5G Toolbox™

Getting Started Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Contents

About 5G

5G Toolbox Product Description

Simulate, analyze, and test 5G communications systems

5G Toolbox provides standard-compliant functions and reference examples for the modeling, simulation, and verification of 5G New Radio (NR) communications systems. The toolbox supports link-level simulation, golden reference verification, conformance testing, and test waveform generation.

With the toolbox you can configure, simulate, measure, and analyze end-to-end 5G NR communications links. You can modify or customize the toolbox functions and use them as reference models for implementing 5G systems and devices.

The toolbox provides functions and reference examples to help you characterize uplink and downlink baseband specifications and simulate the effects of RF designs and interference sources on system performance. You can generate waveforms and customize test benches, either programmatically or interactively using the Wireless Waveform Generator app. With these waveforms, you can verify that your designs, prototypes, and implementations comply with the 3GPP 5G NR specifications.

What Is 5G New Radio?

New Radio (NR) is the air interface supporting the next generation of mobile communication, commonly referred to as fifth generation or 5G.

The predecessors of 5G NR are GSM, UMTS, and LTE, also referred to as second generation (2G), third generation (3G), and fourth generation (4G) technologies, respectively. GSM primarily enabled voice calls. The redesigned interfaces of UMTS and LTE enabled and gradually improved mobile broadband connectivity with high data rates and high efficiency.

5G NR continues on the path of LTE by enabling much higher data rates and much higher efficiency for mobile broadband. However, as a response to the demands of *networked society*, the scope of 5G NR goes beyond mobile broadband connectivity. The main requirement of 5G NR is to enable wireless connectivity everywhere, at any time to anyone and anything.

The wide range of use cases that drive 5G NR are classified by three main scenarios.

* *Enhanced mobile broadband* (eMBB) — This scenario is still the most important usage scenario that addresses human-centric communications. eMBB use cases have various challenges. For example, hot spots require higher data rates, higher user density, and a need for high capacity. Wide area coverage stresses mobility and seamless user experience with lower requirements on data rate and user density.
* *Massive machine type communications* (mMTC) — This scenario addresses pure machine-centric use cases characterized by a large number of connected devices. Typically, the data rate requirement of mMTC applications is low. However, the use cases demand a high connection density locally, low cost, and long battery life.
* *Ultra reliable and low latency communications* (URLLC) — This scenario covers both human-centric communication and critical machine-type communication (C-MTC) that demand low latency, reliability, and high availability. Typical URLLC use cases include 3-D gaming, self driving cars, mission-critical applications, remote medical surgery, and wireless control of industrial equipment.

This classification is based on presently foreseen use cases and identifies key capabilities of 5G NR. Based on these capabilities, the 5G NR interface is designed to easily adapt to unforeseen use cases that will evolve and emerge over time.

Scope of 5G Toolbox

The 5G NR specification is developed by the Third Generation Partnership Project (3GPP). The first release of the standard was frozen in mid-2018 as 3GPP 5G NR Release 15.

5G Toolbox provides implementations for a subset of the 5G NR physical layer specification and channel model specifications. The following diagram highlights the scope of 5G Toolbox in terms of the addressed specifications and their connectivity.
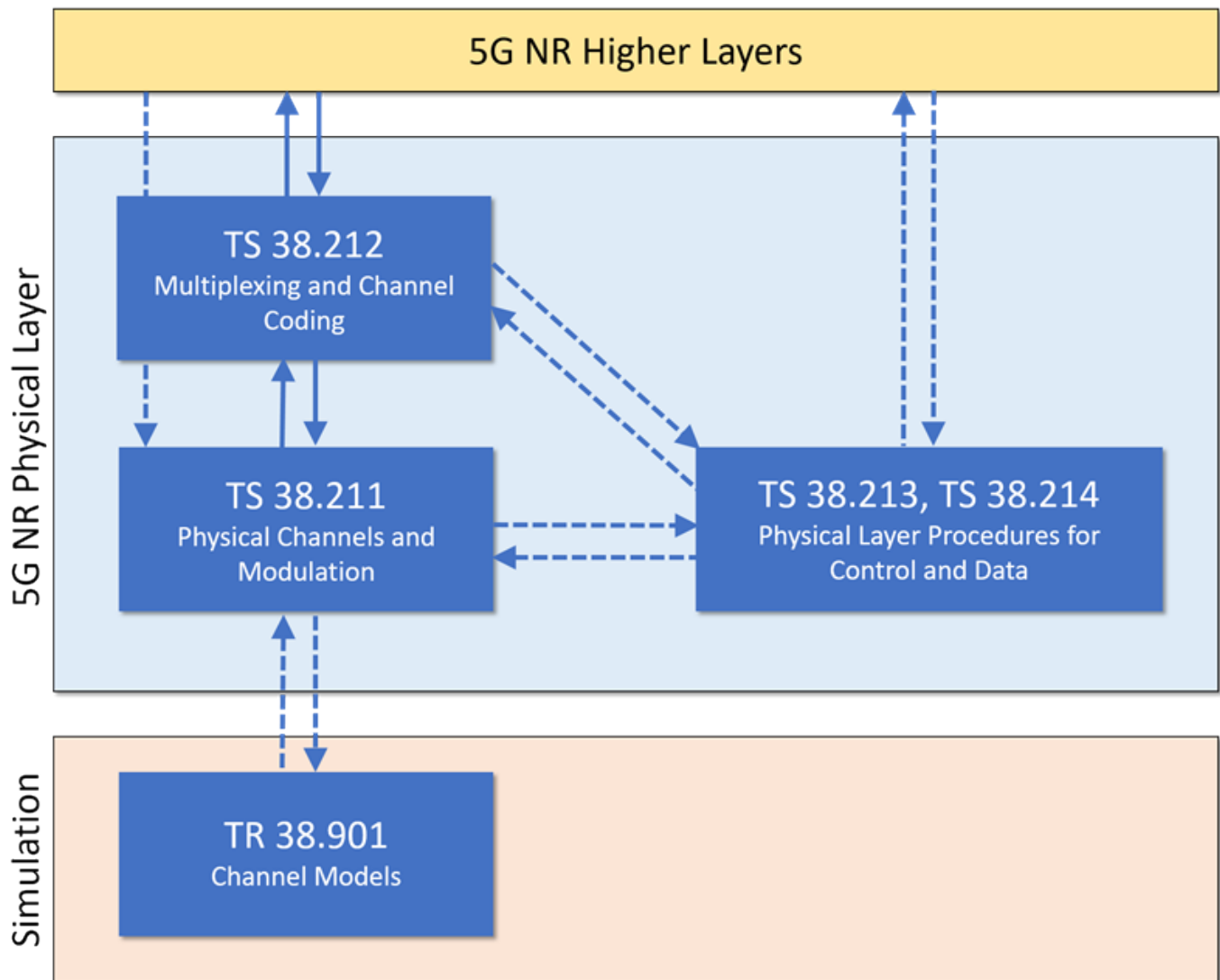
References

[1] Dalman, E., S. Parkvall, and J. Sköld. *4G, LTE-Advanced Pro and The Road to 5G*. Kidlington, Oxford: Academic Press, 2016.

See Also

More About

- "5G Toolbox and the 5G NR Protocol Layers" on page 1-5
- "What Is LTE?" (LTE Toolbox)

External Websites

- https://www.3gpp.org

5G Toolbox and the 5G NR Protocol Layers

The 5G NR radio access network is comprised of these protocol entities:

- Service data adaptation protocol (SDAP)
- Packet data convergence protocol (PDCP)
- Radio link control (RLC)
- Medium access control (MAC)
- Physical layer (PHY)

The SDAP protocol is new in 5G NR compared to the LTE protocol stack. SDAP handles the new QoS framework of the 5G System (in the 5G Core). SDAP applies also to LTE when connected to the 5G Core. The introduction of SDAP enables end-to-end QoS framework that works in both directions.

To meet the desired key capabilities of 5G NR, the other layers of the stack provide various enhancements over their LTE counterparts. The PDCP, RLC, and MAC protocols handle tasks such as header compression, ciphering, segmentation and concatenation, and multiplexing and demultiplexing. PHY handles coding and decoding, modulation and demodulation, and antenna mapping.

This figure shows the 5G NR user plane protocol stack for user equipment (UE) and the NR radio access network node (gNB). 5G Toolbox supports the 5G NR physical layer, including physical channels and signals. The toolbox also supports interfacing with portions of the RLC and MAC layers, including transport channels and logical channels.

QoS Flows

SDAP ↔ SDAP

Radio Bearers

PDCP ↔ PDCP

RLC Channels

RLC ↔ RLC

Logical Channels

MAC ↔ MAC

Transport Channels

PHY ↔ PHY

5G Toolbox™

UE   gNB
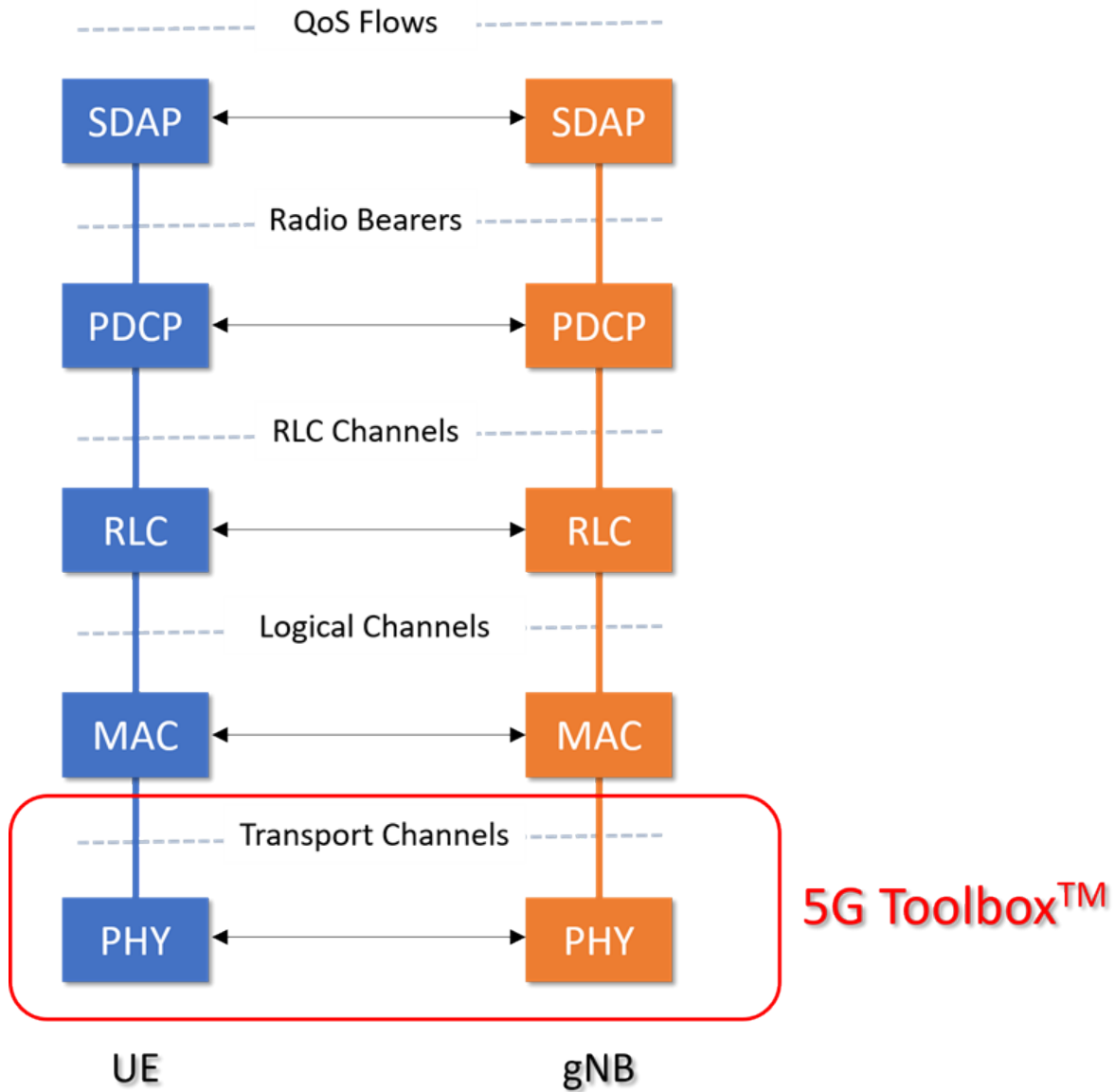
Downlink Channel Mapping

5G NR system downlink data follows the mapping between logical channels, transport channels, and physical channels, as indicated in the diagram. 5G Toolbox provides the red-highlighted downlink functionality for physical channels, transport channels, and control information.

5G Toolbox™
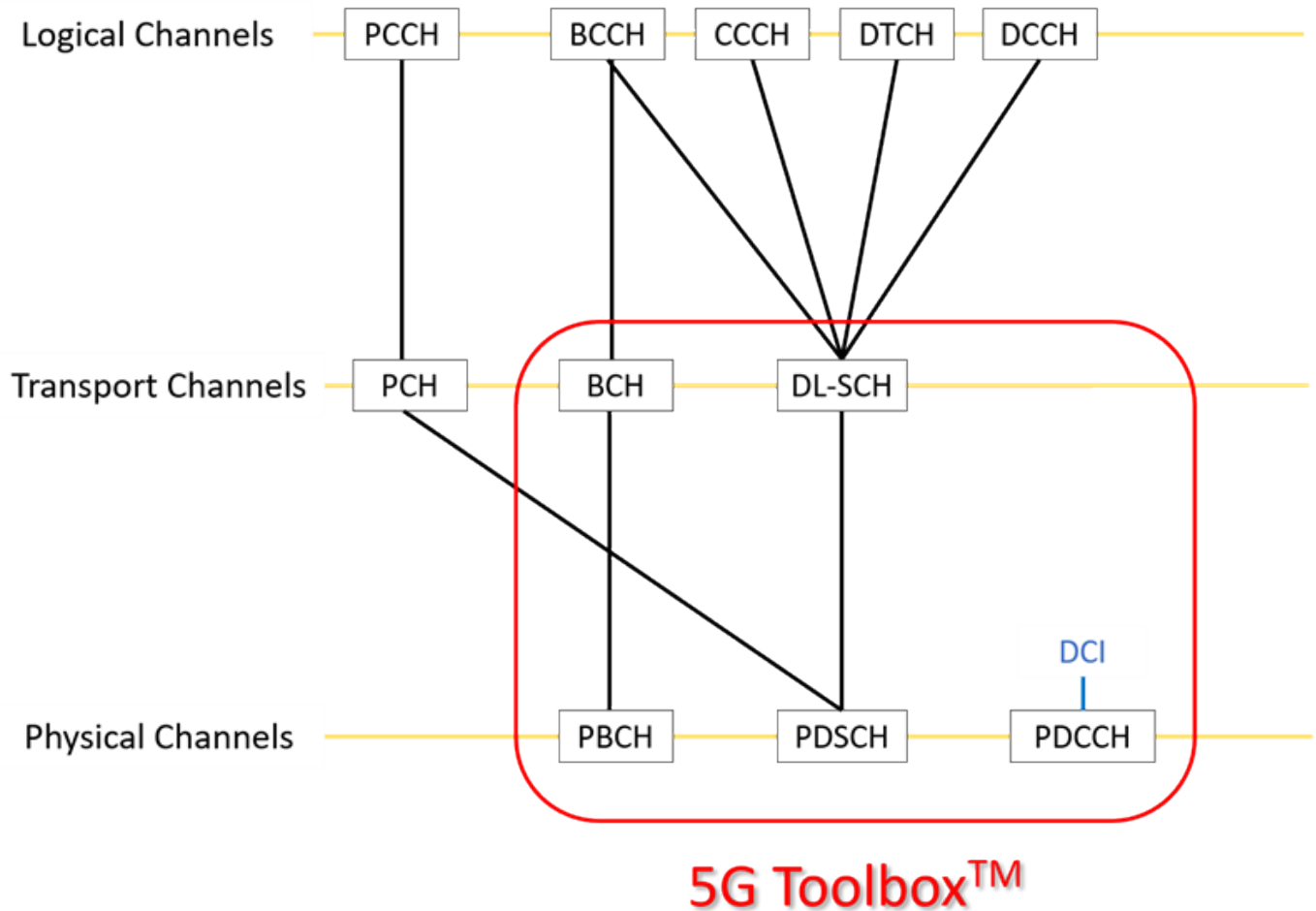
For more details, see "Downlink Channels" or the specific downlink channel:

- "Downlink Physical Signals"
- "Downlink Physical Channels"
- "Downlink Transport Channels"
- "Downlink Control Information"

Uplink Channel Mapping

5G NR system uplink data follows the mapping between logical channels, transport channels, and physical channels, as indicated in the diagram. 5G Toolbox provides the red-highlighted uplink functionality for physical channels, transport channels, and control information.
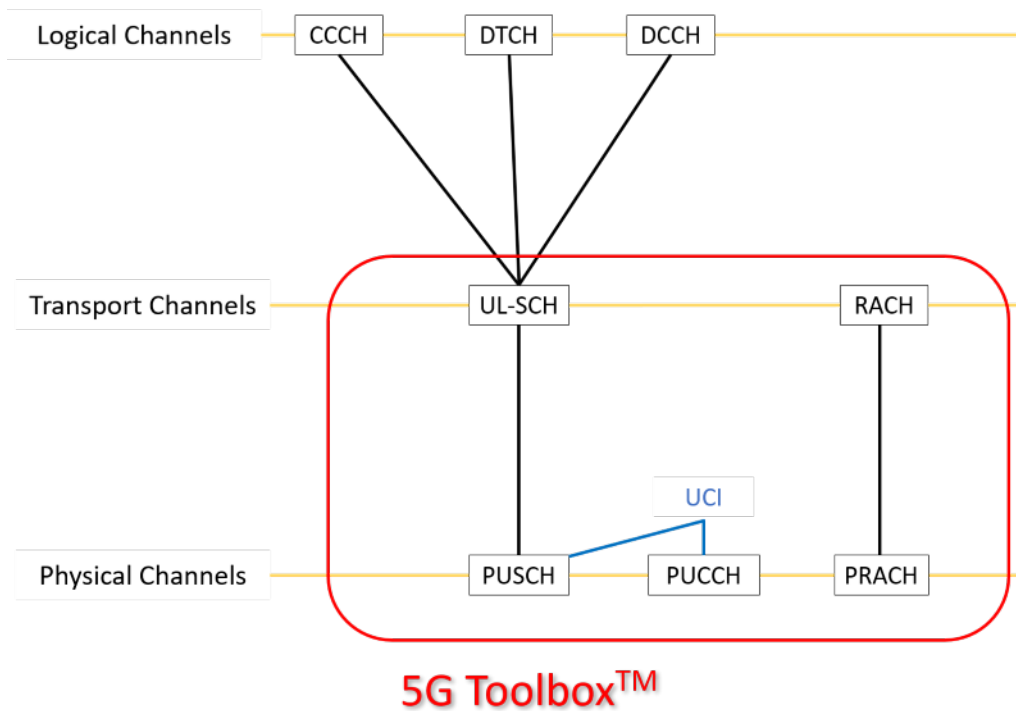
For more details, see "Uplink Channels" or the specific uplink channel:

- "Uplink Physical Signals"
- "Uplink Physical Channels"
- "Uplink Transport Channels"
- "Uplink Control Information"

References

[1] 3GPP TS 38.211. "NR; Physical channels and modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] 3GPP TS 38.212. "NR; Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[3] 3GPP TS 38.300. "NR; NR and NG-RAN Overall Description." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

See Also

More About

- "What Is 5G New Radio?" on page 1-3

External Websites

- https://www.3gpp.org

Tutorials

Synchronization Signal Blocks and Bursts

This example shows how to generate a synchronization signal block (SSB) and generate multiple SSBs to form a synchronization signal burst (SS burst). The channels and signals that form a synchronization signal block (primary and secondary synchronization signals, physical broadcast channel) are created and mapped into a matrix representing the block. Finally a matrix representing a synchronization signal burst is created, and each synchronization signal block in the burst is created and mapped into the matrix.

SS/PBCH block

TS 38.211 Section 7.4.3.1 defines the Synchronization Signal / Physical Broadcast Channel (SS/PBCH) block as 240 subcarriers and 4 OFDM symbols containing the following channels and signals:

*   Primary synchronization signal (PSS)
*   Secondary synchronization signal (SSS)
*   Physical broadcast channel (PBCH)
*   PBCH demodulation reference signal (PBCH DM-RS)

In other documents, for example TS 38.331, the SS/PBCH is termed "synchronization signal block" or "SS block".

Create a 240-by-4 matrix representing the SS/PBCH block:

```
ssblock = zeros([240 4])
```

ssblock = *240×4*

```
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     ⋮
```

Primary Synchronization Signal (PSS)

Create the PSS for a given cell identity:

```
ncellid = 17;
pssSymbols = nrPSS(ncellid)
```

pssSymbols = *127×1*

```
    -1
    -1
    -1
    -1
    -1
    -1
```

```
     1
     1
     1
    -1
     ⋮
```

The variable `pssSymbols` is a column vector containing the 127 BPSK symbols of the PSS.

Create the PSS indices:

```
pssIndices = nrPSSIndices;
```

The variable `pssIndices` is a column vector of the same size as `pssSymbols`. The value in each element of `pssIndices` is the linear index of the location in the SS/PBCH block to which the corresponding symbols in `pssSymbols` should be mapped. Therefore the mapping of the PSS symbols to the SS/PBCH block can be performed with a simple MATLAB assignment, using linear indexing to select the correct elements of the SS/PBCH block matrix. Note that a scaling factor of 1 is applied to the PSS symbols, to represent β_{PSS} in TS 38.211 Section 7.4.3.1.1:

```
ssblock(pssIndices) = 1 * pssSymbols;
```

Plot the SS/PBCH block matrix to show the location of the PSS:

```
imagesc(abs(ssblock));
caxis([0 4]);
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS/PBCH block containing PSS');
```

SS/PBCH block containing PSS
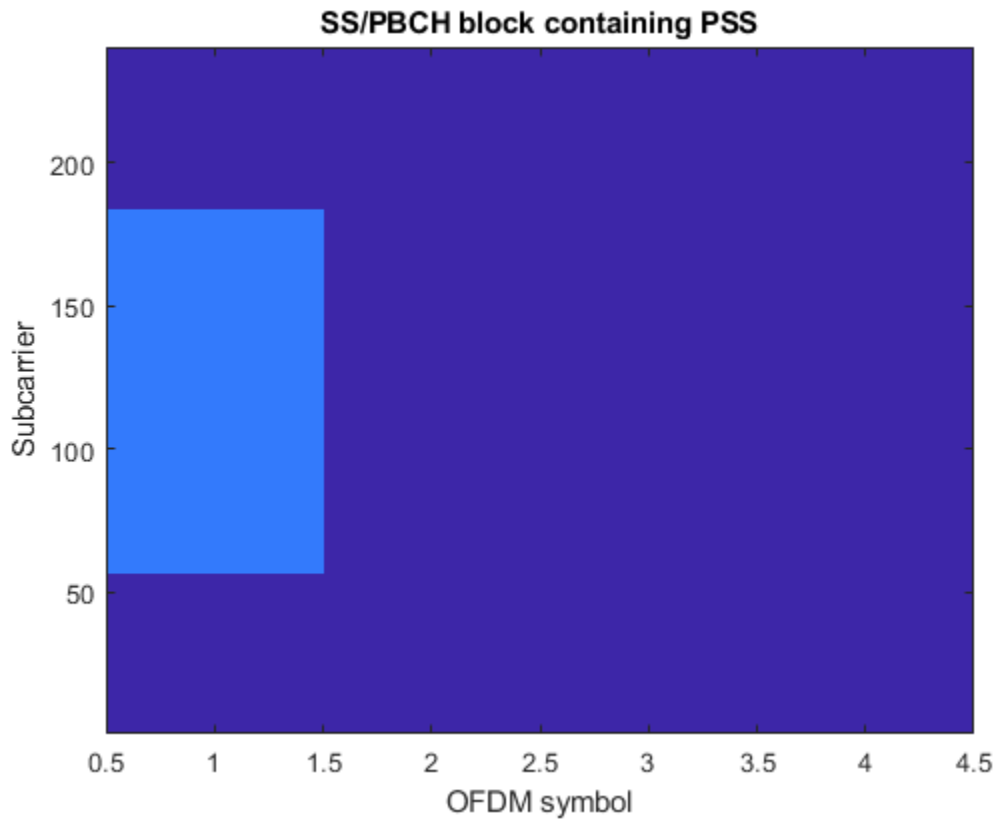
Secondary Synchronization Signal (SSS)

Create the SSS for the same cell identity as configured for the PSS:

```
sssSymbols = nrSSS(ncellid)
```

sssSymbols = *127×1*

```
   -1
    1
   -1
   -1
   -1
    1
   -1
    1
   -1
    1
    ⋮
```

Create the SSS indices and map the SSS symbols to the SS/PBCH block, following the same pattern used for the PSS. Note that a scaling factor of 2 is applied to the SSS symbols, to represent β_{SSS} in TS 38.211 Section 7.4.3.1.2:

```
sssIndices = nrSSSIndices;
ssblock(sssIndices) = 2 * sssSymbols;
```

The default form of the indices is 1-based linear indices, suitable for linear indexing of MATLAB matrices like `ssblock` as already shown. However, the NR standard documents describe the OFDM resources in terms of OFDM subcarrier and symbol subscripts, using 0-based numbering. For convenient cross-checking with the NR standard, the indices functions accept options to allow the indexing style (linear index versus subscript) and base (0-based versus 1-based) to be selected:

```
sssSubscripts = nrSSSIndices('IndexStyle','subscript','IndexBase','0based')
```

sssSubscripts = 127x3 uint32 matrix

```
    56     2     0
    57     2     0
    58     2     0
    59     2     0
    60     2     0
    61     2     0
    62     2     0
    63     2     0
    64     2     0
    65     2     0
     ⋮
```

It can be seen from the subscripts that the SSS is located in OFDM symbol 2 (0-based) of the SS/PBCH block, starting at subcarrier 56 (0-based).

Plot the SS/PBCH block matrix again to show the locations of the PSS and SSS:

```
imagesc(abs(ssblock));
caxis([0 4]);
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS/PBCH block containing PSS and SSS');
```

SS/PBCH block containing PSS and SSS
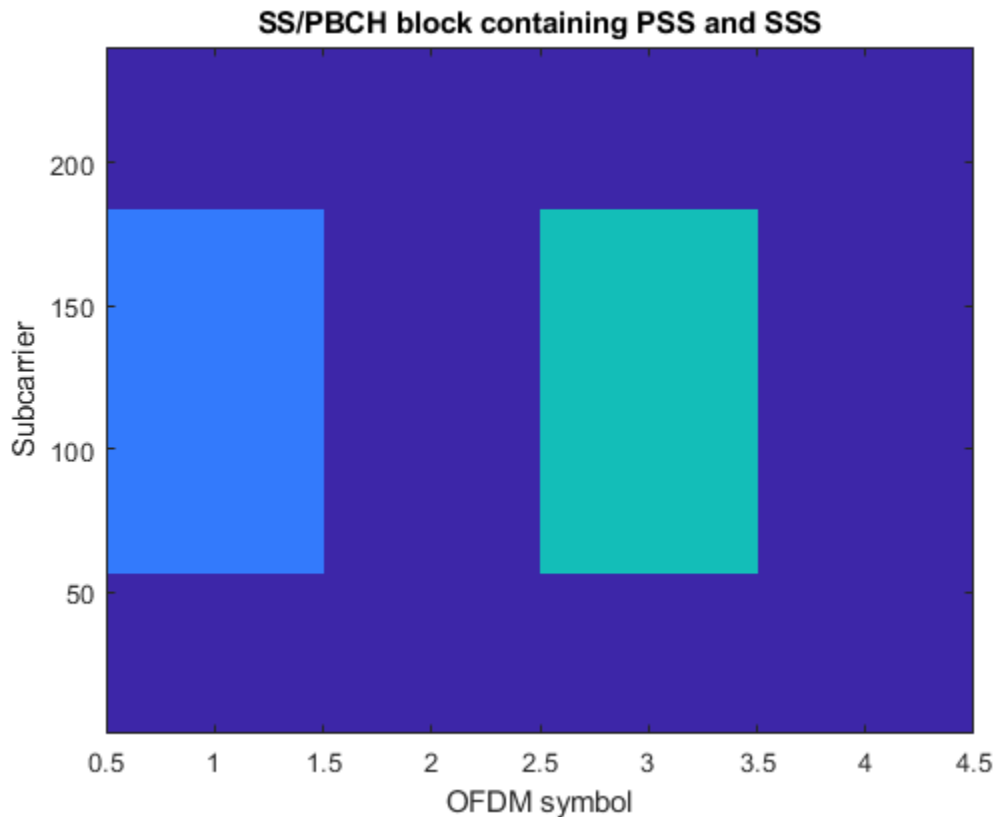
Physical Broadcast Channel (PBCH)

The PBCH carries a codeword of length 864 bits, created by performing BCH encoding of the master information block (MIB). For more information on BCH coding, see the functions `nrBCH` and `nrBCHDecode` and their use in the "NR Cell Search and MIB and SIB1 Recovery" example. Here a PBCH codeword consisting of 864 random bits is used:

```
cw = randi([0 1],864,1);
```

The PBCH modulation consists of the following steps as described in TS 38.211 Section 7.3.3:

- Scrambling
- Modulation
- Mapping to physical resources

Scrambling and modulation

Multiple SS/PBCH blocks are transmitted across half a frame, as described in the cell search procedure in TS 38.213 Section 4.1. Each SS/PBCH block is given an index from $0...L-1$, where L is the number SS/PBCH blocks in the half frame. The scrambling sequence for the PBCH is initialized according to the cell identity `ncellid`, and the subsequence used to scramble the PBCH codeword depends on the value v, 2 or 3 LSBs of SS/PBCH block index, as described in TS 38.211 Section 7.3.3.1. In this example, $v = 0$ is used. The function `nrPBCH` creates the appropriate subsequence of the scrambling sequence, performs scrambling and then performs QPSK modulation:

```
v = 0;
pbchSymbols = nrPBCH(cw,ncellid,v)

pbchSymbols = 432×1 complex

  -0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 + 0.7071i
   0.7071 - 0.7071i
   0.7071 + 0.7071i
   0.7071 + 0.7071i
      ⋮
```

Mapping to resource elements

Create the PBCH indices and map the PBCH symbols to the SS/PBCH block. Note that a scaling factor of 3 is applied to the PBCH symbols, to represent β_{PBCH} in TS 38.211 Section 7.4.3.1.3:

```
pbchIndices = nrPBCHIndices(ncellid);
ssblock(pbchIndices) = 3 * pbchSymbols;
```

Plot the SS/PBCH block matrix again to show the locations of the PSS, SSS and PBCH:

```
imagesc(abs(ssblock));
caxis([0 4]);
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS/PBCH block containing PSS, SSS and PBCH');
```

SS/PBCH block containing PSS, SSS and PBCH
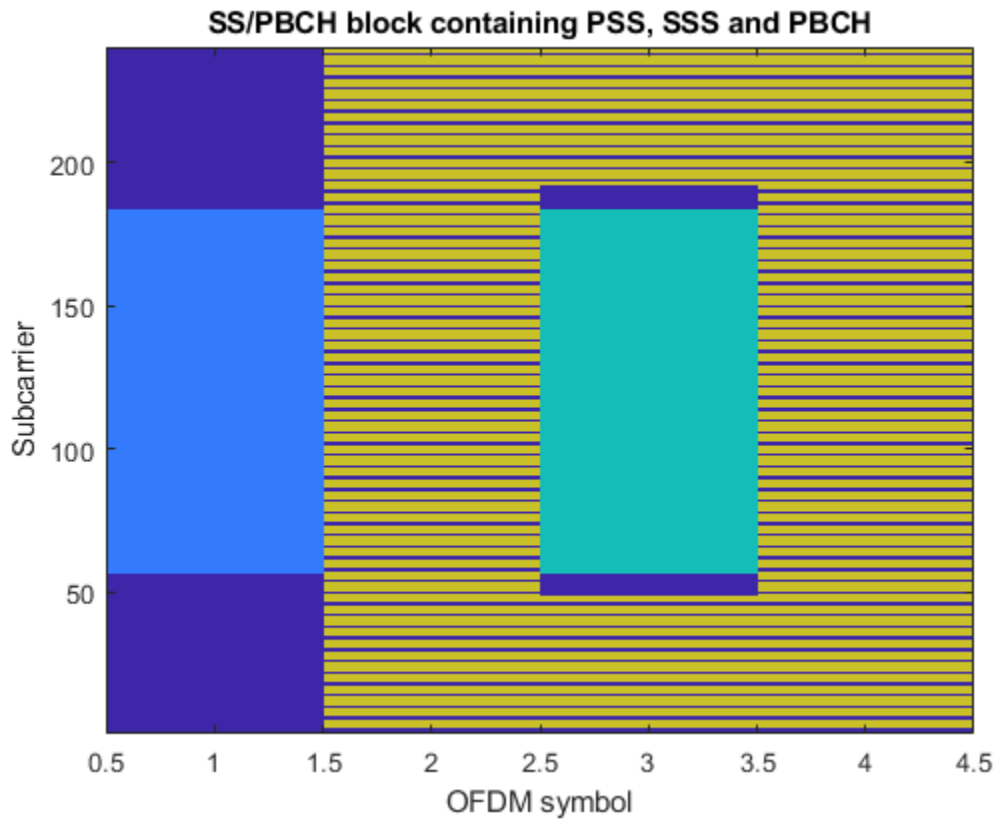
PBCH Demodulation Reference Signal (PBCH DM-RS)

The final component of the SS/PBCH block is the DM-RS associated with the PBCH. Similar to the PBCH, the DM-RS sequence used derives from the SS/PBCH block index and is configured using the variable $\bar{i}_{SSB}$ described in TS 38.211 Section 7.4.1.4.1. Here $\bar{i}_{SSB} = 0$ is used:

```
ibar_SSB = 0;
dmrsSymbols = nrPBCHDMRS(ncellid,ibar_SSB)
```

dmrsSymbols = 144×1 complex

```
   0.7071 - 0.7071i
   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 + 0.7071i
   0.7071 - 0.7071i
   0.7071 + 0.7071i
   0.7071 - 0.7071i
  -0.7071 - 0.7071i
  -0.7071 - 0.7071i
   0.7071 + 0.7071i
     ⋮
```

Note that TS 38.211 Section 7.4.1.4.1 defines an intermediate variable i_{SSB} which is defined identically to v described previously for the PBCH.

Create the PBCH DM-RS indices and map the PBCH DM-RS symbols to the SS/PBCH block. Note that a scaling factor of 4 is applied to the PBCH DM-RS symbols, to represent $\beta_{\text{PBCH}}^{\text{DM}-\text{RS}}$ in TS 38.211 Section 7.4.3.1.3:

```
dmrsIndices = nrPBCHDMRSIndices(ncellid);
ssblock(dmrsIndices) = 4 * dmrsSymbols;
```

Plot the SS/PBCH block matrix again to show the locations of the PSS, SSS, PBCH and PBCH DM-RS:

```
imagesc(abs(ssblock));
caxis([0 4]);
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS/PBCH block containing PSS, SSS, PBCH and PBCH DM-RS');
```
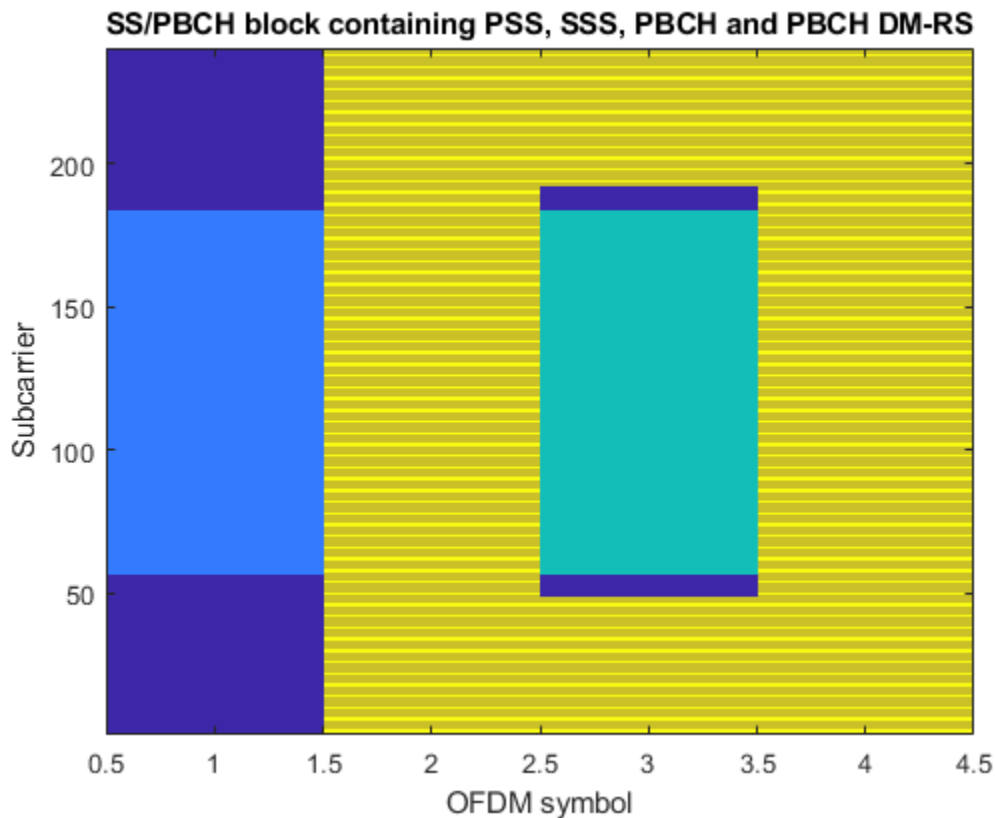

Generating an SS burst

An SS burst, consisting of multiple SS/PBCH blocks, can be generated by creating a larger grid and mapping SS/PBCH blocks into the appropriate locations, with each SS/PBCH block having the correct parameters according to the location.

Create SS burst grid

In the NR standard, OFDM symbols are grouped into slots, subframes and frames. As defined in TS 38.211 Section 4.3.1, there are 10 subframes in a frame, and each subframe has a fixed duration of 1ms. Each SS burst has a duration of half a frame, and therefore spans 5 subframes:

```
nSubframes = 5
```

```
nSubframes = 5
```

TS 38.211 Section 4.3.2 defines each slot as having 14 OFDM symbols (for normal cyclic prefix length) and this is fixed:

```
symbolsPerSlot = 14
```

```
symbolsPerSlot = 14
```

However, the number of slots per subframe varies and is a function of the subcarrier spacing. As the subcarrier spacing increases, the OFDM symbol duration decreases and therefore more OFDM symbols can be fitted into the fixed subframe duration of 1ms.

There are 5 subcarrier spacing configurations $\mu = 0 \ldots 4$, with the corresponding subcarrier spacing being $15 \cdot 2^{\mu}$ kHz. In this example we shall use $\mu = 1$, corresponding to 30 kHz subcarrier spacing:

```
mu = 1
```

```
mu = 1
```

The number of slots per subframe is 2^{μ}, as doubling the subcarrier spacing halves the OFDM symbol duration. Note that definition of a slot in NR is different from LTE: a subframe in LTE consists of 2 slots of 7 symbols (for normal cyclic prefix) whereas in NR, a subframe using the LTE subcarrier spacing ($\mu = 0$, 15 kHz) consists of 1 slot of 14 symbols.

Calculate the total number of OFDM symbols in an SS burst:

```
nSymbols = symbolsPerSlot * 2^mu * nSubframes
```

```
nSymbols = 140
```

Create an empty grid for the whole SS burst :

```
ssburst = zeros([240 nSymbols]);
```

Define SS block pattern

The pattern of SS/PBCH blocks within an SS burst is indirectly specified by the cell search procedure in TS 38.213, which describes the locations in which the UE may detect an SS/PBCH block. There are 5 block patterns, Case A - Case E, which have different subcarrier spacings and are applicable for different carrier frequencies.

Create the indices of the first symbols in the candidate SS/PBCH blocks for block pattern Case B, which has $L = 8$ blocks per burst:

```
n = [0, 1];
firstSymbolIndex = [4; 8; 16; 20] + 28*n;
firstSymbolIndex = firstSymbolIndex(:).'
```

```
firstSymbolIndex = 1×8

     4     8    16    20    32    36    44    48
```

Create SS burst content

Now a loop can be created which generates each SS block and assigns it into the appropriate location of the SS burst. Note the following:

- The code re-uses various variables created earlier in this example (PSS, SSS, and 4 sets of indices)
- The PSS and SSS are independent of the SS/PBCH block index, so can be mapped into the SS block before the loop
- The PBCH indices and PBCH DM-RS indices are independent of the SS/PBCH block index, so do not need updated in the loop
- i_{SSB}, $\overline{i}_{\text{SSB}}$ and v are set up according to the rules in TS 38.211 Sections 7.3.3.1 and 7.4.1.4.1 for the case of $L = 8$.
- Each channel / signal has been scaled in order to give them different colors in the final plot

```
ssblock = zeros([240 4]);
ssblock(pssIndices) = pssSymbols;
ssblock(sssIndices) = 2 * sssSymbols;

for ssbIndex = 1:length(firstSymbolIndex)

    i_SSB = mod(ssbIndex - 1,8);
    ibar_SSB = i_SSB;
    v = i_SSB;

    pbchSymbols = nrPBCH(cw,ncellid,v);
    ssblock(pbchIndices) = 3 * pbchSymbols;

    dmrsSymbols = nrPBCHDMRS(ncellid,ibar_SSB);
    ssblock(dmrsIndices) = 4 * dmrsSymbols;

    ssburst(:,firstSymbolIndex(ssbIndex) + (0:3)) = ssblock;

end
```

Finally, plot the SS burst content:

```
imagesc(abs(ssburst));
caxis([0 4]);
axis xy;
xlabel('OFDM symbol');
ylabel('Subcarrier');
title('SS burst, block pattern Case B');
```

SS burst, block pattern Case B
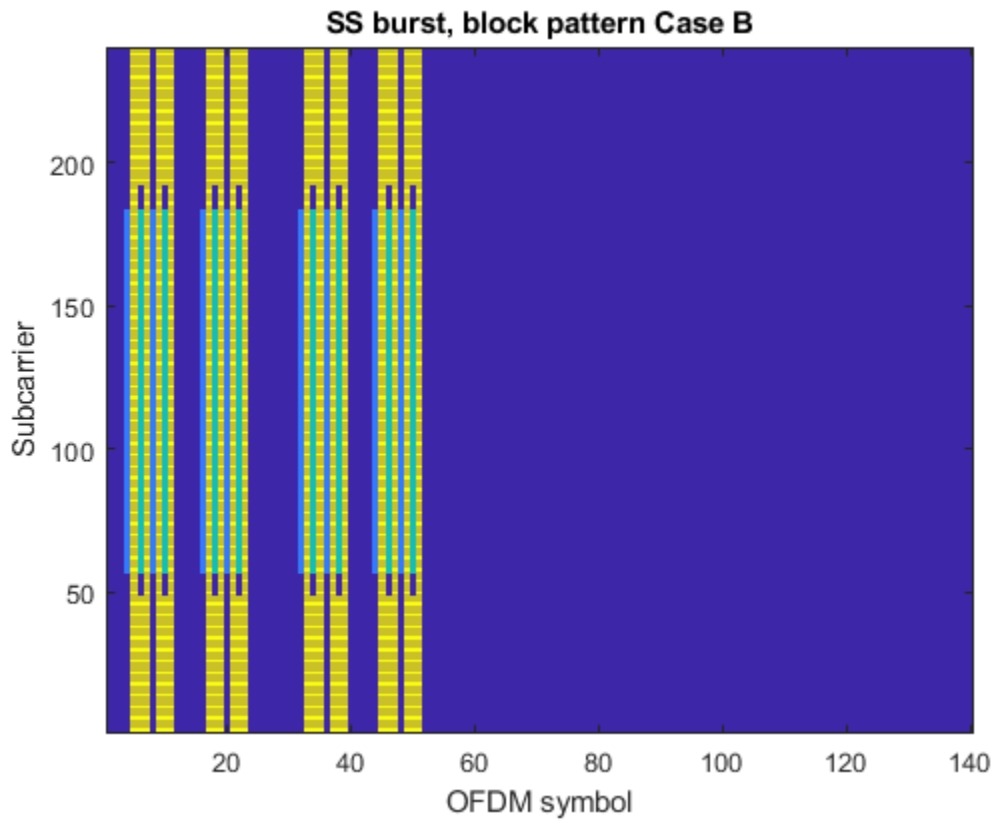
See Also

Functions
nrPBCH | nrPBCHDMRS | nrPBCHDMRSIndices | nrPBCHIndices | nrSSS | nrSSSIndices | nrPSS | nrPSSIndices

More About

- "NR Cell Search and MIB and SIB1 Recovery"

Modeling Downlink Control Information

This example describes the downlink control information (DCI) processing for the 5G New Radio communications system. Starting from a random DCI message, it models the message encoding followed by the physical downlink control channel (PDCCH) processing on the transmit end. Corresponding receiver components recover the transmitted control information elements.

System Parameters

Set parameters for a UE-specific search space.

```
rng(211);               % Set RNG state for repeatability

nID = 23;               % pdcch-DMRS-ScramblingID
rnti = 100;             % C-RNTI for PDCCH in a UE-specific search space
K = 64;                 % Number of DCI message bits
E = 288;                % Number of bits for PDCCH resources
```

DCI Encoding

The DCI message bits based on a downlink format are encoded using the `nrDCIEncode` function, which includes the stages of CRC attachment, polar encoding and rate matching.

```
dciBits = randi([0 1],K,1,'int8');
dciCW = nrDCIEncode(dciBits,rnti,E);
```

PDCCH Symbol Generation

The encoded DCI bits (a codeword) are mapped onto the physical downlink control channel (PDCCH) using the `nrPDCCH` function which generates the scrambled, QPSK-modulated symbols. The scrambling accounts for the user-specific parameters.

```
sym = nrPDCCH(dciCW,nID,rnti);
```

For NR, the PDCCH symbols are then mapped to the resource elements of an OFDM grid which also has PDSCH, PBCH and other reference signal elements. These are followed by OFDM modulation and transmission over a channel. For simplicity, we directly pass the PDCCH symbols over an AWGN channel next.

The following schematic depicts the components used in the example for DCI processing.
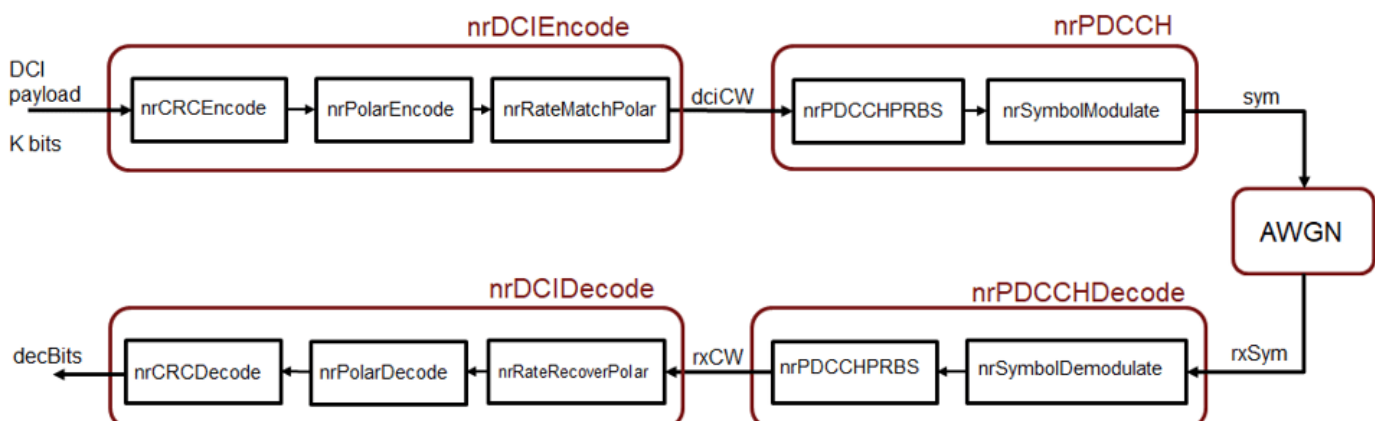
Channel

The PDCCH symbols are transmitted over an AWGN channel with a specified SNR, accounting for the coding rate and QPSK modulation.

```
EbNo = 3;                         % in dB
bps = 2;                          % bits per symbol, 2 for QPSK
EsNo = EbNo + 10*log10(bps);
snrdB = EsNo + 10*log10(K/E);

rxSym = awgn(sym,snrdB,'measured');
```

PDCCH Decoding

The received symbols are demodulated with known user-specific parameters and channel noise variance using the `nrPDCCHDecode` function. The soft output is the log-likelihood ratio for each bit in the codeword.

```
noiseVar = 10.^(-snrdB/10);       % assumes unit signal power
rxCW = nrPDCCHDecode(rxSym,nID,rnti,noiseVar);
```

DCI Decoding

An instance of the received PDCCH codeword is then decoded by the `nrDCIDecode` function. This includes the stages of rate recovery, polar decoding and CRC decoding to recover the transmitted information bits.

```
listLen = 8;                      % polar decoding list length
[decDCIBits,mask] = nrDCIDecode(rxCW,K,listLen,rnti);

isequal(mask,0)

ans = logical
   1


isequal(decDCIBits,dciBits)

ans = logical
   1
```

For a known recipient, the C-RNTI information aids decoding. The output mask value of 0 indicates no errors in the transmission. For the chosen system parameters, the decoded information matches the transmitted information bits.

See Also

Functions
nrDCIDecode | nrDCIEncode | nrPDCCHDecode | nrPDCCH

More About

- "Downlink Control Processing and Procedures"

5G New Radio Polar Coding

This example highlights the new polar channel coding technique chosen for 5G New Radio (NR) communications system. Of the two main types of code constructions specified by 3GPP, this example models the CRC-Aided Polar (CA-Polar) coding scheme. This example describes the main components of the polar coding scheme with individual components for code construction, encoding and decoding along-with rate-matching. It models a polar-coded QPSK-modulated link over AWGN and presents Block-Error-Rate results for different message lengths and code rates for the coding scheme.

Introduction

The selection of polar codes as the channel coding technique for control channels for 5G NR communications system has proven the merits of Arikan's [1] discovery and will establish their application in commercial systems [6]. Based on the concept of channel polarization, this new coding family is capacity achieving as opposed to just capacity approaching. With better or comparable performance than LDPC and turbo codes, it supersedes the tail-biting convolutional codes used in LTE systems for control channels. It is applied for downlink and uplink control information (DCI/UCI) for the enhanced mobile broadband (eMBB) use case, as well as the broadcast channel (BCH). Alternatively, the channel coding scheme for data channels for eMBB is specified to be flexible LDPC for all block sizes.

This example highlights the components to enable a polar coding downlink simulation using QPSK modulation over an AWGN channel. In the following sections, the individual polar coding components are further detailed.

```
s = rng(100);        % Seed the RNG for repeatability
```

Specify the code parameters used for a simulation.

```
% Code parameters
K = 54;              % Message length in bits, including CRC, K > 30
E = 124;             % Rate matched output length, E <= 8192

EbNo = 0.8;          % EbNo in dB
L = 8;               % List length, a power of two, [1 2 4 8]
numFrames = 10;      % Number of frames to simulate
linkDir = 'DL';      % Link direction: downlink ('DL') OR uplink ('UL')
```

Polar Encoding

The following schematic details the transmit-end processing for the downlink, with relevant components and their parameters highlighted.
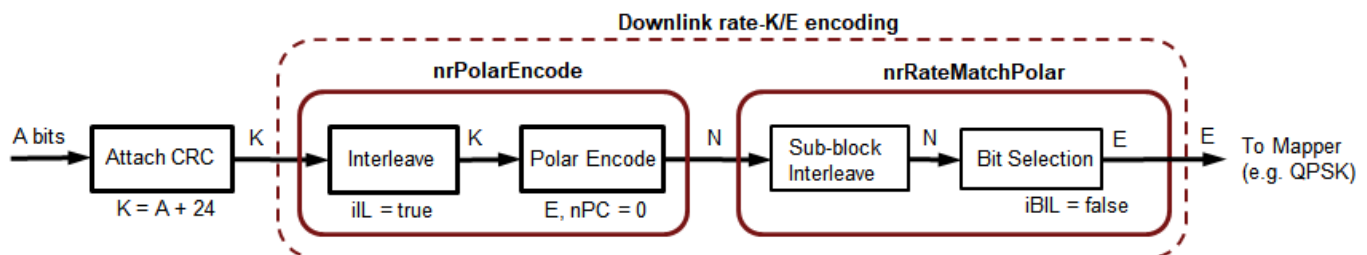
For the downlink, the input bits are interleaved prior to polar encoding. The CRC bits appended at the end of the information bits are thus distributed for the CA-Polar scheme. This interleaving is not specified for the uplink.

The polar encoding uses an SNR-independent method where the reliability of each subchannel is computed offline and the ordered sequence stored for a maximum code length [6]. The nested property of polar codes allows this sequence to be used for any code rate and all code lengths smaller than the maximum code length.

This sequence is computed for given rate-matched output length, E, and information length, K, by the function nrPolarEncode, which implements the non-systematic encoding of the input K bits.

```
if strcmpi(linkDir,'DL')
    % Downlink scenario (K >= 36, including CRC bits)
    crcLen = 24;       % Number of CRC bits for DL, Section 5.1, [6]
    poly = '24C';      % CRC polynomial
    nPC = 0;           % Number of parity check bits, Section 5.3.1.2, [6]
    nMax = 9;          % Maximum value of n, for 2^n, Section 7.3.3, [6]
    iIL = true;        % Interleave input, Section 5.3.1.1, [6]
    iBIL = false;      % Interleave coded bits, Section 5.4.1.3, [6]
else
    % Uplink scenario (K > 30, including CRC bits)
    crcLen = 11;
    poly = '11';
    nPC = 0;
    nMax = 10;
    iIL = false;
    iBIL = true;
end
```

The following schematic details the transmit-end processing for the uplink, for a payload size greater than 19 bits and no code-block segmentation, with relevant components and their parameters highlighted.
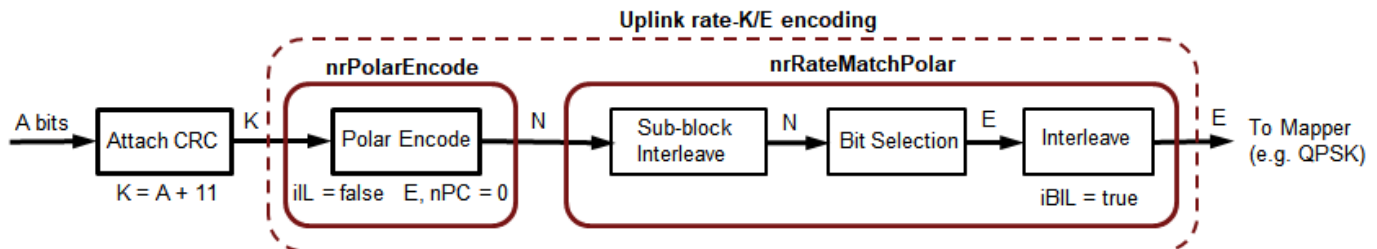

Rate Matching and Rate Recovery

The polar encoded set of bits (N) are rate-matched to output the specified number of bits (E) for resource element mapping [7]. The coded bits are sub-block interleaved and passed to a circular buffer of length N. Depending on the desired code rate and selected values of K, E, and N, either of repetition (E >= N), and puncturing or shortening (E < N) is realized by reading the output bits from the buffer.

- For puncturing, E bits are taken from the end
- For shortening, E bits are taken from the start
- For repetition, E bits are repeated modulo N.

For the downlink, the selected bits are passed on to the modulation mapper, while for the uplink, they are further interleaved prior to mapping. The rate-matching processing is implemented by the function nrRateMatchPolar.

At the receiver end, rate recovery is accomplished for each of the cases

- For puncturing, corresponding LLRs for the bits removed are set to zero
- For shortening, corresponding LLRs for the bits removed are set to a large value
- For repetition, the set of LLRs corresponding to first N bits are selected.

The rate-recovery processing is implemented by the function `nrRateRecoverPolar`.

```
R = K/E;                          % Effective code rate
bps = 2;                          % bits per symbol, 1 for BPSK, 2 for QPSK
EsNo = EbNo + 10*log10(bps);
snrdB = EsNo + 10*log10(R);       % in dB
noiseVar = 1./(10.^(snrdB/10));

% Channel
chan = comm.AWGNChannel('NoiseMethod','Variance','Variance',noiseVar);
```

Polar Decoding

The implicit CRC encoding of the downlink (DCI or BCH) or uplink (UCI) message bits dictates the use of the CRC-Aided Successive Cancellation List Decoding (CA-SCL) [3] as the channel decoder algorithm. It is well known that CA-SCL decoding can outperform turbo or LDPC codes [4] and this was one of the major factors in the adoption of polar codes by 3GPP.

Tal & Vardy [2] describe the SCL decoding algorithm in terms of likelihoods (probabilities). However, due to underflow, the inherent computations are numerically unstable. To overcome this issue, Stimming et.al. [5] offer the SCL decoding solely in the log-likelihood ratio (LLR) domain. The list decoding is characterized by the L parameter, which represents the number of most likely decoding paths retained. At the end of the decoding, the most likely code-path among the L paths is the decoder output. As L is increased, the decoder performance also improves, however, with a diminishing-returns effect.

For an input message which is concatenated with a CRC, CA-SCL decoding prunes out any of the paths for which the CRC is invalid, if at least one path has the correct CRC. This additional insight in the final path selection improves the performance further, when compared to SCL decoding. For the downlink, a CRC of 24 bits is used, while for the uplink CRCs of 6 and 11 bits are specified, which vary on the value of K.

The decoder is implemented by the function `nrPolarDecode`, which supports all three CRC lengths. The decoder function also accounts for the input bit interleaving specified at the transmitter for the downlink, prior to outputting the decoded bits.

```
% Error meter
ber = comm.ErrorRate;
```

Frame Processing Loop

This section shows how the prior described components for polar coding are used in a Block Error Rate (BLER) simulation. The simulation link is highlighted in the following schematic.
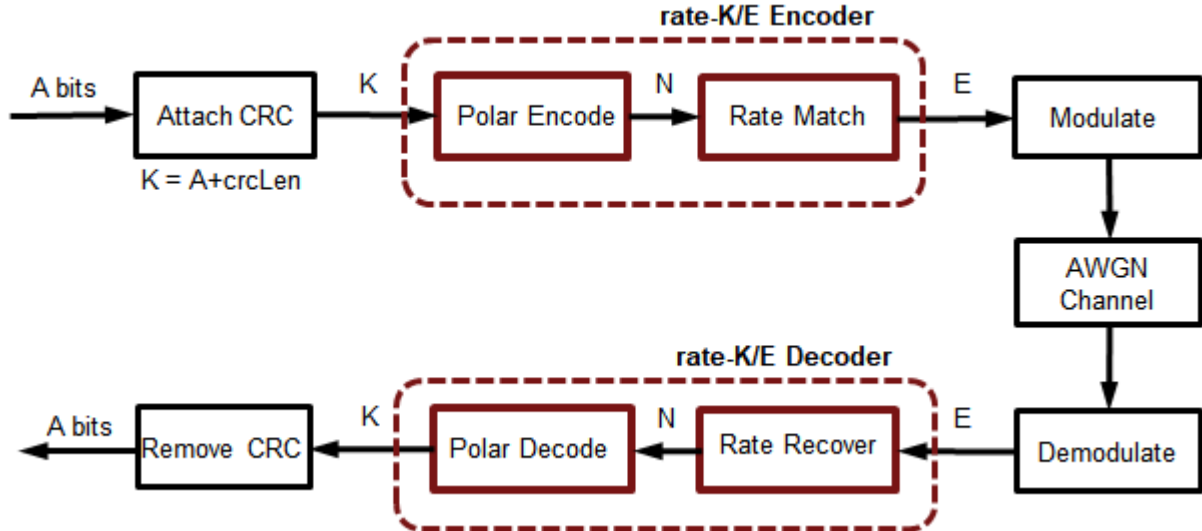
For each frame processed, the following steps are performed:

- `K-crcLen` random bits are generated,
- A CRC is computed and appended to these bits
- The CRC appended bits are polar encoded to the mother code block length
- Rate-matching is performed to transmit `E` bits
- The `E` bits are QPSK modulated
- White Gaussian Noise of specified power is added
- The noisy signal is soft QPSK demodulated to output LLR values
- Rate recovery is performed accounting for either of puncturing, shortening or repetition
- The recovered LLR values are polar decoded using the CA-SCL algorithm, including deinterleaving.
- Off the decoded `K` bits, the first `K-crcLen` bits are compared with those transmitted to update the BLER and bit-error-rate (BER) metrics.

At the end of the simulation, the two performance indicators, BLER and BER, are reported.

```
numferr = 0;
for i = 1:numFrames

    % Generate a random message
    msg = randi([0 1],K-crcLen,1);

    % Attach CRC
    msgcrc = nrCRCEncode(msg,poly);

    % Polar encode
    encOut = nrPolarEncode(msgcrc,E,nMax,iIL);
    N = length(encOut);

    % Rate match
    modIn = nrRateMatchPolar(encOut,K,E,iBIL);
```

```matlab
    % Modulate
    modOut = nrSymbolModulate(modIn,'QPSK');

    % Add White Gaussian noise
    rSig = chan(modOut);

    % Soft demodulate
    rxLLR = nrSymbolDemodulate(rSig,'QPSK',noiseVar);

    % Rate recover
    decIn = nrRateRecoverPolar(rxLLR,K,N,iBIL);

    % Polar decode
    decBits = nrPolarDecode(decIn,K,E,L,nMax,iIL,crcLen);

    % Compare msg and decoded bits
    errStats = ber(double(decBits(1:K-crcLen)), msg);
    numferr = numferr + any(decBits(1:K-crcLen)~=msg);

end

disp(['Block Error Rate: ' num2str(numferr/numFrames) ...
      ', Bit Error Rate: ' num2str(errStats(1)) ...
      ', at SNR = ' num2str(snrdB) ' dB'])

rng(s);     % Restore RNG

Block Error Rate: 0, Bit Error Rate: 0, at SNR = 0.20002 dB
```

Results

To get meaningful results, simulations have to be run for a longer duration. Using scripts which encapsulate the above processing into a function that supports C-code generation, the following results for different code rates and message lengths are presented for both link directions with QPSK modulation.
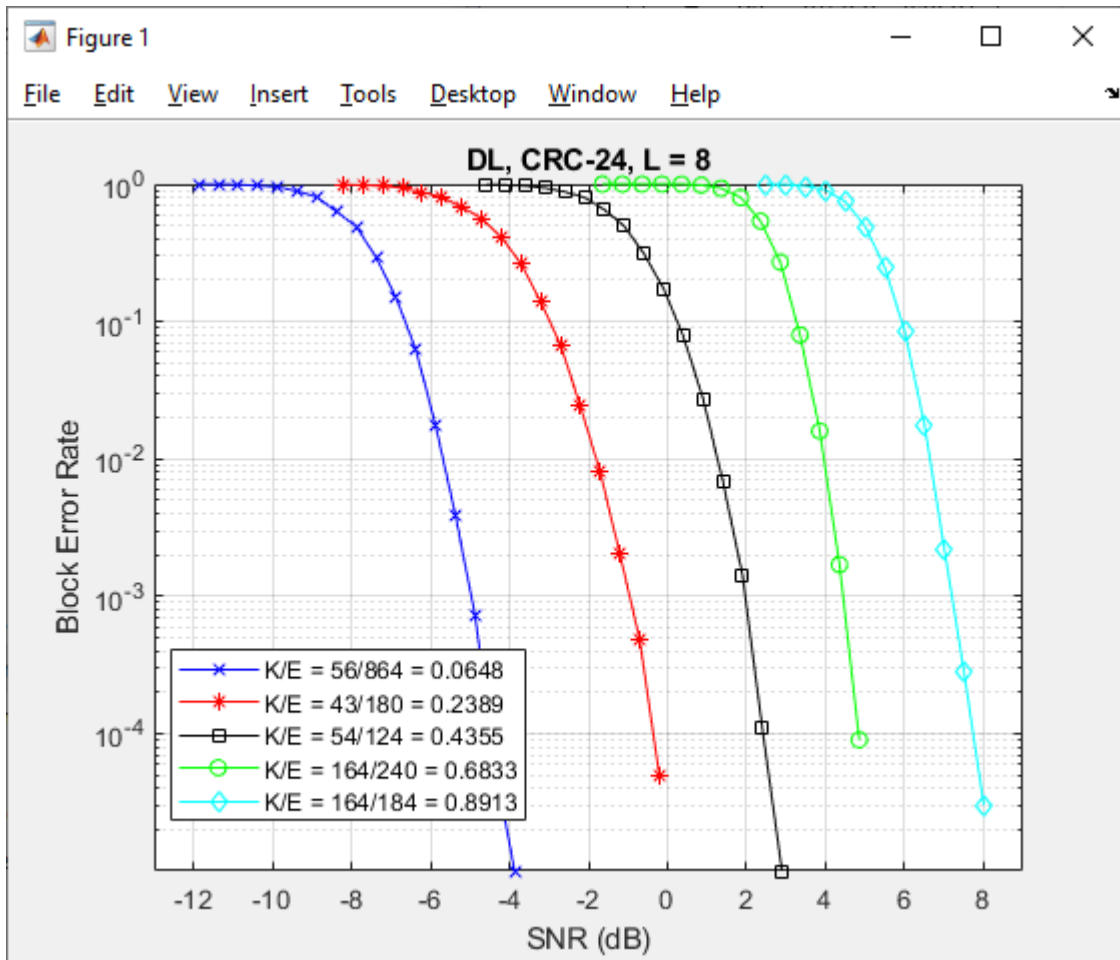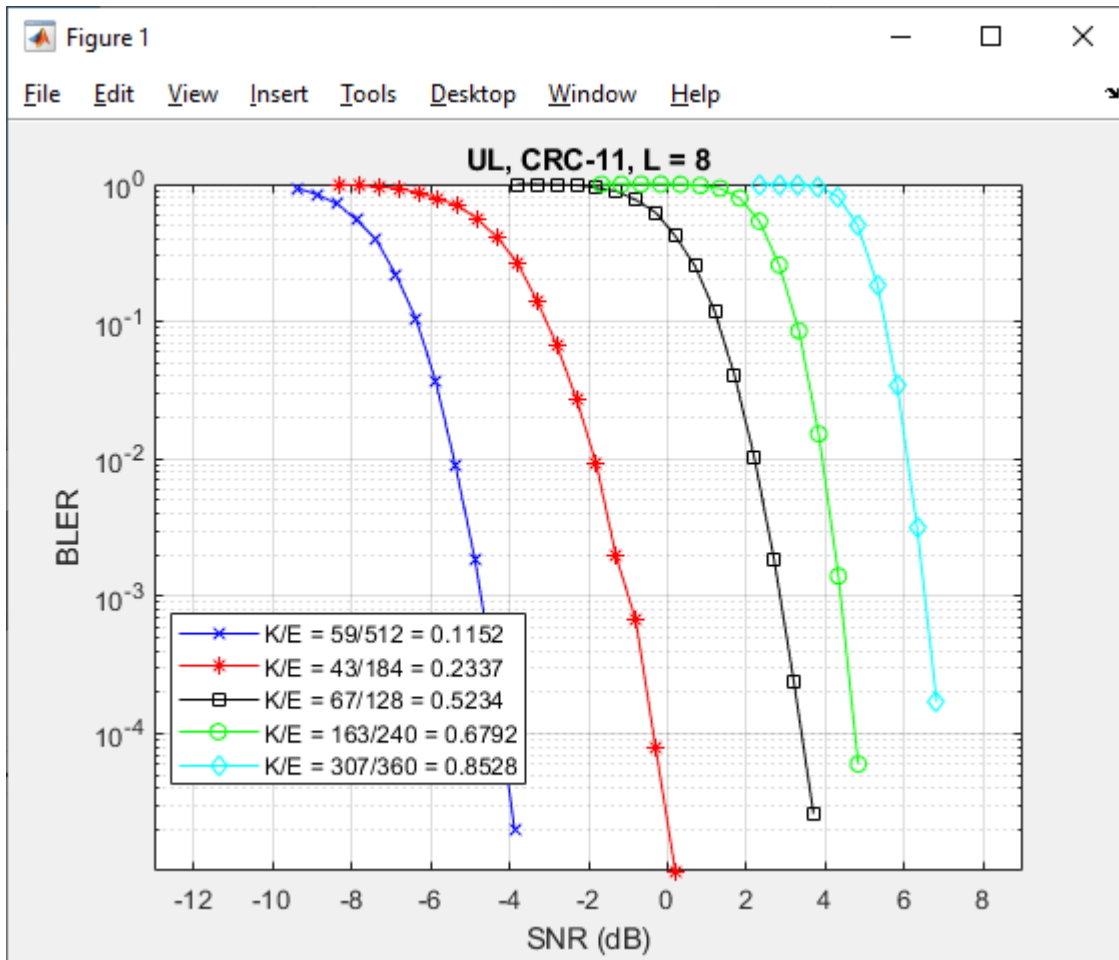
The above results were generated by simulating, for each SNR point, up to 1000 frame errors or a maximum of 100e3 frames, whichever occurred first.

The BLER performance results indicate the suitability of polar codes in a communication link and their implicit support for rate-compatibility at the bit-level granularity.

The use of C-code generation tools for the components reduces the execution time, a key concern for simulations. The C-code generation is enabled by MATLAB Coder™.

Summary and Further Exploration

This example highlights one of the polar coding schemes (CRC-Aided Polar) specified by 3GPP for New Radio control channel information (DCI, UCI) and broadcast channel (BCH). It shows the use of components for all stages of the processing (encoding, rate-matching, rate-recovery and decoding) and uses them in a link with QPSK over an AWGN channel. Highlighted performance results for different code rates and message lengths show agreement to published trends, within parametric and simulation assumption variations.

Explore simple parameter variations (K, E, L) and their effect on BLER performance. The polar coding functions are implemented as open MATLAB® code to enable their application for both downlink/ uplink control information and broadcast channel. The CA-Polar scheme is applicable for both

- Downlink, for all message lengths, and
- Uplink, for `K > 30`, with `crcLen = 11`, `nPC = 0`, `nMax = 10`, `iIL = false`, and `iBIL = true`.

Refer to "Modeling Downlink Control Information" on page 2-13 and "NR Cell Search and MIB and SIB1 Recovery" examples, for the use of polar coding functions within the DCI and BCH functions respectively.

The highlighted polar coding functions also support the Parity-Check polar coding construction and encoding. This is applicable for the uplink with UCI payloads in range $18 <= K <= 25$. This is supported by the uplink control coding functions `nrUCIEncode` and `nrUCIDecode`, which include code-block segmentation as well for appropriate values of K and E.

Selected References

1 Arikan, E., "Channel Polarization: A Method for constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels," IEEE Transactions on Information Theory, vol. 55, No. 7, pp. 3051-3073, July 2009.

2 Tal, I, and Vardy, A., "List decoding of Polar Codes", IEEE Transactions on Information Theory, vol. 61, No. 5, pp. 2213-2226, May 2015.

3 Niu, K., and Chen, K., "CRC-Aided Decoding of Polar Codes," IEEE Communications Letters, vol. 16, No. 10, pp. 1668-1671, Oct. 2012.

4 Niu, K., Chen, K., and Lin, J.R., "Beyond turbo codes: rate compatible punctured polar codes", IEEE International Conference on Communications, pp. 3423-3427, 2013.

5 Stimming, A. B., Parizi, M. B., and Burg, A., "LLR-Based Successive Cancellation List Decoding of Polar Codes", IEEE Transaction on Signal Processing, vol. 63, No. 19, pp.5165-5179, 2015.

6 3GPP TS 38.212. "NR; Multiplexing and channel coding" 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.

7 R1-1711729. "WF on circular buffer of Polar Code", 3GPP TSG RAN WG1 meeting NR Ad-Hoc#2, Ericsson, Qualcomm, MediaTek, LGE. June 2017.

See Also

Functions
nrPolarDecode | nrPolarEncode | nrRateRecoverPolar | nrRateMatchPolar

More About

- "Modeling Downlink Control Information" on page 2-13
- "NR Cell Search and MIB and SIB1 Recovery"

LDPC Processing for DL-SCH and UL-SCH

This example highlights the low-density parity-check (LDPC) coding chain for the 5G NR downlink and uplink shared transport channels (DL-SCH and UL-SCH).

Shared Channel Parameters

The example uses the DL-SCH to describe the processing, which also applies to the UL-SCH.

Select parameters for a transport block transmitted on the downlink shared (DL-SCH) channel.

```
rng(210);                % Set RNG state for repeatability

A = 10000;               % Transport block length, positive integer
rate = 449/1024;         % Target code rate, 0<R<1
rv = 0;                  % Redundancy version, 0-3
modulation = 'QPSK';     % Modulation scheme, QPSK, 16QAM, 64QAM, 256QAM
nlayers = 1;             % Number of layers, 1-4 for a transport block
```

Based on the selected transport block length and target coding rate, DL-SCH coding parameters are determined using the `nrDLSCHInfo` function.

```
% DL-SCH coding parameters
cbsInfo = nrDLSCHInfo(A,rate);
disp('DL-SCH coding parameters')
disp(cbsInfo)
```

```
DL-SCH coding parameters
    CRC: '24A'
      L: 24
    BGN: 1
      C: 2
    Lcb: 24
      F: 244
     Zc: 240
      K: 5280
      N: 15840
```

DL-SCH supports multi-codeword transmission (i.e. two transport blocks) while UL-SCH supports only a single codeword. UL-SCH also supports pi/2-BPSK modulation in addition to those listed above for DL-SCH.

Transport Block Processing using LDPC Coding

Data delivered from the MAC layer to the physical layer is termed as a transport block. For the downlink shared channel (DL-SCH), a transport block goes through the processing stages of:

- CRC attachment,
- Code block segmentation and code block CRC attachment,
- Channel coding using LDPC,
- Rate matching and code block concatenation

before being passed on to the physical downlink shared channel (PDSCH) for scrambling, modulation, layer mapping and resource/antenna mapping. Each of these stages is performed by a function as shown next.

```
% Random transport block data generation
in = randi([0 1],A,1,'int8');

% Transport block CRC attachment
tbIn = nrCRCEncode(in,cbsInfo.CRC);

% Code block segmentation and CRC attachment
cbsIn = nrCodeBlockSegmentLDPC(tbIn,cbsInfo.BGN);

% LDPC encoding
enc = nrLDPCEncode(cbsIn,cbsInfo.BGN);

% Rate matching and code block concatenation
outlen = ceil(A/rate);
chIn = nrRateMatchLDPC(enc,outlen,rv,modulation,nlayers);
```

The output number of bits from the rate matching and code block concatenation process must match the bit capacity of the PDSCH, based on the available resources. In this example, as the PDSCH is not modeled, this is set to achieve the target code rate based on the transport block size previously selected.

Similar processing applies for the UL-SCH, where the physical uplink shared channel (PUSCH) is the recipient of the UL-SCH codeword. The following schematics depict the processing for the two channels.
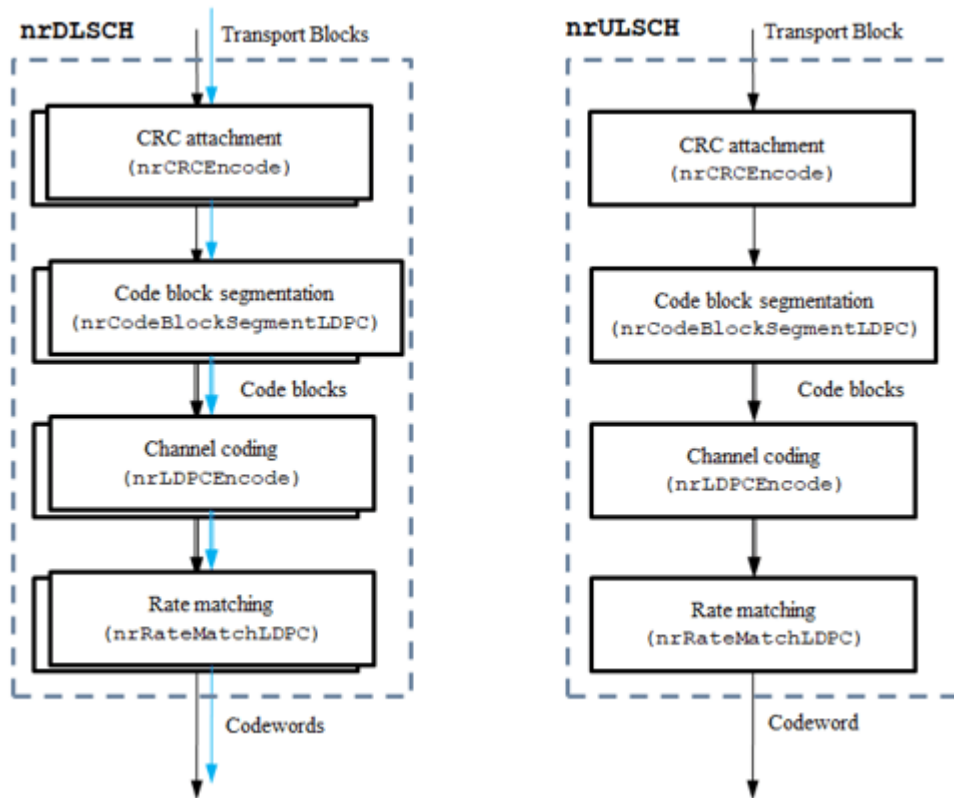
Refer to nrDLSCH and nrULSCH System objects that encapsulate the processing per transport block, with additional support for retransmissions.

Channel

A simple bipolar channel with no noise is used for this example. With the full PDSCH or PUSCH processing, one can consider fading channels, AWGN and other RF impairments as well.

```
chOut = double(1-2*(chIn));
```

Receive Processing using LDPC Decoding

The receive end processing for the DL-SCH channel comprises of the corresponding dual operations to the transmit end that include

- Rate recovery
- LDPC decoding
- Code block desegmentation and CRC decoding
- Transport block CRC decoding

Each of these stages is performed by a function as shown next.

```
% Rate recovery
raterec = nrRateRecoverLDPC(chOut,A,rate,rv,modulation,nlayers);

% LDPC decoding
decBits = nrLDPCDecode(raterec,cbsInfo.BGN,25);

% Code block desegmentation and CRC decoding
[blk,blkErr] = nrCodeBlockDesegmentLDPC(decBits,cbsInfo.BGN,A+cbsInfo.L);

disp(['CRC error per code-block: [' num2str(blkErr) ']'])

% Transport block CRC decoding
[out,tbErr] = nrCRCDecode(blk,cbsInfo.CRC);

disp(['Transport block CRC error: ' num2str(tbErr)])
disp(['Recovered transport block with no error: ' num2str(isequal(out,in))])

CRC error per code-block: [0  0]
Transport block CRC error: 0
Recovered transport block with no error: 1
```

As the displays indicate, there are no CRC errors at both the code-block and transport block levels. This leads to the transport block being recovered and decoded with no errors, as expected, for a noiseless channel.

Refer to `nrDLSCHDecoder` and `nrULSCHDecoder` System objects that encapsulate the receive processing per codeword, with additional soft-combining of retransmissions for improved performance.

See Also

Functions
nrLDPCEncode | nrRateMatchLDPC | nrRateRecoverLDPC | nrLDPCDecode | nrDLSCHInfo | nrCRCEncode | nrCRCDecode | nrCodeBlockSegmentLDPC | nrCodeBlockDesegmentLDPC | nrULSCHInfo | nrDLSCHDecoder | nrDLSCH | nrULSCH | nrULSCHDecoder

More About

- "NR PDSCH Throughput"
- "NR PUSCH Throughput"

Generate Wireless Waveform in Simulink Using App-Generated Block

This example shows how to configure and use the block that is generated using the **Export to Simulink** capability that is available in the **Wireless Waveform Generator** app.

Introduction

The **Wireless Waveform Generator** app is an interactive tool for creating, impairing, visualizing, and exporting waveforms. You can export the waveform to your workspace or to a `.mat` or `.bb` file. You can also export the waveform generation parameters to a runnable MATLAB® script or a Simulink® block. You can use the exported Simulink block to reproduce your waveform in Simulink. This example shows how to use the **Export to Simulink** capability of the app and how to configure the exported block to generate waveforms in Simulink.

Although this example focuses on exporting an OFDM waveform, the same process applies for all of the supported waveform types.

Export Wireless Waveform Configuration to Simulink

Open the **Wireless Waveform Generator** app by clicking the app icon on the **Apps** tab, under **Signal Processing and Communications**. Alternatively, enter `wirelessWaveformGenerator` at the MATLAB command prompt.

In the **Waveform Type** section, select an OFDM waveform by clicking **OFDM**. In the left-most pane of the app, adjust any configuration parameters for the selected waveform. Then export the configuration by clicking **Export** in the app toolstrip and selecting **Export to Simulink**.
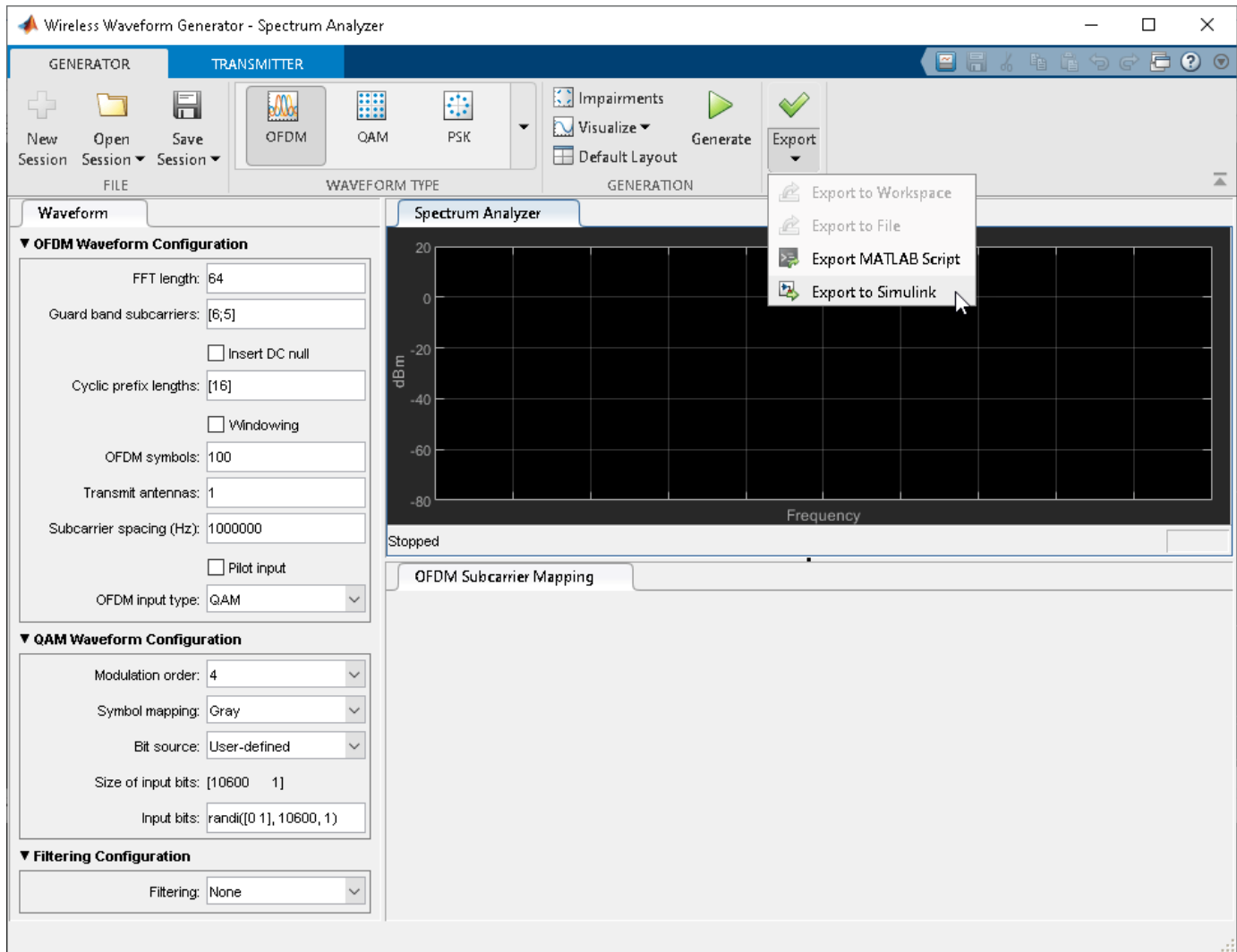
The **Export to Simulink** option creates a Simulink block, which outputs the selected waveform when you run the Simulink model. The block is exported to a new model if no open models exist.

```
modelName = 'WWGExport2SimulinkBlock';
open_system(modelName);
```
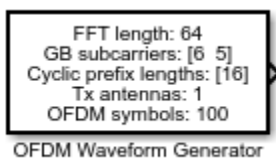

The **Form output after final data value by** block parameter specifies the output after all of the specified signal samples are generated. The value options for this parameter are `Cyclic repetition` and `Setting to zero`. The `Cyclic repetition` option repeats the signal from the beginning after it reaches the last sample in the signal. The `Setting to zero` option generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

The **Waveform sample rate (Fs)** and **Waveform length** block parameters are derived from the waveform configuration that is available in the **Initialization** tab of the Mask Editor dialog box. For further information about the block parameters, see Waveform From Wireless Waveform Generator App. This figure shows the parameters of the exported block.
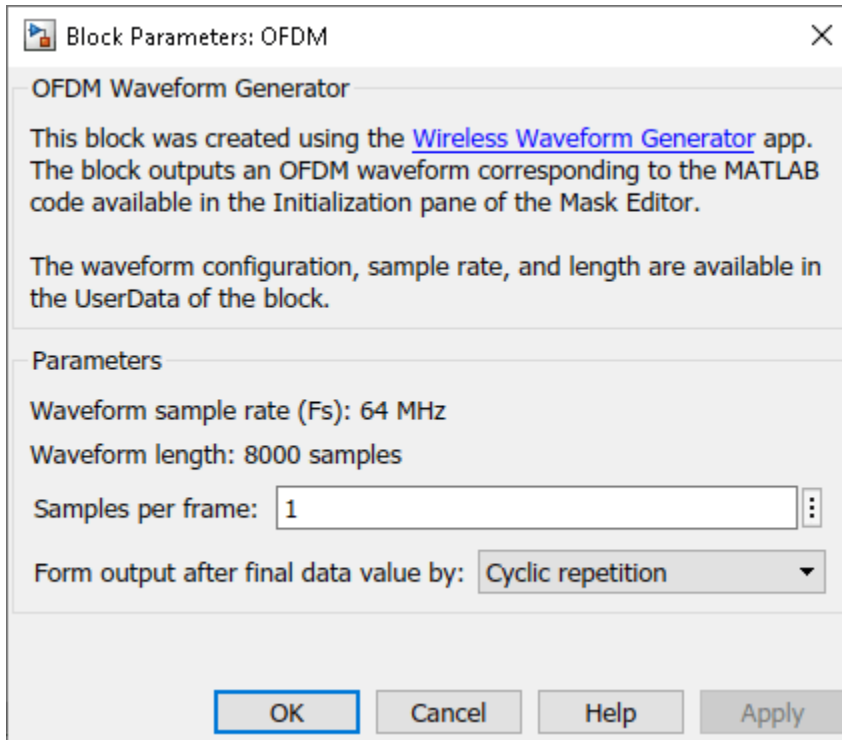

```
close_system(modelName);
```

Connect a Spectrum Analyzer block to the exported block.

```
modelName = 'WWGExport2SimulinkModel';
open_system(modelName);
```
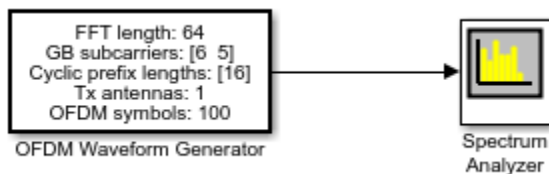

Simulate the model to visualize the waveform using the current configuration.

```
sim(modelName);
```
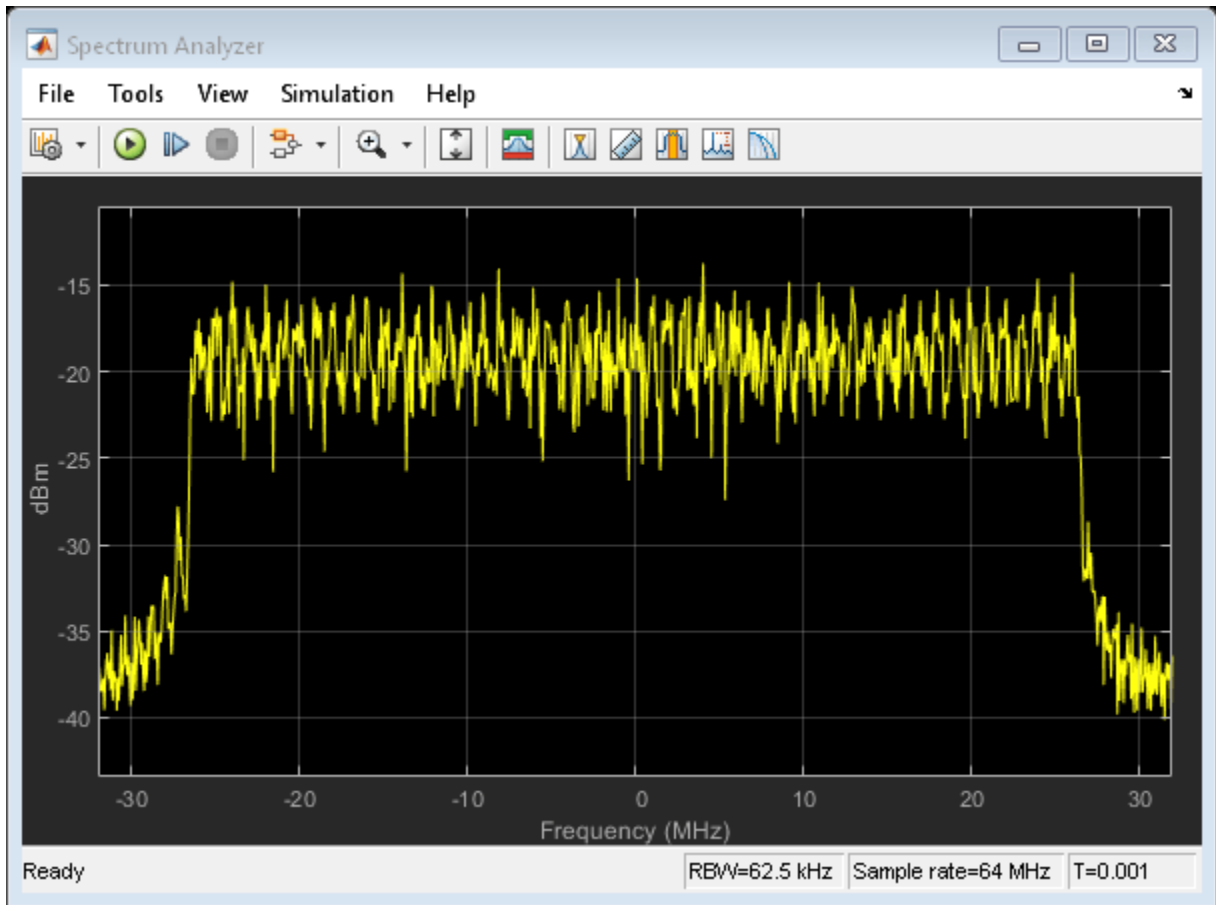
The Spectrum Analyzer block inherits the **Waveform sample rate (Fs)** parameter, which is 64 MHz.

```
close_system(modelName);
```

Modify Wireless Waveform Configuration

When you run the Simulink model, the exported block outputs the waveform generated in the **Initialization** tab of the Mask Editor dialog box for the block. The MATLAB code that initializes the waveform in this tab corresponds to the configuration that you selected in the **Wireless Waveform Generator** app before exporting the block. To modify the configuration of the waveform, choose one of these options:

- Open the **Wireless Waveform Generator** app, select the configuration of your choice, and export a new block. This option provides interaction with an app interface instead of MATLAB code, parameter range validation during the parameterization process, and visualization of the waveform before running the Simulink model.

- Update the configuration parameters that are available in the **Initialization** tab of the Mask Editor dialog box of the exported block. This option requires modifying the MATLAB code available in this tab so that the parameter range validation occurs only when you apply the changes. This option does not provide visualization of the waveform before running the Simulink model. Modifying the waveform parameters using this option is not recommended if you are not familiar with the MATLAB code that generates the selected waveform.

If you choose to modify the configuration by using the **Initialization** tab, you can open it by clicking the exported block, pressing **Ctrl+M** to open the Mask Editor dialog, and clicking the **Initialization** tab.
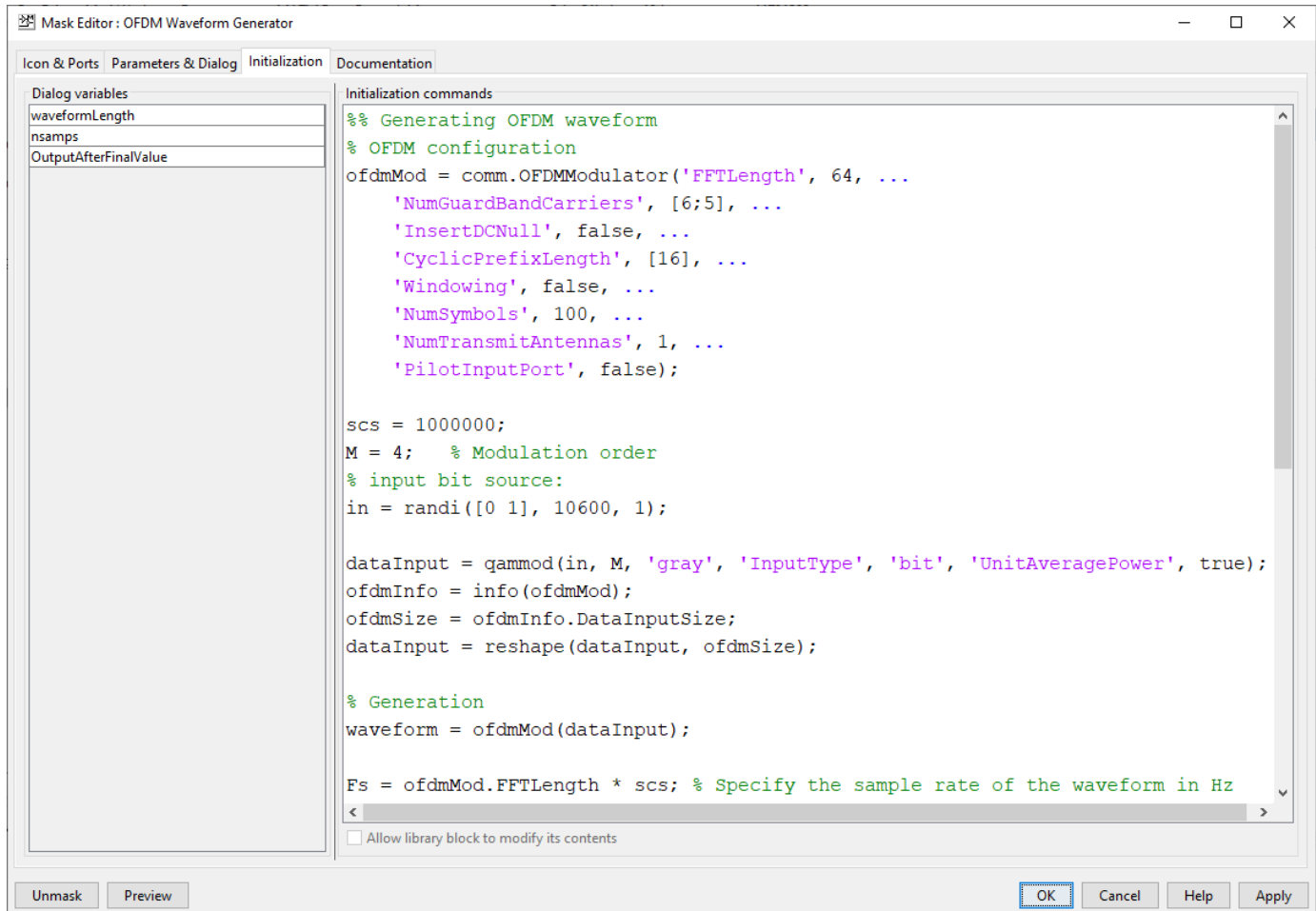
Use the MATLAB code that is available in the **Initialization** tab to update the parameters of your choice. For example, set the subcarrier spacing, `scs`, to 1,500,000 Hz.

Mask Editor : OFDM Waveform Generator

Icon & Ports | Parameters & Dialog | Initialization | Documentation

Dialog variables
- waveformLength
- nsamps
- OutputAfterFinalValue

Initialization commands

```matlab
%% Generating OFDM waveform
% OFDM configuration
ofdmMod = comm.OFDMModulator('FFTLength', 64, ...
    'NumGuardBandCarriers', [6;5], ...
    'InsertDCNull', false, ...
    'CyclicPrefixLength', [16], ...
    'Windowing', false, ...
    'NumSymbols', 100, ...
    'NumTransmitAntennas', 1, ...
    'PilotInputPort', false);

scs = 1500000;
M = 4;    % Modulation order
% input bit source:
in = randi([0 1], 10600, 1);

dataInput = qammod(in, M, 'gray', 'InputType', 'bit', 'UnitAveragePower', true);
ofdmInfo = info(ofdmMod);
ofdmSize = ofdmInfo.DataInputSize;
dataInput = reshape(dataInput, ofdmSize);

% Generation
waveform = ofdmMod(dataInput);

Fs = ofdmMod.FFTLength * scs; % Specify the sample rate of the waveform in Hz
```

☐ Allow library block to modify its contents

Unmask | Preview | OK | Cancel | Help | Apply

Click **OK** to apply the changes and close the Mask Editor dialog box. Simulate the model to visualize the updated waveform.

```matlab
modelName = 'WWGExport2SimulinkModelSCSModified';
sim(modelName);
```
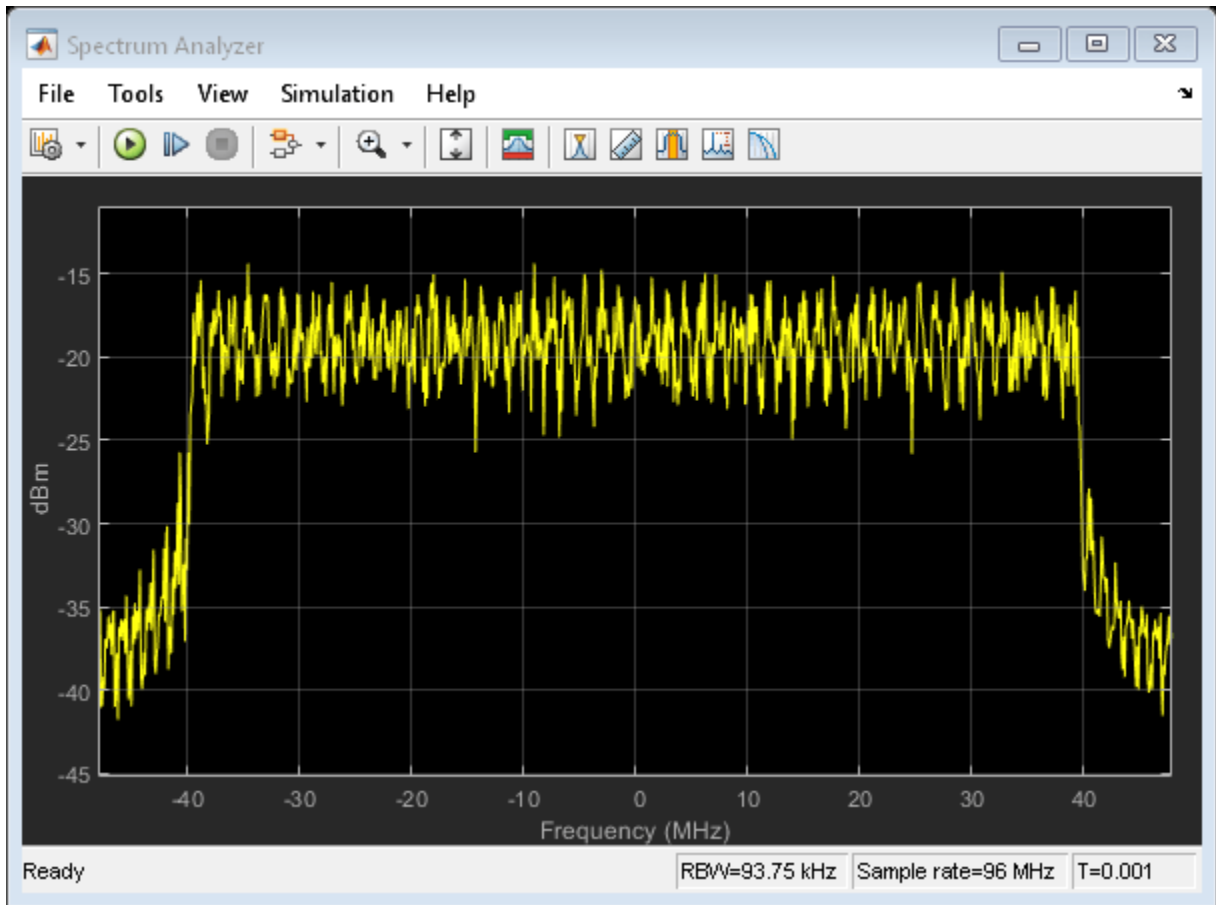
The Spectrum Analyzer block now shows a sample rate of 96 MHz, which is 1.5 times the previous sample rate, as expected.

Share Wireless Waveform Configuration with Other Blocks in the Model

To access read-only block parameters and waveform configuration parameters, use the `UserData` common block property, which is a structure with these fields.

- `WaveformConfig`: Waveform configuration
- `WaveformLength`: Waveform length
- `Fs`: Waveform sample rate

You can access the user data of the exported block by using the `get_param` function.

```
get_param([gcs '/OFDM Waveform Generator'],'UserData')
```

```
ans =

  struct with fields:

    WaveformConfig: [1x1 comm.OFDMModulator]
    WaveformLength: 8000
                Fs: 96000000
```

Store the structure available in the user data in a base workspace variable by using the `InitFcn` in the callback. The `InitFcn` callback is executed during a model update and simulation. To use this callback, click the **MODELING** tab, then click the **Model Settings** dropdown, and click the **Model Properties** option. In the **Callbacks** pane, select the `InitFcn` callback. Assign the user data to a new base workspace variable (for example, `cfg`).
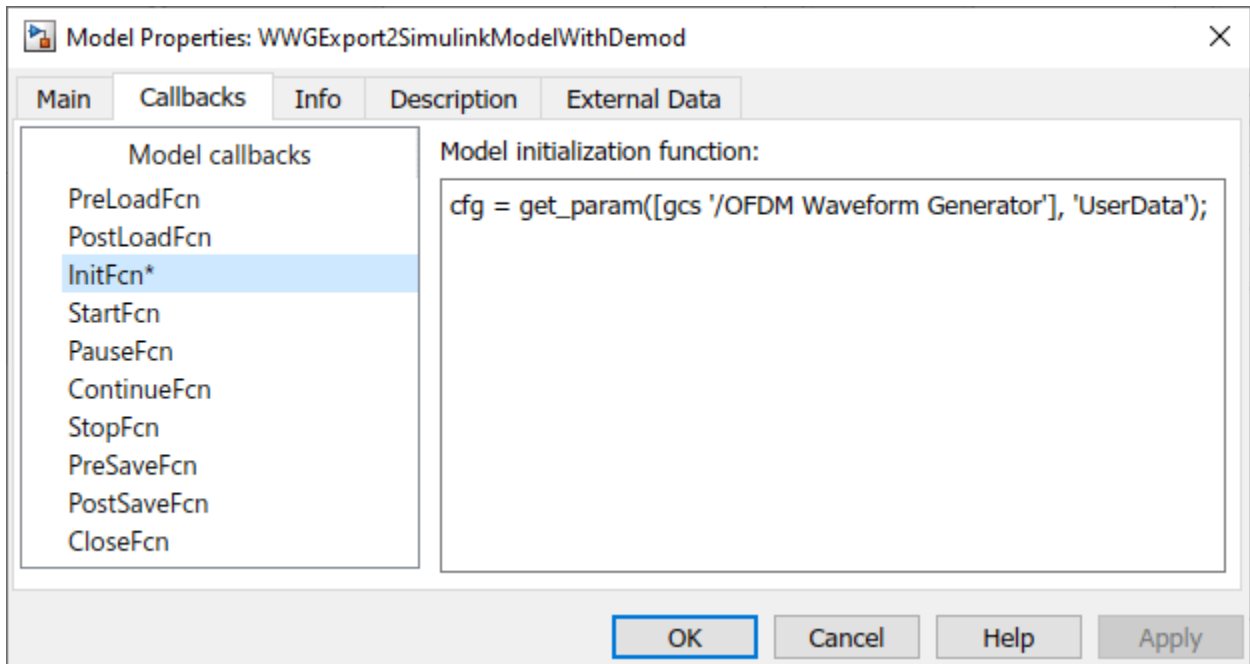
The parameters that are available in the user data of the exported block are updated every time you apply configuration changes in the **Initialization** tab.

To demodulate the OFDM waveform, add an OFDM Demodulator block to the model. Connect an AWGN Channel block between the OFDM Waveform Generator and OFDM Demodulator blocks to add white Gaussian noise to the input signal. Also add a Constellation Diagram block to plot the demodulated symbols.

```
modelName = 'WWGExport2SimulinkModelWithDemod';
open_system(modelName);
```
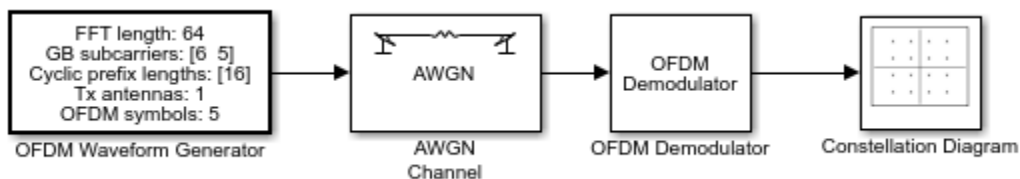

The parameters that are required to configure the OFDM Demodulator block must match the parameters that are used to configure the exported block, (otherwise, demodulation fails). To access the configuration parameters of the exported block, use the variable `cfg`. This figure shows the parameters of the OFDM Demodulator block.

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
Number of guard bands:	cfg.WaveformConfig.NumGuardBandCarriers

☐ Remove DC subcarrier from output

☐ Pilot output port

Cyclic prefix length:	cfg.WaveformConfig.CyclicPrefixLength
Number of OFDM symbols:	cfg.WaveformConfig.NumSymbols
Number of receive antennas:	cfg.WaveformConfig.NumTransmitAntennas

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);
```
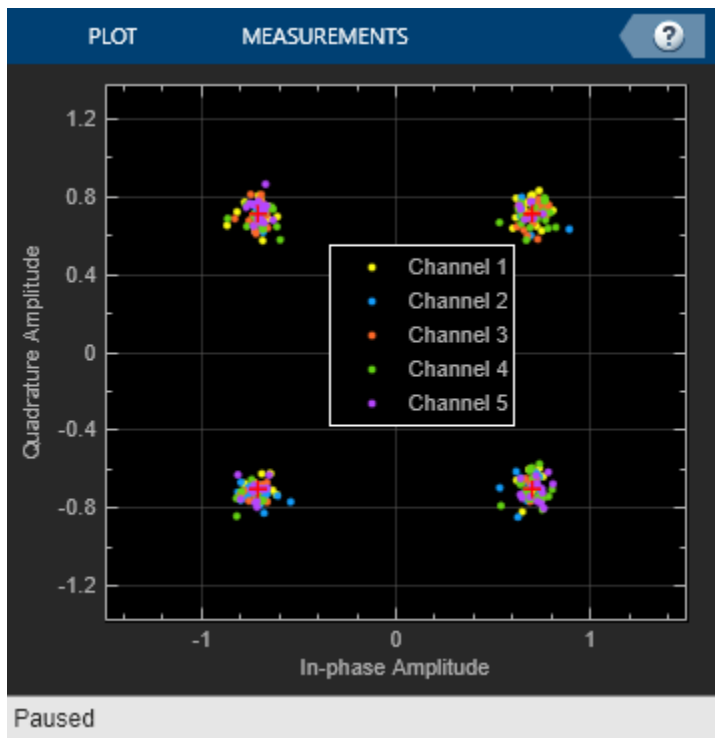
After demodulating the OFDM waveform by using the OFDM Demodulator block, the Constellation Diagram block displays the resulting QAM symbols.

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);
```
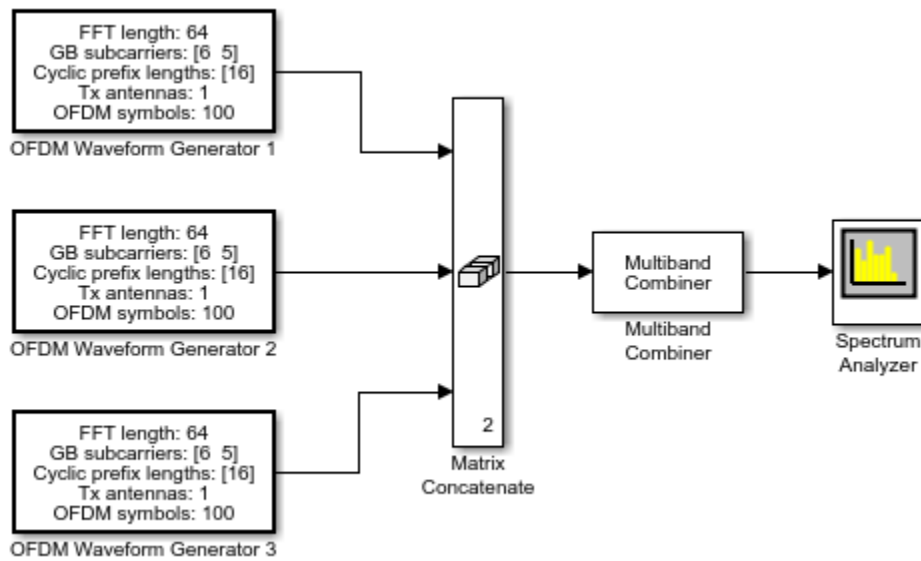
To shift the waveforms in frequency, you might have to increase the sampling rates. The Multiband Combiner block provides the option to oversample the input waveforms before shifting and combining them. This figure shows the parameters of the Multiband Combiner block.

Block Parameters: Multiband Combiner ✕

Multiband Combiner (mask) (link)

Combine baseband signals after shifting them to the desired
frequency bands.

Input signals are interpolated, if needed, to avoid distortion before
shifting them per the frequency offsets parameter values. The
interpolation factor is determined automatically or per the ratio of
output sample rate to input sample rate.

Parameters

Input sample rate (Hz):   cfg1.Fs

Frequency offsets (Hz):   [-80e6 20e6 100e6]

Rate options: Enforce single-rate processing ▼

Output sample rate options: Specify via property ▼

Output sample rate (Hz):   cfg1.Fs*10

Output delay (samples): 120

OK   Cancel   Help   Apply

Simulate the model to visualize the waveforms that are centered at -80, 20, and 100 MHz.

```
sim(modelName);
```
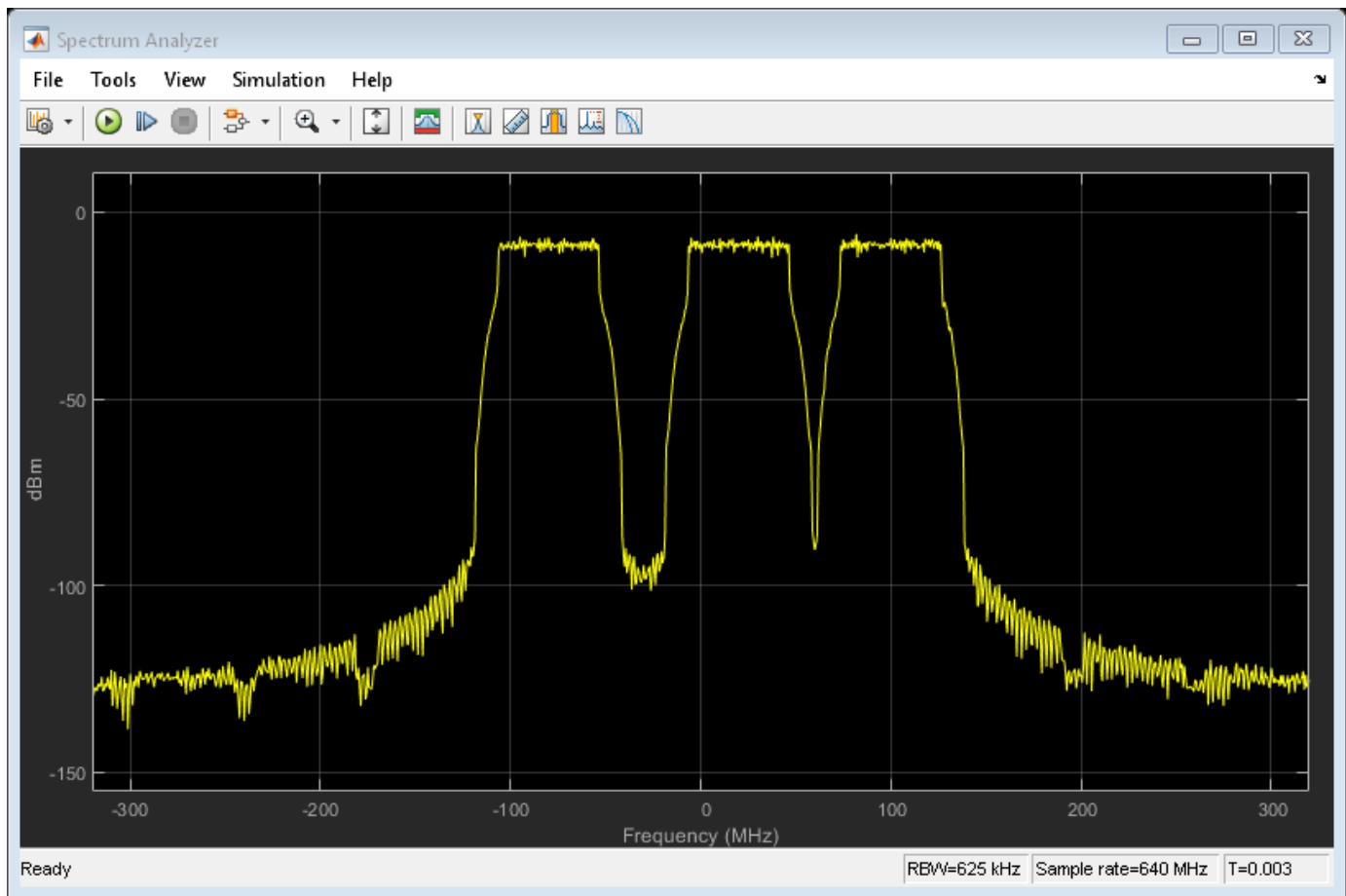
See Also

Apps
5G Waveform Generator

More About

- "Modeling and Testing an NR RF Transmitter"
- "Modeling and Testing an NR RF Receiver with LTE Interference"

Model 5G NR Communication Links

You can model 5G NR shared channel links, including all of the steps from transport block generation to transport channel decoding, by using the 5G Toolbox software. This figure shows the main elements of a downlink communication link.
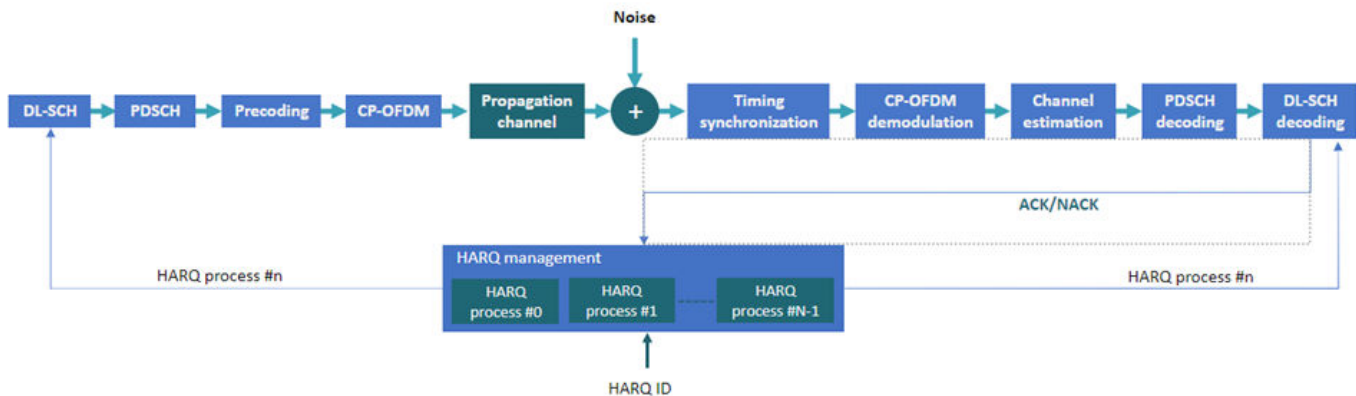
These examples show how to model the main elements of the downlink communication link. Modeling the main elements of the uplink communication link is similar.

- The "Model 5G NR Transport Channels with HARQ" on page 2-41 example describes how to model 5G NR transport channels with multiple hybrid automatic repeat-request (HARQ) processes using the downlink shared channel (DL-SCH) encoder and decoder System objects.
- The "Map 5G Physical Channels and Signals to the Resource Grid" on page 2-49 example shows how to generate, precode, and map 5G NR physical channels and signals to the resource grid.
- The "DL-SCH and PDSCH Transmit and Receive Processing Chain" on page 2-59 example shows how to model the full physical downlink shared channel link, including the coding stages, channel modeling, and the receiver and decoding steps.

See Also

More About
- "Downlink Channels"
- "Uplink Channels"

Model 5G NR Transport Channels with HARQ

This example shows how to model 5G NR transport channels with multiple hybrid automatic repeat-request (HARQ) processes using the downlink shared channel (DL-SCH) encoder and decoder 5G Toolbox™ System objects.

Introduction

This figure shows the link elements that are modeled in this example in the context of a 5G downlink link. These elements are:

- DL-SCH encoding and decoding
- Physical downlink shared channel (PDSCH) encoding and decoding
- HARQ management

The other link elements are not modeled in this example.
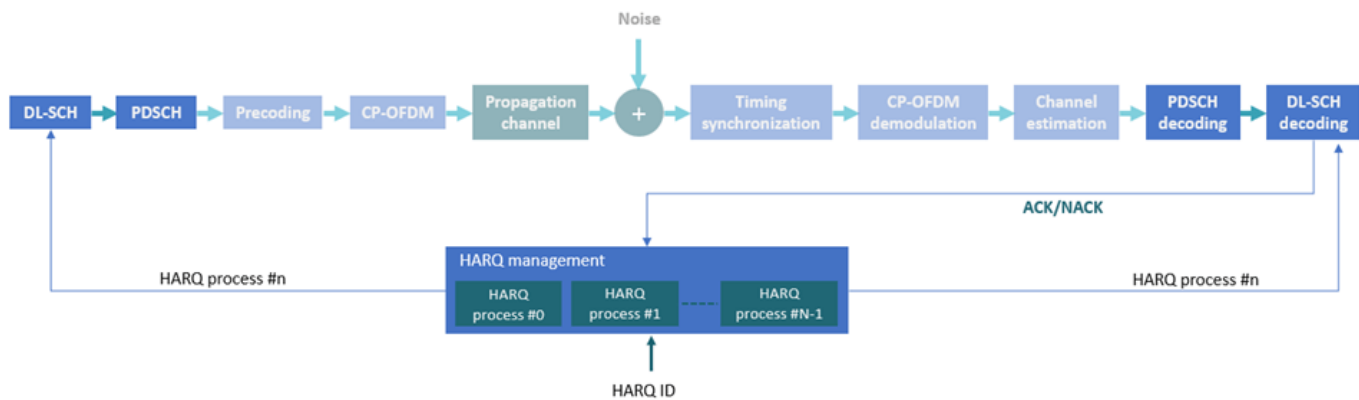
The examples also measures the block error rate (BLER) using an AWGN channel. This figure shows all of the link elements modeled in this example followed by the BLER calculation.
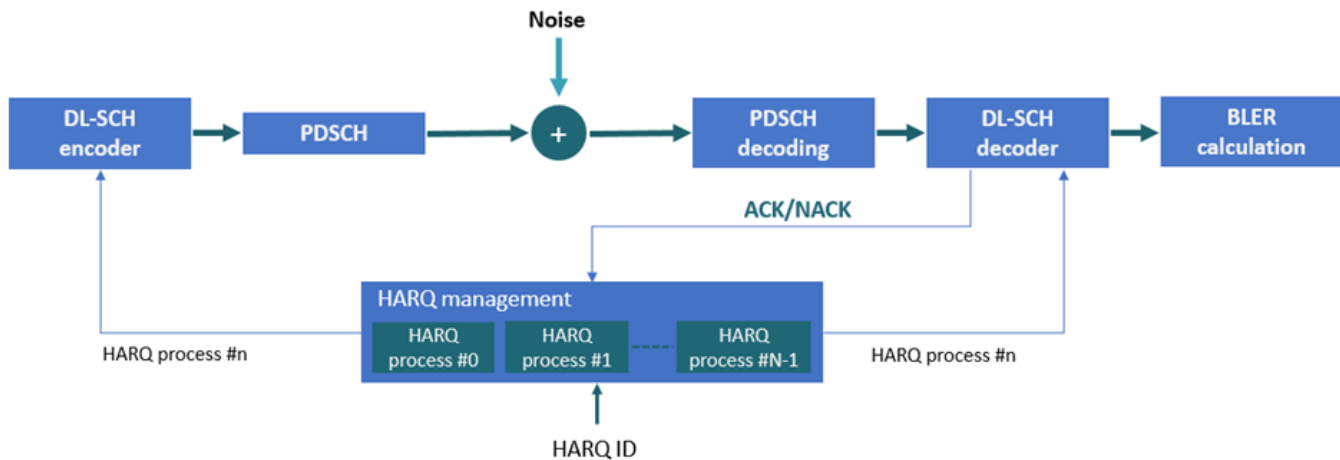
This figure shows that the DL-SCH encoder uses internal buffers to store the transport blocks for each HARQ process and then selects the active HARQ process buffer content for the encoding. The DL-SCH decoder uses a similar buffering mechanism to store and select HARQ processes.

setTransportBlock()   encodeDLSCH()
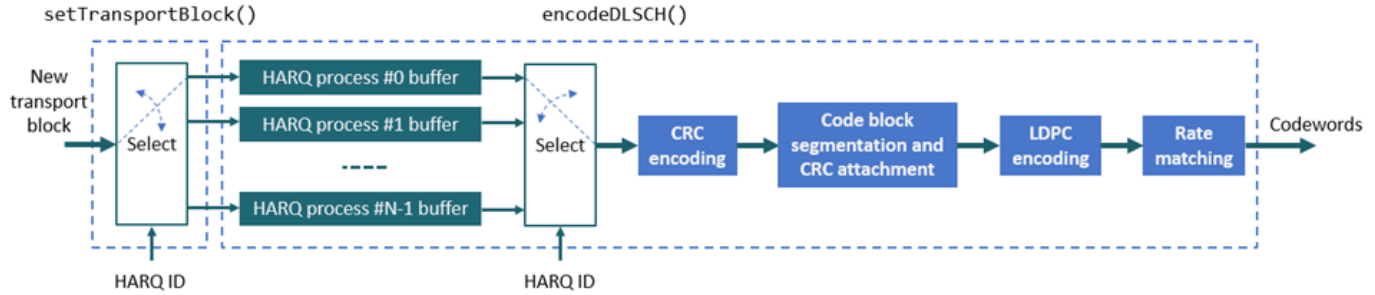
The DL-SCH encoder and decoder do not manage the HARQ processes internally. The example uses the HARQ entity object, `HARQEntity.m`, for HARQ process management. This figure shows the structure of the HARQ entity object
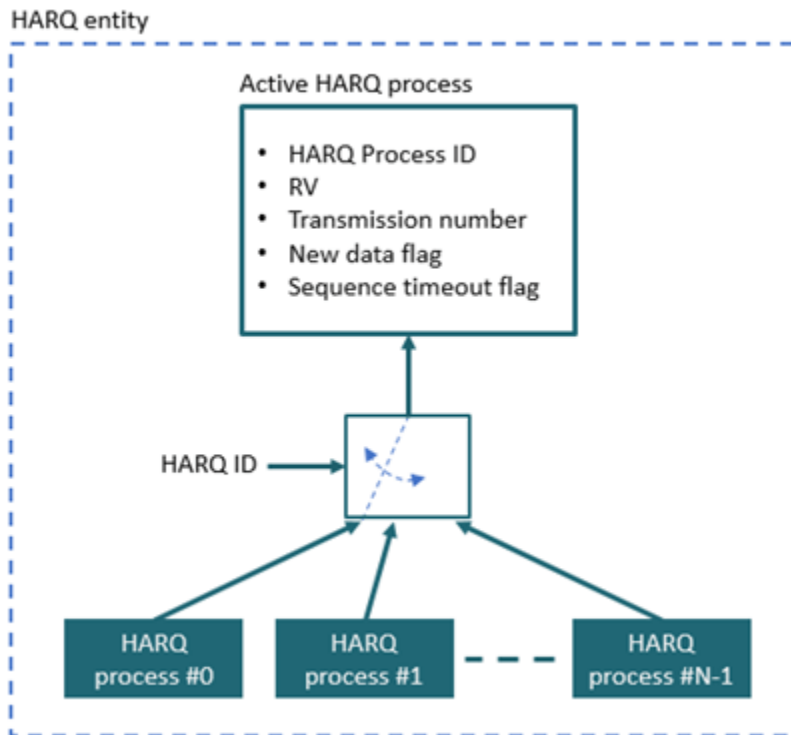
Simulation Parameters

Specify the number of transport blocks to simulate and the signal to noise ratio (SNR).

```
noTransportBlocks = 100;
SNRdB = 7; % SNR in dB
```

Reset random number generator for reproducibility.

```
rng("default");
```

DL-SCH Configuration

Specify the code rate, the number of HARQ processes, and the redundancy values (RVs) sequence. This sequence controls the redundancy version retransmissions in case of error.

```
% DL-SCH parameters
codeRate = 490/1024;
NHARQProcesses = 16; % Number of parallel HARQ processes to use
rvSeq = [0 2 3 1];
```

Create the DL-SCH encoder and decoder objects. To use multiple processes, set the `MultipleHARQProcesses` property to `true` for both objects. To enable retransmissions for multiple HARQ processes, the encoder buffers the input bits. The decoder needs a similar mechanism to enable soft combining of retransmissions for each HARQ process.

```
% Create DL-SCH encoder object
encodeDLSCH = nrDLSCH;
encodeDLSCH.MultipleHARQProcesses = true;
encodeDLSCH.TargetCodeRate = codeRate;

% Create DL-SCH decoder object
decodeDLSCH = nrDLSCHDecoder;
decodeDLSCH.MultipleHARQProcesses = true;
decodeDLSCH.TargetCodeRate = codeRate;
decodeDLSCH.LDPCDecodingAlgorithm = "Normalized min-sum";
decodeDLSCH.MaximumLDPCIterationCount = 6;
```

The DL-SCH encoder and decoder objects can model up to 16 HARQ processes. The encoder and decoder objects use the `HARQprocessID` property of the HARQ entity object to identify the active HARQ process when performing any of these operations.

- Setting new transport block to transmit
- Encoding data
- Resetting soft buffers
- Decoding data

Carrier and PDSCH Configuration

Specify the carrier and PDSCH parameters. These parameters are used for PDSCH encoding and decoding and for calculating the transport block size.

Create a carrier object, specifying the subcarrier spacing (SCS) and the bandwidth (BW).

```
% Numerology
SCS = 15;                          % SCS: 15, 30, 60, 120 or 240 (kHz)
NRB = 52;                          % BW in number of RBs (52 RBs at 15 kHz SCS for 10 MHz BW)

carrier = nrCarrierConfig;
carrier.NSizeGrid = NRB;
carrier.SubcarrierSpacing = SCS;
carrier.CyclicPrefix = "Normal";   % "Normal" or "Extended"
```

Create a PDSCH configuration object. The PDSCH parameters determine the available bit capacity and the transport block size.

```
modulation = "16QAM";              % Modulation scheme
```

```
pdsch = nrPDSCHConfig;
pdsch.Modulation = modulation;
pdsch.PRBSet = 0:NRB-1;              % Assume full band allocation
pdsch.NumLayers = 1;                 % Assume only one layer and one codeword
```

HARQ Management

Create a HARQ entity object to manage the HARQ processes. For each HARQ processes, the object stores these elements:

- HARQ ID number.
- RV.
- Transmission number, which indicates how many times a certain transport block has been transmitted.
- Flag to indicate whether new data is required. New data is required when a transport block is received successfully or if a sequence timeout has occurred (all RV transmissions have failed).
- Flag to indicate whether a sequence timeout has occurred (all RV transmissions have failed).

```
harqEntity = HARQEntity(0:NHARQProcesses-1,rvSeq,pdsch.NumCodewords);
```

The HARQ entity is used to manage the buffers in the DL-SCH encoder and decoder.

BER Simulation

Loop over a number of transport blocks. For each transport block:

- Calculate the transport block size in number of bits.
- Generate new data block or reset buffers in the decoder.
- Apply DL-SCH encoding.
- Modulate bits to symbols.
- Apply AWGN.
- Demodulate soft bits (symbols to soft bits).
- Decode the DL-SCH.
- Update the HARQ processes.

```
% Initialize loop variables
noiseVar = 1./(10.^(SNRdB/10)); % Noise variance
numBlkErr = 0;                  % Number of block errors
numRxBits = [];                 % Number of successfully received bits per transmission
txedTrBlkSizes = [];            % Number of transmitted info bits per transmission

for nTrBlk = 1:noTransportBlocks
    % A transport block or transmission time interval (TTI) corresponds to
    % one slot
    carrier.NSlot = carrier.NSlot+1;
```

Transport Block Size Calculation

Calculate the transport block size.

```
    % Generate PDSCH indices info, which is used to calculate the transport
    % block size
    [~,pdschInfo] = nrPDSCHIndices(carrier,pdsch);
```

```
% Calculate transport block sizes
Xoh_PDSCH = 0;
trBlkSizes = nrTBS(pdsch.Modulation,pdsch.NumLayers,numel(pdsch.PRBSet),pdschInfo.NREPerPRB,
```

Because the PDSCH capacity in bits, `pdsch.G`, is dynamically determined, the actual code rate might not be exactly equal to the target code rate specified by the `TargetCodeRate` property of the `encodeDLSCH` object.

HARQ Processing (Buffer Management)

This section explains the buffer management in the encoder and decoder.

- DL-SCH encoder buffers: Generate a new transport block if new data is required for the active HARQ process. Store the transport block in the corresponding buffer. If no new data is required, the buffered bits in the DL-SCH encoder are used for retransmission.

- DL-SCH decoder buffers: The soft buffers in the receiver store previously received versions of the same transport block. These buffers are cleared automatically upon successful reception (no CRC error). However, if the RV sequence ends without successful decoding, the buffers must be flushed manually by calling the `resetSoftBuffer` object function.

```
% Get new transport blocks and flush decoder soft buffer, as required
for cwIdx = 1:pdsch.NumCodewords
    if harqEntity.NewData(cwIdx)
        % Create and store a new transport block for transmission
        trBlk = randi([0 1],trBlkSizes(cwIdx),1);
        setTransportBlock(encodeDLSCH,trBlk,cwIdx-1,harqEntity.HARQProcessID);

        % If the previous RV sequence ends without successful decoding,
        % flush the soft buffer explicitly
        if harqEntity.SequenceTimeout(cwIdx)
            resetSoftBuffer(decodeDLSCH,cwIdx-1,harqEntity.HARQProcessID);
        end
    end
end
```

DL-SCH Encoding

Encode the DL-SCH transport blocks.

```
codedTrBlock = encodeDLSCH(pdsch.Modulation,pdsch.NumLayers,pdschInfo.G, ...
    harqEntity.RedundancyVersion,harqEntity.HARQProcessID);
```

PDSCH Encoding

Generate the PDSCH symbols.

```
modOut = nrPDSCH(carrier,pdsch,codedTrBlock);
```

AWGN Channel

Add white Gaussian noise.

```
rxSig = awgn(modOut,SNRdB);
```

PDSCH Demodulation

Soft demodulate the received symbols.

```
    rxLLR = nrPDSCHDecode(carrier,pdsch,rxSig,noiseVar);
```

DL-SCH Decoding

Apply DL-SCH decoding.

```
    decodeDLSCH.TransportBlockLength = trBlkSizes;
    [decbits,blkerr] = decodeDLSCH(rxLLR,pdsch.Modulation,pdsch.NumLayers, ...
        harqEntity.RedundancyVersion,harqEntity.HARQProcessID);
```

Results

Store the results to calculate the BLER.

```
    % Store values to calculate throughput (only for active transport blocks)
    if(any(trBlkSizes ~= 0))
        numRxBits = [numRxBits trBlkSizes.*(1-blkerr)];
        txedTrBlkSizes = [txedTrBlkSizes trBlkSizes];
    end

    if blkerr
        numBlkErr = numBlkErr + 1;
    end
```

HARQ Process Update

Update the current HARQ process with the CRC error, and then advance to the next process. This step updates the information related to the active HARQ process in the HARQ entity.

```
    statusReport = updateAndAdvance(harqEntity,blkerr,trBlkSizes,pdschInfo.G);
```

Display information about the current decoding attempt.

```
    disp("Slot "+(nTrBlk)+". "+statusReport);

end % for nTrBlk = 1:noTransportBlocks

Slot 1. HARQ Proc 0: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 2. HARQ Proc 1: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 3. HARQ Proc 2: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 4. HARQ Proc 3: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 5. HARQ Proc 4: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 6. HARQ Proc 5: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 7. HARQ Proc 6: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 8. HARQ Proc 7: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 9. HARQ Proc 8: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 10. HARQ Proc 9: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 11. HARQ Proc 10: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 12. HARQ Proc 11: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 13. HARQ Proc 12: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 14. HARQ Proc 13: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 15. HARQ Proc 14: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 16. HARQ Proc 15: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 17. HARQ Proc 0: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 18. HARQ Proc 1: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 19. HARQ Proc 2: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 20. HARQ Proc 3: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 21. HARQ Proc 4: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 22. HARQ Proc 5: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
```

```
Slot 23. HARQ Proc 6: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 24. HARQ Proc 7: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 25. HARQ Proc 8: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 26. HARQ Proc 9: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 27. HARQ Proc 10: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 28. HARQ Proc 11: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 29. HARQ Proc 12: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 30. HARQ Proc 13: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 31. HARQ Proc 14: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 32. HARQ Proc 15: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 33. HARQ Proc 0: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 34. HARQ Proc 1: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 35. HARQ Proc 2: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 36. HARQ Proc 3: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 37. HARQ Proc 4: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 38. HARQ Proc 5: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 39. HARQ Proc 6: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 40. HARQ Proc 7: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 41. HARQ Proc 8: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 42. HARQ Proc 9: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 43. HARQ Proc 10: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 44. HARQ Proc 11: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 45. HARQ Proc 12: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 46. HARQ Proc 13: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 47. HARQ Proc 14: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 48. HARQ Proc 15: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 49. HARQ Proc 0: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 50. HARQ Proc 1: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 51. HARQ Proc 2: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 52. HARQ Proc 3: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 53. HARQ Proc 4: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 54. HARQ Proc 5: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 55. HARQ Proc 6: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 56. HARQ Proc 7: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 57. HARQ Proc 8: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 58. HARQ Proc 9: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 59. HARQ Proc 10: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 60. HARQ Proc 11: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 61. HARQ Proc 12: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 62. HARQ Proc 13: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 63. HARQ Proc 14: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 64. HARQ Proc 15: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 65. HARQ Proc 0: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 66. HARQ Proc 1: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 67. HARQ Proc 2: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 68. HARQ Proc 3: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 69. HARQ Proc 4: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 70. HARQ Proc 5: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 71. HARQ Proc 6: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 72. HARQ Proc 7: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 73. HARQ Proc 8: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 74. HARQ Proc 9: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 75. HARQ Proc 10: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 76. HARQ Proc 11: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 77. HARQ Proc 12: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 78. HARQ Proc 13: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 79. HARQ Proc 14: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 80. HARQ Proc 15: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
```

```
Slot 81. HARQ Proc 0: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 82. HARQ Proc 1: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 83. HARQ Proc 2: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 84. HARQ Proc 3: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 85. HARQ Proc 4: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 86. HARQ Proc 5: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 87. HARQ Proc 6: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 88. HARQ Proc 7: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 89. HARQ Proc 8: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 90. HARQ Proc 9: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 91. HARQ Proc 10: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 92. HARQ Proc 11: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 93. HARQ Proc 12: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 94. HARQ Proc 13: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 95. HARQ Proc 14: CW0: Retransmission #1 passed (RV=2,CR=0.481509).
Slot 96. HARQ Proc 15: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 97. HARQ Proc 0: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 98. HARQ Proc 1: CW0: Initial transmission failed (RV=0,CR=0.481509).
Slot 99. HARQ Proc 2: CW0: Initial transmission passed (RV=0,CR=0.481509).
Slot 100. HARQ Proc 3: CW0: Initial transmission failed (RV=0,CR=0.481509).
```

BLER Results

Calculate the BLER and the throughput (percentage of successfully received transport blocks). To provide statistically meaningful results, run this simulation for many transport blocks.

```
maxThroughput = sum(txedTrBlkSizes); % Maximum possible throughput
totalNumRxBits = sum(numRxBits,2);   % Number of successfully received bits

disp("Block Error Rate: "+string(numBlkErr/noTransportBlocks))

Block Error Rate: 0.42

disp("Throughput: " + string(totalNumRxBits*100/maxThroughput) + "%")

Throughput: 58%
```

See Also

Related Examples

- "Map 5G Physical Channels and Signals to the Resource Grid" on page 2-49
- "DL-SCH and PDSCH Transmit and Receive Processing Chain" on page 2-59

Map 5G Physical Channels and Signals to the Resource Grid

This example shows how to generate and map 5G New Radio (NR) physical channels and signals to the resource grid using 5G Toolbox ™ features.

Introduction

This figure shows the link elements that are modeled in this example in the context of a 5G downlink link. These elements are:

- Generation of a physical downlink shared channel (PDSCH) and its demodulation reference signal (DM-RS)
- MIMO precoding and mapping of the PDSCH and PDSCH DM-RS to the resource grid
- OFDM modulation
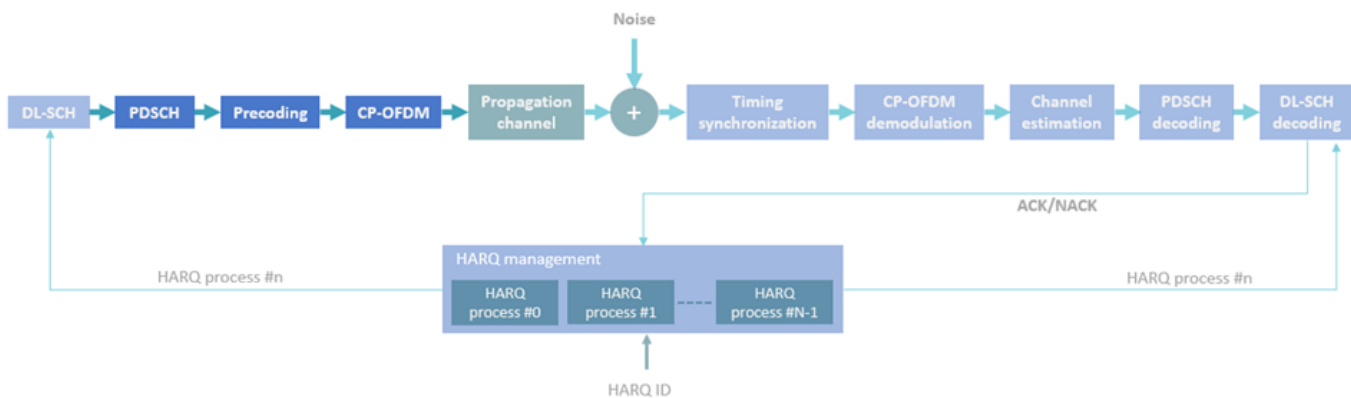
Carrier Configuration

Specify the number of transmit antennas and create a carrier configuration object. This object controls the size of the resource grid. For simplicity, use the default carrier configuration object.

```
nTxAnts = 4;
carrier = nrCarrierConfig

carrier =
  nrCarrierConfig with properties:

             NCellID: 1
    SubcarrierSpacing: 15
         CyclicPrefix: 'normal'
            NSizeGrid: 52
           NStartGrid: 0
                NSlot: 0
               NFrame: 0

  Read-only properties:
       SymbolsPerSlot: 14
     SlotsPerSubframe: 1
        SlotsPerFrame: 10
```

PDSCH and PDSCH DM-RS Configuration

Create a PDSCH configuration object. This object specifies PDSCH-related parameters. Specify 16-QAM modulation, two layers, and full band allocation. This configuration maps the PDSCH into a bandwidth part (BWP) of equal size to the carrier. You can also use this object to specify other time-allocation parameters and DM-RS settings.

```
pdsch = nrPDSCHConfig;
pdsch.Modulation = "16QAM";
pdsch.NumLayers = 2;
pdsch.PRBSet = 0:carrier.NSizeGrid-1; % Full band allocation
```

Display the PDSCH and PDSCH DM-RS parameters.

```
pdsch
```

```
pdsch =
  nrPDSCHConfig with properties:

                NSizeBWP: []
               NStartBWP: []
             ReservedPRB: {[1x1 nrPDSCHReservedConfig]}
              ReservedRE: []
              Modulation: '16QAM'
               NumLayers: 2
             MappingType: 'A'
         SymbolAllocation: [0 14]
                  PRBSet: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... ]
      VRBToPRBInterleaving: 0
           VRBBundleSize: 2
                     NID: []
                    RNTI: 1
                    DMRS: [1x1 nrPDSCHDMRSConfig]
              EnablePTRS: 0
                    PTRS: [1x1 nrPDSCHPTRSConfig]

   Read-only properties:
           NumCodewords: 1
```

```
pdsch.DMRS
```

```
ans =
  nrPDSCHDMRSConfig with properties:

       DMRSConfigurationType: 1
          DMRSReferencePoint: 'CRB0'
            DMRSTypeAPosition: 2
      DMRSAdditionalPosition: 0
                  DMRSLength: 1
             CustomSymbolSet: []
                 DMRSPortSet: []
                    NIDNSCID: []
                       NSCID: 0
     NumCDMGroupsWithoutData: 2
             DMRSDownlinkR16: 0

   Read-only properties:
```

```
                   CDMGroups: [0 0]
                  DeltaShifts: [0 0]
             FrequencyWeights: [2x2 double]
                  TimeWeights: [2x2 double]
      DMRSSubcarrierLocations: [6x2 double]
                   CDMLengths: [2 1]
```

PDSCH Generation

Generate indices to map the PDSCH to the grid.

```
[pdschIndices,pdschInfo] = nrPDSCHIndices(carrier,pdsch);
```

Generate and map random PDSCH bits to PDSCH symbols. The input argument `pdschInfo.G` specifies the bit capacity of the PDSCH, which is the length of the codeword from the channel coding stages. `pdschInfo.G` takes into account the resource elements (REs) available for PDSCH transmission. For simplicity, this example does not include downlink shared channel (DL-SCH) modeling.

```
pdschBits = randi([0 1],pdschInfo.G,1);
```

Generate PDSCH symbols. The PDSCH symbols are stored in a matrix of size N_S-by-ν, where N_S is the number of symbols and ν is the number of layers.

```
pdschSymbols = nrPDSCH(carrier,pdsch,pdschBits);
size(pdschSymbols)
```

ans = *1×2*

```
      8112           2
```

PDSCH DM-RS Generation

Generate DM-RS symbols and indices.

```
dmrsSymbols = nrPDSCHDMRS(carrier,pdsch);
dmrsIndices = nrPDSCHDMRSIndices(carrier,pdsch);
```

Display the constellation plot with the PDSCH and the PDSCH DM-RS symbols.

```
plot(pdschSymbols(:),"o");hold on
plot(dmrsSymbols(:),"xr");hold off
title("PDSCH and PDSCH DM-RS Symbols");xlabel("In-Phase Amplitude");ylabel("Quadrature Amplitude"
legend("PDSCH","PDSCH DM-RS")
```

PDSCH and PDSCH DM-RS Symbols
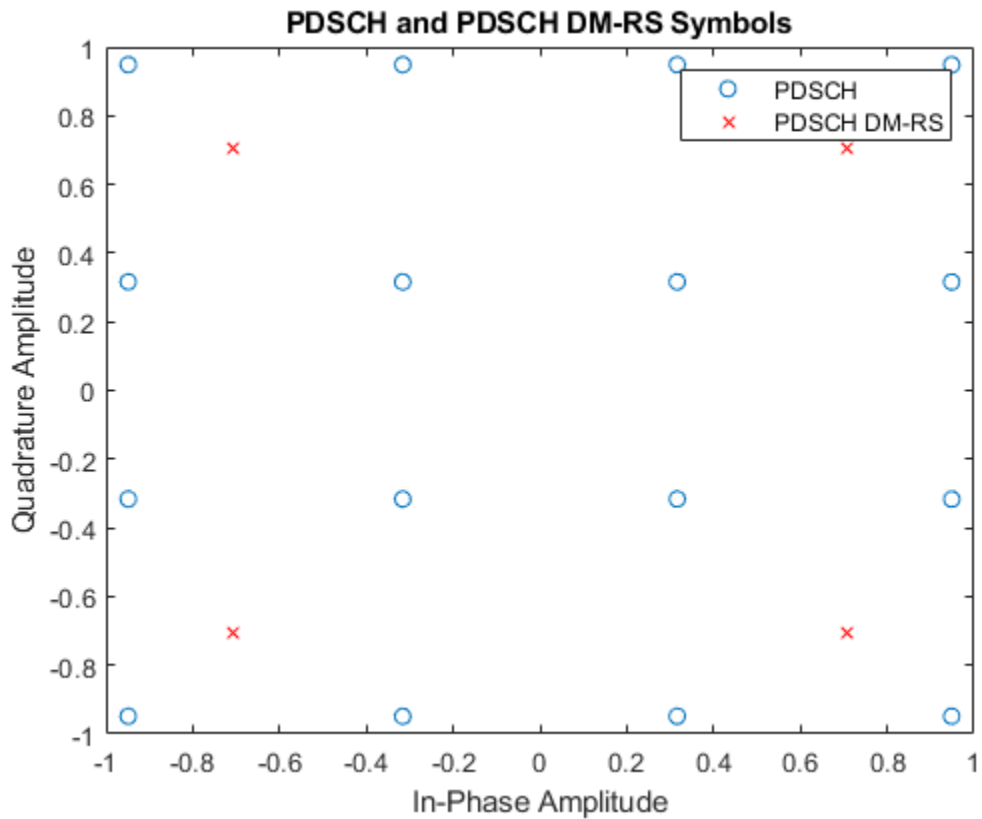
MIMO Precoding and Mapping to the Resource Grid

Apply precoding. Channel measurements determine the precoding weights (also referred to as *beamforming weights*). However, this example does not model the propagation channel. This example assumes that the precoding weights are known.

```
% Precoding weights
W = fft(eye(nTxAnts))/sqrt(nTxAnts);                  % Unitary precoding matrix
w = W(1:pdsch.NumLayers,:)/sqrt(pdsch.NumLayers);     % Normalize by number of layers
```

The precoding matrix, w, must be a matrix of size ν-by-N_{tx}, where ν is the number of layers and N_{tx} is the number of transmit antennas.

```matlab
size(pdschSymbols)
```
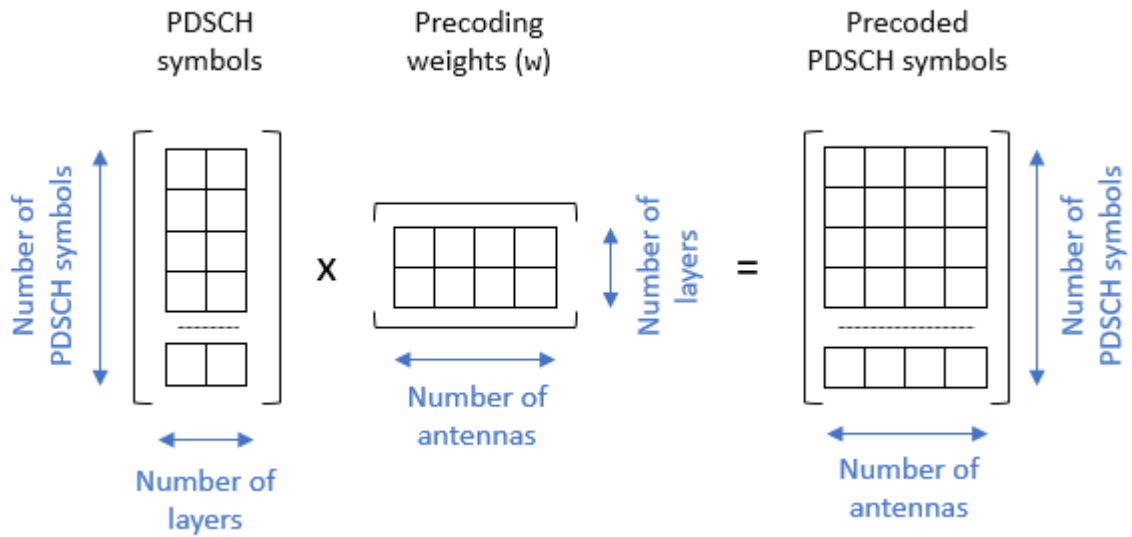
ans = *1×2*

      8112   2

```matlab
size(w)
```

ans = *1×2*

    2   4

Precode the PDSCH symbols.

```matlab
pdschSymbolsPrecoded = pdschSymbols*w;
```

The number of rows in the `pdschSymbolsPrecoded` matrix corresponds to the number of PDSCH symbols and the number of columns corresponds to the number of antennas.

```matlab
size(pdschSymbolsPrecoded)
```

ans = *1×2*

      8112   4

Generate an empty resource grid. This grid spans one slot.

```matlab
pdschGrid = nrResourceGrid(carrier,nTxAnts);
```

When you map the PDSCH symbols to the resource grid, take into account that the PDSCH indices generated by the `nrPDSCHIndices` function refer to layers and not antennas. This format can be useful when you map PDSCH symbols directly to layers. In this case, the resulting resource grids are not precoded.
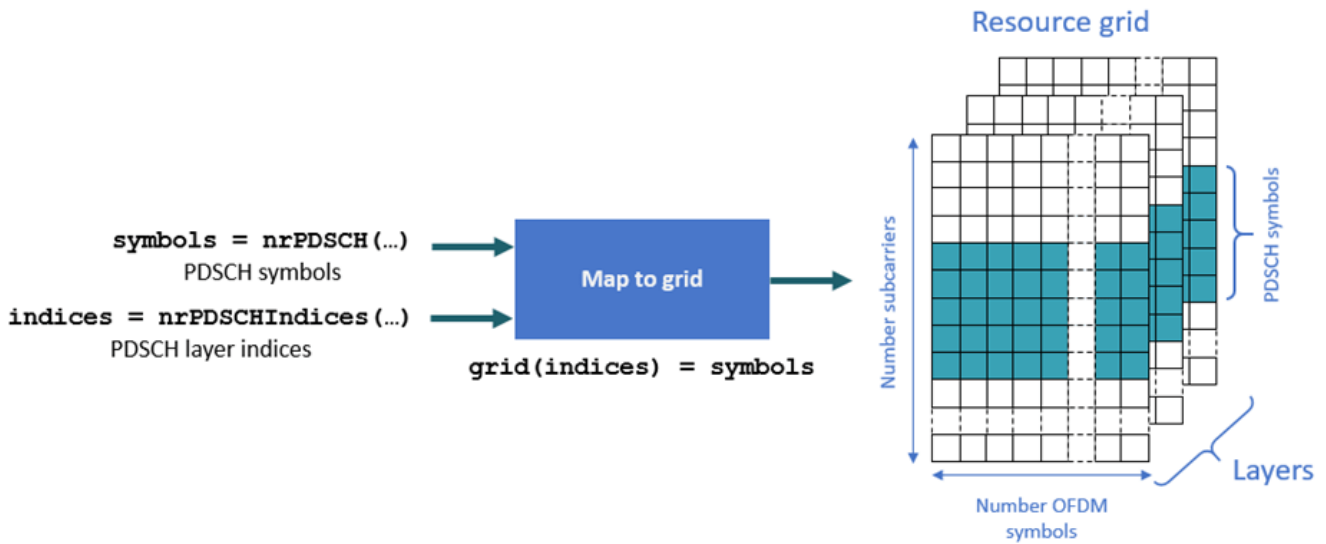
Because this example applies precoding to the PDSCH symbols before mapping to the resource grid, the precoded PDSCH symbols are mapped to antennas and not layers. To convert layer indices to antenna indices, use the `nrExtractResources` function.
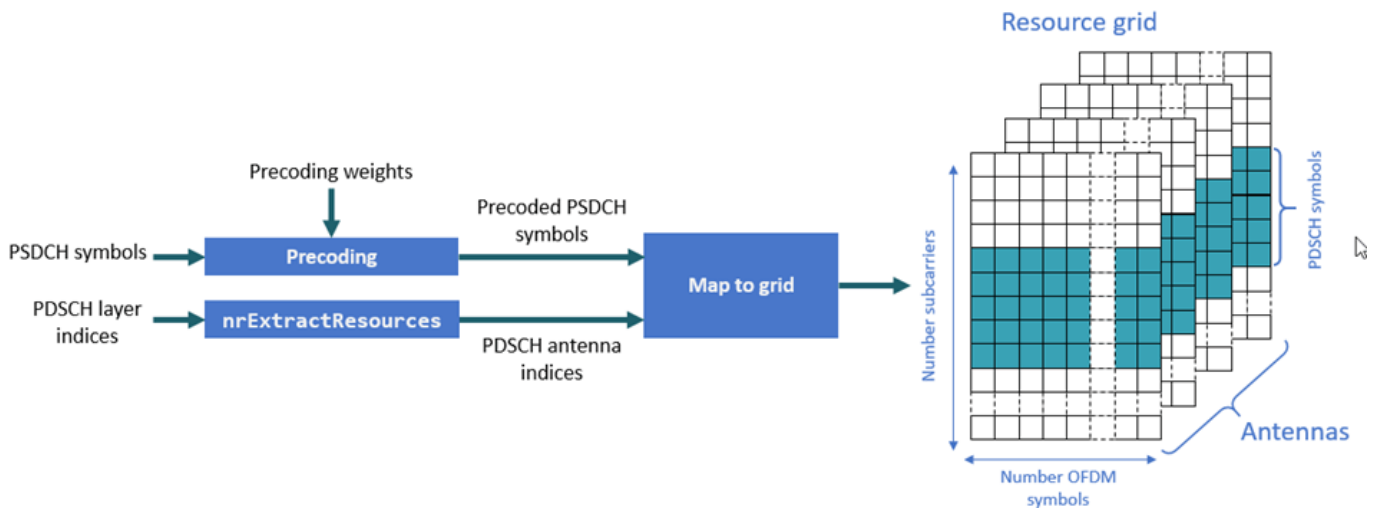

```
[~,pdschAntIndices] = nrExtractResources(pdschIndices,pdschGrid);
pdschGrid(pdschAntIndices) = pdschSymbolsPrecoded;
```

Display the resource grid for the first antenna. The blue gap is left for the DM-RS.

```
imagesc([0 carrier.SymbolsPerSlot-1],[0 carrier.NSizeGrid*12-1],abs(pdschGrid(:,:,1)));
axis xy;title("Resource Grid (First Antenna) - PDSCH");xlabel("OFDM Symbol");ylabel("Subcarrier")
```

Resource Grid (First Antenna) - PDSCH
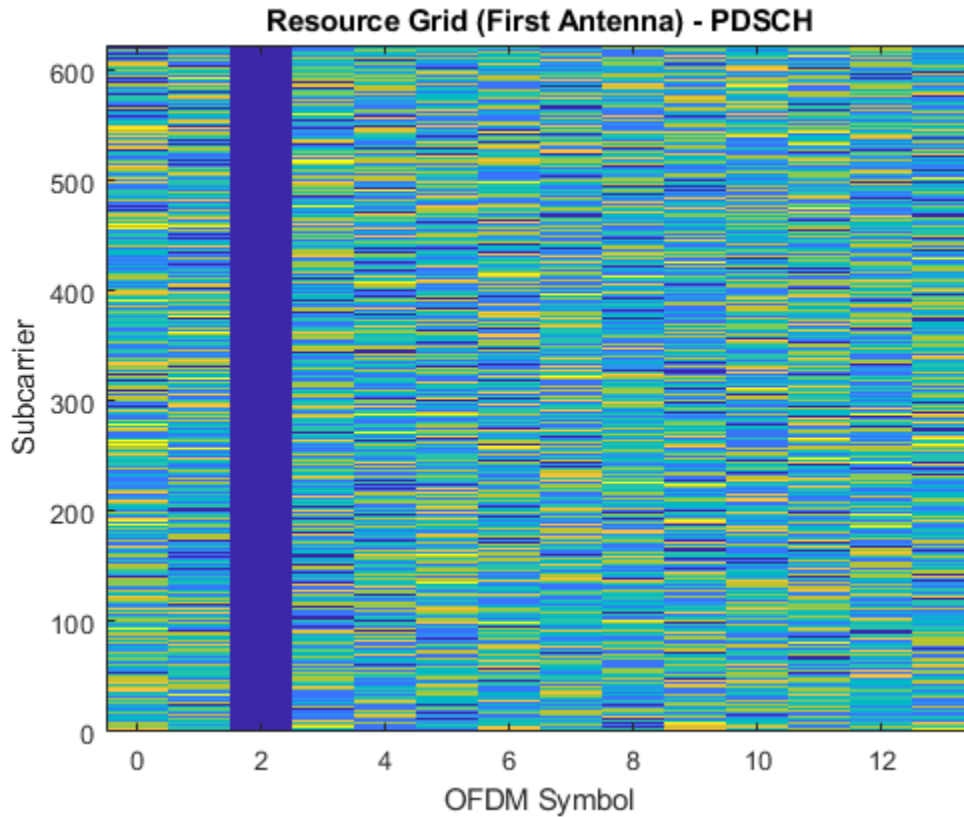
Precode and map the DM-RS symbols to the grid. Similar to the PDSCH indices, the DM-RS indices refer to layers. To convert these layers to multiantenna indices, use the `nrExtractResources` function again.

```
% PDSCH DM-RS precoding and mapping
for p = 1:size(dmrsSymbols,2)
    [~,dmrsAntIndices] = nrExtractResources(dmrsIndices(:,p),pdschGrid);
    pdschGrid(dmrsAntIndices) = pdschGrid(dmrsAntIndices) + dmrsSymbols(:,p)*w(p,:);
end
```

Display the resource grid for the first antenna.

```
imagesc([0 carrier.SymbolsPerSlot-1],[0 carrier.NSizeGrid*12-1],abs(pdschGrid(:,:,1)));
axis xy;title("Resource Grid (First Antenna) - PDSCH and PDSCH DM-RS");
    xlabel("OFDM Symbol");ylabel("Subcarrier")
```
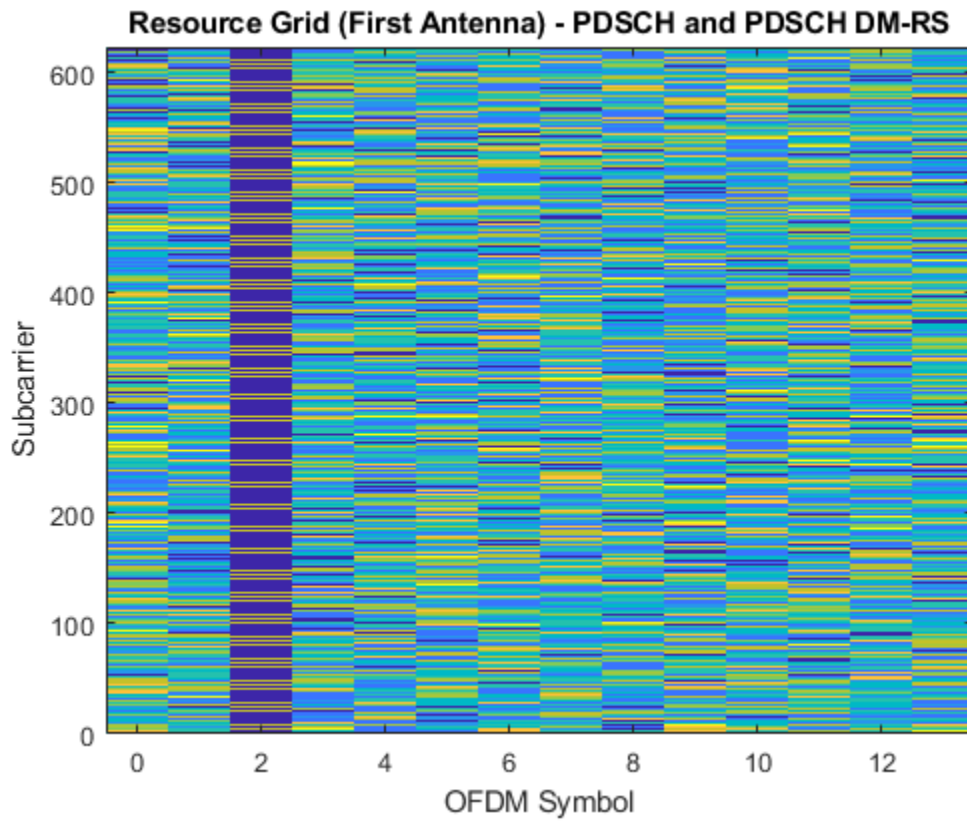
Resource Grid (First Antenna) - PDSCH and PDSCH DM-RS

Display a single resource block (RB) from the resource grid. This view zooms into a single RB and provides a detailed view of the RE contents.

```
imagesc(abs(pdschGrid(1:12,:,1)));view(2)
axis xy;title("Resource Block - PDSCH and PDSCH DM-RS");ylabel("Subcarrier");xlabel("OFDM Symbol"
```

Resource Block - PDSCH and PDSCH DM-RS
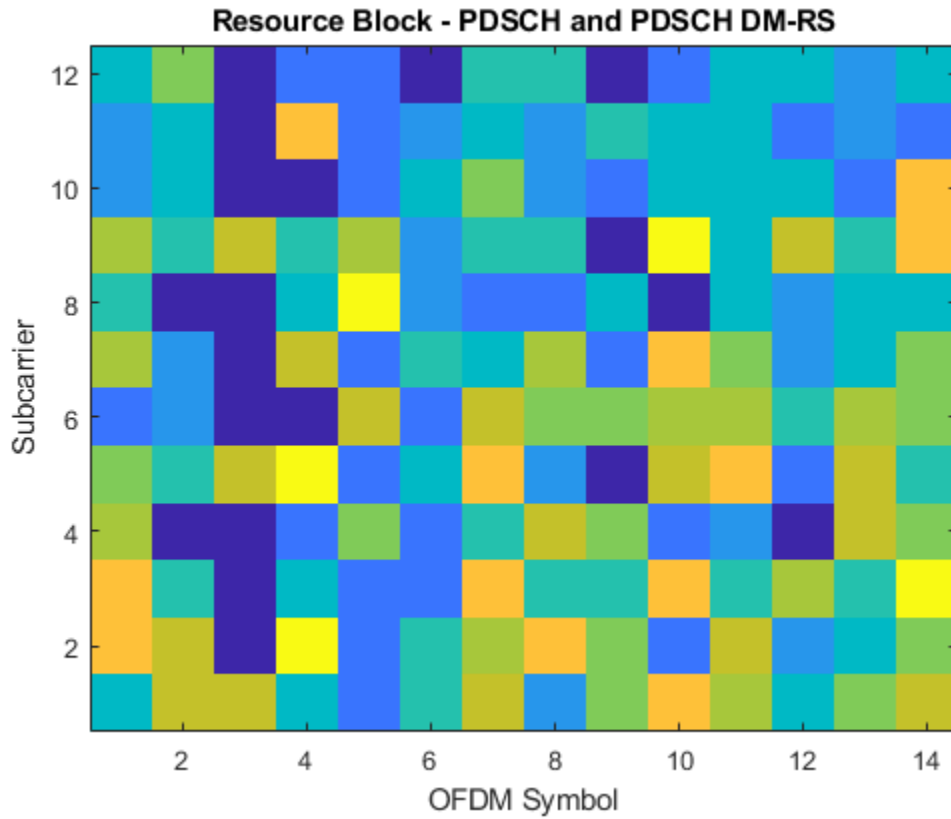
OFDM Modulation

OFDM-modulate the resource grid and display the time-domain waveform for the first antenna.

```
[txWaveform,waveformInfo] = nrOFDMModulate(carrier,pdschGrid);
plot(abs(txWaveform(:,1)));title("Time Domain Waveform (First Antenna)");xlabel("Sample Number")
```

Time Domain Waveform (First Antenna)
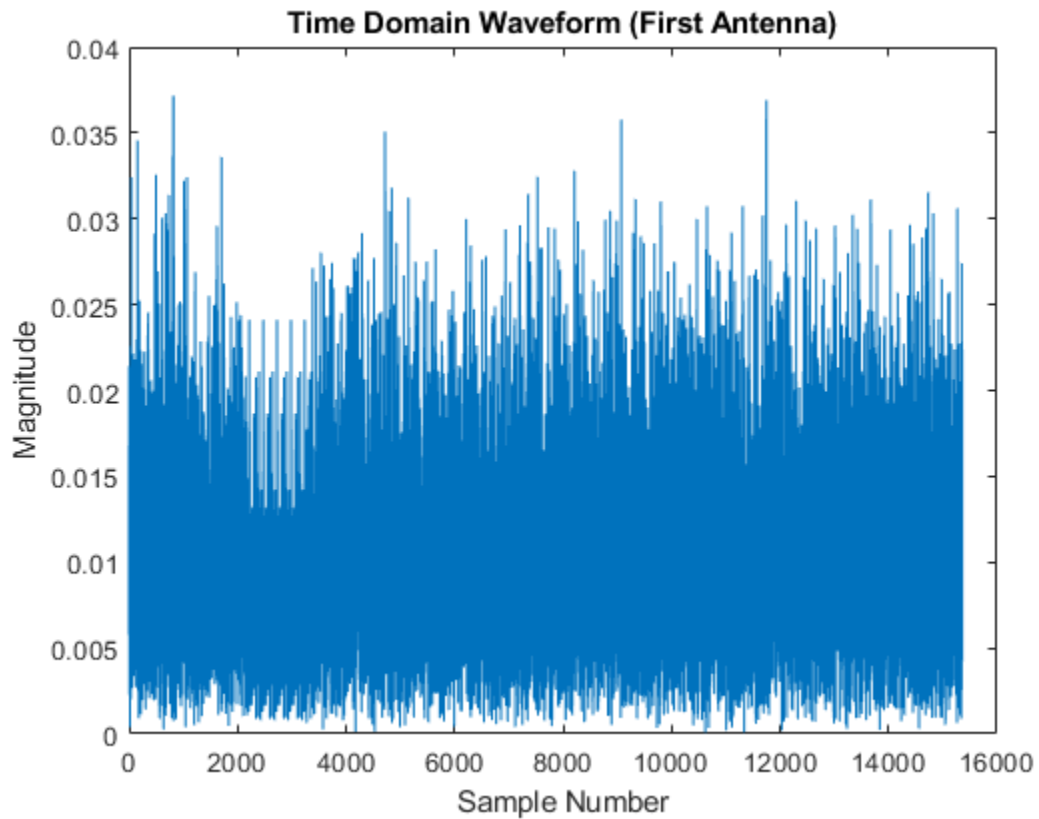
The `waveformInfo` output contains information about the time-domain waveform, such as the sampling rate.

`waveformInfo`

```
waveformInfo = struct with fields:
                   Nfft: 1024
             SampleRate: 15360000
    CyclicPrefixLengths: [80 72 72 72 72 72 72 80 72 72 72 72 72 72]
          SymbolLengths: [1104 1096 1096 1096 1096 1096 1096 1104 1096 ... ]
              Windowing: 36
           SymbolPhases: [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
         SymbolsPerSlot: 14
       SlotsPerSubframe: 1
          SlotsPerFrame: 10
```

See Also

Related Examples

- "Model 5G NR Transport Channels with HARQ" on page 2-41
- "DL-SCH and PDSCH Transmit and Receive Processing Chain" on page 2-59

DL-SCH and PDSCH Transmit and Receive Processing Chain

This example shows how to use 5G Toolbox™ features to model a 5G NR physical downlink shared channel (PDSCH) link, including all of the steps from transport block generation to bit decoding at the receiver end.

Introduction

This diagram shows the downlink shared channel (DL-SCH) and PDSCH transmit and receive processing chain.
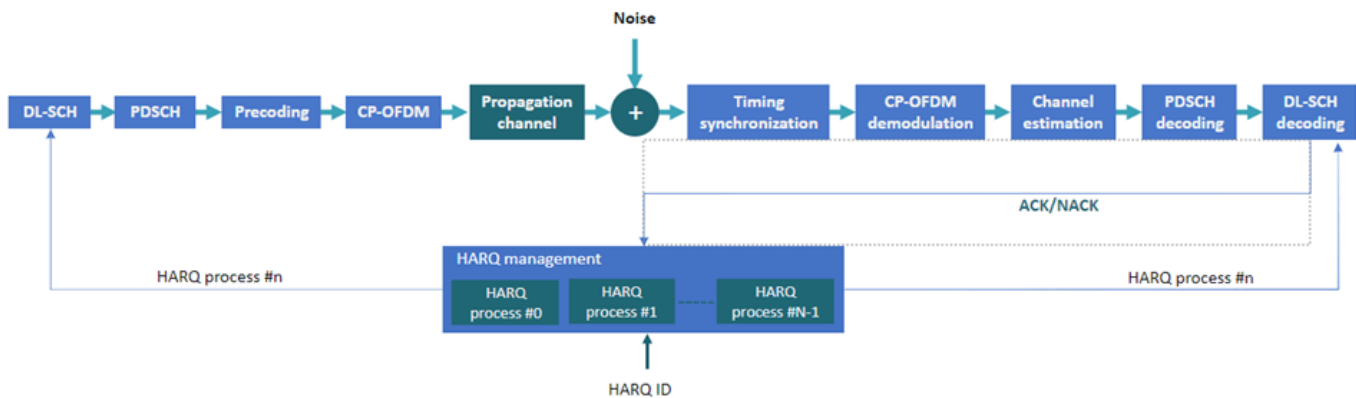
This example shows how to model these elements of a link-level simulation.

- DL-SCH encoding
- Hybrid ARQ (HARQ) management
- PDSCH encoding
- Multiple-input multiple-output (MIMO) precoding
- OFDM modulation
- Propagation channel and noise addition
- Timing synchronization
- OFDM demodulation
- Channel estimation and equalization
- PDSCH decoding
- DL-SCH decoding

For an example of how to use link-level simulation to measure throughput, see "NR PDSCH Throughput".

Simulation Parameters

Specify the signal-to-noise ratio (SNR), number of slots to simulate, and perfect channel estimation flag. To learn more about the SNR definition used in this example, see "SNR Definition Used in Link Simulations".

```
SNRdB = 10;                  % SNR in dB
totalNoSlots = 20;           % Number of slots to simulate
```

```
perfectEstimation = false; % Perfect synchronization and channel estimation
rng("default");            % Set default random number generator for repeatability
```

Carrier Configuration

Create a carrier configuration object. This object controls the numerology, such as, the subcarrier spacing, bandwidth, and cyclic prefix (CP) length. This example uses the default set of properties.

```
carrier = nrCarrierConfig

carrier =
  nrCarrierConfig with properties:

              NCellID: 1
    SubcarrierSpacing: 15
         CyclicPrefix: 'normal'
            NSizeGrid: 52
           NStartGrid: 0
                NSlot: 0
               NFrame: 0

  Read-only properties:
        SymbolsPerSlot: 14
      SlotsPerSubframe: 1
         SlotsPerFrame: 10
```

PDSCH and DM-RS Configuration

Create a PDSCH configuration object. Specify the modulation scheme (16-QAM) and the number of layers (2). Allocate all resource blocks (RBs) to the PDSCH (full band allocation). You can also specify other time-allocation parameters and demodulation reference signal (DM-RS) settings in this object.

```
pdsch = nrPDSCHConfig;
pdsch.Modulation = "16QAM";
pdsch.NumLayers = 2;
pdsch.PRBSet = 0:carrier.NSizeGrid-1;      % Full band allocation
```

Display the PDSCH parameters.

```
pdsch

pdsch =
  nrPDSCHConfig with properties:

                NSizeBWP: []
               NStartBWP: []
             ReservedPRB: {[1x1 nrPDSCHReservedConfig]}
              ReservedRE: []
              Modulation: '16QAM'
               NumLayers: 2
             MappingType: 'A'
        SymbolAllocation: [0 14]
                  PRBSet: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... ]
    VRBToPRBInterleaving: 0
           VRBBundleSize: 2
                     NID: []
                    RNTI: 1
                    DMRS: [1x1 nrPDSCHDMRSConfig]
```

```
          EnablePTRS: 0
                PTRS: [1x1 nrPDSCHPTRSConfig]

  Read-only properties:
          NumCodewords: 1
```

Set the DM-RS parameters. To improve channel estimation, add an additional DM-RS position.

```
pdsch.DMRS.DMRSAdditionalPosition = 1;
```

Set the DM-RS configuration type and the DM-RS length, which determines the number of orthogonal DM-RS sequences or DM-RS ports.

- `DMRSConfigurationType = 1` supports up to 4 DM-RS ports when `DMRSLength = 1`.
- `DMRSConfigurationType = 1` supports up to 8 DM-RS ports when `DMRSLength = 2`.
- `DMRSConfigurationType = 2` supports up to 6 DM-RS ports when `DMRSLength = 1`. This is designed for multi-user MIMO (MU-MIMO).
- `DMRSConfigurationType = 2` supports up to 12 DM-RS ports when `DMRSLength = 2`. This is designed for MU-MIMO.

The maximum number of layers must be less than or equal to the number of DM-RS ports.

```
pdsch.DMRS.DMRSConfigurationType = 1;
pdsch.DMRS.DMRSLength = 2;
pdsch.DMRS                              % Display DM-RS properties

ans =
  nrPDSCHDMRSConfig with properties:

     DMRSConfigurationType: 1
        DMRSReferencePoint: 'CRB0'
          DMRSTypeAPosition: 2
     DMRSAdditionalPosition: 1
                DMRSLength: 2
            CustomSymbolSet: []
               DMRSPortSet: []
                  NIDNSCID: []
                     NSCID: 0
     NumCDMGroupsWithoutData: 2
            DMRSDownlinkR16: 0

  Read-only properties:
                  CDMGroups: [0 0]
                 DeltaShifts: [0 0]
            FrequencyWeights: [2x2 double]
                 TimeWeights: [2x2 double]
     DMRSSubcarrierLocations: [6x2 double]
                  CDMLengths: [2 1]
```

DL-SCH Configuration

Specify the code rate, the number of HARQ processes, and the redundancy version (RV) sequence values. This sequence controls the RV retransmissions in case of error. To disable HARQ retransmissions, you can set `rvSeq` to a fixed value (for example, 0). For more information on how to

model transport channels with HARQ, see "Model 5G NR Transport Channels with HARQ" on page 2-41.

```
NHARQProcesses = 16;      % Number of parallel HARQ processes
rvSeq = [0 2 3 1];
```

Take into account the number of codewords when specifying the coding rate. The number of codewords is a read-only property of the PDSCH configuration object that depends on the number of layers.

- 1 codeword for up to 4 layers

- 2 codewords for more than 4 layers

```
% Coding rate
if pdsch.NumCodewords == 1
    codeRate = 490/1024;
else
    codeRate = [490 490]./1024;
end
```

Create the DL-SCH encoder and decoder objects. To use multiple processes, set the `MultipleHARQProcesses` property to `true` for both objects. You do not need to specify the number of HARQ processes. The DL-SCH encoder and decoder objects can model up to 16 HARQ processes. To identify the active HARQ process when performing operations with the DL-SCH encoder and decoder objects, use the `HARQprocessID` property of the HARQ entity object, defined in the next section.

```
% Create DL-SCH encoder object
encodeDLSCH = nrDLSCH;
encodeDLSCH.MultipleHARQProcesses = true;
encodeDLSCH.TargetCodeRate = codeRate;

% Create DLSCH decoder object
decodeDLSCH = nrDLSCHDecoder;
decodeDLSCH.MultipleHARQProcesses = true;
decodeDLSCH.TargetCodeRate = codeRate;
decodeDLSCH.LDPCDecodingAlgorithm = "Normalized min-sum";
decodeDLSCH.MaximumLDPCIterationCount = 6;
```

HARQ Management

Create a HARQ entity object to manage the HARQ processes and the DL-SCH encoder and decoder buffers. For each HARQ process, a HARQ entity stores these elements:

- HARQ ID number.

- RV.

- Transmission number, which indicates how many times a certain transport block has been transmitted.

- Flag to indicate whether new data is required. New data is required when a transport block is received successfully or if a sequence timeout has occurred (all RV transmissions have failed).

- Flag to indicate whether a sequence timeout has occurred (all RV transmissions have failed).

```
harqEntity = HARQEntity(0:NHARQProcesses-1,rvSeq,pdsch.NumCodewords);
```

Channel Configuration

Specify the number of transmit and receive antennas.

```
nTxAnts = 8;
nRxAnts = 8;

% Check that the number of layers is valid for the number of antennas
if pdsch.NumLayers > min(nTxAnts,nRxAnts)
    error("The number of layers ("+string(pdsch.NumLayers)+") must be smaller than min(nTxAnts,nF
end
```

Create a channel object.

```
channel = nrTDLChannel;
channel.DelayProfile = "TDL-C";
channel.NumTransmitAntennas = nTxAnts;
channel.NumReceiveAntennas = nRxAnts;
```

Set the channel sample rate to that of the OFDM signal. To obtain the sampling rate of the OFDM signal, use the `nrOFDMInfo` function.

```
ofdmInfo = nrOFDMInfo(carrier);
channel.SampleRate = ofdmInfo.SampleRate;
```

Transmission and Reception

Set up a loop to simulate the transmission and reception of slots. Create a
`comm.ConstellationDiagram` to display the constellation of the equalized signal.

```
constPlot = comm.ConstellationDiagram;                                        % Constellation c
constPlot.ReferenceConstellation = getConstellationRefPoints(pdsch.Modulation); % Reference const
constPlot.EnableMeasurements = 1;                                             % Enable EVM meas

% Initial timing offset
offset = 0;

estChannelGrid = getInitialChannelEstimate(channel,carrier);
newPrecodingWeight = getPrecodingMatrix(pdsch.PRBSet,pdsch.NumLayers,estChannelGrid);

for nSlot = 0:totalNoSlots-1
    % New slot
    carrier.NSlot = nSlot;
```

Calculate Transport Block Size

The transport block size is the number of bits to send to the channel coding stages. This value depends on the capacity of the PDSCH. To calculate the transport block size, use the `nrTBS` function.

```
    % Generate PDSCH indices info, which is needed to calculate the transport
    % block size
    [pdschIndices,pdschInfo] = nrPDSCHIndices(carrier,pdsch);

    % Calculate transport block sizes
    Xoh_PDSCH = 0;
    trBlkSizes = nrTBS(pdsch.Modulation,pdsch.NumLayers,numel(pdsch.PRBSet),pdschInfo.NREPerPRB,c
```

HARQ Processing (Buffer Management)

This section explains the buffer management in the encoder and decoder.

- DL-SCH encoder buffers: Generate a new transport block if new data is required for the active HARQ process. Store the transport block in the corresponding buffer. If new data is not required, the DL-SCH encoder uses its buffered bits for retransmission.

- DL-SCH decoder buffers: The soft buffers in the receiver store previously received versions of the same codeword. These buffers are cleared automatically upon successful reception (no CRC error). However, if the RV sequence ends without successful decoding, flush the buffers manually by using the `resetSoftBuffer` object function.

```
% Get new transport blocks and flush decoder soft buffer, as required
for cwIdx = 1:pdsch.NumCodewords
    if harqEntity.NewData(cwIdx)
        % Create and store a new transport block for transmission
        trBlk = randi([0 1],trBlkSizes(cwIdx),1);
        setTransportBlock(encodeDLSCH,trBlk,cwIdx-1,harqEntity.HARQProcessID);

        % If the previous RV sequence ends without successful
        % decoding, flush the soft buffer
        if harqEntity.SequenceTimeout(cwIdx)
            resetSoftBuffer(decodeDLSCH,cwIdx-1,harqEntity.HARQProcessID);
        end
    end
end
```

DL-SCH Encoding

Encode the transport blocks. The transport block is already stored in one of the internal soft buffers of the DL-SCH encoder object.

```
codedTrBlock = encodeDLSCH(pdsch.Modulation,pdsch.NumLayers,pdschInfo.G,harqEntity.Redundancy
```

PDSCH Modulation and MIMO Precoding

Generate PDSCH symbols from the coded transport blocks.

```
pdschSymbols = nrPDSCH(carrier,pdsch,codedTrBlock);
```

Get the precoding weights. This example assumes channel knowledge for precoding. (For an example of how to use the channel estimate at the receiver to calculate the weights used for the transmission in the next slot, see "NR PDSCH Throughput".)

```
precodingWeights = newPrecodingWeight;
```

Precode the PDSCH symbols.

```
pdschSymbolsPrecoded = pdschSymbols*precodingWeights;
```

PDSCH DM-RS Generation

Generate DM-RS symbols and indices.

```
dmrsSymbols = nrPDSCHDMRS(carrier,pdsch);
dmrsIndices = nrPDSCHDMRSIndices(carrier,pdsch);
```

Mapping to Resource Grid

Generate an empty resource grid. This grid represents a slot.

```
pdschGrid = nrResourceGrid(carrier,nTxAnts);
```

The `nrPDSCHIndices` function generates indices that refer to layers and not antennas. This format is useful when mapping PDSCH symbols directly to layers. In this case, the resulting resource grids are not precoded. This figure shows the mapping process of the PDSCH symbols to as many resource grids as layers.
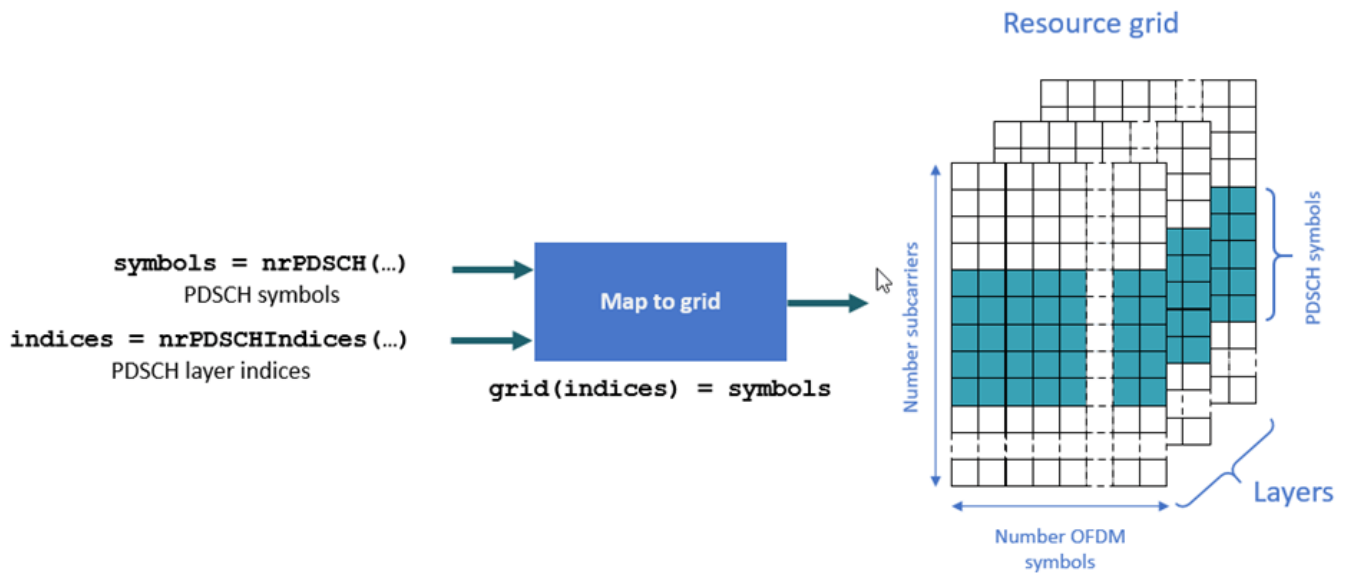
Because this example applies MIMO precoding to the PDSCH symbols before mapping them to the resource grids, the MIMO-precoded PDSCH symbols refer to antennas and not layers. To convert layer indices to antenna indices, use the `nrExtractResources` function. This figure shows the mapping process of MIMO-precoded symbols to as many resource grids as transmit antennas.
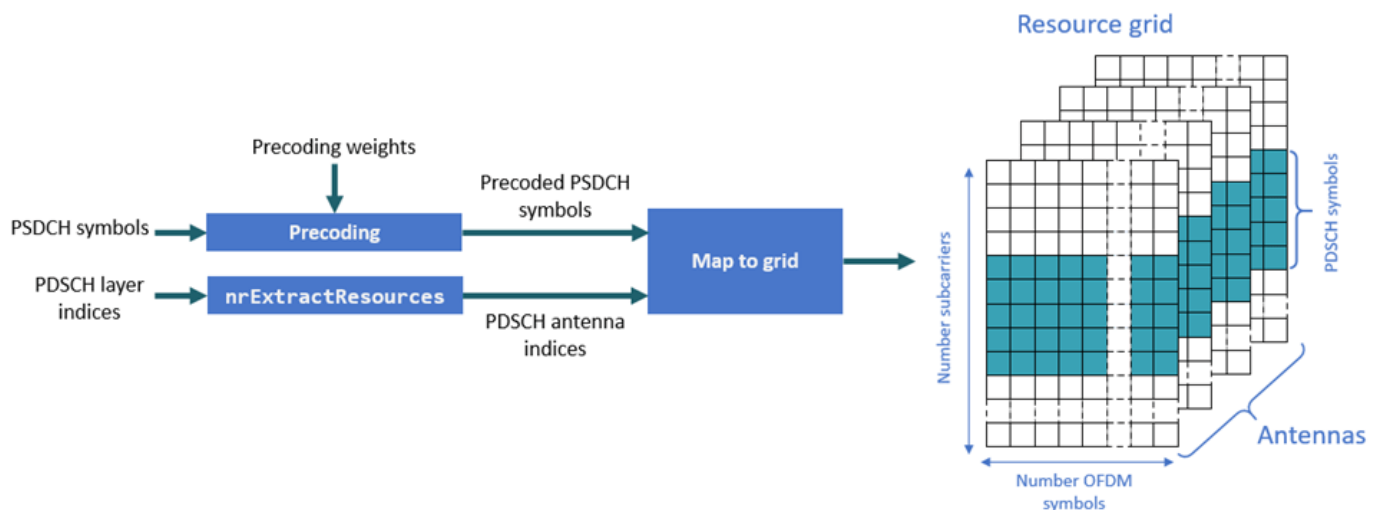
```
[~,pdschAntIndices] = nrExtractResources(pdschIndices,pdschGrid);
pdschGrid(pdschAntIndices) = pdschSymbolsPrecoded;
```

MIMO-precode and map the DM-RS symbols to the resource grid. Similar to the PDSCH indices, the DM-RS indices refer to layers. To convert these layer indices to antenna indices, use the `nrExtractResources` function again.

```
% PDSCH DM-RS precoding and mapping
for p = 1:size(dmrsSymbols,2)
    [~,dmrsAntIndices] = nrExtractResources(dmrsIndices(:,p),pdschGrid);
    pdschGrid(dmrsAntIndices) = pdschGrid(dmrsAntIndices) + dmrsSymbols(:,p)*precodingWeights
end
```

**OFDM Modulation**

OFDM-modulate the resource grid.

```
[txWaveform,waveformInfo] = nrOFDMModulate(carrier,pdschGrid);
```

Propagation Channel

The propagation channel generates N output samples for an input with N samples. However, the block of N output samples includes the channel filter transient (K samples). Because the synchronization stage removes this initial transient, if a slot at the channel output has N samples, N-K samples remain after synchronization. N-K samples are not enough to decode a slot-worth of data. Part of the slot samples are in the channel filter delay line and are not flushed yet. To flush all relevant samples out of the channel filter, pad the input signal with zeros. The maximum delay that the channel filter introduces affects the size of the padding. The padding accounts for the delay introduced by all multipath components and the channel filter implementation delay. This figure shows the need for zero padding before a waveform enters the channel.
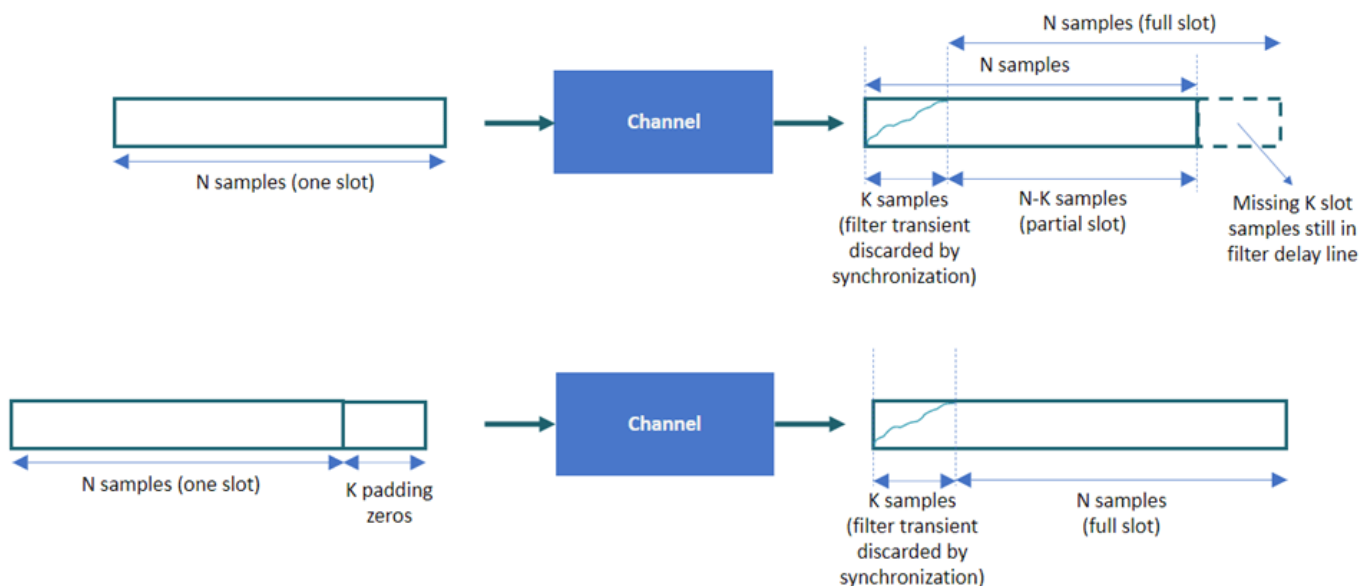
Pad the input signal with enough zeros to ensure that the generated signal is flushed out of the channel filter.

```
chInfo = info(channel);
maxChDelay = ceil(max(chInfo.PathDelays*channel.SampleRate)) + chInfo.ChannelFilterDelay;
txWaveform = [txWaveform; zeros(maxChDelay,size(txWaveform,2))];
```

Send the signal through the channel and add noise.

```
[rxWaveform,pathGains,sampleTimes] = channel(txWaveform);
noise = generateAWGN(SNRdB,nRxAnts,waveformInfo.Nfft,size(rxWaveform));
rxWaveform = rxWaveform + noise;
```

Timing Synchronization

You can perform perfect or practical synchronization.

- Perfect synchronization assumes channel knowledge (`nrPerfectTimingEstimate`). The channel returns information on path gains and path filters impulse response. You can use this information to determine the offset that is associated with the strongest multipath component across all channel snapshots and across all transmit and receive antennas.

- Practical synchronization performs a cross-correlation of the received signal with the PDSCH DM-RS symbols in the time domain (`nrTimingEstimate`). In some adverse cases, this cross-correlation can be weak due to fading or noise, resulting in an erroneous timing offset. The function `hSkipWeakTimingOffset` checks the magnitude of the cross-correlation `mag`. If the cross-correlation is weak, the function ignores the current timing estimate and instead uses the previous estimate (`offset`).

Perform perfect or practical timing estimation and synchronization.

```
if perfectEstimation
    % Get path filters for perfect timing estimation
    pathFilters = getPathFilters(channel);
    [offset,mag] = nrPerfectTimingEstimate(pathGains,pathFilters);
else
    [t,mag] = nrTimingEstimate(carrier,rxWaveform,dmrsIndices,dmrsSymbols);
    offset = hSkipWeakTimingOffset(offset,t,mag);
end
rxWaveform = rxWaveform(1+offset:end,:);
```

OFDM Demodulation

OFDM-demodulate the synchronized signal.

```
rxGrid = nrOFDMDemodulate(carrier,rxWaveform);
```
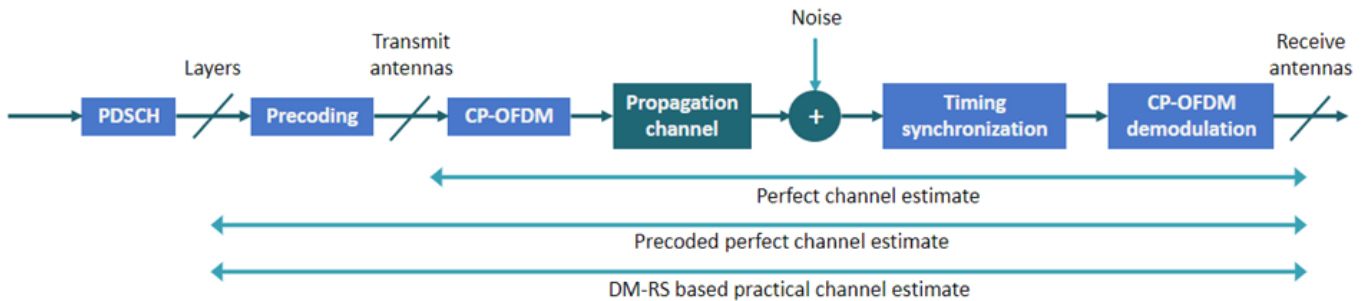
Channel Estimation

Channel estimation provides a representation of the channel effects per resource element (RE). The equalizer uses this information to compensate for the distortion introduced by the channel.

You can perform perfect or practical channel estimation.

- Perfect channel estimation assumes channel knowledge (`nrPerfectChannelEstimate`). The perfect channel estimate represents the channel conditions between the transmit and receive antennas. Because the equalizer requires channel knowledge between the transmit layers and the receive antennas, you must apply precoding to the perfect channel estimate.

- Practical channel estimation uses the PDSCH DM-RS to estimate the channel conditions and uses noise averaging and interpolation to obtain an estimate for all REs in the slot. Because the DM-

RSs are specified per layer, the resulting practical channel estimate represents the channel conditions between the transmit layers and the receive antennas. The practical channel estimate includes the effect of the MIMO precoding operation.

This figure shows the reference points of the channel estimates in the downlink processing chain.

Perform perfect or practical channel estimation.

```
if perfectEstimation
    % Perform perfect channel estimation between transmit and receive
    % antennas.
    estChGridAnts = nrPerfectChannelEstimate(carrier,pathGains,pathFilters,offset,sampleTimes

    % Get perfect noise estimate (from noise realization)
    noiseGrid = nrOFDMDemodulate(carrier,noise(1+offset:end ,:));
    noiseEst = var(noiseGrid(:));

    % Get precoding matrix for next slot
    newPrecodingWeight = getPrecodingMatrix(pdsch.PRBSet,pdsch.NumLayers,estChGridAnts);

    % Apply precoding to estChGridAnts. The resulting estimate is for
    % the channel estimate between layers and receive antennas.
    estChGridLayers = precodeChannelEstimate(estChGridAnts,precodingWeights.');
else
    % Perform practical channel estimation between layers and receive
    % antennas.
    [estChGridLayers,noiseEst] = nrChannelEstimate(carrier,rxGrid,dmrsIndices,dmrsSymbols,'CI

    % Remove precoding from estChannelGrid before precoding
    % matrix calculation
    estChGridAnts = precodeChannelEstimate(estChGridLayers,conj(precodingWeights));

    % Get precoding matrix for next slot
    newPrecodingWeight = getPrecodingMatrix(pdsch.PRBSet,pdsch.NumLayers,estChGridAnts);
end
```
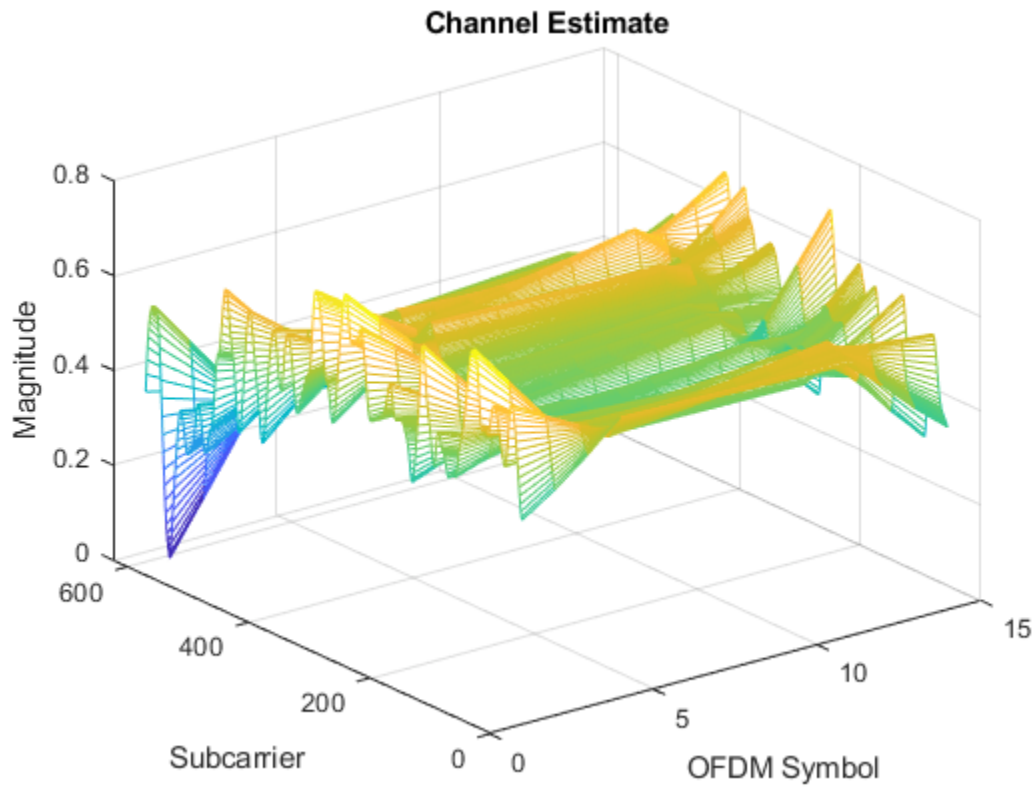
Plot the channel estimate between the first layer and the first receive antenna.

```
mesh(abs(estChGridLayers(:,:,1,1)));
```

Channel Estimate


```
title('Channel Estimate');
xlabel('OFDM Symbol');
ylabel("Subcarrier");
zlabel("Magnitude");
```

At this point, you can use the channel estimate to obtain the precoding matrix for transmission in the next slot. Because this example assumes channel knowledge at the transmitter, you do not need to calculate the precoding matrix at the receiver end. For an example of how to calculate the precoding matrix for data transmission based on a channel estimate at the receiver, see "NR PDSCH Throughput".

Equalization

The equalizer uses the channel estimate to compensate for the distortion introduced by the channel.

Extract the PDSCH symbols from the received grid and associated channel estimates. The `csi` output has channel state information (CSI) for each of the equalized PDSCH symbols. The CSI is a measure of the channel conditions for each PDSCH symbol. Use the CSI to weight the decoded soft bits after PDSCH decoding, effectively increasing the importance of symbols experiencing better channel conditions.

```
[pdschRx,pdschHest] = nrExtractResources(pdschIndices,rxGrid,estChGridLayers);
[pdschEq,csi] = nrEqualizeMMSE(pdschRx,pdschHest,noiseEst);
```
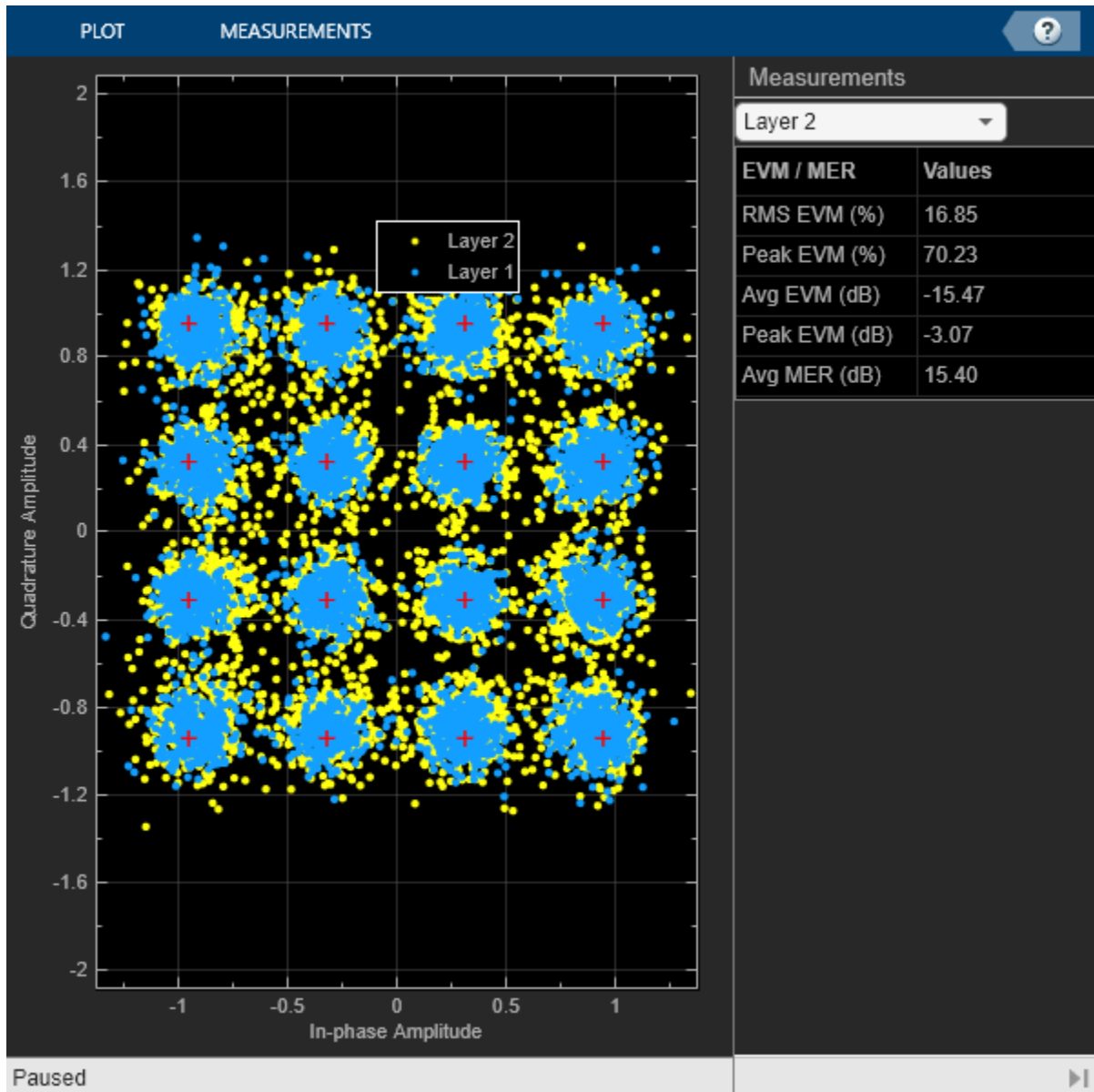
Plot the constellation of the equalized symbols. The plot includes the constellation diagrams for all layers.

```
constPlot.ChannelNames = "Layer "+(pdsch.NumLayers:-1:1);
constPlot.ShowLegend = true;
% Constellation for the first layer has a higher SNR than that for the
% last layer. Flip the layers so that the constellations do not mask
% each other.
constPlot(fliplr(pdschEq));
```


PDSCH Decoding

Decode the equalized PDSCH symbols and obtain the soft bit codewords.

```
[dlschLLRs,rxSymbols] = nrPDSCHDecode(carrier,pdsch,pdschEq,noiseEst);
```

Scale the soft bits or log-likelihood ratios (LLRs) by the CSI. This scaling applies a larger weight to the symbols in the REs with better channel conditions.

```
    % Scale LLRs by CSI
    csi = nrLayerDemap(csi);                                   % CSI layer demapping
    for cwIdx = 1:pdsch.NumCodewords
        Qm = length(dlschLLRs{cwIdx})/length(rxSymbols{cwIdx}); % Bits per symbol
        csi{cwIdx} = repmat(csi{cwIdx}.',Qm,1);                % Expand by each bit per symbol
        dlschLLRs{cwIdx} = dlschLLRs{cwIdx} .* csi{cwIdx}(:);  % Scale
    end
```

DL-SCH Decoding

Decode the LLRs and check for errors.

```
    decodeDLSCH.TransportBlockLength = trBlkSizes;
    [decbits,blkerr] = decodeDLSCH(dlschLLRs,pdsch.Modulation,pdsch.NumLayers, ...
        harqEntity.RedundancyVersion,harqEntity.HARQProcessID);
```

HARQ Process Update

Update the current HARQ process with the resulting block error status, and then advance to the next process. This step updates the information related to the active HARQ process in the HARQ entity.

```
    statusReport = updateAndAdvance(harqEntity,blkerr,trBlkSizes,pdschInfo.G);
```

Summarize HARQ and decoding information for the present slot.

```
    disp("Slot "+(nSlot)+". "+statusReport);
```

```
Slot 0. HARQ Proc 0: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 1. HARQ Proc 1: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 2. HARQ Proc 2: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 3. HARQ Proc 3: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 4. HARQ Proc 4: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 5. HARQ Proc 5: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 6. HARQ Proc 6: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 7. HARQ Proc 7: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 8. HARQ Proc 8: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 9. HARQ Proc 9: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 10. HARQ Proc 10: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 11. HARQ Proc 11: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 12. HARQ Proc 12: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 13. HARQ Proc 13: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 14. HARQ Proc 14: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 15. HARQ Proc 15: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 16. HARQ Proc 0: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 17. HARQ Proc 1: CW0: Initial transmission passed (RV=0,CR=0.482212).
Slot 18. HARQ Proc 2: CW0: Initial transmission passed (RV=0,CR=0.482212).

Slot 19. HARQ Proc 3: CW0: Initial transmission passed (RV=0,CR=0.482212).
```

```
end % for nSlot = 0:totalNoSlots
```

Local Functions

```
function noise = generateAWGN(SNRdB,nRxAnts,Nfft,sizeRxWaveform)
% Generate AWGN for a given value of SNR in dB (SNRDB), which is the
% receiver SNR per RE and antenna, assuming the channel does
% not affect the power of the signal. NRXANTS is the number of receive
% antennas. NFFT is the FFT size used in OFDM demodulation. SIZERXWAVEFORM
% is the size of the receive waveform used to calculate the size of the
```

```
    % noise matrix.

        % Normalize noise power by the IFFT size used in OFDM modulation, as
        % the OFDM modulator applies this normalization to the transmitted
        % waveform. Also normalize by the number of receive antennas, as the
        % channel model applies this normalization to the received waveform by
        % default. The SNR is defined per RE for each receive antenna (TS
        % 38.101-4).
        SNR = 10^(SNRdB/10); % Calculate linear noise gain
        N0 = 1/sqrt(2.0*nRxAnts*double(Nfft)*SNR);
        noise = N0*complex(randn(sizeRxWaveform),randn(sizeRxWaveform));
    end

    function wtx = getPrecodingMatrix(PRBSet,NLayers,hestGrid)
    % Calculate precoding matrix given an allocation and a channel estimate

        % Allocated subcarrier indices
        allocSc = (1:12)' + 12*PRBSet(:).';
        allocSc = allocSc(:);

        % Average channel estimate
        [~,~,R,P] = size(hestGrid);
        estAllocGrid = hestGrid(allocSc,:,:,:);
        Hest = permute(mean(reshape(estAllocGrid,[],R,P)),[2 3 1]);

        % SVD decomposition
        [~,~,V] = svd(Hest);

        wtx = V(:,1:NLayers).';
        wtx = wtx/sqrt(NLayers); % Normalize by NLayers
    end

    function estChannelGrid = getInitialChannelEstimate(channel,carrier)
    % Obtain an initial channel estimate for calculating the precoding matrix.
    % This function assumes a perfect channel estimate

        % Clone of the channel
        chClone = channel.clone();
        chClone.release();

        % No filtering needed to get channel path gains
        chClone.ChannelFiltering = false;

        % Get channel path gains
        [pathGains,sampleTimes] = chClone();

        % Perfect timing synchronization
        pathFilters = getPathFilters(chClone);
        offset = nrPerfectTimingEstimate(pathGains,pathFilters);

        % Perfect channel estimate
        estChannelGrid = nrPerfectChannelEstimate(carrier,pathGains,pathFilters,offset,sampleTimes);
    end

    function refPoints = getConstellationRefPoints(mod)
    % Calculate the reference constellation points for a given modulation
    % scheme.
        switch mod
```

```matlab
        case "QPSK"
            nPts = 4;
        case "16QAM"
            nPts = 16;
        case "64QAM"
            nPts = 64;
        case "256QAM"
            nPts = 256;
    end
    binaryValues = int2bit(0:nPts-1,log2(nPts));
    refPoints = nrSymbolModulate(binaryValues(:),mod);
end

function estChannelGrid = precodeChannelEstimate(estChannelGrid,W)
% Apply precoding matrix W to the last dimension of the channel estimate.

    % Linearize 4-D matrix and reshape after multiplication
    K = size(estChannelGrid,1);
    L = size(estChannelGrid,2);
    R = size(estChannelGrid,3);
    estChannelGrid = reshape(estChannelGrid,K*L*R,[]);
    estChannelGrid = estChannelGrid*W;
    estChannelGrid = reshape(estChannelGrid,K,L,R,[]);

end
```

See Also

Related Examples

- "Model 5G NR Transport Channels with HARQ" on page 2-41
- "Map 5G Physical Channels and Signals to the Resource Grid" on page 2-49

5G NR Waveform Generation

This example provides an overview of 5G NR waveform generation workflows and the waveform types that you can generate using the 5G Toolbox™ product.

Introduction

Using 5G Toolbox features, you can configure and generate these NR waveforms.

- NR test models (NR-TM)
- NR uplink and downlink fixed reference channels (FRCs)
- NR downlink waveforms
- NR uplink waveforms

To configure and generate a waveform with a static set of parameters, you can use either of these workflows. You can use the waveforms generated with these workflows in test and measurement applications.

- Use the 5G Waveform Generator app, which provides a user interface (UI) to configure the waveform. Because 5G waveforms have a large number of parameters, the recommended workflow for parameterizing a waveform is to use this app. The app enables you to generate the waveform directly in the app or to export the waveform configuration to MATLAB® to generate the waveform at the command prompt. The app also enables you to export the waveform configuration to Simulink™.
- Use the `nrWaveformGenerator` function, which provides a programmatic interface to configure the waveform using a configuration object.

To learn how to configure and generate waveforms with dynamically changing parameters (for example, when modeling a 5G link) see the "NR PDSCH Throughput" and "NR PUSCH Throughput" examples.

To learn how to configure and generate physical random access channel (PRACH) waveforms, see the "5G NR PRACH Configuration" and "5G NR PRACH Waveform Generation" examples.

Configure and Generate 5G Waveforms Using the App

The **5G Waveform Generator** app provides a UI to manage the large number of configuration parameters. In the app, you can choose the waveform type, specify the parameters, and generate and export the waveform. The app also enables you to interact with test and measurement equipment. This figure shows a common workflow to generate and export a 5G waveform in this app.
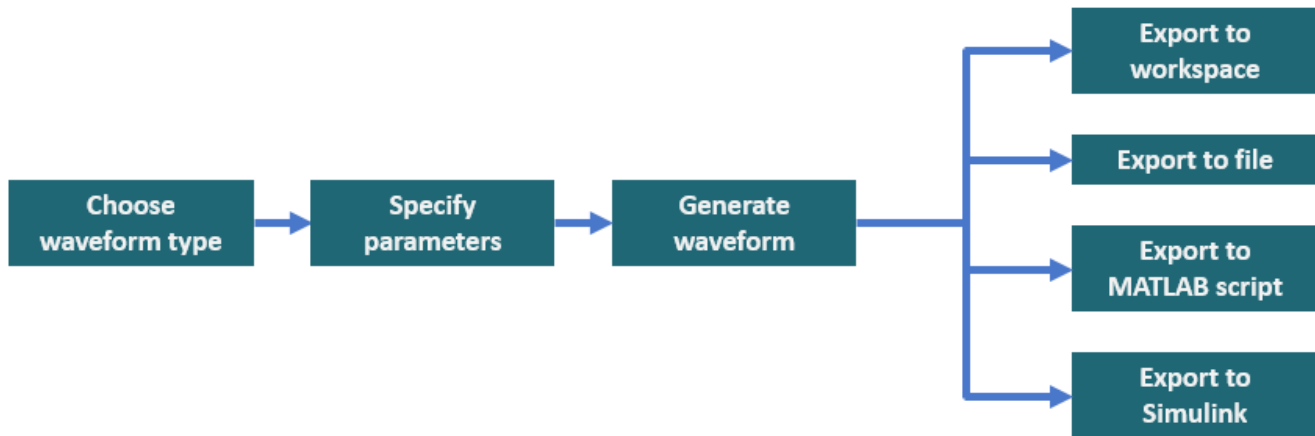
Open 5G Waveform Generator App

Open the **5G Waveform Generator** app by clicking the app icon on the **Apps** tab, under **Signal Processing and Communications**.


Choose Waveform Type

In the app, you can select different waveform types. The downlink and uplink options enable you to fully customize the contents of your waveform. You can also generate NR-TMs, downlink FRCs, and uplink FRCs.
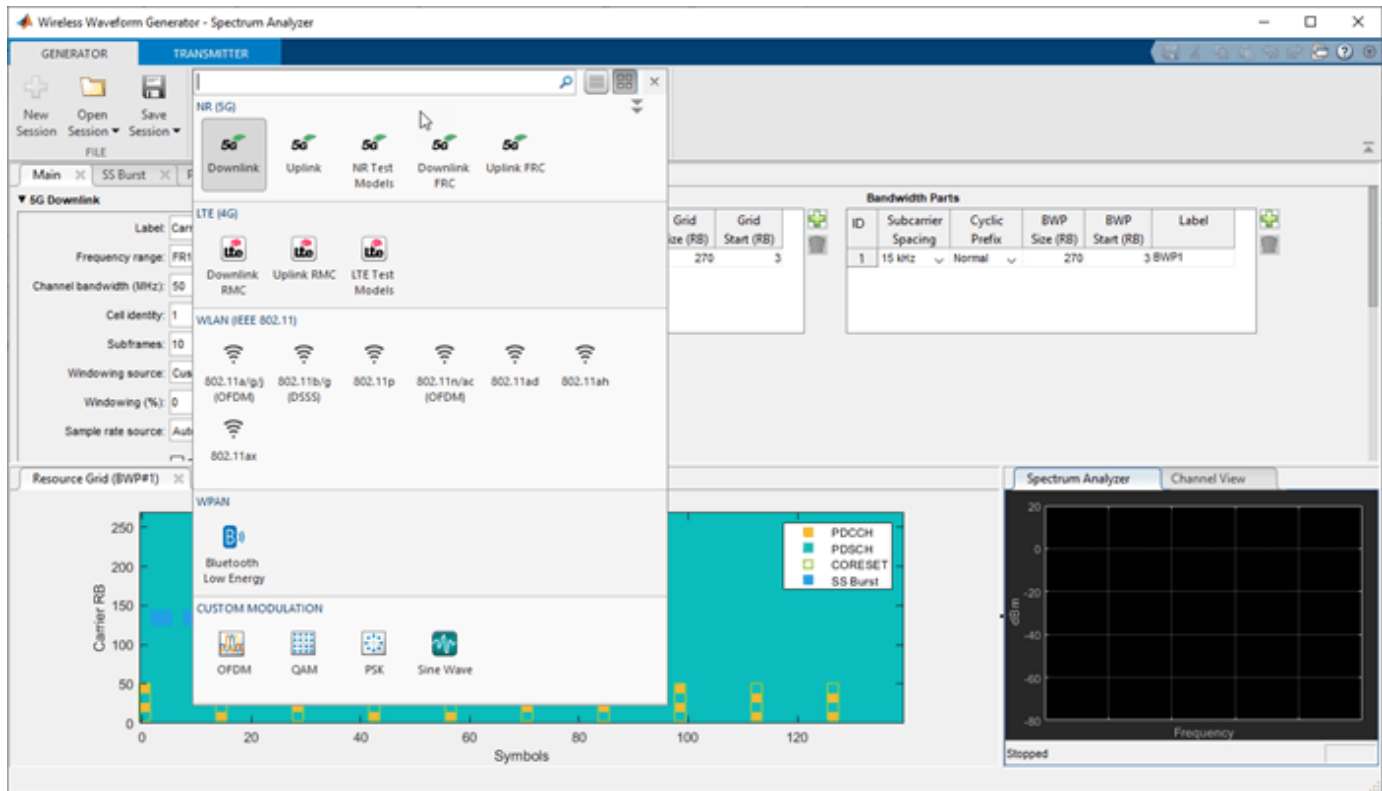
Specify Parameters

In the app, you can specify the parameters. When you set the parameters, the app updates the resource grid visualization content instantly, showing the location of all physical channels in the waveform. The resource grid view contains a union of the locations of all physical channels over all ports (that is, the visualization does not differentiate what each port transmits). Because the maximum resolution of the resource grid is one resource block (RB), the visualization does not show signals using single resource elements (REs).

This figure shows the configuration of two physical downlink shared channels (PDSCHs). The first PDSCH spans all slots and uses physical resource blocks (PRBs) 0 to 100. The second PDSCH is active in slots 0 to 2 and 4 to 6 and uses PRBs 200 to 250.
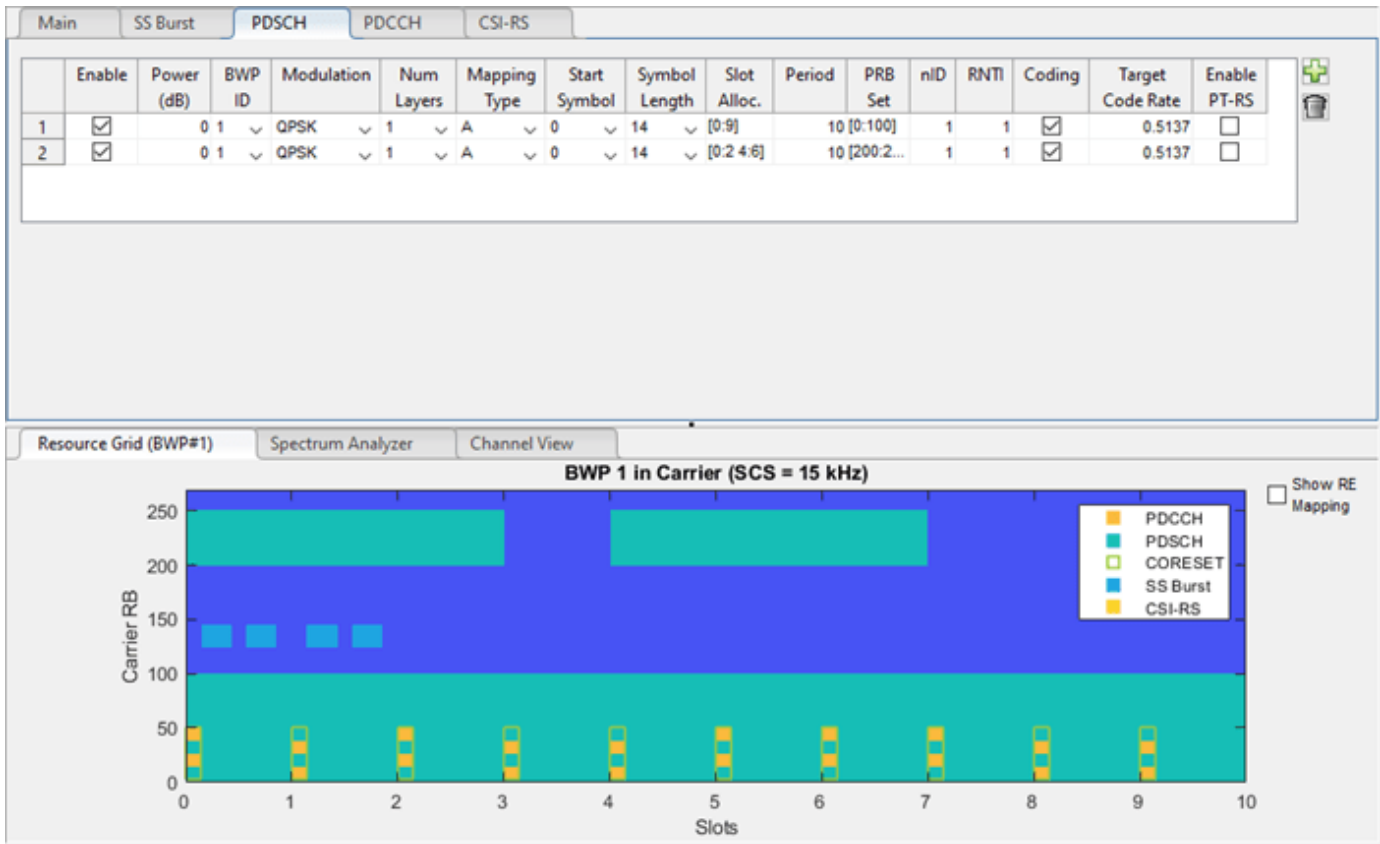
Generate Waveform

To generate the configured waveform in the app, click **Generate.** The app creates the baseband in-phase and quadrature (IQ) component samples internally in the generator. You can see the spectrum of the generated signal in the **Spectrum Analyzer** tab.

Export Waveform

To export the waveform, click **Export** and select one of the available options. You can export the waveform to the workspace, a file, a MATLAB script, or a Simulink model.
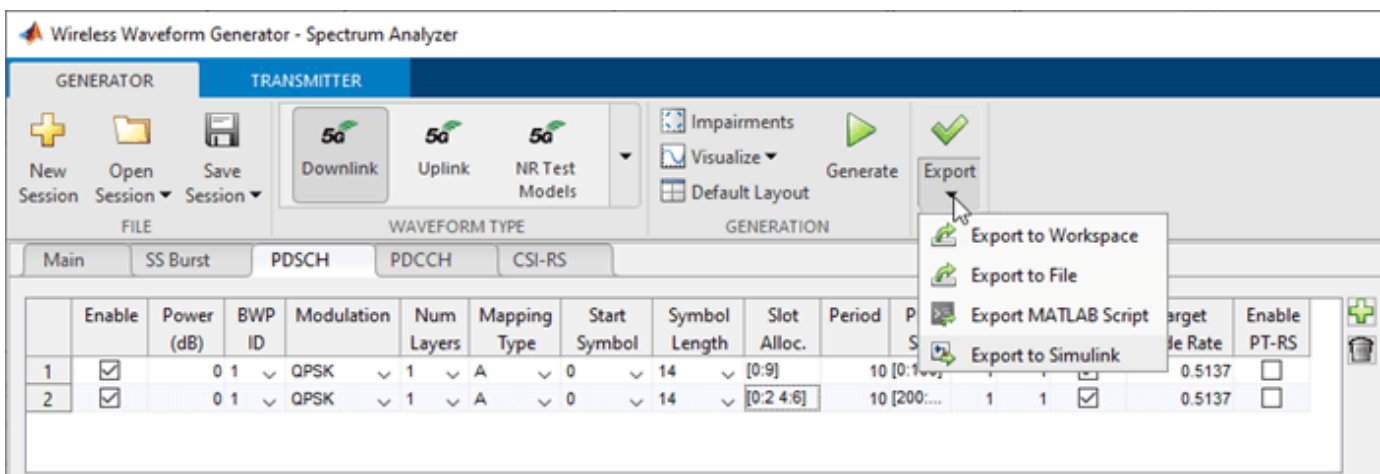
- The **Export to Workspace** option creates a structure in the MATLAB workspace. The structure contains the waveform samples, sampling frequency, configuration parameters, and a string describing the waveform type (downlink, uplink, test model, downlink FRC, or uplink FRC). For example:

```
>> myWaveform

myWaveform =

  struct with fields:

        type: 'Downlink'
      config: [1×1 struct]
          Fs: 61440000
    waveform: [614400×1 double]
```

- The **Export to File** option saves the waveform as a `.mat` or a `.bb` baseband file.
- The **Export to MATLAB Script** option creates a MATLAB script. Run the script to generate the configured waveform at the MATLAB command window.
- The **Export to Simulink** option generates a Waveform From Wireless Waveform Generator App block. Use the block as a waveform source in a Simulink model.

Configure and Generate 5G Waveforms Using MATLAB Code

The `nrWaveformGenerator` function provides a programmatic interface to configure the waveform using a configuration object. Instead of specifying all parameters manually, which is time consuming, you can configure the waveform in the 5G Waveform Generator app and export this configuration to a MATLAB script. You can modify and run this MATLAB script to generate the configured 5G waveform.

Using the app has these benefits.

- The app includes UI controls, such as drop-down lists to select values.
- Some UI controls include validation when you set custom values.
- The grid visualization capability enables you to see what the signal looks like as you specify the parameters.

For example, the code example in this section generates a 5G downlink waveform using the `nrDLCarrierConfig` configuration object. The code that you generate for a downlink waveform using the **Export to MATLAB Script** option in the app also uses the `nrDLCarrierConfig` configuration object.

Create a default downlink waveform configuration object. The `waveconfig` object contains the full waveform specification and is fully configurable.

```
waveconfig = nrDLCarrierConfig

waveconfig =
  nrDLCarrierConfig with properties:

             Label: 'Downlink carrier 1'
    FrequencyRange: 'FR1'
```

```
      ChannelBandwidth: 50
              NCellID: 1
          NumSubframes: 10
      WindowingPercent: 0
            SampleRate: []
      CarrierFrequency: 0
            SCSCarriers: {[1x1 nrSCSCarrierConfig]}
         BandwidthParts: {[1x1 nrWavegenBWPConfig]}
                SSBurst: [1x1 nrWavegenSSBurstConfig]
                CORESET: {[1x1 nrCORESETConfig]}
           SearchSpaces: {[1x1 nrSearchSpaceConfig]}
                  PDCCH: {[1x1 nrWavegenPDCCHConfig]}
                  PDSCH: {[1x1 nrWavegenPDSCHConfig]}
                  CSIRS: {[1x1 nrWavegenCSIRSConfig]}
```

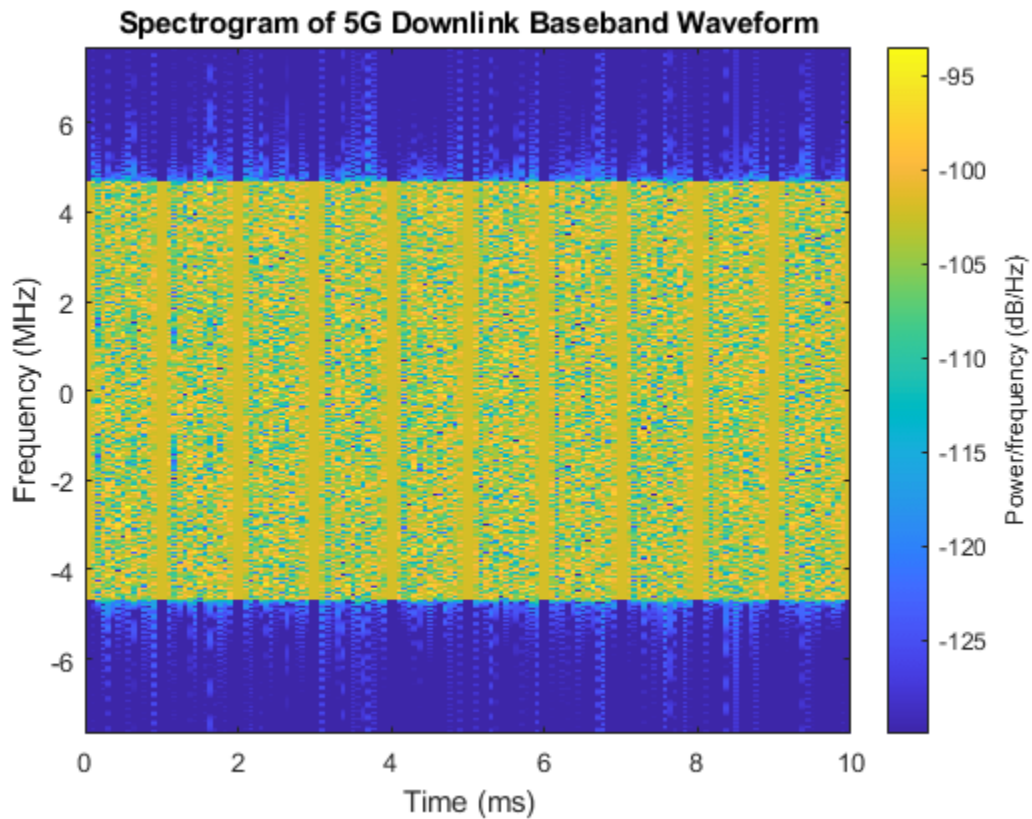After you set the configuration parameters, call the programmatic waveform generator.

```
[waveform,waveformInfo] = nrWaveformGenerator(waveconfig);
```

Plot the spectrogram to visualise the signal in the frequency domain. This waveform includes a full allocation PDSCH, a physical downlink control channel (PDCCH), and the signal synchronization (SS) burst.

```matlab
% Plot spectrogram of waveform for first antenna port
samplerate = waveformInfo.ResourceGrids(1).Info.SampleRate;
nfft = waveformInfo.ResourceGrids(1).Info.Nfft;
figure;
spectrogram(waveform(:,1),ones(nfft,1),0,nfft,'centered',samplerate,'yaxis','MinThreshold',-130)
title('Spectrogram of 5G Downlink Baseband Waveform');
```
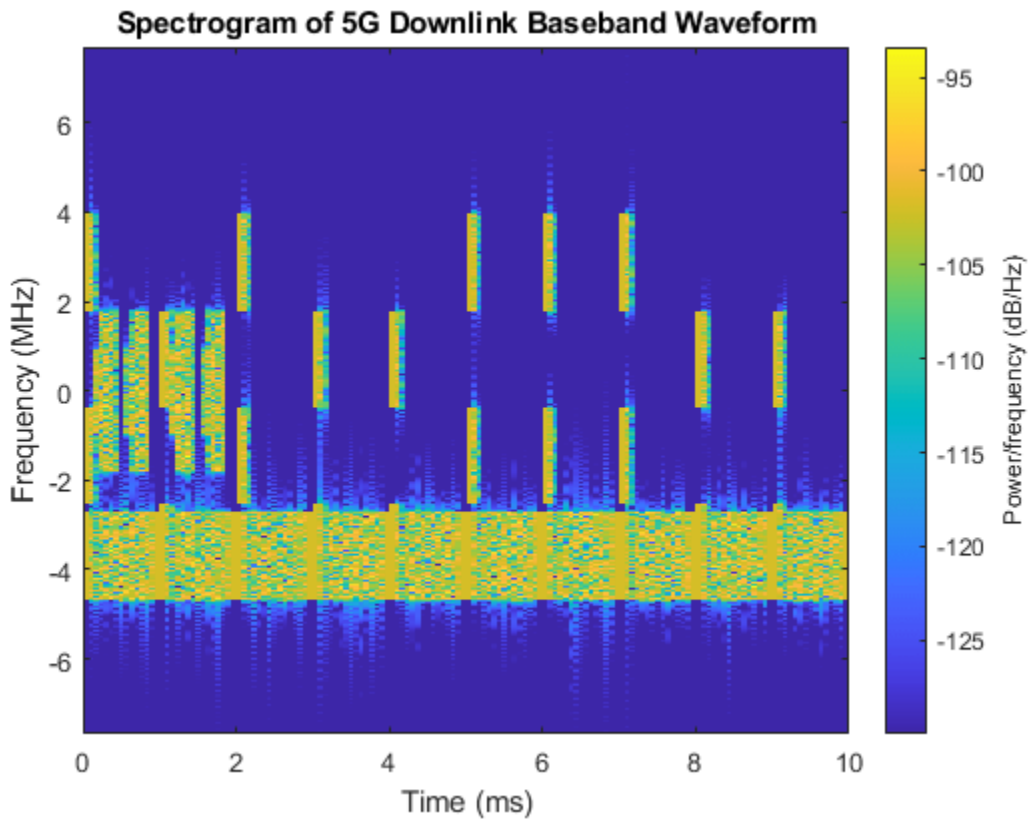
Change the PDSCH allocation to span PRBs 0 to 10. Generate the waveform and plot the spectrogram.

```
waveconfig.PDSCH{1}.PRBSet = 0:10;
[waveform,waveformInfo] = nrWaveformGenerator(waveconfig);

% Plot spectrogram of waveform for first antenna port
samplerate = waveformInfo.ResourceGrids(1).Info.SampleRate;
nfft = waveformInfo.ResourceGrids(1).Info.Nfft;
figure;
spectrogram(waveform(:,1),ones(nfft,1),0,nfft,'centered',samplerate,'yaxis','MinThreshold',-130)
title('Spectrogram of 5G Downlink Baseband Waveform');
```

**Spectrogram of 5G Downlink Baseband Waveform**

You can define multiple instances of the physical channels and signals. Create a second instance of the PDSCH configuration object and set the allocation to span PRBs 40 to 50 and OFDM symbols 2 to 10.

```
mySecondPDSCH = nrWavegenPDSCHConfig;
mySecondPDSCH.PRBSet = 40:50;
mySecondPDSCH.SymbolAllocation = [2 10];
```

Assign the second PDSCH configuration to the waveform configuration. Generate the waveform.

```
waveconfig.PDSCH{2} = mySecondPDSCH;
[waveform,waveformInfo] = nrWaveformGenerator(waveconfig);

% Plot spectrogram of waveform for first antenna port
samplerate = waveformInfo.ResourceGrids(1).Info.SampleRate;
nfft = waveformInfo.ResourceGrids(1).Info.Nfft;
figure;
spectrogram(waveform(:,1),ones(nfft,1),0,nfft,'centered',samplerate,'yaxis','MinThreshold',-130)
title('Spectrogram of 5G Downlink Baseband Waveform');
```

Spectrogram of 5G Downlink Baseband Waveform
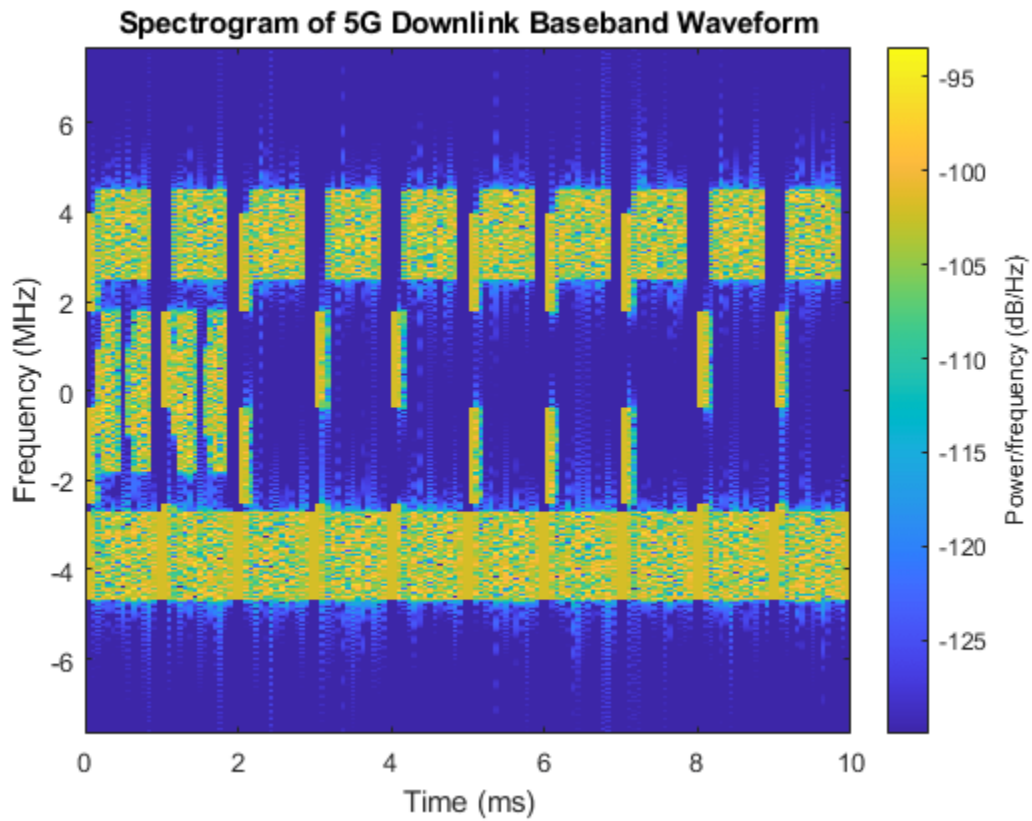
See Also

Related Examples

- "5G NR Downlink Vector Waveform Generation"
- "5G NR Uplink Vector Waveform Generation"
- "5G NR-TM and FRC Waveform Generation"