Beamforming in Multi-Path Channels at 28 GHz

Beamforming is a key component of many 5G technologies including massive MIMO and mmWave communication. In this lab, we will show how to simulate beamforming in frequency domain. We will use typical arrays for 28 GHz.

In going through this lab, you will learn to:

- Construct antenna elements and arrays
- Visualize the element gain pattern and array gain
- Normalize array patterns to account for mutual coupling
- Select beamforming vectors to align to particular paths
- Compute the MIMO channel response in an OFDM system
- Compute the gain in SNR due to element gain and beamforming
- Compare constant beamforming and instantaneous beamforming

Also, along with this file, you will have to make modifications for these files:

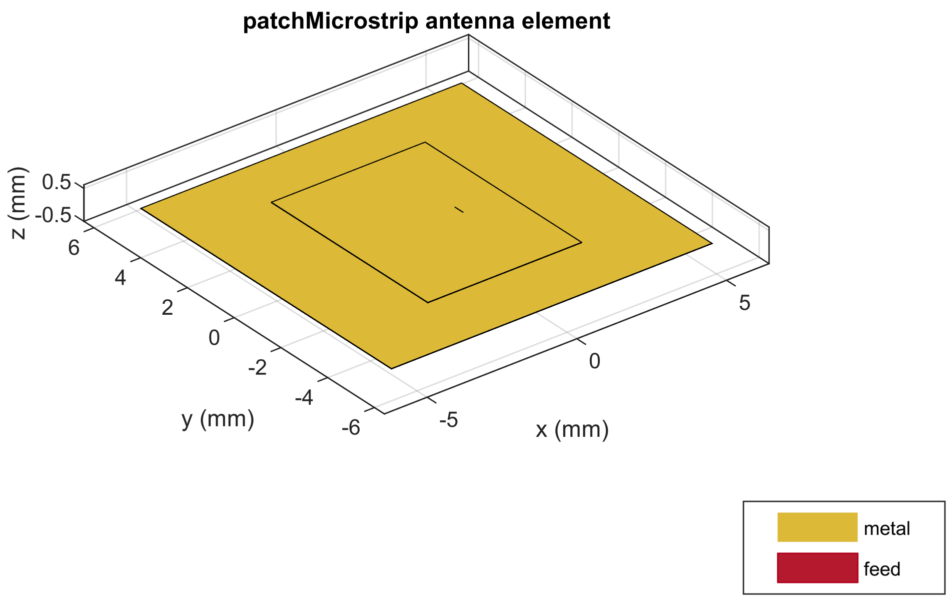
- `ArrayPlatform.m`: A class for an array platform that stores the element, array and local to global coordinate system mapping, It performs the normalization for antenna patterns
- `FDMIMOChan.m`: A class for frequency-domain MIMO channel simulation

Submission: Complete the files. Run the code. Print the outputs to PDF. Combine all files to a single PDF and submit the PDF. Do not submit any other source code.

Creating the Antenna Elements

We will use a simple patch microstrip antenna element which are typical at 28 GHz.

```
fc = 28e9;  % Carrier frequency
elem = design(patchMicrostrip, fc);
show(elem);
```

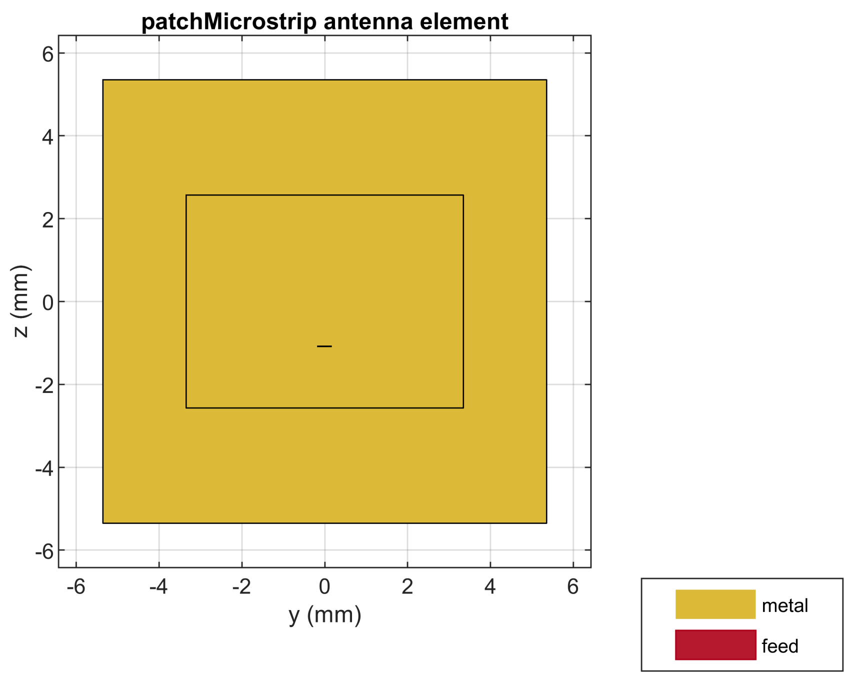


The element is constructed with the boresight in the positive z-axis. Use the `elem.Tilt` and `elem.TiltAxis` to rotate the element so that the boresight is along the x-axis.

```
% TODO:
elem.Tilt = 90;
elem.TiltAxis = [0, 1, 0];
```

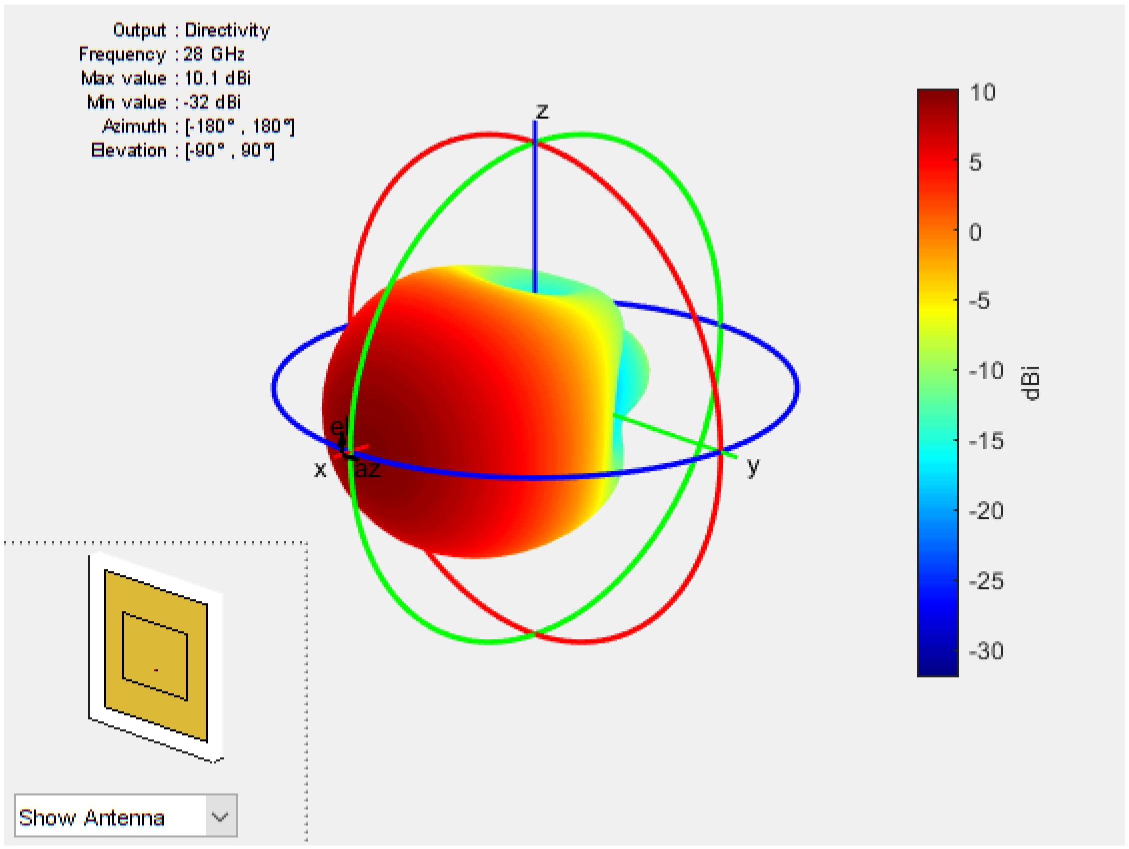
Use the `elem.show() and `view` commands to see the antenna element in 3D see the front face of the patch.

```
% TODO
elem.show();
view([90 0]);
```



Display the directivity pattern of the antenna with `elem.pattern` command. You should see that this is a highly directive antenna with 10.1 dBi gain.

```
% TODO
figure;
elem.pattern(fc);
```



Creating the Antenna Arrays

We next construct the arrays at the gNB and UE. Consruct two URA with the given array dimensions. Use a lambda / 2 separation.

```
% Number of antennas
nantgNB = [4,4];
nantUE = [2,2];
lambda = physconst("Lightspeed")/fc;

% TODO:  Construct a URA
arrgNB = phased.URA(nantgNB, lambda/2, 'Element',elem);
arrUE = phased.URA(nantUE, lambda/2, 'Element',elem);
```

Creating the Array Platform

I have created a skeleton class, `ArrayPlatform,` that performs several useful tasks for the array managment:

- Stores the element and array structure
- Computes the normalization matrix to account for mutual coupling
- Computes the normalized steering vectors in a given set of angles

The main function to be completed is `computeNormMatrix()` method.

```
% TODO:  Complete the computeNormMatrix() method in the ArrayPlatform class
```

Next, complete the `step` method that produces the steering vectors and element gain for a given set of angles.

```
% TODO:  Complete the step() method that computes the steering vectors and element gain.
```

Once you have completed this method, you can build the array platform with the functions

```
arrPlatformgNB = ArrayPlatform('elem', elem, 'arr', arrgNB, 'fc', fc);
arrPlatformgNB.computeNormMatrix();

arrPlatformUE = ArrayPlatform('elem', elem, 'arr', arrUE, 'fc', fc);
arrPlatformUE.computeNormMatrix();
```

We now display the array patterns for different steering angles. We will do this just for the gNB array. First, complete the following code that get beamforing vectors on two different angles.

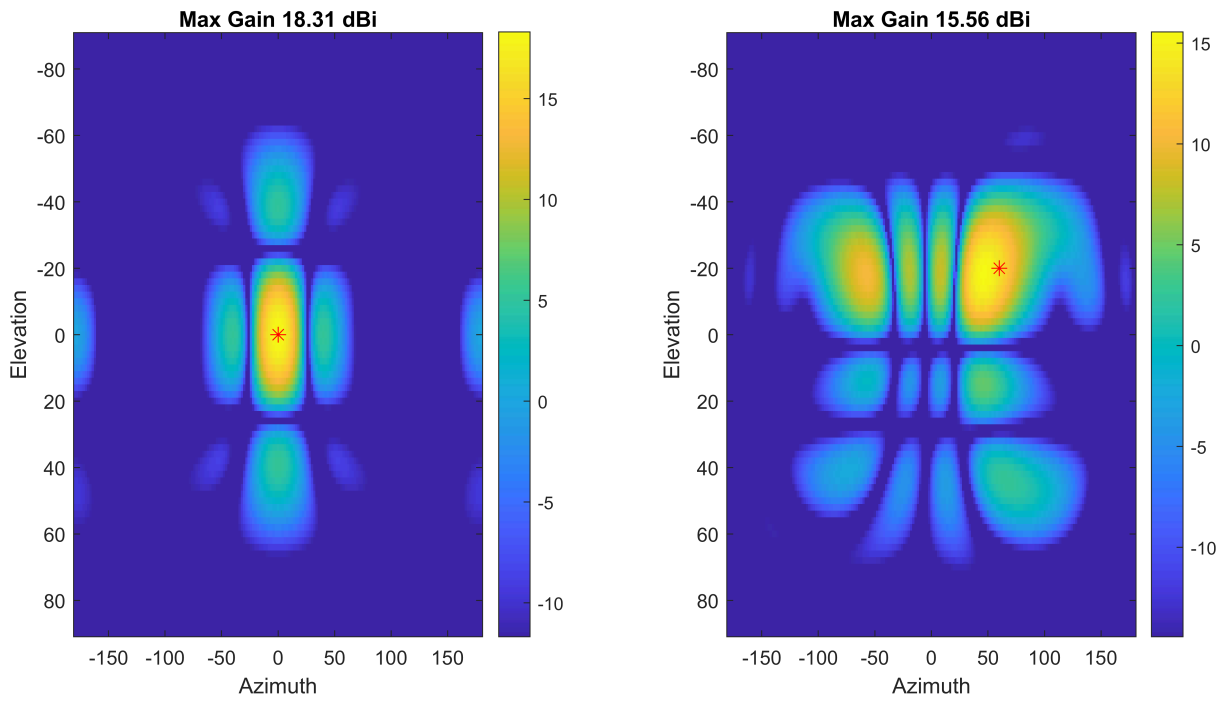
```
% Angles to get the beamforming vectors
az0 = [0, 60];
el0 = [0, -20];
nw = length(az0);

% TODO:  Get the steering vectors for the angles using the step function
[svTx, gains] = arrPlatformgNB.step(az0, el0, true);

% TODO:  Get the BF vectors by taking the conjugate of each SV
% and normalizing
w = zeros(size(svTx));
for i=1:nw
    w_b = conj(svTx(:,i));
    w(:,i) = w_b / norm(w_b);
end
disp(norm(w(:,1)));
```

```
1
```

Next, complete the following code to plot the array patterns.

```
% Angles to plot the array pattern
az = (-180:2:180)';
el = (-90:2:90)';
naz = length(az);
nel = length(el);

% Get grid of values
[azMat, elMat] = meshgrid(az, el);
azVal = azMat(:);
elVal = elMat(:);

% TODO:  Get the normalized steering vectors at each of the plot angles
Sv = arrPlatformgNB.step(azVal', elVal', true);
```

```
clf;
posOld = get(gcf, 'Position');
set(gcf,'Position', [0,0,1000,500]);

% Loop over the beamforming vectors
for i = 1:nw

    w_b = w(:, i);
    % TODO:  Find the array gain by taking the inner product of BF vector
    % w(:,i) with the steering vectors.  Convert the gain to dBi.
    gain = mag2db(abs(sum(w_b.*Sv,1)));

    % TODO:  Find the maximum gain
    gainMax = max(gain);

    % TODO:  Reshape the gain to a nel x naz matrix
    gain = reshape(gain,nel,naz);

    % TODO:  Plot the gain on a subplot. using imagesc()
    %   Set the color limits to [gainMax-30, gainMax]
    %   Add a colorbar
    %   Label the axes
    %   Place a marker on the intended maximum direction
    subplot(1,2,i);
    imagesc(az,el,gain,[gainMax-30 gainMax]); hold on;
    plot(az0(i), el0(i), "Marker","*","MarkerSize",8,"MarkerFaceColor",'r',"MarkerEdgeColor",'r
    title(sprintf("Max Gain %3.2f dBi", gainMax));
    xlabel("Azimuth"); ylabel("Elevation");
    colorbar();
end
```



Visualizing Beamforming Gains on the Channel Paths

We will now compute the gains on a multi-path channel. We use the 3GPP CDL-C model. The path parameters can be obtianed in MATLAB as follows.

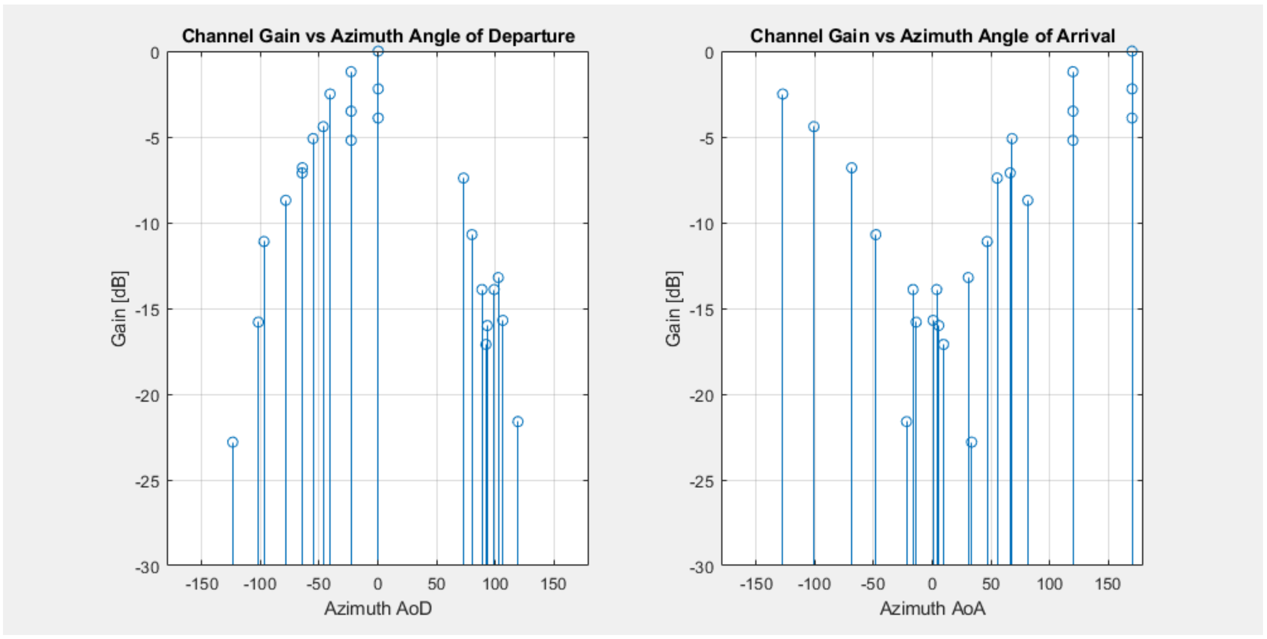
```
dlySpread = 100e-9;  % delay spread in seconds
chan = nrCDLChannel('DelayProfile','CDL-C',...
    'DelaySpread',dlySpread, 'CarrierFrequency', fc, ...
    'NormalizePathGains', true);
chaninfo = info(chan);

gain = chaninfo.AveragePathGains';
aoaAz  = chaninfo.AnglesAoA';
aoaEl = 90-chaninfo.AnglesZoA';
aodAz  = chaninfo.AnglesAoD';
aodEl = 90-chaninfo.AnglesZoD';
dly = chaninfo.PathDelays';
```

To visualize the gains, create two stem plots.

- Gain vs. aodAz
- Gain vs. aoaAz

Use the subplots to put the gains side by side. Also, set the `BaseValue` to -30 to clip very low gains. This is the omni-directional gains before element or beamforming gain has been applied.

```
% TODO
clf;
set(gcf,'Position', [0,0,1000,500]);
subplot(1,2,1);
stem(aodAz, gain, "BaseValue",-30); grid on;
xlabel("Azimuth AoD"); ylabel("Gain [dB]"); xlim([-180 180]);
title("Channel Gain vs Azimuth Angle of Departure");
subplot(1,2,2);
stem(aoaAz, gain, "BaseValue",-30); grid on;
xlabel("Azimuth AoA"); ylabel("Gain [dB]");
title("Channel Gain vs Azimuth Angle of Arrival"); ylim([-30, 0]); xlim([-180 180]);
```



We now rotote the UE to an angle of 135 degrees which is close to the main angle of arrival.

```
azUE = 135;
elUE = -10;
arrPlatformUE.alignAxes(azUE, elUE);
```

Next, get the TX element gains and steering vectors along the AoD and the RX element gains and steering vectors along the AoAs.

```
% TODO:
[svTx, elemGainTx] = arrPlatformgNB.step(aodAz',aodEl',true);
[svRx, elemGainRx] = arrPlatformUE.step(aoaAz',aoaEl',true);
```

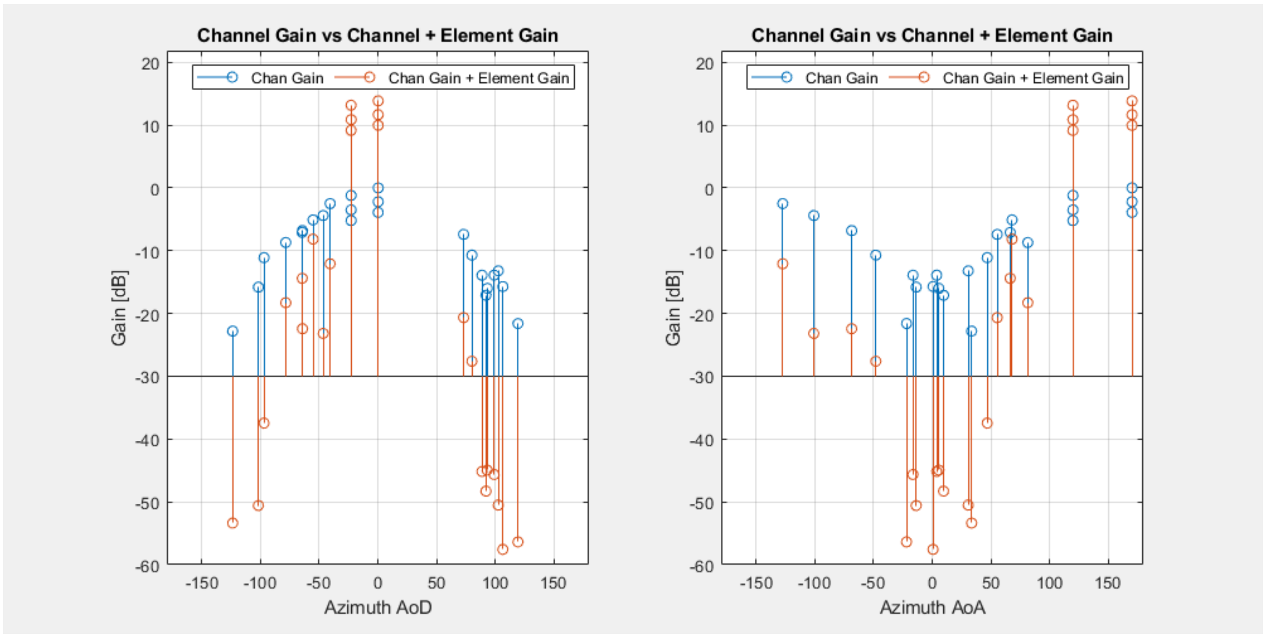
Get the gains along the paths with the element gains but without any array gain.

```
% TODO:
gainElem = gain + elemGainTx' + elemGainRx';
```

Create the same stem plot as before, but on each subplot, plot both `gain` and `gainElem` to see the effect of the element gains. You will see that a few paths are much more pronounced in gain.

```
% TODO
clf;
set(gcf,'Position', [0,0,1000,500]);

subplot(1,2,1);
stem(aodAz, gain, "BaseValue",-30); hold on;
stem(aodAz, gainElem, "BaseValue",-30); grid on; hold off;
xlabel("Azimuth AoD"); ylabel("Gain [dB]");
ylim([-60, max(gainElem)+8]);
```

```
xlim([-180 180]);
legend("Chan Gain","Chan Gain + Element Gain", "Orientation","horizontal");
title("Channel Gain vs Channel + Element Gain");

subplot(1,2,2);
stem(aoaAz, gain, "BaseValue",-30); hold on;
stem(aoaAz, gainElem, "BaseValue",-30); grid on; hold off;
xlabel("Azimuth AoA"); ylabel("Gain [dB]");
ylim([-60, max(gainElem)+8]);
xlim([-180 180]);
legend("Chan Gain","Chan Gain + Element Gain", "Orientation","horizontal");
title("Channel Gain vs Channel + Element Gain");
```



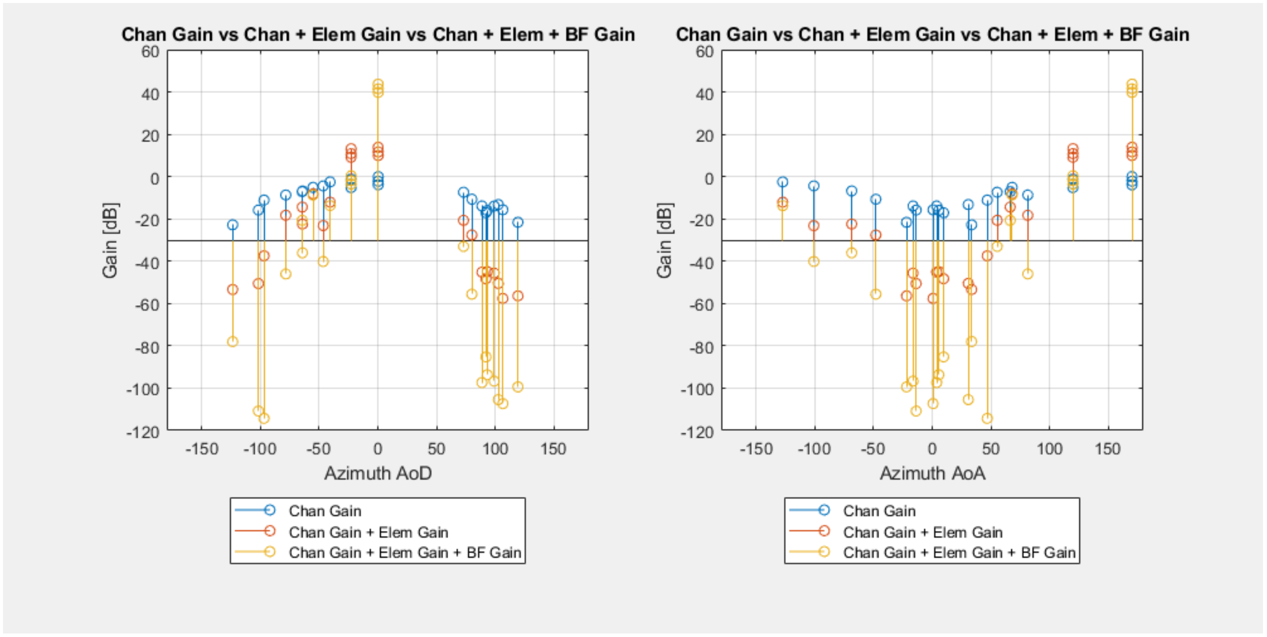
As a simple example for the beamforing, find the path index `i` where `gainElem(i)` is maximized. Then find TX and RX beamforming vectors, `wtx` and `wrx` aligned to the conjugate of the normalized spatial signatures on this path.

```
% TODO
[~, max_ind] = max(gainElem);
w_b = conj(svTx(:, max_ind));
wtx = w_b / norm(w_b);
w_b_rx = conj(svRx(:, max_ind));
wrx = w_b_rx / norm(w_b_rx);
```

Find the gain along each path using the BF vectors from above. Then create the same stem plot as before. You should see that, after the beamforming, there is one path that is dominant.

```
% TODO:  Find the gain along each path
gainBFtx = mag2db(abs(sum(wtx.*svTx,1)))';
gainBFrx = mag2db(abs(sum(wrx.*svRx,1)))';
gainArr = gainElem + gainBFtx + gainBFrx;
```

```
% TODO:  Create the stem plots
clf;
set(gcf,'Position', [0,0,1000,500]);

subplot(1,2,1);
stem(aodAz, gain, "BaseValue",-30); hold on;
stem(aodAz, gainElem,"BaseValue",-30); hold on; grid on;
stem(aodAz, gainArr,"BaseValue",-30); hold off;
xlabel("Azimuth AoD"); ylabel("Gain [dB]");
%ylim([-40, max(gainArr)+8]);
xlim([-180 180]);
legend("Chan Gain","Chan Gain + Elem Gain", "Chan Gain + Elem Gain + BF Gain", "Orientation","v
title("Chan Gain vs Chan + Elem Gain vs Chan + Elem + BF Gain");

subplot(1,2,2);
stem(aoaAz, gain,"BaseValue",-30); hold on;
stem(aoaAz, gainElem,"BaseValue",-30); hold on; grid on;
stem(aoaAz, gainArr,"BaseValue",-30); hold off;
xlabel("Azimuth AoA"); ylabel("Gain [dB]");
%ylim([-40, max(gainArr)+8]);
xlim([-180 180]);
legend("Chan Gain","Chan Gain + Elem Gain", "Chan Gain + Elem Gain + BF Gain", "Orientation","v
title("Chan Gain vs Chan + Elem Gain vs Chan + Elem + BF Gain");
```



Visualizing the OFDM Channel

We conclude by visualizing the OFDM channel with and without beamforming. We use the following parameters

```
fc = 28e9;                    % Carrier frequency
SubcarrierSpacing = 120;      % SCS in kHZ
NRB = 61;  % number of resource blocks
```

```
nscPerRB = 12;  % number of sub-carriers per RB
```

Similar to the previous lab, we get the `carrierConfig` and `waveformConfig` with the `nrCarrierConfig` and `nrOFDMInfo` methods in the 5G Toolbox.

```
carrierConfig = nrCarrierConfig('NSizeGrid', NRB, 'SubcarrierSpacing', SubcarrierSpacing);
waveformConfig = nrOFDMInfo(carrierConfig);
```

Complete the MIMO channel object, `FDMIMO` which produces a OFDM-MIMO channel matrix.

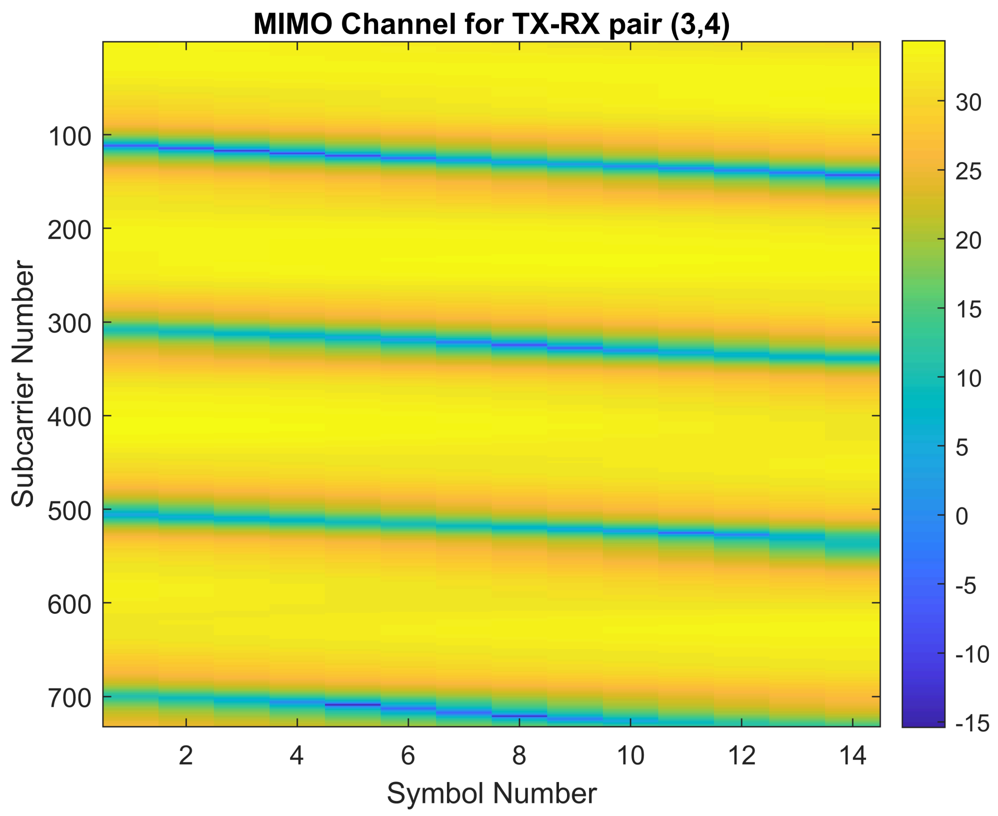
```
        % TODO:  Complete the step method in the FDMIMO class.
```

Once this class is completed, we can create the channel for one slot as follows.

```
Enoise = -5;  % Energy per noise sample in dB
fdchan = FDMIMOChan(carrierConfig, 'txArrPlatform', arrPlatformgNB, 'rxArrPlatform', arrPlatfor
    'aoaAz', aoaAz, 'aodAz', aodAz, 'aoaEl', aoaEl, 'aodEl', aodEl,  ...
    'gain', gain, 'dly', dly, 'fc', fc, 'Enoise', Enoise);

frameNum = 0;
slotNum = 0;
[chanGrid, noiseVar] = fdchan.step(frameNum, slotNum);
```

The above command should yield an array chanGrid of size `nrx x ntx x nsc x nt` representing the MIMO channel matrix components over frequency and time. Plot the SNR per resource element for the channel along a single TX and RX antenna pair, say `chanGrid(3,4,:,:)`. Plot the SNR in dB scale using the `imagesc` command so it can be visualized in time and frequency.

```
% TODO:
figure;
sub_chanGrid = reshape(chanGrid(3,4,:,:),nscPerRB*NRB,14);
snrSing = pow2db(abs(sub_chanGrid).^2/noiseVar);
imagesc(snrSing);
colorbar();
xlabel("Symbol Number"); ylabel("Subcarrier Number");
title("MIMO Channel for TX-RX pair (3,4)");
```

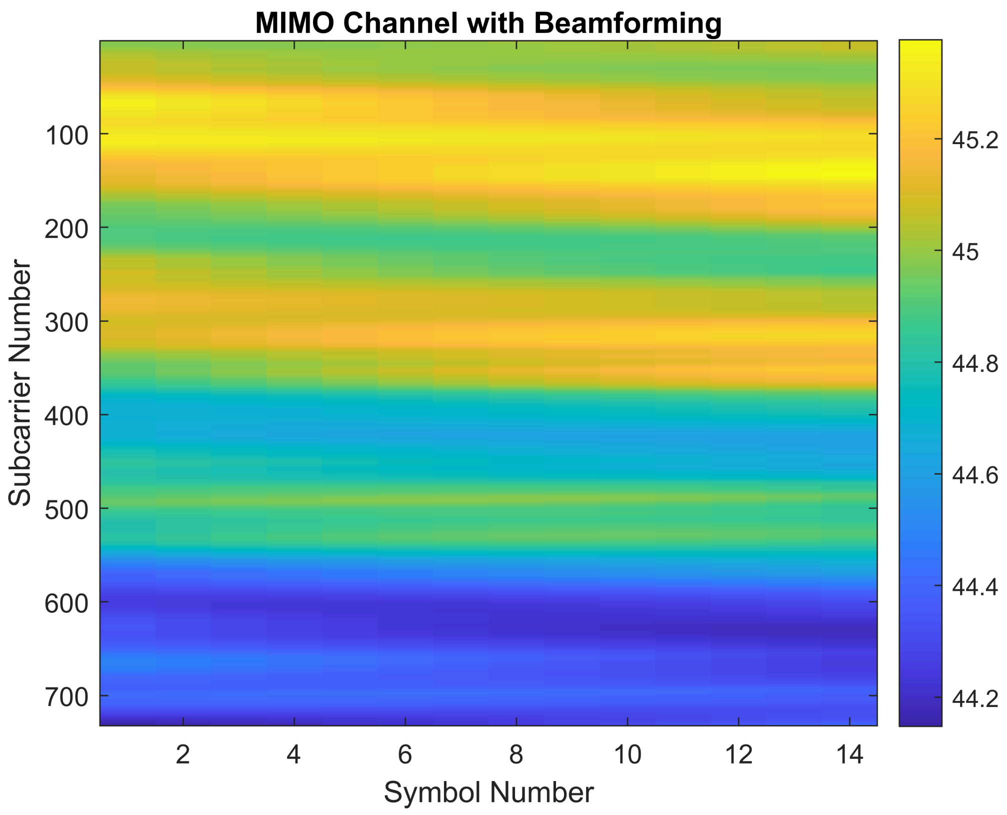


OFDM Beamforming

We next compute the SNR using instantaneous BF. For each point `chanGrid(:,:,n,t)`, find the maximum gain with BF using the SVD. Plot the resulting SNR. You should see that the resulting SNR is much higher and varies very little over time and frequency.

```
% TODO:
nrx = arrPlatformUE.getNumElements();
ntx = arrPlatformgNB.getNumElements();
chan_BF = zeros(nscPerRB*NRB,14);
for i=1:nscPerRB*NRB
    for j=1:14
        H = reshape(chanGrid(:,:,i,j), nrx,ntx);
        [U,S,V] = svd(H);
        wrx = U(:,1);
        wtx = V(:,1);
        c_bf = wrx'*H*wtx;
        chan_BF(i,j) = c_bf;
    end
end

figure;
snrInst = pow2db(abs(chan_BF).^2/noiseVar);
imagesc(snrInst);
colorbar();
xlabel("Symbol Number"); ylabel("Subcarrier Number");
```

```
title("MIMO Channel with Beamforming");
```



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

        fc = 28e9;  % Carrier frequency

        % Element within each array.
        elem = [];

        % Gridded interpolant object for the element gain in linear scale
        elemGainInterp = [];

        % Antenna array.
        arr = [];

        % Steering vector object
        svObj = [];

        % 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);

        % Position in m
        pos = zeros(1,3);

        % Normalization matrix
        Qinv = [];
    end

    methods
        function obj = ArrayPlatform(varargin)
            % Constructor

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



        function computeNormMatrix(obj)
            % The method performs two key tasks:
            % * Measures the element pattern and creates a gridded
            %   interpolant object to interpolate the values at other
            %   angles
            % * Compute the normalization matrix to account for mutual
            %   coupling

            % TODO:  Get the pattern for the element using the elem.pattern
            % method
            [elemGain,az,el] = obj.elem.pattern(obj.fc,'Type','directivity');
            % TODO:  Create the gridded interpolant object
            % for element gain
            obj.elemGainInterp = griddedInterpolant({el,az},elemGain);

            % Get a vector of values with the elements az(i) and el(j).
            [azMat, elMat] = meshgrid(az, el);
            azVal = azMat(:);
            elVal = elMat(:);
            elemGainVal = elemGain(:);

            % TODO:  Create a steering vector object with the array
            obj.svObj = phased.SteeringVector('SensorArray', obj.arr);

            % TODO:  Get the steering vectors
            Sv0 = obj.svObj(obj.fc, [azVal elVal]');

            % TODO:  Compute un-normalized spatial signature
            % by multiplying the steering vectors with the element gain
            elemGainLin = db2mag(elemGainVal);
            SvNoNorm = Sv0.*elemGainLin';

            % TODO:  Compute the normalization matrix by integrating the
            % un-normalized steering vectors
            cosel = cos(deg2rad(elVal));
            Q = (1/length(elVal))*(SvNoNorm.*cosel')*(SvNoNorm') / mean(cosel);

            % Ensure matrix is Hermitian
            Q = (Q+Q')/2;

            % TODO:  Save the inverse matrix square root of Q
            obj.Qinv = sqrtm(Q);

        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.svObj)
                obj.svObj = phased.SteeringVector('SensorArray',obj.arr);
            else
                obj.svObj.release();
            end

        end

        function n = getNumElements(obj)
            % Gets the number of elements
            n = obj.arr.getNumElements();

        end

        function elemPosGlob = getElementPos(obj)
            % Gets the array elements in the global reference frame

            % Get the element position in the local reference frame
            elemPosLoc = obj.arr.getElementPosition();

            % Convert to the global reference frame
            elemPosGlob = local2globalcoord(elemPosLoc, 'rr', ...
                zeros(3,1), obj.axesLoc) + reshape(obj.pos,3,1);
        end

    end

    methods (Access = protected)

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


        end

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

       function [Sv, elemGain] = stepImpl(obj, az, el, relSV)
            % Gets normalized steering vectors and element gains for a set of angles
            % The angles az and el should be columns vectors along which
            % the outputs are to be computed.
            % If the relSV == true, then the steering vector object is
            % released.  This is needed in case the dimensions of the past
            % call are the different from the past one

            % Release the SV
            if nargin < 4
                relSV = true;
            end
            if relSV
```

```
                obj.releaseSV();
            end

            % TODO:  Convert the global angles (az, el) to local
            % angles (azLoc, elLoc).  Use the
            % global2localcoord() method with the 'ss' option.
            locCoord = global2localcoord([az; el; ones(size(az));],'ss',[0;0;0],obj.axesLoc);
            azLoc = locCoord(1,:);
            elLoc = locCoord(2,:);

            % TODO: Get the SV in the local coordinates
            Sv0 = obj.svObj(obj.fc, [azLoc(:) elLoc(:)]');

            % TODO:  Get the directivity gain of the element from the
            % local angles.
            elemGain = obj.elemGainInterp(elLoc, azLoc);
            elemGainLin = db2mag(elemGain(:));
            SvNoNorm = Sv0.*elemGainLin';

            % TODO:  Compute the normalized steering vectors

            Sv = obj.Qinv \ SvNoNorm;

        end

    end


end
```

```
ans = 

  ArrayPlatform with properties:

               fc: 2.8000e+10
             elem: []
   elemGainInterp: []
              arr: []
            svObj: []
           axesAz: 0
           axesEl: 0
          axesLoc: [3×3 double]
              vel: [0 0 0]
              pos: [0 0 0]
             Qinv: []
```

Published with MATLAB® R2020b

```
classdef FDMIMOChan < matlab.System
    % Frequency-domain MIMO multipath channel
    properties
        % Configuration
        carrierConfig;   % Carrier configuration
        waveformConfig;  % Waveform parameters

        % Path parameters
        gain;  % Relative path gain in dB
        dly;   % Delay of each path in seconds
        aodAz, aodEl; % Angle of departure of each path in degrees
        aoaAz, aoaEl; % Angle of arrival of each path in degrees

        % Derived path parameters
        fd;    % Doppler shift for each path
        gainComplex;  % Complex gain of each path
        svTx, svRx;   % Steering vectors for each path
        elemGainTx, elemGainRx;  % Element gains

        % Other parmaters
        fc = 28e9;    % Carrier freq in Hz
        rxVel = [30,0,0]';  % RX velocity vector in m/s
        txVel = [0,0,0]';   % TX velocity vector in m/s
        Enoise = 0;         % Noise energy per sample in dBmJ

        % Symbol times
        symStart;  % symStart(i) = start of symbol i relative to subframe

        % TX and RX array platforms
        txArrPlatform = [];
        rxArrPlatform = [];

    end
    methods
        function obj = FDMIMOChan(carrierConfig, varargin)
            % Constructor

            % Save the carrier configuration
            obj.carrierConfig = carrierConfig;

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

            % Check all the required fields are specified
            fields = {'txArrPlatform', 'rxArrPlatform', 'gain', 'dly', ...
                'aoaAz', 'aodAz', 'aoaEl', 'aoaAz' };
            nfields = length(fields);
            for i = 1:nfields
                fstr = fields{i};
                if isempty(obj.(fstr))
                    e =  MException('FDMIMOChan:missingParam', ...
                        'Parameter %s not speficied', fstr);
                    throw(e);
                end
            end

            % Complex gain for each path using a random initial phase
            % The gains are normalized to an average of one
            npath = length(obj.gain);
            phase = 2*pi*rand(npath, 1);
            obj.gainComplex = db2mag(obj.gain).*exp(1i*phase);

            % Symbol times relative to the start of the subframe
            obj.waveformConfig = nrOFDMInfo(obj.carrierConfig);
            nsym = obj.waveformConfig.SymbolLengths;
            obj.symStart = nsym/obj.waveformConfig.SampleRate;
            obj.symStart = cumsum([0 obj.symStart]');

            % Get Doppler shift for RX
            vc = physconst('Lightspeed');
            [ux, uy, uz] = sph2cart(deg2rad(obj.aoaAz), deg2rad(obj.aoaEl), 1);
            obj.fd = [ux uy uz]*obj.rxVel*obj.fc/vc;

            % Get Doppler shift for TX
            [ux, uy, uz] = sph2cart(deg2rad(obj.aodAz), deg2rad(obj.aodEl), 1);
            obj.fd = obj.fd + [ux uy uz]*obj.txVel*obj.fc/vc;
        end

        function computePathSV(obj)
            % Computes the element gains and steering vectors of each path

            % Call the array platform objects to get the steering vectors
            % and element gains
            [obj.svTx, obj.elemGainTx] = ...
                obj.txArrPlatform.step(obj.aodAz', obj.aodEl',true);
            [obj.svRx, obj.elemGainRx] = ...
                obj.rxArrPlatform.step(obj.aoaAz', obj.aoaEl',true);

        end


    end
    methods (Access = protected)


        function [chanGrid, noiseVar] = stepImpl(obj, frameNum, slotNum)
            % Applies a frequency domain channel and noise
            %
            % Parameters
            % ----------
            % frameNum:  The index of the frame  (1 frame = 10ms)
            % slotNum:  The index of the slot in the frame
            %    This should be 0,...,waveformConfig.SlotsPerFrame
            %
            % Outputs
            % -------
            % chanGrid:  Grid of the channel values
            % noiseVar:  Noise variance

            % Compute the steering vectors and element gains
            obj.computePathSV();

            % Get the number of TX and RX elements
            ntx = obj.txArrPlatform.getNumElements();
            nrx = obj.rxArrPlatform.getNumElements();

            % Get the number of sub-carriers
            nscPerRB = 12;
            nsc = obj.carrierConfig.NSizeGrid * nscPerRB;
            nsym = obj.carrierConfig.SymbolsPerSlot;

            % Compute the frequency of each carrier
            f = (0:nsc-1)'*obj.carrierConfig.SubcarrierSpacing*1e3;

            % Compute slot in sub-frame and sub-frame index
            sfNum = floor(slotNum / obj.waveformConfig.SlotsPerSubframe);
            slotNum1 = mod(slotNum, obj.waveformConfig.SlotsPerSubframe);

            % Compute the time for each symbol
            framePeriod = 0.01;
            sfPeriod = 1e-3;
            t = frameNum*framePeriod + sfPeriod*sfNum + ...
                obj.symStart(slotNum1+1:slotNum1+nsym);

            % Initialize the channel grid to zero
            chanGrid = zeros(nrx, ntx, nsc, nsym);
            npath = length(obj.gain);

            % TODO: Set the channel:
            %
            % chanGrid(j,k,n,t) = MIMO channel matrix from
            %    RX antenna j, TX antenna k, sub-carrier n,
            %    symbol t.
            %
            % This should be a sum of the paths
            %
            % chanGrid(j,k,:,:)
            %   = \sum_i exp(1i*phase)*svRx(j,i)*svTx(k,i)
            %
            % where
            %
            % phase = 2*pi*(f*obj.dly(i) + t'*obj.fd(i));

            for j=1:nrx
                for k=1:ntx
                    for p=1:npath
                        phase = 2*pi*(f*obj.dly(p) + t'*obj.fd(p));
                        chan_path = exp(1i*phase)*obj.svRx(j,p)*obj.svTx(k,p);
                        curr_chan = reshape(chanGrid(j,k,:,:),nsc,nsym);
                        chanGrid(j,k,:,:) = curr_chan + chan_path;
                    end
                end
            end

            % Compute noise variance
            noiseVar = db2pow(obj.Enoise);

        end

    end
end
```

```
Not enough input arguments.

Error in FDMIMOChan (line 39)
            obj.carrierConfig = carrierConfig;
```