Lab: 5G NR Downlink Simulation with Beamforming

Beamforming is an essential component of the millimeter wave (mmWave) communication systems. This lab will demonstrate simple beamforming and channel modeling for a downlink transmissions in the 5G New Radio standard. In performing this lab, you will learn to:

- Model realistic antenna arrays and elements
- Compute beamforming vectors and the associated antenna factors
- Model multi-path fading channels in time-domain
- Modulate and demodulate symbols on the 5G NR downlink data channel using the MATLAB 5G Toolbox.
- Measure the quality of the demodulated symbols
- Organize complex projects into packages

Contents

Packages

Similar to python, you can group code with a common functionality into a MATLAB package. Using packages enables better organized, more modular code. For several of the remaining labs, the repository contains a folder `+nr` containing some routines for simulating 5G NR systems. These functions were mostly helper functions taken from MATLAB's excellent demo on 5G throughput. In this lab, we will use just a small portion of the code.

To use the package, you must first add the folder containing this +nr to your MATLAB path.

```
% TODO:  Use the MATLAB addpath() command to add the folder with the nr
% package.  You can use indirect referencing like '..'.
addpath('../..');
```

Parameters

We will use the following parameters

```
fc = 28e9;           % carrier frequency in Hz
nantUE = [4,4];      % array size at the UE (mobile device)
nantgNB = [8,8];     % array size at the gNB (base station)
snrPerAntenna = -5; % target SNR per sample per antenna
ueVel = [5 0 0];     % UE velocity vector in m/s
subcarrierSpacing = 120;  % sub-carrier spacing in kHz

% Creates simulation parameters for this lab
simParam = PDSCHSimParam('fc', fc);
```

Loading the 3GPP NR channel model

We will load the same channel as in the previous lab.

```
dlySpread = 50e-9;  % delay spread in seconds
chan = nrCDLChannel('DelayProfile','CDL-A',...
    'DelaySpread',dlySpread, 'CarrierFrequency', fc, ...
    'NormalizePathGains', true);
chaninfo = info(chan);

% Get the gains and other path parameters
gain = chaninfo.AveragePathGains;
aoaAz  = chaninfo.AnglesAoA;
aoaEl = 90-chaninfo.AnglesZoA;
aodAz  = chaninfo.AnglesAoD;
aodEl = 90-chaninfo.AnglesZoD;
dly = chaninfo.PathDelays;
```

Create the element

We will use the same patch element on the UE and gNB as before.

```
% Constants
vp = physconst('lightspeed');  % speed of light
lambda = vp/fc;   % wavelength

% Create a patch element
len = 0.49*lambda;
groundPlaneLen = lambda;
elem = patchMicrostrip(...
    'Length', len, 'Width', 1.5*len, ...
    'GroundPlaneLength', groundPlaneLen, ...
    'GroundPlaneWidth', groundPlaneLen, ...
    'Height', 0.01*lambda, ...
    'FeedOffset', [0.25*len 0]);

% Tilt the element so that the maximum energy is in the x-axis
elem.Tilt = 90;
elem.TiltAxis = [0 1 0];
```

Creating an element wrapper class

A problem with the Phased Array Toolbox is that the element patterns are not smoothly interpolated. Also, we want to support antennas that have analytic functions for their pattern. To support this, we will use a wrapper class, InterpPatternAntenna. This class derives from the system.Matlab super-class. Its step method provides the directivity as a function of the angles. We can then replace this class with any other class that provides a formula for the directivity.

```
% TODO:  Complete the setupImpl and stepImpl() methods in
% the InterPatternAntenna class.  This mostly follows the same syntax
% as the previous lab.

% TODO:  Create a wrapper object for the elem
%     elemInterp = InterpPatternAntenna(...);
elemInterp = InterpPatternAntenna(elem, fc);
```

Create the arrays

We next create arrays at the UE and gNB

```
% TODO:  Create two URAs with the sizes nantUE and nantgNB.
% Both arrays should be separated at 0.5 lambda.
%    arrUE0 = URA for the UE
%    arrgNB0 = URA for the gNB
dsep = 0.5*lambda;
arrUE0  = phased.URA(nantUE,dsep,'ArrayNormal','x');
arrgNB0 = phased.URA(nantgNB,dsep,'ArrayNormal','x');
```

Create the array with axes

Similar to the previous lab, we will create a wrapper class around the arrays to handle orientation modeling. The ArrayWithAxes class includes an array along with axes to store the orientation of the array relative to some global coordinate system.

```
% TODO:  Complete the setupImpl() and stepImpl() methods in the
% ArrayWithAxes class.

% TODO:  Create ArrayWithAxes objects for the gNB and UE with the
% arrays, arrUE0 and arrgNB0, and elements, elemInterp.  Also
% set the frequency and velocity.
%    arrgNB = ArrayWithAxes(...);
%    arrUE  = ArrayWithAxes(...);
arrgNB = ArrayWithAxes('arr', arrgNB0, 'elem', elemInterp, 'fc', fc);
arrUE  = ArrayWithAxes('arr', arrUE0, 'elem', elemInterp, 'fc', fc);
arrUE.set('vel', ueVel);
```

Rotate the UE and gNB antennas

Similar to the previous lab, we will rotate the arrays to the path with the maximum gain.

```
% TODO:  Find the index of the path with the maximum gain.
[gainmax, im] = max(gain);

% TODO:  Call the arrUE.alignAxes() and arrgNB.alignAxes() to
% align to the corresponding angles of arrival and departure.
arrUE.alignAxes(aoaAz(im), aoaEl(im));
arrgNB.alignAxes(aodAz(im), aodEl(im));
```

Compute the gains along the paths

To see the potential gain from beamforming, we will compute the array factor and element gain along each path

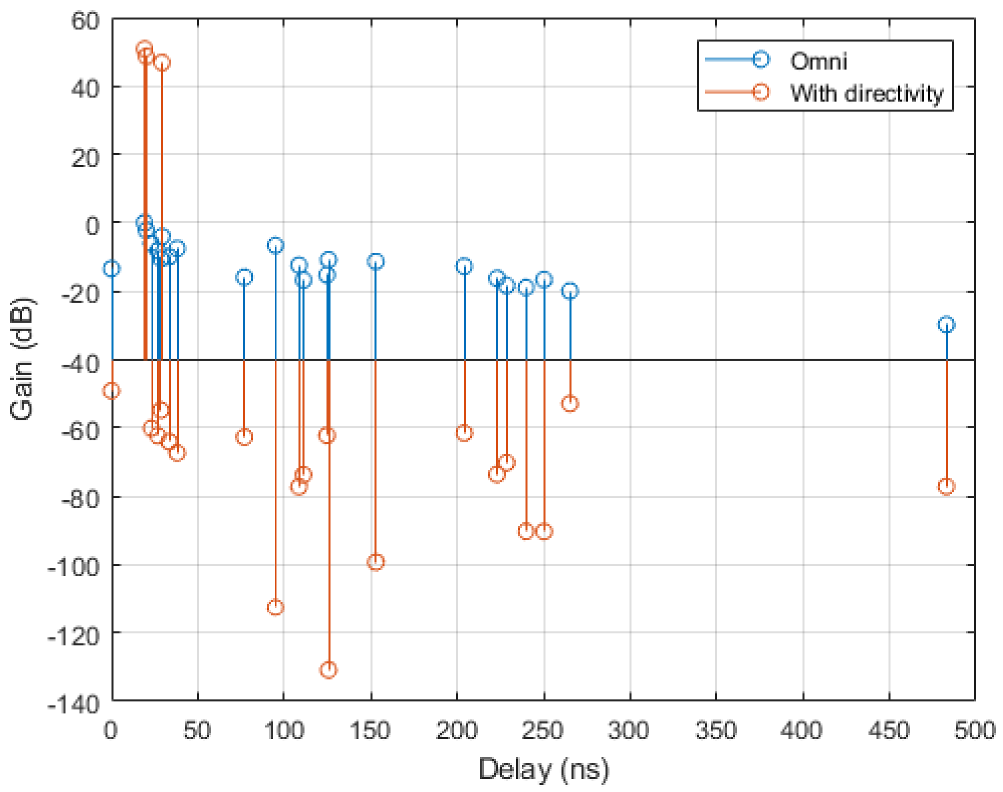
```
% TODO:  Find the spatial signatures and element gains of each path
% based on their angles of arrival and departure.
%   [utx, elemGainTx] = arrgNB.step(...);
%   [urx, elemGainRx] = arrUE.step(...);
[utx, elemGainTx] = arrgNB.step(aodAz, aodEl);
[urx, elemGainRx] = arrUE.step(aoaAz, aoaEl);

% TODO:  Compute the TX beamforming direction at the gNB and RX BF
% direction at the UE.  To keep the beamforming simple, we will align
% the directions to the strongest path.  Thus, the BF directions should
% be complex conjugate of the steering vectors.  They should also be
% normalized.
%     wtx = TX direction at the gNB
%     wrx = RX direction at the UE
wtx = conj(utx(:,im));
wtx = wtx / norm(wtx);
wrx = conj(urx(:,im));
wrx = wrx / norm(wrx);

% TODO:  Compute the array factors at the gNB and UE
% from the BF vectors and spatial signatures, utx and urx.
%    AFgNB(i) = array factor gain on path i in dBi at the gNB
%    AFUE(i) = array factor gain in path i dBi at the UE
AFgNB = 20*log10(abs(wtx.'*utx));
AFUE = 20*log10(abs(wrx.'*urx));

% TODO:  Compute the gain on each path adding the array factors
% and elemement gains.
%    gainDir = gain + ...
gainDir = gain + AFgNB + AFUE + elemGainTx + elemGainRx;

% TODO:  Use the stem plot to plot both the original gain and
% gainDir, the gain with directivity.  Add a legend and label the axes.
% You will see that, with directivity, many of the paths are highly
% attenuated and a few are very significantly amplified.
stem(dly/1e-9, [gain; gainDir]', 'BaseValue', -40);
grid on;
xlabel('Delay (ns)');
ylabel('Gain (dB)');
legend('Omni', 'With directivity');
```



Generate a 5G TX signal

We will now test the array processing by transmitting a 5G downlink signal. Specifically, we will transmit random QPSK symbols on the locations of the PDSCH channel, the channel in the 5G NR standard for data. The lab supplies a simple class, NRgNBTx, to perform this function. Most of the class is implemented and extensively uses commands from the 5G Toolbox.

```
% TODO:  Complete the code in the NRgNBTx.stepImpl() method to
% perform the TX beamforming.


% TODO:  Create a TX object using the NRgNBTx object.  Pass the simParam
% object.
%    tx = NRgNBTx(...);
tx = NRgNBTx(simParam);

% TODO:  Set the BF vector of the TX
tx.set('txBF', wtx);

% TODO:  Generate one slot of symbols using the step() method
%    x = ...
x = tx.step();
```

The slot of data produced by this data contains two channels: * PDSCH: The downlink data channel * DM-RS: Demodulation reference signals for channel estimation at the RX. We will discuss this later.

```
% TODO:  Print the sampling rate in MHz and total time (in us) of the data.
% You can use the info in tx.waveformConfig.
fprintf(1,'Sampling rate = %7.2f MHz\n', ...
    tx.waveformConfig.SamplingRate/1e6);
fprintf(1,'Slot duration = %7.2f us\n', ...
    size(x,1)*1e6/tx.waveformConfig.SamplingRate);
```

```
% TODO:  Print the following:
% * Total number of REs (this can be found from the number of elements in
%   tx.ofdmGridLayer), the time-freq grid for the symbols in the slot
% * Number of REs for the DM-RS:  From tx.dmrsSym
% * Number of REs for the PDSCH:  From tx.pdschSym
fprintf(1,'Total REs:     %d \n', numel(tx.ofdmGridLayer) );
fprintf(1,'REs for DMRS:  %d \n', length(tx.dmrsSym) );
fprintf(1,'REs for PDSCH: %d \n', length(tx.pdschSym) );
```

```
Sampling rate =  122.88 MHz
Slot duration =  125.39 us
Total REs:     8568
REs for DMRS:  204
REs for PDSCH: 7752
```

Create the MIMO multi-path channel

We will now simulate the channel in time-domain. The lab supplies code, MIMOMPChan, which is a MIMO version of the SISOMPChan you created in the previous lab.

```
% TODO:  Complete the code in the stepImpl() method of the code.

% TODO:  Create the MIMOMPChan object with all the AoAs, AoDs, gains
% arrays, delays and sampling rate.
%    chan = ...
chan = MIMOMPChan('aoaAz', aoaAz, 'aoaEl', aoaEl, 'aodAz', aodAz, ...
    'aodEl', aodEl, 'gain', gain, 'rxArr', arrUE, 'txArr', arrgNB, ...
    'dly', dly, 'fsamp', tx.waveformConfig.SamplingRate);

% TODO:  Create the output
%   y = chan.step(...)
y = chan.step(x);
```

Add noise

In multi-antenna receivers, the SNR is typically quoted as the SNR per antenna. Specifically, suppose that ynoisy = y + w. The SNR per antenna is E|y(t,j)|^2/E|w(t,j)|^2. Create a matrix ynoisy using the snrPerAntenna.

```
Erx = mean(abs(y).^2, 'all');
Enoise = 10.^(-0.1*snrPerAntenna)*Erx;
ynoisy = y + sqrt(Enoise/2)*(randn(size(y)) +1i*randn(size(y)));
```

Create a UE receiver

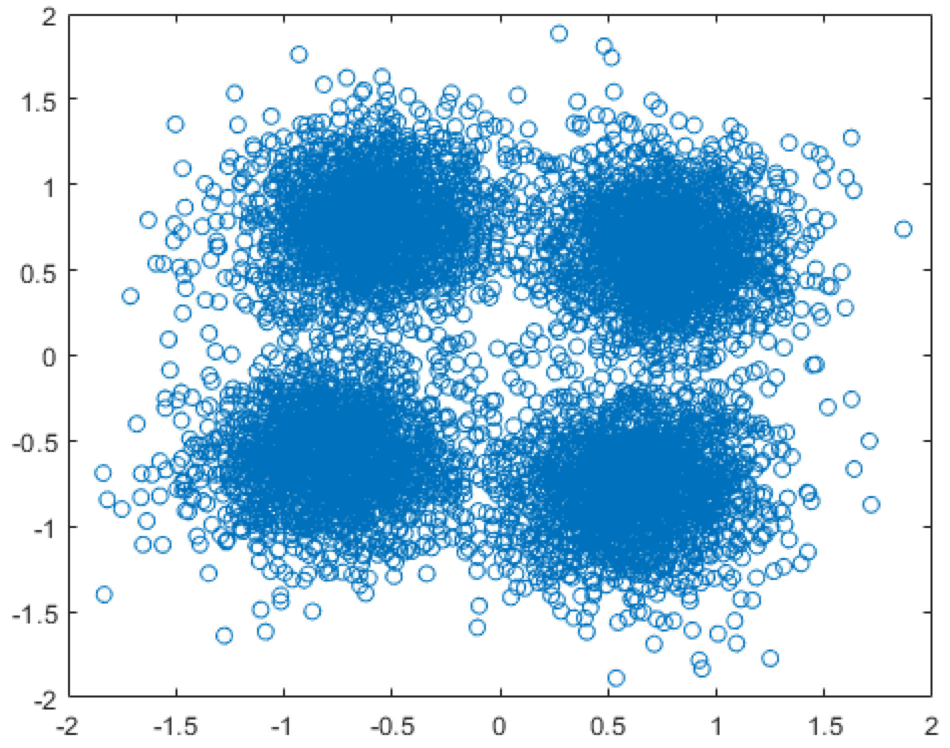
We will now demodulate the noisy symbols. The lab supplies a simple class, NRUErx, to perform this function. Most of the class is implemented and extensively uses commands from the 5G Toolbox. You just have to do a small modification to support BF.

```
% TODO:  Modify the stepImpl() method of NRUERx to implement RX
% beamforming.

% TODO:  Create a RX object with the correct rxBF vector
%   rx = NRUERx(...)
rx = NRUERx(simParam, 'rxBF', wrx);
```

```
% TODO:  Run the rx.step() method with ynoisy to receive the signals
rx.step(ynoisy);

% TODO:  The equalized symbols are now stored in rx.pdschSym.
plot(real(rx.pdschSymEq), imag(rx.pdschSymEq), 'o');
```



Measure the SNR

When you plot the final equalized symbols you will see that there is a lot of noise. Also there is a phase rotation which comes from the Doppler shift that was not corrected. In reality, you would have some carrier frequency offset to remove this. We will also discuss this later. For now, we measure the post-equalized SNR.

One way to measure the post-equalized SNR is: snrEq = 10*log10(E|r|^2 / E| r - h*x |^2) where r is the recived raw symbols (in this case rx.pdschSymRaw) h is the channel estimate (rx.pdschChanEst) and x is the transmitted symbols (tx.pdschSym).

```
% TODO:  Compute and print the post-equalized SNR in dB
Eerr = mean(abs(rx.pdschSymRaw - rx.pdschChanEst.*tx.pdschSym).^2);
Erx = mean(abs(rx.pdschSymRaw).^2);
snrEq = 10*log10(Erx / Eerr);
fprintf(1,'SNR post equalization = %7.2f\n', snrEq);
```

```
SNR post equalization =    8.63
```

Since the RX signal ynoisy already includes the antenna element gain, The post-equalized SNR should be: snrTheory = snrPerAntenna + bwGain + BFGain where BFGain is the BF gain and bwGain is the gain since we are transmitting on tx.waveformConfig.NSubcarriers out of tx.waveformConfig.Nfft FFT frequency bins. Ideally the BF gain is the number of UE antennas. You will notice that snrTheory is a little higher than what we received. This is mostly since the receiver we built here does not compensate for frequency offset.

```
% TODO:  Compute and print snrTheory.
nantrx = prod(nantUE);
bwGain = 10*log10(tx.waveformConfig.Nfft/tx.waveformConfig.NSubcarriers);
snrTheory = snrPerAntenna + bwGain + 10*log10(nantrx);
fprintf(1,'SNR theoretical = %7.2f\n', snrTheory);
```

```
SNR theoretical =    9.28
```

Published with MATLAB® R2020a