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

% **Submission**: Complete all the sections marked `TODO`, and run the cells to make sure your scipt is working. When you are satisfied with the results, publish your code to generate an html file. Print the html file to PDF and submit the PDF.

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
% Sai Meghana Kuchana and Jitendra Project
% package.  You can use indirect referencing like '..'.
```

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);
NLayers = 1;    % number of layers
Modulation = 'QPSK';
NRB = 51;  % number of resource blocks
SubcarrierSpacing = 120;  % SCS in kHZ
```

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.

```
% Creating wrapper object for the elem
elemInterp = InterpPatternAntenna(elem,fc);
```

Create the arrays

We next create arrays at the UE and gNB

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

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.

```
% Create ArrayWithAxes objects for the gNB and UE with the
% arrays, arrUE0 and arrgNB0, and elements, elemInterp.
velgNB = [30,0,0];
velUE = [30,0,0];
arrgNB = ArrayWithAxes('elem', elemInterp, 'arr', arrgNB0, 'vel', velgNB, 'fc', fc);
arrUE  = ArrayWithAxes('elem', elemInterp, 'arr', arrUE0, 'vel', velUE, 'fc', fc);
```

Rotate the UE and gNB antennas

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

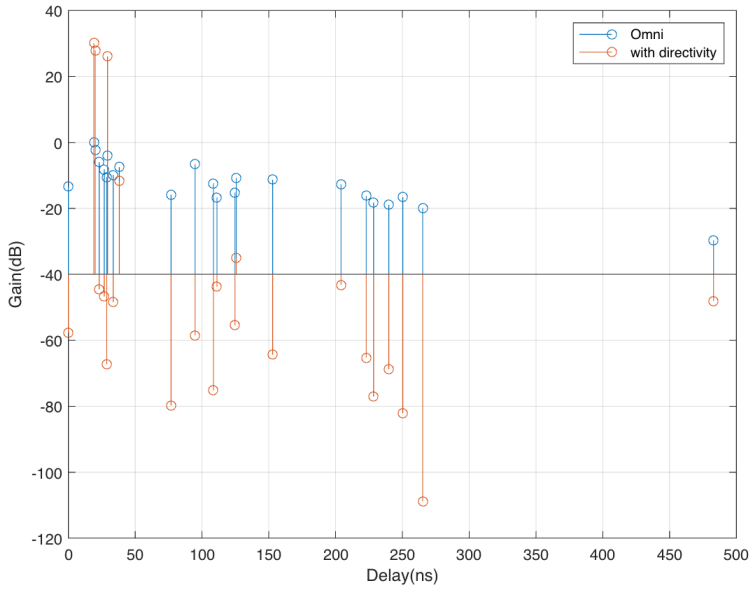
```
% Index of the path with the maximum gain
[~,im] = max(gain);

% 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

```
% Calculating spatial signatures and element gains of each path
% based on their angles of arrival and departure.

[utx, elemGainTx] = arrgNB.step(aoaAz, aoaEl);
[urx, elemGainRx] = arrUE.step(aodAz, aodEl);

% 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);
```

```
% Computing the array factors at the gNB and UE
% from the BF vectors and spatial signatures, utx and urx.
AFgNB = 10*log10( abs(sum(wtx.*utx, 1)).^2 ); % AFgNB(i) = array factor gain on path i in dBi at the gNB
AFUE = 10*log10( abs(sum(wrx.*urx, 1)).^2 ); % AFUE(i) = array factor gain in path i dBi at the UE
gainDir = gain + AFgNB + AFUE;
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.

```
% Creating a TX object using the NRgNBTx object with the simParam
% object.

tx = NRgNBTx(simParam);
% Set the BF vector of the TX
tx.txBF = wtx;

% Generating one slot of symbols using the step() method
x = tx.step();
%imagesc(abs(x));
%colorbar()
```

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.

```
fprintf(1, 'Sampling rate = %7.2f MHz\n', tx.waveformConfig.SampleRate/1e6);
```

```
Sampling rate =  122.88 MHz
```

```
fprintf(1,'Slot duration = %7.2f us\n', size(x,1)*1e6/tx.waveformConfig.SampleRate);
```

```
Slot duration =  125.39 us
```

```
nchan = 2;
fprintf(1,'Total REs: %d \n', numel(tx.ofdmGridLayer));
```

```
Total REs: 8568
```

```
fprintf(1,'REs for DMRS: %d \n', length(tx.dmrsSym));
```

```
REs for DMRS: 306
```

```
fprintf(1,'REs for PDSCH: %d \n', length(tx.pdschSym));
```

```
REs for PDSCH: 7032
```

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.

```
% Creating the MIMOMPChan object with all the AoAs, AoDs, gains
% arrays, delays and sampling rate.
chan = MIMOMPChan('fsamp', tx.waveformConfig.SampleRate, 'dly', dly, 'gain', gainDir, ...
                  'txArr', arrgNB, 'rxArr', arrUE, 'aoaAz', aoaAz, 'aoaEl', aoaEl, ...
                   'aodAz', aodAz, 'aodEl', aodEl);
% y: output of channel
y = chan.step(x);
```

Add noise

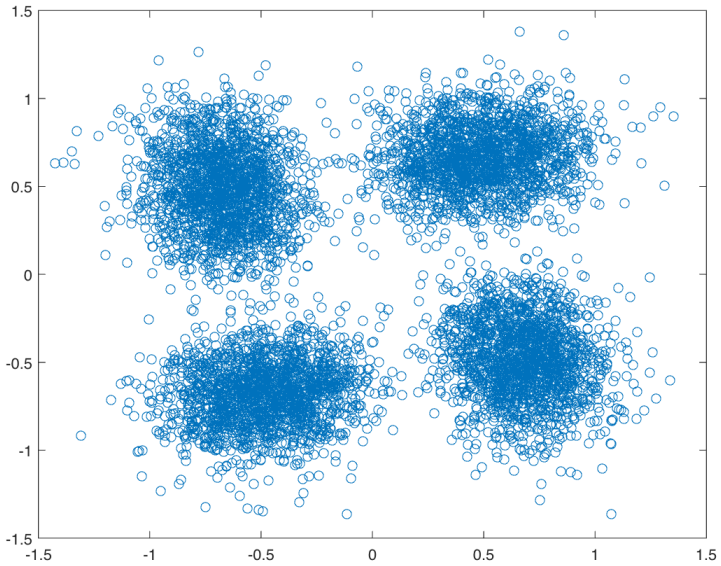
In multi-antenna receivers, the SNR is typically quoted as the SNR per antenna. Specifically, suppose that ynoisy = y + w. The SNR per antenna is Ely(t,j)I^2/Elw(t,j)I^2. Create a matrix ynoisy using the snrPerAntenna.

```
snr = 20;
[n,m] = size(y);
wvar = mean(abs(y(:)).^2)*10^(-0.1*snr);
ynoisy = y + sqrt(wvar/2)*(randn(n,m) + 1i*randn(n,m));
```

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.

```
% Creating a RX object with the correct rxBF vector
rx = NRUERx(simParam, 'rxBF', wrx);
rx.step(ynoisy);
% 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(Elrl^2 / El r - h*x l^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).

```
% 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 =    7.53
```

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.

```
% SnrTheory.
nscPerRB = 12;
nsc = simParam.carrierConfig.NSizeGrid * nscPerRB;
nantrx = prod(nantUE);
bwGain = 10*log10(tx.waveformConfig.Nfft/nsc);
snrTheory = snrPerAntenna + bwGain + 10*log10(nantrx);
fprintf(1,'SNR theoretical = %7.2f\n',snrTheory)
```

```
SNR theoretical =   19.28
```

Loading the Data (Test on real Data)

```
load roomPathData.mat
```

Select a RX Location

```
% Select an approximate RX position in the room
rxPos = [12,-6];   % A strong LOS point
% rxPos = [13,-12];  % A strong NLOS point
%rxPos = [15,-14];  % A weak NLOS point
%rxPos = [16,-15];  % A very weak NLOS point

% Find the index in the ray tracing data closest to the desired position
d = sum((pathData.rxPos - rxPos).^2, 2);
[dmin, indRx] = min(d);
```

Reading the Data

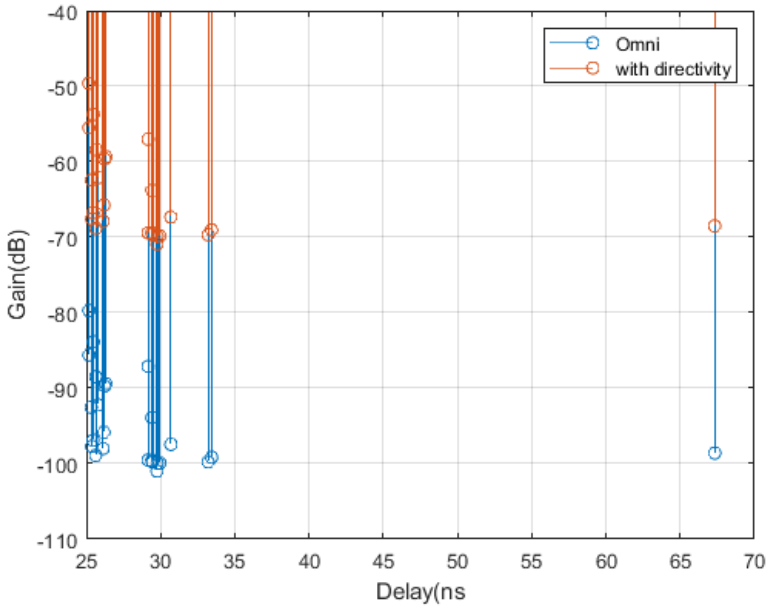
```
gain = pathData.gain(indRx,:);
aoaAz  = pathData.aoaAz(indRx,:);
aoaEl = 90-pathData.aoaEl(indRx,:);
aodAz  = pathData.aodAz(indRx,:);
aodEl = 90-pathData.aodEl(indRx,:);
dly = pathData.dly(indRx,:);
```

We now plot the TX and RX position

```
clf;
noDataValue = 3;  % Use this value for positions with no data

% Legend labels
legendLabels = {'Outage', 'LOS', 'NLOS', 'NoData'};

% Create a mapper for plotting
mapper = DataMapper(pathData.rxPos, pathData.posStep);

% Call the plot.  Use `plotHold=true` to hold the plot so we can overlay
% other items
mapper.plot(pathData.linkState, 'noDataValue', noDataValue, ...
    'LegendLabels', legendLabels, 'plotHold',true);

% Overlay the TX position
txPos = pathData.txPos;
plot(txPos(1), txPos(2), 'ro', 'MarkerSize', 10, 'DisplayName', 'TX');
plot(pathData.rxPos(indRx,1), pathData.rxPos(indRx,2), 'rx', ...
    'MarkerSize', 10, 'DisplayName', 'RX');
hold off;


% Display the legend
legend('Location', 'SouthEast');
```

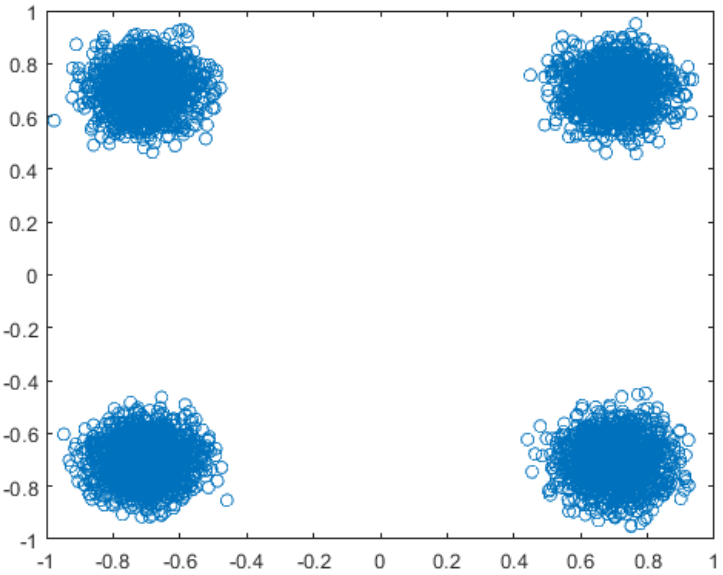
Creating the Antenna Elements

```
velgNB = [30,0,0];
velUE = [30,0,0];
arrgNB = ArrayWithAxes('elem', elemInterp, 'arr', arrgNB0, 'vel', velgNB, 'fc', fc);
arrUE  = ArrayWithAxes('elem', elemInterp, 'arr', arrUE0, 'vel', velUE, 'fc', fc);
[~,im] = max(gain);
arrUE.alignAxes(aoaAz(im),aoaEl(im));
arrgNB.alignAxes(aodAz(im),aodEl(im));
[utx, elemGainTx] = arrgNB.step(aoaAz, aoaEl);
[urx, elemGainRx] = arrUE.step(aodAz, aodEl);

wtx = conj(utx(:,im));
wtx = wtx / norm(wtx);
wrx = conj(urx(:,im));
wrx = wrx / norm(wrx);
AFgNB = 10*log10( abs(sum(wtx.*utx, 1)).^2 );
AFUE = 10*log10( abs(sum(wrx.*urx, 1)).^2 );

gainDir = gain + AFgNB + AFUE;
stem(dly/1e-9,[gain;gainDir]','BaseValue', -40);
grid on;
xlabel('Delay(ns')
ylabel('Gain(dB)')
legend('Omni', 'with directivity');
```




```
% generatin tx signal
```

```
tx = NRgNBTx(simParam);
tx.txBF = wtx;
x = tx.step();

chan = MIMOMPChan('fsamp', tx.waveformConfig.SampleRate, 'dly', dly, 'gain', gainDir, ...
                  'txArr', arrgNB, 'rxArr', arrUE, 'aoaAz', aoaAz, 'aoaEl', aoaEl, ...
                   'aodAz', aodAz, 'aodEl', aodEl);
y = chan.step(x);
%snr = 20;
[n,m] = size(y);
wvar = mean(abs(y(:)).^2)*10^(-0.1*snrPerAntenna);
ynoisy = y + sqrt(wvar/2)*(randn(n,m) + 1i*randn(n,m));
rx = NRUERx(simParam, 'rxBF', wrx);
rx.step(ynoisy);
plot(real(rx.pdschSymEq),imag(rx.pdschSymEq),'o')
```



Measure the SNR

```
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 =   19.61
```

```
% TODO:  Compute and print snrTheory.
nscPerRB = 12;
nsc = simParam.carrierConfig.NSizeGrid * nscPerRB;
nantrx = prod(nantUE);
bwGain = 10*log10(tx.waveformConfig.Nfft/nsc);
snrTheory = snrPerAntenna + bwGain + 10*log10(nantrx);
fprintf(1,'SNR theoretical = %7.2f\n',snrTheory)
```

```
SNR theoretical =   19.28
```

Till now we align beamforming vector to the max gain path now let's fix the TX antenna at some fixed angle and vary the rotate the RX, calculate the post equalization SNR to see the impact

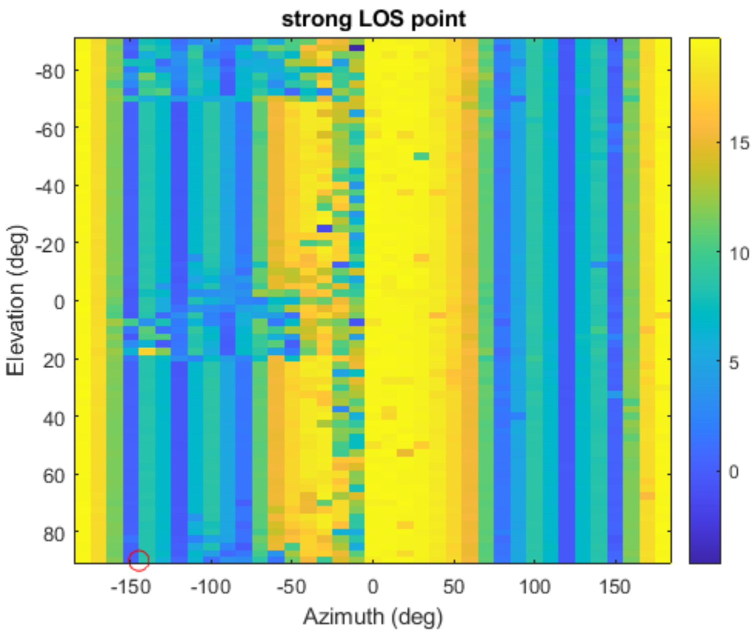
```
% different rotation angle to test
az = (-180:5:180);
el = (-90:5:90);

naz = length(az);
nel = length(el);

npath = length(gain);

% Creating the array
arrgNB = ArrayWithAxes('elem', elemInterp, 'arr', arrgNB0, 'vel', velgNB, 'fc', fc);
arrUE  = ArrayWithAxes('elem', elemInterp, 'arr', arrUE0, 'vel', velUE, 'fc', fc);

% Aligning the tx at some fixed location along the max gain path
```

```
arrgNB.alignAxes(aodAz(im),aodEl(im));


snr = zeros(naz, nel); % to hold the snr values

for i = 1:naz
    for j = 1:nel
        arrUE.alignAxes(az(i),el(j));
        [utx, elemGainTx] = arrgNB.step(aoaAz, aoaEl);
        [urx, elemGainRx] = arrUE.step(aodAz, aodEl);

        wtx = conj(utx(:,im));
        wtx = wtx / norm(wtx);
        wrx = conj(urx(:,im));
        wrx = wrx / norm(wrx);
        %AFgNB = 10*log10( abs(sum(wtx.*utx, 1)).^2 );
        %AFUE = 10*log10( abs(sum(wrx.*urx, 1)).^2 );

        %gainDir = gain + AFgNB + AFUE;

        % generatin tx signal
        tx = NRgNBTx(simParam);
        tx.txBF = wtx;
        x = tx.step();

        chan = MIMOMPChan('fsamp', tx.waveformConfig.SampleRate, 'dly', dly, 'gain', gainDir, ...
                     'txArr', arrgNB, 'rxArr', arrUE, 'aoaAz', aoaAz, 'aoaEl', aoaEl, ...
                     'aodAz', aodAz, 'aodEl', aodEl);
        y = chan.step(x);

        [n,m] = size(y);
        wvar = mean(abs(y(:)).^2)*10^(-0.1*snrPerAntenna);
        ynoisy = y + sqrt(wvar/2)*(randn(n,m) + 1i*randn(n,m));


        rx = NRUERx(simParam, 'rxBF', wrx);
        rx.step(ynoisy);

        Eerr = mean(abs(rx.pdschSymRaw - rx.pdschChanEst.*tx.pdschSym).^2);
        Erx = mean(abs(rx.pdschSymRaw).^2);
        snrEq = 10*log10(Erx/Eerr);

        snr(i,j) = snrEq;
    end
end
```

```
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
```

```
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
Warning: Query data is in MESHGRID format, NDGRID format will yield better performance.
Convert your query data (Xq, Yq) as follows:
   F = griddedInterpolant(. . .);
   Xq = Xq'; Yq = Yq';
   Vq = F(Xq,Yq);
```

```
imagesc(az,el,snr);
colorbar();
hold on;
plot(aoaAz(im),aoaEl(im), 'ro', 'Markersize', 10);
hold off;
xlabel('Azimuth (deg)');
ylabel('Elevation (deg)');

 title(sprintf('strong LOS point'));
```



```
classdef ArrayWithAxes < matlab.System
    % ArrayWithAxes.  Class containing an antenna array and axes.
    properties

        fc = 28e9;  % Carrier frequency

        % Element within each array.  Empty indicates to use an
        % isotropic element.  If non-empty, the object must be derived
        % from the matlab.System class with a step method
        %    dir = elem.step(fc,az,el)
        % that provides the directivity in dBi as a function of the
        % frequency and angles (az,el)
        elem = [];

        % Antenna array.  Empty indicates that there is only one element
        arr = [];

        % Steering vector object
        sv = [];

        % Azimuth and elevation angle of the element peak directivity
        axesAz = 0;
        axesEl = 0;

        % Axes of the element local coordinate frame of reference
        axesLoc = eye(3);

        % Velocity vector in 3D in m/s
        vel = zeros(1,3);
    end

    methods
        function obj = ArrayWithAxes(varargin)
            % Constructor

            % Set key-value pair arguments
            if nargin >= 1
                obj.set(varargin{:});
            end
        end

        function alignAxes(obj,az,el)
            % Aligns the axes to given az and el angles

            % Set the axesAz and axesEl to az and el
            obj.axesAz = az;
            obj.axesEl = el;

            % Creates axes aligned with az and el
            obj.axesLoc = azelaxes(az,el);
        end
```

```
        function dop = doppler(obj,az,el)
            % Computes the Doppler shift of a set of paths
            % The angles of the paths are given as (az,el) pairs
            % in the global frame of reference.

            % Finds unit vectors in the direction of each path
            npath = length(el);
            [u1,u2,u3] = sph2cart(deg2rad(az),deg2rad(el),ones(1,npath));
            u = [u1; u2; u3];

            % Compute the Doppler shift of each path via an inner product
            % of the path direction and velocity vector.
            vcos = obj.vel*u;
            vc = physconst('lightspeed');
            dop = vcos*obj.fc/vc;

        end

        function releaseSV(obj)
            % Creates the steering vector object if it has not yet been
            % created.  Otherwise release it.  This is needed since the
            % sv object requires that it takes the same number of
            % inputs each time.
            if isempty(obj.sv)
                obj.sv = phased.SteeringVector('SensorArray',obj.arr);
            else
                obj.sv.release();
            end

        end


    end

    methods (Access = protected)

        function setupImpl(obj)
            % setup:  This is called before the first step.

            % Creating the steering vector object using the array.
            obj.sv = phased.SteeringVector('SensorArray', obj.arr);
        end

        function releaseImpl(obj)
            % release:  Called to release the object
            obj.elem.release();
            obj.sv.release();
        end

       function [u, elemGain] = stepImpl(obj, az, el)
```

```
            % Gets steering vectors and element gains for a set of angles
            % The angles az and el should be row vectors along which
            % the outputs are to be computed.
            %Sv0 = obj.sv(obj.fc, [az; el]);

            % Release the SV
            obj.releaseSV();

            % Converting the global angles (az, el) to local
            % angles (azLoc, elLoc).  Use the
            % global2localcoord() method with the 'ss' option.
            uglobal = [az' el' ones(length(az),1)]';
            ulocal = global2localcoord(uglobal, 'ss', zeros(3,1), obj.axesLoc);
            azLoc = ulocal(1,:);
            elLoc = ulocal(2,:);

            % Getting the SV in the local coordinates
            u = obj.sv(obj.fc, [azLoc; elLoc]);

            % Getting the directivity gain of the element from the local angles.
             elemGain = obj.elem.step(elLoc,azLoc)';
        end

    end


end
```

```
classdef DataMapper < matlab.mixin.SetGet
    % Class for mapping data for a grid

    properties
        posStep;  % Grid spacing in each direction

        % Data at position n will be mapped to grid point
        % (posInd(n,1), posInd(n,2))
        posInd;   % Index of the

        % Number of steps in each axis
        nsteps;

        % Position along each axis
        posx, posy;

    end

    methods
        function obj = DataMapper(pos, posStep)
            % Constructor
            %
            % pos is a n x 2 list of all possible positions.
            % posStep is the 1 x 2 grid spacing in each direction

            % Compute the indices that each position gets mapped to
            posMin = min(pos);
            posMax = max(pos);
            obj.posStep = posStep;
            obj.nsteps = (posMax - posMin)./posStep + 1;
            obj.posInd = round((pos-posMin)./posStep)+1;

            % Compute the x and y positions for the imagesc function
            obj.posx = posMin(1) + (0:obj.nsteps(1)-1)*posStep(1);
            obj.posy = posMin(2) + (0:obj.nsteps(2)-1)*posStep(2);
        end

        function plot(obj, data, options)
            % Plots data with the positions
            %
            % Data is a vector of size n x 1.  The function will then
            % plot the value of data(i) in pos(i,1), pos(i,2).
            %
            % It can also optionally create dummy legend labels that can
            % then be displayed with the legend command
            arguments
                obj
                data
                options.legendValues = []
                options.legendLabels = []
                options.plotHold logical = false
```

```
            options.addLegend logical = false
            options.legendLocation string = 'southeast'
            options.noDataValue = 0
        end


        % Create an image array for the data
        posImage = repmat(options.noDataValue, ...
            obj.nsteps(2), obj.nsteps(1));
        for i = 1:length(data)
            posImage(obj.posInd(i,2), obj.posInd(i,1)) = data(i);
        end

        % Plot the data
        imagesc(obj.posx, obj.posy, posImage);
        hold on;

        % Plot a dummy rectangle for each color
        if ~isempty(options.legendLabels)
            nlabels = length(options.legendLabels);
            colorArr = parula(nlabels);
            for i = 1:nlabels
                fill([1,1],[1,1], colorArr(i,:), 'DisplayName', ...
                    options.legendLabels{i});
            end
        end

        % Turn off the plot hold if not requested
        if ~options.plotHold
            hold off;
        end

        % Create the legend
        if options.addLegend
            legend('Location', options.legendLocation);
        end
    end
  end
end
```

```
classdef InterpPatternAntenna < matlab.System
    % Wrapper class for an antenna to perform smooth interpolation of the
    % directivity.  This class is necessary since the antenna toolbox
    % generally performs nearest neighbor interpolation which may not
    % be smooth
    properties

        ant;  % Base antenna.  Must support a pattern method
        fc;   % Frequency
        dirInterp;  % Gridded interplant
    end

    methods
        function obj = InterpPatternAntenna(ant, fc, varargin)
            % Constructor
            obj.ant = ant;
            obj.fc = fc;

            % Set key-value pair arguments
            if nargin >= 1
                obj.set(varargin{:});
            end
        end
    end

    methods (Access = protected)

        function setupImpl(obj)
            % setup:  This is called before the first step.
            % We will use this point to create the interpolator.

            % Getting the pattern from ant.pattern
            [elemGain,az,el] = obj.ant.pattern(obj.fc,'Type', 'Directivity');
            % Creating the gridded interpolant object.
             obj.dirInterp = griddedInterpolant({el,az},elemGain);

        end
        function dir = stepImpl(obj, az, el)
            % Computes the directivity along az and el angles in the local
            % reference frame

            % Running the interplationn object to compute the directivity in the local↙
angles
            dir = obj.dirInterp(el,az);
        end
    end
end
```

```
function [hest, w] = kernelReg(ind,hestRaw,nsc,len,sig)
% kernelReg:  Kernel regression with a RBF

% Create the RBF kernel
w = exp(-0.5*(-len:len).^2/sig^2)';

% Place the raw channel esimates in a vector at the locations
% of the indices.
%   y(ind(i))  = hestRaw(i)
%   y0(ind(i)) = 1
y = zeros(nsc,1);
y0 = zeros(nsc,1);
y(ind) = hestRaw;
y0(ind) = 1;

% Get the filter length
len = floor(length(w)/2);

% Filter both raw estimates and the indicators
[z1, z1f] = filter(w,1,y);
z1 = [z1(len+1:end); z1f(1:len)];
[z0, z0f] = filter(w,1,y0);
z0 = [z0(len+1:end); z0f(1:len)];

% Compute the channel estimate
hest = z1./max(1e-8,z0);
end
```

```
classdef MIMOMPChan < matlab.System
    % MIMOMPChan:  MIMO multi-path fading channel
    properties
        fsamp;   % Sample rate in Hz

        % TX and RX antenna arrays
        txArr, rxArr;

        % Path properties
        aoaAz, aoaEl; % Angles of arrival in degrees
        aodAz, aodEl; % Angles of departure in degrees
        gain;  % path gains in dB
        dly;   % delays in seconds
        dop;   % doppler shift of each path in Hz

        % Fractional delay object
        fracDly;

        % Initial set of phases for the next step call
        phaseInit;

        % Previous gains and steering vectors
        utx, urx;
        gainTx, gainRx;


    end

    methods
        function obj = MIMOMPChan(varargin)
            % Constructor:
            % The syntax allows you to call the constructor with syntax of
            % the form:
            %
            %     chan = SISOMPChan('Prop1', Val1, 'Prop2', val2, ...);
            if nargin >= 1
                obj.set(varargin{:});
            end

        end

    end
    methods (Access = protected)
        function setupImpl(obj)
              % setup:  This is called before the first step.

              % Create a dsp.VariableFractionalDelay object
              obj.fracDly = dsp.VariableFractionalDelay(...
                'InterpolationMethod', 'Farrow','FilterLength',8,...
                'FarrowSmallDelayAction','Use off-centered kernel',...
                'MaximumDelay', 1024);
```

```
        end

        function resetImpl(obj)
            % reset:  Called on the first step after reset or release.

            % Reset the fracDly object
            obj.fracDly.reset();

            % Initialize phases, phaseInit, to a row vector of
            % dimension equal to the number of paths with uniform values
            % from 0 to 2pi
            npath = length(obj.gain);
            obj.phaseInit = 2*pi*rand(1,npath);
        end

        function releaseImpl(obj)
            % release:  Called after the release method

            % Release the fracDly object
            obj.fracDly.release();
        end

        function y = stepImpl(obj, x)
            % step:  Run samples through the channel
            % The input, x, should be nsamp x nanttx,


            % The delay in samples
            dlySamp = obj.dly;

            % Getting the TX steering vectors and element gains
            % along the angles of departure using the
            [obj.utx, obj.gainTx] = obj.txArr.step(obj.aoaAz, obj.aoaEl);
            [obj.urx, obj.gainRx] = obj.rxArr.step(obj.aodAz, obj.aodEl);

            % Computing the total gain along each path in linear scale
            gainLin = db2mag(obj.gain + obj.gainTx + obj.gainRx);

            % Initialize variables
            nsamp  = size(x,1);
            nantrx = size(obj.urx,1);
            npath = length(obj.dly);
            y = zeros(nsamp,nantrx);

            % Getting the Doppler shift of each path from the TX and RX
            obj.dop = obj.txArr.doppler(obj.aoaAz, obj.aoaEl)+ obj.rxArr.doppler(obj.↙
aodAz, obj.aodEl);
            % Using the Doppler shifts, compute the phase rotations
            % on each path.  Specifically, if nsamp = length(x), create a
            % (nsamp+1) x npath matrix
            %     phase(i,k) = phase rotation on sample i and path k
```

```
            nsamp = length(x);
            t = (0:1:nsamp)';
            t = t/obj.fsamp;
            oneMat = ones(nsamp+1,1);
            phase = oneMat*obj.phaseInit - t*obj.dop*2*pi;
            % Save the final phase, phase(nsamp+1,:)
            % as phaseInit for the next step.
            obj.phaseInit = phase(nsamp+1,:);
            % Loop over the paths
            for ipath = 1:npath

                % Computing the transmitted signal, x, along the
                % TX spatial signature for path ipath.
                %   z = nsamp x 1 vetor
                z = x*obj.utx(:,ipath);


                % Delaying the path by the dlySamp(ipath) using the fractional delay↙
object
                zdly = obj.fracDly(z,dlySamp(ipath));

                % Multiplying by the gain
                zdly = zdly*gainLin(ipath);

                % Phase rotation
                z1 = exp(1i*phase(1:nsamp,ipath)).*zdly;

                % Multiply by the RX spatial signature and add to y
                y = y + z1*obj.urx(:,ipath)';
            end

        end
    end
end
```

```
classdef NRgNBTx < matlab.System
    % 5G NR gNB transmitter class
    properties
        carrierConfig;  % Carrier configuration
        pdschConfig;     % Default PDSCH config
        waveformConfig;  % Waveform config


        % Coded bits transmitted on PDSCH
        txBits;         % Cell array of TX bits

        % OFDM grids
        ofdmGridLayer;      % Before pre-coding nsc x nsym x nlayers
        ofdmGridAnt;        % Before pre-coding nsc x nsym x nantennas

        % OFDM grid to visualize the type of symbols
        ofdmGridChan;

        % Transmitted data in last slots
        bits;           % TX bits
        pdschSym;       % TX symbols
        dmrsSym;        % TX data symbols

        % Channel
        chanNames;

        % Slot number
        Nslot = 0;

        % TX beamforming vector.  This is fixed.
        txBF;

        % HARQ Process
        %nharq = 8;      % number of HARQ processes

    end

    properties (Constant)
        % Indices for ofdmGridChan indicating the type of symbol


    end

    methods
        function obj = NRgNBTx(simParam, varargin)
            % Constructor

            % Get parameters from simulation parameters
            % Many 5G Toolbox routines do not take classes, the
            % objects need to be converted to older structures.
            obj.carrierConfig = simParam.carrierConfig;
```

```
            obj.pdschConfig = simParam.pdschConfig;
            obj.waveformConfig =  simParam.waveformConfig;

            % Set parameters from constructor arguments
            if nargin >= 1
                obj.set(varargin{:});
            end
        end
      function setAck(obj, iharq)
            % Set that the HARQ transmission was received correctly
            obj.newDataAvail(iharq) = 1;
      end
    end
    methods (Access = protected)

        function x = stepImpl(obj)
            % step implementation.  Creates one slot of samples

            % Set the slot number if the PDSCH config
            obj.carrierConfig.NSlot = obj.Nslot;

            % Create the PDSCH grid before pre-coding
            nscPerRB = 12;
            nsc = obj.carrierConfig.NSizeGrid * nscPerRB;
            obj.ofdmGridLayer = zeros(nsc, ...
                obj.waveformConfig.SymbolsPerSlot, ...
                obj.pdschConfig.NumLayers);
            obj.ofdmGridChan = zeros(nsc, ...
                obj.waveformConfig.SymbolsPerSlot, ...
                obj.pdschConfig.NumLayers);

            % Get information for PDSCH and DM-RS allocations
            pdschIndices = nrPDSCHIndices(obj.carrierConfig, obj.pdschConfig);
            dmrsIndices = nrPDSCHDMRSIndices(obj.carrierConfig, obj.pdschConfig);
            obj.dmrsSym = nrPDSCHDMRS(obj.carrierConfig, obj.pdschConfig);

            % Get the PT-RS symbols and indices and insert them
            % in the TX grid
            ptrsSym = nrPDSCHPTRS(obj.carrierConfig, obj.pdschConfig);
            ptrsInd = nrPDSCHPTRSIndices(obj.carrierConfig, obj.pdschConfig);
            % Generate random bits
            bitsPerSym = 2;
            nsym = length(pdschIndices);
            nbits = bitsPerSym * nsym;
            obj.bits = randi([0 1], nbits, 1);
            obj.txBits = obj.bits;

            % Modulate the bits to symbols
            %M = nr.NRConst.modOrder(obj.pdschConfig.Modulation);
            M = 2^bitsPerSym;
            obj.pdschSym = qammod(obj.bits,M,'InputType','bit',...
```

```
            'UnitAveragePower',true);

        % Map symbols to OFDM grid
        obj.ofdmGridLayer(pdschIndices) = obj.pdschSym;
        obj.ofdmGridLayer(dmrsIndices) = obj.dmrsSym;
        % Map PT-RS symbols and indices and insert them
        % in the OFDM grid

        obj.ofdmGridLayer(ptrsInd) = ptrsSym;


        % Fill the channel with labels of the channels.
        % This is just for visualization
        obj.ofdmGridChan(pdschIndices) = 1;
        obj.ofdmGridChan(dmrsIndices) = 2;
        obj.ofdmGridChan(ptrsInd) = 3;
        obj.chanNames = {'Other', 'PDSCH', 'DM-RS', 'PT-RS'};
        % Perform the OFDM modulation
        xlayer = nrOFDMModulate(obj.carrierConfig, obj.ofdmGridLayer);

        % TX beamforming:  At this point,
        % xlayer will be an nsamp x 1 vector.  Use the TX beamforming
        % vector, obj.txBF, to map this to a  nsamp x nant matrix
        % where nant is the number of TX antennas.

        x = xlayer*obj.txBF';
        % Increment the slot number
        obj.Nslot = obj.Nslot + 1;


    end

  end
end
```

```
classdef NRUERx < matlab.System
    % 5G NR gNB transmitter class
    properties
        carrierConfig;  % Carrier configuration
        pdschConfig;     % Default PDSCH config
        waveformConfig;  % Waveform config

        % OFDM grid b
        ofdmGrid;     % Before pre-coding nsc x nsym x nlayers

        % Channel and noise estimate
        noiseEst;
        chanEstGrid;
        % Channel estimation parameters
        sigFreq = 7;  % Channel smoothing in freq
        sigTime = 3;  % Channel smoothing in time
        lenFreq = 21;  % Filter length in freq
        Wtime;

        % Recived symbols
        pdschChanEst;    % Channel estimate on the PDSCH
        pdschSymRaw;     % Raw symbols before equalization
        pdschSymEq;      % Equalized symbols
        dmrsSym;         % DM-RS Reference symbols

        % Slot number
        Nslot = 0;

        % Test bit parameters
        bitsPerSym = 2;

        % RX beamforming vector.
        rxBF;

        % Timing offset
        offset;
        rxBits;        % RX bits
    end

    methods
        function obj = NRUERx(simParam, varargin)
            % Constructor

            % Get parameters from simulation parameters
            % Many 5G Toolbox routines do not take classes, the
            % objects need to be converted to older structures.
            obj.carrierConfig = simParam.carrierConfig ;
            obj.pdschConfig = simParam.pdschConfig ;
            obj.waveformConfig = simParam.waveformConfig;

            % Set parameters from constructor arguments
```

```
        if nargin >= 1
            obj.set(varargin{:});
        end

    end

    function chanEst(obj, rxGrid)
        % Computes the channel estimate

        % Getting the TX DM-RS symbols and indices
        dmrsSymTx = nrPDSCHDMRS(obj.carrierConfig, obj.pdschConfig);
        dmrsInd = nrPDSCHDMRSIndices(obj.carrierConfig, obj.pdschConfig);

        % Getting RX symbols on the DM-RS
        dmrsSymRx = rxGrid(dmrsInd);

        % Getting the raw channel estimate
        chanEstRaw = dmrsSymRx ./ dmrsSymTx;

        % Getting the symbol numbers and sub-carrier indices of the
        % DM-RS symbols from the DM-RS

        [nsc, nsym, nlayers] = size(rxGrid);
        dmrsSymNum = floor(double(dmrsInd-1) / nsc)+1;
        %   dmrsSymNum(i) = symbol number for the i-th DM-RS symbol
        dmrsScInd = mod((dmrsInd-1), nsc)+1;
        %   dmrsScInd(i) = sub-carrier index for the i-th DM-RS symbol

        % Getting the list of all symbol numbers on which DM-RS was
        % transmitted using the unique command
        dmrsSymNums = unique(dmrsSymNum);
        ndrmsSym = length(dmrsSymNums);

        % We first compute the channel and noise
        % estimate on each of the symbols on which the DM-RS was
        % transmitted.  We will store these in two arrays
        chanEstDmrs = zeros(nsc, ndrmsSym);
        noiseEstDmrs  = zeros(ndrmsSym, 1);

        % Loop over the DM-RS symbols
        for i = 1:ndrmsSym

            I = find(dmrsSymNum == dmrsSymNums(i));

            % Getting the sub-carrier indices and raw channel
            % channel estimate for these RS on the symbol
            ind = dmrsScInd(I);
            raw = chanEstRaw(I);

            % Using kernelReg to compute the channel estimate
            % on that DM-RS symbol.  Use the lenFreq and sigFreq
```

```
                % for the kernel length and sigma.
                chanEstDmrs(:,i) = kernelReg(ind, raw, nsc, ...
                    obj.lenFreq, obj.sigFreq);

                % Computing the noise estimate on the symbol
                % using the residual method
                noiseEstDmrs(i) = mean(abs(raw- chanEstDmrs(ind,i)).^2);

            end
            % Finding the noise estimate over the PDSCH by
            % averaging noiseEstDmrs
            obj.noiseEst = mean(noiseEstDmrs);

            % Finally, we interpolate over time.
            % We will use an estimate of the form
            %    obj.chaneEstGrid = chanEstDrms*W
            % so that
            %    chanEstGrid(k,j) = \sum_i chanEstDmrs(k,i)*W(i,j)
            %
            % We use a kernel estimator
            %
            %     W(i,j) = W0(i,j) / \sum_k W0(k,j)
            %     W0(k,j) = exp(-D(k,j)^2/(2*obj.sigTime^2))
            %     D(k,j) = dmrsSymNum(k) - j
            %
            D = dmrsSymNums - (1:nsym);
            W0 = exp(-D.^2/(2*obj.sigTime^2));
            W = W0 ./ sum(W0,1);

            % Save the time interpolation matrix
            obj.Wtime = W;

            % Create the channel estimate grid
            obj.chanEstGrid = chanEstDmrs*W;

        end
    end
    methods (Access = protected)

        function rxBits = stepImpl(obj, y)

            % RX beamforming by multiplying y with the
            % the RX BF vector.
            %   z = ...
            z = y*obj.rxBF;

            % Get information for PDSCH and DM-RS allocations
            %[pdschIndices,dmrsIndices,dmrsSymbols,pdschIndicesInfo] = ...
            %    nr.hPDSCHResources(obj.carrierConfig, obj.pdschConfig);
            [pdschIndices,pdschIndicesInfo] = nrPDSCHIndices(obj.carrierConfig, obj.↙
pdschConfig);
```

```
            dmrsIndices = nrPDSCHDMRSIndices(obj.carrierConfig, obj.pdschConfig);
            dmrsSymbols = nrPDSCHDMRS(obj.carrierConfig, obj.pdschConfig);

            % Demodulate the RX signal
            obj.ofdmGrid = nrOFDMDemodulate(obj.carrierConfig, z);

            % Get channel estimate.
            obj.chanEst(obj.ofdmGrid);

            % Extract raw symbols and channel estimate on PDSCH
            obj.pdschSymRaw = obj.ofdmGrid(pdschIndices);
            obj.pdschChanEst = obj.chanEstGrid(pdschIndices);
            obj.pdschSymEq = obj.pdschSymRaw ./ obj.pdschChanEst;

            % Demodulate the symbols
            M = 2^obj.bitsPerSym;
            rxBits = qamdemod(obj.pdschSymEq, M, 'OutputType', 'bit',...
                'UnitAveragePower', true);
        end

    end
end
```

```
classdef PDSCHSimParam < matlab.mixin.SetGet
    % PDSCHSimParam:  Parameters for the PDSCHSimParam
    properties

        % Settable parameters.  These should be set in the constructor
        NLayers = 1;    % number of layers
        Modulation = 'QPSK';
        NRB = 51;  % number of resource blocks
        SubcarrierSpacing = 120;  % SCS in kHZ
        fc = 28e9;  % carrier frequency in Hz

        % Carrier parameters.  Right now, we assume only one BWP
        carrierConfig;

        % PDSCH parameters
        pdschConfig;

        % Waveform parameters
        waveformConfig;

    end

    methods
        function obj = PDSCHSimParam(varargin)
            % Constructor

            % Set parameters from constructor arguments
            if nargin >= 1
                obj.set(varargin{:});
            end

            % Set the carrierParam settings
            obj.carrierConfig = nrCarrierConfig(...
                'NSizeGrid', obj.NRB, 'SubcarrierSpacing', obj.SubcarrierSpacing);
            %, ...'fc', obj.fc);

            % Compute the waveform parameters
            % We need to convert the carrier configuration to a structure
            obj.waveformConfig = nrOFDMInfo(obj.carrierConfig);

            % PDSCH parameters.  We allocate all the RBs and all
            % the OFDM symbols in the slot
            res.Symbols = 0;  % Reserve first symbol
            res.PRB = (0:obj.NRB-1);      % Reserve all RBs
            res.Period = 1;  % Reserve on every slot
            obj.pdschConfig = nrPDSCHConfig(...
                'Modulation', 'QPSK', ...
                'PRBSet', (0:obj.NRB-1), ...
                'SymbolAllocation', [1,obj.waveformConfig.SymbolsPerSlot-1], ...
                'EnablePTRS', 1, 'PTRS', nrPDSCHPTRSConfig());
                %'Reserved', res);
```

```
        end



    end
end
```