHE.MCS) ','...
          ' SNR ' num2str(snr(isnr)) ...
          ' completed after ' num2str(numPkt-1) ' packets,'...
          ' PER:' num2str(packetErrorRate(isnr))]);
end
```

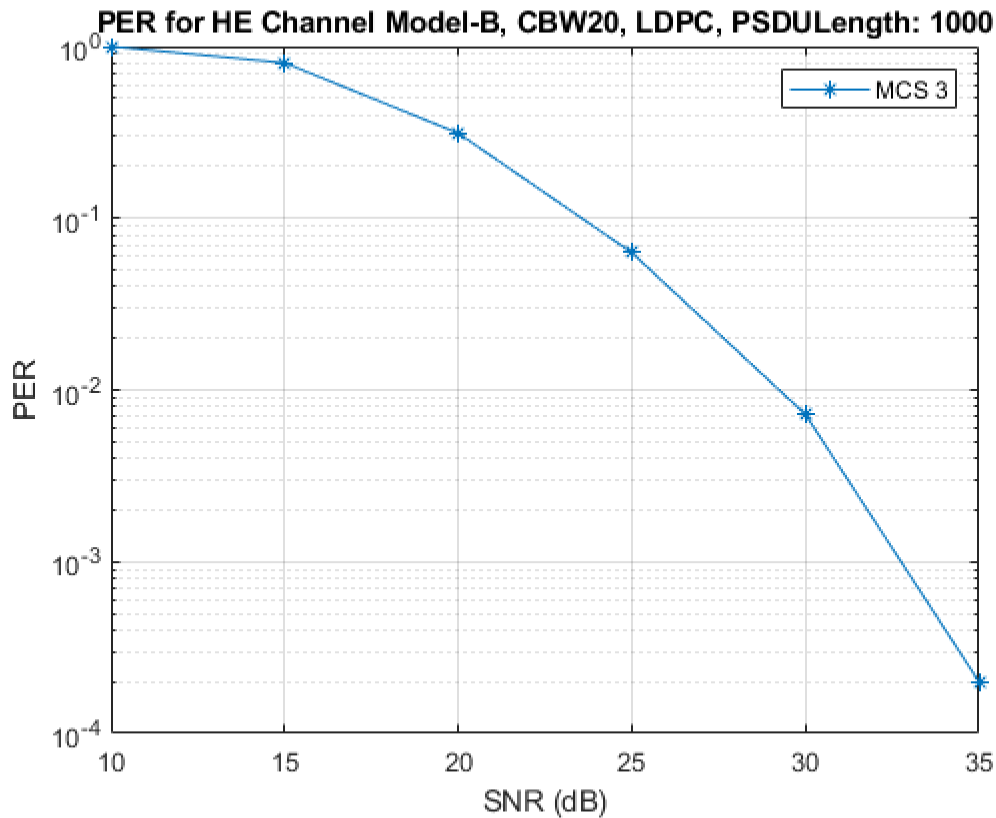
```
MCS 3, SNR 10 completed after 11 packets, PER:1
MCS 3, SNR 15 completed after 17 packets, PER:0.64706
MCS 3, SNR 20 completed after 52 packets, PER:0.21154
MCS 3, SNR 25 completed after 100 packets, PER:0.03
MCS 3, SNR 30 completed after 100 packets, PER:0
MCS 3, SNR 35 completed after 100 packets, PER:0
```

Plot Packet Error Rate vs SNR

```
figure;
semilogy(snr,packetErrorRate,'-*');
hold on;
grid on;
xlabel('SNR (dB)');
ylabel('PER');
dataStr = arrayfun(@(x)sprintf('MCS %d',x),cfgHE.MCS,'UniformOutput',false);
legend(dataStr);
title(sprintf('PER for HE Channel %s, %s, %s, PSDULength: %d',tgaxChannel.DelayProfile,cfgHE.Char
```



The number of packets tested at each SNR point is controlled by two parameters: `maxNumErrors` and `maxNumPackets`. For meaningful results, these values should be larger than those presented in this example. As an example, the figure below was created by running a longer simulation with `maxNumErrors`:1e3 and `maxNumPackets`:1e4.



Appendix

This example uses the following helper functions:

- heEqualizeCombine.m
- heNoiseEstimate.m

Selected Bibliography

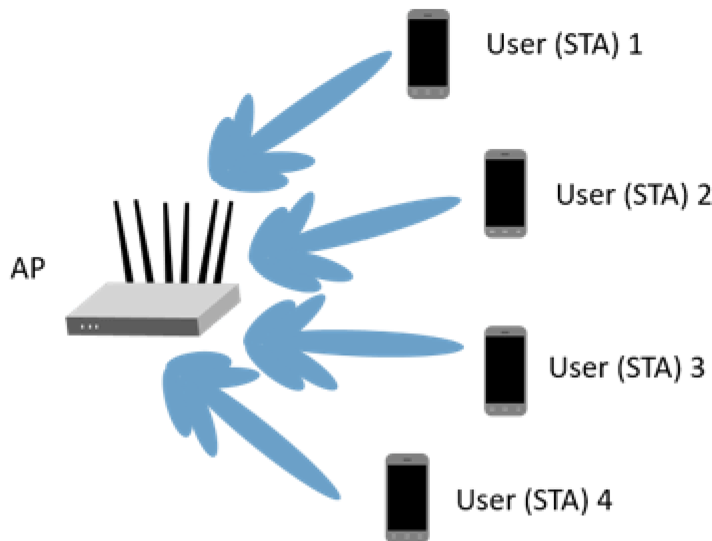
1 IEEE Std 802.11ax™-2021. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 1: Enhancements for High-Efficiency WLAN.

802.11ax Downlink OFDMA and Multi-User MIMO Throughput Simulation

This example shows the transmit and receive processing for an IEEE® 802.11ax™ multi-user downlink transmission over a TGax indoor fading channel. Three transmission modes are simulated: OFDMA, MU-MIMO, and a combination of OFDMA and MU-MIMO.

Introduction

This example simulates a scenario of an access point (AP) transmitting to four stations (STAs) simultaneously using high efficiency (HE) multi-user (MU) format packets as specified in IEEE Std 802.11ax™-2021 [1].



The HE multi-user format can be configured for an OFDMA transmission, a MU-MIMO transmission, or a combination of the two. This flexibility allows an HE-MU packet to transmit to a single user over the whole band, multiple users over different parts of the band (OFDMA), or multiple users over the same part of the band (MU-MIMO).

Three transmission modes are compared for the multi-user downlink scenario:

1. OFDMA - each of the four users is assigned a separate resource unit (RU) and the transmission is beamformed.
2. MU-MIMO - all four users share the full band.
3. Mixed MU-MIMO and OFDMA - Two users share a single RU in a MU-MIMO configuration, and the remaining two users are assigned a single RU each.

For a detailed overview of 802.11ax formats, see the “802.11ax Waveform Generation” on page 1-64 example.

For each transmission mode, the AP transmits a burst of 10 packets, and each STA demodulates and decodes the data intended for it. An evolving TGax indoor MIMO channel with AWGN is modeled between the AP and each STA. The raw AP throughput is provided as a metric to compare the transmission modes and is calculated by counting the number of packets transmitted successfully to

all STAs. The simulation is repeated for different path losses. In this example, all the transmissions are beamformed. Therefore, before simulating the data transmission, the channel between the AP and each station is sounded under perfect conditions to obtain channel state information.

Transmission Configuration

An `wlanHEMUConfig` object is used to configure the transmission of an HE-MU packet. Three transmission configuration objects are specified to define the three different AP transmissions:

1. `cfgMUMIMO` is a MU-MIMO configuration which consists of a single 242-tone RU with 4 users. Each user has one space-time stream.
2. `cfgOFDMA` is an OFDMA configuration which consists of four 52-tone RUs, each with one user. Each user has two space-time streams.
3. `cfgMixed` is a mixed OFDMA and MU-MIMO configuration which consists of one 106-tone RU shared by two users, and two 52-tone RUs, each with one user. The MU-MIMO users each have one space-time stream, and the OFDMA users each have two space-time streams.

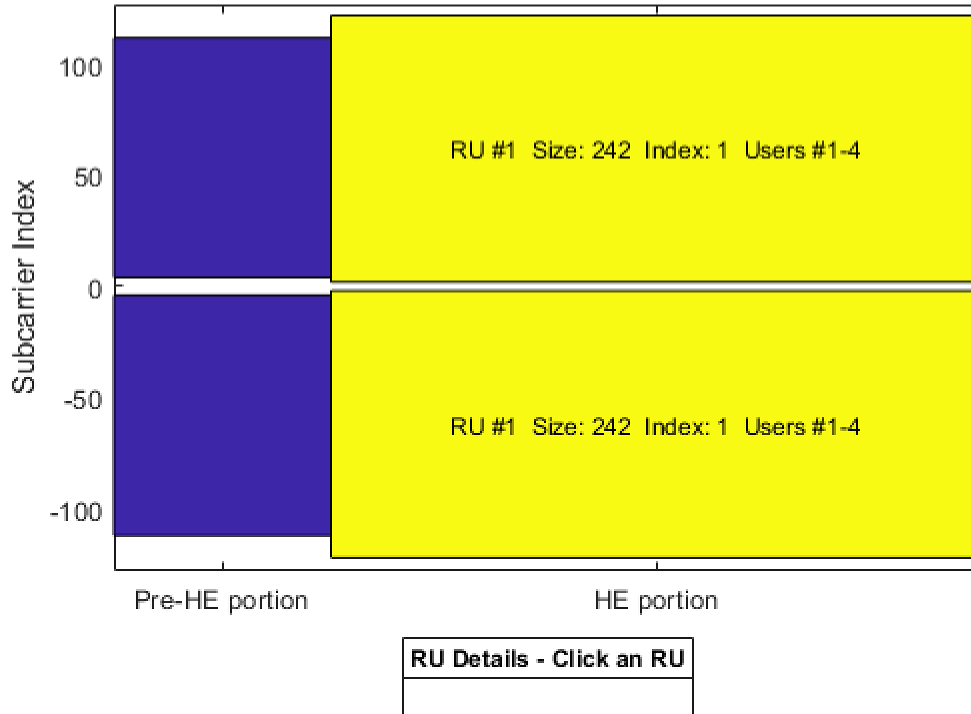
A 20 MHz channel bandwidth is used for all transmissions. Other transmission parameters such as the APEPLength and MCS are the same for all users in all configurations.

First, the MU-MIMO configuration is defined. The allocation index `195` defines a single 242-tone RU, with four users in MU-MIMO. For a description of selecting an allocation index, see the “802.11ax Waveform Generation” on page 1-64 example.

```
% MU-MIMO configuration - 4 users on one 242-tone RU
cfgMUMIMO = wlanHEMUConfig(195);
```

The allocation plot shows a single RU is assigned to all four users.

```
showAllocation(cfgMUMIMO);
```



The transmission parameters for each user are now configured.

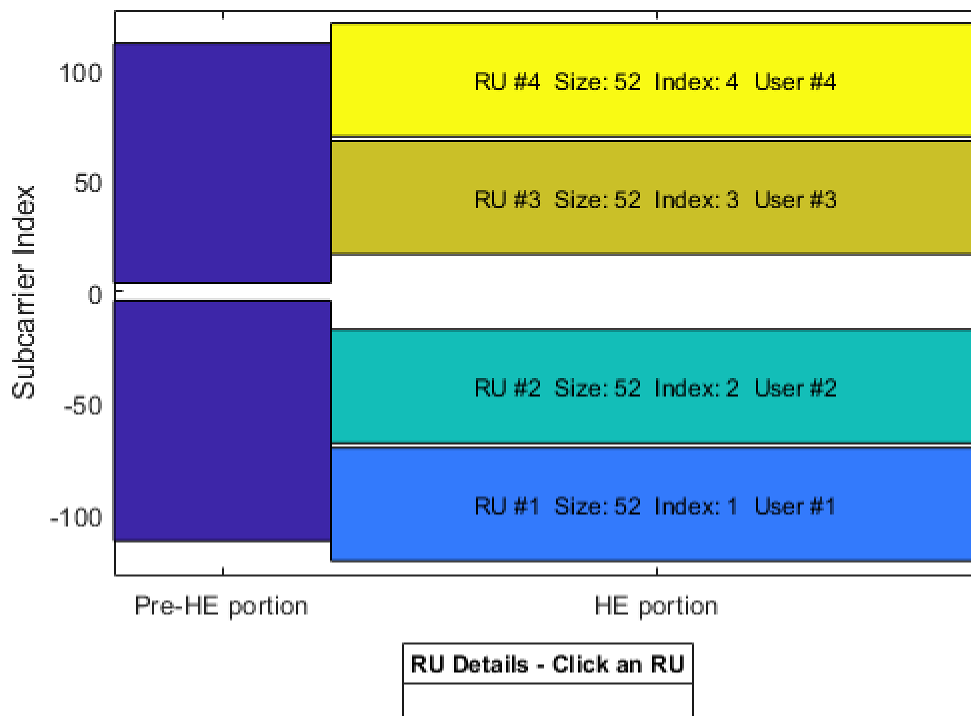
```
numTx = 6; % Number of transmit antennas
guardInterval = 0.8; % Guard interval in Microseconds

% Configure common parameters for all users
cfgMUMIMO.NumTransmitAntennas = numTx;
cfgMUMIMO.GuardInterval = guardInterval;

% Configure per user parameters
% STA #1
cfgMUMIMO.User{1}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{1}.MCS = 4;
cfgMUMIMO.User{1}.APEPLength = 1000;
% STA #2
cfgMUMIMO.User{2}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{2}.MCS = 4;
cfgMUMIMO.User{2}.APEPLength = 1000;
% STA #3
cfgMUMIMO.User{3}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{3}.MCS = 4;
cfgMUMIMO.User{3}.APEPLength = 1000;
% STA #4
cfgMUMIMO.User{4}.NumSpaceTimeStreams = 1;
cfgMUMIMO.User{4}.MCS = 4;
cfgMUMIMO.User{4}.APEPLength = 1000;
```

Next the OFDMA configuration is defined. The allocation index 112 defines four 52-tone RUs, each serving a single user.

```
% OFDMA configuration - 4 users, each on a 52-tone RU
cfgOFDMA = wlanHEMUConfig(112);
```

The allocation plot shows the four RUs, each with a single user. When comparing this allocation plot to the full band MU-MIMO plot, it is apparent that the total number of subcarriers used (4x52 = 208 subcarriers) is less than the MU-MIMO allocation (242 subcarriers). The fewer number of subcarriers allow guards between each OFDMA user.

```
showAllocation(cfgOFDMA);
```



The transmission parameters for each user are now configured.

```
% Configure common parameters for all users
cfgOFDMA.NumTransmitAntennas = numTx;
cfgOFDMA.GuardInterval = guardInterval;

% Configure per user parameters
% STA #1 (RU #1)
cfgOFDMA.User{1}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{1}.MCS = 4;
cfgOFDMA.User{1}.APEPLength = 1000;
% STA #2 (RU #2)
cfgOFDMA.User{2}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{2}.MCS = 4;
```

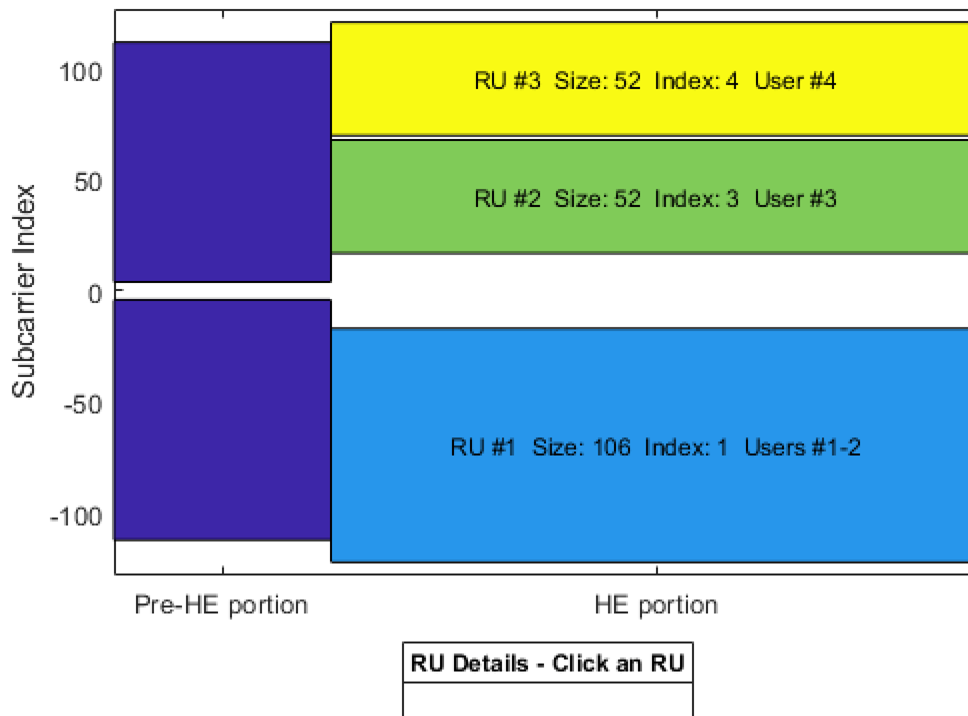
```
cfgOFDMA.User{2}.APEPLength = 1000;
% STA #3 (RU #3)
cfgOFDMA.User{3}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{3}.MCS = 4;
cfgOFDMA.User{3}.APEPLength = 1000;
% STA #4 (RU #4)
cfgOFDMA.User{4}.NumSpaceTimeStreams = 2;
cfgOFDMA.User{4}.MCS = 4;
cfgOFDMA.User{4}.APEPLength = 1000;
```

Finally, the mixed MU-MIMO and OFDMA configuration is defined. The allocation index 25 defines a 106-tone RU with two users, and two 52-tone RUs, each with one user.

```
% Mixed OFDMA and MU-MIMO configuration
cfgMixed = wlanHEMUConfig(25);
```

The allocation plot shows the three RUs, one with 2 users (MU-MIMO), and the others with one user each (OFDMA).

```
showAllocation(cfgMixed);
```



The transmission parameters for each user are now configured.

```
% Configure common parameters for all users
cfgMixed.NumTransmitAntennas = numTx;
cfgMixed.GuardInterval = guardInterval;

% Configure per user parameters
```

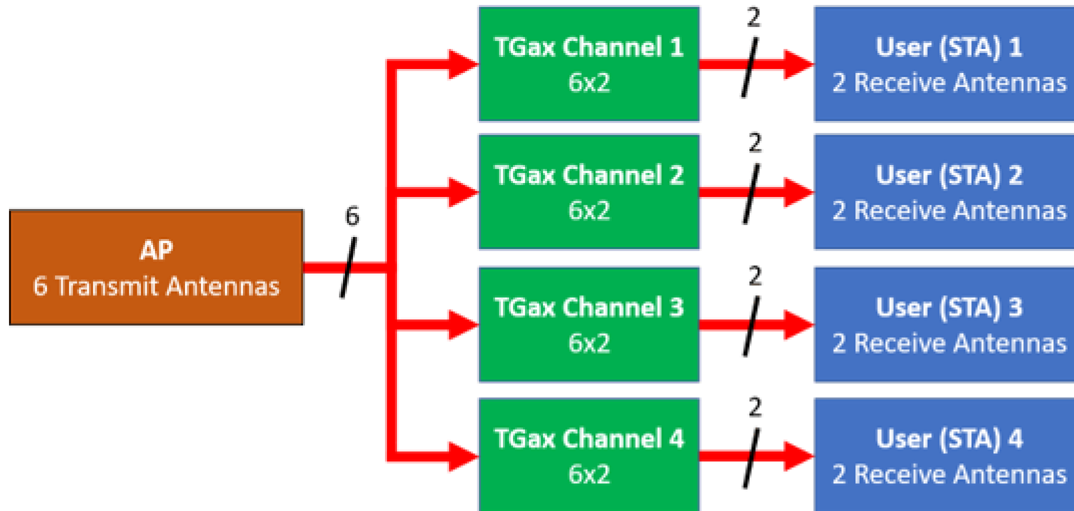
```
% RU #1 has two users (MU-MIMO) and a total of 2 STS (1 per user)
% STA #1 (RU #1)
cfgMixed.User{1}.NumSpaceTimeStreams = 1;
cfgMixed.User{1}.MCS = 4;
cfgMixed.User{1}.APEPLength = 1000;
% STA #2 (RU #1)
cfgMixed.User{2}.NumSpaceTimeStreams = 1;
cfgMixed.User{2}.MCS = 4;
cfgMixed.User{2}.APEPLength = 1000;

% The remaining two users are OFDMA
% STA #3 (RU #2)
cfgMixed.User{3}.NumSpaceTimeStreams = 2;
cfgMixed.User{3}.MCS = 4;
cfgMixed.User{3}.APEPLength = 1000;
% STA #4 (RU #3)
cfgMixed.User{4}.NumSpaceTimeStreams = 2;
cfgMixed.User{4}.MCS = 4;
cfgMixed.User{4}.APEPLength = 1000;
```

Channel Model Configuration

A TGax indoor channel model is used in this example. An individual channel is used to simulate the link between the AP and each user. A TGax channel object, `tgaxBase` is created with properties relevant for all users. In this example, the delay profile (Model-D) and number of receive antennas are common for all users. Model-D is considered non-line of sight when the distance between transmitter and receiver is greater than or equal to 10 meters. This is described further in `wlanTGaxChannel`. A fixed seed is used for the channel to allow repeatability.

```
% Create channel configuration common for all users
tgaxBase = wlanTGaxChannel;
tgaxBase.DelayProfile = 'Model-D';     % Delay profile
tgaxBase.NumTransmitAntennas = numTx;  % Number of transmit antennas
tgaxBase.NumReceiveAntennas = 2;       % Each user has two receive antennas
tgaxBase.TransmitReceiveDistance = 10; % Non-line of sight distance
tgaxBase.ChannelBandwidth = cfgMUMIMO.ChannelBandwidth;
tgaxBase.SampleRate = wlanSampleRate(cfgMUMIMO);
% Set a fixed seed for the channel
tgaxBase.RandomStream = 'mt19937ar with seed';
tgaxBase.Seed = 5;
```

Next a channel is created for each user. The channel for each user is a clone of the `tgaxBase`, but with a unique `UserIndex` property, and is stored in a cell array `tgax`. The `UserIndex` property of each individual channel is set to provide a unique channel for each user. The resultant channels are used in the simulation as shown below.




```
% A cell array stores the channel objects, one per user
numUsers = numel(cfgMixed.User); % Number of users simulated in this example
tgax = cell(1,numUsers);

% Generate per-user channels
for userIdx = 1:numUsers
    tgax{userIdx} = clone(tgaxBase);
    tgax{userIdx}.UserIndex = userIdx; % Set unique user index
end
```

Beamforming Feedback

Transmit beamforming for both OFDMA and MU-MIMO relies on knowledge of the channel state between transmitter and receiver at the beamformer. Feedback of the per-subcarrier channel state is provided by each STA by channel sounding. A null data packet (NDP) is transmitted by the AP, and each STA uses this packet to determine the channel state. The channel state is then fed-back to the AP. The same process is used for 802.11ac™ in the “802.11ac Transmit Beamforming” on page 3-26 and “802.11ac Multi-User MIMO Precoding” on page 1-93 examples, but an HE single user NDP packet is used instead of a VHT packet. In this example, the feedback is considered perfect; there is no noise present for channel sounding and the feedback is uncompressed. The `heUserBeamformingFeedback` helper function detects the NDP and uses channel estimation to determine the channel state information. Singular value decomposition (SVD) is then used to calculate the beamforming feedback.

```
% Create an NDP with the correct number of space-time streams to generate
% enough LTF symbols
cfgNDP = wlanHESUConfig('APEPLength',0,'GuardInterval',0.8); % No data in an NDP
cfgNDP.ChannelBandwidth = tgaxBase.ChannelBandwidth;
cfgNDP.NumTransmitAntennas = cfgMUMIMO.NumTransmitAntennas;
cfgNDP.NumSpaceTimeStreams = cfgMUMIMO.NumTransmitAntennas;

% Generate NDP packet - with an empty PSDU as no data
txNDP = wlanWaveformGenerator([],cfgNDP);

% For each user STA, pass the NDP packet through the channel and calculate
% the feedback channel state matrix by SVD.
staFeedback = cell(1,numUsers);
for userIdx = 1:numel(tgax)
```

```
    % Received waveform at user STA with 50 sample padding. No noise.
    rx = tgax{userIdx}([txNDP; zeros(50,size(txNDP,2))]);

    % Get the full-band beamforming feedback for a user
    staFeedback{userIdx} = heUserBeamformingFeedback(rx,cfgNDP);
end
```

Simulation Parameters

Different path losses are simulated in this example. The same path loss and noise floor is applied to all users. For each path loss simulated, 10 packets are passed through the channel. Packets are separated by 20 microseconds.

```
cfgSim = struct;
cfgSim.NumPackets = 10;       % Number of packets to simulate for each path loss
cfgSim.Pathloss = (96:3:105); % Path losses to simulate in dB
cfgSim.TransmitPower = 30;    % AP transmit power in dBm
cfgSim.NoiseFloor = -89.9;    % STA noise floor in dBm
cfgSim.IdleTime = 20;         % Idle time between packets in us
```

Simulation with OFDMA

The scenario is first simulated with the OFDMA configuration and transmit beamforming.

The steering matrix for each RU is calculated using the feedback from the STAs. The `heMUCalculateSteeringMatrix` helper function calculates the beamforming matrix for an RU given the CSI feedback.

```
% For each RU, calculate the steering matrix to apply
for ruIdx = 1:numel(cfgOFDMA.RU)
    % Calculate the steering matrix to apply to the RU given the feedback
    steeringMatrix = heMUCalculateSteeringMatrix(staFeedback,cfgOFDMA,cfgNDP,ruIdx);

    % Apply the steering matrix to each RU
    cfgOFDMA.RU{ruIdx}.SpatialMapping = 'Custom';
    cfgOFDMA.RU{ruIdx}.SpatialMappingMatrix = steeringMatrix;
end
```

The `heMUSimulateScenario` helper function performs the simulation. The pre-HE preamble of 802.11ax is backwards compatible with 802.11ac, therefore in this example the front-end synchronization components for a VHT waveform are used to synchronize the HE waveform at each STA. For each packet and path loss simulated the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through an evolving TGax channel model and AWGN is added to the received waveform. The channel state is maintained between packets.
3. The packet is detected.
4. Coarse carrier frequency offset is estimated and corrected.
5. Fine timing synchronization is established.
6. Fine carrier frequency offset is estimated and corrected.
7. The HE-LTF is extracted from the synchronized received waveform. The HE-LTF is OFDM demodulated and channel estimation is performed.
8. The HE Data field is extracted from the synchronized received waveform and OFDM demodulated.

9. Common pilot phase tracking is performed to track any residual carrier frequency offset.
10. The phase corrected OFDM symbols are equalized with the channel estimate.
11. Noise estimation is performed using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers.
12. The equalized symbols are demodulated and decoded to recover the PSDU.
13. The recovered PSDU is compared to the transmitted PSDU to determine if the packet has been recovered successfully.

The simulation is run for the OFDMA configuration.

```
disp('Simulating OFDMA...');
throughputOFDMA = heMUSimulateScenario(cfgOFDMA,tgax,cfgSim);
```

```
Simulating OFDMA...
 Pathloss 96.0 dB, AP throughput 66.1 Mbps
 Pathloss 99.0 dB, AP throughput 66.1 Mbps
 Pathloss 102.0 dB, AP throughput 49.6 Mbps
 Pathloss 105.0 dB, AP throughput 16.5 Mbps
```

Simulation with MU-MIMO

Now the scenario is simulated with the MU-MIMO configuration. The `heMUCalculateSteeringMatrix` helper function calculates the beamforming matrix for an RU given the CSI feedback for all users in the MU-MIMO allocation. A zero forcing solution is used to calculate the steering matrix within the helper function.

```
% Calculate the steering matrix to apply to the RU given the feedback
ruIdx = 1; % Index of the one and only RU
steeringMatrix = heMUCalculateSteeringMatrix(staFeedback,cfgMUMIMO,cfgNDP,ruIdx);

% Apply the steering matrix to the RU
cfgMUMIMO.RU{1}.SpatialMapping = 'Custom';
cfgMUMIMO.RU{1}.SpatialMappingMatrix = steeringMatrix;
```

Run the simulation for the MU-MIMO configuration.

```
disp('Simulating MU-MIMO...');
throughputMUMIMO = heMUSimulateScenario(cfgMUMIMO,tgax,cfgSim);
```

```
Simulating MU-MIMO...
 Pathloss 96.0 dB, AP throughput 110.5 Mbps
 Pathloss 99.0 dB, AP throughput 110.5 Mbps
 Pathloss 102.0 dB, AP throughput 63.5 Mbps
 Pathloss 105.0 dB, AP throughput 0.0 Mbps
```

Simulation with Combined MU-MIMO and OFDMA

Finally, the scenario is simulated with the combined MU-MIMO and OFDMA configuration.

The steering matrix for each RU is calculated using the feedback from the STAs, including the MU-MIMO RU. The `heMUCalculateSteeringMatrix` helper function calculates the beamforming matrix for an RU given the CSI feedback.

```
% For each RU calculate the steering matrix to apply
for ruIdx = 1:numel(cfgMixed.RU)
    % Calculate the steering matrix to apply to the RU given the feedback
```

```
    steeringMatrix = heMUCalculateSteeringMatrix(staFeedback,cfgMixed,cfgNDP,ruIdx);

    % Apply the steering matrix to each RU
    cfgMixed.RU{ruIdx}.SpatialMapping = 'Custom';
    cfgMixed.RU{ruIdx}.SpatialMappingMatrix = steeringMatrix;
end
```

Run the simulation for the combined MU-MIMO and OFDMA configuration.

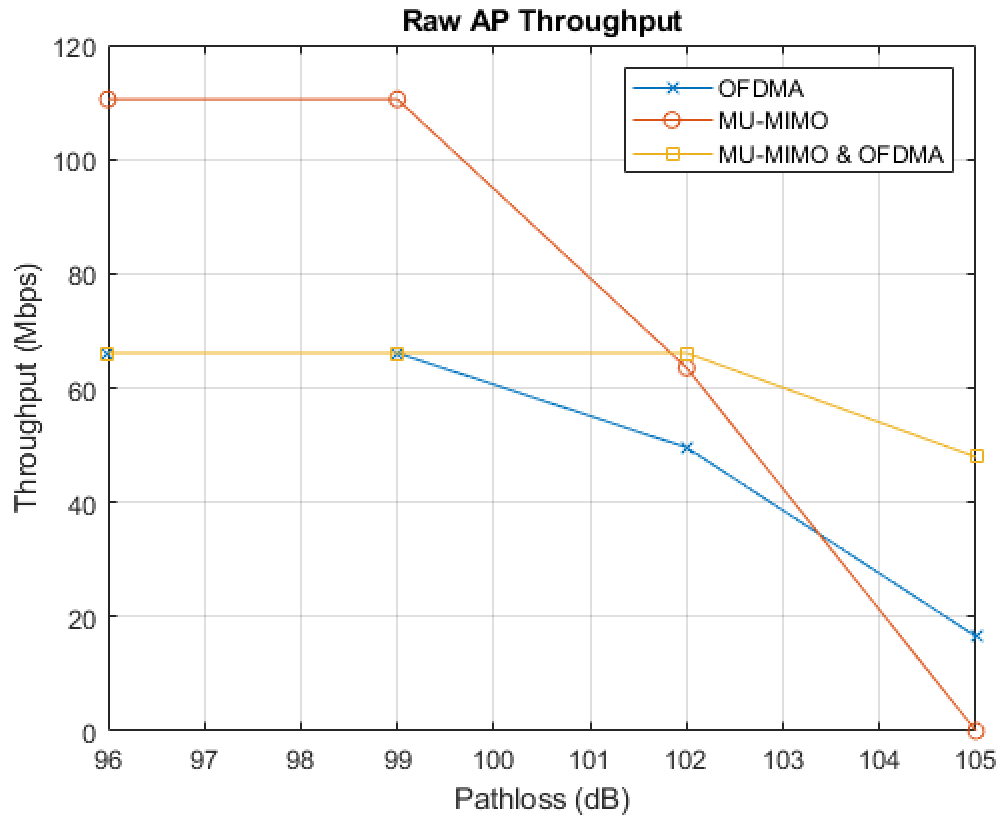
```
disp('Simulating Mixed MU-MIMO and OFDMA...');
throughputMixed = heMUSimulateScenario(cfgMixed,tgax,cfgSim);
```

```
Simulating Mixed MU-MIMO and OFDMA...
 Pathloss 96.0 dB, AP throughput 66.1 Mbps
 Pathloss 99.0 dB, AP throughput 66.1 Mbps
 Pathloss 102.0 dB, AP throughput 66.1 Mbps
 Pathloss 105.0 dB, AP throughput 47.9 Mbps
```

Plot Results

The raw AP throughput for each transmission mode is plotted. The results show for this channel realization at high SNRs (low pathloss) the throughput provided by the MU-MIMO configuration exceeds OFDMA configuration. The packet duration of the MU-MIMO configuration is roughly half that of the OFDMA configuration which provides the throughput gain. As the SNR decreases, the noise dominates and transmit beamforming with OFDMA becomes more effective. The performance of the combined MU-MIMO and OFDMA configuration follow a similar trend to the OFDMA configuration as the packet duration is the same. The performance differs due to different RU sizes and number of space-time streams.

```
% Sum throughput for all STAs and plot for all configurations
figure;
plot(cfgSim.Pathloss,sum(throughputOFDMA,2),'-x');
hold on;
plot(cfgSim.Pathloss,sum(throughputMUMIMO,2),'-o');
plot(cfgSim.Pathloss,sum(throughputMixed,2),'-s');
grid on;
xlabel('Pathloss (dB)');
ylabel('Throughput (Mbps)');
legend('OFDMA','MU-MIMO','MU-MIMO & OFDMA');
title('Raw AP Throughput');
```



Appendix

This example uses these helper functions.

- heEqualizeCombine.m
- heMUCalculateSteeringMatrix.m
- heMUSimulateScenario.m
- heNoiseEstimate.m
- heUserBeamformingFeedback.m

Selected Bibliography

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

See Also

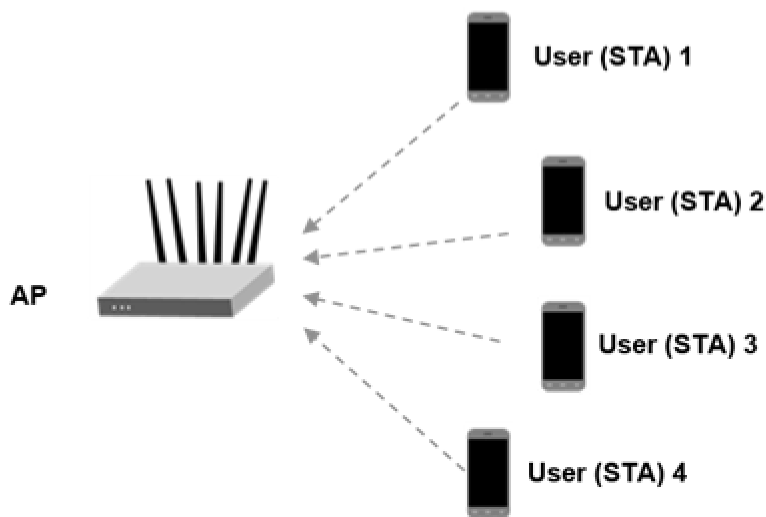
Functions
awgn

802.11ax Packet Error Rate Simulation for Uplink Trigger-Based Format

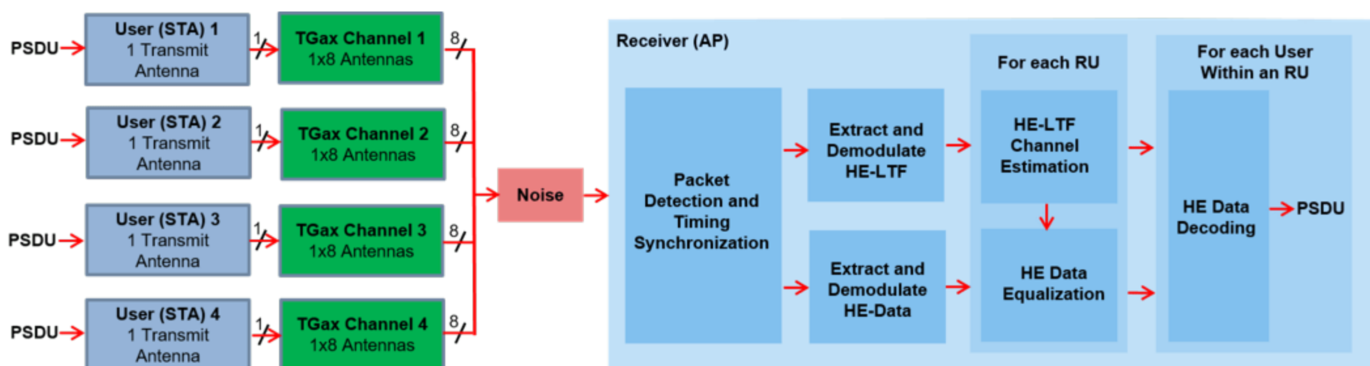
This example shows how to measure the packet error rate of an IEEE® 802.11ax™ high efficiency (HE) uplink, trigger-based (TB) format.

Introduction

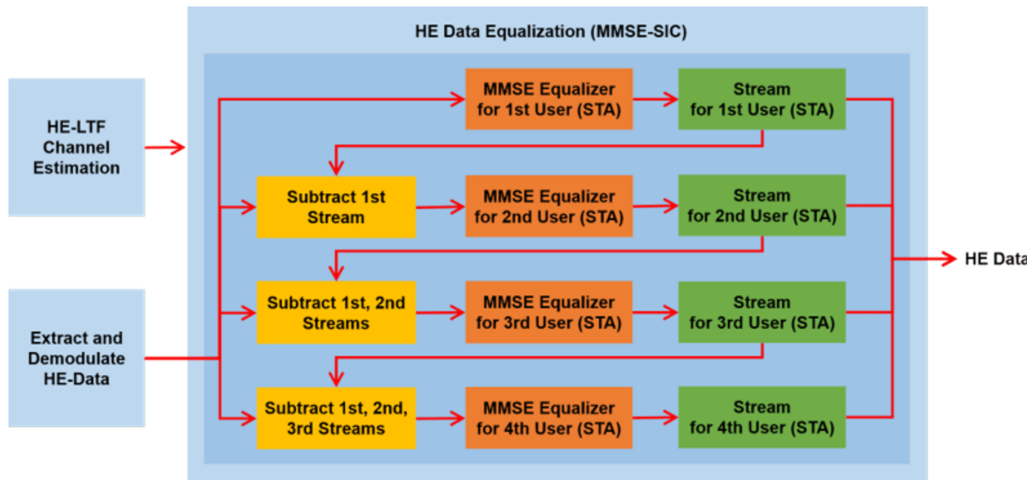
The 802.11ax [1] HE trigger-based (HE TB) format allows for OFDMA or MU-MIMO transmission in the uplink. An HE TB transmission is controlled entirely by an access point (AP). All the parameters required for the transmission are provided in a trigger frame to all STAs participating in the HE TB transmission. Each station (STA) transmits an HE TB packet simultaneously, when triggered by the AP as shown in the following diagram.



In this example an end-to-end simulation is used to determine the packet error rate of an HE TB link for four STAs in a MU-MIMO configuration. At each SNR point multiple packets are transmitted with no impairments apart from channel and noise. The received packets are demodulated and the PSDUs recovered for each STA. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate for all users. Packet detection, timing synchronization and symbol equalization is performed by the receiver. No frequency offset correction is performed in this example. The processing of HE TB processing chain is shown in the following diagram.



The receiver performs a minimum-mean-square-error-based ordered successive interference cancellation (MMSE-SIC) process for data equalization [2]. To avoid error propagation in the cancellation stage, data streams for all STAs are sorted in descending order based on the channel state information and equalized sequentially. This diagram shows the procedure of MMSE-SIC equalization.



Equalization Method

In this example, you can specify the equalization method as 'mmse' or 'mmse-sic'. The default equalizer is 'mmse-sic'.

```
equalizationMethod = 'mmse-sic';
```

User Configuration

In this example the allocation information and transmit parameters for multiple uplink STAs are configured using an `heTBSystemConfig` object.

```
allocationIndex = 195; % Four uplink users in a MU-MIMO configuration
cfgSys = heTBSystemConfig(allocationIndex);
```

In a trigger-based transmission some parameters are the same for all uplink users, while some can differ. The `User` property of `cfgSys` contains a cell array of user configurations. Each element of the cell array is an object which can be configured to set the parameters of individual users. In this example, all users have the same transmission parameters.

```
% These parameters are same for all users in the MU-MIMO system
cfgSys.HELTFType = 4;           % HE-LTF compression mode
cfgSys.GuardInterval = 3.2;     % Guard interval type
cfgSys.SingleStreamPilots = 1; % Single stream pilot transmission of HE-LTF
numRx = 8;                      % Number of receive(AP) antennas

% The individual parameters for each user are specified below
allocInfo = ruInfo(cfgSys);
numUsers = allocInfo.NumUsers; % Number of uplink users

for userIdx = 1:numUsers
    cfgSys.User{userIdx}.NumTransmitAntennas = 1;
    cfgSys.User{userIdx}.NumSpaceTimeStreams = 1;
```

```
    cfgSys.User{userIdx}.SpatialMapping = 'Direct';
    cfgSys.User{userIdx}.MCS = 7;
    cfgSys.User{userIdx}.APEPLength = 1e3;
    cfgSys.User{userIdx}.ChannelCoding = 'LDPC';
end
```

A trigger-based transmission for a single user within the system is configured with an `wlanHETBConfig` object. The transmission configurations for all users are generated using the method `getUserConfig`. A cell array of four HE TB objects is created to describe the transmission of four users.

```
cfgTB = getUserConfig(cfgSys);
```

Simulation Parameters

For each SNR point (dB) in the `snr` vector a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 20:2:24;

% The sample rate and field indices for the HE TB packet is same for all
% users. Here the trigger configuration of the first user is used to get
% the sample rate and field indices of the HE TB PPDU.
fs = wlanSampleRate(cfgTB{1});    % Same for all users
ind = wlanFieldIndices(cfgTB{1}); % Same for all users
```

Channel Configuration

In this example, a TGax NLOS indoor channel model is used with delay profile Model-B. Model-B is considered NLOS when the distance between the transmitter and receiver is greater than or equal to 5 meters. This is described further in `wlanTGaxChannel`. In this example all the STAs are assumed to be at the same distance from the AP.

```
tgaxBase = wlanTGaxChannel;
tgaxBase.SampleRate = fs;
tgaxBase.TransmissionDirection = 'Uplink';
tgaxBase.TransmitReceiveDistance = 10;
chanBW = cfgSys.ChannelBandwidth;
tgaxBase.ChannelBandwidth = chanBW;
tgaxBase.NumReceiveAntennas = numRx;
tgaxBase.NormalizeChannelOutputs = false;
```

An individual channel is created for each of the four users. Each channel is a clone of `tgaxBase`, but with a different `UserIndex` property, and is stored in a cell array `tgax`. The `UserIndex` property of each individual channel is set to provide a unique channel for each user. In this example a random channel realization is used for each packet by randomly varying the `UserIndex` property for each transmitted packet.

```
% A cell array stores the channel objects, one per user
tgax = cell(1,numUsers);
for userIdx = 1:numUsers
    tgax{userIdx} = clone(tgaxBase);
    tgax{userIdx}.NumTransmitAntennas = cfgSys.User{userIdx}.NumTransmitAntennas;
    tgax{userIdx}.UserIndex = userIdx;
end
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate is calculated. The pre-HE preamble of 802.11ax is backwards compatible with 802.11ac™, therefore in this example the timing synchronization components for a VHT waveform are used to synchronize the HE waveform at the receiver. For each user, the following processing steps occur to create a waveform at the receiver containing all four users:

1. To create an HE TB waveform, a PSDU is created and encoded for each user based on predefined user parameters.
2. The waveform for each user is passed through an indoor TGax channel model. Different channel realizations are modeled for different users and packets, by randomly varying the `UserIndex` property of the channel. This results in same spatial correlation properties for all users.
3. The waveforms for all HE TB users are scaled and combined to ensure same SNR for each user after the addition of noise.
4. AWGN is added to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.

At the receiver (AP) the following processing steps occur:

1. The packet is detected.
2. Fine timing synchronization is established. The L-STF, L-LTF and L-SIG samples are provided for fine timing to allow for packet detection at the start or end of the L-STF.
3. The HE-LTF and HE-Data fields for all users are extracted from the synchronized received waveform. The HE-LTF and HE-Data fields are OFDM demodulated.
4. The demodulated HE-LTF is extracted for each RU and channel estimation is performed.
5. Noise estimation is performed using the demodulated data field pilots for each RU.
6. The data field is extracted and equalized for all users within an RU, from the demodulated data field.
7. For each RU, and user within the RU, the spatial streams for a user are demodulated and decoded to recover the transmitted PSDU.

A parfor loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

```
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgSys.ChannelBandwidth,cfgSys.GuardInterval);
numSNR = numel(snr); % Number of SNR points
numPackets = 50; % Number of packets to simulate
packetErrorRate = zeros(numUsers,numSNR);
txPSDU = cell(numUsers);

% parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
for isnr = 1:numSNR
    % Create HE TB object for receiver processing
    cfgHETB = wlanHETBConfig;
    cfgHETB.ChannelBandwidth = cfgSys.ChannelBandwidth;
    cfgHETB.HELTFType = cfgSys.HELTFType;
    cfgHETB.SingleStreamPilots = cfgSys.SingleStreamPilots;

    % 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);

% RU allocation information
sysInfo = ruInfo(cfgSys);

% Simulate multiple packets
numPacketErrors = zeros(numUsers,1);
for pktIdx = 1:numPackets

    % Transmit processing
    rxWaveform = 0;
    packetError = zeros(numUsers,1);
    txPSDU = cell(1,numUsers);

    % Generate random channel realization for each packet by varying
    % the UserIndex property of the channel. This assumes all users
    % have the same number of transmit antennas.
    chPermutations = randperm(numUsers);
    for userIdx = 1:numUsers
        % HE TB config object for each user
        cfgUser = cfgTB{userIdx};

        % Generate a packet with random PSDU
        txPSDU{userIdx} = randi([0 1],getPSDULength(cfgUser)*8,1,'int8');

        % Generate HE TB waveform, containing payload for single user
        txTrig = wlanWaveformGenerator(txPSDU{userIdx},cfgUser);

        % Pass waveform through a random TGax Channel
        channelIdx = chPermutations(userIdx);
        reset(tgax{channelIdx}); % New channel realization
        rxTrig = tgax{channelIdx}([txTrig; zeros(15,size(txTrig,2))]);

        % Scale the transmit power of the user within an RU. This is to
        % ensure same SNR for each user after the addition of noise.
        ruNum = cfgSys.User{userIdx}.RUNumber;
        SF = sqrt(1/sysInfo.NumUsersPerRU(ruNum))*sqrt(cfgUser.RUSize/(sum(sysInfo.RUSizes)))

        % Combine uplink users into one waveform
        rxWaveform = rxWaveform+SF*rxTrig;
    end

    % Pass the waveform through AWGN channel. Account for noise
    % energy in nulls so the SNR is defined per active subcarriers.
    packetSNR = snr(isnr)-10*log10(ofdmInfo.FFTLength/sum(sysInfo.RUSizes));
    rxWaveform = awgn(rxWaveform,packetSNR);

    % Receive processing
    % Packet detect and determine coarse packet offset
    coarsePktOffset = wlanPacketDetect(rxWaveform,chanBW,0,0.05);
    if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
        numPacketErrors = numPacketErrors+1;
        continue; % Go to next loop iteration
    end

    % Extract the non-HT fields and determine fine packet offset
```

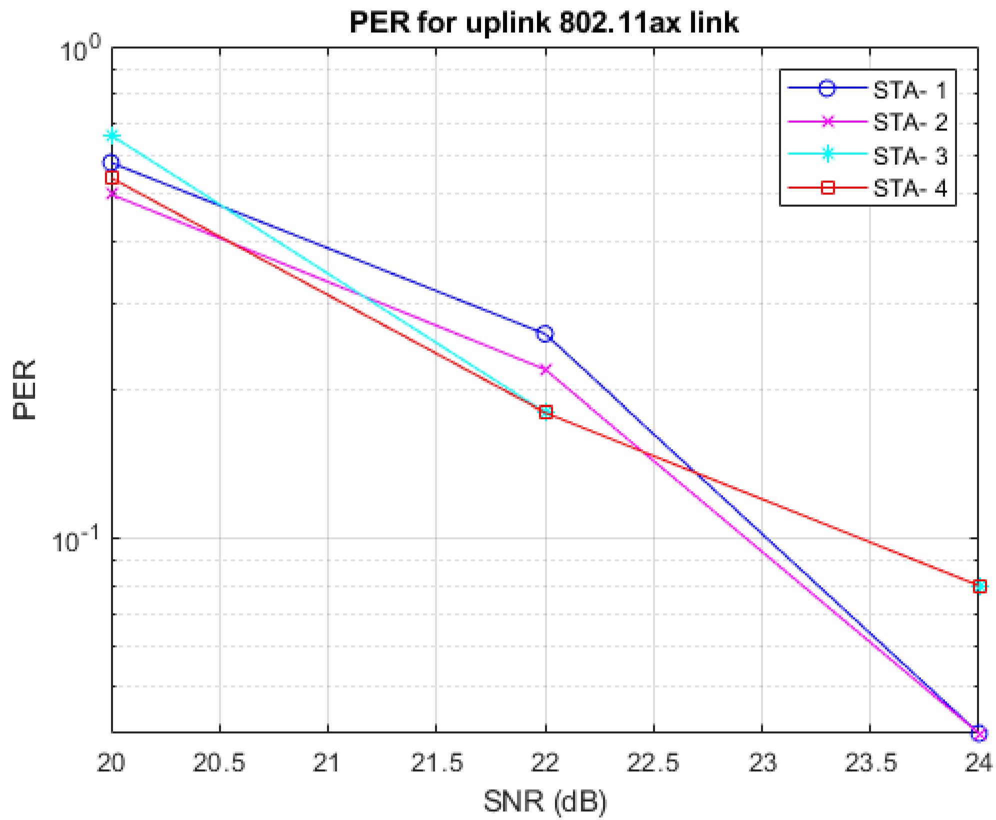
```
nonhtfields = rxWaveform(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

% Determine final packet offset
pktOffset = coarsePktOffset+finePktOffset;

% If packet detected out with the range of expected delays from
% the channel modeling; packet error
if pktOffset>50
    numPacketErrors = numPacketErrors+1;
    continue; % Go to next loop iteration
end

% Extract HE-LTF and HE-Data fields for all RUs
rxLTF = rxWaveform(pktOffset+(ind.HELTF(1):ind.HELTF(2)),:);
rxData = rxWaveform(pktOffset+(ind.HEData(1):ind.HEData(2)),:);

for ruIdx = 1:allocInfo.NumRUs
    % Demodulate HE-LTF and HE-Data field for the RU of interest
    ru = [allocInfo.RUSizes(ruIdx) allocInfo.RUIndices(ruIdx)];
    demodHELTFRU = wlanHEDemodulate(rxLTF,'HE-LTF',chanBW,cfgSys.GuardInterval,cfgSys.HE
    demodHEDataRU = wlanHEDemodulate(rxData,'HE-Data',chanBW,cfgSys.GuardInterval,ru);

    % Configure the relevant properties in HE TB object
    cfgHETB.RUSize = allocInfo.RUSizes(ruIdx);
    cfgHETB.RUIndex = allocInfo.RUIndices(ruIdx);
    cfgHETB.NumSpaceTimeStreams = allocInfo.NumSpaceTimeStreamsPerRU(ruIdx);

    % Channel estimate
    [chanEst,ssPilotEst] = wlanHELTFChannelEstimate(demodHELTFRU,cfgHETB);

    % Get indices of data and pilots within RU (without nulls)
    ruOFDMInfo = wlanHEOFDMInfo('HE-Data',cfgSys.ChannelBandwidth,cfgSys.GuardInterval,
        [allocInfo.RUSizes(ruIdx) allocInfo.RUIndices(ruIdx)]);

    % Estimate noise power in HE fields of each user
    nVarEst = heNoiseEstimate(demodHEDataRU(ruOFDMInfo.PilotIndices,:,:),ssPilotEst,cfgSy

    % Discard pilot subcarriers
    demodDataSym = demodHEDataRU(ruOFDMInfo.DataIndices,:,:);
    chanEstData = chanEst(ruOFDMInfo.DataIndices,:,:);

    % Equalize
    if strcmpi(equalizationMethod,'mmse-sic')
        [eqSym,csi] = heSuccessiveEqualize(demodDataSym,chanEstData,nVarEst,cfgSys,ruIdx
    else
        [eqSym,csi] = heEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgSys);
    end

    for userIdx = 1:allocInfo.NumUsersPerRU(ruIdx)
        % Get TB config object for each user
        userNum = cfgSys.RU{ruIdx}.UserNumbers(userIdx);
        cfgUser = cfgTB{userNum};

        % Get space-time stream indices for the current user
        stsIdx = cfgUser.StartingSpaceTimeStream-1+(1:cfgUser.NumSpaceTimeStreams);

        % Demap and decode bits
```

```
                rxPSDU = wlanHEDataBitRecover(eqSym(:,:,stsIdx),nVarEst,csi(:,stsIdx),cfgUser,'LD

                % PER calculation
                packetError(userNum) = any(biterr(txPSDU{userNum},rxPSDU));
            end
        end
        numPacketErrors = numPacketErrors+packetError;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(:,isnr)= numPacketErrors/numPackets;
    disp(['SNR ' num2str(snr(isnr)) ...
          ' completed for ' num2str(numUsers) ' users']);

end
```

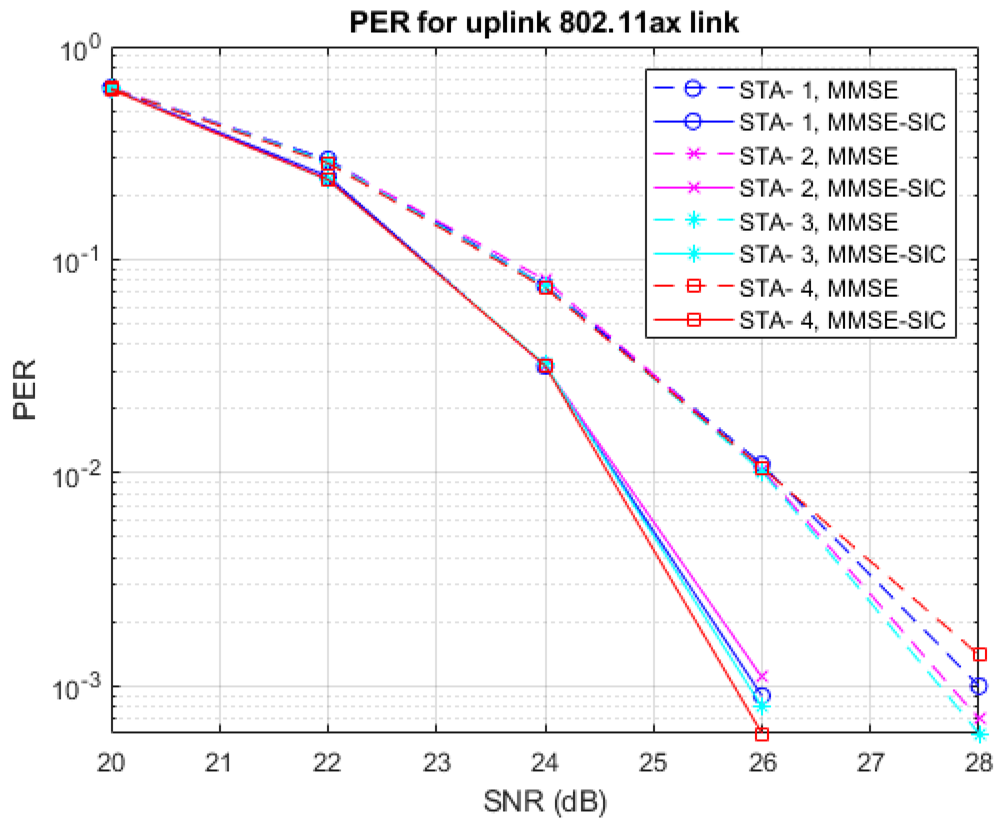
```
SNR 20 completed for 4 users
SNR 22 completed for 4 users
SNR 24 completed for 4 users
```

Plot Packet Error Rate vs SNR

```
markers = 'ox*sd^v><ph+ox*sd^v';
color = 'bmcrgbrkymcrgbrkymc';
figure;

for nSTA = 1:numUsers
    semilogy(snr,packetErrorRate(nSTA,:).',['-' markers(nSTA) color(nSTA)]);
    hold on;
end

grid on;
xlabel('SNR (dB)');
ylabel('PER');
dataStr = arrayfun(@(x)sprintf('STA- %d',x),1:numUsers,'UniformOutput',false);
legend(dataStr);
title('PER for uplink 802.11ax link');
```



The number of packets tested at each SNR point is controlled by `numPackets`. For meaningful results, this value should be larger than those presented in this example. The figure below was created by running a longer simulation with `numPackets`:1e4 and `snr`:20:2:28, which shows the packet error rate of both MMSE equalizer and MMSE-SIC equalizer.



Appendix

This example uses the following helper functions and objects:

- heEqualizeCombine.m
- heNoiseEstimate.m
- heSuccessiveEqualize.m
- heTBRU.m
- heTBSystemConfig.m
- heTBUser.m

Selected Bibliography

1. IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.
2. M. Debbah, B. Muquet, M. de Courville, M. Muck, S. Simoens, and P. Loubaton. A MMSE successive interference cancellation scheme for a new adjustable hybrid spread OFDM system. IEEE 51st Vehicular Technology Conference Proceedings, pp. 745-749, vol. 2, 2000.

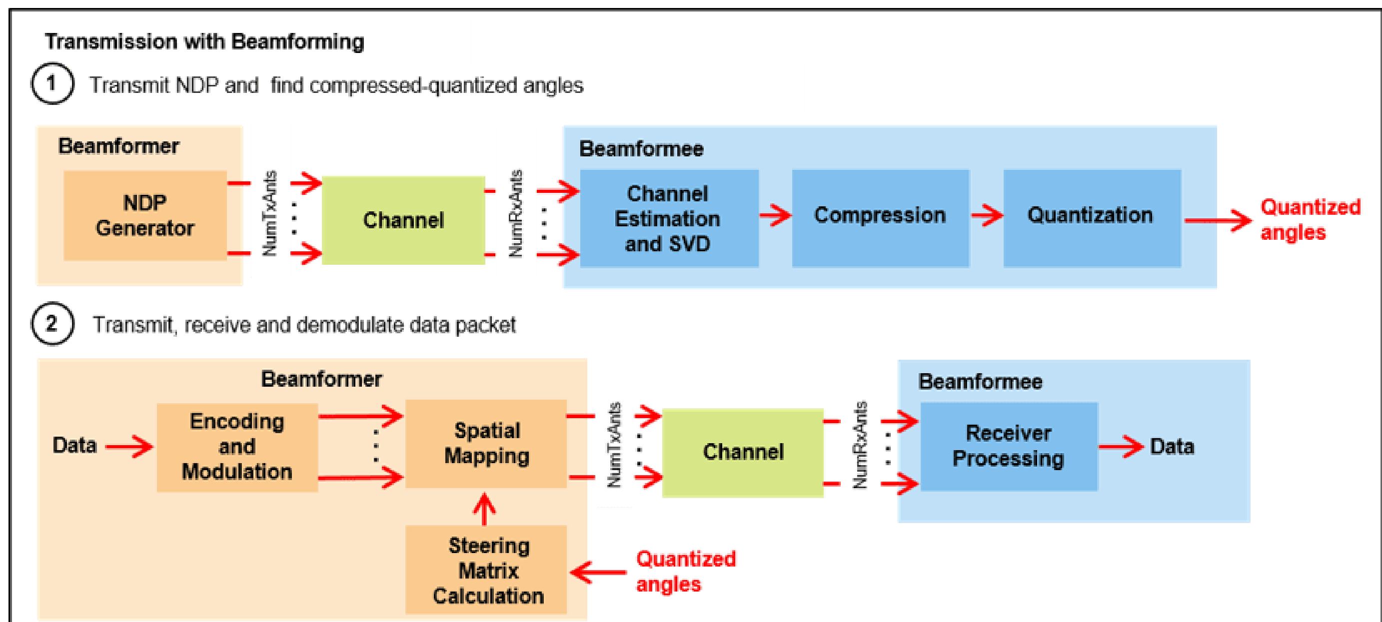
802.11ax Compressed Beamforming Packet Error Rate Simulation

This example shows how to measure the packet error rate of a beamformed IEEE® 802.11ax™ high efficiency single user (HE SU) format link with different beamforming feedback quantization levels.

Introduction

Transmit beamforming focuses energy towards a receiver to improve the SNR of a link. In this scheme, the transmitter is called a beamformer and the receiver is called a beamformee. A steering matrix is used by the beamformer to direct the energy to the beamformee. The steering matrix is calculated using channel state information obtained through channel measurements. These measurements are obtained by sounding the channel between beamformer and beamformee. To sound the channel, the beamformer sends a null data packet (NDP) to the beamformee. The beamformee measures the channel information during sounding to calculate a feedback matrix. This matrix is compressed in the form of quantized angles (phi and psi) and fed back to the beamformer. The beamformer can then calculate the feedback matrix from the quantized angles to create a steering matrix and beamform transmissions to the beamformee. The process of forming steering matrix is shown in “802.11ac Transmit Beamforming” on page 3-26.

This example measures the packet error rate (PER) for an 802.11ax [1] single-user (SU) link comprising a 4x2 MIMO configuration between a transmitter and receiver with two space-time streams. The link utilizes compressed beamforming feedback quantization for different quantization levels and a selection of SNR points. This example does not consider grouping of subcarriers (see Section 9.4.1.65 in [1]).



Waveform Configuration

An HE SU packet is a full-band transmission to a single user. Configure transmit parameters for the HE SU format using a `wlanHESUConfig` object. Configure the object for a 20 MHz channel bandwidth, four transmit antennas, two space-time streams, 16-QAM rate-1/2 (MCS 3), and 4x HE-

LTF compression mode. This example does not model subcarrier beamforming smoothing. To sound all subcarriers and estimate the channel without interpolation for this transmission configuration, use 4x HE-LTF compression mode.

```
NumTxAnts = 4; % Number of transmit antennas
NumSTS = 2;    % Number of space-time streams
NumRxAnts = 2; % Number of receive antennas
cfgHEBase = wlanHESUConfig;
cfgHEBase.ChannelBandwidth = 'CBW20';      % Channel bandwidth
cfgHEBase.NumSpaceTimeStreams = NumSTS;    % Number of space-time streams
cfgHEBase.NumTransmitAntennas = NumTxAnts; % Number of transmit antennas
cfgHEBase.APEPLength = 1e3;                % Payload length in bytes
cfgHEBase.ExtendedRange = false;           % Do not use extended range format
cfgHEBase.Upper106ToneRU = false;          % Do not use upper 106 tone RU
cfgHEBase.PreHESpatialMapping = false;     % Spatial mapping of pre-HE fields
cfgHEBase.GuardInterval = 3.2;             % Guard interval duration
cfgHEBase.HELTFType = 4;                   % HE-LTF compression mode
cfgHEBase.ChannelCoding = 'LDPC';          % Channel coding
cfgHEBase.MCS = 3;                         % Modulation and coding scheme
cfgHEBase.SpatialMapping = 'Custom';       % Custom for beamforming
```

Null Data Packet (NDP) Configuration

Configure the NDP transmission to have data length of zero. Since the NDP is used to obtain the channel state information, set the number of space-time streams equal to the number of transmit antennas and directly map each space-time stream to a transmit antenna.

```
cfgNDP = cfgHEBase;
cfgNDP.APEPLength = 0;                  % NDP has no data
cfgNDP.NumSpaceTimeStreams = NumTxAnts; % For feedback matrix calculation
cfgNDP.SpatialMapping = 'Direct';       % Each TxAnt carries a STS
```

Channel Configuration

This example uses a TGax NLOS indoor channel model with delay profile Model-B. The Model-B profile is considered NLOS when the distance between transmitter and receiver is greater than or equal to five meters. For more information, see `wlanTGaxChannel`.

```
% Create and configure the TGax channel
chanBW = cfgHEBase.ChannelBandwidth;
tgaxChannel = wlanTGaxChannel;
tgaxChannel.DelayProfile = 'Model-B';
tgaxChannel.NumTransmitAntennas = NumTxAnts;
tgaxChannel.NumReceiveAntennas = NumRxAnts;
tgaxChannel.TransmitReceiveDistance = 5; % Distance in meters for NLOS
tgaxChannel.ChannelBandwidth = chanBW;
tgaxChannel.LargeScaleFadingEffect = 'None';
tgaxChannel.NormalizeChannelOutputs = false;
fs = wlanSampleRate(cfgHEBase);
tgaxChannel.SampleRate = fs;
```

Simulation Parameters

This example compares the performance of beamforming with two different resolutions of compression quantization, and without compression. For each quantization resolution, perform an end-to-end simulation with various SNR values to determine the PER. 802.11ax specifies only two sets of quantization resolution for single user beamforming (Table 9-29a in [1]). Set the value of

`codeBookSize` to slect the number of bits used to quantize the beamforming feedback angles (phi and psi) in this simulation. To disable compression set `codeBookSize` to `Inf`. This table shows the quantization levels for each value of `codeBookSize`:

```
codeBookSize            Compression Configuration
--------------------------------------------------------
    0                   NumBitsphi = 4; NumBitspsi = 2
    1                   NumBitsphi = 6; NumBitspsi = 4
   Inf                  No compression
--------------------------------------------------------
```

```
codeBookSize = [0 1 Inf];
```

Set the SNRs to simulate in dB.

```
snr = 10:2:18;
```

Limit the number of packets tested at each SNR point to `maxNumErrors` or `maxNumPackets`, where:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example lead to a very short simulation. For statistically meaningful results, increase these numbers.

```
maxNumErrors = 10;   % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Processing SNR Points

For each SNR point, test packets and calculate the PER. For each packet, perform these steps.

Obtain the steering matrix to create a feedback matrix:

1. Transmit an NDP waveform through an indoor TGax channel model. Model different channel realizations for different packets.
2. Add AWGN to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.
3. Detect the packet at the beamformee.
4. Estimate and correct for coarse carrier frequency offset (CFO).
5. Establish fine timing synchronization.
6. Estimate and correct for fine CFO.
7. Extract the HE-LTF from the synchronized received waveform. OFDM demodulate the HE-LTF and perform channel estimation.
8. Perform singular value decomposition on the estimated channel and calculate the beamforming feedback matrix, V.
9. Compress and quantize the feedback matrix, V, to create a set of angles as specified in the standard.

Transmit a data packet using the recovered steering matrix and decode the beamformed data transmission to recover the PSDU:

1. If modeling compression, convert the quantized angles back to the beamforming feedback matrix, V. Otherwise, use the V matrix calculated by the beamformee. Assume no beamforming feedback delay.
2. Create and encode a PSDU to create a single-packet waveform with the steering matrix set to the beamforming feedback matrix, V.
3. Pass the waveform through the same indoor TGax channel realization as the NDP transmission.
4. Add AWGN to the received waveform.
5. As with NDP, perform synchronization and HE channel estimation.
6. Extract the data field from the synchronized received waveform and OFDM demodulate.
7. Perform common phase error pilot tracking to track any residual carrier frequency offset.
8. Estimate the noise power using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers.
9. Equalize the phase corrected OFDM symbols with the channel estimate.
10. Demodulate and decode the equalized symbols to recover the PSDU.

To parallelize processing of SNR points, you can use a `parfor` loop. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

```
numQuant = numel(codeBookSize);
numSNR = numel(snr); % Number of SNR points
packetErrorRate = zeros(numQuant,numSNR);

% Get occupied subcarrier indices and OFDM parameters
ofdmInfo = wlanHEOFDMInfo('HE-Data',cfgHEBase);

% Indices to extract fields from the PPDU
ind = wlanFieldIndices(cfgHEBase);
indSound = wlanFieldIndices(cfgNDP);

for ibf = 1:numQuant
    switch codeBookSize(ibf) % See P802.11ax/D4.1 Section 9.4.1.64
        case 0
            NumBitsPsi = 2; % Number of bits for psi
            NumBitsPhi = 4; % Number of bits for phi
            disp('End-to-End simulation with compressed beamforming quantization with');
            disp(['Number of Bits for phi = ' num2str(NumBitsPhi) ...
                ' and Number of Bits for psi = ' num2str(NumBitsPsi)]);
        case 1
            NumBitsPsi = 4; % Number of bits for psi
            NumBitsPhi = 6; % Number of bits for phi
            disp('End-to-End simulation with compressed beamforming quantization with');
            disp(['Number of Bits for phi = ' num2str(NumBitsPhi) ...
                ' and Number of Bits for psi = ' num2str(NumBitsPsi)]);
        otherwise
            disp('End-to-End simulation with non-compressed beamforming');
    end

    %parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
    for isnr = 1:numSNR
        % Set random substream index per iteration to ensure that each
        % iteration uses a repeatable set of random numbers
        stream = RandStream('combRecursive','Seed',100);
```

```
stream.Substream = isnr;
RandStream.setGlobalStream(stream);

% Account for noise energy in nulls so the SNR is defined per
% active subcarrier
packetSNR = snr(isnr)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

% Create an instance of the HE configuration object per SNR point
% simulated. This will enable to use parfor
cfgHE = cfgHEBase;

% Loop to simulate multiple packets
numPacketErrors = 0;
numPkt = 1; % Index of packet transmitted
while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
    % Null data packet transmission
    tx = wlanWaveformGenerator([],cfgNDP);

    % Add trailing zeros to allow for channel delay
    txPad = [tx; zeros(50,cfgNDP.NumTransmitAntennas)];

    % Pass through a fading indoor TGax channel
    reset(tgaxChannel); % Reset channel for different realization
    rx = tgaxChannel(txPad);

    % Pass the waveform through AWGN channel
    rx = awgn(rx,packetSNR);

    % Calculate the steering matrix at the beamformee
    V = heUserBeamformingFeedback(rx,cfgNDP,true);

    if isempty(V)
        % User feedback failed, packet error
        numPacketErrors = numPacketErrors+1;
        numPkt = numPkt+1;
        continue; % Go to next loop iteration
    end

    if ~isinf(codeBookSize(ibf))
        % Find quantized angles of the beamforming feedback matrix
        angidx = bfCompressQuantize(V(:,1:NumSTS,:),NumBitsPhi,NumBitsPsi);

        % Calculate steering matrix from the quantized angles at
        % beamformer:
        % Assuming zero delay in transmitting the quantized angles
        % from beamformee to beamformer, the steering matrix is
        % calculated from the quantized angles and is used in the
        % data transmission of beamformer.

        [~,Nc,Nr] = size(V(1,1:NumSTS,:));
        V = bfDecompress(angidx,Nr,Nc,NumBitsPhi,NumBitsPsi);
    end

    steeringMat = V(:,1:NumSTS,:);

    % Beamformed data transmission
    psduLength = getPSDULength(cfgHE); % PSDU length in bytes
    txPSDU = randi([0 1],psduLength*8,1); % Generate random PSDU
```

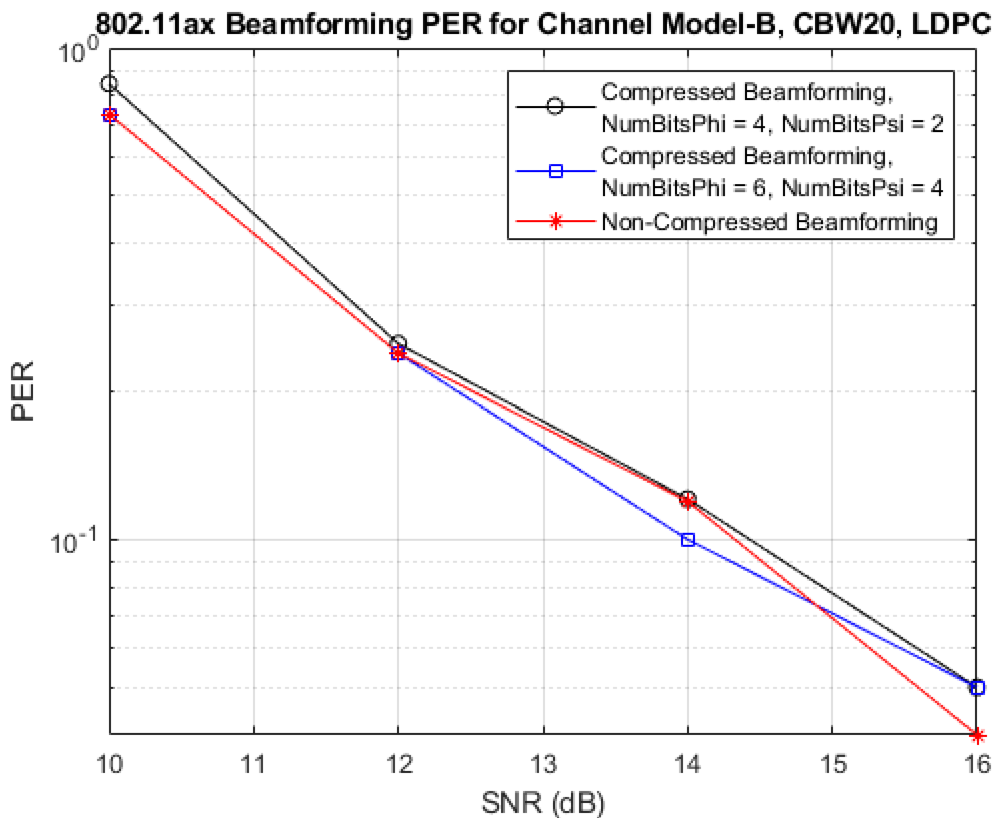
```
cfgHE.SpatialMappingMatrix = steeringMat;
tx = wlanWaveformGenerator(txPSDU,cfgHE);

% Add trailing zeros to allow for channel delay
txPad = [tx; zeros(50,cfgHE.NumTransmitAntennas)];

% Pass through a fading indoor TGax channel
rx = tgaxChannel(txPad);

% Pass the waveform through AWGN channel
rx = awgn(rx,packetSNR);

% Packet detect and determine coarse packet offset
coarsePktOffset = wlanPacketDetect(rx,chanBW);
if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-STF and perform coarse frequency offset correction
lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
coarseFreqOff = wlanCoarseCFOEstimate(lstf,chanBW);
rx = frequencyOffset(rx,fs,-coarseFreqOff);

% Extract the non-HT fields and determine fine packet offset
nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

% Determine final packet offset
pktOffset = coarsePktOffset+finePktOffset;

% If packet detected outwith the range of expected delays from
% the channel modeling; packet error
if pktOffset>50
    numPacketErrors = numPacketErrors+1;
    numPkt = numPkt+1;
    continue; % Go to next loop iteration
end

% Extract L-LTF and perform fine frequency offset correction
rxLLTF = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
fineFreqOff = wlanFineCFOEstimate(rxLLTF,chanBW);
rx = frequencyOffset(rx,fs,-fineFreqOff);

% HE-LTF demodulation and channel estimation
rxHELTF = rx(pktOffset+(ind.HELTF(1):ind.HELTF(2)),:);
heltfDemod = wlanHEDemodulate(rxHELTF,'HE-LTF',cfgHE);
[chanEst,pilotEst] = wlanHELTFChannelEstimate(heltfDemod,cfgHE);

% Data demodulate
rxData = rx(pktOffset+(ind.HEData(1):ind.HEData(2)),:);
demodSym = wlanHEDemodulate(rxData,'HE-Data',cfgHE);

% Pilot phase tracking
% Average single-stream pilot estimates over symbols (2nd dimension)
pilotEstTrack = mean(pilotEst,2);
demodSym = wlanHETrackPilotError(demodSym,pilotEstTrack,cfgHE,'HE-Data');
```

```
            % Estimate noise power in HE fields
            nVarEst = heNoiseEstimate(demodSym(ofdmInfo.PilotIndices,:,:),pilotEstTrack,cfgHE);

            % Extract data subcarriers from demodulated symbols and channel
            % estimate
            demodDataSym = demodSym(ofdmInfo.DataIndices,:,:);
            chanEstData = chanEst(ofdmInfo.DataIndices,:,:);

            % Equalization and STBC combining
            [eqDataSym,csi] = heEqualizeCombine(demodDataSym,chanEstData,nVarEst,cfgHE);

            % Recover data
            rxPSDU = wlanHEDataBitRecover(eqDataSym,nVarEst,csi,cfgHE,'LDPCDecodingMethod','norm

            % Determine if any bits are in error, i.e. a packet error
            packetError = ~isequal(txPSDU,rxPSDU);
            numPacketErrors = numPacketErrors+packetError;
            numPkt = numPkt+1;
        end

        % Calculate packet error rate (PER) at SNR point
        packetErrorRate(ibf,isnr) = numPacketErrors/(numPkt-1);
        disp(['MCS ' num2str(cfgHE.MCS) ','...
            ' SNR ' num2str(snr(isnr)) ...
            ' completed after ' num2str(numPkt-1) ' packets,'...
            ' PER:' num2str(packetErrorRate(ibf,isnr))]);
    end
    disp(newline);
end
```

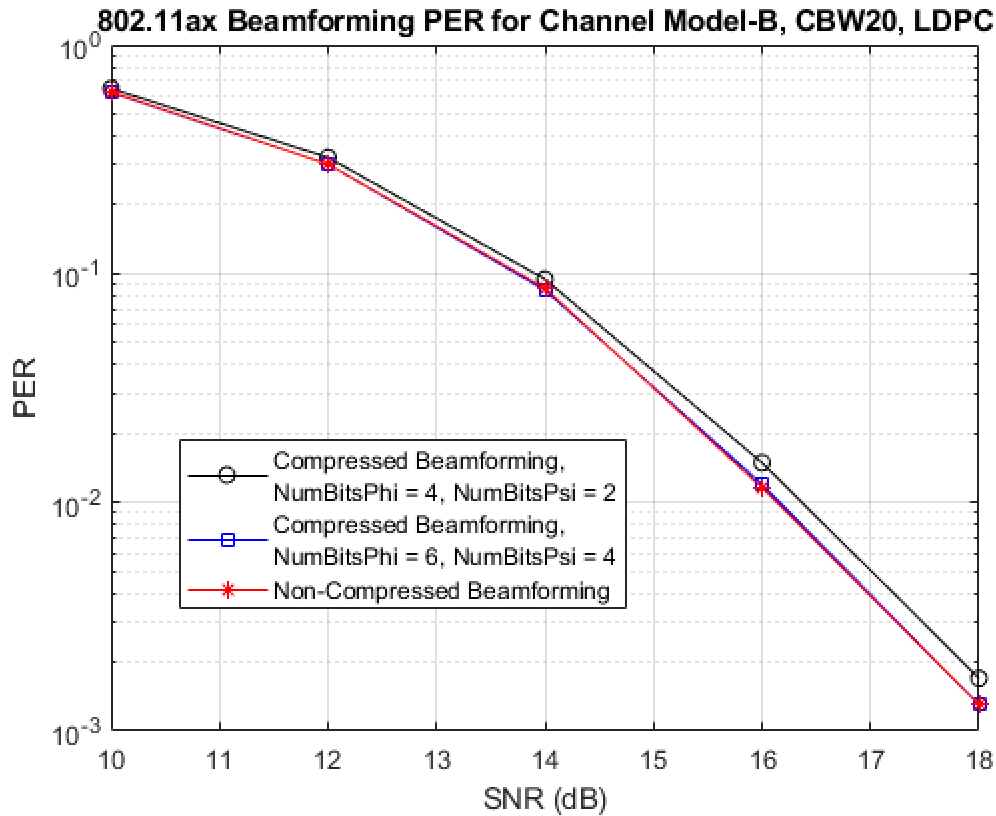
```
End-to-End simulation with compressed beamforming quantization with
Number of Bits for phi = 4 and Number of Bits for psi = 2
MCS 3, SNR 10 completed after 13 packets, PER:0.84615
MCS 3, SNR 12 completed after 44 packets, PER:0.25
MCS 3, SNR 14 completed after 91 packets, PER:0.12088
MCS 3, SNR 16 completed after 100 packets, PER:0.05
MCS 3, SNR 18 completed after 100 packets, PER:0


End-to-End simulation with compressed beamforming quantization with
Number of Bits for phi = 6 and Number of Bits for psi = 4
MCS 3, SNR 10 completed after 15 packets, PER:0.73333
MCS 3, SNR 12 completed after 46 packets, PER:0.23913
MCS 3, SNR 14 completed after 100 packets, PER:0.1
MCS 3, SNR 16 completed after 100 packets, PER:0.05
MCS 3, SNR 18 completed after 100 packets, PER:0


End-to-End simulation with non-compressed beamforming
MCS 3, SNR 10 completed after 15 packets, PER:0.73333
MCS 3, SNR 12 completed after 46 packets, PER:0.23913
MCS 3, SNR 14 completed after 92 packets, PER:0.11957
MCS 3, SNR 16 completed after 100 packets, PER:0.04
MCS 3, SNR 18 completed after 100 packets, PER:0
```

Plot Packet Error Rate vs Signal to Noise Ratio

```
figure;
lineTypes = ["k-o" "b-s" "r-*"];
semilogy(snr,packetErrorRate(1,:),lineTypes(1));
hold on;
grid on;
xlabel('SNR (dB)');
ylabel('PER');
for ibf = 2:numQuant
    semilogy(snr,packetErrorRate(ibf,:),lineTypes(ibf));
end
dataStr = [string(['Compressed Beamforming, ' newline ...
                   'NumBitsPhi = 4, NumBitsPsi = 2' newline])...
           string(['Compressed Beamforming, ' newline ...
                   'NumBitsPhi = 6, NumBitsPsi = 4' newline]) ...
                   "Non-Compressed Beamforming"];
legend(dataStr);
title(sprintf('802.11ax Beamforming PER for Channel %s, %s, %s',tgaxChannel.DelayProfile,cfgHEBas
```



The number of packets tested at each SNR point is controlled by two parameters: `maxNumErrors` and `maxNumPackets`. For meaningful results, increase the values used in this example. The figure below was created by running a longer simulation with `maxNumErrors`:1e3 and `maxNumPackets`:1e4.



Further Exploration

Using 4x HE-LTF compression mode for NDP and data packet transmissions allows all subcarriers to be sounded and the channel to be estimated. When 2x HE-LTF compression mode is used linear interpolation is performed during channel estimation. This example does not perform subcarrier beamforming smoothing. Therefore, if you configure the simulation to use 2x HE-LTF compression, the interpolation performed during channel estimation will not correctly estimate the beamforming matrix for all subcarriers and the PER will increase.

Selected Bibliography

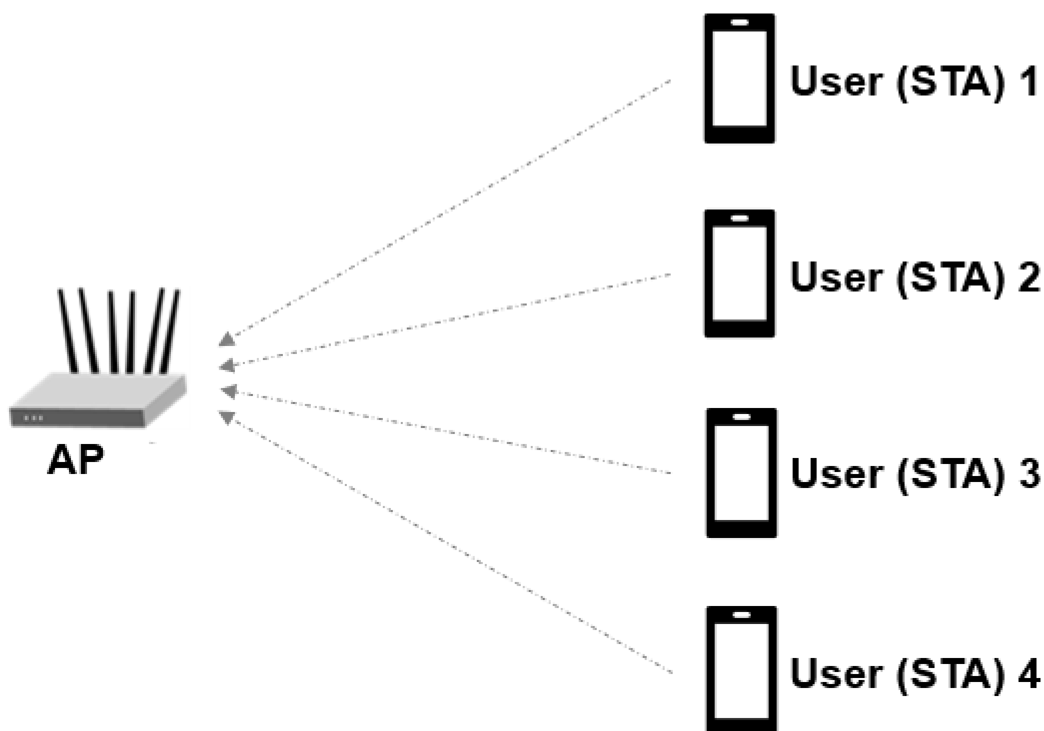
1 IEEE Std 802.11ax™-2021. IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 1: Enhancements for High-Efficiency WLAN.

802.11ax Feedback Status Misdetection Simulation for Uplink Trigger-Based Feedback NDP

This example shows how to measure the probability of misdetecting feedback status information in an uplink high efficiency (HE) trigger-based (TB) feedback null data packet (NDP) transmission from multiple uplink stations (STAs).

Introduction

The 802.11ax [1 on page 6-86] HE TB feedback NDP is a variant of the HE TB physical layer protocol data unit (PPDU). The HE TB feedback NDP transmission is controlled entirely by an access point (AP). All the parameters required for the transmission are provided in a trigger frame of type NDP feedback report poll (NFRP) sent to all STAs participating in the HE TB feedback transmission. Following the transmission of an NFRP trigger frame from the AP, multiple STAs may simultaneously transmit an HE TB feedback NDP, which carries the resource request information (feedback status) as shown in this diagram. For more information on the NDP feedback report procedure, see the NDP Feedback Report Procedure section of the `wlanHETBConfig` reference page.



This example measures the probability of misdetecting feedback status information for an HE TB feedback NDP by comparing the transmitted and received feedback statuses. A misdetection is recorded when the recovered feedback status is incorrect or undetermined. The example performs this measurement for a transmission between four STAs and an AP by using an end-to-end simulation. The STA infers the transmission parameters from the User Info field of the soliciting NFRP trigger frame. At each signal-to-noise ratio (SNR) point the STA transmits multiple packets with no

impairments apart from multipath fading and noise. The STA demodulates the packet and recovers the feedback status. The AP determines the probability of misdetecting feedback status by comparing the recovered and transmitted feedback status.

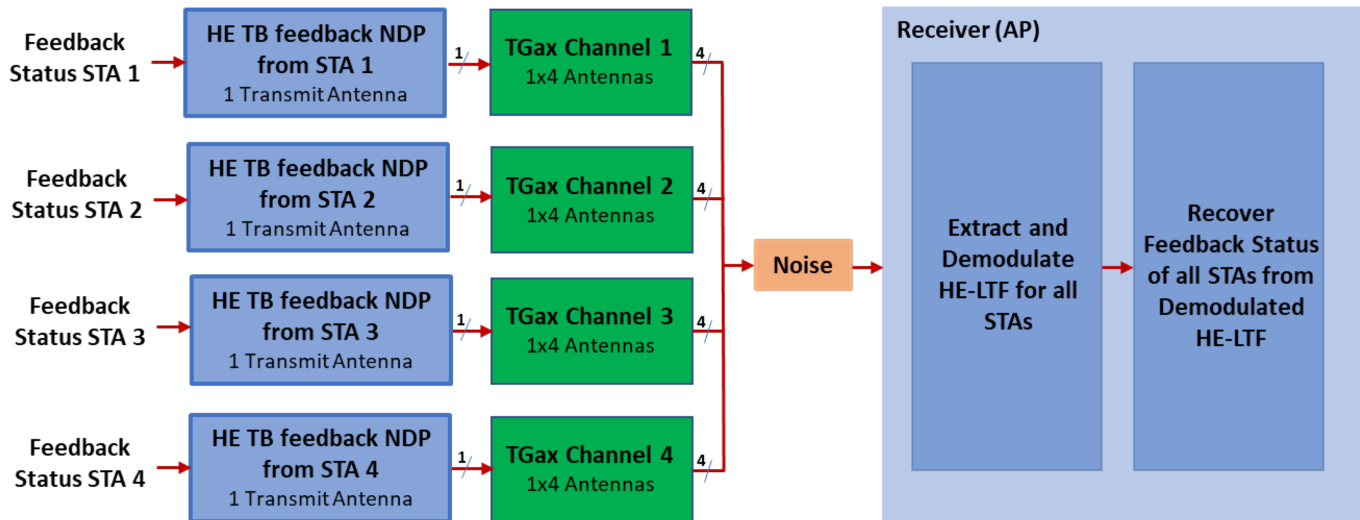
Each STA transmits a waveform by performing these processing steps.

1. Determine if the STA is scheduled to transmit an HE TB feedback NDP.
2. Determine the RU tone set index and starting space time stream number for all transmitting STAs.
3. Generate an HE TB feedback NDP for the STAs.
4. Pass the waveform for each STA through an indoor channel by using the `wlanTGaxChannel` System object (TM) . Model different channel realizations for each user and each packet by randomly varying the UserIndex property of `wlanTGaxChannel`. This process results in the same spatial correlation properties for all STAs.
5. Combine waveforms from all STAs.
6. Add additive white Gaussian noise (AWGN) to the received waveform. The AWGN creates the desired average SNR per subcarrier after orthogonal frequency-division multiplexing (OFDM) demodulation for each STA.

The receiver (AP) performs these processing steps on the received waveform.

1. Use perfect channel delay estimate to synchronize.
2. Extract the HE-LTF from the synchronized waveform and demodulate the HE-LTF.
3. Recover feedback status information from the demodulated HE-LTF symbols [2 on page 6-86].

This figure shows the processing for each link between the STA and AP.



Simulate Uplink Transmission

This section simulates an end-to-end uplink scenario for multiple STAs and SNR points. Specify the number of STAs and the SNR range. This section estimates the probability of misdetecting feedback status for all STAs. The feedback status represents the resource request information from the STA and is defined in Table 26-3 of [1 on page 6-86].

```
% Set transmission parameters
numSTAs          = 4 ; % Number of uplink STAs
STAID            = [1 19 2 4] ; % STA association ID, assigned to each associated STA
multiplexingFlag = [0 1 0 0] ; % Signaled in the trigger frame of type NFRP for each
feedbackStatus   = [1 0 1 0] ; % Resource request information signaled by each STA
startingAID      = 0 ; % Signaled in the trigger frame of type NFRP

% Set simulation parameters
numPackets = 50 ; % Number of packets to simulate
snrRange   = -2:2:6 ; % SNR points (dB)
chanBW     = CBW20 ; % Channel bandwidth
numTx      = 1 ; % Assume same number of transmit antennas for all STAs
numRx      = 4 ; % Number of receive (AP) antennas
```

HE TB Feedback NDP Waveform Configuration

Configure the waveform generator for each STA. The STA performs these steps.

- Check if the STA is scheduled to transmit.
- Calculate the RUToneSetIndex for each STA from `STAID`, `startingAID`, and `chanBW`.
- Calculate starting space time stream number for all STAs from `STAID, startingAID,` and `chanBW`.
- Generate configuration object for all STAs.

```
% Return the index of the transmitting STAs. Calculate RUToneSetIndex and starting space time str
[txSTAIndex,ruToneSetIndexPerSTA,startingSTSNumPerSTA] = heTBNDPMappingParams(chanBW,numSTAs,mult
numTxSTAs = numel(txSTAIndex); % STAs scheduled to transmit
cfgSTA = cell(1,numTxSTAs);

% Generate the configuration object and set the feedback status property for all STAs
cfgBase = wlanHETBConfig('ChannelBandwidth',chanBW,'NumTransmitAntennas',numTx,'SpatialMapping',
cfgNDP = getNDPFeedbackConfiguration(cfgBase);
if numSTAs~=numel(feedbackStatus)
    error('The number of elements in FeedbackStatus must be equal to the number of STAs');
end

for u=1:numTxSTAs
      cfgNDP.RUToneSetIndex = ruToneSetIndexPerSTA(u);
      cfgNDP.StartingSpaceTimeStream = startingSTSNumPerSTA(u);
      cfgNDP.FeedbackStatus = feedbackStatus(txSTAIndex(u));
      cfgSTA{u} = cfgNDP;
end
```

Channel Configuration

This example uses a TGax non-line-of-sight (NLOS) indoor channel model with delay profile Model-D. Model-D is considered NLOS when the distance between the transmitter and receiver is greater than

or equal to ten meters. For more information, see `wlanTGaxChannel`. This example assumes that all STAs are at the same distance from the AP.

```
delayProfile = Model-D ; % TGax channel multipath delay profile
tgaxBase = wlanTGaxChannel;
tgaxBase.DelayProfile = delayProfile;
tgaxBase.SampleRate = wlanSampleRate(cfgSTA{1});
tgaxBase.TransmissionDirection = 'Uplink';
tgaxBase.TransmitReceiveDistance = 10;
tgaxBase.ChannelBandwidth = chanBW;
tgaxBase.NumReceiveAntennas = numRx;
tgaxBase.NormalizeChannelOutputs = false;
tgaxBase.PathGainsOutputPort = true;
```

Create an individual channel for each STA. Each channel is a clone of `tgaxBase`, but with a different `UserIndex` property, and is stored in cell array `tgax`. The `UserIndex` property of each individual channel creates a unique channel for each user. This example uses a random channel realization for each packet by randomly varying the `UserIndex` property of each transmitted packet.

```
% A cell array stores the channel objects, one per STA.
tgax = cell(1,numTxSTAs);
for u=1:numTxSTAs
    tgax{u} = clone(tgaxBase);
    tgax{u}.NumTransmitAntennas = numTx;
    tgax{u}.UserIndex = u;
end
chInfo = info(tgaxBase);
chFilterCoefficients = chInfo.ChannelFilterCoefficients; % Channel filter coefficients
```

Processing SNR Points

This section tests a number of packets at each SNR point and calculates the probability of misdetecting recovered feedback status.

To parallelize processing of the SNR points, you can use a parfor loop. To enable the use of parallel computing for increased speed comment out the 'for' statement and uncomment the 'parfor' statement below.

```
% Processing SNR Points
ofdmInfo = wlanHEOFDMInfo('HE-LTF',cfgNDP);
numSNR = numel(snrRange); % Number of SNR points
misdetectionProbability = zeros(numTxSTAs,numSNR);
ind = wlanFieldIndices(cfgNDP); % Same for all STAs

%parfor isnr=1:numSNR % Use 'parfor' to speed up the simulation
for isnr=1:numSNR
    % 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);
    rxFeedbackStatus = zeros(numPackets,numTxSTAs);
    chDelay = zeros(1,numTxSTAs);

    for pktIdx=1:numPackets
        rxWaveform = 0;
```

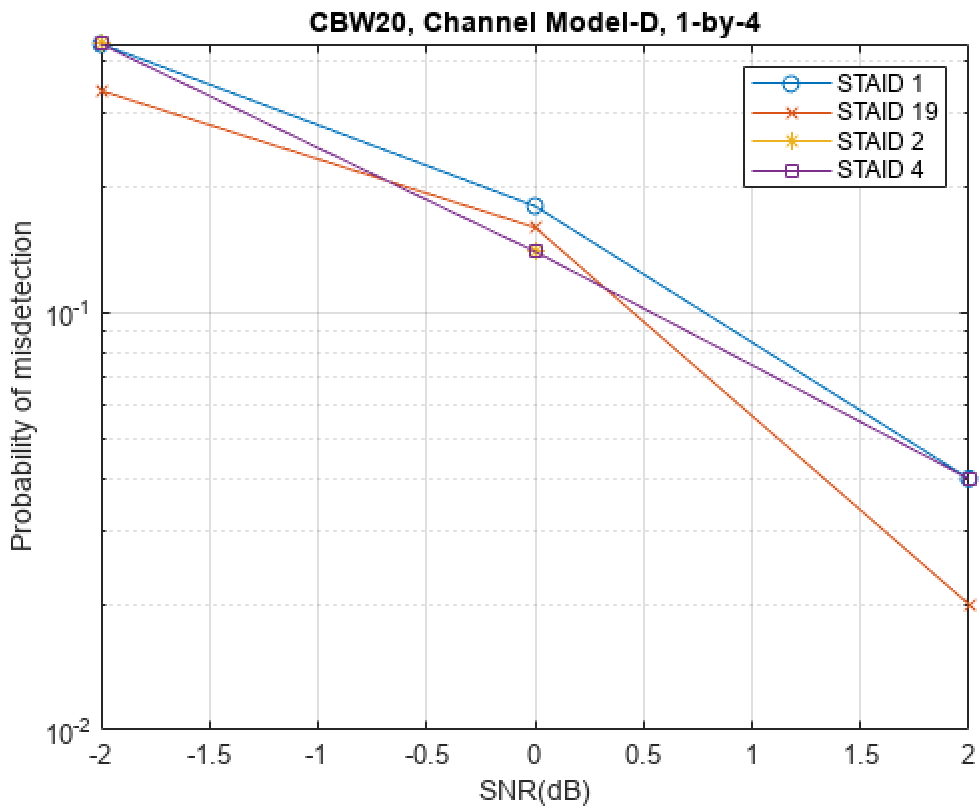
```
        % Generate random channel realization for each packet by varying
        % the UserIndex property of the channel. This assumes all STAs
        % have the same number of transmit antennas.
        chPermutations = randperm(numTxSTAs);
        for u=1:numTxSTAs
            % Generate HE TB feedback NDP waveform for each STA
            txSTA = wlanWaveformGenerator([],cfgSTA{u});

            % Pass waveform through a random TGax Channel
            channelIdx = chPermutations(u);
            reset(tgax{channelIdx}); % New channel realization
            [rxSTA,h] = tgax{channelIdx}([txSTA; zeros(50,size(txSTA,2))]);

            % Perform perfect channel delay estimate to find the start of
            % the packet
            chDelay(u) = channelDelay(h,chFilterCoefficients);

            % Combine uplink waveform from all STAs into one waveform
            rxWaveform = rxWaveform+rxSTA;
        end

        % Synchronize later in time by using the maximum channel delay
        % between all channels as the start of the packet
        pktOffset = max(chDelay(u)); % Packet start index

        % Pass the waveform through AWGN channel. Account for noise
        % energy in unused subcarriers.
        snrVal = snrRange(isnr)-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);
        rxWaveform = awgn(rxWaveform,snrVal);

        % Uplink processing (at the AP)
        rxHELTF = rxWaveform(pktOffset+(ind.HELTF(1):ind.HELTF(2)),:);
        heltfDemod = wlanHEDemodulate(rxHELTF,'HE-LTF',chanBW,cfgNDP.GuardInterval,cfgNDP.HELTFTy

        % Recover feedback status for all STAs
        for u=1:numTxSTAs
            rxFeedbackStatus(pktIdx,u) = wlanHETBNDPFeedbackStatus(heltfDemod,cfgSTA{u});
        end
    end
    % Probability of misdetection per STA
    misdetectionProbability(:,isnr) = 1-sum(rxFeedbackStatus==feedbackStatus(txSTAIndex))/numPac
    disp(['SNR ' num2str(snrRange(isnr)) ' completed']);
end
```

```
SNR -2 completed
SNR 0 completed
SNR 2 completed
SNR 4 completed
SNR 6 completed
```

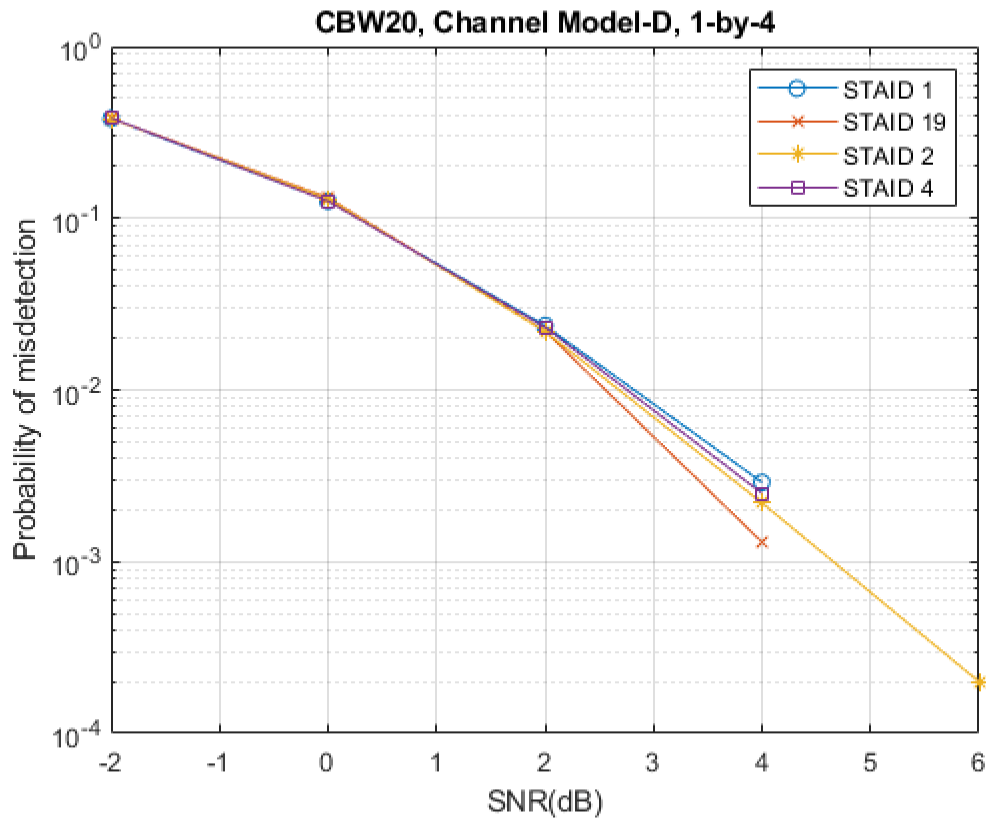
Plot Probability of Misdetection Against SNR

```
markers = 'ox*sd^v><ph+';
numMarkers = numel(markers);
for u=1:numTxSTAs
    semilogy(snrRange,misdetectionProbability(u,:),['-' markers(mod(u-1,numMarkers)+1)]); hold on
end
xlabel('SNR(dB)');
```

```
ylabel('Probability of misdetection');
dataStr = arrayfun(@(x)sprintf('STAID %d',x),STAID(txSTAIndex),'UniformOutput',false);
title(sprintf('%s, Channel %s, %d-by-%d',chanBW,delayProfile,numTx,numRx));
legend(dataStr);
grid on;
```



The number of packets tested at each SNR point depends on `numPackets`. For meaningful results, increase the value of `numPackets`. This figure was created by running a longer simulation with `numPackets`:1e4 and `snrRange`:-2:2:6.



Selected Bibliography

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

2 Montreuil, L. *et al. NDP Short Feedback Design*. IEEE 802.11-17/0044r4, May 2017.

Three-Dimensional Indoor Positioning with 802.11az Fingerprinting and Deep Learning

This example shows how to train a convolutional neural network (CNN) for high-precision positioning by using generated IEEE® 802.11az™ data. Using the trained CNN, you can predict the precise position of multiple stations (STAs) or the room that STAs are located in based on fingerprinting. The example obtains the dataset used for training and validating the CNN in two steps. Channel taps are obtained by using ray tracing for a 3D indoor scenario then 802.11az links are simulated to generate realistic channel state information (CSI).

Introduction

The IEEE 802.11az Wi-Fi™ standard [1 on page 6-103], commonly referred to as next generation positioning (NGP), provides physical layer features that enable enhanced ranging and positioning using classical techniques.

Classical techniques rely on line-of-sight (LOS) conditions to effectively extract temporal information, such as time of arrival (ToA), or spatial information, such as angle of arrival (AoA), from a multipath signal to compute a distance or range between devices. When the range between a minimum of three devices can be measured, trilateration can be used to compute a position estimate. For more information about how to use classical ranging and positioning techniques, see the “802.11az Positioning Using Super-Resolution Time of Arrival Estimation” on page 6-105 example.

Fingerprinting and deep learning techniques can be used in Wi-Fi positioning systems to achieve sub-meter accuracies even in non-line-of-sight (NLOS) multipath environments [2 on page 6-103]. A *fingerprint* typically contains channel state information (CSI), such as a received signal strength indicator (RSSI) or a channel estimate from a received signal, measured at a specific location in an environment [3 on page 6-103].

During the training phase of the network, the example creates a database by sampling the channel fingerprints at multiple known locations in an environment. The network estimates the user location based on a signal received at an unknown location by using the database as a reference.

This example creates a data set of CIR fingerprints by using 802.11az signals in an indoor environment, labeling each fingerprint with its location information. The example trains a CNN to predict STA locations by using a subset of the fingerprints, then evaluates the performance of the trained model by generating predictions of STA locations based on their CIR fingerprint remainder of the data set.

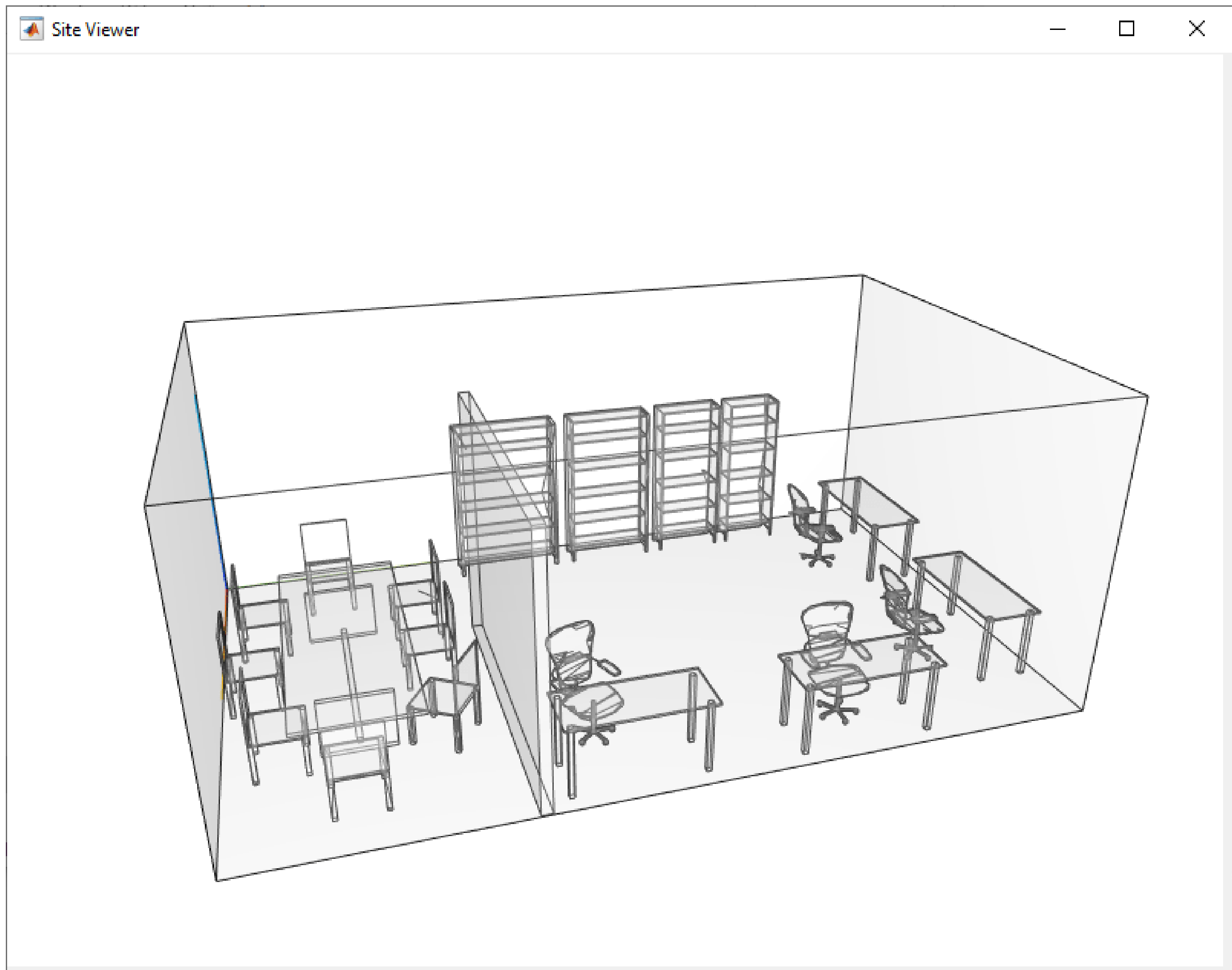
For simplicity, this example uses a small data set, which results in a short simulation time. For more accurate results, use a larger data set. The example provides pretrained models on page 6-99 to show the high levels of performance that can be achieved with more training data.

Simulation Parameters

Indoor Propagation Environment

Generate training data for an indoor office environment specified by the `office.stl` file.

```
mapFileName = "office.stl";
viewer = siteviewer("SceneModel",mapFileName,"Transparency",0.25);
```



The example places four access points (APs) and a number of STAs that you specify in this environment. The example generates the fingerprints based on the propagation channel defined by the environment, then generates the associated channel impulse response by using ray tracing techniques. For more information on ray tracing for indoor environments, see the “Indoor MIMO-OFDM Communication Link using Ray Tracing” example.

AP and STA Parameters

Select the size of the transmit and receive antenna arrays, and the channel bandwidth. These parameters control the quantity of data and its resolution within each fingerprint. Larger antenna arrays produce more channel realizations and more CIRs per fingerprint. A higher bandwidth increases the sample rate of the CIR, capturing it in more detail. Changing these parameters makes the data set incompatible with the pretrained models, as the dimensions of each fingerprint must match the shape of the model input layer.

```
S = RandStream("mt19937ar","Seed",5489); % Set the RNG for reproducibility
RandStream.setGlobalStream(S);
```

```
txArraySize = 4x1 ; % Linear transmit array
rxArraySize = 4x1 ; % Linear receive array
chanBW = CBW40 ;
```

Specify the number of STAs used to map the environment and the distribution of STAs. To distribute STAs uniformly in each dimension as an evenly spaced grid, set `distribution` to `uniform`. To distribute STAs randomly within the environment, set `distribution` to `random`.

When you distribute the STAs uniformly, the example determines the number of STAs by using the value of `staSeparation`, which measures the distance, in meters, between STAs in all dimensions.

When you randomly distribute STAs, specify the number of STAs by setting the value of `numSTAs`.

```
distribution = uniform ;
staSeparation = 0.5 ; % STA separation in meters. This parameter is used only w
numSTAs = 500 ;       % Number of STAs. This parameter is used only when the di
```

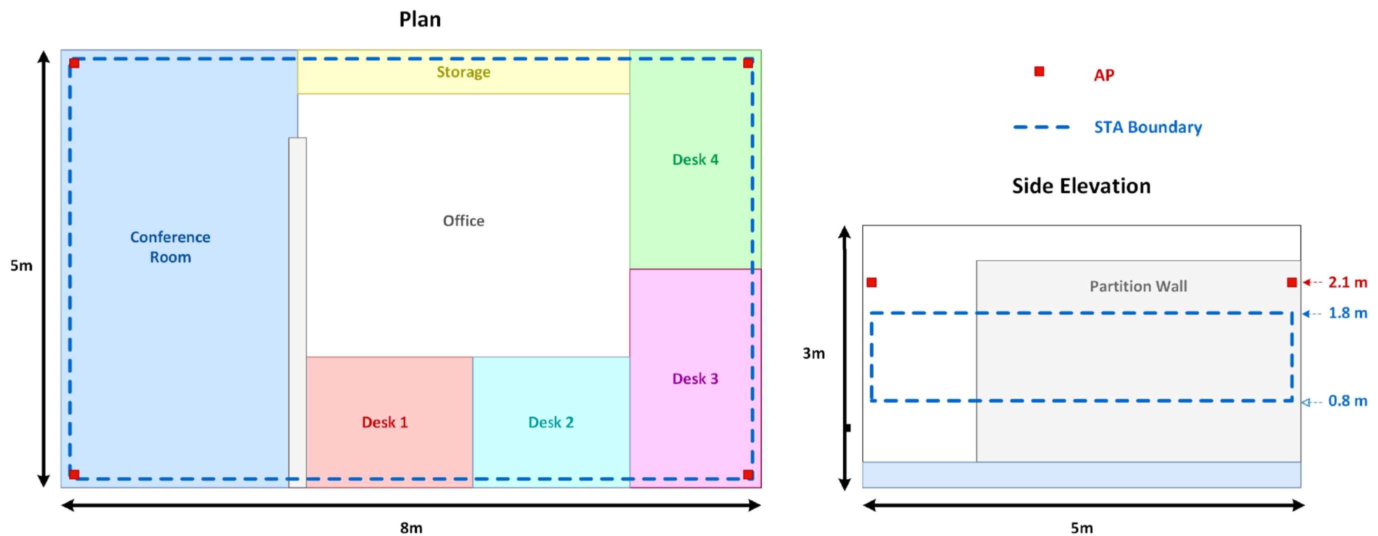
Localization and Positioning

The parameter `task` determines whether the example performs fingerprinting-based localization or 3D positioning [4 on page 6-103]. Fingerprinting-based localization is a multi-class classification task, which predicts the discrete area of the map at which an STA is located. Fingerprinting-based localization is useful for tasks where the detection of the discrete position of an STA, for example, the room of a building or an aisle in a store, is sufficient. 3D positioning is a regression task in which the output of the model is the predicted position of an STA. Positioning can estimate the exact location of the user but can have a higher error for positions across an area when compared to localization.

```
task = localization ;
```

Fingerprinting-based localization task determines the general location of an STA, rather than its precise 3D location. The following diagram shows the layout of the small office with discrete areas used as classes for localization. The red square markers indicate the locations of the APs. The blue dashed box represents the area where the example distributes the STAs during the training process. The example restricts the height of the STAs to a range between 0.8 and 1.8 meters. This range represents a realistic set of values for portable consumer devices. This constraint also minimizes the chance of STAs being placed in unreachable positions.

Office Environment



Training Data Synthesis

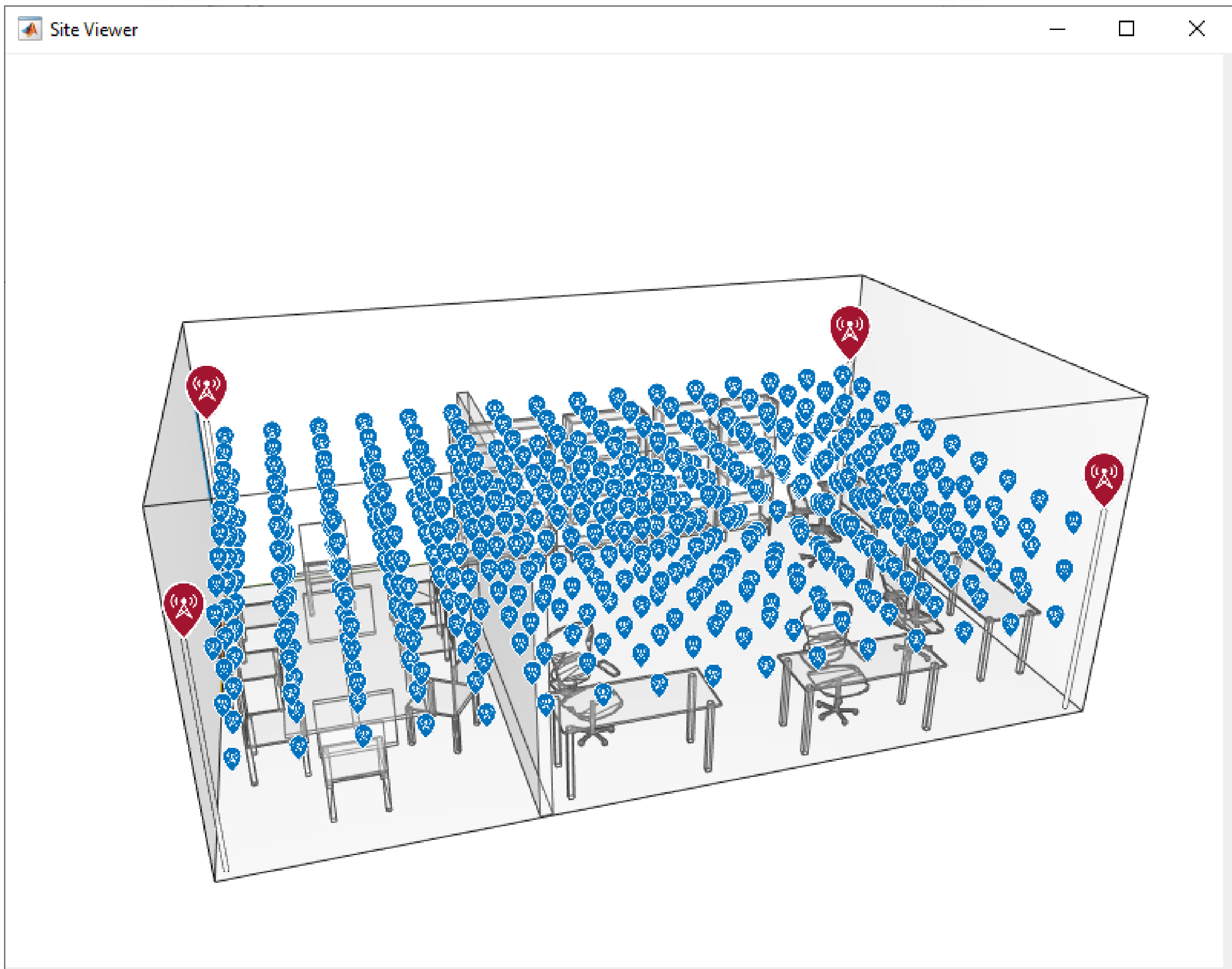
This section shows how to synthesize the data set for training the CNN.

Generate AP and STA Positions

Generate the AP and STA objects and visualize them in the indoor scenario. If you use a file other than `office.stl`, you must adjust the AP and STA locations within the `dlPositioningCreateEnvironment` function to fit the new environment.

```
if distribution == "uniform"
    [APs,STAs] = dlPositioningCreateEnvironment(txArraySize,rxArraySize,staSeparation,"uniform")
else
    [APs,STAs] = dlPositioningCreateEnvironment(txArraySize,rxArraySize,numSTAs,"random");
end
show(APs)
show(STAs,"ShowAntennaHeight",false,"IconSize",[16 16]);
disp(['Simulating a scenario with ',num2str(numel(APs)),' APs and ',char(distribution),'ly distr
```

```
Simulating a scenario with 4 APs and uniformly distributed 480 STAs.
```



Generate Channel Characteristics by Using Ray Tracing Techniques

Set the parameters of the ray propagation model. This example considers only LOS and first-order reflections by setting the `MaxNumReflections` parameter to `1`.Increasing the maximum number of reflections increases the simulation time. To consider only LOS propagation, set the `MaxNumReflections` property to `0`.

```
pm = propagationModel("raytracing", ...
    "CoordinateSystem","cartesian", ...
    "SurfaceMaterial","wood", ...
    "MaxNumReflections",1);
```

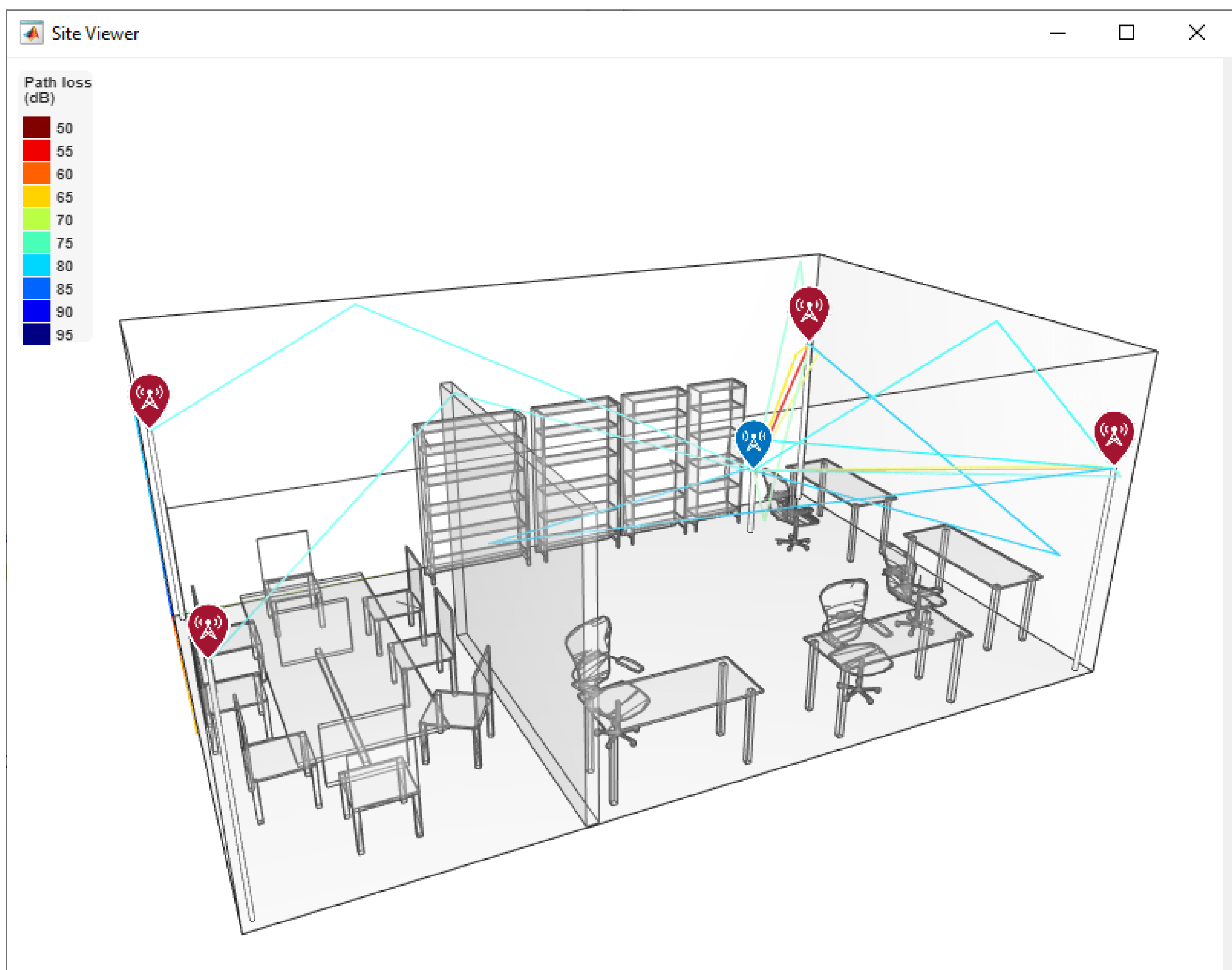
Perform ray tracing analysis for all AP-STA pairs. The `raytrace` function returns the generated rays in a cell array of size N_{AP}-by-N_{STA}, where N_{AP} is the number of APs and N_{STA} is the number of STAs.

```
% Perform ray tracing for all transmitters and receivers in parallel
rays = raytrace(APs,STAs,pm,"Map",mapFileName);
size(rays)
```

```
ans = 1×2

     4   480
```

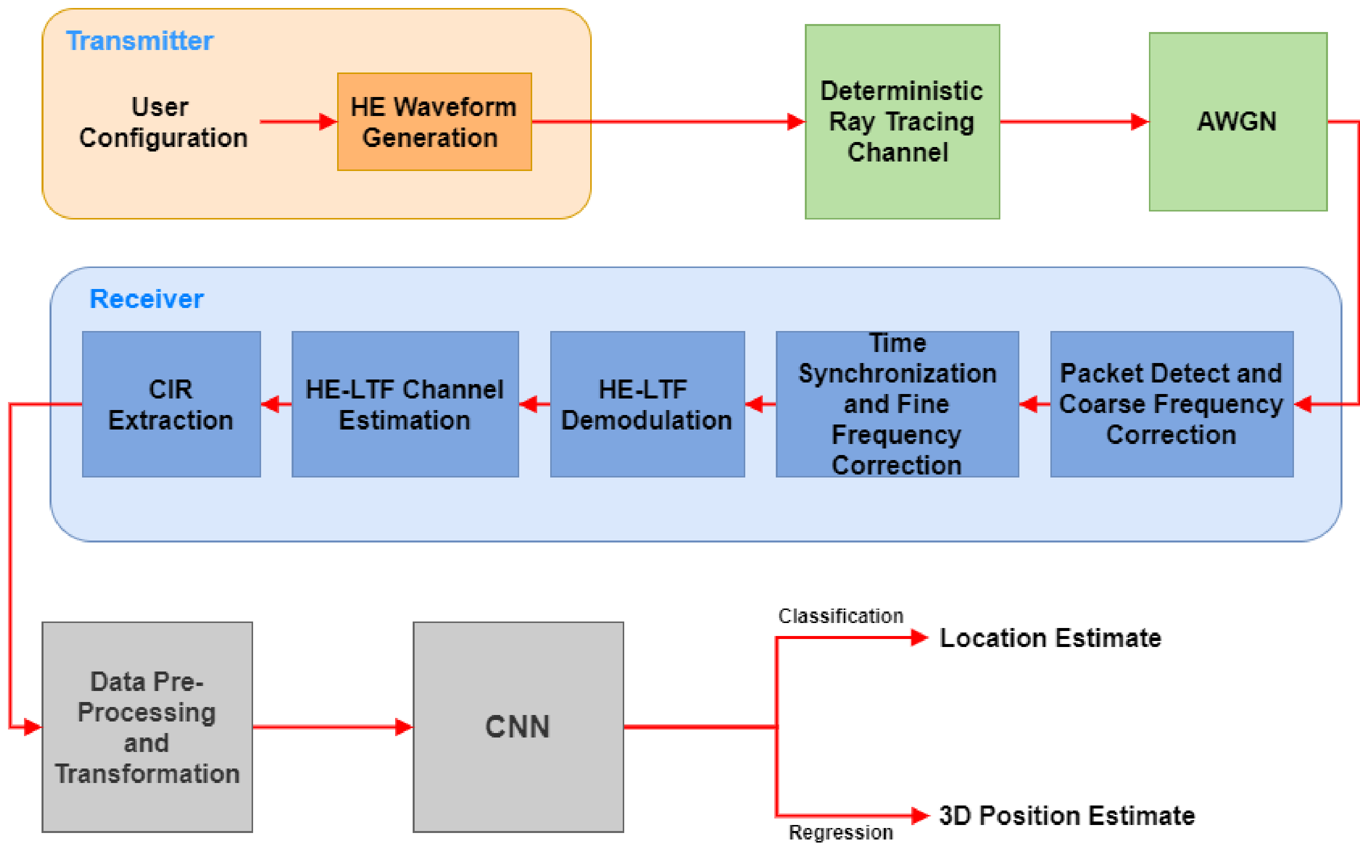
Visualize the calculated rays between all APs and a single STA. The color represents the associated path loss in dB.

```
hide(STAs);
show(STAs(30),'IconSize',[32 32]);
plot([rays{:,30}],'ColorLimits',[50 95]);
```



Generate 802.11az CIR Fingerprint Features and Labels

This section shows how to compute the CIRs for each AP-STA pair by using the calculated rays. This diagram shows the processing chain that the example uses to generate the CIRs.



Each AP transmits an 802.11az packet through a noisy channel, and each STA receives the packet. The example assumes that each STA can differentiate between APs, and that no interference occurs between APs.

Packet reception at a location fails if the path between the location and an AP, or if synchronization fails due to low SNR. In this case, the generated CIR is a vector of zeros.

This example uses the magnitude of each multipath component in the CIR as training data. Therefore, the generated CIRs are real-valued. The example stores the CIRs in a four-dimensional array of size N_s-by-$N_{tx\text{-}rx}$-by-N_{AP}-by-N_r.

- N_s is the number of time-domain samples in the CIR.
- $N_{tx\text{-}rx}$ is the total number of transmit-receive antenna pairs.
- N_{AP} is the number of APs.
- N_r is the number of channel realizations for all SNR points.

The example trains the CNN by combining these features with labels of the STA position or location names.

To simulate variations in the environment, repeat the fingerprinting process under different noise conditions by specifying a range of SNR values.

```
snrs = [10 15 20];
```

Configure the waveform parameters. In particular, set the number of space-time streams (STSs) to the size of the transmit antenna array to ensure that the signal from each transmit antenna contributes to the fingerprint of an STA during channel estimation.

```
cfg = heRangingConfig('ChannelBandwidth',chanBW, ...
    "NumTransmitAntennas",prod(txArraySize), ...
    "SecureHELTF",false, ...
    "GuardInterval",1.6);
cfg.User{1}.NumSpaceTimeStreams = prod(txArraySize);
```

Generate the data set.

```
[features,labels] = dlPositioningGenerateDataSet(rays,STAs,APs,cfg,snrs);
```

```
Generating Dataset: 10% complete.
Generating Dataset: 20% complete.
Generating Dataset: 30% complete.
Generating Dataset: 40% complete.
Generating Dataset: 50% complete.
Generating Dataset: 60% complete.
Generating Dataset: 70% complete.
Generating Dataset: 80% complete.
Generating Dataset: 90% complete.
Generating Dataset: 100% complete.
```

Deep Learning

Deep learning uses neural networks to approximate functions across a diverse range of domains. A CNN is a neural network architecture that uses two-dimensional multichannel images. CNNs preserve and learn features from spatial aspects of the data. The deep learning workflow comprises these steps.

1. Define the training data.
2. Define the neural network model.
3. Configure the learning process.
4. Train the model.
5. Evaluate model performance.

Define Training Data

Neural networks are powerful models that can fit a variety of data. To validate results, split the data set into 80% training data and 20% validation data. Training data is the data that the model learns to fit by adjusting its weighted parameters based on the error of its predictions. Validation data is the data that the example uses to ensure that the model is performing as expected on unseen data and not overfitting the training data.

```
[training,validation] = dlPositioningSplitDataSet(features,labels,0.2);
```

Define Neural Network Model

A typical CNN consists of seven main layers.

1. Input layer, which defines the size and type of the input data
2. Convolutional layer, which performs convolution operations on the layer's input by using a set of filters

3 Batch normalization layer, which prevents unstable gradients by normalizing the activations of a layer
4 Activation (ReLU) layer, which a nonlinear activation function that thresholds the output of the previous functional layer
5 Pooling layer, which extracts and pools feature information
6 Dropout layer, which randomly deactivates a percentage of the parameters of the previous layer during training to prevent overfitting
7 Output layer, which defines the size and type of output data

You can form a deep network by arranging layers from these classes and repeating them in blocks. The CNN in this example consists of four blocks, each with a convolution, batch normalization, ReLU, and average pooling layer. The example adds dropout regularization (20%) before the final layers. This architecture is similar to that of the CNN used in the “Train Convolutional Neural Network for Regression” (Deep Learning Toolbox) example, which predicts the rotated positions of handwritten digit images and demonstrates the diverse applications of deep learning models with minimal changes to their parameters.

Construct the CNN.

```
layers = [

    imageInputLayer(size(training.X, 1:3))

    convolution2dLayer(3,256,"Padding","same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,"Stride",2,"Padding","same")

    convolution2dLayer(3,256,"Padding","same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,"Stride",2,"Padding","same")

    convolution2dLayer(3,256,"Padding","same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,"Stride",2,"Padding","same")

    convolution2dLayer(3,256,"Padding","same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,"Stride",2,"Padding","same")

    dropoutLayer(0.2)];
```

Specify the size of the output layer and the activation function for the output, depending on the selected task.

```
if task == "localization"
    layers = [
        layers
```

```
        fullyConnectedLayer(7)
        softmaxLayer
        classificationLayer];
else % positioning
    layers = [
        layers
        fullyConnectedLayer(3)
        regressionLayer];
end
```

Match the labels given to the network with the expected model output for each task. The features are an image of CIR fingerprints for both tasks.

```
if task == "localization"
    valY = validation.Y.classification;
    trainY = training.Y.classification;
else % positioning
    valY = validation.Y.regression;
    trainY = training.Y.regression;
end
```

Configure Learning Process and Train Model

Set the number of training data samples that the model evaluates during each training iteration. Increase the number of samples when using a larger data set.

```
miniBatchSize = 32;
```

Set the validation frequency so that the network is validated about once per epoch. The network evaluates the unseen samples to test generalization at the end of each validation period.

```
validationFrequency = floor(size(training.X,4)/miniBatchSize);
```

Specify the options that control the training process. The number of epochs controls how many times the example trains the model consecutively on the full training data set. By default, the example trains the model on a GPU if one is available. Using a GPU required Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox).

```
options = trainingOptions("adam", ...
    "MiniBatchSize",miniBatchSize, ...
    "MaxEpochs",3, ...
    "InitialLearnRate", 1e-4,...
    "Shuffle","every-epoch", ...
    "ValidationData",{validation.X,valY'}, ...
    "ValidationFrequency",validationFrequency, ...
    "Verbose",true, ...
    "ResetInputNormalization",true, ...
    "ExecutionEnvironment","auto");
```

Train the model.

```
net = trainNetwork(training.X,trainY',layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
```

```
|==========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|==========================================================================================
|       1 |           1 |       00:00:02 |        6.25% |       40.97% |       2.4116 |       1.6
|       1 |          36 |       00:00:21 |       53.12% |       60.42% |       1.1444 |       1.0
|       2 |          50 |       00:00:28 |       62.50% |              |       1.1303 |
|       2 |          72 |       00:00:40 |       68.75% |       62.85% |       0.8643 |       1.0
|       3 |         100 |       00:00:53 |       62.50% |              |       0.8839 |
|       3 |         108 |       00:00:58 |       78.12% |       70.14% |       0.6926 |       0.8
|==========================================================================================
Training finished: Max epochs completed.
```

Evaluate Model Performance

Investigate the model performance by examining predicted values for the validation data. Predict labels by passing the validation set features through the network, then evaluate the performance of the network by comparing the predicted labels with the validation set labels.

```
if task == "localization"
    YPred = net.classify(validation.X);
else % positioning
    YPred = net.predict(validation.X);
end
```

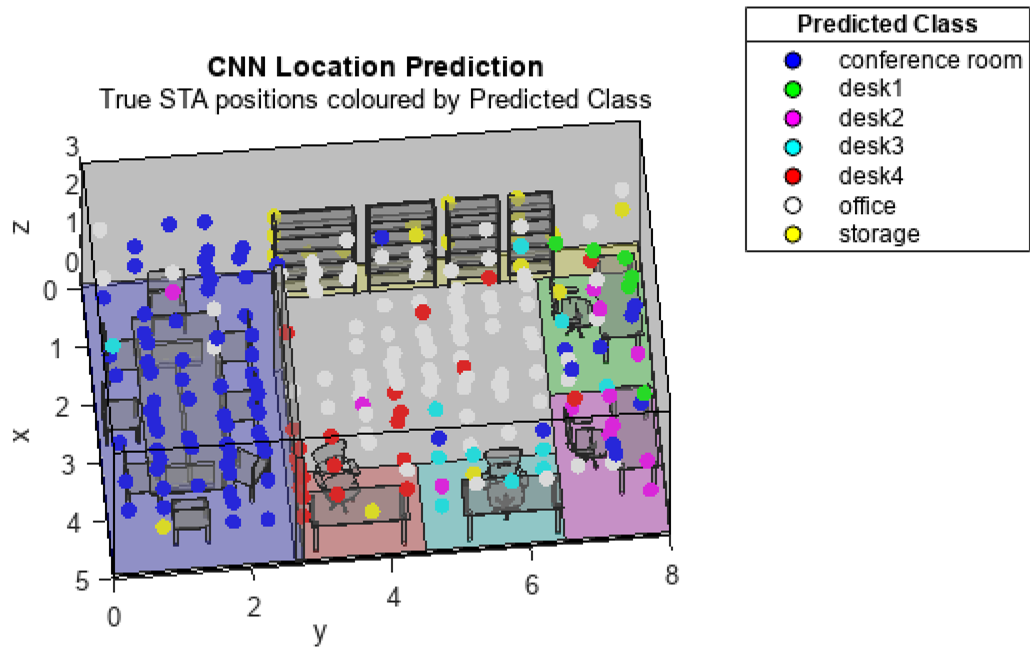
Generate a visual and statistical view of the results.

For localization, the `dlPositioningPlotResults` function generates a three-dimensional map, which displays the true locations of STAs. The color assigned to each STA denotes its predicted location. The function also generates a confusion matrix in which the rows correspond to the predicted class and the columns correspond to the true class. The diagonal cells correspond to observations that are correctly classified. The off-diagonal cells correspond to incorrectly classified observations. If the network performs well, elements in the main diagonal are significantly larger than the other matrix elements.

For positioning, the color assigned to each STA denotes the distance error of the predicted position. The function also generates a cumulative distribution function (CDF). The y-axis measures the proportion of the data for which the measured distance error is less than or equal to the corresponding value on the x-axis.

This example uses a small data set and limited training period, which produces modest results. You can obtain more accurate results by using a CNN pretrained with a large data set on page 6-99.

```
metric = dlPositioningPlotResults(mapFileName,validation.Y,YPred,task);
```


CNN Location Prediction
True STA positions coloured by Predicted Class
Predicted Class
conference room
desk1
desk2
desk3
desk4
office
storage
z
x
y
0
1
2
3
4
5
6
8

Accuracy: 70.8333%

True Class \ Predicted Class	conference_room	desk1	desk2	desk3	desk4	office	storage
conference_room	79		1	2		7	3
desk1	5	5	2	1	1	5	1
desk2	5	3	8	1	1	7	
desk3	2		3	9		5	1
desk4					15	9	1
office	2		2	1	9	74	3
storage						1	14

Pretrained Model

Download a CNN for positioning and localization pretrained with a large data set from https://www.mathworks.com/supportfiles/spc/wlan/Positioning80211azExample/R2021b/pretrained_networks_R2021b.zip. This zip file contains two MAT files:

- localization_cbw40_4x4.mat - CNN trained for localization
- positioning_cbw40_4x4.mat - CNN trained for positioning

All models were trained for 100 epochs with a `miniBatchSize` value of 256 on a data set of fingerprints from a uniform distribution of STAs with a `staSeparation` value of `0.1`. The bandwidth was 40 MHz bandwidth with four-antenna linear arrays in the transmitter and receiver (4×4 MIMO).

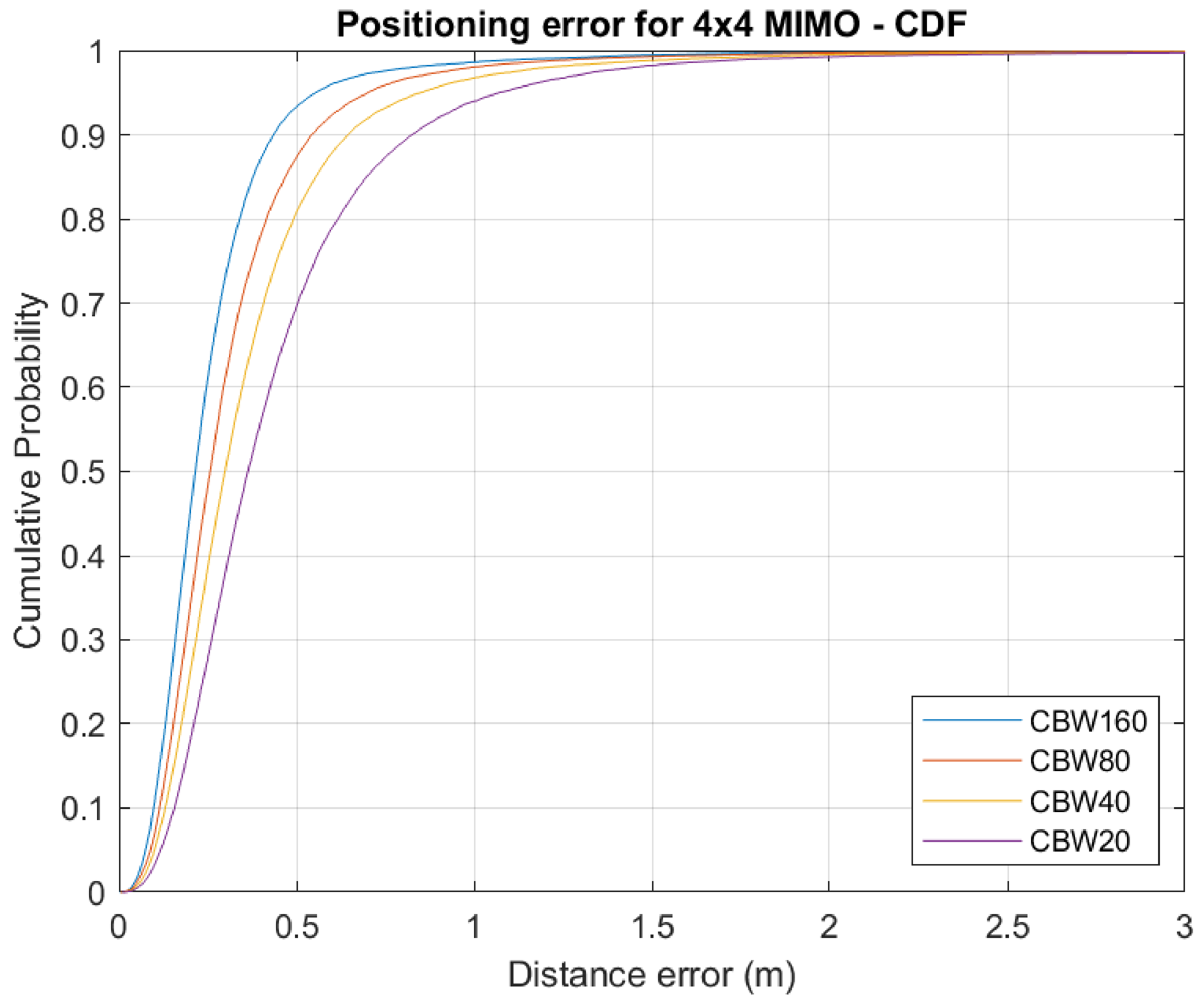
To explore the performance of these pretrained models, download and extract the zip file, load the appropriate MAT file into the workspace, and run the Evaluate Model Performance on page 6-97 section again.

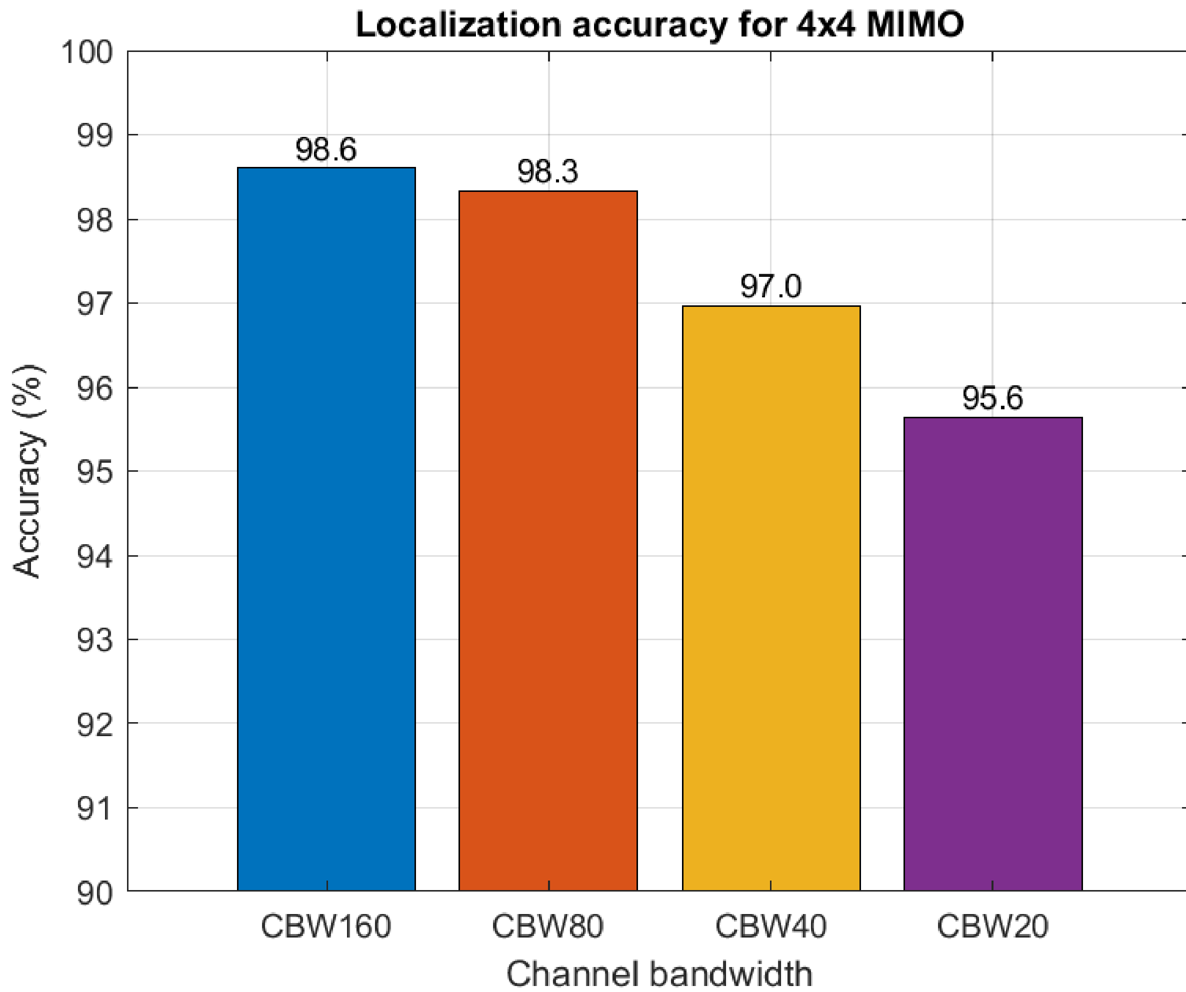
Effect of Antenna Array Size and Bandwidth

The pretrained models show that an increased number of STAs and training epochs improve performance. This section shows the effect of channel bandwidth and size of antenna arrays on performance.

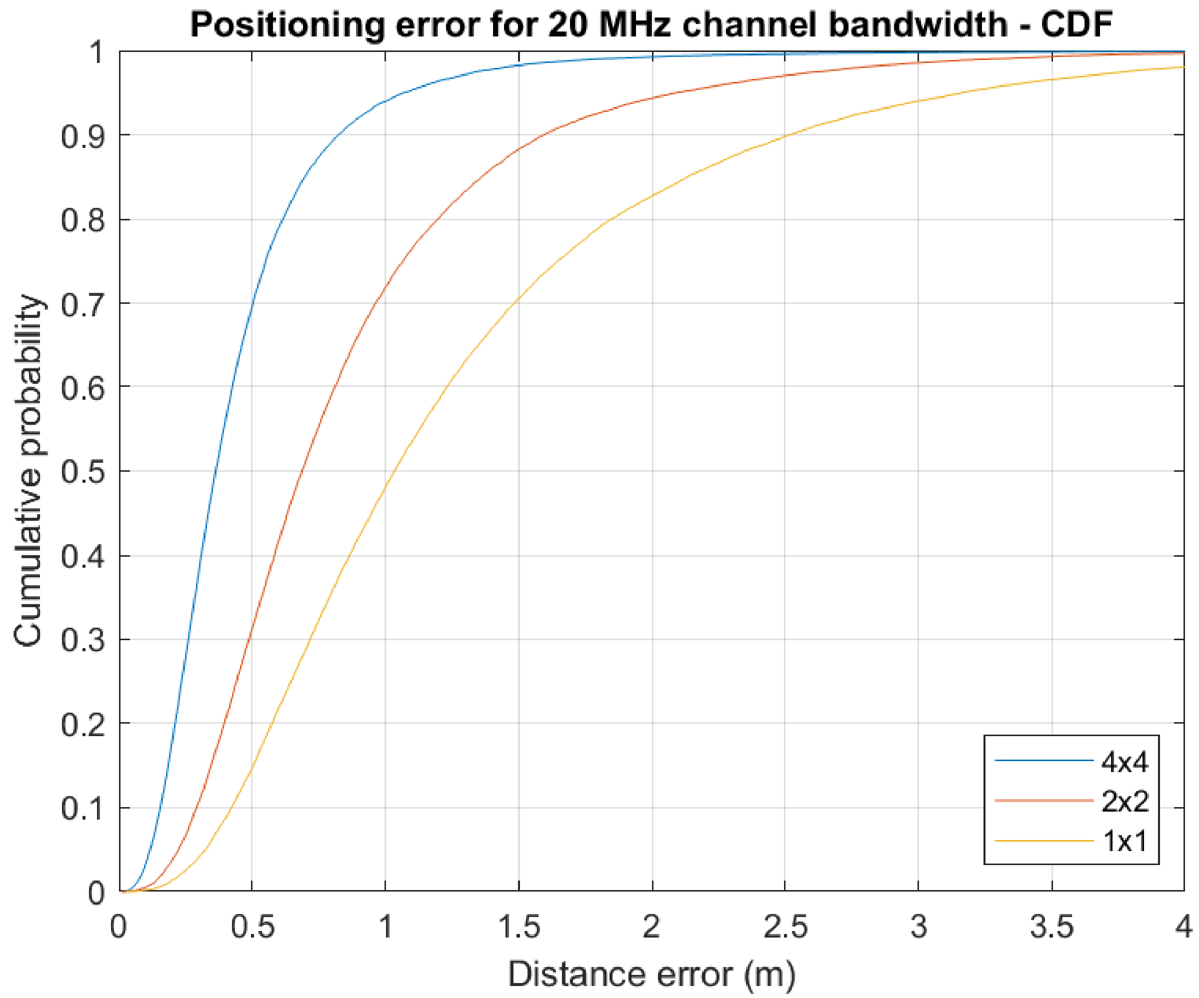
These figures show the impact of channel bandwidth on positioning and localization performance. These results were generated by training a model for 100 epochs with a `miniBatchSize` value of 256 with a data set compiled from a uniform distribution of STAs sampled at intervals of 0.1 meters,

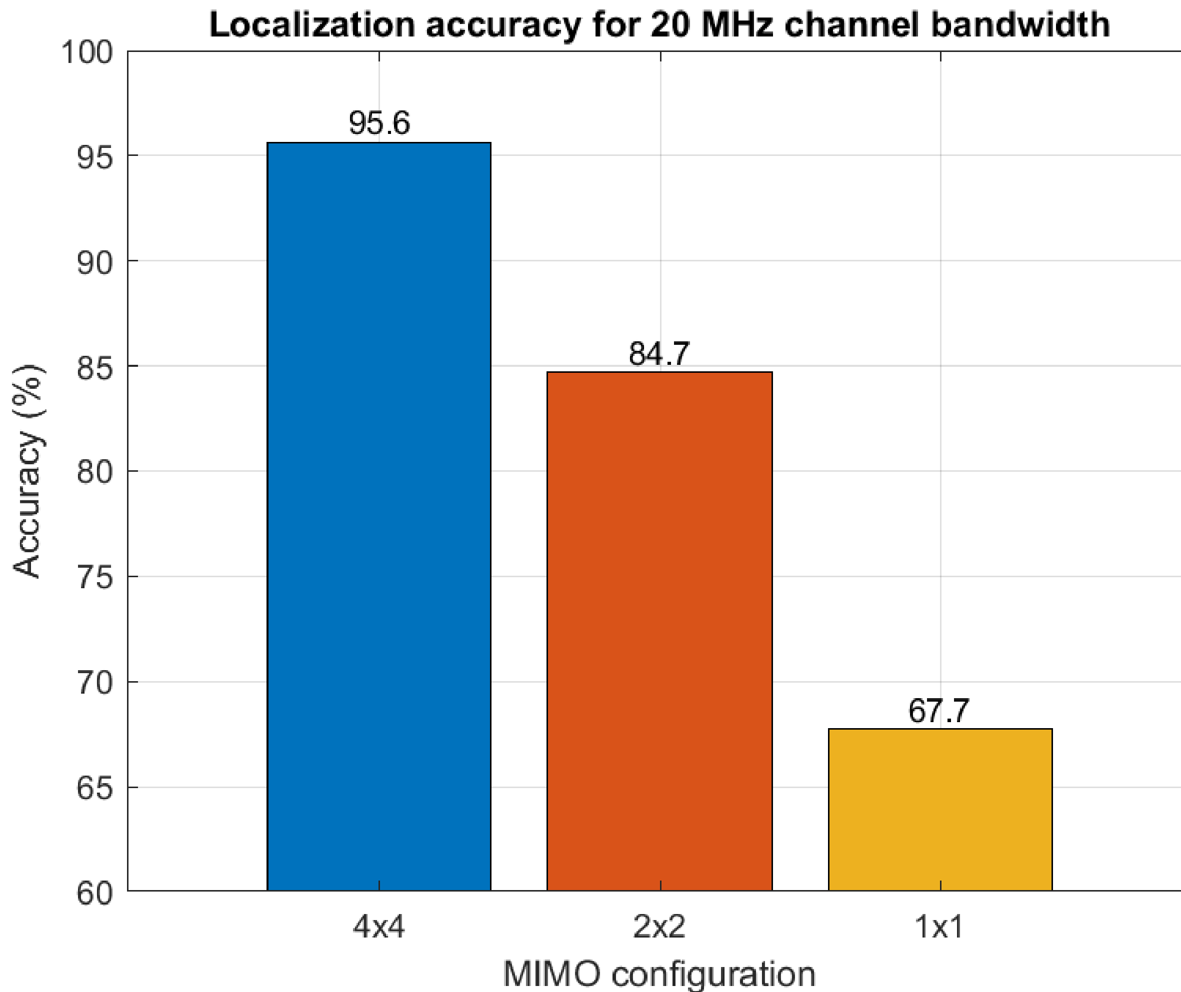
in which each STA and AP contained a four-element linear antenna array. Using a larger bandwidth results in more accurate estimation as a higher sampling rate increases the resolution of CIRs.





These figures show the impact of the number of transmit and receive antennas. These results were generated using a bandwidth of 20 MHz. The accuracy increases for larger antenna arrays due to the presence of more channel information. For example, a 4×4 channel contains 16 channel realizations, whereas a 2×2 channel contains only four. As expected, the CNN performs better when trained with more data.


Positioning error for 20 MHz channel bandwidth - CDF
Cumulative probability
Distance error (m)
4x4
2x2
1x1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.5
1
1.5
2
2.5
3
3.5
4



These results were generated with data samples which were not used to train the CNNs. If data samples used to train the CNN are also used to evaluate performance the results may differ.

References

1 IEEE P802.11az™/D2.6 Draft Standard for Information technology— Telecommunications and information exchange between systems Local and metropolitan area networks— Specific requirements - Amendment 3: Enhancements for positioning.

2 Kokkinis, Akis, Loizos Kanaris, Antonio Liotta, and Stavros Stavrou. “RSS Indoor Localization Based on a Single Access Point.” Sensors 19, no. 17 (August 27, 2019): 3711. https://doi.org/10.3390/s19173711.

3 Wang, Xuyu, Lingjun Gao, Shiwen Mao, and Santosh Pandey. “CSI-Based Fingerprinting for Indoor Localization: A Deep Learning Approach.” IEEE Transactions on Vehicular Technology, 2016, 763–776. https://doi.org/10.1109/TVT.2016.2545523.

4 Groves, Paul D. Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems. Boston: Artech House, 2008.

802.11az Positioning Using Super-Resolution Time of Arrival Estimation

This example shows how to estimate the position of a station (STA) in a multipath environment by using a time-of-arrival-based (ToA-based) positioning algorithm defined in the IEEE® 802.11az™ Wi-Fi™ standard. The example estimates the ToA by using a multiple signal classification (MUSIC) super-resolution approach, then estimates the two-dimensional position of a STA by using trilateration. The example then evaluates and compares the performance of the positioning algorithm at multiple signal-to-noise ratio (SNR) points.

Introduction

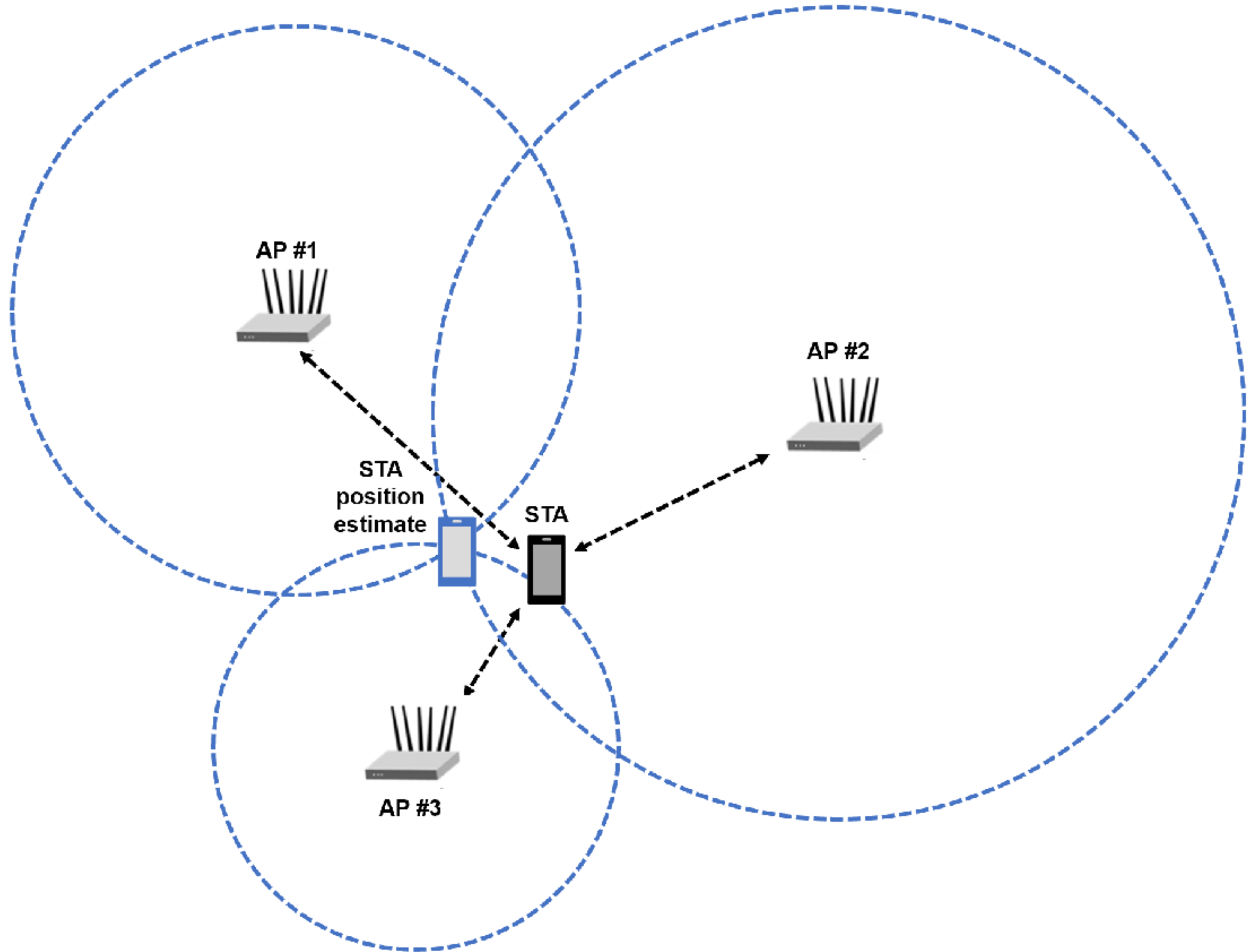
The 802.11az standard [1 on page 6-118], commonly referred to as next generation positioning (NGP), enables a STA to identify its position relative to multiple access points (APs). This standard supports two high-efficiency (HE) ranging physical layer (PHY) protocol data unit (PPDU) formats:

- HE ranging null data packet (NDP)
- HE trigger-based (TB) ranging NDP

The HE ranging NDP and HE TB ranging NDP are the respective analogues of the HE sounding NDP and HE TB feedback NDP PPDU formats, as defined in the 802.11ax™ standard. For more information on these HE PPDU formats, see [2] on page 6-118.

The HE ranging NDP supports the positioning of one or more users with an optional secure HE long training field (HE-LTF) sequence. The single-user HE ranging waveform contains HE-LTF symbols for a single user, which also support an optional secure HE-LTF sequence. The multi-user HE ranging waveform permits only secure HE-LTF symbols for multiple users. To improve position estimation accuracy, single-user and multi-user waveforms can contain multiple repetitions of the HE-LTF symbols. To parameterize and generate HE ranging NDPs, see the “802.11az Waveform Generation” on page 1-48 example.

This example simulates an 802.11az network consisting of a STA and multiple APs. To estimate the position of a STA, the network requires a minimum of three APs. The example simulates a ranging measurement exchange for each STA-AP pair, then trilaterates the position of the STA by using these measurements. The example repeats this simulation for multiple iterations and SNR points. This diagram shows the positioning process in a network with one STA and three APs.

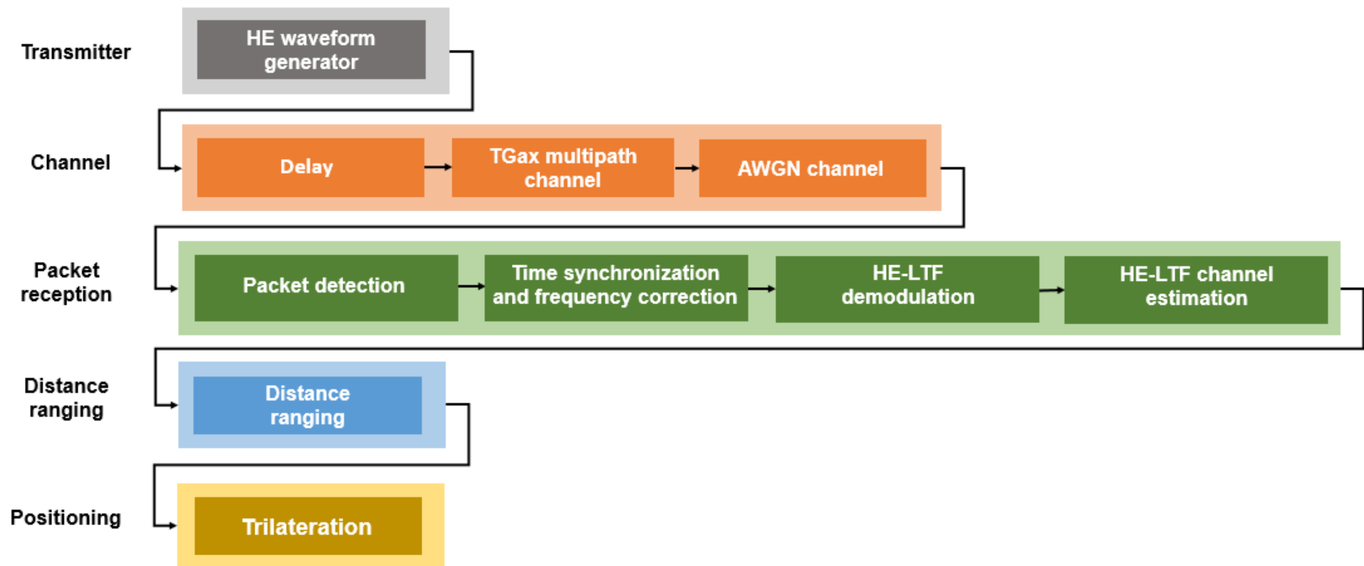


Packet Transmission and Reception

This example models the measurement exchange between the STA and APs by performing these steps.

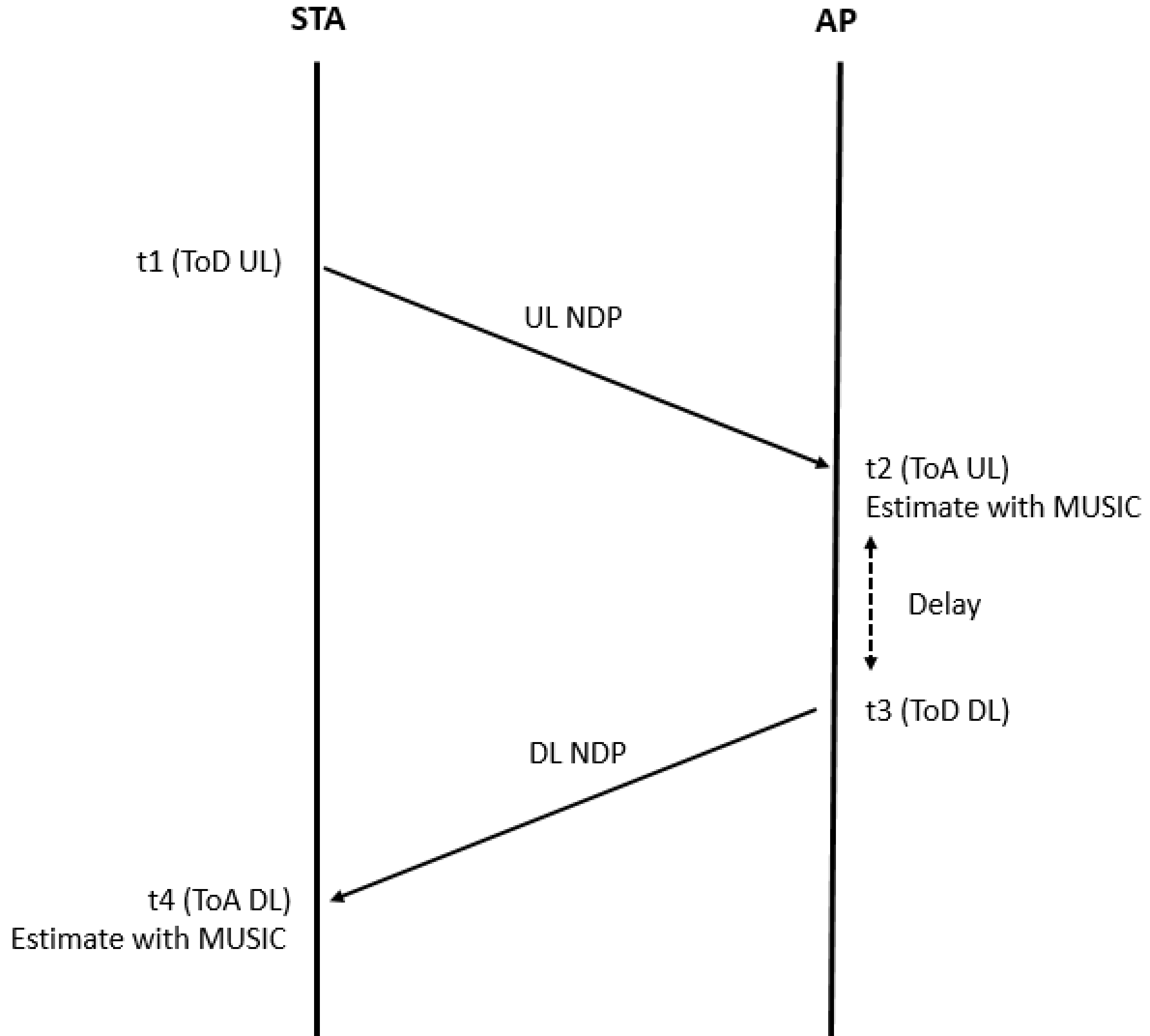
1. Generate a ranging NDP
2. Delay the NDP according to a randomly generated distance between the STA and AP, adding fractional and integer sample delay
3. Pass the waveform through an indoor TGax channel. The example models different channel realizations for different packets
4. Add additive white Gaussian noise (AWGN) to the received waveform. The example uses the same SNR value for all the links between STA and APs
5. Perform synchronization and frequency correction on the received waveform
6. Demodulate the HE-LTF
7. Estimate the channel frequency response from the HE-LTF
8. Estimate the distance by using the MUSIC super-resolution algorithm
9. Combine distance estimates from other STA-AP pairs and trilaterate the position of the STA

This figure illustrates the processing for each STA-AP link.



Distance Ranging

The example performs a distance ranging measurement between the STA and each AP by capturing the timestamps of the NDP. The STA records the time t_1 (UL ToD) at which it transmits the uplink NDP (UL NDP). The AP then captures the time t_2 (UL ToA) at which it receives the UL NDP and records the time t_3 (DL ToD) at which it transmits the downlink NDP (DL NDP). The STA then captures the time t_4 (DL ToA) at which it receives the DL NDP. This diagram illustrates the measurement sounding phase between a STA and a single AP [1 on page 6-118].



The example estimates T_{RTT} the round trip time (RTT), by combining these timestamps.

$$T_{RTT} = (t_4 - t_1) - (t_3 - t_2)$$

The example then computes the distance d between the STA and AP by using this equation.

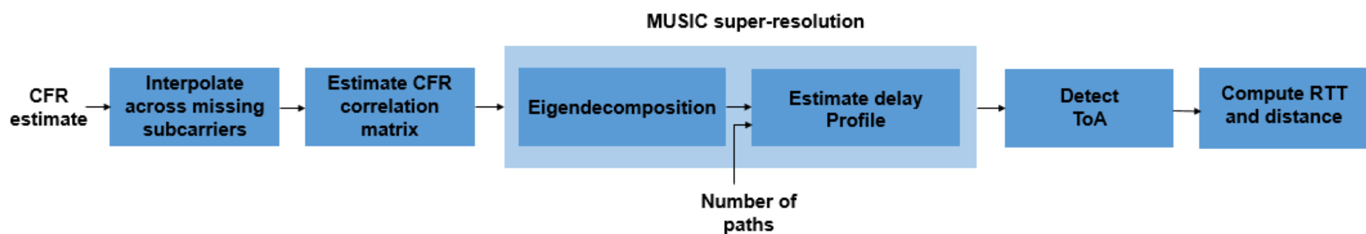
$d = \frac{T_{RTT}}{2}c$, where c is speed of light

The example estimates t_2 and t_4 by using MUSIC super-resolution. To compute these estimates, the example performs these steps [3 on page 6-118].

1. Interpolate across missing subcarriers from the channel frequency response (CFR), assuming uniform subcarrier spacing
2. Estimate the CFR correlation matrix
3. Decorrelate the multipaths by using spatial smoothing

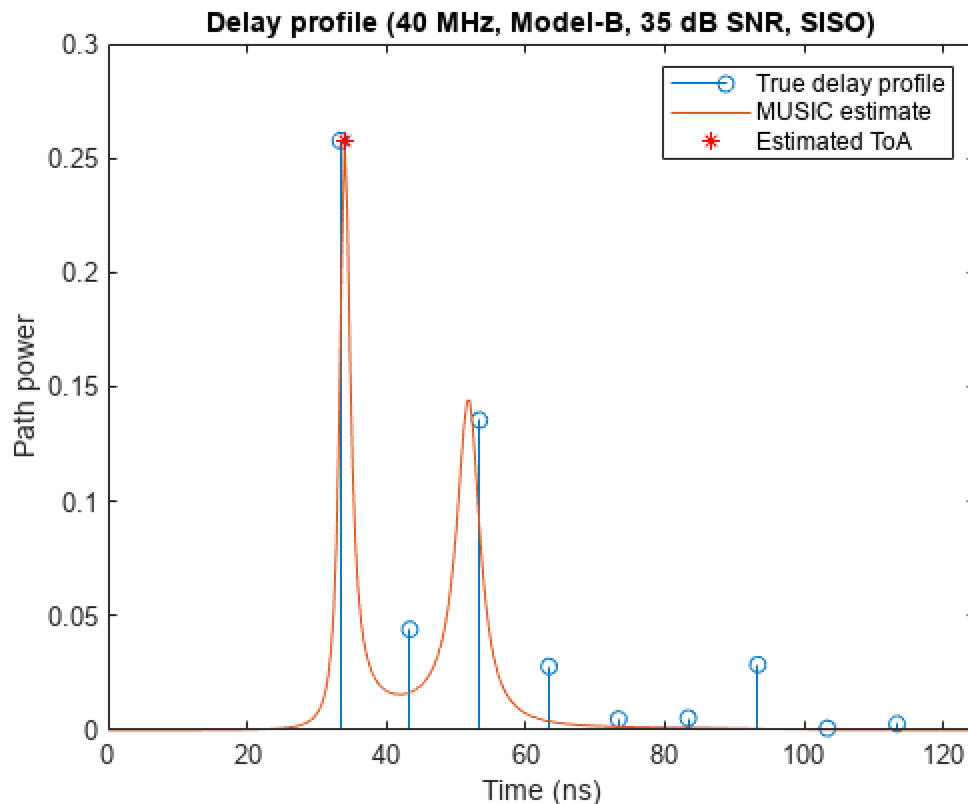
4 Improve the correlation matrix estimate by performing forward-backward averaging. The example assumes that CFR estimates from multiple spatial streams are different CFR snapshots, and uses all the snapshots for correlation matrix estimation.
5 Run the MUSIC algorithm. Perform eigendecomposition on the correlation matrix to separate it into signal and noise subspaces. Estimate the time-domain delay profile by finding all instances where signal and noise subspaces are orthogonal. The example assumes that the precise signal subspace dimension, equal to the number of multipaths, is known.
6 Determine the ToA by finding the first peak of the recovered multipaths in the estimated delay profile, assumed to be the direct-line-of-sight (DLOS) path

This diagram illustrates the distance ranging process.



This plot compares the true multipath delay profile and the MUSIC estimated delay profile for a single 802.11az link simulation.

```
heRangingPlotDelayProfile()
```



Simulation Parameters

This example performs a ranging and positioning simulation for multiple iterations and SNR points. At each iteration, the AP and the STA exchange multiple uplink and downlink packets. The example estimates the ranging error between the AP and the STA for each iteration by comparing the estimated distance between the AP and the STA with the known distance.

Specify the number of iterations, SNR points, and APs in the networks. To estimate the position of a STA, the network requires a minimum of three APs. For each iteration, use a different random set of AP positions, a different channel realization, and a different AWGN profile. The example generates a cumulative distribution function (CDF) for the absolute ranging error by using the ranging measurements from all iterations and all STA-AP pairs.

```
numIterations = 50; % Number of iterations
snrRange = 15:10:35; % SNR points, in dB
numAPs = 3; % Number of APs
```

802.11az Waveform Configuration

Configure waveform generators for each AP and the STA.

```
chanBW = CBW20; % Channel bandwidth
numTx  = 2; % Number of transmit antennas
numRx  = 2; % Number of receive antennas
numSTS = 2; % Number of space-time streams
numLTFRepetitions = 3; % Number of HE-LTF repetitions
```

Configure the HE ranging NDP parameters of the STA.

```
cfgSTABase = heRangingConfig;
cfgSTABase.ChannelBandwidth = chanBW;
cfgSTABase.NumTransmitAntennas = numTx;
cfgSTABase.SecureHELTF = true;
cfgSTABase.User{1}.NumSpaceTimeStreams = numSTS;
cfgSTABase.User{1}.NumHELTFRepetition = numLTFRepetitions;
cfgSTABase.GuardInterval = 1.6;
```

Configure the HE ranging NDP parameters of the APs.

```
cfgAPBase = cell(1,numAPs);
for iAP = 1:numAPs
    cfgAPBase{iAP} = heRangingConfig;
    cfgAPBase{iAP}.ChannelBandwidth = chanBW;
    cfgAPBase{iAP}.NumTransmitAntennas = numTx;
    cfgAPBase{iAP}.SecureHELTF = true;
    cfgAPBase{iAP}.User{1}.NumSpaceTimeStreams = numSTS;
    cfgAPBase{iAP}.User{1}.NumHELTFRepetition = numLTFRepetitions;
    cfgAPBase{iAP}.GuardInterval = 1.6;
end
```

```
ofdmInfo = wlanHEOFDMInfo('HE-LTF',chanBW,cfgSTABase.GuardInterval);
sampleRate = wlanSampleRate(chanBW);
```

Channel Configuration

Configure the WLAN TGax multipath channel by using the `wlanTGaxChannel` System object™. This System object can generate a channel with a dominant direct path, in which the DLOS path is the strongest path, or a channel with a non-dominant direct path, for which the DLOS path is present, but not the strongest path.

```
delayProfile = Model-B; % TGax channel multipath delay profile

carrierFrequency = 5e9; % Carrier frequency, in Hz
speedOfLight = physconst('lightspeed');

chanBase = wlanTGaxChannel;
chanBase.DelayProfile = delayProfile;
chanBase.NumTransmitAntennas = numTx;
chanBase.NumReceiveAntennas = numRx;
chanBase.SampleRate = sampleRate;
chanBase.CarrierFrequency = carrierFrequency;
chanBase.ChannelBandwidth = chanBW;
chanBase.PathGainsOutputPort = true;
chanBase.NormalizeChannelOutputs = false;
```

Get channel filter delay and the number of paths

```
chBaseInfo = info(chanBase);
chDelay = chBaseInfo.ChannelFilterDelay;
numPaths = size(chBaseInfo.PathDelays,2);
```

Ranging Measurement

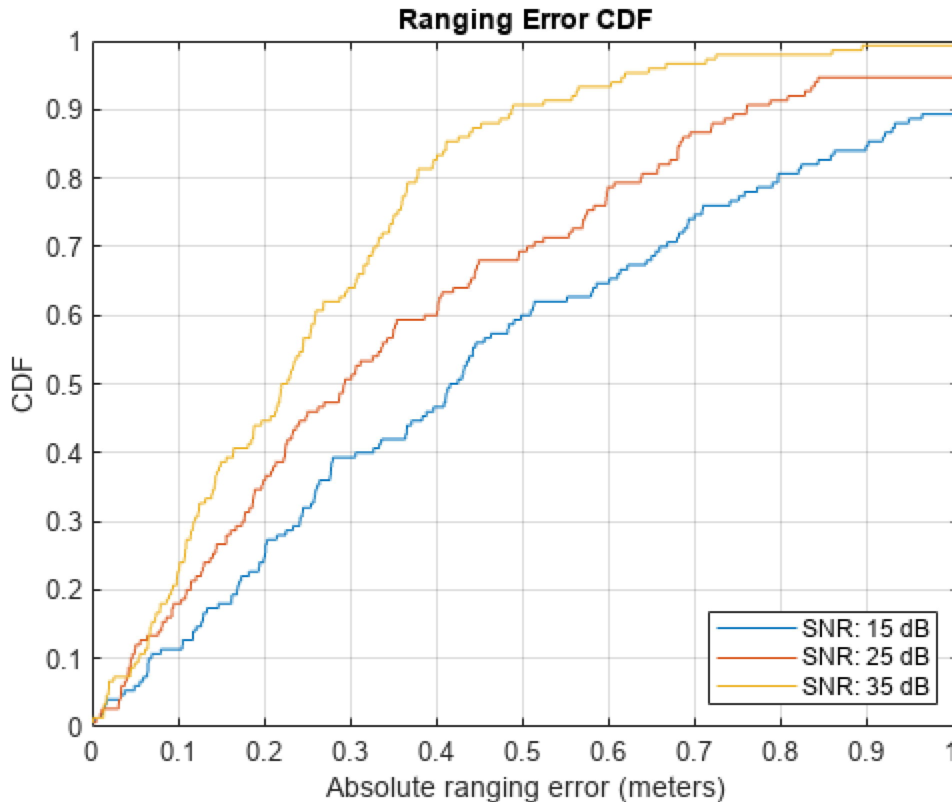
Run a ranging simulation with multiple iterations for all STA-AP pairs. Display the ranging mean absolute error (MAE) and the ranging error CDF for each SNR point.

```
delayULDL = 16e-6; % Time delay between UL NDP ToA and DL NDP ToD, in seconds

numSNR = numel(snrRange);
distEst = zeros(numAPs,numIterations,numSNR);  % Estimated distance
distance = zeros(numAPs,numIterations,numSNR); % True distance
positionSTA = zeros(2,numIterations,numSNR);   % Two-dimensional position of the STA
positionAP= zeros(2,numAPs,numIterations,numSNR); % Two-dimensional positions of the APs
per = zeros(numSNR,1); % Packet error rate (PER)

%parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
for isnr = 1:numSNR

    % Use a separate channel and waveform configuration object for each parfor stream
    chan = chanBase;
    cfgAP = cfgAPBase;
    cfgSTA = cfgSTABase;

    % Initialize ranging error and total failed packet count variables
    rangingError = 0;
    failedPackets = 0;
```

```
        % Set random substream index per iteration to ensure that each
        % iteration uses a repeatable set of random numbers
        stream = RandStream('combRecursive','Seed',123456);
        stream.Substream = isnr;
        RandStream.setGlobalStream(stream);

        % Define the SNR per active subcarrier to account for noise energy in nulls
        snrVal = snrRange(isnr) - 10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

        for iter = 1:numIterations

            % Gennerate random AP positions
            [positionSTA(:,iter,isnr),positionAP(:,:,iter,isnr),distanceAllAPs] = heGeneratePositions
            distance(:,iter,isnr) = distanceAllAPs;

            % Range-based delay
            delay = distance(:,iter,isnr)/speedOfLight;
            sampleDelay = delay*sampleRate;

            % Loop over the number of APs
            for ap = 1:numAPs

                linkType = ["Uplink","Downlink"];

                % ToD of UL NDP (t1)
                todUL = randsrc(1,1,0:1e-9:1e-6);

                % Loop for both UL and DL transmission
                numLinks = numel(linkType);
                txTime = zeros(1,numLinks);

                for l = 1:numLinks
                    if linkType(l) == "Uplink" % STA to AP
                        cfgSTA.UplinkIndication = 1; % For UL
                        % Generate a random secure HE-LTF sequence for the exchange
                        cfgSTA.User{1}.SecureHELTFSequence = dec2hex(randsrc(1,10,(0:15)))';
                        cfg = cfgSTA;
                    else % AP to STA
                        % Generate a random secure HE-LTF sequence for the exchange
                        cfgAP{ap}.User{1}.SecureHELTFSequence = dec2hex(randsrc(1,10,(0:15)))';
                        cfg = cfgAP{ap}; % For DL
                    end

                    % Set different channel for UL and DL, assuming that the channel is not reciproca
                    reset(chan)

                    % Generate HE Ranging NDP transmission
                    tx = heRangingWaveformGenerator(cfg);

                    % Introduce time delay (fractional and integer) in the transmit waveform
                    txDelay = heDelaySignal(tx,sampleDelay(ap));

                    % Pad signal and pass through multipath channel
                    txMultipath = chan([txDelay;zeros(50,cfg.NumTransmitAntennas)]);

                    % Pass waveform through AWGN channel
                    rx = awgn(txMultipath,snrVal);
```

```
                % Perform synchronization and channel estimation
                [chanEstActiveSC,integerOffset] = heRangingSynchronize(rx,cfg);

                % Estimate the transmission time between UL and DL
                if ~isempty(chanEstActiveSC) % If packet detection is successful

                    % Estimate fractional delay with MUSIC super-resolution
                    fracDelay = heRangingTOAEstimate(chanEstActiveSC,ofdmInfo.ActiveFFTIndices,
                                                     ofdmInfo.FFTLength,sampleRate,numPaths);

                    integerOffset = integerOffset - chDelay; % Account for channel filter delay
                    intDelay = integerOffset/sampleRate; % Estimate integer time delay
                    txTime(l) = intDelay + fracDelay; % Transmission time

                else % If packet detection fails
                    txTime(l) = NaN;
                end

            end

            if ~any(isnan(txTime)) % If packet detection succeeds

                % TOA of UL waveform (t2)
                toaUL = todUL + txTime(1);

                % Time of departure of DL waveform (t3)
                todDL = toaUL + delayULDL;

                % TOA DL waveform (t4)
                toaDL = todDL + txTime(2);

                % Compute the RTT
                rtt = (toaDL-todUL) - (todDL-toaUL);

                % Estimate the distance between the STA and AP
                distEst(ap,iter,isnr) = (rtt/2)*speedOfLight;
                % Accumulate error to MAE
                rangingError = rangingError + abs(distanceAllAPs(ap) - distEst(ap,iter,isnr));

            else % If packet detection fails
                distEst(ap,iter,isnr) = NaN;
                failedPackets = failedPackets + 1;
            end

        end
    end
    mae = rangingError/((numAPs*numIterations) - failedPackets); % MAE for successful packets
    per(isnr) = failedPackets/(numAPs*numIterations); % PER
    if(per(isnr) > 0.01) % Use only successful packets for ranging and positioning
        warning('wlan:discardPacket','At SNR = %d dB, %d%% of packets were discarded',snrRange(is
    end
    disp(['At SNR = ',num2str(snrRange(isnr)),' dB, ','Ranging mean absolute error = ',num2str(ma
end

At SNR = 15 dB, Ranging mean absolute error = 0.57082 meters.
At SNR = 25 dB, Ranging mean absolute error = 0.40224 meters.
At SNR = 35 dB, Ranging mean absolute error = 0.25439 meters.
```

```
% Reshape to consider all packets within one SNR point as one dataset
rangingError = reshape(abs(distance - distEst),[numAPs*numIterations,numSNR]);
hePlotErrorCDF(rangingError,snrRange)
xlabel('Absolute ranging error (meters)')
title('Ranging Error CDF')
```



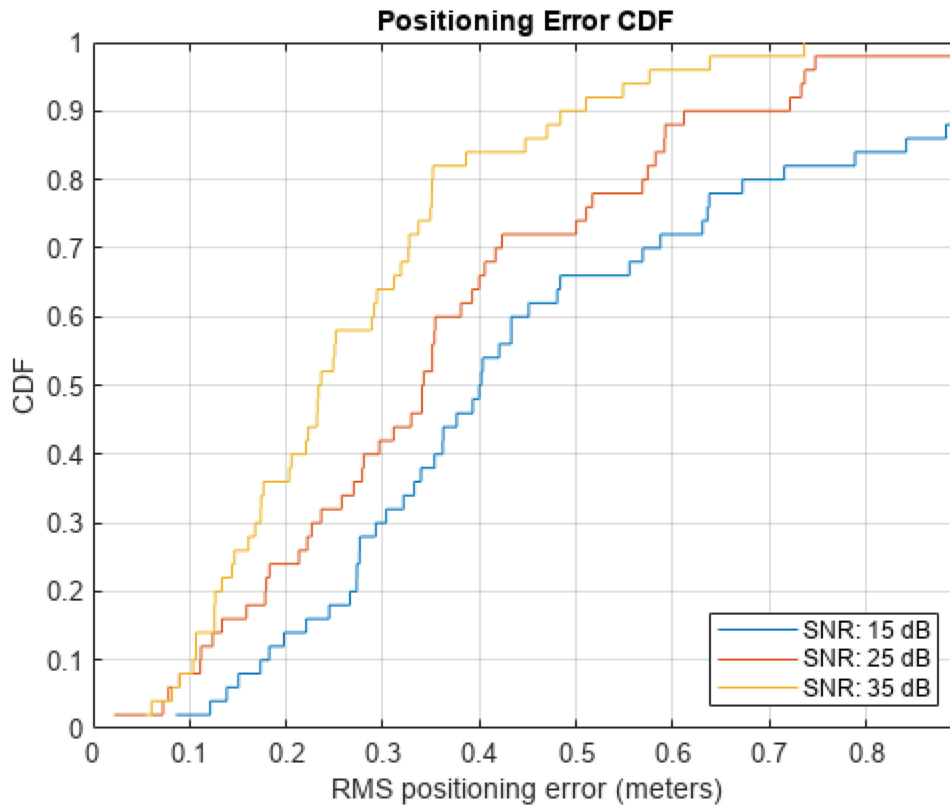
Trilateration

Trilaterate the location of the STA in two dimensions by using the distance estimates, then calculate the positioning root-mean-square error (RMSE) for each iteration by using the STA position estimate. Display the average RMSE and its CDF for each SNR point.

```
positionSTAEst = zeros(2,numIterations,numSNR);
RMSE = zeros(numIterations,numSNR);
for isnr = 1:numSNR
    for i = 1:numIterations
        positionSTAEst(:,i,isnr) = hePositionEstimate(squeeze(positionAP(:,:,i,isnr)),squeeze(dis
    end
    % Find the RMSE for each iteration, then take the mean of all RMSEs
    RMSE = reshape(sqrt(mean(((positionSTAEst-positionSTA).^2),1)),[numIterations numSNR]);
    posEr = mean(RMSE(:,isnr),'all','omitnan');
    disp(['At SNR = ',num2str(snrRange(isnr)),' dB, ', 'Average RMS Positioning error = ', num2s
end
```

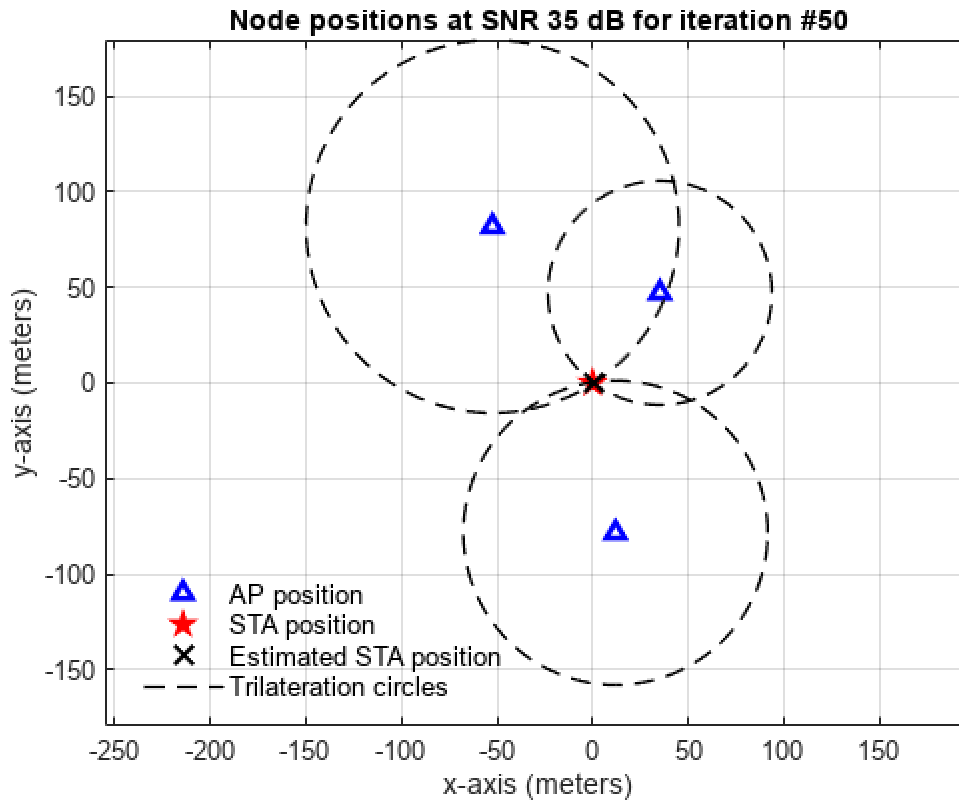
```
At SNR = 15 dB, Average RMS Positioning error = 0.61713 meters.
At SNR = 25 dB, Average RMS Positioning error = 0.40481 meters.
At SNR = 35 dB, Average RMS Positioning error = 0.26968 meters.
```

```
hePlotErrorCDF(RMSE,snrRange)
xlabel('RMS positioning error (meters)')
title('Positioning Error CDF')
```



Plot the location estimate and the trilateration circles of the last iteration.

```
hePlotTrilaterationCircles(positionAP(:,:,numIterations,numSNR),positionSTAEst(:,numIterations,nu
```

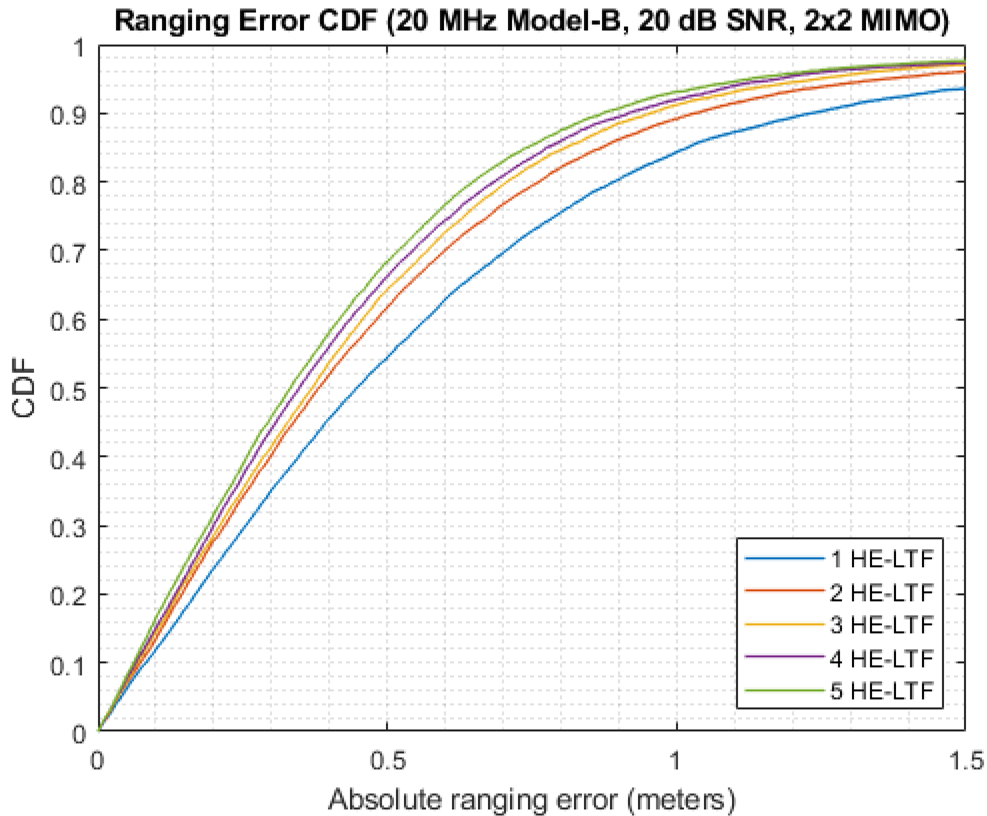


Conclusion

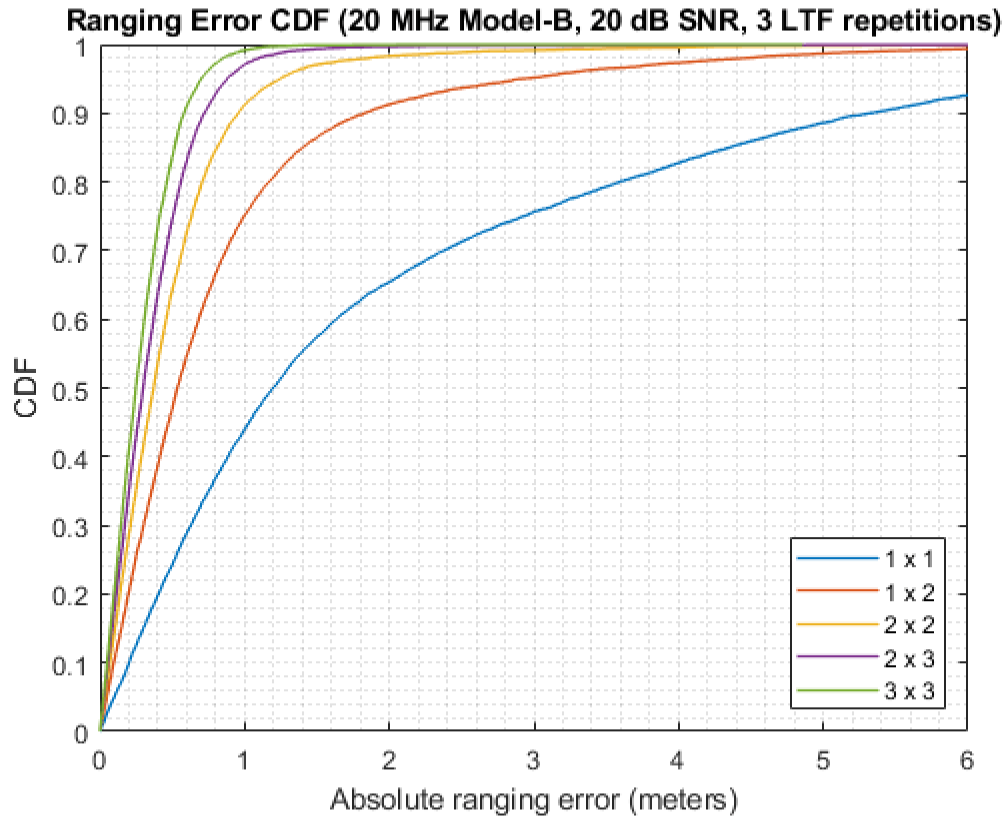
This example shows how to use a positioning algorithm with the IEEE® 802.11az™ standard. In particular, the example shows how to estimate the transmit-receive distance between a STA and AP by using MUSIC super-resolution, and how to locate a STA in two dimensions by using the ranging measurements from multiple STA-AP pairs. The example demonstrates the increase in performance of the positioning system at higher SNRs by computing the positioning estimate at multiple SNR points.

Further Exploration

Besides SNR, several important parameters impact positioning performance, such as HE-LTF repetitions, number of spatial streams, higher bandwidths, and channel delay profiles. This figure shows the impact of HE-LTF repetitions on ranging performance. To generate this figure, run a longer simulation with three randomly placed APs and 4000 iterations for a Model-B 2×2 MIMO channel of 20 MHz at 20 dB SNR. The figure shows that the ranging error decreases with the increase in HE-LTF repetitions. This decrease occurs because HE-LTF repetitions effectively reduce the noise in the CFR by averaging out multiple CFR estimates.



This figure shows the impact of different MIMO configurations on ranging performance. To generate this figure, run a longer simulation with three randomly placed APs and 4000 iterations, generating ranging packets with three HE-LTF repetitions at 20 MHz and specifying a Model-B channel at 20 dB SNR. The figure shows that the ranging error decreases with higher-order MIMO configurations. This decrease occurs because higher-order MIMO configurations produce more CFR snapshots from the different spatial streams available. More CFR snapshots give a better correlation matrix estimate which yields better ToA and distance estimates.



Related Examples

- "Three-Dimensional Indoor Positioning with 802.11az Fingerprinting and Deep Learning" on page 6-87. Trains a convolutional neural network (CNN) for localization and positioning by using Deep Learning Toolbox and IEEE 802.11az data generated with WLAN Toolbox.
- "802.11az Waveform Generation" on page 1-48. Parameterizes and generates IEEE 802.11az high-efficiency (HE) ranging null data packet (NDP) waveforms and highlights some of the key features of the standard.

References

1. IEEE P802.11az™/D2.0 Draft Standard for Information technology— Telecommunications and information exchange between systems Local and metropolitan area networks— Specific requirements - Amendment 3: Enhancements for positioning.
2. IEEE Std 802.11ax™-2021. 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 1: Enhancements for High Efficiency WLAN.
3. Xinrong Li and K. Pahlavan, "Super-resolution TOA estimation with diversity for indoor geolocation," in IEEE Transactions on Wireless Communications, vol. 3, no. 1, pp. 224-234, Jan. 2004, doi: 10.1109/TWC.2003.819035.

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” example.

Detect Router Impersonation Using RF Fingerprinting

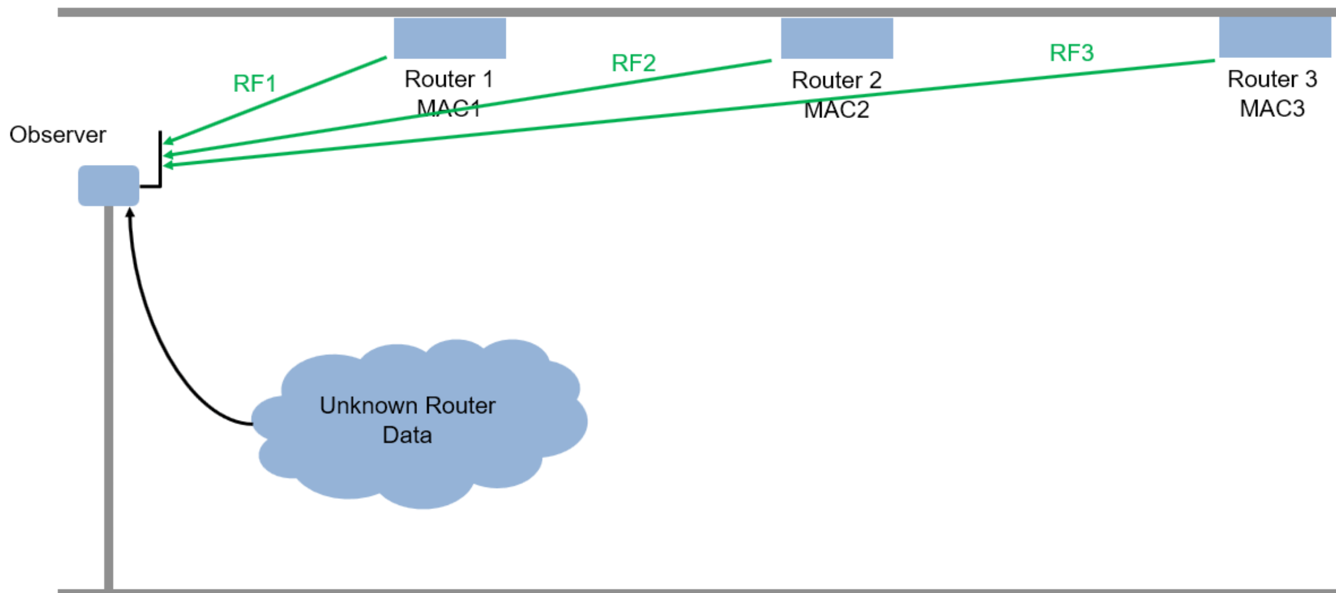
Router impersonation is a form of attack on a WLAN network where a malicious agent tries to impersonate a legitimate router and trick network users to connect to it. Security identification solutions based on simple digital identifiers, such as MAC addresses, IP addresses, and SSID, are not effective 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 6-132, 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.

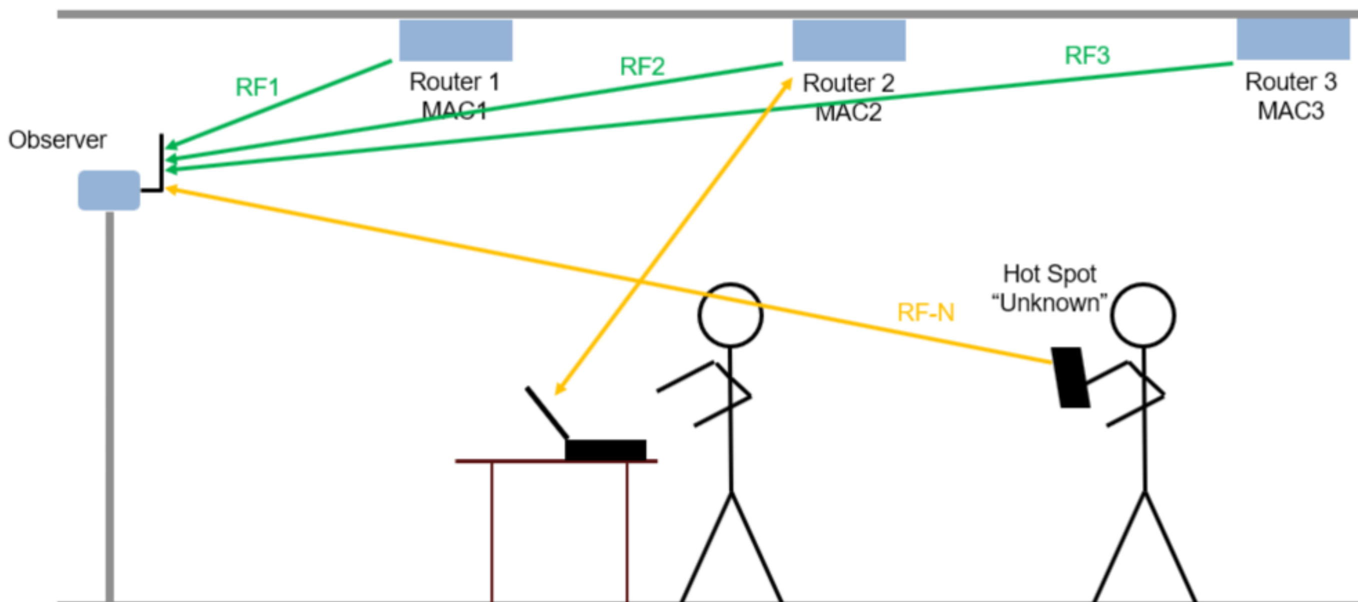
This example simulates a WLAN system with several fixed routers and a fixed observer using the WLAN Toolbox™ and trains a neural network (NN) with the simulated data using Deep Learning Toolbox™.

System Description

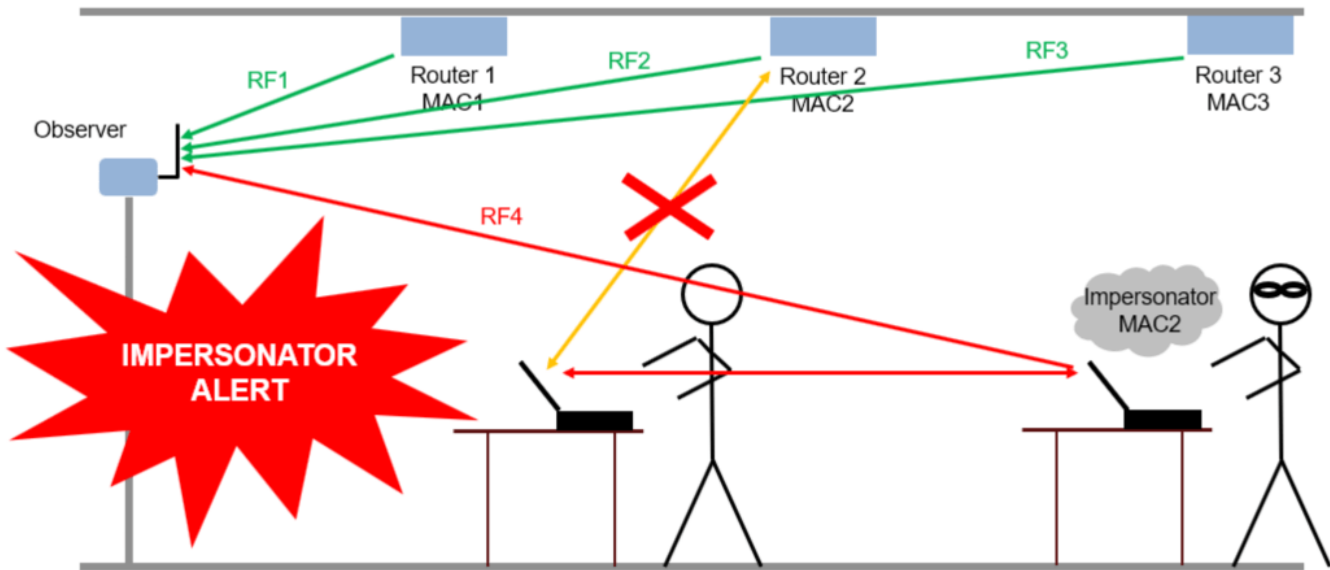
Assume an indoor space with a number of trusted routers with known MAC addresses, which we will refer to as known routers. Also, assume that unknown routers may enter the observation area, some of which may be router impersonators. The class "Unknown" represents any transmitting device that is not contained 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 enable 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 match, 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 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_R2023a.tar. 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 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® Xeon® W-2133 CPU @ 3.6 GHz with 64 GB memory. Training this network takes about 20

seconds with an NVIDIA® GeForce RTX 3080 GPU and about 3 minutes with an Intel® Xeon® W-2133 CPU @ 3.6 GHz.

```
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_R2
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`, `wlanMACFrame`, `wlanNonHTConfig`, and `wlanWaveformGenerator` 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 = wlanChannelFrequency(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 6-131 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
  elapsedTime = seconds(toc);
  elapsedTime.Format = 'hh:mm:ss';
  fprintf('%s - Generating frames for router %d with MAC address %s\n', ...
    elapsedTime, 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
      elapsedTime = seconds(toc);
      elapsedTime.Format = 'hh:mm:ss';
      fprintf('%s - Generated %d/%d frames\n', ...
        elapsedTime, 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 4DA3EE3C8968
00:00:00 - Generating frames for router 2 with MAC address B1077CFE3777
00:00:01 - Generating frames for router 3 with MAC address DB28133A97BF
00:00:01 - Generating frames for router 4 with MAC address B8AF375DAC0F
00:00:01 - Generating frames for router 5 with MAC address AAAAAAAAAAAA
00:00:01 - 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
00:00:02 - Generating frames for router 9 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 10 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 11 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 12 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 13 with MAC address AAAAAAAAAAAA
00:00:03 - 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 frameLength X 2 x 1 x numFrames array, 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 frameLength x 2 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 frameLength x 2 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 6-132.

```
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')
  ]
```

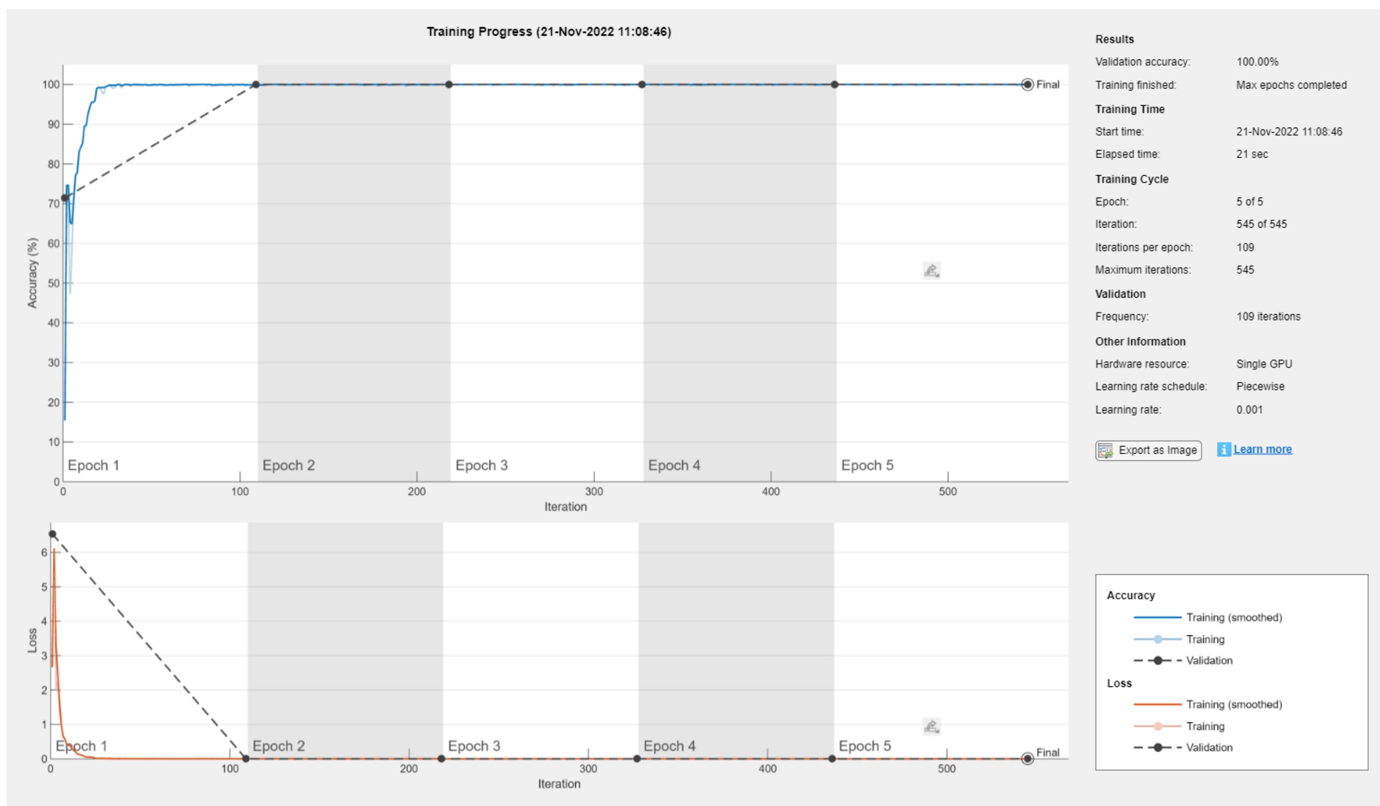
```
layers = 
  18×1 Layer array with layers:

     1   'Input Layer'   Image Input             160×2×1 images
     2   'CNN1'          2-D 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'      2-D Max Pooling         2×1 max pooling with stride [2  1] and padding
     6   'CNN2'          2-D 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'      2-D 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 512. 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'.

```
if trainNow

  miniBatchSize = 512; %#ok<UNRCH>
  iterPerEpoch = floor(numTrainingFramesPerRouter*numTotalRouters/miniBatchSize);

  % Training options
  options = trainingOptions('adam', ...
    'MaxEpochs',5, ...
    'ValidationData',{xValFrames, yVal}, ...
    'ValidationFrequency',iterPerEpoch, ...
    'Verbose',false, ...
    'InitialLearnRate', 0.004, ...
```

```
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor', 0.5, ...
    'LearnRateDropPeriod', 2, ...
    'MiniBatchSize', miniBatchSize, ...
    '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_R2023a.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. The final accuracy is 100%.



Classify test frames and calculate the final accuracy of the neural network.

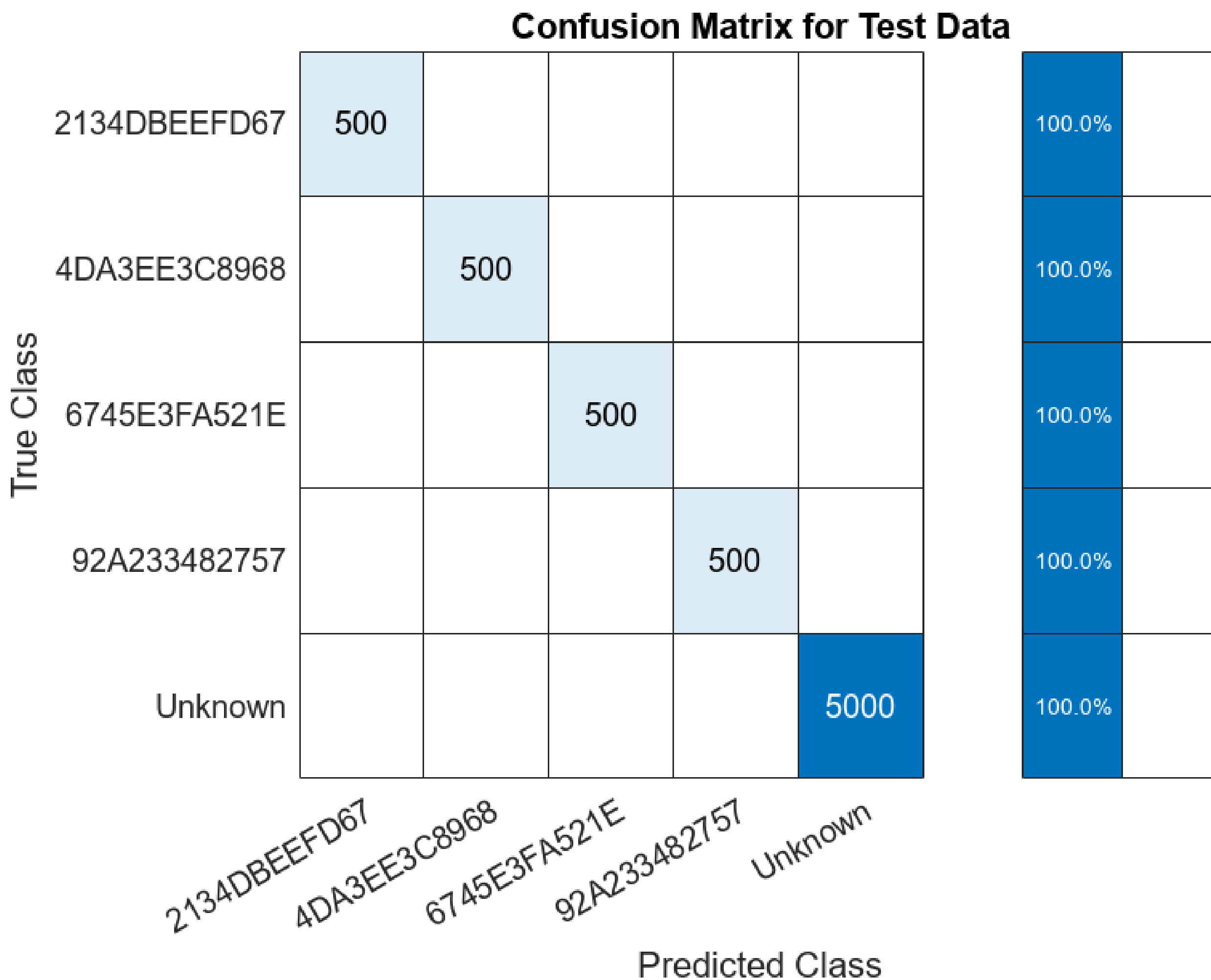
```
% Obtain predicted classes for xTestFrames
yTestPred = classify(simNet,xTestFrames);

% 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) ) + dcOffsetRan
  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 I
        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.Frequency

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.

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” 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” 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.

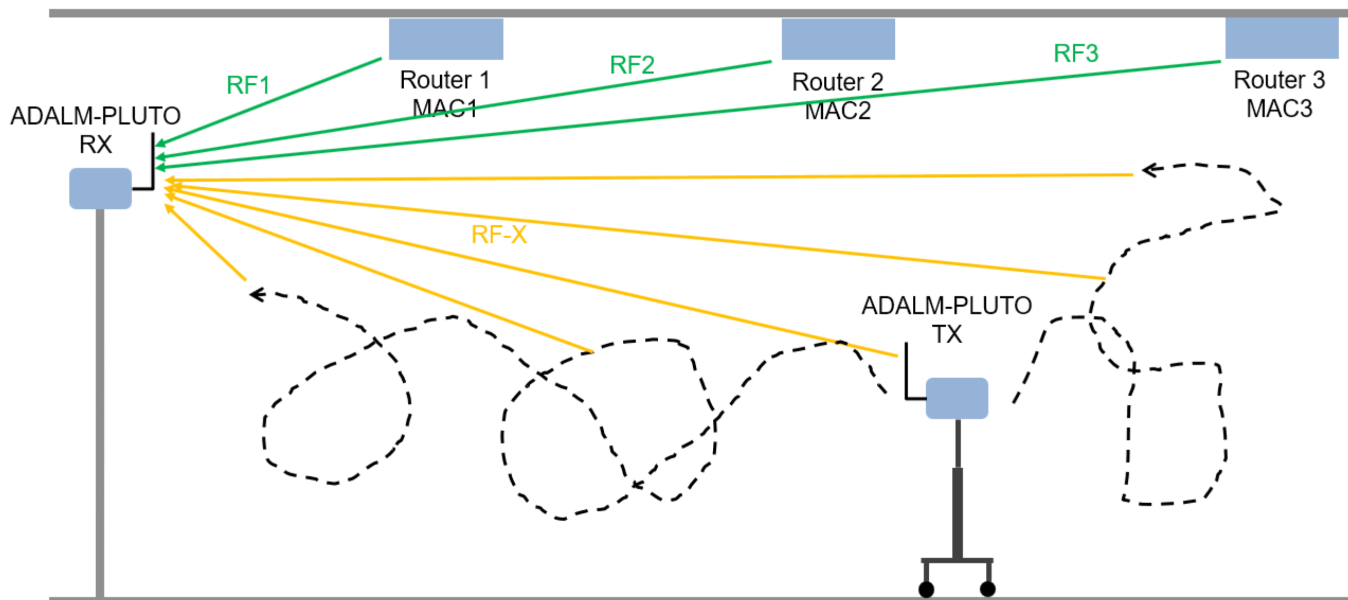
In this example, the data was collected using the scenario depicted in the following figure. The observer is a stationary ADALM-PLUTO radio. Known router data was collected as follows:

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” example.



This example downloads training data and trained network from https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData_R2023a.tar. 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 6-142.

```
rfFingerprintingDownloadData('captured')
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData_R202
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 25 seconds with an NVIDIA® GeForce RTX 3080 GPU and about 2 minutes with an Intel® Xeon W-2133 CPU @ 3.6 GHz.

```
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 *frameLength x 2 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 6-142. 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 frameLength x 2 x 1 x numTrainingFramesPerRouter*numKnownRouters matr
  xTrainingFrames = permute(...
    reshape(xTrainingFrames,[frameLength,numTrainingFramesPerRouter*numKnownRouters, 2, 1]),...
    [1 3 4 2]);

  % Reshape dataset into an frameLength x 2 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')
  ];
```

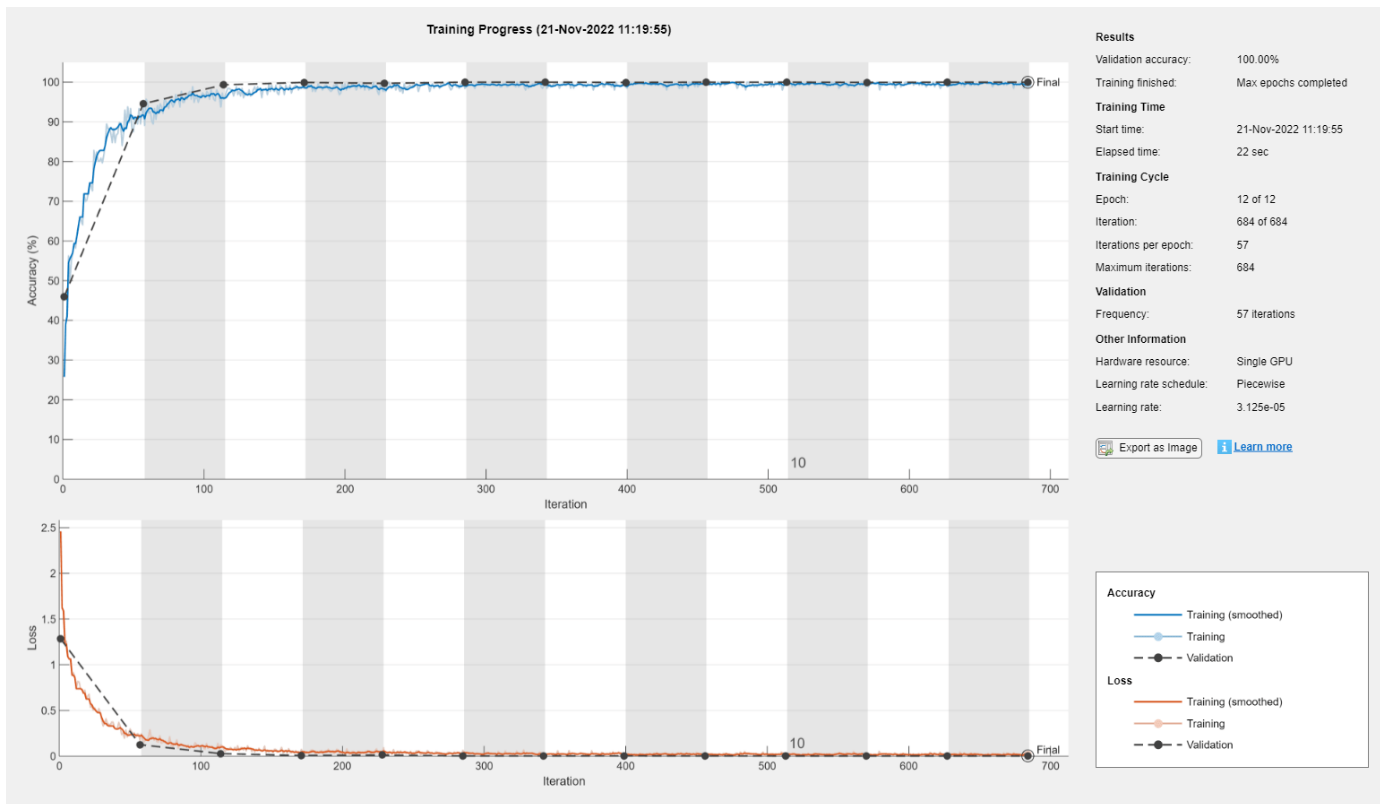
Configure the training options to use ADAM optimizer with a mini-batch size of 256. Use test frames for validation since optimization of hyperparameters were done in [1] on page 6-143.

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'`.

```
if trainNow
  miniBatchSize = 256;
  iterPerEpoch = floor((numTrainingFramesPerRouter*numKnownRouters + numUnknownTrainingFrames)/mi

  options = trainingOptions('adam', ...
    'MaxEpochs', 12, ...
    'ValidationData',{xTestFrames, yTest}, ...
    'ValidationFrequency', iterPerEpoch, ...
    'Verbose',false, ...
```

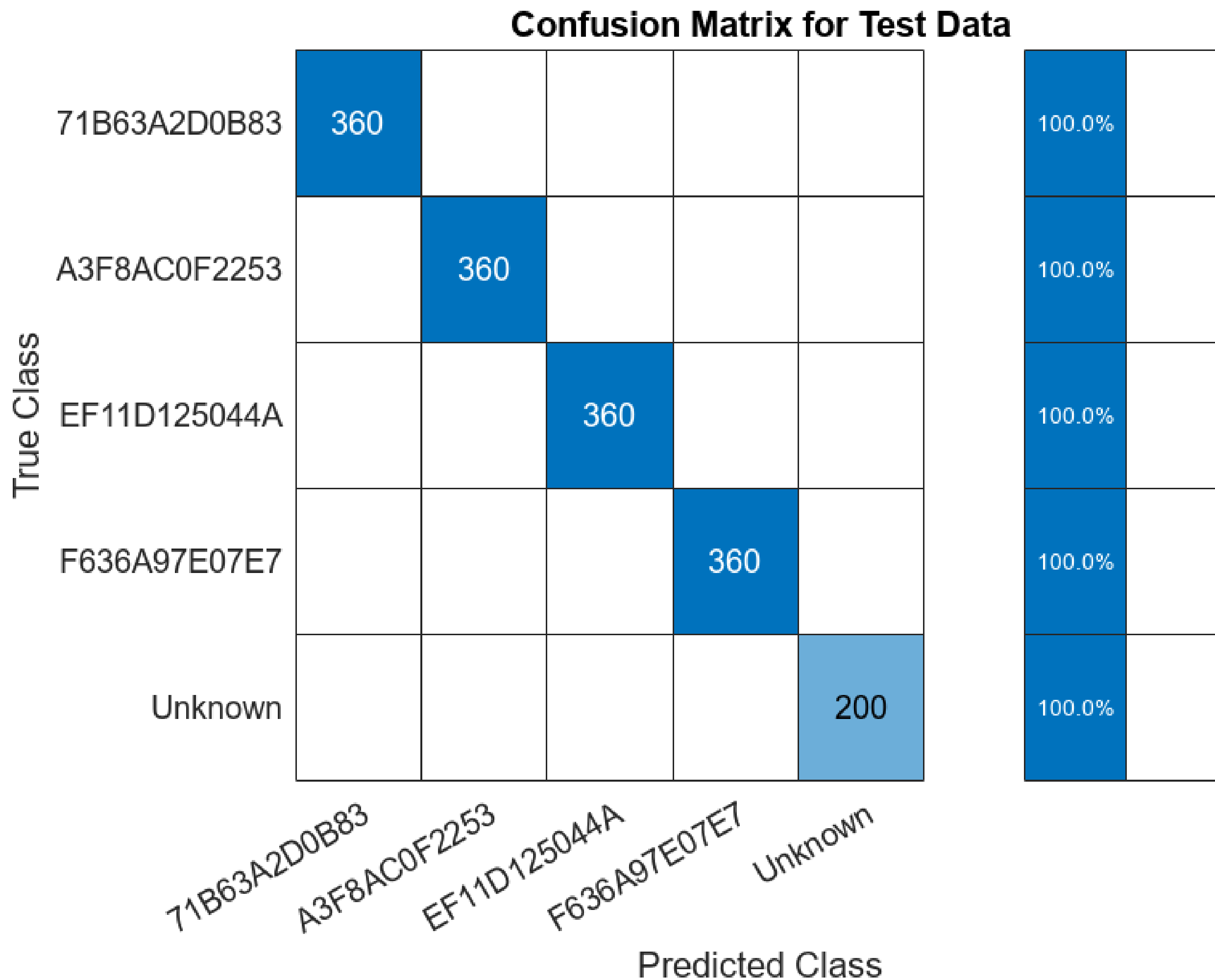
```
    'LearnRateSchedule','piecewise', ...
    'InitialLearnRate', 0.001, ...
    'LearnRateDropFactor', 0.5, ...
    'LearnRateDropPeriod', 2, ...
    'MiniBatchSize', miniBatchSize, ...
    'Plots','training-progress', ...
    'Shuffle', 'every-epoch');

  % Train the network
  capturedDataNet = trainNetwork(xTrainingFrames, yTrain, layers, options);
else
  load('rfFingerprintingCapturedDataTrainedNN_R2023a.mat','capturedDataNet','xTestFrames','yTest
end
```

The following plot shows the training progress of the network run on a computer with a single NVIDIA GeForce RTX 3080 GPU, where the network converged in 12 epochs to 100% accuracy.



Generate the confusion matrix.

```
figure
yTestPred = classify(capturedDataNet,xTestFrames,ExecutionEnvironment='cpu');
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 = wlanChannelFrequency(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, ROUTEI
          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 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
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
```

Further Exploration

Capture data from your own routers as explained in Appendix: Known and Unknown Router Data Collection, on page 6-142 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 “OFDM Beacon Receiver Using Software-Defined Radio” (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.

802.11ad Packet Error Rate Simulation for Control PHY

This example shows how to measure the packet error rate of an IEEE® 802.11ad™ DMG control PHY AWGN link using an end-to-end simulation.

Introduction

In this example an end-to-end simulation is used to determine the packet error rate for an 802.11ad [1] control PHY link with an AWGN channel at a selection of SNR points. At each SNR point multiple packets are transmitted through a noisy channel, de-spread and the PSDUs recovered. The PSDUs are compared to those transmitted to determine the number of packet errors and hence the packet error rate. The receiver assumes perfect synchronization when recovering data bits from the received signal. The following diagram summarizes the processing for each packet.

This example also demonstrates how a `parfor` loop can be used instead of the `for` loop when simulating each SNR point to speed up a simulation. The `parfor` function, as part of the Parallel Computing Toolbox™, executes processing for each SNR in parallel to reduce the total simulation time.

Waveform Configuration

An 802.11ad DMG control PHY transmission is simulated in this example. The DMG format configuration object, `wlanDMGConfig`, contains the format-specific configuration of the transmission. The properties of the object contain the configuration of the transmitted packet. In this example the object is configured to generate a control PHY waveform. The MCS determines the PHY type used, therefore the MCS must be set to 0 to use the control PHY.

```
% Create a format configuration object
cfgDMG = wlanDMGConfig;
cfgDMG.MCS = 0; % MCS 0 represents Control PHY
cfgDMG.PSDULength = 256; % PSDULength in bytes
```

Spectral Filtering

Spectral filtering is used to reduce the out-of-band spectral emissions due to the spread spectrum characteristics of the transmitted waveform. In this example, the waveform is filtered through a raised cosine filter both at the transmitter and receiver using the `comm.RaisedCosineTransmitFilter` and `comm.RaisedCosineReceiveFilter` objects, respectively. To meet the spectral mask requirements, the raised cosine filter is truncated to the duration of eight symbols and the roll-off factor is set to 0.5.

```
% Define the pulse shaping filter characteristics
pulseShaping = true; % Enable pulse shaping
Nsym = 8;            % Filter span in symbol durations
alpha = 0.5;         % Roll-off factor
osps = 4;            % Output samples per symbol
```

```
% Transmit pulse shaping filter
txFilter = comm.RaisedCosineTransmitFilter;
txFilter.RolloffFactor = alpha;
txFilter.FilterSpanInSymbols = Nsym;
txFilter.OutputSamplesPerSymbol = osps;
txFilter.Shape = 'Normal';

% Receive pulse shaping filter
rxFilter = comm.RaisedCosineReceiveFilter;
rxFilter.RolloffFactor = alpha;
rxFilter.DecimationFactor = osps;
rxFilter.InputSamplesPerSymbol = osps;
rxFilter.FilterSpanInSymbols = Nsym;
rxFilter.Shape = 'Normal';
```

Simulation Parameters

For each SNR point in the vector `snrVec` a number of packets are generated, passed through an AWGN channel and demodulated to determine the packet error rate.

```
snrVec = -13.5:0.5:-10.5;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example will lead to a very short simulation. For meaningful results we recommend increasing these numbers.

```
maxNumErrors = 10;   % The maximum number of packet errors at an SNR point
maxNumPackets = 100; % The maximum number of packets at an SNR point
```

Processing SNR Points

For each SNR point a number of packets are tested and the packet error rate calculated.

For each packet the following processing steps occur:

1. A PSDU is created and encoded to create a single packet waveform.
2. AWGN is added to the waveform.
3. The packet is received with perfect synchronization.
4. The header and data fields are extracted from the received waveform and are processed together.
5. The recovered field is de-rotated by pi/2 and is de-spread.
6. The PSDU is recovered from the extracted field.

A `parfor` loop can be used to parallelize processing of the SNR points. To enable the use of parallel computing for increased speed comment out the `for` statement and uncomment the `parfor` statement below.

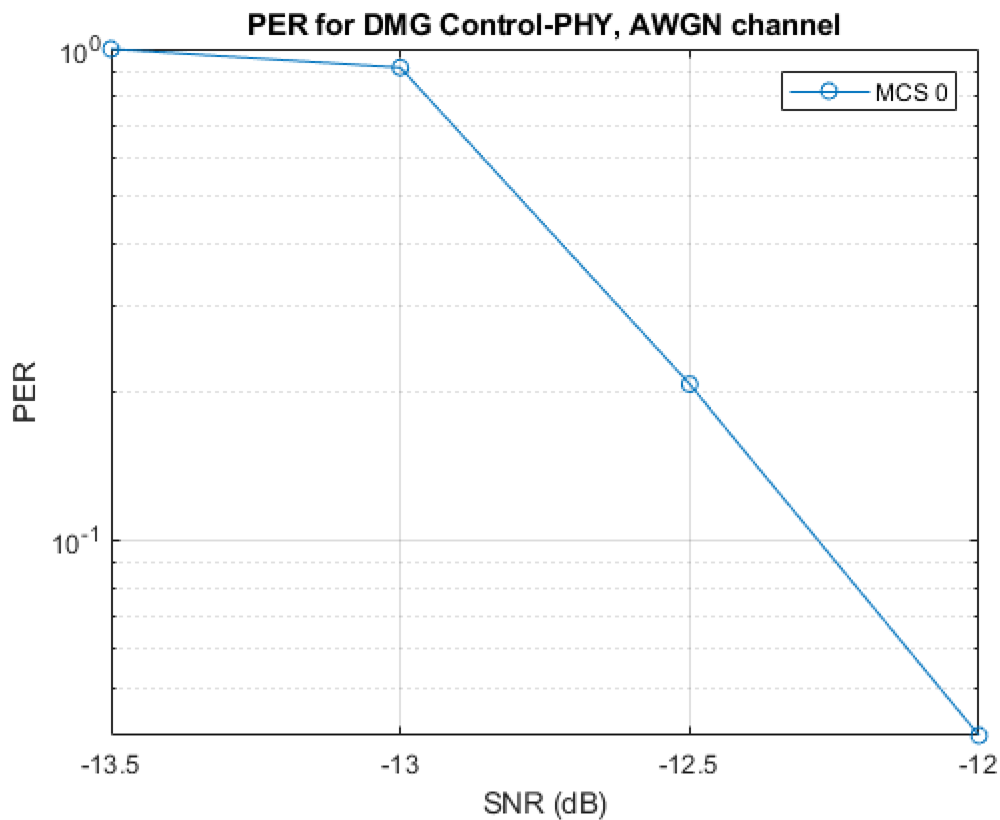
```
numSNR = numel(snrVec); % Number of SNR points
packetErrorRate = zeros(numSNR,1);
```

```
indices = wlanFieldIndices(cfgDMG);

if ~strcmp(phyType(cfgDMG),'Control')
    error('This example only supports DMG Control PHY simulation');
end

%parfor isnr = 1:numSNR % Use 'parfor' to speed up the simulation
for isnr = 1:numSNR     % 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);

    % Noise power
    nVar = 10^(-snrVec(isnr)/10);
    numPacketErrors = 0;
    numPkt = 1; % Index of packet transmitted

    while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
        % Generate a packet waveform
        psdu = randi([0 1],cfgDMG.PSDULength*8,1); % PSDULength in bytes
        tx = wlanWaveformGenerator(psdu,cfgDMG);

        % Transmitter filtering
        if pulseShaping
            % Append zero to compensate for filter group delay
            tx = txFilter([tx; zeros(Nsym,1)]);
            reset(txFilter);
        end

        % Add noise
        rx = awgn(tx,snrVec(isnr));

        % Receiver filtering
        if pulseShaping
            rx = rxFilter(rx);
            reset(rxFilter);
        end

        % Synchronize
        % The received signal is synchronized to the start of the packet by
        % compensating for a known delay due the spectral shaping filters
        if pulseShaping
            offset = Nsym;
        else
            offset = 0; %#ok<UNRCH>
        end

        % Process header and data field together
        rxHeaderDataField = rx(offset+(indices.DMGHeader(1):indices.DMGData(2)));

        % Apply pi/2 de-rotation and de-spread the received signal
        [rxSym,SF] = dmgControlDespread(rxHeaderDataField);

        % Recover the transmitted PSDU from DMG Data field. Scale the noise
        % power by the spreading factor
        dataDecode = wlanDMGDataBitRecover(rxSym,nVar/SF,cfgDMG);
```

```
        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(psdu,dataDecode));
        numPacketErrors = numPacketErrors + packetError;
        numPkt = numPkt+1;
    end

    % Calculate packet error rate (PER) at SNR point
    packetErrorRate(isnr) = numPacketErrors/(numPkt-1);
    disp(['SNR ' num2str(snrVec(isnr))...
          ' completed after ' num2str(numPkt-1) ' packets,'...
          ' PER: ' num2str(packetErrorRate(isnr))]);
end
```

Plot Packet Error Rate vs SNR Results

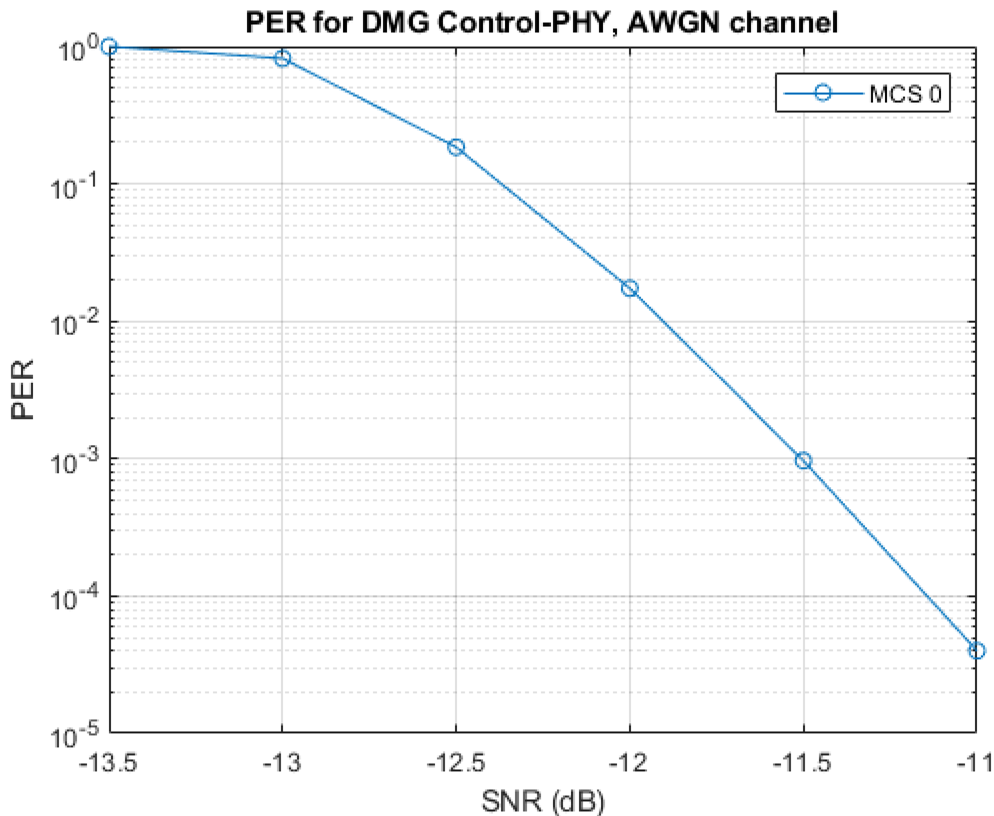
```
figure;
semilogy(snrVec,packetErrorRate,'-o');
grid on;
xlabel('SNR (dB)');
ylabel('PER');
legend('MCS 0');
title('PER for DMG Control-PHY, AWGN channel');
```



Further Exploration

The number of packets tested at each SNR point is controlled by two parameters: `maxNumErrors` and `maxNumPackets`. For meaningful results, it is recommended that these values should be larger

than those presented in this example. Increasing the number of packets simulated allows the PER under different scenarios to be compared. As an example, the figure below was created by running the example for a `PSDULength` of 256 bytes, `maxNumErrors`:1000 and `maxNumPackets`: 100000.



Selected Bibliography

1 IEEE Std 802.11™-2020. 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.

Local Functions

The following local function is used in this example:

- `dmgControlDespread`: De-spread the receive signal

```
function [y,SF] = dmgControlDespread(rx)
    SF = 32; % Spreading factor
    dataField = rx.*exp(-1i*pi/2*(0:size(rx,1)-1).'); % De-rotate symbols
    Ga = wlanGolaySequence(SF); % Generate Golay sequence
    y = (reshape(dataField,SF,length(dataField)/SF)'*Ga)/SF;
end
```

```
SNR -13.5 completed after 11 packets, PER: 1
SNR -13 completed after 12 packets, PER: 0.91667
SNR -12.5 completed after 53 packets, PER: 0.20755
```

```
SNR -12 completed after 100 packets, PER: 0.04
SNR -11.5 completed after 100 packets, PER: 0
SNR -11 completed after 100 packets, PER: 0
SNR -10.5 completed after 100 packets, PER: 0
```

802.11ad Single Carrier Link with RF Beamforming in Simulink

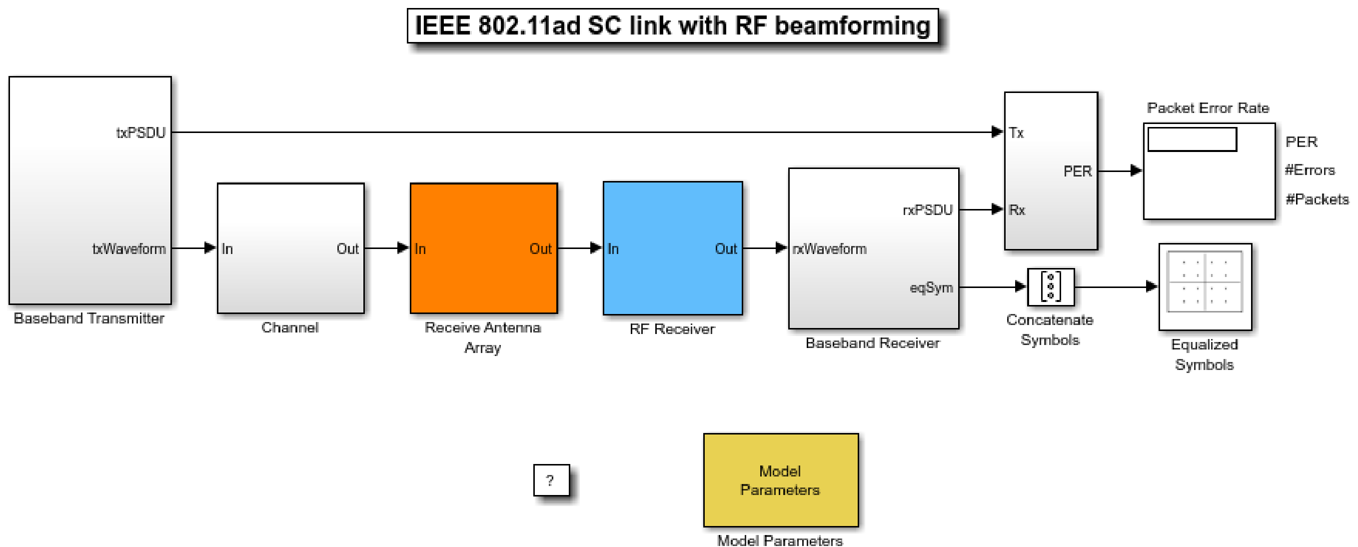
This example shows how to model an IEEE 802.11ad™ single carrier link in Simulink® which includes a phased array antenna with RF beamforming. This example requires the following products:

- WLAN Toolbox™ for baseband transmitter and receiver
- Phased Array System Toolbox™ for receive antenna array
- RF Blockset™ for RF receiver

Introduction

This model simulates an 802.11ad single carrier (SC) [1] link with RF beamforming. Multiple packets are transmitted through free space, then RF beamformed, demodulated and the PLCP service data units (PSDU) are recovered. The PSDUs are compared with those transmitted to determine the packet error rate. The receiver performs packet detection, timing synchronization, carrier frequency offset correction and unique word based phase tracking.

The MATLAB function block allows Simulink models to use MATLAB® functions. In this example, an 802.11ad SC link modeled in Simulink uses WLAN Toolbox functions called using MATLAB function blocks. For an 802.11ad baseband simulation in MATLAB, see the example “802.11ad Packet Error Rate Single Carrier PHY Simulation with TGay Channel” on page 5-2.



System Architecture

The system consists of:

- A baseband transmitter which generates a random PSDU and an 802.11ad SC packet.
- A free space channel.

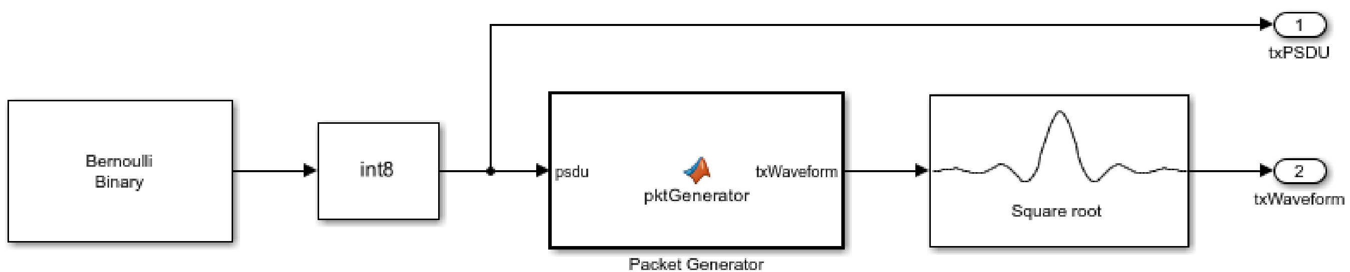
- A receive antenna array which supports up to 16 elements. This module allows control of the array geometry, number of elements in an array, operating frequency, and receiver direction.
- A 16 channel RF receiver module to process the RF signals. This receiver module includes low noise amplifiers, phase shifters, Wilkinson 16:1 combiner, and a down-converter. This module allows control of the beamforming direction used to calculate the corresponding phase shifts.
- A baseband receiver which recovers the transmitted PSDU by performing packet detection, time and frequency synchronization, channel estimation, PSDU demodulation, and decoding.

The system diagnostics includes the display of equalized constellation and the obtained packet error rate.

The following sections describe the transmitter and receiver in more detail.

Baseband Transmitter

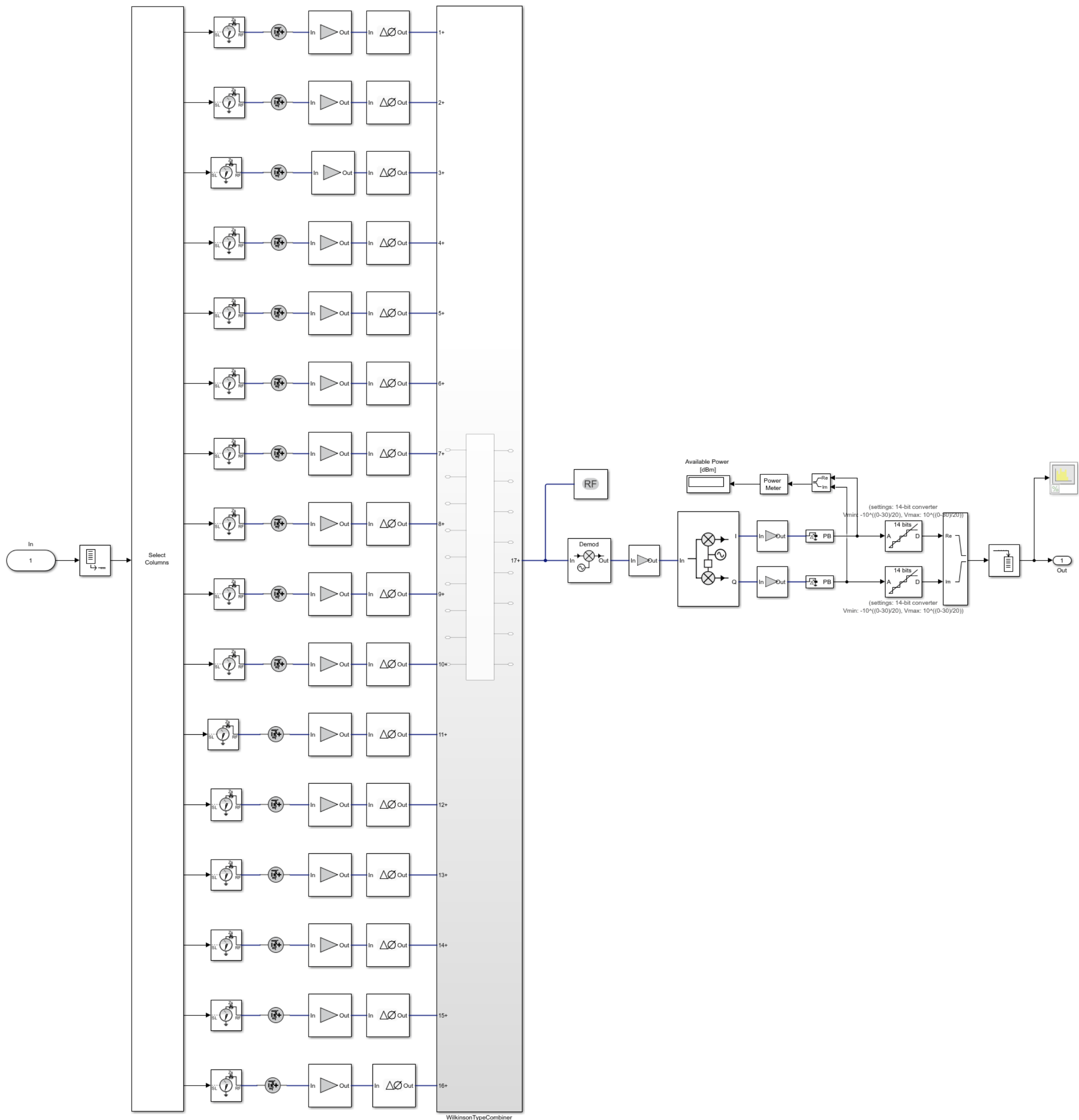
The baseband transmitter block creates a random PSDU and encodes the bits to create a single packet waveform based on the MCS and PSDU length values in the Model Parameters block. The packet generator block uses the function `wlanWaveformGenerator` to encode a packet.



RF Receiver

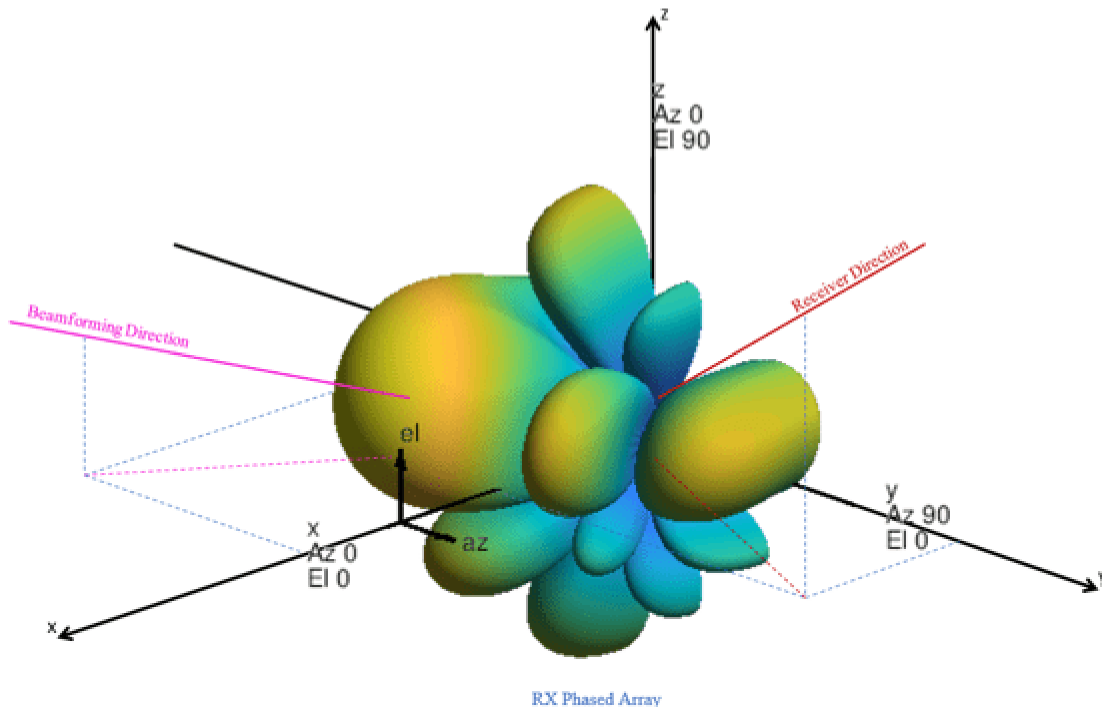
The RF receiver consists of amplifiers, phase shifters, Wilkinson 16:1 combiner and is implemented in superheterodyne fashion.

RF RECEIVE MODULE FOR A 16 CHANNEL PHASED ARRAY with Ideal 16:1 Combiner



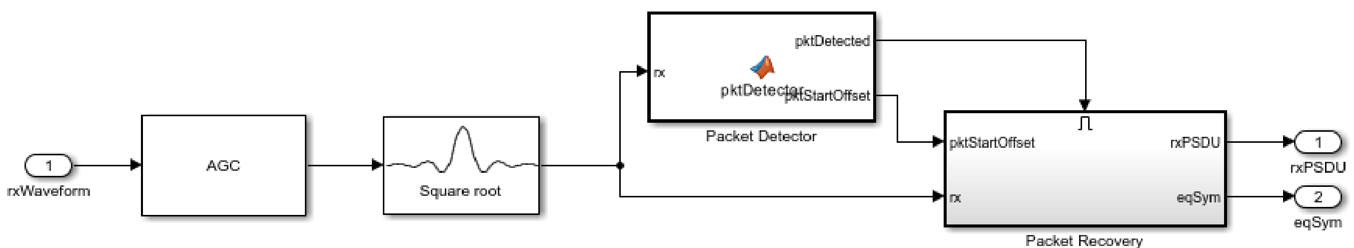
The phase shift applied to each element is calculated based on the beamforming direction. This is provided by the user and indicates the direction of the main beam. The receiver maximizes the SNR when the receiver's main beam points to the transmitter. Transmitter is omnidirectional and the

receiver direction (az,el) indicates the direction of incident signal. The scenario where the receiver direction and the beamforming direction are different is shown. In this case, there will be a reduction in the received signal power leading to high packet error rate (PER) and error vector magnitude (EVM). The results section shows these values.



Baseband Receiver

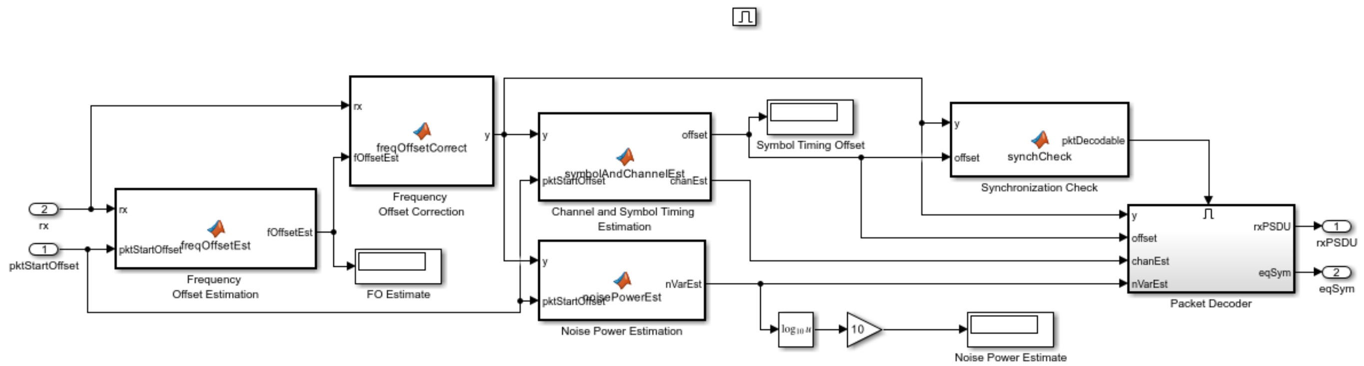
The baseband receiver has two components: packet detection and packet recovery.



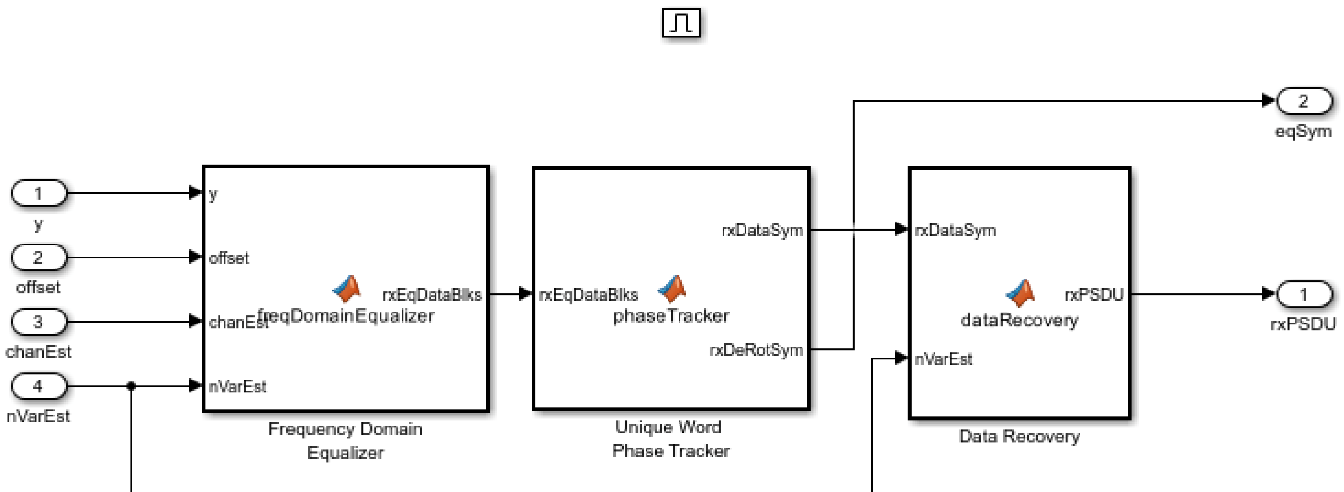
If a packet is detected, the packet recovery subsystem is enabled to process the detected packet.

The packet recovery subsystem processing consists of the following steps:

1. Frequency offset estimation and correction.
2. Symbol timing and channel frequency response estimation.
3. Noise power estimation.
4. Synchronization error checking. This determines whether the packet can be decoded or not.
5. Packet decoding.



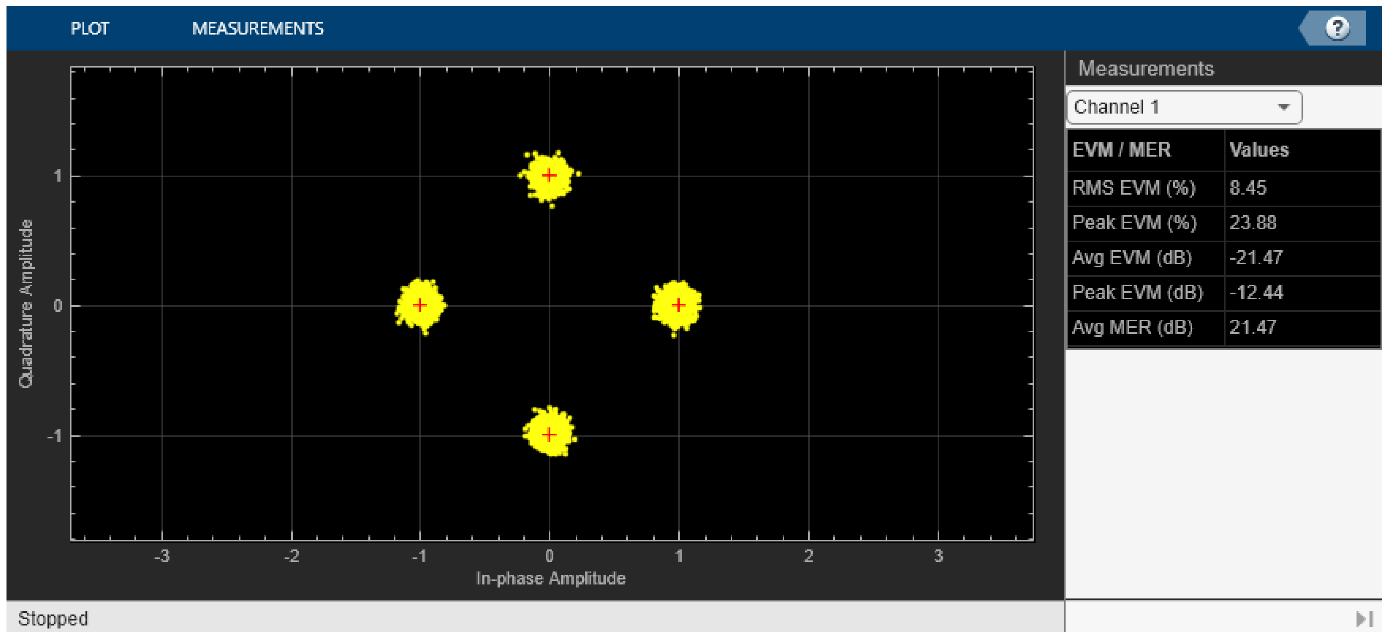
In the packet decoder subsystem, the SC data field is extracted from the synchronized received waveform. Then, the PSDU is recovered using the extracted field, channel, and noise power estimates.



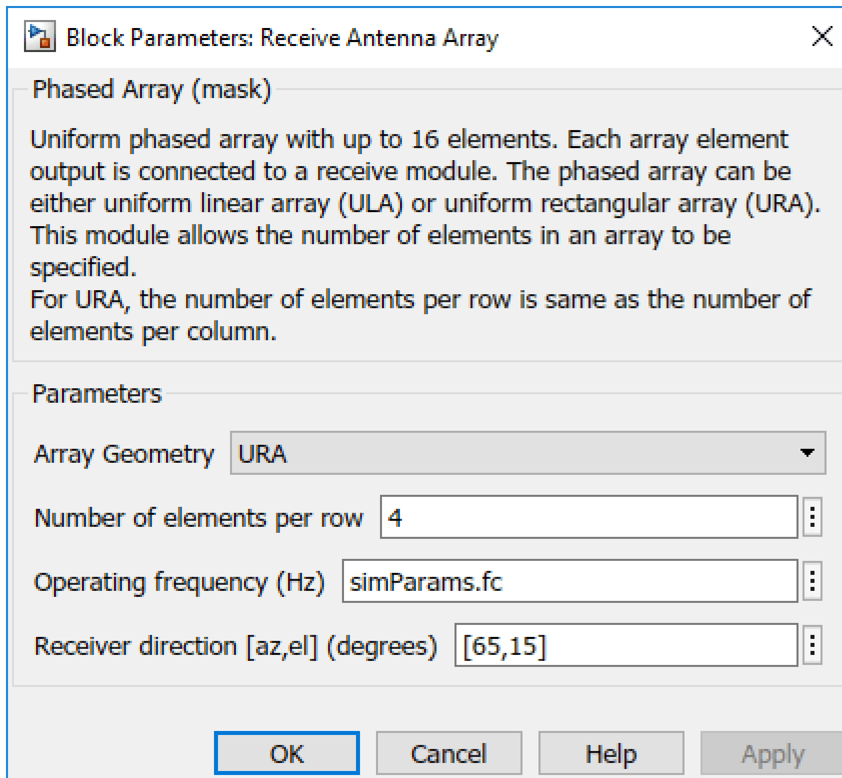
Results

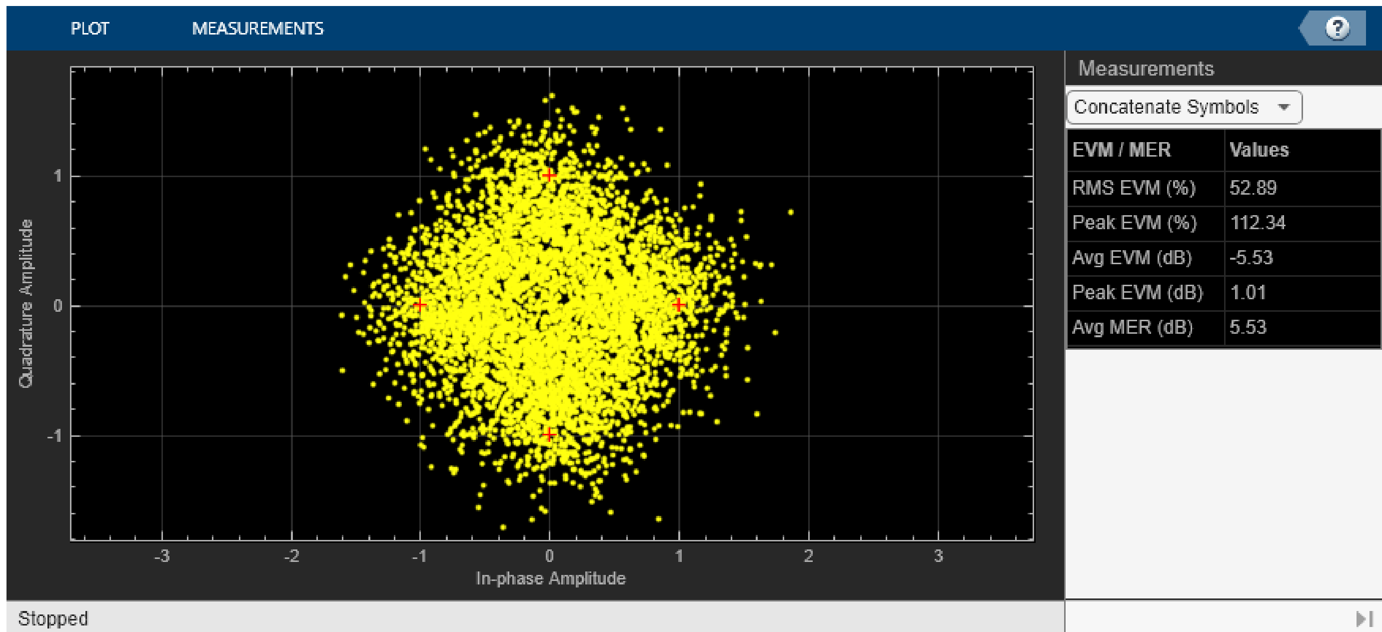
Running the simulation displays the packet error rate. The model updates the PER after processing each packet. The model also displays the equalized symbol constellation along with the EVM measurement. Note that for statistically valid results, long simulation times are required.

By default, the main beam of the receive antenna array points towards the direction: azimuth = 0 deg. and elevation = 0 deg.

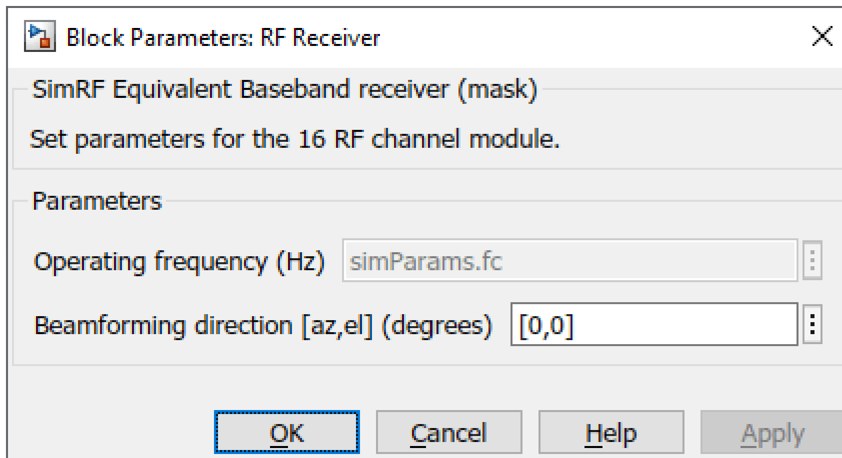


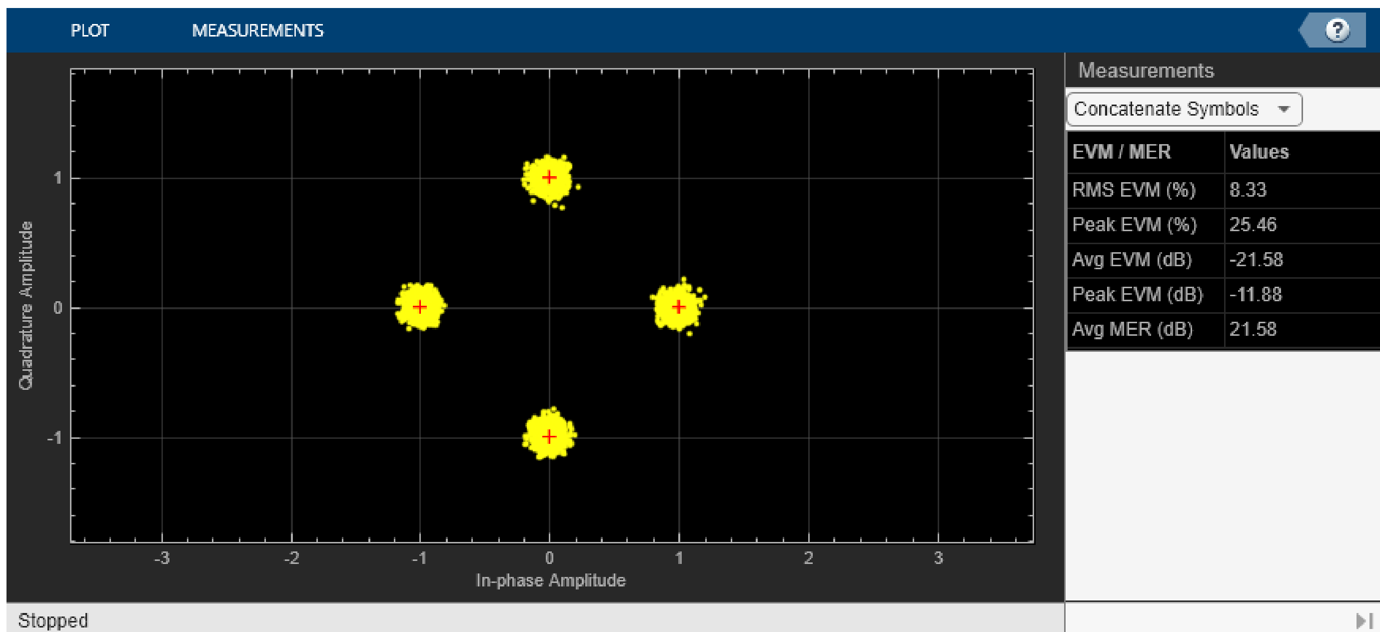
If you change the `Receiver direction` value in the receive antenna array towards a proximity null in the array radiation, the EVM increases and the packets cannot be successfully decoded.





If you change the `Beamforming direction` value in the RF receiver such that the main beam points towards the transmitter, the EVM improves and packets are successfully decoded.





Exploring the Example

- Try changing the signal to noise ratio (SNR) value in the Model Parameters block. Increasing SNR leads to lower packet error rates and improved EVM of equalized symbols constellation. The SNR specified is the signal to noise ratio at the input to the ADC, if a single receive chain is used. The SNR accounts for free space path loss, thermal noise and the noise figure of RF components.

- You can change the array geometry and the number of elements in an array present in the receive antenna array block. Increasing the number of antenna elements improves the EVM. The diversity gain due to receiver antenna array can be observed in the equalized symbols constellation.

Appendix

This example uses the following helper functions:

- dmgCFOEstimate.m
- dmgPacketDetect.m
- dmgSingleCarrierFDE.m
- dmgSTFNoiseEstimate.m
- dmgTimingAndChannelEstimate.m
- dmgUniqueWordPhaseTracking.m

Selected Bibliography

1 IEEE Std 802.11™-2020. 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.

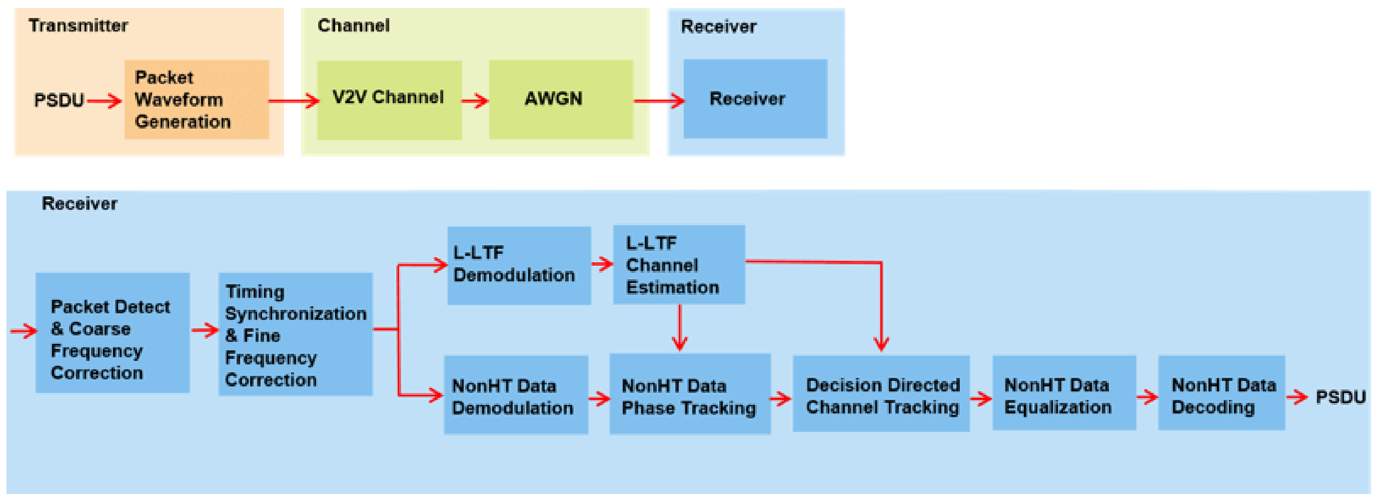
802.11p Packet Error Rate Simulation for a Vehicular Channel

This example shows how to measure the packet error rate (PER) of an IEEE® 802.11p™ link using an end-to-end simulation with a Vehicle-to-Vehicle (V2V) fading channel and additive white Gaussian noise. The PER performance of a receiver with and without channel tracking is compared. In a vehicular environment (high Doppler), a receiver with channel tracking performs better.

Introduction

IEEE 802.11p [1] is an approved amendment to the IEEE 802.11™ standard to enable support for wireless access in vehicular environments (WAVE). Using the half-clocked mode with a 10 MHz channel bandwidth, it operates in 5.85-5.925 GHz bands to support applications for Intelligent Transportation Systems (ITS) [2].

In this example, an end-to-end simulation is used to determine the packet error rate for an 802.11p [1] link with a fading channel at a selection of SNR points with and without channel tracking. For each SNR point, multiple packets are transmitted through a V2V channel, demodulated and the PSDUs are recovered. The PSDUs are compared to those transmitted to determine the number of packet errors. For each packet, packet detection, timing synchronization, carrier frequency offset correction and phase tracking are performed at the receiver. For channel tracking, decision directed channel estimation [3] is used to compensate for the high Doppler spread. The figure below shows the processing chain with channel tracking.



Waveform Configuration

An 802.11p non-HT format transmission is simulated in this example. A non-HT format configuration object contains the format specific configuration of the transmission. This object is created using the `wlanNonHTConfig` function. In this example, the object is configured for a 10 MHz channel bandwidth and QPSK rate 1/2 (MCS 2) operation.

```
% Link parameters
mcs = 2;       % QPSK rate 1/2
psduLen = 500; % PSDU length in bytes

% Create a format configuration object for an 802.11p transmission
cfgNHT = wlanNonHTConfig;
```

```
cfgNHT.ChannelBandwidth = 'CBW10';
cfgNHT.PSDULength = psduLen;
cfgNHT.MCS = mcs;
```

Channel Configuration

The V2V radio channel model defines five scenarios to represent fading conditions within a vehicular environment. In this example, 'Urban NLOS' [4] scenario is used. This corresponds to a scenario with two vehicles crossing each other at an urban blind intersection with building and fences present on the corners.

```
% Create and configure the channel
fs = wlanSampleRate(cfgNHT); % Baseband sampling rate for 10 MHz

chan = V2VChannel;
chan.SampleRate = fs;
chan.DelayProfile = 'Urban NLOS';
```

Simulation Parameters

For each SNR (dB) point in the vector `snr` a number of packets are generated, passed through a channel and demodulated to determine the packet error rate.

```
snr = 15:5:30;
```

The number of packets tested at each SNR point is controlled by two parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each SNR point. When the number of packet errors reaches this limit, the simulation at this SNR point is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each SNR point. It limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example lead to a short simulation. For statistical meaningful results these numbers should be increased.

```
maxNumErrors = 20;   % The maximum number of packet errors at an SNR point
maxNumPackets = 200; % The maximum number of packets at an SNR point

% Set random stream for repeatability of results
s = rng(98);
```

Processing SNR Points

For each SNR point, a number of packets are tested and the packet error rate is calculated. For each packet the following processing steps occur:

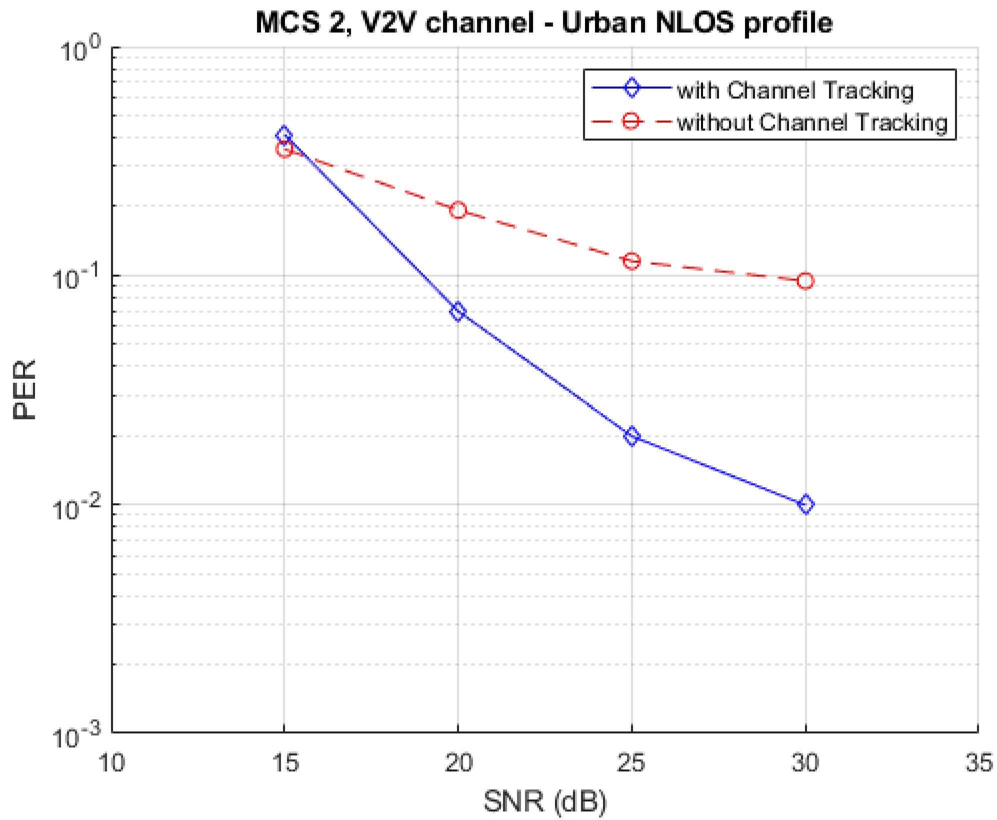
1. A PSDU is created and encoded to create a single packet waveform.
2. The waveform is passed through the channel. Different channel realizations are used for each transmitted packet.
3. AWGN is added to the received waveform to create the desired average SNR per active subcarrier after OFDM demodulation.
4. The per-packet processing includes packet detection, coarse carrier frequency offset estimation and correction, symbol timing and fine carrier frequency offset estimation and correction.
5. The L-LTF is extracted from the synchronized received waveform. The L-LTF is OFDM demodulated and initial channel estimates are obtained.

6 Channel tracking can be enabled using the switch `enableChanTracking`. If enabled, the channel estimates obtained from L-LTF are updated per symbol using decision directed channel tracking as presented in J. A. Fernandez et al in [3]. If disabled, the initial channel estimates from L-LTF are used for the entire packet duration.

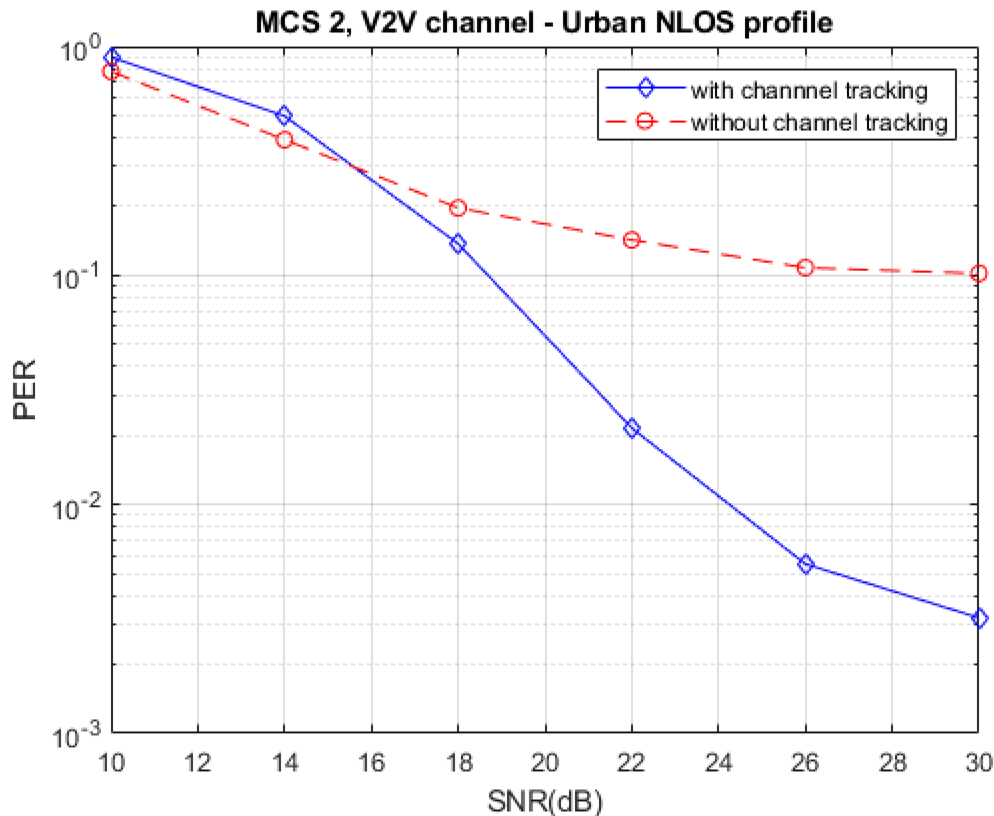
7 The non-HT Data field is extracted from the synchronized received waveform. The PSDU is recovered using the extracted data field and the channel estimates and noise power estimate.

```
% Set up a figure for visualizing PER results
h = figure;
grid on;
hold on;
ax = gca;
ax.YScale = 'log';
xlim([snr(1), snr(end)]);
ylim([1e-3 1]);
xlabel('SNR (dB)');
ylabel('PER');
h.NumberTitle = 'off';
h.Name = '802.11p ';
title(['MCS ' num2str(mcs) ', V2V channel - ' chan.DelayProfile ' profile']);

% Simulation loop for 802.11p link
S = numel(snr);
per_LS = zeros(S,1);
per_STA = per_LS;
for i = 1:S
    enableChanTracking = true;
    % 802.11p link with channel tracking
    per_STA(i) = v2vPERSimulator(cfgNHT, chan, snr(i), ...
        maxNumErrors, maxNumPackets, enableChanTracking);

    enableChanTracking = false;
    % 802.11p link without channel tracking
    per_LS(i) = v2vPERSimulator(cfgNHT, chan, snr(i), ...
        maxNumErrors, maxNumPackets, enableChanTracking);

    semilogy(snr, per_STA, 'bd-');
    semilogy(snr, per_LS, 'ro--');
    legend('with Channel Tracking','without Channel Tracking')
    drawnow;
end

axis([10 35 1e-3 1])
hold off;

% Restore default stream
rng(s);
```

```
SNR 15 dB with channel tracking completed after 51 packets, PER: 0.41176
SNR 15 dB without channel tracking completed after 59 packets, PER: 0.35593
SNR 20 dB with channel tracking completed after 201 packets, PER: 0.069652
SNR 20 dB without channel tracking completed after 109 packets, PER: 0.19266
SNR 25 dB with channel tracking completed after 201 packets, PER: 0.0199
SNR 25 dB without channel tracking completed after 182 packets, PER: 0.11538
SNR 30 dB with channel tracking completed after 201 packets, PER: 0.0099502
SNR 30 dB without channel tracking completed after 201 packets, PER: 0.094527
```



For meaningful results `maxNumErrors`, `maxNumPackets` should be increased. The below plot provides results for `maxNumErrors: 1000` and `maxNumPackets: 10000`.



Further Exploration

Try changing the channel delay profile, the length of the packet or the data rate (`mcs` values) and observe the performance of channel tracking. For some configurations channel tracking provides little performance improvement. For a small number of OFDM symbols (small PSDU length or high MCS), temporal averaging performed during decision directed channel tracking may not be effective. The characteristics of the channel may also limit the performance for higher order modulation schemes (`mcs` > 5).

Appendix

This example uses the following helper functions and objects:

- v2vPERSimulator.m
- V2VChannel.m

Selected Bibliography

1 IEEE Std 802.11p-2010: 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 6: Wireless Access in Vehicular Environments, IEEE, New York, NY, USA, 2010.

2 ETSI, https://www.etsi.org/technologies/automotive-intelligent-transport.

3 J. A. Fernandez, D. D. Stancil and F. Bai, "Dynamic channel equalization for IEEE 802.11p waveforms in the vehicle-to-vehicle channel," 2010 48th Annual Allerton Conference on

Communication, Control, and Computing (Allerton), Allerton, IL, 2010, pp. 542-551. doi: 10.1109/ALLERTON.2010.5706954

4 P. Alexander, D. Haley and A. Grant, "Cooperative Intelligent Transport Systems: 5.9-GHz Field Trials," in Proceedings of the IEEE, vol. 99, no. 7, pp. 1213-1235, July 2011.

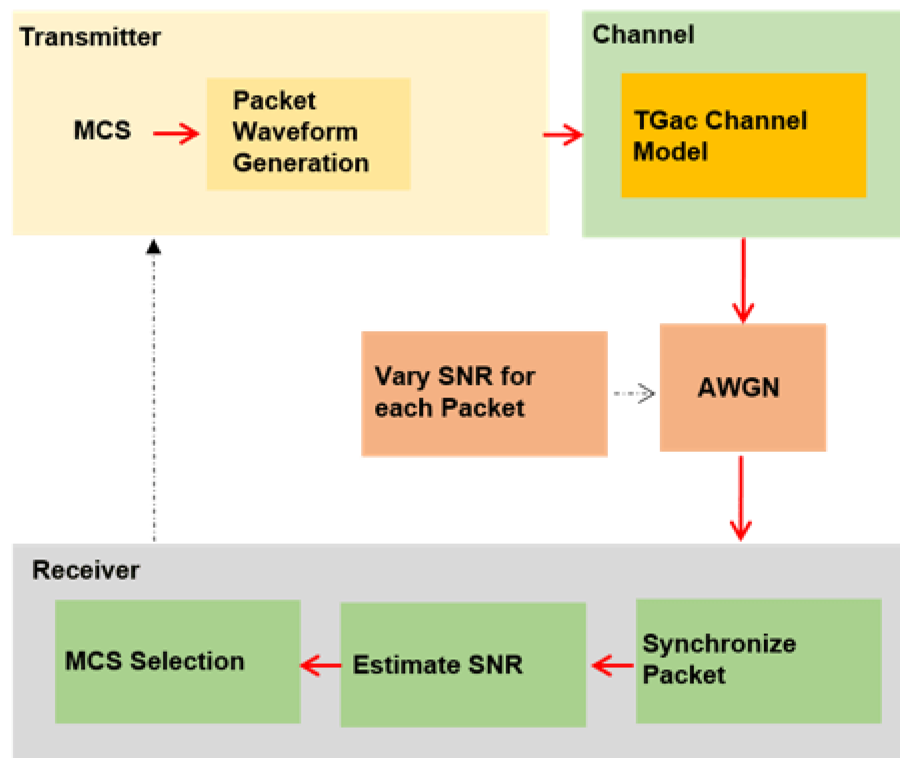
802.11 Dynamic Rate Control Simulation

This example shows dynamic rate control by varying the Modulation and Coding scheme (MCS) of successive packets transmitted over a frequency selective multipath fading channel.

Introduction

The IEEE® 802.11™ standard supports dynamic rate control by adjusting the MCS value of each transmitted packet based on the underlying radio propagation channel. Maximizing link throughput, in a propagation channel that is time varying due to multipath fading or movement of the surrounding objects, requires dynamic variation of MCS. The IEEE 802.11 standard does not define any standardized rate control algorithm (RCA) for dynamically varying the modulation rate. The implementation of RCA is left open to the WLAN device manufacturers. This example uses a closed-loop rate control scheme. A recommended MCS for transmitting a packet is calculated at the receiver and is available at the transmitter without any feedback latency. In a real system this information would be conveyed through a control frame exchange. The MCS is adjusted for each subsequent packet in response to an evolving channel condition with noise power varying over time.

In this example, an IEEE 802.11ac™ [1] waveform consisting of a single VHT format packet is generated using the `wlanWaveformGenerator` function. The waveform is passed through a TGac channel and noise is added. The packet is synchronized and decoded to recover the PSDU. The SNR is estimated and compared against thresholds to determine which MCS is suitable for transmission of the next packet. This figure shows the processing for each packet.



Waveform Configuration

An IEEE 802.11ac VHT transmission is simulated in this example. The VHT waveform properties are specified in a `wlanVHTConfig` configuration object. In this example the object is initially configured

for a 40 MHz channel bandwidth, single transmit antenna and QPSK rate-1/2 (MCS 1). The MCS for the subsequent packets is changed by the algorithm throughout the simulation.

```
cfgVHT = wlanVHTConfig;
cfgVHT.ChannelBandwidth = 'CBW40'; % 40 MHz channel bandwidth
cfgVHT.MCS = 1;                    % QPSK rate-1/2
cfgVHT.APEPLength = 4096;          % APEP length in bytes

% Set random stream for repeatability of results
s = rng(21);
```

Channel Configuration

In this example a TGac N-LOS channel model is used with delay profile Model-D. For Model-D when the distance between the transmitter and receiver is greater than or equal to 10 meters, the model is NLOS. This is described further in `wlanTGacChannel`.

```
tgacChannel = wlanTGacChannel;
tgacChannel.DelayProfile = 'Model-D';
tgacChannel.ChannelBandwidth = cfgVHT.ChannelBandwidth;
tgacChannel.NumTransmitAntennas = 1;
tgacChannel.NumReceiveAntennas = 1;
tgacChannel.TransmitReceiveDistance = 20; % Distance in meters for NLOS
tgacChannel.RandomStream = 'mt19937ar with seed';
tgacChannel.Seed = 0;
tgacChannel.NormalizeChannelOutputs = false;

% Set the sampling rate for the channel
sr = wlanSampleRate(cfgVHT);
tgacChannel.SampleRate = sr;
```

Rate Control Algorithm Parameters

Typically RCAs use channel quality or link performance metrics, such as SNR or packet error rate, for rate selection. The RCA presented in this example estimates the SNR of a received packet. On reception, the estimated SNR is compared against a predefined threshold. If the SNR exceeds the predefined threshold then a new MCS is selected for transmitting the next packet. The `rcaAttack` and `rcaRelease` controls smooth rate changes to avoid changing rates prematurely. The SNR must exceed the `threshold` + `rcaAttack` value to increase the MCS and must be under the `threshold` - `rcaRelease` value to decrease the MCS. In this simulation `rcaAttack` and `rcaRelease` are set to conservatively increase the MCS and aggressively reduce it. For the `threshold` values selected for the scenario simulated in this example, a small number of packet errors are expected. These settings may not be suitable for other scenarios.

```
rcaAttack = 1;  % Control the sensitivity when MCS is increasing
rcaRelease = 0; % Control the sensitivity when MCS is decreasing
threshold = [11 14 19 20 25 28 30 31 35];
snrUp = [threshold inf]+rcaAttack;
snrDown = [-inf threshold]-rcaRelease;
snrInd = cfgVHT.MCS; % Store the start MCS value
```

Simulation Parameters

In this simulation `numPackets` packets are transmitted through a TGac channel, separated by a fixed idle time. The channel state is maintained throughout the simulation, therefore the channel evolves slowly over time. This evolution slowly changes the resulting SNR measured at the receiver. Since the

TGac channel changes very slowly over time, here an SNR variation at the receiver visible over a short simulation can be forced using the `walkSNR` parameter to modify the noise power:

1. Setting `walkSNR` to true generates a varying SNR by randomly setting the noise power per packet during transmission. The SNR walks between 14-33 dB (using the `amplitude` and `meanSNR` variables).
2. Setting `walkSNR` to false fixes the noise power applied to the received waveform, so that channel variations are the main source of SNR changes at the receiver.

```
numPackets = 100; % Number of packets transmitted during the simulation
walkSNR = true;

% Select SNR for the simulation
if walkSNR
    meanSNR = 22;   % Mean SNR
    amplitude = 14; % Variation in SNR around the average mean SNR value
    % Generate varying SNR values for each transmitted packet
    baseSNR = sin(linspace(1,10,numPackets))*amplitude+meanSNR;
    snrWalk = baseSNR(1); % Set the initial SNR value
    % The maxJump controls the maximum SNR difference between one
    % packet and the next
    maxJump = 0.5;
else
    % Fixed mean SNR value for each transmitted packet. All the variability
    % in SNR comes from a time varying radio channel
    snrWalk = 22; %#ok<UNRCH>
end

% To plot the equalized constellation for each spatial stream set
% displayConstellation to true
displayConstellation = false;
if displayConstellation
    ConstellationDiagram = comm.ConstellationDiagram; %#ok<UNRCH>
    ConstellationDiagram.ShowGrid = true;
    ConstellationDiagram.Name = 'Equalized data symbols';
end

% Define simulation variables
snrMeasured = zeros(1,numPackets);
MCS = zeros(1,numPackets);
ber = zeros(1,numPackets);
packetLength = zeros(1,numPackets);
```

Processing Chain

The following processing steps occur for each packet:

1. A PSDU is created and encoded to create a single packet waveform.
2. A fixed idle time is added between successive packets.
3. The waveform is passed through an evolving TGac channel.
4. AWGN is added to the transmitted waveform to create the desired average SNR per active subcarrier after OFDM demodulation.
5. This local function `processPacket` passes the transmitted waveform through the TGac channel, performs receiver processing, and SNR estimation.

6. The VHT-LTF is extracted from the received waveform. The VHT-LTF is OFDM demodulated and channel estimation is performed.
7. The VHT Data field is extracted from the synchronized received waveform.
8. Noise estimation is performed using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers.
9. The estimated SNR for each packet is compared against the threshold, the comparison is used to adjust the MCS for the next packet.
10. The PSDU is recovered using the extracted VHT-Data field.

For simplicity, this example assumes:

1. Fixed bandwidth and antenna configuration for each transmitted packet.
2. There is no explicit feedback packet to inform the transmitter about the suggested MCS setting for the next packet. The example assumes that this information is known to the transmitter before transmitting the subsequent packet.
3. Fixed idle time of 0.5 milliseconds between packets.

```
for numPkt = 1:numPackets
    if walkSNR
        % Generate SNR value per packet using random walk algorithm biased
        % towards the mean SNR
        snrWalk = 0.9*snrWalk+0.1*baseSNR(numPkt)+rand(1)*maxJump*2-maxJump;
    end

    % Generate a single packet waveform
    txPSDU = randi([0,1],8*cfgVHT.PSDULength,1,'int8');
    txWave = wlanWaveformGenerator(txPSDU,cfgVHT,'IdleTime',5e-4);

    % Receive processing, including SNR estimation
    y = processPacket(txWave,snrWalk,tgacChannel,cfgVHT);

    % Plot equalized symbols of data carrying subcarriers
    if displayConstellation && ~isempty(y.EstimatedSNR)
        release(ConstellationDiagram);
        ConstellationDiagram.ReferenceConstellation = wlanReferenceSymbols(cfgVHT);
        ConstellationDiagram.Title = ['Packet ' int2str(numPkt)];
        ConstellationDiagram(y.EqDataSym(:));
        drawnow
    end

    % Store estimated SNR value for each packet
    if isempty(y.EstimatedSNR)
        snrMeasured(1,numPkt) = NaN;
    else
        snrMeasured(1,numPkt) = y.EstimatedSNR;
    end

    % Determine the length of the packet in seconds including idle time
    packetLength(numPkt) = y.RxWaveformLength/sr;

    % Calculate packet error rate (PER)
    if isempty(y.RxPSDU)
        % Set the PER of an undetected packet to NaN
        ber(numPkt) = NaN;
    else
```

```
        [~,ber(numPkt)] = biterr(y.RxPSDU,txPSDU);
    end

    % Compare the estimated SNR to the threshold, and adjust the MCS value
    % used for the next packet
    MCS(numPkt) = cfgVHT.MCS; % Store current MCS value
    increaseMCS = (mean(y.EstimatedSNR) > snrUp((snrInd==0)+snrInd));
    decreaseMCS = (mean(y.EstimatedSNR) <= snrDown((snrInd==0)+snrInd));
    snrInd = snrInd+increaseMCS-decreaseMCS;
    cfgVHT.MCS = snrInd-1;
end
```

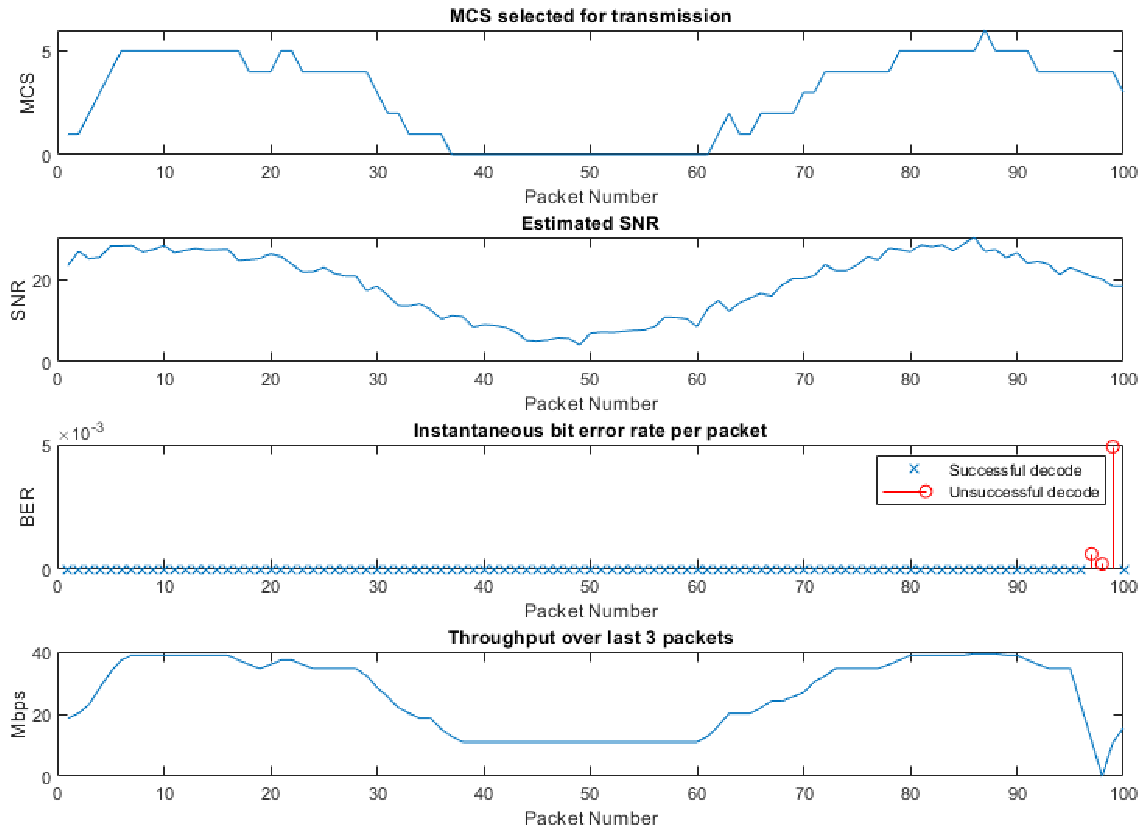
Display and Plot Simulation Results

This example plots the variation of MCS, SNR, BER, and data throughput over the duration of the simulation.

1. The MCS used to transmit each packet is plotted. When compared to the estimated SNR, you can see the MCS selection is dependent on the estimated SNR.
2. The bit error rate per packet depends on the channel conditions, SNR, and MCS used for transmission.
3. The throughput is maximized by varying the MCS according to the channel conditions. The throughput is calculated using a sliding window of three packets. For each point plotted, the throughput is the number of data bits, successfully recovered over the duration of three packets. The length of the sliding window can be increased to further smooth the throughput. You can see drops in the throughput either when the MCS decreases or when a packet error occurs.

```
% Display and plot simulation results
disp(['Overall data rate: ' num2str(8*cfgVHT.APEPLength*(numPackets-numel(find(ber)))/sum(packetL
disp(['Overall packet error rate: ' num2str(numel(find(ber))/numPackets)]);

plotResults(ber,packetLength,snrMeasured,MCS,cfgVHT);

% Restore default stream
rng(s);
```

```
Overall data rate: 20.631 Mbps
Overall packet error rate: 0.03
```



Conclusion and Further Exploration

This example uses a closed-loop rate control scheme where knowledge of the MCS used for subsequent packet transmission is assumed to be available to the transmitter.

In this example the variation in MCS over time due to the received SNR is controlled by the `threshold`, `rcaAttack` and `rcaRelease` parameters. The `rcaAttack` and `rcaRelease` are used as controls to smooth the rate changes, this is to avoid changing rates prematurely. Try changing the `rcaRelease` control to two. In this case, the decrease in MCS is slower to react when channel conditions are not good, resulting in higher BER.

Try setting the `displayConstellation` to true in order to plot the equalized symbols per received packet, you can see the modulation scheme changing over time. Also try setting `walkSNR` to false in order to visualize the MCS change per packet. Here the variability in SNR is only caused by the radio channel, rather than the combination of channel and random walk.

Further exploration includes using an alternate RCA scheme, more realistic MCS variation including changing number of space time streams, packet size and enabling STBC for subsequent transmitted packets.

Selected Bibliography

1 IEEE Std 802.11™-2020. 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.

Local Functions

The following local functions are used in this example:

- `processPacket`: Add channel impairments and decode receive packet
- `plotResults`: Plot the simulation results

```
function Y = processPacket(txWave,snrWalk,tgacChannel,cfgVHT)
    % Pass the transmitted waveform through the channel, perform
    % receiver processing, and SNR estimation.

    chanBW = cfgVHT.ChannelBandwidth; % Channel bandwidth
    % Set the following parameters to empty for an undetected packet
    estimatedSNR = [];
    eqDataSym = [];
    noiseVarVHT = [];
    rxPSDU = [];

    % Get the OFDM info
    ofdmInfo = wlanVHTOFDMInfo('VHT-Data',cfgVHT);

    % Pass the waveform through the fading channel model
    rxWave = tgacChannel(txWave);

    % Account for noise energy in nulls so the SNR is defined per
    % active subcarrier
    packetSNR = snrWalk-10*log10(ofdmInfo.FFTLength/ofdmInfo.NumTones);

    % Add noise
    rxWave = awgn(rxWave,packetSNR);
    rxWaveformLength = size(rxWave,1); % Length of the received waveform

    % Recover packet
    ind = wlanFieldIndices(cfgVHT); % Get field indices
    pktOffset = wlanPacketDetect(rxWave,chanBW); % Detect packet

    if ~isempty(pktOffset) % If packet detected
        % Extract the L-LTF field for fine timing synchronization
        LLTFSearchBuffer = rxWave(pktOffset+(ind.LSTF(1):ind.LSIG(2)),:);

        % Start index of L-LTF field
        finePktOffset = wlanSymbolTimingEstimate(LLTFSearchBuffer,chanBW);

        % Determine final packet offset
        pktOffset = pktOffset+finePktOffset;

        if pktOffset<15 % If synchronization successful
            % Extract VHT-LTF samples from the waveform, demodulate and
            % perform channel estimation
            VHTLTF = rxWave(pktOffset+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
```

```
            demodVHTLTF = wlanVHTLTFDemodulate(VHTLTF,cfgVHT);
            [chanEstVHTLTF,chanEstSSPilots] = wlanVHTLTFChannelEstimate(demodVHTLTF,cfgVHT);

            % Extract VHT data field
            vhtdata = rxWave(pktOffset+(ind.VHTData(1):ind.VHTData(2)),:);

            % Estimate the noise power in VHT data field
            noiseVarVHT = vhtNoiseEstimate(vhtdata,chanEstSSPilots,cfgVHT);

            % Recover equalized symbols at data carrying subcarriers using
            % channel estimates from VHT-LTF
            [rxPSDU,~,eqDataSym] = wlanVHTDataRecover(vhtdata,chanEstVHTLTF,noiseVarVHT,cfgVHT);

            % SNR estimation per receive antenna
            powVHTLTF = mean(VHTLTF.*conj(VHTLTF));
            estSigPower = powVHTLTF-noiseVarVHT;
            estimatedSNR = 10*log10(mean(estSigPower./noiseVarVHT));
        end
    end

    % Set output
    Y = struct( ...
        'RxPSDU',           rxPSDU, ...
        'EqDataSym',        eqDataSym, ...
        'RxWaveformLength', rxWaveformLength, ...
        'NoiseVar',         noiseVarVHT, ...
        'EstimatedSNR',     estimatedSNR);

end

function plotResults(ber,packetLength,snrMeasured,MCS,cfgVHT)
    % Visualize simulation results

    figure('Outerposition',[50 50 900 700])
    subplot(4,1,1);
    plot(MCS);
    xlabel('Packet Number')
    ylabel('MCS')
    title('MCS selected for transmission')

    subplot(4,1,2);
    plot(snrMeasured);
    xlabel('Packet Number')
    ylabel('SNR')
    title('Estimated SNR')

    subplot(4,1,3);
    plot(find(ber==0),ber(ber==0),'x')
    hold on; stem(find(ber>0),ber(ber>0),'or')
    if any(ber)
        legend('Successful decode','Unsuccessful decode')
    else
        legend('Successful decode')
    end
    xlabel('Packet Number')
    ylabel('BER')
    title('Instantaneous bit error rate per packet')
```

```
    subplot(4,1,4);
    windowLength = 3; % Length of the averaging window
    movDataRate = movsum(8*cfgVHT.APEPLength.*(ber==0),windowLength)./movsum(packetLength,windowL
    plot(movDataRate)
    xlabel('Packet Number')
    ylabel('Mbps')
    title(sprintf('Throughput over last %d packets',windowLength))

end
```

7

System-Level Simulation

Spatial Reuse with BSS Coloring in 802.11ax Network Simulation

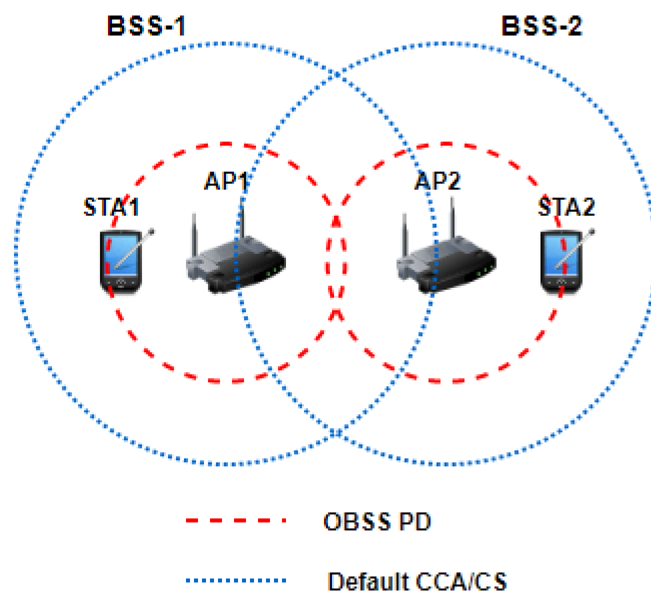
This example demonstrates how to simulate the impact of spatial reuse (SR) with basic service set (BSS) coloring on the throughput of an IEEE® Std 802.11ax™ [1 on page 7-11] network by using WLAN Toolbox™ and the Communications Toolbox™ Wireless Network Simulation Library.

Using this example, you can:

1. Create and configure an 802.11ax network topology consisting of four BSSs. Each BSS includes an access point (AP) and a station (STA).
2. Assign BSS colors to each BSS and configure the overlapping BSS packet detect (OBSS PD) threshold to simulate non-spatial reuse groups OBSS PD, as defined in IEEE Std 802.11ax-2021 [1 on page 7-11].
3. Simulate the network with and without BSS coloring to compare the throughput of each BSS.

IEEE 802.11ax OBSS PD-based Spatial Reuse Operation

In dense IEEE 802.11 networks, multiple BSSs can operate in the same channel due to the limited spectrum. This results in an inefficient paradigm that causes network congestion and slowdown. To optimize efficient reutilization of the frequency spectrum in dense network scenarios, IEEE Std 802.11ax-2021 [1 on page 7-11] introduced the OBSS PD-based SR operation. The OBSS PD-based SR operation improves the network performance of BSSs operating in the same channel by increasing the number of parallel transmissions. To increase the number of parallel transmissions, the standard adjusts the clear channel assessment/carrier sense (CCA/CS) threshold for the detected OBSS transmissions to a new value called the OBSS PD threshold. The OBSS PD threshold is higher than the default CCA/CS threshold. This figure illustrates the SR operation in an OBSS. The network topology consists of two BSSs, each containing an AP and an STA.

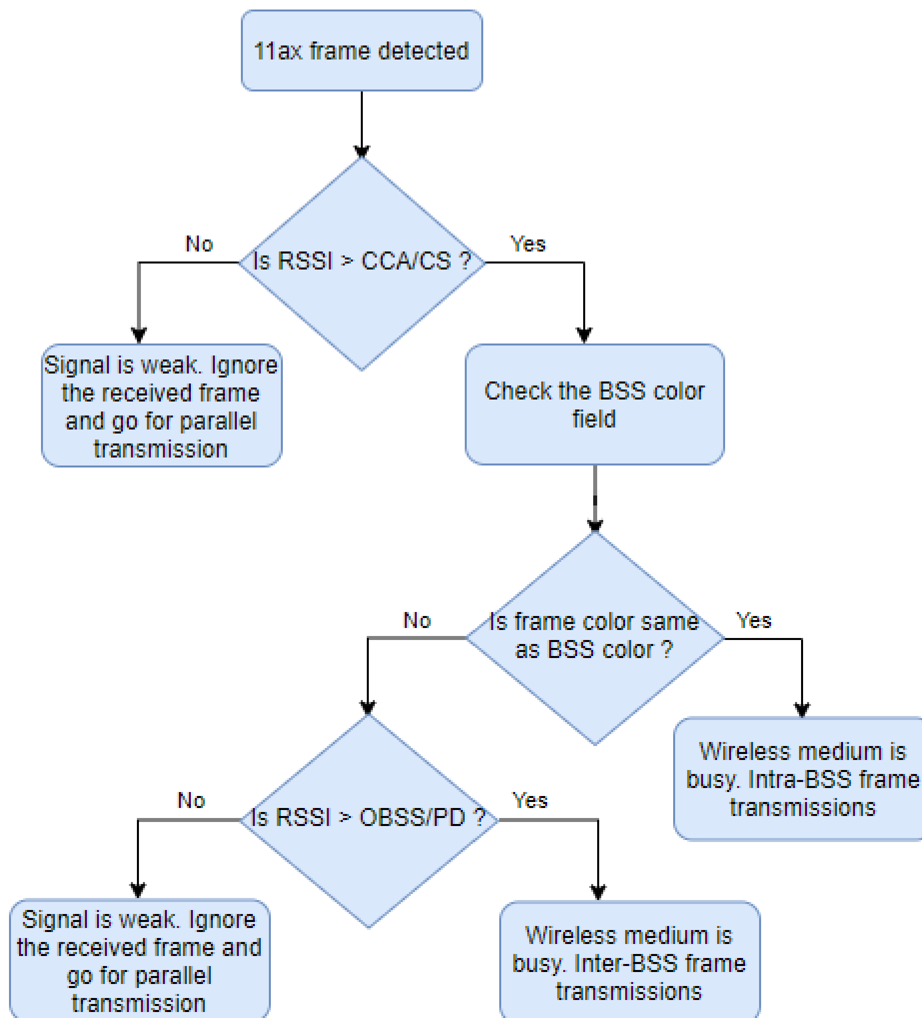


The default CCA/CS threshold (denoted by blue dashed lines) does not enable simultaneous transmissions between AP1 and AP2. If the nodes in BSS-1 occupy the channel for transmission,

transmission in BSS-2 is deferred. However, by optimally selecting the value of the OBSS PD threshold (denoted by red dashed lines), both APs can simultaneously transmit to their corresponding STAs. Therefore, the SR operation improves channel utilization, resulting in better throughput.

BSS Coloring

A receiver device identifies a frame as an intra-BSS frame if the color of the detected frame matches with the BSS color of the receiver. If the receiver identifies the frame as an intra-BSS frame, the transmitter and receiver belong to the same BSS and the receiver defers its transmission using the CSMA/CA procedure. If the detected frame color is different from the BSS color, the frame is an inter-BSS frame and the BSS coloring algorithm checks whether the received signal strength indicator (RSSI) of the received frame is greater than the OBSS PD threshold. If the RSSI value of the received frame is greater than the OBSS PD threshold, the device considers the wireless medium as busy and defers its transmission. If the RSSI value of the received frame is less than the OBSS PD threshold, the device ignores the received frame and continues contending for transmission.



Check for Support Package Installation

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Simulate 802.11ax Network with BSS Coloring

This example demonstrates the communication in an 802.11ax network that, by default, has four BSSs, each containing one AP and one station.

Configure Simulation Parameters

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation, including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again and average the results over multiple simulations.

```
rng(1,"combRecursive")
```

Specify the simulation time in seconds. To visualize a live state transition plot for all of the nodes, set the `showLiveStateTransitionPlot` variable to `true`.

```
simulationTime = 0.5;
showLiveStateTransitionPlot = true;
```

Configure the number of BSSs to create.

```
numBSS = 4;
```

Initialize the wireless network simulator by using the `wirelessNetworkSimulator` object.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Configure Nodes

Specify the BSS color for each BSS. A nonzero BSS color enables spatial reuse at the node. Assign a unique BSS color in the device configuration of each AP. Association of STAs to the AP automatically

configures the STAs with the BSS color of their corresponding APs. Set the OBSS PD threshold for each BSS. This example uses the same OBSS PD threshold value for all the APs and STAs.

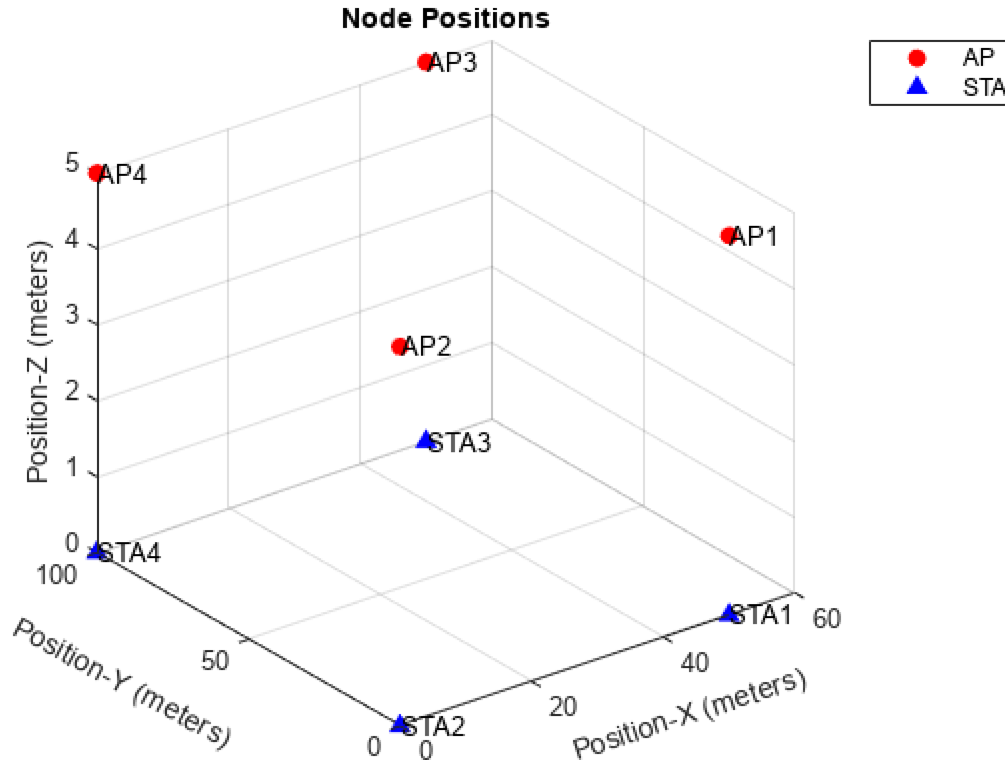
```
bssColor = (1:numBSS);
obssPDThreshold = -72*ones(1,numBSS);
```

Configure the positions of the AP and STA nodes.

```
apPositions = zeros(numBSS, 3);
staPositions = zeros(numBSS, 3);
for bssIdx = 1:numBSS
    apPositions(bssIdx, :) = [50*mod(bssIdx,2) 100*(ceil(bssIdx/2)-1) 5];
    staPositions(bssIdx, :) = [50*mod(bssIdx,2) 100*(ceil(bssIdx/2)-1) 0];
end
```

Visualize the node positions in a 3-D plot by using the `hVisualizeNodePositions` helper function.

```
hVisualizeNodePositions(apPositions, staPositions)
```



For each BSS, create two `wlanDeviceConfig` objects to initialize the configuration parameters for the AP and the STA. Specify the `Mode`, `BSSColor`, `OBSSPDThreshold`, and `DisableRTS` properties of each AP. Specify the `Mode`, `OBSSPDThreshold`, and `DisableRTS` properties of each STA. During the association, each STA automatically adopts the same BSS color as its corresponding AP.

Create the AP and STA nodes by using the `wlanNode` object. Associate each STA to its corresponding AP by using the `associateStations` object function. Configure the AP with continuous downlink application traffic to its associated STA by using the `FullBufferTraffic` option of the `associateStations` object function.

```
for bssIndex = 1:numBSS
    % Configure AP and STA nodes
    apConfig = wlanDeviceConfig(Mode="AP",BSSColor=bssColor(bssIndex),OBSSPDThreshold=obssPDThres
    staConfig = wlanDeviceConfig(Mode="STA",OBSSPDThreshold=obssPDThreshold(bssIndex),DisableRTS=
    % Create AP and STA nodes
    apNodes(bssIndex) = wlanNode(Name=strcat("AP",num2str(bssIndex)),DeviceConfig=apConfig,Positi
    staNodes(bssIndex) = wlanNode(Name=strcat("STA",num2str(bssIndex)),DeviceConfig=staConfig,Pos
    % Associate STAs with AP
    associateStations(apNodes(bssIndex),staNodes(bssIndex),FullBufferTraffic="DL")
end
nodes = [apNodes staNodes];
```

To ensure all the nodes are configured properly, use the `hCheckWLANNodesConfiguration` helper function.

```
hCheckWLANNodesConfiguration(nodes)
```

For more information about node configuration parameters, see the “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31 example.

Configure Wireless Channel

To create a random TGax fading channel between each node, this example uses the `hSLSTGaxMultiFrequencySystemChannel` helper object.

Add the channel model to the wireless network simulator by using the `addChannelModel` object function of the `wirelessNetworkSimulator` object.

```
channel = hSLSTGaxMultiFrequencySystemChannel(nodes);
addChannelModel(networkSimulator,channel.ChannelFcn)
```

Simulation

Add your nodes to the wireless network simulator.

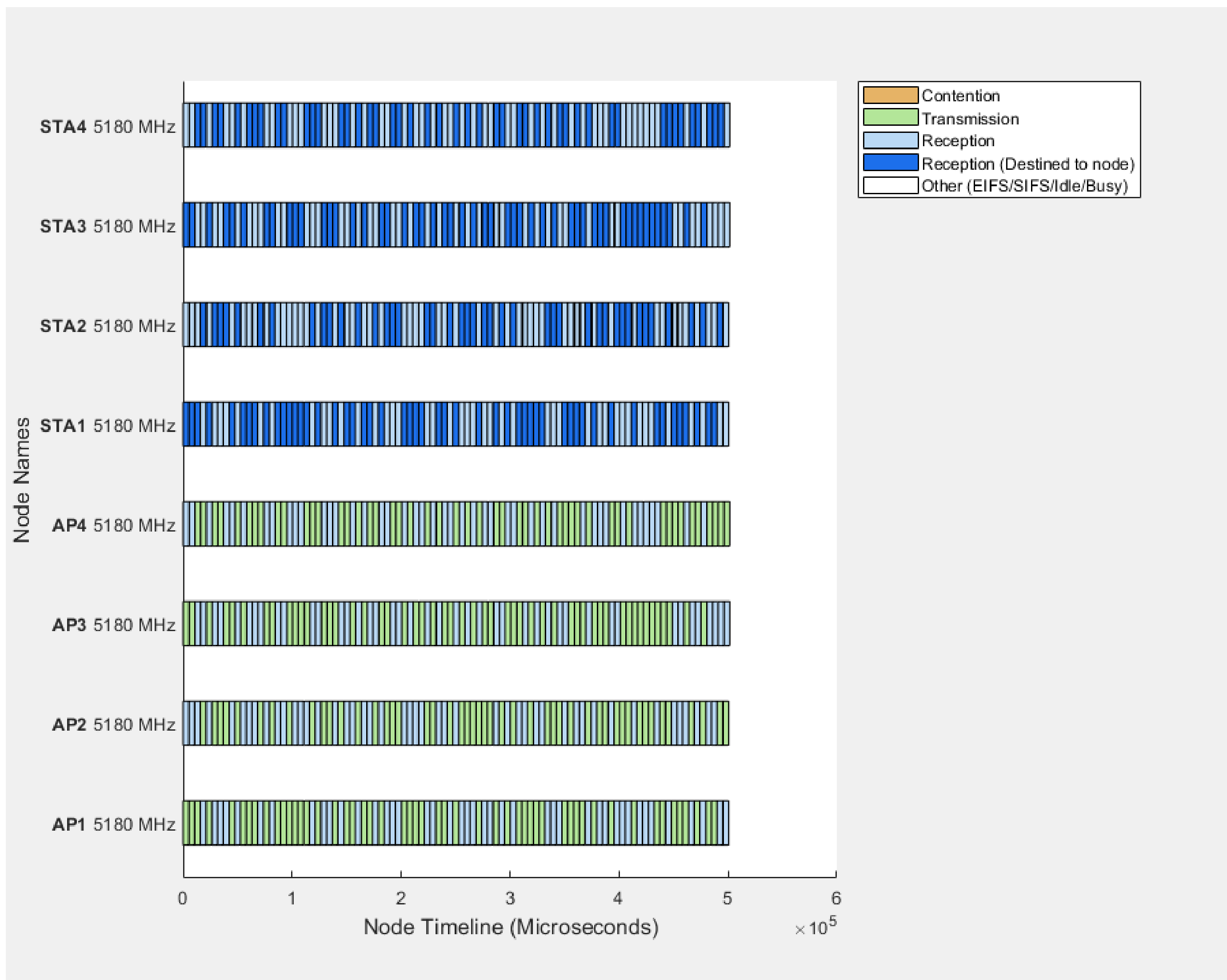
```
addNodes(networkSimulator,nodes)
```

To visualize the state transitions at each node, use the `hSimulationPlotViewer` helper function.

```
hSimulationPlotViewer(showLiveStateTransitionPlot,nodes);
```

Run the simulation for the specified simulation time.

```
run(networkSimulator,simulationTime)
```



Results

Retrieve the APP, MAC, and PHY statistics at each node by using the `statistics` object function. Store the throughput of each AP, calculated by the simulation, in the `perBSSThroughput` variable. Because only the APs are transmitters in this example, the throughput of each AP is the same as the throughput of the corresponding BSS.

```
% Statistics
stats = statistics(nodes);
% Throughput of each AP with SR enabled
perBSSThroughputSR = zeros(1,numBSS);
for bssIndex = 1:numBSS
    perBSSThroughputSR(bssIndex) = (stats(bssIndex).MAC.TransmittedPayloadBytes*8)/(simulationTi
end
```

Simulate 802.11ax Network Without BSS Coloring

In this section of the example, you remove the BSS coloring from the network and simulate it again.

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Configure Nodes

To disable SR at each BSS, set the BSS color to `0`.

```
bssColor = zeros(1,numBSS);
```

Specify the BSS color as `0` in the device configurations of the APs and the STAs. Create the AP and STA nodes and associate each STA to its corresponding AP. Configure the AP with continuous downlink application traffic to its associated STA.

```
for bssIndex = 1:numBSS
    % Configure AP and STA nodes
    apConfig = wlanDeviceConfig(Mode="AP",BSSColor=bssColor(bssIndex),DisableRTS=true);
    staConfig = wlanDeviceConfig(Mode="STA",DisableRTS=true);
    % Create AP and STA nodes
    apNodes(bssIndex) = wlanNode(Name=strcat("AP",num2str(bssIndex)),DeviceConfig=apConfig,Posit
    staNodes(bssIndex) = wlanNode(Name=strcat("STA",num2str(bssIndex)),DeviceConfig=staConfig,Pos
    % Associate STAs with AP
    associateStations(apNodes(bssIndex),staNodes(bssIndex),FullBufferTraffic="DL")
end
nodes = [apNodes staNodes];
```

To ensure all the nodes are configured properly, use the `hCheckWLANNodesConfiguration` helper function.

```
hCheckWLANNodesConfiguration(nodes)
```

Configure Wireless Channel

Add the channel model to the wireless network simulator.

```
channel = hSLSTGaxMultiFrequencySystemChannel(nodes);
addChannelModel(networkSimulator,channel.ChannelFcn)
```

Simulation

Add your nodes to the wireless network simulator.

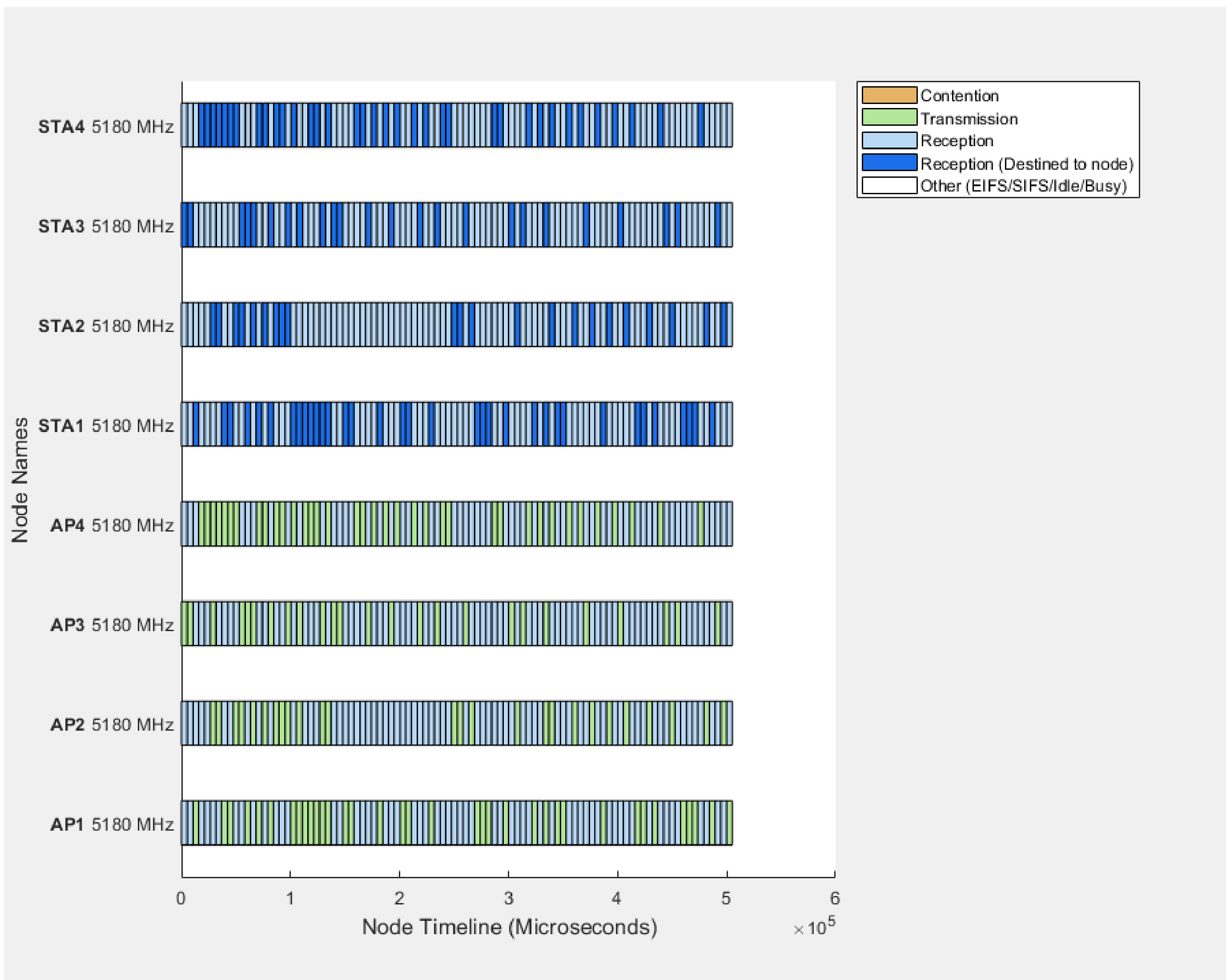
```
addNodes(networkSimulator,nodes)
```

To visualize the state transitions at each node, use the `hSimulationPlotViewer` helper function.

```
hSimulationPlotViewer(showLiveStateTransitionPlot,nodes);
```

Run the simulation for the specified simulation time.

```
run(networkSimulator,simulationTime)
```



Results

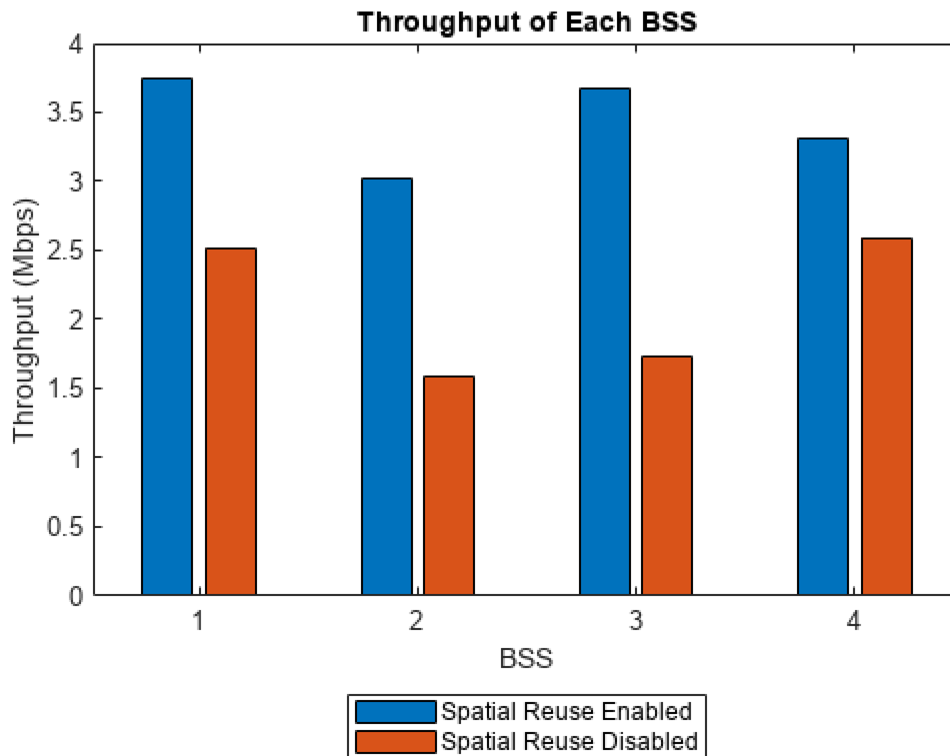
Calculate the throughput of each AP and store it in the `perBSSThroughputNoSR` variable. Because only the APs are transmitters in this example, the throughput of each AP is the same as the throughput of the corresponding BSS.

```
% Statistics
stats = statistics(nodes);

% Throughput of each AP with SR disabled
perBSSThroughputNoSR = zeros(1,numBSS);
for bssIndex = 1:numBSS
    perBSSThroughputNoSR(bssIndex) = (stats(bssIndex).MAC.TransmittedPayloadBytes*8)/(simulation
end
```

Compare Throughput With and Without BSS Coloring at Each BSS

This plot shows the throughput of each BSS with and without BSS coloring. No BSS coloring indicates the SR operation is disabled. Note that the throughput of each BSS improves considerably for the modeled scenario by enabling the SR operation.

```
figure
bar([perBSSThroughputSR; perBSSThroughputNoSR]')
xlabel("BSS")
ylabel("Throughput (Mbps)")
title("Throughput of Each BSS")
legend(["Spatial Reuse Enabled" "Spatial Reuse Disabled"],Location="southoutside")
```



Appendix

The example uses these helper functions and objects.

- `hCheckWLANNodesConfiguration` — Check the node parameters configuration
- `hSimulationPlotViewer` — View the state transition and performance metric figures
- `hSLSTGaxMultiFrequencySystemChannel` — Return a system channel object
- `hSLSTGaxAbstractSystemChannel` — Return a channel manager object for abstracted PHY layer
- `hSLSTGaxSystemChannel` — Return a channel manager object for full PHY layer
- `hSLSTGaxSystemChannelBase` — Return the base channel manager object

- `hVisualizeNodePositions` — Visualize the node positions

References

1. Institute of Electrical and Electronics Engineers (IEEE). 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 1: Enhancements for High-Efficiency WLAN. IEEE 802.11ax-2021. IEEE, May 19, 2021. https://doi.org/10.1109/IEEESTD.2021.9442429.

See Also

Functions
`statistics`

Objects
`wlanNode` | `wirelessNetworkSimulator` | `wlanDeviceConfig`

See Also

More About

- "Get Started with WLAN System-Level Simulation in MATLAB" on page 7-31
- "802.11ax Multinode System-Level Simulation of Residential Scenario" on page 7-40
- "802.11ax Downlink OFDMA Multinode System-Level Simulation" on page 7-12
- "WLAN System-Level Simulation Statistics" on page 7-114

802.11ax Downlink OFDMA Multinode System-Level Simulation

This example shows how to simulate a WLAN multinode downlink (DL) orthogonal frequency-division multiple access (OFDMA) network consisting of an IEEE® 802.11ax™ [1 on page 7-20] access point (AP) and four stations (STAs), by using WLAN Toolbox™ and the Communications Toolbox™ Wireless Network Simulation Library.

Using this example, you can:

- Simulate DL OFDMA communication from the AP to STAs.
- Visualize the time spent by each node in Idle, Contention, Transmission, and Reception state.
- Capture the application layer (APP), medium access control layer (MAC) and physical layer (PHY) statistics for each node.

The simulation results show performance metrics such as throughput, packet latency, and packet loss captured at each node.

Additionally, you can further explore the example by performing these tasks.

- Throughput Comparison of OFDM and OFDMA on page 7-18
- Faster Execution Using Parallel Simulation Runs on page 7-19

OFDM and OFDMA

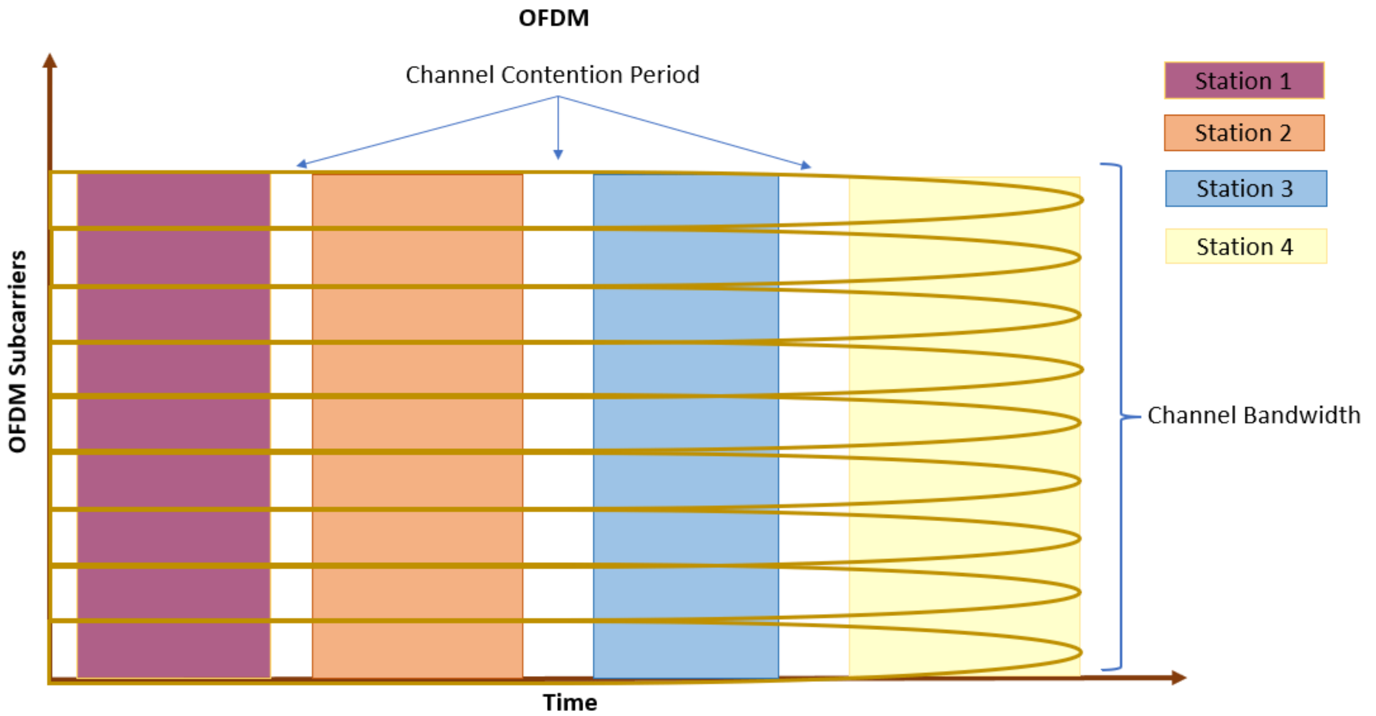
The IEEE 802.11ax standard introduces significant enhancements over the existing 802.11ac standard [2 on page 7-20]. One of the key improvements is OFDMA, which is an extension of OFDM digital modulation technology into a multiuser (MU) environment. The fundamental objective of OFDMA is to efficiently use the available frequency space. OFDMA partitions the channel bandwidth into multiple mutually exclusive sub-bands, called resource units (RUs). By partitioning the channel bandwidth, multiple users can access the channel simultaneously. As a result, 802.11ax supports concurrent transmissions of packets to multiple users.

For example, a conventional 20 MHz channel can be partitioned into a maximum of nine subchannels. An 802.11ax AP can simultaneously transmit packets to nine 802.11ax STAs by using OFDMA. The simultaneous transmission of frames reduces excessive contention overhead at the MAC layer and minimizes the PHY layer preamble overhead. In OFDMA, the AP controls the allocation of RUs.

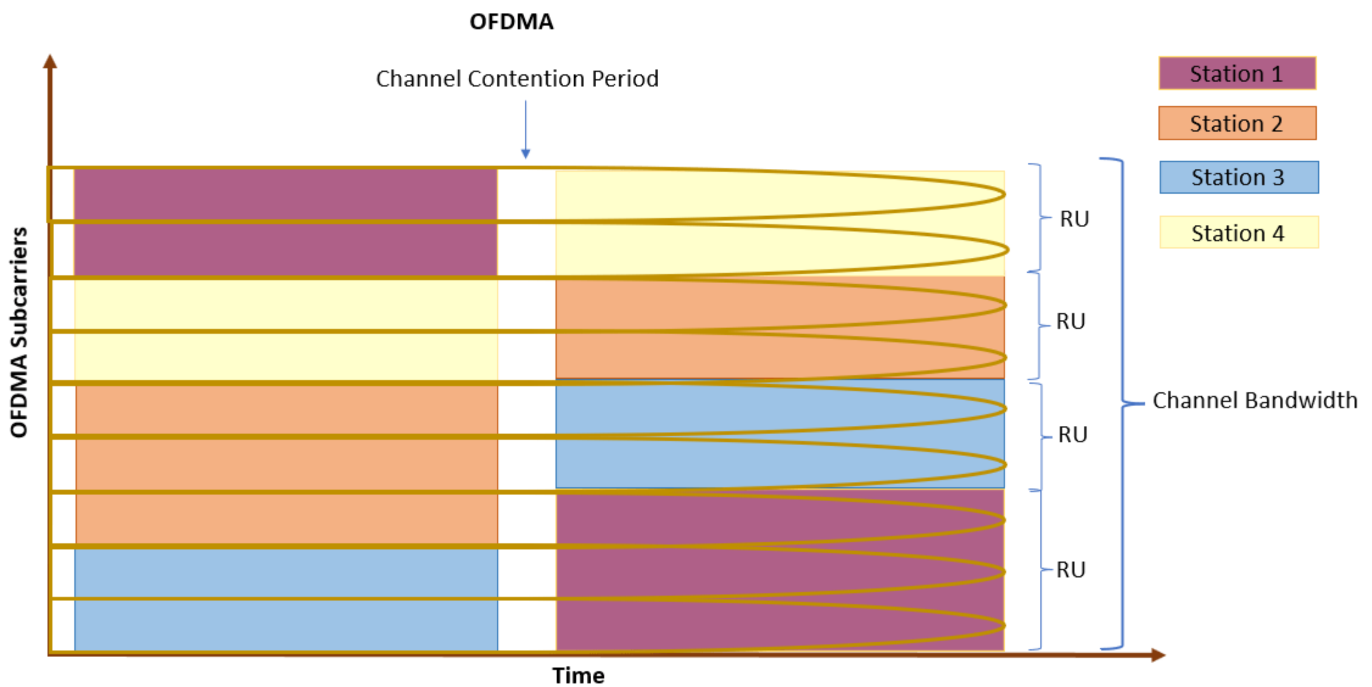
The 802.11ax standard specifies two types of OFDMA:

- **DL OFDMA -** The AP transmits packets to multiple STAs simultaneously using a different RU for each STA.
- **Uplink (UL) OFDMA -** Multiple STAs transmit packets to an AP simultaneously, with each STA using a different RU.

In this figure, the 802.11n/ac/ax AP transmits DL frames to four OFDM STAs independently over time. The AP uses the entire channel bandwidth to communicate with a single OFDM STA. Similarly, an OFDM STA uses the entire channel bandwidth to communicate with an 802.11n/ac/ax AP in an UL OFDM transmission.



This figure shows an OFDMA transmission. The 802.11ax AP partitions the channel bandwidth into RUs for four OFDMA STAs on a continuous basis for simultaneous DL transmissions.



System-Level Simulation Scenario

The example creates, configures, and simulates this 802.11ax system-level scenario, consisting of one AP and four associated STAs.

In the preceding figure:

1. AP transmits DL data to all the STAs simultaneously.
2. STAs respond with a UL acknowledgment frame upon receiving the DL data frames from the AP.

Check for Support Package Installation

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck
```

Configure Simulation Parameters

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation, including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive")
```

Specify the simulation time in seconds. To visualize a live state transition plot for all of the nodes, set the `showLiveStateTransitionPlot` variable to `true`.

```
simulationTime = 0.1;
showLiveStateTransitionPlot = true;
```

This example uses abstracted MAC and PHY at each WLAN node because OFDMA is only supported for abstracted MAC and PHY. For more information about abstracted MAC and PHY, see the “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31 example.

```
PHYAbstractionMethod = tgax-evaluation-met...;
```

Configure WLAN Scenario

Initialize the wireless network simulator by using the `wirelessNetworkSimulator` object.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Nodes

Specify the number of nodes in the network. The example scenario consists of one AP and four associated STAs.

```
numNodes = 5;
numSTAs = numNodes-1;
```

Specify the positions of the AP and STA nodes. The `apPosition` and `staPositions` vectors specify the *x*-, *y*-, and *z*- Cartesian coordinates of the AP and STAs, respectively. Units are in meters.

```
apPosition = [0 0 0];
staPositions = [((30/numSTAs).*(1:numSTAs))' ((30/numSTAs).*(numSTAs:-1:1))' zeros(numSTAs,1)];
```

To model the scenario, create an AP and four STAs using `wlanNode` and `wlanDeviceConfig` objects. The `wlanDeviceConfig` object enables you to specify the configuration parameters for the AP and STAs. Create two WLAN device configuration objects, one for the AP and the other for the STAs. You can use the same device configuration object to configure all the STAs. To configure an AP and a STA, set the `Mode` property of the `wlanDeviceConfig` object to "AP" and "STA" respectively. Configure the `TransmissionFormat`, `MCS`, `DisableRTS`, `MPDUAggregationLimit`, `TransmitPower` properties of AP device configuration object. Note that AP and STA device configuration objects must have the same value of `MPDUAggregationLimit`.

The example simulates these DL OFDMA frame exchange sequences.

- Enable DL MU frames followed by MU block ack request (BAR) trigger frame to simultaneously solicit UL acknowledgment frames from multiple scheduled STAs.
- Optionally enables MU request to send request to send (RTS)/clear to send (CTS) exchanges by specifying the `DisableRTS` property of the `wlanDeviceConfig` object to `false`.

This example uses the round-robin scheduling strategy to select STAs for each transmission. The assignment of RUs is fixed for a given number of users and bandwidth.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",TransmissionFormat="HE-MU-OFDMA", ...  % AP device co
                              MCS=8,DisableRTS=true, ...
                              MPDUAggregationLimit=5, ...
                              TransmitPower=20);
stationCfg = wlanDeviceConfig(Mode="STA",MPDUAggregationLimit=5);                  % STAs device
```

To create one AP and four STA nodes from the specified configuration, use the `wlanNode` objects. Specify the node names, node positions, and device configuration objects. You can use the `wlanNode` object to create multiple nodes at a time by providing multiple node positions. Specify the PHY abstraction method used by the AP and STAs. Configure the `MACFrameAbstraction` property to indicate that the MAC frames are abstracted by the nodes.

```
accessPoint = wlanNode(Name="AP", ...
    Position=apPosition, ...
    DeviceConfig=accessPointCfg,...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=true);
```

```
stations = wlanNode(Name="STA"+(1:numSTAs), ...
    Position=staPositions, ...
    DeviceConfig=stationCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=true);
```

Create a WLAN network consisting of an AP and four STAs.

```
nodes = [accessPoint stations];
```

To ensure all the nodes are configured properly, use the `hCheckWLANNodesConfiguration` helper function.

```
hCheckWLANNodesConfiguration(nodes)
```

Association

Associate all the STAs to the AP by using the `associateStations` object function of the `wlanNode` object.

```
associateStations(accessPoint,stations);
```

Application Traffic

Create `networkTrafficOnOff` objects to generate an On-Off application traffic pattern. Configure the On-Off application traffic by specifying the application data rate and packet size. The number of objects created is equal to the number of AP-STA pairs that have downlink traffic exchange in the example. Add application traffic from the AP node to the corresponding STA node by using the `addTrafficSource` object function and specifying the associated STA as the destination.

```
% Create four networkTrafficOnOff objects, one for each AP-STA pair in the scenario
for staID=1:numNodes-1
    trafficSource(staID) = networkTrafficOnOff(DataRate=100000,PacketSize=100); %#ok<SAGROW>
    addTrafficSource(accessPoint,trafficSource(staID),DestinationNode=stations(staID),AccessCate
end
```

Wireless Channel

To model a random TGax fading channel between each node, this example uses `hSLSTGaxMultiFrequencySystemChannel` helper function. When you model a fading channel, the probability of packet transmission success depends on these factors.

- Quality of the channel between an AP and STA for the selected RU
- Modulation and coding scheme (MCS)

In this example, the RU allocation does not consider the quality of the channel between the AP and each STA. Therefore, if the channel quality between an AP and a STA is poor and the selected MCS is high, packet loss is likely to occur.

Add the channel model to the wireless network simulator by using the `addChannelModel` object function of the `wirelessNetworkSimulator` object.

```
channel = hSLSTGaxMultiFrequencySystemChannel(nodes);
addChannelModel(networkSimulator,channel.ChannelFcn)
```

Simulation

Add your nodes to the wireless network simulator.

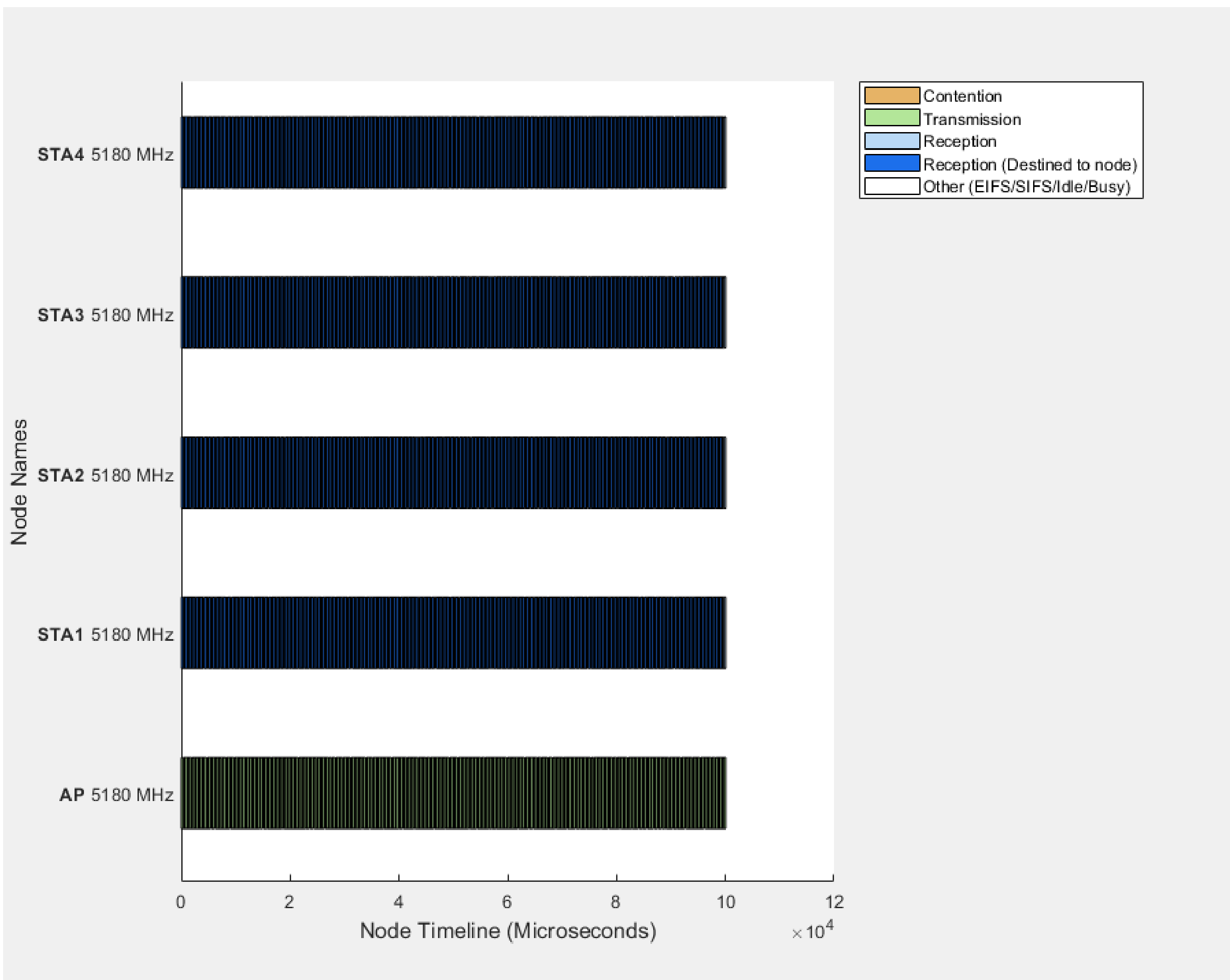
```
addNodes(networkSimulator,nodes)
```

To view the state transition and performance metrics plots, create a `hSimulationPlotViewer` helper object. The simulation shows the plots for all the nodes configured in the simulation. To visualize the state transitions and performance metrics of specific nodes, specify the corresponding node objects as the second argument to the helper object.

```
viewerObj = hSimulationPlotViewer(showLiveStateTransitionPlot,nodes);
```

Run the network simulation for the specified simulation time. The runtime visualization shows the time spent by the AP and the STA in Idle, Contention, Transmission, and Reception state.

```
run(networkSimulator,simulationTime);
```



Results

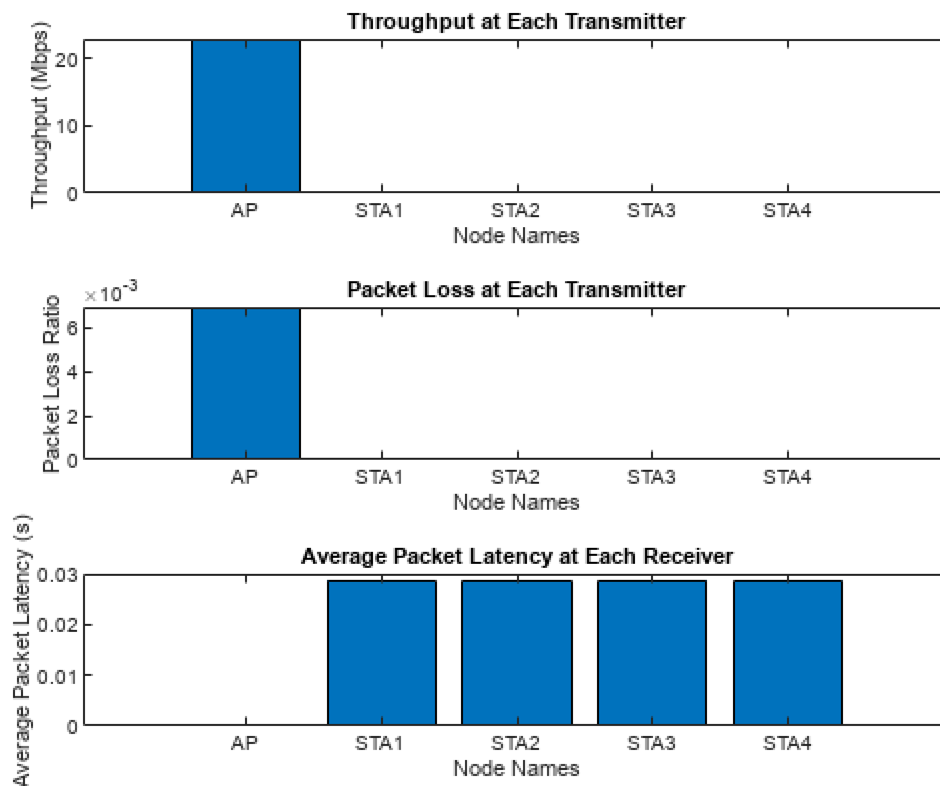
Retrieve the APP, MAC, and PHY statistics at each node by using the `statistics` object function of the `wlanNode` object.

```
stats = statistics(nodes);
```

Plot the performance of each node by using the `plotNetworkStats` object function of the `hSimulationPlotViewer` object. You can visualize these simulation plots.

- Throughput (in Mbps) at each transmitter (AP).
- Packet loss ratio (ratio of unsuccessful data transmissions to the total data transmissions) at each transmitter (AP and STA).
- Average packet latency incurred at each receiver (STAs). The average packet latency shows the average latency that each STA incurs to receive the downlink traffic from the AP.

```
plotNetworkStats(viewerObj,simulationTime)
```



Further Exploration

Throughput Comparison of OFDM and OFDMA

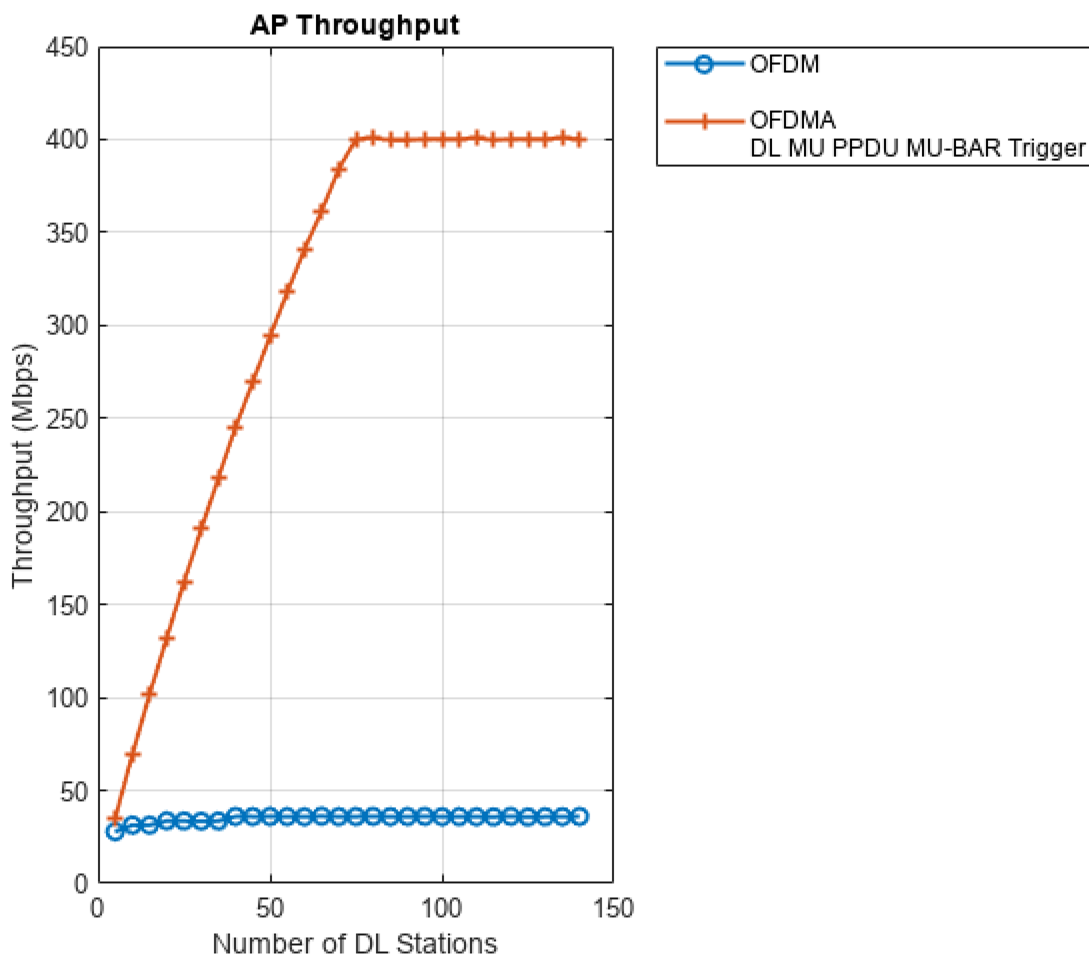
Generate throughput results of these OFDM and OFDMA transmission scenarios by using the `hCompareOFDMvsOFDMAThroughputs` helper function.

- An AP serving a maximum of 140 STAs with OFDM configuration.

- An AP serving a maximum of 140 STAs in an OFDMA configuration.

Plot the throughput results as a function of the number of DL STAs for OFDM and OFDMA configurations. By default, the `hCompareOFDMvsOFDMAThroughputs` helper function plots the static throughput values of OFDM and OFDMA transmissions. To reproduce the results, set the `plotStaticThroughputValues` flag to `false`.

```
plotStaticThroughputValues = true ;
hCompareOFDMvsOFDMAThroughputs(plotStaticThroughputValues);
```



The preceding plot shows the throughput comparison of OFDM and OFDMA. Because of the simultaneous transmissions to multiple users, the throughput obtained by using OFDMA transmission is greater than the throughput obtained by using OFDM transmission. Increasing the number of DL STAs has minimal impact on the OFDM throughput.

Faster Execution Using Parallel Simulation Runs

If you want to run multiple simulations, you can speed up the simulations by enabling parallel computing using the `parfor` loop. The parfor loop is an alternative to the `for` loop. The parfor loop

enables you to execute multiple simulation runs in parallel, thereby reducing the total execution time. To use parfor, you need to install the "Parallel Computing Toolbox"™. For more information about running multiple simulations by using parfor loop, see `hCompareOFDMvsOFDMAThroughputs` helper function.

Appendix

The example uses these helpers:

- `hCheckWLANNodesConfiguration` — Check if the node parameters are configured correctly
- `hSLSTGaxMultiFrequencySystemChannel` — Return a system channel object and set the path loss model
- `hSLSTGaxAbstractSystemChannel` — Return a channel object for abstracted PHY layer
- `hSLSTGaxSystemChannel` — Return a channel object for full PHY layer
- `hSLSTGaxSystemChannelBase` — Return the base channel object
- `hSimulationPlotViewer` — Plot the state transition and performance metrics figures
- `hCompareOFDMvsOFDMAThroughputs` — Retrieve throughput results for OFDM and OFDMA transmission scenarios

References

1. Institute of Electrical and Electronics Engineers (IEEE). 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 1: Enhancements for High-Efficiency WLAN. IEEE 802.11ax-2021. IEEE, May 19, 2021. https://doi.org/10.1109/IEEESTD.2021.9442429.

2. Institute of Electrical and Electronics Engineers (IEEE). 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. IEEE 802.11-2020. IEEE, February 26, 2021. https://doi.org/10.1109/IEEESTD.2021.9363693.

See Also

Functions
`statistics`

Objects
`wirelessNetworkSimulator` | `networkTrafficOnOff` | `wlanNode` | `wlanDeviceConfig`

Related Examples

- "Get Started with WLAN System-Level Simulation in MATLAB" on page 7-31
- "802.11ax Multinode System-Level Simulation of Residential Scenario" on page 7-40
- "Spatial Reuse with BSS Coloring in 802.11ax Network Simulation" on page 7-2
- "WLAN System-Level Simulation Statistics" on page 7-114

Noncollaborative Bluetooth LE Coexistence with WLAN Signal Interference

This example shows how to simulate Bluetooth low energy (LE) noncollaborative coexistence with WLAN interference by using Bluetooth® Toolbox and Communications Toolbox™ Wireless Network Simulation Library.

Using this example, you can:

- Create and configure a Bluetooth LE piconet with Central and Peripheral nodes.
- Analyze the performance of the Bluetooth LE network with and without WLAN interference.
- Visualize Bluetooth LE coexistence with WLAN interference for each Peripheral node by implementing adaptive frequency hopping (AFH).
- Visualize the status (good or bad) and success rate (recent and cumulative) of each channel.

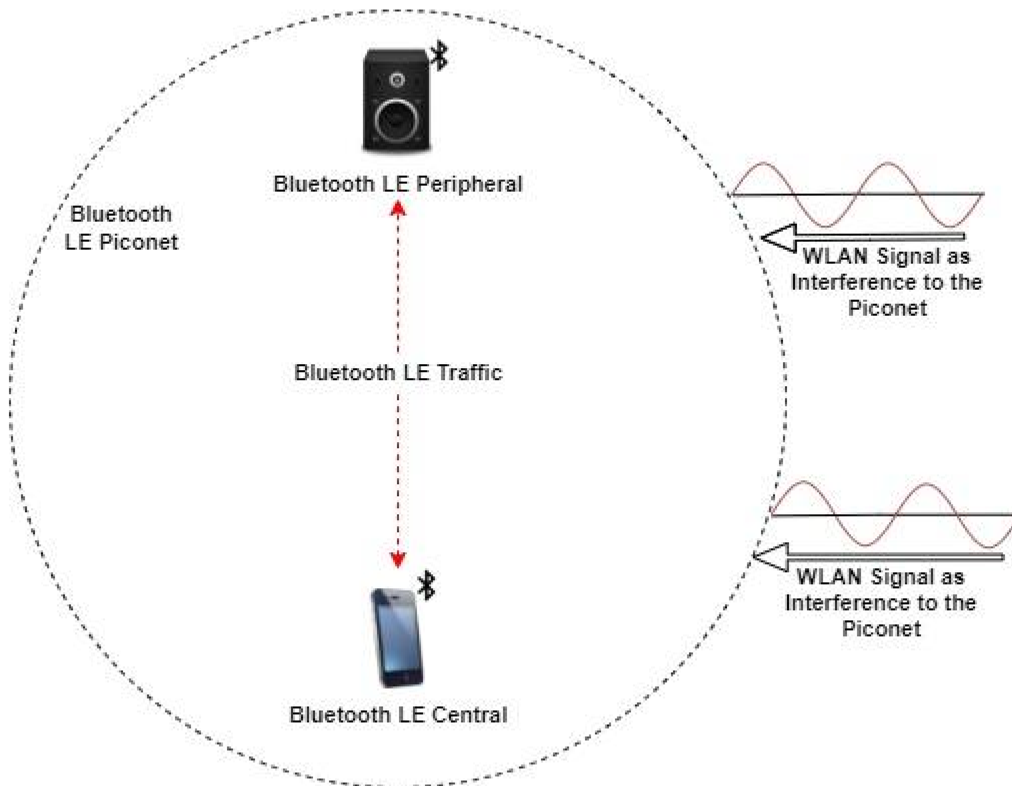
Additionally, you can use this example script to perform these tasks.

- Add WLAN Signal Using WLAN Toolbox Features on page 7-28
- Capture Bluetooth LE Packets to PCAP or PCAPNG file on page 7-28
- Add Custom Channel Classification on page 7-29
- Add Multiple Peripheral Nodes To A Piconet on page 7-29

Noncollaborative Bluetooth LE-WLAN Coexistence Scenario

Interference between Bluetooth and WLAN can be mitigated by two types of coexistence mechanisms: collaborative and noncollaborative. Noncollaborative coexistence mechanisms do not exchange information between two wireless networks. Collaborative coexistence mechanisms collaborate and exchange network-related 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. This example demonstrates a noncollaborative AFH technique deployed between Bluetooth LE and WLAN nodes to mitigate interference. AFH enables a Bluetooth node to adapt to its environment by identifying fixed sources of WLAN interference and excluding them from the list of available channels. For more information about coexistence between Bluetooth LE and WLAN, see “Bluetooth-WLAN Coexistence” (Bluetooth Toolbox).

The Bluetooth LE piconet consists of one Bluetooth LE Central node and one Peripheral node. The scenario consists of two WLAN nodes, which introduce interference in the Bluetooth LE signal. The example simulates this coexistence scenario between Bluetooth LE and WLAN.



Check for Support Package Installation

Check if the 'Communications Toolbox Wireless Network Simulation Library' support package is installed.

```
wirelessnetworkSupportPackageCheck
```

Configure Coexistence Scenario

Set the seed for the random number generator to `1` to ensure repeatability of results. The seed value controls the pattern of random number generation. Initializing the random number generator using the same seed assures the same result. For high-fidelity simulation results, change the seed value and average the results over multiple simulations.

```
rng(1,"twister");
```

Create a wireless network simulator object.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Create a Bluetooth LE node and set the role to `"central"` by using the `bluetoothLENode` (Bluetooth Toolbox) object. Set the properties of the Central node.

```
centralNode = bluetoothLENode("central", ...
    Name="Central Node", ...
    Position=[5 0 0], ...                   % x-, y-, and z-coordinates in meters
    TransmitterPower=0);                    % In dBm
```

Create a Bluetooth LE node and set the role to "peripheral". Set the properties of the Peripheral node.

```
peripheralNode = bluetoothLENode("peripheral", ...
    Name="Peripheral Node", ...
    Position=[10 0 0], ...                          % x-, y-, and z-coordinates in meters
    TransmitterPower=0);                            % In dBm
```

Create a Bluetooth LE configuration object. Set the connection interval, active period, connection offset, and access address for each connection. Connection events are established for every connection interval duration throughout the simulation. The connection offset specifies the offset from the beginning of the connection interval. The active period specifies the active communication period for a connection after which connection event is ended. Assign the configuration to the Central and Peripheral nodes.

```
connectionConfig = bluetoothLEConnectionConfig;
connectionConfig.ConnectionInterval = 0.01;                         % In seconds
connectionConfig.ActivePeriod = 0.01;                               % In seconds
connectionConfig.ConnectionOffset = 0;                              % In seconds
connectionConfig.AccessAddress = "12345678";                        % In hexadecimal
configureConnection(connectionConfig,centralNode,peripheralNode);
```

Configure Application Traffic

Create a networkTrafficOnOff object to generate an On-Off application traffic pattern. Configure the On-Off application traffic pattern at the Central and Peripheral nodes by specifying the application data rate, packet size, and on state duration. Attach application traffic from the Central to the Peripheral nodes.

```
central2PeripheralTrafficSource = networkTrafficOnOff(...
    OnTime=Inf, ...                                                  % In seconds
    DataRate=150, ...                                                % In Kbps
    PacketSize=100, ...                                              % In bytes
    GeneratePacket=true);
addTrafficSource(centralNode,central2PeripheralTrafficSource, ...
    DestinationNode=peripheralNode.Name);
```

Attach application traffic from the Peripheral to the Central nodes.

```
peripheral2CentralTrafficSource = networkTrafficOnOff(...
    OnTime=Inf, ...
    DataRate=150, ...
    PacketSize=100, ...
    GeneratePacket=true);
addTrafficSource(peripheralNode,peripheral2CentralTrafficSource, ...
    DestinationNode=centralNode.Name);
```

Configure WLAN Signal Interference

To add WLAN signal interference, set the enableWLANInterference flag to true.

```
enableWLANInterference = true;
```

Specify the number of WLAN nodes and their positions in the network. The WLAN nodes introduce interference in the network and do not model the PHY and MAC behavior.

Set the properties of the WLAN nodes. Specify the source of WLAN interference by using one of these options.

- "Generated" - To add a WLAN toolbox™ signal to interfere with the communication between Bluetooth LE nodes, select this option and uncomment the WLAN configuration object code.
- "BasebandFile" - To add a WLAN signal from a baseband file with the .bb extension to interfere with the communication between Bluetooth nodes, select this option. You can specify the file name using the BasebandFile input argument. If you do not specify a baseband file, the example adds the default file, WLANHESUBandwidth20 to the WLAN signal.

To determine the path loss of the channel during the transmission, the example uses the distance between the nodes. Create WLAN nodes to introduce interference in the network by using the helperInterferingWLANNode helper object.

```
if enableWLANInterference
    wlanInterferenceSource = BasebandFile ;
    numWLANNodes = 2;
    wlanNodePositions = [0 7 5; 10 8 5];                        % x-, y-, and z-coordinates in r
    wlanCenterFrequency =  [2.412e9; 2.442e9];                  % Center frequency (in Hz) based
    wlanNodes = helperInterferingWLANNode.empty(0,numWLANNodes);
    for wlanIdx=1:numWLANNodes
        wlanNodeObj = helperInterferingWLANNode(...
            WaveformSource=wlanInterferenceSource, ...
            Position=wlanNodePositions(wlanIdx,:), ...
            Name="WLAN node", ...
            TransmitterPower=15, ...                            % In dBm
            CenterFrequency=wlanCenterFrequency(wlanIdx), ...
            Bandwidth = 20e6, ...                               % In Hz
            SignalPeriodicity=0.0029);                          % In seconds
%
% %       To add interfering signal generated using WLAN Toolbox, uncomment this code
%         if strcmpi(wlanInterferenceSource, "Generated")
%             wlanNodeObj.FormatConfig = wlanHTConfig("ChannelBandwidth","CBW20");
%         end
%
        wlanNodes(wlanIdx) = wlanNodeObj;
    end
end
```

Create Bluetooth LE Network

Create a Bluetooth LE network consisting of the Bluetooth LE nodes and any WLAN interfering nodes.

```
bluetoothNodes = [centralNode peripheralNode];
```

Add the Bluetooth and any WLAN nodes to the simulator.

```
addNodes(networkSimulator,bluetoothNodes);
if enableWLANInterference
    addNodes(networkSimulator,wlanNodes);
end
```

Configure Visualization and Channel Classification

Specify the simulation time by using the simulationTime variable. Enable the option to visualize the Bluetooth LE coexistence with WLAN and the channel hopping sequence.

```
simulationTime = 2;             % In seconds
```

To implement channel classification, enable the `enableChannelClassification` variable.

```
enableChannelClassification = true ;
```

Schedule Channel Classification

The Bluetooth LE signal transmitted in a particular channel suffers interference from the WLAN signals. The Bluetooth LE node pseudorandomly selects a new channel from the channel map by using frequency hopping. This example classifies the channels using the AFH algorithm only when you enable channel classification. For each Peripheral node, the Central node periodically classifies the channels as "good" or "bad" based on the total packets received and failed in that channel. If the current number of good channels is less than the preferred number of good channels, the example reclassifies all the bad channels as good channels.

This example implements channel classification by periodic evaluation of the packet failures of each channel. For each Peripheral node, create a channel classifier by using the `helperBluetoothChannelClassification` helper object and schedule the action for each Peripheral node. You can schedule a custom action in the simulation by using the `scheduleAction` object function of the `wirelessNetworkSimulator` object. For example, each time you call the simulator, you can schedule an action to plot the state transitions. Specify the function handle, input argument, absolute simulation time, and periodicity of the callback.

Create a function handle to classify the channel by using the `classifyChannels` object function of the `helperBluetoothChannelClassification` helper object. Schedule the channel classification for the periodicity of the callback by using the `scheduleAction` object function of the `wirelessNetworkSimulator` object. To perform a channel classification for the Peripheral nodes, create and schedule the action for individual destinations.

```
if enableChannelClassification
    classifierObj = helperBluetoothChannelClassification(...
        centralNode,peripheralNode,PERThreshold=50);
    classifyFcn = @(varargin) classifierObj.classifyChannels;
    userData = [];                                                % User data needed
    callAt = 0;                                                   % Absolute simulat
    periodicity = 125e-3;                                         % In seconds

    scheduleAction(networkSimulator,classifyFcn,userData,callAt,periodicity); % Schedule channel
end
```

Visualization

Enable the option to visualize the Bluetooth coexistence with WLAN and the channel hopping sequence.

```
enableVisualization = true ;
```

Initialize coexistence visualization by using the `helperVisualizeCoexistence` helper object. To update the channel map for each channel map update and the status of the channel for each packet reception, listeners are created by using the `addlistener` function of the `helperVisualizeCoexistence` helper object. When the `ChannelMapUpdated` and `PacketReceptionEnded` events are triggered at the Central node object, the listener listens to those events.

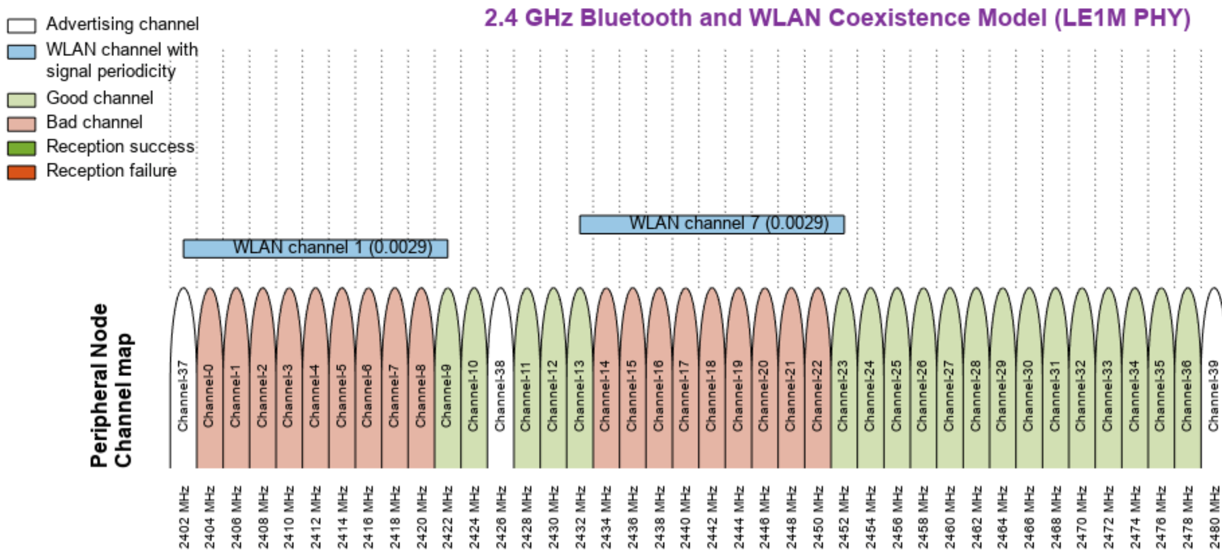
```
if enableVisualization && enableWLANInterference
    coexistenceVisualization = helperVisualizeCoexistence(simulationTime,bluetoothNodes,wlanNodes
elseif enableVisualization && ~enableWLANInterference
    coexistenceVisualization = helperVisualizeCoexistence(simulationTime,bluetoothNodes);
end
```

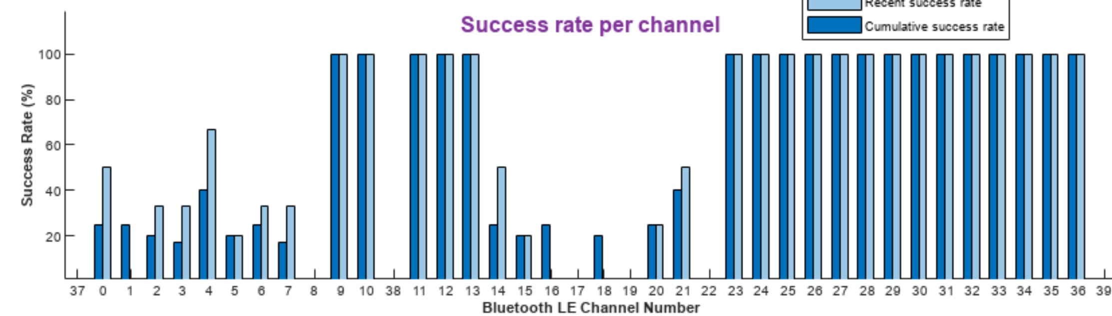
Simulation Results

Run the simulation for the specified time and display the channel hopping sequence in the Bluetooth LE channels and the interference due to the WLAN signals. Visualize the state transitions, status (good or bad), and success rate (recent and cumulative) of each channel. The recent success rate represents the cumulative success rates between each channel classification interval. The overall success rate represents the cumulative success rate throughout the simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channel
```





Retrieve Statistics

The example simulation generates these results.

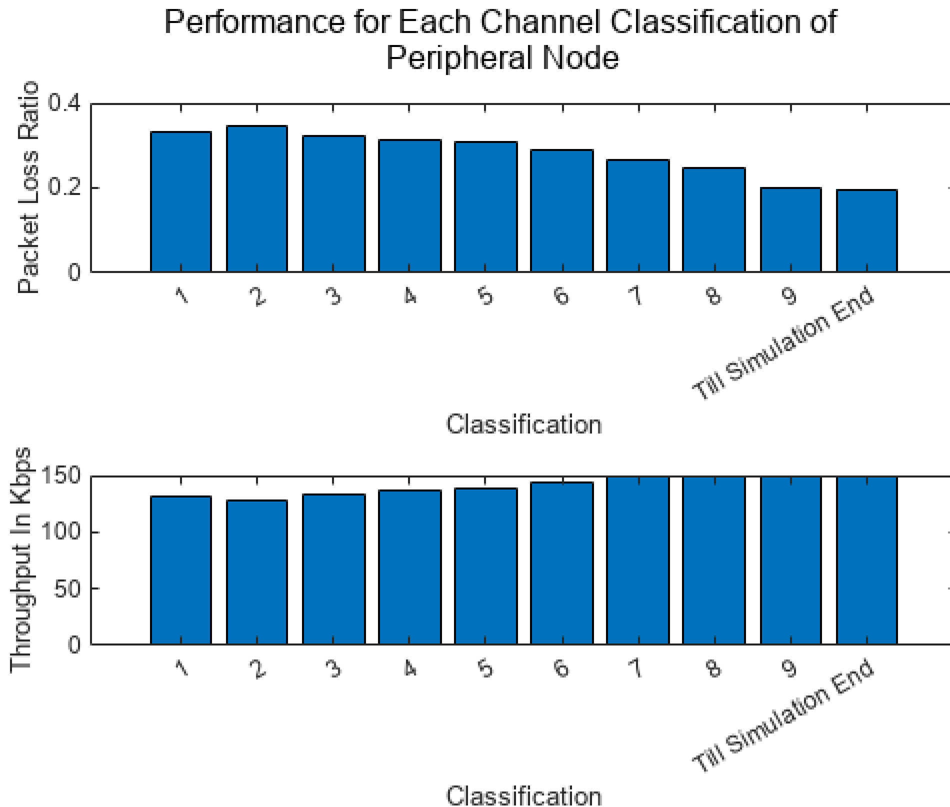
1. A run-time plot for each Central-Peripheral connection pair showing the status (good or bad) and success rate (recent and cumulative) of each channel.

2. Channel classification statistics showing the total number of packets received and corrupted and the status (good or bad) of each channel for each classification interval.
3. A bar plot for each peripheral showing the packet loss ratio and throughput between each channel map update.
4. Application layer (APP), link layer (LL), and PHY statistics for Central and Peripheral nodes.

Retrieve the channel classification statistics by using the `classificationStatistics` object function of the `helperVisualizeCoexistence` helper object. Use this object function to visualize the packet loss ratio and throughput between each channel map update for every Peripheral node.

```
if enableChannelClassification && enableVisualization
    bluetoothLEChannelStats = classificationStatistics(coexistenceVisualization,centralNode,perip
end
```

```
Channel classification statistics of Peripheral Node
                                           Channel 0    Channel 1    Channel 2    Channel 3    Ch
                                           _________    _________    _________    _________    __

    ChannelStatusTillClassification_1          1            1            1            1
    RxPacketsTillClassification_1              0            3            2            2
    RxPacketsFailedTillClassification_1        0            2            2            2
    ChannelStatusTillClassification_2          1            1            1            1
    RxPacketsTillClassification_2              0            4            2            3
    RxPacketsFailedTillClassification_2        0            3            2            3
    ChannelStatusTillClassification_3          1            1            1            1
    RxPacketsTillClassification_3              0            4            5            3
    RxPacketsFailedTillClassification_3        0            3            4            3
    ChannelStatusTillClassification_4          1            0            1            1
    RxPacketsTillClassification_4              2            4            5            3
    RxPacketsFailedTillClassification_4        2            3            4            3
    ChannelStatusTillClassification_5          1            0            0            1
    RxPacketsTillClassification_5              2            4            5            6
    RxPacketsFailedTillClassification_5        2            3            4            5
    ChannelStatusTillClassification_6          1            0            0            0
    RxPacketsTillClassification_6              2            4            5            6
    RxPacketsFailedTillClassification_6        2            3            4            5
    ChannelStatusTillClassification_7          1            0            0            0
    RxPacketsTillClassification_7              4            4            5            6
    RxPacketsFailedTillClassification_7        3            3            4            5
    ChannelStatusTillClassification_8          0            0            0            0
    RxPacketsTillClassification_8              4            4            5            6
    RxPacketsFailedTillClassification_8        3            3            4            5
    ChannelStatusTillClassification_9          0            0            0            0
    RxPacketsTillClassification_9              4            4            5            6
    RxPacketsFailedTillClassification_9        3            3            4            5
    ChannelStatusTillSimulationEnds            0            0            0            0
    RxPacketsTillSimulationEnds                4            4            5            6
    RxPacketsFailedTillSimulationEnds          3            3            4            5
```



Get the Central and Peripheral node statistics by using the `statistics` (Bluetooth Toolbox) object function.

```
centralStats = statistics(centralNode);
peripheralStats = statistics(peripheralNode);
```

The Bluetooth LE Central and Peripheral nodes avoid the interfered channels through channel classification and communicate with each other to avoid packet loss. The success rate is calculated at each Bluetooth LE channel. This example concludes that for high transmit power of a WLAN channel, the success rate of the respective Bluetooth LE channel is low. Therefore, these channels are not used for communication between Bluetooth LE Central and Peripheral nodes.

Further Exploration

Add WLAN Signal Using WLAN Toolbox Features

To add a WLAN signal using WLAN Toolbox features, set the value of `WaveformSource` parameter of the `wlanNodeObj` or a `helperInterferingWLANNode` object to `"Generated"`. Uncomment the code lines in the WLAN Signal Interference on page 7-23 section. You can modify the WLAN packet format configuration object in the `FormatConfig` property of `wlanNodeObj` and assign it to the WLAN node. Set the bandwidth of the signal based on the assigned configuration object.

Capture Bluetooth LE Packets to PCAP or PCAPNG file

To capture the packets at any Bluetooth nodes in a packet capture (PCAP) or packet capture next generation (PCAPNG) file, add the following code before you run the simulation. The packets are

captured into individual PCAP/PCAPNG files for each of the Bluetooth LE node. The name of the PCAP file for a node is in the format: <NodeName_NodeID_yyyyMMdd_HHmmss>.<file_extension>. Node ID is a unique number generated internally for each node. Node ID is a read-only property for all the nodes. Use the `helperPacketCapture` object and specify the Bluetooth LE nodes to capture. Specify the file extension as "pcap" or "pcapng".

```
% clear packetCaptureObj; % Clear the packet capture handle if exists
% packetCaptureObj = helperPacketCapture(bluetoothNodes,"pcap");
```

Add Custom Channel Classification

To add a custom channel classification algorithm, perform these steps:

1. Create a custom channel classification object.
2. Classify the channels by passing the classification function at an absolute simulation time or at a particular periodicity by using the `scheduleAction` object function.
3. Instead of scheduling or calling the classification at certain simulation time instances, you can implement a custom channel classification algorithm by classifying the channels based on the status of the received packets.
4. Update the status of the received packets by using the `updateRxStatus` object function.
5. Classify the channels based on the status of the received packets by using the `classifyChannels` object function.

Add Multiple Peripheral Nodes To A Piconet

To add multiple Peripheral nodes to a piconet, perform these steps:

1. Create Peripheral nodes by using the `bluetoothLENode` (Bluetooth Toolbox) object, setting the `Role` property to `"peripheral"`.
2. Create a connection configuration by using the `bluetoothLEConnectionConfig` (Bluetooth Toolbox) object.
3. Assign the configuration to the Central node and each Peripheral node.
4. Generate and add application traffic at the Central and Peripheral nodes.
5. Create a Bluetooth BR network with all the nodes.
6. Enable channel classification at the Central node for each of the Peripheral nodes by creating an array of classifier objects.
7. Schedule the action for each of the Peripheral nodes. Retrieve the statistics for all Peripheral nodes.

Appendix

The example uses these helper functions:

- helperInterferingWLANNode - Configure and simulate interfering WLAN node
- helperVisualizeCoexistence - Visualize the coexistence model
- helperBluetoothChannelClassification - Create an object to classify the Bluetooth LE channels
- helperPacketCapture - Create an object to capture Bluetooth LE packets as PCAP/PCAPNG file

References

1 Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed September 20, 2022. https://www.bluetooth.com/

2 Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification". Version 5.3. https://www.bluetooth.com/

3 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*

4 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*

5 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

Related Examples

- “PHY Simulation of Bluetooth BR/EDR, LE, and WLAN Coexistence” on page 1-2

Get Started with WLAN System-Level Simulation in MATLAB

This example shows how to model a WLAN network consisting of an IEEE® 802.11ax™ [1 on page 7-38] access point (AP) and a station (STA) by using WLAN Toolbox™ and the Communications Toolbox™ Wireless Network Simulation Library.

Using this example, you can:

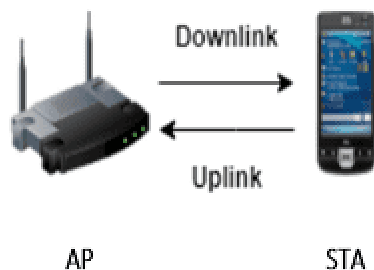
- Simulate a multinode WLAN system by configuring the application layer (APP), medium access control (MAC), and physical layer (PHY) parameters at each node.
- Model uplink and downlink communication between an AP and a STA.
- Switch between the abstracted and full models of MAC and PHY layers.
- Visualize the time spent by each node in Idle, Contention, Transmission, and Reception state.
- Capture the APP, MAC, and PHY statistics for each node.

The simulation results show performance metrics such as throughput, latency, and packet loss.

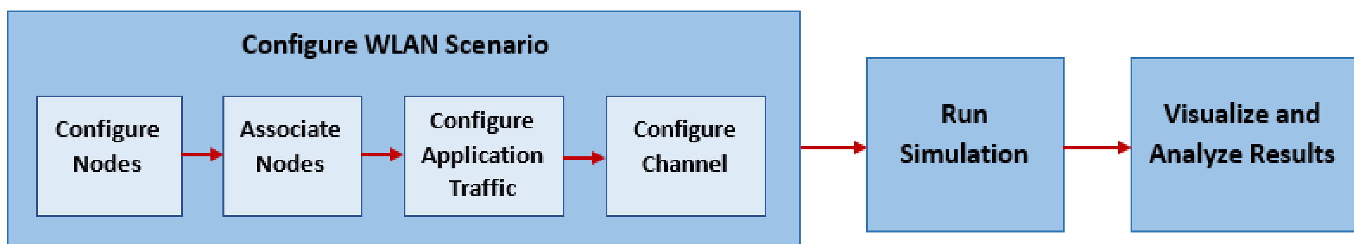
Additionally, you can use this example script to generate, configure and add external application traffic patterns such as On-Off, Video, Voice, and FTP to the WLAN nodes. For more information, see Further Exploration on page 7-37.

WLAN System-Level Simulation

This example shows you how to model a WLAN network with uplink and downlink communication between an AP and a STA. This figure illustrates the example network.



This figure shows the example workflow.



To confirm compliance with the IEEE 802.11 standard [2 on page 7-38], the features in this example are validated with Box-3 and Box-5 scenarios specified in the TGax evaluation methodology [3 on

page 7-38]. The network throughputs that are calculated for TGax simulation scenarios [4 on page 7-38] are validated against the published calibration results from the TGax Task Group.

Check for Support Package Installation

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck
```

Configure Simulation Parameters

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation, including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive")
```

Specify the simulation time in seconds. To visualize a live state transition plot for all of the nodes, set the `showLiveStateTransitionPlot` variable to `true`.

```
simulationTime = 1;
showLiveStateTransitionPlot = true;
```

At the transmitter and receiver, modeling full MAC processing involves complete MAC frame generation at the MAC layer. Similarly, modeling full PHY processing involves complete operations related to waveform transmission and reception through a fading channel. When simulating large networks, full MAC and PHY processing is computationally expensive.

In the abstracted MAC, the node does not generate or decode any frames at the MAC layer. Similarly, in the abstracted PHY, the node does not generate or decode any waveforms at the PHY. MAC and PHY abstraction enable you to minimize the complexity and runtime of system-level simulations. For more information on PHY abstraction, see the “Physical Layer Abstraction for System-Level Simulation” on page 7-90 example.

This table shows you how to switch between the abstracted and full MAC or PHY by configuring the values of the `MACFrameAbstraction` and `PHYAbstractionMethod` properties of `wlanNode` object.

Type of MAC and PHY	Value of MACFrameAbstraction	Value of PHYAbstractionMethod
Abstracted MAC and abstracted PHY (default combination)	true	"tgax-evaluation-methodology" or "tgax-mac-calibration"
Full MAC and abstracted PHY	false	"tgax-evaluation-methodology" or "tgax-mac-calibration"
Full MAC and full PHY	false	"none"
Abstracted MAC and full PHY	This is an invalid combination because full PHY requires a MAC frame (to generate waveform), which the abstracted MAC does not generate.	

If you set the `PHYAbstractionMethod` to `tgax-evaluation-methodology`, the PHY estimates the performance of a link with the TGax channel model by using an effective signal-to-interference-plus-noise ratio (SINR) mapping. Alternatively, if you set the `PHYAbstractionMethod` to `tgax-mac-calibration` the PHY assumes a packet failure on interference without actually calculating the link performance. To use the full PHY, set the value of `PHYAbstractionMethod` to `none`.

```
MACFrameAbstraction = false;
PHYAbstractionMethod = tgax-evaluation-met...;
```

Configure WLAN Scenario

Initialize the wireless network simulator by using the `wirelessNetworkSimulator` object.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Nodes

The `wlanDeviceConfig` object enables you to set the configuration parameters for the AP and STA. Create two WLAN device configuration objects, one for the AP and the other for the STA. Specify the operating mode, modulation and coding scheme, and transmission power (in dBm) for the AP and STA.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",MCS=2,TransmitPower=15); % AP device configuration
stationCfg = wlanDeviceConfig(Mode="STA",MCS=2,TransmitPower=15);    % STA device configuration
```

To create an AP node and a STA node from the specified WLAN device configurations, use `wlanNode` objects. Specify the name and position of the AP and the STA. Specify the PHY abstraction method used for the AP and the STA. Configure the `MACFrameAbstraction` property of the `wlanNode` object to indicate if the MAC frames are abstracted by the nodes.

```
accessPoint = wlanNode(Name="AP", ...
    Position=[10 0 0], ...
    DeviceConfig=accessPointCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
```

```
    MACFrameAbstraction=MACFrameAbstraction);

station = wlanNode(Name="STA", ...
    Position=[20 0 0], ...
    DeviceConfig=stationCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=MACFrameAbstraction);
```

Create a WLAN network consisting of the AP and the STA.

```
nodes = [accessPoint station];
```

To ensure all the nodes are configured properly, use the `hCheckWLANNodesConfiguration` helper function.

```
hCheckWLANNodesConfiguration(nodes)
```

Association and Application Traffic

Associate the STA to the AP by using the `associateStations` object function of the wlanNode object. To configure uplink and downlink application traffic between the AP and STA, use the `FullBufferTraffic` argument of the `associateStations` object function.

```
associateStations(accessPoint,station,FullBufferTraffic="on");
```

Wireless Channel

To model a random TGax fading channel between each node, this example uses the `hSLSTGaxMultiFrequencySystemChannel` helper object. Add the channel model to the wireless network simulator by using the `addChannelModel` object function of the `wirelessNetworkSimulator` object.

```
channel = hSLSTGaxMultiFrequencySystemChannel(nodes);
addChannelModel(networkSimulator,channel.ChannelFcn)
```

Export WLAN MAC Frames to PCAP or PCAPNG File

The example demonstrates packet capture at both AP and STA nodes. Set the `capturePacketsFlag` flag to `true` to capture the packets exchanged during the simulation. The packet capture (PCAP) or packet capture next generation (PCAPNG) file (.pcap or .pcapng, respectively) is a widely used packet capture file format to perform packet analysis. The packets are captured into a PCAP file in this example.

```
capturePacketsFlag = true ;
```

Create a `hExportWLANPackets` helper object to generate PCAP files. Specify the node objects at which you want to capture the packets. The helper object captures transmitted and received packets at each of these nodes and generates a PCAP file for each node. If you want to capture the packets in a PCAPNG file, specify the string "pcapng" as the second argument to the object function call. Note that capturing packets is possible only when `MACFrameAbstraction` flag is set to `false`.

```
if capturePacketsFlag
    clear capturePacketsObj;                          % Clear existing helper object
    capturePacketsObj = hExportWLANPackets(nodes);
end
```

You can visualize and analyze the PCAP or PCAPNG file by using a third-party packet analyzer tool such as Wireshark.

Simulation

Add your nodes to the wireless network simulator.

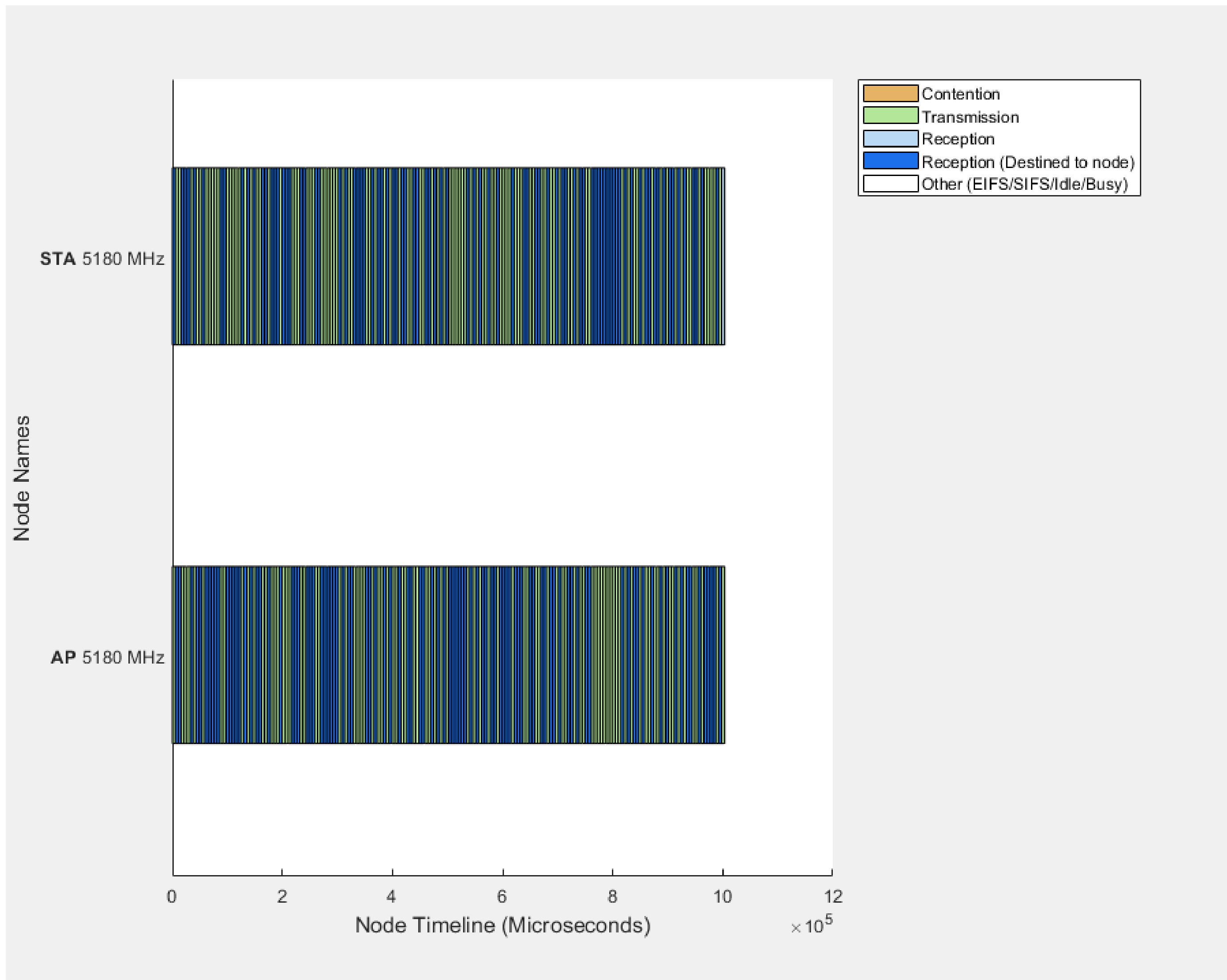
```
addNodes(networkSimulator,nodes)
```

To view the state transition and performance metrics plots, create a `hSimulationPlotViewer` helper object. The simulation shows the plots for all the nodes configured in the simulation. To visualize the state transitions and performance metrics of a specific node, specify the corresponding node object as the second argument to the helper object.

```
viewerObj = hSimulationPlotViewer(showLiveStateTransitionPlot,nodes);
```

Run the network simulation for the specified simulation time. The runtime visualization shows the time spent by the AP and the STA in Idle, Contention, Transmission, and Reception state.

```
run(networkSimulator,simulationTime);
```



Results

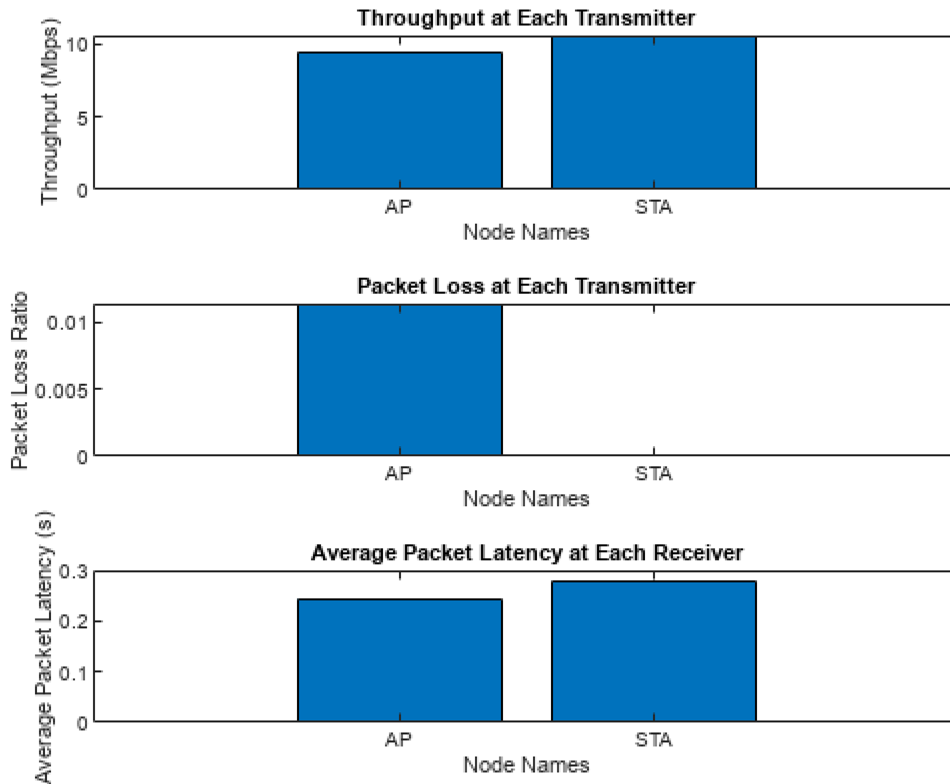
Retrieve the APP, MAC, and PHY statistics at each node by using the `statistics` object function of the `wlanNode` object.

```
stats = statistics(nodes);
```

Plot the performance of each node by using the `plotNetworkStats` object function of the `hSimulationPlotViewer` object. You can visualize these simulation plots.

- Throughput (in Mbps) at each transmitter (AP and STA).
- Packet loss ratio (ratio of unsuccessful data transmissions to the total data transmissions) at each transmitter (AP and STA).
- Average packet latency incurred at each receiver (AP and STA). The average packet latency shows the average latency that the STA incurs to receive the downlink traffic from the AP and the average latency that the AP incurs to receive uplink traffic from the STA.

```
plotNetworkStats(viewerObj,simulationTime)
```



Because the `hExportWLANPackets` helper object does not overwrite the existing PCAP or PCAPNG file, deleting the PCAP objects used in this simulation.

```
if capturePacketsFlag
    delete(capturePacketsObj.PCAPObjList);
end
```

Further Exploration

Configure External Application Traffic

To generate and add external application traffic pattern (On-Off, Video, Voice, and FTP) to the WLAN nodes, modify the traffic configuration in this example. To generate an On-Off application traffic, create two `networkTrafficOnOff` objects for downlink and uplink traffic, respectively. Configure these objects by specifying the application data rate and packet size.

```
% trafficSourceDL = networkTrafficOnOff(DataRate=100000,PacketSize=1500);
% trafficSourceUL = networkTrafficOnOff(DataRate=100000,PacketSize=1500);
```

Attach downlink application traffic from the AP node to the STA node by using the `addTrafficSource` object function of the `wlanNode` object. Similarly, attach uplink application traffic from the STA node to the AP node.

```
% addTrafficSource(accessPoint,trafficSourceDL,DestinationNode=station,AccessCategory=0);
% addTrafficSource(station,trafficSourceUL,DestinationNode=accessPoint,AccessCategory=0);
```

To generate video, voice, and FTP application traffic, use the `networkTrafficVideoConference`, `networkTrafficVoIP`, and `networkTrafficFTP` objects, respectively.

Appendix

This example uses these helpers:

- `hCheckWLANNodesConfiguration` — Check if the node parameters are configured correctly
- `hExportWLANPackets` — Capture MAC frames and write them into a PCAP/PCAPNG file
- `hSLSTGaxMultiFrequencySystemChannel` — Return a system channel object
- `hSLSTGaxAbstractSystemChannel` — Return a channel object for abstracted PHY layer
- `hSLSTGaxSystemChannel` — Return a channel object for full PHY layer
- `hSLSTGaxSystemChannelBase` — Return the base channel object
- `hSimulationPlotViewer` — Plot the state transition and performance metrics figures

References

1. Institute of Electrical and Electronics Engineers (IEEE). 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 1: Enhancements for High-Efficiency WLAN. IEEE 802.11ax-2021. IEEE, May 19, 2021. https://doi.org/10.1109/IEEESTD.2021.9442429.
2. Institute of Electrical and Electronics Engineers (IEEE). 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. IEEE 802.11-2020. IEEE, February 26, 2021. https://doi.org/10.1109/IEEESTD.2021.9363693.
3. Institute of Electrical and Electronics Engineers (IEEE). 11ax Evaluation Methodology. IEEE 802.11-14/0571r12. IEEE, January 2016.
4. Institute of Electrical and Electronics Engineers (IEEE). TGax Simulation Scenarios. IEEE 802.11-14/0980r16. IEEE, 2015.

See Also

Functions
`statistics`

Objects
`wirelessNetworkSimulator` | `networkTrafficOnOff` | `wlanNode` | `wlanDeviceConfig`

See Also

More About

- “Create, Configure, and Simulate an 802.11ax Mesh Network” on page 7-121
- “Simulate an 802.11ax Hybrid Mesh Network” on page 7-127
- “Simulate a Multiband 802.11ax Network” on page 7-124
- “802.11ax Downlink OFDMA Multinode System-Level Simulation” on page 7-12

- "WLAN System-Level Simulation Statistics" on page 7-114

802.11ax Multinode System-Level Simulation of Residential Scenario

This example shows how to model performance of an IEEE® 802.11ax™ [1 on page 7-48] network in a residential scenario by using WLAN Toolbox™ and the Communications Toolbox™ Wireless Network Simulation Library.

Using this example, you can:

- Simulate a residential scenario by configuring the network and channel parameters.
- Simulate a multinode WLAN system and capture the network-related statistics.
- Simulate the network by switching between the abstracted and full models of medium access control (MAC) and physical layer (PHY).

The simulation results show performance metrics such as throughput, latency, and packet loss.

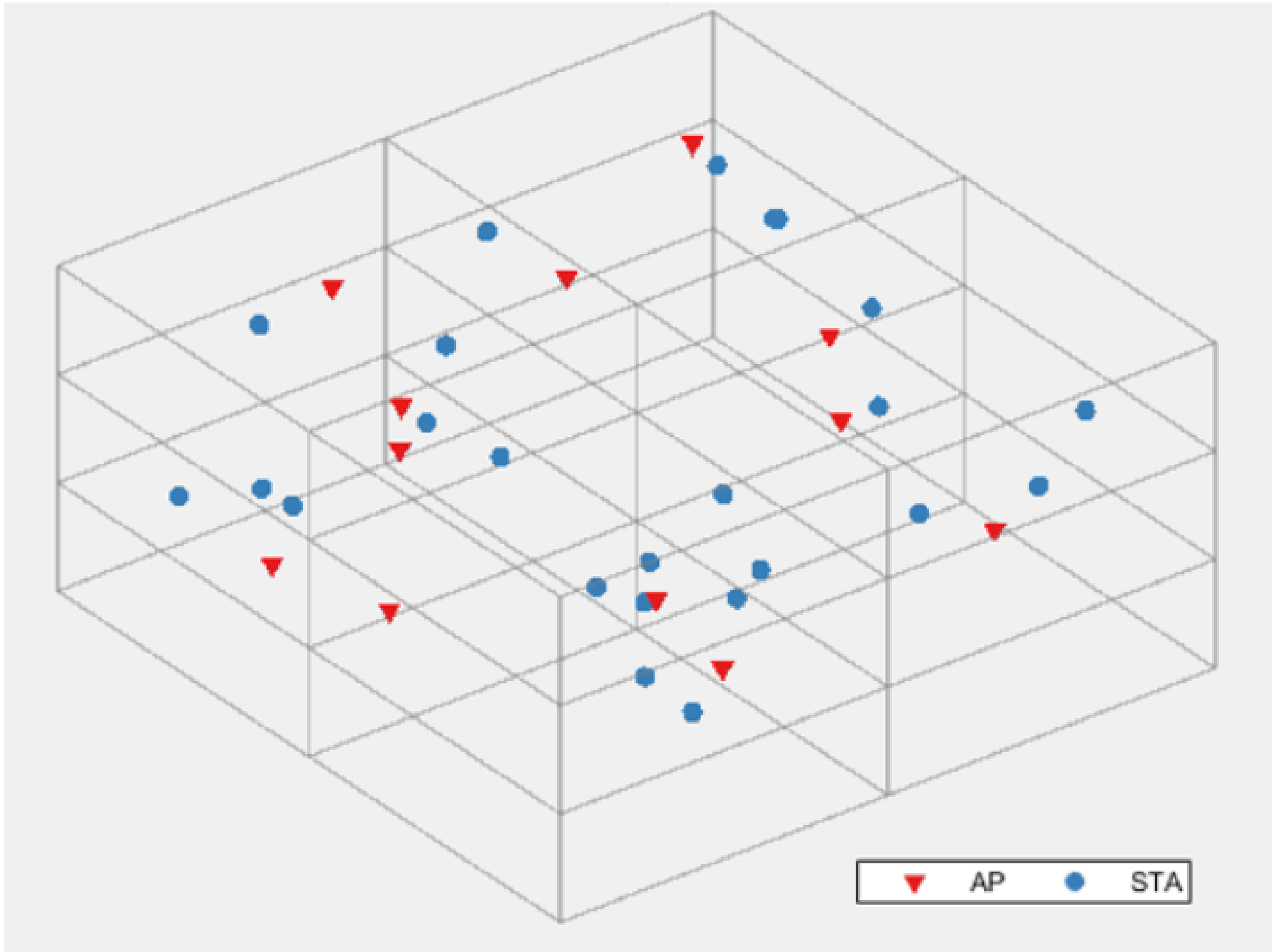
Additionally, you can use this example script to export WLAN MAC frames to a PCAP or PCAPNG file. For more information, see Further Exploration on page 7-47.

Residential Scenario

This example demonstrates a system-level simulation to evaluate the performance of an 802.11ax network in a residential scenario. The residential scenario consists of a building with three floors. These are the characteristics of the residential scenario:

- The spacing between the floors is three meters.
- Each floor consists of four rooms, and the dimensions of each room are 10-by-10-by-3 meters.
- Each room has an access point (AP) and two stations (STAs) placed at random *x*- and *y*-locations at a height of 1.5 meters from the floor.
- Each AP transmits data to the associated STAs in the same room.

The simulation scenario specifies a path loss model based on the distance between the nodes, and the number of walls and floors traversed by the WLAN signal. This figure shows the residential scenario simulated in this example.



To confirm compliance with the IEEE 802.11 standard [2 on page 7-48], the features in this example are validated with Box-3 and Box-5 scenarios specified in the TGax evaluation methodology [3 on page 7-48]. The network throughputs that are calculated for TGax simulation scenarios [4 on page 7-48] are validated against the published calibration results from the TGax Task Group.

Check for Support Package Installation

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck
```

Configuration Parameters

Simulation Parameters

Set the seed for the random number generator to 1. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation, including backoff counter selection at the MAC layer and predicting packet

reception success at the PHY layer. To improve the accuracy of your simulation results after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive")
```

Specify the simulation time in seconds. To visualize a live state transition plot for all of the nodes, set the `showLiveStateTransitionPlot` variable to `true`.

```
simulationTime = 0.12;
showLiveStateTransitionPlot = true;
```

This example uses the abstract models of MAC and PHY layers at all the nodes (APs and STAs) by default. To use the full MAC layer model, set the `MACFrameAbstraction` variable to `false`. Similarly, to use the full PHY layer model, set the `PHYAbstractionMethod` variable to `none`. For more information about MAC and PHY, see the “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31 example.

```
MACFrameAbstraction = true;
PHYAbstractionMethod = tgax-evaluation-met...;
```

Residential Scenario Parameters

The `ScenarioParameters` structure defines the size and layout of the residential building by using these parameters.

- `BuildingLayout` — Specify the number of rooms along the length, breadth, and height of the building.
- `RoomSize` — Specify the size of each room in meters.
- `NumRxPerRoom` — Specify the number of stations per room.

The example assumes each room contains one transmitting AP and two receiving STAs.

```
ScenarioParameters = struct;
ScenarioParameters.BuildingLayout = [2 2 3];
ScenarioParameters.RoomSize = [10 10 3];
ScenarioParameters.NumRxPerRoom = 2;
```

Configure WLAN Scenario

Initialize the wireless network simulator by using the `wirelessNetworkSimulator` object.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Nodes

The example creates a residential scenario consisting of 36 nodes. Node 1 to Node 12 are APs, and Node 13 to Node 36 are STAs. The `hGetIDsAndPositions` helper function returns node IDs and random positions for the AP and STAs within each room. The function returns an array, `nodeIDs`, where each row stores the IDs of the AP and its associated STAs in the same room. The `apPositions` and `staPositions` outputs contain the *x*-, *y*-, and z-Cartesian coordinates of the APs and STAs, respectively. Units are in meters.

```
[nodeIDs,apPositions,staPositions] = hGetIDsAndPositions(ScenarioParameters);
```

The `wlanDeviceConfig` object enables you to set the configuration parameters for the APs and STAs. Create two `wlanDeviceConfig` objects to initialize the configuration parameters for the APs and the STAs. Specify the operating mode, modulation and coding scheme, and the transmission power (in dBm) for the APs and STAs.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",MCS=2,TransmitPower=15);     % AP device configuratior
stationCfg = wlanDeviceConfig(Mode="STA",MCS=2,TransmitPower=15);        % STA device configuratic
```

Create two arrays of `wlanNode` objects, corresponding to the AP nodes and STA nodes, by specifying their `Position` properties as `apPosition` and `staPosition`, respectively. Each array contains a number of objects equal to the number of positions specified by the corresponding value. Specify the `Name`, `DeviceConfig`, `PHYAbstractionMethod`, and `MACFrameAbstraction` properties of `wlanNode` objects.

```
accessPoints = wlanNode(Position=apPositions, ...
    Name="AP"+(1:size(apPositions,1)), ...
    DeviceConfig=accessPointCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=MACFrameAbstraction);
stations = wlanNode(Position=staPositions, ...
    Name="STA"+(1:size(staPositions,1)), ...
    DeviceConfig=stationCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=MACFrameAbstraction);
```

Create a WLAN network consisting of APs and STAs.

```
nodes = [accessPoints stations];
```

To ensure all the nodes are configured properly, use the `hCheckWLANNodesConfiguration` helper function.

```
hCheckWLANNodesConfiguration(nodes)
```

Association and Application Traffic

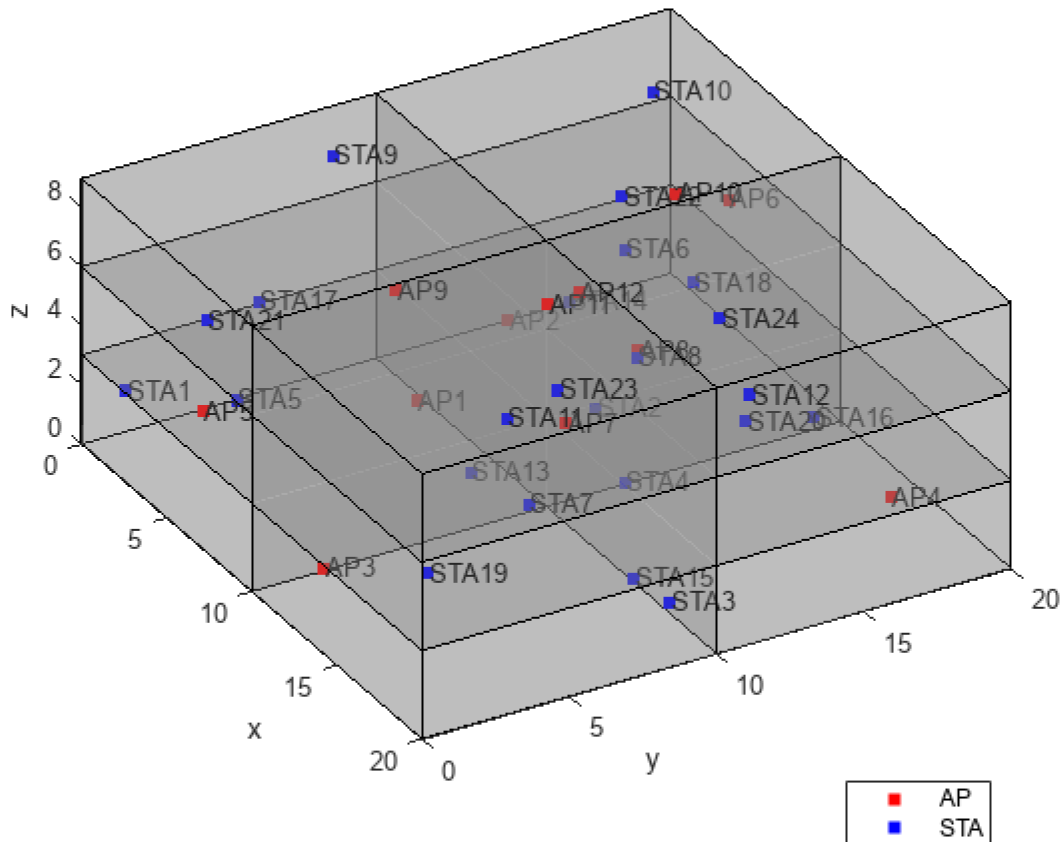
Associate the STAs in each room to the corresponding AP by using the `associateStations` object function of the `wlanNode` object. Configure the APs with continuous application traffic to their associated STAs by using the `FullBufferTraffic` argument.

```
numAPs = prod(ScenarioParameters.BuildingLayout);                                        % One /
for apID = 1:numAPs
    associateStations(nodes(apID),[nodes(nodeIDs(apID,2:end))],FullBufferTraffic="DL");
end
```

Create Network

Create transmitter and receiver sites from the node configurations by using the `hCreateSitesFromNodes` helper function. Create the building geometry from the residential scenario parameters by using the `hTGaxResidentialTriangulation` helper object. Visualize the residential building along with the transmitter and receiver sites by using the `hVisualizeScenario` helper function.

```
[txSites,rxSites] = hCreateSitesFromNodes(nodes);
triangulationObj = hTGaxResidentialTriangulation(ScenarioParameters);
hVisualizeScenario(triangulationObj,txSites,rxSites,apPositions)
```



Wireless Channel

This example uses the TGax residential propagation model to determine path loss between nodes. Path loss is a function of the number of walls, number of floors, and distance between nodes. Create a path loss model by using the `hTGaxResidentialPathLoss` helper function. Obtain the path loss function handle by using the `hCreatePathlossTable` helper function.

```
propModel = hTGaxResidentialPathLoss(Triangulation=triangulationObj,ShadowSigma=0,FacesPerWall=1
[~,pathLossFcn] = hCreatePathlossTable(txSites,rxSites,propModel);
```

To add a channel object to the wireless network simulator, create a `hSLSTGaxMultiFrequencySystemChannel` helper object by using the path loss function handle. Add the residential path loss model to the network simulator by using the `addChannelModel` object function of the `wirelessNetworkSimulator` object.

```
channel = hSLSTGaxMultiFrequencySystemChannel(nodes,PathLossModel="custom",PathLossModelFcn=pathl
addChannelModel(networkSimulator,channel.ChannelFcn)
```

Simulation

Add your nodes to the wireless network simulator.

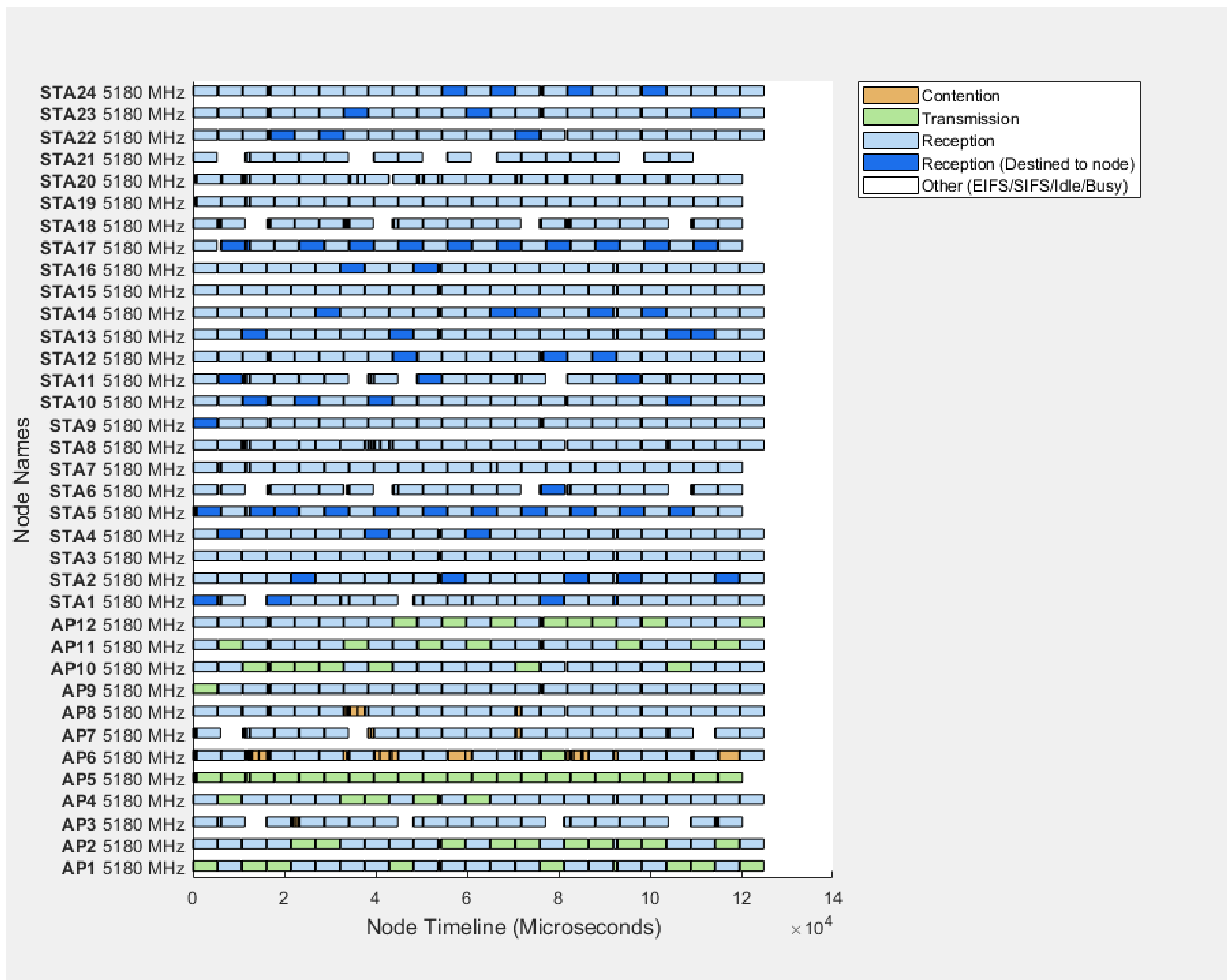
```
addNodes(networkSimulator,nodes)
```

To view the state transition and performance metrics plots, create a `hSimulationPlotViewer` helper object. By default, this helper object shows the plots for all the nodes configured in the simulation. To visualize the state transitions and performance metrics of specific nodes, specify the corresponding node objects as the second argument to the helper object.

```
viewerObj = hSimulationPlotViewer(showLiveStateTransitionPlot,nodes);
```

Run the network simulation for the specified simulation time. The runtime visualization shows the time spent by the AP and the STA in Idle, Contention, Transmission, and Reception state.

```
run(networkSimulator,simulationTime);
```



Results

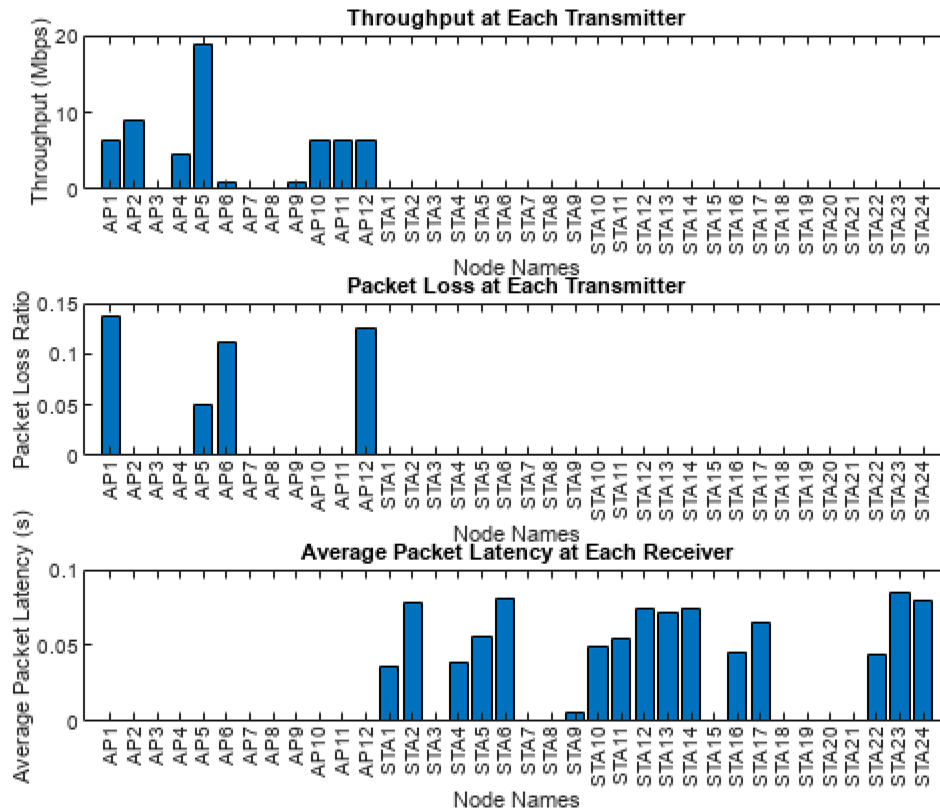
Retrieve the APP, MAC, and PHY statistics at each node by using the `statistics` object function of the `wlanNode` object.

```
stats = statistics(nodes);
```

Plot the performance of each node by using the `plotNetworkStats` function of the `hSimulationPlotViewer` object. You can visualize these simulation plots.

- Throughput (in Mbps) at each transmitter (AP).
- Packet loss ratio (ratio of unsuccessful data transmissions to the total data transmissions) at each transmitter (APs).
- Average packet latency incurred at each receiver (STA). The average packet latency shows the average latency that each STA incurs to receive the downlink traffic from the AP.

```
plotNetworkStats(viewerObj,simulationTime)
```



Further Exploration

Export WLAN MAC Frames to PCAP or PCAPNG File

You can capture packets at both AP and STA nodes by using the `hExportWLANPackets` helper object. Specify the node objects at which you want to capture the packets. To capture packets at multiple nodes, specify the corresponding node objects as an array. You can export the packets captured at the AP and STA nodes to a PCAP or PCAPNG file. For more information about capturing packets, and exporting captured packets to a PCAP or PCAPNG file, see the “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31 example.

Appendix

The example uses these helpers:

- `hGetIDsAndPositions` — Return node IDs and random positions for the APs and STAs
- `hCheckWLANNodesConfiguration` — Check if the node parameters are configured correctly
- `hCreateSitesFromNodes` — Return transmitter and receiver sites
- `hTGaxResidentialTriangulation` — Create the residential scenario geometry
- `hCreatePathlossTable` — Return the path loss function handle in the residential scenario
- `hVisualizeScenario` — Display the residential building along with transmitters and receivers
- `hTGaxIndoorLinkInfo` — Return the number of floors, the number of walls, and the distance between points for a link

- `hExportWLANPackets` — Capture MAC frames and write them into a PCAP or PCAPNG file
- `hTGaxResidentialPathLoss` — Configure and create a residential path loss model
- `hSLSTGaxMultiFrequencySystemChannel` — Return a system channel object and set the path loss model
- `hSLSTGaxAbstractSystemChannel` — Return a channel object for an abstracted PHY layer
- `hSLSTGaxSystemChannel` — Return a channel object for a full PHY layer
- `hSLSTGaxSystemChannelBase` — Return the base channel object
- `hSimulationPlotViewer` — Plot the state transition and performance metric figures

References

1. Institute of Electrical and Electronics Engineers (IEEE). 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 1: Enhancements for High-Efficiency WLAN. IEEE 802.11ax-2021. IEEE, May 19, 2021. https://doi.org/10.1109/IEEESTD.2021.9442429.
2. Institute of Electrical and Electronics Engineers (IEEE). 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. IEEE 802.11-2020. IEEE, February 26, 2021. https://doi.org/10.1109/IEEESTD.2021.9363693.
3. Institute of Electrical and Electronics Engineers (IEEE). 11ax Evaluation Methodology. IEEE 802.11-14/0571r12. IEEE, January 2016.
4. Institute of Electrical and Electronics Engineers (IEEE). TGax Simulation Scenarios. IEEE 802.11-14/0980r16. IEEE, 2015.

See Also

Functions
`statistics`

Objects
`wirelessNetworkSimulator` | `wlanNode` | `wlanDeviceConfig`

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “802.11ax Downlink OFDMA Multinode System-Level Simulation” on page 7-12
- “Spatial Reuse with BSS Coloring in 802.11ax Network Simulation” on page 7-2
- “WLAN System-Level Simulation Statistics” on page 7-114

802.11 MAC and Application Throughput Measurement

This example shows how to measure the MAC and application layer throughput in a multi-node 802.11a/n/ac/ax network using SimEvents®, Stateflow®, and WLAN Toolbox™. The system-level model presented in this example includes functionalities such as configuring the priority of the traffic at the application layer, capability to generate and decode waveforms of Non-HT, HT-MF, VHT, HE-SU and HE-EXT-SU formats, MPDU aggregation and enabling block acknowledgment of MPDUs. The application layer throughput calculated using this model is validated against published calibration results from the TGax Task Group [4] for Box 3 scenarios (Tests 1a, 1b, and 2a) specified in TGax evaluation methodology [3]. The obtained application layer throughput is within the range of minimum and maximum throughput specified in published calibration results [4].

Throughput in 802.11 Networks

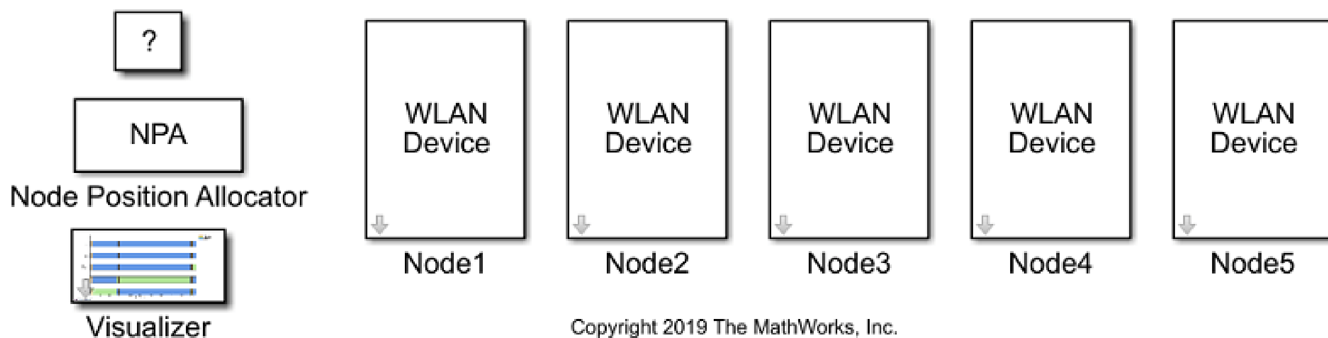
The IEEE® 802.11™ working group is continually adding features to 802.11 specification [1] to improve the throughput and reliability in WLAN networks. Throughput is the amount of data transmitted over a period of time. Medium Access Control (MAC) layer throughput refers to the amount of data successfully transmitted by the MAC layer over a period of time. MAC protocol data unit (MPDU) is the unit of transmission at MAC layer. In 802.11n, MPDU aggregation was introduced to increase the throughput. When MPDU aggregation is supported, MAC layer aggregates multiple MPDUs into an aggregated MPDU (A-MPDU) for transmission. This reduces the overhead of channel contention for transmitting multiple frames, resulting in enhanced throughput. In 802.11ac [1] and 802.11ax [2], the maximum limits for an A-MPDU length were increased resulting in even better throughput in WLAN networks.

Model 802.11 Network

This example models a WLAN network with five nodes as shown in this figure. These nodes implement carrier-sense multiple access with collision avoidance (CSMA/CA) with physical carrier sense and virtual carrier sense. The physical carrier sensing uses the clear channel assessment (CCA) mechanism to determine whether the medium is busy before transmitting. Whereas, the virtual carrier sensing uses the RTS/CTS handshake to prevent the hidden node problem.

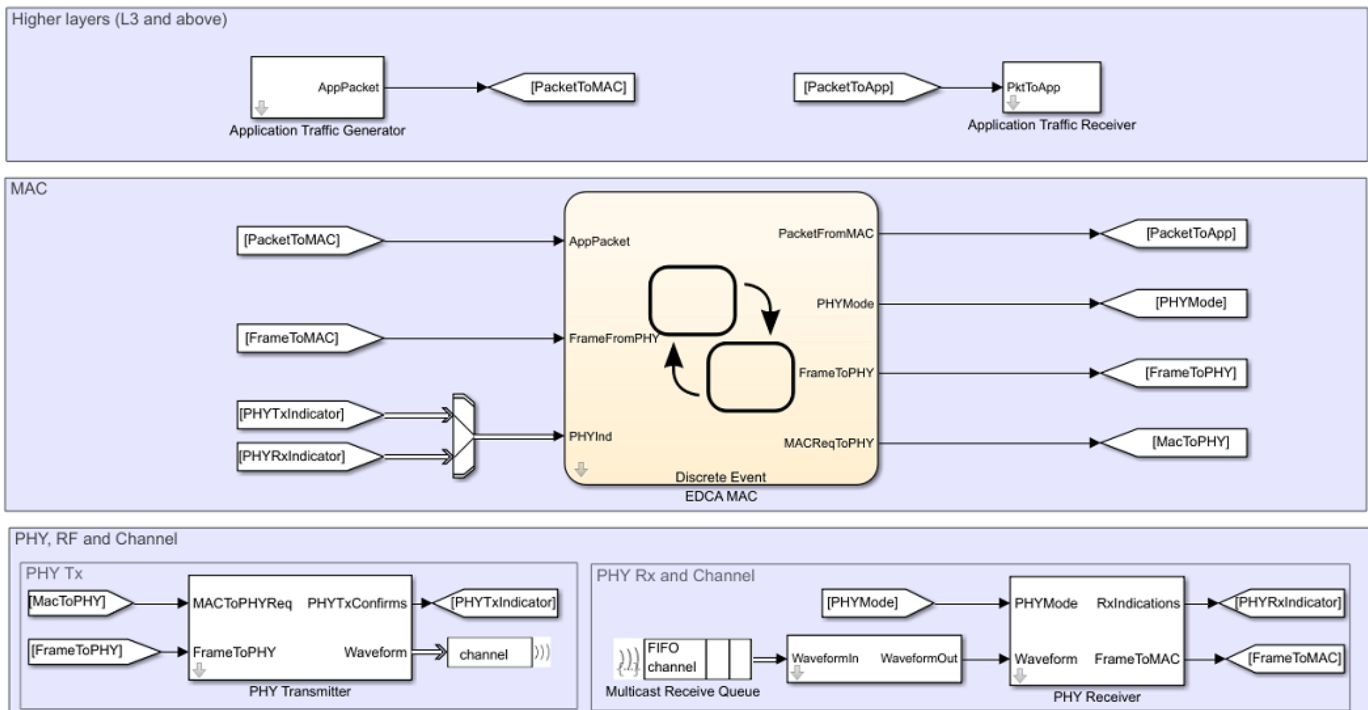
The model in the example displays various statistics such as the number of transmitted, received, and dropped packets at PHY and MAC layers. Moreover, the runtime figures that help in analyzing/estimating the node-level and network-level performance are also displayed in this model. This model is validated against the published calibration results from the TGax Task Group [4] for Box 3 scenarios (Tests 1a, 1b, and 2a) specified in TGax evaluation methodology [3].

WLAN Network



Components of a WLAN Node

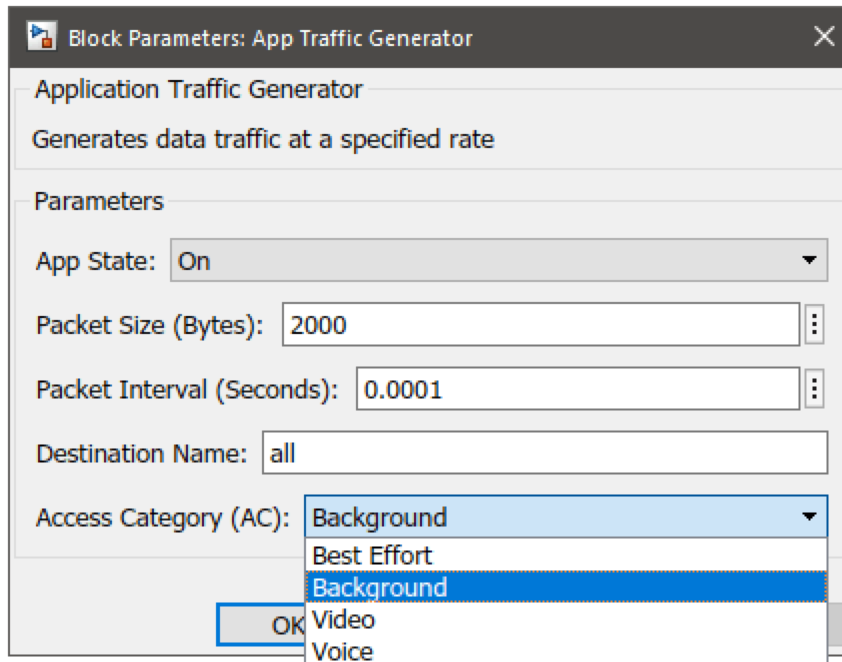
The components of a WLAN node are shown in this figure. The information is retrieved by pressing the arrow button for each node in the above figure.



Application, EDCA MAC, PHY, and Channel Block Capabilities

Application:

The application layer has the capability to generate data with different priority levels as shown in this figure. These priority levels are configured using `Access Category` property in the mask parameters of the Application Traffic Generator block inside a WLAN node. You can also configure the packet size, inter-packet interval, and destination node for the application layer.



EDCA MAC:

The EDCA MAC block used in this example has the following capabilities:

- Generate and decode MAC frames of high efficiency single user (HE-SU), high efficiency extended range single user (HE-EXT-SU), very high throughput (VHT), high throughput mixed format (HT-MF) and Non-HT formats. These formats are configured using the `PHY Tx Format` property in the mask parameters of the MAC EDCA block inside a WLAN node as shown in this figure.
- Aggregate MPDUs to form an A-MPDU. This can be configured by setting `PHY Tx Format` to one of `HT-MF`, `VHT`, `HE-SU`, or `HE-EXT-SU`. In case of `HT-MF`, `MPDU Aggregation` property must also be enabled for A-MPDU generation.
- Acknowledge multiple MPDUs in an A-MPDU with a single block acknowledgment (BA) frame. MAC assumes a pre-configured BA session between the transmitter and the receiver of an A-MPDU.
- Enable/disable acknowledgments. This can be configured using the `Ack Policy` property.
- Maintain separate retry limits for shorter frames (less than RTS threshold) and longer frames (greater than or equal to RTS threshold). These limits can be configured using the `Max Short Retries` and `Max Long Retries` properties.
- Transmit multiple streams of data using the multiple-input multiple-output (MIMO) capability. You can configure this capability using the `Number of Transmit Chains` property. This property is applicable only when the value of `PHY Tx Format` property is set to `VHT`, `HE-SU`, or `HE-EXT-SU`. The MIMO capability can also be used for `HT` format through the `MCS` property. The range of values [0, 7], [8, 15], [16, 23], and [24, 31] correspond to one, two, three, and four streams of data respectively.
- Adapt the data rate according to the channel conditions through the `Rate Adaptation Algorithm` property. This is applicable only when the value of `PHY Tx Format` property is set to `Non-HT`. You can choose between `Auto Rate Fallback (ARF)` and `Minstrel` algorithms. To maintain a constant data rate throughout the simulation, `Fixed-Rate` option is available.

Block Parameters: MAC

EDCA MAC

Models the IEEE 802.11a/n/ac/ax MAC functionality

Parameters

Channel Bandwidth (MHz): 20

PHY Tx Format: HT-MF

MPDU Aggregation: Enable

Ack Policy: Normal Ack

Max A-MPDU Subframes: 64

MCS: 0

RTS Threshold (Bytes): 65535

Max Short Retries: 7

Max Long Retries: 7

Basic Rate Set (Mbps): [6 12 24]

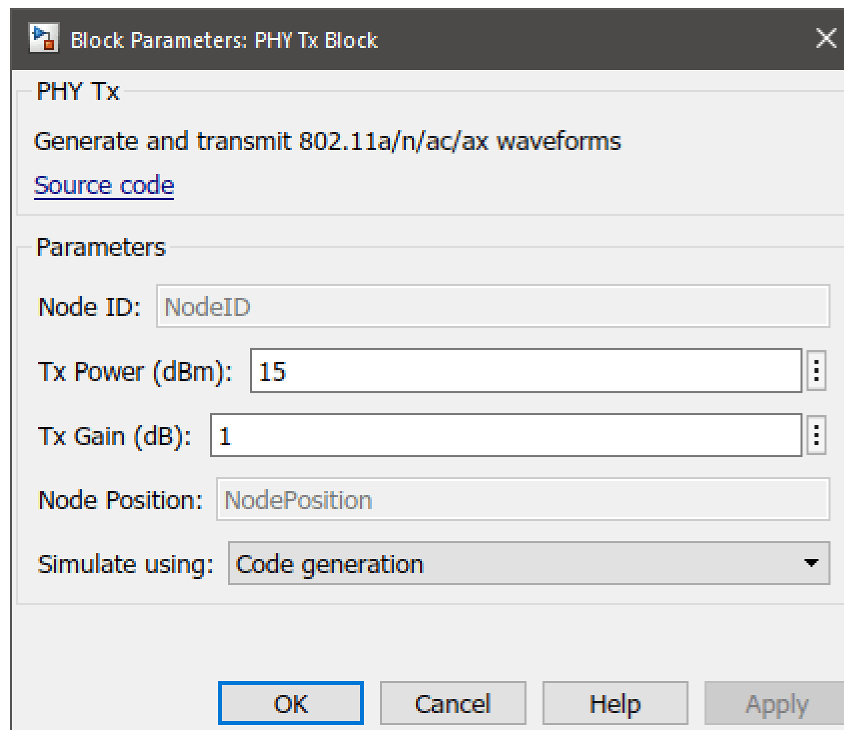
Use 6 Mbps for Control Frames

Tx Queue Size (Per Destination and Per AC): 64

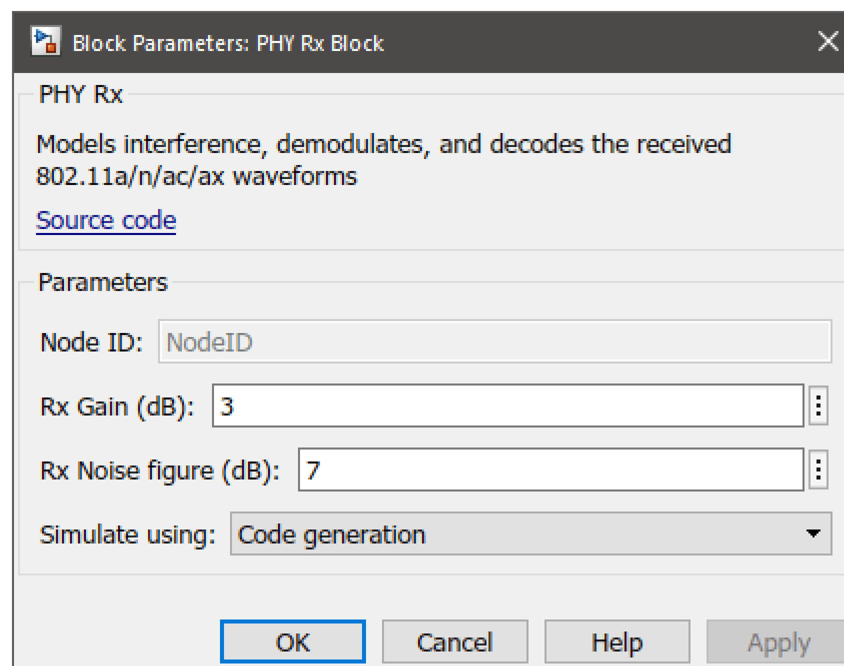
OK Cancel Help Apply

PHY:

The PHY Transmitter and PHY Receiver blocks have the capability to generate and decode waveforms of Non-HT, HT-MF, VHT, HE-SU and HE-EXT-SU formats. You can configure the transmit gain and transmit power using the `Tx Gain` and `Tx Power` properties in the mask parameters of the PHY Transmitter block inside a WLAN node.

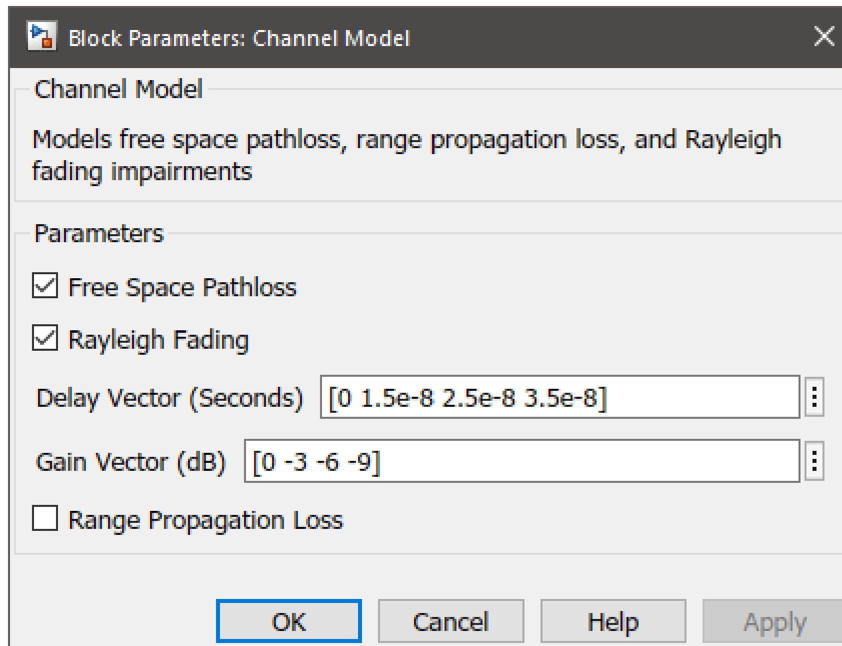


Similarly, you can configure the receive gain and receive noise figure using the `Rx Gain` and `Rx Noise Figure` properties in the mask parameters of the PHY Receiver block inside a WLAN node.



Channel:

Channel impairments determined by free-space path-loss model and Rayleigh multipath fading are added to the transmitted PHY waveform. You can choose to enable or disable these impairment models. In addition to the impairment models, the signal reception range can also be limited by an optional range propagation loss model. To model any of these losses, the channel model must contain both the sender and receiver positions along with the transmitted signal strength. The channel is modeled inside each receiving node, before passing the waveform to the PHY Receiver block.



Throughput Measurement

Throughput varies for different configuration parameters pertaining to the application, MAC & PHY layers. Any change in the configuration may either increase or decrease the throughput. You can vary the combination of these parameters to measure and analyze the throughput.

- `MCS`: PHY data rate
- `PHY Tx Format`: PHY transmission format
- `Packet Size`: Application packet size
- `Max A-MPDU Subframes`: Maximum number of subframes in an A-MPDU
- `Max Tx Queue Size`: MAC transmission queue size

Along with above parameters, you can also vary the node positions, Tx & Rx gains, channel loss, number of nodes in the network, MAC contention parameters, number of transmit chains and rate adaptation algorithms to analyze MAC throughput. This example demonstrates the measurement and analysis of the MAC throughput by varying packet size in the `Application Traffic Generator` block.

Application Packet Size

Throughput is directly proportional to the application packet size. Smaller packet size results in greater number of packets to be transmitted. At the MAC layer, there is an overhead of contention time for each transmitted packet. This is because the MAC layer makes sure that the channel is idle

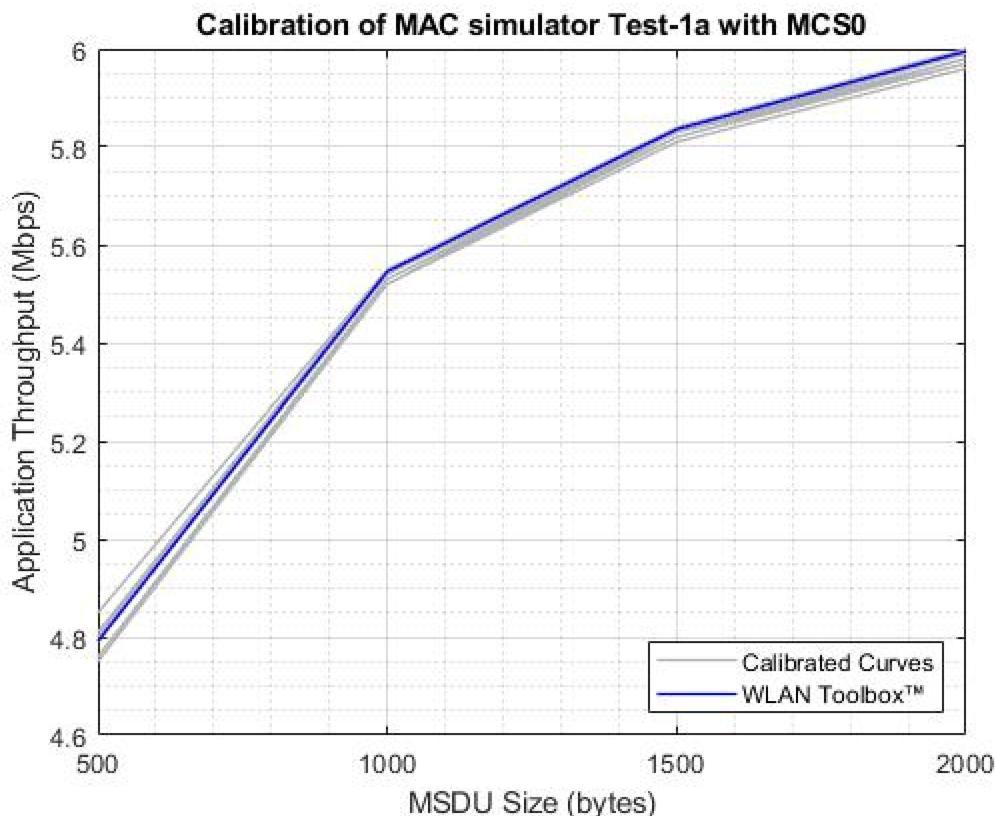
for a specific amount of time (Refer section 10.3.2.3 of [1]) before transmitting any packet. Therefore, as the packet size decreases, the contention overhead increases resulting in lower throughput.

Model Configuration

You can configure the application packet size using these steps:

1. Open model `WLANMACThroughputMeasurementModel.slx`
2. To go inside a node subsystem, click on the downward arrow at the bottom left of the node
3. To open mask parameters of the application, double click on `Application Traffic Generator`
4. To enable application, set `App State` to 'On'
5. Configure the value of `Packet Size`

Run the simulation and observe the throughput. The TGax calibration results for test-1a in [4] are shown below:



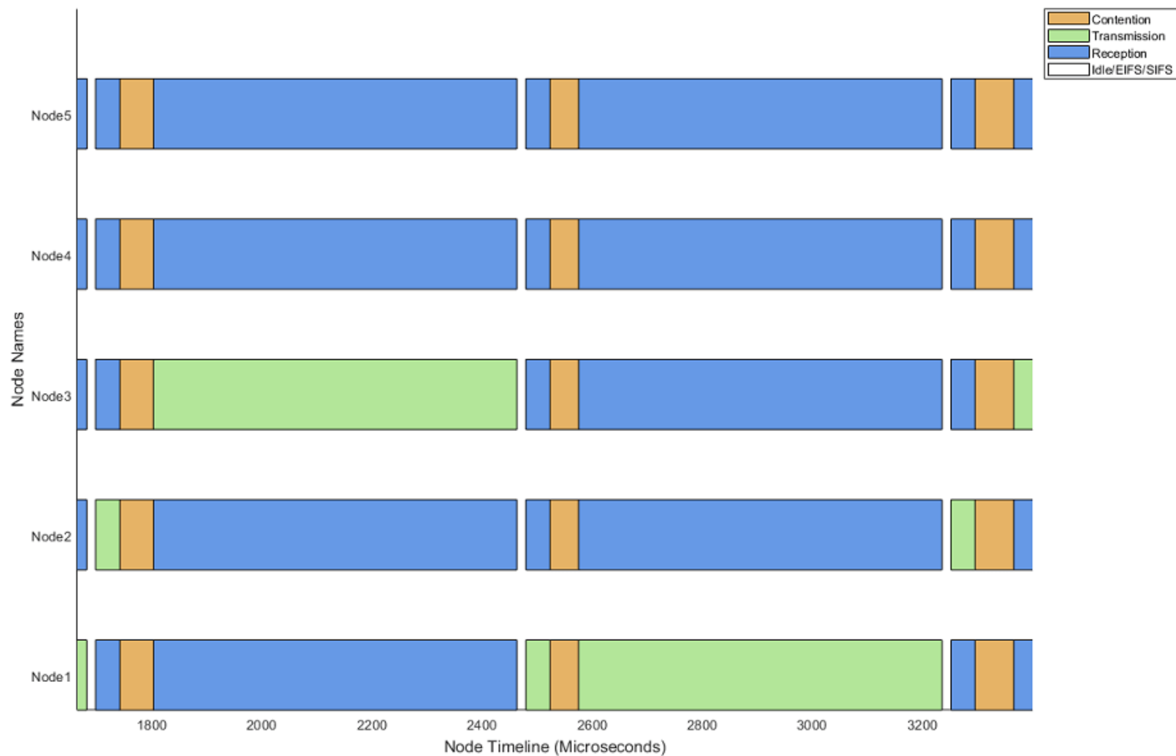
The above plot compares the calibration results for WLAN Toolbox against the published results of other companies listed in [4]. The blue colored curve represents the results of WLAN Toolbox, while the grey colored curves represent the results of other companies.

Simulation Results

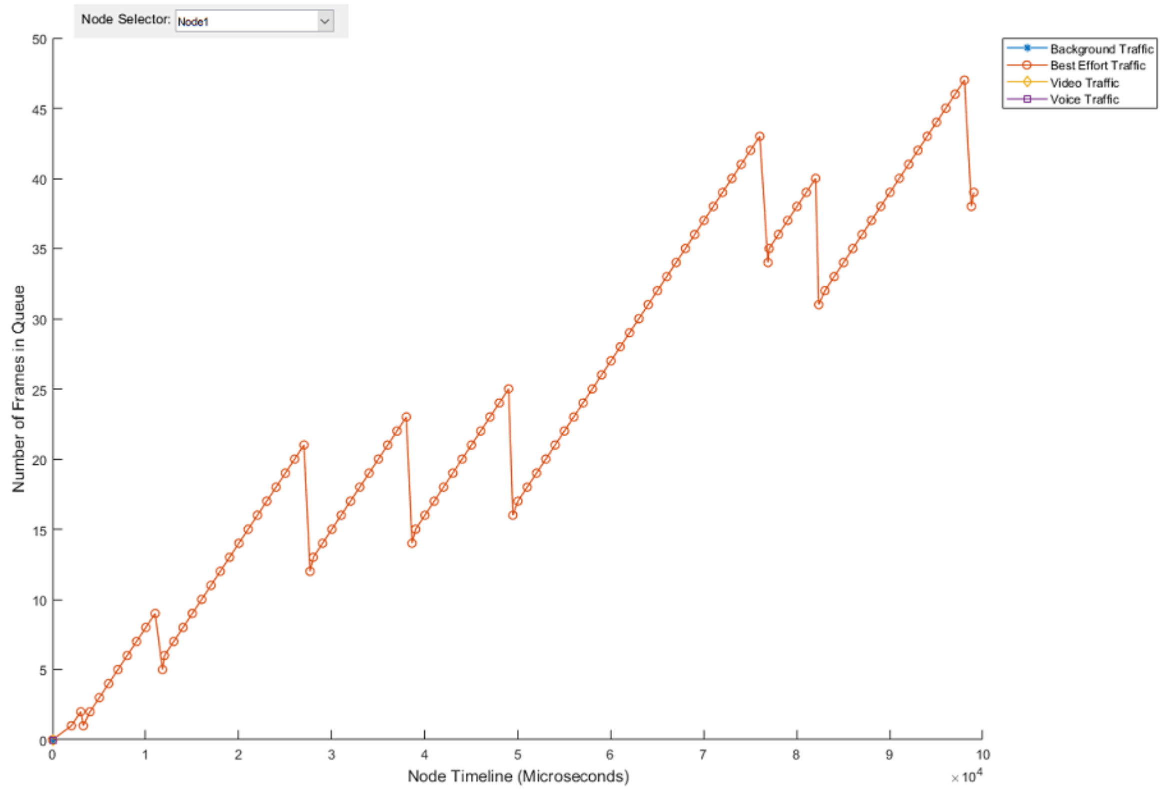
The simulation of the model generates:

1. A run-time visualization showing the time spent on channel contention, transmission, and reception for each node
2. An optional run-time visualization (during the simulation) showing the number of frames queued in MAC transmission queues for a selected node.
3. A bar graph showing metrics for each node such as number of transmitted, received, and dropped packets at PHY and MAC layers
4. A MAT file `statistics.mat` with detailed statistics obtained at each layer for each node

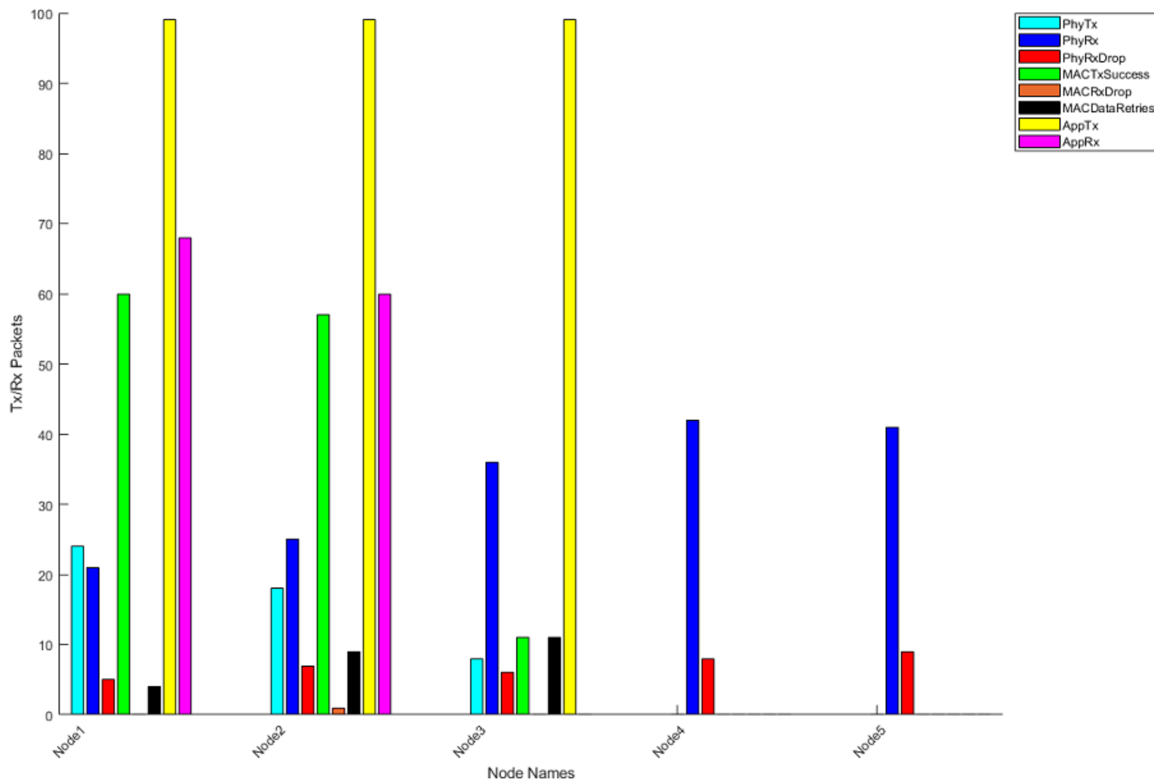
This figure shows MAC state transitions with respect to simulation time.



You can also observe the live state of the MAC layer transmission buffers using the 'Observe MAC queue lengths' button in the above visualization.



This figure shows the network statistics at the end of simulation.



Validating Application Layer Throughput with TGax Calibration Results

The TGax Task Group [4] published application throughput results for different scenarios. You can observe the Layer 3 (above MAC layer) throughput of each node in the network in 'Throughput' column in 'statisticsTable' stored in 'statistics.mat'. The TGax calibration scenarios for MAC simulator published results of application throughput for a User Datagram Protocol (UDP) with Logical Link Control (LLC) layers overhead.

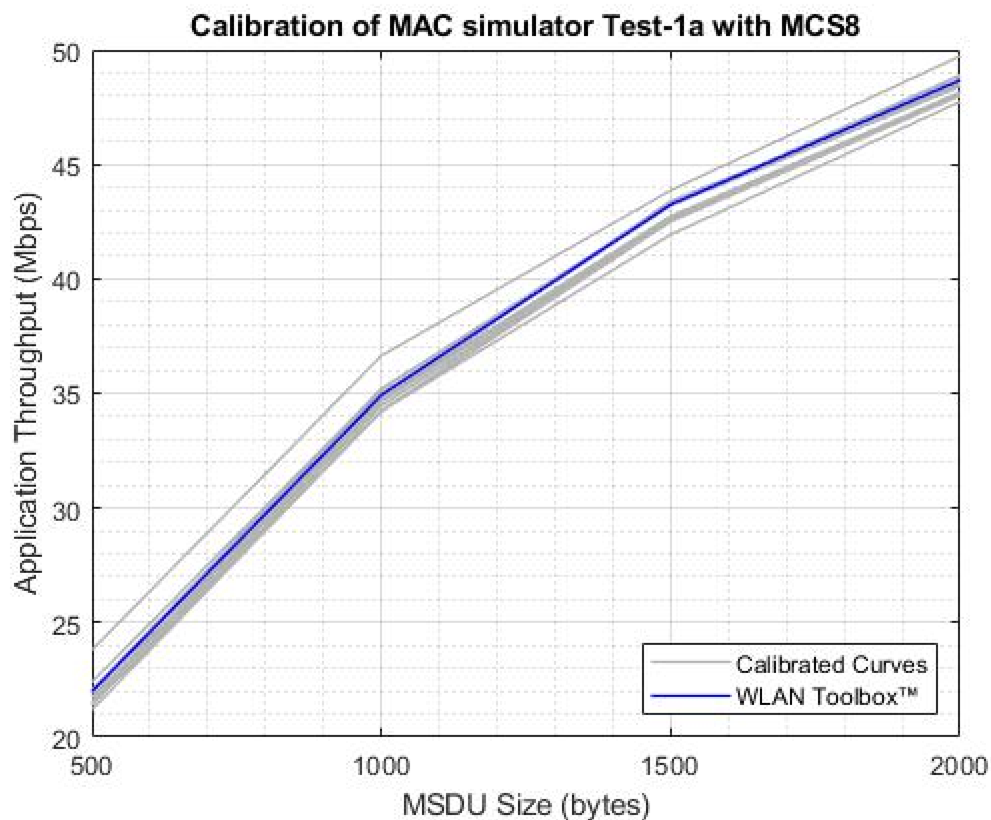
To calculate application throughput from simulation results use the code below:

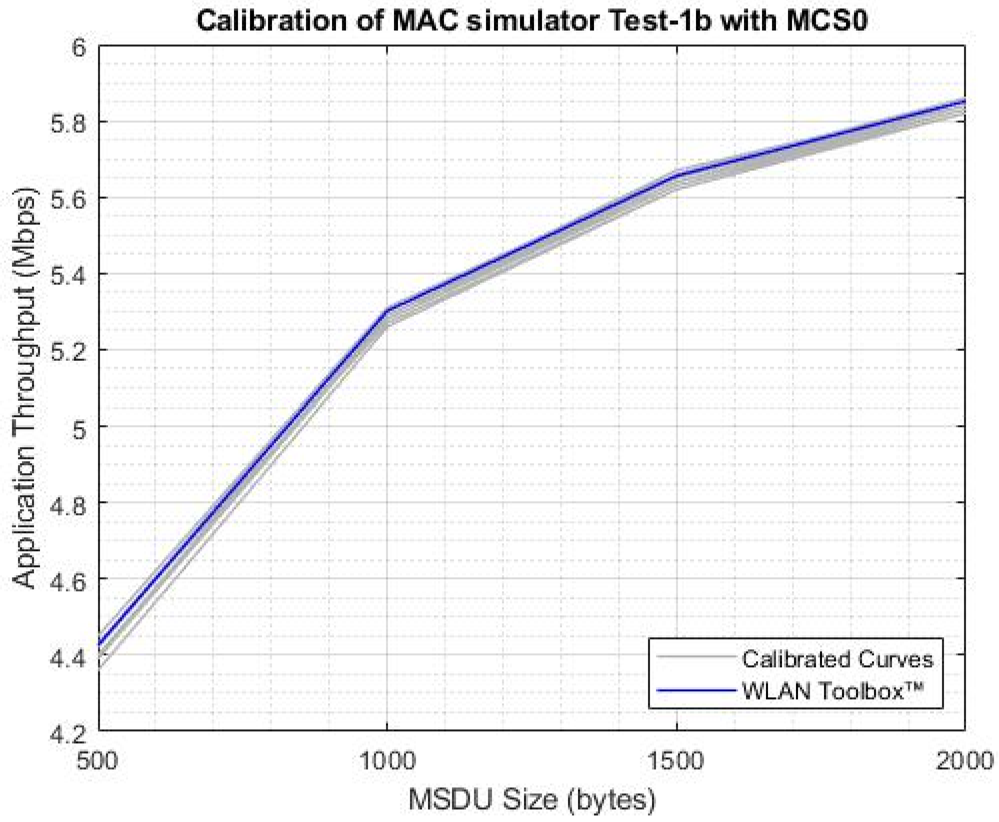
```
% Load statistics.mat (Output of the simulation) file
simulationResults = load('statistics', 'statisticsTable');
% Statistics
stats = simulationResults.statisticsTable;

% Successfully transmitted MAC layer bytes in the network
totalMACTxBytes = sum(stats.MACTxBytes);

% UDP & LLC overheads (bytes)
udpOverhead = 36;
llcOverhead = 8;

% UDP & LLC overhead (bytes) in the network
udpAndLLCOverhead = sum(stats.MACTxSuccess)*(udpOverhead + llcOverhead);

% Successfully transmitted application bytes
```

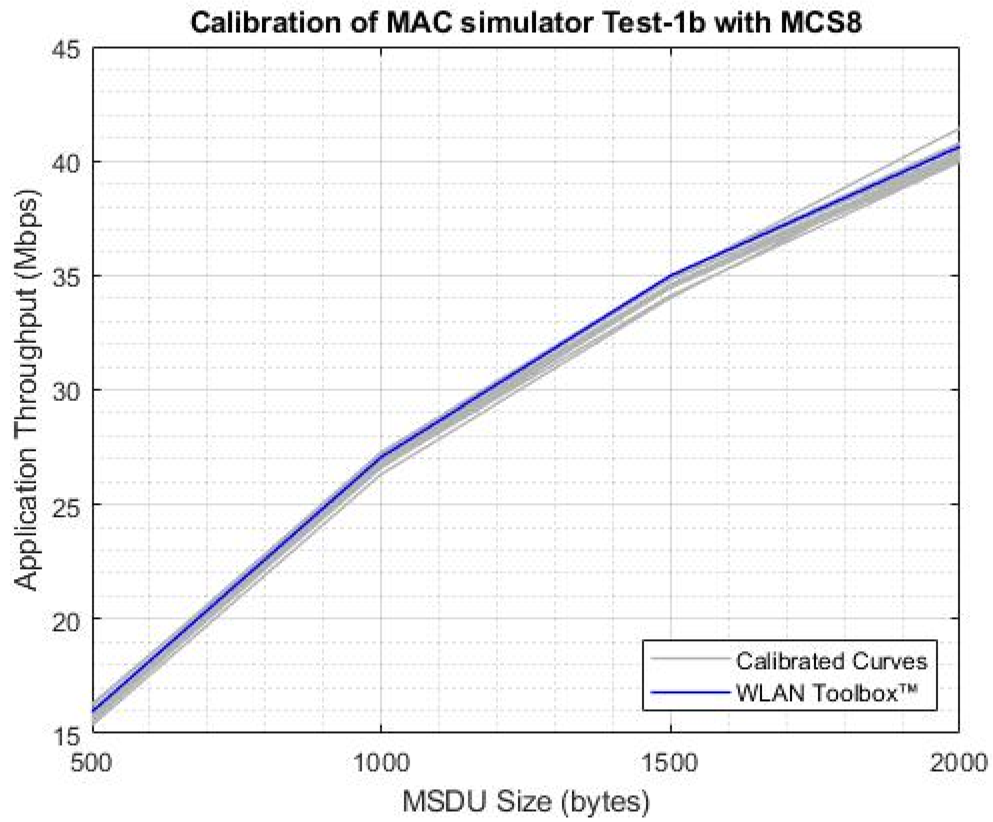
```
totalAppTxBytes = totalMACTxBytes - udpAndLLCOverhead;

% Time at which last transmission is completed in the network (Microseconds)
simulationTime = max(stats.MACRecentFrameStatusTimestamp);

% Application throughput (Mbps)
applicationThroughput = (totalAppTxBytes*8)/simulationTime;
disp(['Application Throughput = ' num2str(applicationThroughput) ' Mbps']);
```

```
Application Throughput = 4.7276 Mbps
```

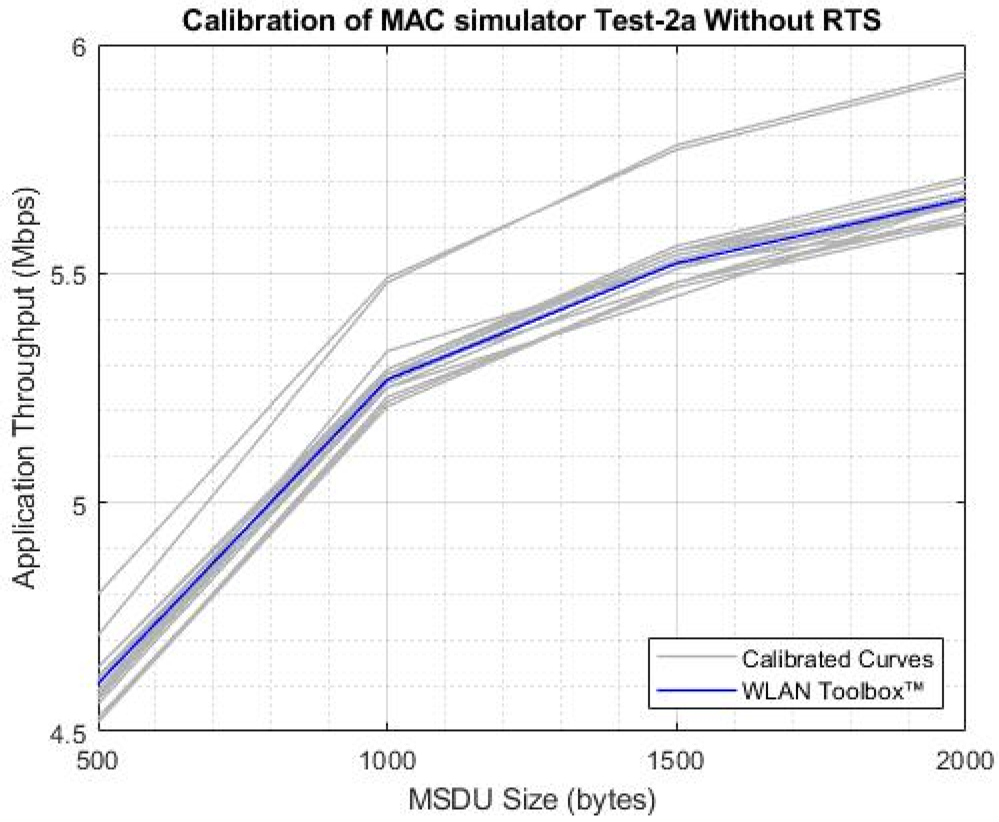
The application throughput for different TGax calibration scenarios is plotted against different MAC service data unit (MSDU) sizes for a simulation time of 30 seconds as shown below:

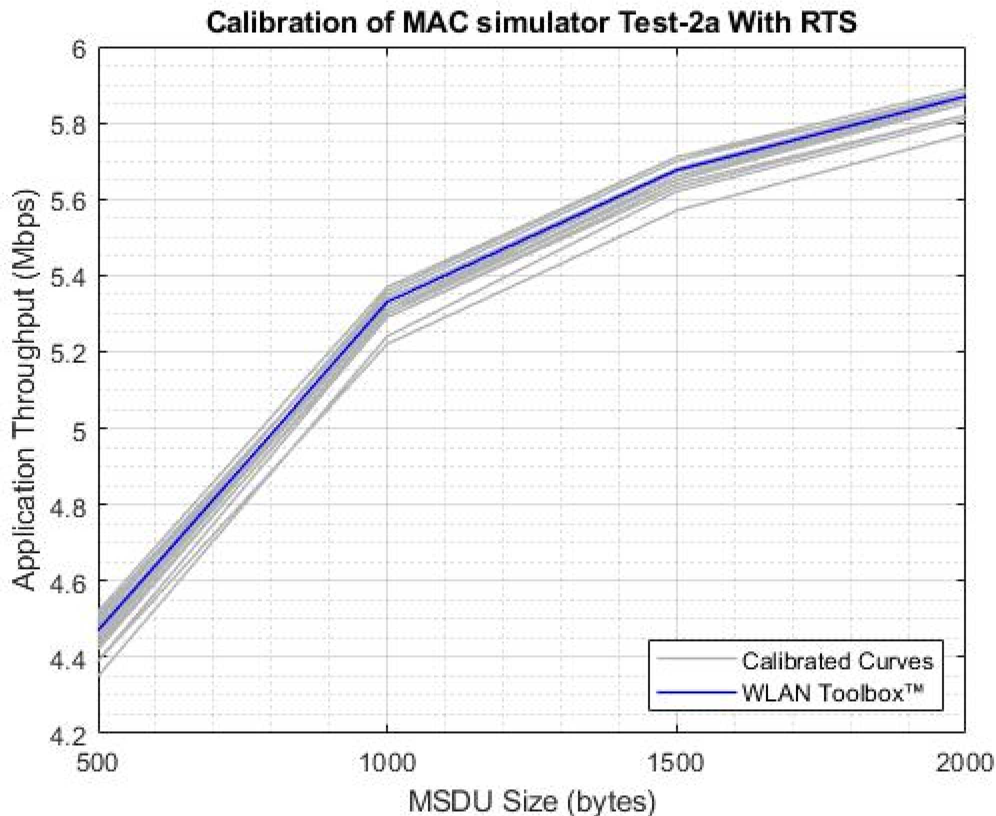



Calibration of MAC simulator Test-1b with MCS0
Application Throughput (Mbps)
6
5.8
5.6
5.4
5.2
5
4.8
4.6
4.4
4.2
500
1000
1500
2000
MSDU Size (bytes)
Calibrated Curves
WLAN Toolbox™



Calibration of MAC simulator Test-1b with MCS8
Application Throughput (Mbps)
45
40
35
30
25
20
15
500
1000
1500
2000
MSDU Size (bytes)
Calibrated Curves
WLAN Toolbox™


Calibration of MAC simulator Test-2a Without RTS
Application Throughput (Mbps)
6
5.5
5
4.5
500
1000
1500
2000
MSDU Size (bytes)
Calibrated Curves
WLAN Toolbox™



Further Exploration

Configuration options

You can change these configuration parameters to further explore this example:

- Application layer: Access category and packet interval
- MAC layer: RTS threshold, Tx queue size, data rate, short retry limit, long retry limit, transmitting frame format, MPDU aggregation, ack policy, number of transmit chains and the rate adaptation algorithms
- PHY: PHY Tx gain, PHY Rx gain, and Rx noise figure
- Channel modeling: Rayleigh fading, free space pathloss, range propagation loss and packet receive range
- Node positions using node position allocator
- The state of each node can be visualized during the run-time through the configuration available in the Visualizer block
- By default, the PHY transmitter and the receiver blocks run in the `Interpreted execution` mode. For longer simulation time, configure all the blocks to `Code generation` mode for better performance.

Related examples

Refer these examples for further exploration:

- To model a multi-node IEEE 802.11ax network with abstracted PHY using SimEvents, refer "802.11ax System-Level Simulation with Physical Layer Abstraction" on page 7-66 example.
- To get started on modeling a multi-node IEEE 802.11 network using MATLAB, refer "Get Started with WLAN System-Level Simulation in MATLAB" on page 7-31
- To model a multi-node IEEE 802.11ax residential scenario using MATLAB, refer "802.11ax Multinode System-Level Simulation of Residential Scenario" on page 7-40

This example enables you to create and configure a multi-node 802.11 network using a Simulink model for analyzing the MAC and application layer throughput. In this model, the MAC throughput obtained through the simulation results is used to calculate the application layer throughput. This model is validated using the Box 3 scenarios (Tests 1a, 1b, and 2a) specified in TGax evaluation methodology [3] to confirm that it complies with the IEEE 802.11 [1]. This example concludes that the calculated application layer throughput is within the range of minimum and maximum throughput specified in published calibration results [4].

Appendix

The helper functions and objects used in this example are:

1. edcaFrameFormats.m: Create an enumeration for PHY frame formats.
2. edcaNodeInfo.m: Return MAC address of a node.
3. edcaPlotQueueLengths.m: Plot MAC queue lengths in the simulation.
4. edcaPlotStats.m: Plot MAC state transitions with respect to simulation times.
5. edcaStats.m: Create an enumeration for simulation statistics.
6. edcaUpdateStats.m: Update statistics of the simulation.
7. helperAggregateMPDUs.m: Generate an A-MPDU, by creating and appending the MPDUs containing the MSDUs in the MSDULIST.
8. helperSubframeBoundaries.m: Return subframes information of an A-MPDU.
9. phyRx.m: Model PHY operations related to packet reception.
10. phyTx.m: Model PHY operations related to packet transmission.
11. edcaApplyFading.m: Apply Rayleigh fading effect on the waveform.
12. heSIGBUserFieldDecode.m: Decode HE-SIG-B user field.
13. heSIGBCommonFieldDecode.m: Decode HE-SIG-B common field.
14. heSIGBMergeSubchannels.m: Merge 20MHz HE-SIG-B subchannels.
15. addMUPadding.m: Add multiuser PSDU padding.
16. macQueueManagement.m: Create a WLAN MAC queue management object.
17. roundRobinScheduler.m: Create a round-robin scheduler object.
18. calculateSubframesCount.m: Return number of subframes to be aggregated.
19. interpretVHTSIGABitsFailCheck.m: Interprets the bits in VHT-SIG-A field
20. rateAdaptationARF.m: Create an auto rate fallback (ARF) algorithm object.
21. rateAdaptationMinstrelNonHT.m: Create a minstrel algorithm object.

References

1. IEEE Std 802.11™-2020. 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.

2 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

3 IEEE 802.11-14/0571r12. "11ax Evaluation Methodology." IEEE P802.11P: Wireless LANs.

4 Baron, Stephane., Nezou, Patrice., Guignard, Romain., and Viger, Pascal. "MAC Calibration Results." Presentation at the IEEE P802.11 - Task Group AX, September 2015.

See Also

More About

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “802.11ax Multinode System-Level Simulation of Residential Scenario” on page 7-40

802.11ax System-Level Simulation with Physical Layer Abstraction

This example demonstrates how to model a multi-node IEEE® 802.11ax™ [1] network with abstracted physical layer (PHY) using SimEvents®, Stateflow®, and WLAN Toolbox™. A PHY abstraction model largely reduces the complexity and the duration of system-level simulations by replacing the actual physical layer computations. This makes it possible to evaluate systems consisting of large number of nodes, resulting in increased scalability. Abstracted PHY models signal-power, gain, delay, loss and interference on each packet without generating physical layer packets, as specified by the TGax Evaluation Methodology [3].

Physical Layer Abstraction

This example shows how to model an 802.11ax network with abstracted PHY. The example presents a variation of the system model used in the example "802.11 MAC and Application Throughput Measurement" on page 7-49. In "802.11 MAC and Application Throughput Measurement" on page 7-49 example, full PHY processing is modeled where waveforms are generated and decoded at the physical layer. However, this example models an abstracted PHY where no waveforms are generated or decoded. Abstracting the physical layer reduces the time taken for simulation at the cost of fidelity. Fidelity refers to the degree of exactness with which the PHY is modeled in the simulation. Simulations that tolerate low fidelity at the physical layer can use the abstracted PHY model.

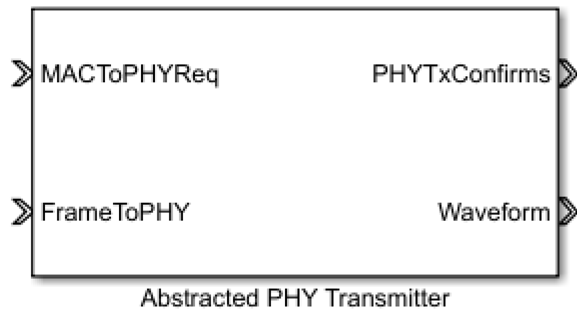
The abstracted PHY operates on pre-computed packet error rate (PER) tables and equations. These tables and equations are used to estimate the corrupted packet without any actual modulation or demodulation of packets, resulting in a low fidelity model. Refer the example "Physical Layer Abstraction for System-Level Simulation" on page 7-90 for more details related to the PHY abstraction.

Abstracted Physical Layer Blocks

This section explains the blocks used for modeling the abstracted PHY and how it fits into the 802.11 [2] network model. Full PHY modeling involves operations related to waveform transmission and reception through a fading channel. Abstracted PHY models signal-power, gain, delay, loss and interference on each packet without generating physical layer packets. This example provides a `PHY Transmitter`, a `Statistical Channel`, and a `PHY Receiver` for modeling an abstracted PHY. These blocks are available in the library wlanAbstractedPHYLib.

Abstracted PHY Transmitter:

The `Abstracted PHY Transmitter` block models the transmit chain of the physical layer. This block consumes the frame and corresponding transmission parameters from the MAC layer. Parameters like transmit power, preamble duration, header duration and payload duration are calculated in the block. This information is passed along with the MAC frame as the metadata to simulate the transmission of a waveform.

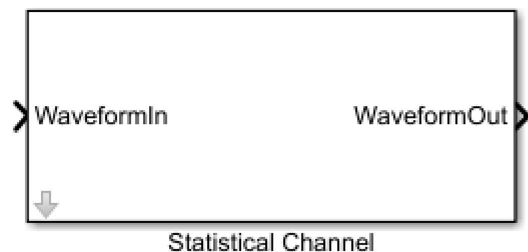


Interfaces to the `Abstracted PHY Transmitter` block are:

- MACToPHYReq: Triggers for indicating transmission start/end requests from MAC layer
- FrameToPHY: MAC frame to be transmitted
- PhyTxConfirms: Confirmation triggers to MAC layer for indicating completion of MAC layer requests
- Waveform: Abstract waveform transmitted into the channel (MAC frame and the metadata)

Statistical Channel:

The `Statistical Channel` block models pathloss, propagation delay, and reception range of the packet. To enable the estimation of loss, delay, and range at each receiver, the `Statistical Channel` block must be modeled inside every node coupled with the `Abstracted PHY Receiver`. Propagation delay is applied on each received packet, and the signal strength of each packet is degraded with optional pathloss. If the receiving node is within the range, the packet is forwarded to the `Abstracted PHY Receiver` with the effective signal strength. The packet is dropped if the receiving node is outside the range of the transmitter.



Interfaces to the `Statistical Channel` are:

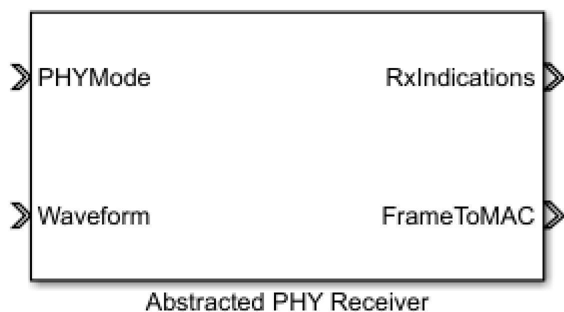
- WaveformIn: Input packet received from a PHY transmitter
- WaveformOut: Output packet intended for PHY receivers after applying channel loss

Abstracted PHY Receiver:

The `Abstracted PHY Receiver` block models the receive chain of the physical layer. This block receives and processes the packet based on the received metadata. The `Abstracted PHY Receiver` block models interference based on the packets received at overlapping timelines. The received packets are processed only at these checkpoints: (a) End of the preamble duration (b) End of

each subframe duration in the payload for aggregated frames (or) end of the payload duration for non-aggregated frames.

This block also provides an option for configuring the level of abstraction through the `PHY Abstraction` mask parameter. You can configure it to `'TGax Evaluation Methodology Appendix 1'` [3] to predict the performance of a link with a TGax channel model using effective SINR mapping. Details of this procedure can be found in the example "Physical Layer Abstraction for System-Level Simulation" on page 7-90. Alternatively, you can configure it to `'TGax Simulation Scenarios MAC Calibration'` [4] to assume a packet failure on interference, without actually calculating the link performance. Note that the option `'TGax Evaluation Methodology Appendix 1'` works for only MCS values in the range [0-9], as the TGax Evaluation Methodology [3] is defined only for these values.

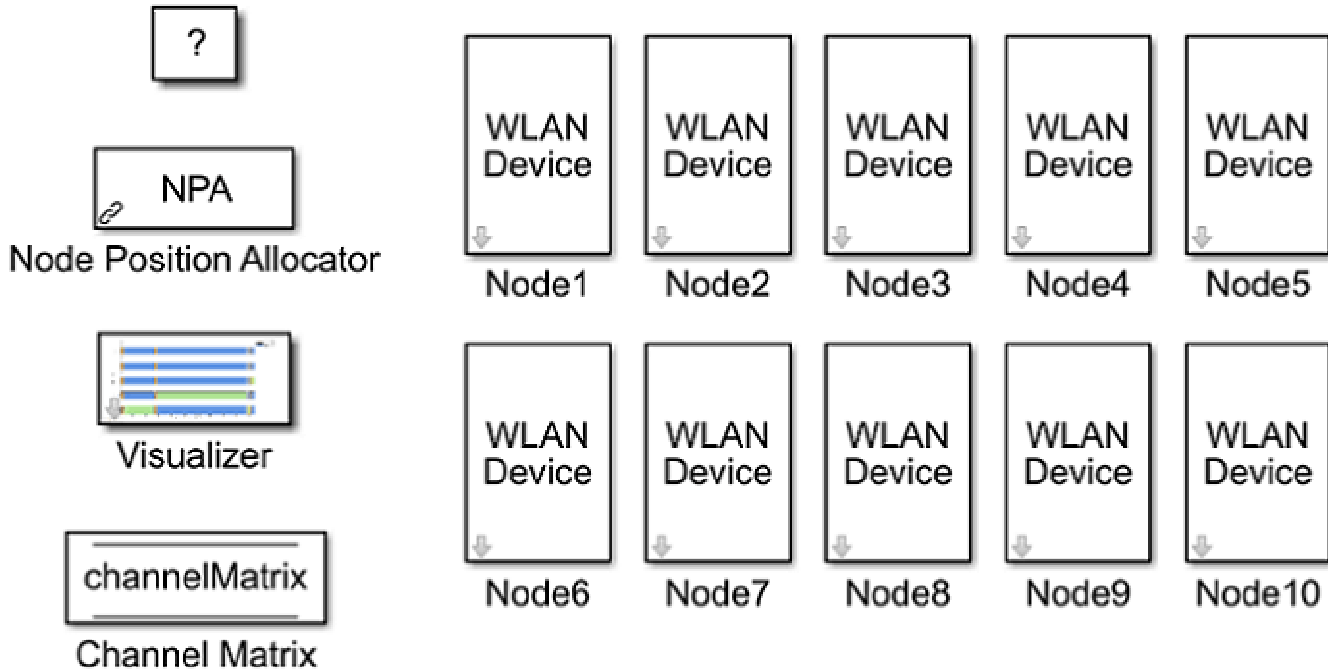


Interfaces to the `Abstracted PHY Receiver` block are:

- PHYMode: Trigger for switching off the receiver function when transmission is in progress
- Waveform: Abstract waveform received from the channel (MAC frame and the metadata)
- RxIndications: Triggers to MAC for indicating channel state shift (busy/idle) events or receive (start/end) events
- FrameToMAC: Received MAC frame

System-Level Simulation

This example simulates a network with 10 nodes in the model, WLANMultiNodeAbstractedPHYModel, as shown in this figure. These nodes implement carrier-sense multiple access with collision avoidance (CSMA/CA) with physical carrier sense and virtual carrier sense. The physical carrier sensing uses the clear channel assessment (CCA) mechanism to determine whether the medium is busy before transmitting. The virtual carrier sensing uses the RTS/CTS handshake to prevent the hidden node problem.

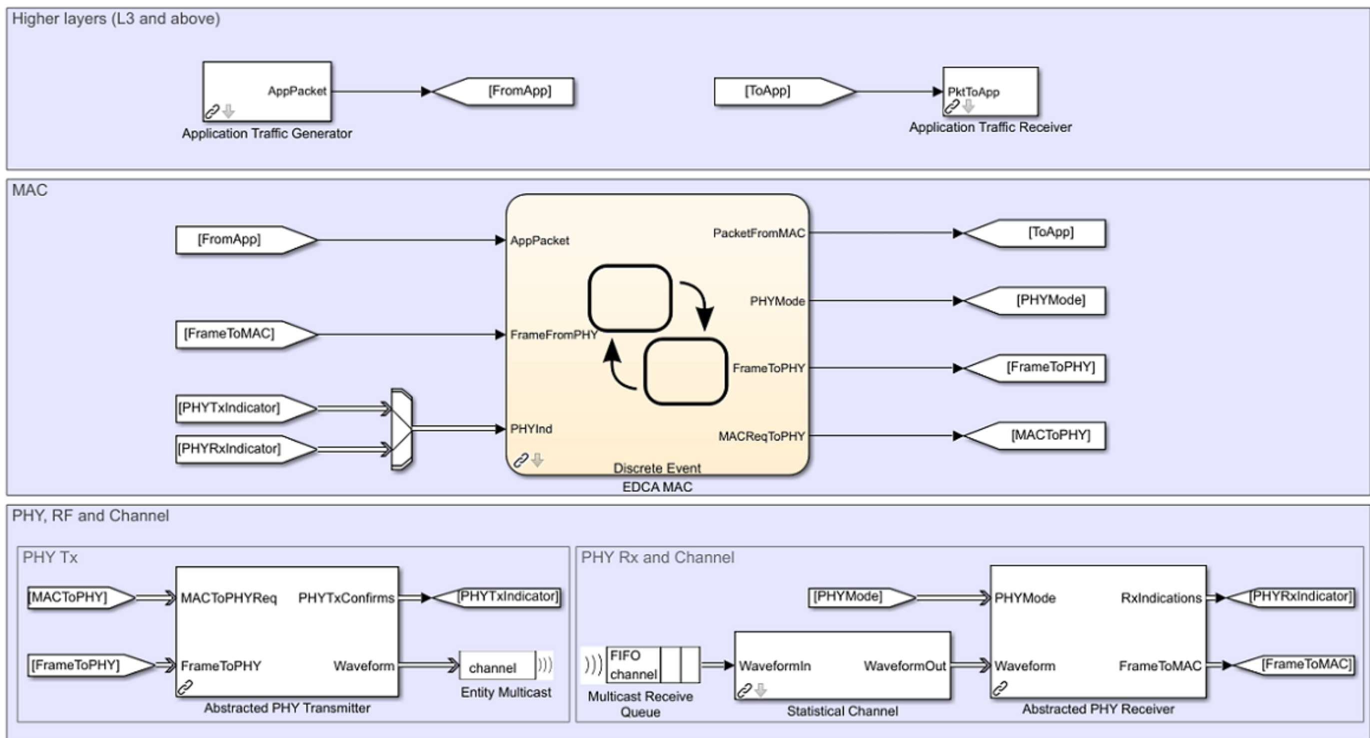


The positions for all the nodes in the network are configured through the node position allocator (`NPA`) block in the model. The state of each node can be visualized during run-time through the configuration available in the `Visualizer` block. The `Channel Matrix` block is a Data Store Memory. On initialization, a TGax channel realization is generated between each pair of nodes in the network and the resulting channel matrix per subcarrier is stored in the block. During the simulation, each receiver node accesses the memory to obtain the channel matrix between itself and a transmitting node to determine the link quality. In this model, nodes 1, 2, 3, 6, 7, and 8 act as both the transmitters and receivers, while nodes 4, 5, 9, and 10 are just passive receivers.

Node Subsystem

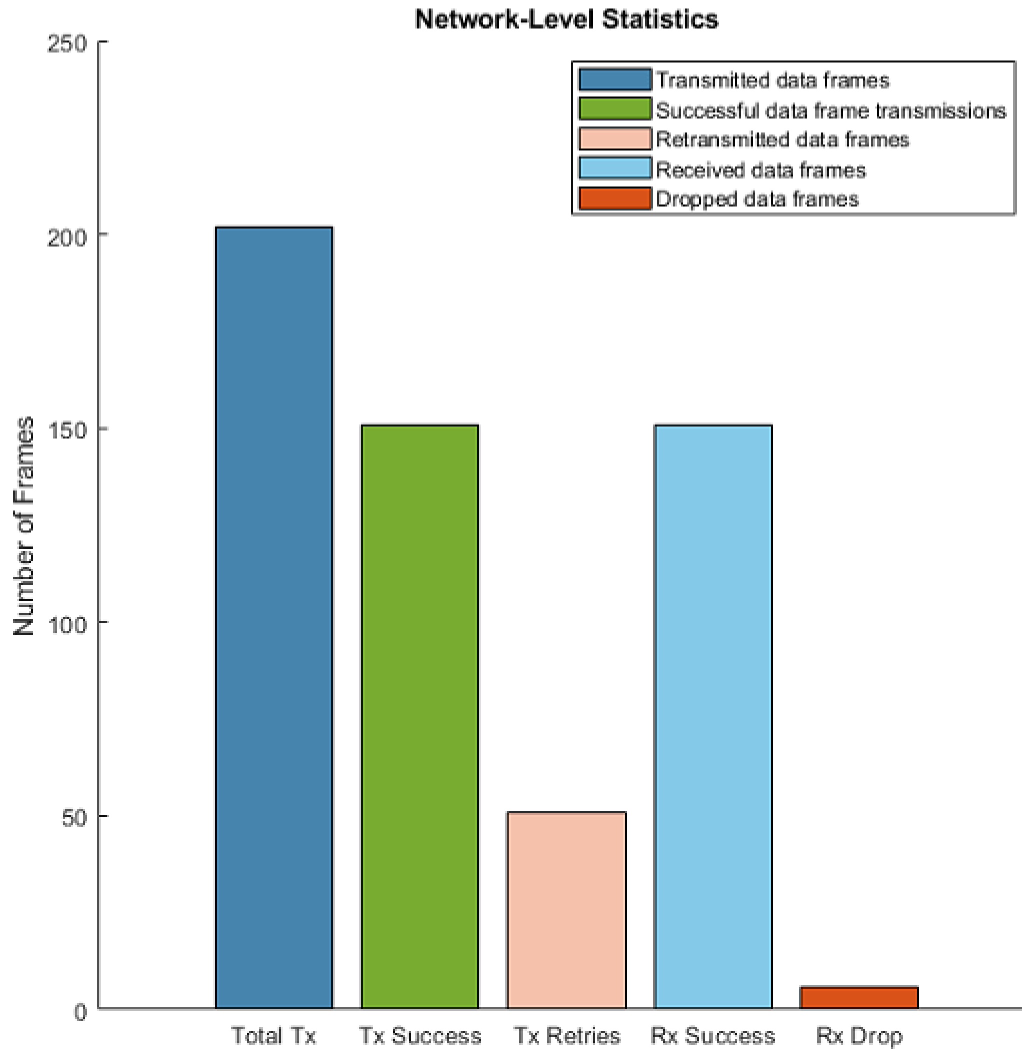
Each node in the above model is a subsystem representing a WLAN device. Each node contains an application layer, a MAC layer and a physical layer. The physical layer is modeled using the abstracted PHY blocks described in the previous section. You can configure a node to transmit and receive packets on a specific channel (frequency) by changing the `Multicast tag` parameter of the `Entity Multicast` and the `Multicast Receive Queue` blocks. By default, all nodes operate on the same channel. You can also configure the receive range for a specific node using the `Packet Receive Range` parameter of the `Statistical Channel` block.

You can easily switch between abstracted PHY blocks available in the wlanAbstractedPHYLib and full PHY processing blocks available in the `wlanFullPHYLib.slx` library of the example “802.11 MAC and Application Throughput Measurement” on page 7-49. The interfaces to the transmitter, receiver and channel blocks remain the same. By default, the abstracted PHY blocks run in the `Interpreted execution` mode. For longer simulation time, configure all the blocks to `Code generation` mode for better performance.



Simulation results

Running the model simulates the WLAN network for the specified simulation time. A plot with network-level statistics (corresponding to MAC layer) is generated at the end of simulation. Detailed node-level statistics (corresponding to application, MAC, and physical layers) are collected during the simulation and saved to a base workspace file `statistics.mat`. You can also enable an optional live visualization, to see the state of each node during run-time, through the mask configuration of the `Visualizer` block.

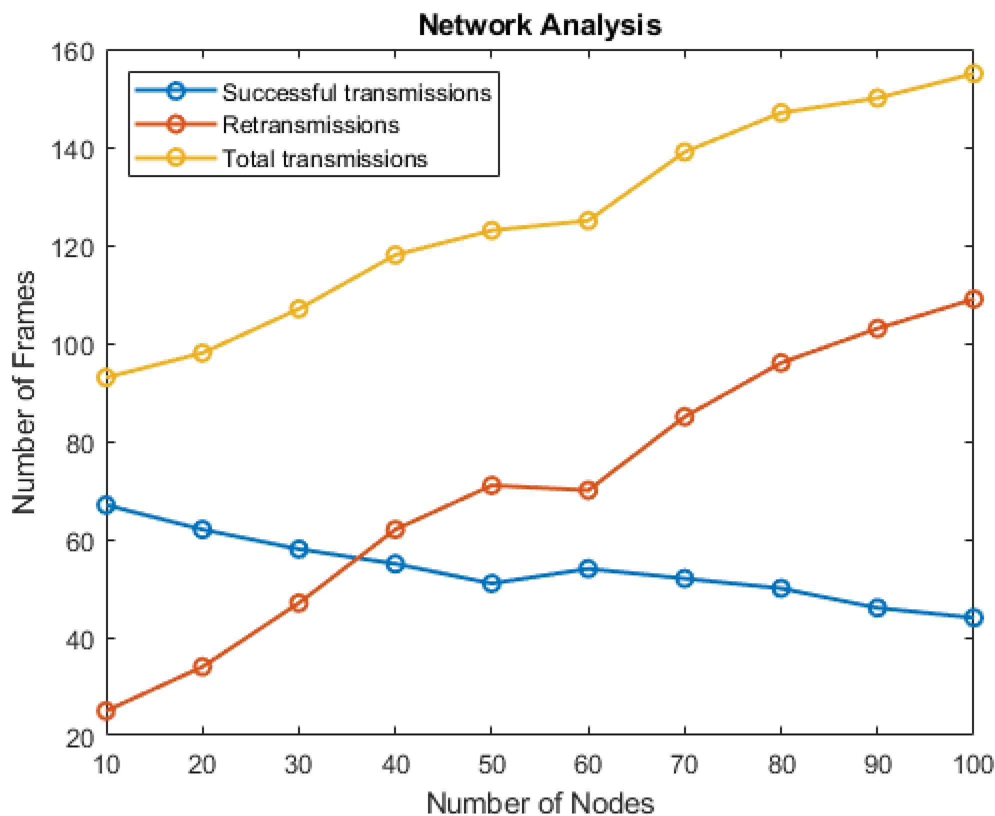


Scalability

The above model shows a network of 10 nodes. You can create a network with a large number of nodes by using the hCreateWLANNetworkModel function. This helper function uses the node subsystem from this example and creates a network of WLAN nodes positioned linearly 10 meters apart from each other. You can create different simulation scenarios and analyze the node-level or network-level statistics with varying number of nodes. For example, the plot below shows the retransmissions and successful transmissions relative to the total transmissions, as the number of nodes in the network increase. The configuration parameters used for collecting the results are:

- Format: HE-SU
- Modulation and coding scheme (MCS) index: 0
- Number of subframes in A-MPDU: 1

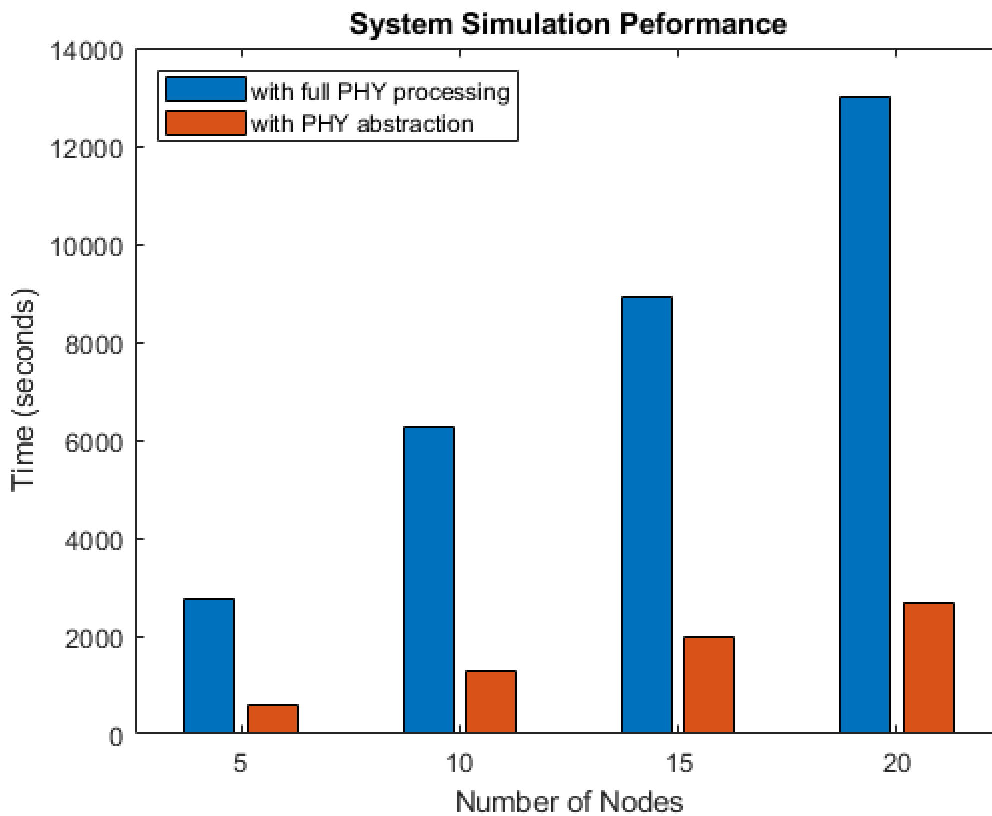
- Distance between nodes: 10 meters
- Path loss: Not applied
- PHY abstraction type: "TGax Evaluation Methodology Appendix 1"
- Range propagation: All the nodes are within range of each other
- Operating frequency: All the nodes operate in the same frequency



The plot below shows that the simulation runs faster with abstracted PHY as compared to full PHY processing, thus making it more scalable. The configuration parameters used for collecting the performance results are:

- Format: HE-SU
- Modulation and coding scheme (MCS) index: 0
- Number of subframes in A-MPDU: 2
- Distance between nodes: 1 meter
- Path loss: Not applied
- PHY abstraction type: "TGax Evaluation Methodology Appendix 1"
- Range propagation: All nodes are within range of each other
- Operating frequency: All the nodes operate in the same frequency
- Simulation mode: `Code generation` mode for all the blocks
- Simulation time: 5 seconds

- Packet generation interval: 0.001 seconds



This example explained the physical layer abstraction and demonstrated a 10-node WLAN network with abstracted PHY. This example shows that a network simulation with abstracted PHY is faster and more scalable compared to using full PHY processing.

Further Exploration

In this example, the A-MPDUs exchanged between the nodes are deaggregated to MPDUs at the receiving node. These MPDUs are exported to packet capture (PCAP) and packet capture next generation (PCAPNG) format file using the `pcapDump` DES block. To use the `pcapDump` DES block, go to wlanSystemLevelComponentsLib

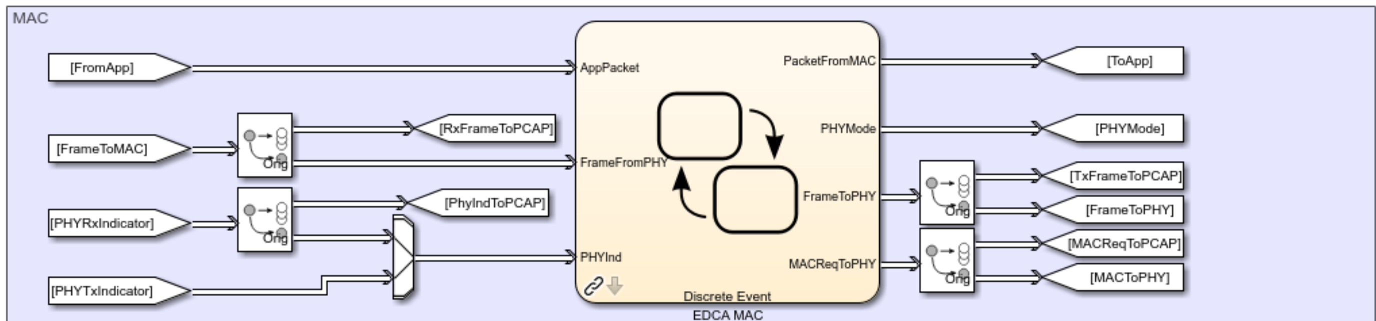
Export to PCAP/PCAPNG Format File

The PCAP/PCAPNG format files contain the packet data of the network. These files are mainly associated with network analyzers like Wireshark [5], a third party tool used to visualize and analyze PCAP/PCAPNG files. The main advantages of using PCAP/PCAPNG files during system level simulations are:

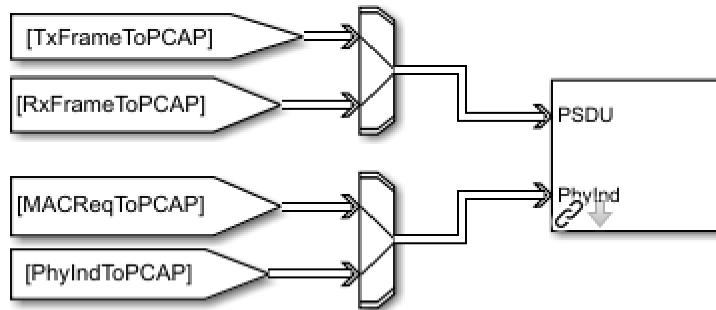
- Monitor the network traffic.
- Visualize and analyze the network characteristics of the data.

To duplicate the MAC layer input entities (received A-MPDUs, `FrameToMAC`, and `PhyRxIndicator` vector) and output entities (transmitted A-MPDUs, `FrameToPHY`, and `MACReqToPHY` vector), use the

Entity Replicator blocks. The MAC layer provides RxFrameToPCAP, PhyIndToPCAP, TxFrameToPCAP, and MACReqToPCAP as inputs to the pcapDump DES block.

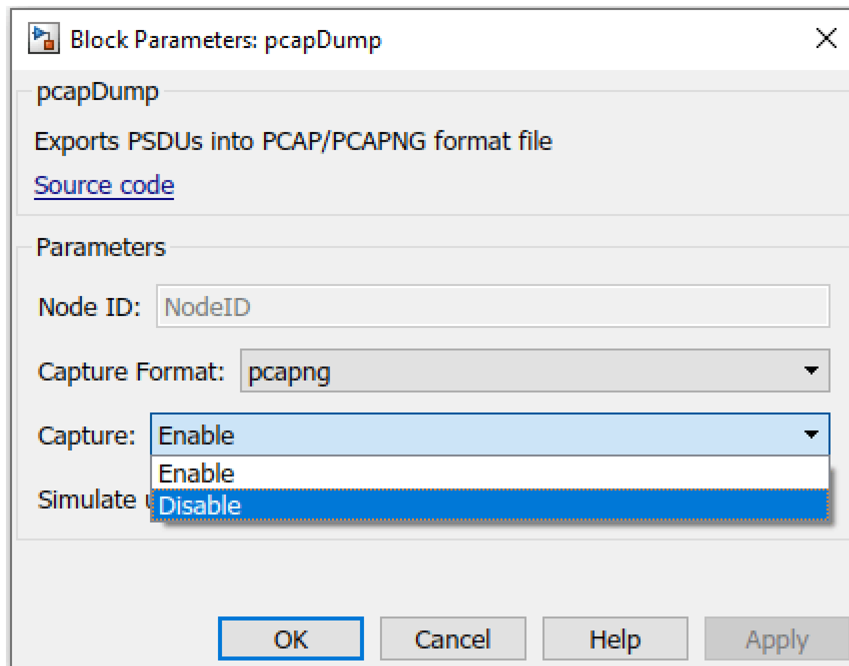


The pcapDump DES block contains two input ports, one for Tx/Rx A-MPDUs and other for Tx/Rx information.



Select the capture format as pcap or pcapng. As the simulation starts, the packets exchanged between the nodes are logged into the selected capture format file.

To capture the packet, double click the pcapDump DES block and select the parameter Capture as Enable.



A new capture file (PCAP/PCAPNG format) is created for every node. The file name corresponds to name of the node. If name of the node is Node1, the captured file name is Node1.pcap or Node1.pcapng.

Appendix

The example uses these helpers:

1. edcaFrameFormats.m: Create an enumeration for PHY frame formats.
2. edcaNodeInfo.m: Return MAC address of a node.
3. edcaPlotQueueLengths.m: Plot MAC queue lengths in the simulation.
4. edcaPlotStats.m: Plot MAC state transitions with respect to simulation times.
5. edcaStats.m: Create an enumeration for simulation statistics.
6. edcaUpdateStats.m: Update statistics of the simulation.
7. helperSubframeBoundaries.m : Return subframe boundaries of an A-MPDU.
8. phyTxAbstracted: Model PHY operations related to packet transmission
9. phyRxAbstracted: Model PHY operations related to packet reception
10. channelBlock: Model the channel for a node
11. addMUPadding.m: Add or remove the padding difference between an HE-SU and HE-MU PSDU
12. macQueueManagement.m: Create a WLAN MAC queue management object
13. roundRobinScheduler.m: Create round-robin scheduler object
14. calculateSubframesCount.m: Calculate the number of subframes required to form MU-PSDU
15. hCreateWLANNetworkModel: Create a WLAN network with given number of nodes
16. hDisplayNetworkStats: Display network level statistics
17. hSetupAbstractChannel: TGax channel setup

18 createRadiotapHeader: Create a radiotap header

19 rateAdaptationARF.m: Create an auto rate fallback (ARF) algorithm object.

20 rateAdaptationMinstrelNonHT.m: Create a minstrel algorithm object.

References

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

2 IEEE Std 802.11™-2020. 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 IEEE 802.11-14/0571r12 - 11ax Evaluation Methodology.

4 IEEE 802.11-14/0980r16 - TGax Simulation Scenarios.

5 Wireshark - Go Deep. https://www.wireshark.org/. Accessed 9 Dec. 2019.

See Also

More About

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “802.11ax Multinode System-Level Simulation of Residential Scenario” on page 7-40

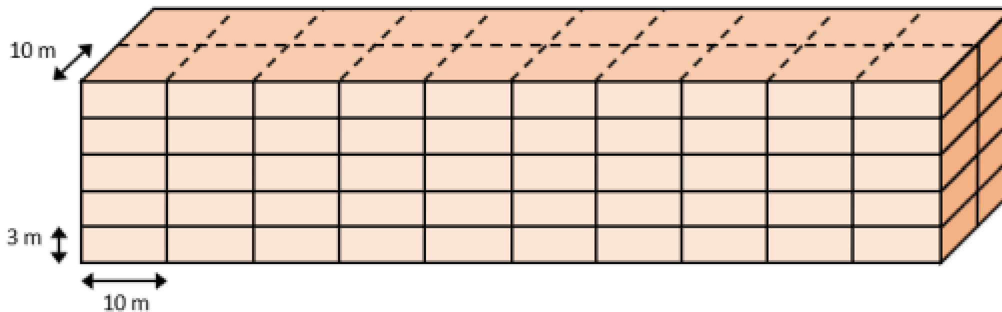
802.11ax PHY-Focused System-Level Simulation

This example shows how to perform a PHY-focused system-level simulation for IEEE® 802.11ax™ (Wi-Fi 6). Part (A) validates the simulation scenario, radio characteristics, and large-scale fading model by comparing against published calibration results. Part (B) estimates the packet error rate of the 802.11ax network by simulating individual links between active nodes under a basic clear channel assessment scheme.

Introduction

In this example the performance of an 802.11ax [1] network in a residential apartment block is evaluated using a PHY-focused system-level simulation.

The residential apartment block simulation scenario is specified in [2]. This consists of a building with five floors, and twenty 10m x 10m x 3m apartments per floor. Each apartment has an access point (AP) and one or more stations (STAs) placed in random xy-locations, a process referred to as 'dropping' nodes. This creates a basic service set (BSS) per apartment which is randomly assigned one of three channels. The simulation scenario specifies a large-scale path loss model based on the distance between nodes, and the number of walls and floors traversed.



The TGax evaluation methodology [3] for 'PHY System Simulation' is followed for this example:

1. APs and STAs are randomly 'dropped' within the scenario.
2. For each pair of nodes, the large-scale path loss is calculated.
3. One or more 'transmission events' is performed. Each transmission event consists of selecting active APs and STAs based on channel access rules and determining the performance of each link.

This example consists of two parts:

In part (A), the 'calibration' stage, the signal-to-interference-plus-noise ratio (SINR) is calculated for multiple 'drops', assuming downlink interfering transmissions. SINR captures long-term radio characteristics. The cumulative distribution function (CDF) of the SINR is compared with published results from the TGax Task Group [4].

In part (B), the 'PHY system-level simulation' stage, for each transmission event the PHY layer is modeled for individual links. A basic clear channel assessment (CCA) scheme is used to control which APs are active. Waveforms for the signal of interest and interference, impaired by fading channel models, are generated and combined. The resultant packets are processed by a receiver to recover the packet of interest. The average packet error rate for the network is calculated.

The two parts of this example can be disabled using the parameters `calibrate` and `systemLevelSimulation`. A figure displaying the simulation scenarios, nodes, active links, and interference is displayed when `showScenarioPlot` is true.

```
calibrate = true;             % To execute Part A calibration test
systemLevelSimulation = true; % To execute Part B system-level simulation
showScenarioPlot = true;      % To show dynamic simulation plotting updates
```

Simulation Parameters

The major simulation parameters are defined as either belonging to Physical Layer (PHY), Medium Access Control Layer (MAC), scenario, or simulation. In this example the PHY and MAC parameters are assumed to be the same for all nodes.

```
PHYParameters = struct;
PHYParameters.TxPower = 20;      % Transmitter power in dBm
PHYParameters.TxGain = 0;        % Transmitter antenna gain in dBi
PHYParameters.RxGain = -2;       % Receiver antenna gain in dBi
PHYParameters.NoiseFigure = 7;   % Receiver noise figure in dB
PHYParameters.NumTxAntennas = 1; % Number of transmitter antennas
PHYParameters.NumRxAntennas = 1; % Number of receiver antennas
PHYParameters.ChannelBandwidth = 'CBW80'; % Bandwidth of system
PHYParameters.TransmitterFrequency = 5e9; % Transmitter frequency in Hz

MACParameters = struct;
MACParameters.NumChannels = 3; % Number of non-overlapping channels
MACParameters.CCALevel = -70;  % Transmission threshold in CCA algorithm (dBm)
```

The scenario parameters define the size and layout of the residential building as per [3].

```
% Number of Rooms in [x,y,z] directions
ScenarioParameters = struct;
ScenarioParameters.BuildingLayout = [10 2 5];

% Size of each room in meters [x,y,z]
ScenarioParameters.RoomSize = [10 10 3];

% Number of receivers per room. Note that only one receiver (STA) can be
% active at any given time.
ScenarioParameters.NumRxPerRoom = 1;
```

The `NumDrops` and `NumTxEventsPerDrop` parameters control the length of the simulation. For this example, these parameters are configured for a short simulation, but for meaningful results these should be increased.

A 'drop' randomly places transmitters and receivers within the scenario and selects the channel for a BSS. A 'transmission' event randomly selects transmitters and receivers for transmissions according to basic channel access rules.

```
SimParameters = struct;
SimParameters.NumDrops = 3;
SimParameters.NumTxEventsPerDrop = 2;
```

Generate Transmitter Sites

Before the main body of the simulation, the transmitter site objects `txsite` are generated and assigned room names of the form 'Room#' for ease of reference. One transmitter (AP) per room is assumed. Each transmitter is assumed to be isotropic.

```
% Total number of transmitters, assuming one transmitter (tx) per room
numTx = prod(ScenarioParameters.BuildingLayout);

% Create transmitter sites with and isotropic antenna element
roomNames = strings(1,numTx);
for siteInd = 1:numTx
    roomNames(siteInd) = "Room " + siteInd;
end
txs = txsite('cartesian','Name',roomNames,...
    'TransmitterFrequency',PHYParameters.TransmitterFrequency, ...
    'TransmitterPower',10.^((PHYParameters.TxPower+PHYParameters.TxGain-30)/10),...
    'Antenna','isotropic');
```

Generate Receiver Sites

The receive site objects `rxsite` are generated and assigned names of the form 'Room#-STA#' for ease of reference. The Scenario parameter `NumRxPerRoom` is used to define how many receivers (STAs) are present in each room. Each receiver is assumed to be isotropic.

```
% Total number of receivers, assuming one transmitter (tx) per room
numRx = numTx*ScenarioParameters.NumRxPerRoom;

% Create receiver sites
roomNames = strings(1,numRx);
for siteInd = 1:numRx
    roomNames(siteInd) = "Room " + (mod(siteInd-1,numTx)+1) + "-" + ceil(siteInd/numTx);
end
rxs = rxsite('cartesian','Name',roomNames,'Antenna','isotropic');

% Receiver noise power in dBm
T = 290; % Temperature (Kelvin)
k = physconst('Boltzmann'); % Boltzmann constant
% Sample rate (Hz)
fs = wlanSampleRate(wlanHESUConfig('ChannelBandwidth',PHYParameters.ChannelBandwidth));
rxNoisePower = 10*log10(k*T*fs)+30+PHYParameters.NoiseFigure;
```

Part A - Align Long-Term Radio Characteristics

In this section, the simulation scenario, radio characteristics, and large-scale fading model are verified by performing the TGax Evaluation Methodology Box 1 Test 2 Downlink Only calibration test [3]. This test calculates the SINR at all receivers (STAs) assuming all transmitters (APs) are active. Multiple drops of transmitters and receivers are performed as part of the simulation. One active receiver is selected per drop.

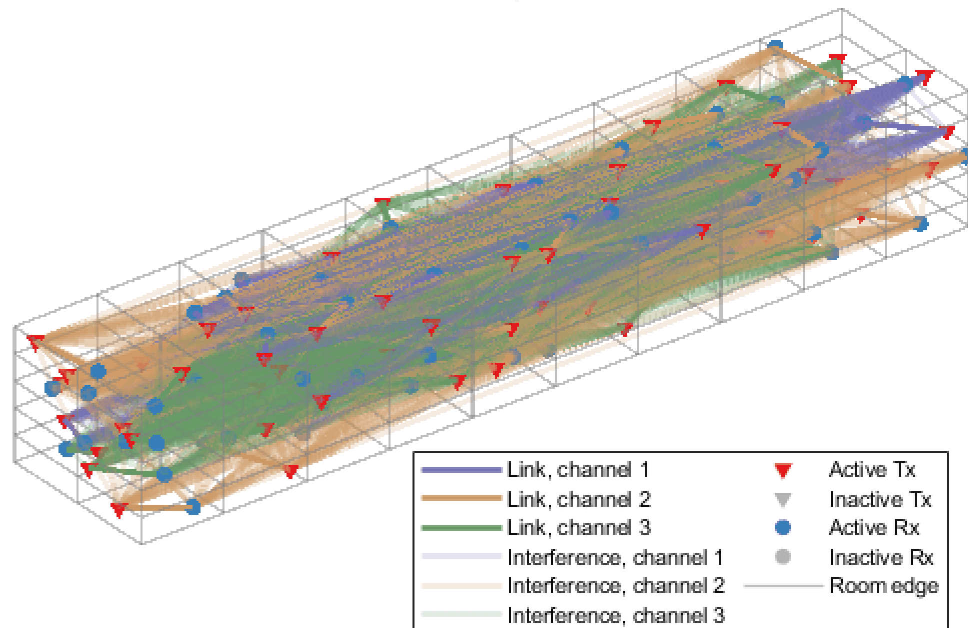
The SINR for each receiver is calculated and aggregated over all drops simulated to generate a CDF curve. This curve is compared with the calibration results provided in [4].

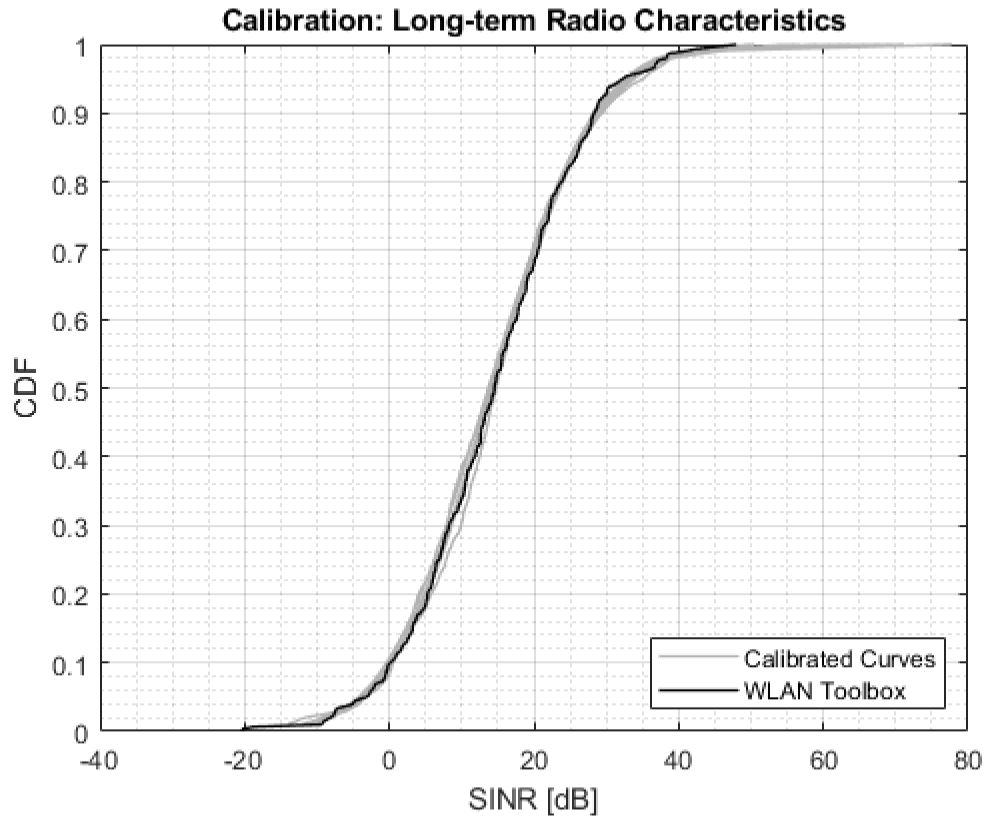
A plot showing the node positions, active links, and interfering links is generated per drop. Individual channels can be hidden and shown in the plot by clicking the corresponding legend entry.

```
seed = rng(6); % Seed random number generator and store state

if showScenarioPlot
    hGrid = tgaxBuildResidentialGrid(ScenarioParameters.RoomSize,ScenarioParameters.BuildingLayou
        numTx,numRx,MACParameters.NumChannels);
end
```

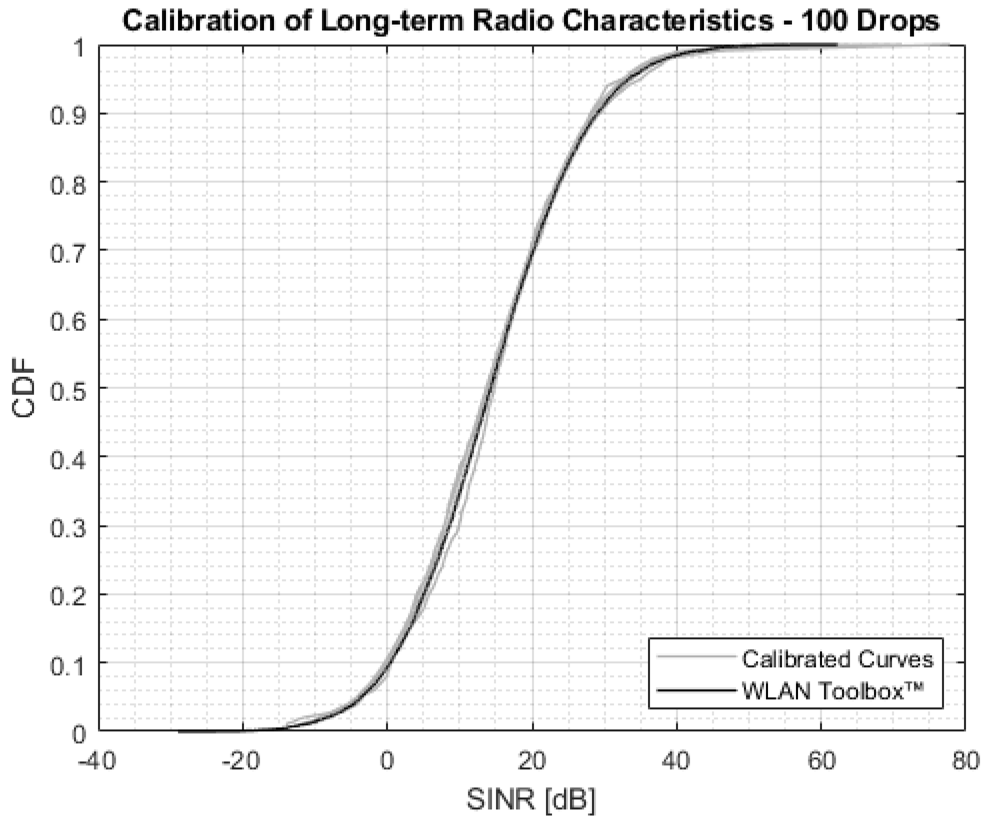
```
if calibrate

fprintf('Running calibration ...\n');

% Pre-allocate output
output = struct;
output.sinr = zeros(SimParameters.NumDrops,numTx); % for storing SINR values
for drop = 1:SimParameters.NumDrops
    % Drop receivers in each room
    [association,txChannels,rxChannels,txPositions,rxPositions] = tgaxDropNodes( ...
        txs,rxs,ScenarioParameters,MACParameters.NumChannels);

    % All transmitters active
    activeTx = true(numTx,1);

    % Only pick one receiver per Room
    rxAlloc = randi([1 ScenarioParameters.NumRxPerRoom],numTx,1);
    activeRx = reshape(rxAlloc==1:ScenarioParameters.NumRxPerRoom,[],1);

    % Generate propagation model
    propModel = TGaxResidential('roomSize',ScenarioParameters.RoomSize);

    % Get the index of the transmitter for each receiver
    tnum = repmat((1:numTx),1,numRx/numTx);

    % SINR calculation - loop over each non-overlapping channel
    activeChannels = unique(txChannels);
    for k = 1:numel(activeChannels)
        % Use kth non-overlapping channel
        tind = txChannels == activeChannels(k);
        rind = false(size(activeRx));
        rind(activeRx) = rxChannels(activeRx) == activeChannels(k);
        % Get the index of the transmitter of interest for each active receiver
        tsigind = tnum(rind);

        % Calculate SNR
        output.sinr(drop,tind) = sinr(rxs(rind),txs(tind),...
            'ReceiverGain',PHYParameters.RxGain,...
            'ReceiverNoisePower',rxNoisePower,...
            'PropagationModel',propModel,...
            'SignalSource',txs(tsigind));
    end

    % Plot nodes and links
    if showScenarioPlot % update plot data
        mask = txChannels==rxChannels';
        tgaxUpdatePlot(hGrid,txPositions,rxPositions,activeTx,activeRx,mask,txChannels,rxChannels
            sprintf('Box 1 Test 2 "downlink only" calibration, drop #%d/%d',drop,SimParameters.Nu
    end
end

% Plot the CDF of SINR and compare with calibration curves
tgaxCalibrationCDF(output.sinr,'SS1Box1Test2','Long-term Radio Characteristics');

fprintf('Calibration complete \n')

end
```

```
Running calibration ...
Calibration complete
```





This example simulates a small number of drops. Therefore, for a more meaningful comparison the number of drops simulated should be increased. The calibration result for 100 drops is shown below:



Part B - PHY-focused System-Level Simulation

In this section, the scenario and path-loss model calibrated in part A are used to perform a PHY-focused system-level simulation and determine the packet error rate for the network. This simulation is described as PHY-focused as the PHY is not abstracted and the MAC is simplified. Each active link is modeled using baseband transmitter and receiver processing. A very simple MAC assumes at each transmission event all transmitters (APs) wish to transmit, and one receiver (STA) per BSS is the recipient. A simple CCA algorithm is used to control channel access between transmitters as specified in Figure 4 of [3]. The CCA algorithm enables random transmitters if the signal power received from transmitters that have already been activated does not exceed the CCA threshold, `MACParameters.CCAThreshold`.

The received signal power from all possible interfering transmitters is calculated at each active receiver. If the received power from an interfering transmitter is above the noise floor of the receiver, then the link is modeled with full baseband transmitter and receiver processing. For each modeled link an HE single-user packet is generated and passed through a TGax Model-D NLOS stationary indoor channel model. At the receiver, the waveforms from the transmitter of interest and all interfering transmitters are scaled by the expected path-loss and combined to create a waveform containing the signal of interest plus interference. All waveforms are time aligned. The receiver performs synchronization, demodulation, and decoding to attempt to recover the payload. The decoded payload is compared to the PSDU transmitted in the BSS to determine if the packet has been recovered successfully.

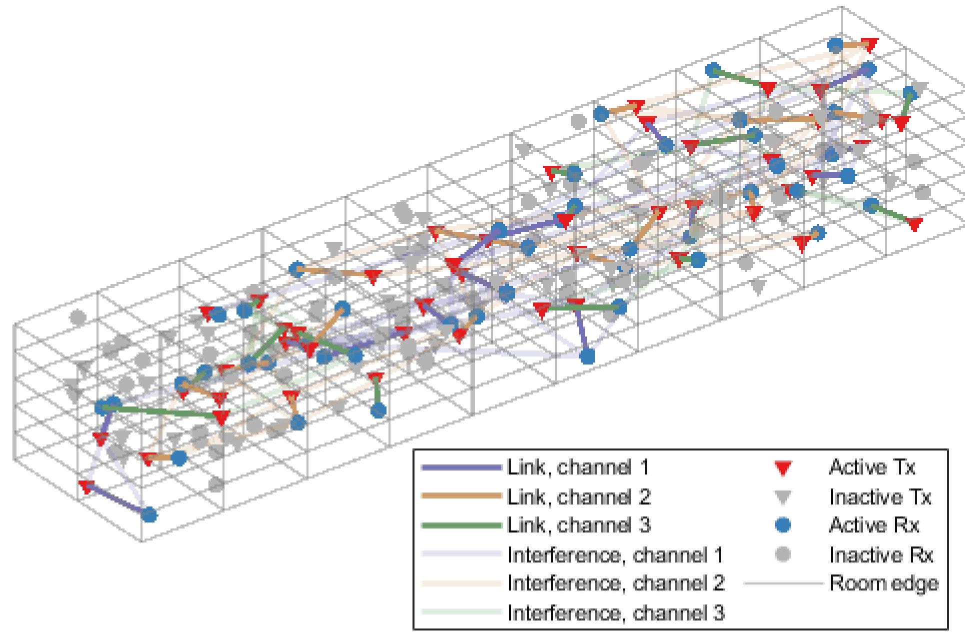
In this example the transmission and channel parameters are assumed to be the same for all nodes. The transmission configuration for all packets is one space-time stream, no space-time block coding and 16-QAM rate-1/2 (MCS 3).

```
if systemLevelSimulation

% Pre-allocate outputs
output.numPkts = zeros(numRx,1);
output.numPktErrors = zeros(numRx,1);
output.sinrMeas = nan(numRx,SimParameters.NumTxEventsPerDrop,SimParameters.NumDrops);
output.sinrEst = nan(numRx,SimParameters.NumTxEventsPerDrop,SimParameters.NumDrops);
output.pktErrorRate = 0;

% For each possible transmitter create a waveform configuration. In this
% example the link and radio parameters are the same for all nodes.
cfgHEBase = wlanHESUConfig;
cfgHEBase.ChannelBandwidth = PHYParameters.ChannelBandwidth; % Channel bandwidth
cfgHEBase.NumTransmitAntennas = PHYParameters.NumTxAntennas; % Number of transmit antennas
cfgHEBase.SpatialMapping = 'Fourier';  % Spatial mapping matrix
cfgHEBase.NumSpaceTimeStreams = 1;     % Number of space-time streams
cfgHEBase.GuardInterval = 0.8;         % Guard interval duration
cfgHEBase.HELTFType = 4;               % HE-LTF compression mode
cfgHEBase.APEPLength = 1e3;            % Payload length in bytes
cfgHEBase.ChannelCoding = 'LDPC';      % Channel coding
cfgHEBase.MCS = 3;                     % Modulation and coding scheme
cfgHE = cell(numTx,1);
for txidx = 1:numTx
    cfgHE{txidx} = cfgHEBase;
end

fprintf('Running System Level Simulation ...\n')
for drop = 1:SimParameters.NumDrops
    fprintf('  Running drop #%d/%d ...\n',drop,SimParameters.NumDrops);

    % Drop receivers in each room
    [association,txChannels,rxChannels,txPositions,rxPositions] = tgaxDropNodes( ...
        txs,rxs,ScenarioParameters,MACParameters.NumChannels);

    % Generate propagation model
    propModel = TGaxResidential('roomSize',ScenarioParameters.RoomSize);

    % Calculate signal strength for all links
    signalStrength = sigstrength(rxs,txs,propModel,'Type','power',...
        'ReceiverGain',PHYParameters.RxGain); % all signal strengths in dBm

    % Threshold signals below noise level to reduce simulation time
    signalStrength(signalStrength < rxNoisePower) = -Inf;

    % Threshold signals that are not on same non-overlapping channel
    signalStrength(~(txChannels == rxChannels')) = -Inf;

    % Mask the transmitter-receiver links that are non-negligible to
    % simulate and get the linear indices
    nonnegligibleMask = signalStrength > -Inf;

    % Reset the non-negligible channels to create a new realization for the
    % current drop
    nonnegligibleIdx = find(nonnegligibleMask)';
```

```
% For each possible active link in a drop create a channel
% configuration. In this example the link and radio parameters are the
% same for all nodes.
tgaxChan = cell(numel(nonnegligibleIdx),1);
for i = 1:numel(nonnegligibleIdx)
    % Index of transmitter for a given link
    txIdx = mod(nonnegligibleIdx(i)-1,numTx)+1;
    % Channel configuration. The channel realization for each link is
    % different as the global random stream is used.
    tgaxChanBase = wlanTGaxChannel;
    tgaxChanBase.DelayProfile = 'Model-D';
    tgaxChanBase.NumTransmitAntennas = cfgHE{txIdx}.NumTransmitAntennas;
    tgaxChanBase.NumReceiveAntennas = PHYParameters.NumRxAntennas;
    tgaxChanBase.TransmitReceiveDistance = 10; % Distance in meters for NLOS
    tgaxChanBase.ChannelBandwidth = cfgHE{txIdx}.ChannelBandwidth;
    tgaxChanBase.LargeScaleFadingEffect = 'None';
    tgaxChanBase.EnvironmentalSpeed = 0; % m/s, stationary
    tgaxChanBase.SampleRate = fs;
    tgaxChanBase.NormalizeChannelOutputs = false;

    % Store in cell array and reset the channel to generate a new
    % response
    tgaxChan{i} = tgaxChanBase;
    reset(tgaxChan{i});
end

for txevent = 1:SimParameters.NumTxEventsPerDrop
    fprintf('    Running transmission event #%d/%d ...\n',txevent,SimParameters.NumTxEventsPe

    % Determine active transmitters and receivers with Clear Channel Assessment
    [activeTx,activeRx] = tgaxCCA(signalStrength,MACParameters.CCALevel);

    % Plot scenario and links
    if showScenarioPlot
        tgaxUpdatePlot(hGrid,txPositions,rxPositions,activeTx,activeRx,nonnegligibleMask,txCh
            sprintf('PHY System-Level Simulation, Drop #%d/%d, Transmission Event #%d/%d', .
            drop,SimParameters.NumDrops,txevent,SimParameters.NumTxEventsPerDrop));
    end

    % Extract elements for active links using activeTx and activeRx
    cfgHEActive = cfgHE(activeTx);
    associationActive = association(activeTx,activeRx);
    nonnegligibleMaskActive = nonnegligibleMask(activeTx,activeRx);
    signalStrengthActive = signalStrength(activeTx,activeRx);

    % Create array containing active channels
    tgaxChanActive = cell(size(associationActive));
    matchIdx = nonnegligibleIdx==find(activeTx&activeRx');
    tgaxChanActive(nonnegligibleMaskActive) = tgaxChan(any(matchIdx,1));

    % Generate a waveform for each non-negligible active link and
    % combine waveforms for each receiver
    [rxWavs,txPSDUActive,signalPower,interfPower] = tgaxGenerateRxWaveforms( ...
        cfgHEActive,tgaxChanActive,nonnegligibleMaskActive,signalStrengthActive,associationAc

    % Run PHY link simulation for each link and determine if the packet
    % has been decoded successfully. The estimated interference power
```

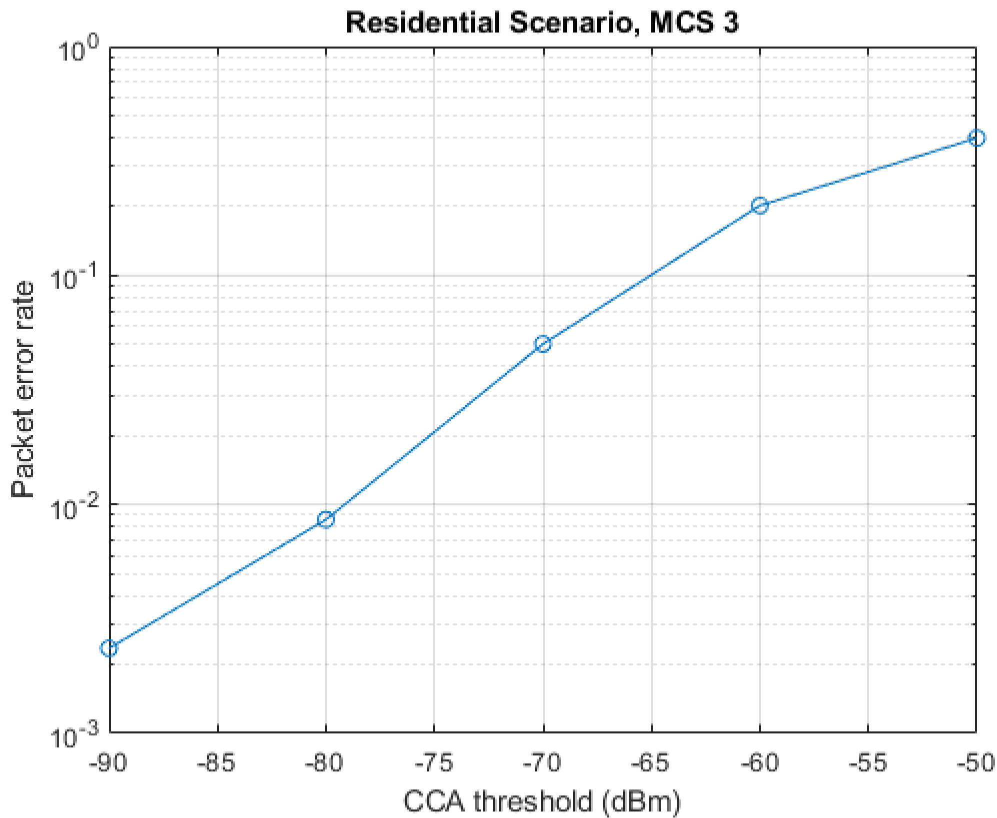
```
        % is passed to the receiver in the place of an interference power
        % measurement algorithm.
        numActiveRxs = sum(activeRx);
        pktError = false(numActiveRxs,1);
        sinrMeas = nan(numActiveRxs,1);
        for rxIdx = 1:numActiveRxs
            [pktError(rxIdx),sinrMeas(rxIdx)] = tgaxModelPHYLink( ...
                rxWavs{rxIdx},cfgHEActive{rxIdx},rxNoisePower,interfPower(rxIdx),txPSDUActive{rx
        end

        % Store output for active receivers
        output.numPktErrors(activeRx) = output.numPktErrors(activeRx)+pktError;
        output.numPkts(activeRx) = output.numPkts(activeRx) + 1;
        output.sinrMeas(activeRx,txevent,drop) = sinrMeas;

        % Calculate the expected SINR at each receiver
        sinrEst = 10*log10(signalPower./(interfPower+10^((rxNoisePower-30)/10)));
        output.sinrEst(activeRx,txevent,drop) = sinrEst;
    end
end

% Calculate average packet error rate
output.pktErrorRate = sum(output.numPktErrors)/sum(output.numPkts);

disp('Simulation complete')
disp(['Average packet error rate for transmitters: ' num2str(output.pktErrorRate)]);

end

rng(seed); % Restore random state
```

```
Running System Level Simulation ...
  Running drop #1/3 ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
  Running drop #2/3 ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
  Running drop #3/3 ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
Simulation complete
Average packet error rate for transmitters: 0.054152
```



Further Exploration

The PHY-focused system-level simulation demonstrated in this example can be used to explore the impact of PHY-level parameters on system performance. For example, the plot below shows the network average packet error rate for different values of CCA threshold for 50 drops and 2 transmission events per drop.



Appendix

This example uses the following helper functions:

- tgaxBuildResidentialGrid.m
- tgaxCalibrationCDF.m
- tgaxCCA.m
- tgaxDropNodes.m
- tgaxGenerateRxWaveforms.m
- tgaxModelPHYLink.m
- TGaxResidential.m
- tgaxUpdatePlot.m

Selected Bibliography

1. IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.
2. IEEE 802.11-14/0980r16 - TGax Simulation Scenarios.
3. IEEE 802.11-14/0571r12 - 11ax Evaluation Methodology.

4 IEEE 802.11-14/0800r30 - Box 1 and Box 2 Calibration Results.

See Also

More About

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “802.11ax Multinode System-Level Simulation of Residential Scenario” on page 7-40

Physical Layer Abstraction for System-Level Simulation

This example demonstrates IEEE® 802.11ax™ physical layer abstraction for system-level simulation. A link quality model and link performance model based on the TGax evaluation methodology are presented and validated by comparing with published results.

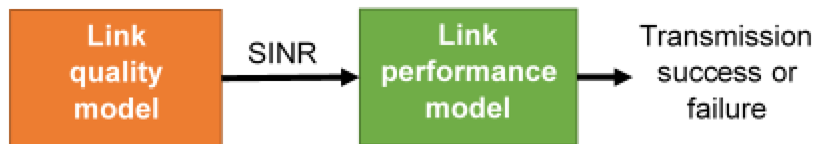
Introduction

Modeling the full physical layer processing at each transmitter and receiver when simulating large networks is computationally expensive. Physical layer abstraction, or link-to-system mapping is a method to run simulations in a timely manner by accurately predicting the performance of a link in a computationally efficient way.

This example demonstrates physical layer abstraction for the data portion of an 802.11ax [1 on page 7-104] packet based on the TGax evaluation methodology [2 on page 7-104].

There are two parts to the physical layer abstraction model [3 on page 7-104, 4 on page 7-104]:

- **The link quality model** calculates the post-equalizer signal to interference and noise ratio (SINR) per subcarrier. For a receiver, this is based on the location and transmission characteristics of the transmitter of interest, and interfering transmissions, and the impact of large- and small-scale fading.
- **The link performance model** predicts the instantaneous packet error rate (PER), and therefore transmission success of an individual packet, given the SINR per subcarrier and coding parameters used for the transmission.



The example is split into two parts:

- Part A on page 7-90 demonstrates the link quality model used to obtain the SINR per subcarrier and validates it by comparing the results for a residential scenario as per the box 2 tests in the TGax evaluation methodology. The objective of box 2 tests is to align the distribution of small and large scale fading channels with MIMO configurations of TGax contributors.
- Part B on page 7-95 demonstrates the link performance model used to estimate the PER, and compares the result of using the abstraction against a link-level simulation with a fading TGax channel model as per the box 0 tests in the TGax evaluation methodology. The objective of box 0 tests is to align PHY abstractions of TGax contributors.

Part A - Link Quality Model

The link quality model implements the box 2 SINR equation from the TGax Evaluation Methodology. The multiple-input-multiple-output (MIMO) SINR per subcarrier (index m) and spatial stream (index j) between the transmitter and receiver of interest is given by

$$SINR_{RX}^{TX}(m,j) = \frac{S_{RX}^{TX}(m,j)}{Is_{RX}^{TX}(m,j) + Io_{RX}^{TX}(m,j) + N(m,j)}.$$

The SINR takes into account the path-loss and fading channels between all transmitters and the receiver, and precoding applied at the transmitters. The power of the signal of interest is given by

$$S_{RX}^{TX}(m,j) = P_{RX}^{TX}\left|[\mathbf{T}_{RX}(m)]_j^H \mathbf{H}_{RX}^{TX}(m)\left[\mathbf{W}^{TX}(m)\right]_j\right|^2,$$

where P_{RX}^{TX} is the received power of the signal of interest,$\mathbf{T}_{RX}$ is the linear receiver filter, $\mathbf{H}_{RX}^{TX}$is the channel matrix between the transmitter and receiver of interest, and $\mathbf{W}^{TX}$ is the precoding matrix applied at the transmitter.

The power of intra-user interference is given by

$$Is_{RX}^{TX}(m,j) = P_{RX}^{TX}\left\|[\mathbf{T}_{RX}(m)]_j^H \mathbf{H}_{\mathrm{RX}}^{\mathrm{TX}}(m)\mathbf{W}^{TX}(m)\right\|^2 - S_{RX}^{TX}(m,j).$$

The power of inter-user interference is given by

$$Io_{RX}^{TX}(m,j) = \sum_k \sum_{i\,\in\,\Omega(k)} P_{RX}^{TX}\left\|[\mathbf{T}_{\mathrm{RX}}(m)]_j^H \mathbf{H}_{RX}^{\mathrm{TX}_i}(m)\mathbf{W}^{\mathrm{TX}_i}(m)\right\|^2,$$

where $\Omega(k)$ is the set of interfering transmitters in the kth basic service set (BSS)

The noise power is given by

$$N(m,j) = \left\|[\mathbf{T}_{RX}(m)]_j\right\|^2 N_0,$$

where N_0 is the noise power spectral density.

Generate a Channel Matrix per Subcarrier

The link quality model requires a channel matrix per subcarrier. Calculate the channel matrix from the path gains returned from the fading channel model `wlanTGaxChannel` by using the `helperPerfectChannelEstimate()` helper function. Efficiently generate path gains by setting the `ChannelFiltering` property of `wlanTGaxChannel` to `false`.

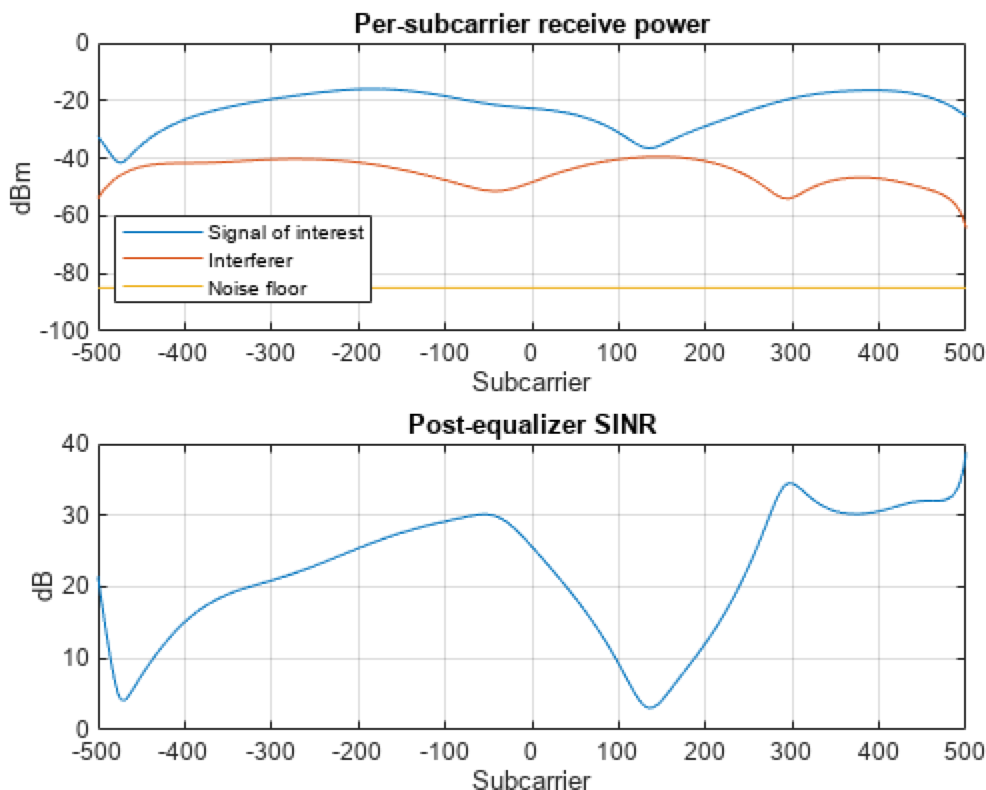
```
sprev = rng('default'); % Seed random number generator and store previous state

% Get an HE OFDM configuration: 80 MHz channel bandwidth, 3.2 us guard
% interval
ofdmInfo = wlanHEOFDMInfo('HE-Data','CBW80',3.2);
k = ofdmInfo.ActiveFrequencyIndices;

% Configure channel to return path gains for one OFDM symbol
tgax = wlanTGaxChannel;
tgax.ChannelBandwidth = 'CBW80';
tgax.SampleRate = 80e6; % MHz
tgax.ChannelFiltering = false;
tgax.NumSamples = ofdmInfo.FFTLength+ofdmInfo.CPLength;

% Generate channel matrix per subcarrier for signal of interest
pathGains = tgax();    % Get path gains
chanInfo = info(tgax); % Get channel info for filter coefficients
chanFilter = chanInfo.ChannelFilterCoefficients;
Hsoi = helperPerfectChannelEstimate(pathGains,chanFilter, ...
    ofdmInfo.FFTLength,ofdmInfo.CPLength,ofdmInfo.ActiveFFTIndices);
```

```
% Generate channel matrix per subcarrier for interfering signal
reset(tgax); % Get a new channel realization
pathGains = tgax();
Hint = helperPerfectChannelEstimate(pathGains,chanFilter, ...
    ofdmInfo.FFTLength,ofdmInfo.CPLength,ofdmInfo.ActiveFFTIndices);
```

SINR Calculation

Calculate and visualize the post-equalizer SINR per subcarrier with the `calculateSINR` and `plotSINR` helper functions.

```
Psoi = -20; % Signal of interest received power (dBm)
Pint = -45; % Interfering signal received power (dBm)
N0 = -85;   % Noise power (dBm)
W = ones(ofdmInfo.NumTones,1); % Precoding matrix (assume no precoding)

sinr = calculateSINR(Hsoi,db2pow(Psoi-30),W,db2pow(N0-30),{Hint},db2pow(Pint-30),{W});

plotSINR(sinr,Hsoi,Psoi,Hint,Pint,N0,k);
```

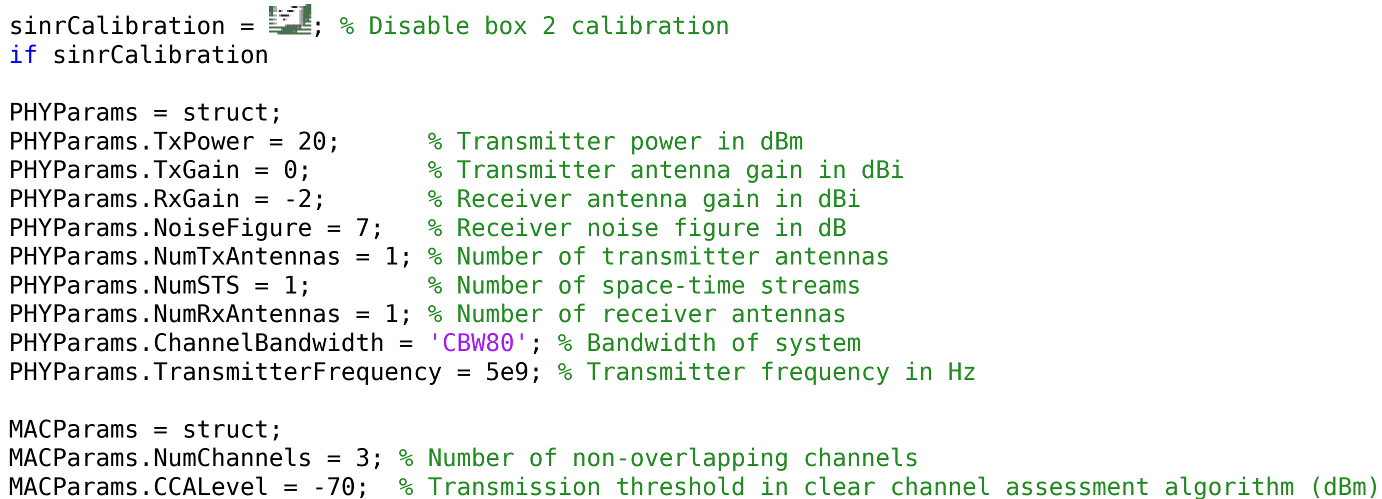


TGax Evaluation Methodology Box 2 - Verify SINR Calibration

This section validates the SINR calculation by comparing the cumulative density function (CDF) of per-subcarrier SINRs with calibration results provided by the TGax working group. We compare the SINR calculation with results published by TGax [5 on page 7-104] for box 2, test 3: "downlink transmission per basic channel access rule" for the residential scenario.

For more information about the scenario, and for the results of long-term SINR calibration see the "802.11ax PHY-Focused System-Level Simulation" on page 7-77 example.

The major simulation parameters are defined as either belonging to Physical Layer (PHY), Medium Access Control Layer (MAC), scenario, or simulation. In this example the PHY and MAC parameters are assumed to be the same for all nodes.



```
sinrCalibration = ; % Disable box 2 calibration
if sinrCalibration

PHYParams = struct;
PHYParams.TxPower = 20;      % Transmitter power in dBm
PHYParams.TxGain = 0;        % Transmitter antenna gain in dBi
PHYParams.RxGain = -2;       % Receiver antenna gain in dBi
PHYParams.NoiseFigure = 7;   % Receiver noise figure in dB
PHYParams.NumTxAntennas = 1; % Number of transmitter antennas
PHYParams.NumSTS = 1;        % Number of space-time streams
PHYParams.NumRxAntennas = 1; % Number of receiver antennas
PHYParams.ChannelBandwidth = 'CBW80'; % Bandwidth of system
PHYParams.TransmitterFrequency = 5e9; % Transmitter frequency in Hz

MACParams = struct;
MACParams.NumChannels = 3; % Number of non-overlapping channels
MACParams.CCALevel = -70;  % Transmission threshold in clear channel assessment algorithm (dBm)
```

The scenario parameters define the size and layout of the residential building as per [6 on page 7-104]. Note only one receiver (STA) can be active at any given time.

```
ScenarioParams = struct;
ScenarioParams.BuildingLayout = [10 2 5]; % Number of rooms in [x,y,z] directions
ScenarioParams.RoomSize = [10 10 3];      % Size of each room in metres [x,y,z]
ScenarioParams.NumRxPerRoom = 1;          % Number of receivers per room.
```

The `NumDrops` and `NumTxEventsPerDrop` parameters control the length of the simulation. In this example, these parameters are configured for a short simulation. A 'drop' randomly places transmitters and receivers within the scenario and selects the channel for a BSS. A 'transmission' event randomly selects transmitters and receivers for transmissions according to the basic clear channel assessment (CCA) rules defined in the evaluation methodology.

```
SimParams = struct;
SimParams.Test = 3; % Downlink transmission per basic channel access rule
SimParams.NumDrops = 3;
SimParams.NumTxEventsPerDrop = 2;
```

The function `box2Simulation` runs the simulation by performing these steps:

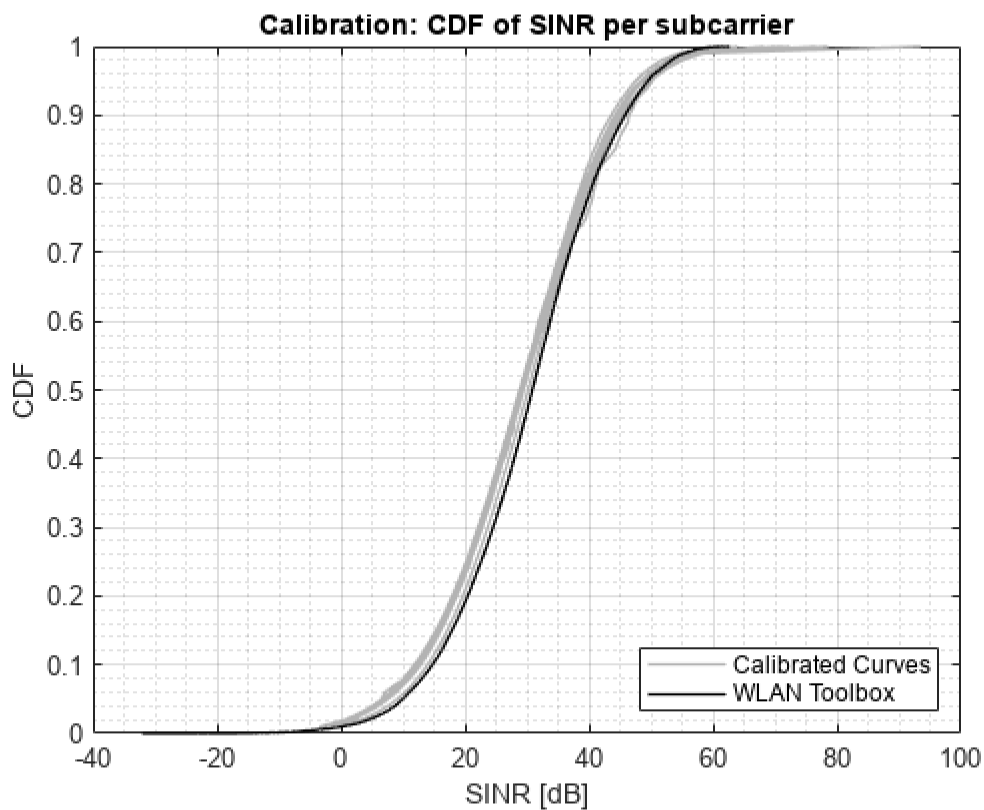
1. Randomly drop transmitters (APs) and receivers (STAs) within the scenario.
2. Calculate the large-scale path loss and generate frequency-selective TGax fading channels for all non-negligible links.
3. For each transmission event, determine active transmitters and receivers based on CCA rules.
4. Calculate and return the SINR per subcarrier and the effective SINR for each active receiver as per box 2, test 3 in the TGax evaluation methodology.

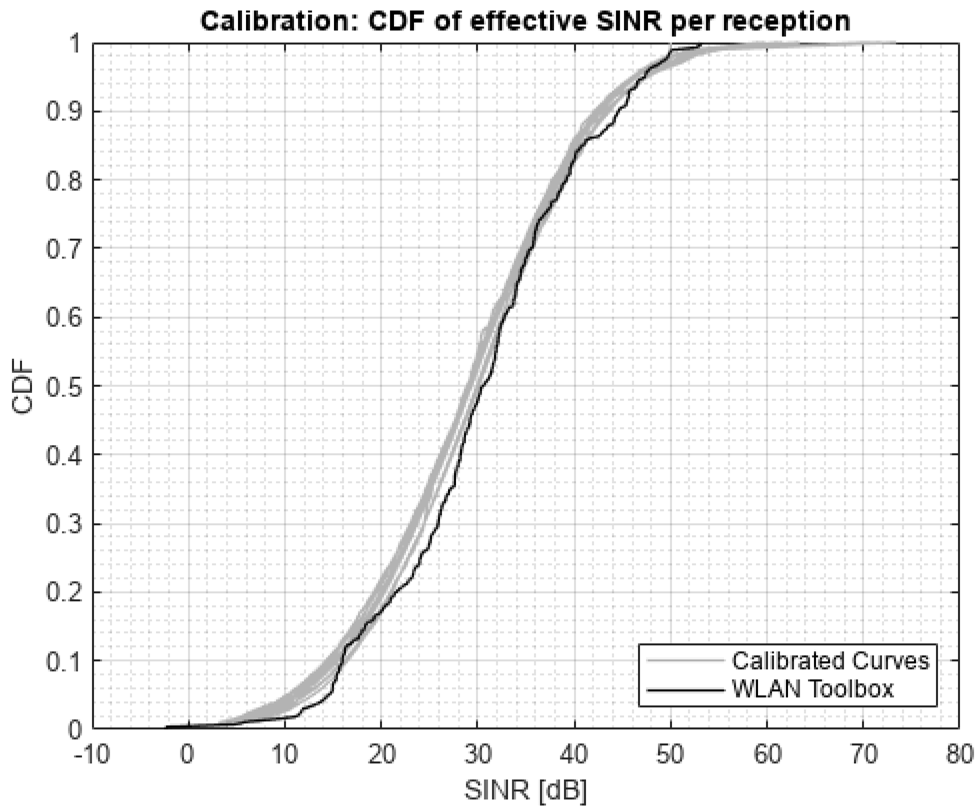
```
box2Results = box2Simulation(PHYParams,MACParams,ScenarioParams,SimParams);
```

Plot the CDF of the SINR per subcarrier and effective SINR (as defined in box 2, test 3) against submitted calibration results.

```
tgaxCalibrationCDF(box2Results.sinr(:), ...
    ['SS1Box2Test' num2str(SimParams.Test) 'A'],'CDF of SINR per subcarrier');
tgaxCalibrationCDF(box2Results.sinrEff(:), ...
    ['SS1Box2Test' num2str(SimParams.Test) 'B'],'CDF of effective SINR per reception');

end
```

```
  Running drop #1/3 ...
    Generating 3518 fading channel realizations ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
  Running drop #2/3 ...
    Generating 3366 fading channel realizations ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
  Running drop #3/3 ...
    Generating 3750 fading channel realizations ...
    Running transmission event #1/2 ...
    Running transmission event #2/2 ...
```



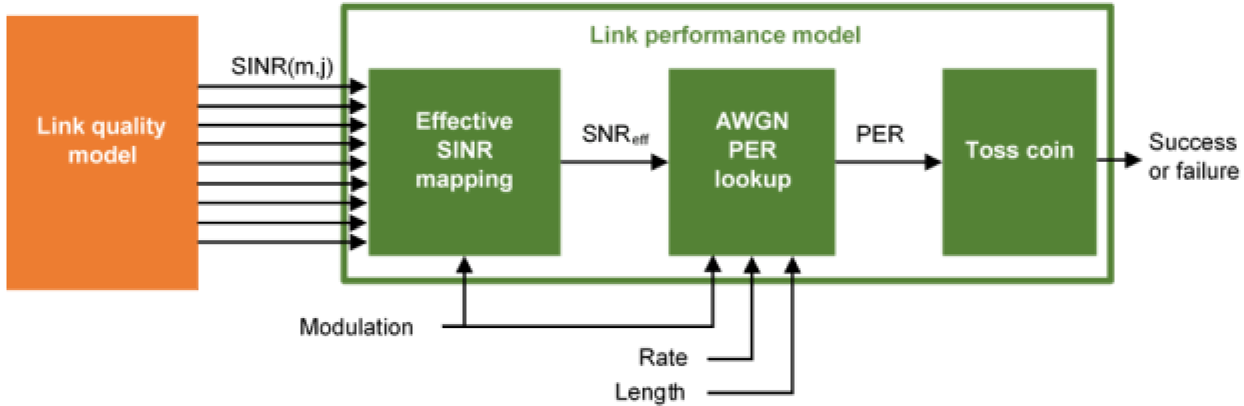


Increase the number of drops for a more accurate comparison.

Part B - Link Performance Model

The link performance model predicts the instantaneous PER given the SINR per subcarrier calculated in Part A on page 7-90 and coding parameters used for the transmission.

Effective SINR mapping and averaging is used to compress the post-equalizer SINR per subcarrier into a single effective SNR. The effective SNR is the SNR that provides an equivalent PER performance with an additive white Gaussian noise (AWGN) channel as with the fading channel. A pre-computed lookup table, generated with WLAN Toolbox™, provides the PER for an SNR under an AWGN channel for a given channel coding, modulation scheme, and coding rate. Once the PER is obtained, a random variable determines whether the packet has been received in error.



The TGax evaluation methodology PER estimation procedure is used in this example considering a single interference event.

The effective SINR is calculated using the received bit mutual information rate (RBIR) mapping function;

$$SINR_{eff} = \alpha\phi^{-1}\left\{\frac{1}{N_{SS}N_{SC}}\sum_{j=1}^{N_{SS}}\sum_{m=1}^{N_{SC}}\phi\left(\frac{SINR(m,j)}{\beta}, M\right), M\right\}.$$

- $\phi(x, M)$ is the RBIR mapping function, which transforms the SINR of each subcarrier to an "information measure" for the modulation scheme M. The RBIR mapping function for BPSK, QPSK, 16QAM, 64QAM and 256QAM is provided in [7 on page 7-104].
- $\phi^{-1}(x, M)$ is the inverse RBIR mapping function, which transforms an "information measure" back to the SNR domain.
- N_{SS} is the number of spatial-streams.
- N_{SC} is the number of subcarriers.
- $SINR$ is the post-equalizer SINR of the mth subcarrier and jth spatial-stream.
- α and β are tuning parameters. The TGax evaluation methodology assumes no tuning therefore in this example we assume these are set to 1.

The PER for a reference data length PER_{PL_0} is obtained by looking up the appropriate AWGN table, $\overline{\quad}LUT$, given the modulation and coding scheme (MCS), channel coding scheme, and reference data length (PL_0)

$$PER_{PL_0} = \overline{\quad}LUT(SNR_{eff}; MCS,\ coding\ scheme,\ PL_0),$$

where the reference data length depends on the channel coding and data length for the transmission PL.

$$PL_0 = \begin{cases} 32\ bytes, & \text{BCC}\ and\ PL < 400\ bytes \\ 1458\ bytes, & \text{BCC}\ and\ PL \geq 400\ bytes \\ 1458\ bytes, & \text{LDPC} \end{cases}.$$

The final estimated PER is then adjusted for the data length:

$$PER_{PL} = 1 - \left(1 - PER_{PL_0}\right)^{\frac{PL}{PL_0}}.$$

The described method assumes the SINR is constant for the duration of the packet. The TGax evaluation methodology describes techniques to deal with time-varying interference and estimate the error rate of media access control protocol data units (MPDUs) within an aggregate MPDU (A-MPDU).

Calculate Effective SINR

Calculate the effective SINR and the PER using the `tgaxLinkPerformanceModel` example helper object.

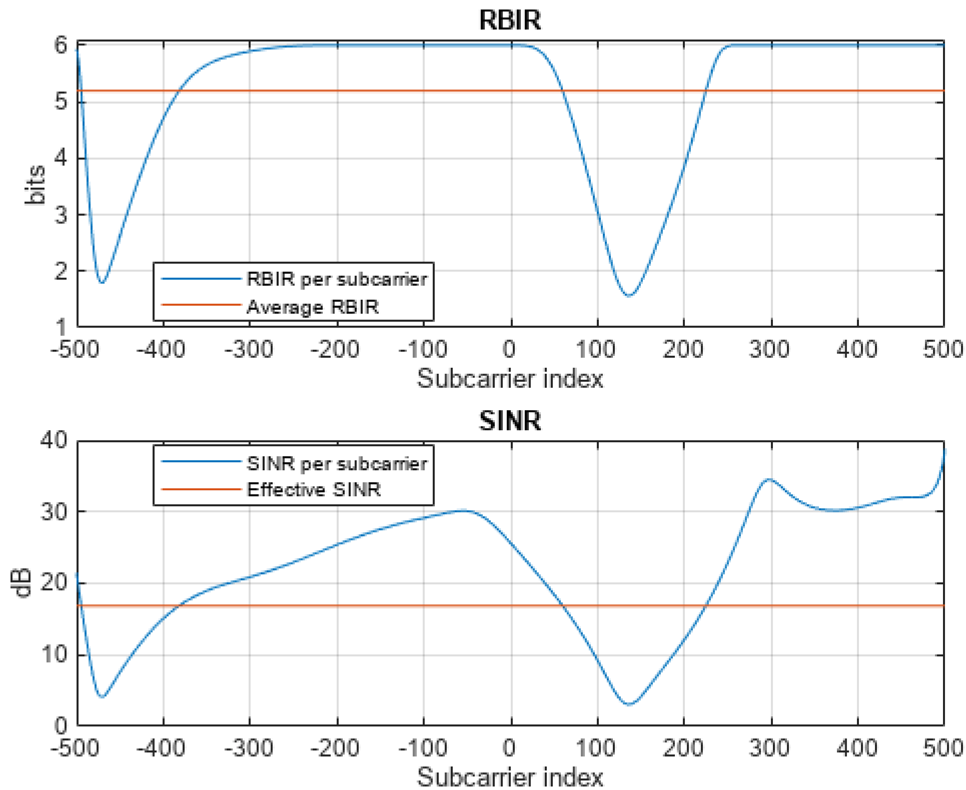
```
Abstraction = tgaxLinkPerformanceModel;
```

The `effectiveSINR` method calculates the effective SINR given the modulation scheme and post equalizer SINR per subcarrier and spatial stream.

```
format = HE SU ;
mcs = 6 ; % MCS 6 is 64-QAM
[snreff,rbir_sc,rbir_av] = effectiveSINR(Abstraction,sinr,format,mcs);
```

The RBIR (information measure) per subcarrier obtained by mapping the SINR per subcarrier, and the average RBIR are shown in the first figure subplot. The effective SINR per subcarrier is obtained by inverse mapping the average RBIR and is shown in the second subplot.

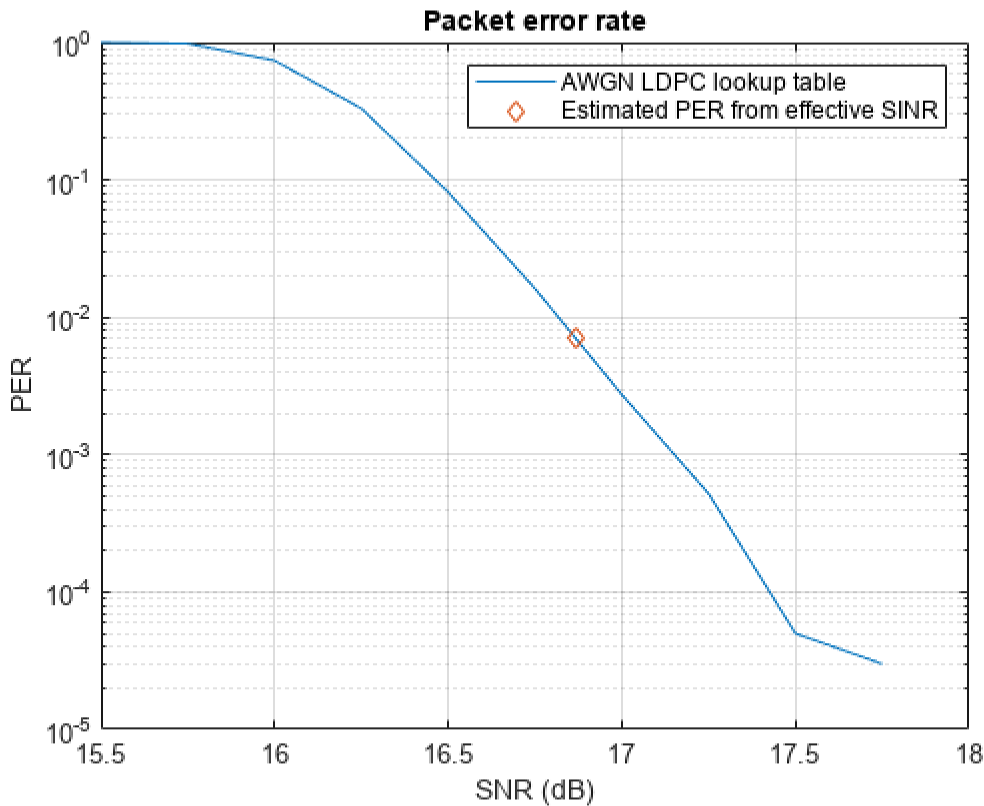
```
plotRBIR(sinr,snreff,rbir_av,rbir_sc,k);
```



Estimate Packet Error Rate

Given the effective SNR, estimate the PER by linearly interpolating and extrapolating a pre-computed AWGN link-level curve in the logarithmic domain, and adjusting for the data length. The `estimatePER` method returns the final PER, `per`, and the AWGN lookup table used, `lut` .

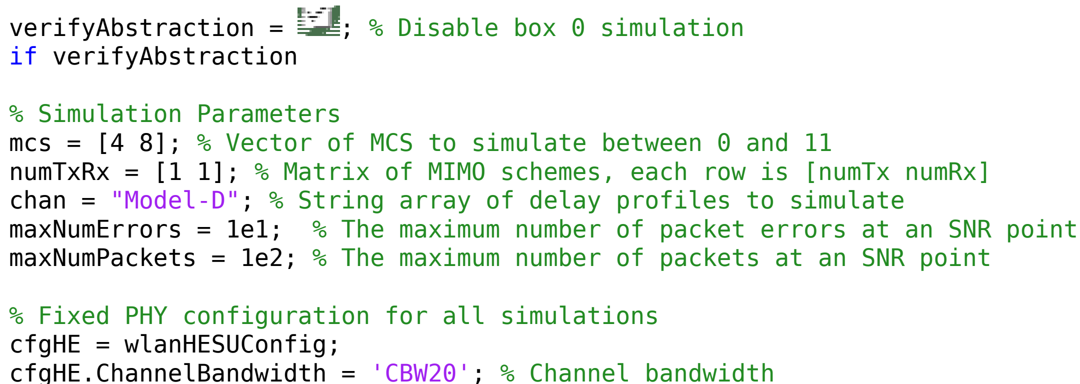
```
channelCoding = LDPC ;
dataLength = 1458 ; % Bytes
[per,~,~,lut] = estimatePER(Abstraction,snreff,format,mcs,channelCoding,dataLength);

% Plot the AWGN lookup table and the estimated PER
figure;
semilogy(lut(:,1),lut(:,2));
grid on
hold on
plot(snreff,per,'d');
legend('AWGN LDPC lookup table','Estimated PER from effective SINR')
title('Packet error rate')
xlabel('SNR (dB)')
ylabel('PER')
```



TGax Evaluation Methodology Box 0 - Verify Effective SNR vs PER Performance

To verify the entire physical-layer abstraction method, the PER from a link-level simulation is compared with the PER estimates using the abstraction. This follows steps 2 and 3 of box 0 testing in the TGax Evaluation Methodology. An 802.11ax single-user link is modeled with perfect synchronization, channel estimation, and no impairments apart from a fading TGax channel model and AWGN. Only errors within data portion of a packet are considered.

In this example the SNRs to simulate are selected based on the MCS, number of transmit and receive antennas, and channel model for the given PHY configuration. The number of space-time streams is assumed to equal the number of transmit antennas. The simulation is configured for a short run; for more meaningful results you should increase the number of packets to simulate.



```
verifyAbstraction = ; % Disable box 0 simulation
if verifyAbstraction

% Simulation Parameters
mcs = [4 8]; % Vector of MCS to simulate between 0 and 11
numTxRx = [1 1]; % Matrix of MIMO schemes, each row is [numTx numRx]
chan = "Model-D"; % String array of delay profiles to simulate
maxNumErrors = 1e1;  % The maximum number of packet errors at an SNR point
maxNumPackets = 1e2; % The maximum number of packets at an SNR point

% Fixed PHY configuration for all simulations
cfgHE = wlanHESUConfig;
cfgHE.ChannelBandwidth = 'CBW20'; % Channel bandwidth
```

```
cfgHE.APEPLength = 1000;          % Payload length in bytes
cfgHE.ChannelCoding = 'LDPC';     % Channel coding

% Generate a structure array of simulation configurations. Each element is
% one SNR point to simulate.
simParams = getBox0SimParams(chan,numTxRx,mcs,cfgHE,maxNumErrors,maxNumPackets);

% Simulate each configuration
results = cell(1,numel(simParams));
% parfor isim = 1:numel(simParams) % Use 'parfor' to speed up the simulation
for isim = 1:numel(simParams)
    results{isim} = box0Simulation(simParams(isim));
end
```

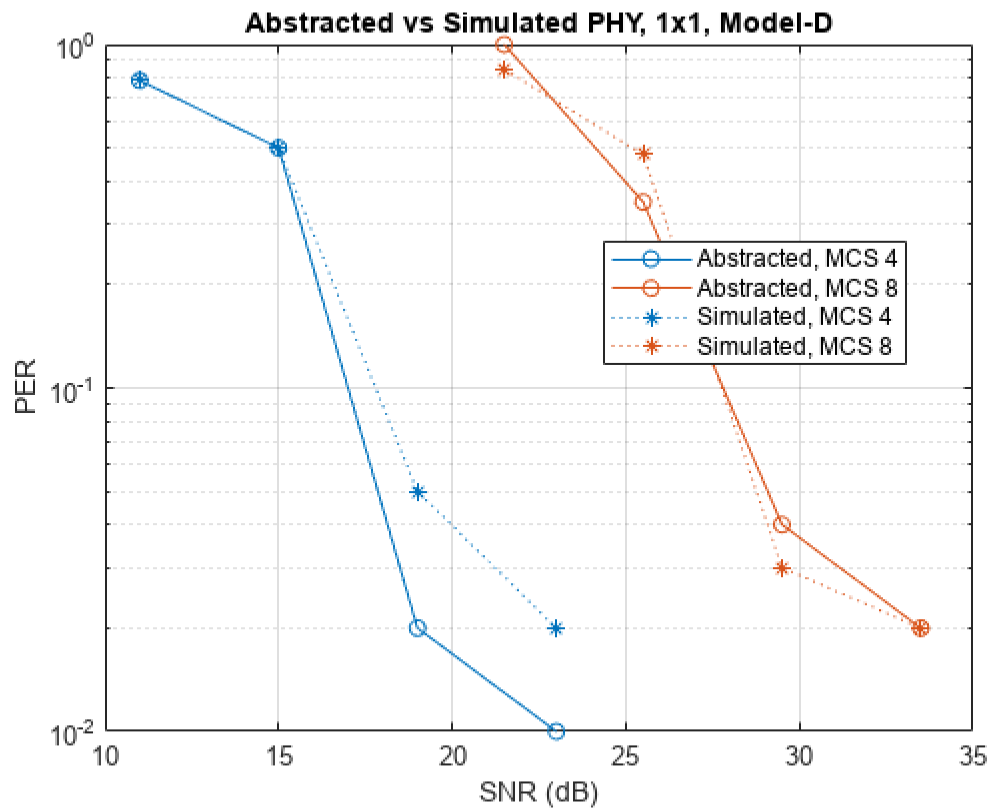
The suitability of the abstraction is determined by comparing the PER calculated by link-level simulation and abstraction. The first figure compares the PERs at each SNR simulated.

```
plotPERvsSNR(simParams,results);
```

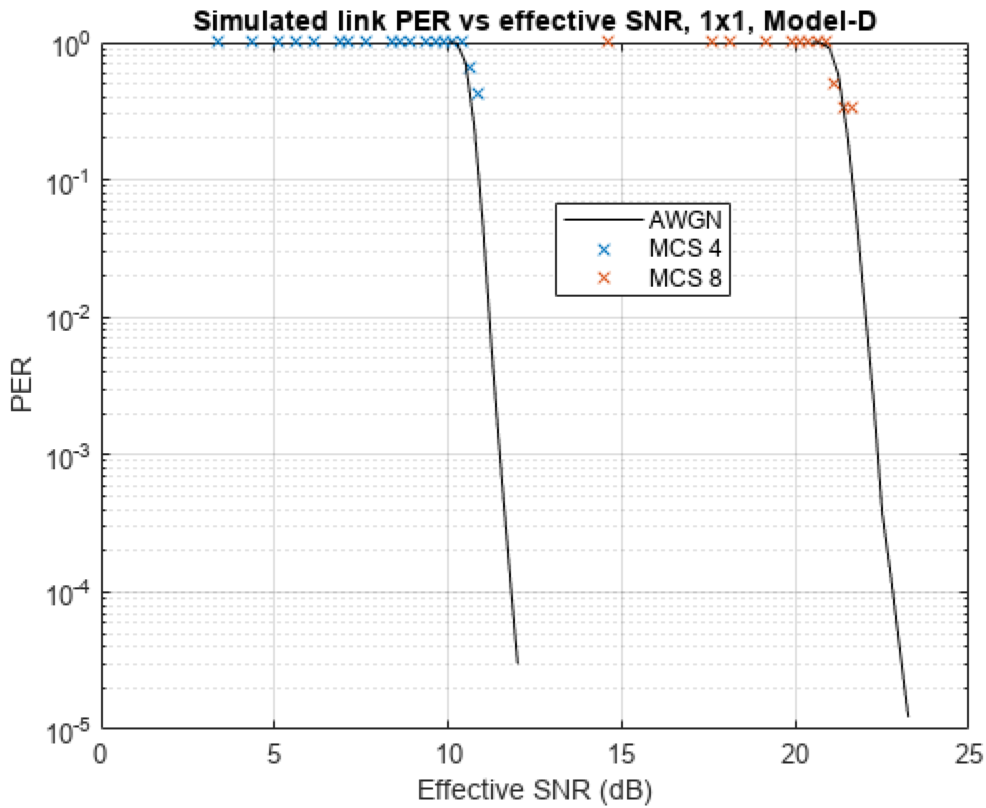
The second figure compares the number of successfully decoded link-level simulation packets with an effective SNR against the reference AWGN curve. If the abstraction is successful the PER should follow the AWGN curve.

```
plotPERvsEffectiveSNR(simParams,results);
end
```

```
Model-D 1-by-1, MCS 4, SNR 11 completed after 14 packets, PER:0.78571
Model-D 1-by-1, MCS 4, SNR 15 completed after 22 packets, PER:0.5
Model-D 1-by-1, MCS 4, SNR 19 completed after 100 packets, PER:0.05
Model-D 1-by-1, MCS 4, SNR 23 completed after 100 packets, PER:0.02
Model-D 1-by-1, MCS 4, SNR 27 completed after 100 packets, PER:0
Model-D 1-by-1, MCS 8, SNR 21.5 completed after 13 packets, PER:0.84615
Model-D 1-by-1, MCS 8, SNR 25.5 completed after 23 packets, PER:0.47826
Model-D 1-by-1, MCS 8, SNR 29.5 completed after 100 packets, PER:0.03
Model-D 1-by-1, MCS 8, SNR 33.5 completed after 100 packets, PER:0.02
Model-D 1-by-1, MCS 8, SNR 37.5 completed after 100 packets, PER:0
```

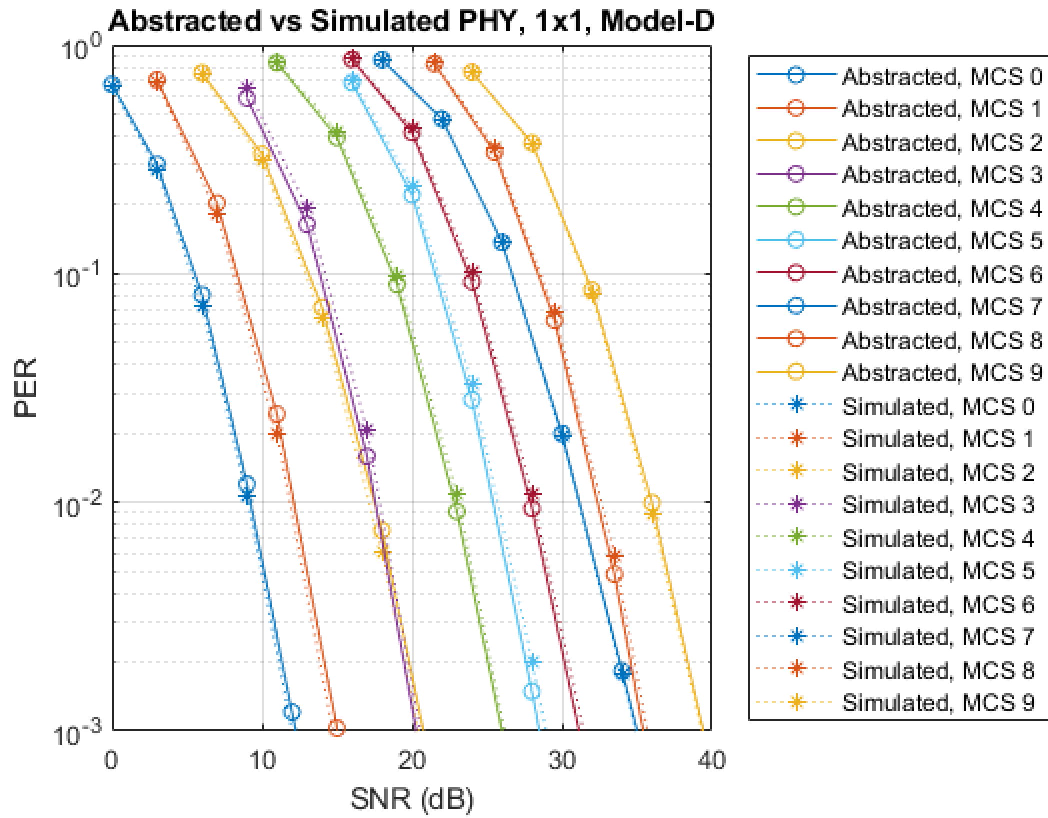


Abstracted vs Simulated PHY, 1x1, Model-D
PER
SNR (dB)
Abstracted, MCS 4
Abstracted, MCS 8
Simulated, MCS 4
Simulated, MCS 8

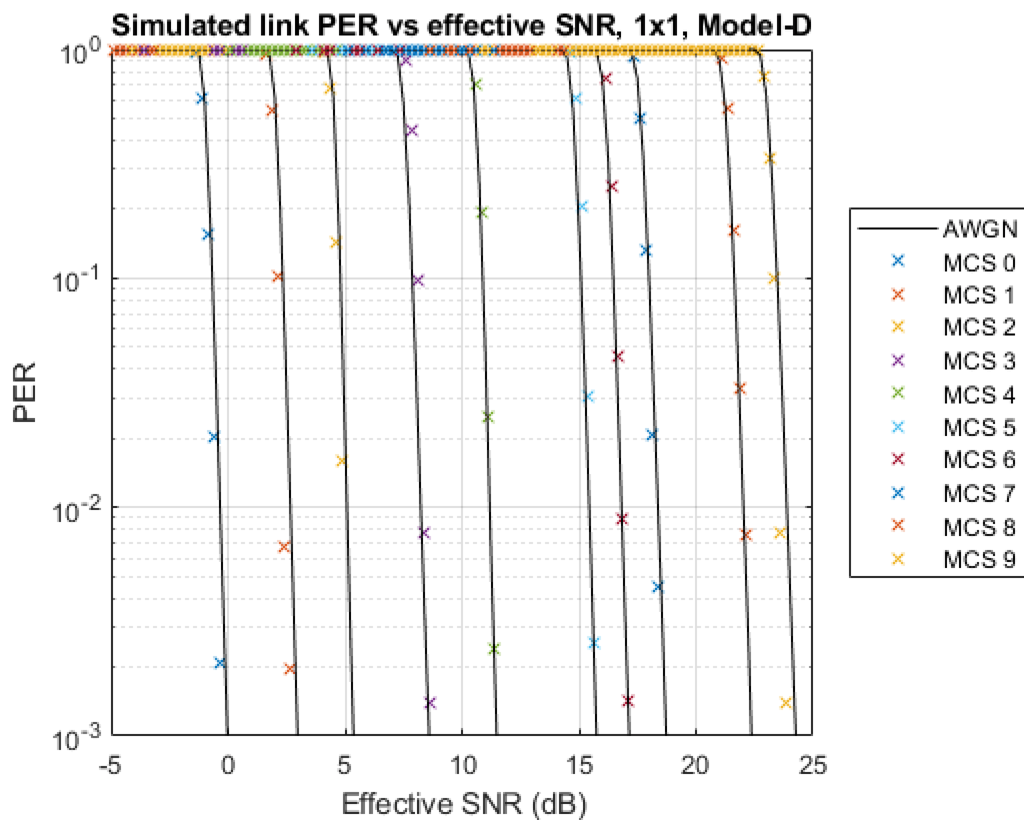



```
rng(sprev) % Restore random state
```

In this example the tuning parameters α and β are set to 1. These could be tuned to further improve the accuracy of the abstraction if desired. The results when simulating 1000 packet errors or 100000 packets for MCS 0 to 9 for a 1458-byte packet without tuning is shown.



Abstracted vs Simulated PHY, 1x1, Model-D
PER
SNR (dB)
10^{0}
10^{-1}
10^{-2}
10^{-3}
0
10
20
30
40
Abstracted, MCS 0
Abstracted, MCS 1
Abstracted, MCS 2
Abstracted, MCS 3
Abstracted, MCS 4
Abstracted, MCS 5
Abstracted, MCS 6
Abstracted, MCS 7
Abstracted, MCS 8
Abstracted, MCS 9
Simulated, MCS 0
Simulated, MCS 1
Simulated, MCS 2
Simulated, MCS 3
Simulated, MCS 4
Simulated, MCS 5
Simulated, MCS 6
Simulated, MCS 7
Simulated, MCS 8
Simulated, MCS 9



Simulated link PER vs effective SNR, 1x1, Model-D
PER
Effective SNR (dB)
10^{0}
10^{-1}
10^{-2}
10^{-3}
-5
0
5
10
15
20
25
AWGN
MCS 0
MCS 1
MCS 2
MCS 3
MCS 4
MCS 5
MCS 6
MCS 7
MCS 8
MCS 9

Further Exploration

To see how the 802.11ax physical layer abstraction described in this example can be used in a system-level simulation, see the “802.11ax System-Level Simulation with Physical Layer Abstraction” on page 7-66 example.

Selected Bibliography

1. IEEE P802.11ax™/D4.1 Draft 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 6: Enhancements for High Efficiency WLAN.
2. IEEE 802.11-14/0571r12 - 11ax Evaluation Methodology.
3. Brueninghaus, Karsten, et al. "Link performance models for system level simulations of broadband radio access systems." *2005 IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*. Vol. 4. IEEE, 2005.
4. Mehlführer, Christian, et al. "The Vienna LTE simulators-Enabling reproducibility in wireless communications research." *EURASIP Journal on Advances in Signal Processing* 2011.1 (2011): 29.
5. IEEE 802.11-14/0800r30 - Box 1 and Box 2 Calibration Results.
6. IEEE 802.11-14/0980r16 - TGax Simulation Scenarios.
7. IEEE 802.11-14/1450r0 - Box 0 Calibration Results

See Also

More About

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “802.11ax Multinode System-Level Simulation of Residential Scenario” on page 7-40

Generate and Visualize FTP Application Traffic Pattern

This example shows how to generate a file transfer protocol (FTP) application traffic pattern based on the IEEE® 802.11ax™ Evaluation Methodology [1 on page 7-110] and the 3GPP TR 36.814 specification [2 on page 7-110].

FTP Application Traffic Model

Multinode communication systems involve modeling of different application traffic models. Each application is characterized by parameters such as the data rate, packet inter arrival time, and packet size. To evaluate various algorithms and protocols, standardization bodies such as IEEE and 3GPP define certain application traffic patterns such as Voice over Internet Protocol (VoIP), video conferencing, and FTP. This example generates and visualizes an FTP application traffic pattern.

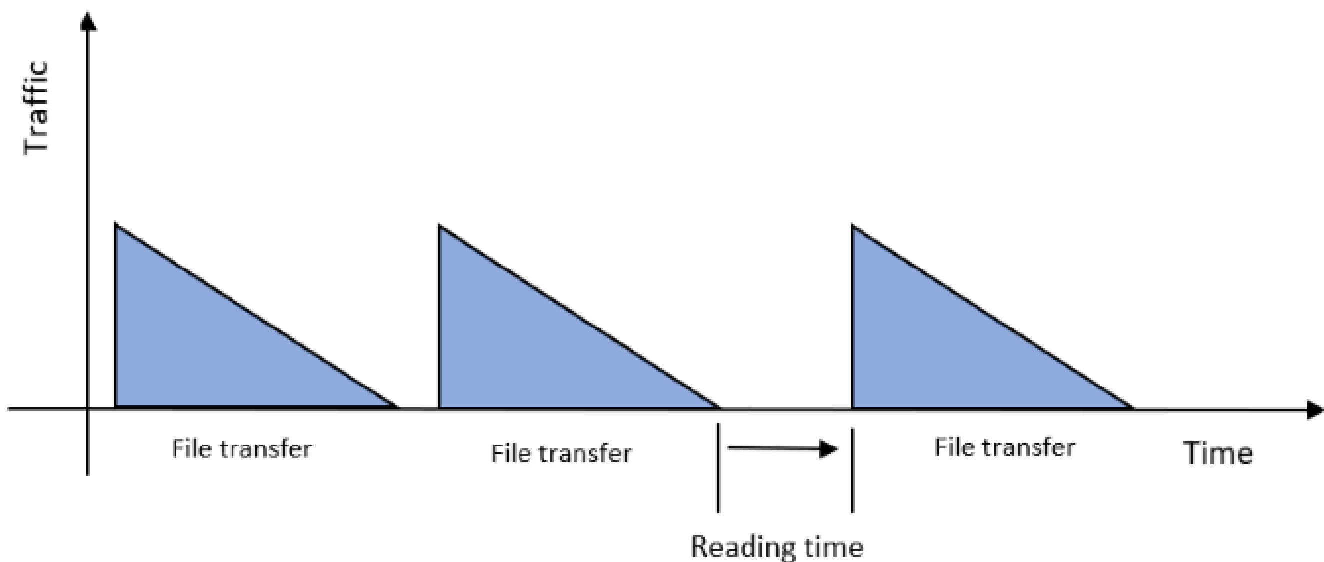
The FTP application traffic pattern is modeled as a sequence of file transfers separated by reading time. The reading time specifies the time interval between two successive file transfers. The file is generated as multiple packets separated by packet inter arrival time. The packet inter arrival time specifies the time interval between two successive packet transfers.

The 11ax Evaluation Methodology [1 on page 7-110] specifies this FTP application traffic model:

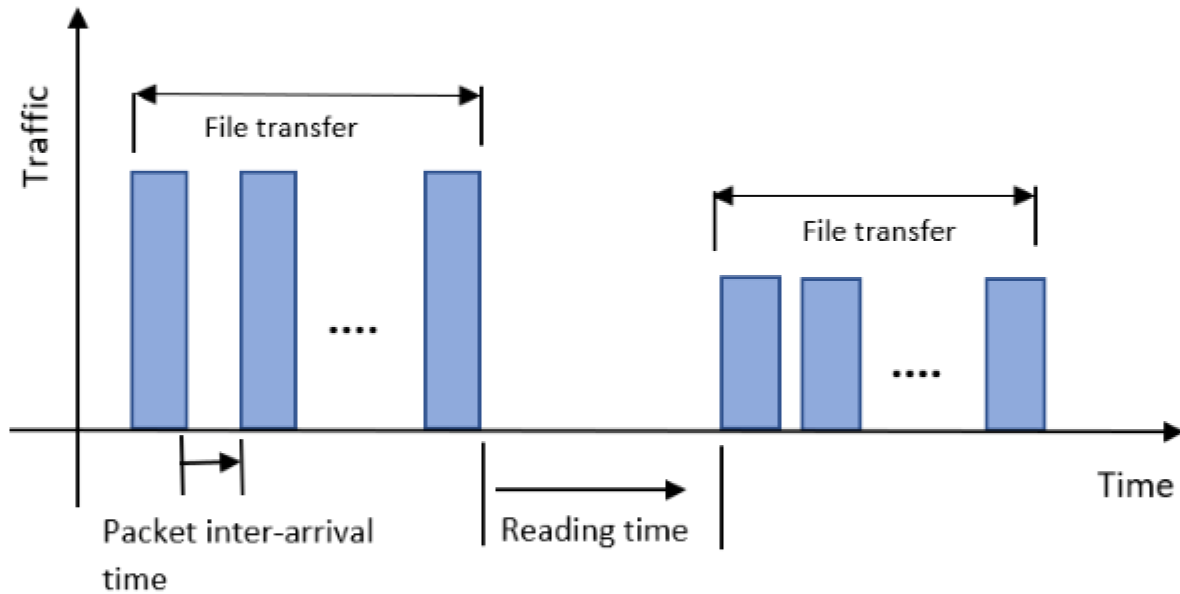
- **Local FTP traffic model -** This model is characterized by truncated Lognormal file size and exponential reading time.

The 3GPP TR 36.814 specification [2 on page 7-110] specifies these FTP application traffic models:

- **FTP traffic model 2 -** This model is characterized by 2/0.5 megabytes file size and exponential reading time. This figure shows the traffic pattern of this FTP model.



- **FTP traffic model 3 -** This model is characterized by a 0.5 megabytes file, exponential reading time, and Poisson packet arrival rate. This figure shows the traffic pattern of this FTP model.



This example demonstrates the local FTP traffic model specified in 11-ax Evaluation Methodology [1 on page 7-110]. Similarly, you can use the FTP traffic models 2 and 3 specified in 3GPP TR 36.814 specification [2 on page 7-110] using the file size and packet arrival rate properties.

Configure FTP Application Traffic Pattern Object

Check if the 'Communications Toolbox Wireless Network Simulation Library' support package is installed.

```
wirelessnetworkSupportPackageCheck
```

Create a configuration object to generate the FTP application traffic pattern.

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

% Create FTP application traffic pattern object with default properties
ftpObj = networkTrafficFTP;

% Set exponential distribution mean value for reading time in milliseconds
ftpObj.ExponentialMean = 50;

% Set truncated Lognormal distribution mu value for file size calculation
ftpObj.LogNormalMu = 10;

% Set truncated Lognormal distribution sigma value for file size calculation
ftpObj.LogNormalSigma = 1;

% Set truncated Lognormal distribution upper limit in Megabytes
ftpObj.UpperLimit = 5;

% Display object
disp(ftpObj);
```

```
  networkTrafficFTP with properties:

               LogNormalMu: 10
            LogNormalSigma: 1
                UpperLimit: 5
           ExponentialMean: 50
    PacketInterArrivalTime: 0
            GeneratePacket: 0
```

Generate and Visualize FTP Application Traffic Pattern

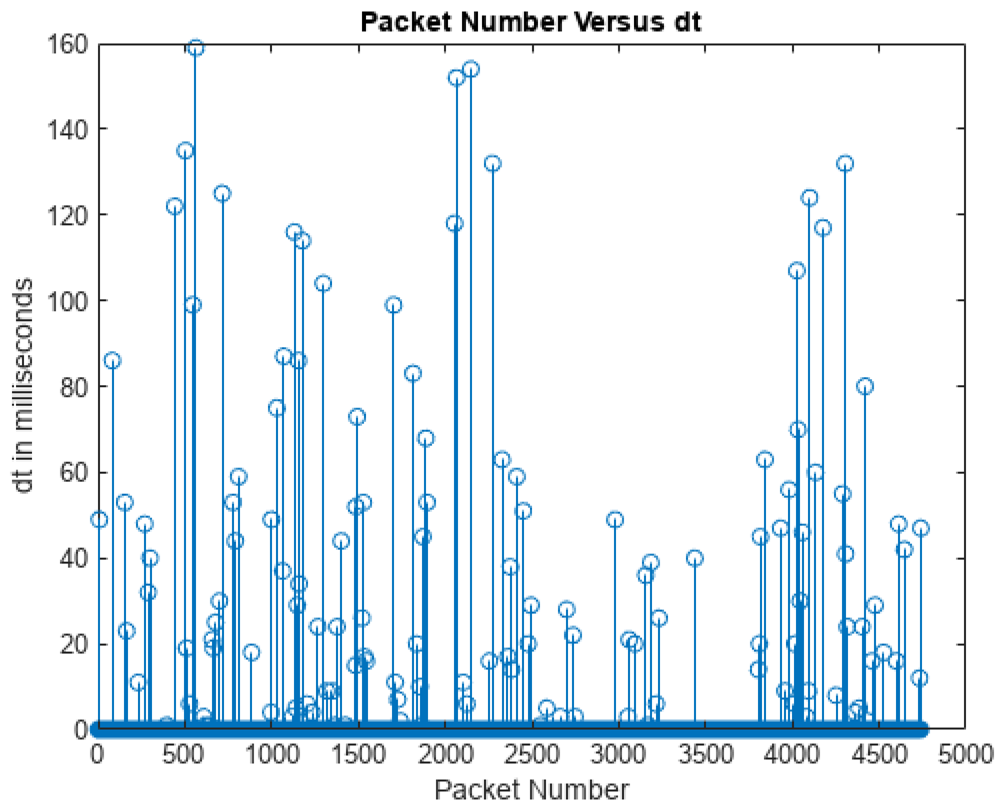
Generate FTP application traffic pattern using the `generate` object function of the `networkTrafficFTP` object.

```
% Set simulation time in milliseconds
simTime = 10000;

% Set step time in milliseconds
stepTime = 1;

% Validate simTime, simTime must be greater than or equal to stepTime
validateattributes(simTime,{'numeric'}, ...
    {'real','scalar','finite','>=',stepTime});

% Time after which the generate method must be invoked again
nextInvokeTime = 0;

% Generated packet count
packetCount = 0;

% Initialize arrays to store outputs for visualization
% Packet generation times in milliseconds
generationTime = zeros(5000,1);

% Time interval between two consecutive packet transfers in milliseconds
packetIntervals = zeros(5000,1);

% Packet sizes in bytes
packetSizes = zeros(5000,1);

% Loop over the simulation time, generating FTP application traffic
% pattern and saving the dt and packet size values for visualization.
while simTime
    if nextInvokeTime <= 0          % Time to generate the packet
        packetCount = packetCount+1; % Increment packet count
        % Call generate method and store outputs for visualization
        [packetIntervals(packetCount), packetSizes(packetCount)] = ...
            generate(ftpObj);
        % Set next invoke time
        nextInvokeTime = packetIntervals(packetCount);
        % Store packet generation time for visualization
        generationTime(packetCount+1) = ...
            generationTime(packetCount) + packetIntervals(packetCount);
    end

    % Update next invoke time
    nextInvokeTime = nextInvokeTime - stepTime;
```

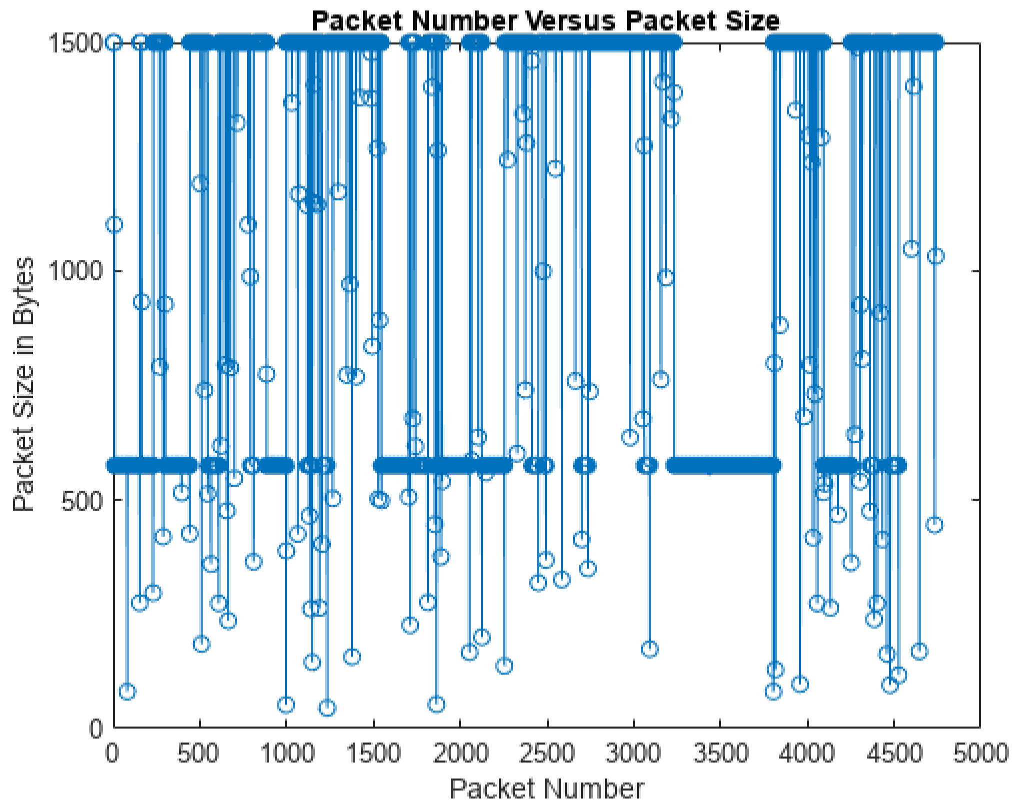
```
    % Update simulation time
    simTime = simTime - stepTime;
end
```

Visualize the generated FTP application traffic pattern. In this plot, `dt` is the time interval between two successive FTP application packets.

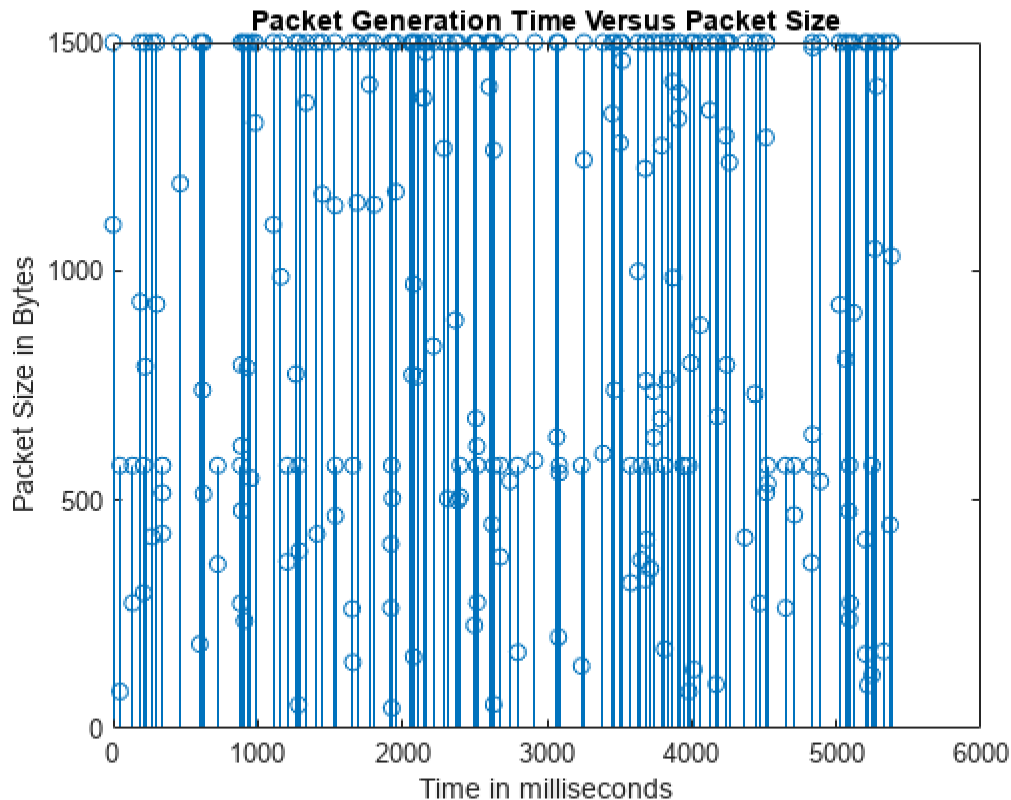
```
% Packet Number Versus Packet Intervals (dt)
% Stem graph to see packet intervals
pktIntervalsFig = figure(Name='Packet intervals',NumberTitle='off');
pktIntervalsAxes = axes(pktIntervalsFig);
stem(pktIntervalsAxes,packetIntervals(1:packetCount));
title(pktIntervalsAxes,'Packet Number Versus dt');
xlabel(pktIntervalsAxes,'Packet Number');
ylabel(pktIntervalsAxes,'dt in milliseconds');
```




```
% Plot to see different packet sizes
pktSizesFig = figure(Name='Packet sizes',NumberTitle='off');
pktSizesAxes = axes(pktSizesFig);
plot(pktSizesAxes,packetSizes(1:packetCount),Marker='o');
title(pktSizesAxes,'Packet Number Versus Packet Size');
xlabel(pktSizesAxes,'Packet Number');
ylabel(pktSizesAxes,'Packet Size in Bytes');
```




```
% Stem graph of FTP application traffic pattern (Packet sizes of
% different files at different packet generation times)
ftpPatternFig = figure(Name='FTP application traffic pattern', ...
    NumberTitle='off');
ftpPatternAxes = axes(ftpPatternFig);
stem(ftpPatternAxes,generationTime(1:packetCount), ...
    packetSizes(1:packetCount),Marker='o');
title(ftpPatternAxes,'Packet Generation Time Versus Packet Size');
ylabel(ftpPatternAxes,'Packet Size in Bytes');
xlabel(ftpPatternAxes,'Time in milliseconds');
```



Further Exploration

This example generates an FTP traffic pattern as per the 11ax Evaluation Methodology [1 on page 7-110] and 3GPP specification [2 on page 7-110]. Similarly, you can use `networkTrafficVoIP`, `networkTrafficOnOff`, and `networkTrafficVideoConference` objects to generate VoIP, On-Off, video conferencing application traffic patterns, respectively. You can use these different application traffic patterns in system-level simulations to model the real-world data traffic.

References

[1] IEEE 802.11-14/0571r12 . "11ax Evaluation Methodology". IEEE P802.11. Wireless LANs.

[2] 3GPP TR 36.814. "Evolved Universal Terrestrial Radio Access (E-UTRA). Further advancements for E-UTRA physical layer aspects". *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

See Also

Objects
`networkTrafficFTP` | `networkTrafficOnOff` | `networkTrafficVideoConference` | `networkTrafficVoIP`

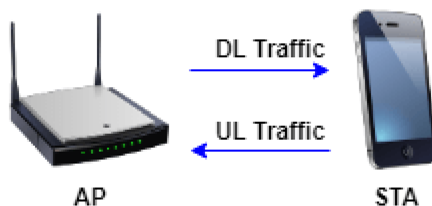
Simulate an 802.11ax Network with Uplink and Downlink Application Traffic

This example shows how to create, configure, and simulate an IEEE® 802.11ax™ network with uplink (UL) and downlink (DL) application traffic.

Using this example, you can:

1. Create and configure an 802.11ax network consisting of an access point (AP) and a station (STA).
2. Associate the STA with the AP.
3. Generate, configure, and add UL and DL on-off application traffic between the STA and the AP.
4. Simulate the 802.11ax network and visualize the statistics.

The example simulates this scenario.



Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Set the seed for the random number generator to 1. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive");
```

Specify the simulation time in seconds.

```
simulationTime = 0.2;
```

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Specify the number of nodes in the network. Set the positions of the nodes as a vector. Units are in meters. Each row of the vector specifies the *x*-, *y*-, and *z*- Cartesian coordinates of a node, starting from the first node.

```
numNodes = 2;
nodePositions = [0 0 0;0 -20 0];
```

Set the configuration parameters of the AP and the STA by using the `wlanDeviceConfig` object.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",MCS=2,TransmitPower=15);    % AP device configuration
stationCfg = wlanDeviceConfig(Mode="STA",MCS=2,TransmitPower=15);       % STA device configuratio
```

Create an AP and a STA from the specified configuration by using the `wlanNode` object.

```
accessPoint = wlanNode(Name="AP", ...
    Position=nodePositions(1,:), ...
    DeviceConfig=accessPointCfg);

stations = wlanNode(Name="STA", ...
    Position=nodePositions(2,:), ...
    DeviceConfig=stationCfg);
```

Create an 802.11ax network consisting of an AP and a STA.

```
nodes = [accessPoint stations];
```

Associate the STA with the AP.

```
associateStations(accessPoint,stations);
```

Generate an on-off application traffic pattern by using the `networkTrafficOnOff` object. Configure the on-off application traffic by specifying the application data rate and packet size. Add UL and DL application traffic between the AP and the STA by using the `addTrafficSource` object function.

Alternatively, you can configure full buffer application traffic between the APs and STAs by using the `FullBufferTraffic` input of the `associateStations` object function.

```
trafficSourceUL = networkTrafficOnOff(DataRate=100000,PacketSize=1500);
addTrafficSource(stations,trafficSourceUL,DestinationNode=accessPoint,AccessCategory=0); % UL tra

trafficSourceDL = networkTrafficOnOff(DataRate=100000,PacketSize=1500);
addTrafficSource(accessPoint,trafficSourceDL,DestinationNode=stations,AccessCategory=0); % DL tra
```

Add nodes to the wireless network simulator.

```
addNodes(networkSimulator,nodes);
```

Run the network simulation for the specified simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channel
```

At each node, the simulation captures the statistics by using the `statistics` object function. The `stats` variable captures the application statistics, MAC layer statistics, physical layer statistics, and mesh forwarding statistics for each node. For more information about these statistics, see the “WLAN System-Level Simulation Statistics” on page 7-114 topic.

```
stats = statistics(nodes)
```

```
stats=1×2 struct array with fields:
    Name
    ID
    App
    MAC
    PHY
    Mesh
```

See Also

Functions
`statistics`

Objects
`wlanDeviceConfig` | `wlanNode`

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “Simulate an 802.11ax Network with Full MAC and Abstracted PHY” on page 7-118
- “Simulate a Multiband 802.11ax Network” on page 7-124
- “Create, Configure, and Simulate an 802.11ax Mesh Network” on page 7-121
- “Simulate an 802.11ax Hybrid Mesh Network” on page 7-127

WLAN System-Level Simulation Statistics

The WLAN system-level simulation captures these statistics as a structure at each node. This structure is returned as an output by the `statistics` object function of the `wlanNode` object.

Default Statistics

If you use the first syntax described on the `statistics` reference page, these tables describe the function's output.

Statistics Structure

Structure Field	Description
`Name`	Name of the WLAN node, returned as a string scalar.
`ID`	ID of the WLAN node, returned as a numeric scalar.
`App`	Application statistics, returned as a structure.
`MAC`	MAC layer statistics, returned as a structure. If the node contains multiple devices, this field returns an array of structures.
`Mesh`	Mesh forwarding statistics, returned as a structure. If the node contains multiple devices, this field returns an array of structures.
`PHY`	Physical layer (PHY) statistics, returned as a structure. If the node contains multiple devices, this field returns an array of structures.

App Structure

Structure Field	Description
`TransmittedPackets`	Total number of packets transmitted by the application layer
`TransmittedBytes`	Total number of bytes transmitted from the application layer
`ReceivedPackets`	Total number of packets received at the application layer
`ReceivedBytes`	Total number of bytes received at the application layer

MAC Structure

Structure Field	Description
TransmittedDataFrames	Number of data MPDUs transmitted from the MAC layer, including retransmissions
TransmittedPayloadBytes	Total number of MSDU bytes transmitted from the MAC layer that are acknowledged
SuccessfulDataTransmissions	Number of acknowledged data MPDUs
RetransmittedDataFrames	Number of data MPDUs retransmitted from MAC layer
TransmittedAMPDUs	Total number of A-MPDUs transmitted from the MAC layer, including retransmissions
TransmittedRTSFrames	Total number of RTS frames transmitted from the MAC layer
TransmittedMURTSFrames	Total number of MU-RTS frames transmitted from the MAC layer
TransmittedCTSFrames	Total number of CTS frames transmitted from the MAC layer
TransmittedMUBARFrames	Total number of MU-BAR frames transmitted from the MAC layer
TransmittedAckFrames	Total number of Ack frames transmitted from the MAC layer
TransmittedBlockAckFrames	Total number of BlockAck frames transmitted from the MAC layer
ReceivedDataFrames	Total number of data MPDUs received by the MAC layer
ReceivedPayloadBytes	Total number of MSDU bytes received by the MAC layer
ReceivedAMPDUs	Total number of A-MPDUs received by the MAC layer, with at least one valid subframe
ReceivedRTSFrames	Total number of RTS frames received by the MAC layer
ReceivedMURTSFrames	Total number of MU-RTS frames received by the MAC layer
ReceivedCTSFrames	Total number of CTS frames received by the MAC layer
ReceivedMUBARFrames	Total number of MU-BAR frames received by the MAC layer
ReceivedAckFrames	Total number of Ack frames received by the MAC layer
ReceivedBlockAckFrames	Total number of BlockAck frames received by the MAC layer
ReceivedFCSValidFrames	Total number of MPDUs received by the MAC layer with valid FCS

Structure Field	Description
`ReceivedFCSFails`	Total number of MPDUs dropped by the MAC layer due to FCS fails
`ReceivedDelimiterCRCFails`	Total number of frames dropped by the MAC layer due to delimiter CRC fails in the A-MPDU subframes

Mesh Structure

Structure Field	Description
`PacketsToBeForwarded`	Number of MSDUs to be forwarded
`PayloadBytesToBeForwarded`	Number of bytes of data to be forwarded
`DroppedPackets`	Number of MSDUs dropped due to: • Insufficient mesh forward hop count • No further path found for forwarding • Receiving a duplicate MSDU

PHY Structure

Structure Field	Description
`TransmittedPackets`	Total number of packets transmitted by the PHY
`TransmittedPayloadBytes`	Total number of payload (PSDU) bytes transmitted by the PHY
`ReceivedPackets`	Total number of packets received at the PHY
`ReceivedPayloadBytes`	Total number of payload (PSDU) bytes received by the PHY
`DroppedPackets`	Total number of packets dropped by the PHY receiver due to: • Low signal power • Preamble, header, or payload decode failures

Additional Statistics

If you use the second syntax detailed on the `statistics` reference page, the function returns these statistics in addition to those described above.

App Structure Additional Fields

Structure Field	Description
`Destinations`	This is an array of structures. Each structure holds statistics that correspond to the destinations of traffic sources that you have added to the node.

Fields of Structures in the Destinations Array

Structure Field	Description
NodeID	Destination node identifier
NodeName	Destination node name
TransmittedPackets	Number of transmitted packets
TransmittedBytes	Number of transmitted bytes

MAC Structure Additional Fields

Structure Field	Description
AccessCategories	This is an array of structures of size 1-by-4. Each structure holds statistics that correspond to one of the four access categories. The four entries of the array correspond to Best Effort, Background, Video, and Voice, respectively.

Fields of Structures in the Access Categories Array

Structure Field	Description
TransmittedDataFrames	Total number of data MPDUs transmitted for the access category from the MAC layer, including retransmissions
TransmittedPayloadBytes	Total number of MSDU bytes transmitted for the access category from the MAC layer that are acknowledged
SuccessfulDataTransmissions	Total number of acknowledged data MPDUs for the access category
RetransmittedDataFrames	Total number of data MPDUs retransmitted for the access category from the MAC layer
RecievedDataFrames	Total number of data MPDUs received for the access category by the MAC layer.
ReceivedPayloadBytes	Total number of MSDU bytes received for the access category by the MAC layer

See Also

Functions
statistics

Objects
wlanDeviceConfig | wlanNode

Simulate an 802.11ax Network with Full MAC and Abstracted PHY

This example shows how to create, configure, and simulate an IEEE® 802.11ax™ network with a full or abstracted model of medium access control (MAC) and the physical layer (PHY). At the transmitter and receiver, modeling full MAC processing involves complete MAC frame generation at the MAC layer. Similarly, modeling full PHY processing involves complete operations related to waveform transmission and reception through a fading channel. When you simulate large networks, full MAC and PHY processing is computationally expensive. In an abstracted MAC, the node does not generate or decode any frames at the MAC layer. Similarly, in an abstracted PHY, the node does not generate or decode any waveforms at the PHY. MAC and PHY abstraction enables you to minimize the complexity and duration of the system-level simulations.

Using this example, you can:

1. Create and configure a two-node 802.11ax network consisting of one access point (AP) and one station (STA).
2. Associate the STA with the AP and add full buffer uplink (UL) application traffic.
3. Configure the AP and STA to implement full MAC and an abstracted PHY.
4. Simulate the 802.11ax network and visualize the statistics.

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation including backoff counter selection at the MAC layer and predicting packet reception success at the PHY. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive");
```

Specify the simulation time in seconds.

```
simulationTime = 0.3;
```

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Specify the number of nodes in the network. Set the positions of the nodes as a vector. Units are in meters. Each row of the vector specifies the *x*-, *y*-, and *z*- Cartesian coordinates of a node, starting from the first node.

```
numNodes = 2;
nodePositions = [0 0 0;20 20 20];
```

The `MACFrameAbstraction` and `PHYAbstractionMethod` properties of the `wlanNode` object enable you to configure MAC and PHY layer abstraction. The valid values for these properties are:

- `MACFrameAbstraction` — `false` (default) or `true`. To use the full MAC, set this property to `true`.
- `PHYAbstractionMethod` — `"tgax-evaluation-methodology"` (default), `"tgax-mac-calibration"`, or `"none"`. To use the abstracted PHY, set this property to `"tgax-evaluation-methodology"` or `"tgax-mac-calibration"`. If you set this property to `"tgax-evaluation-methodology"`, the PHY estimates the performance of a link with the TGax channel model by using an effective signal-to-interference-plus-noise-ratio (SINR) mapping. If you set this property to `"tgax-mac-calibration"`, the PHY assumes a packet failure due to interference without actually calculating the link performance. To use the full PHY, set this property to `"none"`.

```
MACFrameAbstraction = true;
PHYAbstractionMethod = "tgax-mac-calibration";
```

Set the configuration parameters of the AP and the STA by using the `wlanDeviceConfig` object.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",MCS=2,TransmitPower=15);    % AP device configuratio
stationCfg = wlanDeviceConfig(Mode="STA",MCS=2,TransmitPower=15);       % STA device configurati
```

Create an AP and a STA from the specified configuration by using the `wlanNode` object.

```
accessPoint = wlanNode(Name="AP", ...
    Position=nodePositions(1,:), ...
    DeviceConfig=accessPointCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=MACFrameAbstraction);

station = wlanNode(Name=["STA"], ...
    Position=nodePositions(2,:), ...
    DeviceConfig=stationCfg, ...
    PHYAbstractionMethod=PHYAbstractionMethod, ...
    MACFrameAbstraction=MACFrameAbstraction);
```

Create an 802.11ax network consisting of an AP and a STA.

```
nodes = [accessPoint station];
```

Associate the STA with the AP and add full buffer UL application traffic.

```
associateStations(accessPoint,station,FullBufferTraffic="UL");
```

Add nodes to the wireless network simulator.

```
addNodes(networkSimulator,nodes);
```

Run the network simulation for the specified simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channe
```

At each node, the simulation captures the statistics by using the `statistics` object function. The `stats` variable captures the application statistics, MAC layer statistics, physical layer statistics, and mesh forwarding statistics for each node. For more information about these statistics, see the “WLAN System-Level Simulation Statistics” on page 7-114 topic.

```
stats = statistics(nodes)
```

```
stats=1×2 struct array with fields:
    Name
```

```
ID
App
MAC
PHY
Mesh
```

See Also

Functions
`statistics`

Objects
`wlanDeviceConfig` | `wlanNode`

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “Simulate an 802.11ax Network with Uplink and Downlink Application Traffic” on page 7-111
- “Simulate a Multiband 802.11ax Network” on page 7-124
- “Create, Configure, and Simulate an 802.11ax Mesh Network” on page 7-121
- “Simulate an 802.11ax Hybrid Mesh Network” on page 7-127

Create, Configure, and Simulate an 802.11ax Mesh Network

This example shows how to create, configure, and simulate an IEEE® 802.11ax™ mesh network.

Using this example, you can:

1. Create and configure an 802.11ax mesh network consisting of four mesh nodes.
2. Generate, configure, and add on-off application traffic between the mesh nodes.
3. Add mesh paths to route application traffic from the source node to the sink node.
4. Simulate the 802.11ax mesh network and visualize the statistics.

The example simulates this mesh network scenario.

The mesh node 1 (source) generates and transmits application traffic to mesh node 4 (sink) through intermediate relay nodes, mesh node 2, and mesh node 3. The example simulates mesh network communication in the 5 GHz band.

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive");
```

Specify the simulation time in seconds.

```
simulationTime = 0.5;
```

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Specify the names and positions of the mesh nodes.

```
nodeNames = ["MeshNode1","MeshNode2","MeshNode3","MeshNode4"];
nodePositions = [10 0 0; 20 0 0; 30 0 0; 40 0 0];             % x-, y-, and z-coordinates, in me
```

Set the configuration parameters of the mesh nodes by using the `wlanDeviceConfig` object.

```
meshNodeCfg = wlanDeviceConfig(Mode="mesh",BandAndChannel=[5 36],MCS=7,TransmitPower=15);
```

Create the mesh nodes from the specified configuration by using the `wlanNode` object.

```
meshNodes = wlanNode(Name=nodeNames, ...
    Position=nodePositions, ...
    DeviceConfig=meshNodeCfg);
```

Generate an on-off application traffic pattern by using the `networkTrafficOnOff` object. Configure the on-off application traffic by specifying the application data rate and packet size. Add application traffic from mesh node 1 to mesh node 4.

```
trafficSource = networkTrafficOnOff(DataRate=50000,PacketSize=1500);
addTrafficSource(meshNodes(1),trafficSource,DestinationNode=meshNodes(4));
```

Add mesh paths to propagate the application traffic from the source mesh node 1 to the destination mesh node 4. The application traffic flows from mesh node 1 to mesh node 4 through the intermediate relay nodes mesh node 2 and mesh node 3.

```
addMeshPath(meshNodes(1),meshNodes(4),meshNodes(2));
addMeshPath(meshNodes(2),meshNodes(4),meshNodes(3));
addMeshPath(meshNodes(3),meshNodes(4));
```

Add nodes to the wireless network simulator.

```
addNodes(networkSimulator,meshNodes);
```

Run the network simulation for the specified simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channe
```

At each mesh node, the simulation captures the statistics by using the `statistics` object function. The `stats` variable captures the application statistics, MAC layer statistics, physical layer statistics, and mesh forwarding statistics for each node. For more information about these statistics, see the “WLAN System-Level Simulation Statistics” on page 7-114 topic.

```
stats = statistics(meshNodes)
```

```
stats=1×4 struct array with fields:
    Name
    ID
    App
    MAC
    PHY
    Mesh
```

See Also

Functions
`statistics`

Objects
`wlanDeviceConfig` | `wlanNode`

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “Simulate an 802.11ax Network with Uplink and Downlink Application Traffic” on page 7-111
- “Simulate an 802.11ax Network with Full MAC and Abstracted PHY” on page 7-118
- “Simulate a Multiband 802.11ax Network” on page 7-124
- “Simulate an 802.11ax Hybrid Mesh Network” on page 7-127

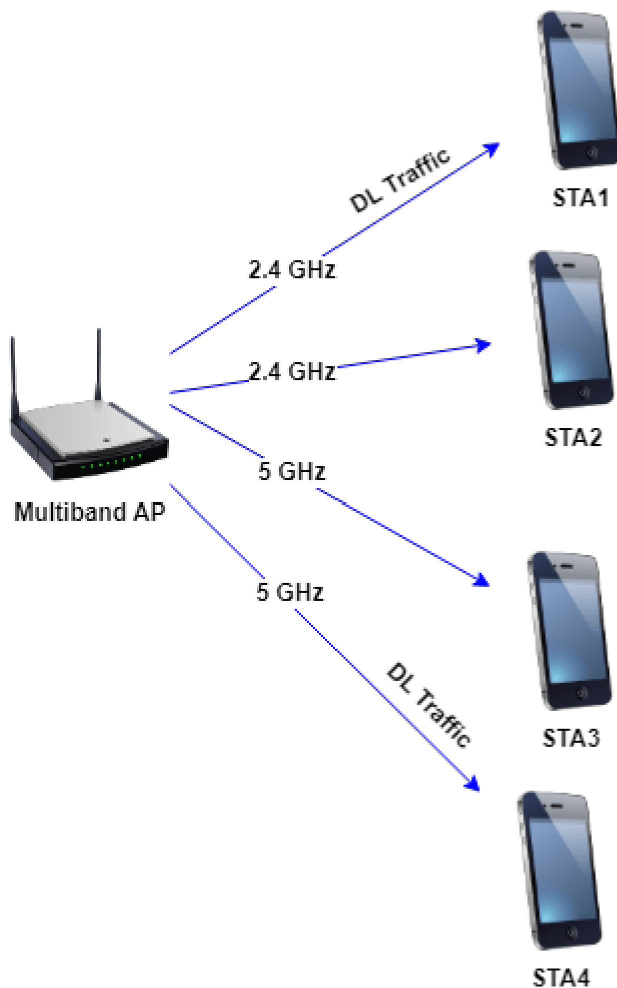
Simulate a Multiband 802.11ax Network

This example shows how to create, configure, and simulate an IEEE® 802.11ax™ network operating in the 2.4 GHz and 5 GHz band.

Using this example, you can:

1. Create and configure an 802.11ax network consisting of a multiband access point (AP) and four stations (STAs).
2. Configure the multiband AP to operate in the 2.4 GHz and 5 GHz bands.
3. Associate the STAs with the AP and add full buffer downlink (DL) application traffic between them.
4. Simulate the multiband band 802.11ax network.

The example simulates this scenario.



STA1 and STA 2 are associated with the multiband AP in the 2.4 GHz band. STA3 and STA4 are associated with the multiband AP in the 5 GHz band. The multiband AP transmits DL application traffic to STA1 and STA4.

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive");
```

Specify the simulation time in seconds.

```
simulationTime = 0.5;
```

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Configure the multiband AP to operate in the 2.4 GHz and 5 GHz bands.

```
accessPointCfg1 = wlanDeviceConfig(Mode="AP",BandAndChannel=[2.4 11]);
accessPointCfg2 = wlanDeviceConfig(Mode="AP",BandAndChannel=[5 36]);
```

Create a multiband AP from the specified configuration by using the `wlanNode` object.

```
accessPoint = wlanNode(Position=[0 0 0],DeviceConfig=[accessPointCfg1,accessPointCfg2]);
```

Configure STA1 and STA2 to operate in the 2.4 GHz band. Configure STA3 and STA4 to operate in the 5 GHz band.

```
stationCfg1 = wlanDeviceConfig(Mode="STA",BandAndChannel=[2.4 11]);
stationCfg2 = wlanDeviceConfig(Mode="STA",BandAndChannel=[5 36]);
```

Create STA1, STA2, STA3, and STA4 from the specified configuration.

```
stations1 = wlanNode(Position=[10 0 0; 0 10 0],DeviceConfig=stationCfg1);  % STA1 and STA2 on 2.4
stations2 = wlanNode(Position=[0 0 10; 10 10 0],DeviceConfig=stationCfg2); % STA3 and STA4 on 5 G
```

Associate STAs with the AP and add full buffer application traffic between them.

```
associateStations(accessPoint,[stations1(1),stations2(2)],FullBufferTraffic="DL");
```

Add nodes to the wireless network simulator.

```
addNodes(networkSimulator,[accessPoint,stations1,stations2]);
```

Run the network simulation for the specified simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channel
```

At each node, the simulation captures the statistics by using the `statistics` object function. The `stats` variable captures the application statistics, MAC layer statistics, physical layer statistics, and

mesh forwarding statistics for each node. For more information about these statistics, see the “WLAN System-Level Simulation Statistics” on page 7-114 topic.

```
stats = statistics([accessPoint,stations1,stations2])
```

```
stats=1×5 struct array with fields:
    Name
    ID
    App
    MAC
    PHY
    Mesh
```

See Also

Functions
statistics

Objects
wlanDeviceConfig | wlanNode

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “Simulate an 802.11ax Network with Uplink and Downlink Application Traffic” on page 7-111
- “Simulate an 802.11ax Network with Full MAC and Abstracted PHY” on page 7-118
- “Create, Configure, and Simulate an 802.11ax Mesh Network” on page 7-121
- “Simulate an 802.11ax Hybrid Mesh Network” on page 7-127

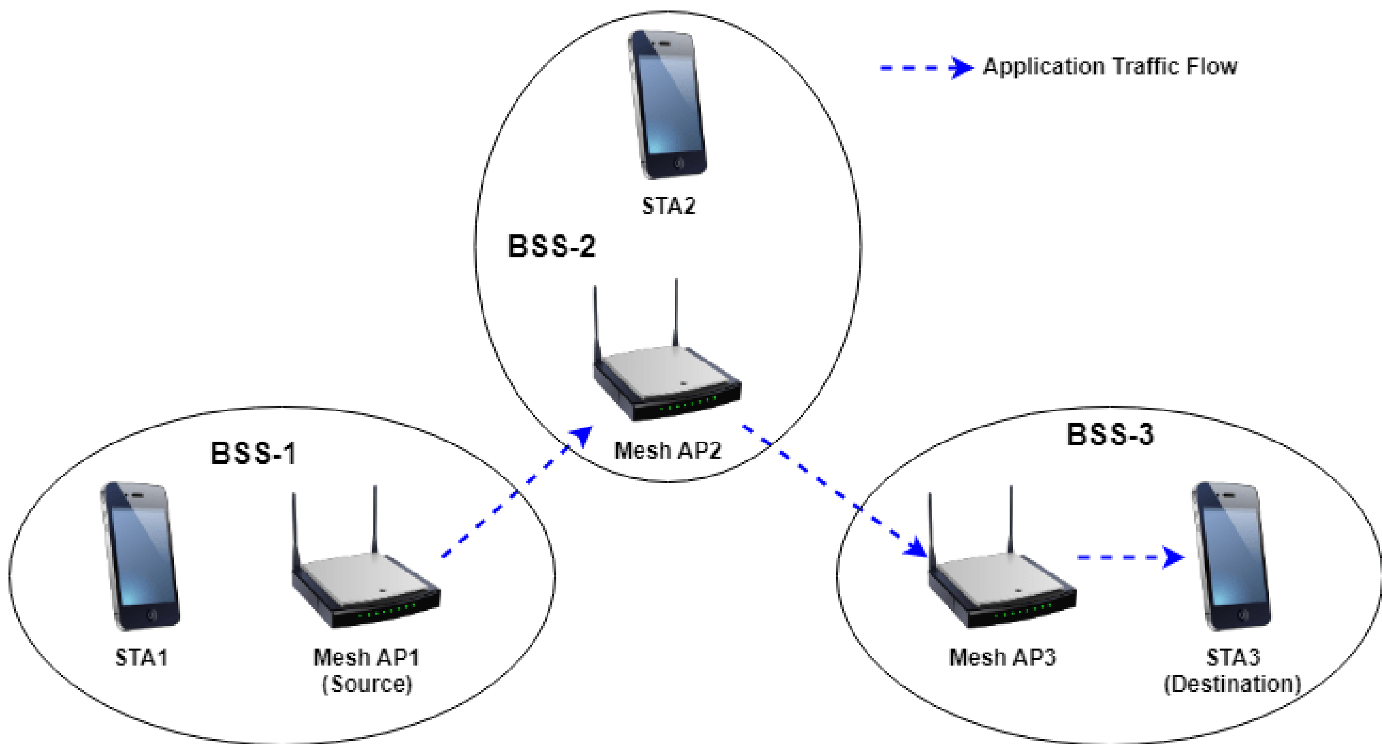
Simulate an 802.11ax Hybrid Mesh Network

This example shows how to create, configure, and simulate an IEEE® 802.11ax™ hybrid mesh network.

Using this example, you can:

1. Create and configure an 802.11ax hybrid mesh network consisting of three mesh access points (APs) and three stations (STAs).
2. Configure the mesh APs to operate in the 2.4 GHz and 6 GHz bands.
3. Associate the STAs with the mesh APs.
4. Generate, configure, and add on-off application traffic between an AP and a STA.
5. Add mesh paths to route the application traffic from the source to the destination.
6. Simulate the 802.11ax hybrid mesh network.

The example simulates this scenario.



STA1, STA2, and STA3 are associated with mesh AP1, mesh AP2, and mesh AP3, respectively. Mesh AP1, mesh AP2, and mesh AP3 together consitute the mesh network. For the mesh network, STA1, STA2, and STA3 are nonmesh nodes. Mesh AP1 is the source that generates and transmits the application traffic to STA3 through the intermediate relay nodes mesh AP2 and mesh AP3.

Check if the Communications Toolbox™ Wireless Network Simulation Library support package is installed. If the support package is not installed, MATLAB® returns an error with a link to download and install the support package.

```
wirelessnetworkSupportPackageCheck;
```

Set the seed for the random number generator to `1`. The seed value controls the pattern of random number generation. The random number generated by the seed value impacts several processes within the simulation including backoff counter selection at the MAC layer and predicting packet reception success at the physical layer. To improve the accuracy of your simulation results, after running the simulation, you can change the seed value, run the simulation again, and average the results over multiple simulations.

```
rng(1,"combRecursive");
```

Specify the simulation time in seconds.

```
simulationTime = 0.5;
```

Initialize the wireless network simulator.

```
networkSimulator = wirelessNetworkSimulator.init;
```

Configure the mesh APs such that the AP mode operates in the 2.4 GHz band and the mesh mode operates in the 6 GHz band.

```
accessPointCfg = wlanDeviceConfig(Mode="AP",BandAndChannel=[2.4 11]);
meshCfg = wlanDeviceConfig(Mode="mesh",BandAndChannel=[6 1]);
```

Create three mesh APs from the specified configuration by using the `wlanNode` object.

```
meshAPs = wlanNode(Name=["Mesh AP1" "Mesh AP2" "Mesh AP3"], ...
    Position=[0 10 0; 0 5 0; 0 0 0], ...
    DeviceConfig=[accessPointCfg meshCfg]);
```

Configure the STAs to operate in the 2.4 GHz band.

```
stationsCfg = wlanDeviceConfig(Mode="STA",BandAndChannel=[2.4 11]);
```

Create three STAs from the specified configuration by using the `wlanNode` object.

```
stations = wlanNode(Name=["STA1" "STA2" "STA3"],Position=[0 0 10; 10 5 0; 10 0 0],DeviceConfig=s
```

Associate STA1 with mesh AP1, STA2 with mesh AP2, and STA3 with mesh AP3.

```
associateStations(meshAPs(1),stations(1));
associateStations(meshAPs(2),stations(2));
associateStations(meshAPs(3),stations(3));
```

Generate an on-off application traffic pattern by using the `networkTrafficOnOff` object. Add the application traffic source from mesh AP1 to STA3 by using the `addTrafficSource` object function.

Alternatively, you can configure full buffer application traffic between the AP and STA by specifying the `FullBufferTraffic` input of the `associateStations` object function.

```
trafficSource = networkTrafficOnOff(DataRate=1e5);
addTrafficSource(meshAPs(1),trafficSource,DestinationNode=stations(3));
```

Add mesh paths to propagate application traffic from mesh AP1 (source) to STA3 (destination). The application traffic flows from mesh AP1 to STA3 through the intermediate relay nodes mesh AP2 and mesh AP3.

```
addMeshPath(meshAPs(1),stations(3),meshAPs(3)); % Mesh path from mesh AP1 to STA3 (non-mesh node
addMeshPath(meshAPs(1),meshAPs(3),meshAPs(2));  % Mesh path from mesh AP1 to mesh AP3 through mes
addMeshPath(meshAPs(2),meshAPs(3));             % Mesh path from mesh AP2 to mesh AP3
```

Add nodes to the wireless network simulator.

```
addNodes(networkSimulator,[meshAPs, stations]);
```

Run the network simulation for the specified simulation time.

```
run(networkSimulator,simulationTime);
```

```
Custom channel model is not added. Using free space path loss (fspl) model as the default channel
```

At each node, the simulation captures the statistics by using the `statistics` object function. The `stats` variable captures the application statistics, MAC layer statistics, physical layer statistics, and mesh forwarding statistics for each node. For more information about these statistics, see the “WLAN System-Level Simulation Statistics” on page 7-114 topic.

```
stats = statistics([meshAPs, stations])
```

```
stats=1×6 struct array with fields:
    Name
    ID
    App
    MAC
    PHY
    Mesh
```

See Also

Functions
`statistics`

Objects
`wlanDeviceConfig` | `wlanNode`

Related Examples

- “Get Started with WLAN System-Level Simulation in MATLAB” on page 7-31
- “Simulate an 802.11ax Network with Uplink and Downlink Application Traffic” on page 7-111
- “Simulate an 802.11ax Network with Full MAC and Abstracted PHY” on page 7-118
- “Simulate a Multiband 802.11ax Network” on page 7-124
- “Create, Configure, and Simulate an 802.11ax Mesh Network” on page 7-121

Test and Measurement

Modeling and Testing an 802.11ax RF Receiver with 5G Interference

The example shows how to characterize the impact of RF impairments on the reception of an IEEE® 802.11ax™ waveform coexisting with an adjacent 5G or 802.11ax interferer. The example generates the baseband waveforms by using WLAN Toolbox™ and 5G Toolbox™ and models the RF receiver by using RF Blockset™. The example does not require 5G Toolbox if it models an 802.11ax interferer.

Introduction

This example characterizes the impact of an adjacent 5G or 802.11ax interferer on the RF reception of an 802.11ax waveform. To evaluate the impact of the interference, the example performs these measurements:

- Error vector magnitude (EVM): vector difference between the ideal (transmitted) signal and the measured (received) signal
- Adjacent channel rejection (ACR): power difference between the desired signal and an interfering signal in the adjacent channel
- Packet error rate (PER): number of packets containing errors divided by the total number of received packets

The impact of the receiver RF impairments such as phase noise and amplifier nonlinearities are also considered.

The example works on a packet-by-packet basis and uses a Simulink model to perform these steps:

1. Generate the baseband 802.11ax waveform (desired) by using WLAN Toolbox features.
2. Generate the baseband 5G waveform (interferer) by using 5G Toolbox features. You can generate an 802.11ax interferer by using WLAN Toolbox features instead. Alternatively, you can remove the interference.
3. Match the sampling rate of the two signals by using the Sample-Rate Match block.
4. To capture spectral regrowth of the combined bandwidth, oversample the waveforms by a factor of 10 by using an FIR Interpolation block.
5. Measure and display the ACR by calculating the power difference between both waveforms.
6. Aggregate both waveforms by using the Multiband Combiner block.
7. Import the combined waveform into the RF Receiver Subsystem block implemented by using RF Blockset blocks. The model provides an RF frequency to the combined waveform to carry the baseband information in RF Blockset.
8. Model the effects of downconverting the combined waveform to an intermediate frequency by using the RF Receiver Subsystem block. This block models the impairments introduced by an RF receiver using RF Blockset blocks.
9. Downsample the desired HE waveform by using an FIR Decimation block to compensate for the upsampling performed by the FIR Interpolation block.
10. Downsample the desired HE waveform by using an FIR Rate Conversion block to compensate for any upsampling performed by the Sample-Rate Match block.
11. Extract the data symbols and measure the EVM by demodulating the baseband waveform.
12. Calculate the PER by extracting the received bits and comparing them to the transmitted bits.

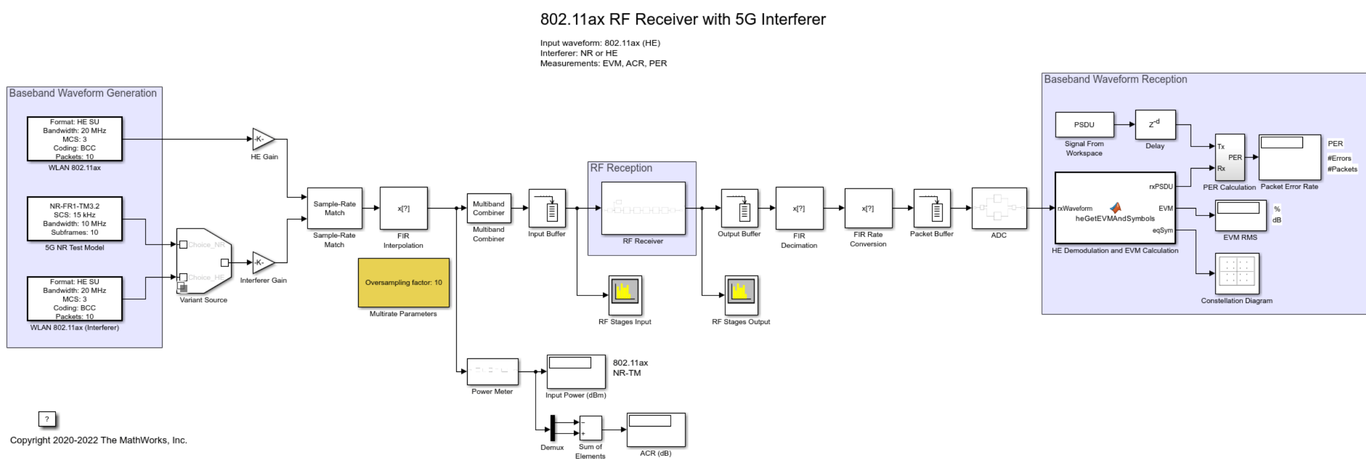
The Simulink model uses WLAN Toolbox, 5G Toolbox, and DSP System Toolbox™ features to process the baseband waveforms (steps 1-6 and 9-12) and uses RF Blockset blocks to model the RF receiver (steps 7 and 8). This model supports `Normal` and `Accelerator` simulation modes.

Simulink Model Structure

The model contains three main parts:

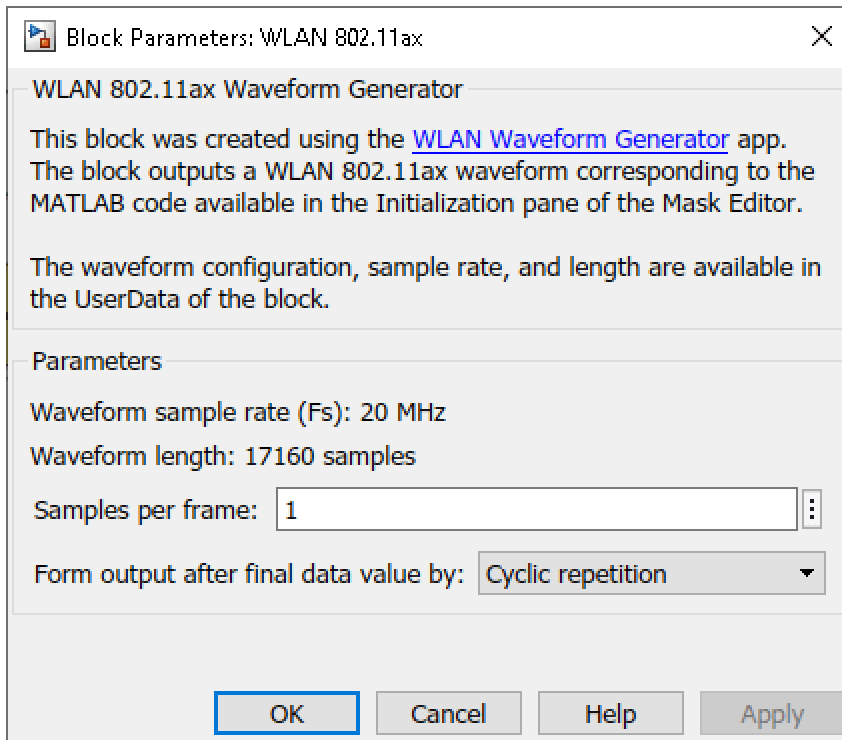
- Baseband Waveform Generation: generates and combines the baseband 802.11ax and 5G waveforms
- RF Reception: models the effects of downconverting the combined waveform to an intermediate frequency
- Baseband Waveform Reception: calculates EVM and PER

```
modelName = 'HERFReceiverNRInterfererModel';
open_system(modelName);
```

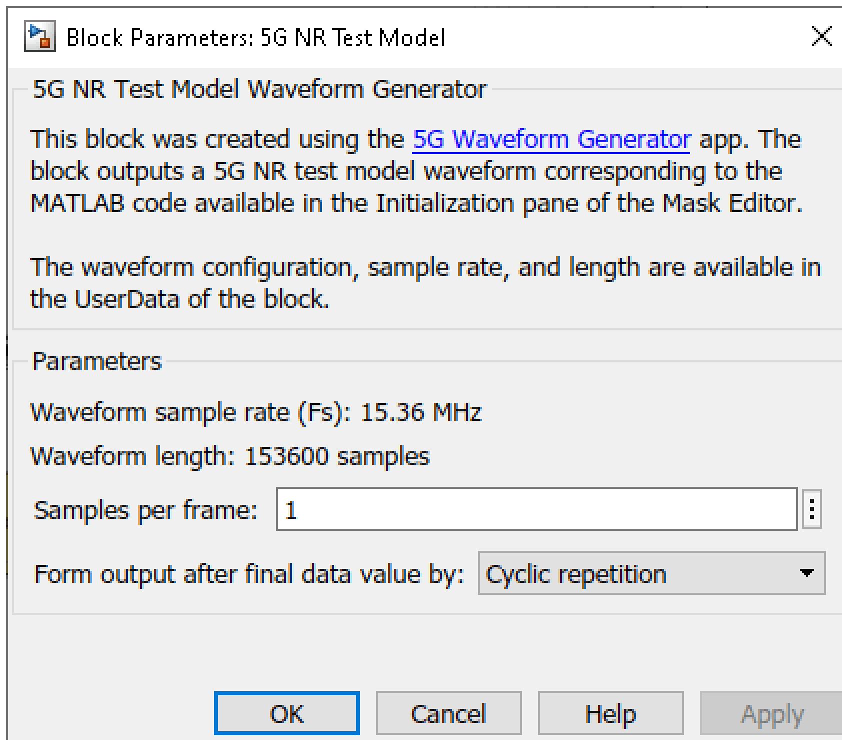


Baseband Waveform Generation

The WLAN 802.11ax block generates standard-compliant high-efficiency single-user (HE SU) waveforms, according to IEEE Std 802.11ax-2021. You can generate this block using the **WLAN Waveform Generator** app. You can access the waveform configuration parameters in the user data of the block. This example uses the **InitFcn** in the **Model callbacks** to store the structure available in the user data in a Base Workspace variable, `HEInfo`. For more information about this block, see Waveform From Wireless Waveform Generator App.



Similarly, the 5G NR Test Model block transmits standard-compliant NR-TM waveforms, as defined in TS 38.141-1. You can generate this block using the **5G Waveform Generator (5G Toolbox)** app. The example also uses the **InitFcn** in the **Model callbacks** to store the NR-TM configuration available in the block user data in a Base Workspace variable, `IntInfo`. For more information about this block, see Waveform From Wireless Waveform Generator App (5G Toolbox).



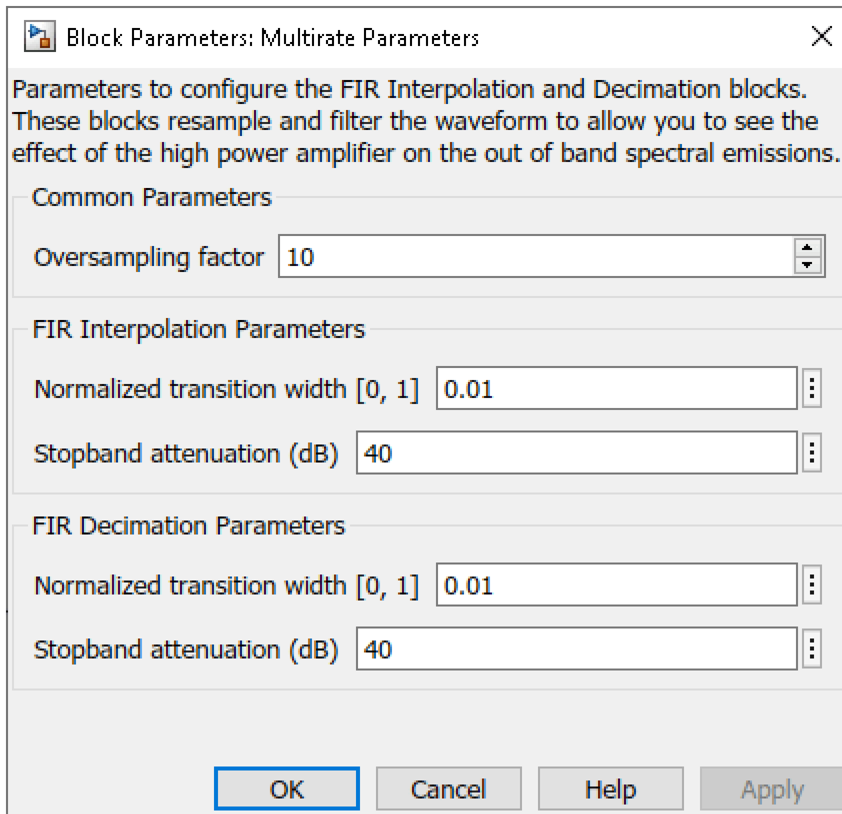
To use a different HE SU waveform, open the **WLAN Waveform Generator** app, select the HE SU configuration, and export a new block. Similarly, to use a different NR-TM waveform, open the **5G Waveform Generator** app, select the NR-TM configuration, and export a new block. For more information on how to generate and use such blocks, see "Generate Wireless Waveform in Simulink Using App-Generated Block" on page 8-62.

Alternatively, you can model an 802.11ax interferer by selecting **Choice_HE** in the Variant Source block. The model shifts the HE interferer so that it is not synchronized with the desired HE waveform.

Control the power of both waveforms by setting the HE Gain and Interferer Gain blocks. To cancel the transmission of the interferer, set the **Gain** parameter of the Interferer Gain block to 0.

The Sample-Rate Match block upsamples the waveform with lower sample rate to match the sample rate of the other waveform. The waveforms must have the same sample rate when combining the waveforms in the Multiband Combiner block. The Sample-Rate Match block also concatenates both waveforms horizontally, one column per waveform.

An FIR Interpolation block oversamples and filters the waveforms to show the effect of the nonlinear impairments on the adjacent channels. To capture at least fifth-order nonlinearities, oversample the combined bandwidth (both waveforms) by a factor of five. As the current combined bandwidth is 35 MHz (20 and 10 MHz bandwidths and a 20 MHz spacing between them), set an **Oversampling factor** of 10 to provide a sample rate of 200 MHz, which is around five times the combined bandwidth. You can set the **Oversampling factor** in the Multirate Parameters block, which provides an interface to configure the parameters of the FIR Interpolation and Decimation blocks.



The Multiband Combiner block frequency shifts and aggregates the oversampled waveforms. To measure the ACR, place the center frequency of the adjacent channel 20, 40, 80, or 160 MHz away from the center frequency of the desired signal, according to IEEE Std 802.11ax-2021. By default, the example centers the HE waveform at baseband (0 Hz) and sets the spacing between the HE and interfering waveforms to 20 MHz. You can adjust the center frequencies by specifying the **Frequency offsets (Hz)** parameter in the Multiband Combiner block. The ACR measurement is displayed in the ACR (dB) block.

Specify Simulation Time

Set the **Stop Time** value in the Simulink model to the time required to transmit and obtain the EVM results and constellation diagram of at least one packet. Considering the waveform configuration selected in the WLAN 802.11ax block, the packet transmission time in this example is 85.8 microseconds. As the filters in the FIR Interpolation, Decimation and Rate Conversion blocks introduce a delay, you can use the **IdleTime** parameter in the Mask Editor of the WLAN 802.11ax block to compensate for the delay.

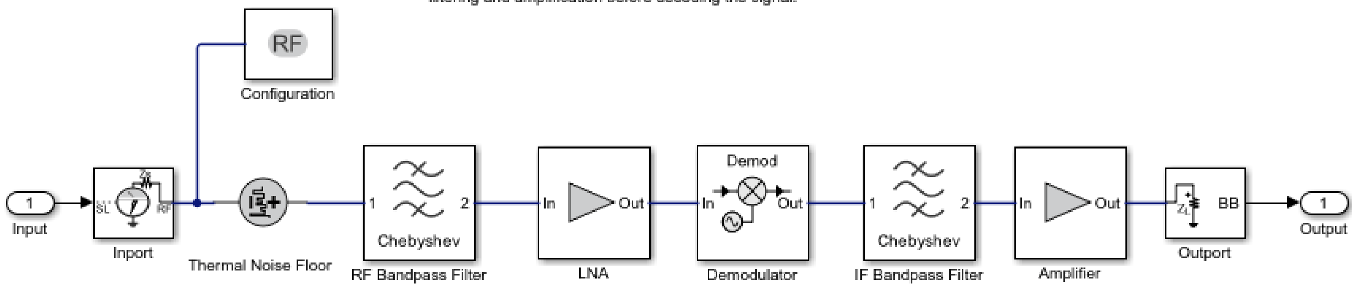
RF Reception

The RF Receiver Subsystem block is based on a superheterodyne receiver architecture. This architecture models the effects of downconverting the combined waveform to an intermediate frequency by characterizing these RF components:

- RF and IF bandpass filters
- Low noise amplifiers

- Demodulator consisting of mixers, phase shifter, and local oscillator

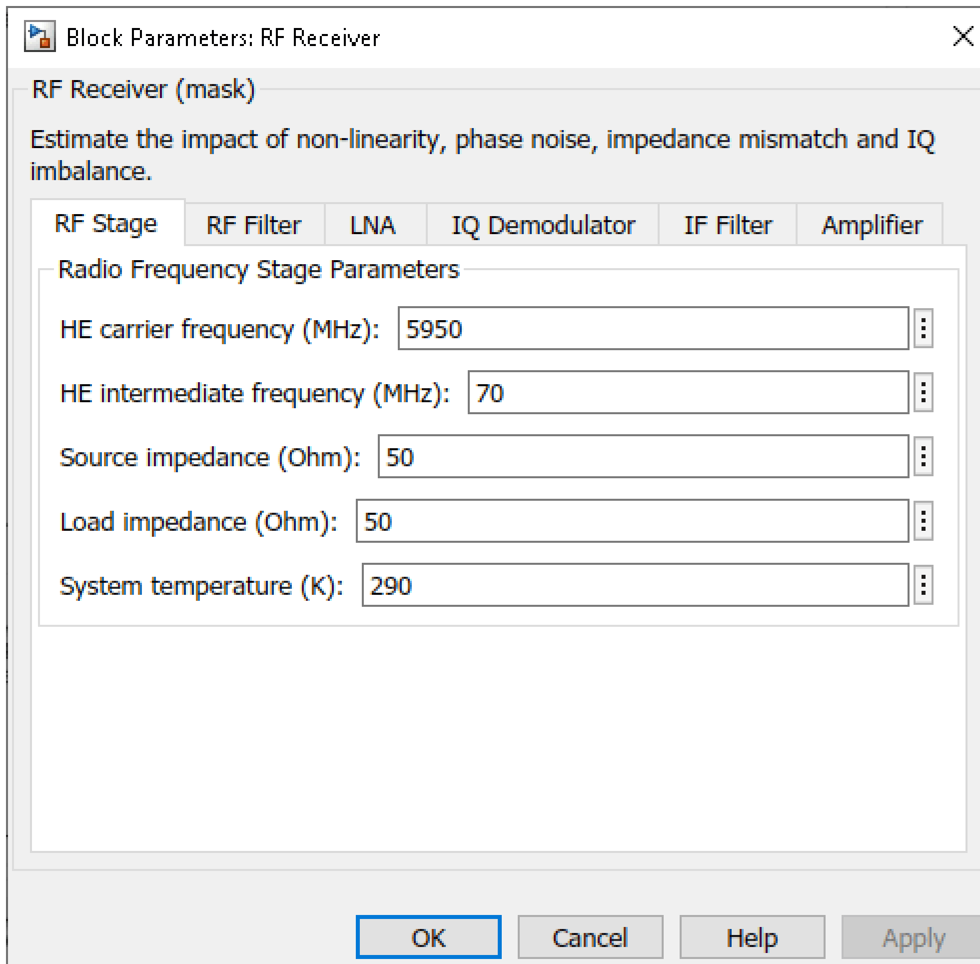
```
set_param(modelName,'Open','off');
set_param([modelName '/RF Receiver'],'Open','on');
```



Use an Input Buffer block to send one sample at a time to the RF Receiver Subsystem block.

The Inport block inside the RF Receiver Subsystem block converts the combined Simulink complex baseband waveform into the RF Blockset Circuit Envelope simulation environment. You can vary the center frequency of the combined RF signal by modifying the **Carrier frequency** parameter of the Inport block. By default, the **Carrier frequency** parameter of the Inport block corresponds to the center frequency of the desired HE waveform and the carrier frequency of the NR waveform is located 20 MHz from the HE carrier. The Outport block converts the RF Blockset signal back into Simulink complex baseband.

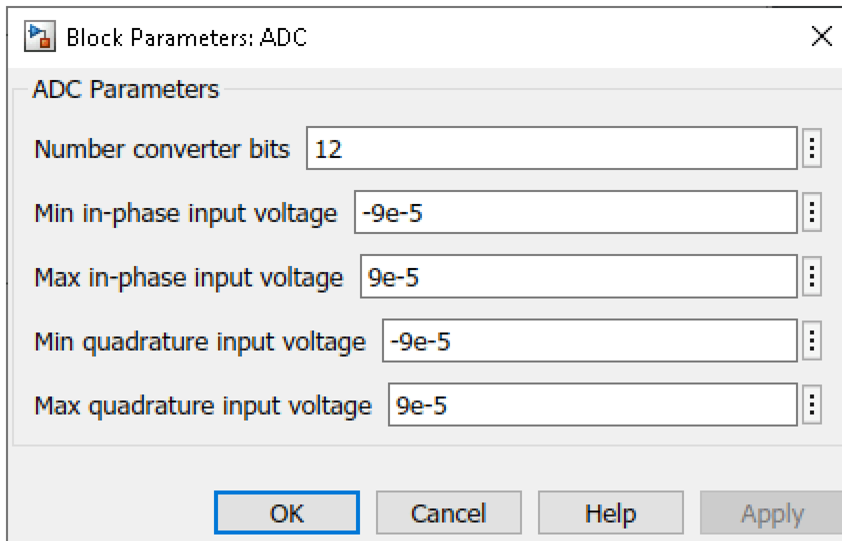
You can configure the RF Receiver components by using the RF Receiver Subsystem block mask.



The RF Receiver Subsystem block models typical impairments, including:

- Phase noise as an effect directly related to the thermal noise within the active devices of the oscillator
- Amplifier nonlinearities due to DC power limitation when the amplifiers work in the saturation region
- Impedance mismatch resulting in signal reflection or an inefficient power transfer

As the current RF Receiver Subsystem block configuration sends one sample at a time, the Output Buffer block (after the RF Receiver Subsystem block) collects all samples within the baseband HE waveform before sending the samples to the HE Demodulation and EVM Calculation block. At the output of the RF Receiver Subsystem block, the FIR Decimation and FIR Rate Conversion blocks downsample the waveform back to its original sampling rate. Additionally, the ADC Subsystem block digitizes the signal. You can modify the parameters of the blocks inside the ADC Subsystem block using its mask.



Baseband Waveform Reception

The HE Demodulation and EVM Calculation block recovers and plots the HE-Data symbols in the Constellation Diagram block by performing frequency and packet offset corrections, channel estimation, pilot phase tracking, OFDM demodulation, and equalization. This block performs these EVM measurements:

- EVM per subcarrier (dB): EVM averaged over the allocated HE-Data symbols within a subcarrier
- EVM per OFDM symbol (dB)
- Overall EVM (dB and %): EVM averaged over all transmitted HE-Data symbols

This block also decodes each packet to recover the transmitted bits. The example compares the recovered bits to those transmitted for each packet to determine the packet error rate for the simulation duration by using the PER Calculation block.

The ACR measurement is displayed in the ACR (dB) block. You can also measure the ACR by calculating the power difference between the **Channel Power** levels of each waveform in the Spectrum Analyzer Input block. To check the **Channel Power** levels of each waveform, set this configuration in the Spectrum Analyzer Input block:

- The **Span (Hz):** must be the bandwidth of the waveform to measure. By default, the example sets this value to 20 MHz, which is the bandwidth of the desired HE waveform.
- The **CF (Hz):** must be 0 for the desired HE waveform or the spacing between both waveforms (defined in the Multiband Combiner block) for the interferer. By default, the example sets this value to 0 Hz to measure the channel power of the desired waveform.

To measure ACR according to IEEE Std 802.11ax-2021, set the desired waveform power 3 dB above the rate-dependent sensitivity specified in Table 27-51 (-71 dBm for the default configuration) and adjust the power level of the interferer waveform to achieve a 10% PER for a PSDU length of 4096 octets.

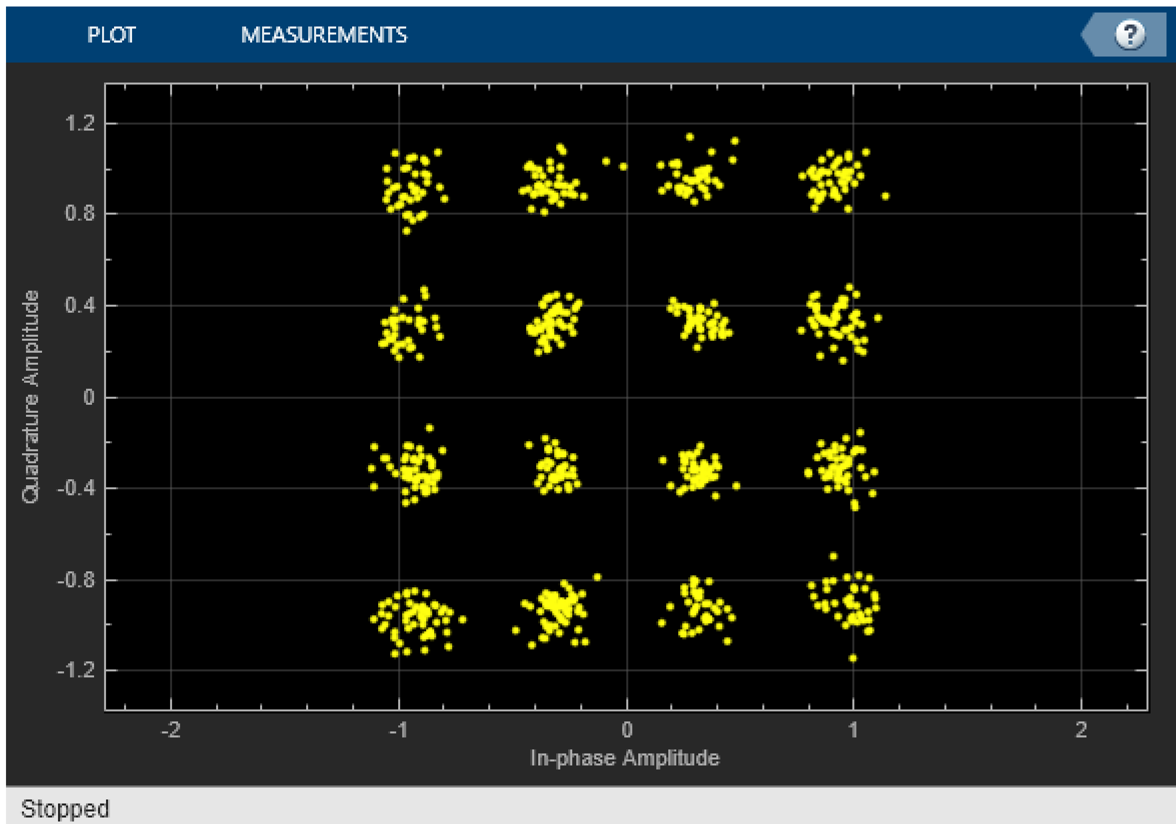
Model Performance

To characterize the impact of the NR interference on the HE reception you can compare the EVM for two different cases: 1) without interference, for example, transmit only the HE waveform; and 2) with

interference, for example, transmit both HE and NR waveforms. You can also measure the ACR in the second case.

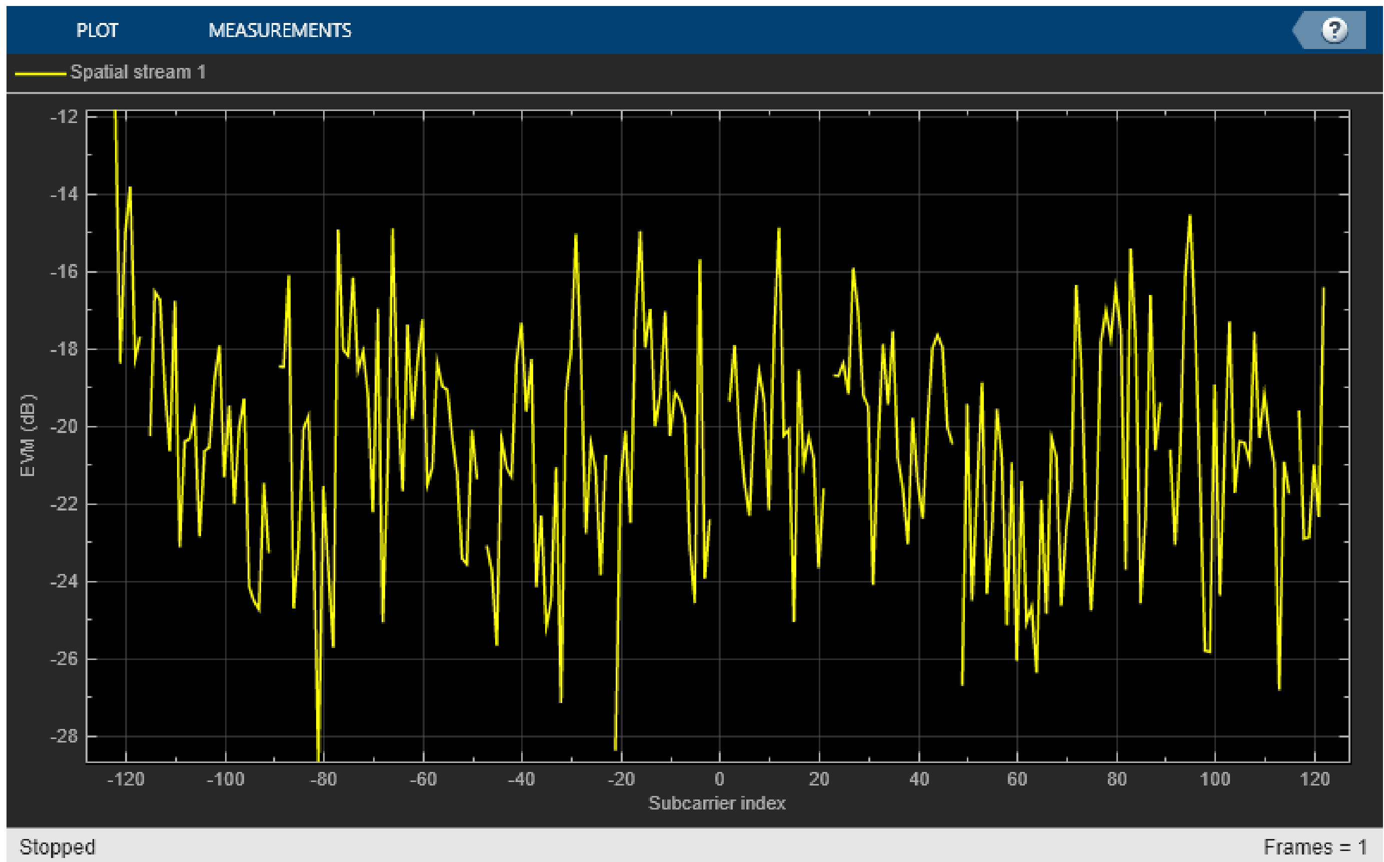
- *Without NR interference (NR gain = 0)*. To eliminate the NR interference, set the **Gain** parameter of the Interferer Gain block to 0. To calculate the EVM and plot the constellation diagram, run the simulation to capture one packet (**Stop Time** equal to 85.5 microseconds for the default configuration).

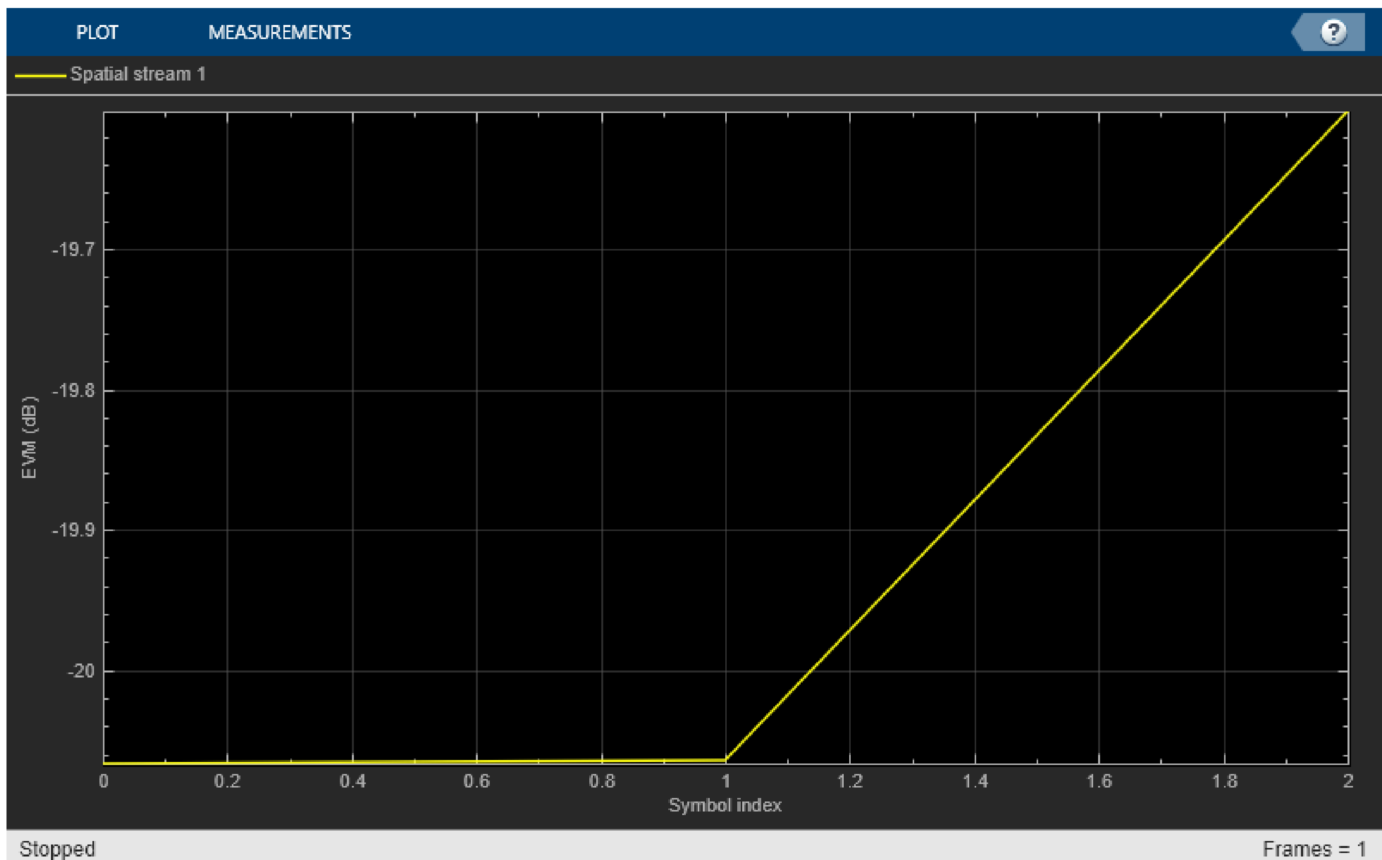
```
set_param([modelName '/Interferer Gain'],'Gain','0');
sim(modelName);
```



ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
dBm
-40
-60
-80
-100
-120
-140
-160
-180
-200
-220
-90
-60
-30
0
30
60
90
Frequency (MHz)
Stopped
Sample Rate = 200.000 MHz Frames = 0 T = 0.00000

ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
dBm
-40
-50
-60
-70
-80
-90
-100
-110
-120
-130
-140
-80
-40
0
40
80
Frequency (MHz)
Stopped
Sample Rate = 200.000 MHz Frames = 0 T = 0.00000

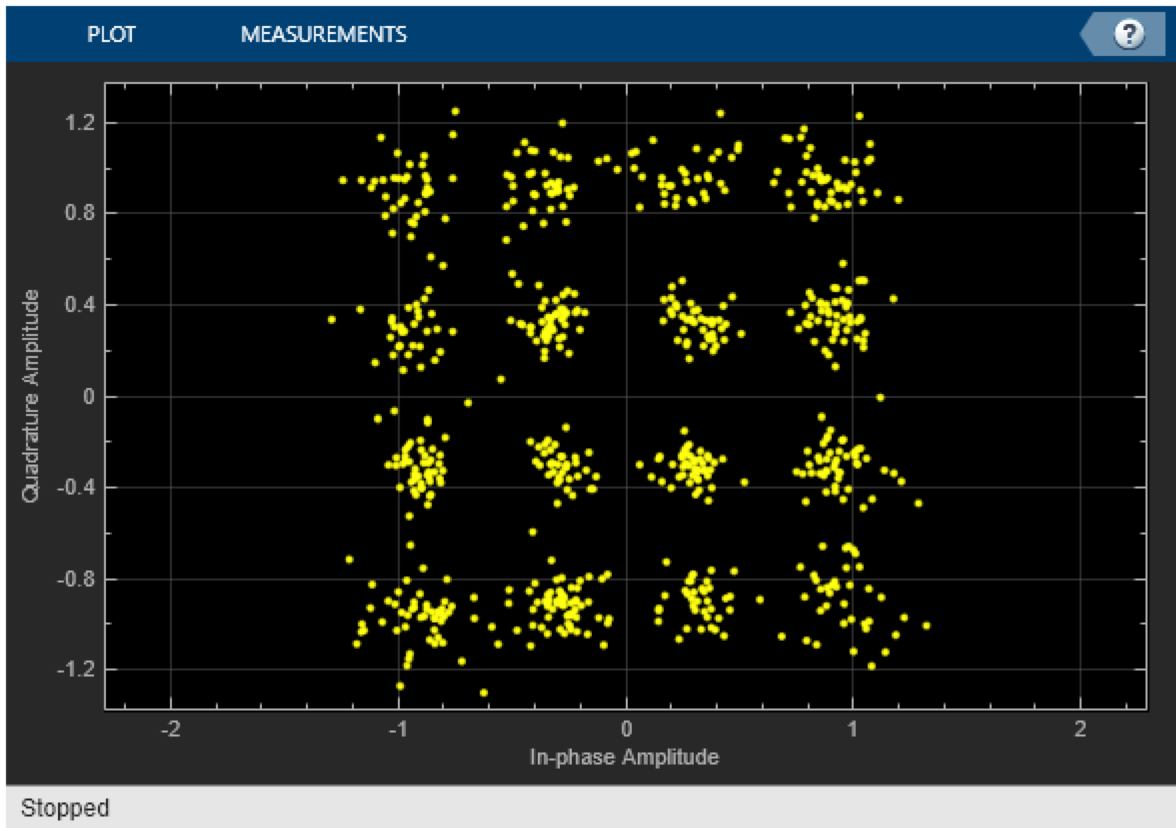

PLOT
MEASUREMENTS
Spatial stream 1
EVM (dB)
-12
-14
-16
-18
-20
-22
-24
-26
-28
-120
-100
-80
-60
-40
-20
0
20
40
60
80
100
120
Subcarrier index
Stopped
Frames = 1



When you disable the interference, the overall EVM is around -20 dB.

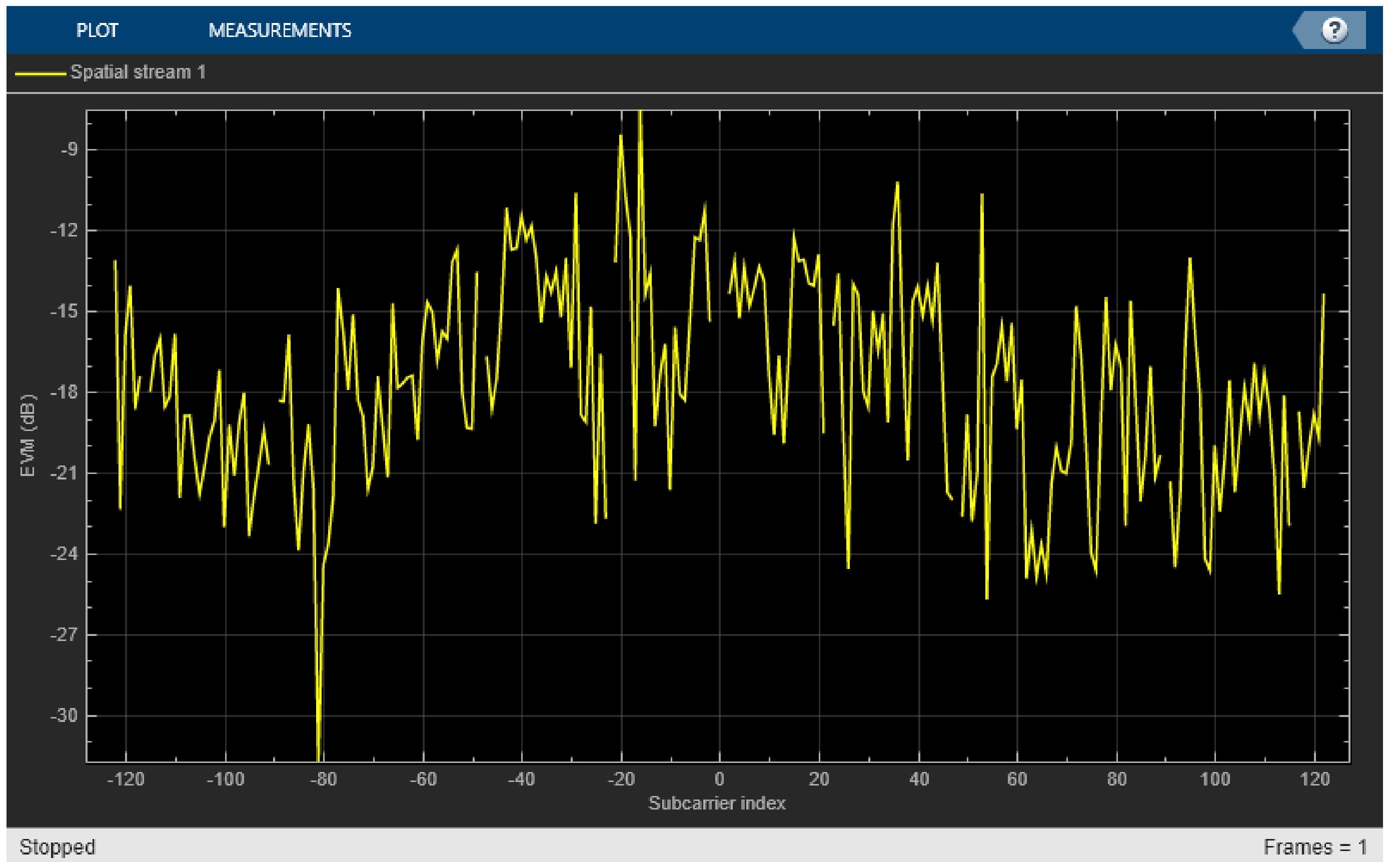
- *With NR interference (NR gain = -37.72 dB)*. To activate the NR interference, set the **Gain** parameter of the Interferer Gain block to any value greater than 0. For example, in order to measure the ACR when the PER is approximately 10% for a PSDU length of 4096 octets, IEEE Std 802.11ax-2021, choose a gain value of around -30 dB and increase the APEP length. If you want to measure the PER for several packet transmissions, for example 100 packets, multiply the current **Stop Time** value by 100. By default, the example transmits one packet and sets the APEP length to 50 bytes.

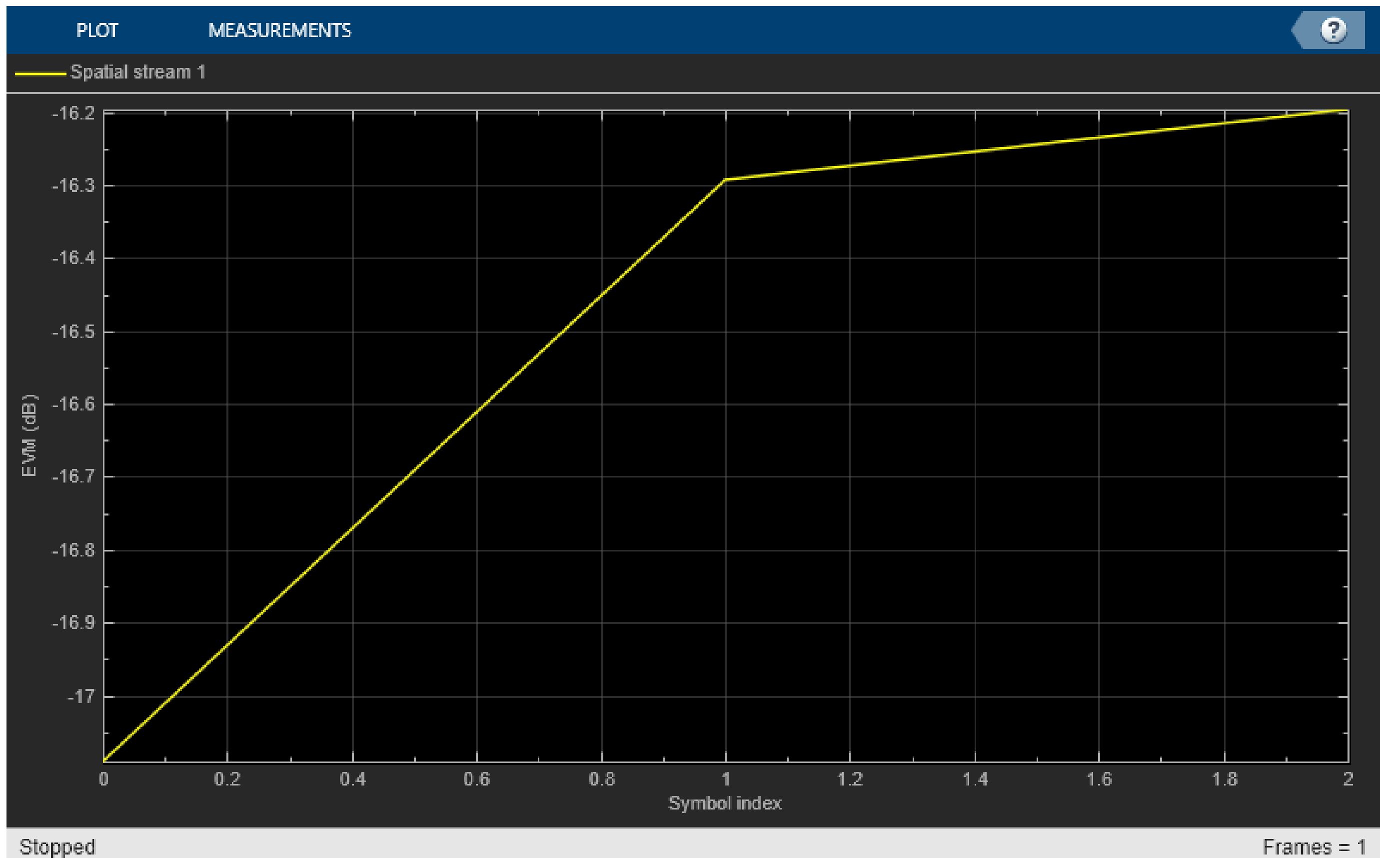
```
set_param([modelName '/Interferer Gain'],'Gain','db2mag(-30)');
sim(modelName);
```


PLOT
MEASUREMENTS
1.2
0.8
0.4
0
-0.4
-0.8
-1.2
Quadrature Amplitude
-2
-1
0
1
2
In-phase Amplitude
Stopped

ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
-40
-60
-80
-100
-120
-140
-160
-180
-200
-220
dBm
-90
-60
-30
0
30
60
90
Frequency (MHz)
Stopped
Sample Rate = 200.000 MHz Frames = 0 T = 0.00000

ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
dBm
-40
-50
-60
-70
-80
-90
-100
-110
-120
-130
-140
-80
-40
0
40
80
Frequency (MHz)
Stopped
Sample Rate = 200.000 MHz Frames = 0 T = 0.00000


PLOT
MEASUREMENTS
Spatial stream 1
EVM (dB)
-9
-12
-15
-18
-21
-24
-27
-30
-120
-100
-80
-60
-40
-20
0
20
40
60
80
100
120
Subcarrier index
Stopped
Frames = 1



Compared to the case without interference, the constellation diagram is more distorted and the overall EVM is around -17 dB.

The ACR is around 38 dB. You can also measure the ACR when the interferer is an HE waveform. In this case, to measure the ACR when the PER is approximately 10% for a PSDU length of 4096 octets, IEEE Std 802.11ax-2021, set the **Gain** value of the Interferer block to around -72.4 dB.

Summary and Further Exploration

This example demonstrates how to model and test the reception of an HE waveform coexisting with an NR waveform or another HE waveform. The RF receiver consists of bandpass filters, amplifiers, and a demodulator. To evaluate the impact of the NR interference, the example modifies the gain of the NR waveform and performs EVM, PER, and ACR measurements. You can explore the impact of altering the RF impairments. For example:

- Increase the phase noise by using **Phase noise offset (Hz)** and **Phase noise level (dBc/Hz)** parameters on the **Demodulator** tab of the RF Receiver Subsystem block.
- Decrease the LO to RF isolation by using the **LO to RF isolation (dB)** parameter on the **Demodulator** tab of the RF Receiver Subsystem block.

This example configures the RF Receiver Subsystem block to work with the default values of the WLAN 802.11ax and 5G NR Test Model blocks and with the HE and NR carriers centered at 5950 MHz and 5970 MHz, respectively. These carriers are within the IEEE 802.11 HE frequency bands (between 1 GHz and 7.125 GHz) and the NR operating band n96, according to IEEE Std 802.11ax-2021 and TS 38.101-1, respectively. If you change the carrier frequencies or the waveform

configurations, check if you need to update the parameters of the RF Receiver Subsystem block as these parameters have been selected to work for the default configuration of the example. For instance, a change in the HE carrier frequency requires revising the **Passband frequencies** and **Stopband frequencies** parameters of the RF Bandpass Filter block inside the RF Receiver. If you increase the waveform bandwidth, check if you need to update the **Impulse response duration** and **Phase noise frequency offset (Hz)** parameters of the Demodulator (RF Blockset) block. The phase noise offset determines the lower limit of the impulse response duration. If the phase noise frequency offset resolution is high for a given impulse response duration, a warning message appears, specifying the minimum duration suitable for the required resolution.

You can use this example as the basis for testing the coexistence between HE and NR or HE waveforms for different RF configurations. You can replace the RF Receiver Subsystem block with another RF subsystem and then configure the model accordingly.

You can also use this example as the basis for performing the coexistence tests defined in IEEE/ANSI C63.27-2017, the American National Standard for Evaluation of Wireless Coexistence. Examples of those tests are:

- Wi-Fi devices operating in the 2.4 GHz ISM band (2.4 to 2.4835 GHz), where the unintended signals (or interferers) are 802.11n or LTE.
- Wi-Fi devices operating in the 5 GHz UNII & ISM bands (5.15 to 5.85 GHz), where the unintended signal (or interferer) is 802.11n.

To transmit the 802.11n or LTE interferers, open the **WLAN Waveform Generator** app or the **LTE Waveform Generator (LTE Toolbox)** app, respectively, and export a new block. To change the frequency band, modify the **HE carrier frequency (MHz)** parameter in the RF Receiver Subsystem block mask. You can also place the adjacent unintended waveform at a different frequency by updating the **Frequency offsets (Hz)** parameter in the Multiband Combiner block.

Bibliography

1. 3GPP TS 38.141-1. "NR; Base Station (BS) conformance testing Part 1: Conducted conformance testing." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network.*
2. IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.
3. 3GPP TS 38.101-1. "NR; User Equipment (UE) radio transmission and reception." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network.*
4. IEEE/ANSI C63.27-2017 - American National Standard for Evaluation of Wireless Coexistence

Modeling and Testing an 802.11ax RF Transmitter

This example shows how to characterize the impact of RF impairments in an 802.11ax transmitter. The example generates a baseband IEEE® 802.11ax™ waveform by using WLAN Toolbox™ and models the RF transmitter by using RF Blockset™.

Introduction

This example characterizes the impact of RF impairments such as in-phase and quadrature (IQ) imbalance, phase noise, and power amplifier (PA) nonlinearities on the transmission of an 802.11ax waveform. To evaluate the impact of these impairments, the example performs these measurements:

- Error vector magnitude (EVM): vector difference at a given time between the ideal (transmitted) signal and the measured (received) signal
- Spectral mask: test that ensures a transmission in one channel does not cause substantial interference in adjacent channels
- Occupied bandwidth: bandwidth that contains 99% of the total integrated power of the signal, centered on the assigned channel frequency
- Channel power: filtered mean power centered on the assigned channel frequency
- Complementary cumulative distribution function (CCDF): probability that the signal's instantaneous power is at a specified level above its average power
- Peak-to-average power ratio (PAPR): relation between the peak power of the signal and its average power

The example works on a packet-by-packet basis and uses a Simulink model to perform these steps:

1. Generate the baseband 802.11ax waveform by using WLAN Toolbox features.
2. Oversample and filter the waveform by using an FIR Interpolation block.
3. Import the baseband waveform into the RF Transmitter Subsystem block implemented by using RF Blockset blocks. The model uses an RF intermediate frequency to carry the baseband information in RF Blockset.
4. Model the effects of upconverting the waveform to the carrier frequency by using the RF Transmitter Subsystem block. This block models the impairments introduced by an RF transmitter using RF Blockset blocks.
5. Calculate the occupied bandwidth and channel power and depict the spectral mask by using the Spectrum Analyzer block.
6. Compute the CCDF and PAPR by using the CCDF and PAPR block.
7. Downsample and filter the waveform by using an FIR Decimation block.
8. Extract the data symbols and measure the EVM by demodulating the baseband waveform.

The Simulink model uses WLAN Toolbox and DSP System Toolbox™ features to process the baseband signal (steps 1, 2, and 5-8) and uses RF Blockset blocks to model the RF transmitter (steps 3 and 4). This model supports Normal and Accelerator simulation modes.

Simulink Model Structure

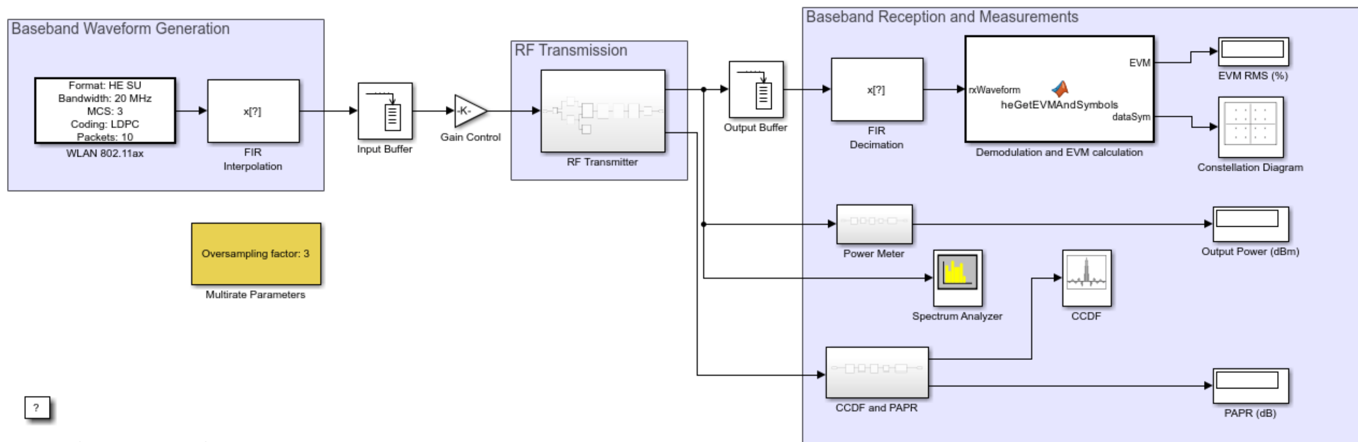
The model contains three main parts:

- Baseband Waveform Generation: generates the baseband 802.11ax waveforms
- RF Transmission: models the effects of upconverting the waveform to the carrier frequency
- Baseband Waveform Reception: performs the RF measurements and calculates EVM by demodulating the baseband waveform

```
modelName = 'HERFTransmitterModel';
open_system(modelName);
```

802.11ax RF Transmitter

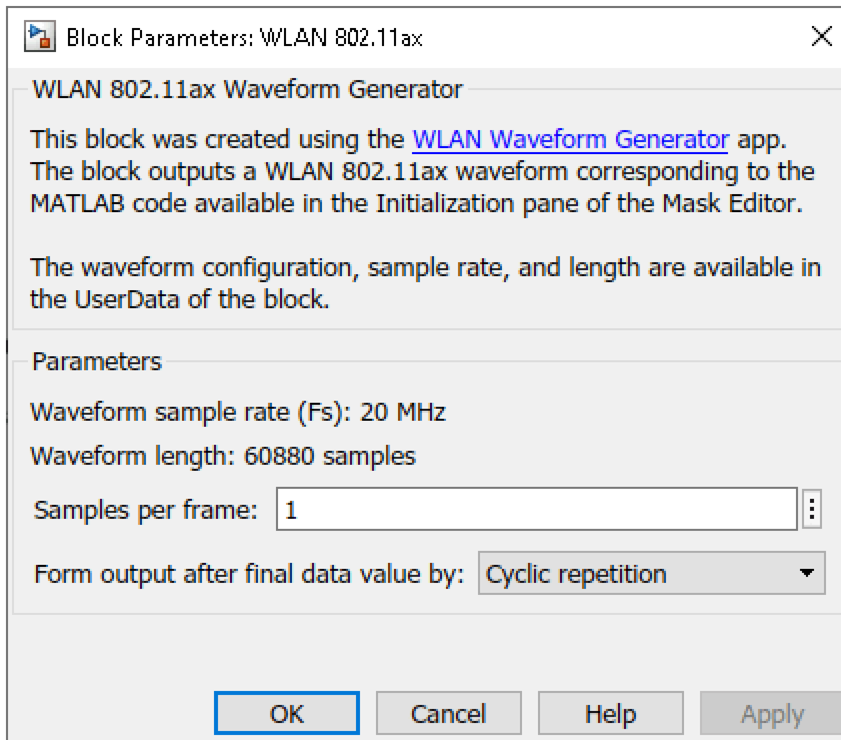
Input waveform: 802.11ax (HE)
Measurements: EVM, channel power, occupied bandwidth, spectral mask, CCDF, PAPR



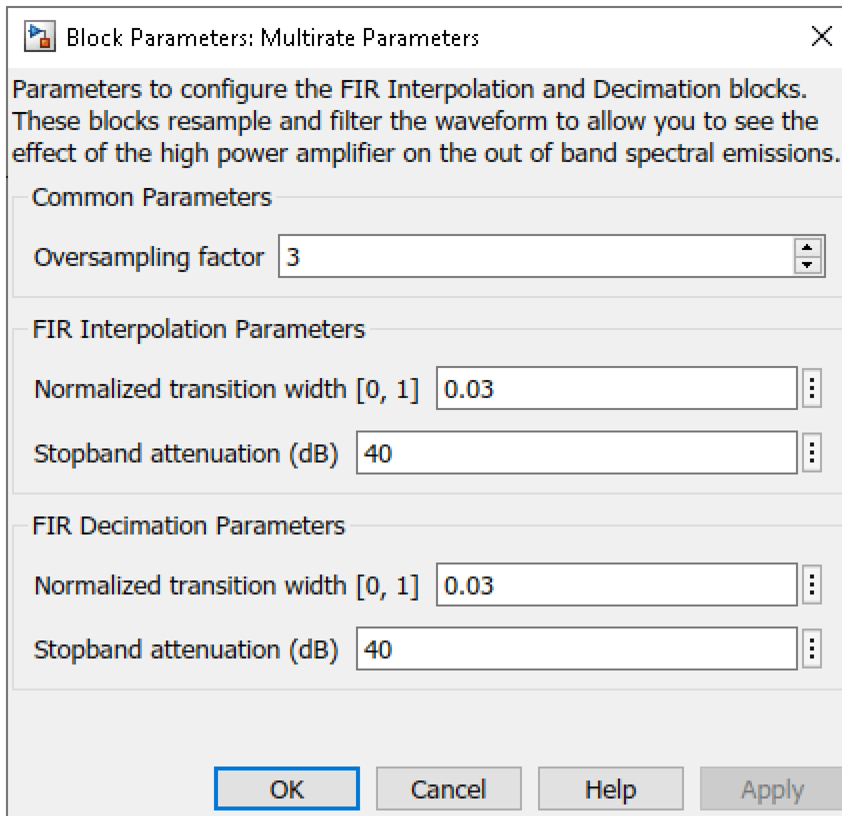
Copyright 2020-2022 The MathWorks, Inc.

Baseband Waveform Generation

The WLAN 802.11ax block generates standard-compliant high-efficiency single-user (HE SU) waveforms, according to IEEE Std 802.11ax™-2021. You can generate this block using the **WLAN Waveform Generator** app. You can access the waveform configuration parameters in the user data of the block. This example uses the **InitFcn** in the **Model callbacks** to store the structure available in the user data in a Base Workspace variable, `HEInfo`. For more information about this block, see Waveform From Wireless Waveform Generator App.



To display the effect of the high-power amplifier (HPA) on the out-of-band spectral emissions, the FIR Interpolation block oversamples and filters the waveform. At the output of the RF Transmitter Subsystem block, the FIR Decimation block downsamples the waveform back to its original sampling rate. The Multirate Parameters block provides an interface to configure the parameters of the FIR Interpolation and Decimation blocks.



Specifying Simulation Time

Set the **Stop Time** value in the Simulink model to the time required to transmit and obtain the EVM results and constellation diagram of, at least, one packet. Considering the waveform configuration selected in the WLAN 802.11ax block, the packet transmission time in this example is 304.4 microseconds. As the filters in the FIR Interpolation and Decimation blocks introduce a delay, you can use the **IdleTime** parameter in the Mask Editor of the WLAN 802.11ax block to compensate for the delay.

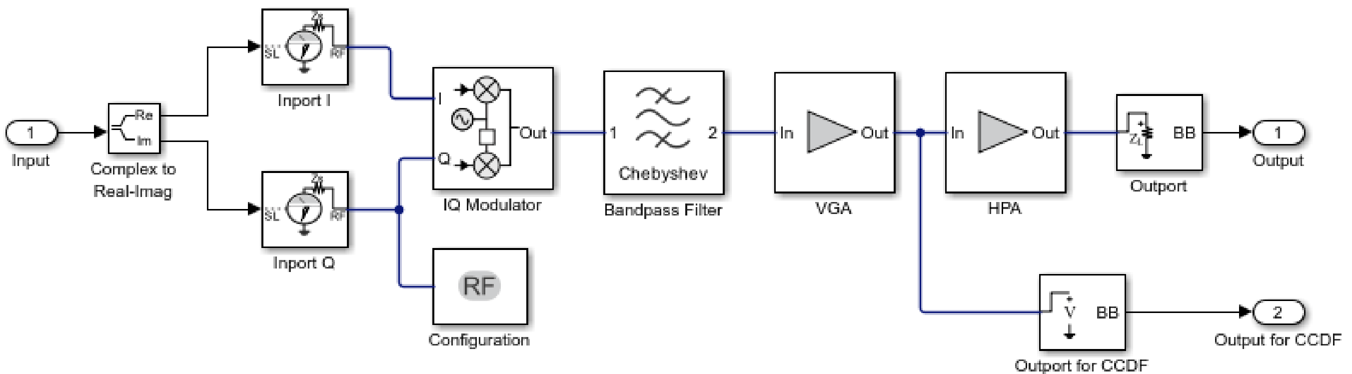
RF Transmission

The RF Transmitter Subsystem block is based on a superheterodyne transmitter architecture. This architecture models the effects of upconverting the waveform to the carrier frequency by characterizing these RF components:

- IQ modulator consisting of mixers, a phase shifter, and a local oscillator
- Bandpass filter
- Power amplifier

In addition to these components, this RF Transmitter Subsystem block also includes a variable gain amplifier (VGA) to control the input back-off (IBO) level of the HPA.

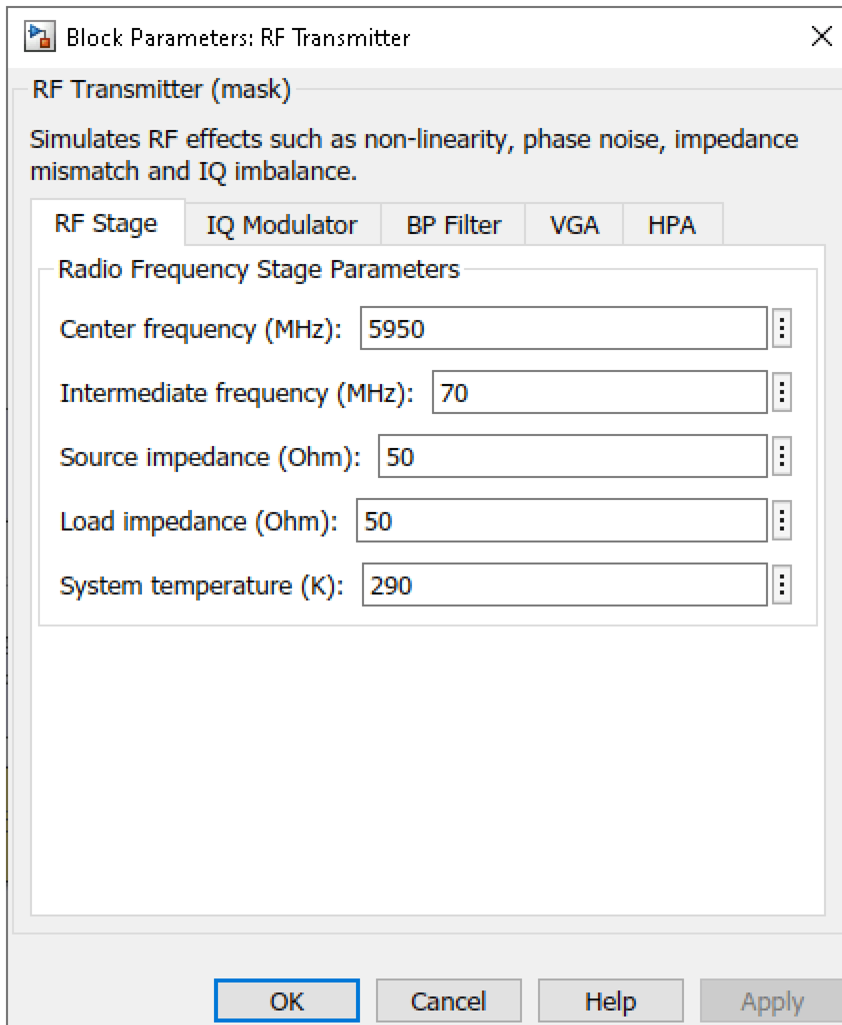
```
set_param(modelName,'Open','off');
RFTransmitterBlock = [modelName '/RF Transmitter'];
set_param(RFTransmitterBlock,'Open','on');
```



Use an Input Buffer block to send one sample at a time to the RF Transmitter Subsystem block.

The Inport block inside the RF Transmitter Subsystem block converts the Simulink complex baseband waveform into the RF Blockset Circuit Envelope simulation environment. The **Carrier frequencies** parameter of the Inport block specifies the center frequency of the carrier in the RF Blockset domain. The Outport block converts the RF Blockset signal back into Simulink complex baseband.

You can configure the RF Transmitter components by using the RF Transmitter Subsystem block mask.



The RF Transmitter Subsystem block models typical impairments, including:

- I/Q imbalance as a result of gain or phase mismatches between the parallel sections of the transmitter chain dealing with the IQ signal paths
- Phase noise as an effect directly related to the thermal noise within the active devices of the oscillator
- PA nonlinearities due to DC power limitation when the amplifier works in the saturation region

Adapt the power level of the baseband waveform to the RF configuration by adding a Gain Control block after the Input Buffer block.

As the current RF Transmitter Subsystem block configuration sends one sample at a time, the Output Buffer block (after the RF Transmitter Subsystem block) collects all samples within an HE SU packet before sending the samples onto the Demodulation and EVM calculation block.

Baseband Waveform Reception

The Demodulation and EVM calculation block recovers and plots the HE Data symbols in the Constellation Diagram block by performing frequency and packet offset corrections, channel

estimation, pilot phase tracking, OFDM demodulation, and equalization. This block also performs these EVM measurements:

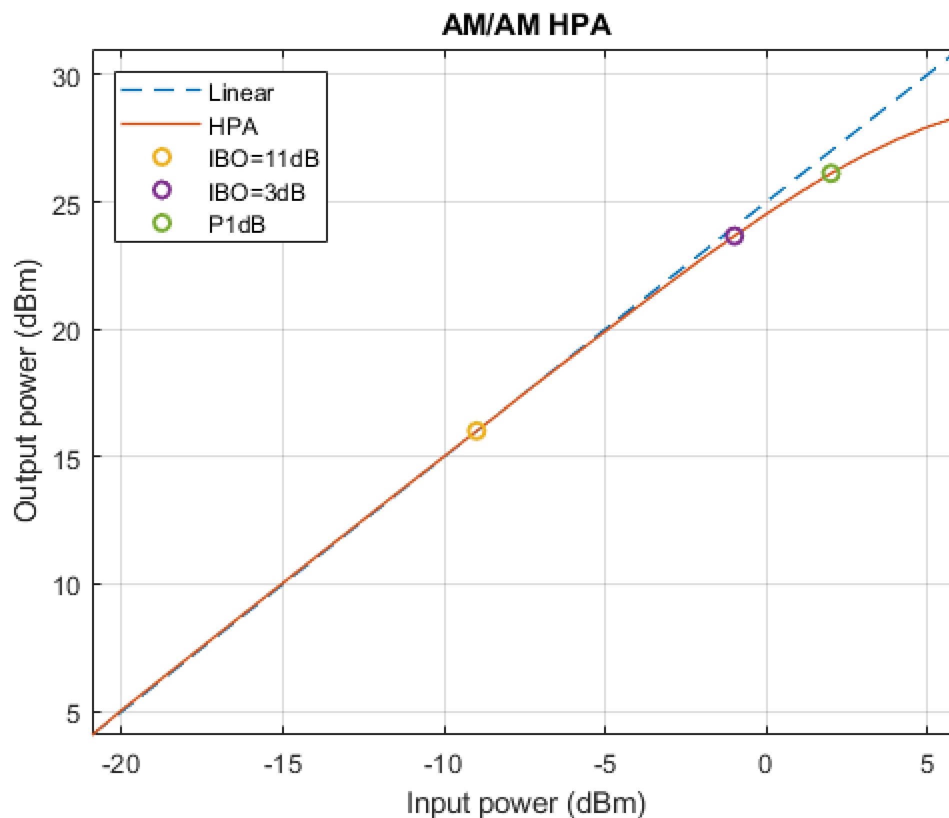
- EVM per subcarrier (dB): EVM averaged over the allocated HE Data symbols within a subcarrier
- EVM per OFDM symbol (dB)
- Overall EVM (dB and %): EVM averaged over all transmitted HE Data symbols

The Spectrum Analyzer block depicts the spectral mask according to IEEE Std 802.11ax™-2021, Section 27.3.19.1. The CCDF and PAPR block, connected at the input of the HPA block, depicts the CCDF and PAPR measurements. The Power Meter block measures the RF waveform channel power, which is displayed in the Output Power (dBm) block.

Effect of Power Amplifier Nonlinearities

To characterize the impact of HPA nonlinearities in the EVM evaluation, you can measure the amplitude-to-amplitude modulation (AM/AM) of the HPA. The AM/AM refers to the output power levels in terms of the input power levels. The helper function `hePlotHPACurve` displays the AM/AM characteristic of the HPA selected for this model.

```
hePlotHPACurve();
figHPA = gcf;
```

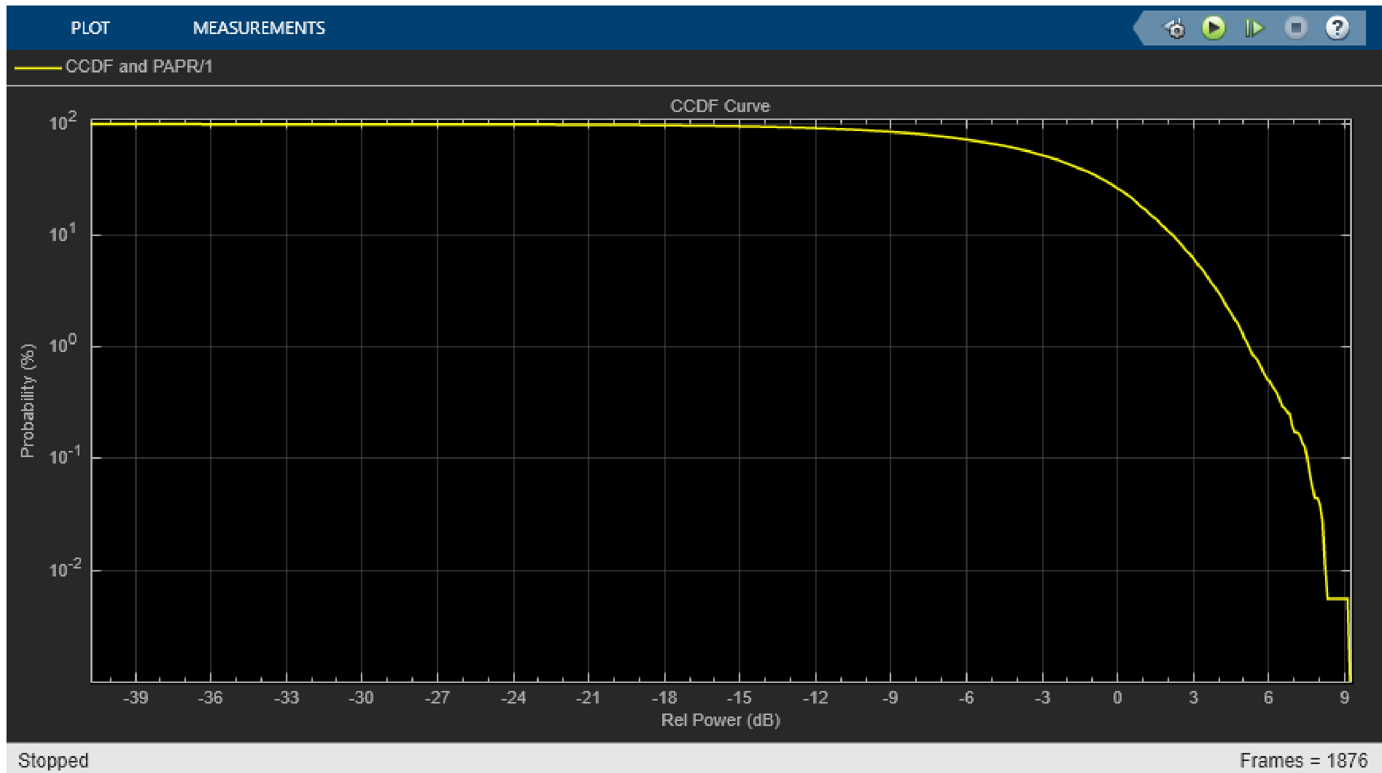


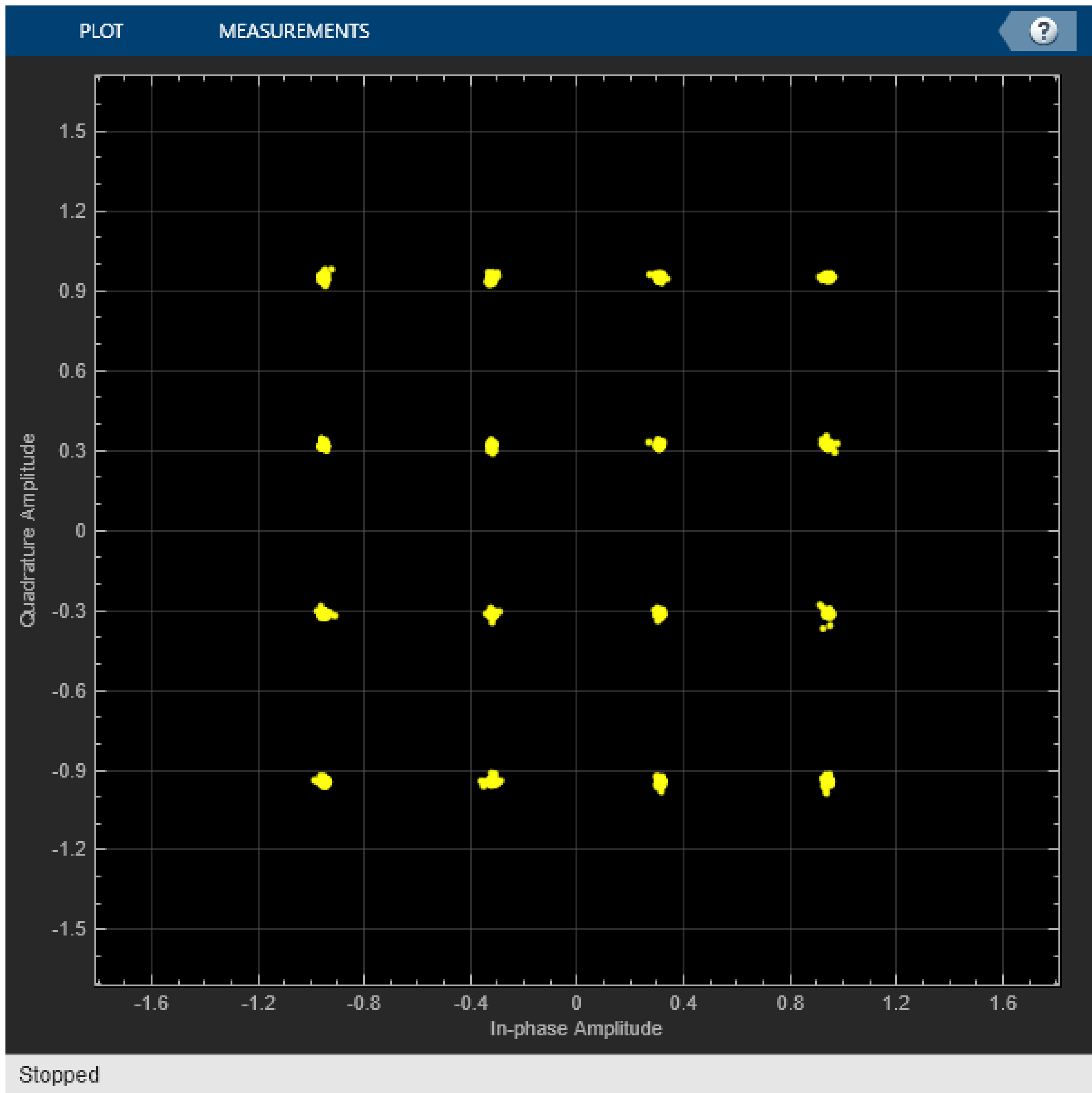
P1dB is the power at 1 dB compression point and is normally used as a reference when selecting the IBO level of the HPA. You can see the HPA impact on the RF Transmitter Subsystem block by analyzing the EVM results for different operating points of the HPA. For example, compare the case when IBO = 11 dB, corresponding to HPA operating in the linear region, with the case when IBO = 3

dB, corresponding to HPA operating in saturation. The gain of the VGA controls the IBO level. To keep a VGA linear behavior using the default parameters, select gain values lower than 15 dB.

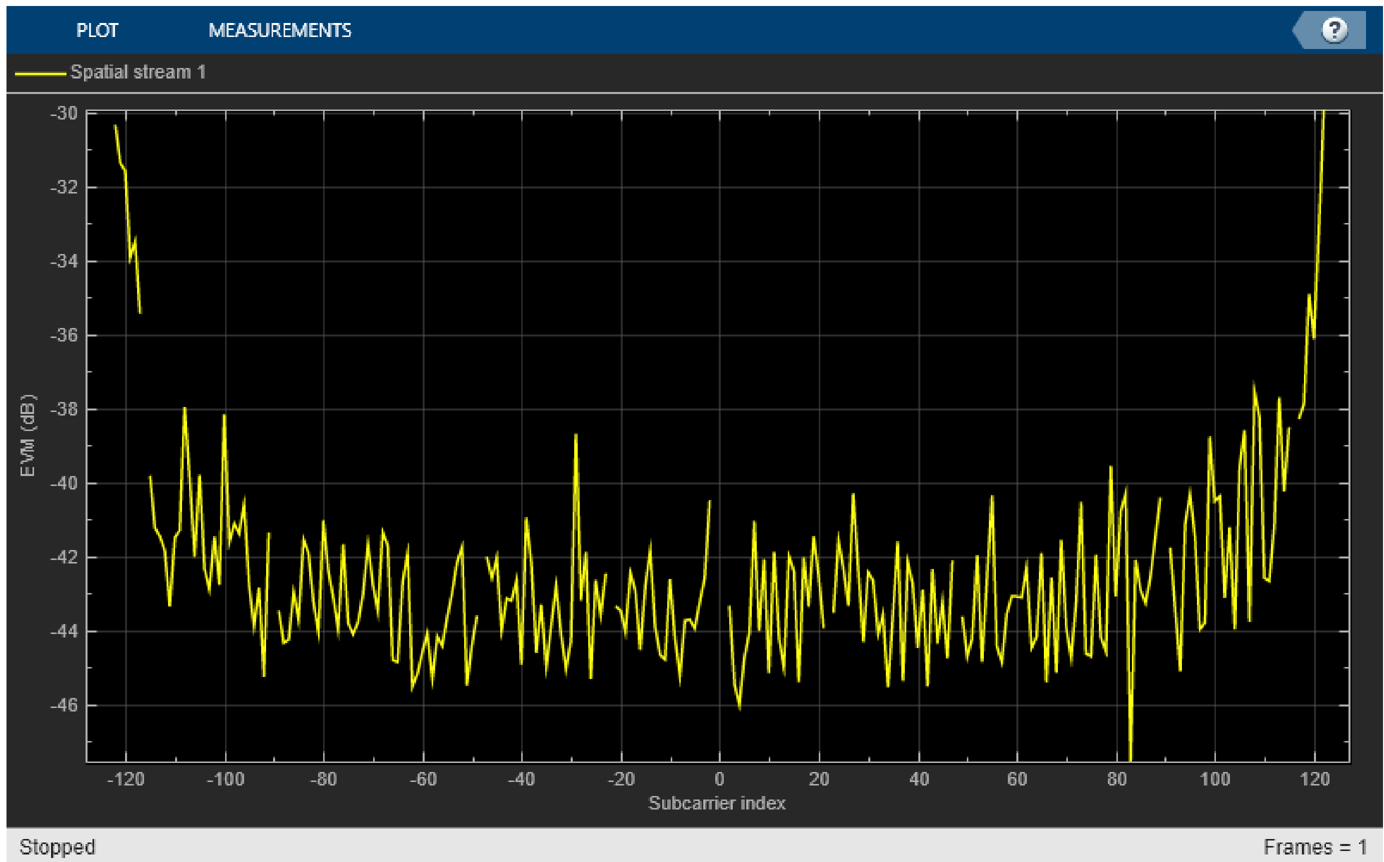
- *Linear HPA (IBO = 11 dB)*. To operate at an IBO level of 11 dB, set the **Available power gain** parameter of the VGA block to 5 dB. To calculate the EVM and plot the constellation diagram, run the simulation to capture one packet (Stop Time equal to 304.4 us for the default configuration).

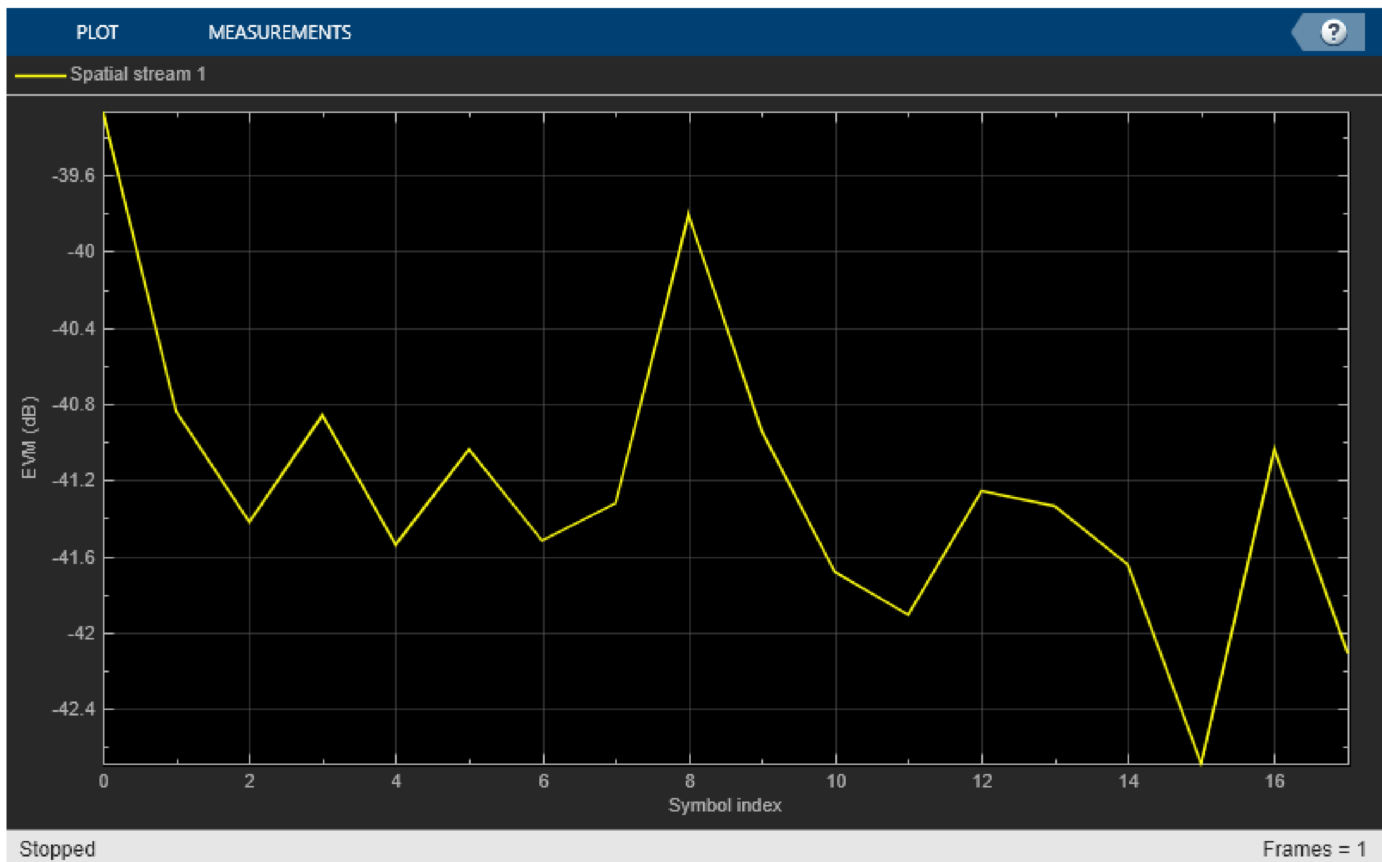
```
set_param(RFTransmitterBlock,'vgaGain','5');
sim(modelName);
```




PLOT
MEASUREMENTS
Quadrature Amplitude
In-phase Amplitude
1.5
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
-1.5
-1.6
-1.2
-0.8
-0.4
0
0.4
0.8
1.2
1.6
Stopped

ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
dBm
-20
-10
0
10
20
Frequency (MHz)
Stopped
Sample Rate = 60.0000 MHz Frames = 0 T = 0.00000

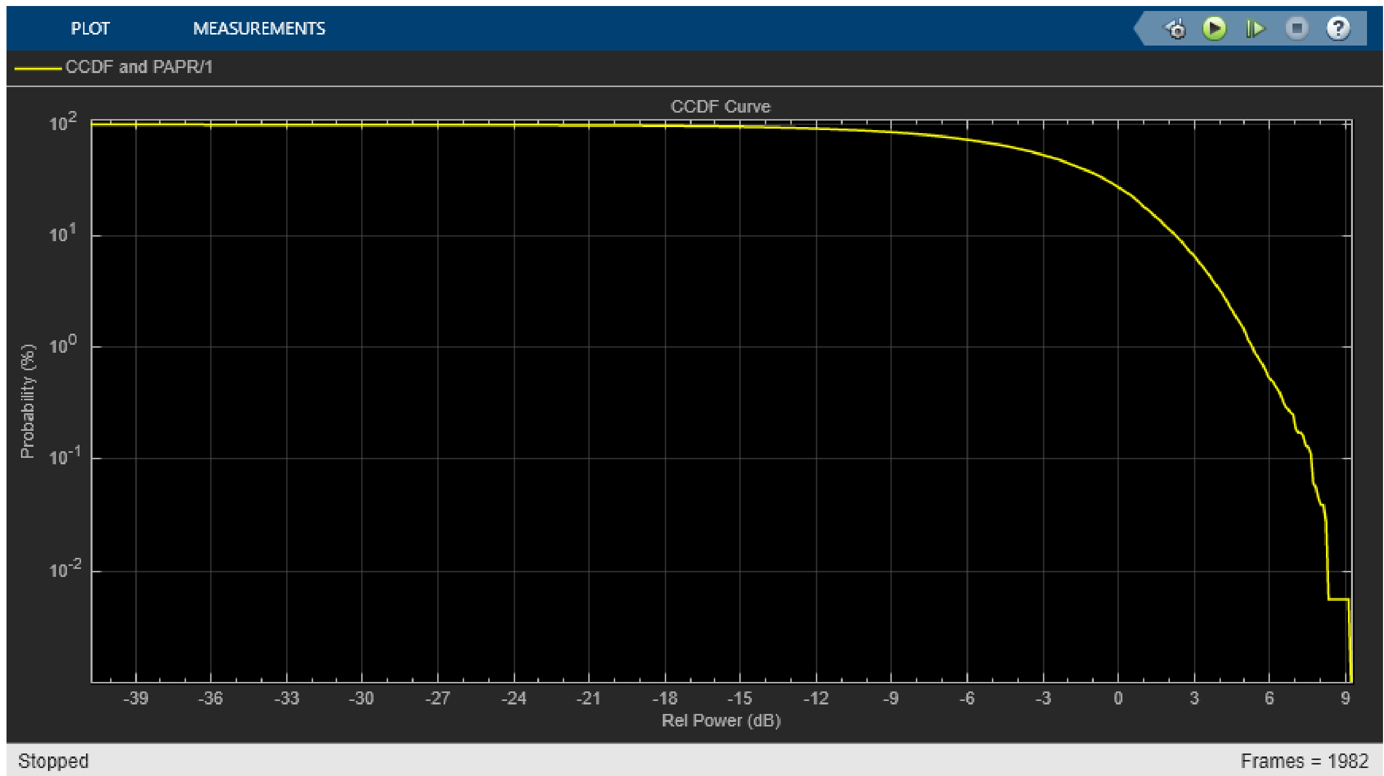

PLOT
MEASUREMENTS
Spatial stream 1
EVM (dB)
Subcarrier index
Stopped
Frames = 1



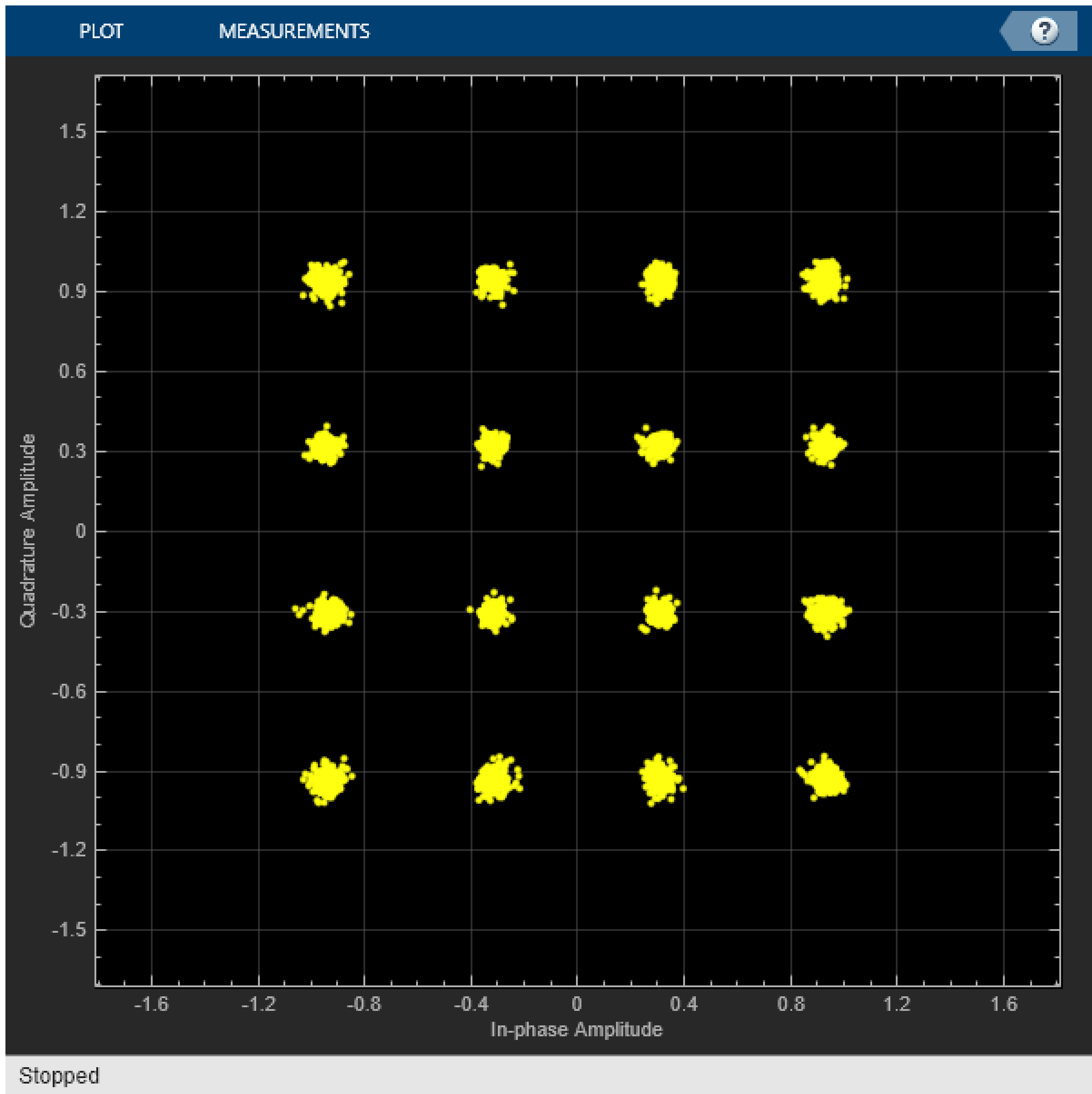
According to IEEE Std 802.11ax™-2021, Table 27-49, the allowed relative constellation error (EVM) in an HE SU PPDU when the dual carrier modulation, or **DCM** parameter in the WLAN 802.11ax block Mask Editor, is equal to `false` and the Modulation/coding, or **MCS** parameter in the WLAN 802.11ax block Mask Editor, is equal to 3 (16-QAM, 1/2) is -16 dB. As the overall EVM, around -41 dB, is lower than -16 dB, this architecture falls within the requirements of IEEE Std 802.11ax™-2021.

- *Nonlinear HPA (IBO = 3 dB)*. To operate at an IBO level of 3 dB, set the **Available power gain** parameter of the VGA block to 13 dB.

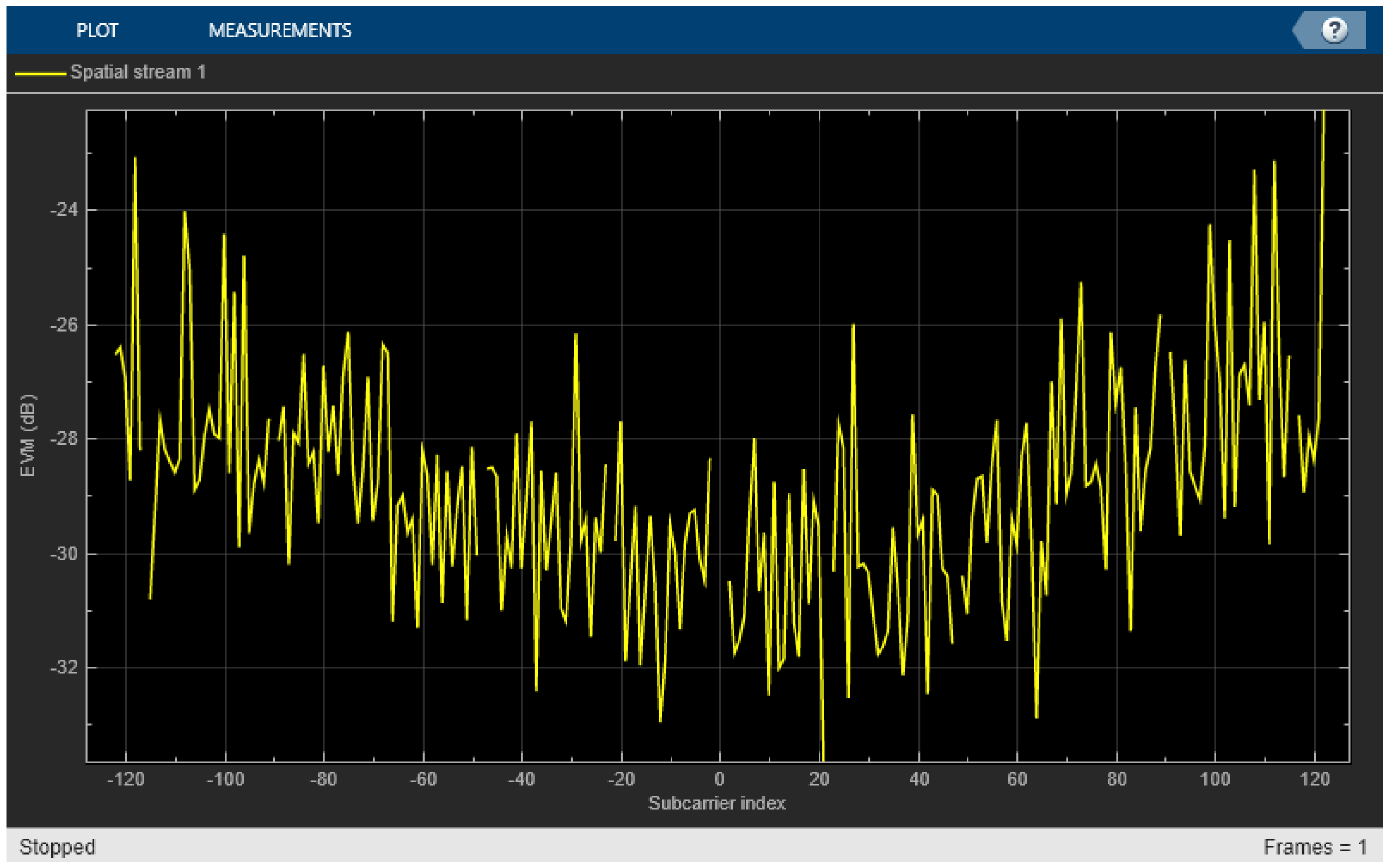
```
set_param(RFTransmitterBlock,'vgaGain','13');
sim(modelName);

% Restore to default parameters
set_param(RFTransmitterBlock,'vgaGain','5');
```

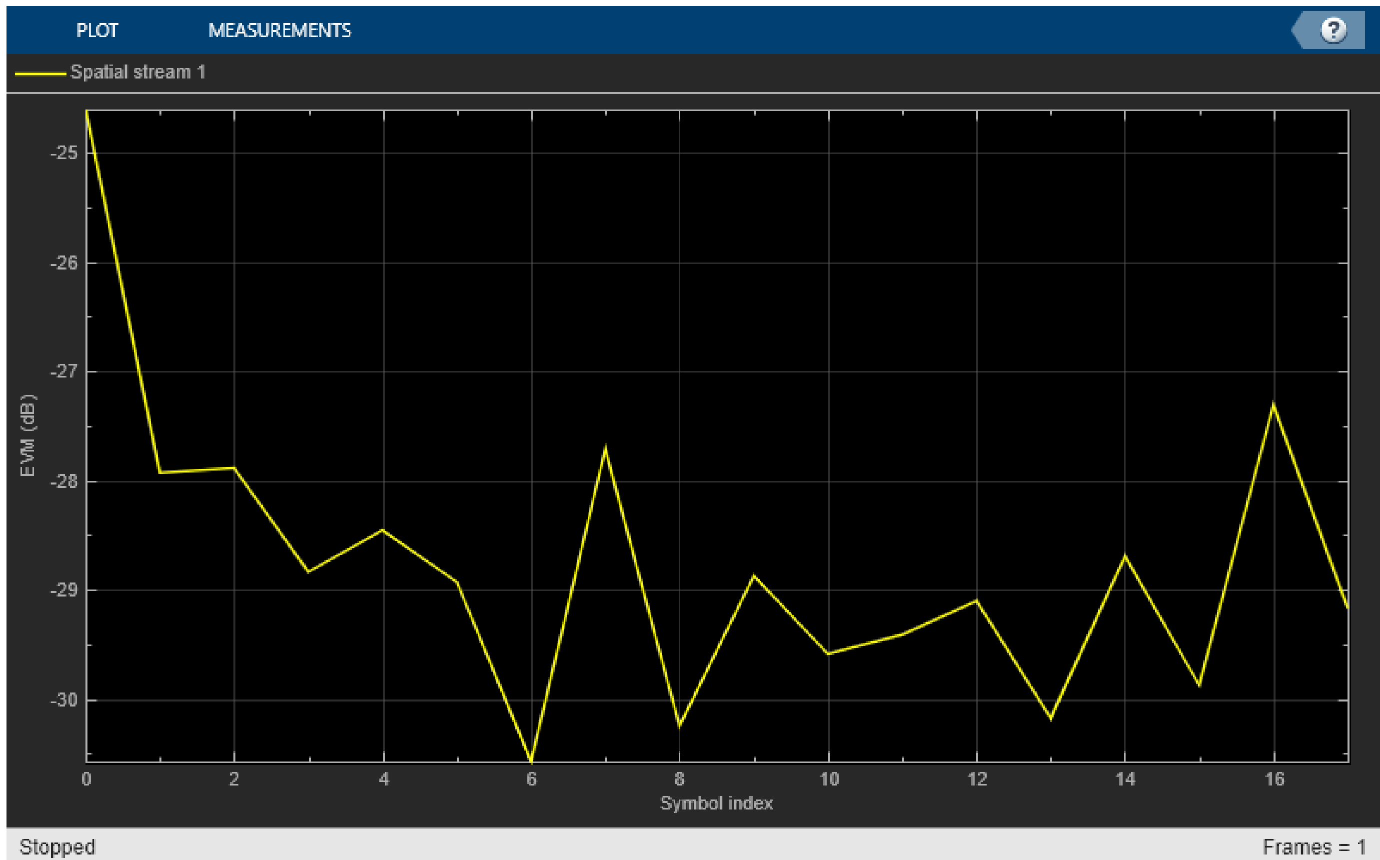


PLOT
MEASUREMENTS
CCDF and PAPR/1
CCDF Curve
Probability (%)
Rel Power (dB)
Stopped
Frames = 1982


PLOT
MEASUREMENTS
1.5
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
-1.5
Quadrature Amplitude
-1.6
-1.2
-0.8
-0.4
0
0.4
0.8
1.2
1.6
In-phase Amplitude
Stopped

ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
dBm
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
-20
-10
0
10
20
Frequency (MHz)
Stopped
Sample Rate = 60.0000 MHz Frames = 0 T = 0.00000


PLOT
MEASUREMENTS
Spatial stream 1
EVM (dB)
-24
-26
-28
-30
-32
-120
-100
-80
-60
-40
-20
0
20
40
60
80
100
120
Subcarrier index
Stopped
Frames = 1



Compared to the previous case, the constellation diagram is more distorted. In terms of measurements, the overall EVM, around -28 dB, is still lower than -16 dB, so it also falls within the requirements of IEEE Std 802.11ax™-2021.

Summary and Further Exploration

This example demonstrates how to model and test the transmission of an 802.11ax waveform. The RF Transmitter Subsystem block consists of a bandpass filter, amplifiers and an IQ modulator. The example highlights the effect of HPA nonlinearities on the performance of the RF Transmitter Subsystem block. You can explore the impact of altering other impairments as well. For example:

- Increase I/Q imbalance by using the **I/Q gain mismatch (dB)** and **I/Q phase mismatch (Deg)** parameters on the **IQ Modulator** tab of the RF Transmitter Subsystem block.
- Increase the phase noise by using **Phase noise offset (Hz)** and **Phase noise level (dBc/Hz)** parameters on the **IQ Modulator** tab of the RF Transmitter Subsystem block.

The RF Transmitter Subsystem block is configured to work with the current HE waveform parameters selected in the WLAN 802.11ax block and with the RF carrier centered at 5950 MHz. This carrier is within the IEEE 802.11 HE STA frequency bands (between 1 GHz and 7.125 GHz, according to IEEE Std 802.11ax™-2021. If you modify the **Center frequency (MHz)** parameter of the RF Transmitter Subsystem block or the waveform configuration of the WLAN 802.11ax block, check if you need to update the parameters of the RF Transmitter components and the FIR filters as these parameters are set to work with the current example configuration. For instance, a change in the carrier frequency requires revising the **Passband frequencies** and **Stopband frequencies** parameters of the Bandpass Filter block inside the RF Transmitter. If you increase the waveform bandwidth, check if

you need to update the **Impulse response duration** and **Phase noise frequency offset (Hz)** parameters of the IQ Modulator (RF Blockset) block. The phase noise offset determines the lower limit of the impulse response duration. If the phase noise frequency offset resolution is high for a given impulse response duration, a warning message appears, specifying the minimum duration suitable for the required resolution.

You can use this example as the basis for testing HE waveforms for different RF configurations. You can replace the RF Transmitter Subsystem block by another RF subsystem and then configure the model accordingly.

To use a different HE SU waveform, open the **WLAN Waveform Generator** app, select the HE SU configuration, and export a new block. For more information on how to generate and use this block, see "Generate Wireless Waveform in Simulink Using App-Generated Block" on page 8-62.

References

1 IEEE Std 802.11ax™-2021. 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 1: Enhancements for High-Efficiency WLAN.

802.11ac Receiver Minimum Input Sensitivity Test

This example shows how to measure the receiver minimum input sensitivity as specified in Section 21.3.18.1 of the IEEE Std 802.11™-2020 standard [1].

Introduction

The receiver minimum sensitivity test ensures a device under test (DUT) receives data with a defined maximum packet error rate (PER) of 10% at a defined minimum signal power. The minimum signal power depends on the channel bandwidth and modulation and coding scheme (MCS) as specified in Table 21-25 of IEEE Std 802.11™-2020 [1]:

MCS	Modulation	Rate	Minimum Sensitivity [dBm]			
			20 MHz	40 MHz	80 MHz	160 MHz
0	BPSK	1/2	-82	-79	-76	-73
1	QPSK	1/2	-79	-76	-73	-70
2	QPSK	3/4	-77	-74	-71	-68
3	16-QAM	1/2	-74	-71	-68	-65
4	16-QAM	3/4	-70	-67	-64	-61
5	64-QAM	2/3	-66	-63	-60	-57
6	64-QAM	3/4	-65	-62	-59	-56
7	64-QAM	5/6	-64	-61	-58	-55
8	256-QAM	3/4	-59	-56	-53	-50
9	256-QAM	5/6	-57	-54	-51	-48

When the test is performed with hardware, each input antenna port on the DUT is connected through a cable to a single output antenna port of a transmitter. To perform the test, specify these parameters for the test waveform:

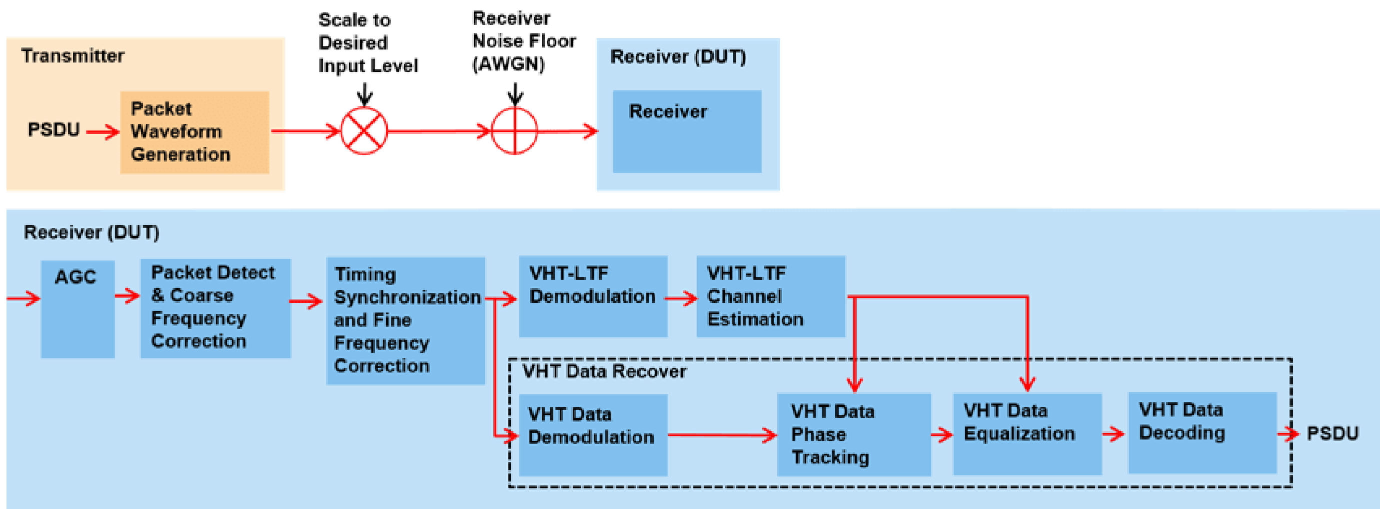
- Number of spatial streams - equal to the number of transmit antennas
- PSDU length, in bytes - 4096
- Space-time block coding (STBC) - disabled
- Guard interval, in nanoseconds - 800
- Channel coding - binary convolutional coding (BCC)

This example shows how to simulate the test by using WLAN Toolbox™. VHT packets stimulate a receiver at a range of input levels below the minimum sensitivity level. The example then measures the packet error rate for each sensitivity level.

The example simulates the test by performing these steps over a range of sensitivity levels:

- Generate and scale packets to the desired signal level
- Add white Gaussian noise is to create a noise floor at the receiver
- Demodulate the noisy packets to recover PSDUs.
- Compare recovered PSDUs to those transmitted to determine the number of packet errors and hence the packet error rate.

Automatic gain control (AGC), packet detection, timing synchronization, carrier frequency offset correction, noise estimation and phase tracking are performed by the example receiver. This diagram demonstrates processing for each packet:



Test Parameters

Configure a transmission for the test by using a VHT configuration object. This example measures the minimum sensitivity for a 160 MHz transmission with 64-QAM rate 5/6 modulation and coding. The simulated DUT has 2 receive antennas. Test different configurations by changing these parameters.

```
cfgVHT = wlanVHTConfig;              % Create VHT transmission configuration
cfgVHT.ChannelBandwidth = 'CBW160'; % Bandwidth
cfgVHT.MCS = 7;                      % 64-QAM, rate 5/6
NumReceiveAntennas = 2;              % Number of receive antennas
```

The test requires these fixed transmission parameters.

```
cfgVHT.APEPLength = 4096; % Bytes
cfgVHT.STBC = false;
cfgVHT.NumTransmitAntennas = NumReceiveAntennas;
cfgVHT.NumSpaceTimeStreams = NumReceiveAntennas;
cfgVHT.SpatialMapping = 'Direct';
cfgVHT.GuardInterval = 'Long';
```

Simulation Parameters

A receiver processes VHT packets at a range of input levels below the minimum input sensitivity level. Specify the range of offsets to test in the vector `testInputLevelOffsets`.

```
testInputLevelOffsets = [-10 -9 -8 -7]; % dB
```

Control the number of packets tested at each sensitivity by specifying these parameters:

1. `maxNumErrors` is the maximum number of packet errors simulated at each input level. When the number of packet errors reaches this limit, the simulation at this sensitivity level is complete.
2. `maxNumPackets` is the maximum number of packets simulated at each input level and limits the length of the simulation if the packet error limit is not reached.

The numbers chosen in this example lead to a very short simulation. Increase `maxNumErrors` and `maxNumPackets` for meaningful results.

```
maxNumErrors = 20;
maxNumPackets = 200;
```

Signal Power Setup

The minimum sensitivity test specifies a maximum PER for a measured input level per receive antenna. In this simulation the receiver processes a test signal with a specified input level in dBm. Generate the test signal using the `wlanWaveformGenerator` function. The `wlanWaveformGenerator` function normalizes the waveform such that the power for all antennas sums to 0 dBm. Therefore, scale the output of the waveform generator to create the desired input level.

```
% Receiver minimum input level sensitivity for 20 MHz, Table 21-25. The
% sensitivity increases by 3dB for double the bandwidth.
rxMinSensitivityTable = [-82 -79 -77 -74 -70 -66 -65 -64 -59 -57]; % dBm

% Get minimum input sensitivity given MCS and bandwidth
fs = wlanSampleRate(cfgVHT);  % Baseband sampling rate (Hz)
B = floor(10*log10((fs/20e6))); % Scalar for bandwidth
rxMinSensitivity = rxMinSensitivityTable(cfgVHT.MCS+1)+B; % dBm
disp(['Minimum sensitivity for MCS' num2str(cfgVHT.MCS) ', ' ...
    num2str(fs/1e6) ' MHz: ' num2str(rxMinSensitivity,'%2.1f') ' dBm'])
```

```
Minimum sensitivity for MCS7, 160 MHz: -55.0 dBm
```

Define the range of input levels below the minimum level to test using `testInputLevels`.

```
testInputLevels = rxMinSensitivity+testInputLevelOffsets; % dBm
```

Calculate a voltage scalar, `A`, to scale the generated waveform for each test level. The power per receive antenna port is measured during the simulation to confirm the input signal level is correct.

```
A = 10.^((testInputLevels-30)/20);      % Voltage gain (attenuation)
A = A*sqrt(cfgVHT.NumTransmitAntennas); % Account for generator scaling
```

Noise Configuration

Add thermal noise at the receiver. The height of the noise floor determines the SNR at the receiver, as the input signal level is fixed for this test. The noise figure of the receiver determines the level of noise floor.

```
NF = 6;         % Noise figure (dB)
T = 290;        % Ambient temperature (K)
BW = fs;        % Bandwidth (Hz)
k = 1.3806e-23; % Boltzmann constant (J/K)
noiseFloor = 10*log10(k*T*BW)+NF; % dB
disp(['Receiver noise floor: ' num2str(noiseFloor+30,'%2.1f') ' dBm'])
```

```
Receiver noise floor: -85.9 dBm
```

Add noise to the waveform using an AWGN channel, `comm.AWGNChannel`.

```
awgnChannel = comm.AWGNChannel('NoiseMethod','Variance', ...
    'Variance',10^(noiseFloor/10));
```

Input Level Sensitivity Simulation

Calculate the packet error rate for each input level by simulating multiple packets.

For each packet perform the following processing steps:

1. Create and encode a PSDU to create a single packet waveform.
2. Create the desired input level in dBm by scaling the waveform.
3. Measure the power of the received waveform.
4. Add AWGN to the received waveform.
5. Boost the signal prior to processing by passing through an automatic gain control.
6. Detect the packet.
7. Estimate and correct coarse carrier frequency offset.
8. Establish fine timing synchronization.
9. Estimate and correct fine carrier frequency offset.
10. Extract and OFDM demodulate the VHT-LTF and perform channel estimation.
11. Extract the VHT Data field and recover the PSDU.

```
ind = wlanFieldIndices(cfgVHT); % For accessing fields within the packet
chanBW = cfgVHT.ChannelBandwidth;
rng(0);  % Set random state for repeatability

agc = comm.AGC; % Automatic gain control

S = numel(testInputLevels);
packetErrorRate = zeros(S,1);
rxAntennaPower = zeros(S,1);
for i=1:S
    disp(['Simulating ' num2str(testInputLevels(i),'%2.1f') ...
        ' dBm input level...']);

    % Loop to simulate multiple packets
    numPacketErrors = 0;
    measuredPower = zeros(maxNumPackets,1); % Average power per antenna
    numPkt = 1; % Index of packet transmitted
    while numPacketErrors<=maxNumErrors && numPkt<=maxNumPackets
        % Generate a packet waveform
        txPSDU = randi([0 1],cfgVHT.PSDULength*8,1); % PSDULength in bytes
        tx = wlanWaveformGenerator(txPSDU,cfgVHT);

        % Scale input signal to desired level
        rx = tx.*A(i);

        % Measure the average power at the antenna connector in Watts
        measuredPower(numPkt) = mean(mean(rx.*conj(rx)));

        % Add noise floor at receiver
        rx = awgnChannel(rx);

        % Pass each channel through AGC
        for ic = 1:size(rx,2)
            rx(:,ic) = agc(rx(:,ic));
            reset(agc);
        end

        % Packet detect and determine coarse packet offset
        coarsePktOffset = wlanPacketDetect(rx,chanBW);
        if isempty(coarsePktOffset) % If empty no L-STF detected; packet error
            numPacketErrors = numPacketErrors+1;
```

```
            numPkt = numPkt+1;
            continue; % Go to next loop iteration
        end

        % Extract L-STF and perform coarse frequency offset correction
        lstf = rx(coarsePktOffset+(ind.LSTF(1):ind.LSTF(2)),:);
        coarseFreqOff = wlanCoarseCFOEstimate(lstf,chanBW);
        rx = frequencyOffset(rx,fs,-coarseFreqOff);

        % Extract the non-HT fields and determine fine packet offset
        nonhtfields = rx(coarsePktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
        finePktOffset = wlanSymbolTimingEstimate(nonhtfields,chanBW);

        % Determine final packet offset
        pktOffset = coarsePktOffset+finePktOffset;
        % if packet detected out of a reasonable range (>50 samples);
        % packet error
        if pktOffset>50
            numPacketErrors = numPacketErrors+1;
            numPkt = numPkt+1;
            continue; % Go to next loop iteration
        end

        % Extract L-LTF and perform fine frequency offset correction
        lltf = rx(pktOffset+(ind.LLTF(1):ind.LLTF(2)),:);
        fineFreqOff = wlanFineCFOEstimate(lltf,chanBW);
        rx = frequencyOffset(rx,fs,-fineFreqOff);

        % Extract VHT-LTF samples from the waveform, demodulate and perform
        % channel estimation
        vhtltf = rx(pktOffset+(ind.VHTLTF(1):ind.VHTLTF(2)),:);
        vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,cfgVHT);
        [chanEst,chanEstSSPilots] = wlanVHTLTFChannelEstimate(vhtltfDemod,cfgVHT);

        % Extract VHT Data samples from the waveform
        vhtdata = rx(pktOffset+(ind.VHTData(1):ind.VHTData(2)),:);

        % Estimate the noise power in VHT data field
        nEstVHT = vhtNoiseEstimate(vhtdata,chanEstSSPilots,cfgVHT);

        % Recover the transmitted PSDU in VHT Data
        rxPSDU = wlanVHTDataRecover(vhtdata,chanEst,nEstVHT,cfgVHT, ...
            'LDPCDecodingMethod','norm-min-sum');

        % Determine if any bits are in error, i.e. a packet error
        packetError = any(biterr(txPSDU,rxPSDU));
        numPacketErrors = numPacketErrors+packetError;
        numPkt = numPkt+1;
    end

    % Calculate packet error rate (PER) at input level point
    packetErrorRate(i) = numPacketErrors/(numPkt-1);
    disp(['  Completed after ' ...
        num2str(numPkt-1) ' packets, PER: ' ...
        num2str(packetErrorRate(i))]);

    % Calculate average input power per antenna
    rxAntennaPower(i) = 10*log10(mean(measuredPower(1:(numPkt-1))))+30;
```

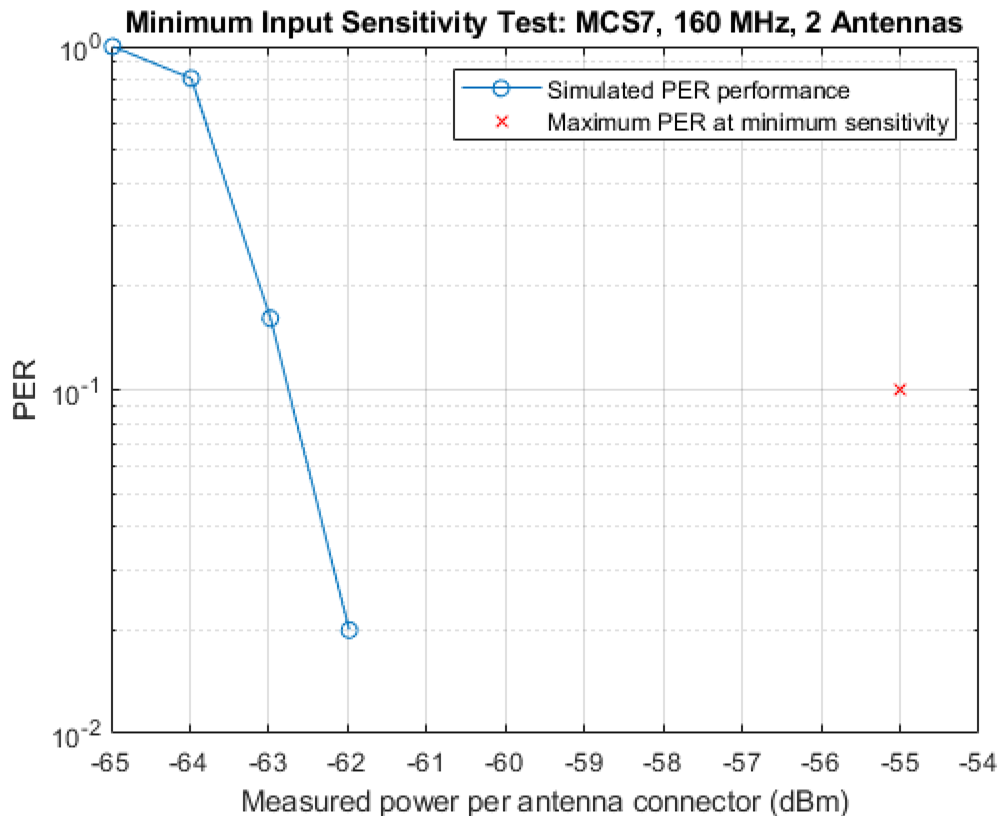
```
    disp(['  Measured antenna connector power: ' ...
        num2str(rxAntennaPower(i),'%2.1f') ' dBm']);
end
```

```
Simulating -65.0 dBm input level...
  Completed after 21 packets, PER: 1
  Measured antenna connector power: -65.0 dBm
Simulating -64.0 dBm input level...
  Completed after 26 packets, PER: 0.80769
  Measured antenna connector power: -64.0 dBm
Simulating -63.0 dBm input level...
  Completed after 130 packets, PER: 0.16154
  Measured antenna connector power: -63.0 dBm
Simulating -62.0 dBm input level...
  Completed after 200 packets, PER: 0.02
  Measured antenna connector power: -62.0 dBm
```

Analysis and Further Exploration

Plot the PER for tested input signal levels with the maximum PER at minimum sensitivity.

```
figure
semilogy(rxAntennaPower,packetErrorRate,'o-')
hold on
semilogy(rxMinSensitivity,0.1,'rx')
currentxlim = xlim(gca);
xlim([currentxlim(1) currentxlim(2)+1])
grid on
xlabel('Measured power per antenna connector (dBm)');
ylabel('PER');
legend('Simulated PER performance','Maximum PER at minimum sensitivity');
title(sprintf(['Minimum Input Sensitivity Test: MCS%d, %d MHz, ' ...
    '%d Antennas'],cfgVHT.MCS,fs/1e6,cfgVHT.NumTransmitAntennas))
```



The plot reveals the simulated 10% PER is just under 8 dB lower than the minimum sensitivity specified by the test. This difference is due to the implementation margin allowed by the test. The implementation margin allows for algorithmic degradations due to impairments and the receiver noise figure when compared to ideal AWGN performance [2]. In this example only AWGN is added as an impairment. Therefore, only the algorithmic performance of front-end synchronization, channel estimation and phase tracking in the presence of AWGN use the implementation margin. If more impairments are included in the simulation the PER waterfall in the plot will move right towards the minimum sensitivity and the margin will decrease.

The number of packets tested at each SNR point is controlled by two parameters; `maxNumErrors` and `maxNumPackets`. For meaningful results, you should use larger numbers than those used in this example.

Selected Bibliography

1. IEEE Std 802.11™-2020. 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.
2. Perahia, Eldad, and Robert Stacey. Next Generation Wireless LANS: 802.11n and 802.11ac. Cambridge University Press, 2013.

802.11ac Transmitter Measurements

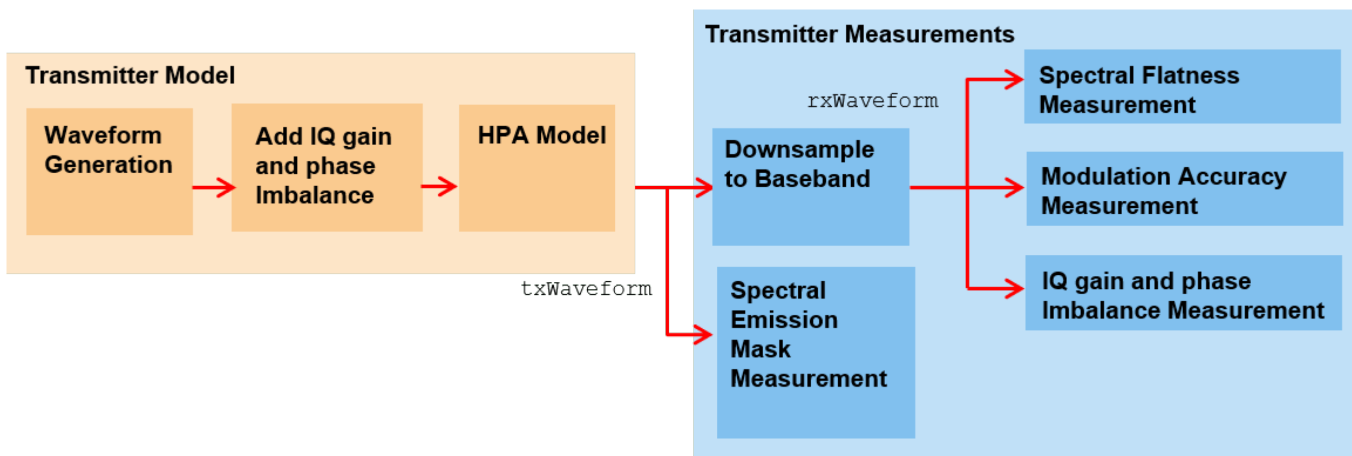
This example shows how to perform these transmitter measurements on an IEEE® 802.11ac™ waveform:

- Modulation accuracy
- Spectrum emission mask
- Spectrum flatness
- In-phase and quadrature (IQ) gain and phase imbalance

Introduction

The transmitter modulation accuracy, required spectrum mask, and required spectral flatness for a given configuration is specified in Section 21.3.17 of [1 on page 8-61]. This example shows how to perform these measurements on a waveform. This example also models, measures, and corrects the IQ gain and phase imbalance. Generate the waveform with WLAN Toolbox™ or use a captured waveform with a spectrum analyzer.

The example generates 20 upsampled VHT packets with an 80 MHz channel bandwidth and a 10 microsecond gap between packets. Each packet contains random data and uses 256-QAM modulation. Add the IQ gain and phase imbalance to the filtered waveform. Use a high-power amplifier (HPA) model to introduce inband distortion and spectral regrowth. Perform the spectral emission mask measurement on the upsampled waveform after the high-power amplifier modeling. Downsample and correct the waveform with the estimated IQ gain and phase imbalance. Measure the error vector magnitude (EVM) of the VHT Data field to determine the modulation accuracy. Additionally, measure the spectral flatness. This diagram shows the workflow contained in the example.



802.11ac VHT Packet Configuration

This example generates an IEEE 802.11ac waveform consisting of multiple VHT format packets. Use the VHT format configuration object, `wlanVHTConfig`, to configure transmission properties of a VHT packet. This example configures the VHT waveform for a 80 MHz bandwidth. Because this example does not use space-time block coding, it can measure the modulation accuracy per spatial stream.

```
cfgVHT = wlanVHTConfig;            % Create packet configuration
cfgVHT.ChannelBandwidth = 'CBW80'; % 80 MHz
cfgVHT.NumTransmitAntennas = 1;    % One transmit antenna
cfgVHT.NumSpaceTimeStreams = 1;    % One space-time stream
cfgVHT.STBC = false;               % No STBC so one spatial stream
cfgVHT.MCS = 8;                    % Modulation: 256-QAM
cfgVHT.APEPLength = 3000;          % A-MPDU length pre-EOF padding in bytes
```

Waveform Generation

Generate the VHT waveform for the specified bits and configuration by using the `wlanWaveformGenerator` function, specifying the desired oversampling factor, number of packets, and idle time between each packet.

```
osf = 3;          % 3x oversampling factor
numPackets = 20;  % Generate 20 packets
idleTime = 10e-6; % 10 microseconds idle time between packets
```

Create random bits for all packets, `data`, and pass as an argument to `wlanWaveformGenerator` along with the VHT packet configuration object `cfgVHT`. This configures the waveform generator to synthesize an 802.11ac VHT waveform. Additionally, configure the waveform generator by using name-value pairs to generate multiple oversampled packets with a specified idle time between each packet.

```
% Create random data; PSDULength is in bytes
savedState = rng(0); % Set random state
data = randi([0 1],cfgVHT.PSDULength*8*numPackets,1);

% Generate a multi-packet waveform
txWaveform = wlanWaveformGenerator(data,cfgVHT, ...
    'OversamplingFactor',osf,'NumPackets',numPackets,'IdleTime',idleTime);

fs = wlanSampleRate(cfgVHT); % Baseband sampling rate of the waveform
```

Add Impariments

IQ Imbalance Modeling

IQ imbalance arises when a front-end component does not respect the power balance or the orthogonality between the I and Q branches. This example adds IQ gain and phase imbalance to the transmitted waveform based on the flag `modelIQImbalance`. At the receiver, estimate the IQ gain and phase imbalance and correct the waveform as per the compensation scheme specified in [5 on page 8-61].

```
modelIQImbalance = true; % Set to true to add IQ gain and phase imbalance

if modelIQImbalance
    iqGaindB = 1; % IQ gain imbalance in dB, specify from the range [-1 1]
    iqPhaseDeg = 1; % IQ phase imbalance in degrees, specify from the range [-2 2]
    iqGainLin = db2mag(iqGaindB); % Convert gain from dB to linear value
    txWaveform = real(txWaveform) + 1i*imag(txWaveform)*iqGainLin*exp(1j*iqPhaseDeg*pi/180); % A
end
```

High-Power Amplifier Modeling

The high-power amplifier introduces nonlinear behavior in the form of inband distortion and spectral regrowth. This example simulates the power amplifiers by using the Rapp model in 802.11ac [2 on page 8-61], which introduces AM/AM distortion.

Model the amplifier by using the `comm.MemorylessNonlinearity` object, and configure reduced distortion by specifying a backoff, `hpaBackoff`, such that the amplifier operates below its saturation point. You can increase the backoff to reduce EVM for higher MCS values.

```
pSaturation = 25; % Saturation power of a power amplifier in dBm
hpaBackoff = 13; % Power amplifier backoff in dB

% Create and configure a memoryless nonlinearity to model the amplifier
nonLinearity = comm.MemorylessNonlinearity;
nonLinearity.Method = 'Rapp model';
nonLinearity.Smoothness = 3; % p parameter
nonLinearity.LinearGain = -hpaBackoff;
nonLinearity.OutputSaturationLevel = db2mag(pSaturation-30);

% Apply the model to each transmit antenna
txWaveform = nonLinearity(txWaveform);
```

Thermal Noise

Add thermal noise to each transmit antenna by using the `comm.ThermalNoise` object with a noise figure of 6 dB [3 on page 8-61].

```
thNoise = comm.ThermalNoise('NoiseMethod','Noise Figure','SampleRate',fs*osf,'NoiseFigure',6);
for i = 1:cfgVHT.NumTransmitAntennas
    txWaveform(:,i) = thNoise(txWaveform(:,i));
end
```

Modulation Accuracy (EVM), Spectral Flatness and IQ Imbalance Measurements

Downsampling and Filtering

Resample the oversampled waveform down to baseband for physical layer processing and EVM and spectral flatness measurements, applying a low-pass anti-aliasing filter before downsampling. The impact of the low-pass filter is visible in the spectral flatness measurement. The anti-aliasing filter is designed so that all active subcarriers are within the filter passband.

Design resampling filter.

```
aStop = 40; % Stopband attenuation
ofdmInfo = wlanVHTOFDMInfo('VHT-Data',cfgVHT); % OFDM parameters
SCS = fs/ofdmInfo.FFTLength; % Subcarrier spacing
txbw = max(abs(ofdmInfo.ActiveFrequencyIndices))*2*SCS; % Occupied bandwidth
[L,M] = rat(1/osf);
maxLM = max([L M]);
R = (fs-txbw)/fs;
TW = 2*R/maxLM; % Transition width
firdec = designMultirateFIR(L,M,TW,aStop,'SystemObject',true);
```

Resample the waveform to baseband.

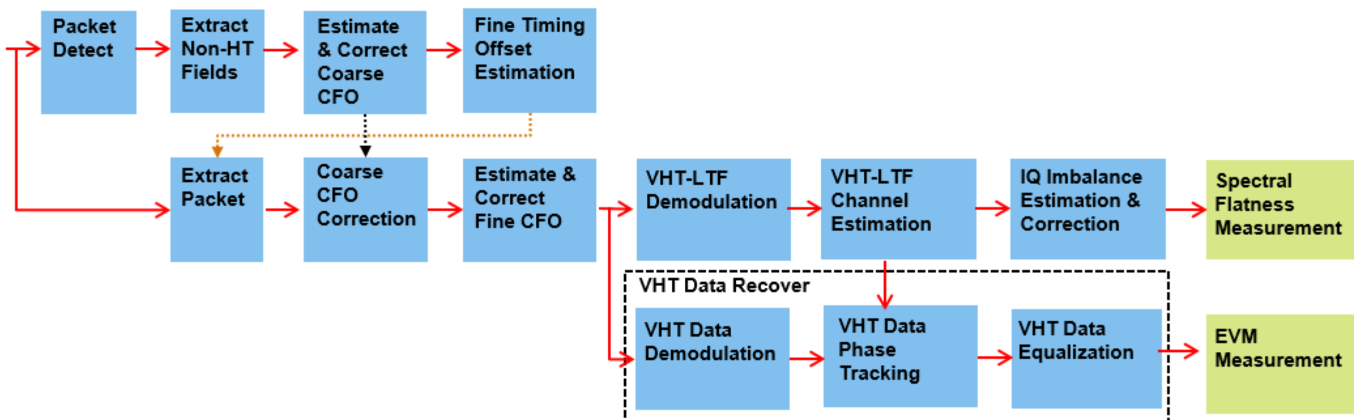
```
rxWaveform = firdec(txWaveform);
```

Receiver Processing

This section detects, synchronizes, and extracts each packet in `rxWaveform`, then measures the EVM, spectral flatness, and IQ imbalance. For each packet, the example performs these steps:

- Detect the start of the packet
- Extract the non-HT fields
- Estimate and correct coarse carrier frequency offset (CFO)
- Perform fine symbol timing estimate by using the frequency-corrected non-HT fields
- Extract the packet from the waveform by using the fine symbol timing offset
- Correct the extracted packet with the coarse CFO estimate
- Extract the L-LTF, then estimate the fine CFO and correct for the whole packet
- Extract the VHT-LTF and perform channel estimation for each of the transmit streams
- Measure the IQ imbalance from the channel estimate and perform correction on the channel estimate
- Measure the spectral flatness by using the channel estimate
- Extract and OFDM demodulate the VHT data field
- Perform noise estimation by using the demodulated data field pilots and single-stream channel estimate at pilot subcarriers
- Phase-correct and equalize the VHT data field by using the channel and noise estimates
- Correct the equalized data subcarriers with the IQ imbalance estimate
- For each data-carrying subcarrier in each spatial stream, find the closest constellation point and measure the EVM

The diagram shows the processing chain:



The VHT-LTF symbols include pilot symbols to allow for phase tracking, but this example does not perform phase tracking.

Test the spectral flatness for each packet by measuring the deviation in the magnitude of individual subcarriers in the channel estimate against the average [1 on page 8-61]. Plot these deviations for each packet using the helper function vhtTxPlotSpectralFlatness. Plot the average EVM per data-carrying subcarrier and the equalized symbols for each packet.

Demodulate, equalize, and decode the VHT Data symbols by using the `wlanVHTDataRecover` function. Parameterize this function to perform pilot phase tracking and zero-forcing equalization as required by the standard. This example measures the modulation accuracy from the equalized symbols.

This example makes two different EVM measurements using two instances of `comm.EVM`.

- RMS EVM per packet, which comprises averaging the EVM over subcarriers, OFDM symbols, and spatial streams.
- RMS EVM per subcarrier per spatial stream for a packet. Because this configuration maps spatial streams directly to antennas, this measurement can help detect frequency-dependent impairments, which may affect individual RF chains differently. This measurement averages the EVM over OFDM symbols only.

```
% Setup EVM measurements
[EVMPerPkt,EVMPerSC] = vhtEVMSetup(cfgVHT);
```

This code configures objects and variables for processing.

```
% Get indices for accessing each field within the time-domain packet
ind = wlanFieldIndices(cfgVHT);

rxWaveformLength = size(rxWaveform,1);
pktLength = double(ind.VHTData(2));

% Define the minimum length of data we can detect; length of the L-STF in
% samples
minPktLen = double(ind.LSTF(2)-ind.LSTF(1))+1;

% Setup the measurement plots
[hSF,hCon,hEVM] = vhtTxSetupPlots(cfgVHT);

rmsEVM = zeros(numPackets,1);
pktOffsetStore = zeros(numPackets,1);

rng(savedState); % Restore random state
```

Detect and process packets within the received waveform, `rxWaveform` by using a while loop, which performs these steps.

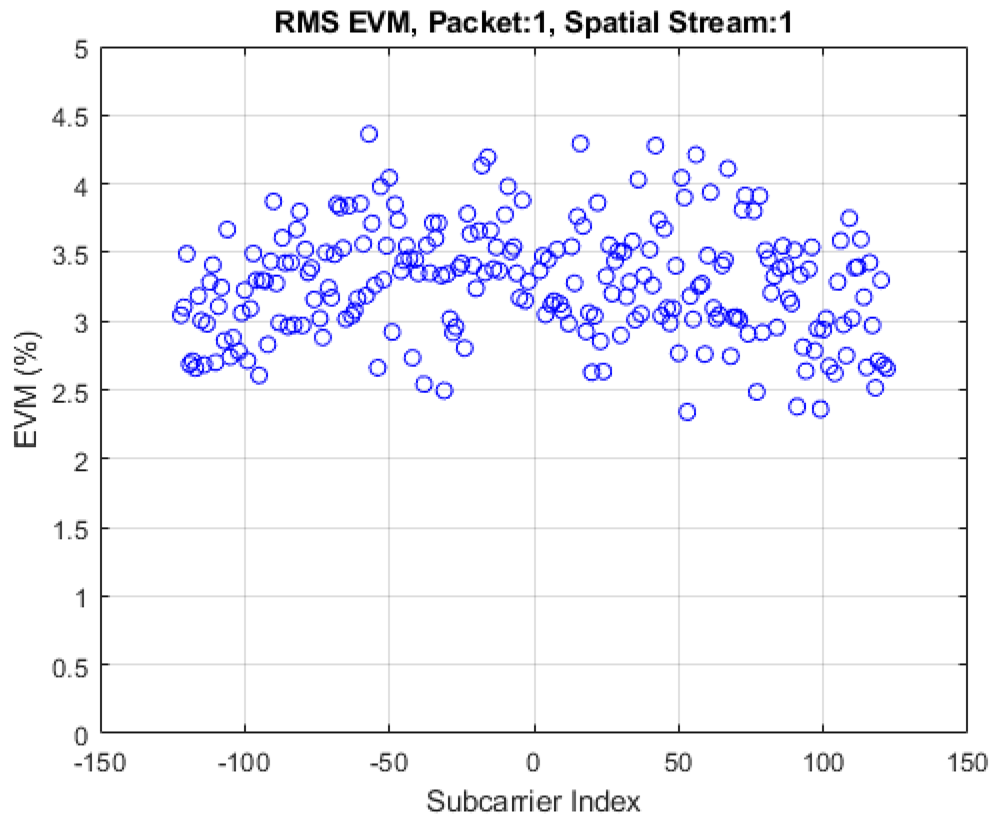
- Detect a packet by indexing into `rxWaveform` with the sample offset, `searchOffset`
- Detect and process the first packet within `rxWaveform`
- Detect and process the next packet by incrementing the sample index offset, `searchOffset`
- Repeat until no further packets are detected

```
pktNum = 0;
searchOffset = 0; % Start at first sample (no offset)
while (searchOffset+minPktLen)<=rxWaveformLength
    % Packet detect
    pktOffset = wlanPacketDetect(rxWaveform,cfgVHT.ChannelBandwidth);
    % Packet offset from start of waveform
    pktOffset = searchOffset+pktOffset;
    % If no packet detected or offset outwith bounds of waveform then stop
    if isempty(pktOffset) || (pktOffset<0) || ...
            ((pktOffset+ind.LSIG(2))>rxWaveformLength)
        break;
    end

    % Extract non-HT fields and perform coarse frequency offset correction
    % to allow for reliable symbol timing
    nonht = rxWaveform(pktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
```

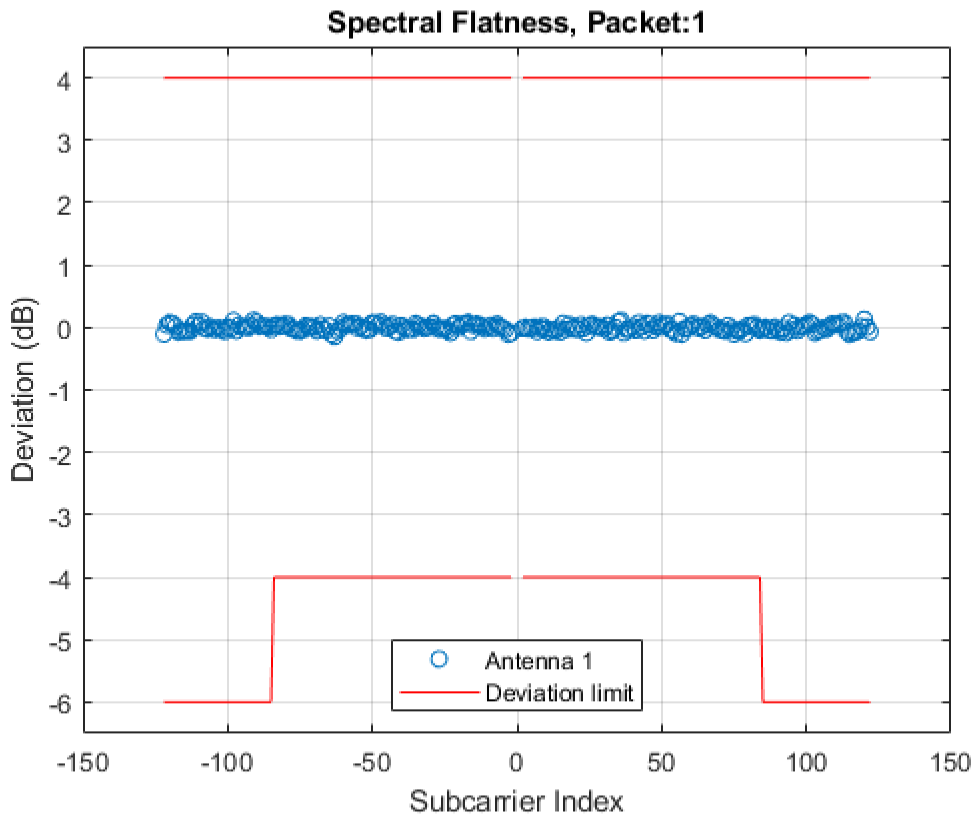
```
coarsefreqOff = wlanCoarseCFOEstimate(nonht,cfgVHT.ChannelBandwidth);
nonht = frequencyOffset(nonht,fs,-coarsefreqOff);

% Determine offset between the expected start of L-LTF and actual start
% of L-LTF
lltfOffset = wlanSymbolTimingEstimate(nonht,cfgVHT.ChannelBandwidth);
% Determine packet offset
pktOffset = pktOffset+lltfOffset;
% If offset is without bounds of waveform skip samples and continue
% searching within remainder of the waveform
if (pktOffset<0) || ((pktOffset+pktLength)>rxWaveformLength)
    searchOffset = pktOffset+double(ind.LSTF(2))+1;
    continue;
end

% Timing synchronization complete; extract the detected packet
rxPacket = rxWaveform(pktOffset+(1:pktLength),:);
pktNum = pktNum+1;
disp(['  Packet ' num2str(pktNum) ' at index: ' num2str(pktOffset+1)]);

% Apply coarse frequency correction to the extracted packet
rxPacket = frequencyOffset(rxPacket,fs,-coarsefreqOff);

% Perform fine frequency offset correction on the extracted packet
lltf = rxPacket(ind.LLTF(1):ind.LLTF(2),:); % Extract L-LTF
fineFreqOff = wlanFineCFOEstimate(lltf,cfgVHT.ChannelBandwidth);
rxPacket = frequencyOffset(rxPacket,fs,-fineFreqOff);

% Extract VHT-LTF samples, demodulate and perform channel estimation
vhtltf = rxPacket(ind.VHTLTF(1):ind.VHTLTF(2),:);
vhtltfDemod = wlanVHTLTFDemodulate(vhtltf,cfgVHT);

% Channel estimate and single stream channel estimate
[chanEst,chanEstSSPilots] = wlanVHTLTFChannelEstimate(vhtltfDemod,cfgVHT);

% Perform IQ gain and phase imbalance estimation
[gainEst,phaseEst,alphaEst,betaEst,gamma,dataRot] = ...
    helperIQImbalanceEstimate(chanEst,cfgVHT);
fprintf('    Measured IQ gain & phase imbalance: %2.2f dB, %2.2f deg\n',gainEst,phaseEst);

% Perform IQ gain and phase imbalance correction on channel
% estimates
chanEst = chanEst./(alphaEst + betaEst.*gamma); % As specified in Equation-29 of [5]

% Spectral flatness test
[pass,deviation,testsc] = wlanSpectralFlatness(chanEst,'VHT',cfgVHT.ChannelBandwidth);
if pass
    disp('    Spectral flatness passed');
else
    disp('    Spectral flatness failed');
end
% Plot deviation against limits
vhtTxPlotSpectralFlatness(deviation,testsc,pktNum,hSF);

% Extract VHT Data samples from the waveform
vhtdata = rxPacket(ind.VHTData(1):ind.VHTData(2),:);

% Estimate the noise power in VHT data field
```

```
    noiseVarVHT = vhtNoiseEstimate(vhtdata,chanEstSSPilots,cfgVHT);

    % Extract VHT Data samples and perform OFDM demodulation, equalization
    % and phase tracking
    [~,~,eqSym] = wlanVHTDataRecover(vhtdata,chanEst,noiseVarVHT,cfgVHT,...
        'EqualizationMethod','ZF','PilotPhaseTracking','PreEQ','LDPCDecodingMethod','norm-min-sur

    % Perform IQ gain and phase imbalance correction on VHT data
    eqSym = eqSym.*dataRot; % Carrier rotation on data subcarriers
    eqSym = ((conj(alphaEst)*eqSym)-(betaEst*conj(eqSym(end:-1:1,:,:))))/((abs(alphaEst)^2)-(abs

    % Compute RMS EVM over all spatial streams for packet
    rmsEVM(pktNum) = EVMPerPkt(eqSym);
    fprintf('    RMS EVM: %2.2f%%, %2.2fdB\n',rmsEVM(pktNum),20*log10(rmsEVM(pktNum)/100));

    % Compute RMS EVM per subcarrier and spatial stream for the packet
    evmPerSC = EVMPerSC(eqSym); % Nst-by-1-by-Nss

    % Plot RMS EVM per subcarrier and equalized constellation
    vhtTxEVMConstellationPlots(eqSym,evmPerSC,cfgVHT,pktNum,hCon,hEVM);

    % Store the offset of each packet within the waveform
    pktOffsetStore(pktNum) = pktOffset;

    % Increment waveform offset and search remaining waveform for a packet
    searchOffset = pktOffset+pktLength+minPktLen;
end
```

```
  Packet 1 at index: 25

    Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg

    Spectral flatness passed

    RMS EVM: 3.30%, -29.63dB
```



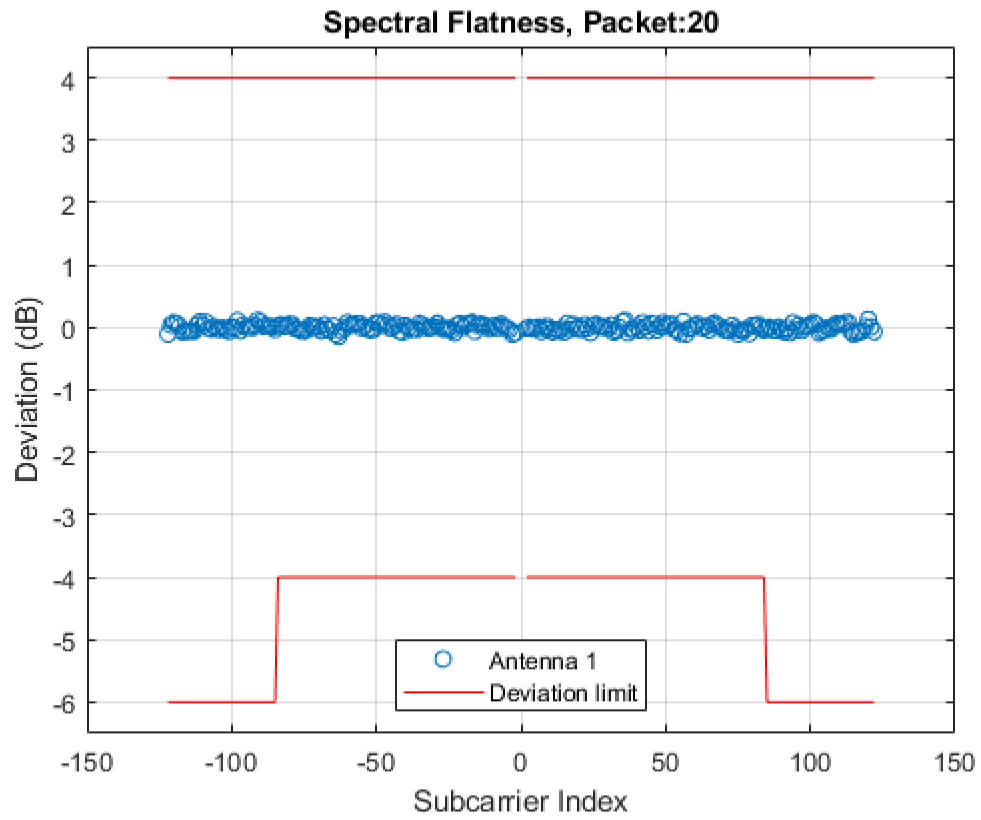

```
Packet 2 at index: 9785

  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg

  Spectral flatness passed
```



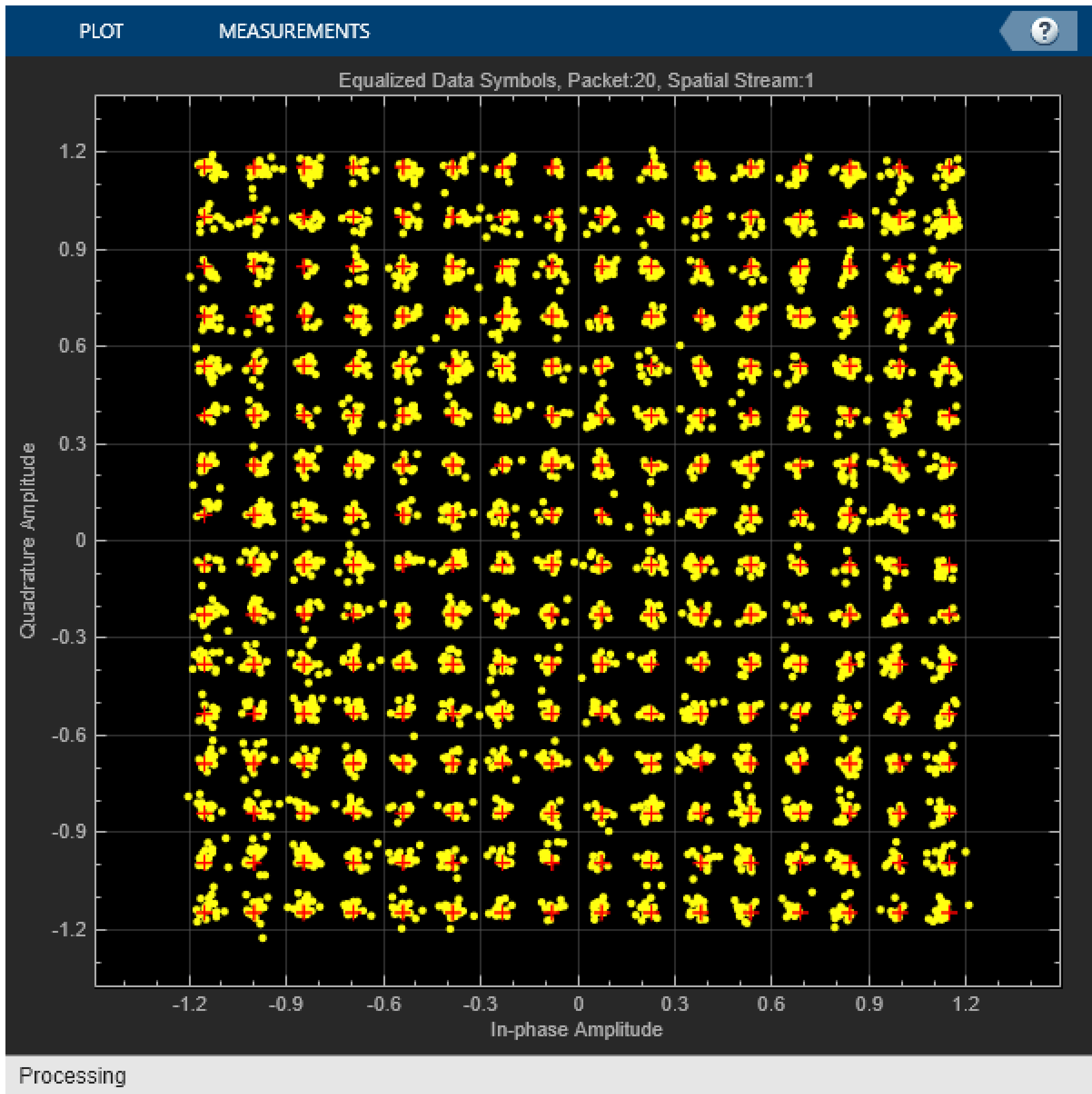

```
  RMS EVM: 3.07%, -30.24dB
Packet 3 at index: 19545
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.91%, -30.71dB
Packet 4 at index: 29305
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 3.04%, -30.33dB
Packet 5 at index: 39065
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.99%, -30.50dB
Packet 6 at index: 48825
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
```

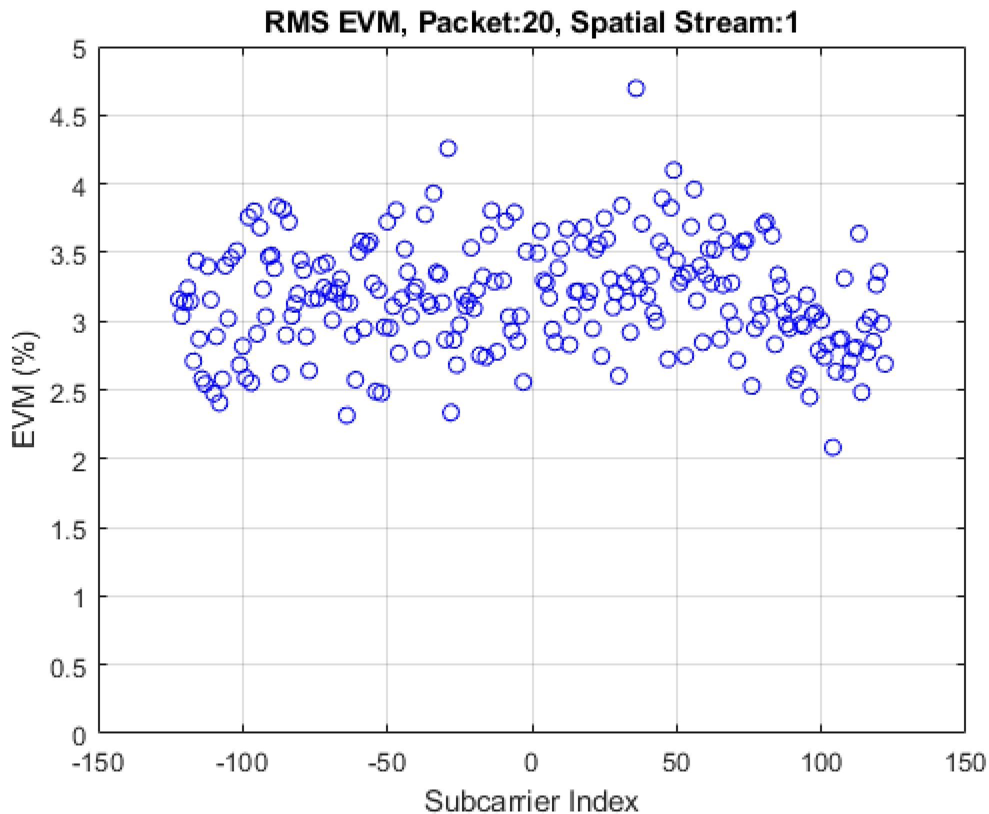
```
  RMS EVM: 2.67%, -31.49dB
Packet 7 at index: 58585
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.93%, -30.65dB
Packet 8 at index: 68345
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.81%, -31.03dB
Packet 9 at index: 78105
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.93%, -30.65dB
Packet 10 at index: 87865
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.73%, -31.28dB
Packet 11 at index: 97625
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.90%, -30.76dB
Packet 12 at index: 107385
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.94%, -30.63dB
Packet 13 at index: 117145
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.84%, -30.94dB
Packet 14 at index: 126905
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
```

```
  RMS EVM: 2.91%, -30.73dB
Packet 15 at index: 136665
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 3.09%, -30.20dB
Packet 16 at index: 146425
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.60%, -31.69dB
Packet 17 at index: 156185
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 2.99%, -30.49dB
Packet 18 at index: 165945
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 3.20%, -29.90dB
Packet 19 at index: 175705
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
  RMS EVM: 3.24%, -29.79dB
Packet 20 at index: 185465
  Measured IQ gain & phase imbalance: 0.98 dB, 0.98 deg
  Spectral flatness passed
```




```
RMS EVM: 3.20%, -29.91dB
```


PLOT
MEASUREMENTS
Equalized Data Symbols, Packet:20, Spatial Stream:1
Quadrature Amplitude
In-phase Amplitude
1.2
0.9
0.6
0.3
0
-0.3
-0.6
-0.9
-1.2
Processing




```
if pktNum>0
    fprintf('Average EVM for %d packets: %2.2f%%, %2.2fdB\n', ...
        pktNum,mean(rmsEVM(1:pktNum)),20*log10(mean(rmsEVM(1:pktNum))/100));
else
    disp('No complete packet detected');
end
```

```
Average EVM for 20 packets: 2.96%, -30.56dB
```

Transmit Spectrum Emission Mask Measurement

This section measures the spectral mask of the filtered and impaired waveform after high-power amplifier modeling.

The transmitter spectral mask test [4 on page 8-61] uses a time-gated spectral measurement of the VHT Data field. The example extracts the VHT Data field of each packet from the oversampled waveform, `txWaveform`, by using the start indices of each packet within the waveform. Any delay introduced in the baseband processing chain used to determine the packet indices must be accounted for when gating the VHT data field within `txWaveform`. Concatenate the extracted VHT Data fields in preparation for measurement.

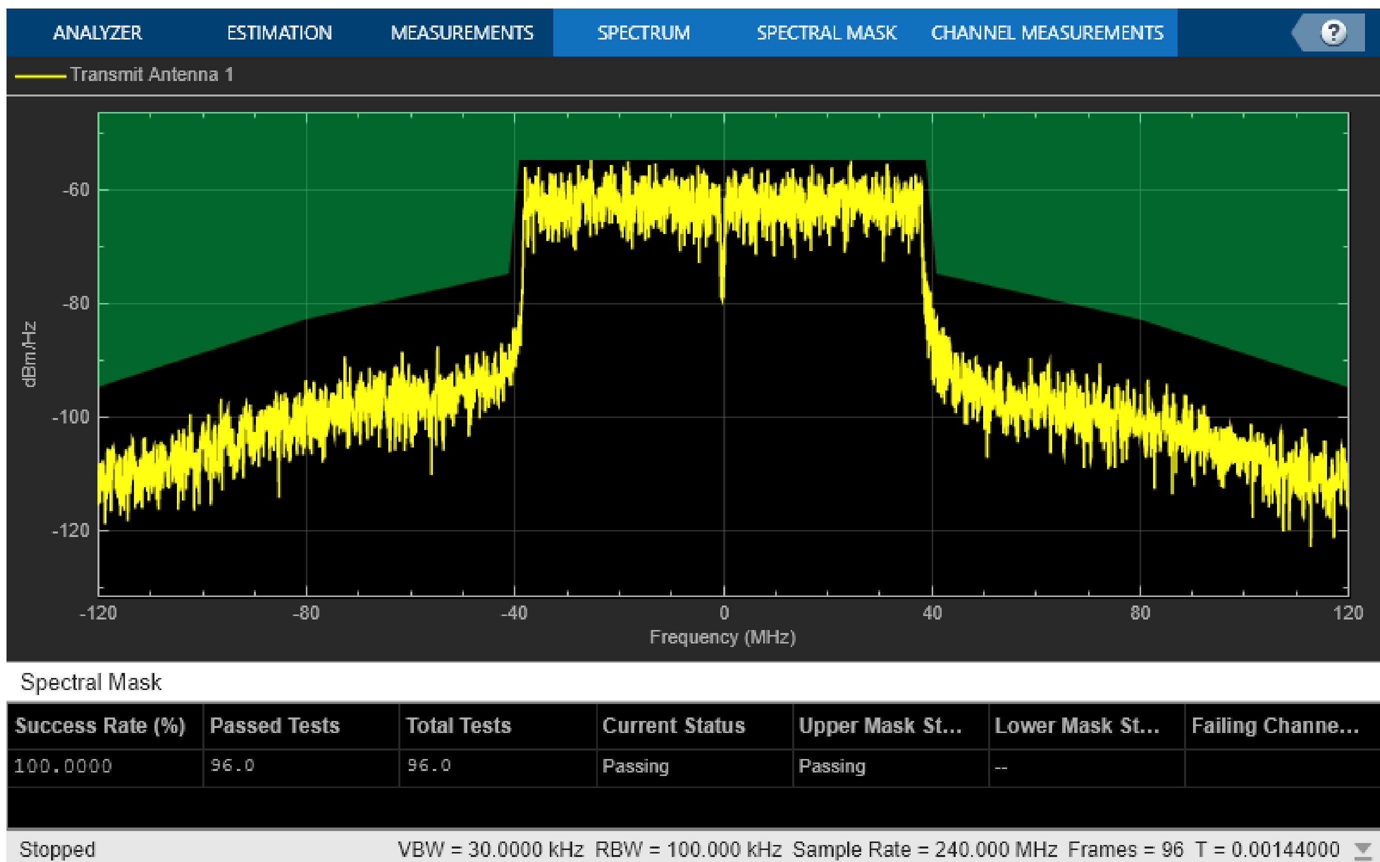
```
% Indices for accessing each field within the time-domain packet
ind = wlanFieldIndices(cfgVHT,'OversamplingFactor', osf);
startIdx = ind.VHTData(1); % Start of Data
endIdx = ind.VHTData(2);   % End of Data
idleNSamps = idleTime*fs*osf; % Idle time samples
```

```
perPktLength = endIdx+idleNSamps;

idx = zeros(endIdx-startIdx+1,numPackets);
for i = 1:numPackets
    % Start of packet in txWaveform, accounting for the filter delay
    pktOffset = (i-1)*perPktLength;
    % Indices of non-HT Data in txWaveform
    idx(:,i) = pktOffset+(startIdx:endIdx);
end
gatedVHTData = txWaveform(idx(:),:);
```

The 802.11ac standard specifies the spectral mask relative to the peak power spectral density. The helper function helperSpectralMaskTest generates a plot which overlays the required mask with the measured PSD.

```
if pktNum>0
    helperSpectralMaskTest(gatedVHTData,fs,osf);
end
```

```
Spectrum mask passed
```



Conclusion and Further Exploration

This example plots four results: spectral flatness, RMS EVM per subcarrier, equalized constellation, and spectral mask.

The high-power amplifier model introduces significant inband distortion and spectral regrowth which is visible in the EVM results, noisy constellation and out-of-band emissions in the spectral mask plot. Try increasing the high-power amplifier backoff and note the improved EVM, constellation and lower out-of-band emissions.

Try using different values for `iqGaindB` and `iqPhaseDeg` and note the impact on EVM and constellation.

Related examples:

- “802.11be Transmitter Measurements” on page 3-2
- “802.11ba WUR Waveform Generation and Analysis” on page 3-14
- “802.11ad Transmitter Spectral Emission Mask Testing” on page 8-74
- “802.11p Spectral Emission Mask Testing” on page 8-79
- “Recover and Analyze Packets in 802.11 Waveform” on page 4-2

Selected Bibliography

1. IEEE Std 802.11™-2020 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.
2. Loc and Cheong. IEEE P802.11 Wireless LANs. TGac Functional Requirements and Evaluation Methodology Rev. 16. 2011-01-19.
3. Perahia, E., and R. Stacey. Next Generation Wireless LANs: 802.11n and 802.11ac. 2nd Edition. United Kingdom: Cambridge University Press, 2013.
4. Archambault, Jerry, and Shravan Surineni. "IEEE 802.11 spectral measurements using vector signal analyzers." RF Design 27.6 (2004): 38-49.
5. M. Janaswamy, N. K. Chavali and S. Batabyal, "Measurement of transmitter IQ parameters in HT and VHT wireless LAN systems," 2016 International Conference on Signal Processing and Communications (SPCOM), Bangalore.

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

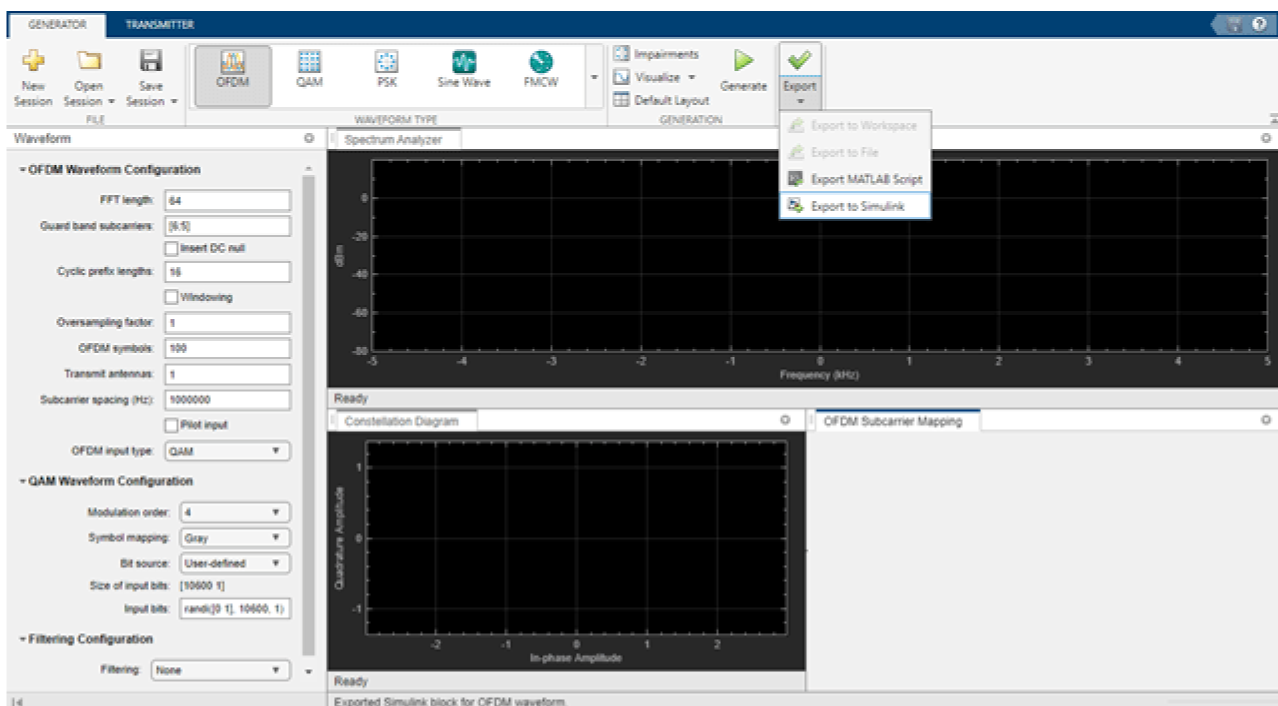
The **Wireless Waveform Generator** app is an interactive tool for creating, impairing, visualizing, and exporting waveforms. You can export the waveform to your workspace or to a `.mat` or `.bb` file. You can also export the waveform generation parameters to a runnable MATLAB® script or a Simulink® 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

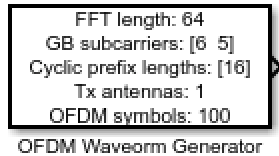
Open the **Wireless Waveform Generator** app by clicking the app icon on the **Apps** tab, under **Signal Processing and Communications**. Alternatively, enter `wirelessWaveformGenerator` at the MATLAB command prompt.

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**.



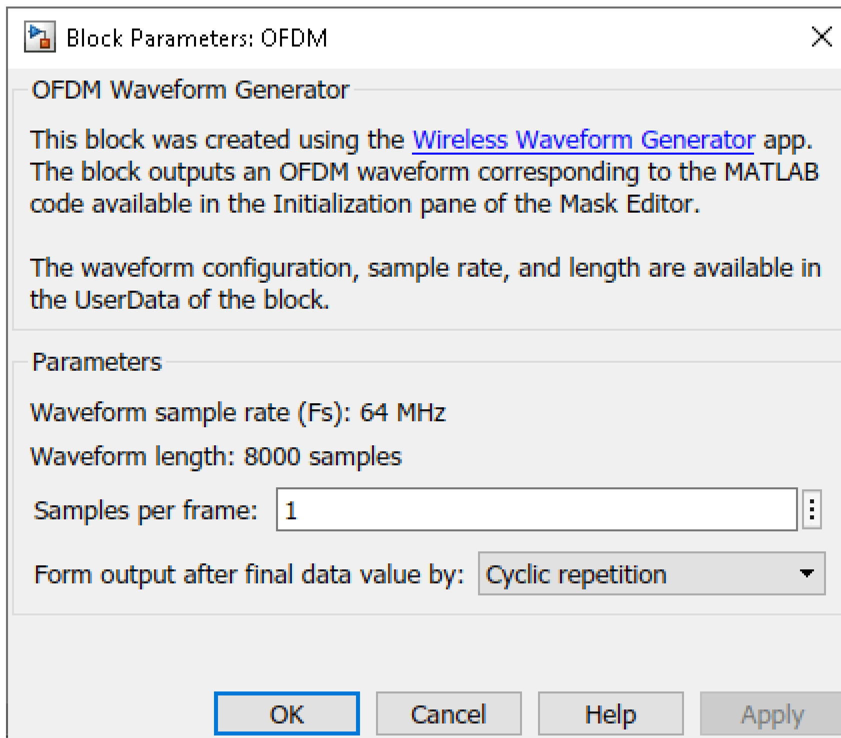
The **Export to Simulink** option creates a Simulink block, which outputs the selected waveform when you run the Simulink model. The block is exported to a new model if no open models exist.

```
modelName = 'WWGExport2SimulinkBlock';
open_system(modelName);
```




```
% Copyright 2021-2023 The MathWorks, Inc.
```

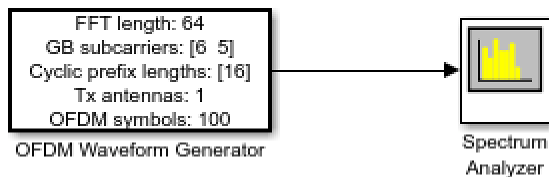
The **Form output after final data value by** block parameter specifies the output after all of the specified signal samples are generated. The value options for this parameter are `Cyclic repetition` and `Setting to zero`. The `Cyclic repetition` option repeats the signal from the beginning after it 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 **Code** 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.

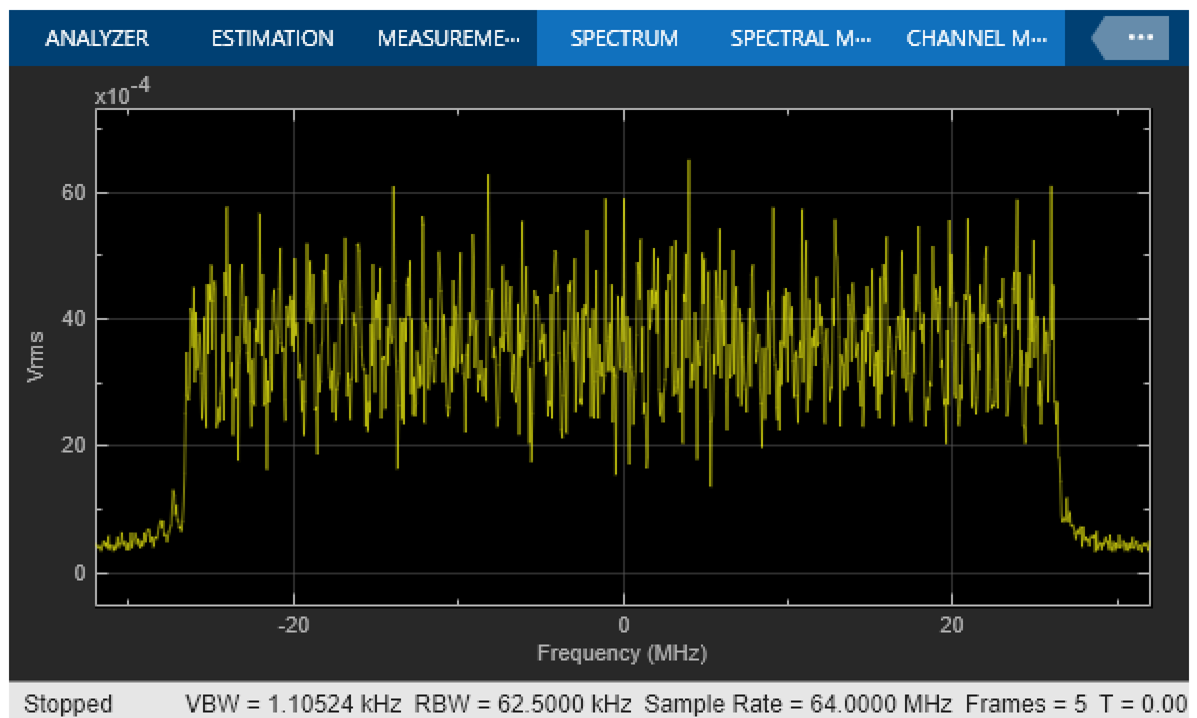
```
modelName = 'WWGExport2SimulinkModel';
open_system(modelName);
```




```
% Copyright 2021-2023 The MathWorks, Inc.
```

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

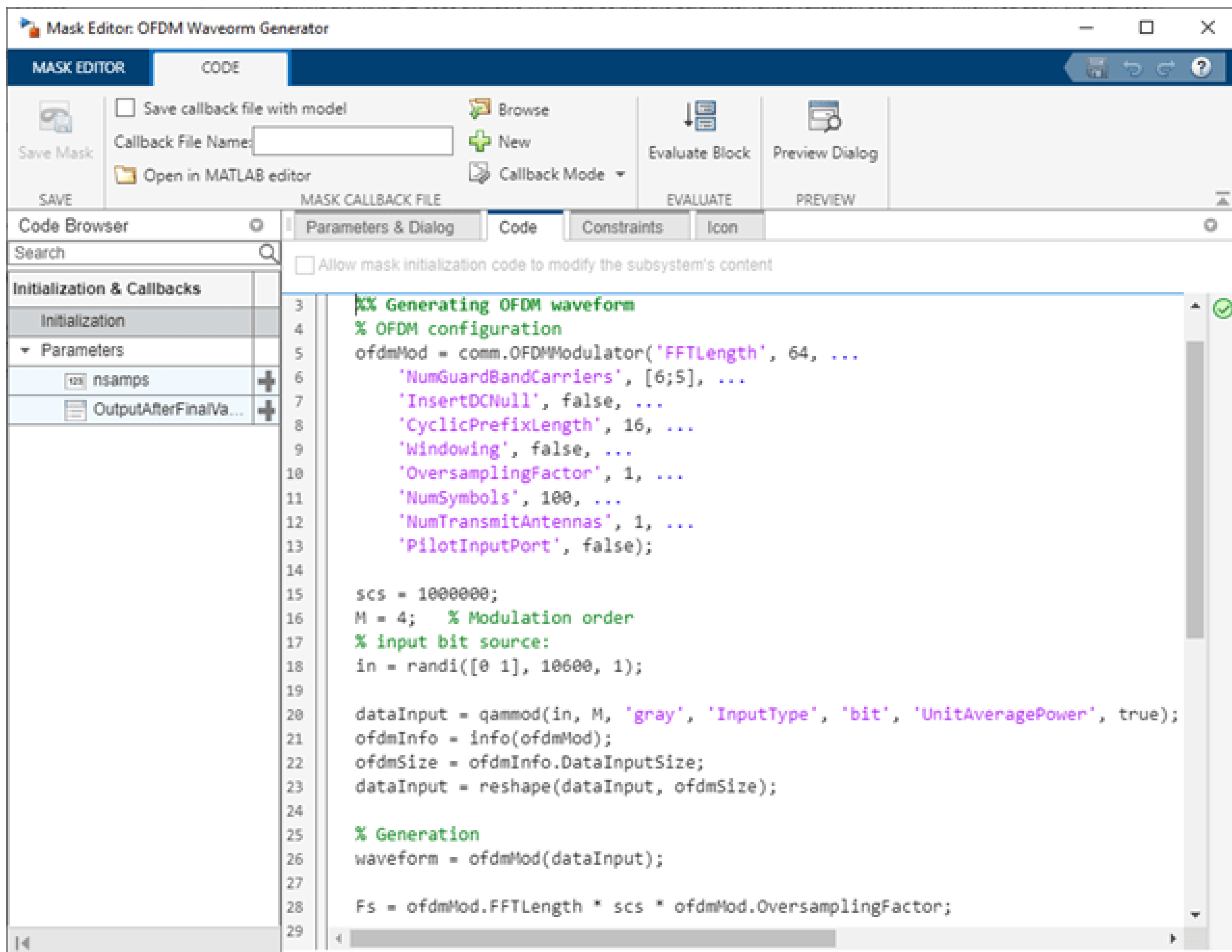
When you run the Simulink model, the exported block outputs the waveform generated in the **Code** tab of the Mask Editor dialog box for the block. The MATLAB code that initializes the waveform in this tab corresponds to the configuration that you selected in the **Wireless Waveform Generator** app before 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,

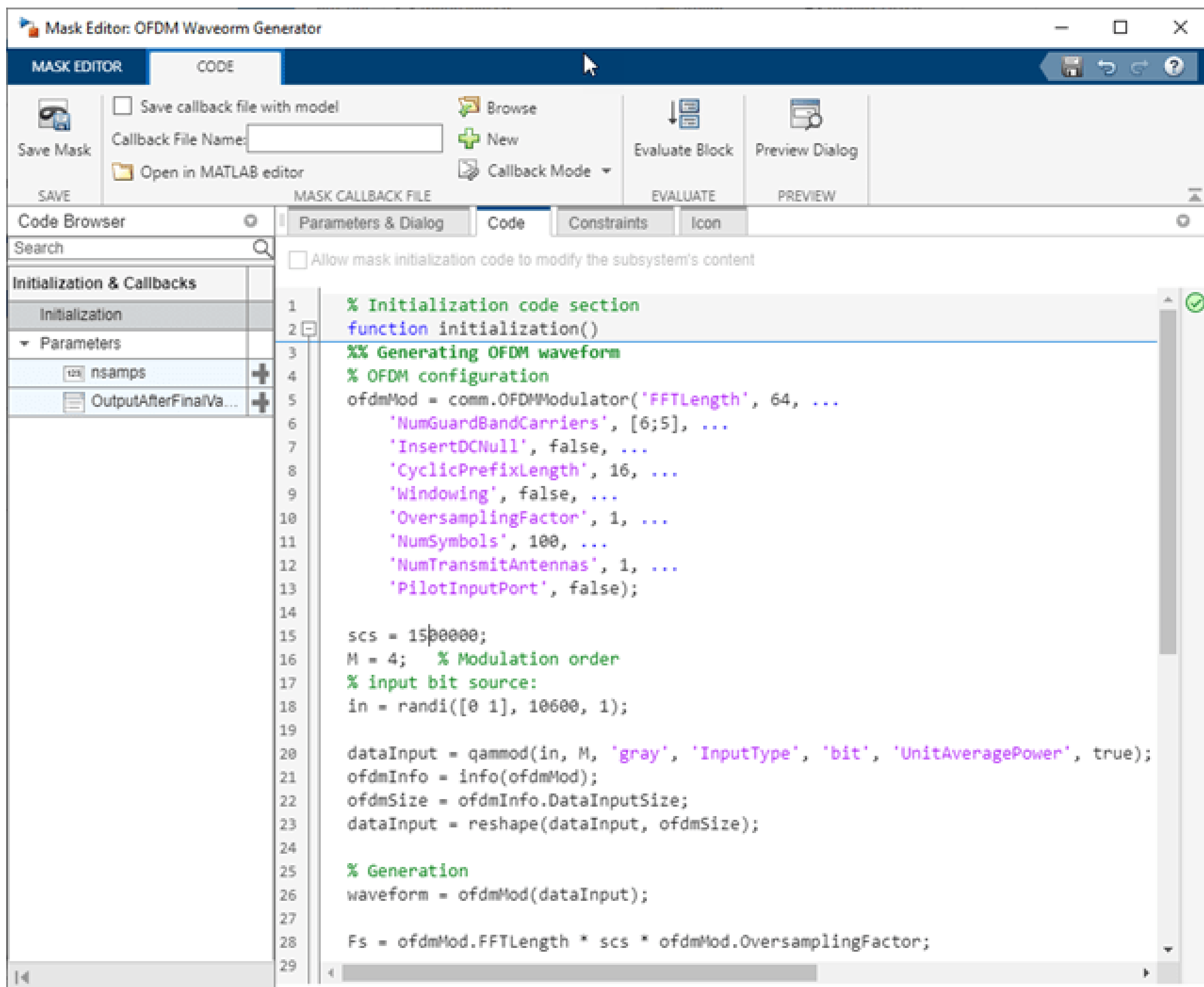
parameter range validation during the parameterization process, and visualization of the waveform before running the Simulink model.

- Update the configuration parameters that are available in the **Code** 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.

You can update the configuration in the **Code** tab of the Mask Editor. To open the Mask Editor, click the exported block and press **Ctrl+M**.

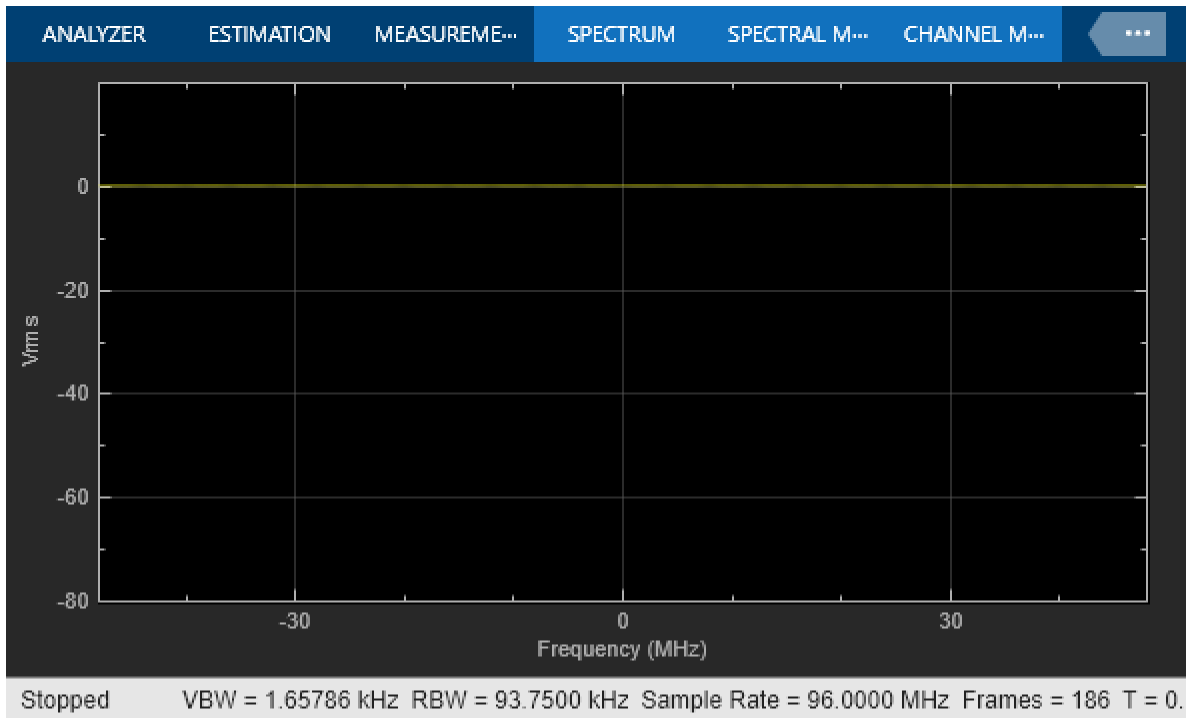


Use the MATLAB code that is available in the **Code** tab to update the parameters of your choice. For example, set the subcarrier spacing, `scs`, to 1,500,000 Hz.



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.

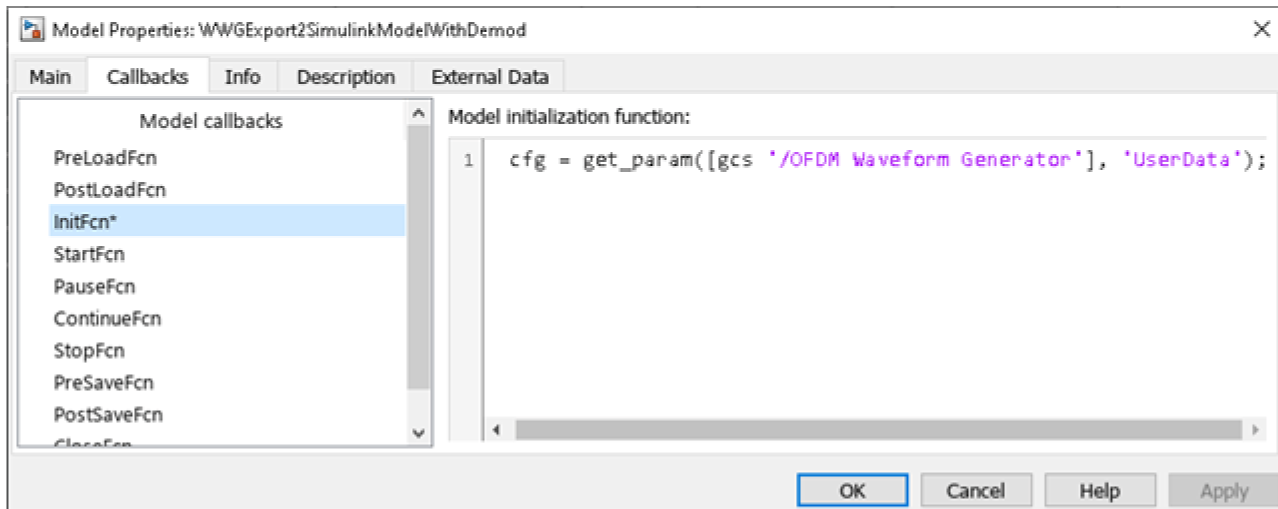
- `WaveformConfig`: Waveform configuration
- `WaveformLength`: Waveform length
- `Fs`: Waveform sample rate

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
```

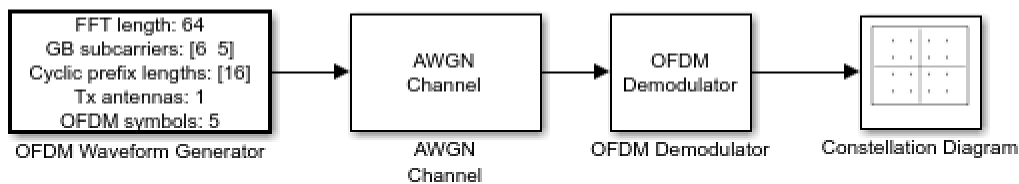
Store the structure available in the user data in a base workspace variable by using the `InitFcn` in the callback. The `InitFcn` callback is executed during a model update and simulation. To use this callback, click the **MODELING** tab, then click the **Model Settings** dropdown, and click the **Model 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 **Code** 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);
```



% Copyright 2021-2023 The MathWorks, Inc.

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.

Block Parameters: OFDM Demodulator

OFDM Demodulator

Apply OFDM demodulation to the input signal.

Enable pilot signal output to separate it from the data signal after demodulation.

Source code

Parameters

FFT length: cfg.WaveformConfig.FFTLength 64

Number of guard bands: cfg.WaveformConfig.NumGuardBandCarriers [6;5]

Remove DC subcarrier from output

Pilot output port

Cyclic prefix length: cfg.WaveformConfig.CyclicPrefixLength 16

Oversampling factor: 1

Number of OFDM symbols: cfg.WaveformConfig.NumSymbols 5

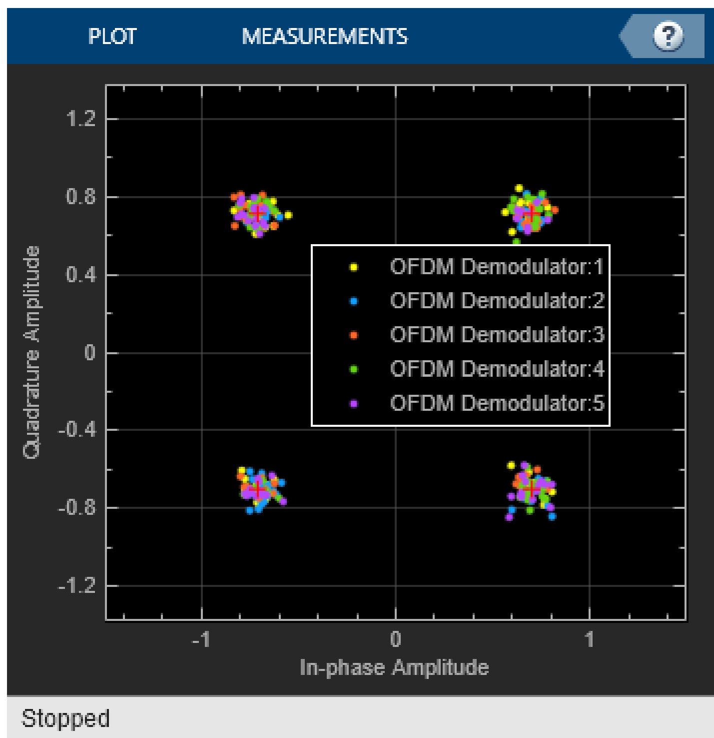
Number of receive antennas: cfg.WaveformConfig.NumTransmitAntennas 1

Simulate using: Interpreted execution

OK Cancel Help Apply

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);
```

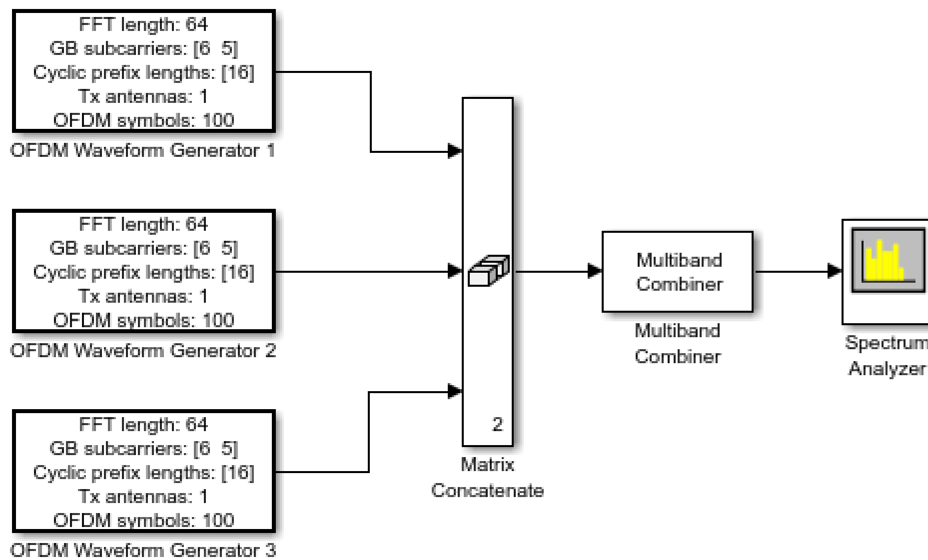


After demodulating the OFDM waveform by using the OFDM Demodulator block, the Constellation Diagram block displays the resulting QAM symbols.

Generate Multicarrier Waveforms

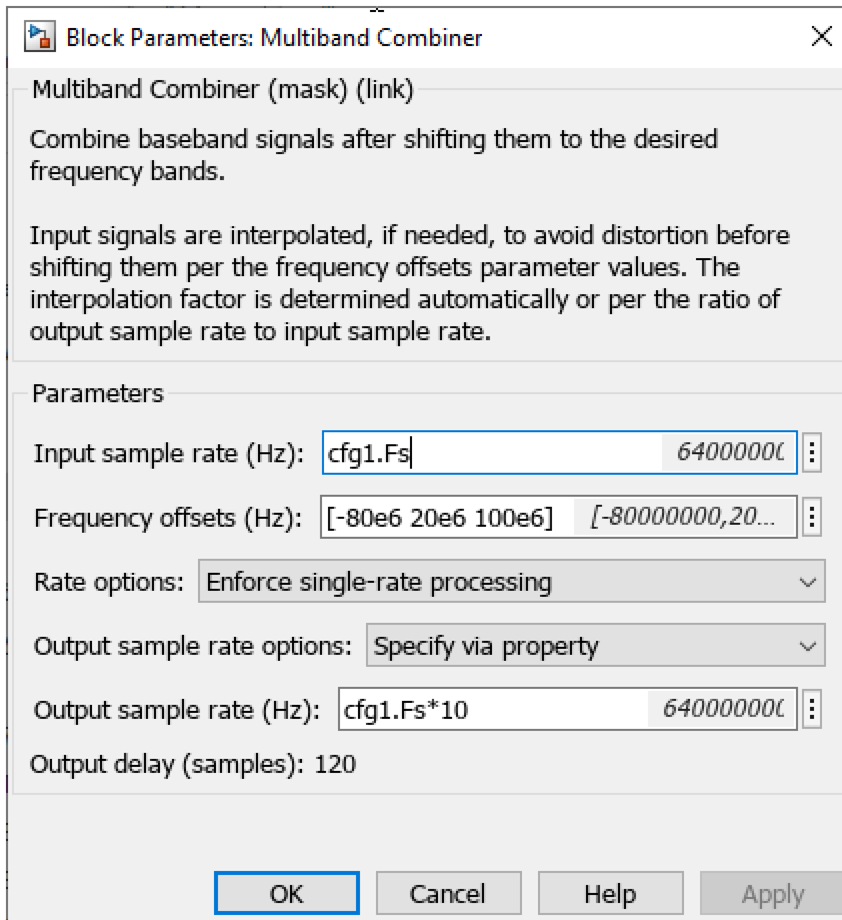
For multicarrier generation, the sampling rates for all of the waveforms must be the same. To shift the waveforms to a carrier offset and aggregate them, you can use the Multiband Combiner block.

```
modelName = 'WWGExport2SimulinkMulticarrier';
open_system(modelName);
```




```
% Copyright 2021-2023 The MathWorks, Inc.
```

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

Blocks
Waveform From Wireless Waveform Generator App

Related Examples

- “Modeling and Testing an 802.11ax RF Transmitter” on page 8-21
- “Modeling and Testing an 802.11ax RF Receiver with 5G Interference” on page 8-2

802.11ad Transmitter Spectral Emission Mask Testing

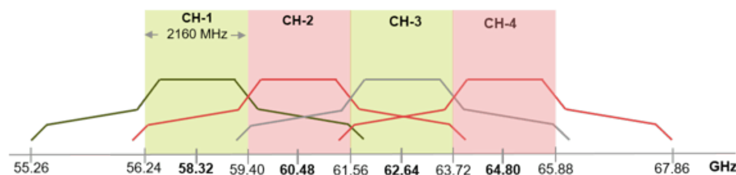
This example shows how to perform pulse shaping and spectrum emission mask testing on an IEEE® 802.11ad™ waveform.

Introduction

IEEE 802.11ad [1] standard, commonly referred to as directional multi-gigabit (DMG), provides data throughput up to 7 Gbps using the 60 GHz industrial, scientific, and medical (ISM) frequency band. The DMG standard supports three PHY types:

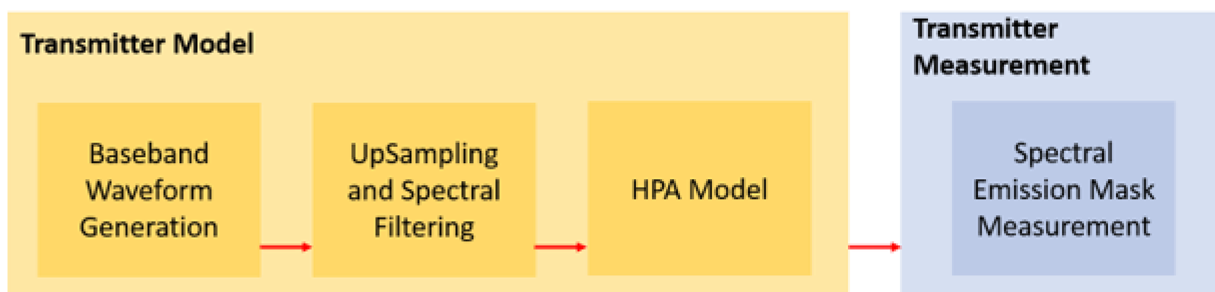
- A Control PHY using MCS 0
- A Single Carrier (SC) PHY using MCS 1 to MCS 12 and low power SC PHY using MCS 25 to MCS 32
- An OFDM PHY using MCS 13 to MCS 24.

DMG defines four 2.16 GHz wide operating channels, typically in the 57-66 GHz band. The spectral mask test, as demonstrated in this example, ensures a transmission in one channel does not cause substantial interference in adjacent channels. The DMG channelization is shown in the figure below.



The SC DMG PHY uses single-carrier (SC) modulation for low cost, short range applications. This example shows how pulse shaping and spectrum mask measurements can be performed on an SC DMG modulated waveform. The waveform is generated using WLAN Toolbox™, but a waveform captured with a spectrum analyzer could also be used. The transmitter spectrum mask and required spectral flatness for DMG configuration is specified in IEEE 802.11ad [1], Section 20.3.2.

This example generates five DMG SC packets, each separated by a one microsecond gap. Random data is used in each packet and pi/2-16QAM modulation is used. To meet the spectral mask requirements, the baseband waveform is upsampled and filtered to reduce the out-of-band emissions. A high power amplifier (HPA) model is used to introduce inband distortion and spectral regrowth. The spectral emission mask measurement is performed on the upsampled waveform after the HPA modeling. The test schematic is illustrated in the following diagram:



DMG, Single Carrier Packet Configuration

In this example, an IEEE 802.11ad waveform consisting of multiple DMG SC packets is generated. The DMG SC waveform properties are specified in a `wlanDMGConfig` configuration object. The object is configured for an MCS index of 5, with no `TrainingLength` fields appended to the packets. Per the test requirement (specified in IEEE 802.11ad Section 21.3.2), the `PSDULength` is set to 20000 for the packet to ensure that the transmit spectral mask is measured on a DMG packet longer than 10 microseconds.

```
cfgDMG = wlanDMGConfig;    % DMG packet configuration
cfgDMG.MCS = 5;            % SC PHY with pi/2-BPSK modulation
cfgDMG.PSDULength = 20000; % Length in Bytes
```

Baseband Waveform Generation

The waveform generator can be configured to generate one or more packets with idle time between each packet. In this example, `wlanWaveformGenerator` is configured to generate five packets filled with random payload data. Each packet is separated by a one microsecond idle period in between and a random scrambler seed is used to generate each packet.

```
% Set random stream for repeatability of results
s = rng(98765);

% Generate a multi-packet waveform
idleTime = 1e-6; % One microsecond idle time between packets
numPackets = 5;  % Generate five packets

% Create random bits for all payload data; PSDULength is in bytes
psdu = randi([0 1],cfgDMG.PSDULength*8*numPackets,1);

% Override the ScramblerInitialization property of the DMG configuration
% object by specifying the scrambler initialization
genWaveform = wlanWaveformGenerator(psdu,cfgDMG,...
    'IdleTime',idleTime, ...
    'NumPackets',numPackets, ...
    'ScramblerInitialization',randi([1 127],numPackets,1));

% Get the sampling rate of the waveform
fs = wlanSampleRate(cfgDMG);
disp(['Baseband sampling rate: ' num2str(fs/1e6) ' Msps']);
```

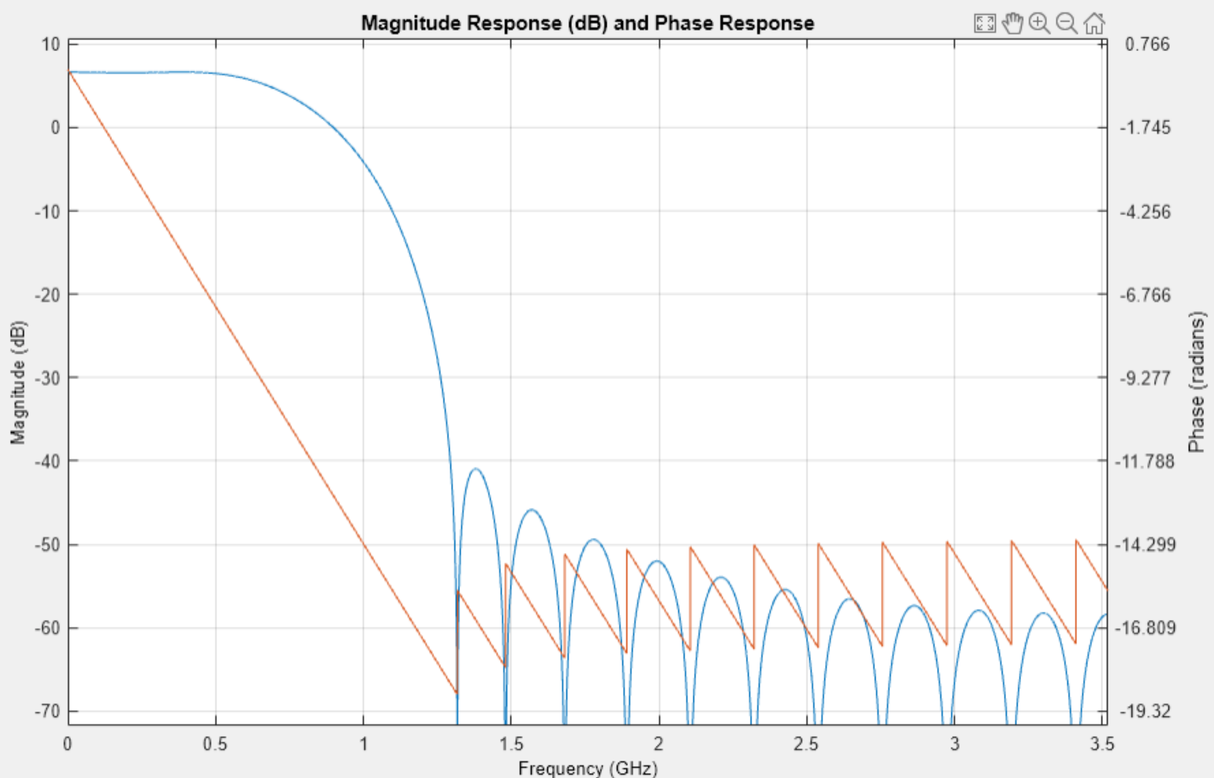
```
Baseband sampling rate: 1760 Msps
```

Oversampling and Filtering

Spectral filtering is used to reduce the out-of-band spectral emissions due to the spread spectrum characteristics of the transmitted waveform and spectral regrowth caused by the HPA in an RF chain. The waveform must be oversampled to model the effect of the HPA on the waveform and view the out-of-band spectral emissions. In this example, the waveform is oversampled and filtered through a raised cosine filter using `comm.RaisedCosineTransmitFilter`. To meet the spectral mask requirements, the raised cosine filter is truncated to the duration of eight symbols and the roll-off factor is set to 0.5.

```
% Define the pulse shaping filter characteristics
Nsym = 8;   % Filter span in symbol durations
beta = 0.5; % Roll-off factor
osps = 4;   % Output samples per symbol
```

```
% Create raised cosine transmit filter system object
rcosFlt = comm.RaisedCosineTransmitFilter(...
    'Shape','Normal', ...
    'RolloffFactor',beta, ...
    'FilterSpanInSymbols',Nsym, ...
    'OutputSamplesPerSymbol',osps);

% Filter transmit signal for pulse shaping
filterWaveform = rcosFlt([genWaveform; zeros(Nsym/2,1)]);

% Plot the magnitude and phase response of the pulse shaping filter
h = fvtool(rcosFlt,'Analysis','freq');
h.FS = osps*fs;                  % Set sampling rate
h.NormalizedFrequency = 'off'; % Plot responses against frequency
```



High Power Amplifier Modeling

Within an RF chain, the HPA is a necessary component but it introduces nonlinear behavior in the form of inband distortion and spectral regrowth. The Rapp model, described in [2], can be used to model an 802.11ad power amplifier. The Rapp model causes AM/AM distortion and is modeled with `comm.MemorylessNonlinearity`. The HPA is backed-off to operate below the saturation point to reduce distortion.

```
hpaBackoff = 0.5; % Power Amplifier backoff in dB

% Create and configure a memoryless nonlinearity to model HPA
```

```
nonLinearity = comm.MemorylessNonlinearity;
nonLinearity.Method = 'Rapp model';
nonLinearity.Smoothness = 0.81;                          % Smoothness factor
nonLinearity.LinearGain = 20*log10(4.65) - hpaBackoff; % Small signal gain
nonLinearity.OutputSaturationLevel = 0.58;             % Saturation level

% Apply the model
txWaveform = nonLinearity(filterWaveform);
```

Transmit Spectrum Emission Mask Measurement

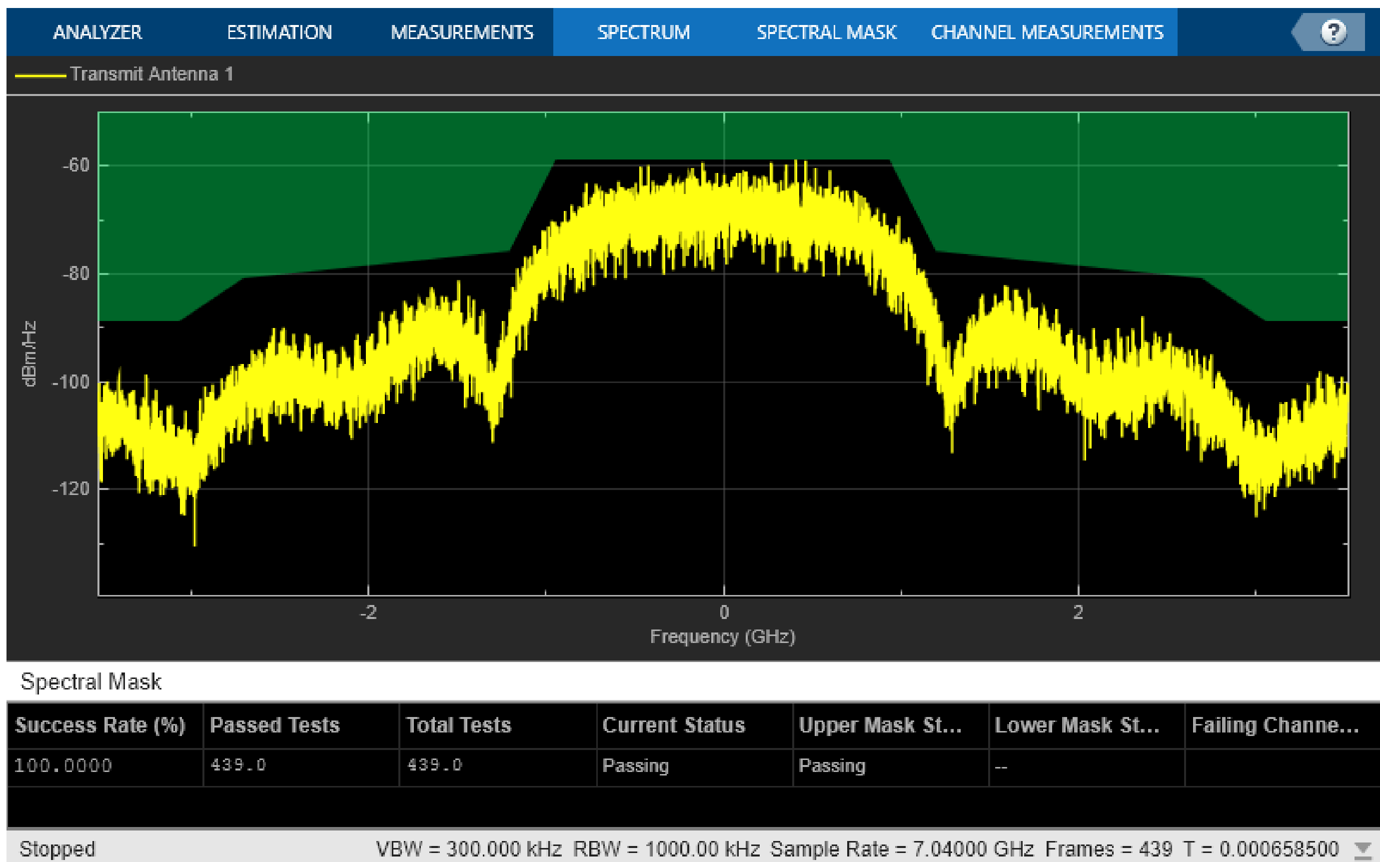
IEEE 802.11ad [1], Section 20.3.2, specifies the transmit spectral mask that all DMG waveforms must adhere to and describes the packet characteristics. According to the test definition, packets should have no training fields appended and be greater than 10 microseconds in duration.

```
dBrLimits = [-30 -30 -22 -17 0 0 -17 -22 -30 -30];
fLimits = [-Inf -3.06 -2.7 -1.2 -0.94 0.94 1.2 2.7 3.06 Inf] * 1e3;
rbw = 1e6; % Resolution bandwidth in Hz
vbw = 300e3; % Video bandwidth in Hz
```

Use the helper function `helperSpectralMaskTest` to generate a plot that overlays the required spectral mask with the measured PSD. It checks the transmitted PSD levels to be within the specified mask levels and displays a pass/fail status after the test.

```
helperSpectralMaskTest(txWaveform,fs,osps,dBrLimits,fLimits,rbw,vbw);

% Restore default stream
rng(s);
```

```
Spectrum mask passed
```



Conclusion and Further Exploration

The transmit spectral mask for a DMG SC waveform in the 60 GHz band for a 2.16 GHz channel bandwidth is shown in this example. It also illustrates that the spectrum of the transmitted signal satisfies regulatory restrictions by falling within the spectral mask after pulse shaping. A similar result can be generated for DMG Control and OFDM PHYs.

The HPA model and the spectral filtering affect the out-of-band emissions in the spectral mask plot. For Single Carrier and Control PHY, you can try using different pulse shaping filter parameters and/or decrease or increase the smoothness factor.

For information on other transmitter measurements like modulation accuracy and spectral flatness, refer to the following examples:

- “802.11ac Transmitter Measurements” on page 8-46
- “802.11p Spectral Emission Mask Testing” on page 8-79

Selected Bibliography

1. IEEE Std 802.11™-2020. 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.
2. Eldad Perahia, et. al. TGad Evaluation Methodology, IEEE 802.11-09/0296r16

802.11p Spectral Emission Mask Testing

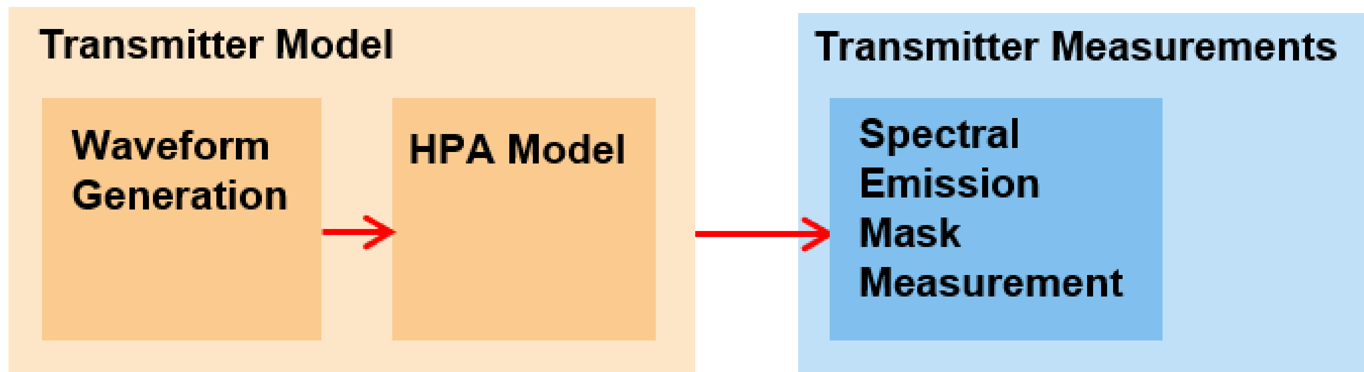
This example shows how to perform spectrum emission mask tests for an IEEE® 802.11p™ transmitted waveform.

Introduction

IEEE 802.11p is an approved amendment to the IEEE 802.11™ standard to enable support for wireless access in vehicular environments (WAVE). Using the half-clocked mode with a 10 MHz channel bandwidth, it operates at the 5.85-5.925 GHz bands, for which it defines additional spectral emission masks.

This example shows how to perform spectral mask measurements on a waveform generated by using WLAN Toolbox™ software. Alternatively, you can perform this measurement on a waveform captured with a spectrum analyzer.

The example generates a waveform consisting of three 10 MHz IEEE 802.11p packets separated by a 32 microsecond gap. Each packet contains random data and uses 16-QAM. The example oversamples the waveform using a larger IFFT than required for the nominal baseband rate and does not perform spectral filtering, and uses a high-power amplifier (HPA) model, which introduces in-band distortion and spectral regrowth. The example performs spectral emission mask measurement on the upsampled waveform after HPA modeling. This diagram shows the test setup.



Non-HT Packet Configuration

Set the non-high-throughput (non-HT) transmission parameters by using the `wlanNonHTConfig` object, specifying 10 MHz bandwidth operation as used by IEEE 802.11p.

```
cfgNHT = wlanNonHTConfig;          % Create packet configuration
cfgNHT.ChannelBandwidth = 'CBW10'; % 10 MHz
cfgNHT.MCS = 4;                    % Modulation 16QAM, rate-1/2
cfgNHT.PSDULength = 1000;          % PSDU length in bytes
```

Waveform Generation

This section configures and generates a waveform containing three packets with an idle time of 32 microseconds between each packet.

To model the effect of a HPA on the waveform and view the out-of-band spectral emissions, the waveform must be oversampled. This example generates the waveform using a larger IFFT than

required for the nominal baseband rate, resulting in an oversampled waveform. The example does not perform spectral filtering.

Set random stream for repeatability of results, specify an oversampling factor, and generate random data of the required PSDU length.

```
s = rng(98765);
osf = 3;
idleTime = 32e-6;
numPackets = 3;
data = randi([0 1],cfgNHT.PSDULength*8*numPackets,1);
```

Generate the multi-packet waveform and calculate the nominal baseband sample rate.

```
genWaveform = wlanWaveformGenerator(data,cfgNHT, ...
                OversamplingFactor=osf, ...
                NumPackets=numPackets,...
                IdleTime=idleTime);
fs = wlanSampleRate(cfgNHT);
```

HPA Modeling

The HPA introduces nonlinear behavior in the form of inband distortion and spectral regrowth. This example simulates the power amplifiers by using the Rapp model in 802.11ac, which introduces AM/AM distortion.

Model the amplifier by using `comm.MemorylessNonlinearity` object, and configure reduced distortion by specifying a back-off, `hpaBackoff`, such that the amplifier operates below its saturation point. You can increase the backoff to reduce EVM for higher MCS values.

```
pSaturation = 25; % Saturation power of a power amplifier in dBm
hpaBackoff = 16; % dB
```

Create and configure a memoryless nonlinearity to model the amplifier.

```
nonLinearity = comm.MemorylessNonlinearity;
nonLinearity.Method = 'Rapp model';
nonLinearity.Smoothness = 3; % p parameter
nonLinearity.LinearGain = -hpaBackoff; % dB
nonLinearity.OutputSaturationLevel = db2mag(pSaturation-30);
```

Apply the model to the transmit waveform.

```
txWaveform = nonLinearity(genWaveform);
```

Transmit Spectrum Emission Mask Measurement

This section performs the spectral emission mask test non-HT Data field.

The IEEE 802.11p standard classifies stations according to the allowed maximum transmit powers (in mW). For the four different classes of stations, the standard defines four different spectral emission masks, and defines the spectral mask limits relative to the peak power spectral density (PSD). This example measures the spectrum emission mask for a Class A station.

```
% IEEE Std 802.11-2016 Annex D.2.3, Table D-5: Class A STA
dBrLimits = [-40  -40 -28 -20  -10 0   0  -10 -20 -28 -40 -40];
fLimits   = [-Inf -15 -10 -5.5 -5 -4.5 4.5 5  5.5  10  15 Inf];
```

Extract the non-HT Data field of each packet from the oversampled `txWaveform` by using the start index of each packet and concatenate the extracted non-HT Data fields in preparation for measurement.

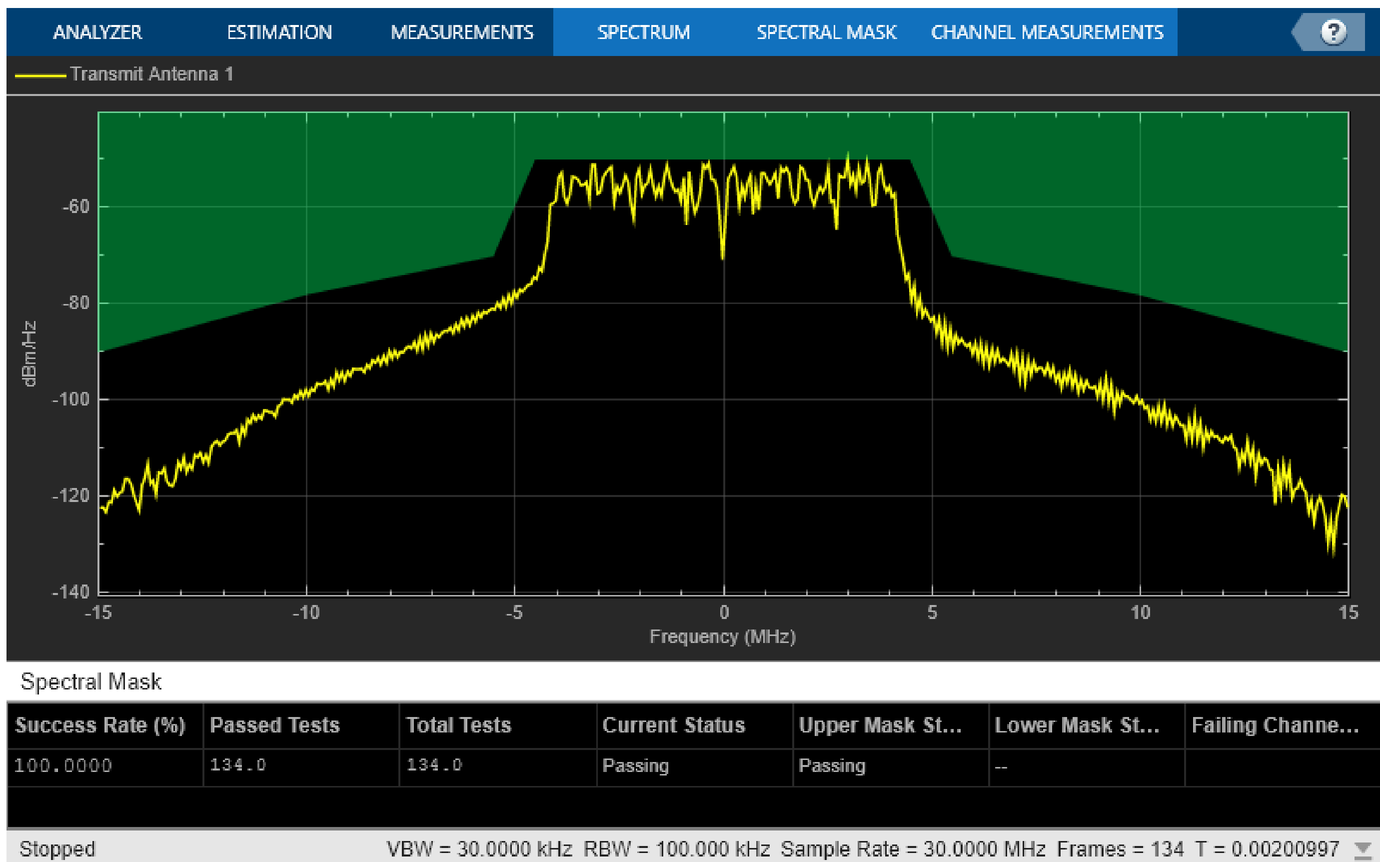
```
ind = wlanFieldIndices(cfgNHT, OversamplingFactor=osf);
startIdx = ind.NonHTData(1); % Start of non-HT Data
endIdx = ind.NonHTData(2); % End of non-HT Data
idleNSamps = idleTime*fs*osf; % Idle time samples
perPktLength = endIdx+idleNSamps;

idx = zeros(endIdx-startIdx+1, numPackets);
for i = 1:numPackets
    % Start of packet in txWaveform, accounting for the filter delay
    pktOffset = (i-1)*perPktLength;
    % Indices of non-HT Data in txWaveform
    idx(:,i) = pktOffset+(startIdx:endIdx);
end
% Select the Data field for the individual packets
gatedNHTDataTx = txWaveform(idx(:), :);
```

The `helperSpectralMaskTest` function overlays the required spectral mask with the measured PSD and checks that the transmitted PSD levels is within the specified mask levels, displaying the result as a pass or fail.

Evaluate PSD and test for compliance.

```
helperSpectralMaskTest(gatedNHTDataTx,fs,osf,dBrLimits,fLimits);
```

```
Spectrum mask passed
```



Restore the default random stream.

```
rng(s);
```

Conclusion and Further Exploration

This example shows how to measure the transmit spectral mask for Class A Stations at the 5.85-5.925 GHz bands for a 10 MHz channel spacing, and how to ensure that the peak spectral density of the transmitted signal falls within the spectral mask to satisfy regulatory restrictions. You can generate a similar result for a 5 MHz channel spacing.

The HPA model affects the out-of-band emissions in the spectral mask plot. For different station classes with higher relative dB values, try increasing the backoff for lower emissions.

For information on other transmitter measurements like modulation accuracy and spectral flatness, and other formats, refer to the following examples:

- “802.11be Transmitter Measurements” on page 3-2
- “802.11ba WUR Waveform Generation and Analysis” on page 3-14
- “802.11ad Transmitter Spectral Emission Mask Testing” on page 8-74
- “802.11ac Transmitter Measurements” on page 8-46
- “Recover and Analyze Packets in 802.11 Waveform” on page 4-2

Selected Bibliography

1. IEEE Std 802.11-2016: 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, IEEE, New York, NY, USA, 1999-2016.
2. IEEE Std 802.11p-2010: 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 6: Wireless Access in Vehicular Environments, IEEE, New York, NY, USA, 2010.
3. Archambault, Jerry, and Shravan Surineni. "IEEE 802.11 spectral measurements using vector signal analyzers." RF Design 27.6 (2004): 38-49.

9

Code Generation and Deployment

What is C Code Generation from MATLAB?

You can use WLAN Toolbox together with MATLAB® Coder™ to:

- Create a MEX file to speed up your MATLAB application.
- Generate ANSI®/ISO® compliant C/C++ source code that implements your MATLAB functions and models.
- Generate a standalone executable that runs independently of MATLAB on your computer or another platform.

In general, the code you generate using the toolbox is portable ANSI C code. In order to use code generation, you need a MATLAB Coder license. For more information, see “Get Started with MATLAB Coder” (MATLAB Coder).

Using MATLAB Coder

Creating a MATLAB Coder MEX file can substantially accelerate your MATLAB code. It is also a convenient first step in a workflow that ultimately leads to completely standalone code. When you create a MEX file, it runs in the MATLAB environment. Its inputs and outputs are available for inspection just like any other MATLAB variable. You can then use MATLAB tools for visualization, verification, and analysis.

The simplest way to generate MEX files from your MATLAB code is by using the `codegen` function at the command line. For example, if you have an existing function, `myfunction.m`, you can type the commands at the command line to compile and run the MEX function. `codegen` adds a platform-specific extension to this name. In this case, the `"mex"` suffix is added.

```
codegen myfunction.m
myfunction_mex;
```

Within your code, you can run specific commands either as generated C code or by using the MATLAB engine. In cases where an isolated command does not yet have code generation support, you can use the `coder.extrinsic` command to embed the command in your code. This means that the generated code reenters the MATLAB environment when it needs to run that particular command. This is also useful if you want to embed commands that cannot generate code (such as plotting functions).

To generate standalone executables that run independently of the MATLAB environment, create a MATLAB Coder project inside the MATLAB Coder Integrated Development Environment (IDE). Alternatively, you can call the `codegen` command in the command line environment with appropriate configuration parameters. A standalone executable requires you to write your own `main.c` or `main.cpp` function. See “Generating Standalone C/C++ Executables from MATLAB Code” (MATLAB Coder) for more information.

C/C++ Compiler Setup

Before using `codegen` to compile your code, you must set up your C/C++ compiler. For 32-bit Windows platforms, MathWorks® supplies a default compiler with MATLAB. If your installation does not include a default compiler, you can supply your own compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website. Install a compiler that is suitable for your platform, then read “Setting Up the C or C++ Compiler” (MATLAB Coder).

After installation, at the MATLAB command prompt, run `mex -setup`. You can then use the `codegen` function to compile your code.

Functions and System Objects That Support Code Generation

All WLAN Toolbox functions and System objects support code generation.

See Also

Functions
`codegen` | `mex`

More About

- "Code Generation Workflow" (MATLAB Coder)
- Generate C Code from MATLAB Code Video

Code Generation of WLAN Toolbox Features

This example shows how to generate MEX files and efficient C/C++ code for the WLAN Toolbox™ waveform generator function and verify its correct behavior. Additionally, it shows how to work around limitations of code generation for WLAN Toolbox channel models. All WLAN Toolbox functions and System objects™ are supported for C/C++ code generation.

Introduction

With MATLAB Coder™, you can generate efficient, portable C source code, standalone executables, and MEX functions for deployment in desktop and embedded applications. You can speed up your MATLAB code using MEX functions or integrate generated source code and static or dynamic libraries into your C/C++ code projects. See “Setting Up the C or C++ Compiler” (MATLAB Coder) for more information about how to set up the C/C++ compiler. In order to use code generation, you need a MATLAB Coder license.

This example uses the `codegen` function to generate code for the WLAN Toolbox waveform generator and the TGax channel System object. First, prepare the required input arguments of the waveform generator and configure MATLAB Coder to generate MEX files. Next, compare the output of the generated MEX files and the original MATLAB code. Finally, the example shows how to work around limitations of code generation affecting WLAN Toolbox.

Input Arguments Definition

For code generation, you must specify the size and type of the input arguments to your entry-point function. In this case, you specify the input arguments of the `wlanWaveformGenerator` function and packet format configuration object. The mandatory arguments of the waveform generator are the data bits of the physical layer service data unit (PSDU) and the waveform format configuration. Additional inputs such as the idle time, scrambler initialization, and window transition time depend on the format configuration. Specify the additional inputs as name-value pairs. See `wlanWaveformGenerator` for more information.

You can use example values of the input arguments and the `codegen` function automatically derives their size, class, and complexity. However, the size of the input data vector depends on properties of the format configuration such as the channel bandwidth and coding. Use the `coder.typeof` function to define a variable-size input data vector that adapts to the format configuration properties as required.

Thus, you can run the generated code for multiple parameter sets of the format configuration object. For more information about specifying variable-size input arguments, see “Generate Code for Variable-Size Data” (MATLAB Coder). In this case, specify only the first dimension as variable-size.

```
variableDims = [1 0];
```

Variable-size vectors can be upper bounded or unbounded. To allow for full flexibility, you can select the upper bound to be the maximum aggregated MAC protocol data unit (A-MPDU) length, i.e., 6500631 bytes.

```
upperBound = 6500631*8; % Upper bound in bits
```

Next, use `coder.typeof` to define an input type as a column vector of doubles with a maximum size of `upperBound`-by-1, with the first dimension variable-size.

```
inputBits = coder.typeof(double(0),[upperBound 1],variableDims);
```

As the inputBits data type is double, you can only use double-precision values for the input bits of the generated MEX and C/C++ code.

Next, create a high-efficiency single-user (HE SU) format configuration object with default parameters and its corresponding configuration structure using the coder.typeof function. Use this structure to specify the type and size of any property of the HE SU format configuration object.

```
cfgHESU = wlanHESUConfig;
cfgHESUcg = coder.typeof(cfgHESU);
```

There are two possible forward-error-correction (FEC) coding types for the HE-Data: low-density parity-check, specified by setting the ChannelCoding property of cfgHESU to 'LDPC', and binary convolutional coding, specified as 'BCC'. Since these are character arrays of different lengths, use the coder.typeof function to define a variable-length row vector of maximum size 1-by-4 characters.

```
cfgHESUcg.Properties.ChannelCoding = coder.typeof('LDPC',[1 4],[0 1]);
```

Use the same strategy for the channel bandwidth property. In this case, the longest character array corresponds to 'CBW320', so six characters are enough to cover all other possible cases, i.e., 'CBW20', 'CBW40', 'CBW80', 'CBW160', and 'CBW320'.

```
cfgHESUcg.Properties.ChannelBandwidth = coder.typeof('CBW320',[1 6],[0 1]);
```

Specify optional arguments of the waveform generator such as the windowing transition time using name-value pairs. Use coder.Constant to define the name of the argument because it is a string literal that is not expected to change.

```
WindowTransitionTime_Name = coder.Constant('WindowTransitionTime');
```

Use an example value from which the codegen function derives its type and size. This technique is only suitable for fixed-size input arguments.

```
WindowTransitionTime_Value = 0;
```

Code Generation for the WLAN Toolbox Waveform Generator

Once the input arguments are specified for code generation, configure MATLAB Coder to generate MEX files and C/C++ code. A MEX file acts as an interface to the generated C/C++ code that can run in MATLAB. MEX file generation is usually the first step in the code generation workflow as it provides a convenient way for verifying the generated C/C++ code.

You can generate a MEX file, C/C++ code, a dynamic library, or a standalone executable by creating a MATLAB Coder configuration object and specifying the build type as 'MEX', 'LIB', 'DLL', or 'EXE', respectively.

```
BuildType = 'MEX';
cfgCoder = coder.config(BuildType);
```

Generate a report containing useful information about the code generation process.

```
cfgCoder.GenerateReport = true;
```

The name of the generated MEX file is the entry-point function name appended by the suffix 'mex', that is, wlanWaveformGenerator_mex. To specify a different name for your MEX file, use the option -o output_file_name. You can find the generated MEX files in your local folder and the generated C/C++ code in the folder codegen\mex\wlanWaveformGenerator.

Users of Microsoft Visual C++ product family may see the C4101 compiler warning indicating an unreferenced local variable.

```
inputArgs = {inputBits,cfgHESUcg,WindowTransitionTime_Name,WindowTransitionTime_Value}; %#ok<NASG
codegen wlanWaveformGenerator -args inputArgs -config cfgCoder -o wlanWaveformGenerator_HESU_mex
```

```
Code generation successful: To view the report, open('codegen\mex\wlanWaveformGenerator\html\repo
```

Next, verify that the generated MEX file behaves as expected when you use different configurations of the `ChannelBandwidth` and `ChannelCoding` by comparing its output to that of the `wlanWaveformGenerator`.

```
% Create a HE SU format configuration object specifying the channel
% bandwidth, coding, number of antennas and streams.
cfgHESU = wlanHESUConfig('ChannelBandwidth','CBW20','ChannelCoding','BCC', ...
                         'NumTransmitAntennas',2,'NumSpaceTimeStreams',2);

% Set the value of the window transition time
WindowTransitionTime = 1e-09;

% Create a PSDU of size defined by the getPSDULength function
inputBits = randi([0 1],getPSDULength(cfgHESU)*8,1);

% Run the wlanWaveformGenerator
waveformMAT = wlanWaveformGenerator(inputBits,cfgHESU,'WindowTransitionTime',WindowTransitionTime

% Run the generated MEX file
waveformMEX = wlanWaveformGenerator_HESU_mex(inputBits,cfgHESU,'WindowTransitionTime',WindowTrans

% Compare the outputs of the wlanWaveformGenerator and generated MEX file
difference = waveformMAT - waveformMEX;

% Check the results are consistent
if max(abs(difference),[],'all') > 1e-10
    error('The MEX file generated does not produce the same results as wlanWaveformGenerator.')
else
    disp('The outputs of the generated MEX file and the wlanWaveformGenerator are equal.')
end
```

```
The outputs of the generated MEX file and the wlanWaveformGenerator are equal.
```

Additional Options for Code Generation

In addition to the reporting options used above, you can configure MATLAB Coder with more advanced options.

- Generate C/C++ code only but do not build object code or MEX files setting `cfgCoder.GenCodeOnly = true`. This can save time when you iterate between modifying MATLAB code and inspecting the generated C/C++ code. To generate a MEX file, set this value to `false`.
- Configure MATLAB Coder for optimized C/C++ code generation by setting `cfgCoder.BuildConfiguration = 'Faster Runs'`. This option is available only for `'LIB'`, `'DLL'`, and `'EXE'` build types.

See the `coder.config` (MATLAB Coder) reference page for more information.

Limitations of Code Generation

“WLAN Channel Models” are System objects, which are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Consider these limitations for code generation of System objects:

- Nontunable property values must be constant and can only be assigned once before you call the object, including the assignment in the constructor.
- You cannot pass in a System object™ to an entry-point function.

See “System Objects in MATLAB Code Generation” (MATLAB Coder) for more information about rules and limitations of System objects™ for code generation.

If you try to generate code for the entry-point function named `hCustomChannelNT`, which creates a `wlanTGaxChannel` with a bandwidth specified by the input argument `BW` and filters the signal `inputSignal`, the code generation process fails.

```
function signalOut = hCustomChannelNT(BW, signalIn)

    % Create a TGax channel with the appropriate bandwidth
    ch = wlanTGaxChannel('ChannelBandwidth',BW);
    % Filter the input signal
    signalOut = ch(signalIn);

end
```

The error message indicates: `Failed to compute constant value for nontunable property 'ChannelBandwidth'`.

The `ChannelBandwidth` property of the `wlanTGaxChannel` object is not a constant value because it depends on the input argument `BW` of the `hCustomChannelNT` function. You can work around this by setting `BW` to be a constant value.

```
inputSignal = coder.typeof(complex(0),[Inf 1],[1 0]); %#ok<NASGU>
codegen hCustomChannelNT -args {coder.Constant('CBW320'),inputSignal};
```

```
Code generation successful.
```

However, if you specify a constant value for the channel bandwidth, you cannot change the channel bandwidth at runtime. The following call `hCustomChannelNT_mex('CBW20',inputSignal)` fails because the specified argument `BW` is not `'CBW320'`. It is possible to work around this for a limited variability of the input arguments using a switch-case block. The function `hCustomChannel` contains a switch-case where each of the cases creates a `wlanTGaxChannel` with the appropriate channel bandwidth.

```
function signalOut = hCustomChannel(ChannelBandwidth,signalIn)
%hCustomChannel Creates a TGax channel with the appropriate bandwidth, sample rate and filter the

switch ChannelBandwidth
    case 'CBW20'
        fs = 20e6; % Input signal sample rate
        ch = wlanTGaxChannel('ChannelBandwidth','CBW20','SampleRate',fs);
```

```
        signalOut = ch(signalIn);
    case 'CBW40'
        fs = 40e6; % Input signal sample rate
        ch = wlanTGaxChannel('ChannelBandwidth','CBW40','SampleRate',fs);
        signalOut = ch(signalIn);
    case 'CBW80'
        fs = 80e6; % Input signal sample rate
        ch = wlanTGaxChannel('ChannelBandwidth','CBW80','SampleRate',fs);
        signalOut = ch(signalIn);
    case 'CBW160'
        fs = 160e6; % Input signal sample rate
        ch = wlanTGaxChannel('ChannelBandwidth','CBW160','SampleRate',fs);
        signalOut = ch(signalIn);
    case 'CBW320'
        fs = 320e6; % Input signal sample rate
        ch = wlanTGaxChannel('ChannelBandwidth','CBW320','SampleRate',fs);
        signalOut = ch(signalIn);
    otherwise
        error('Invalid bandwidth configuration.')
end

end
```

Note that `hCustomChannel` creates a new channel object every time it is used, so it does not preserve the state of the channel.

Next, generate code for `hCustomChannel` and select multiple channel bandwidths at runtime.

```
inputSignal = coder.typeof(complex(0),[Inf 1],[1 0]);
BW = coder.typeof('CBW320',[1 6],[0 1]);
inputArgs = {BW,inputSignal};
codegen hCustomChannel -args inputArgs

cfgHESU = wlanHESUConfig;
cfgHESU.ChannelBandwidth = 'CBW20';

inputSignal = wlanWaveformGenerator_HESU_mex([1 0 1]',cfgHESU,'WindowTransitionTime',0);

rng('default') % Set the default random number generator for reproduction purposes
outputSignalMEX = hCustomChannel_mex('CBW20',inputSignal);
rng('default')
outputSignalMAT = hCustomChannel('CBW20',inputSignal);

% Calculate the difference between the MEX and MATLAB files outputs
difference = outputSignalMAT - outputSignalMEX;

% Check the results are consistent
if max(abs(difference),[],'all') > 1e-10
    error('The generated MEX file does not produce the same results as hCustomChannel.')
else
    disp('The outputs of the generated MEX file and hCustomChannel are equal.')
end
```

```
Code generation successful.

The outputs of the generated MEX file and hCustomChannel are equal.
```

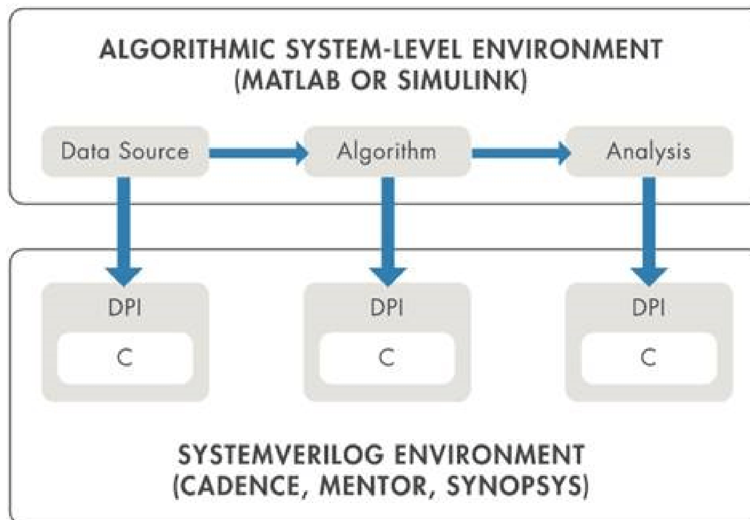
Change the channel bandwidth and pass the signal through the channel.

```
cfgHESU.ChannelBandwidth = 'CBW40';
inputSignal = wlanWaveformGenerator_HESU_mex([1 0 1]',cfgHESU,'WindowTransitionTime',0);

rng('default') % Set the default random number generator for reproduction purposes
outputSignalMEX = hCustomChannel_mex('CBW40',inputSignal);
rng('default')
outputSignalMAT = hCustomChannel('CBW40',inputSignal);

% Calculate the difference between the MEX and MATLAB files outputs
difference = outputSignalMAT - outputSignalMEX;

% Check the results are consistent
if max(abs(difference),[],'all') > 1e-10
    error('The MEX file generated does not produce the same results as hCustomChannel.')
else
    disp('The outputs of the generated MEX file and hCustomChannel are equal.')
end
```

```
The outputs of the generated MEX file and hCustomChannel are equal.
```

Further Investigation

You can integrate the generated code with the SystemVerilog Direct Programming Interface (DPI) to export MATLAB algorithms to ASIC or FPGA verification environments including Synopsys VCS®, Cadence Incisive or Xcelium, and Mentor Graphics ModelSim or Questa. You can automatically generate SystemVerilog DPI components from MATLAB functions using MATLAB Coder™ with HDL Verifier™. The following workflow can be used with MATLAB functions that generate stimuli and perform analysis or with a MATLAB function that is a behavioral golden reference for the device under test (DUT) in the ASIC or FPGA verification environment. See “SystemVerilog DPI Component Generation” (HDL Verifier) (HDL Verifier) to learn more.



10

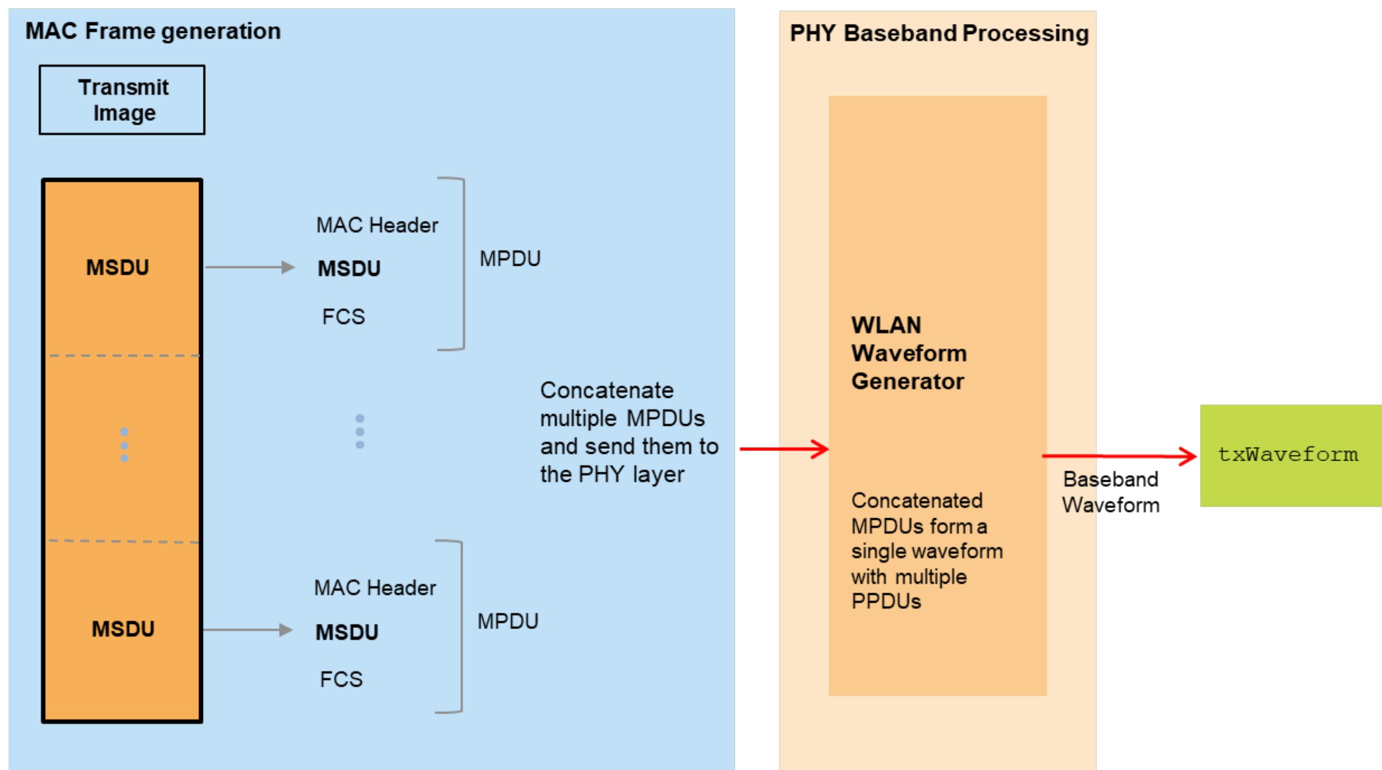
Software-Defined Radio

Image Transmission and Reception Using 802.11 Waveform and SDR

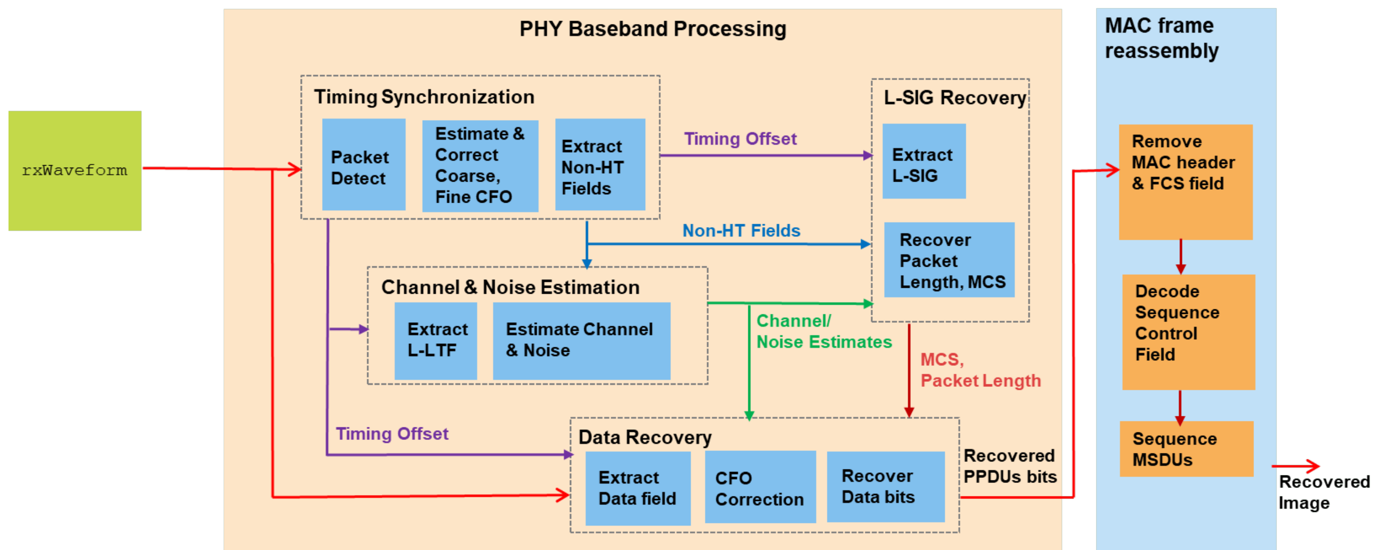
This example shows how to encode and pack an image file into WLAN packets for transmission and subsequently decode the packets to retrieve the image. The example also shows how to use a software-defined radio (SDR) for over-the-air transmission and reception of the WLAN packets.

Introduction

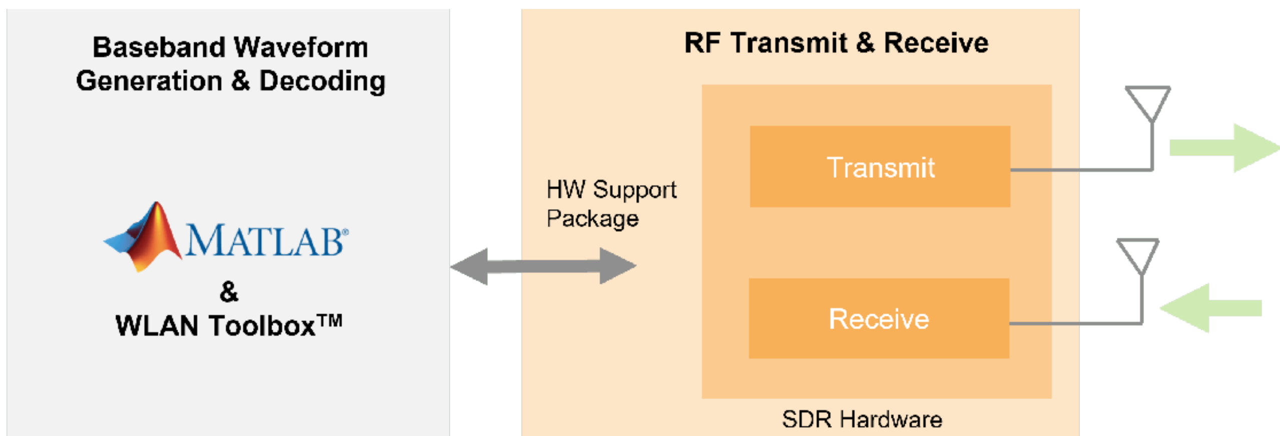
This example imports and segments an image file into multiple medium access control (MAC) service data units (MSDUs). It passes each MSDU to the `wlanMACFrame` function to create a MAC protocol data unit (MPDU). This function also utilizes a `wlanMACFrameConfig` object as an input, which sequentially numbers the MPDUs through the `SequenceNumber` property. The example then passes the MPDUs to the physical (PHY) layer as PHY Layer Service Data Units (PSDUs). Each PSDU uses a single non-high-throughput (non-HT), 802.11a™ WLAN packet for transmission. This example creates a WLAN baseband waveform using the `wlanWaveformGenerator` function. This function utilizes multiple PSDUs and processes each to form a series of physical layer convergence procedure (PLCP) protocol data units (PPDUs) ready for transmission.



The generated waveform then passes through an additive white Gaussian noise (AWGN) channel to simulate an over-the-air transmission. Subsequently, the `wlanMPDUDecode` function takes the noisy waveform and decodes it. Then, the `SequenceNumber` property in the recovered MAC frame configuration object allows the example to sequentially order the extracted MSDUs. The information bits in the multiple received MSDUs combine to recover the transmitted image. This diagram shows the receiver processing.



Alternatively, the generated WLAN waveform can be transmitted over the air and received using these supported SDRs.



Communications Toolbox Support Package for Analog Devices® ADALM-Pluto Radio

- “Installation and Setup” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio)
- “Supported Hardware” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio)

Communications Toolbox Support Package for USRP® Embedded Series Radio

- “Installation and Setup” (Communications Toolbox Support Package for USRP Embedded Series Radio)
- “Hardware Support” (Communications Toolbox Support Package for USRP Embedded Series Radio)

Communications Toolbox Support Package for Xilinx® Zynq®-Based Radio

- "Installation and Setup" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)
- "Hardware Support" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)

Example Setup

Before running the example, set the `channel` variable to be one of these options:

- `OverTheAir`: Use an SDR to transmit and receive the WLAN waveform
- `GaussianNoise`: Pass the transmission waveform through an AWGN channel (default)
- `NoImpairments`: Pass the transmission waveform through with no impairments

If you set `channel` to `OverTheAir`, set frequency band and channel, transmission gain, reception gain, and desired SDR:

- Set to `Pluto` to use the ADALM-Pluto Radio (default)
- Set to `E3xx` to use the USRP Embedded Series Radio
- Set to `AD936x` or `FMCOMMS5` to use the Xilinx Zynq-Based Radio

```
channel = GaussianNoise;
if strcmpi(channel,"OverTheAir")
    deviceName = Pluto;
    channelNumber = 5;
    frequencyBand = 2.4;
    txGain = -10;
    rxGain = 10;
elseif strcmpi(channel,"GaussianNoise")
    % Specify SNR of received signal for a simulated channel
    SNR = 20;
end
```

Configure all the scopes and figures for the example.

```
% Setup handle for image plot
if ~exist('imFig','var') || ~ishandle(imFig) %#ok<SUSENS>
    imFig = figure;
    imFig.NumberTitle = 'off';
    imFig.Name = 'Image Plot';
    imFig.Visible = 'off';
else
    clf(imFig); % Clear figure
    imFig.Visible = 'off';
end

% Setup Spectrum viewer
spectrumScope = spectrumAnalyzer( ...
    SpectrumType='power-density', ...
    Title='Received Baseband WLAN Signal Spectrum', ...
    YLabel='Power spectral density', ...
```

```
    Position=[69 376 800 450]);

% Setup the constellation diagram viewer for equalized WLAN symbols
refQAM = wlanReferenceSymbols('64QAM');
constellation = comm.ConstellationDiagram(...
    Title='Equalized WLAN Symbols',...
    ShowReferenceConstellation=true,...
    ReferenceConstellation=refQAM,...
    Position=[878 376 460 460]);
```

Transmitter Design

These steps describe the general procedure of the WLAN transmitter.

1. Import an image file and convert it to a stream of decimal bytes
2. Generate a baseband WLAN signal using the WLAN Toolbox
3. Pack the data stream into multiple 802.11a packets

If using an SDR, these steps describe the setup of the SDR transmitter.

1. Prepare the baseband signal for transmission using the SDR hardware
2. Send the baseband data to the SDR hardware for upsampling and continuous transmission at the desired center frequency

Prepare Image File

Read data from the image file, scale it for transmission, and convert it to a stream of decimal bytes. The scaling of the image reduces the quality by decreasing the size of the binary data stream.

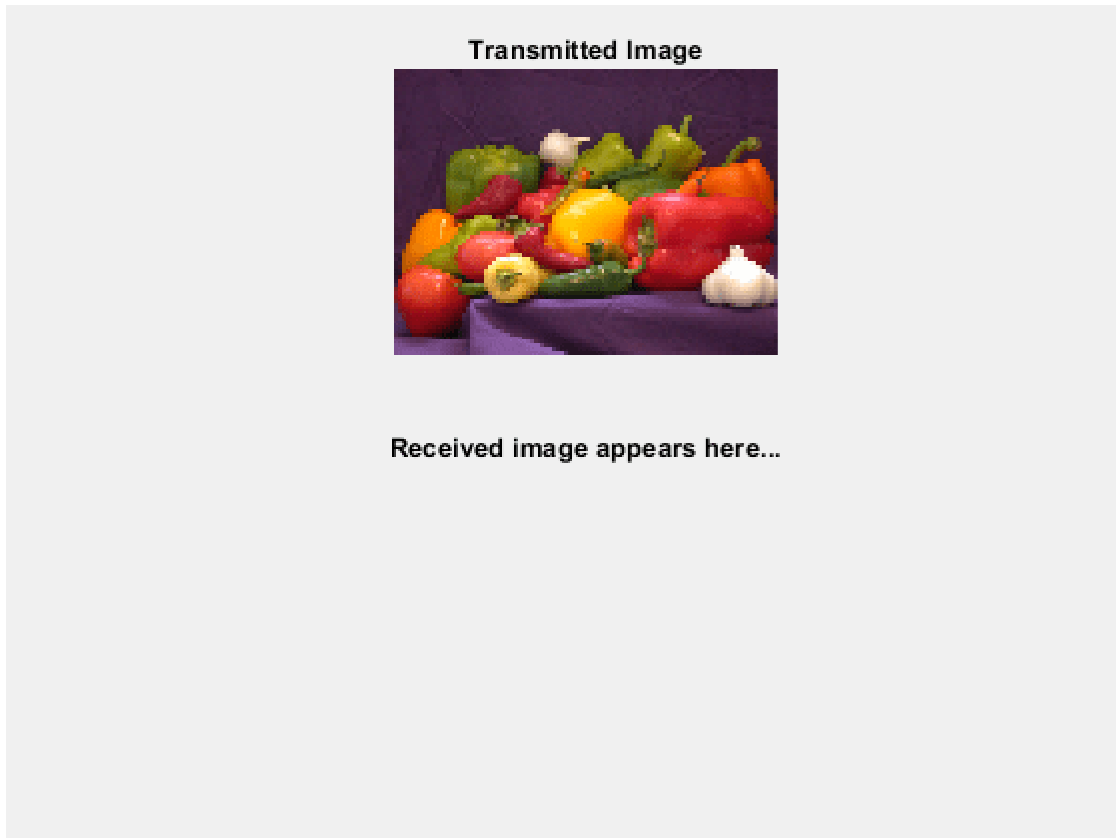
The size of the binary data stream impacts the number of WLAN packets required for the transmission of the image data. The number of WLAN packets generated for transmission depends on these factors.

1. The image scaling, set when importing the image file
2. The length of the data carried in a packet, specified by the `msduLength` variable
3. The modulation and coding scheme (MCS) value of the transmitted packet

The combination of the scaling factor and MSDU length determines the number of WLAN radio packets required for transmission. Setting `scale` to `0.2` and `msduLength` to `2304` requires the transmission of 11 WLAN radio packets. Increasing the scaling factor or decreasing the MSDU length will result in the transmission of more packets.

```
% Input an image file and convert to binary stream
fileTx = peppers.png;                              % Image file name
fData = imread(fileTx);                            % Read image data from file
scale = 0.2;                                       % Image scaling factor
origSize = size(fData);                            % Original input image size
scaledSize = max(floor(scale.*origSize(1:2)),1);   % Calculate new image size
heightIx = min(round(((1:scaledSize(1))-0.5)./scale+0.5),origSize(1));
widthIx = min(round(((1:scaledSize(2))-0.5)./scale+0.5),origSize(2));
fData = fData(heightIx,widthIx,:);                 % Resize image
imsize = size(fData);                              % Store new image size
```

```
txImage = fData(:);

% Plot transmit image
imFig.Visible = 'on';
subplot(211);
imshow(fData);
title('Transmitted Image');
subplot(212);
title('Received image appears here...');
set(gca,'Visible','off');
```




```
set(findall(gca, 'type', 'text'), 'visible', 'on');
```

Fragment Transmit Data

Split the data stream (`txImage`) into smaller transmit units (MSDUs) of size `msduLength`. Then, create an MPDU for each transmit unit using the `wlanMACFrame` function. Each call to this function creates an MPDU corresponding to the given MSDU and the frame configuration object. Next, create the frame configuration object using `wlanMACFrameConfig` to configure the sequence number of the MPDU. All the MPDUs are then sequentially passed to the physical layer for transmission.

To ensure that the MSDU size of the transmission does not exceed the standard-specified maximum, set the `msduLength` field to `2304` bytes. To make all MPDUs the same size, append zeros to the data in the last MPDU.

```
msduLength = 2304 ; % MSDU length in bytes
numMSDUs = ceil(length(txImage)/msduLength);
padZeros = msduLength-mod(length(txImage),msduLength);
txData = [txImage;zeros(padZeros,1)];
txDataBits = double(int2bit(txData,8,false));

% Divide input data stream into fragments
bitsPerOctet = 8;
data = zeros(0,1);

for i=0:numMSDUs-1

    % Extract image data (in octets) for each MPDU
    frameBody = txData(i*msduLength+1:msduLength*(i+1),:);

    % Create MAC frame configuration object and configure sequence number
    cfgMAC = wlanMACFrameConfig(FrameType='Data',SequenceNumber=i);

    % Generate MPDU
    [psdu, lengthMPDU]= wlanMACFrame(frameBody,cfgMAC,OutputFormat='bits');

    % Concatenate PSDUs for waveform generation
    data = [data; psdu]; %#ok<AGROW>

end
```

Generate 802.11a Baseband WLAN Signal

Synthesize a non-HT waveform using the `wlanWaveformGenerator` function with a non-HT format configuration object created by the `wlanNonHTConfig` object. In this example, the configuration object has a 20 MHz bandwidth, one transmit antenna and 64QAM rate 2/3 (MCS 6).

```
nonHTcfg = wlanNonHTConfig;       % Create packet configuration
nonHTcfg.MCS = 6;                 % Modulation: 64QAM Rate: 2/3
nonHTcfg.NumTransmitAntennas = 1; % Number of transmit antenna
chanBW = nonHTcfg.ChannelBandwidth;
nonHTcfg.PSDULength = lengthMPDU; % Set the PSDU length
```

Initialize the scrambler with a random integer for each packet.

```
scramblerInitialization = randi([1 127],numMSDUs,1);
```

Set the oversampling factor to `1.5` to generate the waveform at 30 MHz for transmission.

```
osf = 1.5 ;

sampleRate = wlanSampleRate(nonHTcfg); % Nominal sample rate in Hz

% Generate baseband NonHT packets separated by idle time
txWaveform = wlanWaveformGenerator(data,nonHTcfg, ...
    NumPackets=numMSDUs,IdleTime=20e-6, ...
    ScramblerInitialization=scramblerInitialization,...
    OversamplingFactor=osf);
```

Configure SDR for Transmission

If using an SDR, set the transmitter gain parameter (`txGain`) to reduce transmission quality and impair the received waveform.

Create an `sdrTransmitter` object using the `sdrtx` function. Set the center frequency, sample rate, and gain to the corresponding properties of the `sdrTransmitter` object. For an 802.11a signal on channel 5 in the 2.4 GHz frequency band, the corresponding center frequency is 2.432 GHz as defined in section 16.3.6.3 of the IEEE Std 802.11-2016.

The `sdrTransmitter` object uses the transmit repeat functionality to transmit the baseband WLAN waveform in a loop from the double data rate (DDR) memory on the SDR.

```
if strcmpi(channel,"OverTheAir")

    % Transmitter properties
    sdrTransmitter = sdrtx(deviceName);
    sdrTransmitter.BasebandSampleRate = sampleRate*osf;
    sdrTransmitter.CenterFrequency = wlanChannelFrequency(channelNumber,frequencyBand);
    sdrTransmitter.Gain = txGain;

    % Pass the SDR I/O directly to host skipping FPGA on Zynq Radio or USRP
    % Embedded Series Radio
    if ~strcmpi(deviceName,"Pluto")
        sdrTransmitter.ShowAdvancedProperties = true;
        sdrTransmitter.BypassUserLogic = true;
    end

    fprintf('\nGenerating WLAN transmit waveform:\n')

    % Scale the normalized signal to avoid saturation of RF stages
    powerScaleFactor = 0.8;
    txWaveform = txWaveform.*(1/max(abs(txWaveform))*powerScaleFactor);

    % Transmit RF waveform
    transmitRepeat(sdrTransmitter,txWaveform);
end
```

The `transmitRepeat` function transfers the baseband WLAN packets with idle time to the SDR, and stores the signal samples in hardware memory. The example then transmits the waveform continuously over the air until the release of the transmit object.

Receiver Design

The steps listed below describe the general structure of the WLAN receiver.

1. If using SDR hardware, capture multiple packets of the transmitted WLAN signal
2. Detect a packet
3. Coarse carrier frequency offset is estimated and corrected
4. Fine timing synchronization is established. The L-STF, L-LTF and L-SIG samples are provided for fine timing to allow to adjust the packet detection at the start or end of the L-STF
5. Fine carrier frequency offset is estimated and corrected
6. Perform a channel estimation for the received signal using the L-LTF
7. Detect the format of the packet

8. Decode the L-SIG field to recover the MCS value and the length of the data portion
9. Decode the data field to obtain the transmitted data within each packet
10. Decode the received PSDU and check if the frame check sequence (FCS) passed for the PSDU
11. Order the decoded MSDUs based on the `SequenceNumber` property in the recovered MAC frame configuration object
12. Combine the decoded MSDUs from all the transmitted packets to form the received image

This example plots the power spectral density (PSD) of the received waveform, and shows visualizations of the equalized data symbols and the received image.

Receiver Setup

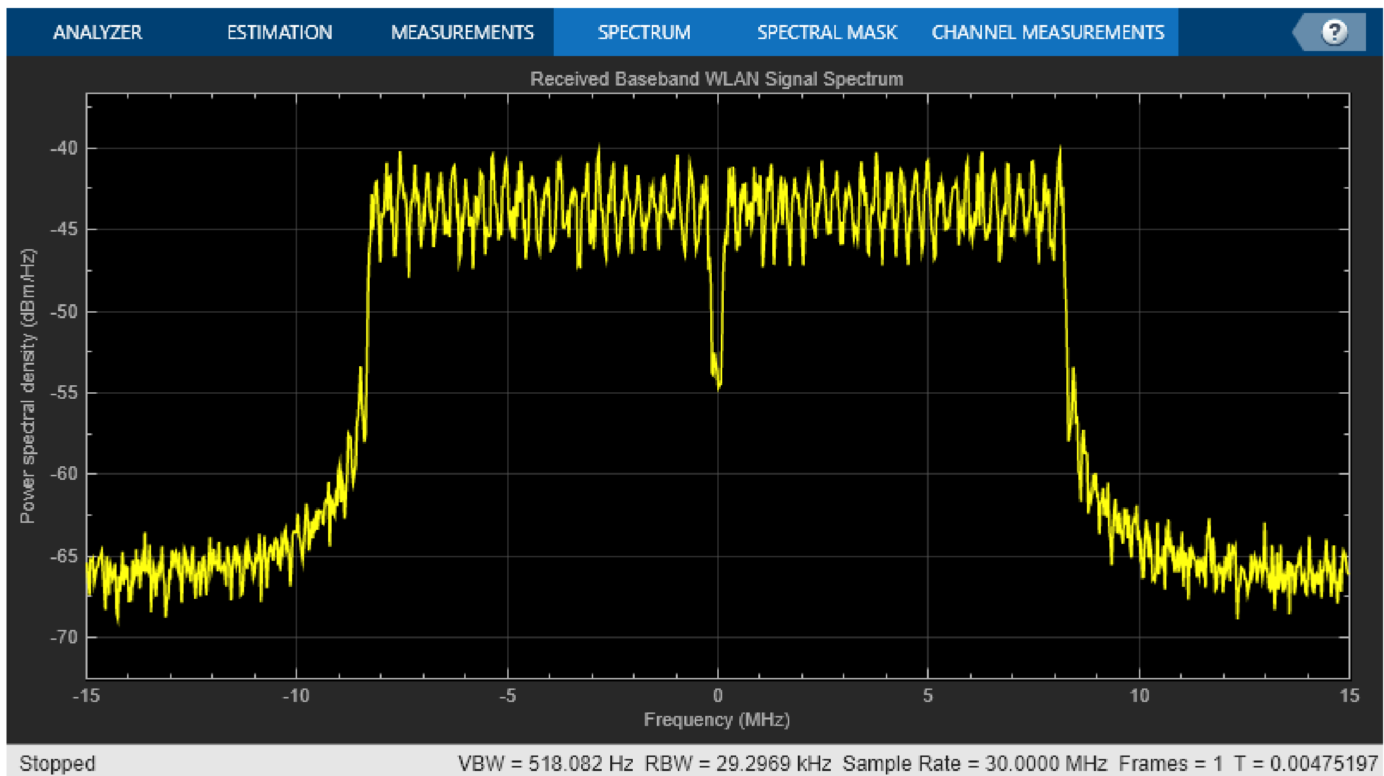
If using an SDR, create an `sdrReceiver` object using the `sdrrx` function. Set the center frequency, sample rate, and output data type to the corresponding properties of the `sdrReceiver` object.

Otherwise, apply gaussian noise to the `txWaveform` using the `awgn` function or pass the `txWaveform` straight through to receiver processing.

```
if strcmpi(channel,"OverTheAir")

    sdrReceiver = sdrrx(deviceName);
    sdrReceiver.BasebandSampleRate = sdrTransmitter.BasebandSampleRate;
    sdrReceiver.CenterFrequency = sdrTransmitter.CenterFrequency;
    sdrReceiver.OutputDataType = 'double';
    sdrReceiver.GainSource = 'Manual';
    sdrReceiver.Gain = rxGain;

    if ~strcmpi(deviceName,"Pluto")
        sdrReceiver.ShowAdvancedProperties = true;
        sdrReceiver.BypassUserLogic = true;
    end

    % Configure the capture length equivalent to twice the length of the
    % transmitted signal, this is to ensure that PSDUs are received in order.
    % On reception the duplicate MAC fragments are removed.
    sdrReceiver.SamplesPerFrame = 2*length(txWaveform);
    fprintf('\nStarting a new RF capture.\n')

    rxWaveform = capture(sdrReceiver,sdrReceiver.SamplesPerFrame,'Samples');
elseif strcmpi(channel,"GaussianNoise")
    rxWaveform = awgn(txWaveform,SNR,'measured');
else % No Impairments
    rxWaveform = txWaveform;
end
```

Show the power spectral density of the received waveform.

```
spectrumScope.SampleRate = sampleRate*osf;
spectrumScope(rxWaveform);
release(spectrumScope);
```



Receiver Processing

Design a rate conversion filter for resampling the waveform to the nominal baseband rate for receiver processing using the `designMultirateFIR` function.

```
aStop = 40; % Stopband attenuation
ofdmInfo = wlanNonHTOFDMInfo('NonHT-Data',nonHTcfg); % OFDM parameters
SCS = sampleRate/ofdmInfo.FFTLength; % Subcarrier spacing
txbw = max(abs(ofdmInfo.ActiveFrequencyIndices))*2*SCS; % Occupied bandwidth
[L,M] = rat(1/osf);
maxLM = max([L M]);
R = (sampleRate-txbw)/sampleRate;
TW = 2*R/maxLM; % Transition width
b = designMultirateFIR(L,M,TW,aStop);
```

Resample the oversampled waveform back to 20 MHz for processing using the `dsp.FIRRateConverter` System object and the filter designed above.

```
firrc = dsp.FIRRateConverter(L,M,b);
rxWaveform = firrc(rxWaveform);
```

If using an SDR, the SDR continuously transmits the 802.11 waveform over-the-air in a loop. The first packet received by the `sdrReceiver` may not be the first transmitted packet. This means that the packets may be decoded out of sequence. To enable the received packets to be recombined in the correct order, their sequence number must be determined. The `wlanMPDUDecode` function decodes the MPDU from the decoded PSDU bits of each packet and outputs the MSDU as well as the recovered MAC frame configuration object `wlanMACFrameConfig`. The `SequenceNumber` property in the recovered MAC frame configuration object can be used for ordering the MSDUs in the transmitted sequence.

To display each packet's decoded L-SIG contents, the EVM measurements, and sequence number, check the `displayFlag` box.

```
displayFlag = 
```

```
;
```

Set up required variables for receiver processing.

```
rxWaveformLen = size(rxWaveform,1);
searchOffset = 0; % Offset from start of the waveform in samples
```

Get the required field indices within a PSDU.

```
ind = wlanFieldIndices(nonHTcfg);
Ns = ind.LSIG(2)-ind.LSIG(1)+1; % Number of samples in an OFDM symbol

% Minimum packet length is 10 OFDM symbols
lstfLen = double(ind.LSTF(2)); % Number of samples in L-STF
minPktLen = lstfLen*5;
pktInd = 1;
fineTimingOffset = [];
packetSeq = [];
rxBit = [];

% Perform EVM calculation
evmCalculator = comm.EVM(AveragingDimensions=[1 2 3]);
evmCalculator.MaximumEVMOutputPort = true;
```

Use a `while` loop to process the received out-of-order packets.

```
while (searchOffset+minPktLen)<=rxWaveformLen
    % Packet detect
    pktOffset = wlanPacketDetect(rxWaveform,chanBW,searchOffset,0.5);

    % Adjust packet offset
    pktOffset = searchOffset+pktOffset;
    if isempty(pktOffset) || (pktOffset+double(ind.LSIG(2))>rxWaveformLen)
        if pktInd==1
            disp('** No packet detected **');
        end
        break;
    end

    % Extract non-HT fields and perform coarse frequency offset correction
    % to allow for reliable symbol timing
    nonHT = rxWaveform(pktOffset+(ind.LSTF(1):ind.LSIG(2)),:);
    coarseFreqOffset = wlanCoarseCFOEstimate(nonHT,chanBW);
    nonHT = frequencyOffset(nonHT,sampleRate,-coarseFreqOffset);

    % Symbol timing synchronization
    fineTimingOffset = wlanSymbolTimingEstimate(nonHT,chanBW);

    % Adjust packet offset
    pktOffset = pktOffset+fineTimingOffset;

    % Timing synchronization complete: Packet detected and synchronized
    % Extract the non-HT preamble field after synchronization and
    % perform frequency correction
```

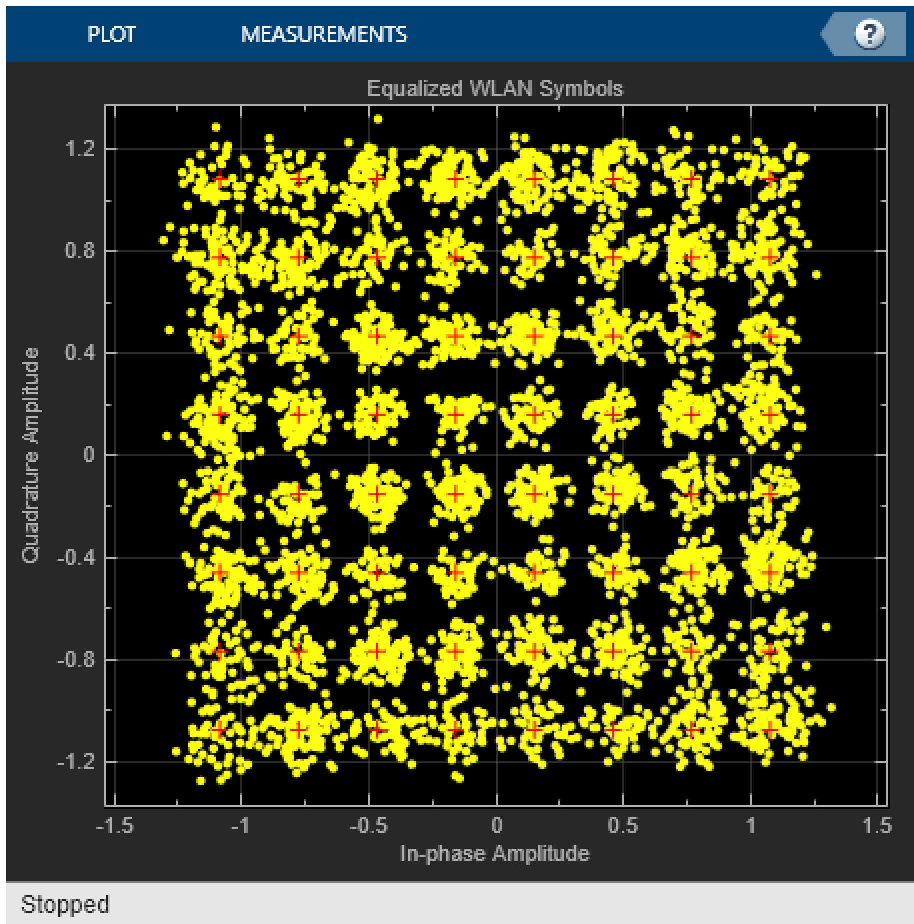
```
if (pktOffset<0) || ((pktOffset+minPktLen)>rxWaveformLen)
    searchOffset = pktOffset+1.5*lstfLen;
    continue;
end
fprintf('\nPacket-%d detected at index %d\n',pktInd,pktOffset+1);

% Extract first 7 OFDM symbols worth of data for format detection and
% L-SIG decoding
nonHT = rxWaveform(pktOffset+(1:7*Ns),:);
nonHT = frequencyOffset(nonHT,sampleRate,-coarseFreqOffset);

% Perform fine frequency offset correction on the synchronized and
% coarse corrected preamble fields
lltf = nonHT(ind.LLTF(1):ind.LLTF(2),:);           % Extract L-LTF
fineFreqOffset = wlanFineCFOEstimate(lltf,chanBW);
nonHT = frequencyOffset(nonHT,sampleRate,-fineFreqOffset);
cfoCorrection = coarseFreqOffset+fineFreqOffset; % Total CFO

% Channel estimation using L-LTF
lltf = nonHT(ind.LLTF(1):ind.LLTF(2),:);
demodLLTF = wlanLLTFDemodulate(lltf,chanBW);
chanEstLLTF = wlanLLTFChannelEstimate(demodLLTF,chanBW);

% Noise estimation
noiseVarNonHT = wlanLLTFNoiseEstimate(demodLLTF);

% Packet format detection using the 3 OFDM symbols immediately
% following the L-LTF
format = wlanFormatDetect(nonHT(ind.LLTF(2)+(1:3*Ns),:), ...
    chanEstLLTF,noiseVarNonHT,chanBW);
disp(['  ' format ' format detected']);
if ~strcmp(format,'Non-HT')
    fprintf('  A format other than Non-HT has been detected\n');
    searchOffset = pktOffset+1.5*lstfLen;
    continue;
end

% Recover L-SIG field bits
[recLSIGBits,failCheck] = wlanLSIGRecover( ...
    nonHT(ind.LSIG(1):ind.LSIG(2),:), ...
    chanEstLLTF,noiseVarNonHT,chanBW);

if failCheck
    fprintf('  L-SIG check fail \n');
    searchOffset = pktOffset+1.5*lstfLen;
    continue;
else
    fprintf('  L-SIG check pass \n');
end

% Retrieve packet parameters based on decoded L-SIG
[lsigMCS,lsigLen,rxSamples] = helperInterpretLSIG(recLSIGBits,sampleRate);

if (rxSamples+pktOffset)>length(rxWaveform)
    disp('** Not enough samples to decode packet **');
    break;
end
```

```
% Apply CFO correction to the entire packet
rxWaveform(pktOffset+(1:rxSamples),:) = frequencyOffset(...
    rxWaveform(pktOffset+(1:rxSamples),:),sampleRate,-cfoCorrection);

% Create a receive Non-HT config object
rxNonHTcfg = wlanNonHTConfig;
rxNonHTcfg.MCS = lsigMCS;
rxNonHTcfg.PSDULength = lsigLen;

% Get the data field indices within a PPDU
indNonHTData = wlanFieldIndices(rxNonHTcfg,'NonHT-Data');

% Recover PSDU bits using transmitted packet parameters and channel
% estimates from L-LTF
[rxPSDU,eqSym] = wlanNonHTDataRecover(rxWaveform(pktOffset+...
    (indNonHTData(1):indNonHTData(2)),:), ...
    chanEstLLTF,noiseVarNonHT,rxNonHTcfg);

constellation(reshape(eqSym,[],1)); % Current constellation
release(constellation);

refSym = wlanClosestReferenceSymbol(eqSym,rxNonHTcfg);
[evm.RMS,evm.Peak] = evmCalculator(refSym,eqSym);

% Decode the MPDU and extract MSDU
[cfgMACRx,msduList{pktInd},status] = wlanMPDUDecode(rxPSDU,rxNonHTcfg); %#ok<*SAGROW>

if strcmp(status,'Success')
    disp('  MAC FCS check pass');

    % Store sequencing information
    packetSeq(pktInd) = cfgMACRx.SequenceNumber;

    % Convert MSDU to a binary data stream
    rxBit{pktInd} = int2bit(hex2dec(cell2mat(msduList{pktInd})),8,false);

else % Decoding failed
    if strcmp(status,'FCSFailed')
        % FCS failed
        disp('  MAC FCS check fail');
    else
        % FCS passed but encountered other decoding failures
        disp('  MAC FCS check pass');
    end

    % Since there are no retransmissions modeled in this example, we
    % extract the image data (MSDU) and sequence number from the MPDU,
    % even though FCS check fails.

    % Remove header and FCS. Extract the MSDU.
    macHeaderBitsLength = 24*bitsPerOctet;
    fcsBitsLength = 4*bitsPerOctet;
    msduList{pktInd} = rxPSDU(macHeaderBitsLength+1:end-fcsBitsLength);

    % Extract and store sequence number
    sequenceNumStartIndex = 23*bitsPerOctet+1;
    sequenceNumEndIndex = 25*bitsPerOctet-4;
    conversionLength = sequenceNumEndIndex-sequenceNumStartIndex+1;
```

```
        packetSeq(pktInd) = bit2int(rxPSDU(sequenceNumStartIndex:sequenceNumEndIndex),conversionL

        % MSDU binary data stream
        rxBit{pktInd} = double(msduList{pktInd});
    end

    % Display decoded information
    if displayFlag
        fprintf('  Estimated CFO: %5.1f Hz\n\n',cfoCorrection); %#ok<*UNRCH>

        disp('  Decoded L-SIG contents: ');
        fprintf('                            MCS: %d\n',lsigMCS);
        fprintf('                         Length: %d\n',lsigLen);
        fprintf('    Number of samples in packet: %d\n\n',rxSamples);

        fprintf('  EVM:\n');
        fprintf('    EVM peak: %0.3f%%  EVM RMS: %0.3f%%\n\n', ...
            evm.Peak,evm.RMS);

        fprintf('  Decoded MAC Sequence Control field contents:\n');
        fprintf('    Sequence number: %d\n\n',packetSeq(pktInd));
    end

    % Update search index
    searchOffset = pktOffset+double(indNonHTData(2));

    % Finish processing when a duplicate packet is detected. The
    % recovered data includes bits from duplicate frame
    % Remove the data bits from the duplicate frame
    if length(unique(packetSeq)) < length(packetSeq)
        rxBit = rxBit(1:length(unique(packetSeq)));
        packetSeq = packetSeq(1:length(unique(packetSeq)));
        break
    end

    pktInd = pktInd+1;
end
```

```
Packet-1 detected at index 7

  Non-HT format detected

  L-SIG check pass

  MAC FCS check pass

Packet-2 detected at index 8647

  Non-HT format detected

  L-SIG check pass

  MAC FCS check pass

Packet-3 detected at index 17287

  Non-HT format detected

  L-SIG check pass
```

```
  MAC FCS check pass
Packet-4 detected at index 25927
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-5 detected at index 34567
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-6 detected at index 43207
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-7 detected at index 51847
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-8 detected at index 60487
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-9 detected at index 69127
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-10 detected at index 77767
  Non-HT format detected
  L-SIG check pass
  MAC FCS check pass
Packet-11 detected at index 86407
  Non-HT format detected
  L-SIG check pass
```




```
MAC FCS check pass
```

If using an SDR, release the `sdrTransmitter` and `sdrReceiver` objects to stop the continuous transmission of the 802.11 waveform and to allow for any modification of the SDR object properties.

```
if strcmpi(channel,"OverTheAir")
    release(sdrTransmitter);
    release(sdrReceiver);
end
```

Reconstruct Image

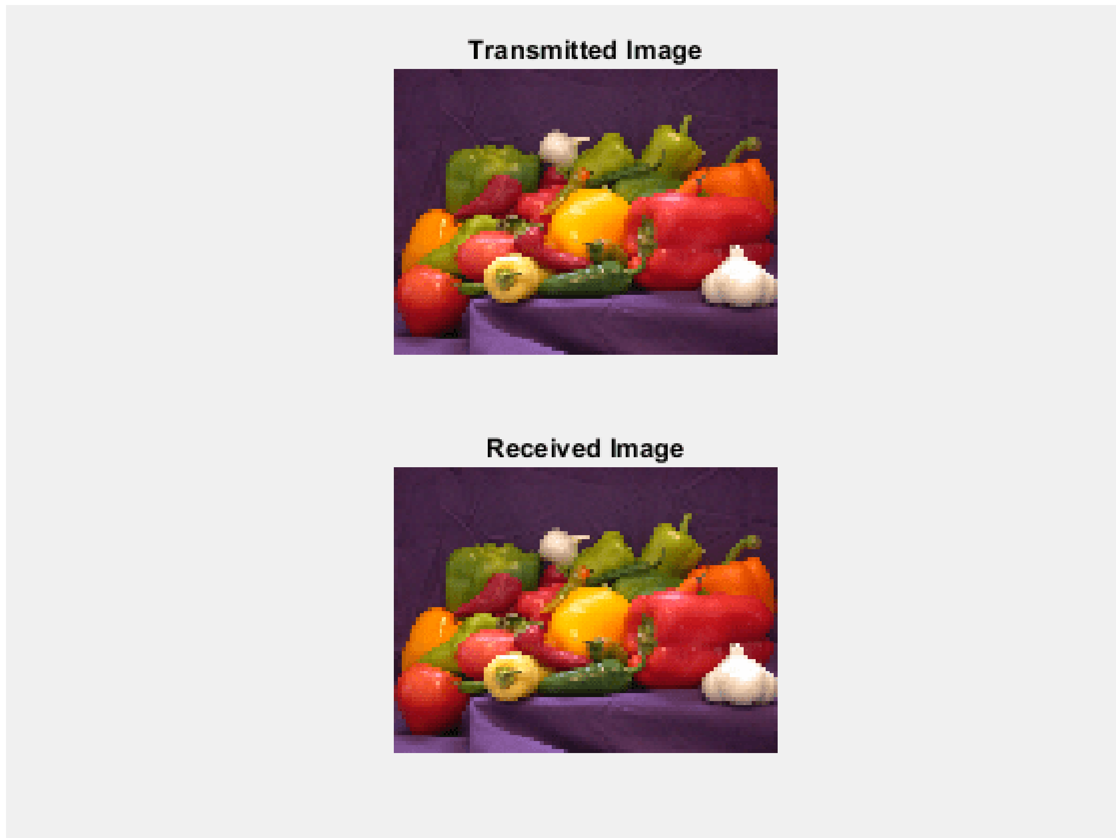
Reconstruct the image using the received MAC frames.

```
if ~(isempty(fineTimingOffset) || isempty(pktOffset))

    % Convert decoded bits from cell array to column vector
    rxData = cat(1,rxBit{:});
    % Remove any extra bits
    rxData = rxData(1:end-(mod(length(rxData),msduLength*8)));
    % Reshape such that each column length has bits equal to msduLength*8
    rxData = reshape(rxData,msduLength*8,[]);

    % Remove duplicate packets if any. Duplicate packets are located at the
    % end of rxData
```

```
    if length(packetSeq)>numMSDUs
        numDupPackets = size(rxData,2)-numMSDUs;
        rxData = rxData(:,1:end-numDupPackets);
    end

    % Initialize variables for while loop
    startSeq = [];
    i=-1;

    % Only execute this if one of the packet sequence values have been decoded
    % accurately
    if any(packetSeq<numMSDUs)
        while isempty(startSeq)
            % This searches for a known packetSeq value
            i = i + 1;
            startSeq = find(packetSeq==i);
        end
        % Circularly shift data so that received packets are in order for image reconstruction. I
        % is assumed that all packets following the starting packet are received in
        % order as this is how the image is transmitted.
        rxData = circshift(rxData,[0 -(startSeq(1)-i-1)]); % Order MAC fragments

        % Perform bit error rate (BER) calculation on reordered data
        bitErrorRate = comm.ErrorRate;
        err = bitErrorRate(double(rxData(:)), ...
            txDataBits(1:length(reshape(rxData,[],1))));
        fprintf('  \nBit Error Rate (BER):\n');
        fprintf('          Bit Error Rate (BER) = %0.5f\n',err(1));
        fprintf('          Number of bit errors = %d\n',err(2));
        fprintf('    Number of transmitted bits = %d\n\n',length(txDataBits));
    end

    decData = bit2int(reshape(rxData(:),8,[]),8,false)';

    % Append NaNs to fill any missing image data
    if length(decData)<length(txImage)
        numMissingData = length(txImage)-length(decData);
        decData = [decData;NaN(numMissingData,1)];
    else
        decData = decData(1:length(txImage));
    end

    % Recreate image from received data
    fprintf('\nConstructing image from received data.\n');
    receivedImage = uint8(reshape(decData,imsize));
    % Plot received image
    if exist('imFig','var') && ishandle(imFig) % If Tx figure is open
        figure(imFig); subplot(212);
    else
        figure; subplot(212);
    end
    imshow(receivedImage);
    title(sprintf('Received Image'));
end
```

```
Bit Error Rate (BER):
```

```
         Bit Error Rate (BER) = 0.00000
         Number of bit errors = 0
    Number of transmitted bits = 202752
Constructing image from received data.
```



Further Exploration

- If using an SDR, modify `txGain` on page 10-4 or `rxGain` on page 10-4 to observe the difference in the EVM and BER after signal reception and processing. You may see errors in the displayed, received image.
- If running the example without an SDR, modify `SNR` on page 10-9 to observe the difference in the EVM and BER after signal reception and processing.
- Increase the scaling factor (`scale` on page 10-5) to improve the quality of the received image by generating more transmit bits. This also increases the number of transmitted PPDUs.
- Decrease MSDU size (`msduLength` on page 10-6) to decrease the number of bits transmitted per packet and observe the differences in EVM and BER when a signal is received with a lower SNR.

SDR Troubleshooting

- ADALM-PLUTO Radio “Common Problems and Fixes” (Communications Toolbox Support Package for Analog Devices ADALM-Pluto Radio)
- USRP Embedded Series Radio “Common Problems and Fixes” (Communications Toolbox Support Package for USRP Embedded Series Radio)

- Xilinx Zynq-Based Radio “Common Problems and Fixes” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)

OFDM Wi-Fi Scanner Using SDR Preamble Detection

This example shows how to retrieve information about Wi-Fi® networks using a software-defined radio (SDR) and preamble detection. The example scans over the 5 GHz channels and uses an SDR preamble detector to detect and capture orthogonal frequency-division multiplexing (OFDM) packets from the air. The example then decodes the OFDM packets to determine which packets are access point (AP) beacons. The AP beacon information includes the service set identifier (SSID), media access control (MAC) address (also known as the basic SSID, or BSSID), AP channel bandwidth, and 802.11 standard used by the AP.

Introduction

This example scans through a set of Wi-Fi channels to detect AP beacons that are transmitted on 20 MHz subchannels. The scanning procedure uses a preamble detector on an NI™ USRP™ radio.

The scanning procedure comprises of these steps.

- Configure the `preambleDetector` (Wireless Testbench) object with a preamble that is generated from the legacy long training field (L-LTF).
- Set the frequency band and channels for the preamble detector to scan.
- Scan each specified channel and with each successful detection of an OFDM packet, capture a waveform for a set duration.
- Process the waveform in MATLAB® by searching for beacon frames in the captured waveform and extracting relevant information from each successfully decoded beacon frame.
- Display key information about the detected APs.

Set Up Radio

Call the `radioConfigurations` function. The function returns all available radio setup configurations that you saved using the Radio Setup wizard. For more information, see “Connect and Set Up NI USRP Radios” (Wireless Testbench).

```
savedRadioConfigurations = radioConfigurations;
```

To update the dropdown menu with your saved radio setup configuration names, click **Update**. Then select the radio to use with this example.

```
savedRadioConfigurationNames = [string({savedRadioConfigurations.Name})];
radio = MyRadio [Update];
```

Configure Preamble Detector

Create a preamble detector object with the specified radio. Because the object requires exclusive access to radio hardware resources, before running this example for the first time, clear any other object associated with the specified radio. In subsequent runs, to speed up the execution time of the example, reuse your new workspace object.

```
if ~exist("pd","var")
    pd = preambleDetector(radio);
end
```

To update the dropdown menu with the antennas available for capture on your radio, call the `hCaptureAntennas` helper function. Then select the antenna to use with this example.

```
captureAntennaSelection = hCaptureAntennas(radio);
pd.Antennas = RF0:RX2;
```

To increase the capture sample rate to 40 MHz, specify an oversampling factor of 2.

```
osf = 2;
pd.SampleRate = 20e6*osf;
pd.CaptureDataType = "double";
pd.ThresholdMethod = "adaptive";
```

Configure Preamble For Radio

The 802.11 standard requires that all Wi-Fi APs must transmit OFDM beacons using non-high throughput (non-HT) packets over a 20 MHz bandwidth. Therefore, generate a 20 MHz L-LTF waveform and use one long training symbol from the generated waveform as the preamble to detect WLAN OFDM packets.

```
cbw = "CBW20";
cfg = wlanNonHTConfig(ChannelBandwidth=cbw);
lltf = wlanLLTF(cfg,OversamplingFactor=osf);
```

Extract the first long training symbol from the L-LTF waveform.

```
cyclicPrefixLength = 1.6e-6*pd.SampleRate;
trainingSymbolLength = 3.2e-6*pd.SampleRate;
preamble = lltf(cyclicPrefixLength+1:cyclicPrefixLength+trainingSymbolLength);
```

Because the preamble detector requires the preamble to be between –1 and 1, normalize and set the preamble.

```
preamble = preamble/sqrt(sum(abs(preamble).^2));
pd.Preamble = preamble;
```

To capture the entire first non-HT packet, you must set the trigger offset to a negative value. Since you created a matched filter based on the long training symbol in the L-LTF waveform, the offset is at least one legacy short training field (L-STF), one L-LTF cyclic prefix, and one long training symbol.

```
lstfLength = 8e-6*pd.SampleRate;
pd.TriggerOffset = -(lstfLength + cyclicPrefixLength + trainingSymbolLength + 5);
```

Tune Preamble Detector

Configure the adaptive threshold gain, the adaptive threshold offset, and the radio gain values of the preamble detector for the local environment. Configuring these values requires manual tuning by exploring the trigger points provided by the `plotThreshold` function. For more information on tuning these values, see “Triggered Capture Using Preamble Detection” (Wireless Testbench).

For tuning the preamble detector, specify a channel in the 5 GHz band with a known OFDM packet.

In the 5 GHz band, the valid channel numbers are 1–200. However, the valid 20 MHz control channels for APs that use the 5 GHz band are 32, 36, 40, 44, 48, 52, 56, 60, 64, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140, 144, 149, 153, 157, 161, 165, 169, 173, 177.

```
band = 5;
channel = 100 ;
pd.CenterFrequency = wlanChannelFrequency(channel,band);
```

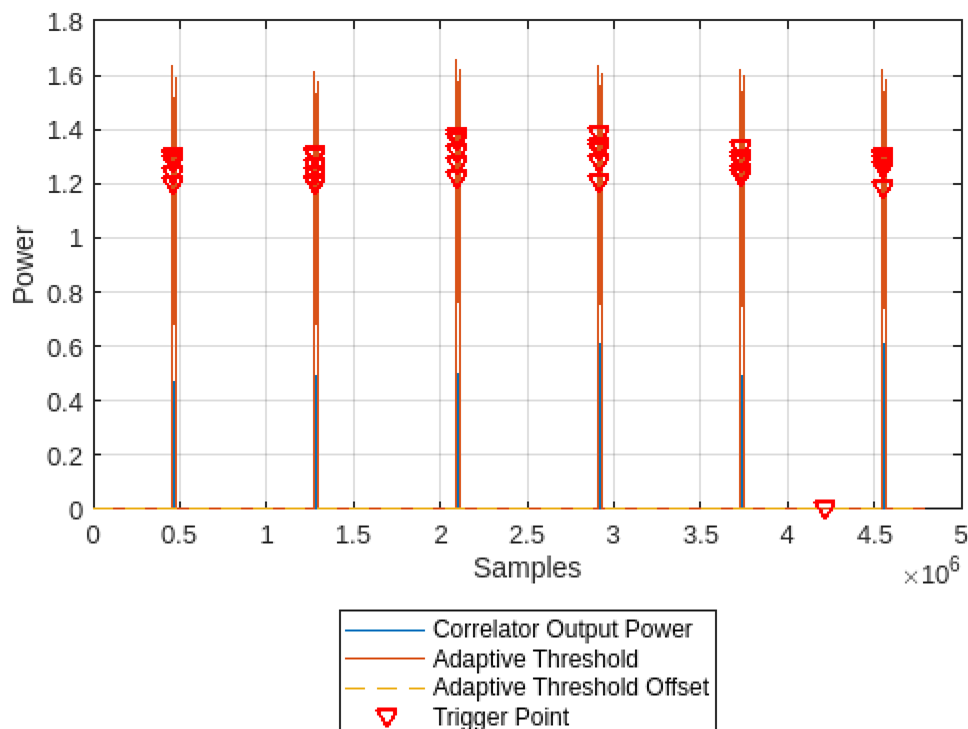
Adjust these values for tuning the preamble detector.

```
pd.AdaptiveThresholdGain = 0.3 ;
pd.AdaptiveThresholdOffset = 0.0005 ;
pd.RadioGain = 60 ;
```

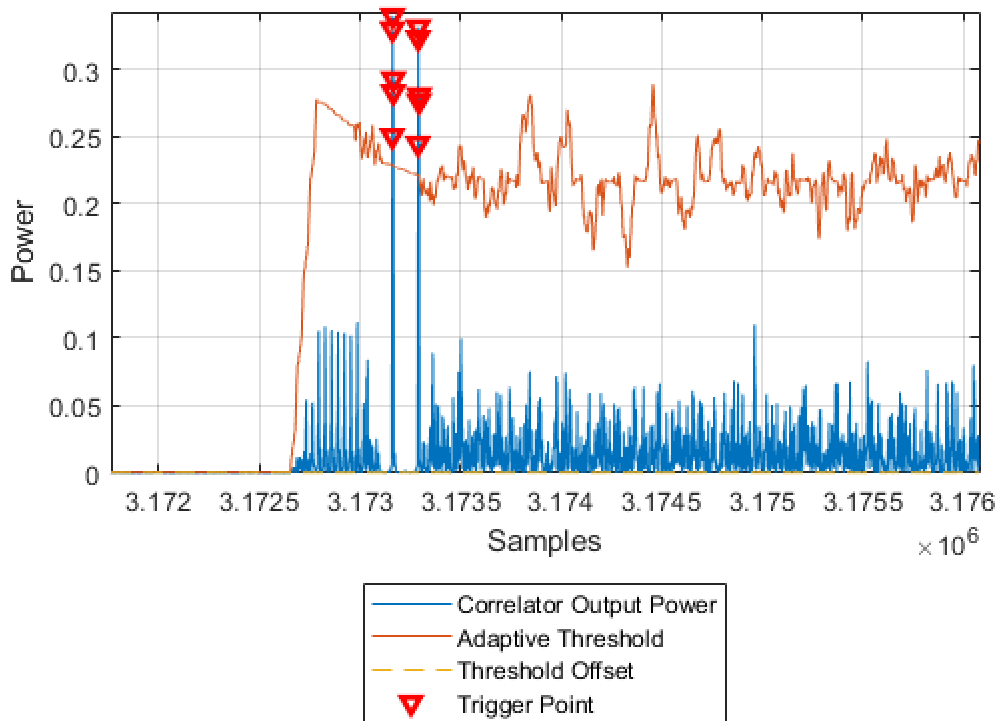
Plot the filter output power, adaptive threshold, and trigger points of the reconfigured preamble detector. The generated figure contains two trigger points for each OFDM packet. Each trigger point corresponds to a long training symbol.

When you generate a `plotThreshold` figure, if you do not have at least two trigger points for each OFDM packet, readjust the adaptive threshold gain, the adaptive threshold offset, and the radio gain until there are at least two trigger points per OFDM packet.

```
captureDuration = milliseconds(120);
plotThreshold(pd,captureDuration);
```



Inspect the trigger points by zooming in along the x-axis of the plot. For example, this figure shows a zoomed-in view of an OFDM packet with the trigger points on the correlation peaks.



Scan Wi-Fi Channels

Specify Scanning Region

Specify the channels in the 5 GHz band for the SDR to scan.

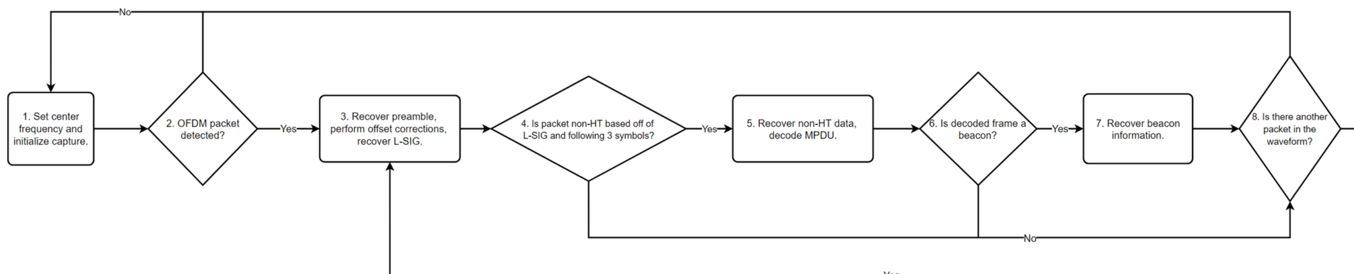
```
band = 5;
channels = [32 36 40 44 48 52 56 60 ( ;
```

Generate the center frequencies associated with the selected channels and band values.

```
centerFrequencies = wlanChannelFrequency(channels,band);
```

Receiver Design

This diagram shows an overview of the receiver for scanning the selected channels and frequency band.



These steps provide further information on the diagram.

1. Set the center frequency of the preamble detector, then initialize the detection and capture of a waveform for a set duration.
2. Check if the preamble detector detects an OFDM packet.
3. Determine and apply frequency and timing corrections on the waveform, then attempt to recover the legacy signal (L-SIG) field bits.
4. Check that the packet format is non-HT.
5. From the recovered L-SIG, extract the modulation and coding scheme (MCS) and the length of the PLCP service data unit (PSDU). Then recover the non-HT data and subsequently decode the MAC protocol data unit (MPDU).
6. Using the recovered MAC frame configuration, check if the non-HT packet is a beacon.
7. Recover the SSID, BSSID, vendor of the AP, SNR, primary 20 MHz channel, current channel center frequency index, supported channel width, frequency band, and wireless standard used by the AP.
8. Check if the waveform contains another packet that you can decode.

Initialize Variables

When you call the `capture` function to detect and capture a signal, you must specify the length of the capture and the signal detection timeout. Since beacons transmit every 100 ms, set `captureLength` to `milliseconds(100)` and `timeout` to `milliseconds(100)`.

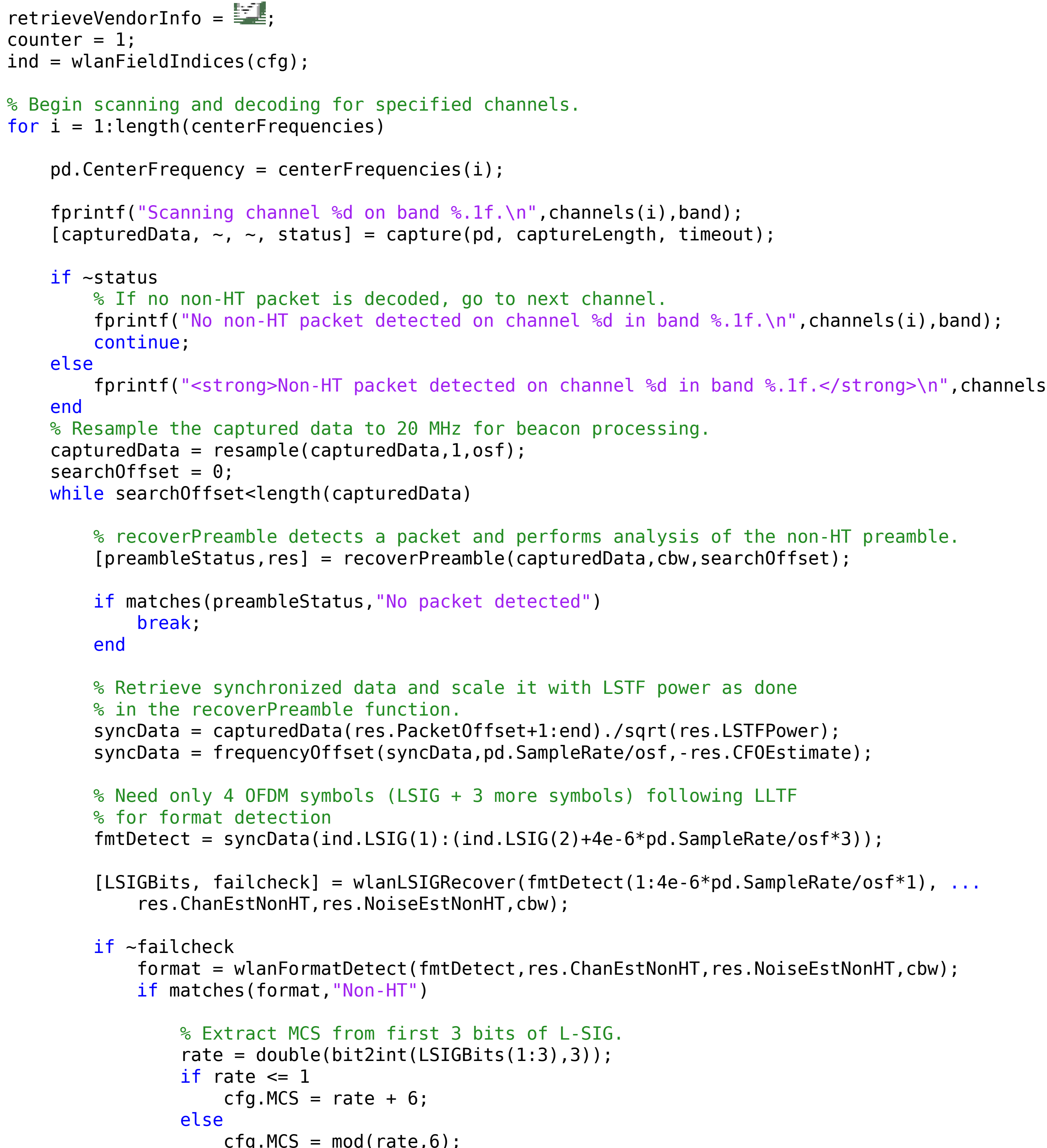
```
captureLength = milliseconds(100);
timeout = milliseconds(100);
```

Create a structure (`APs`) for storing this information for each successfully decoded beacon.

- SSID
- BSSID
- Vendor of AP
- Signal-to-noise ratio (SNR)
- Primary 20 MHz channel
- Current channel center frequency
- Channel width
- Frequency band
- Operating mode supported by the AP
- MAC frame configuration
- Waveform in which the beacon exists
- Index value at which the non-HT beacon packet begins in the captured waveform

```
APs = struct(...
    "SSID",[],"BSSID",[],"Vendor",[],"SNR_dB",[],"Beacon_Channel",[], ...
    "Operating_Channel",[],"Channel_Width_MHz",[],"Band",[],"Mode",[], ...
    "MAC_Config",wlanMACFrameConfig,"Waveform",[],"Offset",[]);
```

To determine the hardware manufacturer of the AP, select the `retrieveVendorInfo` box. Selecting the `retrieveVendorInfo` box downloads the organizationally unique identifier (OUI) CSV file from the IEEE® Registration Authority website for vendor AP identification.



```
retrieveVendorInfo = ;
counter = 1;
ind = wlanFieldIndices(cfg);

% Begin scanning and decoding for specified channels.
for i = 1:length(centerFrequencies)

    pd.CenterFrequency = centerFrequencies(i);

    fprintf("Scanning channel %d on band %.1f.\n",channels(i),band);
    [capturedData, ~, ~, status] = capture(pd, captureLength, timeout);

    if ~status
        % If no non-HT packet is decoded, go to next channel.
        fprintf("No non-HT packet detected on channel %d in band %.1f.\n",channels(i),band);
        continue;
    else
        fprintf("<strong>Non-HT packet detected on channel %d in band %.1f.</strong>\n",channels
    end
    % Resample the captured data to 20 MHz for beacon processing.
    capturedData = resample(capturedData,1,osf);
    searchOffset = 0;
    while searchOffset<length(capturedData)

        % recoverPreamble detects a packet and performs analysis of the non-HT preamble.
        [preambleStatus,res] = recoverPreamble(capturedData,cbw,searchOffset);

        if matches(preambleStatus,"No packet detected")
            break;
        end

        % Retrieve synchronized data and scale it with LSTF power as done
        % in the recoverPreamble function.
        syncData = capturedData(res.PacketOffset+1:end)./sqrt(res.LSTFPower);
        syncData = frequencyOffset(syncData,pd.SampleRate/osf,-res.CFOEstimate);

        % Need only 4 OFDM symbols (LSIG + 3 more symbols) following LLTF
        % for format detection
        fmtDetect = syncData(ind.LSIG(1):(ind.LSIG(2)+4e-6*pd.SampleRate/osf*3));

        [LSIGBits, failcheck] = wlanLSIGRecover(fmtDetect(1:4e-6*pd.SampleRate/osf*1), ...
            res.ChanEstNonHT,res.NoiseEstNonHT,cbw);

        if ~failcheck
            format = wlanFormatDetect(fmtDetect,res.ChanEstNonHT,res.NoiseEstNonHT,cbw);
            if matches(format,"Non-HT")

                % Extract MCS from first 3 bits of L-SIG.
                rate = double(bit2int(LSIGBits(1:3),3));
                if rate <= 1
                    cfg.MCS = rate + 6;
                else
                    cfg.MCS = mod(rate,6);
```

```
            end

            % Determine PSDU length from L-SIG.
            cfg.PSDULength = double(bit2int(LSIGBits(6:17),12,0));
            ind.NonHTData = wlanFieldIndices(cfg,"NonHT-Data");

            if double(ind.NonHTData(2)-ind.NonHTData(1))> ...
                    length(syncData(ind.NonHTData(1):end))
                % Exit while loop as full packet not captured.
                break;
            end

            nonHTData = syncData(ind.NonHTData(1):ind.NonHTData(2));
            bitsData = wlanNonHTDataRecover(nonHTData,res.ChanEstNonHT, ...
                res.NoiseEstNonHT,cfg);
            [cfgMAC, ~, decodeStatus] = wlanMPDUDecode(bitsData,cfg, ...
                SuppressWarnings=true);

            % Extract information about channel from the beacon.
            if ~decodeStatus && matches(cfgMAC.FrameType,"Beacon")
                fprintf("Beacon detected on channel %d in band %.1f.\n",channels(i),band);

                % Populate the table with information about the beacon.
                if isempty(cfgMAC.ManagementConfig.SSID)
                    APs(counter).SSID = "Hidden";
                else
                    APs(counter).SSID = string(cfgMAC.ManagementConfig.SSID);
                end

                APs(counter).BSSID = string(cfgMAC.Address3);
                if retrieveVendorInfo
                    APs(counter).Vendor = determineVendor(cfgMAC.Address3);
                else
                    APs(counter).Vendor = "Skipped"; %#ok<UNRCH>
                end
                [APs(counter).Mode, APs(counter).Channel_Width_MHz, operatingChannel] = ...
                    determineMode(cfgMAC.ManagementConfig.InformationElements);

                if isempty(operatingChannel)
                    % Default to scanning channel if operating channel
                    % cannot be determined.
                    operatingChannel = channels(i);
                end

                APs(counter).Beacon_Channel = channels(i);
                APs(counter).Operating_Channel = operatingChannel;
                APs(counter).SNR_dB = res.LLTFSNR;
                APs(counter).MAC_Config = cfgMAC;
                APs(counter).Offset = res.PacketOffset;
                APs(counter).Waveform = capturedData;
                counter = counter + 1;
            end
            % Shift packet search offset for next iteration of while loop.
            searchOffset = res.PacketOffset + double(ind.NonHTData(2));
        else
            % Packet is NOT non-HT; shift packet search offset by 10 OFDM symbols (minimum
            % packet length of non-HT) for next iteration of while loop.
            searchOffset = res.PacketOffset + 4e-6*pd.SampleRate/osf*10;
```

```
            end
        else
            % L-SIG recovery failed; shift packet search offset by 10 OFDM symbols (minimum
            % packet length of non-HT) for next iteration of while loop.
            searchOffset = res.PacketOffset + 4e-6*pd.SampleRate/osf*10;
        end
    end
end
```

```
Scanning channel 32 on band 5.0.

No non-HT packet detected on channel 32 in band 5.0.

Scanning channel 36 on band 5.0.

No non-HT packet detected on channel 36 in band 5.0.

Scanning channel 40 on band 5.0.

Non-HT packet detected on channel 40 in band 5.0.

Scanning channel 44 on band 5.0.

No non-HT packet detected on channel 44 in band 5.0.

Scanning channel 48 on band 5.0.

Non-HT packet detected on channel 48 in band 5.0.

Beacon detected on channel 48 in band 5.0.
Beacon detected on channel 48 in band 5.0.
Beacon detected on channel 48 in band 5.0.
Beacon detected on channel 48 in band 5.0.
Beacon detected on channel 48 in band 5.0.

Scanning channel 52 on band 5.0.

No non-HT packet detected on channel 52 in band 5.0.

Scanning channel 56 on band 5.0.

No non-HT packet detected on channel 56 in band 5.0.

Scanning channel 60 on band 5.0.

Non-HT packet detected on channel 60 in band 5.0.

Beacon detected on channel 60 in band 5.0.
Beacon detected on channel 60 in band 5.0.
Beacon detected on channel 60 in band 5.0.

Scanning channel 64 on band 5.0.

No non-HT packet detected on channel 64 in band 5.0.

Scanning channel 100 on band 5.0.

Non-HT packet detected on channel 100 in band 5.0.

Beacon detected on channel 100 in band 5.0.
Beacon detected on channel 100 in band 5.0.
Beacon detected on channel 100 in band 5.0.
```

```
Beacon detected on channel 100 in band 5.0.
Beacon detected on channel 100 in band 5.0.

Scanning channel 104 on band 5.0.

No non-HT packet detected on channel 104 in band 5.0.

Scanning channel 108 on band 5.0.

No non-HT packet detected on channel 108 in band 5.0.

Scanning channel 112 on band 5.0.

Non-HT packet detected on channel 112 in band 5.0.

Scanning channel 116 on band 5.0.

No non-HT packet detected on channel 116 in band 5.0.

Scanning channel 120 on band 5.0.

Non-HT packet detected on channel 120 in band 5.0.

Beacon detected on channel 120 in band 5.0.
Beacon detected on channel 120 in band 5.0.
Beacon detected on channel 120 in band 5.0.
Beacon detected on channel 120 in band 5.0.

Scanning channel 124 on band 5.0.

No non-HT packet detected on channel 124 in band 5.0.

Scanning channel 128 on band 5.0.

No non-HT packet detected on channel 128 in band 5.0.

Scanning channel 132 on band 5.0.

No non-HT packet detected on channel 132 in band 5.0.

Scanning channel 136 on band 5.0.

No non-HT packet detected on channel 136 in band 5.0.

Scanning channel 140 on band 5.0.

No non-HT packet detected on channel 140 in band 5.0.

Scanning channel 144 on band 5.0.

No non-HT packet detected on channel 144 in band 5.0.

Scanning channel 149 on band 5.0.

No non-HT packet detected on channel 149 in band 5.0.

Scanning channel 153 on band 5.0.

No non-HT packet detected on channel 153 in band 5.0.

Scanning channel 157 on band 5.0.

No non-HT packet detected on channel 157 in band 5.0.
```

```
Scanning channel 161 on band 5.0.

No non-HT packet detected on channel 161 in band 5.0.

Scanning channel 165 on band 5.0.

No non-HT packet detected on channel 165 in band 5.0.

Scanning channel 169 on band 5.0.

No non-HT packet detected on channel 169 in band 5.0.

Scanning channel 173 on band 5.0.

No non-HT packet detected on channel 173 in band 5.0.

Scanning channel 177 on band 5.0.

No non-HT packet detected on channel 177 in band 5.0.
```

Convert the `APs` structure to a table and display the information specified in step 7 on page 10-24 by using the local function `generateBeaconTable`.

```
detectedBeaconsInfo = generateBeaconTable(APs,band)
```

```
detectedBeaconsInfo=17×9 table
            SSID                   BSSID                     Vendor                 SNR (dB)    Prima
    _____________________    ______________    ____________________________    ________    ______

    "ClassForKids Secure"    "9A1898BEB142"    "Unknown"                        13.532
    "ClassForKids Guest"     "9E1898BEB142"    "Unknown"                        14.314
    "ClassForKids Music"     "921898BEB142"    "Unknown"                        13.211
    "Test SSID"              "961898BEB142"    "Unknown"                        13.671
    "Hidden"                 "A61898BEB142"    "Unknown"                        13.902
    "w-inside"               "B0B867F3D9B0"    "Hewlett Packard Enterprise"     12.655
    "w-mobile"               "B0B867F3D9B1"    "Hewlett Packard Enterprise"     13.278
    "w-guest"                "B0B867F3D9B2"    "Hewlett Packard Enterprise"     13.225
    "wlan1234_5"             "04D4C451C584"    "ASUSTek COMPUTER INC."          37.074
    "wlan1234_5"             "04D4C451C584"    "ASUSTek COMPUTER INC."          33.754
    "wlan1234_5"             "04D4C451C584"    "ASUSTek COMPUTER INC."          34.933
    "wlan1234_5"             "04D4C451C584"    "ASUSTek COMPUTER INC."          36.619
    "wlan1234_5"             "04D4C451C584"    "ASUSTek COMPUTER INC."          36.484
    "w-inside"               "B0B867F6B2D0"    "Hewlett Packard Enterprise"     29.415
    "w-mobile"               "B0B867F6B2D1"    "Hewlett Packard Enterprise"     29.295
    "w-guest"                "B0B867F6B2D2"    "Hewlett Packard Enterprise"     28.261
      ⋮
```

Further Exploration

- The `detectedBeaconsInfo` table shows only key information about the APs. To get further information about the beacons, such as data rates supported by the AP, explore the MAC frame configuration in the `APs` structure.
- If you have access to a configurable AP, change the channel width of your AP and rerun the example to confirm the channel width.

Local Functions

These functions assist in processing the incoming beacons.

```
function vendor = determineVendor(mac)
% DETERMINEVENDOR returns the vendor name of the AP by extracting the
% organizationally unique identifier (OUI) from the specified MAC address.

persistent ouis

vendor = strings(0);
try
    if isempty(ouis)
        if ~exist("oui.csv","file")
            disp("Downloading oui.csv from IEEE Registration Authority...")
            % Increase websave timeout if necessary
            options = weboptions("Timeout",5);
            websave("oui.csv","http://standards-oui.ieee.org/oui/oui.csv",options);
        end
        ouis = readtable("oui.csv",VariableNamingRule="preserve");
    end

    % Extract OUI from MAC Address.
    oui = mac(1:6);

    % Extract vendors name based on OUI.
    vendor = string(cell2mat(ouis.("Organization Name")(matches(ouis.Assignment,oui))));

catch ME
    % Rethrow caught error as warning.
    warning(ME.message+"\nTo skip the determineVendor function call, set retrieveVendorInfo to fa
end

if isempty(vendor)
    vendor = "Unknown";
end

end

function [mode,bw,operatingChannel] = determineMode(informationElements)
% DETERMINEMODE determines the 802.11 standard that the AP uses.
% The function checks for the presence of HT, VHT, and HE capability
% elements and determines the 802.11 standard that the AP uses. The element
% IDs are defined in IEEE Std 802.11-2020 and IEEE Std 802.11ax-2021.


elementIDs = cell2mat(informationElements(:,1));
IDs = elementIDs(:,1);

if any(IDs==255)
    if any(elementIDs(IDs==255,2)==35)
        % HE Packet Format
        mode = "802.11ax";
    else
        mode = "Unknown";
    end
    vhtElement = informationElements{IDs==192,2};
    htElement = informationElements{IDs==61,2};
    [bw,operatingChannel] = determineChannelWidth(htElement,vhtElement);
elseif any(IDs==191)
    % VHT Packet Format
    mode = "802.11ac";
```

```
        vhtElement = informationElements{IDs==192,2};
        htElement = informationElements{IDs==61,2};
        [bw,operatingChannel] = determineChannelWidth(htElement,vhtElement);
    elseif any(IDs==45)
        % HT Packet Format
        mode = "802.11n";
        htElement = informationElements{IDs==61,2};
        [bw,operatingChannel] = determineChannelWidth(htElement);
    else
        % Non-HT Packet Format
        % Exclude b as only DSSS is supported
        mode ="802.11a/g/j/p";
        bw = "Unknown";
        operatingChannel = [];
    end

    end

    function [bw,operatingChannel] = determineChannelWidth(htElement,varargin)
    % DETERMINECHANNELWIDTH returns the bandwidth of the channel from the
    % beacons HT/VHT operation information elements as defined in IEEE Std 802.11-2020
    % Section 9.4.2.56 and Section 9.4.2.158.

    msbFirst = false;

    % IEEE Std 802.11-2020 Figure 9-382 and Table 9-190 define each bit in
    % htOperationInfoBits
    % Convert to bits to get STA channel width value in 3rd bit.
    htOperationInfoBits = int2bit(htElement(2),5*8,msbFirst);
    operatingChannel = 0;

    if nargin == 2
        % IEEE Std 802.11-2020 Figure 9-163 and Table 9-274 define each octet
        % in vhtElement
        vhtElement = varargin{1};

        % VHT Operation Channel Width Field
        CW = vhtElement(1);
        % Channel Center Frequency Segment 0
        CCFS0 = vhtElement(2);
        % Channel Center Frequency Segment 1
        CCFS1 = vhtElement(3);

        % IEEE Std 802.11-2020 Table 11-23 defines the logic below
        if htOperationInfoBits(3) == 0
            bw = "20";
            operatingChannel = CCFS0;
        elseif CW == 0
            % HT Operation Channel Width Field is 1
            bw = "40";
            operatingChannel = CCFS0;
        elseif CCFS1 == 0
            % HT Operation Channel Width Field is 1 and
            % VHT Operation Channel Width Field is 1
            bw = "80";
            operatingChannel = CCFS0;
        elseif abs(CCFS1 - CCFS0) == 8
            % HT Operation Channel Width Field is 1 and
```

```
            % VHT Operation Channel Width Field is 1 and
            % CCFS1 is greater than 0
            bw = "160";
            operatingChannel = CCFS1;
        else
            % HT Operation Channel Width Field is 1 and
            % VHT Operation Channel Width Field is 1 and
            % CCFS1 is greater than 0 and
            % |CCFS1 - CCFS0| is greater than 16
            bw = "80+80";
        end
    end

    if operatingChannel == 0
        if htOperationInfoBits(3) == 1
            bw = "40";
            secondaryChannelOffset = bit2int(htOperationInfoBits(1:2),2,false);
            if secondaryChannelOffset == 1
                % Secondary Channel is above the primary channel.
                operatingChannel = htElement(1) + 2;
            elseif secondaryChannelOffset == 3
                % Secondary Channel is below the primary channel.
                operatingChannel = htElement(1) - 2;
            else
                warning("Could not determine operating channel.")
            end

        else
            bw = "20";
            operatingChannel = htElement(1);
        end
    end

    end

    function tbl = generateBeaconTable(APs,band)
    % GENERATEBEACONTABLE converts the access point structure to a table and
    % cleans up the variable names.

    tbl = struct2table(APs,"AsArray",true);
    tbl.Band = repmat(band,length(tbl.SSID),1);
    tbl = renamevars(tbl,["SNR_dB","Beacon_Channel","Operating_Channel","Channel_Width_MHz"], ...
        ["SNR (dB)","Primary 20 MHz Channel","Current Channel Center Frequency Index", ...
        "Channel Width (MHz)"]);
    tbl = tbl(:,1:9);

    end
```

OFDM Beacon Receiver Using Software-Defined Radio

This example shows how to retrieve information about WiFi networks on the 5 GHz band using a software-defined radio (SDR). The example scans over the 5 GHz beacon channels and captures waveforms for analysis within MATLAB®. The example then decodes the OFDM packets to determine which packets are access point (AP) beacons. The AP beacon information includes the service set identifier (SSID), media access control (MAC) address (also known as the basic SSID, or BSSID), AP channel bandwidth, and 802.11 standard used by the AP.

Introduction

This example scans through a set of WiFi channels in the 5 GHz band to detect AP beacons that are transmitted on 20 MHz subchannels.

The scanning procedure comprises these steps.

- Set the frequency band and channels for the SDR to capture.
- Capture a waveform for a set duration for each specified channel.
- Process the waveform in MATLAB by searching for beacon frames in the captured waveform and extracting relevant information from each successfully decoded beacon frame.
- Display key information about the detected APs.

This example supports these SDRs for capturing waveforms in the 5 GHz band.

- ADALM-Pluto from the Communications Toolbox Support Package for Analog Devices® ADALM-Pluto Radio
- USRP™ E310/E312 from the Communications Toolbox Support Package for USRP™ Embedded Series Radio
- AD936x/FMCOMMS5 from the Communications Toolbox Support Package for Xilinx® Zynq®-Based Radio
- USRP™ N200/N210/USRP2/N320/N321/B200/B210/X300/X310 from the Communications Toolbox Support Package for USRP™ Radio

Alternatively, if an SDR is not available for waveform capture, the example supports importing a file with a precaptured waveform.

Example Setup

Before running the example, ensure that you have installed the appropriate support package for the SDR that you intend to use and that you have set up the hardware.

The `ReceiveOnSDR` field of the `rxsim` structure determines whether the example receives a waveform off the air or imports a waveform from a MAT file.

```
rxsim.ReceiveOnSDR = 
```

```
;
```

Specify the file name of a precaptured waveform in the `fileName` variable. Confirm that the MAT file contains these variables: `capturedWaveforms`, `channels`, `radioSampleRate`, and `band`.

```
fileName = "capturedBeacons.mat";
```

By default, the example processes waveforms that are stored in a MAT file. If using an SDR to perform a live scan of the 5 GHz band, specify the SDR name, radio identifier, radio sample rate, channel numbers, and other parameters. The valid channel numbers for the 5 GHz band are between 1 and 200. However, the valid 20 MHz control channels for an AP are 32, 36, 40, 44, 48, 52, 56, 60, 64, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140, 144, 149, 153, 157, 161, 165, 169, 173, and 177.

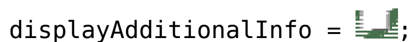
```
if rxsim.ReceiveOnSDR
    rxsim.SDRDeviceName = AD936x ;          % SDR that is used for waveform receptio
    rxsim.RadioIdentifier = 192.168.3.2 ;   % Value used to identify radio, for exam
    rxsim.RadioSampleRate = 20000000 ;      % Configured for 20e6 Hz as this is beac
    rxsim.RadioGain = 50 ;
    rxsim.FrequencyBand = 5;
    rxsim.ChannelNumbers = [32, 36, 40, 44, 48, 52, 56, ;   % Default scans all 5 GHz beacon channel
    rxsim.ReceiveAntenna = 1 ;              % Configure to work with only a single a
    rxsim.CaptureTime = milliseconds(100) ; % Value expected to be of type duration

    % Derived Parameters
    rxsim.CenterFrequencies = wlanChannelFrequency(rxsim.ChannelNumbers,rxsim.FrequencyBand);
    rxsim.NumSamplesToCapture = seconds(rxsim.CaptureTime)*rxsim.RadioSampleRate;

else
    rx = load(fileName);

    rxsim.ChannelNumbers = rx.channels;
    rxsim.RadioSampleRate = rx.radioSampleRate;
    rxsim.FrequencyBand = rx.band;

    % Derived Parameters
    rxsim.NumSamplesToCapture = size(rx.capturedWaveforms,1);
end
osf = rxsim.RadioSampleRate/20e6;
```

Set Optional Information to Display

To determine the hardware manufacturer of the AP, select the `retrieveVendorInfo` box. Selecting the `retrieveVendorInfo` box downloads the organizationally unique identifier (OUI) CSV file from the IEEE® Registration Authority website for vendor AP identification.

retrieveVendorInfo = ;

To display additional packet information such as payload size, code rate, and modulation for all successfully decoded non-HT packets, select the `displayAdditionalInfo` box.

displayAdditionalInfo = ;

To display a spectrum and spectrogram for the captured waveform, select the `displayScope` box.

displayScope = ;

Scan 5 GHz Channels

Initialize SDR object

This example communicates with the radio hardware using the object pertaining to the selected radio.

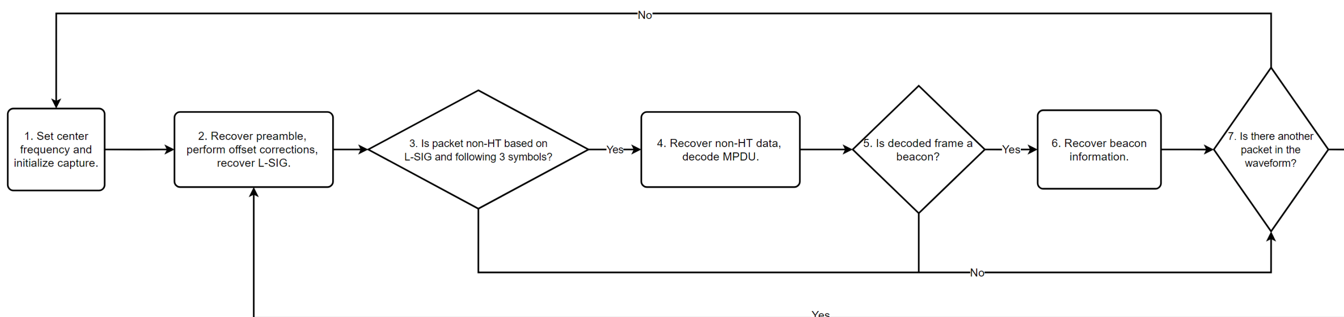
Create an SDR object by either calling `sdrrx` or `comm.SDRuReceiver`. Then, apply the parameters set in the `rxsim` structure above to the properties of that object.

```
if rxsim.ReceiveOnSDR
    if matches(rxsim.SDRDeviceName, ["AD936x", "FMCOMMS5", "Pluto", "E3xx"])
        sdrReceiver = sdrrx( ...
            rxsim.SDRDeviceName, ...
            BasebandSampleRate=rxsim.RadioSampleRate, ...
            GainSource="Manual");
        if matches(rxsim.SDRDeviceName, ["AD936x", "FMCOMMS5", "E3xx"])
            % When capturing a waveform, skip the FPGA and have the data
            % sent straight to the host
            sdrReceiver.ShowAdvancedProperties = true;
            sdrReceiver.BypassUserLogic = true;
            sdrReceiver.IPAddress = rxsim.RadioIdentifier;
        else
            sdrReceiver.RadioID = rxsim.RadioIdentifier;
        end
    else
        % For the USRP SDRs
        sdrReceiver = comm.SDRuReceiver( ...
            Platform=rxsim.SDRDeviceName, ...
            EnableBurstMode=true);
        % Determine burst capture values
        % Each frame will have 100 symbols
        sdrReceiver.SamplesPerFrame = 4e-6*rxsim.RadioSampleRate*100;
        sdrReceiver.NumFramesInBurst = rxsim.NumSamplesToCapture/sdrReceiver.SamplesPerFrame;
        [sdrReceiver.MasterClockRate, sdrReceiver.DecimationFactor] = ...
            hGetUSRPRateInformation(rxsim.SDRDeviceName,rxsim.RadioSampleRate);
        if matches(rxsim.SDRDeviceName, ["B200", "B210"])
            % Change the serial number as needed for USRP B200/B210
```

```
            sdrReceiver.SerialNum = rxsim.RadioIdentifier;
        else
            sdrReceiver.IPAddress = rxsim.RadioIdentifier;
        end
    end
    sdrReceiver.Gain = rxsim.RadioGain;
    sdrReceiver.OutputDataType = "double";
    sdrReceiver.ChannelMapping = rxsim.ReceiveAntenna;
end
```

Receiver Design

This diagram shows an overview of the receiver for scanning the selected channels and frequency band and recovering beacon information.



These steps provide further information on the diagram.

1. Set the center frequency of the SDR, then initialize the capture of a waveform for a set duration.
2. Determine and apply frequency and timing corrections on the waveform, then attempt to recover the legacy signal (L-SIG) field bits.
3. Check that the packet format is non-HT.
4. From the recovered L-SIG, extract the modulation and coding scheme (MCS) and the length of the PLCP service data unit (PSDU). Then recover the non-HT data and subsequently decode the MAC protocol data unit (MPDU).
5. Using the recovered MAC frame configuration, check if the non-HT packet is a beacon.
6. Recover the SSID, BSSID, vendor of the AP, SNR, primary 20 MHz channel, current channel center frequency index, supported channel width, frequency band, and wireless standard used by the AP.
7. Check if the waveform contains another packet that you can decode.

Begin Packet Capture and Processing

Create a structure (`APs`) for storing this information for each successfully decoded beacon.

- SSID
- BSSID
- Vendor of AP
- Signal-to-noise ratio (SNR)

- Primary 20 MHz channel
- Current channel center frequency index
- Channel width
- Frequency band
- Operating mode supported by the AP
- MAC frame configuration
- Waveform in which the beacon exists
- Index value at which the non-HT beacon packet begins in the captured waveform

```
APs = struct(...
    "SSID",[],"BSSID",[],"Vendor",[],"SNR_dB",[],"Beacon_Channel",[], ...
    "Operating_Channel",[],"Channel_Width_MHz",[],"Band",[],"Mode",[], ...
    "MAC_Config",wlanMACFrameConfig,"Waveform",[],"Offset",[]);
```

Initialize a non-HT config object to use for processing incoming waveforms.

```
cbw = "CBW20";
cfg = wlanNonHTConfig(ChannelBandwidth=cbw);
ind = wlanFieldIndices(cfg);
indexAP = 1;
```

Begin scanning and decoding for specified channels.

```
for i = 1:length(rxsim.ChannelNumbers)

    fprintf("<strong>Scanning channel %d on band %.1f.</strong>\n",rxsim.ChannelNumbers(i),rxsim
    if rxsim.ReceiveOnSDR
        sdrReceiver.CenterFrequency = rxsim.CenterFrequencies(i);
        capturedData = captureWaveform(sdrReceiver,rxsim.NumSamplesToCapture);
    else
        capturedData = rx.capturedWaveforms(:,i);
    end

    % Display spectrum and spectrogram
    if displayScope %#ok<*UNRCH>
        scope = spectrumAnalyzer(ViewType="spectrum-and-spectrogram",SampleRate=rxsim.RadioSample
            TimeSpanSource="property",TimeSpan=rxsim.NumSamplesToCapture/rxsim.RadioSampleRate);
        scope(capturedData);
    end

    % Resample the captured data to 20 MHz for beacon processing.
    if osf ~= 1
        capturedData = resample(capturedData,20e6,rxsim.RadioSampleRate);
    end

    searchOffset = 0;
    while searchOffset<length(capturedData)

        % recoverPreamble detects a packet and performs analysis of the non-HT preamble.
        [preambleStatus,res] = recoverPreamble(capturedData,cbw,searchOffset);

        if matches(preambleStatus,"No packet detected")
            break;
        end
```

```
% Retrieve synchronized data and scale it with LSTF power as done
% in the recoverPreamble function.
syncData = capturedData(res.PacketOffset+1:end)./sqrt(res.LSTFPower);
syncData = frequencyOffset(syncData,rxsim.RadioSampleRate/osf,-res.CFOEstimate);

% Need only 4 OFDM symbols (LSIG + 3 more symbols) following LLTF
% for format detection
fmtDetect = syncData(ind.LSIG(1):(ind.LSIG(2)+4e-6*rxsim.RadioSampleRate/osf*3));

[LSIGBits, failcheck] = wlanLSIGRecover(fmtDetect(1:4e-6*rxsim.RadioSampleRate/osf*1), .
    res.ChanEstNonHT,res.NoiseEstNonHT,cbw);

if ~failcheck
    format = wlanFormatDetect(fmtDetect,res.ChanEstNonHT,res.NoiseEstNonHT,cbw);
    if matches(format,"Non-HT")

        % Extract MCS from first 3 bits of L-SIG.
        rate = double(bit2int(LSIGBits(1:3),3));
        if rate <= 1
            cfg.MCS = rate + 6;
        else
            cfg.MCS = mod(rate,6);
        end

        % Determine PSDU length from L-SIG.
        cfg.PSDULength = double(bit2int(LSIGBits(6:17),12,0));
        ind.NonHTData = wlanFieldIndices(cfg,"NonHT-Data");

        if double(ind.NonHTData(2)-ind.NonHTData(1))> ...
                length(syncData(ind.NonHTData(1):end))
            % Exit while loop as full packet not captured.
            break;
        end

        nonHTData = syncData(ind.NonHTData(1):ind.NonHTData(2));
        bitsData = wlanNonHTDataRecover(nonHTData,res.ChanEstNonHT, ...
            res.NoiseEstNonHT,cfg);
        [cfgMAC, ~, decodeStatus] = wlanMPDUDecode(bitsData,cfg, ...
            SuppressWarnings=true);

        % Print additional information on all successfully packets
        if ~decodeStatus && displayAdditionalInfo
           payloadSize = floor(length(bitsData)/8);
           [modulation,coderate] = getRateInfo(cfg.MCS);

           fprintf("Payload Size: %d | Modulation: %s | Code Rate: %s \n",payloadSize,mod
           fprintf("Type: %s | Sub-Type: %s",cfgMAC.getType,cfgMAC.getSubtype);

        end

        % Extract information about channel from the beacon.
        if ~decodeStatus && matches(cfgMAC.FrameType,"Beacon")
            % Populate the table with information about the beacon.
            if isempty(cfgMAC.ManagementConfig.SSID)
                APs(indexAP).SSID = "Hidden";
            else
                APs(indexAP).SSID = string(cfgMAC.ManagementConfig.SSID);
            end
```

```
                    fprintf("<strong>%s beacon detected on channel %d in band %.1f.</strong>\n",A

                    APs(indexAP).BSSID = string(cfgMAC.Address3);
                    if retrieveVendorInfo
                        APs(indexAP).Vendor = determineVendor(cfgMAC.Address3);
                    else
                        APs(indexAP).Vendor = "Skipped";
                    end
                    [APs(indexAP).Mode, APs(indexAP).Channel_Width_MHz, operatingChannel] = ...
                        determineMode(cfgMAC.ManagementConfig.InformationElements);

                    if isempty(operatingChannel)
                        % Default to scanning channel if operating channel
                        % cannot be determined.
                        operatingChannel = rxsim.ChannelNumbers(i);
                    end

                    APs(indexAP).Beacon_Channel = rxsim.ChannelNumbers(i);
                    APs(indexAP).Operating_Channel = operatingChannel;
                    APs(indexAP).SNR_dB = res.LLTFSNR;
                    APs(indexAP).MAC_Config = cfgMAC;
                    APs(indexAP).Offset = res.PacketOffset;
                    APs(indexAP).Waveform = capturedData;
                    indexAP = indexAP + 1;
                end
                % Shift packet search offset for next iteration of while loop.
                searchOffset = res.PacketOffset + double(ind.NonHTData(2));
            else
                % Packet is NOT non-HT; shift packet search offset by 10 OFDM symbols (minimum
                % packet length of non-HT) for next iteration of while loop.
                searchOffset = res.PacketOffset + 4e-6*rxsim.RadioSampleRate/osf*10;
            end
        else
            % L-SIG recovery failed; shift packet search offset by 10 OFDM symbols (minimum
            % packet length of non-HT) for next iteration of while loop.
            searchOffset = res.PacketOffset + 4e-6*rxsim.RadioSampleRate/osf*10;
        end
    end
end
```

```
Scanning channel 52 on band 5.0.

WLAN_5G beacon detected on channel 52 in band 5.0.

Downloading oui.csv from IEEE Registration Authority...

Scanning channel 56 on band 5.0.

w-inside beacon detected on channel 56 in band 5.0.
w-mobile beacon detected on channel 56 in band 5.0.
w-guest beacon detected on channel 56 in band 5.0.

Scanning channel 157 on band 5.0.

w-inside beacon detected on channel 157 in band 5.0.
w-mobile beacon detected on channel 157 in band 5.0.
w-guest beacon detected on channel 157 in band 5.0.
```

Convert the `APs` structure to a table and display the information specified in step 6 on page 10-36 by using the local function `generateBeaconTable`.

```
detectedBeaconsInfo = generateBeaconTable(APs,rxsim.FrequencyBand,retrieveVendorInfo)
```

```
detectedBeaconsInfo=7×9 table
        SSID            BSSID                    Vendor                  SNR (dB)    Primary 20 MHz Cl
    ___________    _______________    ______________________________    ________    _________________

    "WLAN_5G"      "04D4C451C584"     "ASUSTek COMPUTER INC."             34.57                    52
    "w-inside"     "B0B867F6B2D0"     "Hewlett Packard Enterprise"        26.24                    56
    "w-mobile"     "B0B867F6B2D1"     "Hewlett Packard Enterprise"       26.251                    56
    "w-guest"      "B0B867F6B2D2"     "Hewlett Packard Enterprise"       25.843                    56
    "w-inside"     "B0B867F3D9B0"     "Hewlett Packard Enterprise"       31.592                   157
    "w-mobile"     "B0B867F3D9B1"     "Hewlett Packard Enterprise"       31.971                   157
    "w-guest"      "B0B867F3D9B2"     "Hewlett Packard Enterprise"         33.3                   157
```

```
if rxsim.ReceiveOnSDR
    release(sdrReceiver);
end
```

Further Exploration

- The `detectedBeaconsInfo` table shows only key information about the APs. To get further information about the beacons, such as data rates supported by the AP, explore the MAC frame configuration in the `APs` structure.
- If you have access to a configurable AP, change the channel width of your AP and rerun the example to confirm the channel width.

Local Functions

These functions assist in processing the incoming beacons.

```
function waveform = captureWaveform(sdrReceiver,numSamplesToCapture)
% CAPTUREWAVEFORM returns a column vector of complex values given an
% SDRRECEIVER object and a scalar NUMSAMPLESTOCAPTURE value.
    % For a comm.SDRuReceiver object, use the burst capture technique to
    % acquire the waveform
    if isa(sdrReceiver,'comm.SDRuReceiver')
        waveform = complex(zeros(numSamplesToCapture,1));
        samplesPerFrame = sdrReceiver.SamplesPerFrame;
        for i = 1:sdrReceiver.NumFramesInBurst
            waveform(samplesPerFrame*(i-1)+(1:samplesPerFrame)) = sdrReceiver();
        end
    else
        waveform = capture(sdrReceiver,numSamplesToCapture);
    end
end

function [modulation,coderate] = getRateInfo(mcs)
% GETRATEINFO returns the modulation scheme as a character array and the
% code rate of a packet given a scalar integer representing the modulation
% coding scheme
switch mcs
    case 0 % BPSK
        modulation = 'BPSK';
        coderate = '1/2';
```

```
    case 1 % BPSK
        modulation = 'BPSK';
        coderate = '3/4';
    case 2 % QPSK
        modulation = 'QPSK';
        coderate = '1/2';
    case 3 % QPSK
        modulation = 'QPSK';
        coderate = '3/4';
    case 4 % 16QAM
        modulation = '16QAM';
        coderate = '1/2';
    case 5 % 16QAM
        modulation = '16QAM';
        coderate = '3/4';
    case 6 % 64QAM
        modulation = '64QAM';
        coderate = '2/3';
    otherwise % 64QAM
        modulation = '64QAM';
        coderate = '3/4';
end
end

function vendor = determineVendor(mac)
% DETERMINEVENDOR returns the vendor name of the AP by extracting the
% organizationally unique identifier (OUI) from the specified MAC address.

persistent ouis

vendor = strings(0);
try
    if isempty(ouis)
        if ~exist("oui.csv","file")
            disp("Downloading oui.csv from IEEE Registration Authority...")
            options = weboptions("Timeout",10);
            websave("oui.csv","http://standards-oui.ieee.org/oui/oui.csv",options);

        end
        ouis = readtable("oui.csv",VariableNamingRule="preserve");
    end

    % Extract OUI from MAC Address.
    oui = mac(1:6);

    % Extract vendors name based on OUI.
    vendor = string(cell2mat(ouis.("Organization Name")(matches(ouis.Assignment,oui))));

catch ME
    % Rethrow caught error as warning.
    warning(ME.message+"\nTo skip the determineVendor function call, set retrieveVendorInfo to fa
end

if isempty(vendor)
    vendor = "Unknown";
end

end
```

```
function [mode,bw,operatingChannel] = determineMode(informationElements)
% DETERMINEMODE determines the 802.11 standard that the AP uses.
% The function checks for the presence of HT, VHT, and HE capability
% elements and determines the 802.11 standard that the AP uses. The element
% IDs are defined in IEEE Std 802.11-2020 and IEEE Std 802.11ax-2021.


elementIDs = cell2mat(informationElements(:,1));
IDs = elementIDs(:,1);

if any(IDs==255)
    if any(elementIDs(IDs==255,2)==35)
        % HE Packet Format
        mode = "802.11ax";
    else
        mode = "Unknown";
    end
    vhtElement = informationElements{IDs==192,2};
    htElement = informationElements{IDs==61,2};
    [bw,operatingChannel] = determineChannelWidth(htElement,vhtElement);
elseif any(IDs==191)
    % VHT Packet Format
    mode = "802.11ac";
    vhtElement = informationElements{IDs==192,2};
    htElement = informationElements{IDs==61,2};
    [bw,operatingChannel] = determineChannelWidth(htElement,vhtElement);
elseif any(IDs==45)
    % HT Packet Format
    mode = "802.11n";
    htElement = informationElements{IDs==61,2};
    [bw,operatingChannel] = determineChannelWidth(htElement);
else
    % Non-HT Packet Format
    % Exclude b as only DSSS is supported
    mode ="802.11a/g/j/p";
    bw = "Unknown";
    operatingChannel = [];
end

end

function [bw,operatingChannel] = determineChannelWidth(htElement,varargin)
% DETERMINECHANNELWIDTH returns the bandwidth of the channel from the
% beacons operation information elements as defined in IEEE Std 802.11-2020
% Table 11-23.

msbFirst = false;

% Convert to bits to get STA channel width value in 3rd bit.
htOperationInfoBits = int2bit(htElement(2),5*8,msbFirst);
operatingChannel = 0;

if nargin == 2
    vhtElement = varargin{1};

    % VHT Operation Channel Width Field
    CW = vhtElement(1);
```

```
    % Channel Center Frequency Segment 0
    CCFS0 = vhtElement(2);
    % Channel Center Frequency Segment 1
    CCFS1 = vhtElement(3);

    if htOperationInfoBits(3) == 0
        bw = "20";
        operatingChannel = CCFS0;
    elseif CW == 0
        % HT Operation Channel Width Field is 1
        bw = "40";
        operatingChannel = CCFS0;
    elseif CCFS1 == 0
        % HT Operation Channel Width Field is 1 and
        % VHT Operation Channel Width Field is 1
        bw = "80";
        operatingChannel = CCFS0;
    elseif abs(CCFS1 - CCFS0) == 8
        % HT Operation Channel Width Field is 1 and
        % VHT Operation Channel Width Field is 1 and
        % CCFS1 is greater than 0
        bw = "160";
        operatingChannel = CCFS1;
    else
        % HT Operation Channel Width Field is 1 and
        % VHT Operation Channel Width Field is 1 and
        % CCFS1 is greater than 0 and
        % |CCFS1 - CCFS0| is greater than 16
        bw = "80+80";
    end
end

if operatingChannel == 0
    if htOperationInfoBits(3) == 1
        bw = "40";
        secondaryChannelOffset = bit2int(htOperationInfoBits(1:2),2,false);
        if secondaryChannelOffset == 1
            % Secondary Channel is above the primary channel.
            operatingChannel = htElement(1) + 2;
        elseif secondaryChannelOffset == 3
            % Secondary Channel is below the primary channel.
            operatingChannel = htElement(1) - 2;
        else
            warning("Could not determine operating channel.")
        end

    else
        bw = "20";
        operatingChannel = htElement(1);
    end
end

end

function tbl = generateBeaconTable(APs,band,retrieveVendorInfo)
% GENERATEBEACONTABLE converts the access point structure to a table and
% cleans up the variable names.
```

```
tbl = struct2table(APs,"AsArray",true);
tbl.Band = repmat(band,length(tbl.SSID),1);
tbl = renamevars(tbl,["SNR_dB","Beacon_Channel","Operating_Channel","Channel_Width_MHz"], ...
    ["SNR (dB)","Primary 20 MHz Channel","Current Channel Center Frequency Index", ...
    "Channel Width (MHz)"]);
if retrieveVendorInfo
    tbl = tbl(:,1:9);
else
    tbl = tbl(:,[1:2,4:9]);
end

end
```