# Phased Array System Toolbox™

## User's Guide

# MATLAB&SIMULINK®

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

**Phased Arrays**

## Antenna and Microphone Elements

**1**

# Array Geometries and Analysis

**2**

# Signal Radiation and Collection

**3**

**7**

**8**

# Environment and Target Models

## 9

# Coordinate Systems and Motion Modeling

## 10

**Using Polarization**

# 11

# Phased Arrays

# Antenna and Microphone Elements

# Isotropic Antenna Element

| In this section... |
| --- |
| "Support for Isotropic Antenna Elements" on page 1-2 |
| "Backbaffled Isotropic Antenna" on page 1-2 |
| "Response of Backbaffled Isotropic Antenna Element" on page 1-5 |

## Support for Isotropic Antenna Elements

An isotropic antenna element radiates equal power in all directions. If the antenna element is backbaffled, the antenna radiates equal power in all directions for which the azimuth angle satisfies – $90 \leq \varphi \leq 90$ and zero power in all other directions. To construct an isotropic antenna, use the `phased.IsotropicAntennaElement` System object™. When you use this object, you must specify these antenna properties:

FPGA

- The operating frequency range of the antenna using the `FrequencyRange` property.
- Whether or not the response of the antenna is backbaffled at azimuth angles outside the interval *[–90,90]* using the `BackBaffled` property.

You can determine the voltage response of the isotropic antenna element at specified frequencies and angles by executing the System object.

## Backbaffled Isotropic Antenna

This example shows how to construct a backbaffled isotropic antenna element with a uniform frequency response over a range of azimuth angles from [-180,180] degrees and elevation angles from [-90,90] degrees. The antenna operates between 300 MHz and 1 GHz. Show the antenna pattern at 1 GHz.

```
fc = 1e9;
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[300e6 1e9],'BackBaffled',false);
pattern(antenna,fc,[-180:180],[-90:90],'CoordinateSystem','polar',...
    'Type','power')
```

### 3D Response Pattern
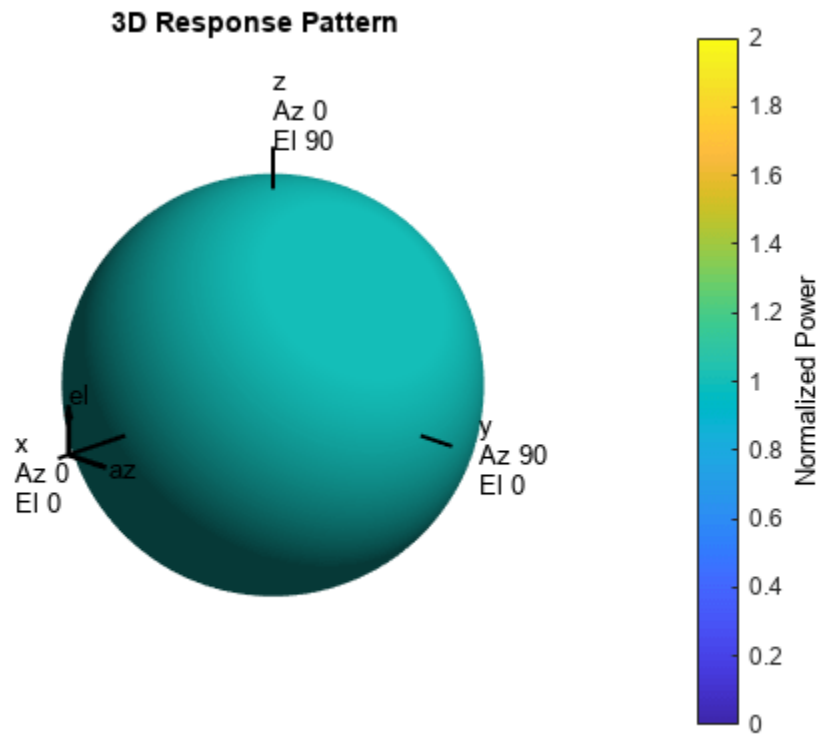


Using the antenna `pattern` method, plot the antenna response at zero degrees elevation for all azimuth angles at 1 GHz.

```
pattern(antenna,1e9,[-180:180],0,'CoordinateSystem','rectangular',...
    'Type','powerdb')
```

**Azimuth Cut (elevation angle = 0.0°)**
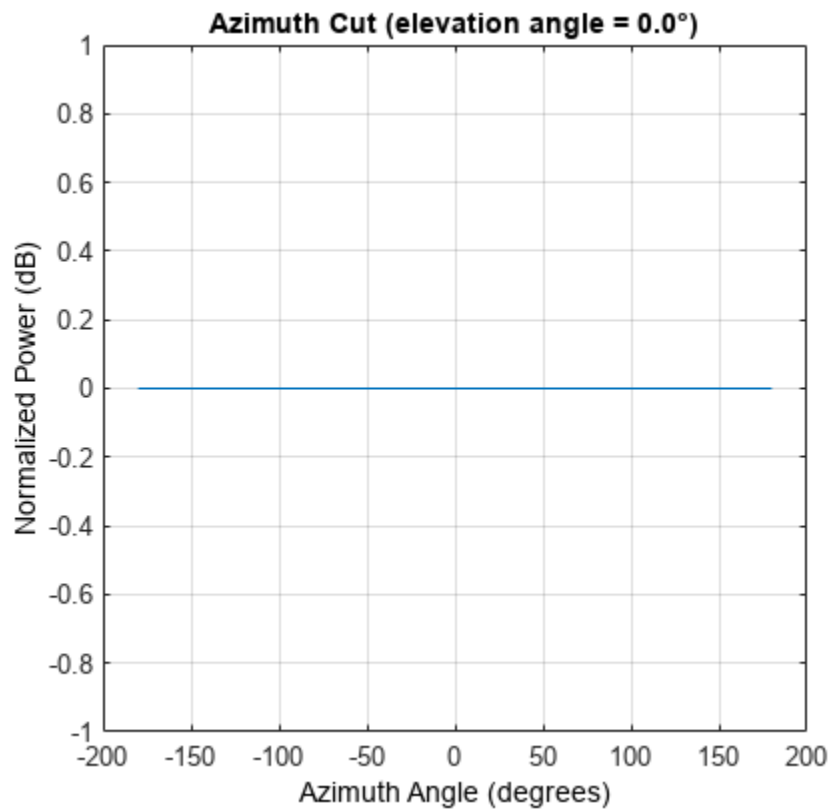


Setting the `BackBaffled` property to `true` restricts the antenna response to azimuth angles in the interval [-90,90] degrees. In this case, plot the antenna response in three dimensions.

```
antenna.BackBaffled = true;
pattern(antenna,fc,[-180:180],[-90:90],'CoordinateSystem','polar',...
    'Type','power')
```

## Response of Backbaffled Isotropic Antenna Element

This example shows how to design a backbaffled isotropic antenna element and obtain its response. First, construct an X-band isotropic antenna element that operates from 8 to 12 GHz setting the `Backbaffle` property to `true`. Obtain the antenna element response at 4, 10, and 14 GHz at azimuth angles between -100 and 100 degrees in 50 degree increments. All elevation angles are by default equal to zero.

```
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[8e9 12e9],'BackBaffled',true);
respfreqs = [6:4:14]*1e9;
respazangles = -100:50:100;
anresp = antenna(respfreqs,respazangles)
```

```
anresp = 5×3

     0     0     0
     0     1     0
     0     1     0
     0     1     0
     0     0     0
```

The antenna response in `anresp` is a matrix having row dimension equal to the number of azimuth angles in `respazangles` and column dimension equal to the number of frequencies in `respfreqs`.

The response voltage in the first and last columns of `anresp` are zero because those columns contain the antenna response at 6 and 14 GHz, respectively. These frequencies lie outside the antenna operating frequency range. Similarly, the first and last rows of `anresp` contain all zeros because `BackBaffled` property is set to `true`. The first and last row contain the antenna response at azimuth angles outside of [-90,90].

To obtain the antenna response at nonzero elevation angles, input the angles to the object as a 2-by-M matrix where each column is an angle in the form `[azimuth;elevation]`.

```
release(antenna)
respelangles = -90:45:90;
respangles = [respazangles; respelangles];
anresp = antenna(respfreqs,respangles)
```

anresp = *5×3*

```
     0     1     0
     0     1     0
     0     1     0
     0     1     0
     0     1     0
```

Notice that `anresp(1,2)` and `anresp(5,2)` represent the antenna voltage response at the azimuth-elevation angle pairs (-100,-90) and (100,90) degrees. Although the azimuth angles lie in the baffled region, because the elevation angles are equal to +/- 90 degrees, the responses are unity. In this case, the resulting elevation cut degenerates to a point.

# Cosine Antenna Element

| **In this section...** |
| --- |
| "Support for Cosine Antenna Elements" on page 1-7 |
| "Concentrating Cosine Antenna Response" on page 1-7 |
| "Plot 3-D Response of Cosine Antenna Element" on page 1-8 |

## Support for Cosine Antenna Elements

The `phased.CosineAntennaElement` object models an antenna element whose response follows a cosine function raised to a specified power in both the azimuth and elevation directions.

The object returns the field response (also called field pattern)

$$f(az, el) = \cos^m(az)\cos^n(el)$$

of the cosine antenna element.

In this expression

- $az$ is the azimuth angle.
- $el$ is the elevation angle.
- The exponents $m$ and $n$ are real numbers greater than or equal to zero.

The response is defined for azimuth and elevation angles between –90° and 90°, inclusive, and is always positive. There is no response at the backside of a cosine antenna. The cosine response pattern achieves a maximum value of 1 at 0° azimuth and 0° elevation. Larger exponent values narrow the response pattern of the element and increase the directivity.

The power response (or power pattern) is the squared value of the field response.

$$P(az, el) = \cos^{2m}(az)\cos^{2n}(el)$$

When you use the cosine antenna element, you specify the exponents of the cosine pattern using the `CosinePower` property and the operating frequency range of the antenna using the `FrequencyRange` property.

## Concentrating Cosine Antenna Response

This example shows the effect of concentrating the cosine antenna response by increasing the exponent of the cosine factor. The example computes and plots the cosine response for exponents equal to 1 and 2 for a single angle between -90 and 90 degrees. The angle can represent azimuth or elevation.

```
theta = -90:.01:90;
costh1 = cosd(theta);
costh2 = costh1.^2;
plot(theta,costh1)
hold on
plot(theta,costh2,'r')
```

```
hold off
legend('Exponent = 1','Exponent = 2','location','northeast');
xlabel('Angle (degrees')
ylabel('Response')
```
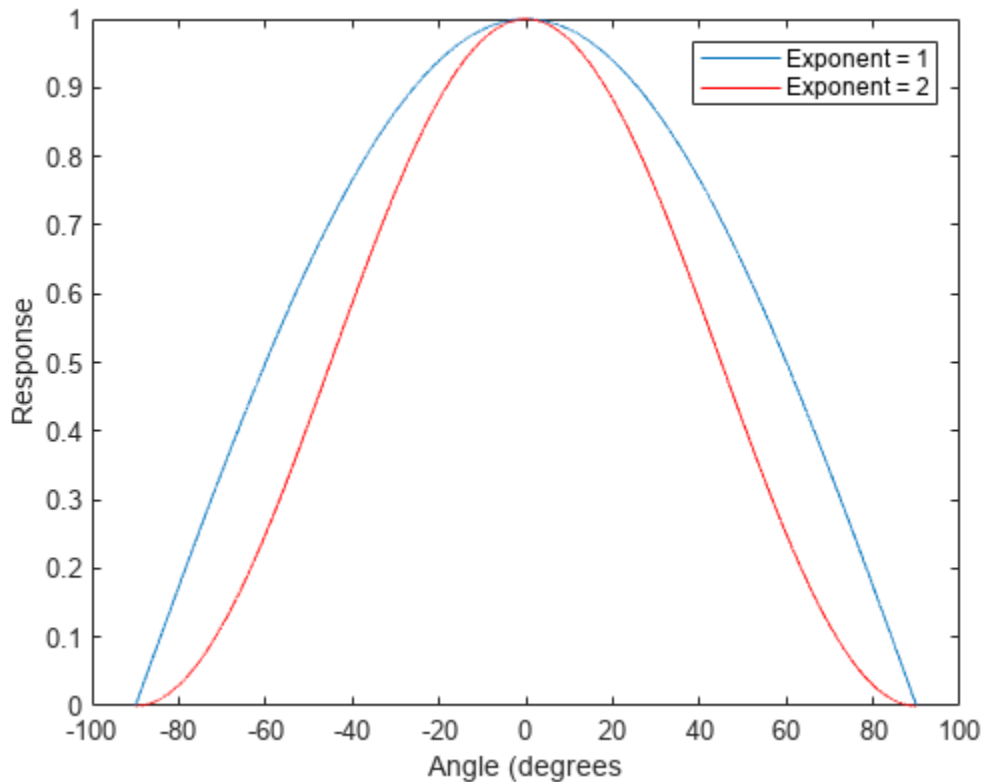


## Plot 3-D Response of Cosine Antenna Element

This example shows how to construct an antenna with a cosine-squared response in both azimuth and elevation. The operating frequency range of the antenna is 1 to 10 GHz. Plot the 3-D antenna response at 5 GHz.

```
sCos = phased.CosineAntennaElement(...
    'FrequencyRange',[1 10]*1e9,'CosinePower',[2 2]);
pattern(sCos,5e9,[-180:180],[-90:90],'CoordinateSystem',...
    'Polar','Type','powerdb')
```

3D Response Pattern
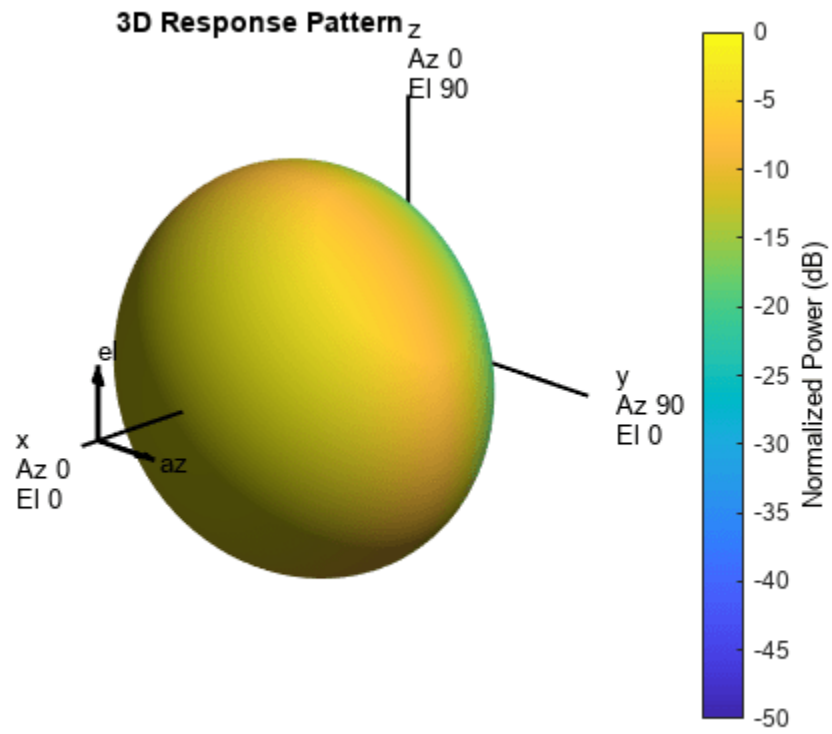
# Custom Antenna Element

| In this section... |
| --- |
| "Support for Custom Antenna Elements" on page 1-10 |
| "Antenna with Custom Radiation Pattern" on page 1-10 |

## Support for Custom Antenna Elements

The `phased.CustomAntennaElement` object enables you to model a custom antenna element. When you use `phased.CustomAntennaElement`, you must specify these aspects of the antenna:

- Operating frequency vector for the antenna element
- Frequency response of the element at the frequencies in the operating frequency vector
- Azimuth angles and elevation angles where the custom response is evaluated
- Magnitude radiation pattern. This pattern shows the spatial response of the antenna at the azimuth and elevation angles you specify.

> **Tip** You can import a radiation pattern that uses $u/v$ coordinates or $\varphi/\theta$ angles, instead of azimuth/elevation angles. To use such a pattern with `phased.CustomAntennaElement`, first convert your pattern to azimuth/elevation form. Use `uv2azelpat` or `phitheta2azelpat` to do the conversion. For an example, see Antenna Array Analysis with Custom Radiation Pattern.

For your custom antenna element, the antenna response depends on the frequency response and radiation pattern. Specifically, the frequency and spatial responses are interpolated separately using nearest-neighbor interpolation and then multiplied together to produce the total response. To avoid interpolation errors, the range of azimuth angles should include +/– 180 degrees and the range of elevation angles should include +/– 90 degrees.

## Antenna with Custom Radiation Pattern

Create a custom antenna element object. The radiation pattern has a cosine dependence on elevation angle but is independent of azimuth angle.

```
az = -180:90:180;
el = -90:45:90;
elresp = cosd(el);
magpattern = mag2db(repmat(elresp',1,numel(az)));
phasepattern = zeros(size(magpattern));
antenna = phased.CustomAntennaElement('AzimuthAngles',az,...
    'ElevationAngles',el,'MagnitudePattern',magpattern, ...
    'PhasePattern',phasepattern);
```

Display the radiation pattern.

```
disp(antenna.MagnitudePattern)

    -Inf      -Inf      -Inf      -Inf      -Inf
  -3.0103   -3.0103   -3.0103   -3.0103   -3.0103
        0         0         0         0         0
```

```
   -3.0103   -3.0103   -3.0103   -3.0103   -3.0103
      -Inf      -Inf      -Inf      -Inf      -Inf
```

Calculate the antenna response at the azimuth-elevation pairs *(-30,0)* and *(-45,0)* at 500 MHz.

```
ang = [-30 0; -45 0];
resp = antenna(500.0e6,ang);
disp(resp)
```

```
   0.7071
   1.0000
```

The following code illustrates how nearest-neighbor interpolation is used to find the antenna voltage response in the two directions. The total response is the product of the angular response and the frequency response.

```
g = interp2(deg2rad(antenna.AzimuthAngles),...
    deg2rad(antenna.ElevationAngles),...
    db2mag(antenna.MagnitudePattern),...
    deg2rad(ang(1,:))', deg2rad(ang(2,:))','nearest',0);
h = interp1(antenna.FrequencyVector,...
    db2mag(antenna.FrequencyResponse),500e6,'nearest',0);
antresp = h.*g;
```

Compare the value of `antresp` to the response of the antenna.

```
disp(mag2db(antresp))
```

```
   -3.0103
         0
```

# Omnidirectional Microphone

| **In this section...** |
| --- |
| "Support for Omnidirectional Microphones" on page 1-12 |
| "Backbaffled Omnidirectional Microphone" on page 1-12 |

## Support for Omnidirectional Microphones

An omnidirectional microphone has a response which is equal to one in all nonbaffled directions. The `phased.OmnidirectionalMicrophoneElement` object enables you to model an omnidirectional microphone. When you use this object, you must specify these aspects of the microphone:

- The operating frequency range of the microphone using the `FrequencyRange` property.
- Whether the response of the microphone is baffled at azimuth angles outside the interval *[–90,90]* degrees using the `BackBaffled` property.

## Backbaffled Omnidirectional Microphone

Construct an omnidirectional microphone element having a response within the human audible frequency range of 20 to 20,000 Hz. Baffle the microphone response for azimuth angles outside of +/- 90 degrees. Plot in polar form the microphone power response at 1 kHz.

```
freq = 1e3;
microphone = phased.OmnidirectionalMicrophoneElement(...
    'BackBaffled',true,'FrequencyRange',[20 20e3]);
pattern(microphone,freq,[-180:180],[-90:90],'CoordinateSystem','polar','Type','power');
```

**3D Response Pattern**
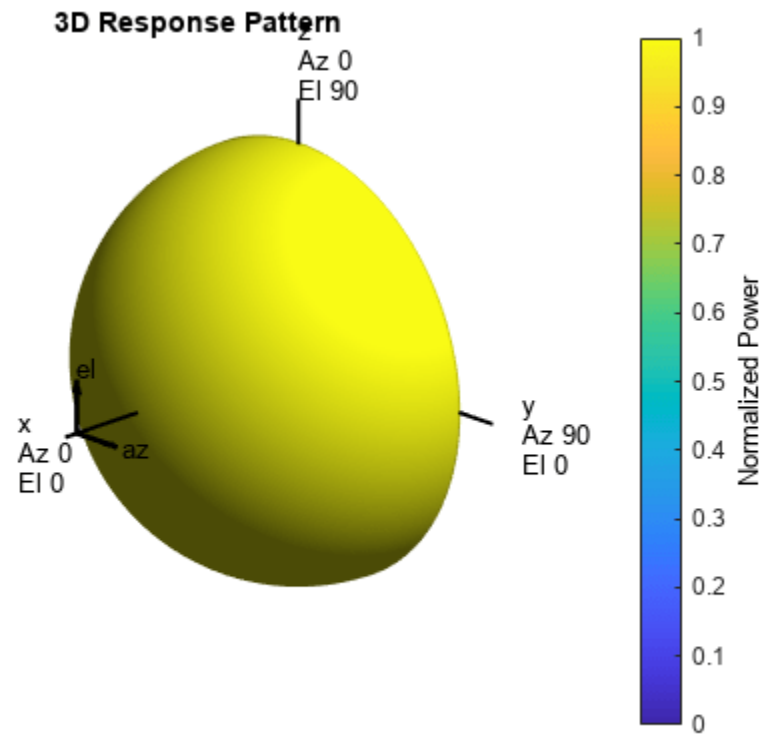


In many applications, you sometimes need to examine the microphone directionality, or polar pattern. To obtain an azimuth cut, set the elevation argument of the `pattern` method to a single angle such as zero.

```
pattern(microphone,freq,[-180:180],0,'CoordinateSystem','polar','Type','power');
```

**Azimuth Cut (elevation angle = 0.0°)**



Normalized Power, Broadside at 0.00 °
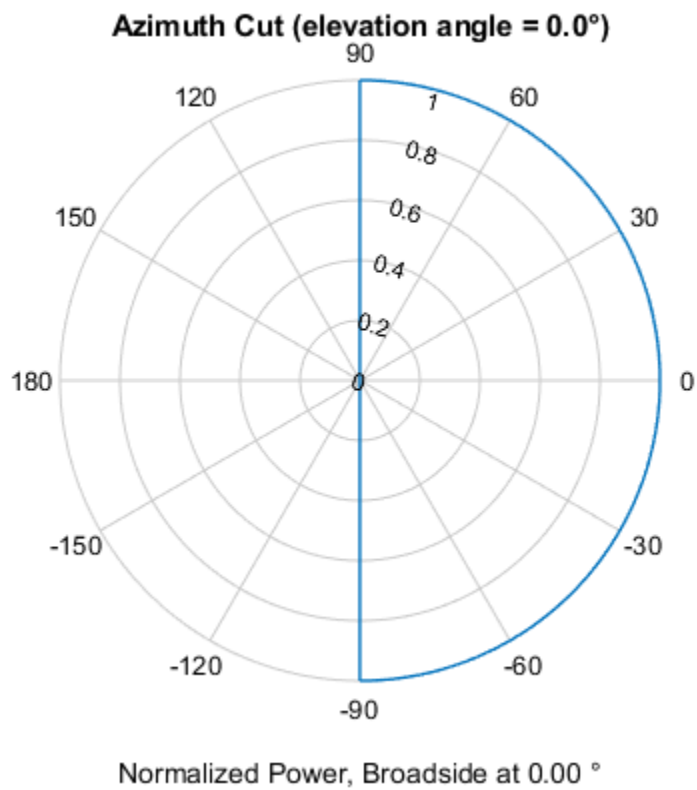
To obtain an elevation cut, set the azimuth argument of the `pattern` method to a single angle such as zero.

```
pattern(microphone,freq,0,[-90:90],'CoordinateSystem','polar','Type','power');
```

**Elevation Cut (azimuth angle = 0.0°)**



Normalized Power, Broadside at 0.00 °
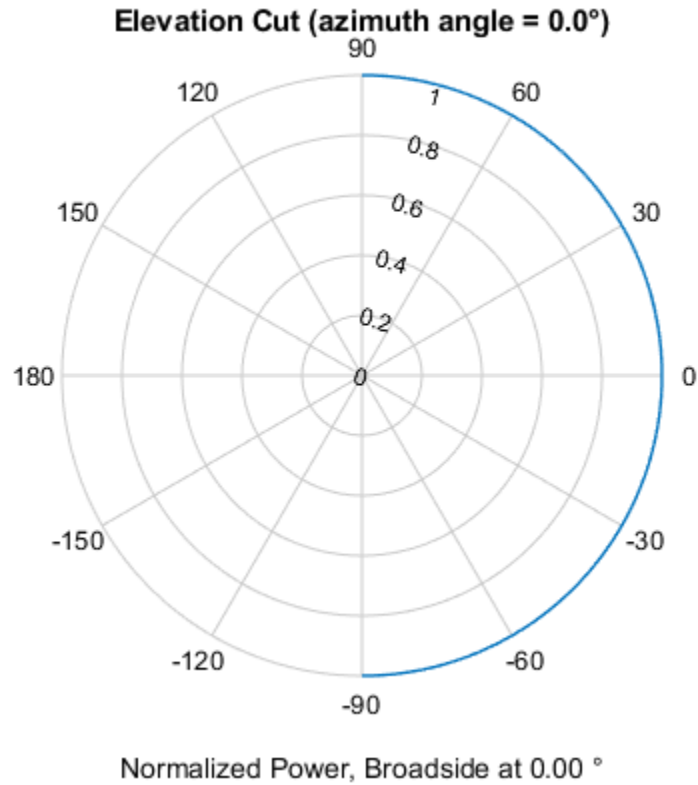
Obtain the microphone magnitude response at the specified azimuth angles and frequencies. By default, when the `ang` argument is a single row, the elevation angles are 0 degrees. Note the response is unity at all azimuth angles and frequencies, as expected.

```
freqs = [100:250:1e3];
ang = [-90:30:90];
response = microphone(freqs,ang)
```

response = 7×4

```
1      1      1      1
1      1      1      1
1      1      1      1
1      1      1      1
1      1      1      1
1      1      1      1
1      1      1      1
```

# Custom Microphone Element

| **In this section...** |
| --- |
| "Support for Custom Microphone Elements" on page 1-16 |
| "Custom Cardioid Microphone Pattern" on page 1-16 |

## Support for Custom Microphone Elements

You can model a microphone with a custom response pattern using `phased.CustomMicrophoneElement` System object. The total response of a custom microphone element is a combination of its frequency response and spatial response. `phased.CustomMicrophoneElement` calculates both responses using nearest neighbor interpolation and then multiplies them to form the total response. When the `PolarPatternFrequencies` property value is nonscalar, the object specifies multiple polar patterns. In this case, the interpolation uses the polar pattern that is measured closest to the specified frequency. When you use `phased.CustomMicrophoneElement`, you must specify these microphone attributes.:

- Frequencies where you specify your response using the `FrequencyVector` property.
- Response corresponding to the specified frequencies using the `FrequencyResponse` property.
- Frequencies and angles at which the microphone's polar pattern is measured.
- Magnitude response of the microphone.

## Custom Cardioid Microphone Pattern

Create a custom cardioid microphone, and plot the power response pattern at 500 and 800 Hz.

```
sCustMic = phased.CustomMicrophoneElement;
sCustMic.PolarPatternFrequencies = [500 1000];
sCustMic.PolarPattern = mag2db([...
    0.5+0.5*cosd(sCustMic.PolarPatternAngles);...
    0.6+0.4*cosd(sCustMic.PolarPatternAngles)]);

pattern(sCustMic,[500,800],[-180:180],0,'Type','powerdb')
```

Azimuth Cut (elevation angle = 0.0°)

Normalized Power (dB), Broadside at 0.00 °
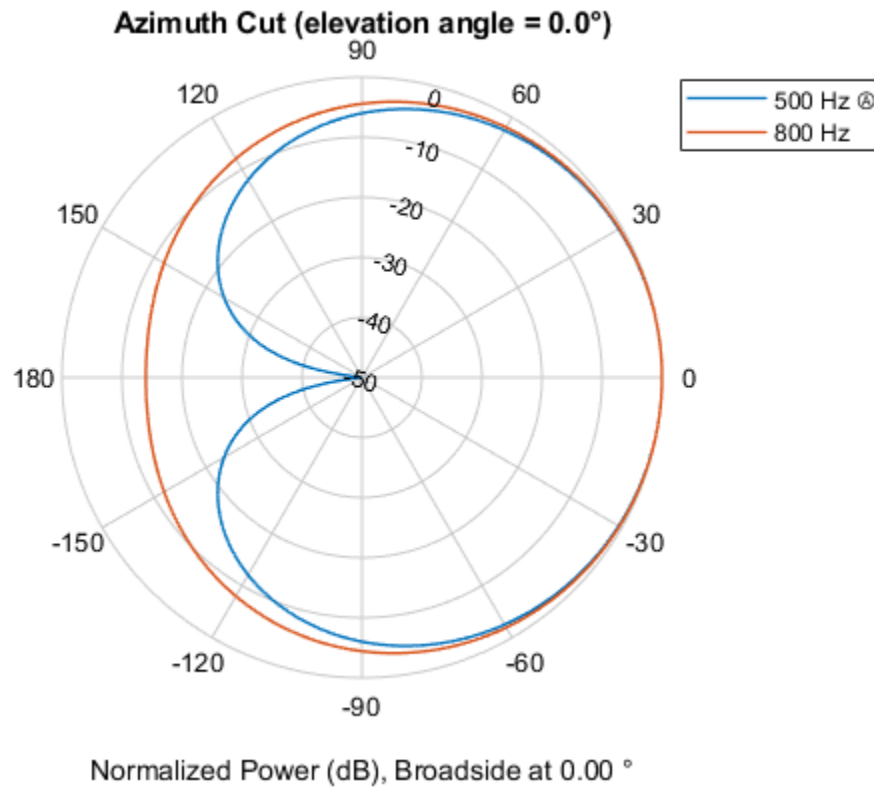
## See Also

## Related Examples
*   "Microphone ULA Array" on page 2-10

# Short-dipole Antenna Element

When you want to explicitly study the effects of polarization in a radar or communication system, you need to specify an antenna that can generate polarized radiation. One such antenna is the short-dipole antenna, created by using the `phased.ShortDipoleAntennaElement`.

The simplest polarized antenna is the dipole antenna which consist of a split length of wire coupled at the middle to a coaxial cable. The simplest dipole, from a mathematical perspective, is the Hertzian dipole, in which the length of wire is much shorter than a wavelength. A diagram of the short dipole antenna of length $L$ appears in the next figure. This antenna is fed by a coaxial feed which splits into two equal length wires of length $L/2$. The current, $I$, moves along the $z$-axis and is assumed to be the same at all points in the wire.



The electric field in the far field has the form

$$E_r = 0$$
$$E_H = 0$$
$$E_V = -\frac{iZ_0IL}{2\lambda}\text{cosel}\,\frac{e^{-ikr}}{r}$$

The next example computes the vertical and horizontal polarization components of the field. The vertical component is a function of elevation angle and is axially symmetric. The horizontal component vanishes everywhere.

## Short-Dipole Polarization Components

Compute the vertical and horizontal polarization components of the field created by a short-dipole antenna pointed along the *z*-direction. Plot the components as a function of elevation angle from 0° to 360°.

Create the `phased.ShortDipoleAntennaElement` System object™.

```
antenna = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[1,2]*1e9,'AxisDirection','Z');
```

Compute the antenna response. Because the elevation angle argument to `antenna` is restricted to ±90°, compute the responses for 0° azimuth and then for 180° azimuth. Combine the two responses in the plot. The operating frequency of the antenna is 1.5 GHz.

```
el = -90:90;
az = zeros(size(el));
fc = 1.5e9;
resp = antenna(fc,[az;el]);
az = 180.0*ones(size(el));
resp1 = antenna(fc,[az;el]);
```

Overlay the responses in the same figure.

```
figure(1)
subplot(121)
polarplot(el*pi/180.0,abs(resp.V.'),'b')
hold on
polarplot((el+180)*pi/180.0,abs(resp1.V.'),'b')
str = sprintf('%s\n%s','Vertical Polarization','vs Elevation Angle');
title(str)
hold off
subplot(122)
polarplot(el*pi/180.0,abs(resp.H.'),'b')
hold on
polarplot((el+180)*pi/180.0,abs(resp1.H.'),'b')
str = sprintf('%s\n%s','Horizontal Polarization','vs Elevation Angle');
title(str)
hold off
```

The plot shows that the horizontal component vanishes, as expected.

# Crossed-dipole Antenna Element

Another antenna that produces polarized radiation is the crossed-dipole antenna, created by using the `phased.CrossedDipoleAntennaElement` System object.

You can use a cross-dipole antenna to generate circularly-polarized radiation. The crossed-dipole antenna consists of two identical but orthogonal short-dipole antennas that are phased 90° apart. A diagram of the crossed dipole antenna appears in the following figure. The electric field created by a crossed-dipole antenna constructed from a $y$-directed short dipole and a $z$-directed short dipole has the form

$$E_r = 0$$

$$E_H = -\frac{iZ_0IL}{2\lambda}\text{cosaz}\frac{e^{-ikr}}{r}$$

$$E_V = \frac{iZ_0IL}{2\lambda}(\text{sinelsinaz} + i\text{cosel})\frac{e^{-ikr}}{r}$$

The polarization ratio $E_V/E_H$, when evaluated along the $x$-axis, is just $-i$ which means that the polarization is exactly RHCP along the $x$-axis. It is predominantly RHCP when the observation point is close to the $x$-axis. Moving away from the $x$-axis, the field becomes a mixture of LHCP and RHCP polarizations. Along the $-x$-axis, the field is LHCP polarized. The figure illustrates, for a point near the $x$, that the field is primarily RHCP.

## LHCP and RHCP Polarization Components

This example plots the right-hand and left-hand circular polarization components of fields generated by a crossed-dipole antenna at 1.5 GHz. You can see how the circular polarization changes from pure RHCP at 0 degrees azimuth angle to pure LHCP at 180 degrees azimuth angle, both at 0 degrees elevation angle.

Create the `phased.CrossedDipoleAntennaElement` object.

```
fc = 1.5e9;
antenna = phased.CrossedDipoleAntennaElement('FrequencyRange',[1,2]*1e9);
```

Compute the left-handed and right-handed circular polarization components from the antenna response.

```
az = [-180:180];
el = zeros(size(az));
resp = antenna(fc,[az;el]);
cfv = pol2circpol([resp.H.';resp.V.']);
clhp = cfv(1,:);
crhp = cfv(2,:);
```

Plot both circular polarization components at 0 degrees elevation.

```
polar(az*pi/180.0,abs(clhp))
hold on
polar(az*pi/180.0,abs(crhp))
title('LHCP and RHCP vs Azimuth Angle')
legend('LHCP','RHCP')
hold off
```

LHCP and RHCP vs Azimuth Angle

# Gaussian Antenna as Approximation for Spiral Antenna

This example shows how a Gaussian antenna can approximate the radiation pattern of a spiral antenna. Spiral antennas are known for their wideband behavior and are often used in wideband communications. The Gaussian antenna approximation is widely used in the literature.

Create a spiral antenna using Antenna Toolbox™. The antenna is in the *y-z* plane and has inner and outer radii of 0.65 mm and 40 mm, respectively.

```
se = spiralArchimedean( ...
    InnerRadius=0.65e-3,OuterRadius=40e-3, ...
    Tilt=90,TiltAxis="y");
show(se)
```
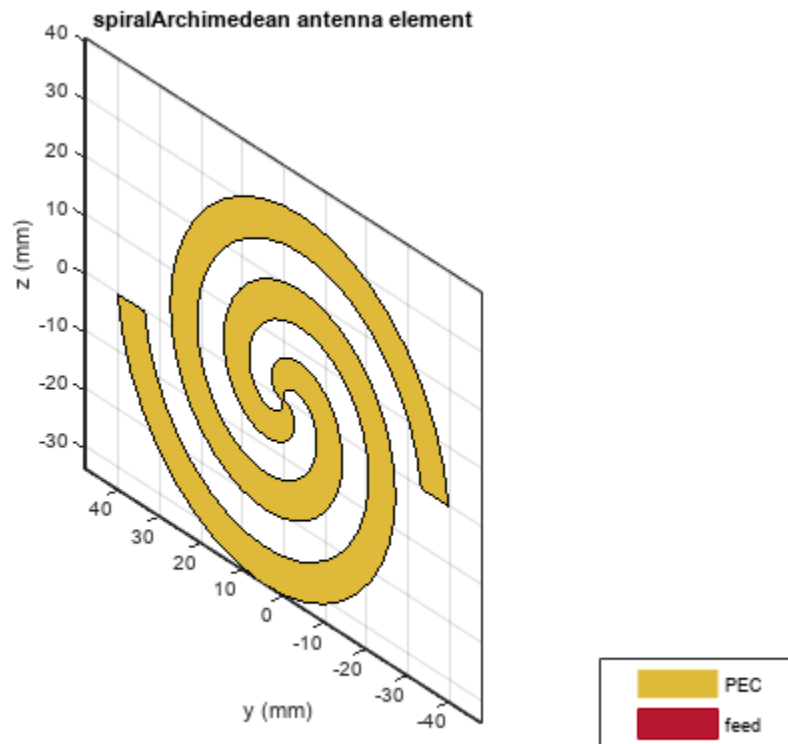


Compute the radiation pattern for the spiral antenna at an operating frequency of 4 GHz. Specify a range of azimuth angles from –90° to 90° and zero elevation. Normalize the pattern so that its maximum value is 0 dB.

```
fc = 4e9;
az = -90:0.5:90;

sePattern = pattern(se,fc,az,0, ...
    CoordinateSystem="rectangular",Type="powerdb");

sePatternNorm = sePattern - max(sePattern);
```

Use Phased Array System Toolbox™ to create a Gaussian element with the same beamwidth as the spiral antenna. Compute its radiation pattern, which is normalized by construction.

```
seBw = beamwidth(se,fc,az,0);
ge = phased.GaussianAntennaElement(Beamwidth=seBw);

gePattern = pattern(ge,fc,az,0, ...
    CoordinateSystem="rectangular",Type="powerdb");
```

Find the smallest positive azimuth angle at which the patterns differ by about 3 dB.

```
idx = find((abs(sePatternNorm.' - gePattern) >= 3) & (az' >= 0),1);
az3dB = az(idx);
```

Plot the patterns for the spiral and Gaussian antennas. Overlay the 3-dB points. The Gaussian antenna pattern matches the spiral antenna pattern well out to about 75 degrees and thus can be used as an excellent approximation of a spiral antenna.

```
plot(az,sePatternNorm,az,gePattern)

xline([-az3dB az3dB],'--')
xlabel("Azimuth Angle, az (degrees)")
ylabel("Power (dB)")
legend("Spiral","Gaussian","3-dB Point")
```
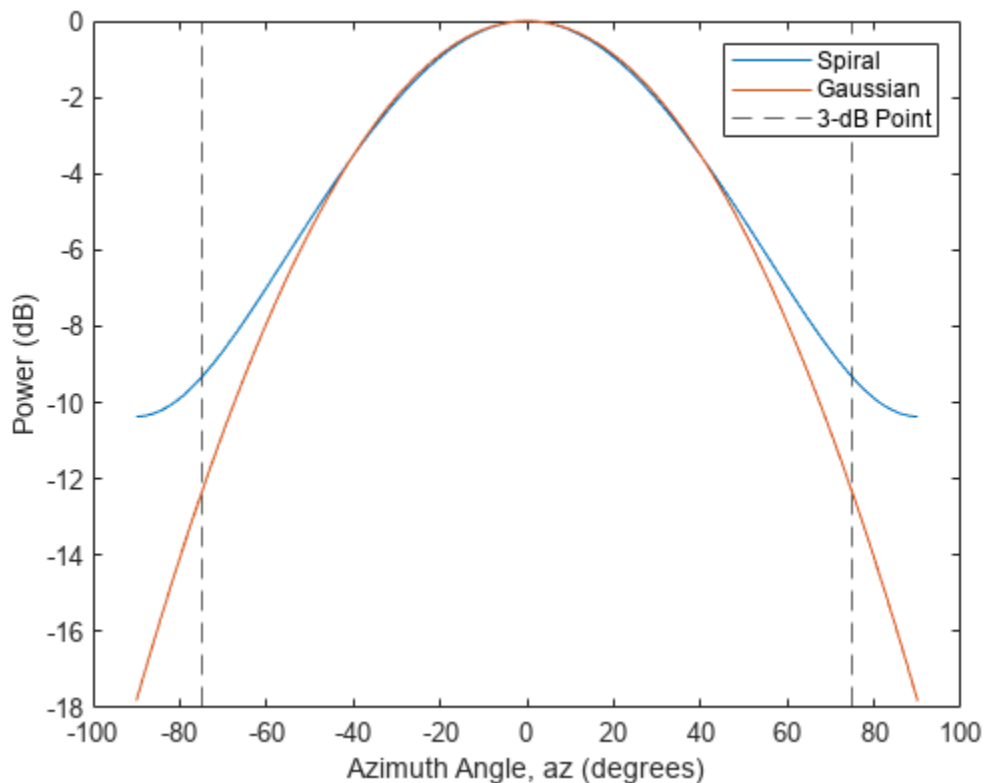
**See Also**

phased.GaussianAntennaElement | spiralArchimedean

# Using Antenna Toolbox with Phased Array Systems

When you create antenna arrays such as a uniform linear array (ULA), you can use antennas that are built into Phased Array System Toolbox™. Alternatively, you can use Antenna Toolbox™ antennas. Antenna Toolbox antennas provide realistic models of physical antennas. They are designed using method of moments. Phased array antennas represent more idealized antennas that are useful for radar performance analysis and higher level modelling. Some phased array antennas cannot be physically realized, such as the isotropic antenna but are still conceptually useful. You can build and analyze systems using both types of antennas in an identical manner. This example shows how to construct a phased array with either Phased Array System Toolbox or Antenna Toolbox antennas.

When you use an Antenna Toolbox antenna in a Phased Array System Toolbox System object™, the antenna response will be normalized by the maximum value of the antenna output over all directions. The maximum value is obtained by finding the maximum of the antenna pattern sampled every five degrees in azimuth and elevation.

### Construct ULA of Crossed-Dipole Antennas from Phased Array System Toolbox

Start by creating a uniform linear array (ULA) of crossed-dipole antennas from Phased Array System Toolbox. Crossed-dipole antennas are used to produce circularly-polarized signals. In this case, set the operating frequency to 2 GHZ and draw the power pattern. Use the `pattern` method of the `phased.CrossedDipoleAntennaElement` System object.

```
fc = 2.0e9;
crosseddipoleantenna = phased.CrossedDipoleAntennaElement('FrequencyRange',[500,2500]*1e6);
pattern(crosseddipoleantenna,fc,[-180:180],0,...
    'Type','powerdb')
```
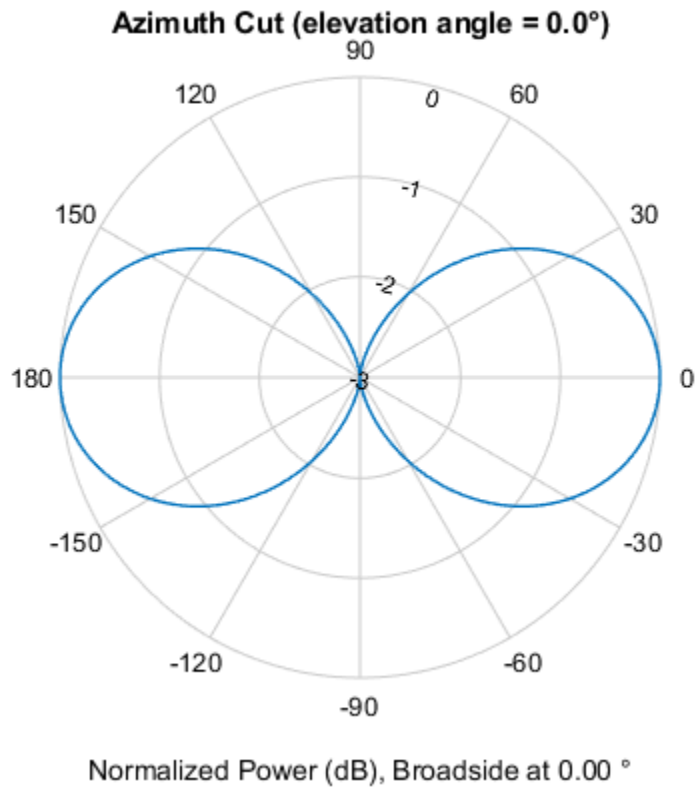
The main axis of this antenna points along the *x*-axis.

Then, create an 11-element ULA array of crossed-dipole antennas. Specify the element spacing to be 0.4 wavelengths. Taper the array using a Taylor window. Then, draw the array pattern as a function of azimuth at 0 degrees elevation. Use the `pattern` method of the `phased.ULA` System object.

```
c = physconst('LightSpeed');
elemspacing = 0.4*c/fc;
nElements = 11;
array1 = phased.ULA('Element',crosseddipoleantenna,'NumElements',nElements,...
    'ElementSpacing',elemspacing,'Taper',taylorwin(nElements)');
pattern(array1,fc,[-180:180],0,'PropagationSpeed',c,...
        'Type','powerdb')
```

Normalized Power (dB), Broadside at 0.00 °
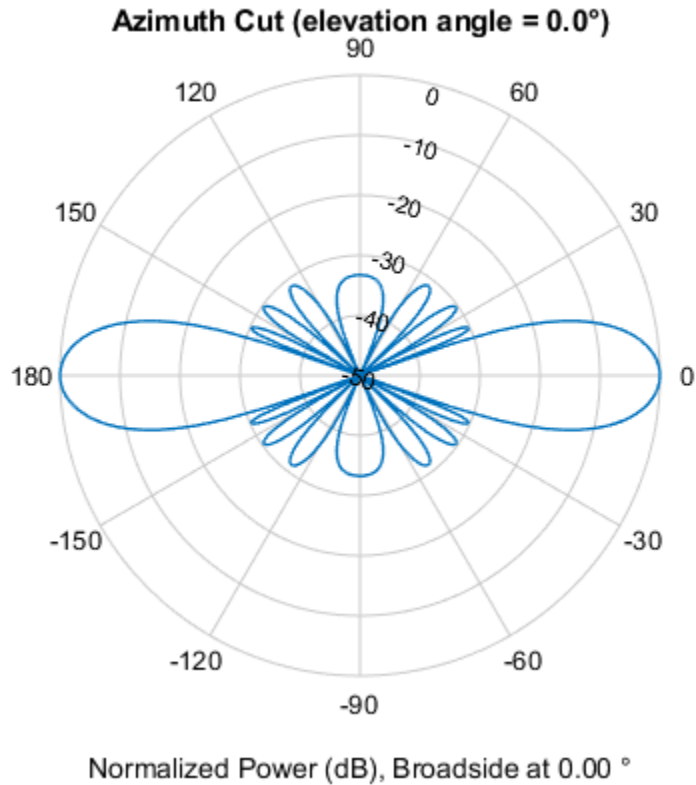
**Construct ULA of Helix Antennas from Antenna Toolbox**

Next, create a uniform linear array (ULA) using the helix antenna from Antenna Toolbox. Helix antennas also produce circularly polarized radiation. Helix antennas are created using the `helix` function.

First, specify a 4-turn helix antenna having a 28.0 mm radius and 1.2 mm width. The `TiltAxis` and `Tilt` properties let you orient the antenna with respect to the local coordinate system. In this example, orient the main response axis (MRA) along the $x$ -axis to coincide with the MRA of the cross-dipole main axis. By default, the MRA of the antenna points in the $z$ -direction. Rotate the MRA around the $y$ -axis by 90 degrees.

```
radius = 0.028;
width = 1.2e-3;
nturns = 4;
helixantenna = helix('Radius',radius,'Width',width,'Turns',nturns,...
    'TiltAxis',[0,1,0],'Tilt',90);
```

You can view the shape of the helix antenna use the `show` function from Antenna Toolbox.

```
show(helixantenna)
```
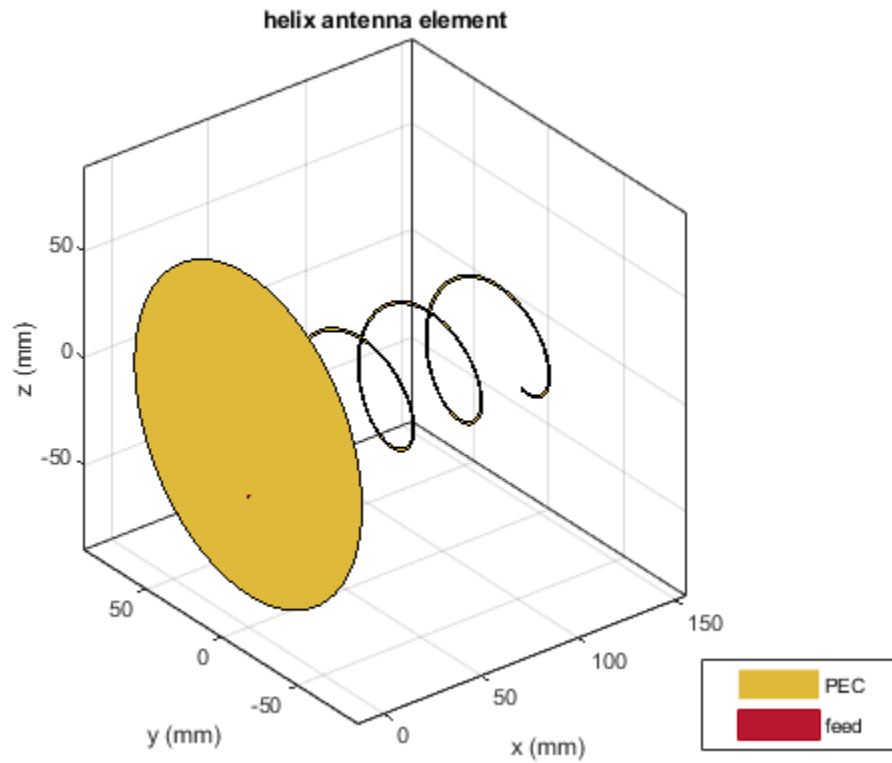
Then, draw the azimuth antenna pattern at 0 degrees elevation at the operating frequency of 2 GHz. Use the `pattern` function from Antenna Toolbox.

```
pattern(helixantenna,fc,[-180:180],0,...
    'Type','powerdb')
```
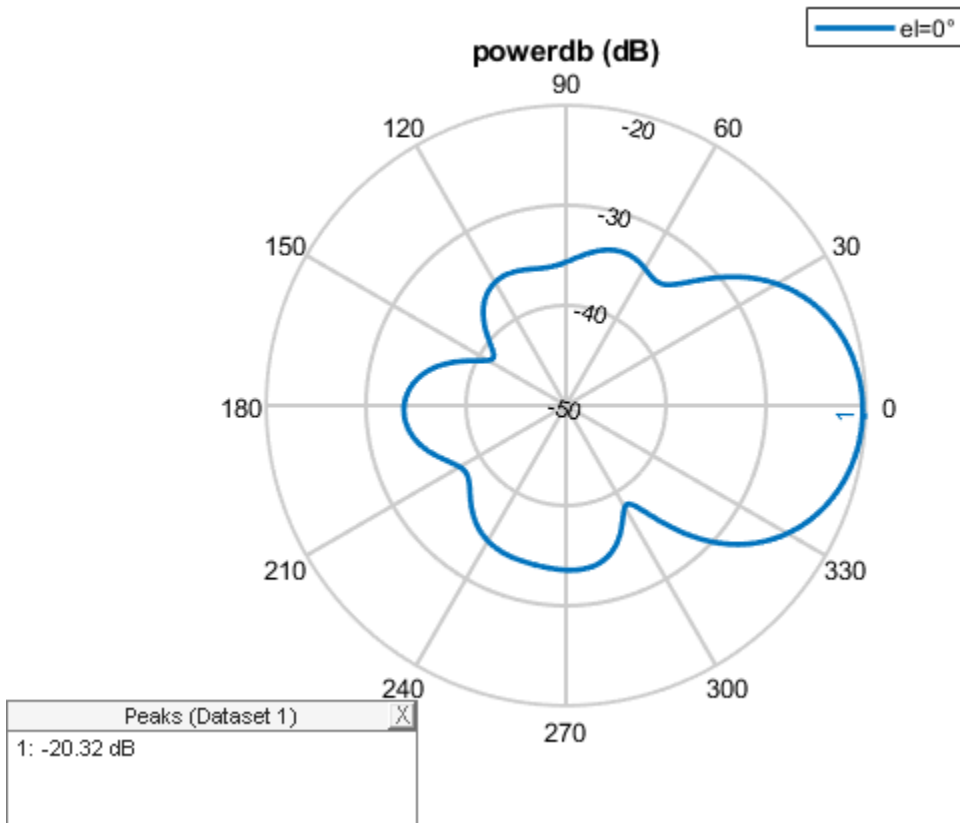
Next, construct an 11-element tapered uniform linear array of helix antennas with elements spaced at 0.4 wavelengths. Taper the array with a Taylor window. You can use the same `phased.ULA` System object from Phased Array System Toolbox to create this array.

```
array2 = phased.ULA('Element',helixantenna,'NumElements',nElements,...
    'ElementSpacing',elemspacing,'Taper',taylorwin(nElements)');
```

Plot the array pattern as a function of azimuth using the ULA pattern `method` which has the same syntax as the Antenna Toolbox `pattern` function.

```
pattern(array2,fc,[-180:180],0,'PropagationSpeed',c,...
        'Type','powerdb')
```

**Azimuth Cut (elevation angle = 0.0°)**

Normalized Power (dB), Broadside at 0.00 °
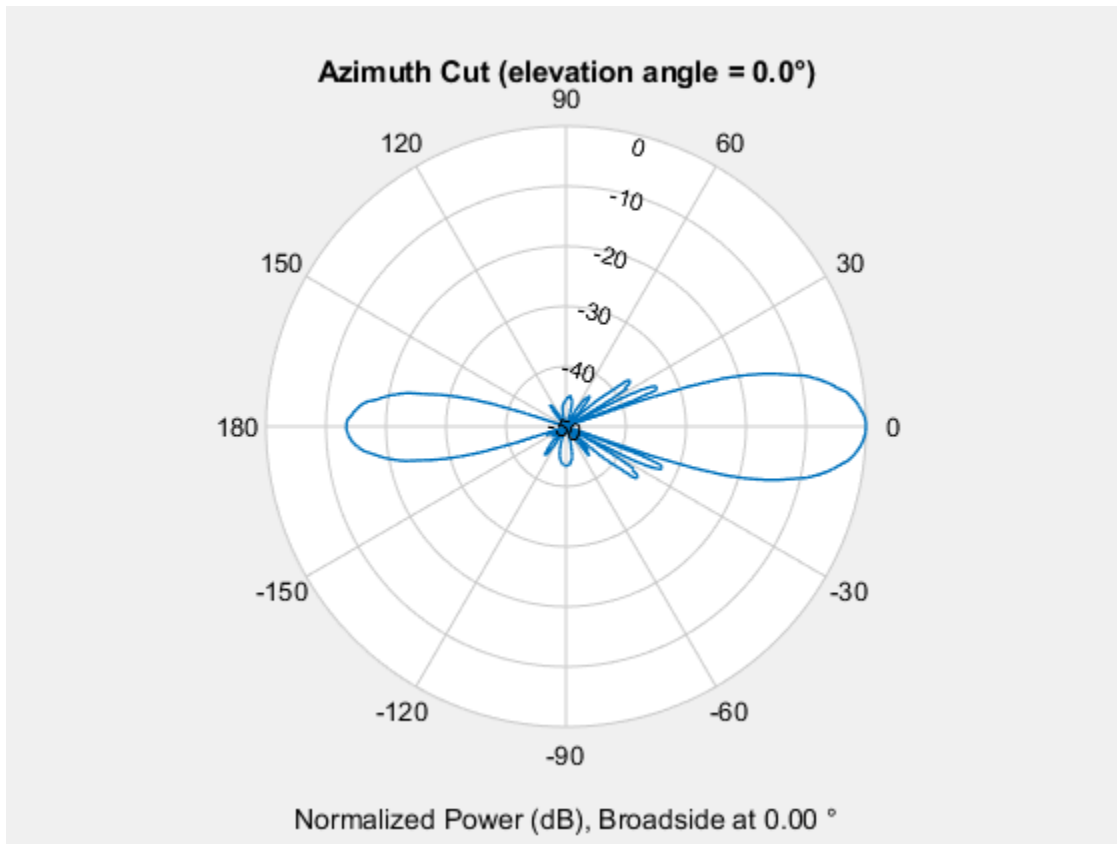
**Compare Patterns**

Comparing the two array patterns shows that they are similar along the mainlobe. The backlobe of the helix antenna array pattern is almost 15 dB smaller than that of the crossed-dipole array. This is due to the presence of the ground plane of the helix antenna which reduces backlobe transmission.

# Sinc Antenna as Approximation for Array Response Pattern

This example shows how a sinc antenna element can approximate the radiation pattern of an array.

Create a 5-by-2 rectangular array of isotropic elements designed to work at an operating frequency of 3 GHz. Specify an element spacing equal to half the wavelength.

```
fc = 3e9;
lambda = freq2wavelen(fc);
array = phased.URA('ElementSpacing',lambda/2,'Size',[5 2]);
```

Compute the radiation pattern of the array. Specify a range of azimuth angles from –90° to 90° and zero elevation. Plot the unnormalized azimuth pattern in polar coordinates.

```
az = -90:0.5:90;

recopts = {'CoordinateSystem','rectangular','Type','powerdb'};
polopts = {'CoordinateSystem','polar','Type','powerdb'};

arrayPatternAz = pattern(array,fc,az,0,recopts{:});

pattern(array,fc,az,0,polopts{:})
```
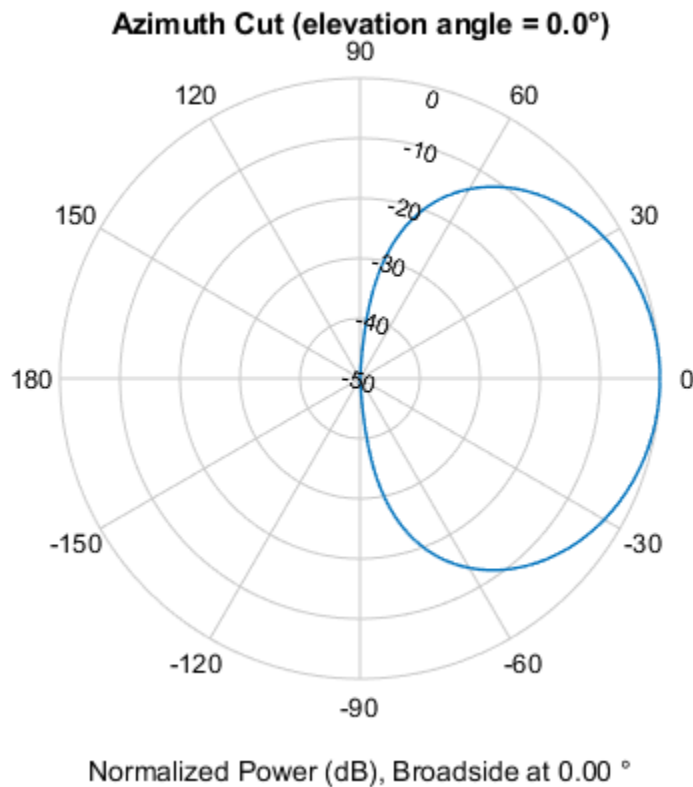


Normalize the pattern so that its maximum value is 0 dB. Compute the azimuth beamwidth.

```
arrayPatternAzNorm = arrayPatternAz - max(arrayPatternAz);
arrayAzBw = beamwidth(array,fc);
```

Compute the radiation for a range of elevation angles from –90° to 90° and zero azimuth. Normalize the pattern and compute the beamwidth. Plot the unnormalized elevation pattern in polar coordinates.

```
el = -90:0.5:90;
```

```
arrayPatternEl = pattern(array,fc,0,el,recopts{:});
```

```
arrayPatternElNorm = arrayPatternEl - max(arrayPatternEl);
arrayElBw = beamwidth(array,fc,'Cut','Elevation');
```

```
pattern(array,fc,0,el,polopts{:})
```
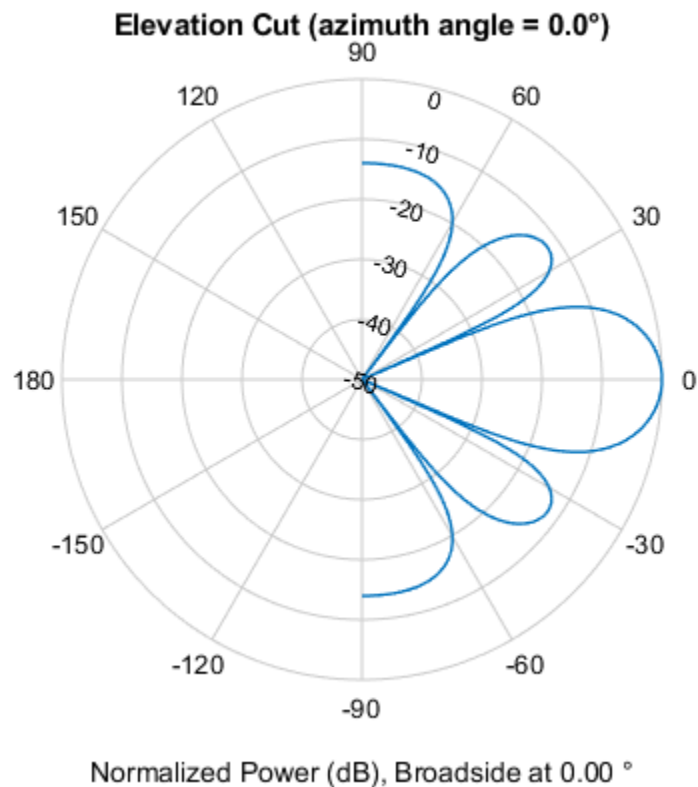


Create a sinc antenna element with the same beamwidth as the array. Compute the azimuth and elevation patterns. The patterns are normalized by construction.

```
se = phased.SincAntennaElement('Beamwidth',[arrayAzBw arrayElBw]);
```

```
sePatternAz = pattern(se,fc,az,0,recopts{:});
sePatternEl = pattern(se,fc,0,el,recopts{:});
```

Find the smallest positive azimuth angle at which the patterns differ by about 3 dB. Plot the azimuth patterns in rectangular coordinates. Overlay the 3-dB points. The sinc azimuth pattern matches the array azimuth pattern well out to about 60 degrees.

```
idxAz = find((abs(arrayPatternAzNorm - sePatternAz) >= 3) & (az' >= 0),1);
az3dB = az(idxAz);

figure
plot(az,arrayPatternAzNorm,az,sePatternAz)
xline([-az3dB az3dB],'--')

title("Az Cut: Normalized Array and Sinc Antenna Patterns")
xlabel("Azimuth Angle, az (degrees)")
ylabel("Power (dB)")
legend("Array","Sinc","3-dB Point",Location="south")
ylim([-50 0])
```
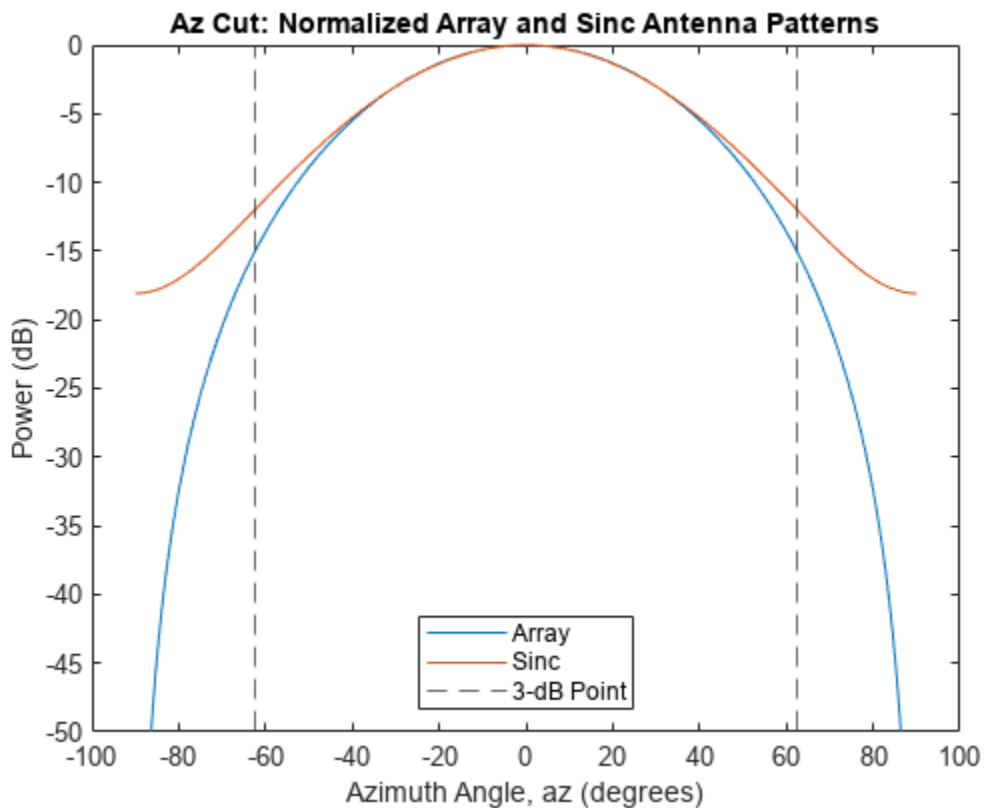


Repeat the process for the elevation patterns. The sinc pattern is a good approximation out to about 20 degrees.

```
idxEl = find((abs(arrayPatternElNorm - sePatternEl) >= 3) & (el' >= 0),1);
el3dB = el(idxEl);

plot(el,arrayPatternElNorm,az,sePatternEl)
xline([-el3dB el3dB],'--')

title("El Cut: Normalized Array and Sinc Antenna Patterns")
xlabel("Elevation Angle, el (degrees)")
ylabel("Power (dB)")
legend("Array","Sinc","3-dB Point")
ylim([-50 0])
```

El Cut: Normalized Array and Sinc Antenna Patterns
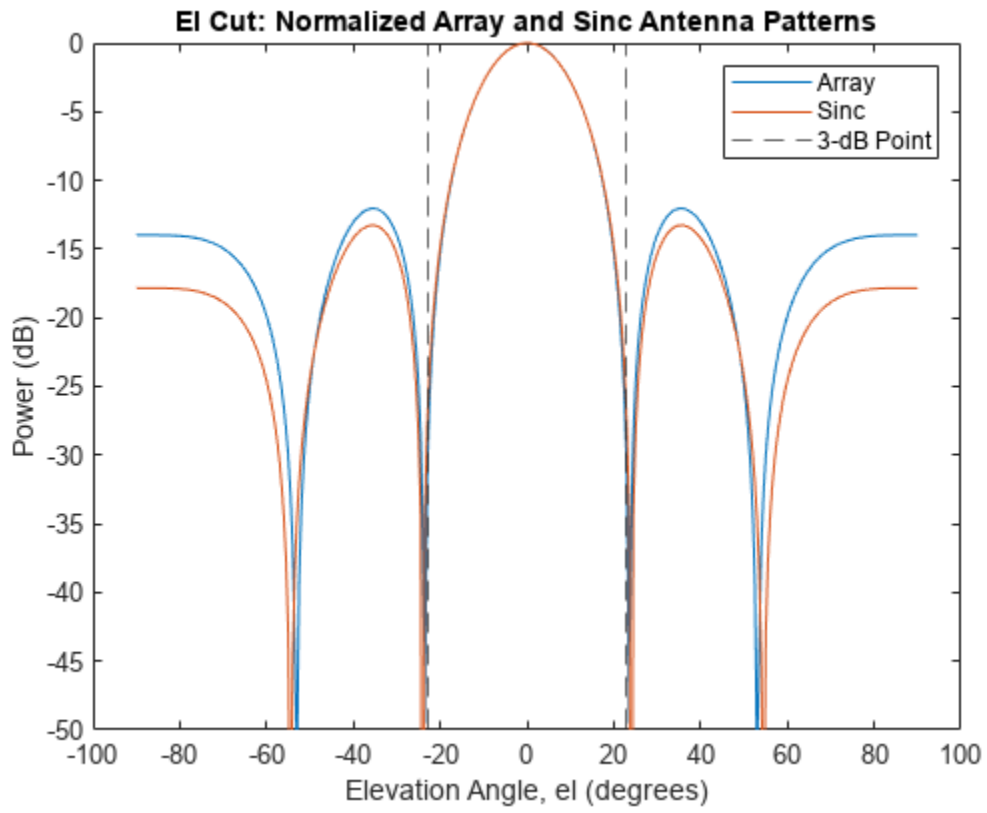
## See Also
`phased.SincAntennaElement` | `phased.URA`

# Array Geometries and Analysis

# Uniform Linear Array

## Support for Uniform Linear Arrays

The uniform linear array (ULA) arranges identical sensor elements along a line in space with uniform spacing. You can design a ULA with `phased.ULA`. When you use this object, you must specify these aspects of the array:

- Sensor elements of the array
- Spacing between array elements
- Number of elements in the array

## Positions of ULA Array Elements

Create and view a ULA having four isotropic antenna elements separated by 0.5 meters

```
array = phased.ULA('NumElements',4,'ElementSpacing',0.5);
viewArray(array);
```

**Array Geometry**
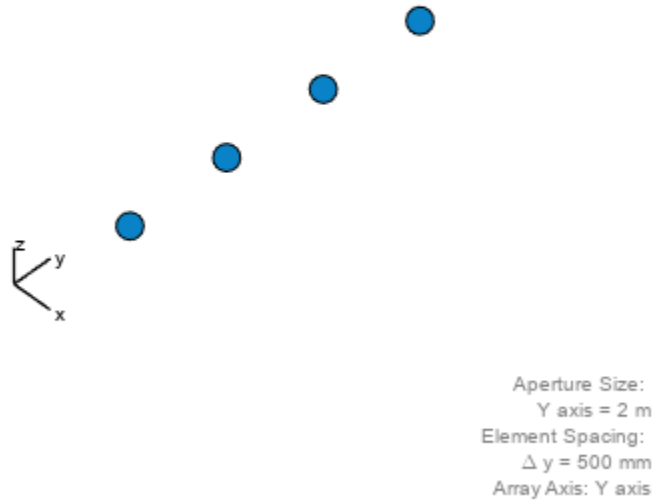


You can return the coordinates of the array sensor elements in the form `[x;y;z]` by using the `getElementPosition` method. See "Rectangular Coordinates" on page 10-2 for toolbox conventions.

```
sensorpos = getElementPosition(array);
```

`sensorpos` is a 3-by-4 matrix with each column representing the position of a sensor element. Note that the *y*-axis is the array axis. The positive *x*-axis is the array look direction (0 degrees broadside). The elements are symmetric with the respect to the phase center of the array.

## ULA Array Elements

The default element for a ULA is the `phased.IsotropicAntennaElement` object. You can specify a different element using the `Element` property.

## Array Element Responses

Obtain the responses of the elements of a 4-element ULA array at 1 GHz.

Specify isotopic antennas for the array elements. Then, specify a 4-element ULA. Obtain the response by executing the System object™.

```
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[3e8 1e9]);
```

```
array = phased.ULA('NumElements',4,'ElementSpacing',0.5,...
    'Element',antenna);
freq = 1e9;
azangles = -180:180;
response = array(freq,azangles);
```

`response` is a 4-by-361 matrix where each column contains the responses at each azimuth angle. Matrix rows correspond to the four elements. Because the elements of the ULA are isotropic antennas, `response` is a matrix of ones.

## Signal Delay Between Array Elements

This example computes the delay between elements of a 4-element ULA using the `phased.ElementDelay` System object™. Assume that the incident waveform satisfies the far-field condition. The delays are computed with respect to the phase center of the array. By default, `phased.ElementDelay` assumes that the incident waveform is an electromagnetic wave propagating at the speed of light.

Construct 4-element ULA using value-only syntax and compute the delay for a signal incident on the array from -90° azimuth and 0° elevation. Delay units are in seconds.

```
array = phased.ULA(4);
delay = phased.ElementDelay('SensorArray',array);
tau = delay([-90;0])
```

tau = *4×1*
$10^{-8}$ ×

    -0.2502
    -0.0834
     0.0834
     0.2502

`tau` is a 4-by-1 vector of delays with respect to the phase center of the array, which is the origin of the local coordinate system *(0;0;0)*. See "Global and Local Coordinate Systems" on page 10-17 for a description of global and local coordinate systems. Negative delays indicate that the signal arrives at an element before reaching the phase center of the array. Because the waveform arrives from an azimuth angle of -90°, the signal arrives at the first and second elements of the ULA before it reaches the phase center resulting in negative delays for these elements.

If the signal is incident on the array at 0° broadside from a far-field source, the signal illuminates all elements of the array simultaneously resulting in zero delay.

```
tau = delay([0;0])
```

tau = *4×1*

     0
     0
     0
     0

If the incident signal is an acoustic pressure waveform propagating at the speed of sound, you can calculate the element delays by setting the `PropagationSpeed` property to 340 m/s. This value is a typical speed of sound at sea level.

```
delay = phased.ElementDelay('SensorArray',array,...
    'PropagationSpeed',340);
tau = delay([90;0])
```

```
tau = 4×1

    0.0022
    0.0007
   -0.0007
   -0.0022
```

## Steering Vector

The *steering vector* represents the relative phase shifts for the incident far-field waveform across the array elements. You can determine these phase shifts with the `phased.SteeringVector` object.

For a single carrier frequency, the steering vector for a ULA consisting of *N* elements is:

$$\begin{pmatrix} e^{-j2\pi f\tau_1} \\ e^{-j2\pi f\tau_2} \\ e^{-j2\pi f\tau_3} \\ . \\ . \\ . \\ e^{-j2\pi f\tau_N} \end{pmatrix}$$

where $\tau_n$ denotes the time delay relative to the array phase center at the n-th array element.

**Compute ULA Steering Vector**

Compute the steering vector for a 4-element ULA at an operating frequency of 1 GHz. Assume that the waveform is incident on the array from 45° azimuth and 10° elevation.

```
fc = 1e9;
array = phased.ULA(4);
steervec = phased.SteeringVector('SensorArray',array);
sv = steervec(fc,[45;10])
```

```
sv = 4×1 complex

  -0.0495 + 0.9988i
  -0.8742 + 0.4856i
  -0.8742 - 0.4856i
  -0.0495 - 0.9988i
```

You can also compute the steering vector with the following equivalent code.

```
delay = phased.ElementDelay('SensorArray',array);
tau = delay([45;10]);
exp(-1i*2*pi*fc*tau)
```

```
ans = 4×1 complex

  -0.0495 + 0.9988i
  -0.8742 + 0.4856i
  -0.8742 - 0.4856i
  -0.0495 - 0.9988i
```

## Array Response

To obtain the array response, which is a weighted-combination of the steering vector elements for each incident angle, use the `phased.ArrayResponse` System object.

**ULA Array Response**

Construct a four-element ULA with elements spaced at 0.25 m. Obtain the array magnitude response (absolute value of the complex-valued array response) for azimuth angles *(-180:180)* at 1 GHz. Then, plot the normalized magnitude response in dB.

```
fc = 1e9;
array = phased.ULA('NumElements',4,'ElementSpacing',0.25);
azangles = -180:180;
response = phased.ArrayResponse('SensorArray',array);
resp = abs(response(fc,azangles));
plot(azangles,mag2db((resp/max(resp))))
grid on
title('Azimuth Cut at Zero Degrees Elevation')
xlabel('Azimuth Angle (degrees)')
```

**Azimuth Cut at Zero Degrees Elevation**
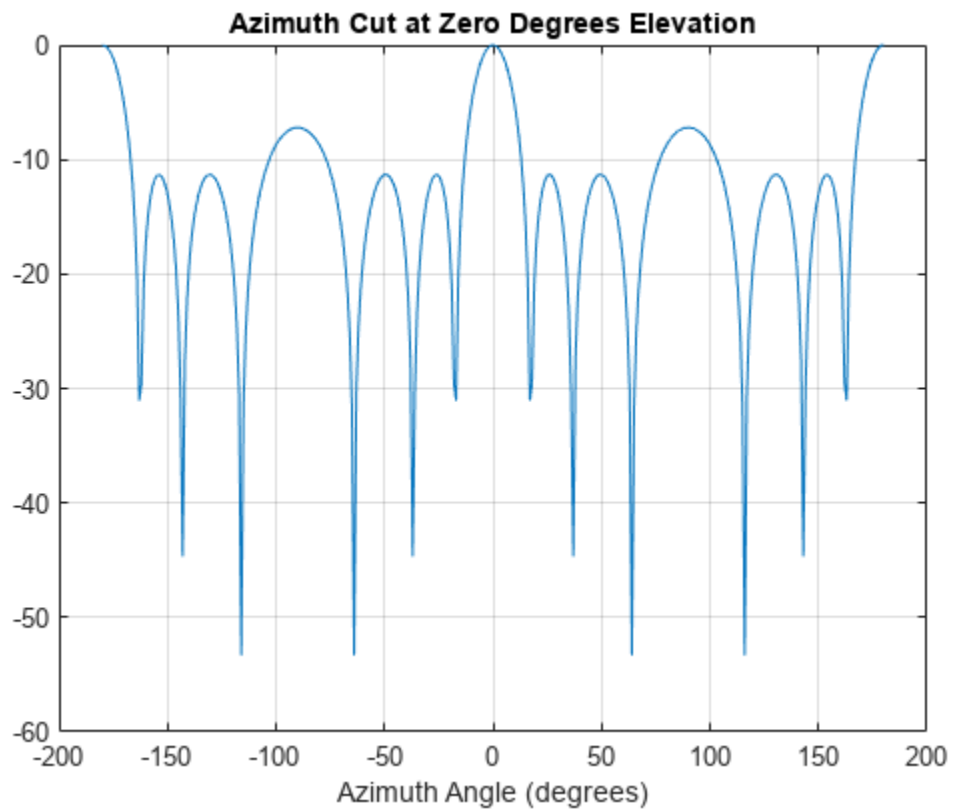


Visualize the array response using the `pattern` method. Create a 3-D plot of the response in *UV* space; other plotting options are available.

```
pattern(array,fc,[-1:.01:1],[-1:.01:1],'CoordinateSystem','uv',...
    'PropagationSpeed',physconst('Lightspeed'))
```

3D Directivity Pattern in u-v space
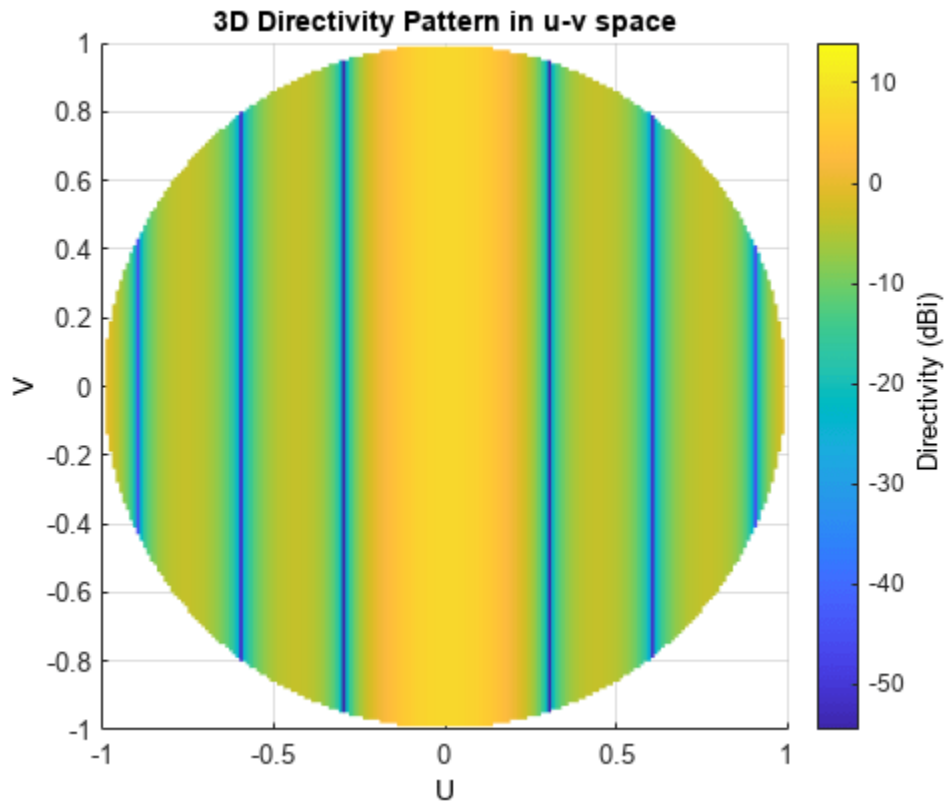
## Reception of Plane Wave Across Array

You can simulate the effects of phase shifts across your array using the `collectPlaneWave` method of any array System object.

The `collectPlaneWave` method modulates input signals by the element of the steering vector corresponding to an array element. Stated differently, `collectPlaneWave` accounts for phase shifts across elements in the array based on the angle of arrival. However, `collectPlaneWave` does not account for the response of individual elements in the array.

### Plane-Wave Reception Across ULA

Simulate the reception of a 100-Hz sine wave modulated by a carrier frequency of 1 GHz at a 4-element ULA. Assume the angle of arrival of the signal is *(-90;0)*.

```
array = phased.ULA(4);
t = unigrid(0,0.001,0.01,'[)');
x = cos(2*pi*100*t)';
y = collectPlaneWave(array,x,[-90;0],1e9,physconst('LightSpeed'));
```

The preceding code is equivalent to the following.

```
steervec = phased.SteeringVector('SensorArray',array);
sv = steervec(1e9,[-90;0]);
y1 = x*sv.';
```

**See Also**

**Related Examples**

- "Microphone ULA Array" on page 2-10

# Microphone ULA Array

This example shows how to construct and visualize a four-element ULA with custom cardioid microphone elements. Specify the polar pattern frequencies as 500 and 1000 Hz.

Create a microphone element with a cardioid response pattern. Use the default values of the `FrequencyVector` property.

```
freq = [500 1000];
microphone = phased.CustomMicrophoneElement(...
    'PolarPatternFrequencies',freq);
microphone.PolarPattern= mag2db([...
    0.5+0.5*cosd(microphone.PolarPatternAngles);...
    0.6+0.4*cosd(microphone.PolarPatternAngles)]);
```

Plot the polar pattern of the microphone at 0.5 kHz and 1 kHz.

```
pattern(microphone,freq,[-180:180],0,'CoordinateSystem','polar','Type','powerdb',...
    'Normalize',true);
```
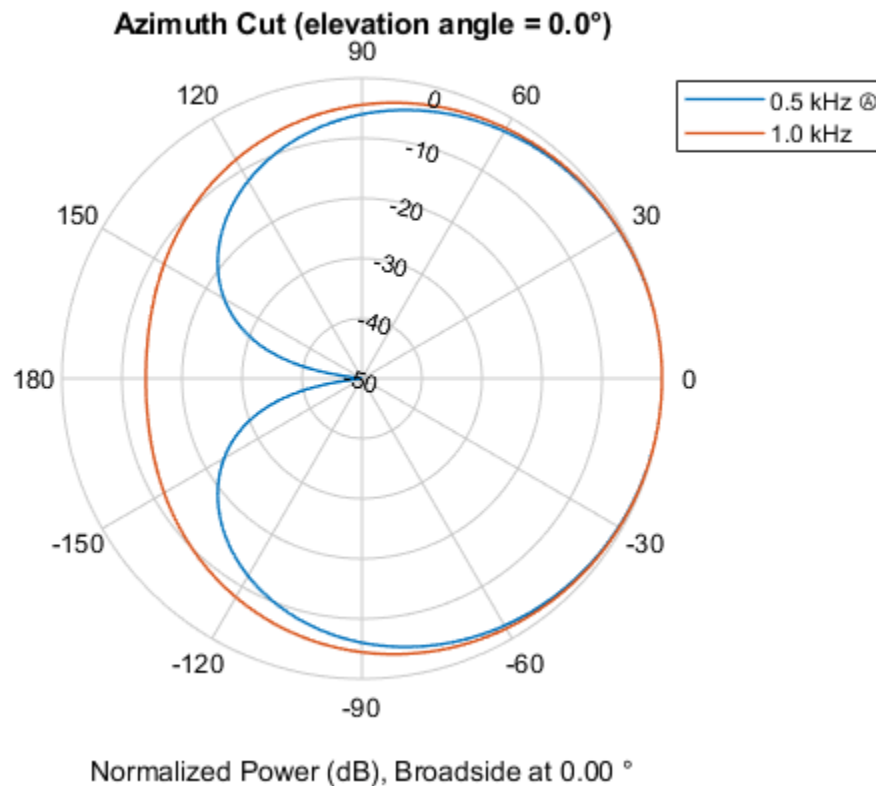


Construct a ULA of custom microphone elements.

```
array = phased.ULA('NumElements',4,'ElementSpacing',0.5,...
    'Element',microphone);
```

Plot the response of the array at 0.5 kHz and 1 kHz.

```
pattern(array,freq,[-180:180],0,'CoordinateSystem','polar','Type','powerdb',...
    'Normalize',true,'PropagationSpeed',340.0);
```



Azimuth Cut (elevation angle = 0.0°)

Normalized Power (dB), Broadside at 0.00 °
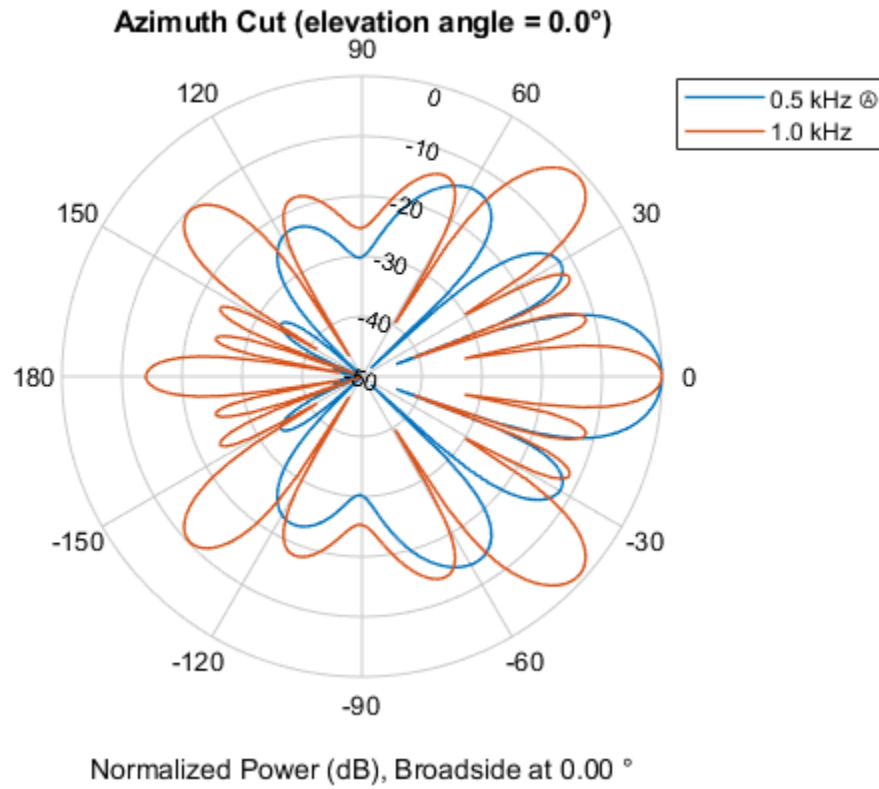
# Uniform Rectangular Array

| **In this section...** |
| --- |
| "Support for Uniform Rectangular Arrays" on page 2-12 |
| "Uniform Rectangular Array of Isotropic Antenna Elements" on page 2-12 |

## Support for Uniform Rectangular Arrays

You can implement a uniform rectangular array (URA) with `phased.URA`. Array elements are distributed in the *yz*-plane with the array look direction along the positive *x*-axis. When you use `phased.URA`, you must specify these aspects of the array:

- Sensor elements of the array
- Number of rows and the spacing between them
- Number of columns and the spacing between them
- Geometry of the planar lattice, which can be rectangular or triangular

## Uniform Rectangular Array of Isotropic Antenna Elements

This example shows you how to create a uniform rectangular array (URA) and obtain information about the element positions, the array response, and inter-element time delays. Then, simulate the reception of two sine waves coming from different directions. Both signals have a 1GHz carrier frequency.

### Create the URA and obtain the element positions

Create and view a six-element URA with two elements along the *y*-axis and three elements along the *z*-axis. Use a rectangular lattice, with the default spacing of 0.5 meters along both the row and column dimensions of the array. Each element is an isotropic antenna element, which is the default element type for a URA.

```
fc = 1e9;
array = phased.URA([3,2]);
viewArray(array)
```

**Array Geometry**



```
Aperture Size:
  Y axis = 1.0 m
  Z axis = 1.5 m
Element Spacing:
  Δ y = 500 mm
  Δ z = 500 mm
```
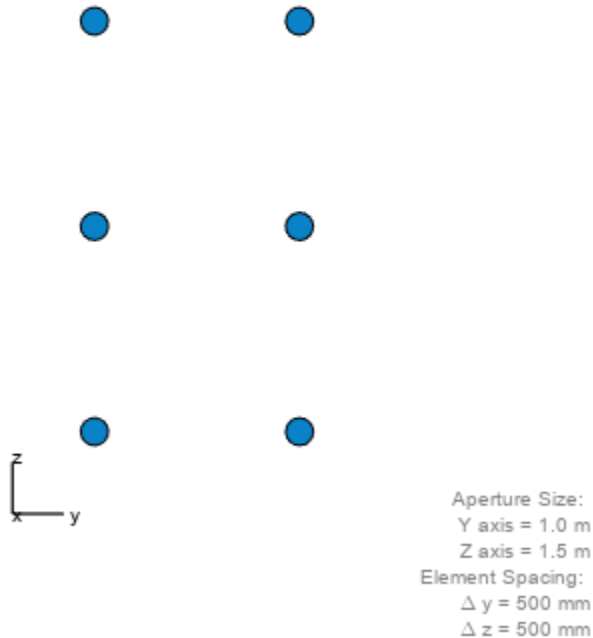
```
pos = getElementPosition(array);
```

The *x*-coordinate is zero for all elements of the array.

**Compute the element delays**

Calculate the element delays for signals arriving from +45° and -45° azimuth and 0° elevation.

```
delay = phased.ElementDelay('SensorArray',array);
ang = [45,-45];
tau = delay(ang);
```

The first column of `tau` contains the element delays for the signal incident on the array from +45° azimuth. The second column contains the delays for the signal arriving from -45°. The delays are equal in magnitude but opposite in sign, as expected.

**Compute the received signals**

The following code simulates the reception of two sinusoidal waves arriving from far field sources. One signal is a 100-Hz sine wave arriving from 20° azimuth and 10° elevation. The second signal is a 300-Hz sine wave arriving from -30° azimuth and 5° elevation.

```
t = linspace(0,1,1000);
x1 = cos(2*pi*100*t)';
x2 = cos(2*pi*300*t)';
ang1 = [20;10];
ang2 = [-30;5];
recsig = collectPlaneWave(array,[x1 x2],[ang1 ang2],fc);
```

Each column of `recsig` represents the received signals at the corresponding element of the URA.

**Plot the array response in 3D**

You can plot the array response using the `pattern` method.

```
pattern(array,fc,[-180:180],[-90:90],'PropagationSpeed',physconst('LightSpeed'),...
    'CoordinateSystem','rectangular','Type','powerdb')
```
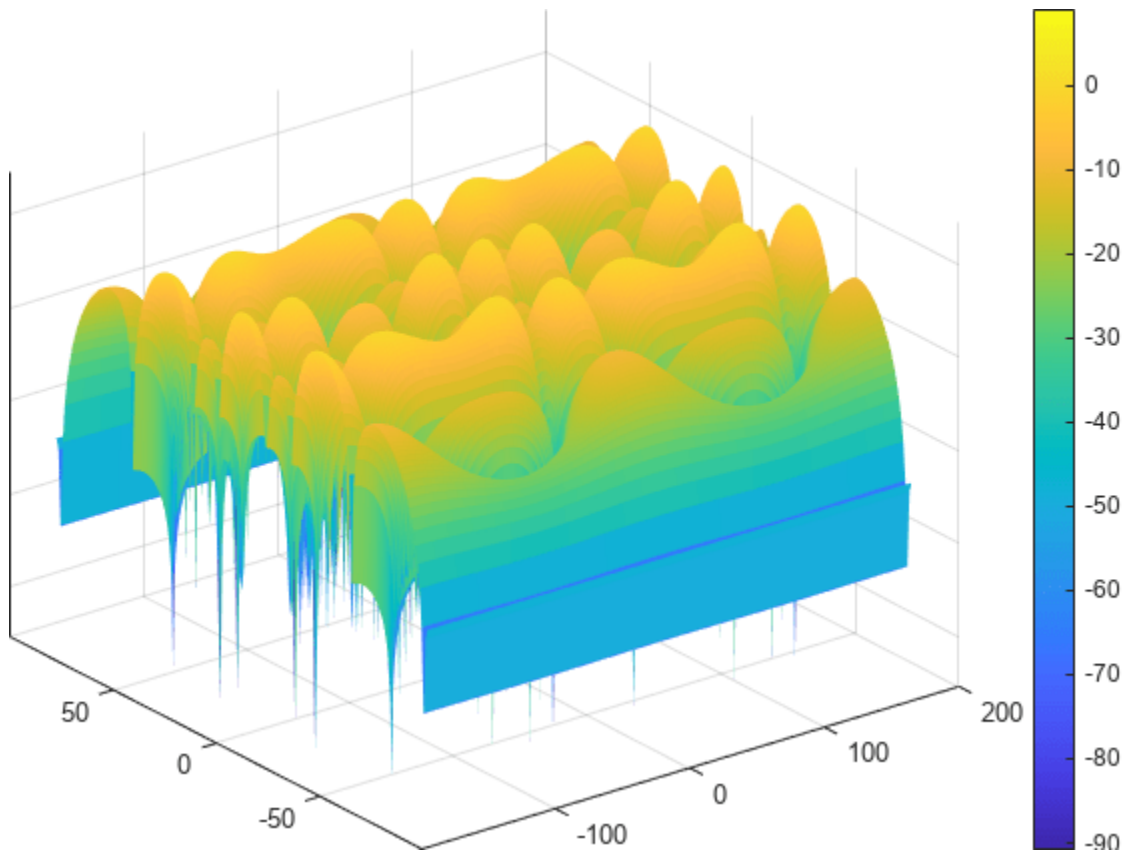
# Conformal Array

| In this section... |
| --- |
| |
| |
| |
| |

## Support for Arrays with Custom Geometry

The `phased.ConformalArray` object lets you model a phased array with arbitrary geometry. For example, you can use `phased.ConformalArray` to design:

- A planar array with a nonrectangular geometry, such as a circular array
- An array with nonuniform geometry, such as a linear array with variable spacing
- A nonplanar array

When you use `phased.ConformalArray`, you must specify these aspects of the array:

- Sensor element of the array
- Element positions
- Direction normal to each array element

## Create Default Conformal Array

To create a conformal array with default properties, use this command:

```
array = phased.ConformalArray

array =
  phased.ConformalArray with properties:

             Element: [1x1 phased.IsotropicAntennaElement]
     ElementPosition: [3x1 double]
       ElementNormal: [2x1 double]
               Taper: 1
```

This default conformal array consists of a single `phased.IsotropicAntennaElement` antenna located at the origin of the local coordinate system. The direction normal to the sensor element is 0° azimuth and 0° elevation.

## Uniform Circular Array Created from Conformal Array

This example shows how to construct a 60-element uniform circular array. In constructing a uniform circular array, you can use either the `phased.UCA` or the `phased.ConformalArray` System objects. The conformal array approach is more general because it allows you to point the array elements in

arbitrary directions. A UCA restricts the array element directions to lie in the plane of the array. This example illustrates how you can use the `phased.ConformalArray` System object™ to create any other array shape. Assume an operating frequency of 400 MHz. Tune the array by specifying the arclength between the elements to be $0.5\lambda$ where $\lambda$ is the wavelength corresponding to the operating frequency. Array elements lie in the x-y-plane. Element normal directions are set to $(\phi_n, 0)$ where $\phi_n$ is the azimuth angle of the $n^{th}$ array element.

Set the number of elements and the operating frequency of the array.

```
N = 60;
fc = 400e6;
```

Compute the element spacing in radians.

```
theta = 360/N;
thetarad = deg2rad(theta);
```

Choose the radius so that the inter-element arclength is one-half wavelength.

```
arclength = 0.5*(physconst('LightSpeed')/fc);
radius = arclength/thetarad;
```

Compute the element azimuth angles. Azimuth angles must lie in the range $(-180°, 180°)$.

```
ang = (0:N-1)*theta;
ang(ang >= 180.0) = ang(ang >= 180.0) - 360.0;
array = phased.ConformalArray;
array.ElementPosition = [radius*cosd(ang);...
    radius*sind(ang);...
    zeros(1,N)];
array.ElementNormal = [ang;zeros(1,N)];
```

Show the UCA array geometry.

```
viewArray(array)
```

**Array Geometry**



Array Span:
X axis = 7.157 m
Y axis = 7.157 m
Z axis = 0.000 m
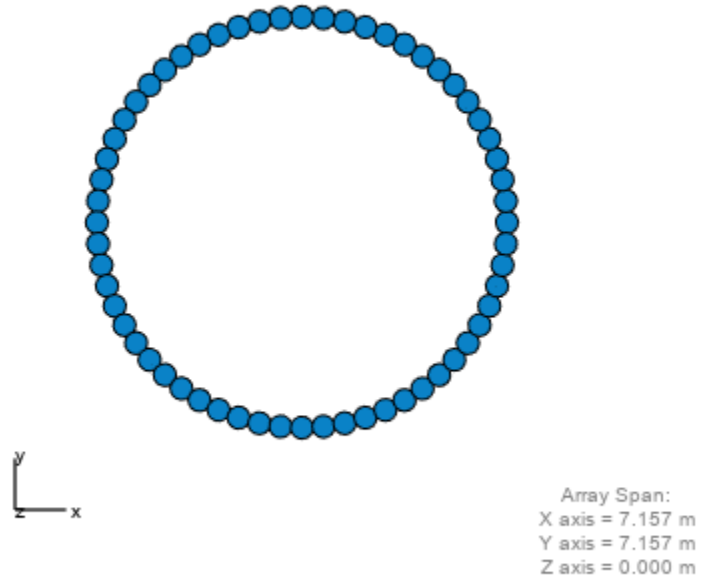
Plot the array response pattern at 1 GHz.

```
pattern(array,1e9,[-180:180],0,'PropagationSpeed',physconst('LightSpeed'),...
    'CoordinateSystem','polar','Type','powerdb','Normalize',true)
```

Azimuth Cut (elevation angle = 0.0°)

Normalized Power (dB), Broadside at 0.00 °
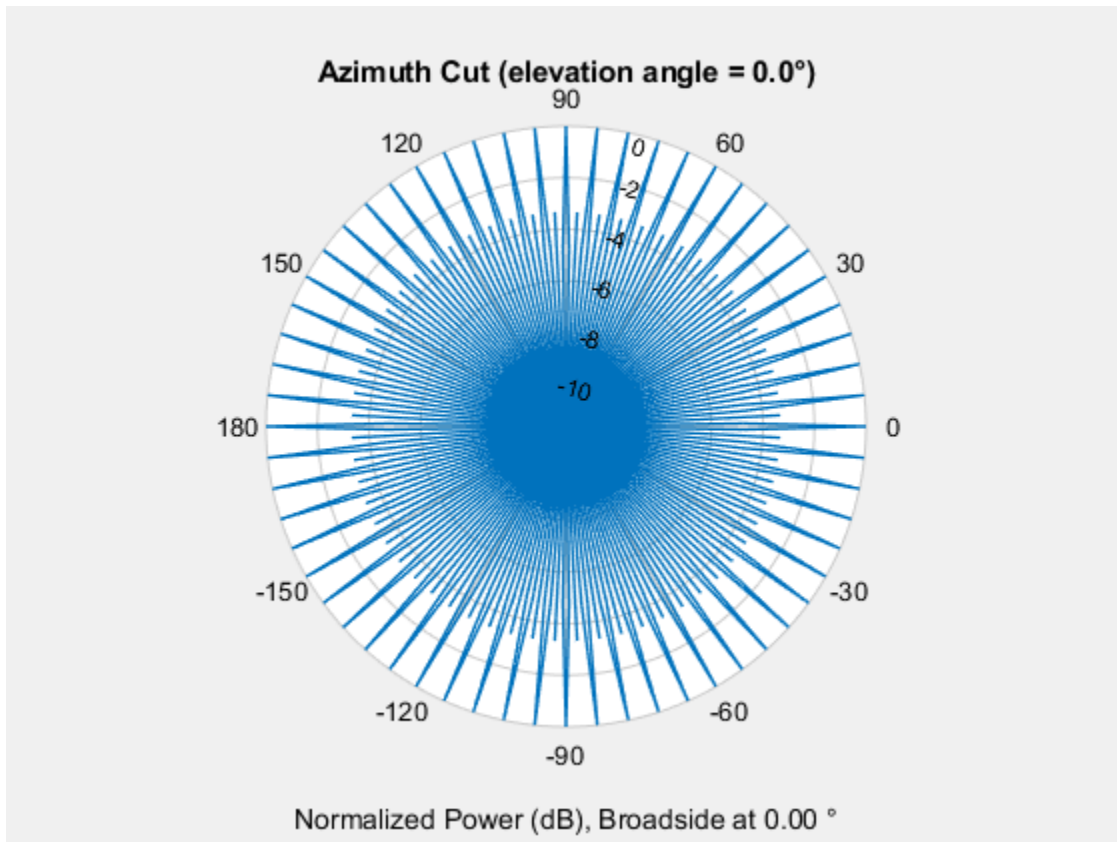
## Custom Antenna Array

This example shows how to construct and visualize a custom-geometry array containing antenna elements with a custom radiation pattern. The radiation pattern of each element is constant over each azimuth angle and has a cosine pattern for the elevation angles.

Define the custom antenna element and plot its radiation pattern.

```
az = -180:180;
el = -90:90;
fc = 3e8;
elresp = cosd(el);
antenna = phased.CustomAntennaElement('AzimuthAngles',az,...
    'ElevationAngles',el,...
    'MagnitudePattern',repmat(elresp',1,numel(az)));
pattern(antenna,3e8,0,el,'CoordinateSystem','polar','Type','powerdb',...
    'Normalize',true);
```

Elevation Cut (azimuth angle = 0.0°)

Normalized Power (dB), Broadside at 0.00 °
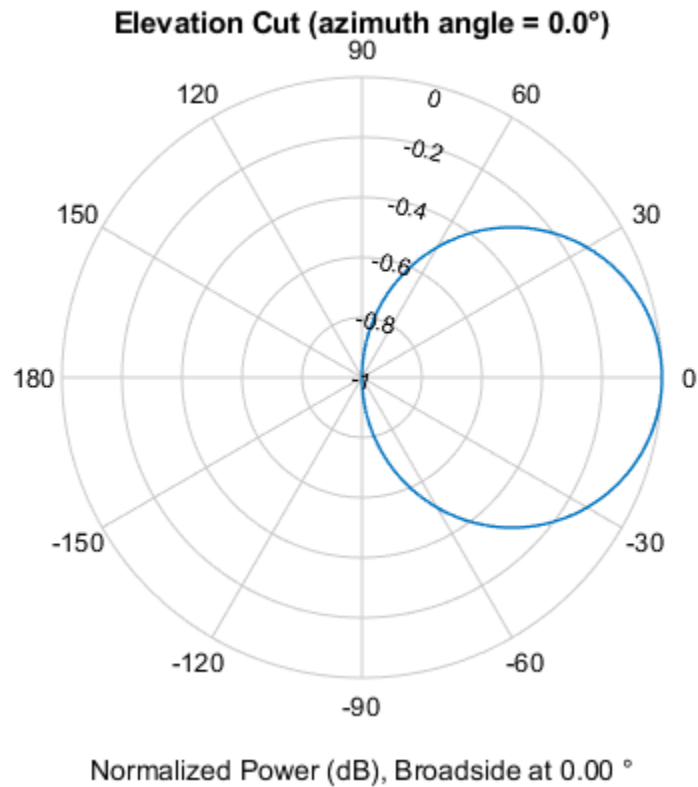
Define the locations and normal directions of the elements. All elements lie in the z-plane. The elements are located at *(1;0;0)* , *(0;1;0)*, and *(0;-1;0)* meters. The element normal azimuth angles are 0°, 120°, and -120°, respectively. All normal elevation angles are 0°.

```
xpos = [1 0 0];
ypos = [0 1 -1];
zpos = [0 0 0];
normal_az = [0 120 -120];
normal_el = [0 0 0];
```

Define a conformal array with those elements.

```
array = phased.ConformalArray('Element',antenna,...
    'ElementPosition',[xpos; ypos; zpos],...
    'ElementNormal',[normal_az; normal_el]);
```

Plot the positions and normal directions of the elements.

```
viewArray(array,'ShowNormals',true)
view(0,90)
```

Array Geometry

Array Span:
X axis = 1 m
Y axis = 2 m
Z axis = 0 m
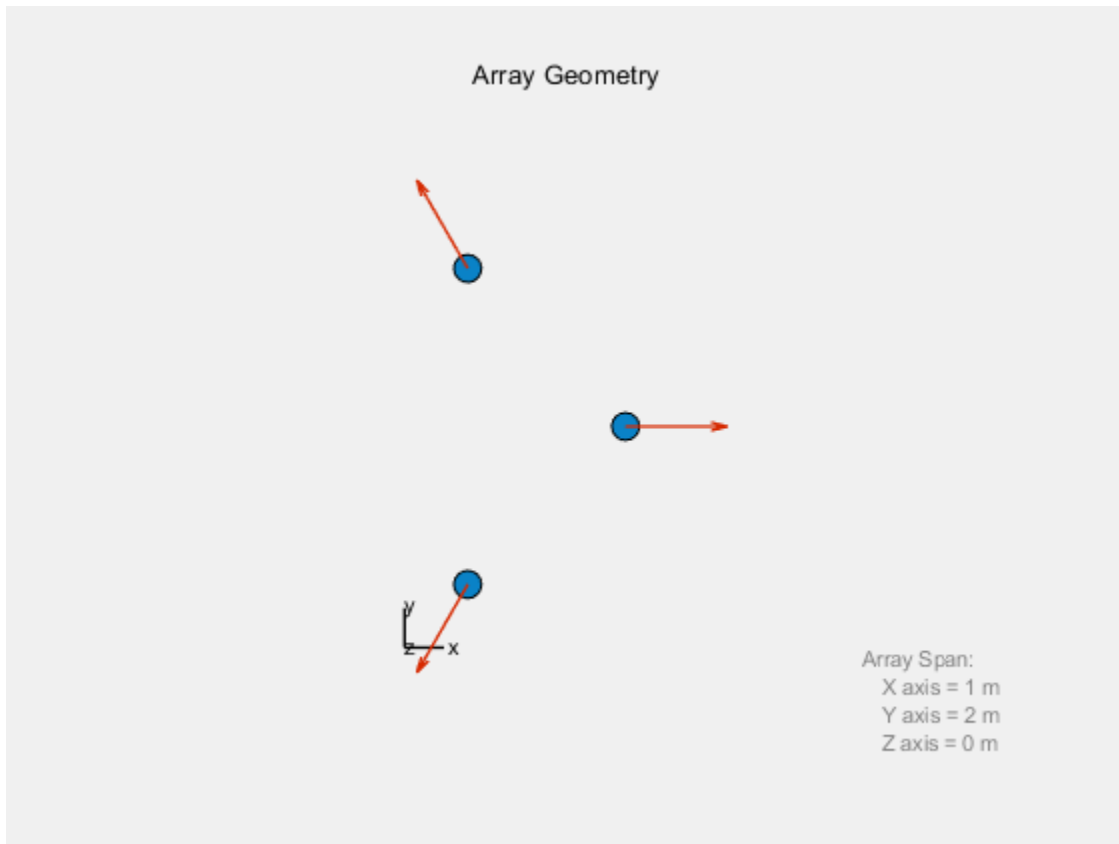
```
pattern(array,fc,az,el,'CoordinateSystem','polar','Type','powerdb',...
    'Normalize',true,'PropagationSpeed',physconst('LightSpeed'))
```
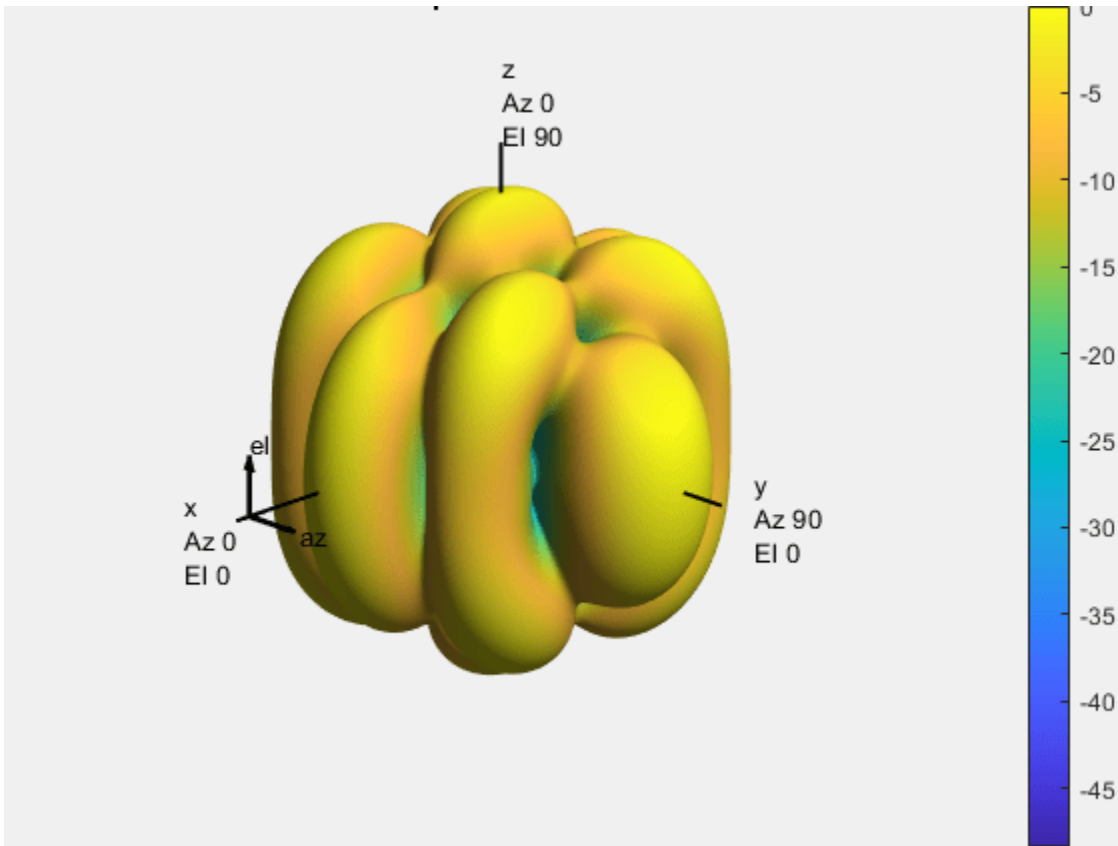
# Subarrays Within Arrays

## Definition of Subarrays

In Phased Array System Toolbox software, a subarray is an accessible subset of array elements. When you use an array that contains subarrays, you can access measurements from the subarrays but not from the individual elements. Similarly, you can perform processing at the subarray level but not at the level of the individual elements. As a result, the system has fewer degrees of freedom than if you controlled the system at the level of the individual elements.

## Benefits of Using Subarrays

Radar applications often use subarrays because operations, such as phase shifting and analog-to-digital conversion, are too expensive to implement for each element. It is less expensive to group the elements of an array through hardware, thus creating subarrays within the array. Grouping elements through hardware limits access to measurements and processing to the subarray level.

## Support for Subarrays Within Arrays

To work with subarrays, you must define the array and the subarrays within it. You can either define the array first or begin with the subarray. Choose one of these approaches:

- Define one subarray, and then build a larger array by arranging copies of the subarray. The subarray can be a ULA, URA, or conformal array. The copies are identical, except for their location and orientation. You can arrange the copies spatially in a grid or a custom layout.

  When you use this approach, you build the large array by creating a `phased.ReplicatedSubarray` System object. This object stores information about the subarray and how the copies of it are arranged to form the larger array.

- Define an array, and then partition it into subarrays. The array can be a ULA, URA, or conformal array. The subarrays do not need to be identical. A given array element can be in more than one subarray, leading to overlapped subarrays.

  When you use this approach, you partition your array by creating a `phased.PartitionedArray` System object. This object stores information about the array and its subarray structure.

After you create a `phased.ReplicatedSubarray` or `phased.PartitionedArray` object, you can use it to perform beamforming, steering, or other operations. To do so, specify your object as the value of the `SensorArray` or `Sensor` property in objects that have such a property and that support subarrays. Objects that support subarrays in their `SensorArray` or `Sensor` property include:

- phased.AngleDopplerResponse
- phased.ArrayGain
- phased.ArrayResponse
- phased.Collector
- phased.ConstantGammaClutter
- phased.MVDRBeamformer
- phased.PhaseShiftBeamformer
- phased.Radiator
- phased.STAPSMIBeamformer
- phased.SteeringVector
- phased.SubbandPhaseShiftBeamformer
- phased.WidebandCollector

## Rectangular Array Partitioned into Linear Subarrays

This example shows how to set up a rectangular array containing linear subarrays. The example also finds the phase centers of the subarrays.

Create a 2-by-3 rectangular array.

```
array = phased.URA('Size',[2 3]);
```

Plot the positions of the array elements in the *yz*-plane (all *x*-coordinates are zero.) Include labels that indicate the numbering of the elements. The numbering is important for selecting which elements are included in each subarray.

```
viewArray(array,'ShowIndex','All')
```

**Array Geometry**
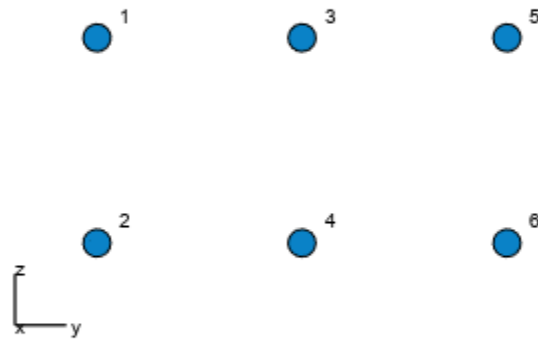


Aperture Size:
Y axis = 1.5 m
Z axis = 1.0 m
Element Spacing:
$\Delta$ y = 500 mm
$\Delta$ z = 500 mm

Create and view an array consisting of three 2-element linear subarrays each parallel to the *z*-axis. Use the indices from the plot to form the matrix for the `SubarraySelection` property. The `getSubarrayPosition` method returns the phase centers of the three subarrays.

```
subarray1 = [1 1 0 0 0 0; 0 0 1 1 0 0; 0 0 0 0 1 1];
partitionedarray1 = phased.PartitionedArray('Array',array,...
    'SubarraySelection',subarray1);
viewArray(partitionedarray1)
```

**Array Geometry**
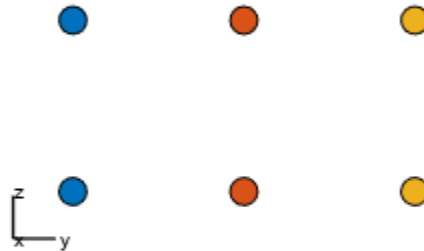


Array Span:
  X axis =    0 mm
  Y axis = 1000 mm
  Z axis =  500 mm

```
subarray1pos = getSubarrayPosition(partitionedarray1)
```

subarray1pos = *3×3*

```
         0            0            0
   -0.5000            0       0.5000
         0            0            0
```

Create and view another array consisting of two 3-element linear subarrays parallel to the *y*-axis. Using the `getSubarrayPosition` method, find the phase centers of the two subarrays.

```
subarray2 = [0 1 0 1 0 1; 1 0 1 0 1 0];
partitionedarray2 = phased.PartitionedArray('Array',array,...
    'SubarraySelection',subarray2);
viewArray(partitionedarray2)
```

**Array Geometry**
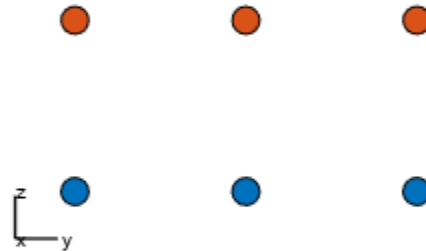


```
Array Span:
  X axis =    0 mm
  Y axis = 1000 mm
  Z axis =  500 mm
```

```
subarraypos2 = getSubarrayPosition(partitionedarray2)

subarraypos2 = 3×2

         0         0
         0         0
   -0.2500    0.2500
```

## Linear Subarray Replicated to Form Rectangular Array

This example shows how to arrange copies of a linear subarray to form a rectangular array.

Create a 4-element linear array parallel to the *y*-axis.

```
array = phased.ULA('NumElements',4);
```

Create a rectangular array by arranging two copies of the linear array.

```
replsubarray = phased.ReplicatedSubarray('Subarray',array,'GridSize',[2 1]);
```

Plot the positions of the array elements (filled circles) and the phase centers of the subarrays (unfilled circles). The elements lie in the *yz*-plane because all the *x*-coordinates are zero.

```
viewArray(replsubarray);
hold on;
```

```
subarraypos = getSubarrayPosition(replsubarray);
sx = subarraypos(1,:);
sy = subarraypos(2,:);
sz = subarraypos(3,:);
scatter3(sx,sy,sz,'ro')
hold off
```

**Array Geometry**
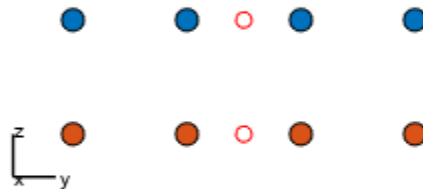


```
Array Span:
  X axis =    0 mm
  Y axis = 1500 mm
  Z axis =  500 mm
```

## Linear Subarray Replicated in a Custom Grid

This example shows how to arrange copies of a linear subarray in a triangular layout.

Create a 4-element linear array.

```
antenna = phased.CosineAntennaElement('CosinePower',1);
array = phased.ULA('NumElements',4,'Element',antenna);
```

Create a larger array by arranging three copies of the linear array. Define the phase centers and normal directions of the three copies explicitly.

```
vertex_ang = [60 180 -60];
vertex = 2*[cosd(vertex_ang); sind(vertex_ang); zeros(1,3)];
subarray_pos = 1/2*[...
   (vertex(:,1)+vertex(:,2)) ...
   (vertex(:,2)+vertex(:,3)) ...
   (vertex(:,3)+vertex(:,1))];
```

**2-27**

```
replsubarray = phased.ReplicatedSubarray('Subarray',array,...
    'Layout','Custom',...
    'SubarrayPosition',subarray_pos,...
    'SubarrayNormal',[120 0;-120 0;0 0].');
```

Plot the positions of the array elements (blue) and the phase centers (red) of the subarrays. The plot is in the xy-plane because all the z-coordinates are zero.

```
viewArray(replsubarray,'ShowSubarray',[])
hold on
scatter3(subarray_pos(1,:),subarray_pos(2,:),...
    subarray_pos(3,:),'ro')
hold off
```

**Array Geometry**



Array Span:
X axis = 2.150 m
Y axis = 2.482 m
Z axis = 0.000 m
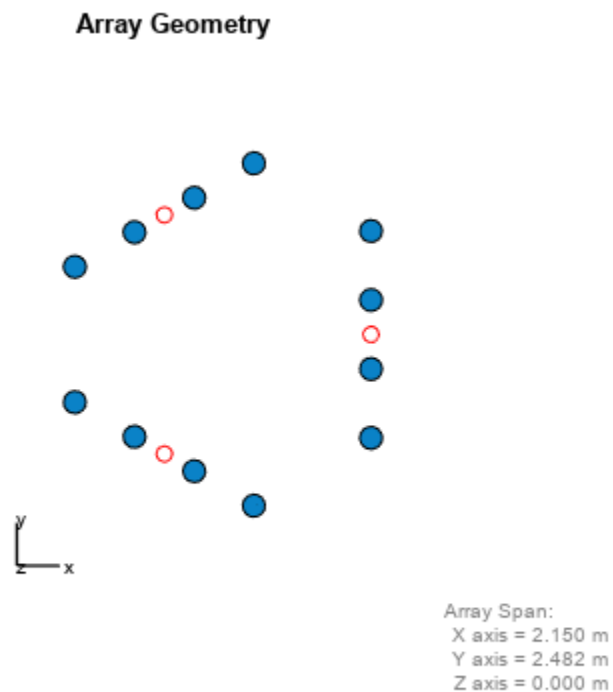
## See Also

## Related Examples
- "Subarrays in Phased Array Antennas" on page 18-124

# Plot Array Directivity Using Sensor Array Analyzer App

The `sensorArrayAnalyzer` app lets you examine important properties of a phased array such as its shape and directivity.

**Open sensorArrayAnalyzer App**

When you type `sensorArrayAnalyzer` from the command line or select the app from the **App Toolstrip**, an interactive window opens. The default window shows the geometry of a 4-element uniform linear array. You can then select various options to analyze different arrays, other element types, geometry, and directivity.



**Create 3D Directivity Plot of 4-by-4 URA**

As an example, use the app to create a 4-by-4 uniform rectangular array of cosine antenna elements and then show the array directivity. Space the elements 0.4 wavelengths apart.

1   Set the **Array** to URA.

2   Set the **Element** to `Cosine`.

3   Set **Cosine Power** to [2 2].

**4**    Set the **Size** of the array to [4 4].

**5**    Set the **Element Spacing** to [0.4 0.4]. Set the element spacing units to wavelength.

**6**    Set **Propagation Speed** to physconst('Lightspeed').

**7**    From the **Steering** pull down menu, set the **Steer Angle** to [-30,0] to steer the array -30 degrees in azimuth.

**8**    In the Plot menu, select 3D Pattern.

Then, you will see a plot of array directivity similar to this.
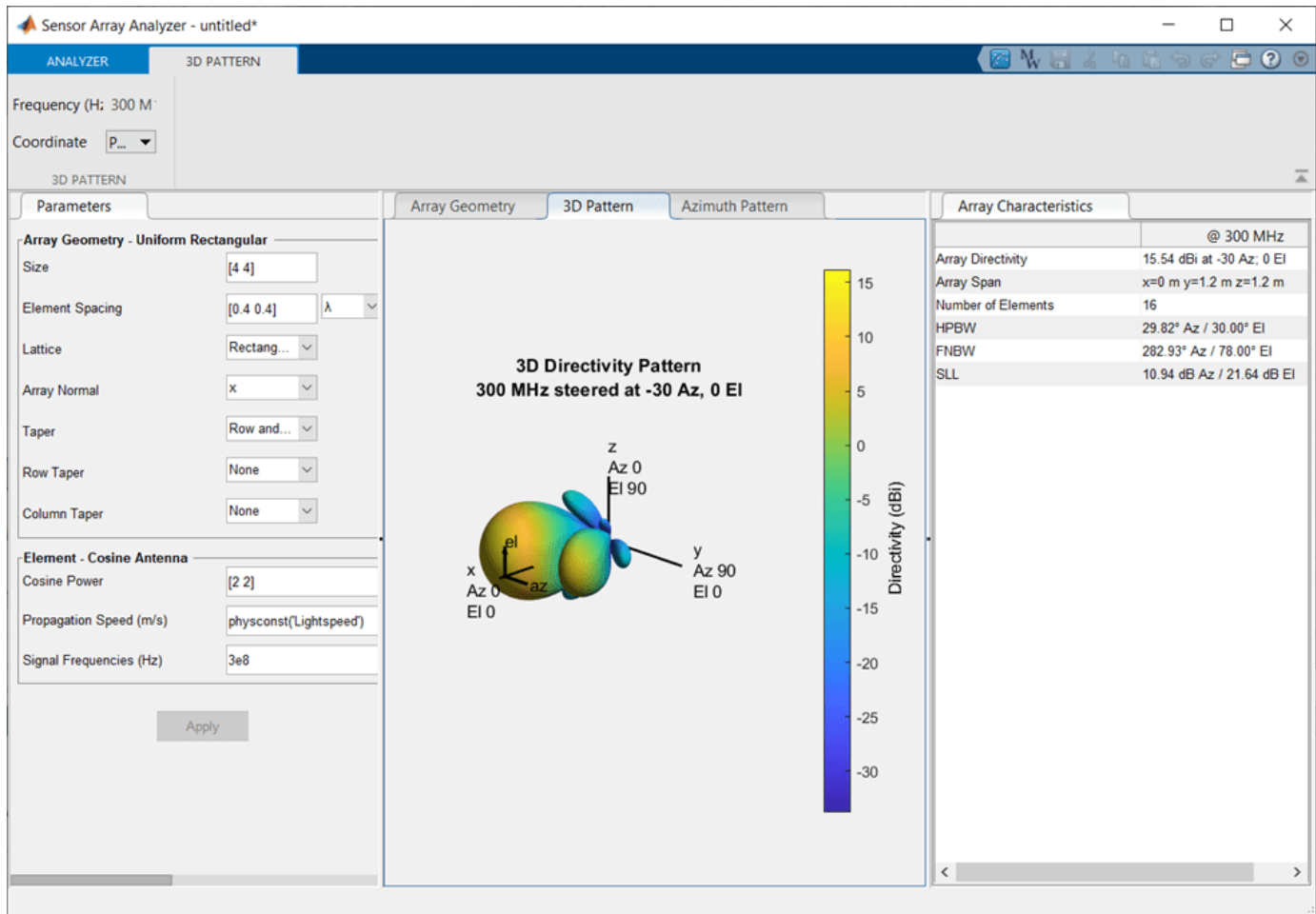
# Signal Radiation and Collection

# Signal Radiation

| In this section... |
|---|
| "Support for Modeling Signal Radiation" on page 3-2 |
| "Radiate Signal with Uniform Linear Array" on page 3-2 |

## Support for Modeling Signal Radiation

You can use the `phased.Radiator` and `phased.Collector` objects to model narrowband signal radiation and collection with an array. The array can be a single microphone or antenna element, or an array of sensor elements.

To radiate a signal from a sensor array, use `phased.Radiator`. When you use this object, you must specify these aspects of the radiator:

- Whether the output of all sensor elements is combined
- Operating frequency of the array
- Propagation speed of the wave
- Sensor (single element) or sensor array
- Whether or not to apply weights to signals radiated by different elements in the array. You apply weights, when you execute the System object.

## Radiate Signal with Uniform Linear Array

Construct a radiator using a two-element ULA with elements spaced 0.5 meters apart (the default ULA). The operating frequency is 300 MHz, the propagation speed is the speed of light, and the element outputs are combined to simulate the far field radiation pattern.

```
array = phased.ULA('NumElements',2,'ElementSpacing',0.5);
radiator = phased.Radiator('Sensor',array,...
    'OperatingFrequency',300e6,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'CombineRadiatedSignals',true);
```

Create a signal to radiate and propagate to the far field at an angle of *(45°,0°)* .

```
x = [1 -1 1 -1]';
y = radiator(x,[45;0]);
```

The far field signal results from multiplying the signal by the *array pattern*. The array pattern is the product of the *array element pattern* and the *array factor*. For a uniform linear array, the array factor is the superposition of elements in the steering vector `phased.SteeringVector`.

The following code produces an identical far-field signal by explicitly using the array factor.

```
array = phased.ULA('NumElements',2,'ElementSpacing',0.5);
steervec = phased.SteeringVector('SensorArray',array,...
    'IncludeElementResponse',true);
sv = steervec(300e6,[45;0]);
y1 = x*sum(sv);
```

Compare y1 to y.

```
disp(y1-y)
     0
     0
     0
     0
```

# Signal Collection

| **In this section...** |
| --- |

## Support for Modeling Signal Collection

To model the collection of a signal with a sensor element or sensor array, you can use the `phased.Collector` or `phased.WidebandCollector`. Both collector objects assume that incident signals have propagated to the location of the array elements, but have not been received by the array. In other words, the collector objects do not model the actual reception by the array. See "Receiver Preamp" on page 4-30 for signal effects related to the gain and internal noise of the array's receiver.

In many array processing applications, the ratio of the signal's bandwidth to the carrier frequency is small. Expressed as a percentage, this ratio does not exceed a few percent. Examples include radar applications where a pulse waveform is modulated by a carrier frequency in the microwave range. These are *narrowband* signals. For narrowband signals, you can express the steering vector as a function of a single frequency, the carrier frequency. For narrowband signals, the `phased.Collector` object is appropriate.

In other applications, the narrowband assumption is not justified. In many acoustic and sonar applications, the wave impinging on the array is a pressure wave that is unmodulated. It is not possible to express the steering vector as a function of a single frequency. In these cases, the subband approach implemented in `phased.WidebandCollector` is appropriate. The wideband collector decomposes the input into subbands and computes the steering vector for each subband.

When you use the narrowband collector, `phased.Collector`, you must specify these aspects of the collector:

- Operating frequency of the array
- Propagation speed of the wave
- Sensor (single element) or sensor array
- Type of incoming wave. Choices are `'Plane'` and `'Unspecified'`. If you select `'Plane'`, the input signals are multiple plane waves impinging on the entire array. Each plane wave is received by all collecting elements. If you select `'Unspecified'`, the input signal are individual waves impinging on individual sensors.
- Whether to apply weights to signals collected by different elements in the array. If you want to apply weights, you specify them when you execute the System object.

When you use the wideband collector, `phased.WidebandCollector`, you must specify these aspects of the collector:

- Carrier frequency
- Whether the signal is demodulated to the baseband

- Operating frequency of the array
- Propagation speed of the wave
- Sampling rate
- Sensor (single element) or sensor array
- Type of incoming wave. Choices are `'Plane'` and `'Unspecified'`. If you select `'Plane'`, the input signals are multiple plane waves impinging on the entire array. Each plane wave is received by all collecting elements. If you select `'Unspecified'`, the input signal are individual waves impinging on individual sensors.
- Whether to apply weights to signals collected by different elements in the array. If you want to apply weights, you specify them when you execute the System object.

## Narrowband Collector for Uniform Linear Array

This example shows how to construct a narrowband collector that models a plane wave impinging on a two-element uniform linear array. The array has an element spacing of 0.5 m (default for a ULA). The operating frequency of the array is 300 MHz.

Create the array and collector System objects.

```
array = phased.ULA('NumElements',2,'ElementSpacing',0.5);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',3e8,'Wavefront','Plane');
```

Create the signal and simulate reception from an angle of *(45;0)*.

```
x =[1 -1 1 -1]';
y = collector(x,[45;0])
```

*y = 4×2 complex*

```
   0.4433 - 0.8964i   0.4433 + 0.8964i
  -0.4433 + 0.8964i  -0.4433 - 0.8964i
   0.4433 - 0.8964i   0.4433 + 0.8964i
  -0.4433 + 0.8964i  -0.4433 - 0.8964i
```

In the preceding case, the collector object multiplies the input signal, `x`, by the corresponding element of the steering vector for the two-element ULA. The following code produces the response in an equivalent manner. First, create the ULA and then create the steering vector. Compare with the previous result.

```
array = phased.ULA('NumElements',2,'ElementSpacing',0.5);
steeringvec = phased.SteeringVector('SensorArray',array);
sv = steeringvec(3e8,[45;0]);
x =[1 -1 1 -1]';
y1 = x*sv.'
```

*y1 = 4×2 complex*

```
   0.4433 - 0.8964i   0.4433 + 0.8964i
  -0.4433 + 0.8964i  -0.4433 - 0.8964i
   0.4433 - 0.8964i   0.4433 + 0.8964i
```

```
  -0.4433 + 0.8964i  -0.4433 - 0.8964i
```

## Narrowband Collector for a Single Antenna Element

The `Sensor` property of a `phased.Collector` System object™ can specify a single antenna element. In this example, create a custom antenna element using the `phased.CustomAntennaElement` System object. The antenna element has a cosine response over elevation angles from *(-90°,90°)*. Plot the polar pattern response of the antenna at 1 GHz on an elevation cut at 0° azimuth. Display the antenna voltage response at 0° azimuth and 45° elevation.

```
fc = 1e9;
antenna = phased.CustomAntennaElement;
antenna.AzimuthAngles = -180:180;
antenna.ElevationAngles = -90:90;
antenna.MagnitudePattern = mag2db(...
    repmat(cosd(antenna.ElevationAngles)',1,numel(antenna.AzimuthAngles)));
resp = antenna(fc,[0;45])
```

```
resp = 0.7071
```

```
pattern(antenna,fc,0,[-90:90],'Type','powerdb')
```



Elevation Cut (azimuth angle = 0.0°)

Normalized Power (dB), Broadside at 0.00 °
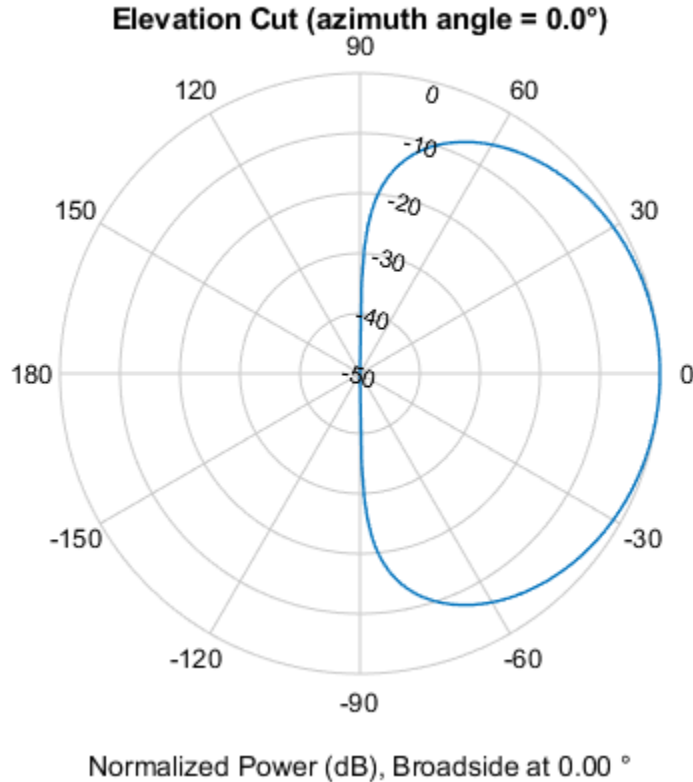
The antenna voltage response at 0° azimuth and 45° elevation is *cos(45°)* as expected.

Assume a narrowband sinusoidal input incident on the antenna element from 0° azimuth and 45° elevation. Determine the signal collected at the element.

```
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',fc);
x =[1 -1 1 -1]';
y = collector(x,[0;45])
```

*y = 4×1*

```
    0.7071
   -0.7071
    0.7071
   -0.7071
```

## Wideband Signal Collection

This example shows how to simulate the reception of a wideband acoustic signal by a single omnidirectional microphone element.

```
x = randn(10,1);
c = 340.0;
microphone = phased.OmnidirectionalMicrophoneElement(...
    'FrequencyRange',[20 20e3],'BackBaffled',true);
collector = phased.WidebandCollector('Sensor',microphone,...
    'PropagationSpeed',c,'SampleRate',50e3,...
    'ModulatedInput',false);
y = collector(x,[30;10]);
```

**4**

# Waveforms, Transmitter, and Receiver

# Rectangular Pulse Waveforms

## Definition of Rectangular Pulse Waveform

Define the following function of time:

$$a(t) = \begin{cases} 1 & 0 \le t \le \tau \\ 0 & \text{otherwise} \end{cases}$$

Assume that a radar transmits a signal of the form:

$$x(t) = a(t)\sin(\omega_c t)$$

where $\omega_c$ denotes the carrier frequency. Note that $a(t)$ represents an on-off rectangular amplitude modulation of the carrier frequency. After demodulation, the complex envelope of $x(t)$ is the real-valued rectangular pulse $a(t)$ of duration $\tau$ seconds.

## How to Create Rectangular Pulse Waveforms

To create a rectangular pulse waveform, use `phased.RectangularWaveform`. You can customize certain characteristics of the waveform, including:

- Sampling rate
- Pulse duration
- Pulse repetition frequency
- Number of samples or pulses in each vector that represents the waveform

## Rectangular Waveform Plot

This example shows how to create a rectangular pulse waveform variable using `phased.RectangularWaveform`. The example also plots the pulse and finds the bandwidth of the pulse.

Construct a rectangular pulse waveform with a duration of 50 μs, a sample rate of 1 MHz, and a pulse repetition frequency (PRF) of 10 kHz.

```
waveform = phased.RectangularWaveform('SampleRate',1e6,...
    'PulseWidth',50e-6,'PRF',10e3);
```

Plot a single rectangular pulse by calling `plot` directly on the rectangular waveform variable. `plot` is a method of `phased.RectangularWaveform`. This method produces an annotated graph of your pulse waveform.

```
plot(waveform)
```



**Rectangular pulse waveform: real part, pulse 1**
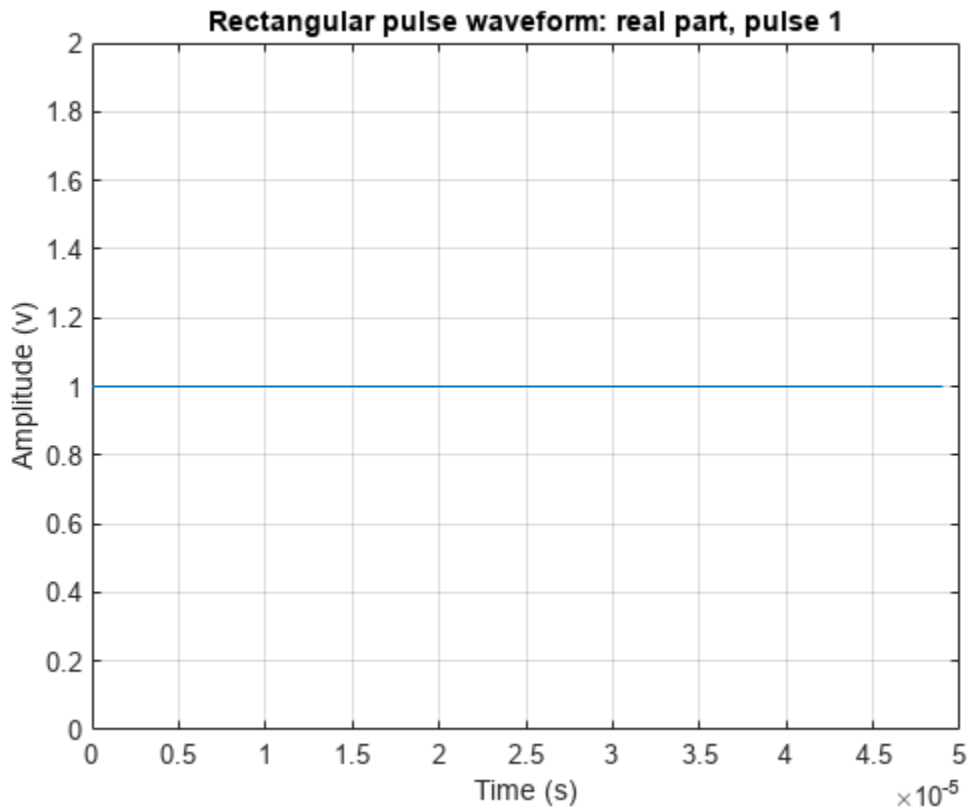
Find the bandwidth of the rectangular pulse.

```
bw = bandwidth(waveform)
```

```
bw = 20000
```

The bandwidth, `bw`, of a rectangular pulse in hertz is approximately the reciprocal of the pulse duration `1/sRect.PulseWidth`.

## Pulses of Rectangular Waveform

This example shows how to create rectangular pulse waveform signals having different durations. The example plots two pulses of each waveform.

Create a rectangular pulse with a duration of 100 µs and a PRF of 1 kHz. Set the number of pulses in the output equal to two.

```
waveform = phased.RectangularWaveform('PulseWidth',100e-6,...
    'PRF',1e3,'OutputFormat','Pulses','NumPulses',2);
```

Make a copy of your rectangular pulse and change the pulse width in your original waveform to 10 µs.

```
waveform2 = clone(waveform);
waveform.PulseWidth = 10e-6;
```

sRect and sRect1 now specify different rectangular pulses because you changed the pulse width of waveform.

Execute the System objects to return two pulses of your rectangular pulse waveforms.

```
y = waveform();
y2 = waveform2();
```

Plot the real part of the waveforms.

```
totaldur = 2*1/waveform.PRF;
totnumsamp = totaldur*waveform.SampleRate;
t = unigrid(0,1/waveform.SampleRate,totaldur,'[)');
subplot(2,1,1)
plot(t.*1000,real(y))
axis([0 totaldur*1e3 0 1.5])
title('Two 10-\musec duration pulses (PRF = 1 kHz)')
set(gca,'XTick',0:0.2:totaldur*1e3)
subplot(2,1,2)
plot(t.*1000,real(y2))
axis([0 totaldur*1e3 0 1.5])
xlabel('Milliseconds')
title('Two 100-\musec duration pulses (PRF = 1 kHz)')
set(gca,'XTick',0:0.2:totaldur*1e3)
```
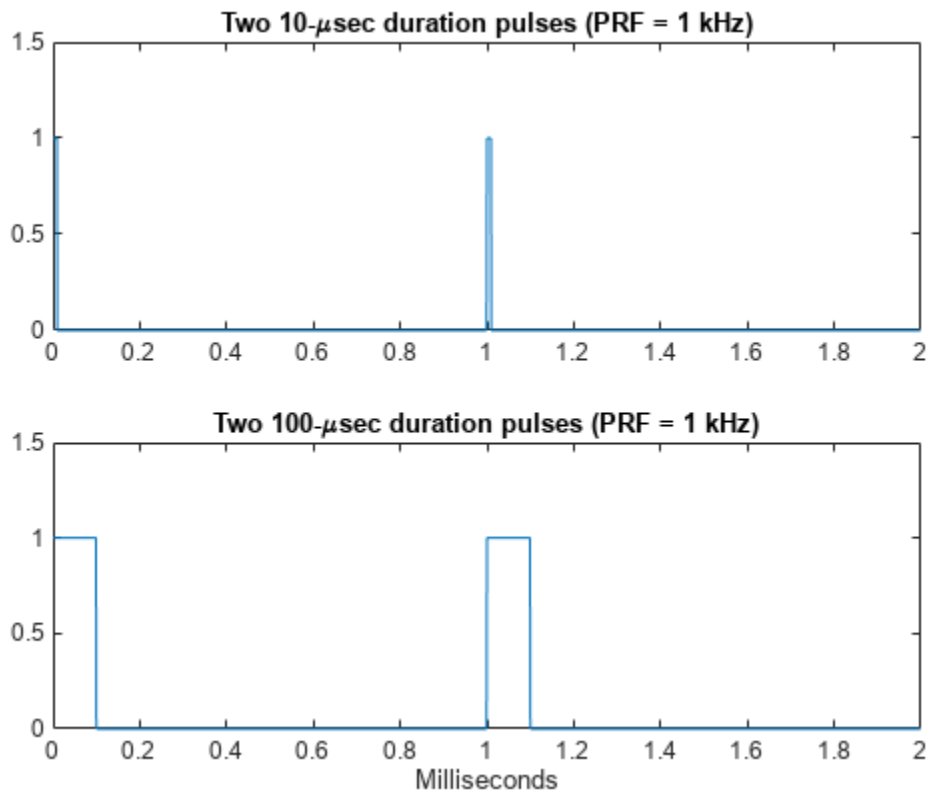
# Linear Frequency Modulated Pulse Waveforms

## Benefits of Using Linear FM Pulse Waveform

Increasing the duration of a transmitted pulse increases its energy and improves target detection capability. Conversely, reducing the duration of a pulse improves the range resolution of the radar.

For a rectangular pulse, the duration of the transmitted pulse and the processed echo are effectively the same. Therefore, the range resolution of the radar and the target detection capability are coupled in an inverse relationship.

Pulse compression techniques enable you to decouple the duration of the pulse from its energy by effectively creating different durations for the transmitted pulse and processed echo. Using a linear frequency modulated pulse waveform is a popular choice for pulse compression.

## Definition of Linear FM Pulse Waveform

The complex envelope of a linear FM pulse waveform with increasing instantaneous frequency is:

$$\tilde{x}(t) = a(t)e^{j\pi(\beta/\tau)t^2}$$

where $\beta$ is the bandwidth and $\tau$ is the pulse duration.

If you denote the phase by $\Theta(t)$, the instantaneous frequency is:

$$\frac{1}{2\pi}\frac{d\Theta(t)}{dt} = \frac{\beta}{\tau}t$$

which is a linear function of $t$ with slope equal to $\beta/\tau$.

The complex envelope of a linear FM pulse waveform with decreasing instantaneous frequency is:

$$\tilde{x}(t) = a(t)e^{-j\pi\beta/\tau(t^2 - 2\tau t)}$$

Pulse compression waveforms have a time-bandwidth product, $\beta\tau$, greater than 1.

## How to Create Linear FM Pulse Waveforms

To create a linear FM pulse waveform, use `phased.LinearFMWaveform`. You can customize certain characteristics of the waveform, including:

- Sample rate
- Duration of a single pulse
- Pulse repetition frequency
- Sweep bandwidth
- Sweep direction (up or down), corresponding to increasing and decreasing instantaneous frequency
- Envelope, which describes the amplitude modulation of the pulse waveform. The envelope can be rectangular or Gaussian.

  - The rectangular envelope is as follows, where $\tau$ is the pulse duration.

  $$a(t) = \begin{cases} 1 & 0 \le t \le \tau \\ 0 & \text{otherwise} \end{cases}$$

  - The Gaussian envelope is:

  $$a(t) = e^{-t^2/\tau^2} \quad t \ge 0$$

- Number of samples or pulses in each vector that represents the waveform

## Create Linear FM Pulse Waveform

This example shows how to create a linear FM pulse waveform using `phased.LinearFMWaveform`. The example illustrates how to specify property settings.

Create a linear FM pulse with a sample rate of 1 MHz, a pulse duration of 50 μs with an increasing instantaneous frequency, and a sweep bandwidth of 100 kHz. The pulse repetition frequency is 10 kHz and the amplitude modulation is rectangular.

```
waveform = phased.LinearFMWaveform('SampleRate',1e6,...
    'PulseWidth',50e-6,'PRF',10e3,...
    'SweepBandwidth',100e3,'SweepDirection','Up',...
    'Envelope','Rectangular',...
    'OutputFormat','Pulses','NumPulses',1);
```

## Linear FM Pulse Waveform Plot

This example shows how to plot a linear FM *(LFM)* pulse waveform. The LFM waveform has a duration of 100 microseconds, a bandwidth of 200 kHz, and a PRF of 4 kHz. Use the default values for the other properties. Compute the time-bandwidth product. Plot the real part of the waveform and plot one full pulse repetition interval.

```
waveform = phased.LinearFMWaveform('PulseWidth',100e-6,...
    'SweepBandwidth',200e3,'PRF',4e3);
```

Display the time-bandwidth product of the FM sweep.

```
disp(waveform.PulseWidth*waveform.SweepBandwidth)
```

```
    20
```

Plot the real part of the waveform.

```
plot(waveform)
```
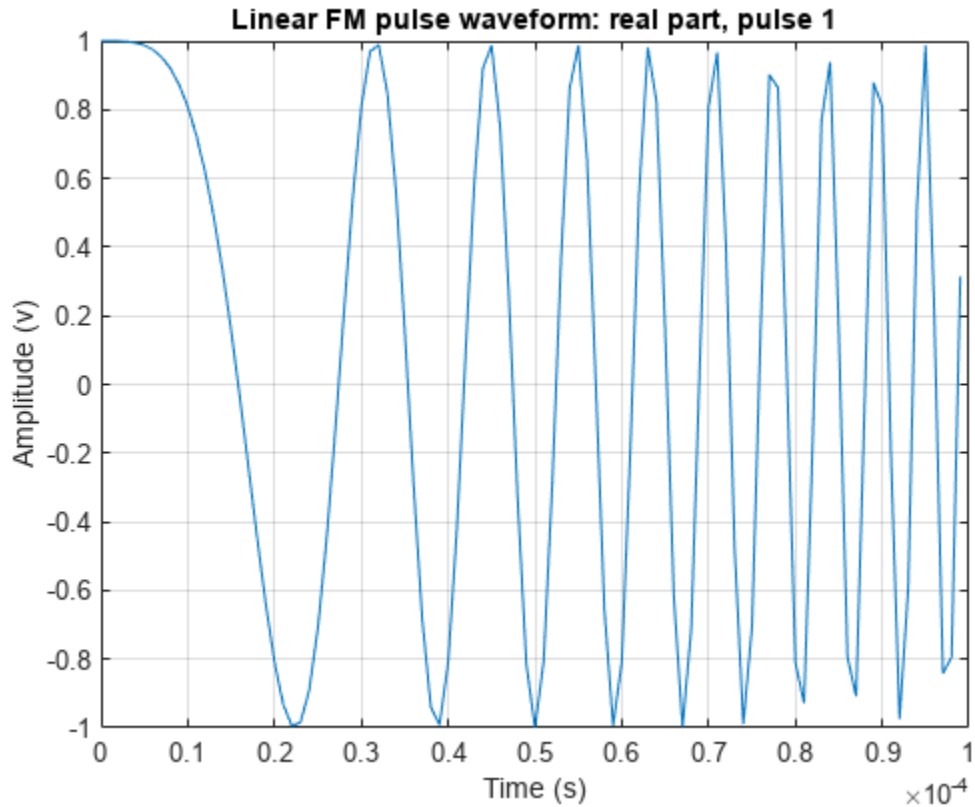


Use the `step` method to obtain one full repetition interval of the signal. Plot the real and imaginary parts.

```
y = waveform();
t = unigrid(0,1/waveform.SampleRate,1/waveform.PRF,'[)');
figure
subplot(2,1,1)
plot(t,real(y))
axis tight
title('Real Part')
subplot(2,1,2)
plot(t,imag(y))
xlabel('Time (s)')
title('Imaginary Part')
axis tight
```
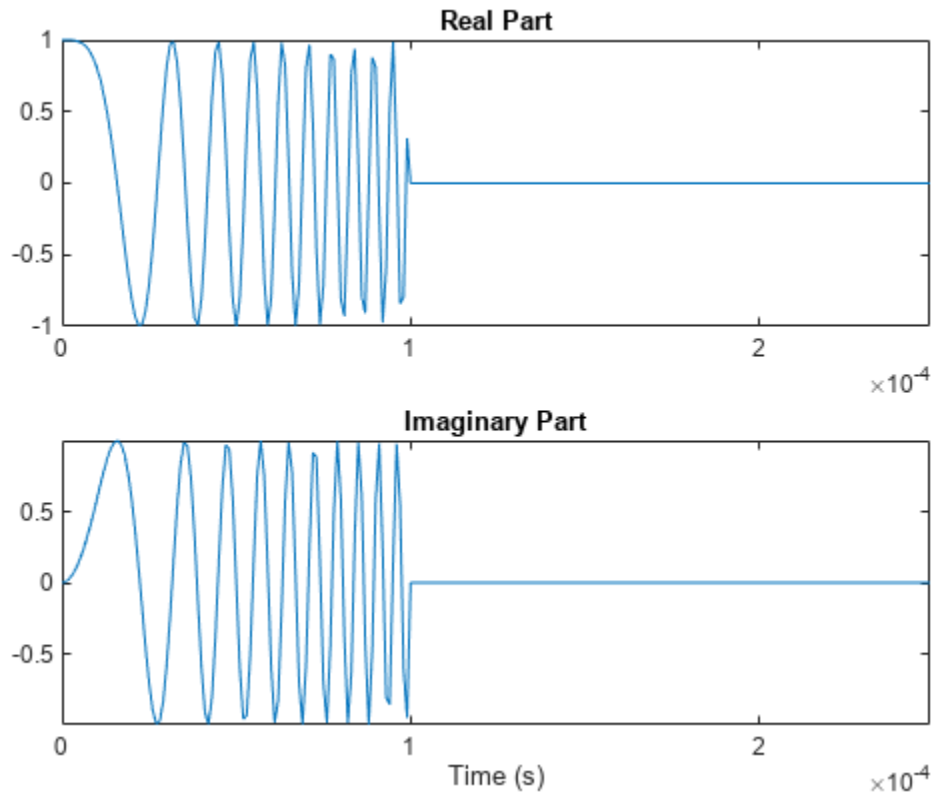
## Ambiguity Function of Linear FM Waveform

This example shows how to plot the ambiguity function of a linear FM pulse waveform.

Define and set up the linear FM waveform.

```
waveform = phased.LinearFMWaveform('PulseWidth',100e-6,...
    'SweepBandwidth',2e5,'PRF',1e3);
```

Generate samples of the waveform.

```
wav = waveform();
```

Create a 3-D surface plot of the ambiguity function for the waveform.

```
[afmag_lfm,delay_lfm,doppler_lfm] = ambgfun(wav,...
    waveform.SampleRate,waveform.PRF);
surf(delay_lfm*1e6,doppler_lfm/1e3,afmag_lfm,...
    'LineStyle','none')
axis tight
grid on
view([140,35])
colorbar
xlabel('Delay \tau (\mus)')
```

```
ylabel('Doppler f_d (kHz)')
title('Linear FM Pulse Waveform Ambiguity Function')
```
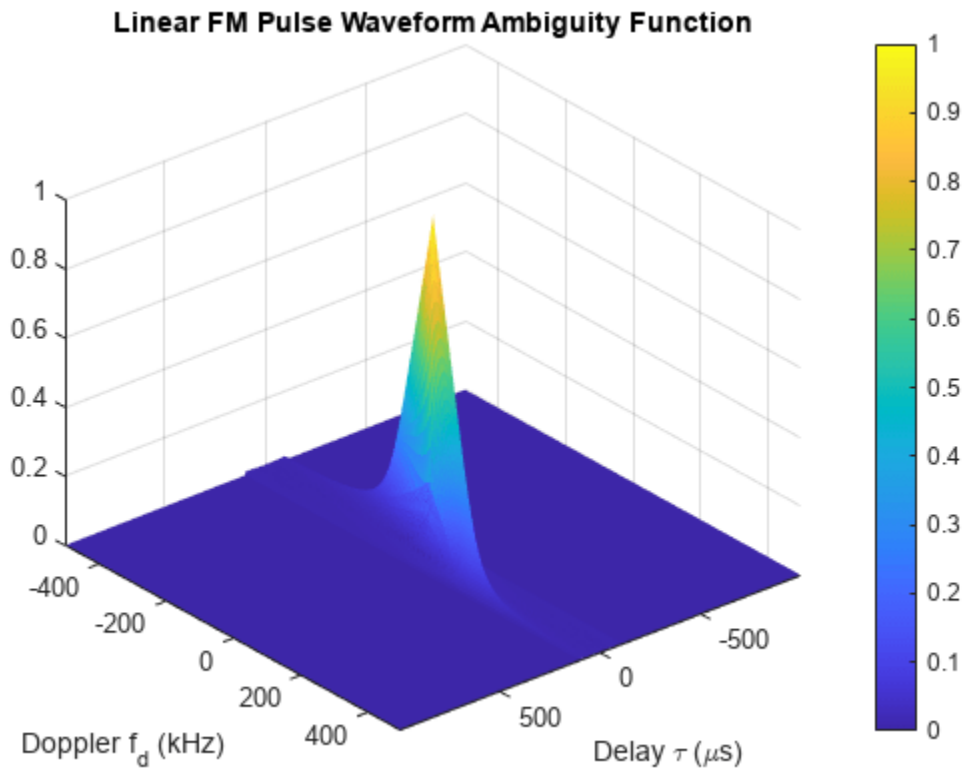


The surface has a narrow ridge that is slightly tilted. The tilt indicates better resolution in the zero delay cut.

## Compare Autocorrelation for Rectangular and Linear FM Waveforms

This example shows how to compute and plot the ambiguity function magnitudes for a rectangular and linear FM pulse waveform. The zero Doppler cut (magnitudes of the autocorrelation sequences) illustrates pulse compression in the linear FM pulse waveform.

Create a rectangular waveform and a linear FM pulse waveform having the same duration and PRF. Generate samples of each waveform.

```
rectwaveform = phased.RectangularWaveform('PRF',20e3);
lfmwaveform = phased.LinearFMWaveform('PRF',20e3);
xrect = rectwaveform();
xlfm = lfmwaveform();
```

Compute the ambiguity function magnitudes for each waveform.

```
[ambrect,delay] = ambgfun(xrect,rectwaveform.SampleRate,rectwaveform.PRF,...
    'Cut','Doppler');
```

```
ambfm = ambgfun(xlfm,lfmwaveform.SampleRate,lfmwaveform.PRF,...
    'Cut','Doppler');
```

Plot the ambiguity function magnitudes.

```
subplot(211)
stem(delay,ambrect)
title('Autocorrelation of Rectangular Pulse')
axis([-5e-5 5e-5 0 1])
set(gca,'XTick',1e-5*(-5:5))
subplot(212)
stem(delay,ambfm)
xlabel('Delay (seconds)')
title('Autocorrelation of Linear FM Pulse')
axis([-5e-5 5e-5 0 1])
set(gca,'XTick',1e-5*(-5:5))
```
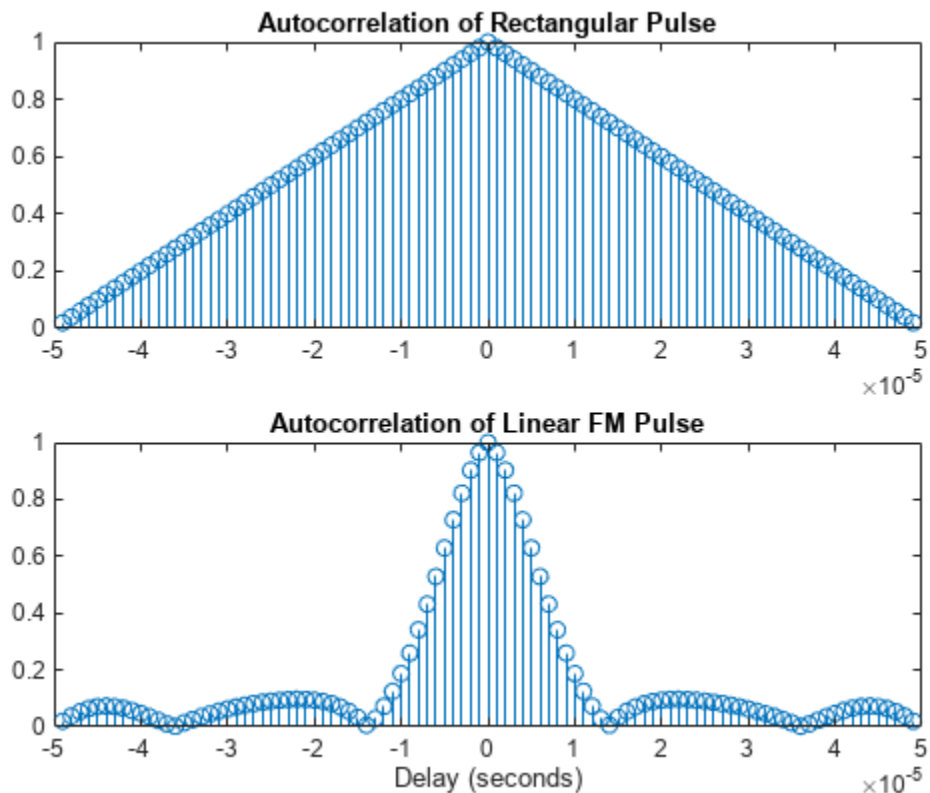


## See Also

## Related Examples

- "Waveform Analysis Using the Ambiguity Function" on page 18-155

# Stepped FM Pulse Waveforms

A stepped frequency pulse waveform consists of a series of *N* narrowband pulses. The frequency is increased from step to step by a fixed amount, Δf, in Hz.

Similar to linear FM pulse waveforms, stepped frequency waveforms are a popular pulse compression technique. Using this approach enables you to increase the range resolution of the radar without sacrificing target detection capability.

To create a stepped FM pulse waveform, use `phased.SteppedFMWaveform`.

The stepped frequency pulse waveform has the following modifiable properties:

- `SampleRate` — Sampling rate in Hz
- `PulseWidth` — Pulse duration in seconds
- `PRF` — Pulse repetition frequency in Hz
- `FrequencyStep` — Frequency step in Hz
- `NumSteps` — Number of frequency steps
- `OutputFormat` — Output format in pulses or samples
- `NumSamples` — Number of samples in the output when the `OutputFormat` property is `'Samples'`
- `NumPulses` — Number of pulses in the output when the `OutputFormat` property is `'Pulses'`

## Create and Plot Stepped FM Pulse Waveform

This example shows how to create and plot a 5-step stepped FM pulse waveform using the `phased.SteppedFM` System object™. Set the pulse width (duration) to 50 µs, the pulse repetition frequency (PRF) to 10 kHz, and the frequency step size to 20 kHz. The sampling rate is 1 MHz. By default, the `OutputFormat` property is set to `'Pulses'` and `NumPulses` is one.

```
waveform = phased.SteppedFMWaveform('SampleRate',1e6,...
    'PulseWidth',50e-6,'PRF',10e3,...
    'FrequencyStep',20e3,'NumSteps',5);
```

Use the `bandwidth` method to show that the bandwidth of the stepped FM pulse waveform equals the product of the frequency step size and the number of steps.

```
bandwidth(waveform)
```

```
ans =

    100000
```

Because the `OutputFormat` property is set to 'Pulses' and the `NumPulses` property is set to one, executing the System object returns one pulse repetition interval (PRI). The pulse duration within that interval is set by the `PulseWidth` property. The signal in the remainder of the PRI consists of zeros.

The frequency of the initial pulse is zero Hz (DC). Each time you execute the System object, the frequency of the narrowband pulse increments by the value of the `FrequencyStep` property. If you

execute the System object more times than the value of the `NumSteps` property, the process repeats, starting over with the DC pulse.

Execute the System object to return successively higher frequency pulses. Plot the pulses one by one in the same figure window. Pause the loop to visualize the increment in frequency with each execution of the System object. Execute the System object one more time than the number of pulses to demonstrate that the process starts over with the DC pulse.

This figure shows the pulse plot for the last iteration of the loop.

```
t = unigrid(0,1/waveform.SampleRate,1/waveform.PRF,'[)');
for i = 1:waveform.NumSteps
    plot(t,real(waveform()))
    pause(0.5)
    axis tight
end
```
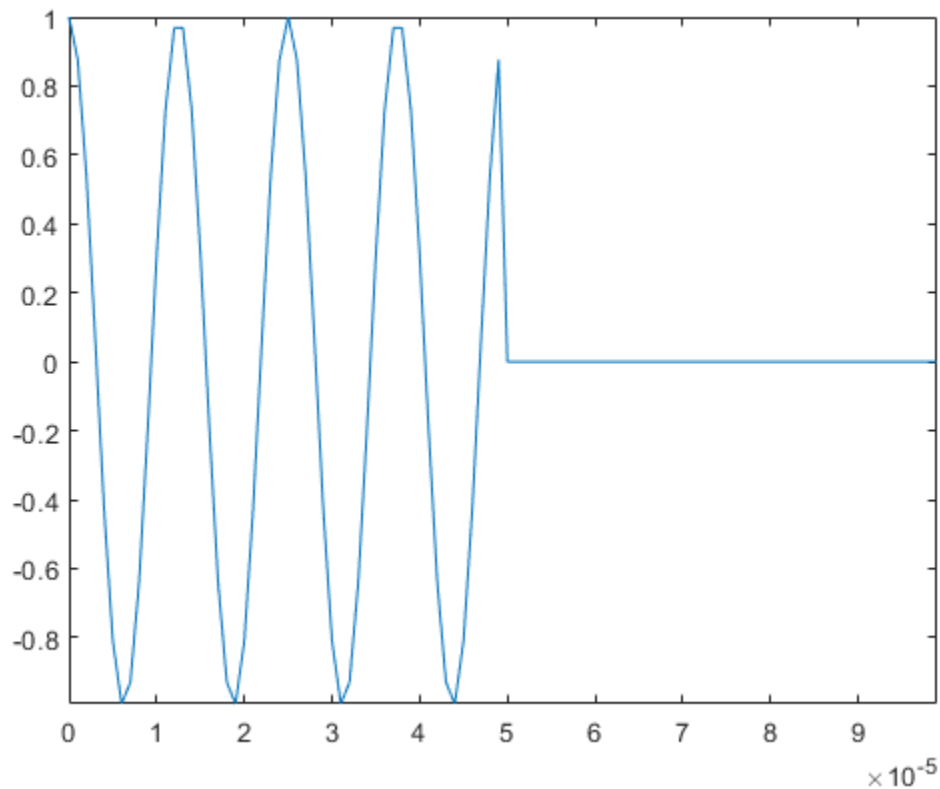


This plot shows how the pulse returns to its DC value.

```
plot(t,waveform())
```
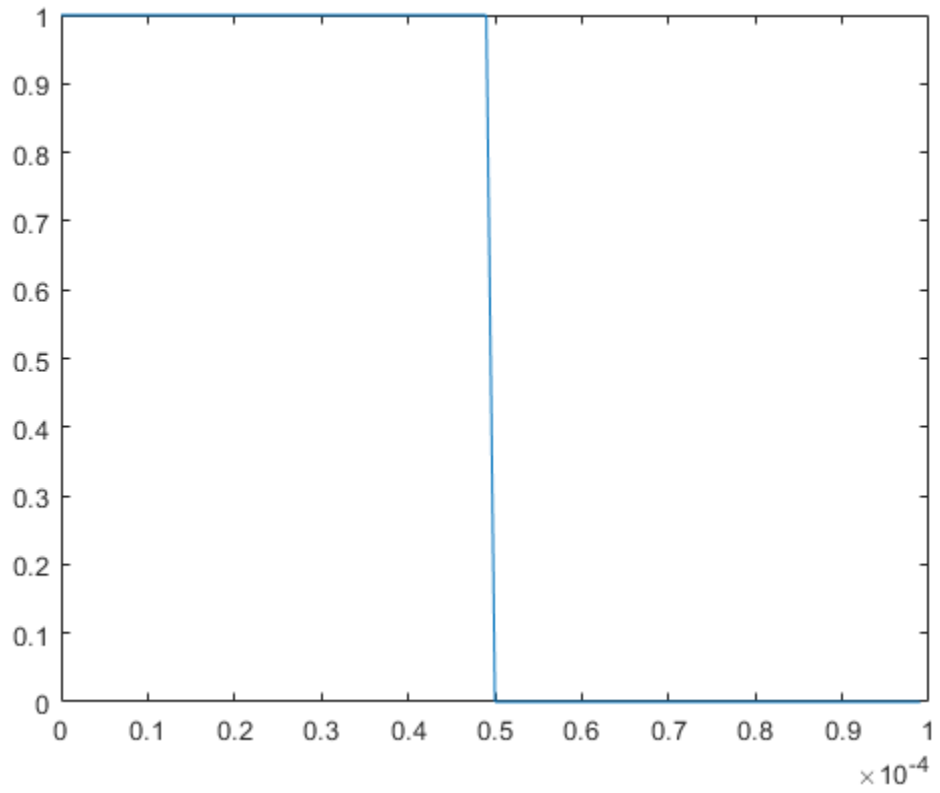
# FMCW Waveforms

| In this section... |
|---|
| "Benefits of Using FMCW Waveform" on page 4-15 |
| "How to Create FMCW Waveforms" on page 4-15 |
| "Double Triangular Sweep" on page 4-15 |

## Benefits of Using FMCW Waveform

Radar systems that use frequency-modulated, continuous-wave (FMCW) waveforms are typically smaller and less expensive to manufacture than pulsed radar systems. FMCW waveforms can estimate the target range effectively, whereas the simplest continuous-wave waveforms cannot.

FMCW waveforms are common in automotive radar systems and ground-penetrating radar systems.

## How to Create FMCW Waveforms

To create an FMCW waveform, use `phased.FMCWWaveform`. You can customize certain characteristics of the waveform, including:

- Sample rate.
- Period and bandwidth of the FM sweep. These quantities can cycle through multiple values during your simulation.

  **Tip**  To find targets up to a given maximum range, `r`, you can typically use a sweep period of approximately `5*range2time(r)` or `6*range2time(r)`. To achieve a range resolution of `delta_r`, use a bandwidth of at least `range2bw(delta_r)`.

- Sweep shape. This shape can be sawtooth (up or down) or triangular.

  **Tip**  For moving targets, you can use a triangular sweep to resolve ambiguity between range and Doppler.

`phased.FMCWWaveform` assumes that all frequency modulations are linear. For triangular sweeps, the slope of the down sweep is the opposite of the slope of the up sweep.

## Double Triangular Sweep

This example shows how to sample an FMCW waveform with a double triangular sweep in which the two sweeps have different slopes. Then, the example plots a spectrogram.

Create an FMCW waveform object for which the `SweepTime` and `SweepBandwidth` properties are vectors of length two. For each period, the waveform alternates between the pairs of corresponding sweep time and bandwidth values.

```
st = [1e-3 1.1e-3];
bw = [1e5 9e4];
waveform = phased.FMCWWaveform('SweepTime',st,...
```

```
'SweepBandwidth',bw,'SweepDirection','Triangle',...
'SweepInterval','Symmetric','SampleRate',2e5,...
'NumSweeps',4);
```

Compute samples from four sweeps (two periods). In a triangular sweep, each period consists of an up sweep and down sweep.

```
x = waveform();
```

Plot the spectrogram.

```
[S,F,T] = spectrogram(x,32,16,32,waveform.SampleRate);
image(T,fftshift(F),fftshift(mag2db(abs(S))))
xlabel('Time (sec)')
ylabel('Frequency (Hz)')
```
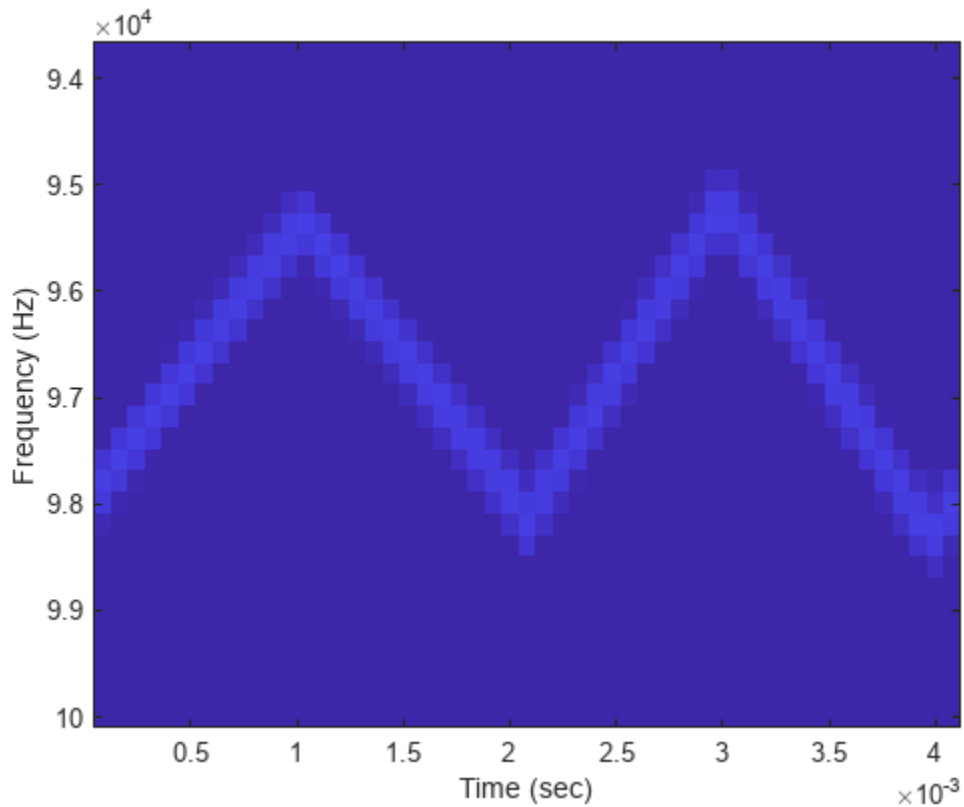
# Phase-Coded Waveforms

## When to Use Phase-Coded Waveforms

Situations in which you might use a phase-coded waveform instead of another type of waveform include:

- When a rectangular pulse cannot provide both of these characteristics:

  - Short enough pulse for good range resolution
  - Enough energy in the signal to detect the reflected echo at the receiver

- When two or more radar systems are close to each other and you want to reduce interference among them.

- When digital processing suggests using a waveform with a discrete set of phases. For example, a Barker-coded waveform is a bi-phase waveform.

Conversely, you might use another waveform instead of a phase-coded waveform in the following situations:

- When you need to detect or track high-speed targets

  Phase-coded waveforms tend to perform poorly when signals have Doppler shifts.

- When the hardware requirements for phase-coded waveforms are prohibitively expensive

## How to Create Phase-Coded Waveforms

To create a phase-coded waveform, use `phased.PhaseCodedWaveform`. You can customize certain characteristics of the waveform, including:

- Type of phase code
- Number of chips
- Chip width
- Sample rate
- Pulse repetition frequency (PRF)
- Sequence index (Zadoff-Chu code only)

After you create a `phased.PhaseCodedWaveform` object, you can plot the waveform using the `plot` method of this class. You can also generate samples of the waveform using the `step` method.

For a full list of properties and methods, see the `phased.PhaseCodedWaveform` reference page.

# Basic Radar Using Phase-Coded Waveform

Instead of the rectangular waveform used in the "Signal Simulation" example, you can use a phase-coded waveform. To do so, replace the `phased.RectangularWaveform` System object™ with `phased.PhaseCodedWaveform`.

```
waveform = phased.PhaseCodedWaveform('Code','Frank','NumChips',4,...
    'ChipWidth',1e-6,'PRF',5e3,'OutputFormat','Pulses',...
    'NumPulses',1);
```

Then, redefine the pulse width, `tau` , using the properties of the new waveform.

```
tau = waveform.ChipWidth*waveform.NumChips;
```

The remainder of the code is almost identical to the code in the original examples and is presented here without comments. For a detailed explanation of how the code works, see the original "Signal Simulation" example.

```
antenna = phased.IsotropicAntennaElement('FrequencyRange',[1e9 10e9]);
target = phased.RadarTarget('Model','Nonfluctuating',...
    'MeanRCS',0.5,'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
transmitterplatform = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0]);
targetplatform = phased.Platform('InitialPosition',[7000; 5000; 0],...
    'Velocity',[-15;-10;0]);
[tgtrng,tgtang] = rangeangle(targetplatform.InitialPosition,...
    transmitterplatform.InitialPosition);

% Use the radar equation to compute the transmitted power needed for a given Pd and Pfa.
numpulses = 10;
Pd = 0.9;
Pfa = 1e-6;
maxrange = 1.5e4;
lambda = physconst('LightSpeed')/target.OperatingFrequency;
RCS = 0.5;
Ts = 290;
SNR = albersheim(Pd,Pfa,10);
Gain = 20;
dbterms = db2pow(SNR - 2*Gain);
Pt = (4*pi)^3*physconst('Boltzmann')*Ts/tau/RCS/(lambda^2)*maxrange^4*dbterms;

transmitter = phased.Transmitter('PeakPower',50e3,'Gain',20,...
    'LossFactor',0,'InUseOutputPort',true,...
    'CoherentOnTransmit',true);
radiator = phased.Radiator('Sensor',antenna,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
collector = phased.Collector('Sensor',antenna,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'Wavefront','Plane','OperatingFrequency',4e9);
receiver = phased.ReceiverPreamp('Gain',20,'NoiseFigure',2,...
    'ReferenceTemperature',290,'SampleRate',1e6,...
    'EnableInputPort',true,'SeedSource','Property','Seed',1e3);
channel = phased.FreeSpace(...
    'PropagationSpeed',physconst('LightSpeed'),...
```

```matlab
    'OperatingFrequency',4e9,'TwoWayPropagation',false,...
    'SampleRate',1e6);

T = 1/waveform.PRF;
txpos = transmitterplatform.InitialPosition;

rxsig = zeros(waveform.SampleRate*T,numpulses);
for n = 1:numpulses
    [tgtpos,tgtvel] = targetplatform(T);
    [tgtrng,tgtang] = rangeangle(tgtpos,txpos);
    sig = waveform();
    [sig,txstatus] = transmitter(sig);
    sig = radiator(sig,tgtang);
    sig = channel(sig,txpos,tgtpos,[0;0;0],tgtvel);
    sig = target(sig);
    sig = channel(sig,tgtpos,txpos,tgtvel,[0;0;0]);
    sig = collector(sig,tgtang);
    rxsig(:,n) = receiver(sig,~txstatus);
end
rxsig = pulsint(rxsig,'noncoherent');
t = unigrid(0,1/receiver.SampleRate,T,'[)');
rangegates = (physconst('LightSpeed')*t)/2;
plot(rangegates,rxsig)
hold on
xlabel('Meters'); ylabel('Power')
ylim = get(gca,'YLim');
plot([tgtrng,tgtrng],[0 ylim(2)],'r')
hold off
```
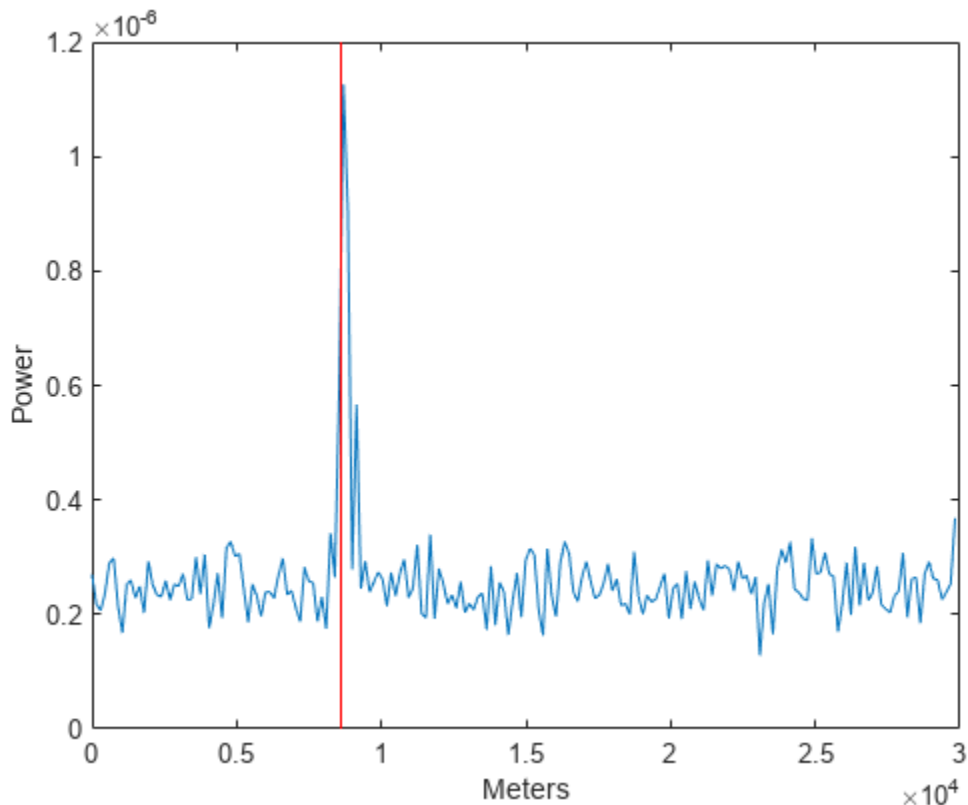
# Waveforms with Staggered PRFs

| In this section... |
| --- |
| "When to Use Staggered PRFs" on page 4-21 |
| "Linear FM Waveform with Staggered PRF" on page 4-21 |

## When to Use Staggered PRFs

Using a nonconstant PRF has important applications in radar. This approach is called *PRF* or *PRI staggering*.

Uses of staggered PRFs include

- Removal of Doppler ambiguities, or *blind speeds*, where Doppler frequencies that are multiples of the PRF are aliased to zero
- Mitigation of the effects of jamming

To implement a staggered PRF, configure your waveform object using a vector instead of a scalar for the PRF property value.

## Linear FM Waveform with Staggered PRF

This example shows how to model a linear FM pulse waveform with two PRFs of 1 and 2 kHz. Set the sweep bandwidth to 200 kHz and the duration of 100 µs. The sample rate is 1 MHz. Output 5 pulses.

```
prfs = [1e3 2e3];
waveform = phased.LinearFMWaveform('PRF',prfs,'SweepBandwidth',200e3,...
    'PulseWidth',100e-6,'NumPulses',5);
sig = waveform();
T = length(sig)*(1/waveform.SampleRate);
t = unigrid(0,1/waveform.SampleRate,T,'[)');
plot(t.*1000,real(sig))
set(gca,'xtick',[0 1 1.5 2.5 3]);
xlabel('millisec')
```
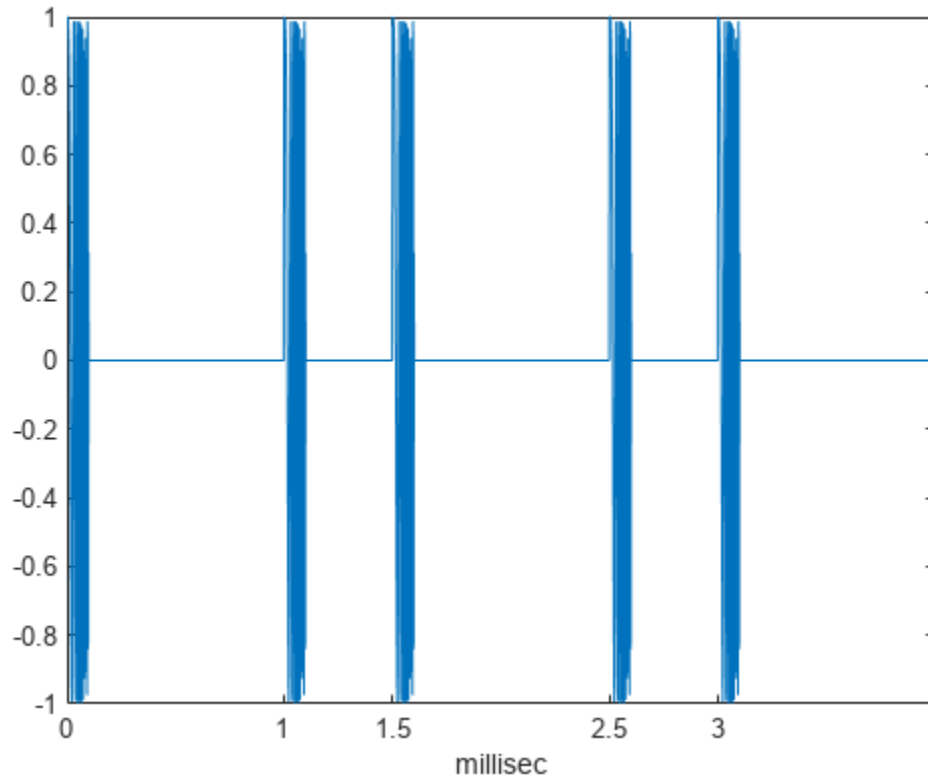
# Plot Spectrogram Using Pulse Waveform Analyzer App

The `pulseWaveformAnalyzer` is a MATLAB® App that lets you explore important properties of a signal such as its waveform, spectrum, and ambiguity function.

**Open pulseWaveformAnalyzer App**

When you type `pulseWaveformAnalyzer` from the command line or select the app from the **App Toolstrip**, an interactive window opens. The default window shows a rectangular waveform and its spectrum. You can then select various options to analyze different waveforms.

pulseWaveformAnalyzer
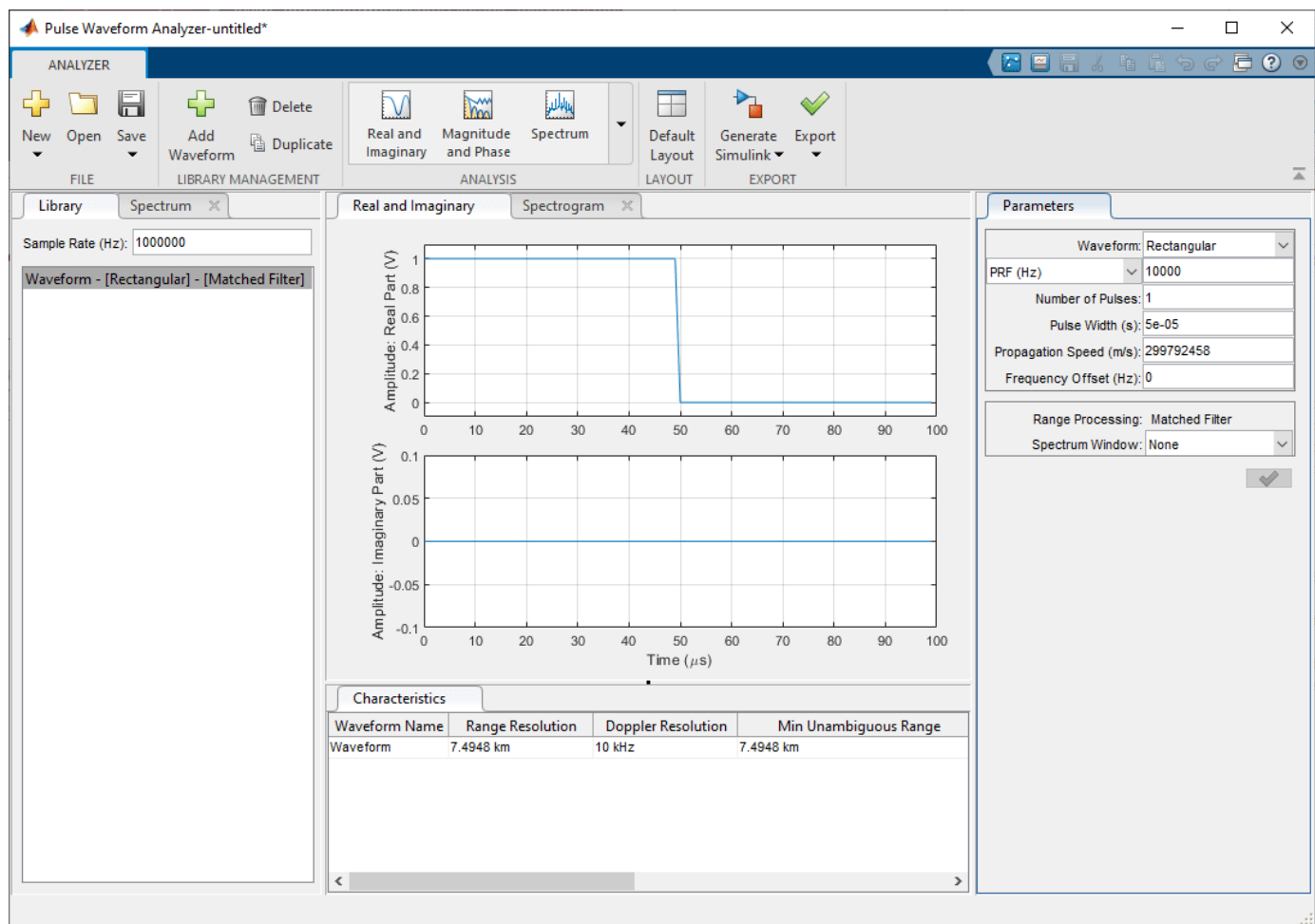


**Show the spectrogram of baseband FMCW signal**

As an example, use the app to show the spectrogram of a continuous FMCW waveform.

1   Set the **Waveform** to FMCW
2   Set the **Sweep Interval** to `Symmetric`
3   Set the **Number of Sweeps** to 4

**4** In the **Analysis** tab, select `Spectrogram`

Then, you will see a plot of the spectrogram of the signal similar to this.
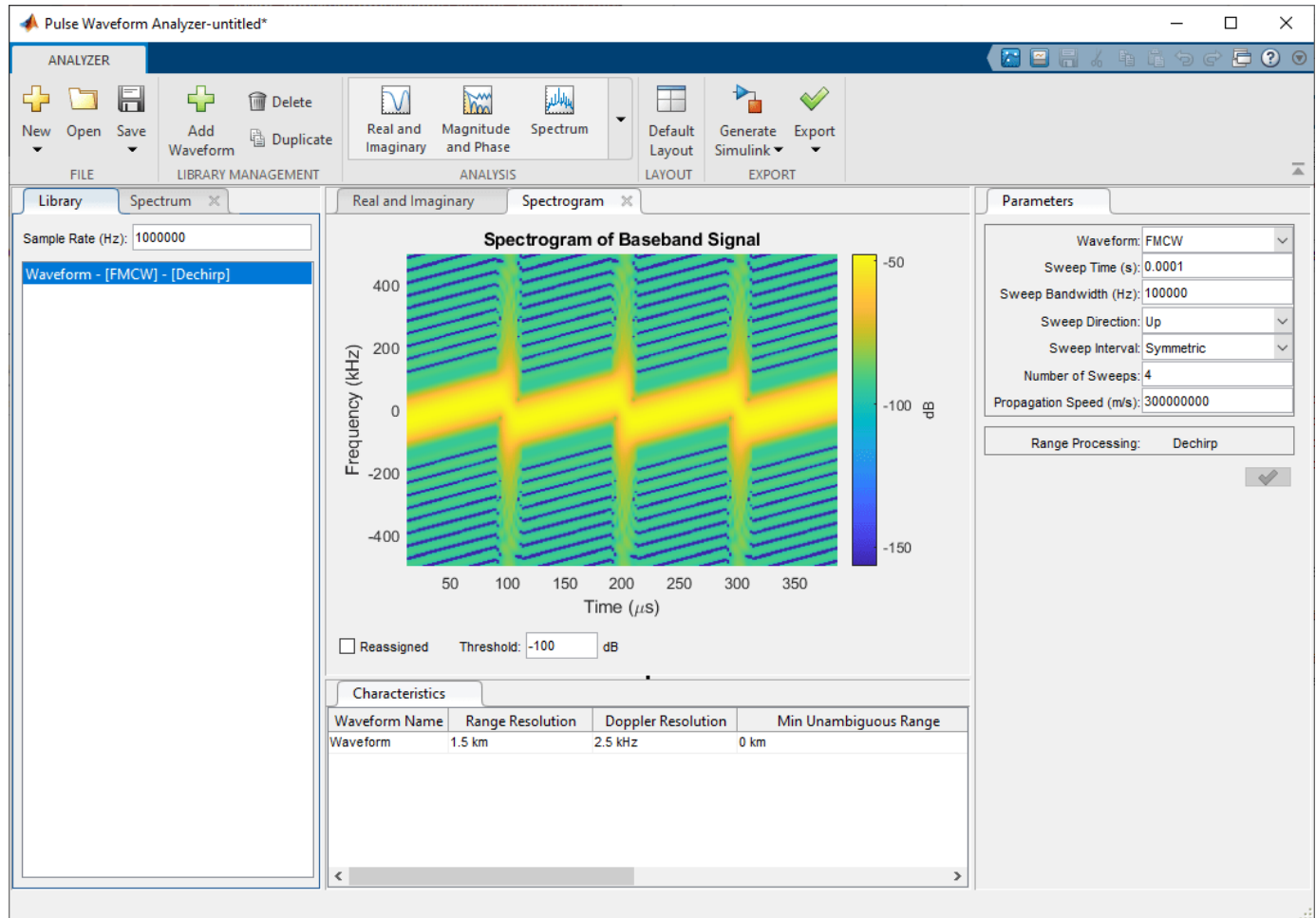
# Transmitter

| **In this section...** |
| --- |
| |
| |

## Transmitter Object

The `phased.Transmitter` object lets you model key components of the *radar equation* including the peak transmit power, the transmit gain, and a system loss factor.

While the preceding functionality is important in applications dependent on amplitude such as signal detectability, Doppler processing depends on the phase of the complex envelope. In order to accurately estimate the radial velocity of moving targets, it is important that the radar operates in either a *fully coherent* or *pseudo-coherent* mode. In the fully coherent, or *coherent on transmit*, mode, the phase of the transmitted pulses is constant. Constant phase provides you with a reference to detect Doppler shifts.

A transmitter that applies a random phase to each pulse creates *phase noise* that can obscure Doppler shifts. If the components of the radar do not enable you to maintain constant phase, you can create a pseudo-coherent, or *coherent on receive* radar by keeping a record of the random phase errors introduced by the transmitter. The receiver can correct for these errors by modulation of the complex envelope. The `phased.Transmitter` object enables you to model both coherent on transmit and coherent on receive behavior.

The transmitter object has the following modifiable properties:

- `PeakPower` — Peak transmit power in watts
- `Gain` — Transmit gain in decibels
- `LossFactor` — Loss factor in decibels
- `InUseOutputPort` — Track transmitter's status. Setting this property to `true` outputs a vector of 1s and 0s indicating when transmitter is on and off. In a monostatic radar, the transmitter and receiver cannot operate simultaneously.
- `CoherentOnTransmit` — Preserve *coherence* among transmitter pulses. Setting this property to `true` (the default) models the operation of a fully coherent transmitter where the pulse-to-pulse phase is constant. Setting this property to `false` introduces random phase noise from pulse to pulse and models the operation of a non-coherent transmitter.
- `PhaseNoiseOutputPort` — Output the random pulse phases introduced by non-coherent operation of the transmitter. This property only applies if the `CoherentOnTransmit` property is `false`. By keeping a record of the random pulse phases, you can create a *pseudo-coherent*, or *coherent on receive* radar.

**Transmit Linear FM Pulse**

Amplify and transmit a linear FM pulse.

Construct a transmitter with a peak transmit power of 1000 watts, a transmit gain of 20 decibels (dB), and a loss factor of 0 dB. Set the `InUseOutPutPort` property to `true` to record the transmitter

status. Pulse waveform values are scaled based on the peak transmit power and the ratio of the transmitter gain to loss factor.

```
transmitter = phased.Transmitter('PeakPower',1e3,'Gain',20,...
    'LossFactor',0,'InUseOutputPort',true)

transmitter =
  phased.Transmitter with properties:

             PeakPower: 1000
                  Gain: 20
            LossFactor: 0
       InUseOutputPort: true
     CoherentOnTransmit: true
```

The waveform scaling factor is `sqrt(PeakPower*db2pow(Gain - LossFactor))`

Construct a linear FM pulse waveform for transmission. Use a 100 µsec linear FM pulse having a bandwidth of 200 kHz. Use the default sweep direction and sample rate. Set the pulse repetition frequency (PRF) to 2 kHz. Obtain one pulse by setting the `NumPulses` property of the `phased.LinearFMWaveform` object to unity.

```
waveform = phased.LinearFMWaveform('PulseWidth',100e-6,'PRF',2e3,...
    'SweepBandwidth',200e3,'OutputFormat','Pulses','NumPulses',1);
```

Generate the pulse by executing the `phased.LinearFMWaveform` waveform System object™. Then, transmit the pulse by executing the `phased.Transmitter` System object.

```
wf = waveform();
[txoutput,txstatus] = transmitter(wf);
t = unigrid(0,1/waveform.SampleRate,1/waveform.PRF,'[)');
subplot(211)
plot(t,real(txoutput))
axis tight
grid on
ylabel('Amplitude')
title('Transmitter Output (real part) - One PRI')
subplot(212)
plot(t,txstatus)
axis([0 t(end) 0 1.5])
xlabel('Seconds')
grid on
ylabel('Off-On Status')
set(gca,'ytick',[0 1])
title('Transmitter Status')
```
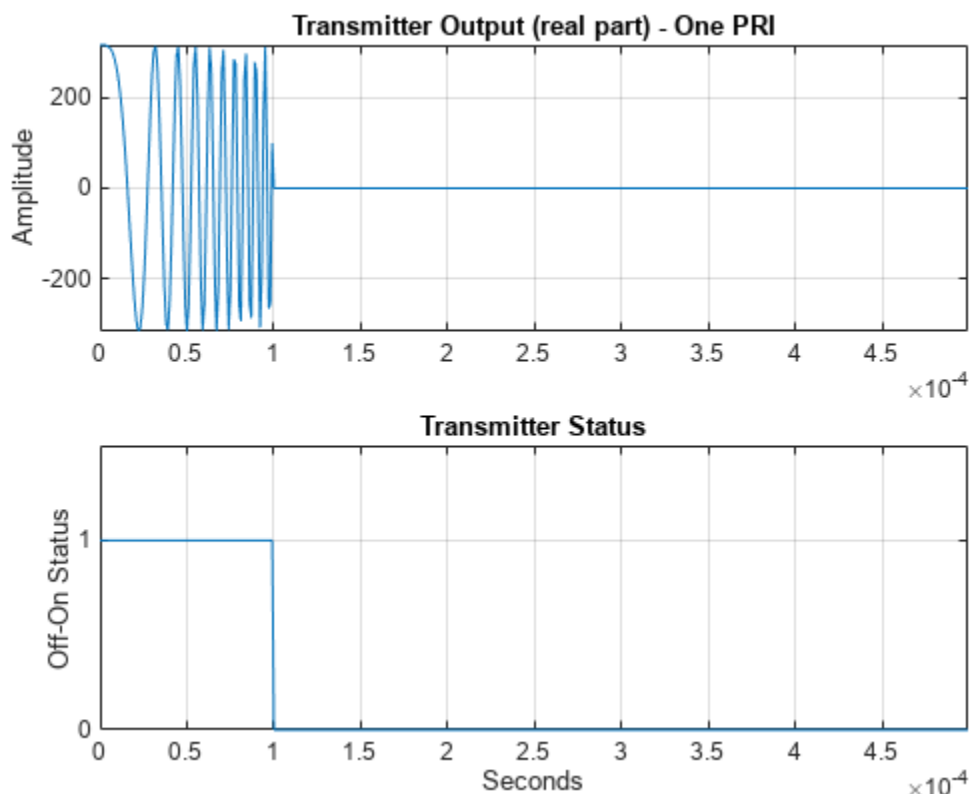
## Phase Noise

To model a coherent on receive radar, you can set the `CoherentOnTransmit` property to `false` and the `PhaseNoiseOutputPort` property to `true`. You can output the random phase added to each sample when you execute the System object.

**Transmit Rectangular Pulse with Phase Noise**

This example illustrates adding phase noise to a rectangular pulse waveform having five pulses. A random phase is added to each sample of the waveform. Compute the phase of the output waveform and compare the phase to the phase noise returned when executing the System object™.

For convenience, set the gain of the transmitter to 0 dB, the peak power to 1 W, and seed the random number generator to ensure reproducible results.

```
waveform = phased.RectangularWaveform('NumPulses',5);
transmitter = phased.Transmitter('CoherentOnTransmit',false,...
    'PhaseNoiseOutputPort',true,'Gain',0,'PeakPower',1,...
    'SeedSource','Property','Seed',1000);
wf = waveform();
[txtoutput,phnoise] = transmitter(wf);
phdeg = rad2deg(phnoise);
phdeg(phdeg>180)= phdeg(phdeg>180) - 360;
plot(wf)
```

```
title('Input Waveform')
axis([0 length(wf) 0 1.5])
ylabel('Amplitude')
grid on
```



**Input Waveform**
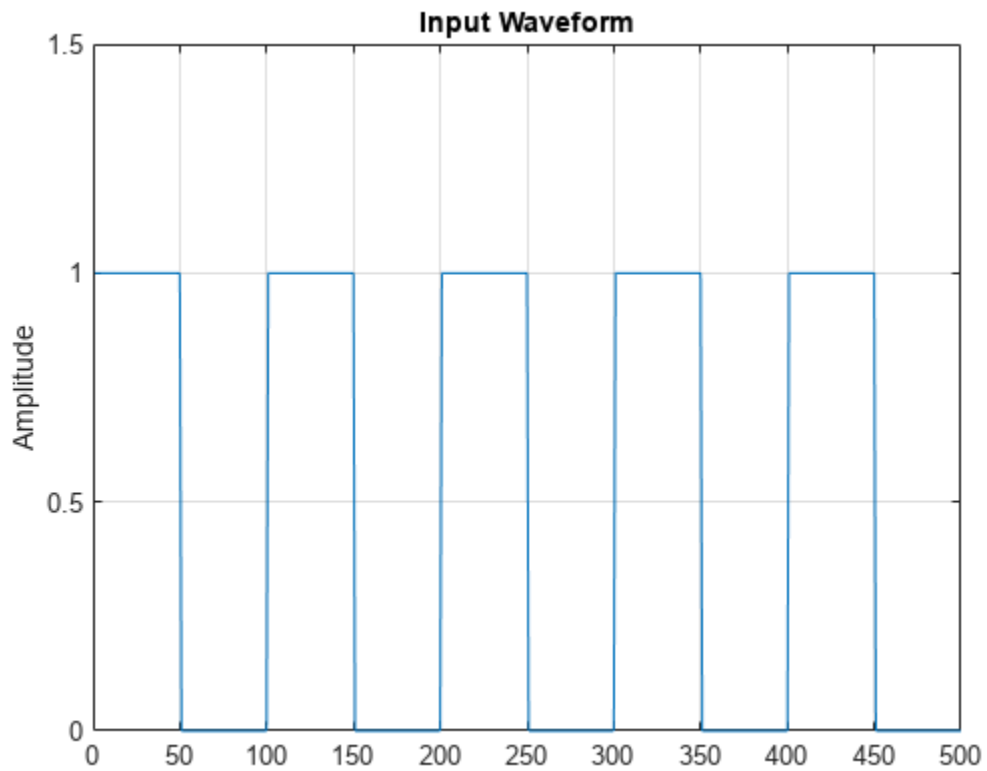
```
subplot(2,1,1)
plot(rad2deg(atan2(imag(txtoutput),real(txtoutput))))
title('Phase of the Output')
ylabel('Degrees')
axis([0 length(wf) -180 180])
grid on
subplot(2,1,2)
plot(phdeg)
title('Phase Noise'); ylabel('Degrees')
axis([0 length(wf) -180 180])
grid on
```

### Phase of the Output



### Phase Noise
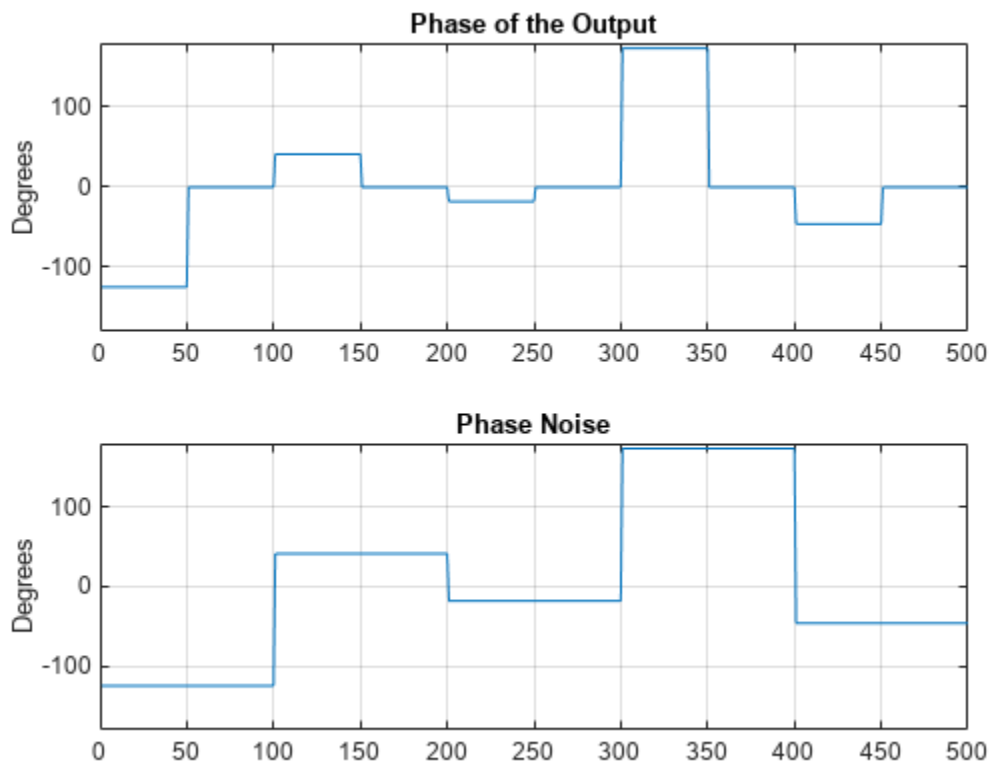


The first figure shows the waveform. The phase of each pulse at the input to the transmitter is zero. In the second figure, the top plot shows the phase of the transmitter output waveform. The bottom plot shows the phase added to each sample. Focus on the first 100 samples. The pulse waveform is equal to 1 for samples 1-50 and 0 for samples 51-100. The added random phase is a constant -124.7° for samples 1-100, but this affects the output only when the pulse waveform is nonzero. In the output waveform, you see that the output waveform has a phase of -124.7° for samples 1-50 and 0 for samples 51-100. Examining the transmitter output and phase noise for samples where the input waveform is nonzero, you can see that the phase output the System object and the phase of the transmitter output agree.

# Receiver Preamp

## Operation of Receiver Preamp

The `phased.ReceiverPreamp` object lets you model the effects of gain and component-based noise on the signal-to-noise ratio (SNR) of received signals. `phased.ReceiverPreamp` operates on baseband signals. The object is not intended to model system effects at RF or intermediate frequency (IF) stages.

## Configuring Receiver Preamp

The `phased.ReceiverPreamp` object has the following modifiable properties:

- `EnableInputPort` — A logical property that enables you to specify when the receiver is on or off. Input the actual status of the receiver as a vector to `step`. This property is useful when modeling a monostatic radar system. In a monostatic radar, it is important to ensure the transmitter and receiver are not operating simultaneously. See `phased.Transmitter` and "Transmitter" on page 4-25.
- `Gain` — Gain in dB ($G_{dB}$)
- `LossFactor` — Loss factor in dB ($L_{dB}$)
- `NoiseMethod` — Specify noise input as noise power or noise temperature
- `NoiseFigure` — Receiver noise figure in dB ($F_{dB}$)
- `ReferenceTemperature` — Receiver reference temperature in kelvin ($T$)
- `SampleRate` — Sample rate ($f_s$)
- `NoisePower` — Noise power specified in Watts ($\sigma^2$)
- `NoiseComplexity` — Specify noise as real-valued or complex-valued
- `EnableInputPort` — Add input to specify when the receiver is active
- `PhaseNoiseInputPort` — Add input to specify phase noise for coherent on receive receiver
- `SeedSource` — Lets you specify random number generator seed
- `Seed` — Random number generator seed

The output signal, *y[n]*, of the `phased.ReceiverPreamp` System object equals the input signal scaled by the ratio of receiver amplitude gain to amplitude loss plus additive noise

$$y[n] = \frac{G}{L}x[n] + \frac{\sigma}{\sqrt{2}}w[n]$$

where *x[n]* is the complex-valued input signal and *w[n]* is unit-variance noise complex-valued noise.

When the input signal is real-valued, the output signal, *y[n]*, equals the real-valued input signal scaled by the ratio of receiver amplitude gain to amplitude loss plus real-valued additive noise

$$y[n] = \frac{G}{L}x[n] + \sigma w[n]$$

.

The amplitude gain, *G*, and loss, *L*, can be express in terms of the input dB parameters by

$$G = 10^{G_{dB}/20}$$

$$L = 10^{L_{dB}/20}$$

.

respectively.

The additive noise for the receiver is modeled as a zero-mean complex white Gaussian noise vector with variance, $\sigma^2$, equal to the noise power. The real and imaginary parts of the noise vector each have variance equal to 1/2 the noise power.

You can set the noise power directly by choosing the `NoiseMethod` property to be `'Noise power'` and then setting the `NoisePower` property to a real positive number. Alternatively, you can set the noise power using the system temperature by choosing the `NoiseMethod` property to be `'Noise temperature'`. Then

$$\sigma^2 = k_B B T F$$

where $k_B$ is Boltzmann's constant, *B* is the noise bandwidth which is equal to the sample rate, $f_s$, *T* is the system temperature, and *F* is the noise figure in power units.

The noise figure, *F*, is a dimensionless quantity that indicates how much a receiver deviates from an ideal receiver in terms of internal noise. An ideal receiver produces thermal noise power defined by noise bandwidth and temperature. In terms of power units, the noise figure $F = 10^{F_{dB}/10}$. A noise figure of 0 dB indicates that the noise power of a receiver equals the noise power of an ideal receiver. Because an actual receiver cannot exhibit a noise power value less than an ideal receiver, the noise figure is always greater than or equal to one. In decibels, the noise figure must be greater than or equal to zero.

To model the effect of the receiver preamp on the signal, `phased.ReceiverPreamp` computes the *effective system noise temperature* by taking the product of the reference temperature, *T*, and the noise figure *F* in power units. See `systemp` for details.

## Model Receiver Effects on Sinusoidal Input

Specify a `phased.ReceiverPreamp` System object™ with a gain of 20 dB, a noise figure of 5 dB, and a reference temperature of 290 degrees kelvin.

```
receiver = phased.ReceiverPreamp('Gain',20,...
    'NoiseFigure',5,'ReferenceTemperature',290,...
    'SampleRate',1e6,'SeedSource','Property','Seed',1e3);
```

Assume a 100-Hz sine wave input with an amplitude of 1 microvolt. Because the Phased Array System Toolbox™ assumes that all modeling is done at baseband, use a complex exponential as the input when executing the System object.

```
t = unigrid(0,0.001,0.1,'[)');
x = 1e-6*exp(1i*2*pi*100*t).';
y = receiver(x);
```

The output of the `phased.ReceiverPreamp.step` method is complex-valued as expected.

Now show how the same output can be produced from the multiplicative amplitude gain and additive noise. First assume that the noise bandwidth equals the sample rate of the receiver preamp (1 MHz). Then, the noise power is equal to:

```
NoiseBandwidth = receiver.SampleRate;
noisepow = physconst('Boltzmann')*...
    systemp(receiver.NoiseFigure,receiver.ReferenceTemperature)*NoiseBandwidth;
```

The noise power is the variance of the additive white noise. To determine the correct amplitude scaling of the input signal, note that the gain is 20 dB. Because the loss factor in this case is 0 dB, the scaling factor for the input signal is found by solving the following equation for the multiplicative gain $G$ from the gain in dB, $G_{dB}$:

$$G = 10^{(G_{dB}/20)}$$

```
G = 10^(receiver.Gain/20)
```

```
G = 10
```

The gain is 10. By scaling the input signal by a factor of ten and adding complex white Gaussian noise with the appropriate variance, you produce an output equivalent to the preceding call to `phased.ReceiverPreamp.step` (use the same seed for the random number generation).

```
rng(1e3);
y1 = G*x + sqrt(noisepow/2)*(randn(size(x))+1j*randn(size(x)));
```

Compare a few values of `y` to `y1`.

```
disp(y1(1:10) - y(1:10))
```

```
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

# Model Coherent-on-Receive Behavior

When modeling a coherent-on-receive monostatic radar, use the `EnableInputPort` and `PhaseNoiseInputPort` properties. In a monostatic radar, the transmitter and receiver cannot operate simultaneously. Therefore, it is important to keep track of when the transmitter is active so that you can disable the receiver at those times. By setting the `EnableInputPort` to `true`, you can input a record of when the transmitter is active as an argument when executing the `phased.ReceiverPreamp` System object.

In a coherent-on-receive radar, the receiver corrects for the phase noise introduced at the transmitter by using the record of those phase errors. By setting the `PhaseNoiseInputPort` property to `true`, you can input a record of the transmitter phase errors as an argument when executing the `phased.ReceiverPreamp` System object.

## Coherent-on-Receive for Rectangular Pulse

To illustrate coherent-on-receive, construct a rectangular pulse waveform with five pulses. The waveform pulse repetition frequency (PRF) is 10 kHz and the pulse width is 50 µs. The pulse repetition interval (PRI) is exactly two times the pulse width so the transmitter alternates between active and inactive time intervals of the same duration. For convenience, set the gains on both the transmitter and receiver to 0 dB and the peak power on the transmitter to 1 W.

Use the `PhaseNoiseOutputPort` and `InUseOutputPort` properties of the transmitter to record the phase noise and the status of the transmitter.

Enable the `EnableInputPort` and `PhaseNoiseInputPort` properties of the receiver preamp to determine when the receiver is active and to correct for the phase noise introduced at the transmitter.

Delay the output of the transmitter using the `delayseq` function to simulate the waveform arriving at the receiver preamp when the transmitter is inactive and the receiver is active.

```
waveform = phased.RectangularWaveform('NumPulses',5);
transmitter = phased.Transmitter('CoherentOnTransmit',false,...
    'PhaseNoiseOutputPort',true,'Gain',0,'PeakPower',1,...
    'SeedSource','Property','Seed',1000,'InUseOutputPort',true);
wf = waveform();
[troutput,trstatus,phasenoise] = transmitter(wf);
troutput = delayseq(troutput,waveform.PulseWidth,...
    waveform.SampleRate);
receiver = phased.ReceiverPreamp('Gain',0,...
    'PhaseNoiseInputPort',true,'EnableInputPort',true);
y = receiver(troutput,~trstatus,phasenoise);
subplot(2,1,1)
plot(real(troutput))
title('Delayed Transmitter Output with Phase Noise')
ylabel('Amplitude')
subplot(2,1,2)
plot(real(y))
xlabel('Samples')
ylabel('Amplitude')
title('Received Signal with Phase Correction')
```

**5**
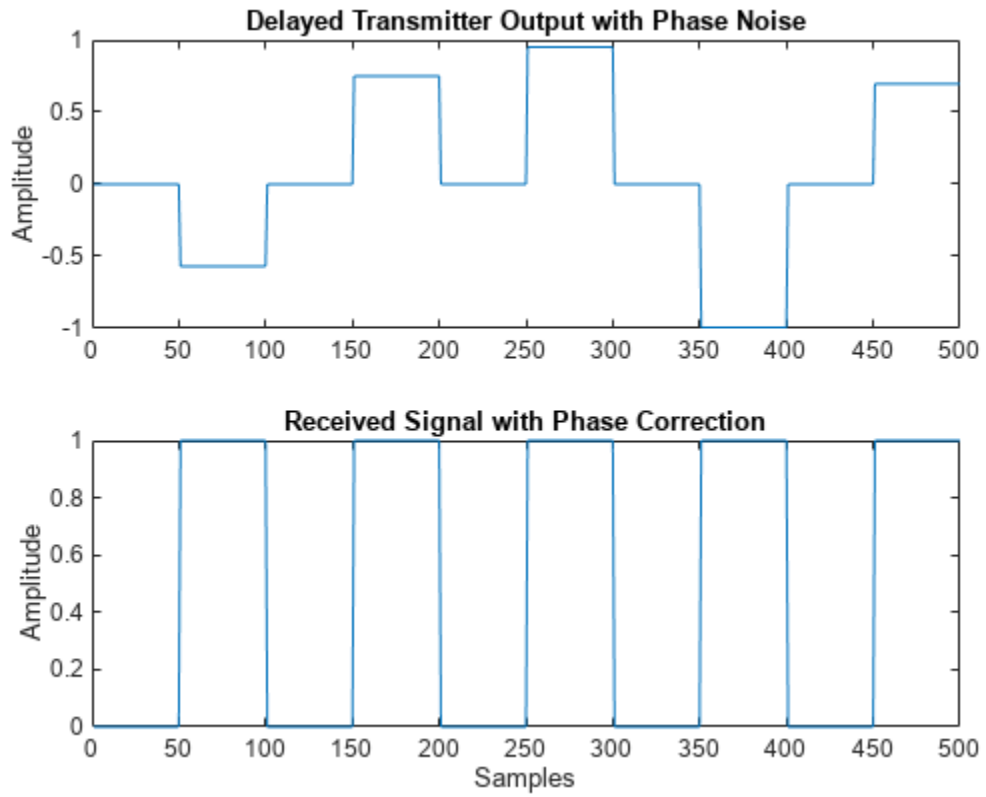
# Beamforming

# Beamforming Overview

| In this section... |
| --- |
| "Conventional Beamforming" on page 5-2 |
| "Optimal and Adaptive Beamforming" on page 5-3 |

Beamforming is the spatial equivalent of frequency filtering and can be grouped into two classes: data independent (conventional) and data-dependent (adaptive). All beamformers are designed to emphasize signals coming from some directions and suppress signals and noise arriving from other directions.

Phased Array System Toolbox provides nine different beamformers. This table summarizes the main properties of the beamformers.

| Beamformer Name | Conventional or Adaptive | Bandwidth | Processing Domain |
| --- | --- | --- | --- |
| phased.PhaseShiftBeamformer | Conventional | Narrowband | Time domain |
| phased.TimeDelayBeamformer | Conventional | Wideband | Time domain |
| phased.SubbandPhaseShiftBeamformer | Conventional | Wideband | Frequency domain |
| phased.LCMVBeamformer | Adaptive | Narrowband | Frequency domain |
| phased.MVDRBeamformer | Adaptive | Narrowband | Frequency domain |
| phased.FrostBeamformer | Adaptive | Wideband | Time domain |
| phased.GSCBeamformer | Adaptive | Wideband | Time domain |
| phased.TimeDelayLCMVBeamformer | Adaptive | Wideband | Time domain |
| phased.SubbandMVDRBeamformer | Adaptive | Wideband | Frequency domain |

## Conventional Beamforming

Conventional beamforming, also called classical beamforming, is the easiest to understand. Conventional beamforming techniques include delay-and-sum beamforming, phase-shift beamforming, subband beamforming, and filter-and-sum beamforming. These beamformers are similar because the weights and parameters that define the beampattern are fixed and do not depend on the array input data. The weights are chosen to produce a specified array response to the signals and interference in the environment. A signal arriving at an array has different times of arrival at each sensor. For example, plane waves arriving at a linear array have a time delay that is a linear function of distance along the array. Delay-and-sum beamforming compensates for these delays by applying a reverse delay to each sensor. If the time delay is accurately computed, the signals from each sensor add constructively.

Finding the compensating delay at each sensor requires accurate knowledge of the sensor locations and signal direction. The delay-and-sum beamformer can be implemented in the frequency domain or in the time domain. When the signal is narrowband, time delay becomes a phase shift in the frequency domain and is implement by multiplying each sensor signal by a frequency-dependent compensatory phase shift. This algorithm is implemented in the `phased.PhaseShiftBeamformer`. For broadband signals, there are several approaches. One approach is to delay the signal in time by a discrete number of samples. A problem with this method is that the degree of resolution that you can distinguish is determined by the sampling rate of your data, because you cannot resolve delay differences less than the sampling interval. Because this technique only works if the sampling rate is high, you must increase the sampling frequency well beyond the Nyquist frequency so that the true delay is very close to a sample time. A second method interpolates the signal between samples. Time delay beamforming is implemented in `phased.TimeDelayBeamformer`. A third method Fourier transforms the signals to the frequency domain, applies a linear phase shift, and converts the signal back into the time domain. Phase-shift beamforming is performed at each frequency band (see `phased.SubbandPhaseShiftBeamformer`).

Beamforming is not limited to plane waves but can be applied even when there is wavefront curvature. In this case, the source lies in the near field. Perhaps the term beamforming is no longer appropriate. You can use the source-array geometry to compute the phase shift for each point in space and then apply this phase shift at each sensor element.

The advantage of a conventional beamformer is simplicity and ease of implementation. Another advantage is its robustness against pointing errors and signal direction errors. A disadvantage is its broad main lobe which decreases resolution of closely spaced sources or targets. A second disadvantage is that it has large sidelobes that allow interference sources to leak into the main beam.

## Optimal and Adaptive Beamforming

The second class of beamformers consists of the data-dependent beamformers. The terms optimal or adaptive beamformers are sometimes used for this class interchangeably but they are not quite the same. Optimal beamformers apply weights that are determined by optimizing some quantity. The MVDR beamformer determines the beamforming weights, $w$, by maximizing the signal-to-noise +interference ratio of the array output

$$\frac{|\mathbf{w'}\mathbf{s}|^2}{\mathbf{w'}\mathbf{R_n}\mathbf{w}} = A^2 \frac{|\mathbf{w'}\mathbf{a}|^2}{\mathbf{w'}\mathbf{R_n}\mathbf{w}}$$

where $s$ represents the signal values at the sensors, $a$ represents the source steering vector, and $A^2$ represents the source power at the array. $R_n$ is the noise+interference covariance matrix. Because the SNR is invariant under any scale factor applied to the weights, an equivalent formulation of this criterion is to minimize the noise output $w'R._nw$ subject to a constraint

$$\mathbf{w'}\mathbf{R_n}\mathbf{w} \quad \text{s.t.} \quad \mathbf{w'}\mathbf{a} = 1$$

The solution of this equation is

$$\mathbf{w}_{\text{opt}} = \frac{\mathbf{R_n}^{-1}\mathbf{a}}{\mathbf{a'}\mathbf{R_n}^{-1}\mathbf{a}}$$

and yields the minimum variance distortionless response (MVDR) beamformer. Because of the constraint, beamformer preserves the desired signal while minimizing contributions to the array output due to noise and interference. The MVDR beamformer is implemented in

`phased.MVDRBeamformer`. A broadband version is implemented in
`phased.SubbandMVDRBeamformer`.

There are several advantages to the MVDR beamformer.

- The beamformer incorporates the noise and interference into an optimal solution.
- The beamformer has higher spatial resolution than a conventional beamformer.
- The beamformer puts nulls in the direction of any interference sources.
- Sidelobes are smaller and smoother.

There are two major disadvantages to the MVDR beamformer. The MVDR beamformer is sensitive to errors in either the array parameters or arrival direction. The MVDR beamformer is susceptible to self-nulling. In addition, trying to use MVDR as an adaptive beamformer requires a matrix inversion every time the noise and interference statistics change. When there are many array elements, the inversion can be computationally expensive.

In practical applications, an accurate steering vector and an accurate covariance matrix are not always available. Generally, all that is available is the sampled covariance matrix. This deficiency can lead to both inadequate interference suppression and distortion of the desired signal. In this case, the true signal direction is slightly off from the beam pointing direction. Then the actual signal is treated as interference.

However it often turns out that the noise is not separable from the signal and it is impossible to determine $R_n$. In that case, you can estimate a sample covariance matrix from the data.

$$\widehat{\mathbf{R}}_{\mathbf{x}} = \frac{1}{K}\sum_{k=1}^{K} x(k)x'(k)$$

and minimizes $w'R_x w$ instead. Minimizing this quantity leads to the minimum power distortionless response (MPDR) beamformer. If the data vector, $x$, contains the signal and the estimated data covariance matrix is perfect and the steering vector of the desired signal is known exactly, the MPDR beamformer is equivalent to the MVDR beamformer. However, MPDR degrades more severely when $R_x$ is estimated from insufficient data or the signal arrival vector is not known precisely.

Rewrite the direction constraint in the form $a'w = 1$ by transposing both sides. This equivalent form suggests that it possible to include multiple constraints by using a matrix constraint $Cw = d$ where $C$ is now a constraint matrix and $d$ represents the signal gains due to the constraints. This is the form used in the linear constraint minimum variance (LCMV) beamformer. The LCMV beamformer is a generalization of MVDR beamforming and is implemented in `phased.LCMVBeamformer` and `phased.TimeDelayLCMVBeamformer`. There are several different approaches to specifying constraints such as amplitude and derivative constraints. You can, for example, specify weights that suppress interfering signals arriving from a particular direction while passing signals from a different direction without distortion. The optimal LCMV weights are determined by the equation

$$\mathbf{w}_{\text{opt}} = \mathbf{R}_n{}^{-1}\mathbf{C}'(\mathbf{C}\mathbf{R}_n{}^{-1}\mathbf{C}')^{-1}\mathbf{d}$$

The advantages and disadvantages of the MVDR beamformer also apply to the LCMV beamformer.

While MVDR and LCMV are adaptive in principle, re-computation of the weights requires the inversion of a potentially large covariance matrix when the array has many elements. The Frost and generalized sidelobe cancelers are reformulations of LCMV that convert the constrained optimization into minimizing an unconstrained form and then compute the weights recursively. This approach

removes any need to invert a covariance matrix. See `phased.FrostBeamformer` and `phased.GSCBeamformer`.

# Conventional Beamforming

| **In this section...** |
| --- |
| "Uses for Beamformers" on page 5-6 |
| "Support for Conventional Beamforming" on page 5-6 |
| "Narrowband Phase Shift Beamformer for a ULA" on page 5-6 |

## Uses for Beamformers

You can use a beamformer to spatially filter the arriving signals. Accentuating or attenuating signals that arrive from specific directions helps you distinguish between signals of interest and interfering signals from other directions.

## Support for Conventional Beamforming

You can implement a narrowband phase shift beamformer using `phased.PhaseShiftBeamformer`. When you use this object, you must specify these aspects of the situation you are simulating:

- Sensor array
- Signal propagation speed
- System operating frequency
- Beamforming direction

For wideband beamformers, see "Wideband Beamforming" on page 5-15.

## Narrowband Phase Shift Beamformer for a ULA

This example shows how to create and beamform a 10-element ULA. Assume the carrier frequency is 1 GHz. Set the array element spacing to be one-half the carrier wavelength.

```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
array = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
```

By default, the ULA elements are isotropic antennas created by the `phased.IsotropicAntennaElement` System object™. Set the frequency range of the antenna elements so that the carrier frequency lies within the operating range.

```
array.Element.FrequencyRange = [8e8 1.2e9];
```

Simulate a test signal. For this example, use a simple rectangular pulse.

```
t = linspace(0,0.3,300)';
testsig = zeros(size(t));
testsig(201:205) = 1;
```

Assume the rectangular pulse is incident on the ULA from an angle of 30° azimuth and 0° elevation. Use the `collectPlaneWave` function of the ULA System object to simulate reception of the pulse waveform from the specified angle.

```
angle_of_arrival = [30;0];
x = collectPlaneWave(array,testsig,angle_of_arrival,fc);
```

The signal |x| is a matrix with ten columns. Each column represents the received signal at one of the array elements.

Add complex-valued Gaussian noise to the signal |x|. Reset the default random number stream for reproducible results. Plot the magnitudes of the received pulses at the first four elements of the ULA.

```
rng default
npower = 0.5;
x = x + sqrt(npower/2)*(randn(size(x)) + 1i*randn(size(x)));
subplot(221)
plot(t,abs(x(:,1)))
title('Element 1 (magnitude)')
axis tight
ylabel('Magnitude')
subplot(222)
plot(t,abs(x(:,2)))
title('Element 2 (magnitude)')
axis tight
ylabel('Magnitude')
subplot(223)
plot(t,abs(x(:,3)))
title('Element 3 (magnitude)')
axis tight
xlabel('Seconds')
ylabel('Magnitude')
subplot(224)
plot(t,abs(x(:,4)))
title('Element 4 (magnitude)')
axis tight
xlabel('Seconds')
ylabel('Magnitude')
```
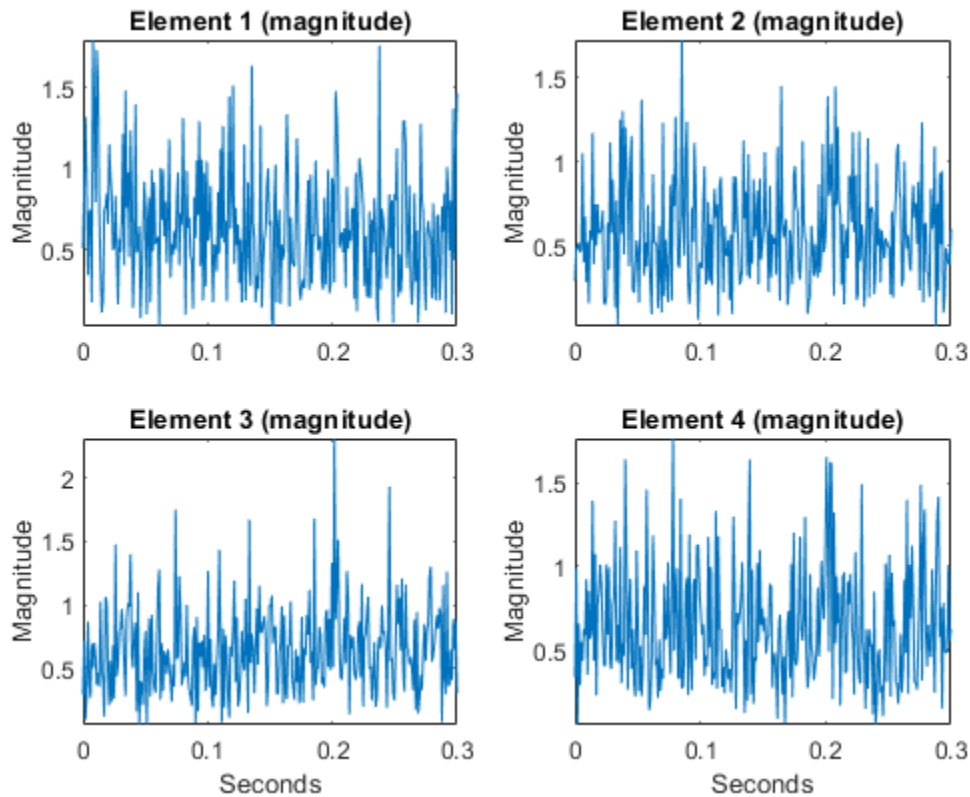
Construct a phase-shift beamformer. Set the `WeightsOutputPort` property to `true` to output the spatial filter weights that point the beamformer to the angle of arrival.

```
beamformer = phased.PhaseShiftBeamformer('SensorArray',array, ...
    'OperatingFrequency',1e9,'Direction',angle_of_arrival, ...
    'WeightsOutputPort',true);
```

Execute the phase shift beamformer to compute the beamformer output and to compute the applied weights.

```
[y,w] = beamformer(x);
```

Plot the magnitude of the output waveform along with the noise-free original waveform for comparison.

```
subplot(211)
plot(t,abs(testsig))
axis tight
title('Original Signal')
ylabel('Magnitude')
subplot(212)
plot(t,abs(y))
axis tight
title('Received Signal with Beamforming')
ylabel('Magnitude')
xlabel('Seconds')
```

**Original Signal**

**Received Signal with Beamforming**
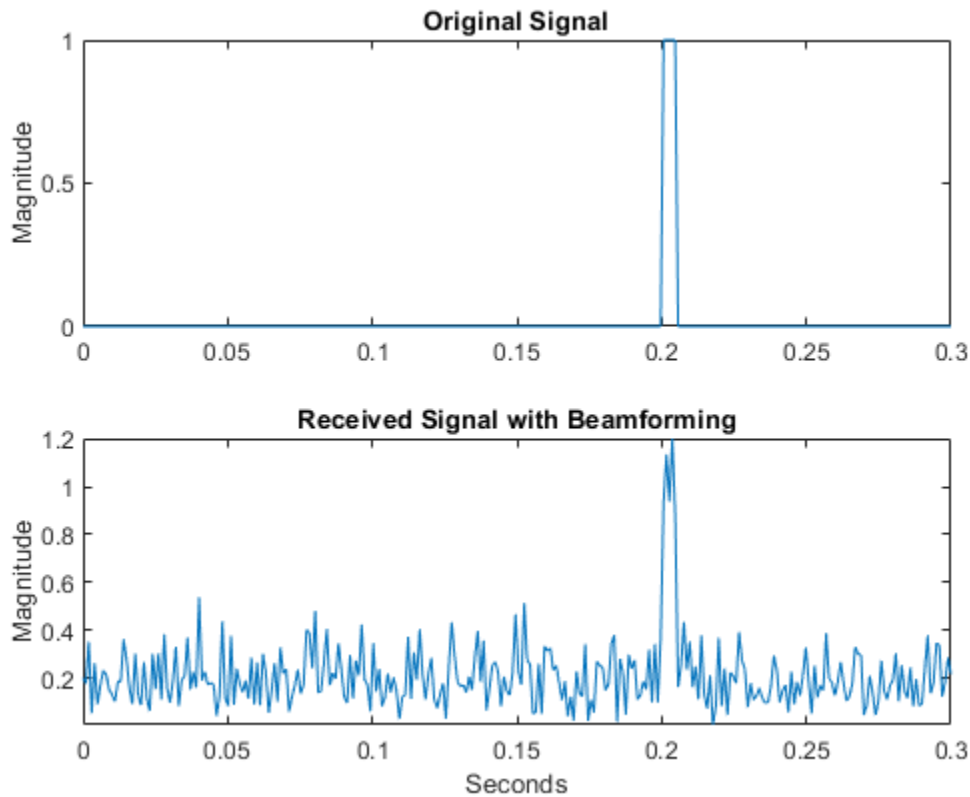
To examine the effect of beamforming weights on the array response, plot the array normalized power response with and without beamforming weights.

```
azang = -180:30:180;
subplot(211)
pattern(array,fc,[-180:180],0,'CoordinateSystem','rectangular',...
    'Type','powerdb','PropagationSpeed',physconst('LightSpeed'))
xticks(azang)
set(gca,'Fontsize',9)
title('Array Response without Beamforming Weights')
subplot(212)

pattern(array,fc,[-180:180],0,'CoordinateSystem','rectangular',...
    'Type','powerdb','PropagationSpeed',physconst('LightSpeed'),...
    'Weights',w)
xticks(azang)
set(gca,'Fontsize',9)
title('Array Response with Beamforming Weights')
```

Array Response without Beamforming Weights

Array Response with Beamforming Weights
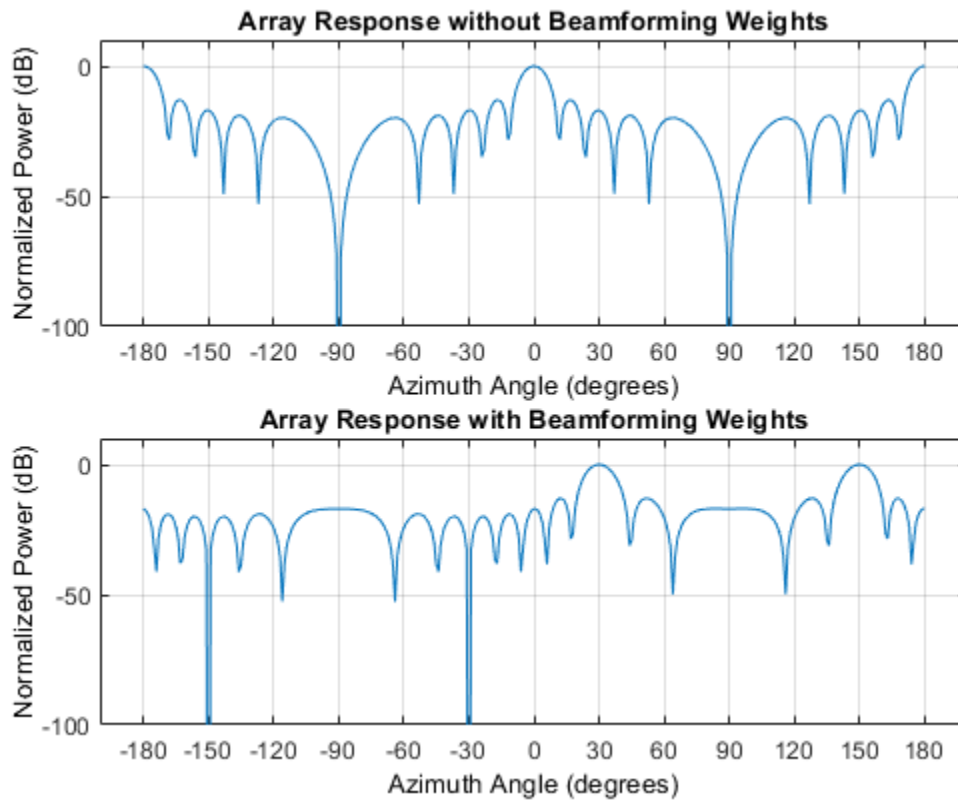
## See Also

## Related Examples

- "Conventional and Adaptive Beamformers" on page 18-201

# Adaptive Beamforming

## Benefits of Adaptive Beamforming

"Narrowband Phase Shift Beamformer for a ULA" on page 5-6 uses weights chosen independent of any data received by the array. The weights in the narrowband phase shift beamformer steer the array response in a specified direction. However, they do not account for any interference scenarios. As a result, these conventional beamformers are susceptible to interference signals. Such interference signals can be a particular problem if they occur at sidelobes of the array response.

By contrast, adaptive, or statistically optimum, beamformers can account for interference signals. An adaptive beamformer algorithm chooses the weights based on the statistics of the received data. For example, an adaptive beamformer can improve the SNR by using the received data to place nulls in the array response. These nulls are placed at angles corresponding to the interference signals.

## Support for Adaptive Beamforming

Phased Array System Toolbox software provides these adaptive beamformers:

- Linearly constrained minimum variance (LCMV) beamformers
- Minimum variance distortionless response (MVDR) beamformers
- Frost beamformers

## Nulling with LCMV Beamformer

This example shows how to use an LCMV beamformer to point a null of the array response in the direction of an interfering source. The array is a 10-element uniform linear array (ULA). By default, the ULA elements are isotropic antennas created by the `phased.IsotropicAntennaElement` System object™. Set the frequency range of the antenna elements so that the carrier frequency lies within the operating range. The carrier frequency is 1 GHz.

```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
array = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
array.Element.FrequencyRange = [8e8 1.2e9];
```

Simulate a test signal using a simple rectangular pulse.

```
t = linspace(0,0.3,300)';
testsig = zeros(size(t));
testsig(201:205) = 1;
```

Assume the rectangular pulse is incident on the ULA from an angle of 30° azimuth and 0° elevation. Use the `collectPlaneWave` function of the ULA System object to simulate reception of the pulse waveform from the incident angle.

```
angle_of_arrival = [30;0];
x = collectPlaneWave(array,testsig,angle_of_arrival,fc);
```

The signal x is a matrix with ten columns. Each column represents the received signal at one of the array elements.

Construct a conventional phase-shift beamformer. Set the WeightsOutputPort property to true to output the spatial filter weights.

```
convbeamformer = phased.PhaseShiftBeamformer('SensorArray',array,...
    'OperatingFrequency',1e9,'Direction',angle_of_arrival,...
    'WeightsOutputPort',true);
```

Add complex-valued white Gaussian noise to the signal x. Set the default random number stream for reproducible results.

```
rng default
npower = 0.5;
x = x + sqrt(npower/2)*(randn(size(x)) + 1i*randn(size(x)));
```

Create a 10W interference source. Specify the barrage jammer to have an effective radiated power of 10 W. The interference signal from the barrage jammer is incident on the ULA from an angle of 120° azimuth and 0° elevation. Use the collectPlaneWave function of the ULA System object to simulate reception of the jammer signal.

```
jamsig = sqrt(10)*randn(300,1);
jammer_angle = [120;0];
jamsig = collectPlaneWave(array,jamsig,jammer_angle,fc);
```

Add complex-valued white Gaussian noise to simulate noise contributions not directly associated with the jamming signal. Again, set the default random number stream for reproducible results. This noise power is 0 dB below the jammer power. Beamform the signal using a conventional beamformer.

```
noisePwr = 1e-5;
rng(2008);
noise = sqrt(noisePwr/2)*...
    (randn(size(jamsig)) + 1j*randn(size(jamsig)));
jamsig = jamsig + noise;
rxsig = x + jamsig;
[yout,w] = convbeamformer(rxsig);
```

Implement the adaptive LCMV beamformer using the same ULA array. Use the target-free data, jamsig, as training data. Output the beamformed signal and the beamformer weights.

```
steeringvector = phased.SteeringVector('SensorArray',array,...
    'PropagationSpeed',physconst('LightSpeed'));
LCMVbeamformer = phased.LCMVBeamformer('DesiredResponse',1,...
    'TrainingInputPort',true,'WeightsOutputPort',true);
LCMVbeamformer.Constraint = steeringvector(fc,angle_of_arrival);
LCMVbeamformer.DesiredResponse = 1;
[yLCMV,wLCMV] = LCMVbeamformer(rxsig,jamsig);
```

Plot the conventional beamformer output and the adaptive beamformer output.

```
subplot(211)
plot(t,abs(yout))
axis tight
title('Conventional Beamformer')
```

```
ylabel('Magnitude')
subplot(212)
plot(t,abs(yLCMV))
axis tight
title('LCMV (Adaptive) Beamformer')
xlabel('Seconds')
ylabel('Magnitude')
```

**Conventional Beamformer**

**LCMV (Adaptive) Beamformer**
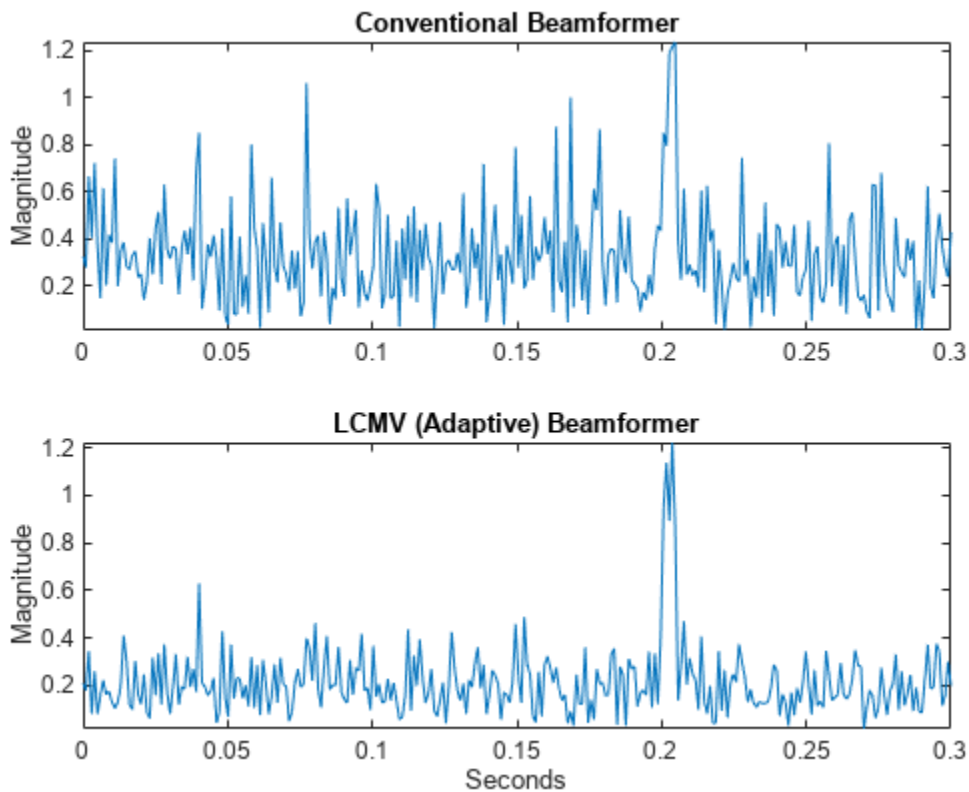
The adaptive beamformer significantly improves the SNR of the rectangular pulse at 0.2 s.

Using conventional and LCMV weights, plot the responses for each beamformer.

```
subplot(211)
pattern(array,fc,[-180:180],0,'PropagationSpeed',physconst('LightSpeed'),...
    'CoordinateSystem','rectangular','Type','powerdb','Normalize',true,...
    'Weights',w)
title('Array Response with Conventional Beamforming Weights');
subplot(212)
pattern(array,fc,[-180:180],0,'PropagationSpeed',physconst('LightSpeed'),...)
    'CoordinateSystem','rectangular','Type','powerdb','Normalize',true,...
    'Weights',wLCMV)
title('Array Response with LCMV Beamforming Weights');
```
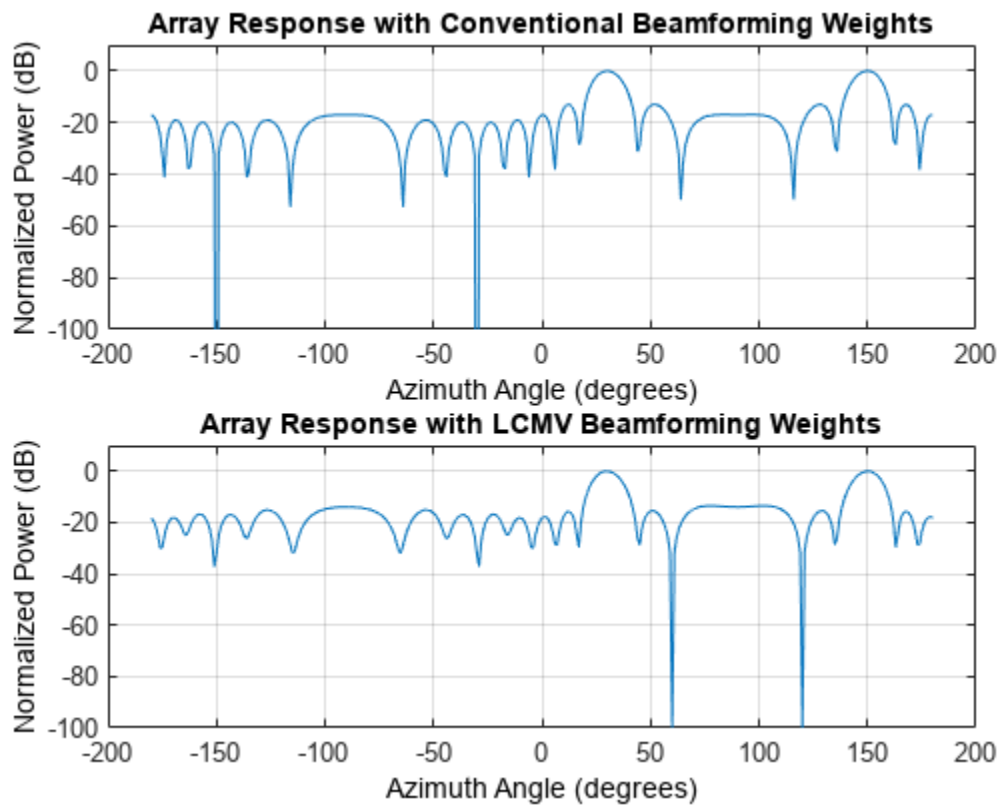
The adaptive beamform places a null at the arrival angle of the interference signal, 120°.

## See Also

phased.FrostBeamformer | phased.LCMVBeamformer | phased.MVDRBeamformer

## Related Examples

- "Conventional and Adaptive Beamformers" on page 18-201

# Wideband Beamforming

| In this section... |
| --- |
| "Support for Wideband Beamforming" on page 5-15 |
| "Time-Delay Beamforming of Microphone ULA Array" on page 5-15 |
| "Visualization of Wideband Beamformer Performance" on page 5-16 |

## Support for Wideband Beamforming

Beamforming achieved by multiplying the sensor input by a complex exponential with the appropriate phase shift only applies for narrowband signals. In the case of wideband, or broadband, signals, the steering vector is not a function of a single frequency. Wideband processing is commonly used in microphone and acoustic applications.

Phased Array System Toolbox software provides conventional and adaptive wideband beamformers. They include:

- phased.FrostBeamformer
- phased.SubbandPhaseShiftBeamformer
- phased.TimeDelayBeamformer
- phased.TimeDelayLCMVBeamformer

See "Acoustic Beamforming Using a Microphone Array" on page 18-185 for an example of using wideband beamforming to extract speech signals in noise.

## Time-Delay Beamforming of Microphone ULA Array

This example shows how to perform wideband conventional time-delay beamforming with a microphone array of omnidirectional elements. Create an acoustic (pressure wave) chirp signal. The chirp signal has a bandwidth of 1 kHz and propagates at a speed of 340 m/s at ground level.

```
c = 340;
t = linspace(0,1,50e3)';
sig = chirp(t,0,1,1000);
```

Collect the acoustic chirp with a ten-element ULA. Use omnidirectional microphone elements spaced less than one-half the wavelength at the 50 kHz sampling frequency. The chirp is incident on the ULA with an angle of 60° azimuth and 0° elevation. Add random noise to the signal.

```
microphone = phased.OmnidirectionalMicrophoneElement(...
    'FrequencyRange',[20 20e3]);
array = phased.ULA('Element',microphone,'NumElements',10,...
    'ElementSpacing',0.01);
collector = phased.WidebandCollector('Sensor',array,'SampleRate',5e4,...
    'PropagationSpeed',c,'ModulatedInput',false);
sigang = [60;0];
rsig = collector(sig,sigang);
rsig = rsig + 0.1*randn(size(rsig));
```

Apply a wideband conventional time-delay beamformer to improve the SNR of the received signal.

```
beamformer = phased.TimeDelayBeamformer('SensorArray',array,...
    'SampleRate',5e4,'PropagationSpeed',c,'Direction',sigang);
y = beamformer(rsig);

subplot(2,1,1)
plot(t(1:5000),real(rsig(1:5e3,5)))
axis([0,t(5000),-0.5,1])
title('Signal (real part) at the 5th element of the ULA')
subplot(2,1,2)
plot(t(1:5000),real(y(1:5e3)))
axis([0,t(5000),-0.5,1])
title('Signal (real part) with time-delay beamforming')
xlabel('Seconds')
```



## Visualization of Wideband Beamformer Performance

This example shows how to plot the response of an acoustic microphone element and an array of microphone elements to validate the performance of a beamformer. The array must maintain an acceptable array pattern throughout the bandwidth.

Create an 11-element uniform linear array (ULA) of microphones using cosine antenna elements as microphones. The `phased.CosineAntennaElement` System object™ is general enough to be used as a microphone element as well because it creates or receives a scalar field. You need to change the

response frequencies to the audible range. In addition make sure the `PropagationSpeed` parameter in the array `pattern` methods are set to the speed of sound in air.

```
c = 340;
freq = [1000 2750];
fc = 2000;
numels = 11;
microphone = phased.CosineAntennaElement('FrequencyRange',freq);
array = phased.ULA('NumElements',numels,...
    'ElementSpacing',0.5*c/fc,'Element',microphone);
```

Plot the response pattern of the microphone element over a set of frequencies.

```
plotFreq = linspace(min(freq),max(freq),15);
pattern(microphone,plotFreq,[-180:180],0,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb')
```



This plot shows that the element pattern is constant over the entire bandwidth.

Plot the response pattern of an 11-element array over the same set of frequencies.

```
pattern(array,plotFreq,[-180:180],0,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb','PropagationSpeed',c)
```

This plot shows that the element pattern mainlobe decreases with frequency.

Apply a subband phase shift beamformer to the array. The direction of interest is 30° azimuth and 0° elevation. There are 8 subbands.

```
direction = [30;0];
numbands = 8;
beamformer = phased.SubbandPhaseShiftBeamformer('SensorArray',array,...
    'Direction',direction,...
    'OperatingFrequency',fc,'PropagationSpeed',c,...
    'SampleRate',1e3,...
    'WeightsOutputPort',true,'SubbandsOutputPort',true,...
    'NumSubbands',numbands);
rx = ones(numbands,numels);
[y,w,centerfreqs] = beamformer(rx);
```

Plot the response pattern of the array using the weights and center frequencies from the beamformer.

```
pattern(array,centerfreqs.',[-180:180],0,'Weights',w,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb','PropagationSpeed',c)
```

**Azimuth Cut (elevation angle = 0.0°)**



The above plot shows the beamformed pattern at the center frequency of each subband.

Plot the response pattern at three frequencies in two-dimensions.

```
centerfreqs = fftshift(centerfreqs);
w = fftshift(w,2);
idx = [1,5,8];
pattern(array,centerfreqs(idx).',[-180:180],0,'Weights',w(:,idx),'CoordinateSystem','rectangular
    'PlotStyle','overlay','Type','powerdb','PropagationSpeed',c)
legend('Location','South')
```

This plot shows that the main beam direction remains constant while the beamwidth decreases with frequency.

# Time-Delay Beamforming of Microphone ULA Array

This example shows how to perform wideband conventional time-delay beamforming with a microphone array of omnidirectional elements. Create an acoustic (pressure wave) chirp signal. The chirp signal has a bandwidth of 1 kHz and propagates at a speed of 340 m/s at ground level.

```
c = 340;
t = linspace(0,1,50e3)';
sig = chirp(t,0,1,1000);
```

Collect the acoustic chirp with a ten-element ULA. Use omnidirectional microphone elements spaced less than one-half the wavelength at the 50 kHz sampling frequency. The chirp is incident on the ULA with an angle of 60° azimuth and 0° elevation. Add random noise to the signal.

```
microphone = phased.OmnidirectionalMicrophoneElement(...
    'FrequencyRange',[20 20e3]);
array = phased.ULA('Element',microphone,'NumElements',10,...
    'ElementSpacing',0.01);
collector = phased.WidebandCollector('Sensor',array,'SampleRate',5e4,...
    'PropagationSpeed',c,'ModulatedInput',false);
sigang = [60;0];
rsig = collector(sig,sigang);
rsig = rsig + 0.1*randn(size(rsig));
```

Apply a wideband conventional time-delay beamformer to improve the SNR of the received signal.

```
beamformer = phased.TimeDelayBeamformer('SensorArray',array,...
    'SampleRate',5e4,'PropagationSpeed',c,'Direction',sigang);
y = beamformer(rsig);

subplot(2,1,1)
plot(t(1:5000),real(rsig(1:5e3,5)))
axis([0,t(5000),-0.5,1])
title('Signal (real part) at the 5th element of the ULA')
subplot(2,1,2)
plot(t(1:5000),real(y(1:5e3)))
axis([0,t(5000),-0.5,1])
title('Signal (real part) with time-delay beamforming')
xlabel('Seconds')
```

# Visualization of Wideband Beamformer Performance

This example shows how to plot the response of an acoustic microphone element and an array of microphone elements to validate the performance of a beamformer. The array must maintain an acceptable array pattern throughout the bandwidth.

Create an 11-element uniform linear array (ULA) of microphones using cosine antenna elements as microphones. The `phased.CosineAntennaElement` System object™ is general enough to be used as a microphone element as well because it creates or receives a scalar field. You need to change the response frequencies to the audible range. In addition make sure the `PropagationSpeed` parameter in the array `pattern` methods are set to the speed of sound in air.

```
c = 340;
freq = [1000 2750];
fc = 2000;
numels = 11;
microphone = phased.CosineAntennaElement('FrequencyRange',freq);
array = phased.ULA('NumElements',numels,...
    'ElementSpacing',0.5*c/fc,'Element',microphone);
```

Plot the response pattern of the microphone element over a set of frequencies.

```
plotFreq = linspace(min(freq),max(freq),15);
pattern(microphone,plotFreq,[-180:180],0,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb')
```

This plot shows that the element pattern is constant over the entire bandwidth.

Plot the response pattern of an 11-element array over the same set of frequencies.

```
pattern(array,plotFreq,[-180:180],0,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb','PropagationSpeed',c)
```



**Azimuth Cut (elevation angle = 0.0°)**

This plot shows that the element pattern mainlobe decreases with frequency.

Apply a subband phase shift beamformer to the array. The direction of interest is 30° azimuth and 0° elevation. There are 8 subbands.

```
direction = [30;0];
numbands = 8;
beamformer = phased.SubbandPhaseShiftBeamformer('SensorArray',array,...
    'Direction',direction,...
    'OperatingFrequency',fc,'PropagationSpeed',c,...
    'SampleRate',1e3,...
    'WeightsOutputPort',true,'SubbandsOutputPort',true,...
    'NumSubbands',numbands);
rx = ones(numbands,numels);
[y,w,centerfreqs] = beamformer(rx);
```

Plot the response pattern of the array using the weights and center frequencies from the beamformer.

```
pattern(array,centerfreqs.',[-180:180],0,'Weights',w,'CoordinateSystem','rectangular',...
    'PlotStyle','waterfall','Type','powerdb','PropagationSpeed',c)
```

The above plot shows the beamformed pattern at the center frequency of each subband.

Plot the response pattern at three frequencies in two-dimensions.

```
centerfreqs = fftshift(centerfreqs);
w = fftshift(w,2);
idx = [1,5,8];
pattern(array,centerfreqs(idx).',[-180:180],0,'Weights',w(:,idx),'CoordinateSystem','rectangular
    'PlotStyle','overlay','Type','powerdb','PropagationSpeed',c)
legend('Location','South')
```
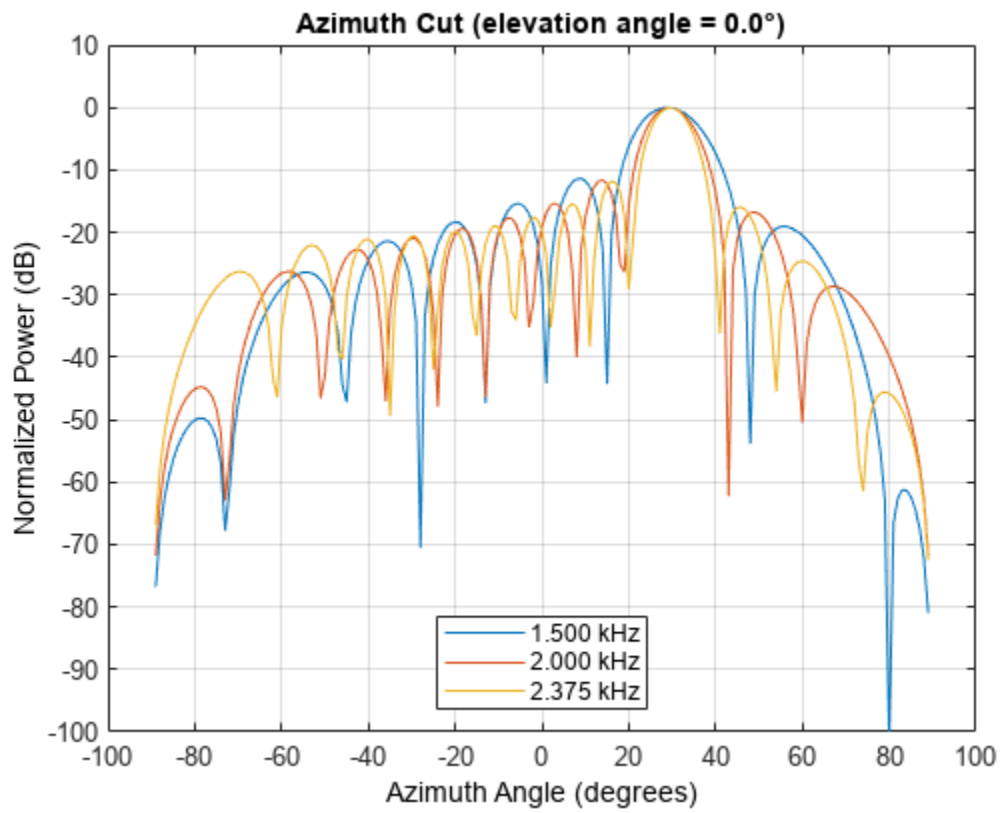
This plot shows that the main beam direction remains constant while the beamwidth decreases with frequency.

# FPGA-Based Minimum-Variance Distortionless-Response (MVDR) Beamformer

This example shows how to implement a minimum-variance distortionless-response (MVDR) beamformer suitable for hardware.

For more information on beamformers, see "Conventional and Adaptive Beamformers" on page 18-272.

**MVDR Objective**

The MVDR beamformer preserves the gain in the direction of arrival of a desired signal and attenuates interference from other directions [1], [2].

Given readings from a sensor array, such as the uniform linear array (ULA) in the following diagram, form data matrix $A$ from samples of the array, where $a(t)$ is an $n$-by-1 column vector of readings from the array sampled at time $t$, and $a(t)^H$ is one row of matrix $A$. Many more samples are taken than there are elements in the array. This results in the number of rows in $A$ being much greater than the number of columns. An estimate of the covariance matrix is $A^H A$, where $A^H$ is the Hermitian or complex-conjugate transpose of $A$.
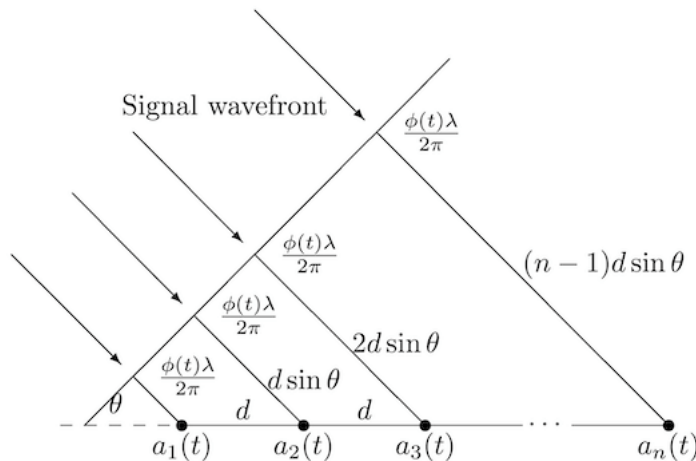


Compute the MVDR beamformer response by solving the following equation for $x$, where $b$ is a steering vector pointing in the direction of the desired signal.

$$(A^H A)x = b$$

The MVDR weight vector $w$ is computed from $x$ and $b$ using the following equation, which normalizes $x$ to preserve the gain in the direction of arrival of the desired signal.

$$w = \frac{x}{b^H x}$$

The MVDR system response is the inner product between the MVDR weight vector $w$ and a current sample from the sensor array $a(t)$.

$$y = w^H a(t)$$

**HDL-Optimized MVDR**

The three equations in the previous section are implemented by the three primary blocks in the following model. The rate changes give the matrix solve additional clock cycles to update before the next input sample. The number of clock cycles between a valid input and when the complex matrix solve block is ready is twice its input wordlength to allow time for CORDIC iterations, plus 15 cycles for internal delays.

```
load_system('MVDRBeamformerHDLOptimizedModel');
open_system('MVDRBeamformerHDLOptimizedModel/MVDR - HDL Optimized')
```
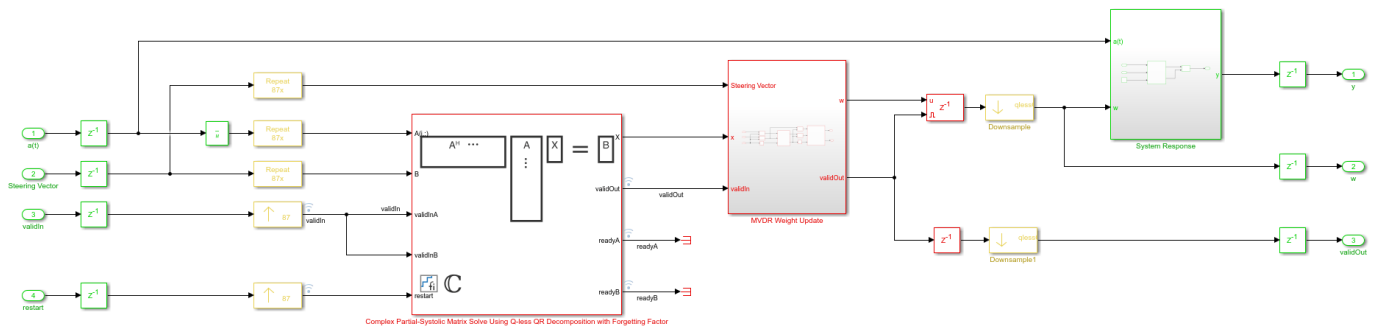


Instead of forming data matrix $A$ and computing the Cholesky factorization of covariance matrix $A^H A$, the upper-triangular matrix of the QR decomposition of $A$ is computed directly and updated as each data vector $a(t)$ streams in from the sensor array. Because the data is updated indefinitely, a forgetting factor is applied after each factorization. To integrate with an equivalent of a matrix of $m$ rows, the forgetting factor $\alpha$ should be set to

$$\alpha = \exp(-1/(2m)).$$

This example simulates the equivalent of a matrix with $m = 300$ rows, so the forgetting factor is set to 0.9983.

The **Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor** block is implemented using the method found in [3]. The upper-triangular matrix $R$ from the QR decomposition of $A$ is identical to the Cholesky factorization of $A^H A$ except the signs of values on the diagonal. Solving the matrix equation $(A^H A)x = b$ by computing the Cholesky factorization of $A^H A$ is not as efficient or as numerically sound as computing the QR decomposition of $A$ directly [4].

**Run Model**

Open and simulate the model.

```
open_system('MVDRBeamformerHDLOptimizedModel')
scopes = find_system('MVDRBeamformerHDLOptimizedModel','BlockType','Scope');
close_system(scopes);
```
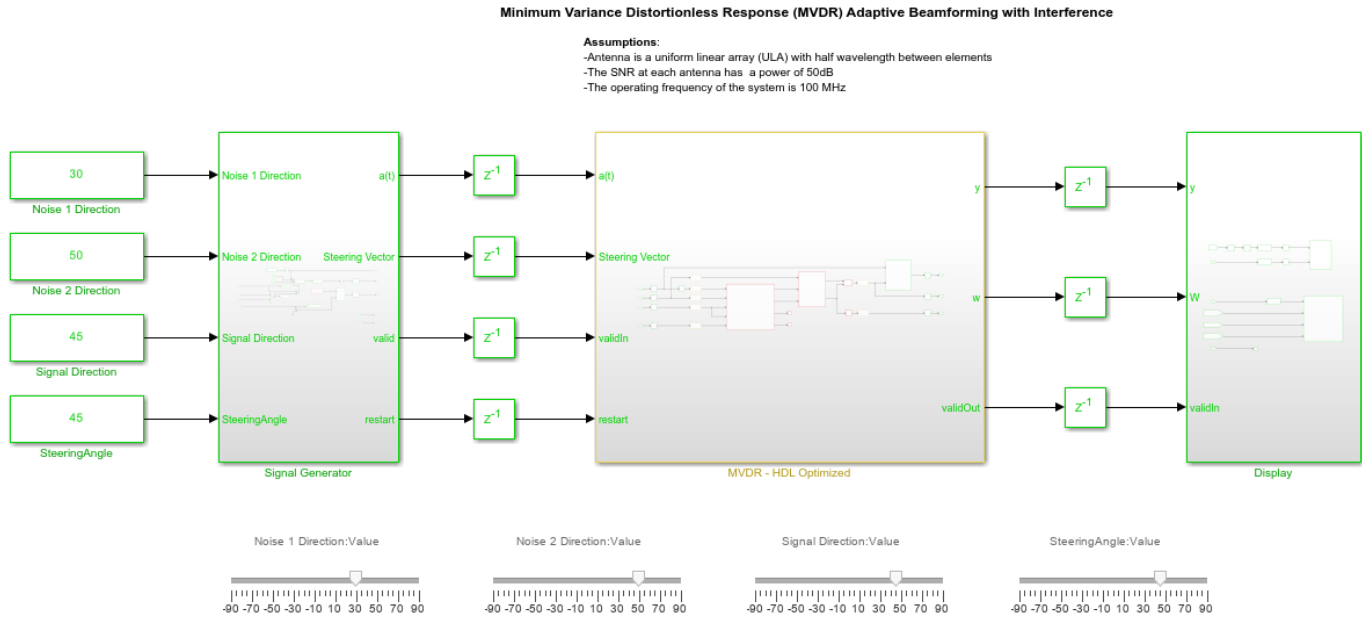
As the model is simulating, you can adjust the signal direction, steering angle and noise directions by dragging the sliders, or by editing the constant values.

When the signal direction and steering angle are aligned as indicated by the blue and green lines, you can see that the beam pattern has a gain of 0 dB. The noise sources are nulled as indicated by the red lines.
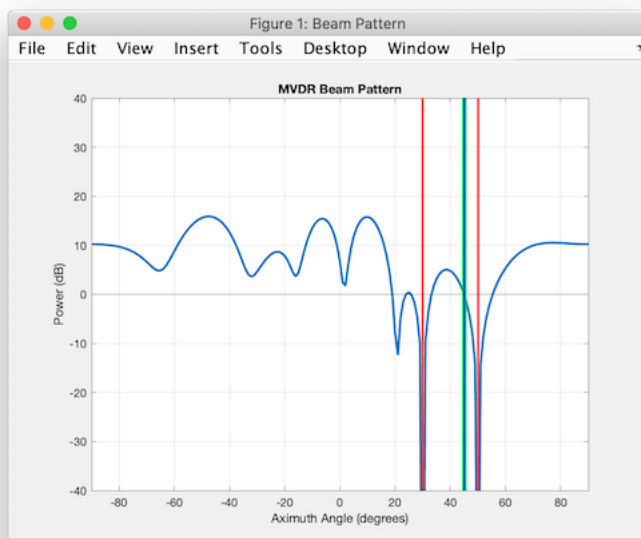
The desired pulse appears when the noise sources are nulled. This example simulates with the same latency as the hardware, so you can see the signal settle over time as the simulation starts and when the directions are changed.
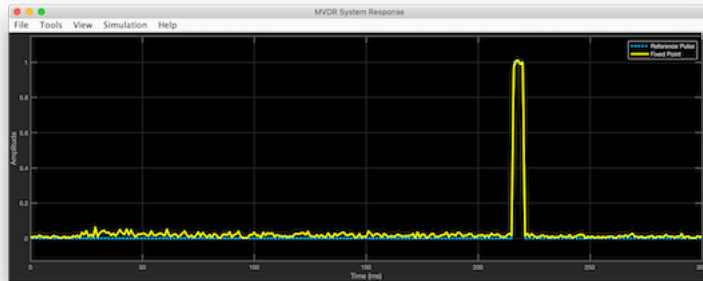


**Set Parameters**

The parameters for the beamformer are set in the model workspace. You can modify the parameters by editing and running the `setMVDRExampleModelWorkspace` function.

**References**

[1] V. Behar et al. "Parameter Optimization of the adaptive MVDR QR-based beamformer for jamming and multipath supression in GPS/GLONASS receivers". In: Proc. 16th Saint Petersburg International Conference on Integrated Navigation systems. Saint Petersburg, Russia, May 2009, pp. 325--334.

[2] Jack Capon. "High-resolution frequency-wavenumber spectrum analysis". In: vol. 57. 1969, pp. 1408--1418.

[3] C.M. Rader. "VLSI Systolic Arrays for Adaptive Nulling". In: IEEE Signal Processing Magazine (July 1996), pp. 29--49.

[4] Charles F. Van Loan. Introduction to Scientific Computing: A Matrix-Vector Approach Using Matlab. Second edition. Prentice-Hall, 2000. isbn: 0-13-949157-0.

**6**

# Direction-of-Arrival Estimation

# Beamscan Direction-of-Arrival Estimation

This example shows how to use the nonparametric beamscan technique to estimate the directions of arrival (DOA) of two signals. The beamscan algorithm estimates the DOAs by scanning the array beam over a region of interest. The algorithm computes the output power for each beamscan angle and identifies the maxima as the DOA estimates.

Construct a uniform linear array (ULA) consisting of ten isotropic antenna elements. The carrier frequency of the incoming narrowband sources is 1 GHz.

```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
antenna = phased.IsotropicAntennaElement('FrequencyRange',[8e8 1.2e9]);
array = phased.ULA('Element',antenna,'NumElements',10,'ElementSpacing',lambda/2);
```

The incident wavefield consists of linear FM pulses from two sources. The DOAs of the two sources are 30° azimuth and 60° azimuth. Both sources have elevation angles of 0°.

```
waveform = phased.LinearFMWaveform('SweepBandwidth',1e5,...
    'PulseWidth',5e-6,'OutputFormat','Pulses','NumPulses',1);
sig1 = waveform();
sig2 = sig1;
ang1 = [30; 0];
ang2 = [60;0];
arraysig = collectPlaneWave(array,[sig1 sig2],[ang1 ang2],fc);
rng default
npower = 0.01;
noise = sqrt(npower/2)*...
    (randn(size(arraysig)) + 1i*randn(size(arraysig)));
rxsig = arraysig + noise;
```

Implement a beamscan DOA estimator. Scan the azimuth angles from −90° to 90°. Output the DOA estimates, and plot the spatial spectrum. The locations of the two largest peaks of the spectrum identify the DOAs of the signals.

```
estimator = phased.BeamscanEstimator('SensorArray',array,...
    'OperatingFrequency',fc,'ScanAngles',-90:90,...
    'DOAOutputPort',true,'NumSignals',2);
[y,sigang] = estimator(rxsig);
disp(sigang)
```

```
    64      28
```

Plot the spatial spectrum as a function of broadside angle.

```
plotSpectrum(estimator)
```

Beamscan Spatial Spectrum
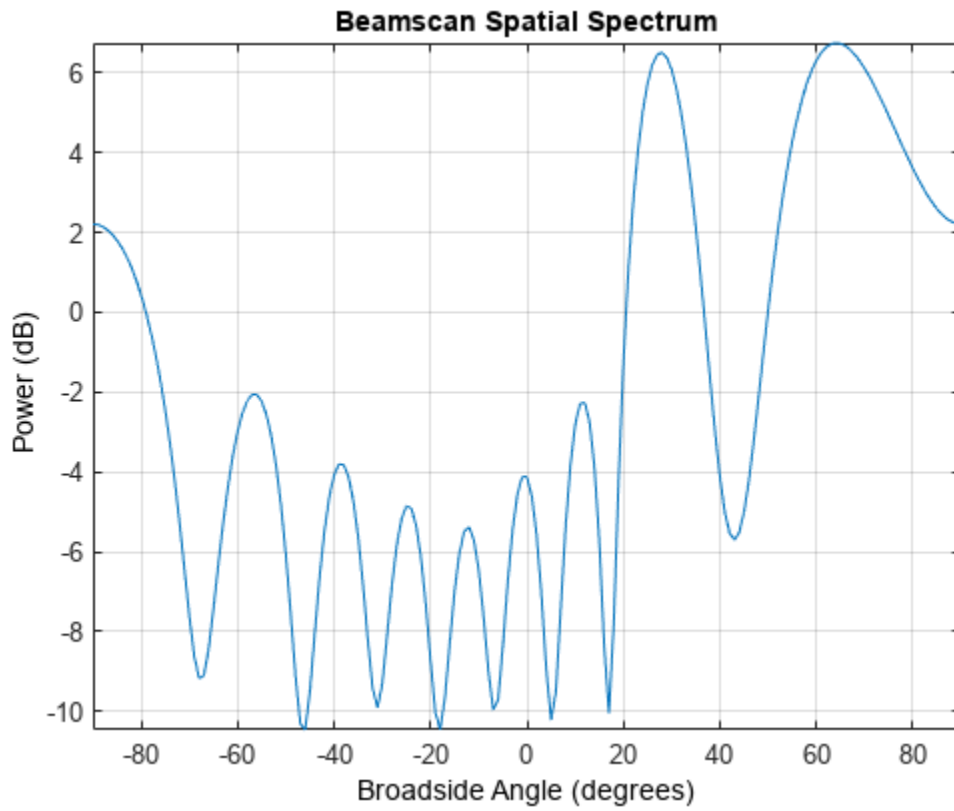
## See Also

## Related Examples

- "Super-Resolution DOA Estimation" on page 6-4

# Super-Resolution DOA Estimation

This example shows how to estimate angles of arrival from two separate signal sources when both angles fall within the main lobe of the array response a uniform linear array (ULA). In this case, a beamscan DOA estimator cannot resolve the two sources. However, a super-resolution DOA estimator using the root MUSIC algorithm is able to do so.

Plot the array response of the ULA. Zoom in on the main lobe.

```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
array = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
array.Element.FrequencyRange = [8e8 1.2e9];
plotResponse(array,fc,physconst('LightSpeed'))
axis([-25 25 -30 0]);
```
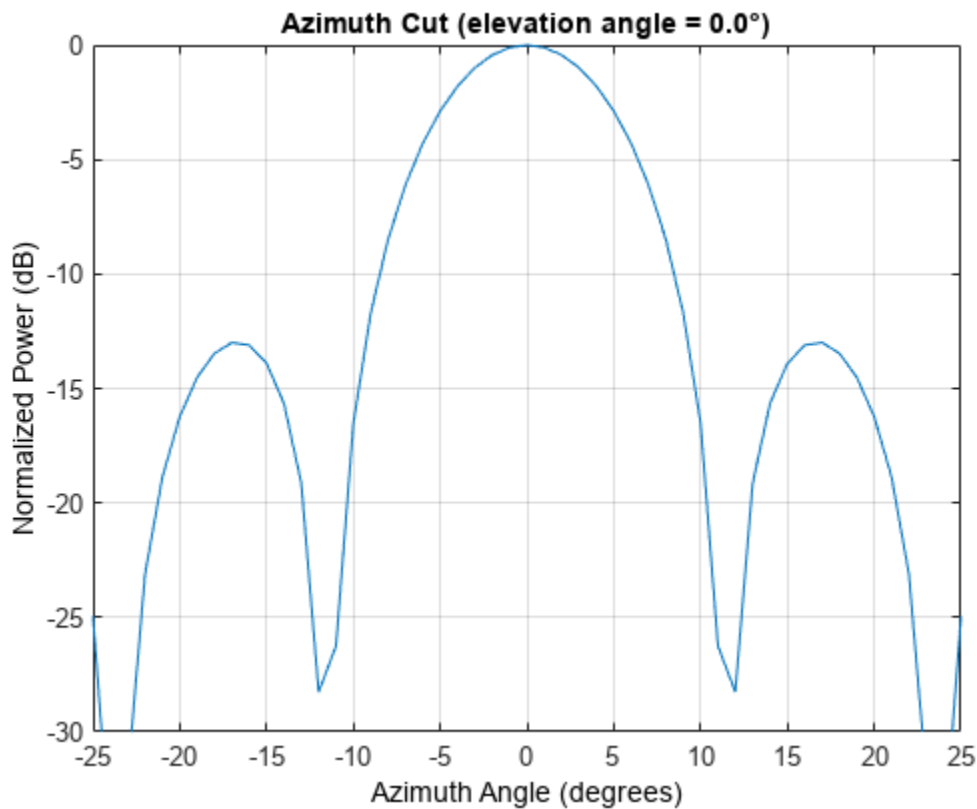


Receive two signal sources with DOAs separated by 10°.

```
ang1 = [30; 0];
ang2 = [40; 0];
Nsnapshots = 1000;
rng default
npower = 0.01;
rxsig = sensorsig(getElementPosition(array)/lambda,...
    Nsnapshots,[ang1 ang2],npower);
```

Estimate the directions of arrival using the beamscan estimator. Because both DOAs fall inside the main lobe of the array response, the beamscan DOA estimator cannot resolve them as separate sources.

```
beamscanestimator = phased.BeamscanEstimator('SensorArray',array,...
    'OperatingFrequency',fc,'ScanAngles',-90:90,...
    'DOAOutputPort',true,'NumSignals',2);
[~,sigang] = beamscanestimator(rxsig);
plotSpectrum(beamscanestimator)
```
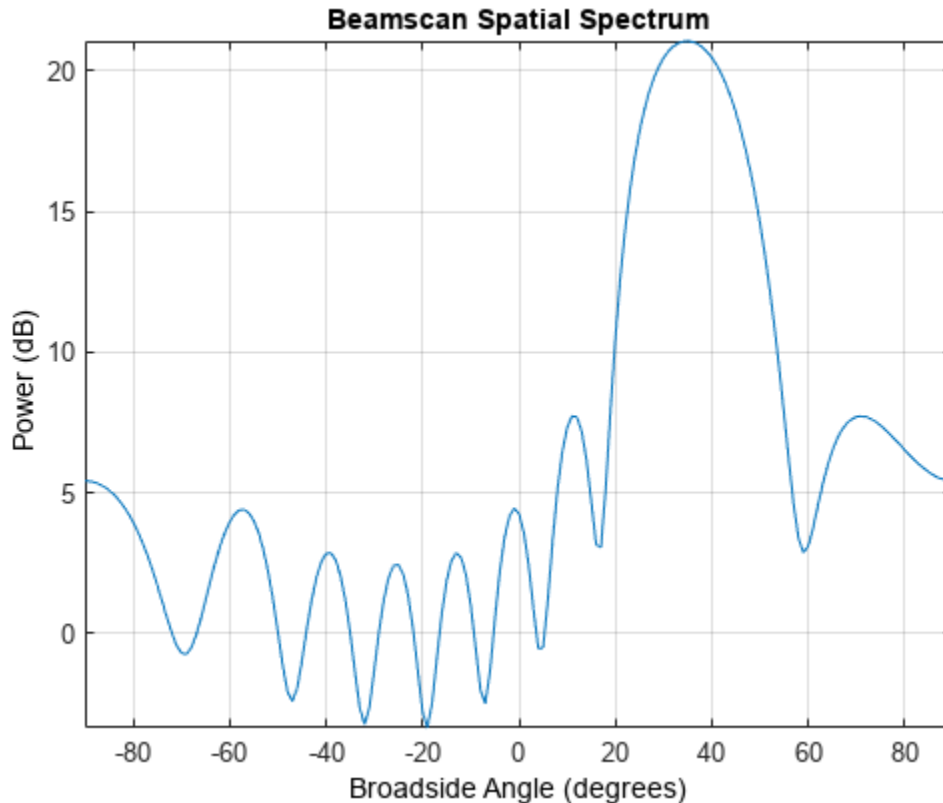


Use the super-resolution DOA estimator to estimate the two directions. This estimator offers better resolution than the nonparametric beamscan estimator.

```
MUSICestimator = phased.RootMUSICEstimator('SensorArray',array,...
    'OperatingFrequency',fc,'NumSignalsSource','Property',...
    'NumSignals',2,'ForwardBackwardAveraging',true);
doa_est = MUSICestimator(rxsig)
```

doa_est = *1×2*

   40.0091   30.0048

This estimator correctly identifies the two distinct directions of arrival.

## See Also

`phased.RootMUSICEstimator`

## Related Examples

# MUSIC Super-Resolution DOA Estimation

| In this section... |
| --- |
| "Signal Model" on page 6-7 |
| "Signal and Noise Subspaces" on page 6-8 |
| "Root-MUSIC" on page 6-9 |
| "Spatial Smoothing of Correlated Sources" on page 6-9 |

MUltiple SIgnal Classification (MUSIC) is a high-resolution direction-finding algorithm based on the eigenvalue decomposition of the sensor covariance matrix observed at an array. MUSIC belongs to the family of subspace-based direction-finding algorithms.

## Signal Model

The signal model relates the received sensor data to the signals emitted by the source. Assume that there are $D$ uncorrelated or partially correlated signal sources, $s_d(t)$. The sensor data, $x_m(t)$, consists of the signals, as received at the array, together with noise, $n_m(t)$. A sensor data snapshot is the sensor data vector received at the $M$ elements of an array at a single time $t$.

$$x(t) = As(t) + n(t)$$
$$s(t) = [s_1(t), s_2(t), ..., s_D(t))]'$$
$$A = [a(\theta_1) | a(\theta_2) | ... | a(\theta_D)]$$

- $x(t)$ is an $M$-by-1 vector of received snapshot of sensor data which consist of signals and additive noise.

- $A$ is an $M$-by-$D$ matrix containing the arrival vectors. An arrival vector consists of the relative phase shifts at the array elements of the plane wave from one source. Each column of $A$ represents the arrival vector from one of the sources and depends on the direction of arrival, $\theta_d$. $\theta_d$ is the direction of arrival angle for the $d$th source and can represents either the broadside angle for linear arrays or the azimuth and elevation angle for planar or 3D arrays.

- $s(t)$ is a $D$-by-1 vector of source signal values from $D$ sources.

- $n(t)$ is an $M$-by-1 vector of sensor noise values.

An important quantity in any subspace method is the sensor covariance matrix, $R_x$, derived from the received signal data. When the signals are uncorrelated with the noise, the sensor covariance matrix has two components, the signal covariance matrix and the noise covariance matrix.

$$R_x = E\left\{xx^H\right\} = AR_sA^H + \sigma_n^2 I$$

where $R_s$ is the source covariance matrix. The diagonal elements of the source covariance matrix represent source power and the off-diagonal elements represent source correlations.

$$R_s = E\left\{ss^H\right\}$$

For uncorrelated sources or even partially correlated sources, $R_s$ is a positive-definite Hermitian matrix and has full rank, $D$, equal to the number of sources.

The signal covariance matrix, $AR_sA^H$, is an $M$-by-$M$ matrix, also with rank $D < M$.

An assumption of the MUSIC algorithm is that the noise powers are equal at all sensors and uncorrelated between sensors. With this assumption, the noise covariance matrix becomes an $M$-by-$M$ diagonal matrix with equal values along the diagonal.

Because the true sensor covariance matrix is not known, MUSIC estimates the sensor covariance matrix, $R_x$, from the *sample* sensor covariance matrix. The sample sensor covariance matrix is an average of multiple snapshots of the sensor data

$$R_x = \frac{1}{T} \sum_{k=1}^{T} x(t)x(t)^H,$$

where $T$ is the number of snapshots.

## Signal and Noise Subspaces

Because $AR_sA^H$ has rank $D$, it has $D$ positive real eigenvalues and $M - D$ zero eigenvalues. The eigenvectors corresponding to the positive eigenvalues span the signal subspace, $U_s = [v_1,...,v_D]$. The eigenvectors corresponding to the zero eigenvalues are orthogonal to the signal space and span the null subspace, $U_n = [u_{D+1},...,u_N]$. The arrival vectors also belong to the signal subspace, but they are eigenvectors. Eigenvectors of the null subspace are orthogonal to the eigenvectors of the signal subspace. Null-subspace eigenvectors, $u_i$, satisfy this equation:

$$AR_sA^Hu_i = 0 \Rightarrow u^HAR_sA^Hu_i = 0 \Rightarrow \left(A^Hu_i\right)^H R_s\left(A^Hu_i\right) = 0 \Rightarrow A^Hu_i = 0$$

Therefore the arrival vectors are orthogonal to the null subspace.

When noise is added, the eigenvectors of the sensor covariance matrix with noise present are the same as the noise-free sensor covariance matrix. The eigenvalues increase by the noise power. Let $v_i$ be one of the original noise-free signal space eigenvectors. Then

$$R_xv_i = AR_sA^Hv_i + \sigma_0^2Iv_i = \left(\lambda_i + \sigma_0^2\right)v_i$$

shows that the signal space eigenvalues increase by $\sigma_0^2$.

The null subspace eigenvectors are also eigenvectors of $R_x$. Let $u_i$ be one of the null eigenvectors. Then

$$R_xu_i = AR_sA^Hu_i + \sigma_0^2Iu_i = \sigma_0^2u_i$$

with eigenvalues of $\sigma_0^2$ instead of zero. The null subspace becomes the noise subspace.

MUSIC works by searching for all arrival vectors that are orthogonal to the noise subspace. To do the search, MUSIC constructs an arrival-angle-dependent power expression, called the MUSIC pseudospectrum:

$$P_{MUSIC}(\vec{\phi}) = \frac{1}{a^H(\vec{\phi})U_nU_n^Ha(\vec{\phi})}$$

When an arrival vector is orthogonal to the noise subspace, the peaks of the pseudospectrum are infinite. In practice, because there is noise, and because the true covariance matrix is estimated by the sampled covariance matrix, the arrival vectors are never exactly orthogonal to the noise subspace. Then, the angles at which $P_{MUSIC}$ has finite peaks are the desired directions of arrival.

Because the pseudospectrum can have more peaks than there are sources, the algorithm requires that you specify the number of sources, $D$, as a parameter. Then the algorithm picks the $D$ largest peaks. For a uniform linear array (ULA), the search space is a one-dimensional grid of broadside angles. For planar and 3D arrays, the search space is a two-dimensional grid of azimuth and elevation angles.

## Root-MUSIC

For a ULA, the denominator in the pseudospectrum is a polynomial in $e^{ikd\cos\varphi}$, but can also be considered a polynomial in the complex plane. In this cases, you can use root-finding methods to solve for the roots, $z_i$. These roots do not necessarily lie on the unit circle. However, Root-MUSIC assumes that the $D$ roots closest to the unit circle correspond to the true source directions. Then you can compute the source directions from the phase of the complex roots.

## Spatial Smoothing of Correlated Sources

When some of the $D$ source signals are correlated, $R_s$ is rank deficient, meaning that it has fewer than $D$ nonzero eigenvalues. Therefore, the number of zero eigenvalues of $AR_sA^H$ exceeds the number, $M - D$, of zero eigenvalues for the uncorrelated source case. MUSIC performance degrades when signals are correlated, as occurs in a multipath propagation environment. A way to compensate for correlation is to use spatial smoothing.

Spatial smoothing takes advantage of the translation properties of a uniform array. Consider two correlated signals arriving at an $L$-element ULA. The source covariance matrix, $R_s$ is a singular 2-by-2 matrix. The arrival vector matrix is an $L$-by-2 matrix

$$A_1 = \begin{bmatrix} 1 & 1 \\ e^{ikd\cos\varphi_1} & e^{ikd\cos\varphi_2} \\ \vdots & \vdots \\ e^{i(L-1)kd\cos\varphi_1} & e^{i(L-1)kd\cos\varphi_2} \end{bmatrix} = [a(\varphi_1) | a(\varphi_2)]$$

for signals arriving from the broadside angles $\varphi_1$ and $\varphi_2$. The quantity $k$ is the signal wave number. $a(\varphi)$ represents an arrival vector at the angle $\varphi$.

You can create a second array by translating the first array along its axis by one element distance, $d$. The arrival matrix for the second array is

$$A_2 = \begin{bmatrix} e^{ikd\cos\varphi_1} & e^{ikd\cos\varphi_2} \\ e^{i2kd\cos\varphi_1} & e^{i2kd\cos\varphi_2} \\ \vdots & \vdots \\ e^{iLkd\cos\varphi_1} & e^{iLkd\cos\varphi_2} \end{bmatrix} = \left[ e^{ikd\cos\varphi_1} a(\varphi_1) \middle| e^{ikd\cos\varphi_2} a(\varphi_2) \right]$$

where the arrival vectors are equal to the original arrival vectors but multiplied by a direction-dependent phase shift. When you translate the original array $J - 1$ more times, you get $J$ copies of the array. If you form a single array from all these copies, then the length of the single array is $M = L + (J - 1)$.

In practice, you start with an $M$-element array and form $J$ overlapping subarrays. The number of elements in each subarray is $L = M - J + 1$. The following figure shows the relationship between the overall length of the array, $M$, the number of subarrays, $J$, and the length of each subarray, $L$.
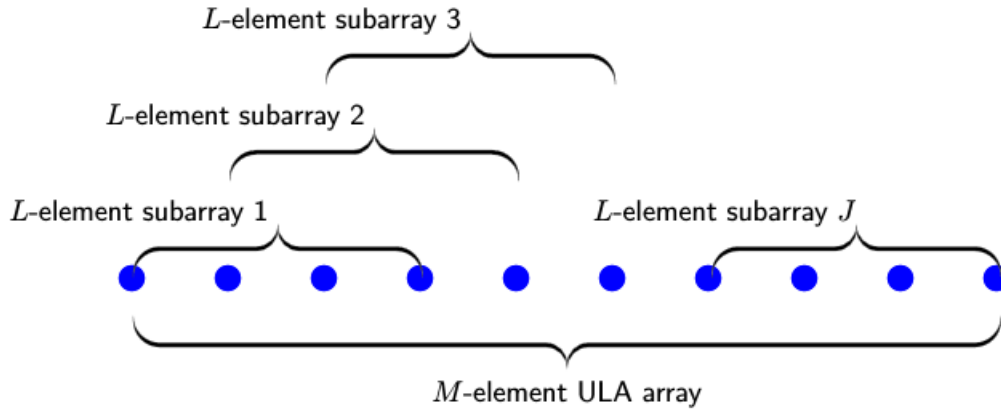
For the $p$th subarray, the source signal arrival matrix is

$$A_p = \left[ e^{ik(p-1)dcos\varphi_1} a(\varphi_1) \,\middle|\, e^{ik(p-1)dcos\varphi_2} a(\varphi_2) \right]$$

$$= [a(\varphi_1) \,|\, a(\varphi_2)] \begin{bmatrix} e^{ik(p-1)dcos\varphi_1} & 0 \\ 0 & e^{ik(p-1)dcos\varphi_2} \end{bmatrix} = A_1 P^{p-1}$$

$$P = \begin{bmatrix} e^{ikdcos\varphi_1} & 0 \\ 0 & e^{ikdcos\varphi_2} \end{bmatrix}.$$

The original arrival vector matrix is postmultiplied by a diagonal phase matrix.

The last step is averaging the signal covariance matrices over all $J$ subarrays to form the averaged signal covariance matrix, $R^{avg}{}_s$. The average signal covariance matrix depends on the smoothed source covariance matrix, $R^{smooth}$.

$$R_s^{avg} = A_1 \left( \frac{1}{J} \sum_{p=1}^{J} P^{p-1} R_s \left( P^{p-1} \right)^H \right) A_1^H = A_1 R^{smooth} A_1^H$$

$$R^{smooth} = \frac{1}{J} \sum_{p=1}^{J} P^{p-1} R_s \left( P^{p-1} \right)^H.$$

You can show that the diagonal elements of the smoothed source covariance matrix are the same as the diagonal elements of the original source covariance matrix.

$$R_{ii}^{smooth} = \frac{1}{J} \sum_{p=1}^{J} \left( P^{p-1} \right)_{im} (R_s)_{mn} \left( P^{p-1} \right)_{ni}^H = \frac{1}{J} \sum_{p=1}^{J} R_s = (R_s)_{ii}$$

However, the off-diagonal elements are reduced. The reduction factor is the beam pattern of a $J$-element array.

$$R_{ij}^{smooth} = \frac{1}{J} \sum_{p=1}^{J} e^{ikd(p-1)(\cos\varphi_1 - \cos\varphi_2)} (R_s)_{ij} = \frac{1}{J} \frac{\sin(kdJ(\cos\varphi_1 - \cos\varphi_2))}{\sin(kd(\cos\varphi_1 - \cos\varphi_2))} (R_s)_{ij}$$

In summary, you can reduce the degrading effect of source correlation by forming subarrays and using the smoothed covariance matrix as input to the MUSIC algorithm. Because of the beam pattern, larger angular separation of sources leads to reduced correlation.

Spatial smoothing for linear arrays is easily extended to 2D and 3D uniform arrays.

# Target Tracking Using Sum-Difference Monopulse Radar

This example shows how to use the `phased.SumDifferenceMonopulseTracker` System object™ to track a moving target. The `phased.SumDifferenceMonopulseTracker` tracker solves for the direction of a target from signals arriving on a uniform linear array (ULA). The sum-difference monopulse algorithm requires a prior estimate of the target direction which is assumed to be close to the actual direction. In a tracker, the current estimate serves as the prior information for the next estimate. The target is a narrowband 500 MHz emitter moving at a constant velocity of 800 kph. For a ULA array, the steering vector depends only upon the broadside angle. The broadside angle is the angle between the source direction and a plane normal to the linear array. Any arriving signal is specified by its broadside angle.

**Create the target platform and define its motion**

Assume the target is located at `[0,10000,20000]` with respect to the radar in the radar's local coordinate system. Assume that the target is moving along the y-axis toward the radar at 800 kph.

```
x0 = [0,10000,20000].';
v0 = -800;
v0 = v0*1000/3600;
targetplatform = phased.Platform(x0,[0,v0,0].');
```

**Set up the ULA array**

The monopulse tracker uses a ULA array which consists of 8 isotropic antenna elements. The element spacing is set to one-half the signal wavelength.

```
fc = 500e6;
c = physconst('LightSpeed');
lam = c/fc;
antenna = phased.IsotropicAntennaElement('FrequencyRange',[100e6,800e6],...
    'BackBaffled',true);
array = phased.ULA('Element',antenna,'NumElements',8,...
    'ElementSpacing',lam/2);
```

Assume a narrowband signal. This kind of signal can be simulated using the `phased.SteeringVector` System object.

```
steervec = phased.SteeringVector('SensorArray',array);
```

**Tracking Loop**

Initialize the tracking loop. Create the `phased.SumDifferenceMonopulseTracker` System object.

```
tracker = phased.SumDifferenceMonopulseTracker('SensorArray',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
```

At each time step, compute the broadside angle of the target with respect to the array. Set the step time to 0.5 seconds.

```
T  = 0.5;
nsteps = 40;
t = (1:nsteps)*T;
```

Setup data vectors for storing and displaying results

```
rng = zeros(1,nsteps);
broadang_actual = zeros(1,nsteps);
broadang_est = zeros(1,nsteps);
angerr = zeros(1,nsteps);
```

Step through the tracking loop. First provide an estimate of the initial broadside angle. In this simulation, the actual broadside angle is known but add an error of five degrees.

```
[tgtrng,tgtang_actual] = rangeangle(x0,[0,0,0].');
broadang0 = az2broadside(tgtang_actual(1),tgtang_actual(2));
broadang_prev = broadang0 + 5.0; % add some sort of error
```

1   Compute the actual broadside angle, `broadang_actual`.
2   Compute the signal, `signl`, from the actual broadside angle, using the `phased.SteeringVector` System object.
3   Using the `phased.SumDifferenceMonopulseTracker` tracker, estimate the broadside angle, `broadang_est`, from the signal. The broadside angle derived from a previous step serves as an initial estimate for the current step.
4   Compute the difference between the estimated broadside angle, `broadang_est`, and actual broadside angle, `broadang_actual`. This is a measure of how good the solution is.

```
for n = 1:nsteps
    x = targetplatform(T);
    [rng(n),tgtang_actual] = rangeangle(x,[0,0,0].');
    broadang_actual(n) = az2broadside(tgtang_actual(1),tgtang_actual(2));
    signl = steervec(fc,broadang_actual(n)).';
    broadang_est(n) = tracker(signl,broadang_prev);
    broadang_prev = broadang_est(n);
    angerr(n) = broadang_est(n) - broadang_actual(n);
end
```

**Results**

Plot the range as a function of time showing the point of closest approach.

```
plot(t,rng/1000,'-o')
xlabel('time (sec)')
ylabel('Range (km)')
```
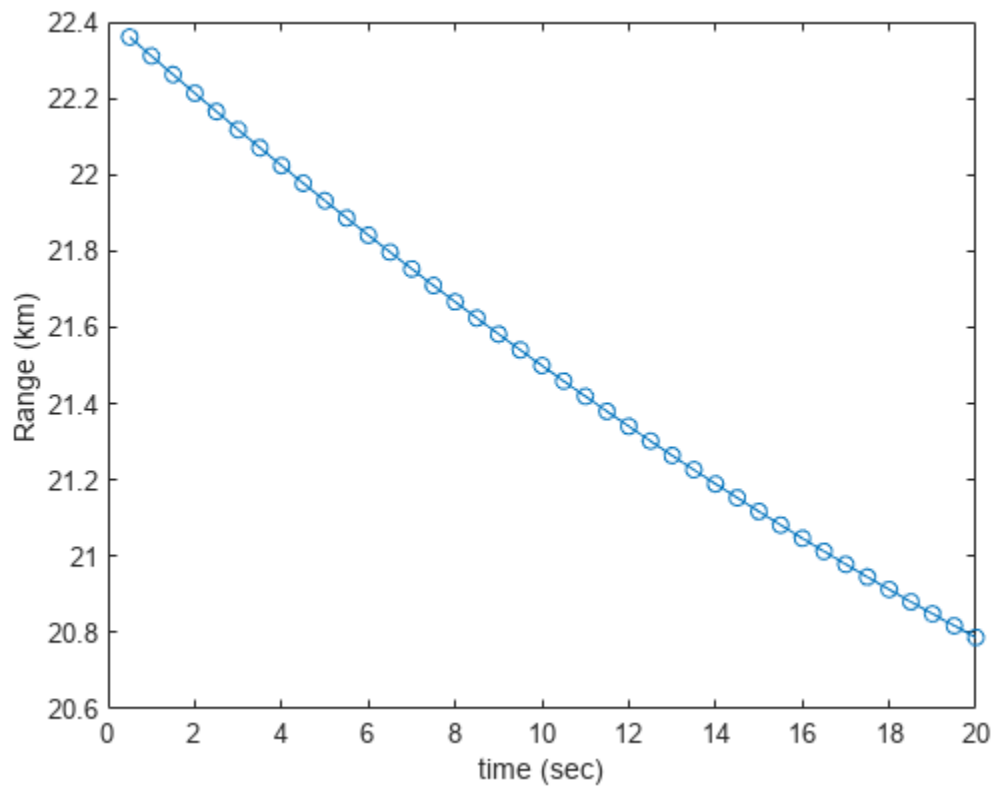
Plot the estimated broadside angle as a function of time.

```
plot(t,broadang_actual,'-o')
xlabel('time (sec)')
ylabel('Broadside angle (deg)')
```
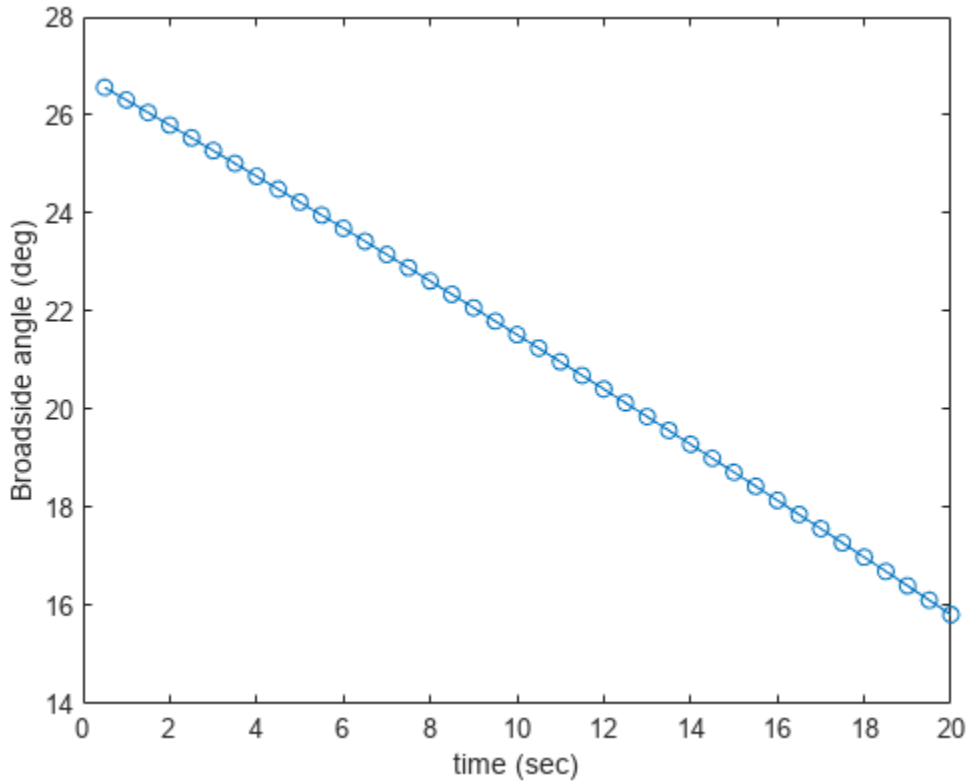
A monopulse tracker cannot solve for the direction angle if the angular separation between samples is too large. The maximum allowable angular separation is approximately one-half the null-to-null beamwidth of the array. For an 8-element, half-wavelength-spaced ULA, the half-beamwidth is approximately 14.3 degrees at broadside. In this simulation, the largest angular difference between samples is

```
maxangdiff = max(abs(diff(broadang_est)));
disp(maxangdiff)
```

```
    0.2942
```

The angular separation between samples is less than the half-beamwidth.

Plot the angle error. This is the difference between the estimated angle and the actual angle. The plot shows a very small error, on the order of microdegrees.

```
plot(t,angerr,'-o')
xlabel('time (sec)')
ylabel('Angle error (deg)')
```
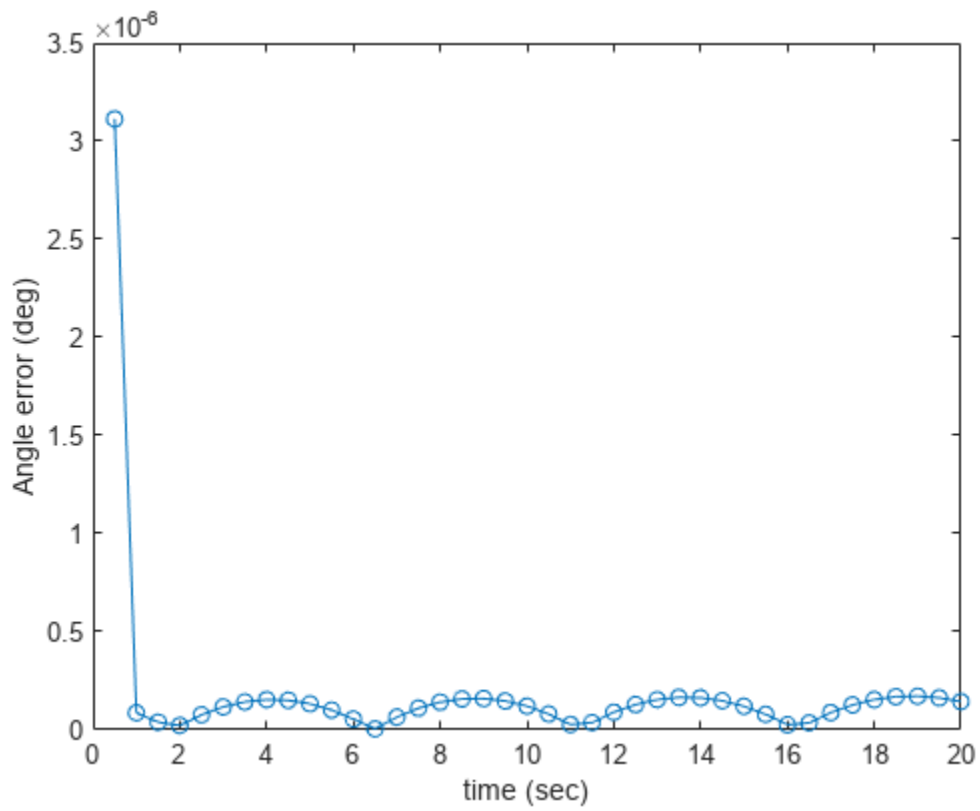
## See Also

**Functions**
musicdoa

**Objects**
phased.MUSICEstimator | phased.MUSICEstimator2D

# Space-Time Adaptive Processing (STAP)

# Angle-Doppler Response

| In this section... |
| --- |
| "Benefits of Visualizing Angle-Doppler Response" on page 7-2 |
| "Angle-Doppler Response of Stationary Array to Stationary Target" on page 7-2 |
| "Angle-Doppler Response to Stationary Target at Moving Array" on page 7-4 |

## Benefits of Visualizing Angle-Doppler Response

Visualizing a signal in the angle-Doppler domain can help you identify characteristics of the signal in direction and speed. You can distinguish among targets moving at various speeds in various directions. If a transmitter platform is stationary, returns from stationary targets map to zero in the Doppler domain while returns from moving targets exhibit a nonzero Doppler shift. If you visualize the array response in the angle-Doppler domain, a stationary target produces a response at a specified angle and zero Doppler.

You can use the `phased.AngleDopplerResponse` object to visualize the angle-Doppler response of input data. The `phased.AngleDopplerResponse` object uses a conventional narrowband (phase shift) beamformer and an FFT-based Doppler filter to compute the angle-Doppler response.

## Angle-Doppler Response of Stationary Array to Stationary Target

Display the angle-Doppler response of a stationary array to a stationary target. The array is a six-element uniform linear array (ULA) located at the global origin *(0,0,0)*. The target is located at *(5000,5000,0)* meters and has a nonfluctuating radar cross section (RCS) of 1 square meter.

Construct the objects needed to simulate the target response at the array.

```
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('LightSpeed')/4e9;
array = phased.ULA(6,'Element',antenna,'ElementSpacing',lambda/2);
waveform = phased.RectangularWaveform('PulseWidth',2e-006,...
    'PRF',5e3,'SampleRate',1e6,'NumPulses',1);
radiator = phased.Radiator('Sensor',array,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
txplatform = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0]);
target = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[0;0;0]);
freespace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',1e6,'Gain',40);
transmitter = phased.Transmitter('PeakPower',1e4,...
    'InUseOutputPort',true,'Gain',40);
```

Propagate ten rectangular pulses to and from the target, and collect the responses at the array.

```
PRF = 5e3;
NumPulses = 10;
wav = waveform();
tgtloc = targetplatform.InitialPosition;
txloc = txplatform.InitialPosition;
M = waveform.SampleRate*1/PRF;
N = array.NumElements;
rxsig = zeros(M,N,NumPulses);

for n = 1:NumPulses
    % get angle to target
    [~,tgtang] = rangeangle(tgtloc,txloc);
    % transmit pulse
    [txsig,txstatus] = transmitter(wav);
    % radiate pulse
    txsig = radiator(txsig,tgtang);
    % propagate pulse to target
    txsig = freespace(txsig,txloc,tgtloc,[0;0;0],[0;0;0]);
    % reflect pulse off stationary target
    txsig = target(txsig);
    % propagate pulse to array
    txsig = freespace(txsig,tgtloc,txloc,[0;0;0],[0;0;0]);
    % collect pulse
    rxsig(:,:,n) = collector(txsig,tgtang);
    % receive pulse
    rxsig(:,:,n) = receiver(rxsig(:,:,n),~txstatus);
end
```

Find and plot the angle-Doppler response. Then, add the label +Target at the expected azimuth angle and Doppler frequency.

```
tgtdoppler = 0;
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,...
    physconst('LightSpeed')/(2*waveform.SampleRate));
snapshot = shiftdim(rxsig(tgtcell,:,:)); % Remove singleton dim
response = phased.AngleDopplerResponse('SensorArray',array,...
    'OperatingFrequency',4e9, ...
    'PropagationSpeed',physconst('LightSpeed'),...
    'PRF',PRF, 'ElevationAngle',tgtelang);
plotResponse(response,snapshot);
text(tgtazang,tgtdoppler,'+Target');
```
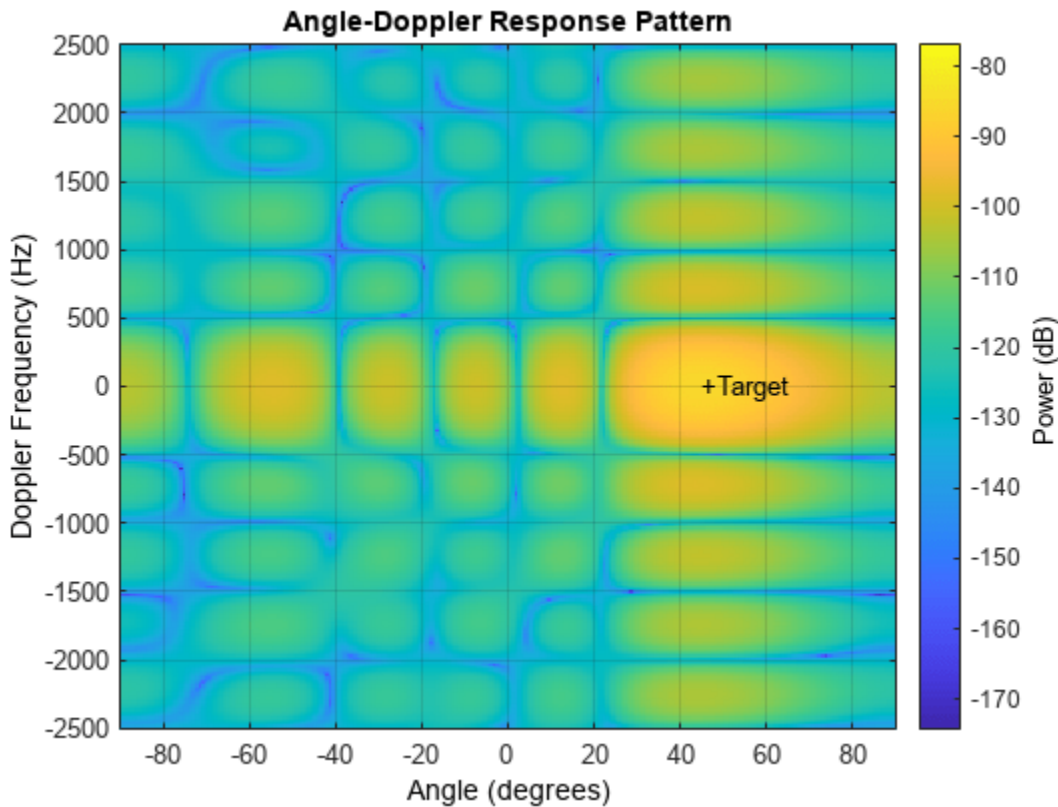
As expected, the angle-Doppler response shows the greatest response at zero Doppler and 45° azimuth.

## Angle-Doppler Response to Stationary Target at Moving Array

This example illustrates the nonzero Doppler shift exhibited by a stationary target in the presence of array motion. In general, this nonzero shift complicates the detection of slow-moving targets because the motion-induced Doppler shift and spread of the clutter returns obscure the Doppler shifts of such targets.

The scenario in this example is identical to that of "Angle-Doppler Response of Stationary Array to Stationary Target" on page 7-2 except that the ULA is moving at a constant velocity. For convenience, the MATLAB® code to set up the objects is repeated. Notice that the `InitialPosition` and `Velocity` properties of the `txplatform` System object™ have changed. The `InitialPosition` property value is set to simulate an airborne ULA. The motivation for selecting the particular value of the `Velocity` property is explained in "Applicability of DPCA Pulse Canceller" on page 7-7.

```
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('LightSpeed')/4e9;
array = phased.ULA(6,'Element',antenna,'ElementSpacing',lambda/2);
waveform = phased.RectangularWaveform('PulseWidth',2e-006,...
    'PRF',5e3,'SampleRate',1e6,'NumPulses',1);
radiator = phased.Radiator('Sensor',array,...
```

```
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
vy = (array.ElementSpacing*waveform.PRF)/2;
txplatform = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
target = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
tgtvel = [0;0;0];
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',tgtvel);
freespace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',1e6,'Gain',40);
transmitter = phased.Transmitter('PeakPower',1e4,...
    'InUseOutputPort',true,'Gain',40);
```

Transmit ten rectangular pulses toward the target as the ULA is moving. Then, collect the received echoes.

```
PRF = 5e3;
NumPulses = 10;
wav = waveform();
tgtloc = targetplatform.InitialPosition;
M = waveform.SampleRate*1/PRF;
N = array.NumElements;
rxsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/waveform.SampleRate,1/PRF,'[)');
rangebins = (physconst('LightSpeed')*fasttime)/2;

for n = 1:NumPulses
    % move transmitter
    [txloc,txvel] = txplatform(1/PRF);
    % get angle to target
    [~,tgtang] = rangeangle(tgtloc,txloc);
    % transmit pulse
    [txsig,txstatus] = transmitter(wav);
    % radiate pulse
    txsig = radiator(txsig,tgtang);
    % propagate pulse to target
    txsig = freespace(txsig,txloc,tgtloc,txvel,tgtvel);
    % reflect pulse off stationary target
    txsig = target(txsig);
    % propagate pulse to array
    txsig = freespace(txsig,tgtloc,txloc,tgtvel,txvel);
    % collect pulse
    rxsig(:,:,n) = collector(txsig,tgtang);
    % receive pulse
    rxsig(:,:,n) = receiver(rxsig(:,:,n),~txstatus);
end
```

Calculate the target angles and range with respect to the ULA. Then, calculate the Doppler shift induced by the motion of the phased array.

```
sp = radialspeed(tgtloc,tgtvel,txloc,txvel);
tgtdoppler = 2*speed2dop(sp,lambda);
```

```
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
```

The two-way Doppler shift is approximately 1626 Hz. The azimuth angle is 45° and is identical to the value obtained in the stationary ULA example.

Plot the angle-Doppler response.

```
tgtcell = val2ind(tgtrng,...
    physconst('LightSpeed')/(2*waveform.SampleRate));
snapshot = shiftdim(rxsig(tgtcell,:,:)); % Remove singleton dim
hadresp = phased.AngleDopplerResponse('SensorArray',array,...
    'OperatingFrequency',4e9, ...
    'PropagationSpeed',physconst('LightSpeed'),...
    'PRF',PRF, 'ElevationAngle',tgtelang);
plotResponse(hadresp,snapshot);
text(tgtazang,tgtdoppler,'+Target');
```
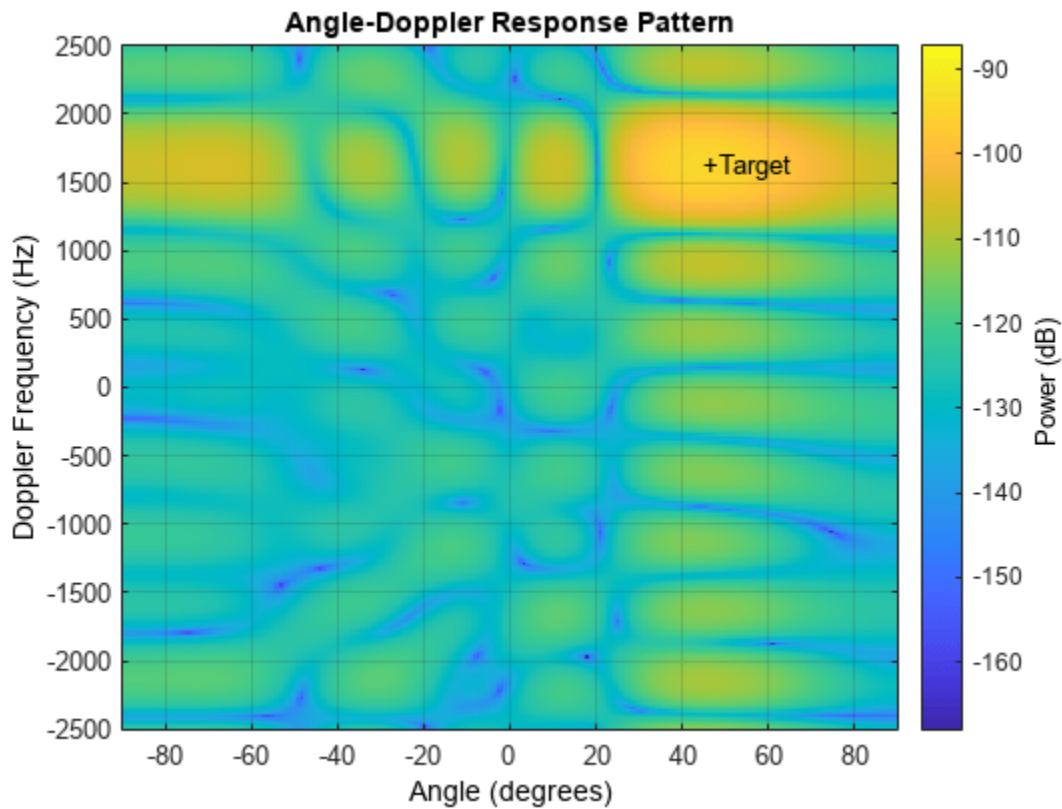


The angle-Doppler response shows the greatest response at 45° azimuth at the expected Doppler shift.

# Displaced Phase Center Antenna Pulse Canceller

| **In this section...** |
| --- |
| "When to Use the DPCA Pulse Canceller" on page 7-7 |
| "DPCA Pulse Canceller to Reject Clutter" on page 7-7 |

## When to Use the DPCA Pulse Canceller

In a *moving target indication* (MTI) radar, clutter returns can make it more difficult to detect and track the targets of interest. A rudimentary way to mitigate the effects of clutter returns in such a system is to implement a displaced phase center antenna (DPCA) pulse canceller on the slow-time data.

You can implement a DPCA pulse canceller with `phased.DPCACanceller`. This implementation assumes that the entire array is used on transmit. On receive, the array is divided into two subarrays. The phase centers of the subarrays are separated by twice the distance the platform moves in one pulse repetition interval.

### Applicability of DPCA Pulse Canceller

The DPCA pulse canceller is applicable when both these conditions are true:

- Clutter is stationary across pulses.
- The motion satisfies

$$vT = d/2 \tag{7-1}$$

  where:

  - $v$ indicates the speed of the platform
  - $T$ represents the pulse repetition interval
  - $d$ indicates the interelement spacing of the array

## DPCA Pulse Canceller to Reject Clutter

This example implements a DPCA pulse canceller for clutter rejection. Assume you have an airborne radar platform modeled by a six-element ULA operating at 4 GHz. The array elements are spaced at one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses two microseconds in duration with a PRF of 5 kHz. The platform moves along the array axis with a speed equal to one-half the product of the element spacing and the PRF. DPCA pulse cancellation is applicable because $vT = d/2$ where

- $v$ indicates the speed of the platform
- $T$ represents the pulse repetition interval
- $d$ indicates the interelement spacing of the array

The target has a nonfluctuating RCS of 1 square meter and moves with a constant velocity vector of *(15,15,0)*.

```matlab
PRF = 5e3;
fc = 4e9;
fs = 1e6;
c = physconst('LightSpeed');
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
array = phased.ULA(6,'Element',antenna,'ElementSpacing',lambda/2);
waveform = phased.RectangularWaveform('PulseWidth',2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
radiator = phased.Radiator('Sensor',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (array.ElementSpacing*PRF)/2;
transmitplatform = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);

target = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
channel = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'Gain',40);
transmitter = phased.Transmitter('PeakPower',1e4,...
    'InUseOutputPort',true,'Gain',40);

% Load simulated clutter data
load clutterdata
```

Propagate ten rectangular pulses to target and back, and collect the responses at the array. Also, compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain.

```matlab
NumPulses = 10;
wav = waveform();
M = fs/PRF;
N = array.NumElements;
rxsig = zeros(M,N,NumPulses);
csig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c*fasttime)/2;

for n = 1:NumPulses
    [txloc,txvel] = transmitplatform(1/PRF); % move transmitter
    [tgtloc,tgtvel] = targetplatform(1/PRF); % move target
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig1,txstatus] = transmitter(wav);  % transmit pulse
    txsig = radiator(txsig1,tgtang); % radiate pulse
    txsig = channel(txsig,txloc,tgtloc,...
        txvel,tgtvel); % propagate to target
    txsig = target(txsig);  % reflect off target
    txsig = channel(txsig,tgtloc,txloc,...
        tgtvel,txvel); % propagate to array
```

```
    rxsig(:,:,n) = collector(txsig,tgtang); % collect pulse
    rxsig(:,:,n) = receiver(rxsig(:,:,n) + csig(:,:,n),...
        ~txstatus); % receive pulse plus clutter return
end
```

Determine the target range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc,tgtvel,txloc,txvel);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,c/(2*fs));
```

Use noncoherent pulse integration to visualize the signal received by the ULA for the first of the ten pulses. Mark the target range gate with a vertical dashed line.

```
firstpulse = pulsint(rxsig(:,:,1),'noncoherent');
plot([tgtrng/1e3 tgtrng/1e3],[0 0.1],'-.',rangebins/1e3,firstpulse)
title('Noncoherent Integration of 1st Pulse at the ULA')
xlabel('Range (km)')
ylabel('Magnitude');
```
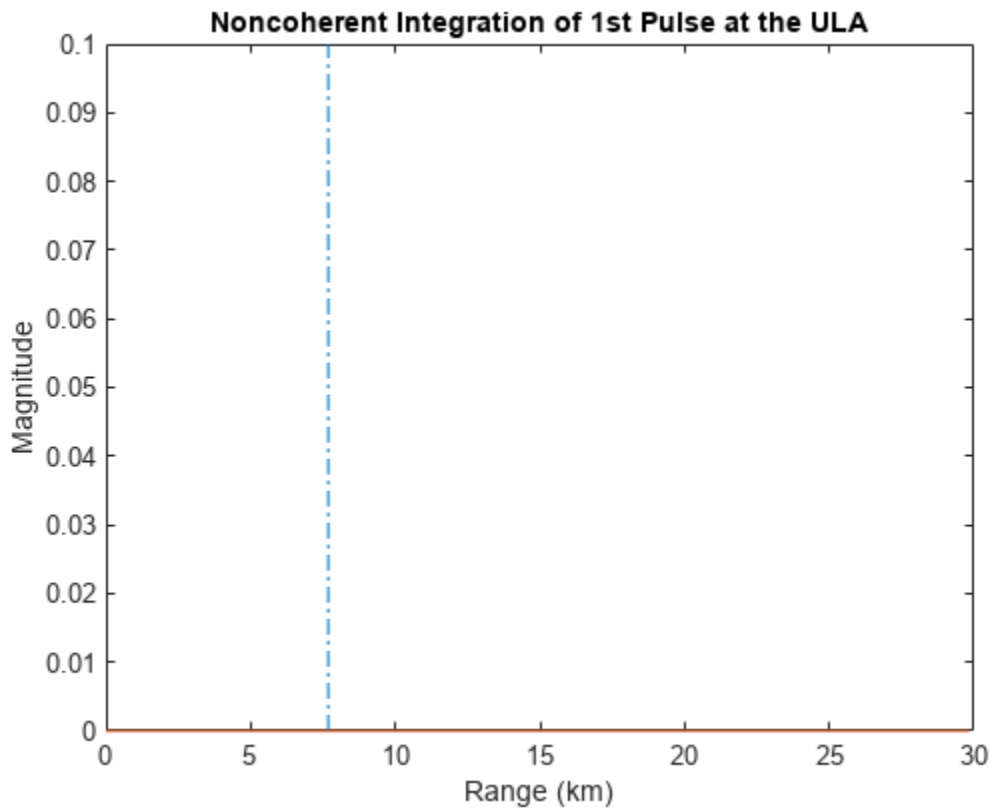


The large-magnitude clutter returns obscure the presence of the target. Apply the DPCA pulse canceller to reject the clutter.

```
canceller = phased.DPCACanceller('SensorArray',array,'PRF',PRF,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'Direction',[0;0],'Doppler',tgtdoppler,...
    'WeightsOutputPort',true);
[y,w] = canceller(rxsig,tgtcell);
```

Plot the result of applying the DPCA pulse canceller. Mark the target range gate with a vertical dashed line.

```
plot([tgtrng/1e3,tgtrng/1e3],[0 3.5e-5],'-.',rangebins/1e3,abs(y))
title('DPCA Canceller Output')
xlabel('Range (km)')
ylabel('Magnitude')
```
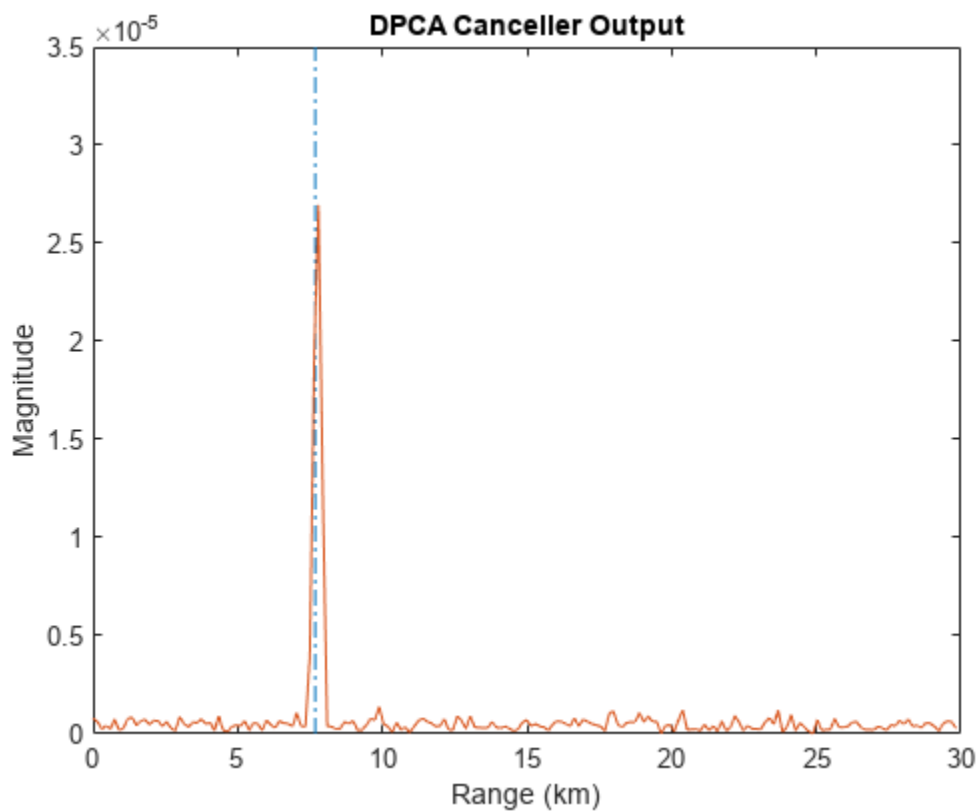


The DPCA pulse canceller has significantly rejected the clutter. As a result, the target is visible at the expected range gate.

# Adaptive Displaced Phase Center Antenna Pulse Canceller

| In this section... |
|---|
| "When to Use the Adaptive DPCA Pulse Canceller" on page 7-11 |
| "Adaptive DPCA Pulse Canceller To Reject Clutter and Interference" on page 7-11 |

## When to Use the Adaptive DPCA Pulse Canceller

Consider an airborne radar system that needs to suppress clutter returns and possibly jammer interference. Under any of the following conditions, you might choose an adaptive DPCA (ADPCA) pulse canceller for suppressing these effects.

- Jamming and other interference effects are substantial. The DPCA pulse canceller is susceptible to interference because the DPCA pulse canceller does not use the received data.

- The sample matrix inversion (SMI) algorithm is inapplicable because of computational expense or a rapidly changing environment.

The `phased.ADPCAPulseCanceller` object implements an ADPCA pulse canceller. This pulse canceller uses the data received from two consecutive pulses to estimate the space-time interference covariance matrix. In particular, the object lets you specify:

- The number of training cells. The algorithm uses training cells to estimate the interference. In general, a larger number of training cells leads to a better estimate of interference.

- The number of guard cells close to the target cells. The algorithm recognizes guard cells to prevent target returns from contaminating the estimate of the interference.

## Adaptive DPCA Pulse Canceller To Reject Clutter and Interference

This example implements an adaptive DPCA pulse canceller for clutter and interference rejection. The scenario is identical to the one in "DPCA Pulse Canceller to Reject Clutter" on page 7-7 except that noise equivalent to a stationary broadband barrage jammer is added at the location *(3.5e3,1e3,0)*. The jammer has an effective radiated power of 1 kW.

To repeat the scenario for convenience, the airborne radar platform is a six-element ULA operating at 4 GHz. The array elements are spaced at one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses of two µs duration with a PRF of 5 kHz. The platform is moving along the array axis with a speed equal to one-half the product of the element spacing and the PRF. ADPCA pulse cancellation is applicable because $vT = d/2$ where

- $v$ indicates the speed of the platform
- $T$ represents the pulse repetition interval
- $d$ indicates the interelement spacing of the array

The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of (15,15,0).

```
PRF = 5e3;
fc = 4e9;
fs = 1e6;
```

```
c = physconst('LightSpeed');
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
array = phased.ULA(6,'Element',antenna,'ElementSpacing',lambda/2);
waveform = phased.RectangularWaveform('PulseWidth', 2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
radiator = phased.Radiator('Sensor',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (array.ElementSpacing * PRF)/2;
transmitterplatform = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
% Load simulated constant gamma clutter
load clutterdata
target = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
% add jammer signal with 200 samples per frame and an ERP of 1000 W.
jamsig = sqrt(1000)*randn(200,1);

jammerplatform = phased.Platform(...
    'InitialPosition',[3.5e3; 1e3; 0],'Velocity',[0;0;0]);
channel = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'Gain',40);
transmitter = phased.Transmitter('PeakPower',1e4,...
    'InUseOutputPort',true,'Gain',40);
```

Propagate the ten rectangular pulses to the target and back and collect the responses at the array. Compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain. Also, propagate the jamming signal from the jammer location to the airborne ULA.

```
NumPulses = 10;
wav = waveform();
M = fs/PRF;
N = array.NumElements;
rxsig = zeros(M,N,NumPulses);
%csig = zeros(M,N,NumPulses);
jsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c * fasttime)/2;
receiver.SeedSource = 'Property';
receiver.Seed = 56113;
jamloc = jammerplatform.InitialPosition;

for n = 1:NumPulses
    [txloc,txvel] = transmitterplatform(1/PRF); % move transmitter
    [tgtloc,tgtvel] = targetplatform(1/PRF); % move target
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = transmitter(wav); % transmit pulse
```

```
    txsig = radiator(txsig,tgtang); % radiate pulse
    txsig = channel(txsig,txloc,tgtloc,...
        txvel,tgtvel); % propagate pulse to target
    txsig = target(txsig); % reflect off target
    txsig = channel(txsig,tgtloc,txloc,...
        tgtvel,txvel); % propagate to array
    rxsig(:,:,n) = collector(txsig,tgtang); % collect pulse

    [~,jamang] = rangeangle(jamloc,txloc); % angle from jammer to transmitter
    jamsig = channel(jamsig,jamloc,txloc,...
        [0;0;0],txvel); % propagate jammer signal
    jsig(:,:,n) = collector(jamsig,jamang); % collect jammer signal

    rxsig(:,:,n) = receiver(...
        rxsig(:,:,n) + csig(:,:,n) + jsig(:,:,n),...
        ~txstatus); % receive pulse plus clutter return plus jammer signal
end
```

Determine the target range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc, targetplatform.Velocity, ...
    txloc, transmitterplatform.Velocity);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,c/(2*fs));
```

Process the array responses using the nonadaptive DPCA pulse canceller. To do so, construct the DPCA object, and apply it to the received signals.

```
canceller = phased.DPCACanceller('SensorArray',array,'PRF',PRF,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'Direction',[0;0],'Doppler',tgtdoppler,...
    'WeightsOutputPort',true);
[y,w] = canceller(rxsig,tgtcell);
```

Plot the DPCA result with the target range marked by a vertical dashed line. Notice how the presence of the interference signal has obscured the target.

```
plot([tgtrng/1e3,tgtrng/1e3],[0 7e-2],'-.',rangebins/1e3,abs(y))
axis tight
xlabel('Range (km)')
ylabel('Magnitude')
title('DPCA Canceller Output with Jamming')
```

**7-13**
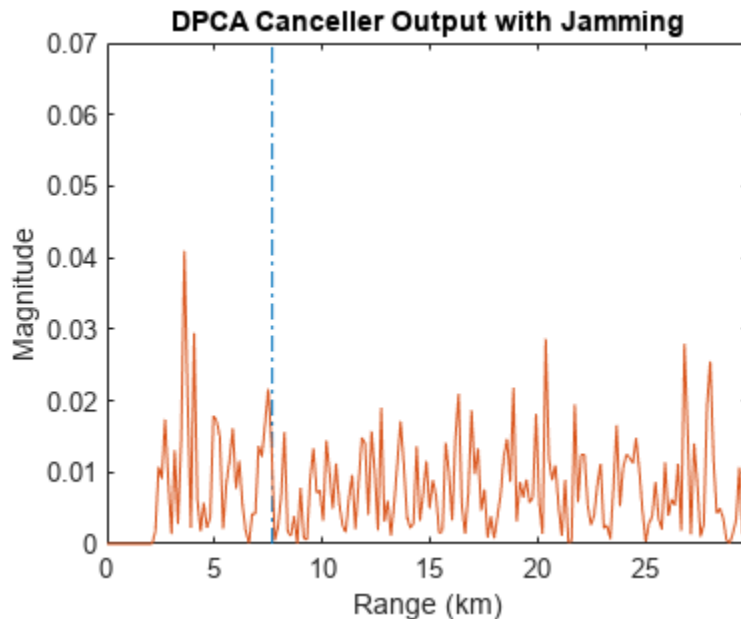
Apply the adaptive DPCA pulse canceller. Use 100 training cells and 4 guard cells, two on each side of the target range gate.

```
canceller = phased.ADPCACanceller('SensorArray',array,'PRF',PRF,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'Direction',[0;0],'Doppler',tgtdoppler,...
    'WeightsOutputPort',true,'NumGuardCells',4,...
    'NumTrainingCells',100);
[y_adpca,w_adpca] = canceller(rxsig,tgtcell);
```

Plot the result with the target range marked by a vertical dashed line. Notice how the adaptive DPCA pulse canceller enables you to detect the target in the presence of the jamming signal.

```
plot([tgtrng/1e3,tgtrng/1e3],[0 4e-7],'-.',rangebins/1e3,abs(y_adpca))
axis tight
title('ADPCA Canceller Output with Jamming')
xlabel('Range (km)')
ylabel('Magnitude')
```

ADPCA Canceller Output with Jamming
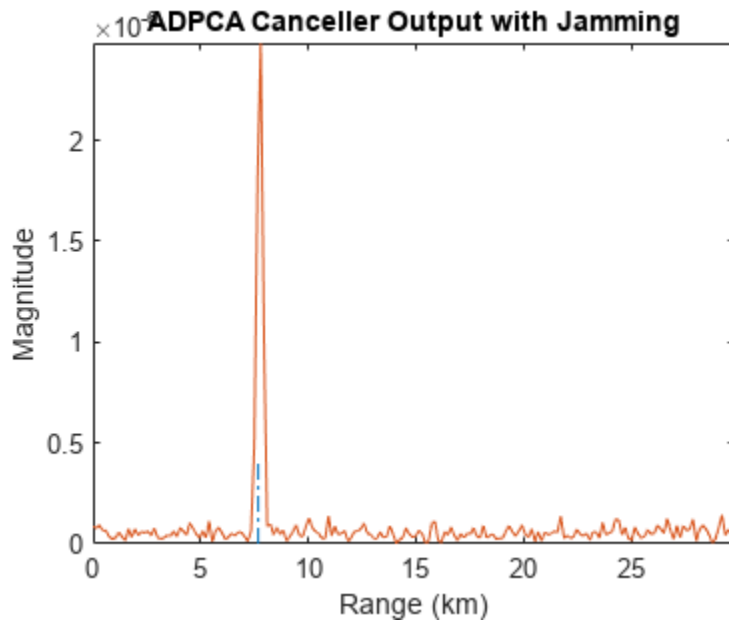
Examine the angle-Doppler response. Notice the presence of the clutter ridge in the angle-Doppler plane and the null at the jammer's broadside angle for all Doppler frequencies.

```
angdopplerersponse = phased.AngleDopplerResponse('SensorArray',array,...
    'OperatingFrequency',fc,...
    'PropagationSpeed',c,...
    'PRF',PRF,'ElevationAngle',tgtelang);
plotResponse(angdopplerersponse,w_adpca)
title('Angle-Doppler Response with ADPCA Pulse Cancellation')
text(az2broadside(jamang(1),jamang(2)) + 10,...
    0,'\leftarrow Interference Null')
```



Angle-Doppler Response with ADPCA Pulse Cancellation
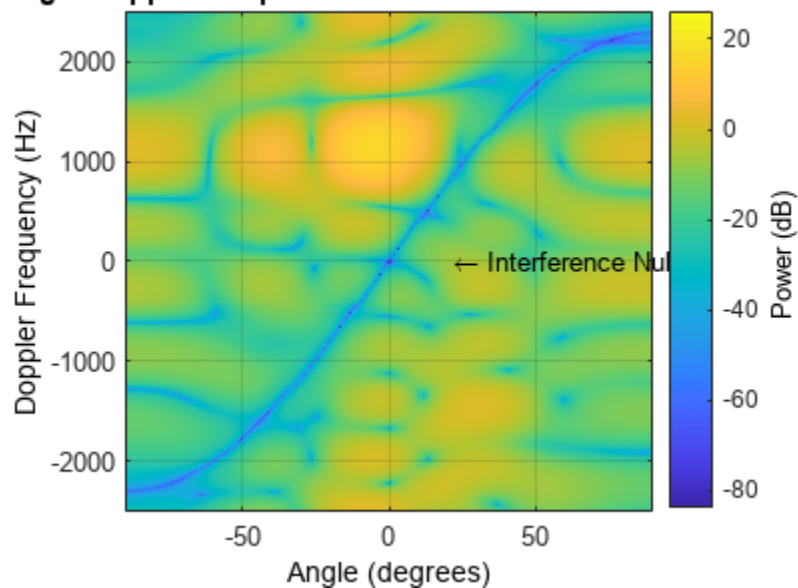
# Sample Matrix Inversion Beamformer

| In this section... |
| --- |
| "When to Use the SMI Beamformer" on page 7-16 |
| "Sample Matrix Inversion Beamformer" on page 7-16 |

## When to Use the SMI Beamformer

In situations where an airborne radar system needs to suppress clutter returns and jammer interference, the system needs a more sophisticated algorithm than a DPCA pulse canceller can provide. One option is the sample matrix inversion (SMI) algorithm. SMI is the optimum STAP algorithm and is often used as a baseline for comparison with other algorithms.

The SMI algorithm is computationally expensive and assumes a stationary environment across many pulses. If you need to suppress clutter returns and jammer interference with less computation, or in a rapidly changing environment, consider using an ADPCA pulse canceller instead.

The `phased.STAPSMIBeamformer` object implements the SMI algorithm. In particular, the object lets you specify:

* The number of training cells. The algorithm uses training cells to estimate the interference. In general, a larger number of training cells leads to a better estimate of interference.
* The number of guard cells close to the target cells. The algorithm recognizes guard cells to prevent target returns from contaminating the estimate of the interference.

## Sample Matrix Inversion Beamformer

This scenario is identical to the one presented in "Adaptive DPCA Pulse Canceller To Reject Clutter and Interference" on page 7-11. You can run the code for both examples to compare the ADPCA pulse canceller with the SMI beamformer. The example details and code are repeated for convenience.

To repeat the scenario for convenience, the airborne radar platform is a six-element ULA operating at 4 GHz. The array elements are spaced at one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses two µs in duration with a PRF of 5 kHz. The platform is moving along the array axis with a speed equal to one-half the product of the element spacing and the PRF. The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of *(15,15,0)*. A stationary broadband barrage jammer is located at *(3.5e3,1e3,0)*. The jammer has an effective radiated power of 1 kW.

```
PRF = 5e3;
fc = 4e9;
fs = 1e6;
c = physconst('LightSpeed');
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
array = phased.ULA(6,'Element',antenna,'ElementSpacing',lambda/2);
waveform = phased.RectangularWaveform('PulseWidth', 2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
radiator = phased.Radiator('Sensor',array,...
    'PropagationSpeed',c,...
```

```matlab
    'OperatingFrequency',fc);
collector = phased.Collector('Sensor',array,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (array.ElementSpacing * PRF)/2;
transmitterplatform = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
% Load simulated constant gamma clutter
load clutterdata
target = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
% add jammer signal with 200 samples per frame and an ERP of 1000 W.
jamsig = sqrt(1000)*randn(200,1);

jammerplatform = phased.Platform(...
    'InitialPosition',[3.5e3; 1e3; 0],'Velocity',[0;0;0]);
channel = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
receiverpreamp = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'Gain',40);
transmitter = phased.Transmitter('PeakPower',1e4,...
    'InUseOutputPort',true,'Gain',40);
```

Propagate the ten rectangular pulses to and from the target and collect the responses at the array. Compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain. Also, propagate the jamming signal from the jammer location to the airborne ULA.

```matlab
NumPulses = 10;
wav = waveform();
M = fs/PRF;
N = array.NumElements;
rxsig = zeros(M,N,NumPulses);
%csig = zeros(M,N,NumPulses);
jsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c*fasttime)/2;
clutter.SeedSource = 'Property';
clutter.Seed = 40543;
jammer.SeedSource = 'Property';
jammer.Seed = 96703;
receiverpreamp.SeedSource = 'Property';
receiverpreamp.Seed = 56113;
jamloc = jammerplatform.InitialPosition;

for n = 1:NumPulses
    [txloc,txvel] = transmitterplatform(1/PRF); % move transmitter
    [tgtloc,tgtvel] = targetplatform(1/PRF); % move target
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = transmitter(wav); % transmit pulse
    %csig(:,:,n) = clutter(txsig(abs(txsig)>0)); % collect clutter

    txsig = radiator(txsig,tgtang); % radiate pulse
    txsig = channel(txsig,txloc,tgtloc,...
        txvel,tgtvel); % propagate pulse to target
    txsig = target(txsig); % reflect off target
    txsig = channel(txsig,tgtloc,txloc,...
```

```
        tgtvel,txvel); % propagate to array
    rxsig(:,:,n) = collector(txsig,tgtang); % collect pulse

    %jamsig = jammer(); % generate jammer signal
    [~,jamang] = rangeangle(jamloc,txloc); % angle from jammer to transmitter
    jamsig = channel(jamsig,jamloc,txloc,...
        [0;0;0],txvel); % propagate jammer signal
    jsig(:,:,n) = collector(jamsig,jamang); % collect jammer signal

    rxsig(:,:,n) = receiverpreamp(...
        rxsig(:,:,n) + csig(:,:,n) + jsig(:,:,n),...
        ~txstatus); % receive pulse plus clutter return plus jammer signal
end
```

Determine the target's range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc, targetplatform.Velocity, ...
    txloc, transmitterplatform.Velocity);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,c/(2 * fs));
```

Construct an SMI beamformer object. Use 100 training cells, 50 on each side of the target range gate. Use four guard cells, two range gates in front of the target cell and two range gates beyond the target cell. Obtain the beamformer response and weights.

```
tgtang = [tgtazang; tgtelang];
beamformer = phased.STAPSMIBeamformer('SensorArray',array,...
    'PRF',PRF,'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'Direction',tgtang,'Doppler',tgtdoppler,...
    'WeightsOutputPort',true,...
    'NumGuardCells',4,'NumTrainingCells',100);
[y,weights] = beamformer(rxsig,tgtcell);
```

Plot the resulting array output after beamforming.

```
plot([tgtrng,tgtrng],[0 5e-6],'-.',rangebins,abs(y))
axis tight
title('SMI Beamformer Output')
xlabel('Range (meters)')
ylabel('Magnitude')
```
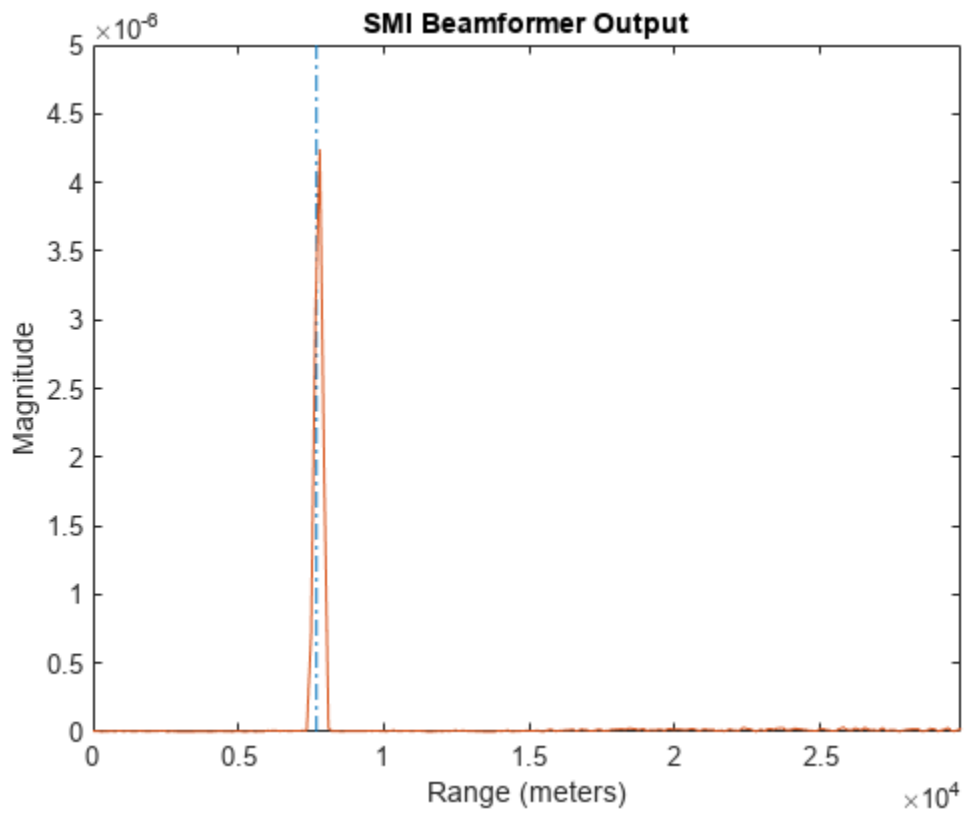
Plot the angle-Doppler response with the beamforming weights.

```
response = phased.AngleDopplerResponse('SensorArray',array,...
    'OperatingFrequency',4e9,'PRF',PRF,...
    'PropagationSpeed',physconst('LightSpeed'));
plotResponse(response,weights)
title('Angle-Doppler Response with SMI Beamforming Weights')
```

Angle-Doppler Response with SMI Beamforming Weights

**8**
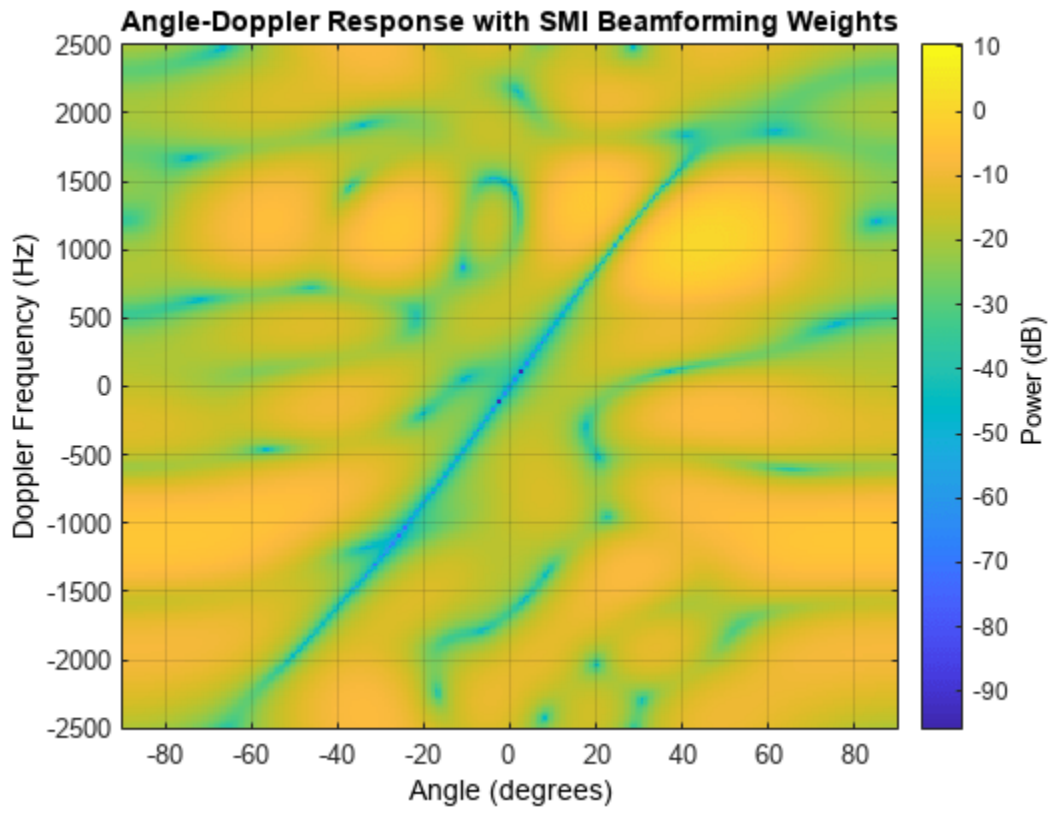
# Detection

# Neyman-Pearson Hypothesis Testing

| **In this section...** |
| --- |
| "Purpose of Hypothesis Testing" on page 8-2 |
| "Support for Neyman-Pearson Hypothesis Testing" on page 8-2 |
| "Threshold for Real-Valued Signal in White Gaussian Noise" on page 8-2 |
| "Threshold for Two Pulses of Real-Valued Signal in White Gaussian Noise" on page 8-3 |
| "Threshold for Complex-Valued Signals in Complex White Gaussian Noise" on page 8-4 |

## Purpose of Hypothesis Testing

In phased-array applications, you sometimes need to decide between two competing hypotheses to determine the reality underlying the data the array receives. For example, suppose one hypothesis, called the null hypothesis, states that the observed data consists of noise only. Suppose another hypothesis, called the alternative hypothesis, states that the observed data consists of a deterministic signal plus noise. To decide, you must formulate a decision rule that uses specified criteria to choose between the two hypotheses.

## Support for Neyman-Pearson Hypothesis Testing

When you use Phased Array System Toolbox software for applications such as radar and sonar, you typically use the Neyman-Pearson (NP) optimality criterion to formulate your hypothesis test.

When you choose the NP criterion, you can use `npwgnthresh` to determine the threshold for the detection of deterministic signals in white Gaussian noise. The optimal decision rule derives from a likelihood ratio test (LRT). An LRT chooses between the null and alternative hypotheses based on a ratio of conditional probabilities.

`npwgnthresh` enables you to specify the maximum false-alarm probability as a constraint. A false alarm means determining that the data consists of a signal plus noise, when only noise is present.

For details about the statistical assumptions the `npwgnthresh` function makes, see the reference page for that function.

## Threshold for Real-Valued Signal in White Gaussian Noise

This example shows how to compute empirically the probability of false alarm for a real-valued signal in white Gaussian noise.

Determine the required signal-to-noise (SNR) in decibels for the NP detector when the maximum tolerable false-alarm probability is 10^-3.

```
Pfa = 1e-3;
T = npwgnthresh(Pfa,1,'real');
```

Determine the actual detection threshold corresponding to the desired false-alarm probability, assuming the variance is 1.

```
variance = 1;
threshold = sqrt(variance * db2pow(T));
```

Verify empirically that the detection threshold results in the desired false-alarm probability under the null hypothesis. To do so, generate 1 million samples of a Gaussian random variable, and determine the proportion of samples that exceed the threshold.

```
rng default
N = 1e6;
x = sqrt(variance) * randn(N,1);
falsealarmrate = sum(x > threshold)/N
```

falsealarmrate = 9.9500e-04

Plot the first 10,000 samples. The red horizontal line shows the detection threshold.

```
x1 = x(1:1e4);
plot(x1)
line([1 length(x1)],[threshold threshold],'Color','red')
xlabel('Sample')
ylabel('Value')
```
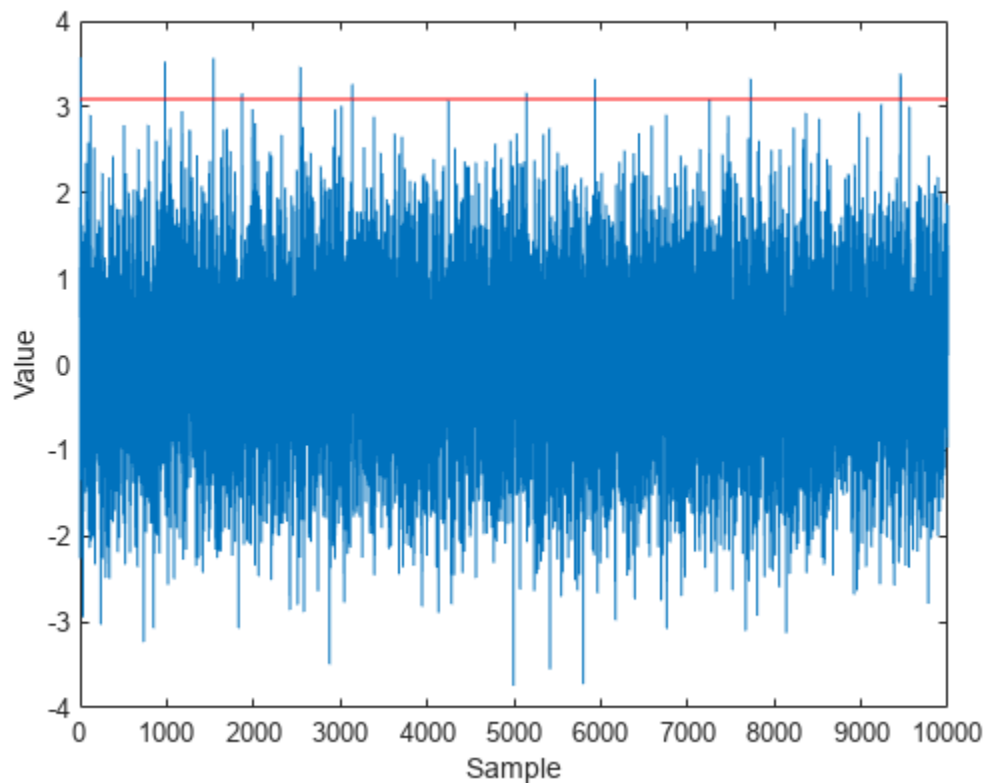


You can see that few sample values exceed the threshold. This result is expected because of the small false-alarm probability.

## Threshold for Two Pulses of Real-Valued Signal in White Gaussian Noise

This example shows how to empirically verify the probability of false alarm in a system that integrates two real-valued pulses. In this scenario, each integrated sample is the sum of two samples, one from each pulse.

Determine the required SNR for the NP detector when the maximum tolerable false-alarm probability is $10^{-3}$.

```
pfa = 1e-3;
T = npwgnthresh(pfa,2,'real');
```

Generate two sets of one million samples of a Gaussian random variable.

```
rng default
variance = 1;
N = 1e6;
pulse1 = sqrt(variance)*randn(N,1);
pulse2 = sqrt(variance)*randn(N,1);
intpuls = pulse1 + pulse2;
```

Compute the proportion of samples that exceed the threshold.

```
threshold = sqrt(variance*db2pow(T));
falsealarmrate = sum(intpuls > threshold)/N
```

```
falsealarmrate = 9.8900e-04
```

The empirical false alarm rate is very close to .001

## Threshold for Complex-Valued Signals in Complex White Gaussian Noise

This example shows how to empirically verify the probability of false alarm in a system that uses *coherent detection* of complex-valued signals. Coherent detection means that the system utilizes information about the phase of the complex-valued signals.

Determine the required SNR for the NP detector in a coherent detection scheme with one sample. Use a maximum tolerable false-alarm probability of $10^{-3}$.

```
pfa = 1e-3;
T = npwgnthresh(pfa,1,'coherent');
```

Test that this threshold empirically results in the correct false-alarm rate The sufficient statistic in the complex-valued case is the real part of the received sample.

```
rng default
variance = 1;
N = 1e6;
x = sqrt(variance/2)*(randn(N,1)+1j*randn(N,1));
threshold = sqrt(variance*db2pow(T));
falsealarmrate = sum(real(x)>threshold)/length(x)
```

```
falsealarmrate = 9.9500e-04
```

# Receiver Operating Characteristics

Receiver Operating Characteristic (ROC) curves present graphical summaries of a detector's performance. You can generate ROC curves using the `rocpfa` and `rocsnr` functions.

If you are interested in examining the effect of varying the false-alarm probability on the probability of detection for a fixed SNR, you can use `rocsnr`. For example, the threshold SNR for the Neyman-Pearson detector of a single sample in real-valued Gaussian noise is approximately 13.5 dB. Use `rocsnr` to plot the probability of detection varies as a function of the false-alarm rate at that SNR.

```
T = npwgnthresh(1e-6,1,'real');
rocsnr(T,'SignalType','real')
```
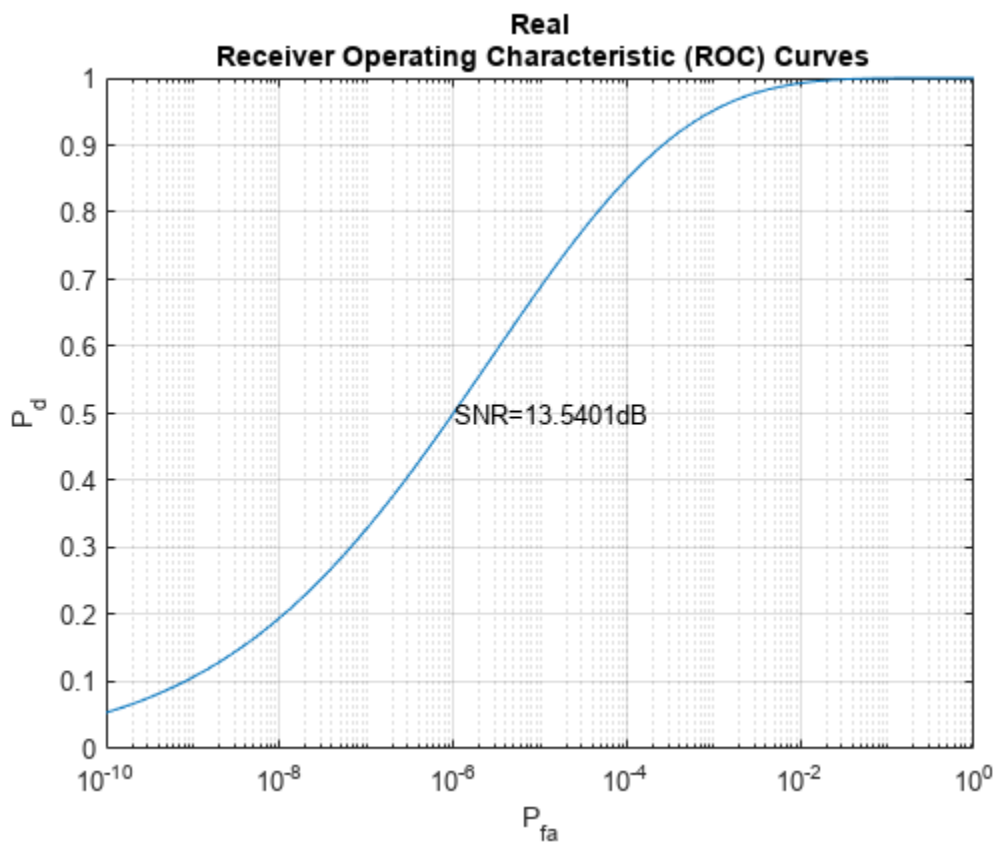


The ROC curve lets you easily read off the probability of detection for a given false-alarm rate.

You can use `rocsnr` to examine detector performance for different received signal types at a fixed SNR.

```
SNR = 13.54;
[Pd_real,Pfa_real] = rocsnr(SNR,'SignalType','real',...
    'MinPfa',1e-8);
[Pd_coh,Pfa_coh] = rocsnr(SNR,...
    'SignalType','NonfluctuatingCoherent',...
    'MinPfa',1e-8);
[Pd_noncoh,Pfa_noncoh] = rocsnr(SNR,'SignalType',...
```

```
    'NonfluctuatingNoncoherent','MinPfa',1e-8);
semilogx(Pfa_real,Pd_real)
hold on
grid on
semilogx(Pfa_coh,Pd_coh,'r')
semilogx(Pfa_noncoh,Pd_noncoh,'k')
xlabel('False-Alarm Probability')
ylabel('Probability of Detection')
legend('Real','Coherent','Noncoherent','location','southeast')
title('ROC Curve Comparison for Nonfluctuating RCS Target')
hold off
```
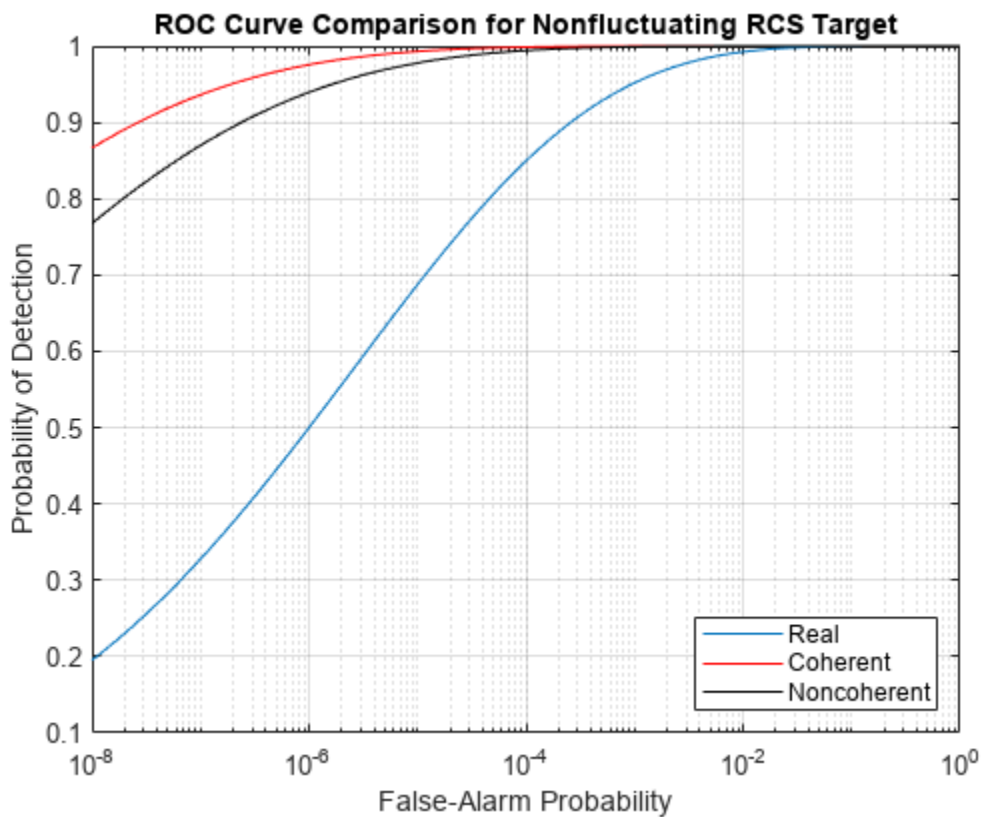


The ROC curves clearly demonstrate the superior probability of detection performance for coherent and noncoherent detectors over the real-valued case.

The `rocsnr` function accepts an SNR vector input letting you quickly examine a number of ROC curves.

```
SNRs = (6:2:12);
rocsnr(SNRs,'SignalType','NonfluctuatingNoncoherent')
```

Nonfluctuating Noncoherent
Receiver Operating Characteristic (ROC) Curves
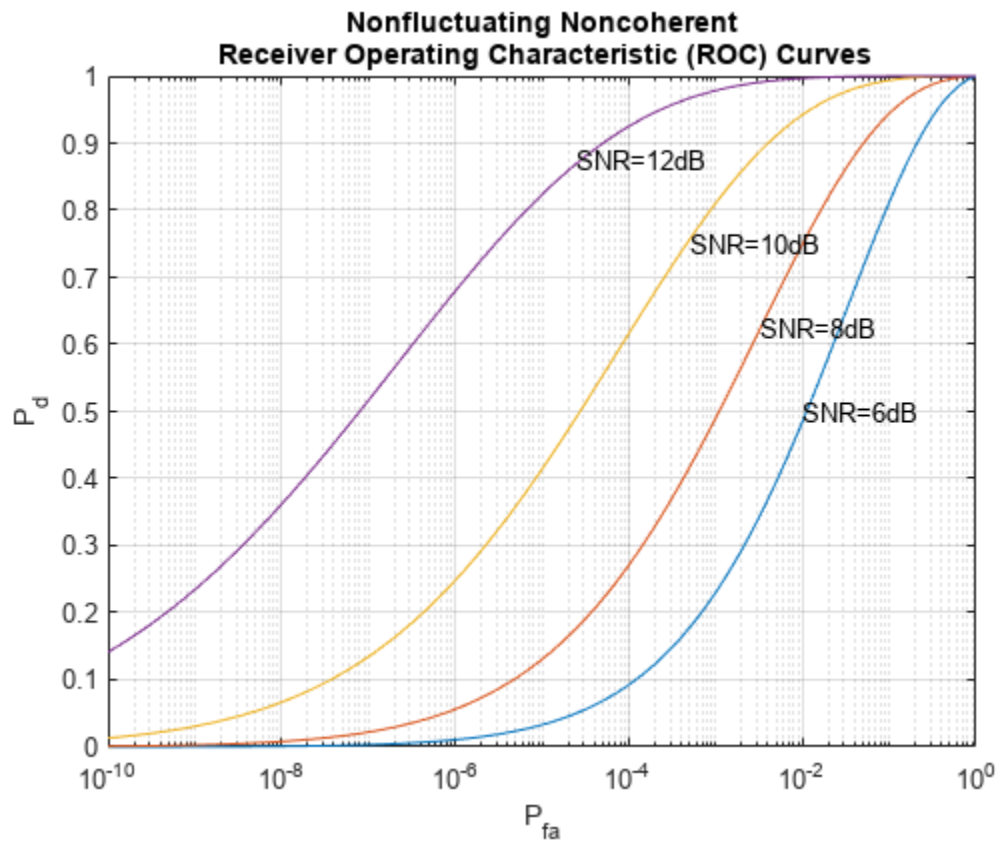
The graph shows that, as the SNR increases, the supports of the probability distributions under the null and alternative hypotheses become more disjointed. Therefore, for a given false-alarm probability, the probability of detection increases.

You can examine the probability of detection as a function of SNR for a fixed false-alarm probability with `rocpfa`. To obtain ROC curves for a Swerling I target model at false-alarm probabilities of *(1e-6,1e-4,1e-2,1e-1)*, use

```
Pfa = [1e-6 1e-4 1e-2 1e-1];
rocpfa(Pfa,'SignalType','Swerling1')
```
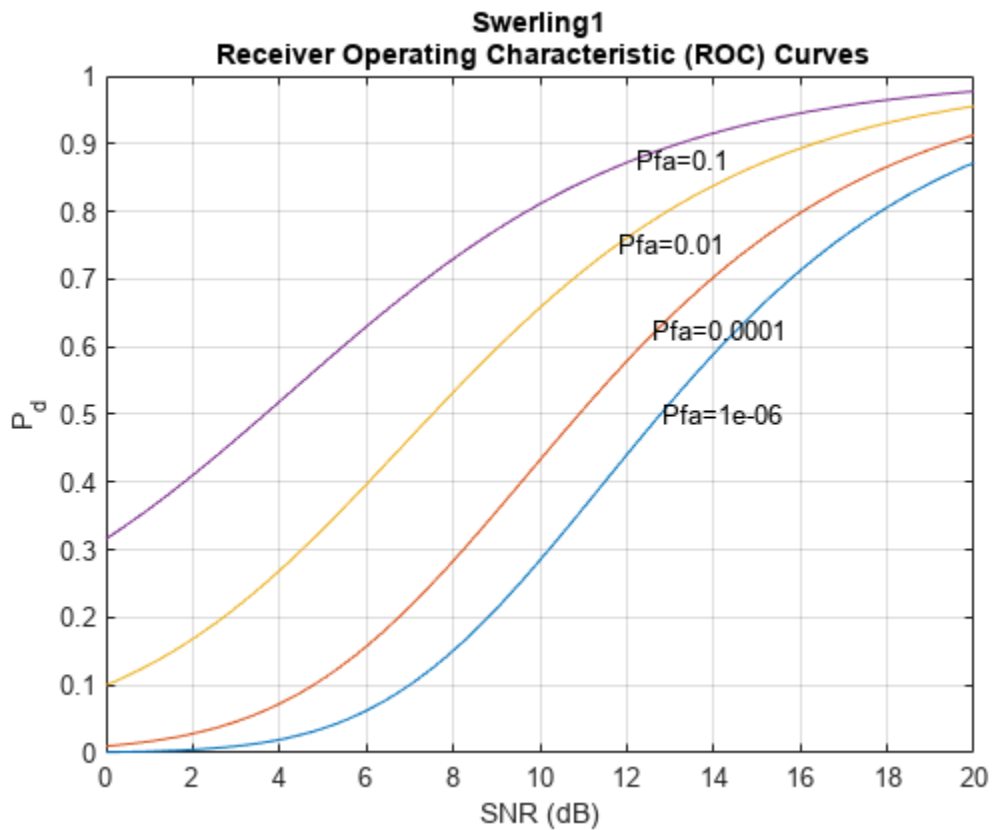
Use `rocpfa` to examine the effect of SNR on the probability of detection for a detector using noncoherent integration with a false-alarm probability of *1e-4*. Assume the target has a nonfluctuating RCS and that you are integrating over 5 pulses.

```
[Pd,SNR] = rocpfa(1e-4,...
    'SignalType','NonfluctuatingNoncoherent',...
    'NumPulses',5);
figure;
plot(SNR,Pd); xlabel('SNR (dB)');
ylabel('Probability of Detection'); grid on;
title('Nonfluctuating Noncoherent Detector (5 Pulses)');
```
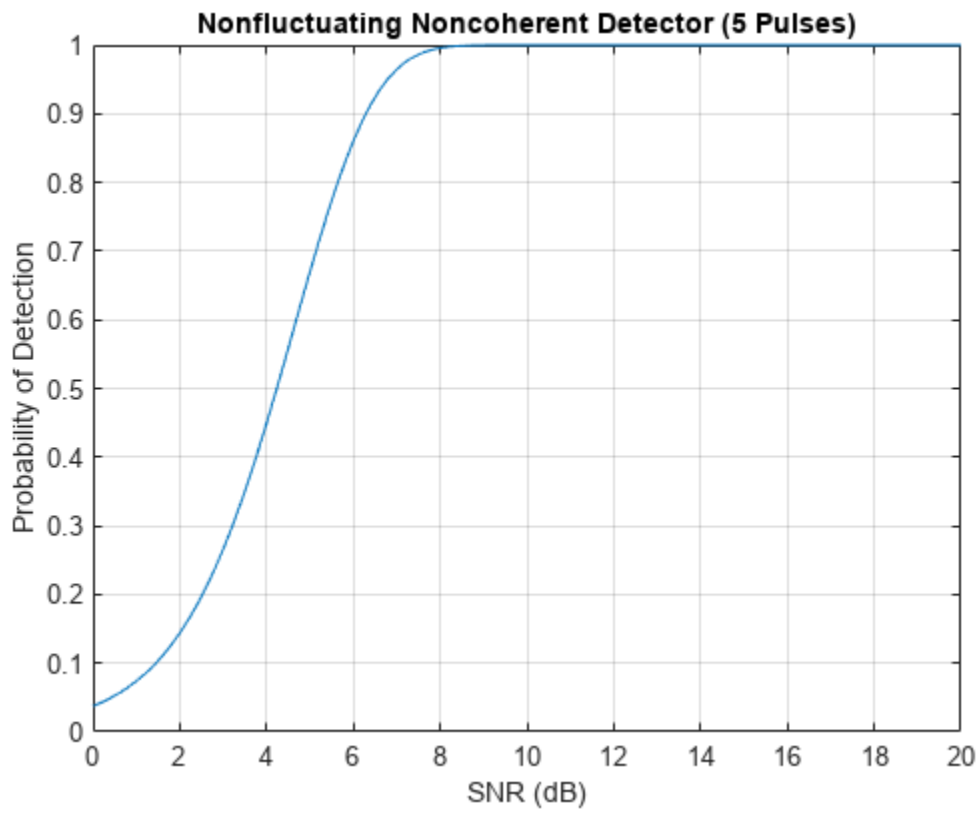
**See Also**

**Related Examples**

- Detector Performance Analysis using ROC Curves

# Monte Carlo ROC Simulation

This example shows how to generate a receiver operating characteristic (ROC) curve of a radar system using a Monte Carlo simulation. The receiver operating characteristic determines how well the system can detect targets while rejecting large spurious signal values when a target is absent (false alarms). A detection system will declare presence or absence of a target by comparing the received signal value to a preset threshold. The probability of detection (*Pd*) of a target is the probability that the instantaneous signal value is larger than the threshold whenever a target is actually present. The probability of false alarm (*Pfa*) is the probability that the signal value is larger than the threshold when a target is absent. In this case, the signal is due to noise and its properties depend on the noise statistics. The Monte Carlo simulation generates a very large number of radar returns with and without a target present. The simulation computes *Pd* and *Pfa* are by counting the proportion of signal values in each case that exceed the threshold.

A ROC curve plots *Pd* as a function of *Pfa*. The shape of a ROC curve depends on the received SNR of the signal. If the arriving signal SNR is known, then the ROC curve shows how well the system performs in terms of *Pd* and *Pfa*. If you specify *Pd* and *Pfa*, then you can determine how much power is needed to achieve this requirement.

You can use the function `rocsnr` to compute theoretical ROC curves. This example shows a ROC curve generated by a Monte Carlo simulation of a single-antenna radar system and compares that curve with a theoretical curve.

### Specify Radar Requirements

Set the desired probability of detection to be 0.9 and the probability of false alarm to be $10^{-6}$. Set the maximum range of the radar to 4000 meters and the range resolution to 50 meters. Set the actual target range to 3000 meters. Set the target radar cross-section to 1.5 square meters and set the operating frequency to 10 GHz. All computations are performed in baseband.

```
c = physconst('LightSpeed');
pd = 0.9;
pfa = 1e-6;
max_range = 4000;
target_range = 3000.0;
range_res = 50;
tgt_rcs = 1.5;
fc = 10e9;
lambda = c/fc;
```

Any simulation that computes *Pfa* and *pd* requires processing of many signals. To keep memory requirements low, process the signals in chunks of pulses. Set the number of pulses to process to 45000 and set the size of each chunk to 10000.

```
Npulse = 45000;
Npulsebuffsize = 10000;
```

### Select Waveform and Signal Parameters

Calculate the waveform pulse bandwidth using the pulse range resolution. Calculate the pulse repetition frequency from the maximum range. Because the signal is baseband, set the sampling frequency to twice the bandwidth. Calculate the pulse duration from the pulse bandwidth.

```
pulse_bw = c/(2*range_res);
prf = c/(2*max_range);
```

```
fs = 2*pulse_bw;
pulse_duration = 10/pulse_bw;
waveform = phased.LinearFMWaveform('PulseWidth',pulse_duration,...
    'SampleRate',fs,'SweepBandwidth',...
    pulse_bw,'PRF',prf);
```

Achieving a particular *Pd* and *Pfa* requires that sufficient signal power arrive at the receiver after the target reflects the signal. Compute the minimum SNR needed to achieve the specified probability of false alarm and probability of detection by using the Albersheim equation.

```
snr_min = albersheim(pd,pfa);
```

To achieve this SNR, sufficient power must be transmitted to the target. Use the radar equation to estimate the peak transmit power, `peak_power`, required to achieve the specified SNR in dB for the target at a range of 3000 meters. The received signal also depends on the target radar cross-section (RCS). which is assumed to follow a nonfluctuating model (Swerling 0). Set the radar to have identical transmit and receive gains of 20 dB. The radar equation is given

```
txrx_gain = 20;
peak_power = ((4*pi)^3*noisepow(1/pulse_duration)*target_range^4*...
    db2pow(snr_min))/(db2pow(2*txrx_gain)*tgt_rcs*lambda^2)
```

```
peak_power = 293.1827
```

**Set Up the Transmitter System Objects**

Create System objects that make up the transmission part of the simulation: radar platform, antenna, transmitter, and radiator.

```
antennaplatform = phased.Platform(...
    'InitialPosition',[0; 0; 0],...
    'Velocity',[0; 0; 0]);
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[5e9 15e9]);
transmitter = phased.Transmitter(...
    'Gain',txrx_gain,...
    'PeakPower',peak_power,...
    'InUseOutputPort',true);
radiator = phased.Radiator(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);
```

**Set Up the Target System Object**

Create a target System object™ corresponding to an actual reflecting target with a non-zero target cross-section. Reflections from this target will simulate actual radar returns. In order to compute false alarms, create a second target System object with zero radar cross section. Reflections from this target are zero except for noise.

```
target{1} = phased.RadarTarget(...
    'MeanRCS',tgt_rcs,...
    'OperatingFrequency',fc);
targetplatform{1} = phased.Platform(...
    'InitialPosition',[target_range; 0; 0]);
target{2} = phased.RadarTarget(...
    'MeanRCS',0,...
    'OperatingFrequency',fc);
```

```
targetplatform{2} = phased.Platform(...
    'InitialPosition',[target_range; 0; 0]);
```

**Set Up Free-Space Propagation System Objects**

Model the propagation environment from the radar to the targets and back.

```
channel{1} = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
channel{2} = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

**Set Up Receiver System Objects**

Specify the noise by setting the `NoiseMethod` property to `'Noise temperature'` and the `ReferenceTemperature` property to 290 K.

```
collector = phased.Collector(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);
receiver = phased.ReceiverPreamp(...
    'Gain',txrx_gain,...
    'NoiseMethod','Noise temperature',...
    'ReferenceTemperature',290.0,...
    'NoiseFigure',0,...
    'SampleRate',fs,...
    'EnableInputPort',true);
receiver.SeedSource = 'Property';
receiver.Seed = 2010;
```

**Specify Fast-Time Grid**

The fast-time grid is the set of time samples within one pulse repetition time interval. Each sample corresponds to a range bin.

```
fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
rangebins = c*fast_time_grid/2;
```

**Create Transmitted Pulse from Waveform**

Create the waveform you want to transmit.

```
wavfrm = waveform();
```

Create the transmitted signal that includes transmitted antenna gains.

```
[sigtrans,tx_status] = transmitter(wavfrm);
```

Create matched filter coefficients from the waveform System object. Then create the matched filter System object.

```
MFCoeff = getMatchedFilter(waveform);
matchingdelay = size(MFCoeff,1) - 1;
filter = phased.MatchedFilter(...
```

```
        'Coefficients',MFCoeff,...
        'GainOutputPort',false);
```

**Compute Target Range Bin**

Compute the target range, and then compute the index into the range bin array. Because the target and radar are stationary, use the same values of position and velocity throughout the simulation loop. You can assume that the range bin index is constant for the entire simulation.

```
ant_pos = antennaplatform.InitialPosition;
ant_vel = antennaplatform.Velocity;
tgt_pos = targetplatform{1}.InitialPosition;
tgt_vel = targetplatform{1}.Velocity;
[tgt_rng,tgt_ang] = rangeangle(tgt_pos,ant_pos);
rangeidx = val2ind(tgt_rng,rangebins(2)-rangebins(1),rangebins(1));
```

**Loop Over Pulses**

Create a signal processing loop. Each step is accomplished by executing the System objects. The loop processes the pulses twice, once for the target-present condition and once for target-absent condition.

**1**   Radiate the signal into space using `phased.Radiator`.

**2**   Propagate the signal to the target and back to the antenna using `phased.FreeSpace`.

**3**   Reflect the signal from the target using `phased.Target`.

**4**   Receive the reflected signals at the antenna using `phased.Collector`.

**5**   Pass the received signal though the receive amplifier using `phased.ReceiverPreamp`. This step also adds the random noise to the signal.

**6**   Match filter the amplified signal using `phased.MatchedFilter`.

**7**   Store the matched filter output at the target range bin index for further analysis.

```
rcv_pulses = zeros(length(sigtrans),Npulsebuffsize);
h1 = zeros(Npulse,1);
h0 = zeros(Npulse,1);
Nbuff = floor(Npulse/Npulsebuffsize);
Nrem = Npulse - Nbuff*Npulsebuffsize;
for n = 1:2 % H1 and H0 Hypothesis
    trsig = radiator(sigtrans,tgt_ang);
    trsig = channel{n}(trsig,...
        ant_pos,tgt_pos,...
        ant_vel,tgt_vel);
    rcvsig = target{n}(trsig);
    rcvsig = collector(rcvsig,tgt_ang);

    for k = 1:Nbuff
        for m = 1:Npulsebuffsize
            rcv_pulses(:,m) = receiver(rcvsig,~(tx_status>0));
        end
        rcv_pulses = filter(rcv_pulses);
        rcv_pulses = buffer(rcv_pulses(matchingdelay+1:end),size(rcv_pulses,1));
        if n == 1
            h1((1:Npulsebuffsize) + (k-1)*Npulsebuffsize) = rcv_pulses(rangeidx,:).';
        else
            h0((1:Npulsebuffsize) + (k-1)*Npulsebuffsize) = rcv_pulses(rangeidx,:).';
        end
```

```
        end
        if (Nrem > 0)
            for m = 1:Nrem
                rcv_pulses(:,m) = receiver(rcvsig,~(tx_status>0));
            end
            rcv_pulses = filter(rcv_pulses);
            rcv_pulses = buffer(rcv_pulses(matchingdelay+1:end),size(rcv_pulses,1));
             if n == 1
                h1((1:Nrem) + Nbuff*Npulsebuffsize) = rcv_pulses(rangeidx,1:Nrem).';
            else
                h0((1:Nrem) + Nbuff*Npulsebuffsize) = rcv_pulses(rangeidx,1:Nrem).';
             end
        end
end
```

**Create Histogram of Matched Filter Outputs**

Compute histograms of the target-present and target-absent returns. Use 100 bins to give a rough estimate of the spread of signal values. Set the range of histogram values from the smallest signal to the largest signal.

```
h1a = abs(h1);
h0a = abs(h0);
thresh_low = min([h1a;h0a]);
thresh_hi  = max([h1a;h0a]);
nbins = 100;
binedges = linspace(thresh_low,thresh_hi,nbins);
figure
histogram(h0a,binedges)
hold on
histogram(h1a,binedges)
hold off
title('Target-Absent Vs Target-Present Histograms')
legend('Target Absent','Target Present')
```
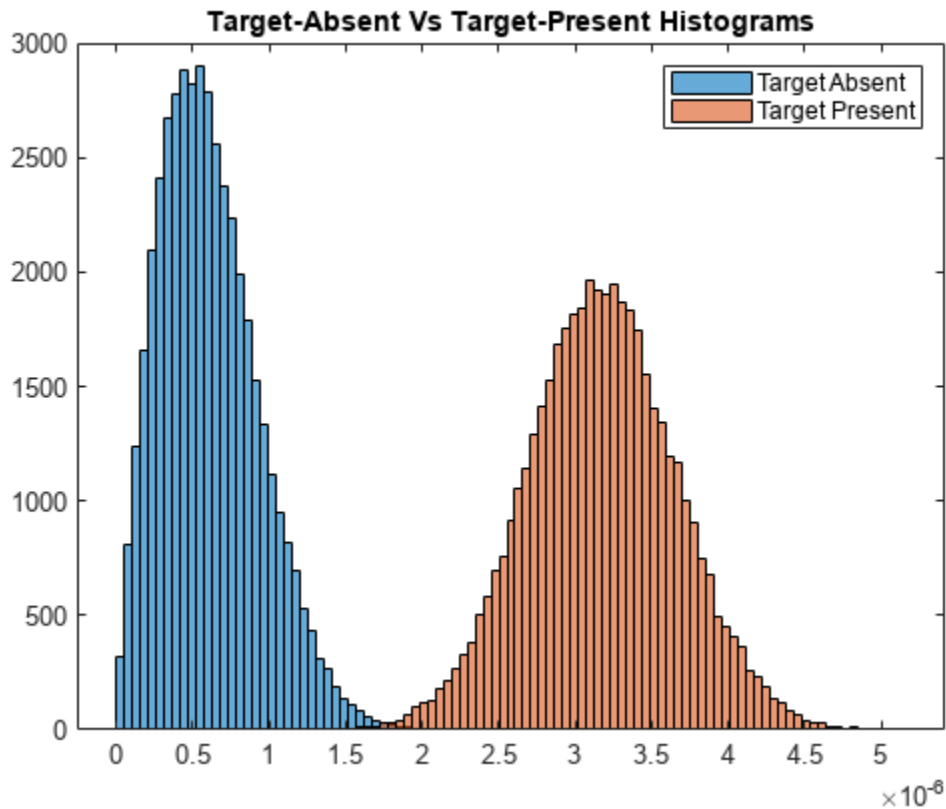
**Compare Simulated and Theoretical *Pd* and *Pfa***

To compute *Pd* and *Pfa*, calculate the number of instances that a target-absent return and a target-present return exceed a given threshold. This set of thresholds has a finer granularity than the bins used to create the histogram in the previous simulation. Then, normalize these counts by the number of pulses to get an estimate of the probabilities. The vector `sim_pfa` is the simulated probability of false alarm as a function of the threshold, `thresh`. The vector `sim_pd` is the simulated probability of detection, also a function of the threshold. The receiver sets the threshold so that it can determine whether a target is present or absent. The histogram above suggests that the best threshold is around 1.8.

```
nbins = 1000;
thresh_steps = linspace(thresh_low,thresh_hi,nbins);
sim_pd = zeros(1,nbins);
sim_pfa = zeros(1,nbins);
for k = 1:nbins
    thresh = thresh_steps(k);
    sim_pd(k) = sum(h1a >= thresh);
    sim_pfa(k) = sum(h0a >= thresh);
end
sim_pd = sim_pd/Npulse;
sim_pfa = sim_pfa/Npulse;
```

To plot the experimental ROC curve, you must invert the Pfa curve so that you can plot *Pd* against *Pfa*. You can invert the *Pfa* curve only when you can express *Pfa* as a strictly monotonic decreasing function of `thresh`. To express *Pfa* this way, find all array indices where the *Pfa* is the constant over neighboring indices. Then, remove these values from the *Pd* and *Pfa* arrays.

```
pfa_diff = diff(sim_pfa);
idx = (pfa_diff == 0);
sim_pfa(idx) = [];
sim_pd(idx) = [];
```

Limit the smallest Pfa to $10^{-6}$.

```
minpfa = 1e-6;
N = sum(sim_pfa >= minpfa);
sim_pfa = fliplr(sim_pfa(1:N)).';
sim_pd = fliplr(sim_pd(1:N)).';
```

Compute the theoretical *Pfa* and *Pd* values from the smallest *Pfa* to 1. Then plot the theoretical *Pfa* curve.

```
[theor_pd,theor_pfa] = rocsnr(snr_min,'SignalType',...
    'NonfluctuatingNoncoherent',...
    'MinPfa',minpfa,'NumPoints',N,'NumPulses',1);
semilogx(theor_pfa,theor_pd)
hold on
semilogx(sim_pfa,sim_pd,'r.')
title('Simulated and Theoretical ROC Curves')
xlabel('Pfa')
ylabel('Pd')
grid on
legend('Theoretical','Simulated','Location','SE')
```
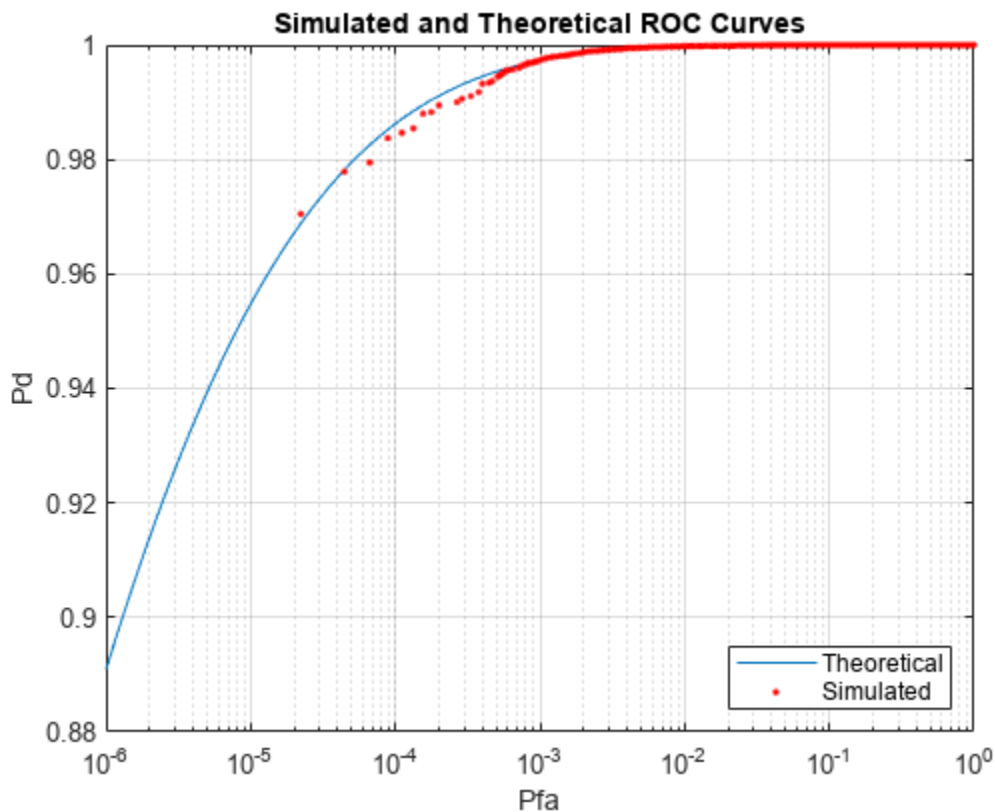
**Improve Simulation Using One Million Pulses**

In the preceding simulation, *Pd* values at low *Pfa* do not fall along a smooth curve and do not even extend down to the specified operating regime. The reason for this is that at very low *Pfa* levels, very few, if any, samples exceed the threshold. To generate curves at low *Pfa*, you must use a number of samples on the order of the inverse of *Pfa*. This type of simulation takes a long time. The following curve uses one million pulses instead of 45,000. To run this simulation, repeat the example, but set `Npulse` to 1000000.
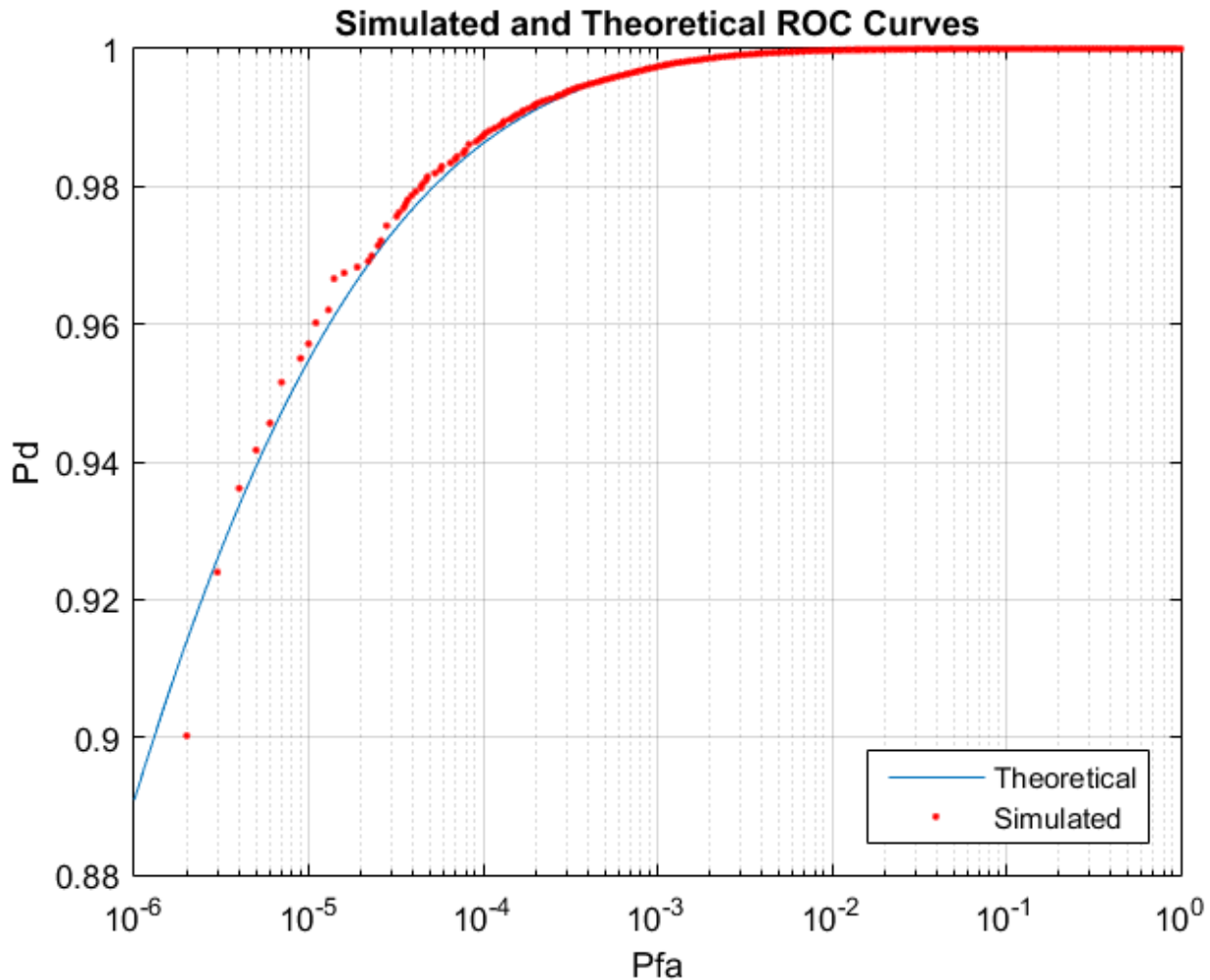
# Matched Filtering

| In this section... |
| --- |
| "Reasons for Using Matched Filtering" on page 8-18 |
| "Support for Matched Filtering" on page 8-18 |
| "Matched Filtering of Linear FM Waveform" on page 8-18 |
| "Matched Filtering to Improve SNR for Target Detection" on page 8-20 |

## Reasons for Using Matched Filtering

You can see from the results in "Receiver Operating Characteristics" on page 8-5 that the probability of detection increases with increasing SNR. For a deterministic signal in white Gaussian noise, you can maximize the SNR at the receiver by using a filter matched to the signal. The matched filter is a time-reversed and conjugated version of the signal. The matched filter is shifted to be causal.

## Support for Matched Filtering

Use `phased.MatchedFilter` to implement a matched filter.

When you use `phased.MatchedFilter`, you can customize characteristics of the matched filter such as the matched filter coefficients and window for spectrum weighting. If you apply spectrum weighting, you can specify the coverage region and coefficient sample rate; Taylor, Chebyshev, and Kaiser windows have additional properties you can specify.

## Matched Filtering of Linear FM Waveform

This example compares the results of matched filtering with and without spectrum weighting. Spectrum weighting is often used with linear FM waveforms to reduce the sidelobes in the time domain.

Create a linear FM waveform with a duration of 0.1 milliseconds, a sweep bandwidth of 100 kHz, and a pulse repetition frequency of 5 kHz. Add noise to the linear FM pulse and filter the noisy signal using a matched filter. This example applies a matched filter with and without spectrum weighting.

Specify the waveform.

```
waveform = phased.LinearFMWaveform('PulseWidth',1e-4,'PRF',5e3,...
    'SampleRate',1e6,'OutputFormat','Pulses','NumPulses',1,...
    'SweepBandwidth',1e5);
wav = getMatchedFilter(waveform);
```

Create a matched filter with no spectrum weighting, and a matched filter that uses a Taylor window for spectrum weighting.

```
filter = phased.MatchedFilter('Coefficients',wav);
taylorfilter = phased.MatchedFilter('Coefficients',wav,...
    'SpectrumWindow','Taylor');
```

Create the signal and add noise.

```
sig = waveform();
rng(17)
x = sig + 0.5*(randn(length(sig),1) + 1j*randn(length(sig),1));
```

Filter the noisy signal separately with each of the filters.

```
y = filter(x);
y_taylor = taylorfilter(x);
```

Plot the real parts of the waveform and noisy signal.

```
t = linspace(0,numel(sig)/waveform.SampleRate,...
    waveform.SampleRate/waveform.PRF);
subplot(2,1,1)
plot(t,real(sig))
title('Input Signal')
xlim([0 max(t)])
grid on
ylabel('Amplitude')
subplot(2,1,2)
plot(t,real(x))
title('Input Signal + Noise')
xlim([0 max(t)])
grid on
xlabel('Time (sec)')
ylabel('Amplitude')
```
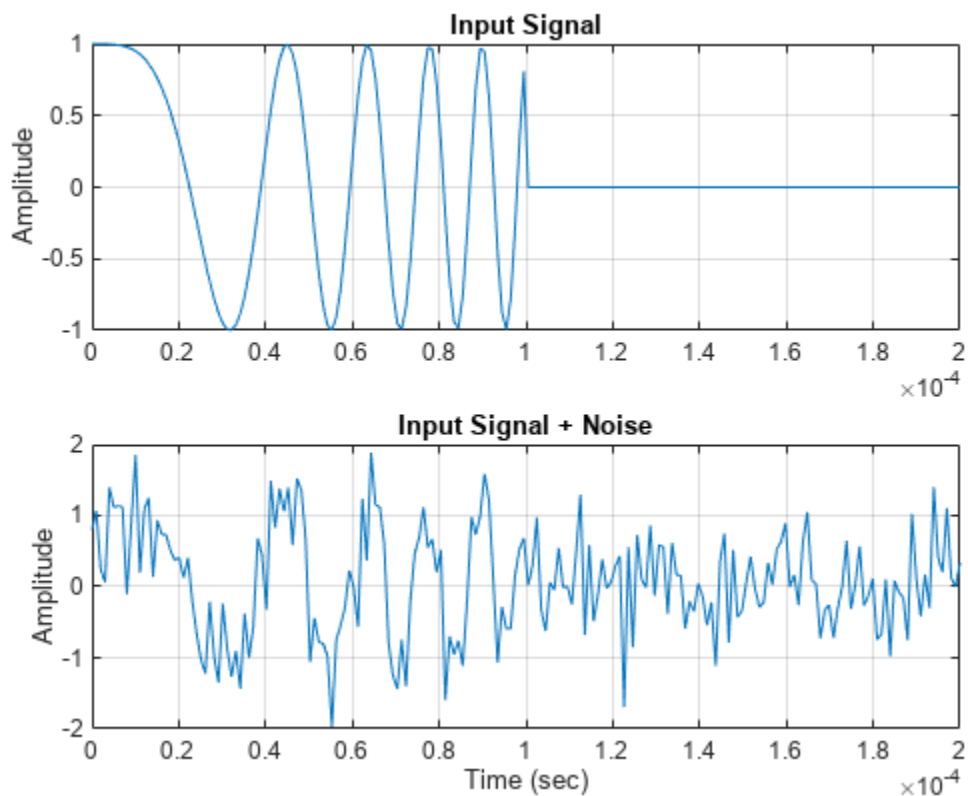


Plot the magnitudes of the two matched filter outputs.

```
plot(t,abs(y),'b--')
title('Matched Filter Output')
xlim([0 max(t)])
grid on
hold on
plot(t,abs(y_taylor),'r-')
ylabel('Magnitude')
xlabel('Seconds')
legend('No Spectrum Weighting','Taylor Window')
hold off
```
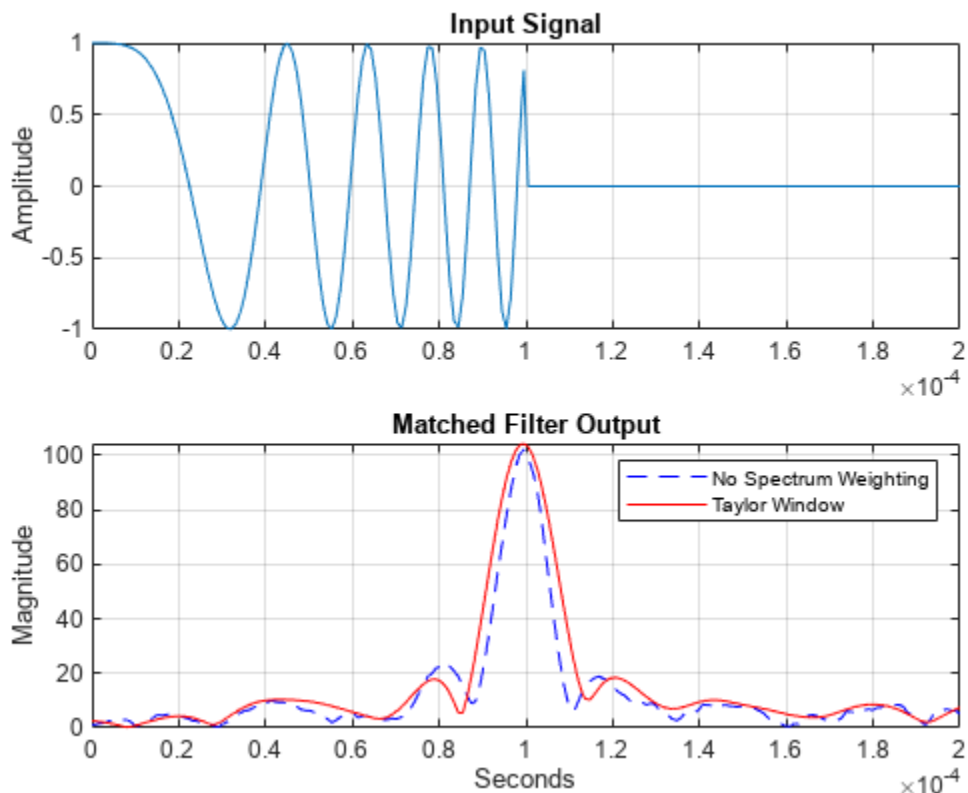


## Matched Filtering to Improve SNR for Target Detection

This example shows how to improve the SNR by performing matched filtering.

Place an isotropic antenna element at the global origin *(0;0;0)*. Then, place a target with a nonfluctuating RCS of 1 square meter approximately 7 km from the transmitter at *(5000;5000;10)*. Set the operating (carrier) frequency to 10 GHz. To simulate a monostatic radar, set the `InUseOutputPort` property on the transmitter to `true`. Calculate the range and angle from the transmitter to the target.

```
antenna = phased.IsotropicAntennaElement('FrequencyRange',[5e9 15e9]);
transmitter = phased.Transmitter('Gain',20,'InUseOutputPort',true);
fc = 10e9;
```

```
target = phased.RadarTarget('Model','Nonfluctuating',...
    'MeanRCS',1,'OperatingFrequency',fc);
txloc = [0;0;0];
tgtloc = [5000;5000;10];
transmitterplatform = phased.Platform('InitialPosition',txloc);
targetplatform = phased.Platform('InitialPosition',tgtloc);
[tgtrng,tgtang] = rangeangle(targetplatform.InitialPosition,...
    transmitterplatform.InitialPosition);
```

Create a rectangular pulse waveform 25 μs in duration with a PRF of 10 kHz. Use a single pulse for this example. Determine the maximum unambiguous range for the given PRF. Use the radar equation to determine the peak power required to detect a target. This target has an RCS of 1 square meter at the maximum unambiguous range for the transmitter operating frequency and gain. The SNR is based on a desired false-alarm rate of 1e-6 for a noncoherent detector.

```
waveform = phased.RectangularWaveform('PulseWidth',25e-6,...
    'OutputFormat','Pulses','PRF',10e3,'NumPulses',1);
c = physconst('LightSpeed');
maxrange = c/(2*waveform.PRF);
SNR = npwgnthresh(1e-6,1,'noncoherent');

lambda = physconst('LightSpeed')/target.OperatingFrequency;
Ts = 290;
dbterms = db2pow(SNR - 2*transmitter.Gain);
Pt = (4*pi)^3*physconst('Boltzmann')*Ts/waveform.PulseWidth/target.MeanRCS/(lambda^2)*maxrange^4
```

Set the peak transmit power to the output value from the radar equation.

```
transmitter.PeakPower = Pt;
```

Create radiator and collector objects that operate at 10 GHz. Create a free space path for the propagation of the pulse to and from the target. Then, create a receiver and a matched filter for the rectangular waveform.

```
radiator = phased.Radiator('PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
channel = phased.FreeSpace('PropagationSpeed',c,...
    'OperatingFrequency',fc,'TwoWayPropagation',false);
collector = phased.Collector('PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SeedSource','Property','Seed',2e3);
filter = phased.MatchedFilter(...
    'Coefficients',getMatchedFilter(waveform),...
    'GainOutputPort',true);
```

After you create all the objects that define your model, you can propagate the pulse to and from the target. Collect the echo at the receiver, and implement the matched filter to improve the SNR.

Generate waveform.

```
wf = waveform();
```

Transmit waveform.

```
[wf,txstatus] = transmitter(wf);
```

Radiate pulse toward the target.

```
wf = radiator(wf,tgtang);
```

Propagate pulse toward the target.

```
wf = channel(wf,txloc,tgtloc,[0;0;0],[0;0;0]);
```

Reflect it off the target.

```
wf = target(wf);
```

Propagate the pulse back to transmitter.

```
wf = channel(wf,tgtloc,txloc,[0;0;0],[0;0;0]);
```

Collect the echo.

```
wf = collector(wf,tgtang);
```

Receive target echo.

```
rx_puls = receiver(wf,~txstatus);
[mf_puls,mfgain] = filter(rx_puls);
```

Get group delay of matched filter.

```
Gd = length(filter.Coefficients)-1;
```

The group delay is constant.

Shift the matched filter output.

```
mf_puls=[mf_puls(Gd+1:end); mf_puls(1:Gd)];
subplot(2,1,1)
t = unigrid(0,1e-6,1e-4,'[)');
rangegates = c.*t;
rangegates = rangegates/2;
plot(rangegates,abs(rx_puls))
title('Received Pulse')
ylabel('Amplitude')
hold on
plot([tgtrng, tgtrng], [0 max(abs(rx_puls))],'r')
subplot(2,1,2)
plot(rangegates,abs(mf_puls))
title('With Matched Filtering')
xlabel('Meters')
ylabel('Amplitude')
hold on
plot([tgtrng, tgtrng], [0 max(abs(mf_puls))],'r')
hold off
```
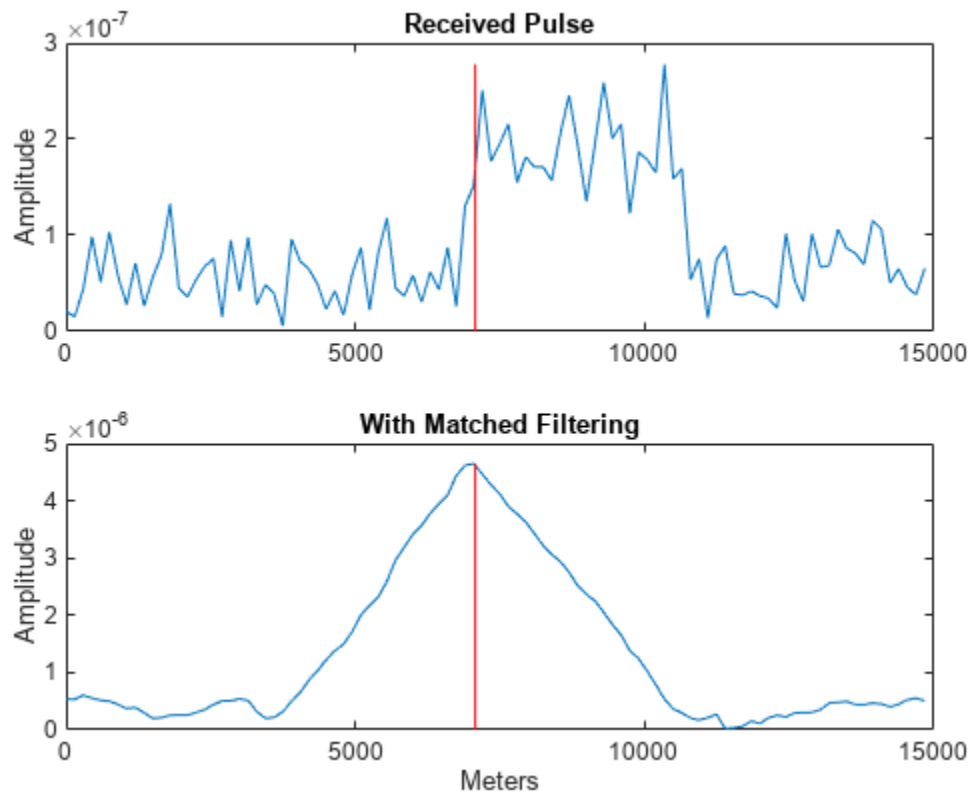
# Stretch Processing

| **In this section...** |
| --- |
| "Reasons for Using Stretch Processing" on page 8-24 |
| "Support for Stretch Processing" on page 8-24 |
| "Stretch Processing Procedure" on page 8-24 |

## Reasons for Using Stretch Processing

The linear FM waveform is popular in radar systems because its large time-bandwidth product can provide good range resolution. However, the large bandwidth of this waveform makes digital matched filtering difficult because it requires expensive, high-quality analog-to-digital converters. Stretch processing, also known as deramping or dechirping, is an alternative to matched filtering. Stretch processing provides pulse compression by looking for the return within a predefined range interval of interest. Stretch processing typically occurs in the analog domain. Unlike matched filtering, stretch processing reduces the bandwidth requirement of subsequent processing.

## Support for Stretch Processing

The `phased.StretchProcessor` System object implements stretch processing. You can use this object as part of a simulation that uses `phased.LinearFMWaveform` or directly with your own data.

## Stretch Processing Procedure

The typical procedure for stretch processing is as follows:

1   Choose a range interval of interest, centered on a reference range. Stretch processing focuses on this interval instead of the entire range span that the pulse can cover.

2   Define and configure a stretch processor object. The configuration includes the reference range, length of the range interval of interest, characteristics of the linear FM waveform, and signal propagation speed.

  •  If you are using a `phased.LinearFMWaveform` object to implement the linear FM waveform, use the `getStretchProcessor` method to define and automatically configure a stretch processor object.

  •  Otherwise, create a `phased.StretchProcessor` object directly, and set its properties as needed.

3   Perform stretch processing by calling the `step` method on your stretch processor object. You provide your received signal as an input argument. The `step` method generates a reference signal and correlates it with your received signal.

4   Compute a periodogram of the output from `step`, and identify the peak frequencies. You can use the following features to help you perform this step:

  •  `periodogram`
  •  `psd`
  •  `findpeaks`

5   Convert each peak frequency to the corresponding range value, using the `stretchfreq2rng` function.

## See Also

`phased.StretchProcessor` | `phased.LinearFMWaveform` | `stretchfreq2rng` | `periodogram` | `findpeaks`

## Related Examples

*   Range Estimation Using Stretch Processing

# FMCW Range Estimation

The purpose of FMCW range estimation is to estimate the range of a target. For example, a radar for collision avoidance in an automobile needs to estimate the distance to the nearest obstacle. FMCW range estimation algorithms can vary in the details, but the typical high-level procedure is as follows:

1   **Dechirp** — Dechirp the received signal by mixing it with the transmitted signal. If you use the `dechirp` function, the transmitted signal is the reference signal.

2   **Find beat frequency** — From the dechirped signal, extract the beat frequency or pair of beat frequencies. If the FMCW signal has a sawtooth shape (up-sweep or down-sweep sawtooth shape), you extract one beat frequency. If the FMCW signal has a triangular sweep, you extract up-sweep and down-sweep beat frequencies.

Extracting beat frequencies can use a variety of algorithms. For example, you can use the following features to help you perform this step:

- `pwelch` or `periodogram`
- `psd`
- `findpeaks`
- `rootmusic`
- `phased.CFARDetector`

3   **Compute range** — Use the beat frequency or frequencies to compute the corresponding range value. The `beat2range` function can perform this computation.

While developing your algorithm, you might also perform these auxiliary tasks:

- Visualize targets in the range-Doppler domain, using the `phased.RangeDopplerResponse` System object.

- Determine whether you need to compensate for range-Doppler coupling. Such coupling can occur if the target is moving relative to the radar. You can use the `rdcoupling` function to compute the range offset due to range-Doppler coupling. If the range offset is not negligible, common compensation techniques include:

  - Subtracting the range offset from your initial range estimate
  - Having the FMCW signal use a triangle sweep instead of an up sweep or down sweep

- Explore the relationships among your system's range requirements and parameters of the FMCW waveform. You can use these functions:

  - `range2time`
  - `time2range`
  - `range2bw`

## See Also

`phased.FMCWWaveform` | `phased.RangeDopplerResponse` | `time2range` | `range2time` | `range2bw` | `dechirp` | `beat2range` | `range2beat` | `rdcoupling` | `periodogram` | `findpeaks` | `pwelch` | `rootmusic`

## Related Examples

- *"Automotive Adaptive Cruise Control Using FMCW Technology"* (Radar Toolbox)

# Range-Doppler Response

## Benefits of Producing Range-Doppler Response

Visualizing a signal in the range-Doppler domain can help you intuitively understand connections among targets. From a range-Doppler map, you can:

- See how far away the targets are and how quickly they are approaching or receding.
- Distinguish among targets moving at various speeds at various ranges, in particular:

  - If a transmitter platform is stationary, a range-Doppler map shows a response from stationary targets at zero Doppler.
  - For targets that are moving relative to the transmitter platform, the range-Doppler map shows a response at nonzero Doppler values.

You can also use the range-Doppler response in nonvisual ways. For example, you can perform peak detection in the range-Doppler domain and use the information to resolve the range-Doppler coupling of an FMCW radar system.

## Support for Range-Doppler Processing

You can use the `phased.RangeDopplerResponse` object to compute and visualize the range-Doppler response of input data. This object performs range processing in fast time, followed by Doppler processing in slow time. The object configuration and syntax typically depend on the kind of radar system.

### Pulsed Radar Systems

This procedure is used typically to produce a range-Doppler response for a pulsed radar system. (In the special case of linear FM pulses, the procedure in "FMCW Radar Systems" on page 8-29 is an alternative option.)

1 Create a `phased.RangeDopplerResponse` object, setting the `RangeMethod` property to `'Matched Filter'`.

2 Customize these characteristics, or accept default values for any of them:

- Signal propagation speed
- Sample rate
- Length of the FFT for Doppler processing
- Characteristics of the window for Doppler weighting, if any
- Doppler domain output preference in terms of radial speed or Doppler shift frequency. (If you select radial speed, also specify the signal carrier frequency.)

3 Organize your data, x, into a matrix. The columns in this matrix correspond to separate, consecutive pulses.

**4** Use `plotResponse` to plot the range-Doppler response or `step` to obtain data representing the range-Doppler response. Include x and matched filter coefficients in your syntax when you call `plotResponse` or `step`.

For examples, see the `step` reference page or "Range-Speed Response Pattern of Target" on page 8-29.

**FMCW Radar Systems**

This procedure is used typically to produce a range-Doppler response for an FMCW radar system. You can also use this procedure for a system that uses linear FM pulsed signals. In the case of pulsed signals, you typically use stretch processing to dechirp the signal.

**1** Create a `phased.RangeDopplerResponse` object, setting the `RangeMethod` property to `'Dechirp'`.

**2** Customize these characteristics, or accept default values for any of them:

- Signal propagation speed
- Sample rate
- FM sweep slope
- Whether the processor should dechirp or decimate your signal
- Length of the FFT for range processing. The algorithm performs an FFT to translate the dechirped data into the beat frequency domain, which provides range information.
- Characteristics of the window for range weighting, if any
- Length of the FFT for Doppler processing
- Characteristics of the window for Doppler weighting, if any
- Doppler domain output preference in terms of radial speed or Doppler shift frequency. (If you select radial speed, also specify the signal carrier frequency.)

**3** Organize your data, x, into a matrix in which the columns correspond to sweeps or pulses that are separate and consecutive.

In the case of an FMCW waveform with a triangle sweep, the sweeps alternate between positive and negative slopes. However, `phased.RangeDopplerResponse` is designed to process consecutive sweeps of the same slope. To apply `phased.RangeDopplerResponse` for a triangle-sweep system, use one of the following approaches:

- Specify a positive `SweepSlope` property value, with x corresponding to upsweeps only. The true values of Doppler or speed are half of what `step` returns or `plotResponse` plots.
- Specify a negative `SweepSlope` property value, with x corresponding to downsweeps only. The true values of Doppler or speed are half of what `step` returns or `plotResponse` plots.

**4** Use `plotResponse` to plot the range-Doppler response or `step` to obtain data representing the range-Doppler response. Include x in the syntax when you call `plotResponse` or `step`. If your data is not already dechirped, also include a reference signal in the syntax.

For an example, see the `plotResponse` reference page.

## Range-Speed Response Pattern of Target

This example shows how to visualize the speed and range of a target in a pulsed radar system that uses a rectangular waveform.

Place an isotropic antenna element at the global origin *(0,0,0)*. Then, place a target with a nonfluctuating RCS of 1 square meter at *(5000,5000,10)*, which is approximately 7 km from the transmitter. Set the operating (carrier) frequency to 10 GHz. To simulate a monostatic radar, set the `InUseOutputPort` property on the transmitter to `true`. Calculate the range and angle from the transmitter to the target.

```
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[5e9 15e9]);
transmitter = phased.Transmitter('Gain',20,'InUseOutputPort',true);
fc = 10e9;
target = phased.RadarTarget('Model','Nonfluctuating',...
    'MeanRCS',1,'OperatingFrequency',fc);
txloc = [0;0;0];
tgtloc = [5000;5000;10];
antennaplatform = phased.Platform('InitialPosition',txloc);
targetplatform = phased.Platform('InitialPosition',tgtloc);
[tgtrng,tgtang] = rangeangle(targetplatform.InitialPosition,...
    antennaplatform.InitialPosition);
```

Create a rectangular pulse waveform 2µs in duration with a PRF of 10 kHz. Determine the maximum unambiguous range for the given PRF. Use the radar equation to determine the peak power required to detect a target. This target has an RCS of 1 square meter at the maximum unambiguous range for the transmitter operating frequency and gain. The SNR is based on a desired false-alarm rate of $1e^{-6}$ for a noncoherent detector.

```
waveform = phased.RectangularWaveform('PulseWidth',2e-6,...
    'OutputFormat','Pulses','PRF',1e4,'NumPulses',1);
c = physconst('LightSpeed');
maxrange = c/(2*waveform.PRF);
SNR = npwgnthresh(1e-6,1,'noncoherent');
lambda = c/target.OperatingFrequency;
maxrange = c/(2*waveform.PRF);
tau = waveform.PulseWidth;
Ts = 290;
dbterm = db2pow(SNR - 2*transmitter.Gain);
Pt = (4*pi)^3*physconst('Boltzmann')*Ts/tau/target.MeanRCS/lambda^2*maxrange^4*dbterm;
```

Set the peak transmit power to the value obtained from the radar equation.

```
transmitter.PeakPower = Pt;
```

Create radiator and collector objects that operate at 10 GHz. Create a free space path for the propagation of the pulse to and from the target. Then, create a receiver.

```
radiator = phased.Radiator(...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
channel = phased.FreeSpace(...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,'TwoWayPropagation',false);
collector = phased.Collector(...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SeedSource','Property','Seed',2e3);
```

Propagate 25 pulses to and from the target. Collect the echoes at the receiver, and store them in a 25-column matrix named rx_puls.

```
numPulses = 25;
rx_puls = zeros(100,numPulses);
```

Simulation loop

```
for n = 1:numPulses
```

Generate waveform

```
    wf = waveform();
```

Transmit waveform

```
    [wf,txstatus] = transmitter(wf);
```

Radiate pulse toward the target

```
    wf = radiator(wf,tgtang);
```

Propagate pulse toward the target

```
    wf = channel(wf,txloc,tgtloc,[0;0;0],[0;0;0]);
```

Reflect it off the target

```
    wf = target(wf);
```

Propagate the pulse back to transmitter

```
    wf = channel(wf,tgtloc,txloc,[0;0;0],[0;0;0]);
```

Collect the echo

```
    wf = collector(wf,tgtang);
```

Receive the target echo

```
    rx_puls(:,n) = receiver(wf,~txstatus);
```

```
end
```

Create a range-Doppler response object that uses the matched filter approach. Configure this object to show radial speed rather than Doppler frequency. Use plotResponse to plot the range versus speed.

```
rangedoppler = phased.RangeDopplerResponse(...
    'RangeMethod','Matched Filter',...
    'PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc);
plotResponse(rangedoppler,rx_puls,getMatchedFilter(waveform))
```
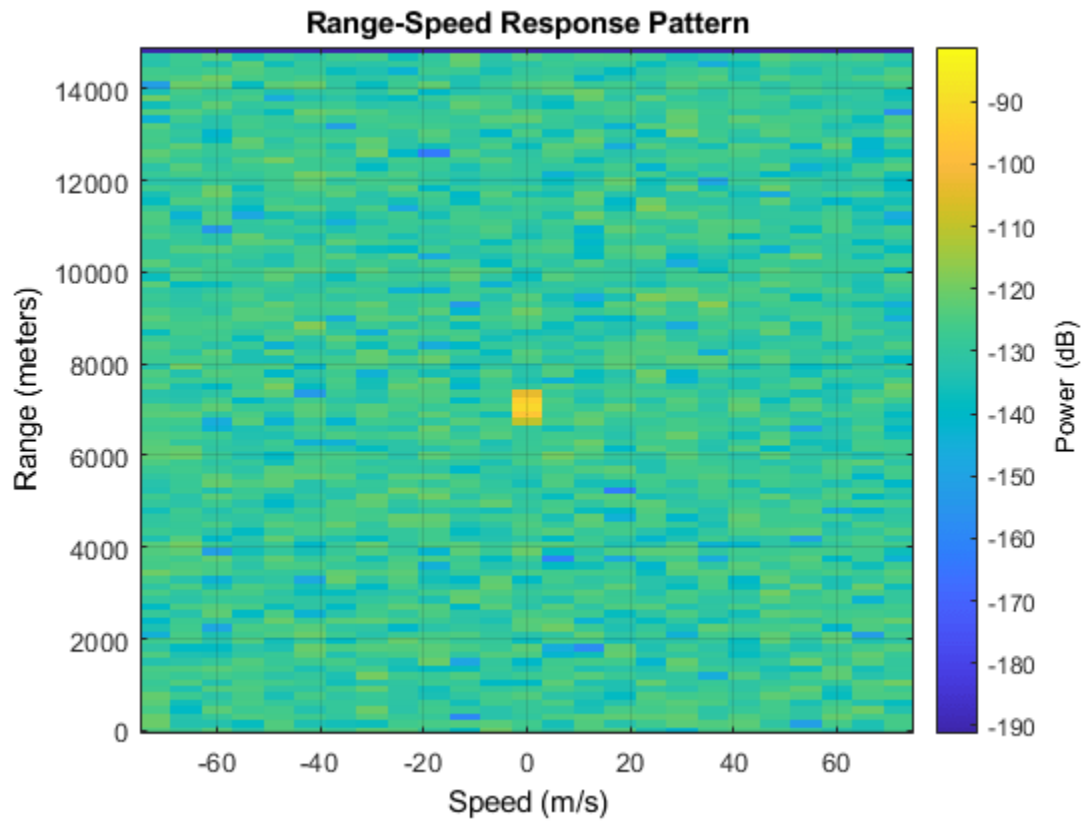
The plot shows the stationary target at a range of approximately 7000 m.

## See Also

phased.RangeDopplerResponse

## Related Examples

- "Automotive Adaptive Cruise Control Using FMCW Technology" (Radar Toolbox)

# Constant False-Alarm Rate (CFAR) Detectors

| In this section... |
|---|
| |
| |
| |
| |
| |

## False Alarm Rate for CFAR Detectors

In the Neyman-Pearson framework, the probability of detection is maximized subject to the constraint that the false-alarm probability does not exceed a specified level. The false-alarm probability depends on the noise variance. Therefore, to calculate the false-alarm probability, you must first estimate the noise variance. If the noise variance changes, you must adjust the threshold to maintain a constant false-alarm rate. *Constant false-alarm rate detectors* implement adaptive procedures that enable you to update the threshold level of your test when the power of the interference changes.

To motivate the need for an adaptive procedure, assume a simple binary hypothesis test where you must decide between the signal-absent and signal-present hypotheses for a single sample. The signal has amplitude 4, and the noise is zero-mean Gaussian with unit variance.

$$H_0 : x = w, \quad w \sim N(0,1)$$

$$H_1 : x = 4 + w$$

First, set the false-alarm rate to 0.001 and determine the threshold.

```
T = npwgnthresh(1e-3,1,'real');
threshold = sqrt(db2pow(T))
```

```
threshold =

    3.0902
```

Check that this threshold yields the desired false-alarm rate probability and then compute the probability of detection.

```
pfa = 0.5*erfc(threshold/sqrt(2))
pd = 0.5*erfc((threshold-4)/sqrt(2))
```

```
pfa =

   1.0000e-03
```

```
pd =
```

```
    0.8185
```

Next, assume that the noise power increases by 6.02 dB, doubling the noise variance. If your detector does not adapt to this increase in variance by determining a new threshold, your false-alarm rate increases significantly.

```
pfa = 0.5*erfc(threshold/2)
```

```
pfa =

    0.0144
```

The following figure shows the effect of increasing the noise variance on the false-alarm probability for a fixed threshold.

```
noisevar = 1:0.1:10;
noisepower = 10*log10(noisevar);
pfa = 0.5*erfc(threshold./sqrt(2*noisevar));
semilogy(noisepower,pfa./1e-3)
grid on
title('Increase in P_{FA} due to Noise Variance')
ylabel('Increase in P_{FA} (Orders of Magnitude)')
xlabel('Noise Power Increase (dB)')
```
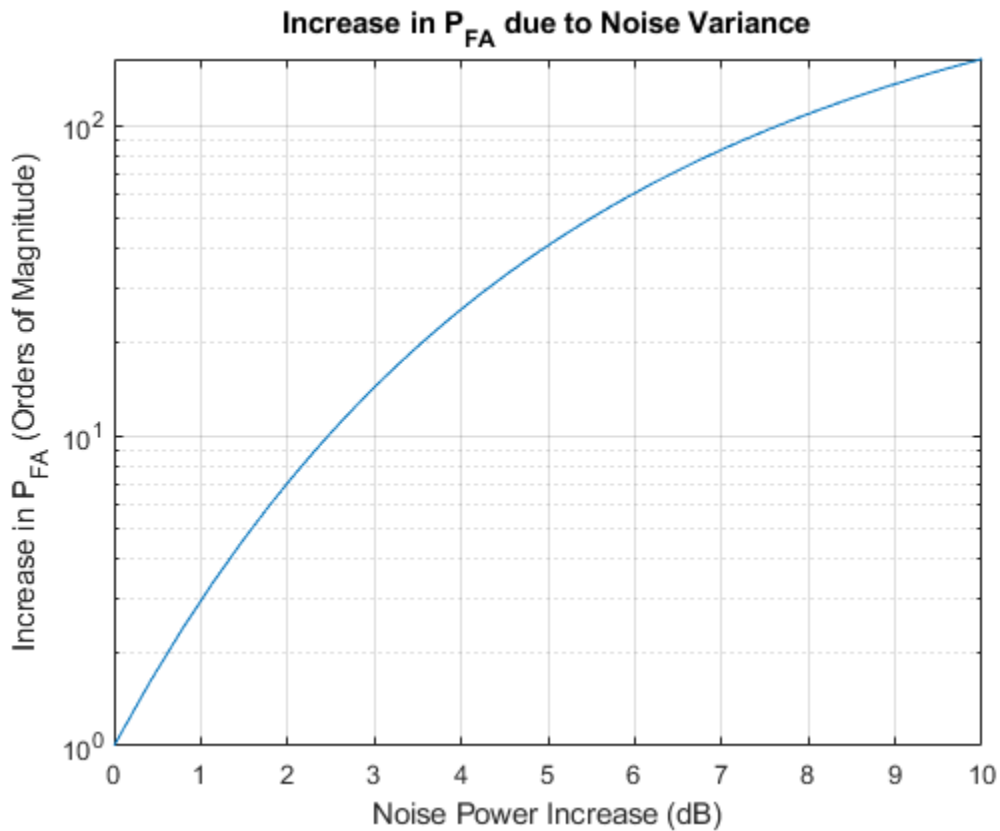
## Cell-Averaging CFAR Detector

The cell-averaging CFAR detector estimates the noise variance for the range cell of interest, or *cell under test*, by analyzing data from neighboring range cells designated as *training cells*. The noise characteristics in the training cells are assumed to be identical to the noise characteristics in the cell under test (CUT).

This assumption is key in justifying the use of the training cells to estimate the noise variance in the CUT. Additionally, the cell-averaging CFAR detector assumes that the training cells do not contain any signals from targets. Thus, the data in the training cells are assumed to consist of noise only.

To make these assumptions realistic:

* It is preferable to have some buffer, or guard cells, between the CUT and the training cells. The buffer provided by the guard cells *guards* against signal leaking into the training cells and adversely affecting the estimation of the noise variance.

* The training cells should not represent range cells too distant in range from the CUT, as the following figure illustrates.
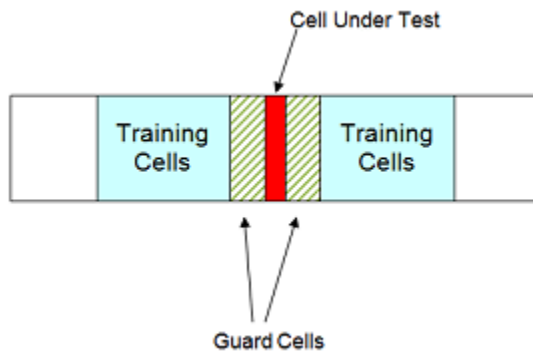
The optimum estimator for the noise variance depends on distributional assumptions and the type of detector. Assume the following:

**1** You are using a square-law detector.

**2** You have a Gaussian, complex-valued, random variable (RV) with independent real and imaginary parts.

**3** The real and imaginary parts each have mean zero and variance equal to $\sigma^2/2$.

---

**Note** If you denote this RV by Z=U+jV, the squared magnitude $|Z|^2$ follows an exponential distribution with mean $\sigma^2$.

---

If the samples in training cells are the squared magnitudes of such complex Gaussian RVs, you can use the sample mean as an estimator of the noise variance.

To implement cell-averaging CFAR detection, use `phased.CFARDetector`. You can customize characteristics of the detector such as the numbers of training cells and guard cells, and the probability of false alarm.

## CFAR Detector Adaptation to Noisy Input Data

This example shows how to create a CFAR detector and test its ability to adapt to the statistics of input data. The test uses noise-only trials. By using the default square-law detector, you can determine how close the empirical false-alarm rate is to the desired false-alarm probability.

Create a CFAR detector object with two guard cells, 20 training cells, and a false-alarm probability of 0.001. By default, this object assumes a square-law detector with no pulse integration.

```
detector = phased.CFARDetector('NumGuardCells',2,...
    'NumTrainingCells',20,'ProbabilityFalseAlarm',1e-3);
```

There are 10 training cells and 1 guard cell on each side of the *cell under test* (CUT). Set the CUT index to 12.

```
CUTidx = 12;
```

Seed the random number generator for a reproducible set of input data.

```
rng(1000);
```

Set the noise variance to 0.25. This value corresponds to an approximate -6 dB SNR. Generate a 23-by-10000 matrix of complex-valued, white Gaussian rv's with the specified variance. Each row of the matrix represents 10,000 Monte Carlo trials for a single cell.

```
Ntrials = 1e4;
variance = 0.25;
Ncells = 23;
inputdata = sqrt(variance/2)*(randn(Ncells,Ntrials)+1j*randn(Ncells,Ntrials));
```

Because the example implements a square-law detector, take the squared magnitudes of the elements in the data matrix.

```
Z = abs(inputdata).^2;
```

Provide the output from the square-law operator and the index of the cell under test to the CFAR detector.

```
Z_detect = detector(Z,CUTidx);
```

The output, `Z_detect`, is a logical vector with 10,000 elements. Sum the elements in `Z_detect` and divide by the total number of trials to obtain the empirical false-alarm rate.

```
pfa = sum(Z_detect)/Ntrials
```

```
pfa = 0.0013
```

The empirical false-alarm rate is 0.0013, which corresponds closely to the desired false-alarm rate of 0.001.

## Extensions of Cell-Averaging CFAR Detector

The cell-averaging algorithm for a CFAR detector works well in many situations, but not all. For example, when targets are closely located, cell averaging can cause a strong target to mask a weak target nearby. The `phased.CFARDetector` System object supports the following CFAR detection algorithms.

| Algorithm | Typical Usage |
|---|---|
| Cell-averaging CFAR | Most situations |
| Greatest-of cell-averaging CFAR | When it is important to avoid false alarms at the edge of clutter |
| Smallest-of cell-averaging CFAR | When targets are closely located |
| Order statistic CFAR | Compromise between greatest-of and smallest-of cell averaging |

## Detection Probability for CFAR Detector

This example shows how to compare the probability of detection resulting from two CFAR algorithms. In this scenario, the order statistic algorithm detects a target that the cell-averaging algorithm does not.

Create a CFAR detector that uses the cell-averaging CFAR algorithm.

```
Ntraining = 10;
Nguard = 2;
Pfa_goal = 0.01;
detector = phased.CFARDetector('Method','CA',...
    'NumTrainingCells',Ntraining,'NumGuardCells',Nguard,...
    'ProbabilityFalseAlarm',Pfa_goal);
```

The detector has 2 guard cells, 10 training cells, and a false-alarm probability of 0.01. This object assumes a square-law detector with no pulse integration.

Generate a vector of input data based on a complex-valued white Gaussian random variable.

```
Ncells = 23;
Ntrials = 100000;
inputdata = 1/sqrt(2)*(randn(Ncells,Ntrials) + ...
    1i*randn(Ncells,Ntrials));
```

In the input data, replace rows 8 and 12 to simulate two targets for the CFAR detector to detect.

```
inputdata(8,:) = 3*exp(1i*2*pi*rand);
inputdata(12,:) = 9*exp(1i*2*pi*rand);
```

Because the example implements a square-law detector, take the squared magnitudes of the elements in the input data vector.

```
Z = abs(inputdata).^2;
```

Perform the detection on rows 8 through 12.

```
Z_detect = detector(Z,8:12);
```

The `Z_detect` matrix has five rows. The first and last rows correspond to the simulated targets. The three middle rows correspond to noise.

Compute the probability of detection of the two targets. Also, estimate the probability of false alarm using the noise-only rows.

```
Pd_1 = sum(Z_detect(1,:))/Ntrials
```

```
Pd_1 = 0
```

```
Pd_2 = sum(Z_detect(end,:))/Ntrials
```

```
Pd_2 = 1
```

```
Pfa = max(sum(Z_detect(2:end-1,:),2)/Ntrials)
```

```
Pfa = 6.0000e-05
```

The `0` value of `Pd_1` indicates that this detector does not detect the first target.

Change the CFAR detector so it uses the order statistic CFAR algorithm with a rank of 5.

```
release(detector);
detector.Method = 'OS';
detector.Rank = 5;
```

Repeat the detection and probability computations.

```
Z_detect = detector(Z,8:12);
Pd_1 = sum(Z_detect(1,:))/Ntrials
```

Pd_1 = 0.5820

```
Pd_2 = sum(Z_detect(end,:))/Ntrials
```

Pd_2 = 1

```
Pfa = max(sum(Z_detect(2:end-1,:),2)/Ntrials)
```

Pfa = 0.0066

Using the order statistic algorithm instead of the cell-averaging algorithm, the detector detects the first target in about 58% of the trials.

# Measure Intensity Levels Using the Intensity Scope

This tutorial shows you how to measure the intensity of signals using the UI of the intensity scope. First, create an intensity scope. You can start with the example below, "RTI and DTI Displays in Full Radar Simulation" on page 8-41, or you can create your own scope. When this example launches, range-time-intensity (RTI) and Doppler-time-intensity (DTI) display windows open. This tutorial focuses on the RTI display so you can close the DTI window once the processing loop completes. This figure shows the RTI display after processing has completed. The display shows three tracks.
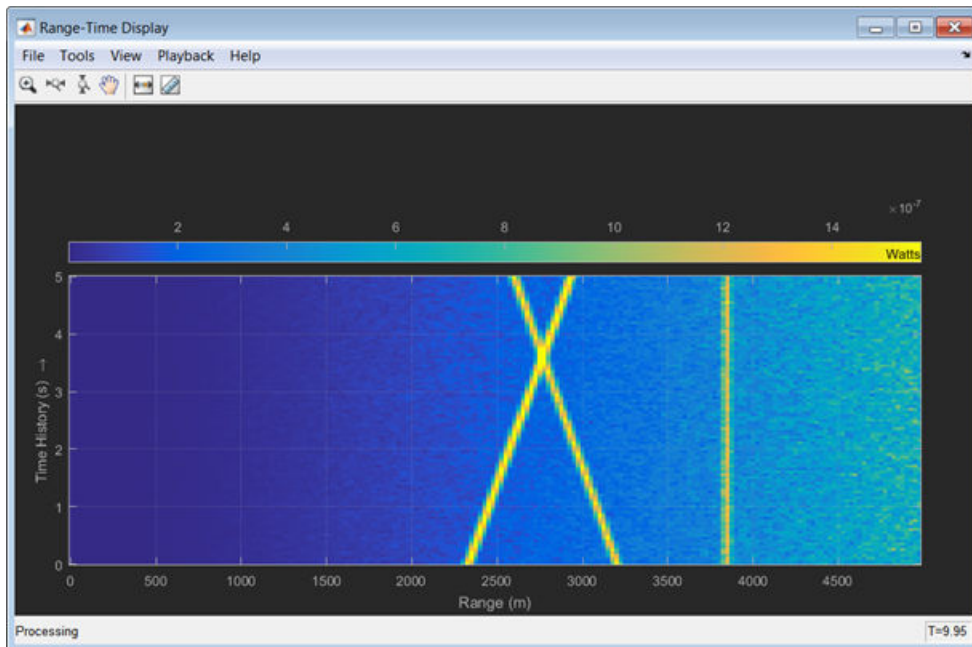


To examine the data, click the Cursor Measurement button  in the Toolbar. You see two cursors, each of which is represented by pairs of cross-hairs. To distinguish cursors, one pair consists of solid lines and the second pair consists of dashed lines and are tagged with a **1** or a **2**.
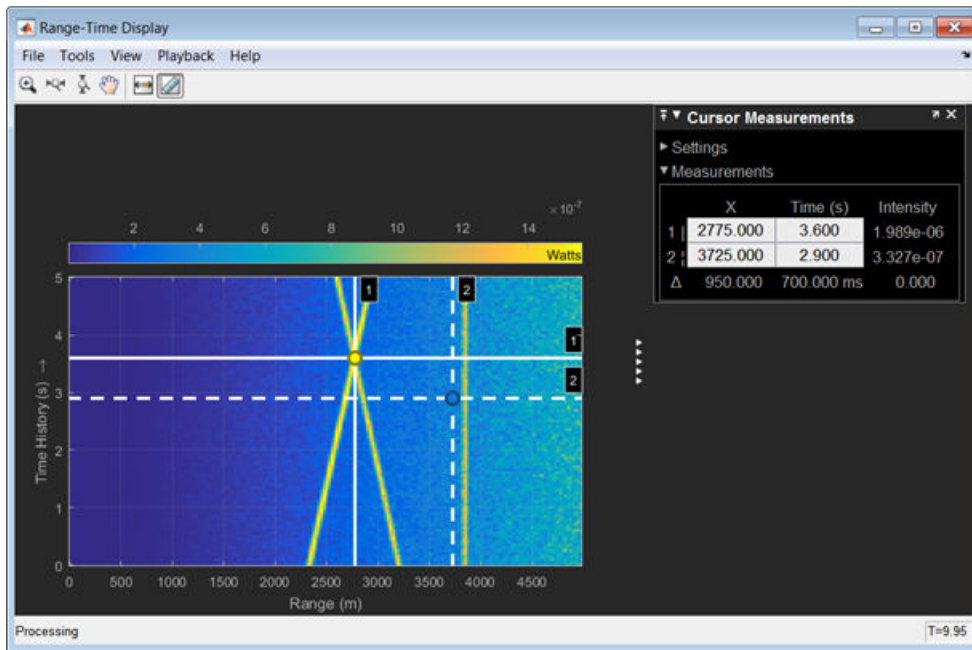
Cursor 1 has solid cross-hairs and overlays the intersection of two signal lines. Cursor 2 has dashed cross-hairs and overlays a signal-free region. The **Cursor Measurements** pane shows the coordinates of the cursors in time and range (labelled **X**) and the intensities at these positions. *Cursor 1* is located at a range of 2775 meters and a time of 3.6 seconds. The signal intensity at this point is 1.989e-6 watts. *Cursor 2* is located at a range of 3725 meters and a time of 2.9 seconds. The signal intensity at this point is 3.327e-7 watts. You can move the cursors to any positions of interest and obtain the intensity values.

## RTI and DTI Displays in Full Radar Simulation

Use the `phased.IntensityScope` System Object™ to display the detection output of a radar system simulation. The radar scenario contains a stationary single-element monostatic radar and three moving targets.

### Set Radar Operating Parameters

Set the maximum range, peak power range resolution, operating frequency, transmitter gain, and target radar cross-section.

```
max_range = 5000;
range_res = 50;
fc = 10e9;
tx_gain = 20;
peak_power = 5500.0;
```

Choose the signal propagation speed to be the speed of light, and compute the signal wavelength corresponding to the operating frequency.

```
c = physconst('LightSpeed');
lambda = c/fc;
```

Compute the pulse bandwidth from the range resolution. Set the sampling rate, `fs`, to twice the pulse bandwidth. The noise bandwidth is also set to the pulse bandwidth. The radar integrates a number of pulses set by `num_pulse_int`. The duration of each pulse is the inverse of the pulse bandwidth.

```
pulse_bw = c/(2*range_res);
pulse_length = 1/pulse_bw;
fs = 2*pulse_bw;
noise_bw = pulse_bw;
num_pulse_int = 10;
```

Set the pulse repetition frequency to match the maximum range of the radar.

```
prf = c/(2*max_range);
```

**Create System Objects for the Model**

Choose a rectangular waveform.

```
waveform = phased.RectangularWaveform('PulseWidth',pulse_length,...
    'PRF',prf,'SampleRate',fs);
```

Set the receiver amplifier characteristics.

```
amplifier = phased.ReceiverPreamp('Gain',20,'NoiseFigure',0,...
    'SampleRate',fs,'EnableInputPort',true,'SeedSource','Property',...
    'Seed',2007);
transmitter = phased.Transmitter('Gain',tx_gain,'PeakPower',peak_power,...
    'InUseOutputPort',true);
```

Specify the radar antenna as a single isotropic antenna.

```
antenna = phased.IsotropicAntennaElement('FrequencyRange',[5e9 15e9]);
```

Set up a monostatic radar platform.

```
radarplatform = phased.Platform('InitialPosition',[0; 0; 0],...
    'Velocity',[0; 0; 0]);
```

Set up the three target platforms using a single System object.

```
targetplatforms = phased.Platform(...
    'InitialPosition',[2000.66 3532.63 3845.04; 0 0 0; 0 0 0], ...
    'Velocity',[150 -150 0; 0 0 0; 0 0 0]);
```

Create the radiator and collector System objects.

```
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc);
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',fc);
```

Set up the three target RCS properties.

```
targets = phased.RadarTarget('MeanRCS',[1.6 2.2 1.05],'OperatingFrequency',fc);
```

Create System object to model two-way freespace propagation.

```
channels= phased.FreeSpace('SampleRate',fs,'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

Define a matched filter.

```
MFcoef = getMatchedFilter(waveform);
filter = phased.MatchedFilter('Coefficients',MFcoef,'GainOutputPort',true);
```

**Create Range and Doppler Bins**

Set up the fast-time grid. Fast time is the sampling time of the echoed pulse relative to the pulse transmission time. The range bins are the ranges corresponding to each bin of the fast time grid.

```
fast_time = unigrid(0,1/fs,1/prf,'[)');
range_bins = c*fast_time/2;
```

To compensate for range loss, create a time varying gain System Object.

```
gain = phased.TimeVaryingGain('RangeLoss',2*fspl(range_bins,lambda),...
    'ReferenceLoss',2*fspl(max_range,lambda));
```

Set up Doppler bins. Doppler bins are determined by the pulse repetition frequency. Create an FFT System object for Doppler processing.

```
DopplerFFTbins = 32;
DopplerRes = prf/DopplerFFTbins;
fft = dsp.FFT('FFTLengthSource','Property',...
    'FFTLength',DopplerFFTbins);
```

**Create Data Cube**

Set up a reduced data cube. Normally, a data cube has fast-time and slow-time dimensions and the number of sensors. Because the data cube has only one sensor, it is two-dimensional.

```
rx_pulses = zeros(numel(fast_time),num_pulse_int);
```

**Create IntensityScope System Objects**

Create two `IntensityScope` System objects, one for Doppler-time-intensity and the other for range-time-intensity.

```
dtiscope = phased.IntensityScope('Name','Doppler-Time Display',...
    'XLabel','Velocity (m/sec)', ...
    'XResolution',dop2speed(DopplerRes,c/fc)/2, ...
    'XOffset',dop2speed(-prf/2,c/fc)/2,...
    'TimeResolution',0.05,'TimeSpan',5,'IntensityUnits','Mag');
rtiscope = phased.IntensityScope('Name','Range-Time Display',...
    'XLabel','Range (m)', ...
    'XResolution',c/(2*fs), ...
    'TimeResolution',0.05,'TimeSpan',5,'IntensityUnits','Mag');
```

**Run the Simulation Loop over Multiple Radar Transmissions**

Transmit 2000 pulses. Coherently process groups of 10 pulses at a time.

For each pulse:

1  Update the radar position and velocity `radarplatform`
2  Update the target positions and velocities `targetplatforms`
3  Create the pulses of a single wave train to be transmitted `transmitter`
4  Compute the ranges and angles of the targets with respect to the radar

**5** Radiate the signals to the targets `radiator`

**6** Propagate the pulses to the target and back `channels`

**7** Reflect the signals off the target `targets`

**8** Receive the signal `sCollector`

**9** Amplify the received signal `amplifier`

**10** Form data cube

For each set of 10 pulses in the data cube:

**1** Match filter each row (fast-time dimension) of the data cube.

**2** Compute the Doppler shifts for each row (slow-time dimension) of the data cube.

```
pri = 1/prf;
nsteps = 200;
for k = 1:nsteps
    for m = 1:num_pulse_int
        [ant_pos,ant_vel] = radarplatform(pri);
        [tgt_pos,tgt_vel] = targetplatforms(pri);
        sig = waveform();
        [s,tx_status] = transmitter(sig);
        [~,tgt_ang] = rangeangle(tgt_pos,ant_pos);
        tsig = radiator(s,tgt_ang);
        tsig = channels(tsig,ant_pos,tgt_pos,ant_vel,tgt_vel);
        rsig = targets(tsig);
        rsig = collector(rsig,tgt_ang);
        rx_pulses(:,m) = amplifier(rsig,~(tx_status>0));
    end

    rx_pulses = filter(rx_pulses);
    MFdelay = size(MFcoef,1) - 1;
    rx_pulses = buffer(rx_pulses((MFdelay + 1):end), size(rx_pulses,1));
    rx_pulses = gain(rx_pulses);
    range = pulsint(rx_pulses,'noncoherent');
    rtiscope(range);
    dshift = fft(rx_pulses.');
    dshift = fftshift(abs(dshift),1);
    dtiscope(mean(dshift,2));
    radarplatform(.05);
    targetplatforms(.05);
end
```
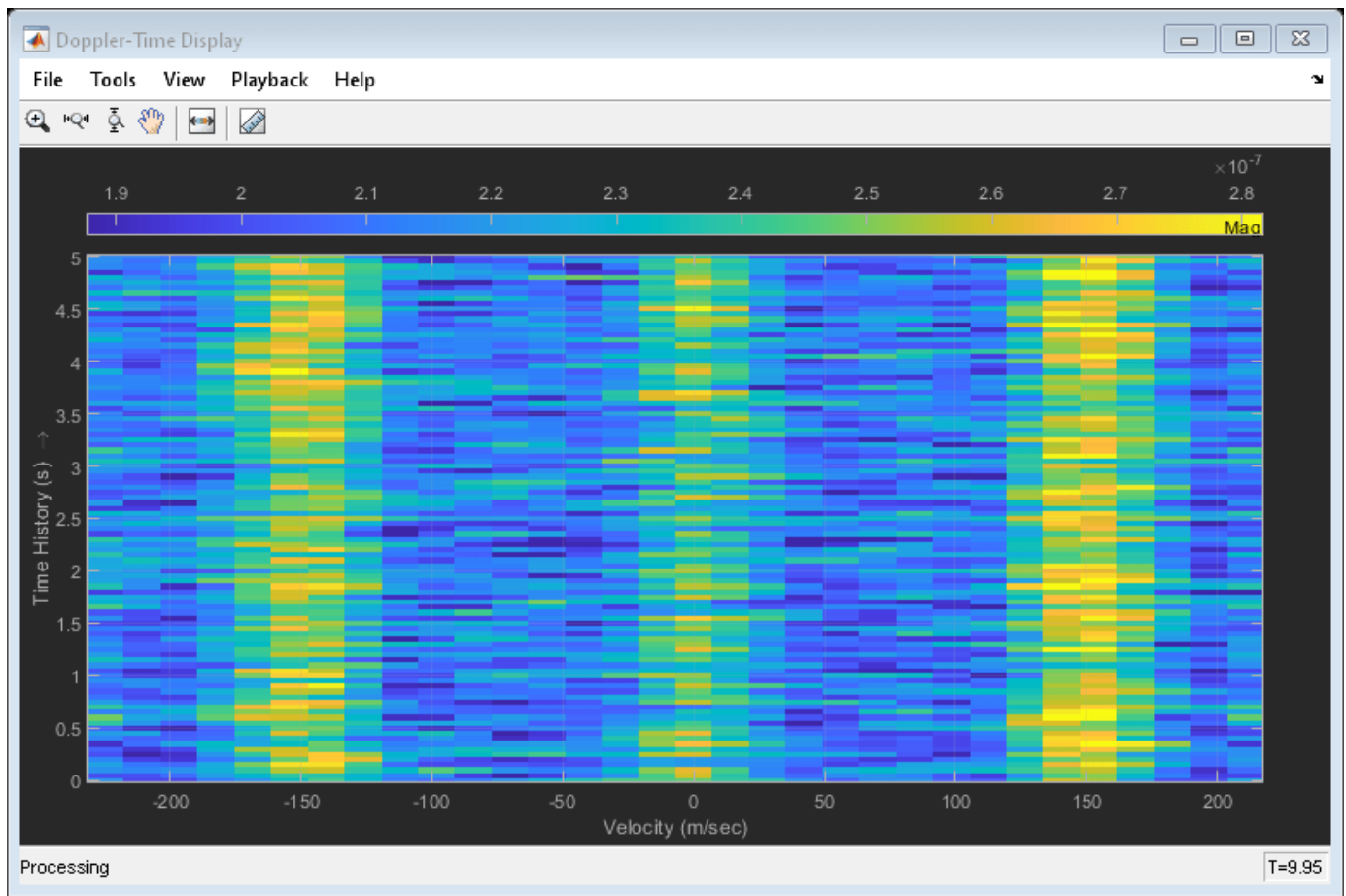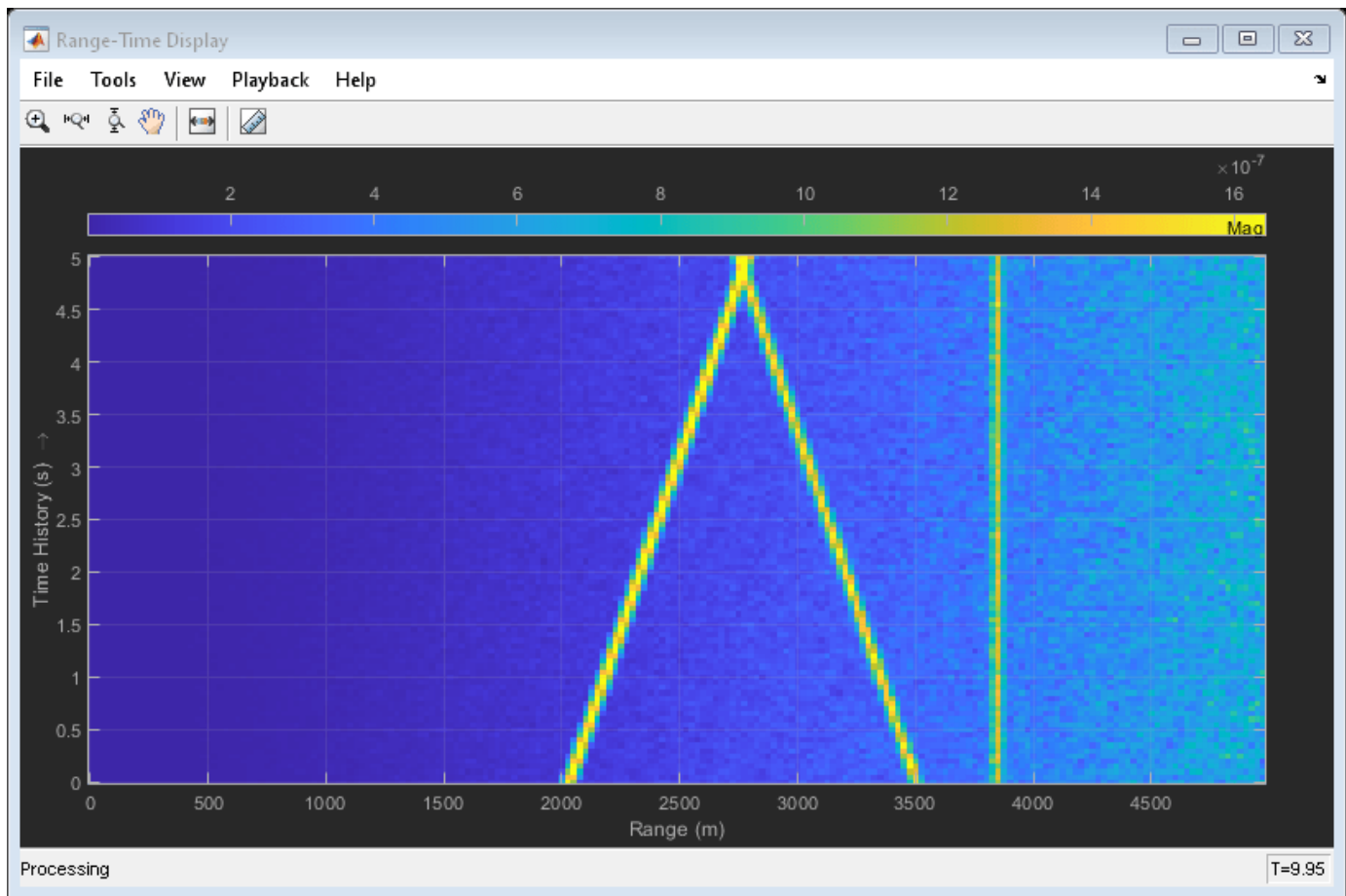
All of the targets lie on the x-axis. Two targets are moving along the x-axis and one is stationary. Because the radar is at the origin, you can read the target speed directly from the Doppler-Time Display window. The values agree with the specified velocities of -150, 150, and 0 m/sec.

# Environment and Target Models

# Free Space Path Loss

| **In this section...** |
| --- |
| |
| |
| |
| |
| |

## Support for Modeling Propagation in Free Space

Propagation environments have significant effects on the amplitude, phase, and shape of propagating space-time wavefields. In some cases, you may want to simulate a system that propagates narrowband signals through free space. If so, you can use the `phased.FreeSpace` System object to model the range-dependent time delay, phase shift, Doppler shift, and gain effects.

Consider this object as a point-to-point propagation channel. By setting object properties, you can customize certain characteristics of the environment and the signals propagating through it, including:

- Propagation speed and sampling rate of the signal you are propagating
- Signal carrier frequency
- Whether the object models one-way or two-way propagation

Each time you call a `phased.FreeSpace` object, you specify not only the signal to propagate, but also the location and velocity of the signal origin and destination.

You can use `fspl` to determine the free space path loss, in decibels, for a given distance and wavelength.

## Free Space Path Loss in dB

Assume a transmitter is located at *(1000,250,10)* in the global coordinate system. Assume a target is located at *(3000,750,20)*. The transmitter operates at 1 GHz. Determine the free space path loss in decibels for a narrowband signal propagating to and from the target.

```
[tgtrng,~] = rangeangle([3000; 750; 20],[1000; 250; 10]);
```

Determine the wavelength for 1 GHz.

```
lambda = physconst('LightSpeed')/1e9;
```

Solve for the two-way loss in dB using `fspl`. The two-way loss is twice the one-way loss.

```
L = 2*fspl(tgtrng,lambda)
```

```
L = 197.4635
```

The free space path loss in decibels is approximately 197.5 dB.

Alternatively, you can compute the loss directly from

```
Loss = 2*pow2db((4*pi*tgtrng/lambda)^2)

Loss = 197.4635
```

## Propagate Linear FM Pulse Waveform to Target and Back

Construct a linear FM pulse waveform of 50 ms duration with a bandwidth of 100 kHz. Model the range-dependent time delay and amplitude loss incurred during two-way propagation. The pulse propagates between the transmitter located at *(1000,250,10)* and a target located at *(3000,750,20)*. The signals propagate at the speed of light.

```
waveform = phased.LinearFMWaveform('SweepBandwidth',1e5,...
    'PulseWidth',5e-5,'OutputFormat','Pulses',...
    'NumPulses',1,'SampleRate',1e6,'PRF',1e4);
signal = waveform();
channel = phased.FreeSpace('SampleRate',1e6,...
    'TwoWayPropagation',true,'OperatingFrequency',1e9);
y = channel(signal,[1000; 250; 10],[3000; 750; 20],[0;0;0],[0;0;0]);
```

Plot the magnitude of the transmitted and received pulse to show the amplitude loss and time delay.

```
t = unigrid(0,1/waveform.SampleRate,1/waveform.PRF,'[)');
subplot(2,1,1)
plot(t.*1e6,abs(signal))
title('Magnitude of Transmitted Pulse')
xlabel('Time (microseconds)')
ylabel('Magnitude')
subplot(2,1,2)
plot(t.*1e6,abs(y))
title('Magnitude of Received Pulse')
xlabel('Time (microseconds)')
ylabel('Magnitude')
```
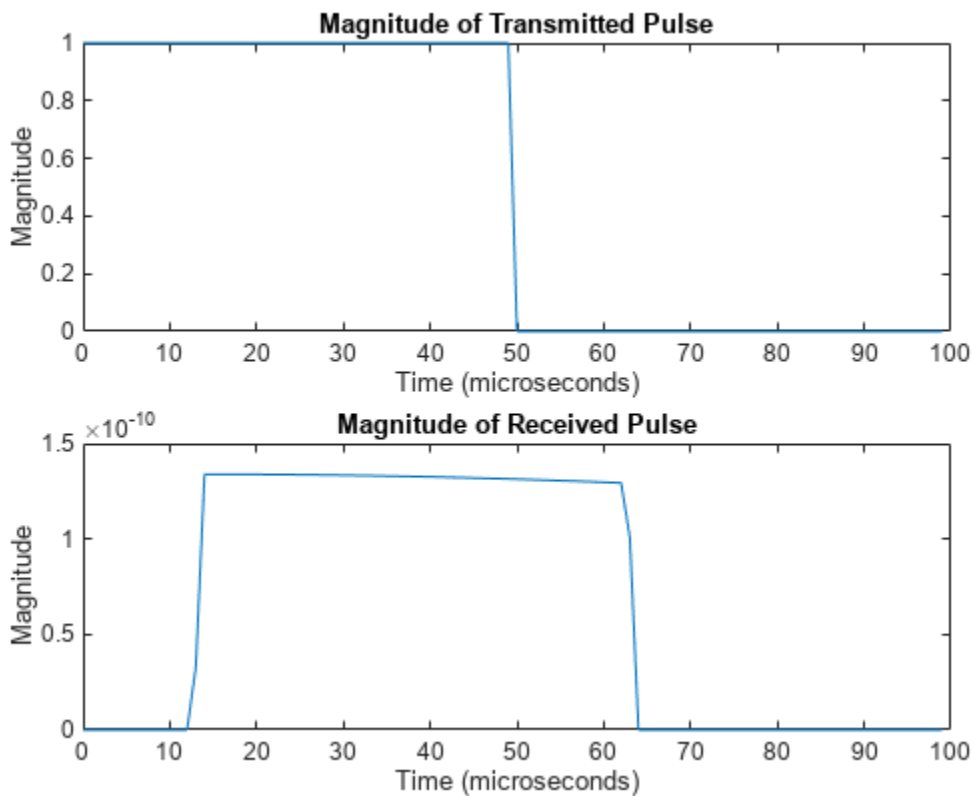
The delay in the received pulse is approximately 14 µs, the expected value for a distance of 4.123 km.

## One-Way and Two-Way Propagation

The `TwoWayPropagation` property of the `phased.FreeSpace` System object™ lets you simulate either one- or two-way propagation. The following example demonstrates how to use this property for a single linear FM pulse propagating to a target and back. The sensor is a single isotropic radiating antenna operating at 1 GHz located at *(1000,250,10)*. The target is located at *(3000,750,20)* and has a nonfluctuating RCS of 1 square meter.

The following code constructs the required System objects and calculates the range and angle from the antenna to the target.

```
waveform = phased.LinearFMWaveform('SweepBandwidth',1e5,...
    'PulseWidth',5e-5,'OutputFormat','Pulses',...
    'NumPulses',1,'SampleRate',1e6);
antenna = phased.IsotropicAntennaElement('FrequencyRange',[500e6 1.5e9]);
transmitter = phased.Transmitter('PeakPower',1e3,'Gain',20);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',1e9);
channel = phased.FreeSpace('SampleRate',1e6,...
    'TwoWayPropagation',true,'OperatingFrequency',1e9);
target = phased.RadarTarget('MeanRCS',1,'Model','Nonfluctuating');
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',1e9);
sensorpos = [3000;750;20];
```

```
tgtpos = [1000;250;10];
[tgtrng,tgtang] = rangeangle(sensorpos,tgtpos);
```

Because the `TwoWayPropagation` property is set to `true`, you compute the total propagation only once.

Compute the radiated signal.

```
pulse = waveform();
pulse = transmitter(pulse);
pulse = radiator(pulse,tgtang);
```

Propagate the pulse to the target and back.

```
pulse = channel(pulse,sensorpos,tgtpos,[0;0;0],[0;0;0]);
pulse = target(pulse);
sig = collector(pulse,tgtang);
```

Alternatively, you can break up the two-way propagation into two separate one-way propagation paths. You do so by setting the `TwoWayPropagation` property to `false`.

```
channel1 = phased.FreeSpace('SampleRate',1e9,...
    'TwoWayPropagation',false,'OperatingFrequency',1e6);
```

Radiate the pulse.

```
pulse = waveform();
pulse = transmitter(pulse);
pulse = radiator(pulse,tgtang);
```

Propagate the pulse from the antenna to the target.

```
pulse = channel1(pulse,sensorpos,tgtpos,[0;0;0],[0;0;0]);
pulse = target(pulse);
```

Propagate the reflected pulse from the target to the antenna.

```
pulse = channel(pulse,tgtpos,sensorpos,[0;0;0],[0;0;0]);
sig = collector(pulse,tgtang);
```

## Propagate Signal from Stationary Radar to Moving Target

This example shows how to propagate a signal in free space from a stationary radar to a moving target.

Define the signal sample rate, propagation speed, and carrier frequency. Define the signal as a sinusoid of frequency 150 Hz. Set the sample rate to 1 kHz and the carrier frequency to 300 MHz. The propagation speed is the speed of light.

```
fs = 1.0e3;
c = physconst('Lightspeed');
fc = 300e3;
f = 150.0;
N = 1024;
t = (0:N-1)'/fs;
x = exp(1i*2*pi*f*t);
```

Assume the target is approaching the radar at 300.0 m/s, and the radar is stationary. Find the Doppler shift that corresponds to this relative speed.

```
v = 1000.0;
dop = speed2dop(v,c/fc)
```

dop = 1.0007

From the formula, the one-way Doppler shift is 1 Hz.

Create a `phased.FreeSpace` System object™, and use it to propagate the signal from the radar to the target. Assume the radar is at (0, 0, 0) and the target is at (100, 0, 0).

```
channel = phased.FreeSpace('SampleRate',fs,...
    'PropagationSpeed',c,'OperatingFrequency',fc);
origin_pos = [0;0;0];
dest_pos = [100;0;0];
origin_vel = [0;0;0];
dest_vel = [-v;0;0];
y = channel(x,origin_pos,dest_pos,origin_vel,dest_vel);
```

Plot the spectrum of the transmitted signal. The peak at 150 Hz reflects the frequency of the signal.

```
window = 64;
ovlp = 32;
[Pxx,F] = pwelch(x,window,ovlp,N,fs);
plot(F,10*log10(Pxx))
xlabel('Frequency (Hz)')
ylabel('Power/Frequency (dB/Hz)')
title('Transmitted Signal')
```

**Transmitted Signal**
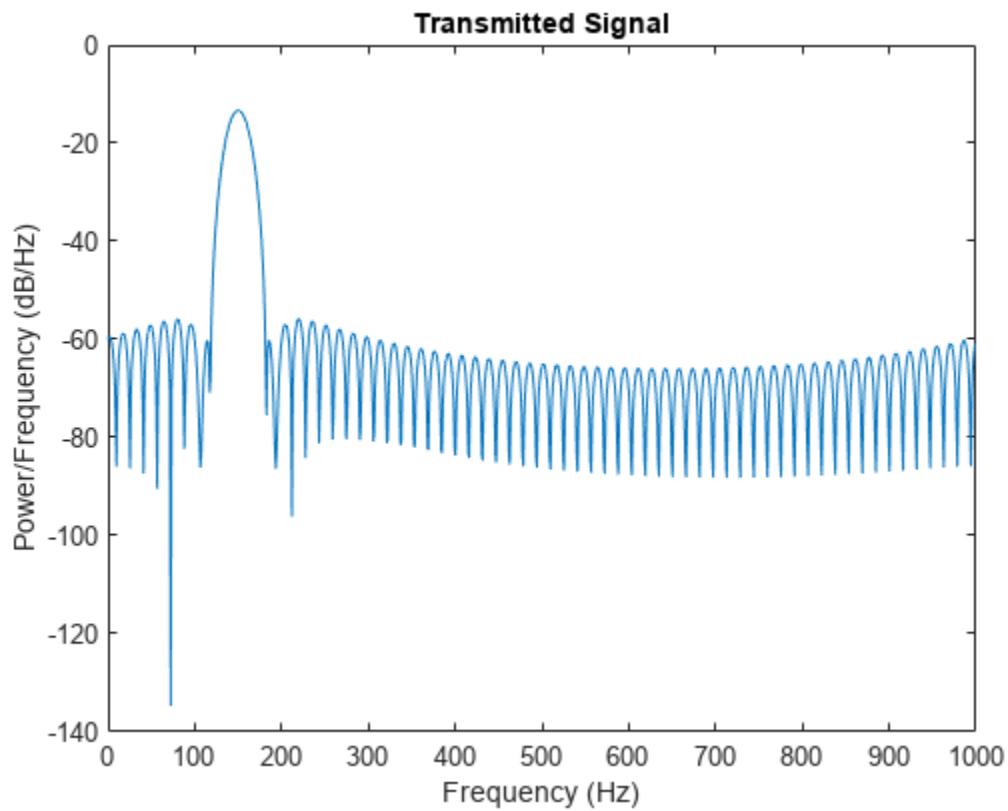


Plot the spectrum of the propagated signal. The peak at 250 Hz reflects the frequency of the signal plus the Doppler shift of 100 Hz.

```
window = 64;
ovlp = 32;
[Pyy,F] = pwelch(y,window,ovlp,N,fs);
plot(F,10*log10(Pyy))
grid
xlabel('Frequency (Hz)')
ylabel('Power/Frequency (dB/Hz)')
title('Propagated Signal')
```

**Propagated Signal**
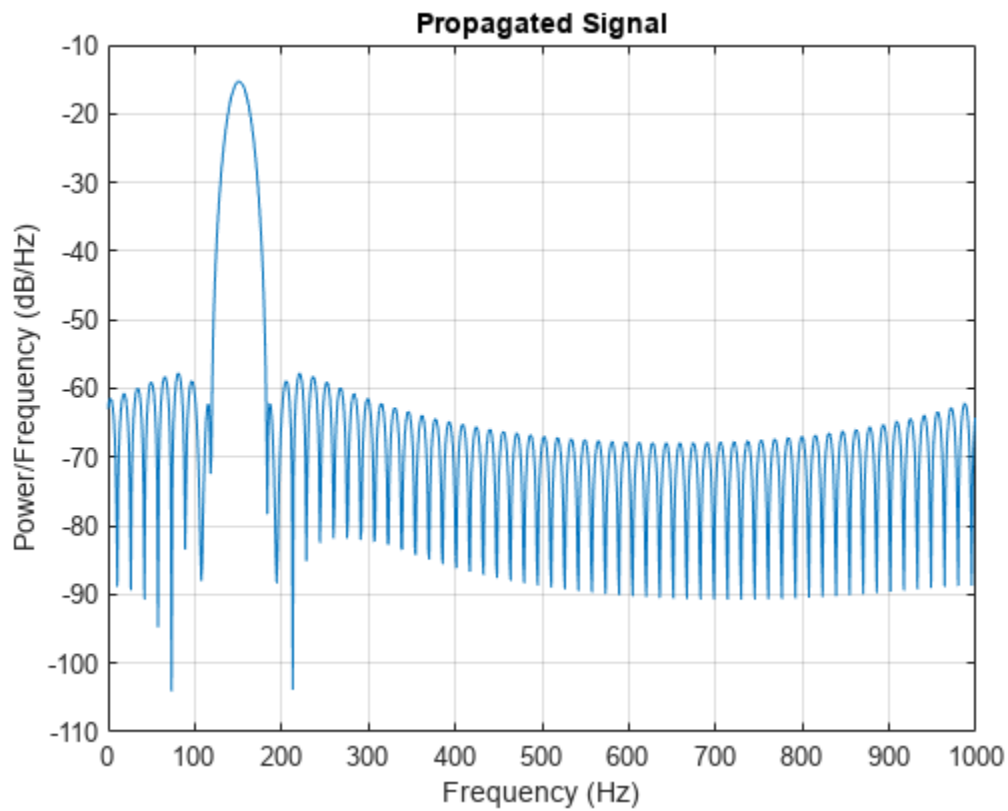
The Doppler shift is too small to see. Overlay the two spectra in the region of 150 Hz.

```
figure
idx = find(F>=130 & F<=170);
plot(F(idx),10*log10(Pxx(idx)),'b')
grid
hold on
plot(F(idx),10*log10(Pyy(idx)),'r')
hold off
xlabel('Frequency (Hz)')
ylabel('Power/Frequency (dB/Hz)')
title('Transmitted and Propagated Signals')
legend('Transmitted','Propagated')
```
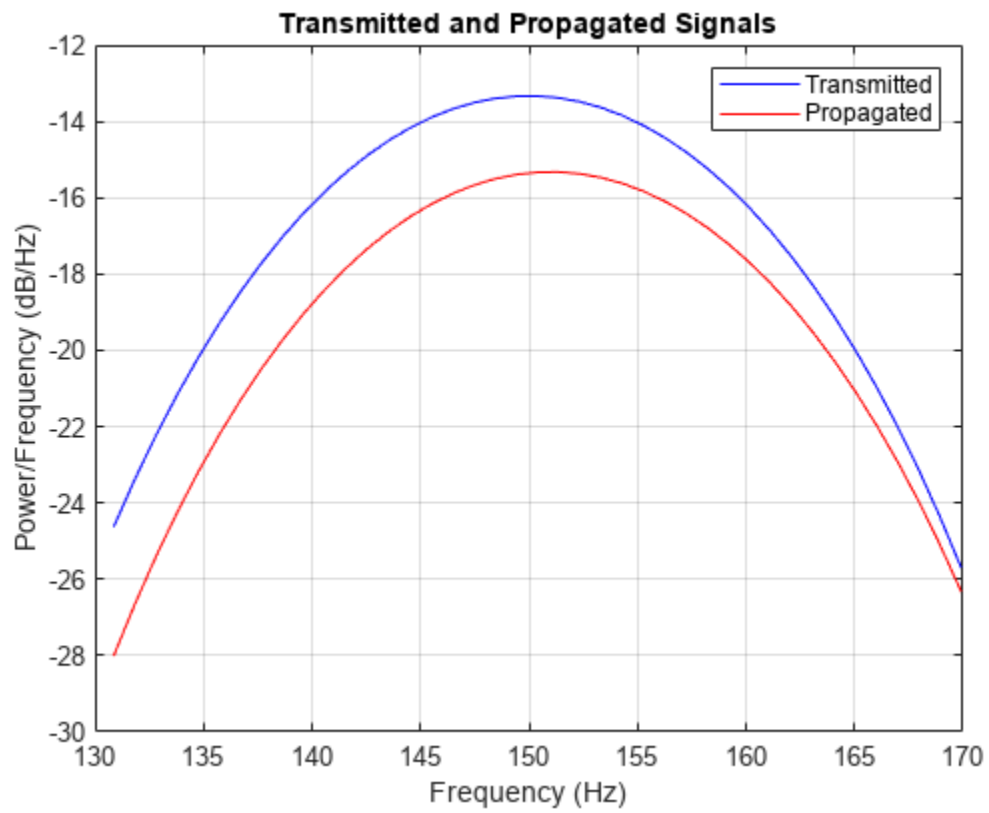
The peak shift appears to be approximately 1 Hz.

# Two-Ray Multipath Propagation

A two-ray propagation channel is the next step up in complexity from a free-space channel and is the simplest case of a multipath propagation environment. The free-space channel models a straight-line *line-of-sight* path from point 1 to point 2. In a two-ray channel, the medium is specified as a homogeneous, isotropic medium with a reflecting planar boundary. The boundary is always set at $z = 0$. There are at most two rays propagating from point 1 to point 2. The first ray path propagates along the same line-of-sight path as in the free-space channel (see the `phased.FreeSpace` System object). The line-of-sight path is often called the *direct path*. The second ray reflects off the boundary before propagating to point 2. According to the Law of Reflection , the angle of reflection equals the angle of incidence. In short-range simulations such as cellular communications systems and automotive radars, you can assume that the reflecting surface, the ground or ocean surface, is flat.

The figure illustrates two propagation paths. From the source position, $s_s$, and the receiver position, $s_r$, you can compute the arrival angles of both paths, $\theta'_{los}$ and $\theta'_{rp}$. The arrival angles are the elevation and azimuth angles of the arriving radiation with respect to a local coordinate system. In this case, the local coordinate system coincides with the global coordinate system. You can also compute the transmitting angles, $\theta_{los}$ and $\theta_{rp}$. In the global coordinates, the angle of reflection at the boundary is the same as the angles $\theta_{rp}$ and $\theta'_{rp}$. The reflection angle is important to know when you use angle-dependent reflection-loss data. You can determine the reflection angle by using the `rangeangle` function and setting the reference axes to the global coordinate system. The total path length for the line-of-sight path is shown in the figure by $R_{los}$ which is equal to the geometric distance between source and receiver. The total path length for the reflected path is $R_{rp} = R_1 + R_2$. The quantity $L$ is the ground range between source and receiver.
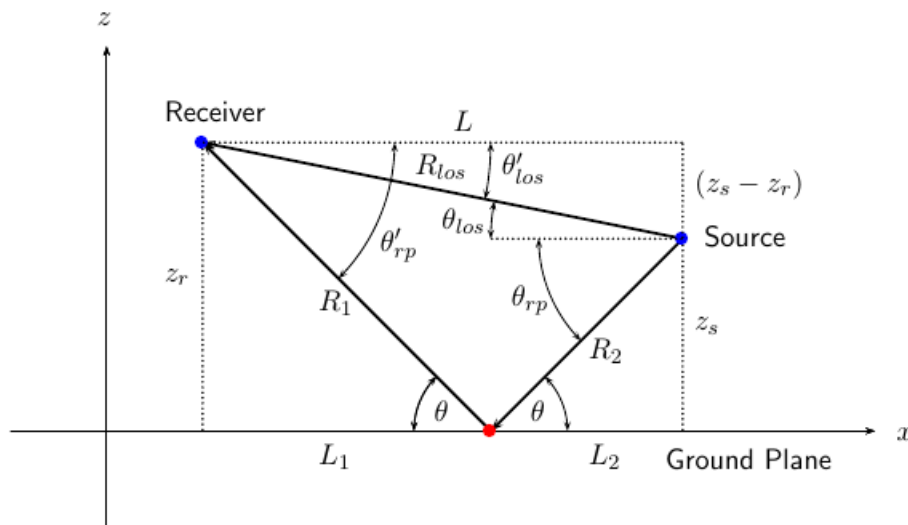


You can easily derive exact formulas for path lengths and angles in terms of the ground range and object heights in the global coordinate system.

$$\vec{R} = \vec{x}_s - \vec{x}_r$$

$$R_{los} = \left| \vec{R} \right| = \sqrt{(z_r - z_s)^2 + L^2}$$

$$R_1 = \frac{z_r}{z_r + z_z}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_2 = \frac{z_s}{z_s + z_r}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_{rp} = R_1 + R_2 = \sqrt{(z_r + z_s)^2 + L^2}$$

$$\tan\theta_{los} = \frac{(z_s - z_r)}{L}$$

$$\tan\theta_{rp} = -\frac{(z_s + z_r)}{L}$$

$$\theta'_{los} = -\theta_{los}$$

$$\theta'_{rp} = \theta_{rp}$$

# Free-Space Propagation of Wideband Signals

Propagate a wideband signal with three tones in an underwater acoustic with constant speed of propagation. You can model this environment as free space. The center frequency is 100 kHz and the frequencies of the three tones are 75 kHz, 100 kHz, and 125 kHz, respectively. Plot the spectrum of the original signal and the propagated signal to observe the Doppler effect. The sampling frequency is 100 kHz.

```
c = 1500;
fc = 100e3;
fs = 100e3;
relfreqs = [-25000,0,25000];
```

Set up a stationary radar and moving target and compute the expected Doppler.

```
rpos = [0;0;0];
rvel = [0;0;0];
tpos = [30/fs*c; 0;0];
tvel = [45;0;0];
dop = -tvel(1)./(c./(relfreqs + fc));
```

Create a signal and propagate the signal to the moving target.

```
t = (0:199)/fs;
x = sum(exp(1i*2*pi*t.'*relfreqs),2);
channel = phased.WidebandFreeSpace(...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'SampleRate',fs);
y = channel(x,rpos,tpos,rvel,tvel);
```

Plot the spectra of the original signal and the Doppler-shifted signal.

```
periodogram([x y],rectwin(size(x,1)),1024,fs,'centered')
ylim([-150 0])
legend('original','propagated');
```
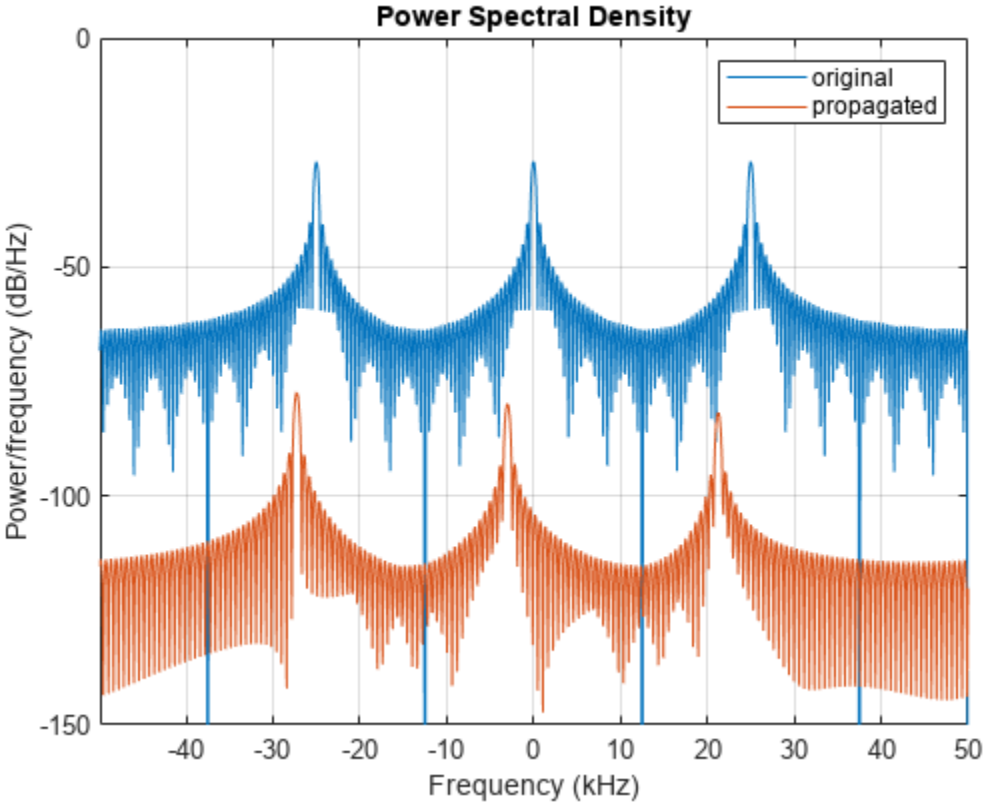
For this wideband signal, you can see that the magnitude of the Doppler shift increases with frequency. In contrast, for narrowband signals, the Doppler shift is assumed constant over the band.

# Radar Target

## Radar Target Properties

The `phased.RadarTarget` System object models a reflected signal from a target. The target may have a nonfluctuating or fluctuating radar cross section (RCS). This object has the following modifiable properties:

- `MeanRCSSource` — Source of the target's mean radar cross section
- `MeanRCS` — Target's mean RCS
- `Model` — Statistical model for the target's RCS
- `PropagationSpeed` — Signal propagation speed
- `OperatingFrequency` — Operating frequency
- `SeedSource` — Source of the seed for the random number generator to generate the target's random RCS values
- `Seed` — Seed for the random number generator

## Gain for Nonfluctuating RCS Target

Create a radar target with a nonfluctuating RCS of 1 square meter and an operating frequency of 1 GHz. Specify a wave propagation speed equal to the speed of light.

```
sigma = 1.0;
target = phased.RadarTarget('Model','nonfluctuating','MeanRCS',sigma,...
    'PropagationSpeed',physconst('LightSpeed'),'OperatingFrequency',1e9);
```

For a nonfluctuation target, the reflected waveform equals the incident waveform scaled by the gain

$$G = \sqrt{\frac{4\pi\sigma}{\lambda^2}}$$

Here, σ represents the mean target RCS, and λ is the wavelength of the operating frequency.

Set the signal incident on the target to be a vector of ones to obtain the gain factor used by the `phased.RadarTarget` System object™.

```
x = ones(10,1);
y = target(x)
```

y = *10×1*

```
   11.8245
   11.8245
   11.8245
   11.8245
   11.8245
   11.8245
   11.8245
   11.8245
   11.8245
```

```
11.8245
```

Compute the gain from the formula to verify that the output of the System object equals the theoretical value.

```
lambda = target.PropagationSpeed/target.OperatingFrequency;
G = sqrt(4*pi*sigma/lambda^2)
```

```
G = 11.8245
```

## Fluctuating RCS Targets

The previous examples used nonfluctuating values for the target's RCS. This model is not valid in many scenarios. There are several cases where the RCS exhibits relatively small or large magnitude fluctuations. These fluctuations can occur rapidly on pulse-to-pulse, or more slowly, on scan-to-scan time scales:

- **Several small randomly distributed reflectors with no dominant reflector** — This target, at close range or when the radar uses pulse-to-pulse frequency agility, can exhibit large magnitude rapid (pulse-to-pulse) fluctuations in the RCS. That same complex reflector at long range with no frequency agility can exhibit large magnitude fluctuations in the RCS over a longer time scale (scan-to-scan).
- **Dominant reflector along with several small reflectors** — The reflectors in this target can exhibit small magnitude fluctuations on pulse-to-pulse or scan-to-scan time scales, subject to:
  - How rapidly the aspect changes
  - Whether the radar uses frequency agility

To account for significant fluctuations in the RCS, you need to use statistical models. The four *Swerling* models, described in the following table, are widely used to cover these kinds of fluctuating-RCS cases.

| Swerling Case Number | Description |
|---|---|
| I | Scan-to-scan decorrelation. Rayleigh/exponential PDF — A number of randomly distributed scatterers with no dominant scatterer. |
| II | Pulse-to-pulse decorrelation. Rayleigh/exponential PDF — A number of randomly distributed scatterers with no dominant scatterer. |
| III | Scan-to-scan decorrelation — Chi-square PDF with 4 degrees of freedom. A number of scatterers with one scatterer dominant. |
| IV | Pulse-to-pulse decorrelation — Chi-square PDF with 4 degrees of freedom. A number of scatterers with one scatterer dominant. |

You can simulate a Swerling target model by setting the `Model` property. Call the object with the `UPDATERCS` input argument set to `true` or `false`. Setting `UPDATERCS` to `true` updates the RCS value according to the specified probability model each time you call `step`. If you set `UPDATERCS` to `false`, the previous RCS value is used.

## Model Pulse Reflection from Nonfluctuating Target

This example creates and transmits a linear FM waveform with a 1 GHz carrier frequency. The waveform is transmitted and collected by an isotropic antenna with a back-baffled response. The waveform propagates to and from a target with a nonfluctuating RCS of 1 square meter. The target is located at a range of 1.414 km from the antenna at an azimuth angle of 45° and an elevation of 0°.

Set up the radar system.

```
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
antennapos = phased.Platform('InitialPosition',[0;0;0]);
targetpos = phased.Platform('InitialPosition',[1000; 1000; 0]);
waveform = phased.LinearFMWaveform('PulseWidth',100e-6);
transmitter = phased.Transmitter('PeakPower',1e3,'Gain',40);
radiator = phased.Radiator('OperatingFrequency',1e9, ...
    'Sensor',antenna);
channel = phased.FreeSpace('OperatingFrequency',1e9,...
    'TwoWayPropagation',true);
target = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',1e9);
collector = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',antenna);
```

Compute the transmitted and received waveforms.

```
wav = waveform();
txwav = transmitter(wav);
radwav = radiator(txwav,[0 0]');
propwav = channel(radwav,antennapos.InitialPosition,...
    targetpos.InitialPosition,[0;0;0],[0;0;0]);
reflwav = target(propwav);
collwav = collector(reflwav,[45 0]');
```

# Swerling 1 Target Models

The example presents a scenario consisting of a rotating monostatic radar and a target with a radar cross-section described by a Swerling 1 model. In this example, the radar and target are stationary.

**Swerling 1 versus Swerling 2 Models**

For Swerling 1 and Swerling 2 target models, the total RCS arises from many independent small scatterers of approximately equal individual RCS. The total RCS may vary with every pulse in a scan (Swerling 2) or may be constant over a complete scan consisting of multiple pulses (Swerling 1). In either case, the statistics obey a chi-squared probability density function with two degrees of freedom.

**Dwell Time and Radar Scan**

For simplicity, start with a rotating radar having a rotation time of 5 seconds corresponding to a rotation rate or scan rate of 72 degrees/sec.

```
Trot = 5.0;
rotrate = 360/Trot;
```

The radar has a main half-power beam width (HPBW) of 3.0 degrees. During the time that a target is illuminated by the main beam, radar pulses strike the target and reflect back to the radar. The time period during which the target is illuminated is called the dwell time. This time period is also called a scan. The example will process 3 scans of the target.

```
HPBW = 3.0;
Tdwell = HPBW/rotrate;
Nscan = 3;
```

The number of pulses that arrive on target during the dwell time depends upon the pulse repetition frequency (PRF). PRF is the inverse of the pulse repetition interval (PRI). Assume 5000 pulses are transmitted per second.

```
prf = 5000.0;
pri = 1/prf;
```

The number of pulses in one dwell time is

```
Np = floor(Tdwell*prf);
```

**Set up a Swerling 1 radar model**

You create a Swerling 1 target by using the `RadarTarget` System object™. To effect a Swerling 1 model, set the `Model` property of the `phased.RadarTarget` System object to either `'Swerling1'` or `'Swerling2'`. Both are equivalent. Then, at the first call to the object at the beginning of the scan, set the `updatercs` argument to `true`. Set `updatercs` to `false` for the remaining calls to the object during the scan. This procedure means that the radar cross section is only updated at the beginning of a scan and remains constant for the remainder of the scan.

Set the target model to `'Swerling1'`.

```
tgtmodel = 'Swerling1';
```

**Set up radar model System object components**

Set up the radiating antenna. Assume the operating frequency of the antenna is 1 GHz.

```
fc = 1e9;
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
radiator = phased.Radiator('OperatingFrequency',fc, ...
    'Sensor',antenna);
```

Specify the location of the stationary antenna.

```
radarplatform = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
targetplatform = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the object.

```
waveform = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifier.

```
transmitter = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
channel = phased.FreeSpace('OperatingFrequency',fc,'TwoWayPropagation',true);
```

Specify the radar target to have a mean RCS of 1 m2 and be of the Swerling model type 1 or 2. You can use Swerling 1 or 2 interchangeably.

```
target = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
collector = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',antenna);
```

Define a matched filter to process the incoming signal.

```
wav = waveform();
filter = phased.MatchedFilter('Coefficients',getMatchedFilter(waveform));
```

**Processing loop for 3 scans of a Swerling 1 target**

**1**  Generate waveform with unit amplitude

**2**  Amplify the transmit waveform

**3**  Radiate the waveform in the desired direction to the target

**4**  Propagate the waveform to and from the target

**5**  Reflect waveform from radar target

**6**  Collect radiation to create received signal

**7**  Match filter received signal

Provide memory for radar return amplitudes.

```
z = zeros(Nscan,Np);
tp = zeros(Nscan,Np);
```

Enter the loop. Set `updatercs` to `true` only for the first pulse of the scan.

```
for m = 1:Nscan
    t0 = (m-1)*Trot;
    t = t0;
    for k = 1:Np

        if k == 1
            updatercs = true;
        else
            updatercs = false;
        end
        t = t + pri;
        txwav = transmitter(wav);
```

Find the radar and target positions

```
        [xradar,vradar] = radarplatform(t);
        [xtgt,vtgt] = targetplatform(t);
```

Radiate waveform to target

```
        [~,ang] = rangeangle(xtgt,xradar);
        radwav = radiator(txwav,ang);
```

Propagate waveform to and from the target

```
        propwav = channel(radwav,xradar,xtgt,vradar,vtgt);
```

Reflect waveform from target. Set the `updatercs` flag.

```
        reflwav = target(propwav,updatercs);
```

Collect the received waveform

```
        collwav = collector(reflwav,ang);
```

Apply matched filter to incoming signal

```
        y = filter(collwav);
        z(m,k) = max(abs(y));
        tp(m,k) = t;

    end
end
```

**Plot the pulse amplitudes**

Plot the amplitudes of the pulses for the scan as a function of time.

```
plot(tp(:),z(:),'.')
xlabel('Time (sec)')
ylabel('Pulse Amplitudes')
```

Notice that the pulse amplitudes are constant within a scan.

# Swerling Target Models

The example illustrates the use of Swerling target models to describe the fluctuations in radar cross-section. The scenario consists of a rotating monostatic radar and a target having a radar cross-section described by a Swerling 2 model. In this example, the radar and target are stationary.

**Swerling 1 versus Swerling 2 Models**

In Swerling 1 and Swerling 2 target models, the total RCS arises from many independent small scatterers of approximately equal individual RCS. The total RCS may vary with every pulse in a scan (Swerling 2) or may be constant over a complete scan consisting of multiple pulses (Swerling 1). In either case, the statistics obey a chi-squared probability density function with two degrees of freedom.

**Dwell Time and Radar Scan**

For simplicity, start with a rotating radar having a rotation time of 5 seconds corresponding to a rotation or scan rate of 72 degrees/sec.

```
Trot = 5.0;
scanrate = 360/Trot;
```

The radar has a main half-power beam width (HPBW) of 3.0 degrees. During the time that a target is illuminated by the main beam, radar pulses strike the target and reflect back to the radar. The time period during which the target is illuminated is called the dwell time. This time is also called a scan. The radar will process 3 scans of the target.

```
HPBW = 3.0;
Tdwell = HPBW/scanrate;
Nscan = 3;
```

The number of pulses that arrive on target during the dwell time depends upon the pulse repetition frequency (PRF). PRF is the inverse of the pulse repetition interval (PRI). Assume 5000 pulses are transmitted per second.

```
prf = 5000.0;
pri = 1/prf;
```

The number of pulses in one dwell time is

```
Np = floor(Tdwell*prf);
```

**Set up a Swerling 2 model**

You create a Swerling 2 target by using the `RadarTarget` System object™. To effect a Swerling 2 model, set the `Model` property of the `phased.RadarTarget` System object to either `'Swerling1'` or `'Swerling2'`. Both are equivalent. Then, at every call to the object, set the `updatercs` argument to `true`. This argument means that the radar cross-section is updated at every pulse.

Set the target model to `'Swerling1'` .

```
tgtmodel = 'Swerling2';
```

**Set up radar model System object components**

Set up the radiating antenna. Assume the operating frequency of the antenna is 1 GHz.

```
fc = 1e9;
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
radiator = phased.Radiator('OperatingFrequency',fc,'Sensor',antenna);
```

Specify the location of the stationary antenna.

```
radarplatform = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
targetplatform = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the object.

```
waveform = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifier.

```
transmitter = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
channel = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',true);
```

Specify the radar target to have a mean RCS of 1 m2 and be of the Swerling model type 1 or 2. You can use Swerling 1 or 2 interchangeably.

```
target = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
collector = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',antenna);
```

Define a matched filter to process the incoming signal.

```
wav = waveform();
filter = phased.MatchedFilter(...
    'Coefficients',getMatchedFilter(waveform));
```

**Processing loop for 3 scans of a Swerling 2 target**

1  Generate waveform with unit amplitude
2  Amplify the transmit waveform
3  Radiate the waveform in the desired direction to the target
4  Propagate the waveform to and from the radar target
5  Reflect waveform from radar target.
6  Collect radiation to create received signal
7  Match filter received signal

Provide memory for radar return amplitudes.

```
z = zeros(Nscan,Np);
tp = zeros(Nscan,Np);
```

Enter the loop. Set `updatercs` to `true` only for the first pulse of the scan.

```
for m = 1:Nscan
    t0 = (m-1)*Trot;
    t = t0;
    updatercs = true;
    for k = 1:Np

        t = t + pri;
        txwav = transmitter(wav);
```

Find the radar and target positions

```
        [xradar,vradar] = radarplatform(t);
        [xtgt,vtgt] = targetplatform(t);
```

Radiate waveform to target

```
        [~,ang] = rangeangle(xtgt,xradar);
        radwav = radiator(txwav,ang);
```

Propagate waveform to and from the target

```
        propwav = channel(radwav,radarplatform.InitialPosition,...
            targetplatform.InitialPosition,[0;0;0],[0;0;0]);
```

Reflect waveform from target. Set the `updatercs` flag.

```
        reflwav = target(propwav,updatercs);
```

Collect the received waveform

```
        collwav = collector(reflwav,ang);
```

Apply matched filter to incoming signal

```
        y = filter(collwav);
        z(m,k) = max(abs(y));
        tp(m,k) = t;

    end
end
```

**Plot the pulse amplitudes**

Plot the amplitudes of the pulses for the scan as a function of time.

```
plot(tp(:),z(:),'.')
xlabel('Time (sec)')
ylabel('Pulse Amplitude')
```
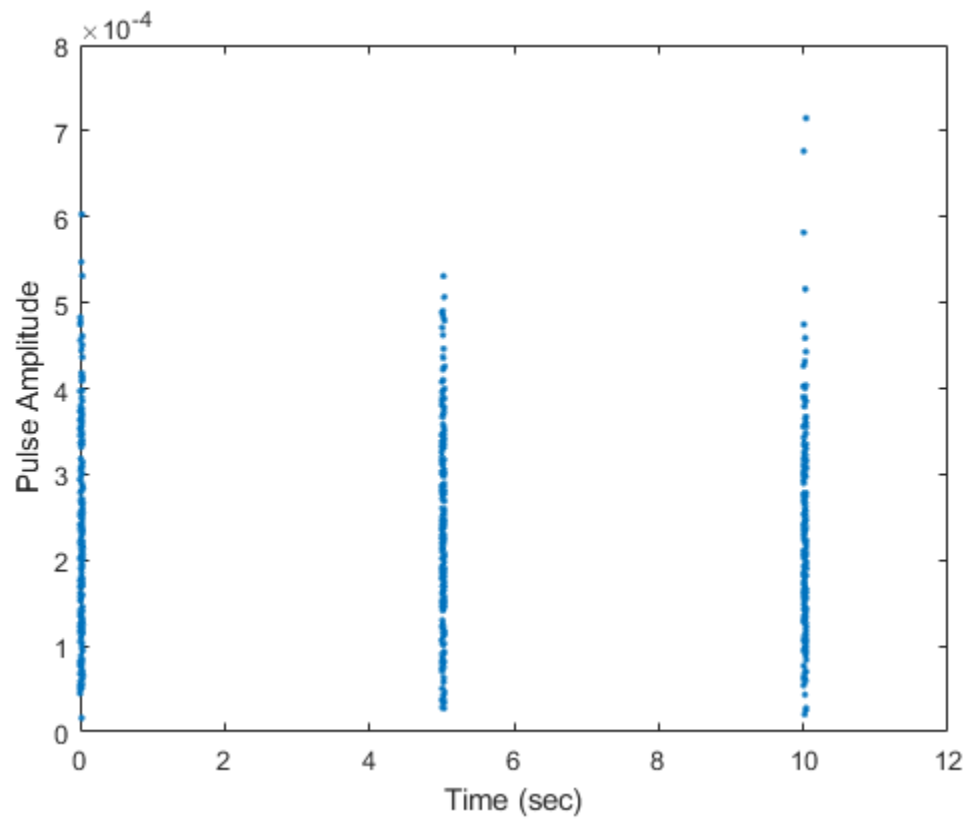
Notice that the pulse amplitudes vary within a scan.

**Histogram the received pulse amplitudes**

```
figure;
hist(z(:),25)
xlabel('Pulse Amplitude')
ylabel('Count')
```
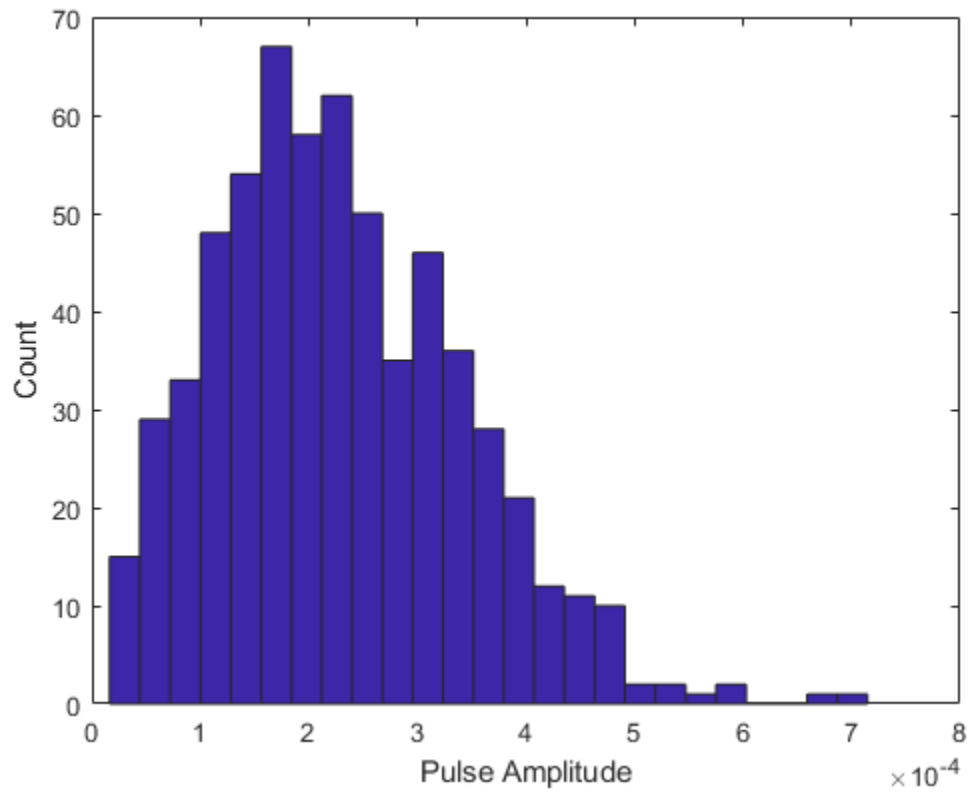
# Swerling 3 Target Models

The example presents a scenario of a rotating monostatic radar and a target having a radar cross-section described by a Swerling 3 model. In this example, the radar and target are stationary.

**Swerling 3 versus Swerling 4 Models**

In Swerling 3 and Swerling 4 target models, the total RCS arises from a target consisting of one large scattering surface with several other small scattering surfaces. The total RCS may vary with every pulse in a scan (Swerling 4) or may be constant over a complete scan consisting of multiple pulses (Swerling 3). In either case, the statistics obey a chi-squared probability density function with *four* degrees of freedom.

**Dwell Time and Radar Scan**

For simplicity, start with a rotating radar having a rotation time of 5 seconds corresponding to a rotation or scan rate of 72 degrees/sec.

```
Trot = 5.0;
scanrate = 360/Trot;
```

The radar has a main half-power beam width (HPBW) of 3.0 degrees. During the time that a target is illuminated by the main beam, radar pulses strike the target and reflect back to the radar. The time period during which the target is illuminated is called the dwell time. This time is also called a scan. The radar will process 3 scans of the target.

```
HPBW = 3.0;
Tdwell = HPBW/scanrate;
Nscan = 3;
```

The number of pulses that arrive on target during the dwell time depends upon the pulse repetition frequency (PRF). PRF is the inverse of the pulse repetition interval (PRI). Assume 5000 pulses are transmitted per second.

```
prf = 5000.0;
pri = 1/prf;
```

The number of pulses in one dwell time is

```
Np = floor(Tdwell*prf);
```

**Set up Swerling 3 radar model**

You create a Swerling 3 target by using the RadarTarget System object™. To effect a Swerling 3 model, set the Model property of the phased.RadarTarget System object to either 'Swerling3' or 'Swerling4'. Both are equivalent. Then, at the first call to the object at the beginning of the scan, set the updatercs argument to true. Set updatercs to false for the remaining calls to the object during the scan. This procedure means that the radar cross section is only updated at the beginning of a scan and remains constant for the remainder of the scan.

Set the target model to 'Swerling3' .

```
tgtmodel = 'Swerling3';
```

**Set up radar model System object components**

Set up the radiating antenna. Assume the operating frequency of the antenna is 1 GHz.

```
fc = 1e9;
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
radiator = phased.Radiator('OperatingFrequency',fc,'Sensor',antenna);
```

Specify the location of the stationary antenna.

```
radarplatform = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
targetplatform = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the object.

```
waveform = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifier.

```
transmitter = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
channel = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',true);
```

Specify the radar target to have a mean RCS of 1 m2 and be of the Swerling model type 3 or 4. You can use Swerling 3 or 4 interchangeably.

```
target = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
collector = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',antenna);
```

Define a matched filter to process the incoming signal.

```
wav = waveform();
filter = phased.MatchedFilter('Coefficients',getMatchedFilter(waveform));
```

**Processing loop for 3 scans of a Swerling 3 target**

1  Generate waveform with unit amplitude
2  Amplify the transmit waveform
3  Radiate the waveform in the desired direction to the target
4  Propagate the waveform to and from the radar target
5  Reflect waveform from radar target
6  Collect radiation to create received signal
7  Match filter received signal

Provide memory for radar return amplitudes.

```
z = zeros(Nscan,Np);
tp = zeros(Nscan,Np);
```

Enter the loop. Set `updatercs` to `true` only for the first pulse of the scan.

```
for m = 1:Nscan
    t0 = (m-1)*Trot;
    t = t0;
    for k = 1:Np

        if k == 1
            updatercs = true;
        else
            updatercs = false;
        end
        t = t + pri;
        txwav = transmitter(wav);
```

Find the radar and target positions

```
        [xradar,vradar] = radarplatform(pri);
        [xtgt,vtgt] = targetplatform(pri);
```

Radiate waveform to target

```
        [~,ang] = rangeangle(xtgt,xradar);
        radwav = radiator(txwav,ang);
```

Propagate waveform to and from the target

```
        propwav = channel(radwav,xradar,xtgt,vradar,vtgt);
```

Reflect waveform from target. Set the `updatercs` flag.

```
        reflwav = target(propwav,updatercs);
```

Collect the received waveform

```
        collwav = collector(reflwav,ang);
```

Apply matched filter to incoming signal

```
        y = filter(collwav);
        z(m,k) = max(abs(y));
        tp(m,k) = t;

    end
end
```

**Plot the pulse amplitudes**

Plot the amplitudes of the pulses for the scan as a function of time.

```
plot(tp(:),z(:),'.')
xlabel('Time (sec)')
ylabel('Pulse Amplitude')
```

Notice that the pulse amplitudes are constant within a scan.

# Swerling 4 Target Models

The example presents a scenario of a rotating monostatic radar and a target having a radar cross-section described by a Swerling 4 model. In this example, the radar and target are stationary.

**Swerling 3 versus Swerling 4 Targets**

In Swerling 3 and Swerling 4 target models, the total RCS arises from a target consisting of one large scattering surface with several other small scattering surfaces. The total RCS may vary with every pulse in a scan (Swerling 4) or may be constant over a complete scan consisting of multiple pulses (Swerling 3). In either case, the statistics obey a chi-squared probability density function with *four* degrees of freedom.

**Dwell Time and Radar Scan**

For simplicity, start with a rotating radar having a rotation time of 5 seconds corresponding to a rotation or scan rate of 72 degrees/sec.

```
Trot = 5.0;
scanrate = 360/Trot;
```

The radar has a main half-power beam width (HPBW) of 3.0 degrees. During the time that a target is illuminated by the main beam, radar pulses strike the target and reflect back to the radar. The time period during which the target is illuminated is called the dwell time. This time is also called a scan. The radar will process 3 scans of the target.

```
HPBW = 3.0;
Tdwell = HPBW/scanrate;
Nscan = 3;
```

The number of pulses that arrive on target during the dwell time depends upon the pulse repetition frequency (PRF). PRF is the inverse of the pulse repetition interval (PRI). Assume 5000 pulses are transmitted per second.

```
prf = 5000.0;
pri = 1/prf;
```

The number of pulses in one dwell time is

```
Np = floor(Tdwell*prf);
```

**Set up a Swerling 4 model**

You create a Swerling 4 target by calling the `RadarTarget` System object™. To effect a Swerling 4 model, set the `Model` property of the `phased.RadarTarget` System object to either `'Swerling3'` or `'Swerling4'`. Both are equivalent. Then, at every call to the object, set the `updatercs` argument to `true`. This argument means that the radar cross section is updated for every pulse in the scan.

Set the target model to `'Swerling4'` .

```
tgtmodel = 'Swerling4';
```

**Set up radar model System object components**

Set up the radiating antenna. Assume the operating frequency of the antenna is 1 GHz.

```
fc = 1e9;
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
radiator = phased.Radiator('OperatingFrequency',fc, ...
    'Sensor',antenna);
```

Specify the location of the stationary antenna.

```
radarplatform = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
targetplatform = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the object.

```
waveform = phased.LinearFMWaveform('PulseWidth',50e-6, ...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifier.

```
transmitter = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
channel = phased.FreeSpace('OperatingFrequency',fc, ...
    'TwoWayPropagation',true);
```

Specify the radar target to have a mean RCS of 1 m2 and be of the Swerling model type 1 or 2. You can use Swerling 1 or 2 interchangeably.

```
target = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc, ...
    'Model',tgtmodel);
```

Set up the radar collector.

```
collector = phased.Collector('OperatingFrequency',1e9, ...
    'Sensor',antenna);
```

Define a matched filter to process the incoming signal.

```
wav = waveform();
filter = phased.MatchedFilter('Coefficients',getMatchedFilter(waveform));
```

**Processing loop for 3 scans of a Swerling 4 target**

**1** Generate waveform with unit amplitude

**2** Amplify the transmit waveform

**3** Radiate the waveform in the desired direction to the target

**4** Propagate the waveform to and from the radar target

**5** Reflect waveform from radar target.

**6** Collect radiation to create received signal

**7** Match filter received signal

Provide memory for radar return amplitudes.

```
z = zeros(Nscan,Np);
tp = zeros(Nscan,Np);
```

Enter the loop. Set `updatercs` to `true` only for all pulses of the scan.

```
for m = 1:Nscan
    t0 = (m-1)*Trot;
    t = t0;
    updatercs = true;
    for k = 1:Np

        t = t + pri;
        txwav = transmitter(wav);
```

Find the radar and target positions

```
        [xradar,vradar] = radarplatform(pri);
        [xtgt,vtgt] = targetplatform(pri);
```

Radiate waveform to target

```
        [~,ang] = rangeangle(xtgt,xradar);
        radwav = radiator(txwav,ang);
```

Propagate waveform to and from the target

```
        propwav = channel(radwav,xradar,xtgt,vradar,vtgt);
```

Reflect waveform from target. Set the `updatercs` flag.

```
        reflwav = target(propwav,updatercs);
```

Collect the received waveform

```
        collwav = collector(reflwav,ang);
```

Apply matched filter to incoming signal

```
        y = filter(collwav);
        z(m,k) = max(abs(y));
        tp(m,k) = t;

    end
end
```

**Plot the pulse amplitudes**

Plot the amplitudes of the pulses for the scan as a function of time.

```
plot(tp(:),z(:),'.')
xlabel('Time (sec)')
ylabel('Pulse Amplitude')
```

Notice that the pulse amplitudes vary within a scan.

**Histogram the received pulse amplitudes**

```
hist(z(:),25)
xlabel('Pulse Amplitude')
```

**10**
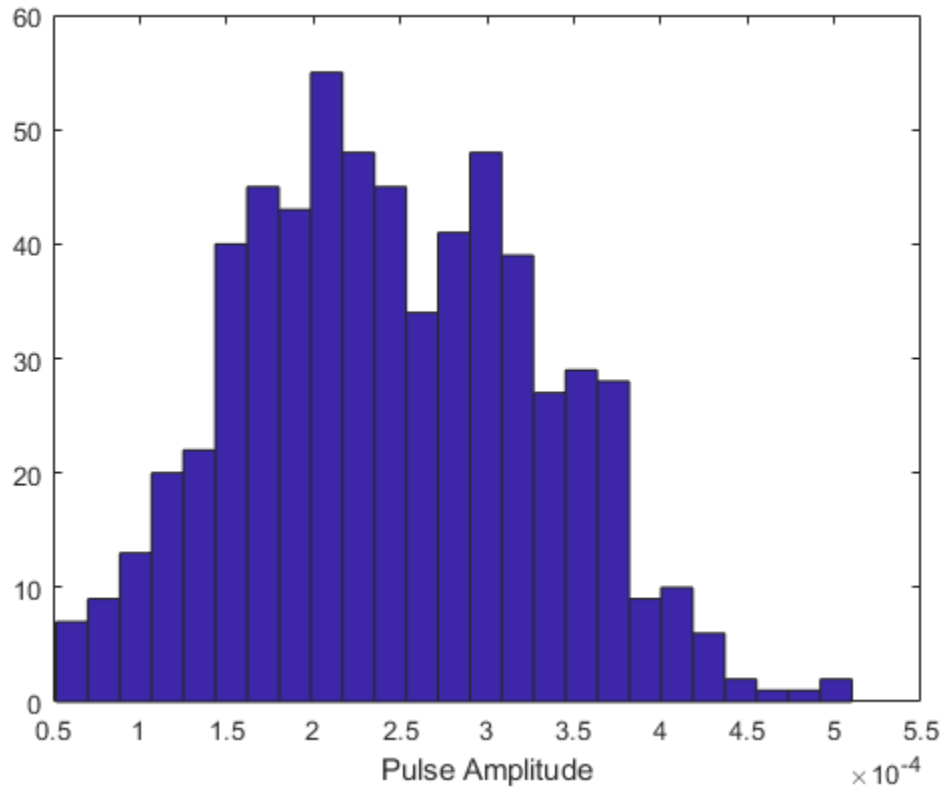
# Coordinate Systems and Motion Modeling

# Rectangular Coordinates

## Definitions of Coordinates

Construct a rectangular, or Cartesian, coordinate system for three-dimensional space by specifying three mutually orthogonal coordinate axes. The following figure shows one possible specification of the coordinate axes.
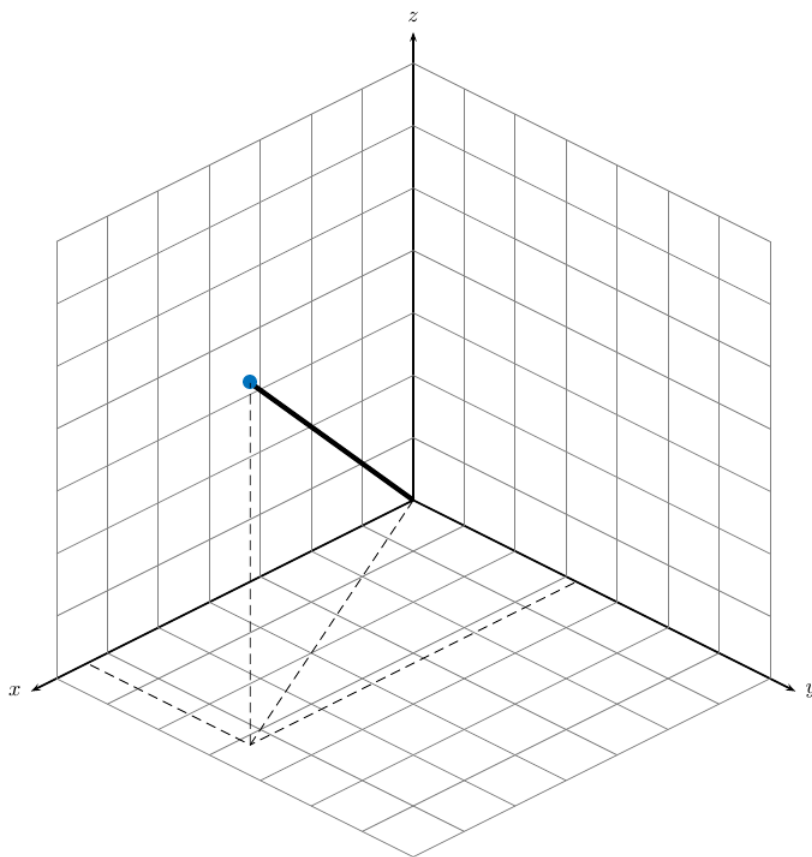


Rectangular coordinates specify a position in space in a given coordinate system as an ordered 3-tuple of real numbers, $(x,y,z)$, with respect to the origin $(0,0,0)$. Considerations for choosing the origin are discussed in "Global and Local Coordinate Systems" on page 10-17.

You can view the 3-tuple as a point in space, or equivalently as a vector in three-dimensional Euclidean space. Viewed as a vector space, the coordinate axes are basis vectors and the vector gives

the direction to a point in space from the origin. Every vector in space is uniquely determined by a linear combination of the basis vectors. The most common set of basis vectors for three-dimensional Euclidean space are the standard unit basis vectors:

$\{[1\ 0\ 0], [0\ 1\ 0], [0\ 0\ 1]\}$

## Notation for Vectors and Points

In Phased Array System Toolbox software, you specify both coordinate axes and points as column vectors.

---

**Note** In this software, all coordinate vectors are column vectors. For convenience, the documentation represents column vectors in the format [x y z] without transpose notation.

---

Both the vector notation [x y z] and point notation (x,y,z) are used interchangeably. The interpretation of the column vector as a vector or point depends on the context. If the column vector specifies the axes of a coordinate system or direction, it is a vector. If the column vector specifies coordinates, it is a point.

## Orthogonal Basis and Euclidean Norm

Any three linearly independent vectors define a basis for three-dimensional space. However, this software assumes that the basis vectors you use are orthogonal.

The standard distance measure in space is the $l^2$ norm, or Euclidean norm. The Euclidean norm of a vector [x y z] is defined by:

$$\sqrt{x^2 + y^2 + z^2}$$

The Euclidean norm gives the length of the vector measured from the origin as the hypotenuse of a right triangle. The distance between two vectors [$x_0\ y_0\ z_0$] and [$x_1\ y_1\ z_1$] is:

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

## Orientation of Coordinate Axes

Given an orthonormal set of basis vectors representing the coordinate axes, there are multiple ways to orient the axes. The following figure illustrates one such orientation, called a *right-handed* coordinate system. The arrows on the coordinate axes indicate the positive directions.
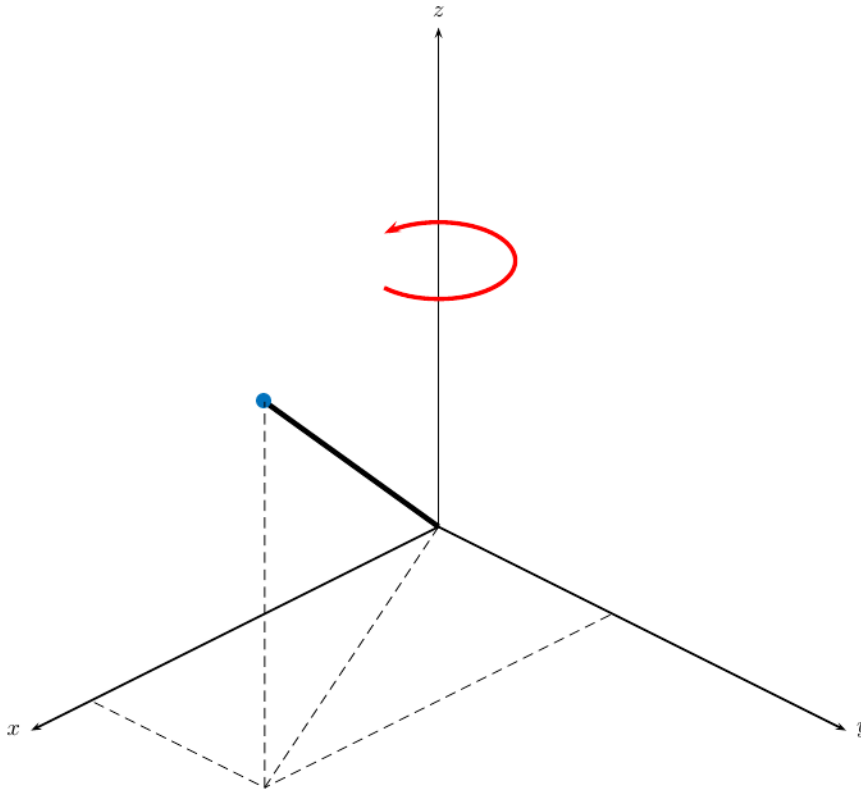
If you take your right hand and point it along the positive *x*-axis with your palm facing the positive *y*-axis and extend your thumb, your thumb indicates the positive direction of the *z*-axis.

## Rotations and Rotation Matrices

In transforming vectors in three-dimensional space, rotation matrices are often encountered. Rotation matrices are used in two senses: they can be used to rotate a vector into a new position or they can be used to rotate a coordinate basis (or coordinate system) into a new one. In this case, the vector is left alone but its components in the new basis will be different from those in the original basis. In Euclidean space, there are three basic rotations: one each around the x, y and z axes. Each rotation is specified by an angle of rotation. The rotation angle is defined to be positive for a rotation that is counterclockwise when viewed by an observer looking along the rotation axis towards the origin. Any arbitrary rotation can be composed of a combination of these three *(Euler's rotation theorem)*. For example, you can rotate a vector in any direction using a sequence of three rotations:
$\mathbf{v}' = A\mathbf{v} = R_z(\gamma)R_y(\beta)R_x(\alpha)\mathbf{v}$.

The rotation matrices that rotate a vector around the x, y, and z-axes are given by:

- Counterclockwise rotation around x-axis

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

- Counterclockwise rotation around y-axis

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

- Counterclockwise rotation around z-axis

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The following three figures show what positive rotations look like for each rotation axis:
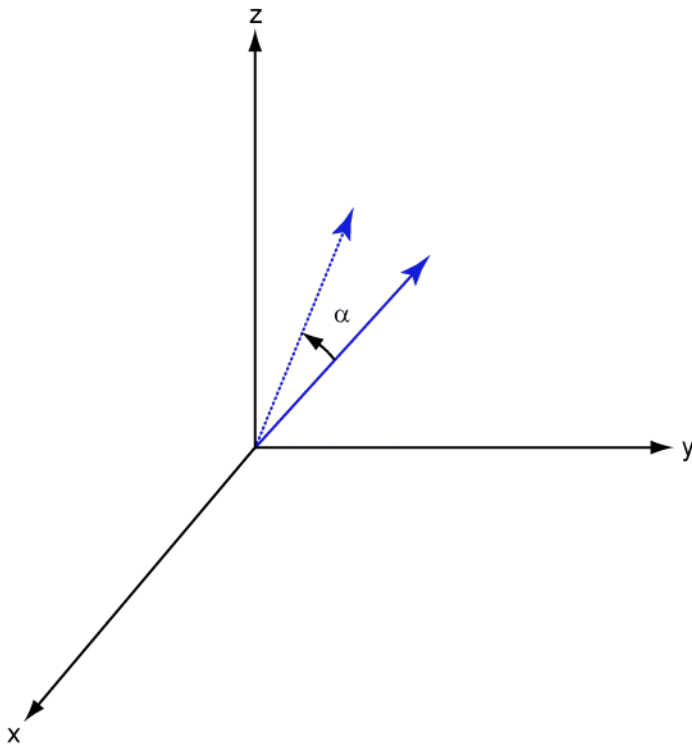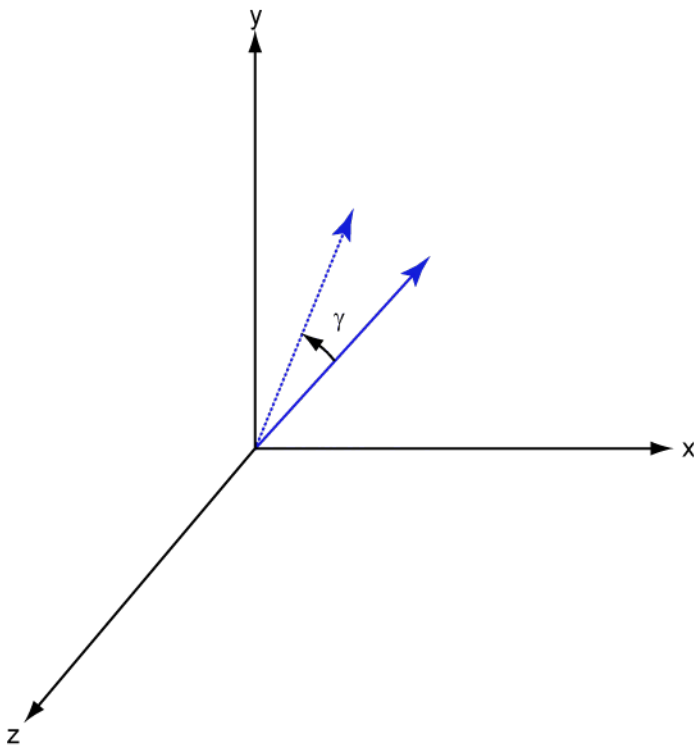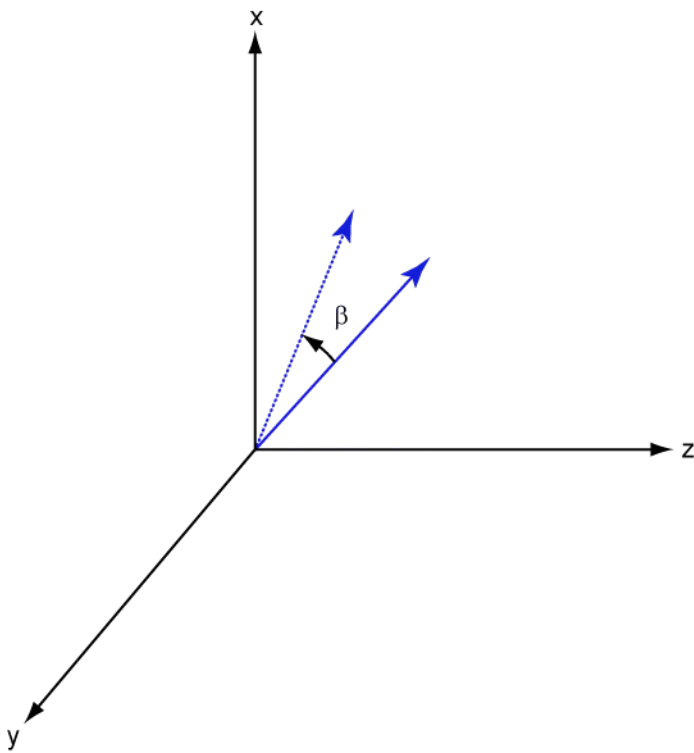
For any rotation, there is an inverse rotation satisfying $A^{-1}A = 1$. For example, the inverse of the x-axis rotation matrix is obtained by changing the sign of the angle:

$$R_x^{-1}(\alpha) = R_x(-\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} = R_x'(\alpha)$$

This example illustrates a basic property: the inverse rotation matrix is the transpose of the original. Rotation matrices satisfy *A'A = 1*, and consequently *det(A) = 1*. Under rotations, vector lengths are preserved as well as the angles between vectors.

We can think of rotations in another way. Consider the original set of basis vectors, **i**, **j**, **k**, and rotate them all using the rotation matrix *A*. This produces a new set of basis vectors **i'**, **j'** **k'** related to the original by:

**i'** = *A***i**

**j'** = *A***j**

**k'** = *A***k**

Using the transpose, you can write the new basis vectors as a linear combinations of the old basis vectors:

$$\begin{bmatrix} \mathbf{i'} \\ \mathbf{j'} \\ \mathbf{k'} \end{bmatrix} = A' \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \end{bmatrix}$$

Now any vector can be written as a linear combination of either set of basis vectors:

$$\mathbf{v} = v_x \mathbf{i} + v_y \mathbf{j} + v_z \mathbf{k} = v'_x \mathbf{i'} + v'_y \mathbf{j'} + v'_z \mathbf{k'}$$

Using algebraic manipulation, you can derive the transformation of components for a fixed vector when the basis (or coordinate system) rotates. This transformation uses the transpose of the rotation matrix.

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = A^{-1} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = A' \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

The next figure illustrates how a vector is transformed as the coordinate system rotates around the x-axis. The figure after shows how this transformation can be interpreted as a rotation *of the vector* in the opposite direction.
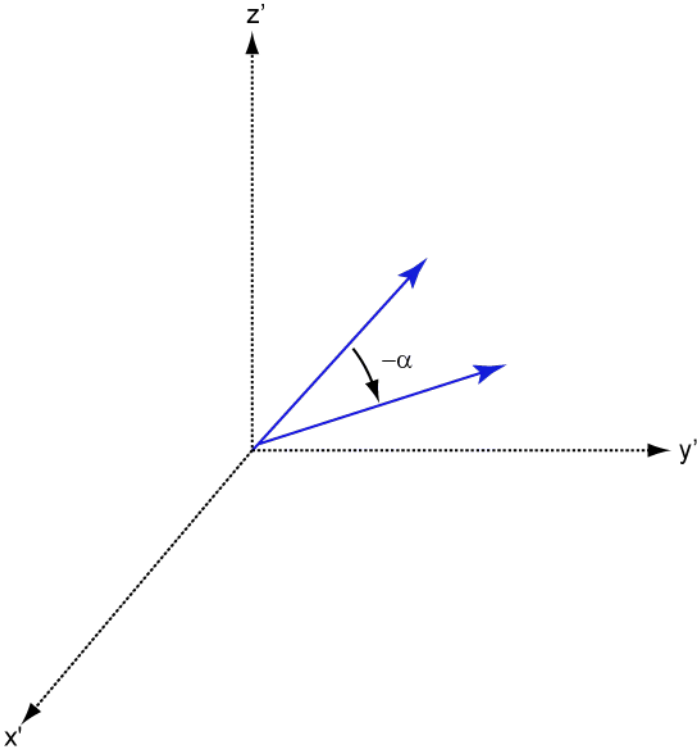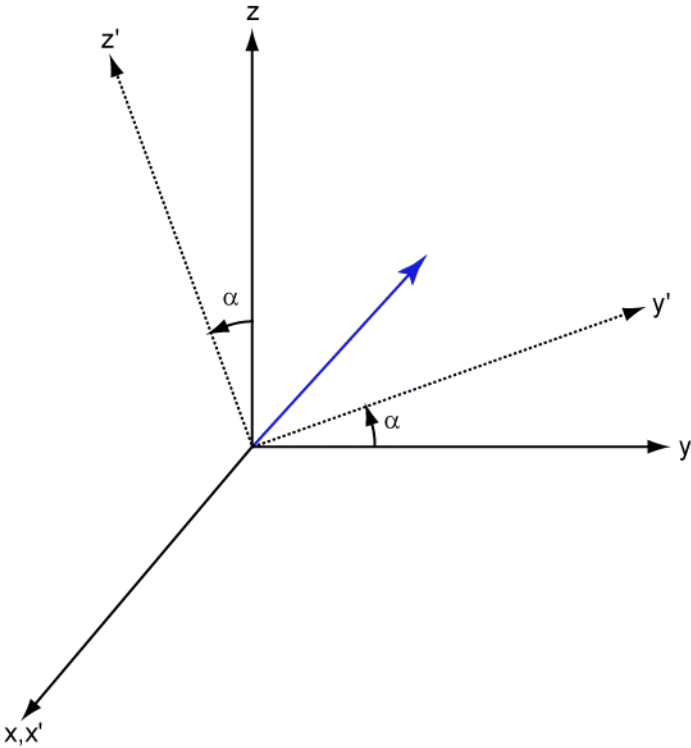
**See Also**

**More About**

*   "Global and Local Coordinate Systems" on page 10-17

# Spherical Coordinates

## Support for Spherical Coordinates

Spherical coordinates describe a vector or point in space with a distance and two angles. The distance, $R$, is the usual Euclidean norm. There are multiple conventions regarding the specification of the two angles. They include:

- Azimuth and elevation angles
- Phi and theta angles
- $u$ and $v$ coordinates

Phased Array System Toolbox software natively supports the azimuth/elevation representation. The software also provides functions for converting between the azimuth/elevation representation and the other representations. See "Phi and Theta Angles" on page 10-11 and "U and V Coordinates" on page 10-12.

## Azimuth and Elevation Angles

In Phased Array System Toolbox software, the predominant convention for spherical coordinates is as follows:

- Use the azimuth angle, $az$, and the elevation angle, $el$, to define the location of a point on the unit sphere.
- Specify all angles in degrees.
- List coordinates in the sequence ($az,el,R$).

The azimuth angle of a vector is the angle between the $x$-axis and the orthogonal projection of the vector onto the $xy$ plane. The angle is positive in going from the $x$ axis toward the $y$ axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the $xy$-plane. The angle is positive when going toward the positive $z$-axis from the $xy$ plane. By default, the boresight direction of an element or array is aligned with the positive $x$-axis. The boresight direction is the direction of the main lobe of an element or array.

**Note** The elevation angle is sometimes defined in the literature as the angle a vector makes with the positive $z$-axis. The MATLAB® and Phased Array System Toolbox products do not use this definition.

This figure illustrates the azimuth angle and elevation angle for a vector shown as a green solid line.
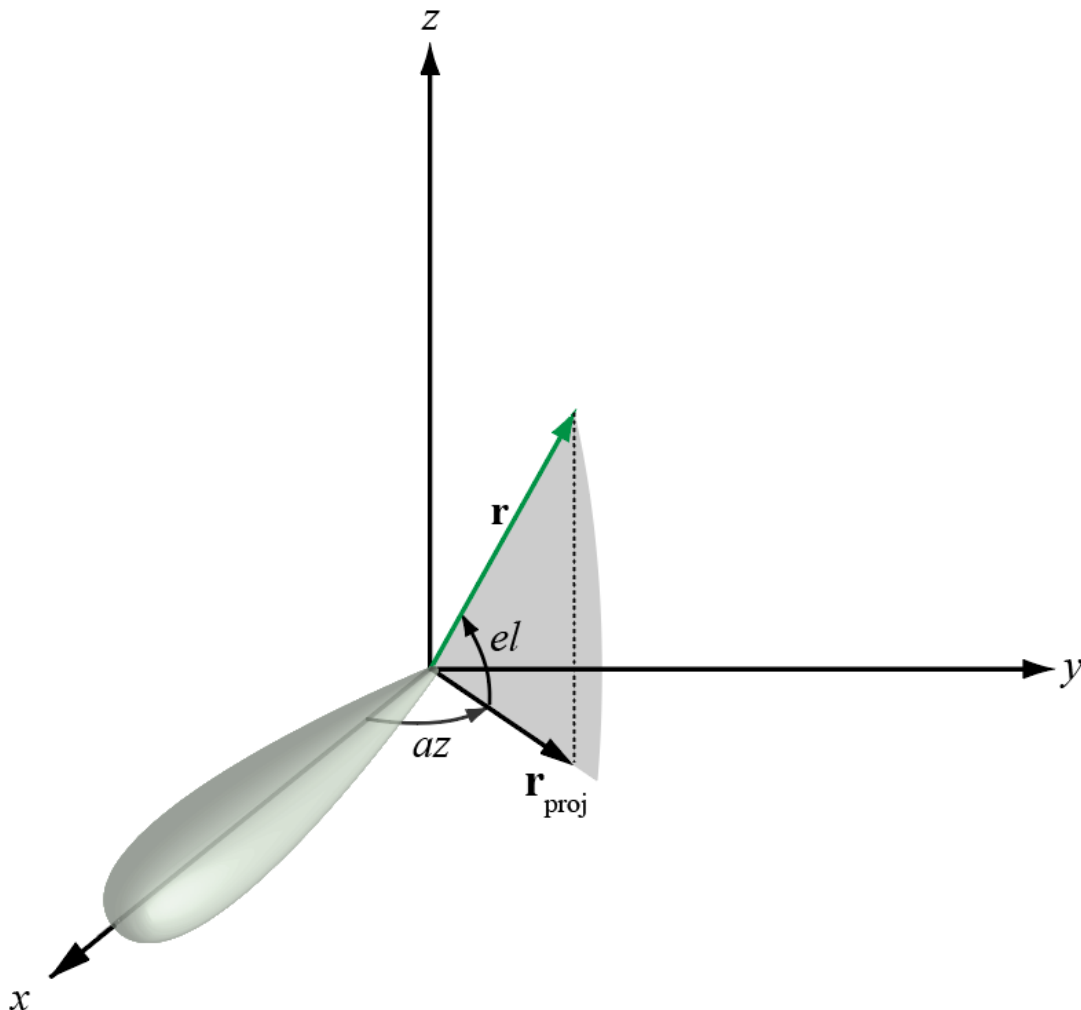
## Phi and Theta Angles

As an alternative to azimuth and elevation angles, you can use angles denoted by $\varphi$ and $\theta$ to express the location of a point on the unit sphere. To convert the $\varphi/\theta$ representation to and from the corresponding azimuth/elevation representation, use coordinate conversion functions, `phitheta2azel` and `azel2phitheta`.

The phi angle ($\varphi$) is the angle from the positive *y*-axis to the vector's orthogonal projection onto the *yz* plane. The angle is positive toward the positive *z*-axis. The phi angle is between 0 and 360 degrees. The theta angle ($\theta$) is the angle from the *x*-axis to the vector itself. The angle is positive toward the *yz* plane. The theta angle is between 0 and 180 degrees.

The figure illustrates phi and theta for a vector that appears as a green solid line.
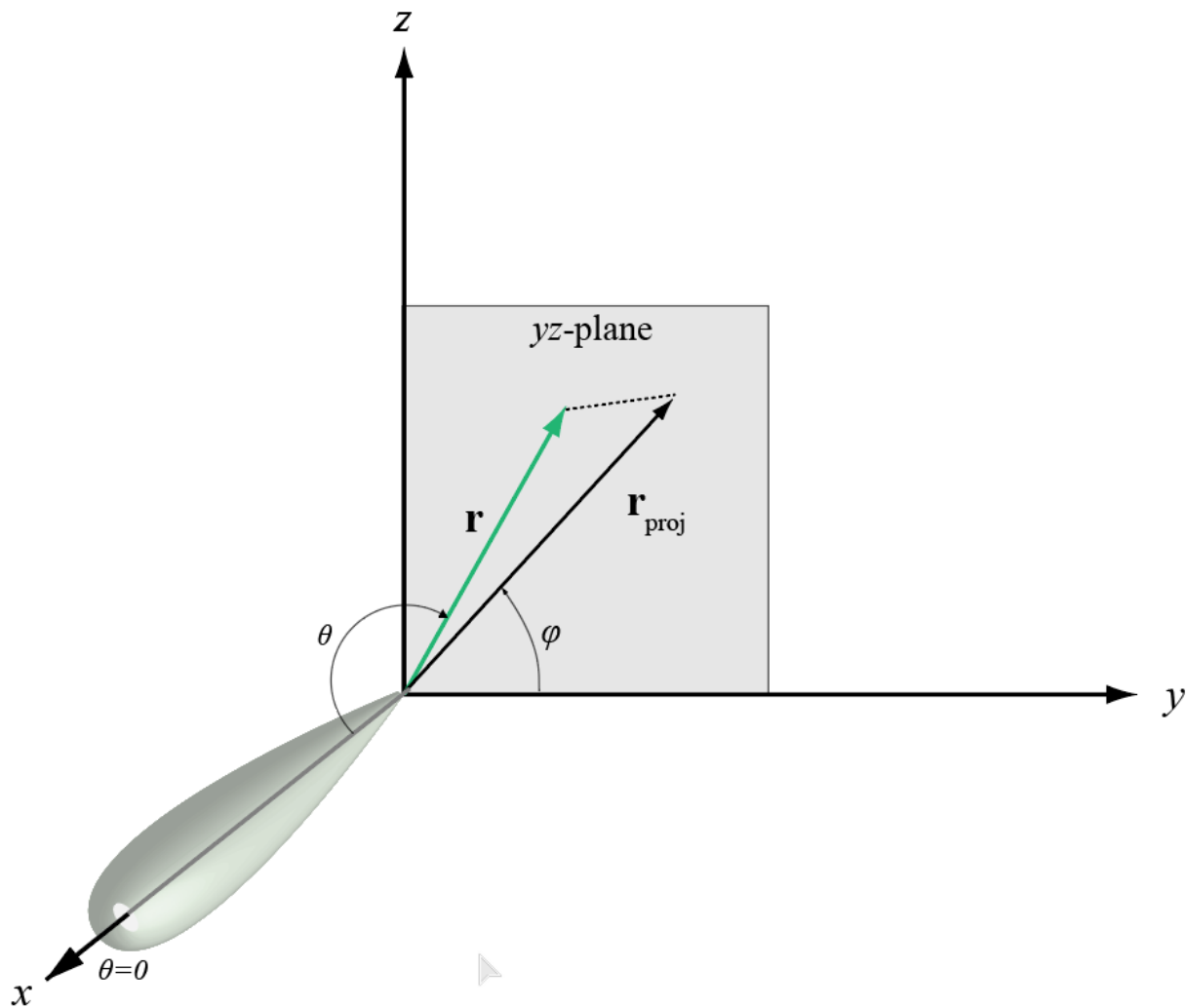
The coordinate transformations between φ/θ and *az/el* are described by the following equations

$\sin el = \sin\phi \sin\theta$
$\tan az = \cos\phi \tan\theta$
$\cos\theta = \cos el \cos az$
$\tan\phi = \tan el / \sin az$

## U and V Coordinates

In radar applications, it is often useful to parameterize the hemisphere $x \geq 0$ using coordinates denoted by *u* and *v*.

- To convert the φ/θ representation to and from the corresponding *u/v* representation, use coordinate conversion functions `phitheta2uv` and `uv2phitheta`.

- To convert the azimuth/elevation representation to and from the corresponding *u*/*v* representation, use coordinate conversion functions `azel2uv` and `uv2azel`.

You can define *u* and *v* in terms of φ and θ:

$$u = \sin\theta\cos\phi$$
$$v = \sin\theta\sin\phi$$

In these expressions, φ and θ are the phi and theta angles, respectively.

To convert azimuth and elevation to *u* and *v* use the transformation

$$u = \cos el \sin az$$
$$v = \sin el$$

which is valid only in the range *abs(az)≤=90*.

The values of *u* and *v* satisfy the inequalities

$$-1 \le u \le 1$$
$$-1 \le v \le 1$$
$$u^2 + v^2 \le 1$$

Conversely, the phi and theta angles can be written in terms of *u* and *v* using

$$\tan\phi = v/u$$
$$\sin\theta = \sqrt{u^2 + v^2}$$

The azimuth and elevation angles can also be written in terms of *u* and *v*:

$$\sin el = v$$
$$\tan az = \frac{u}{\sqrt{1 - u^2 - v^2}}$$

## Conversion between Rectangular and Spherical Coordinates

The following equations define the relationships between rectangular coordinates and the (*az*,*el*,*R*) representation used in Phased Array System Toolbox software.

To convert rectangular coordinates to (*az*,*el*,*R*):

$$R = \sqrt{x^2 + y^2 + z^2}$$
$$az = \tan^{-1}(y/x)$$
$$el = \tan^{-1}(z/\sqrt{x^2 + y^2})$$

To convert (*az*,*el*,*R*) to rectangular coordinates:

$$x = R\cos(el)\cos(az)$$
$$y = R\cos(el)\sin(az)$$
$$z = R\sin(el)$$

When specifying a target's location with respect to a phased array, it is common to refer to its distance and direction from the array. The distance from the array corresponds to *R* in spherical coordinates. The direction corresponds to the azimuth and elevation angles.

---

**Tip**  To convert between rectangular coordinates and (*az*,*el*,*R*), use the MATLAB functions `cart2sph` and `sph2cart`. These functions specify angles in radians. To convert between degrees and radians, use `deg2rad` and `rad2deg`.

---

## Broadside Angles

Broadside angles are useful when describing the response of uniform linear arrays (ULA) to impinging signals. The array response depends directly on the broadside angle and not on the azimuth and elevation angles individually. The figure below shows a ULA with elements along the *y*-axis shown in blue. Consider a plane orthogonal to the ULA axis at the origin. The broadside angle, β, is the angle between the plane and the signal direction. To compute the broadside angle, construct a line from any point on the signal path to the plane, orthogonal to the plane. The angle between these two lines is the broadside angle and lies in the interval [–90°,90°]. The broadside angle is positive when measured toward the positive direction of the array axis. When β is zero, the signal path is orthogonal to the array axis. When β = ±90, the signal path lies along the array axis. All signal paths having the same broadside angle form a cone around the ULA axis.
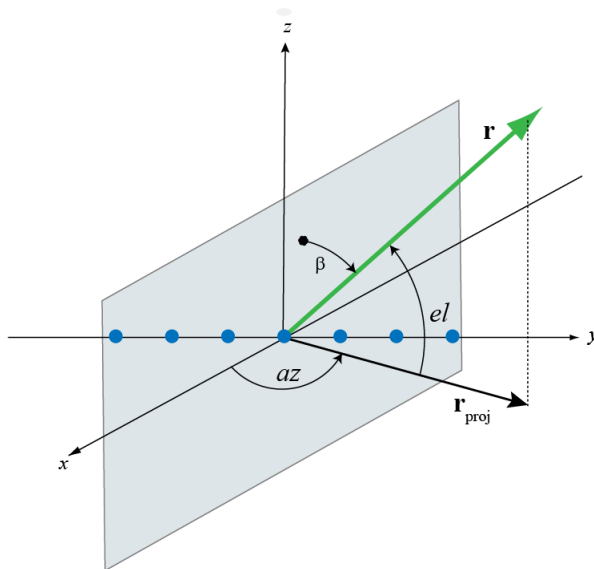


The conversion from azimuth angle, *az*, and elevation angle, *el*, to broadside angle, β, is

$$\beta = \sin^{-1}(\sin(az)\cos(el))$$

This equation shows that

- For an elevation angle of zero, the broadside angle equals the azimuth angle.
- Elevation angles equally above and below the *xy* plane result in identical broadside angles.

You can convert from broadside angle to azimuth angle but you must specify the elevation angle

$$az = \sin^{-1}\left(\frac{\sin\beta}{\cos(el)}\right)$$

Because the signals paths for a given broadside angle, $\beta$, form a cone around the array axis, you cannot specify the elevation angle arbitrarily. The elevation angle and broadside angle must satisfy

$$|el| + |\beta| \leq 90$$

As an example, the figure below shows a ULA with elements spaced $d$ meters apart along the $y$-axis. The ULA is illuminated by a plane wave emitted from a point source in the far field. Choose the azimuth angle to be 90°. Then the signal direction lies in the $yz$-plane and satisfies $\sin\beta = \cos(el)$. The broadside angle is the complement of the elevation angle.
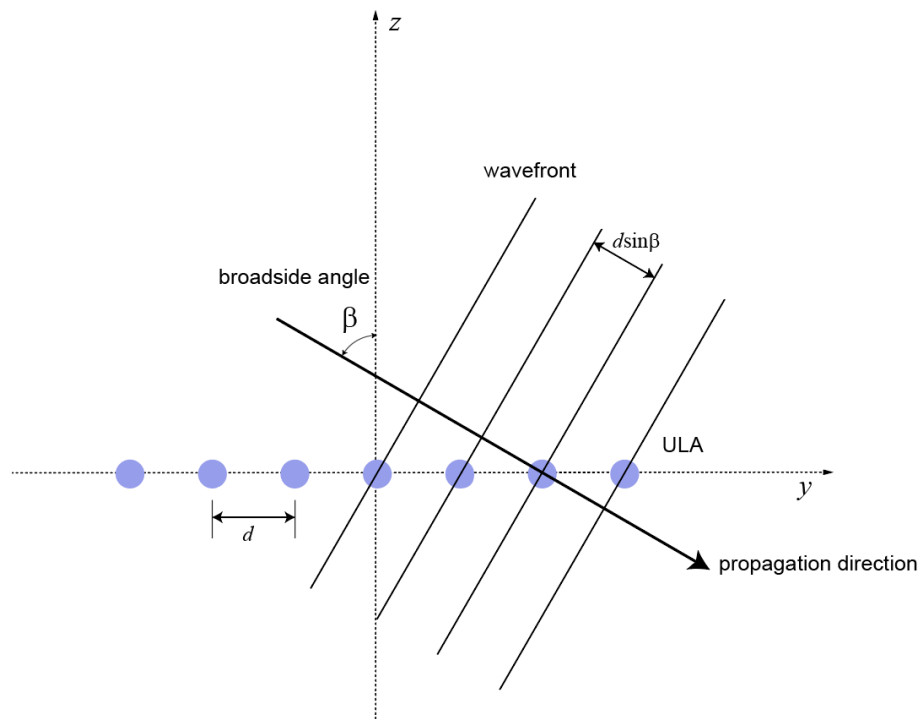


Because of the angle of arrival, the array elements are not simultaneously illuminated by the plane wave. The additional distance the incident wave travels between array elements is $d\,\sin\beta$ where $d$ is the distance between array elements. The constant time delay, $\tau$, between array elements is

$$\tau = \frac{d\sin\beta}{c},$$

where $c$ is the speed of the wave.

For broadside angles of ±90°, the signal is incident on the array parallel to the array axis and the time delay between sensors equals $\pm d/c$. For a broadside angle of zero, the plane wave illuminates all elements of the ULA simultaneously and the time delay between elements is zero.

Phased Array System Toolbox software provides functions `az2broadside` and `broadside2az` for converting between azimuth and broadside angles.

## Convert Between Broadside Angles and Azimuth and Elevation

The following examples show how to use the `az2broadside` and `broadside2az` functions.

A target is located at an azimuth angle of 45° and at an elevation angle of 60° relative to a ULA. Determine the corresponding broadside angle.

```
bsang = az2broadside(45,60)
```

bsang = 20.7048

Calculate the azimuth for an incident signal arriving at a broadside angle of 45° and an elevation of 20°.

```
az = broadside2az(45,20)
```

az = 48.8063

# Global and Local Coordinate Systems

| **In this section...** |
| --- |
| "Global Coordinate System" on page 10-17 |
| "Local Coordinate Systems" on page 10-17 |
| "Converting Between Global and Local Coordinate Systems" on page 10-29 |
| "Convert Local Spherical Coordinates to Global Rectangular Coordinates" on page 10-29 |
| "Convert Global Rectangular Coordinates to Local Spherical Coordinates" on page 10-30 |

## Global Coordinate System

The *global* coordinate system describes the arena in which your radar or sonar simulation takes place. Within this arena, you can place radar or sonar transmitters and receivers, and targets. These objects can be either stationary or moving. You specify the location and motion of these objects in global coordinates.

You can model the motion of all objects using the `phased.Platform` System object. This System object computes the position and speed of objects using constant-velocity or constant-acceleration models.

You can model the signals that propagate between objects in your scenario. The ray paths that connect transmitters, targets, and receivers are specified in global coordinates. You can propagate signals using these System objects: `phased.FreeSpace`, `phased.WidebandFreeSpace`, `phased.LOSChannel`, or `phased.WidebandLOSChannel`. If you model two-ray multipath propagation using `twoRayChannel`, the boundary plane is set at $z = 0$ in the global coordinate system.

## Local Coordinate Systems

When signals interact with sensors or targets, the interaction is almost always specified as a function of the sensor or target local coordinates. Local coordinate systems are fixed to the antennas and microphones, phased arrays, and targets. They move and rotate with the object. Local coordinates are commonly adapted to the shape and symmetry of the object.

Because signals propagate in the global coordinate system, you need to be able to convert local coordinates to the global coordinates. You do this by constructing a 3-by-3 orthonormal matrix of coordinate axes. The matrix columns represent the three orthogonal direction vectors of the local coordinates expressed in the global coordinate system. The coordinate axes of a local coordinate system must be orthonormal, but they need not be parallel to the global coordinate axes.

When you need to compute the range and arrival angles of a signal, you can use the `rangeangle` function. When you call this function with the source and receiver position expressed in global coordinates, the function returns the range and arrival angles, azimuth and elevation, with respect to the axes of the global system. However, when you pass the orientation matrix as an additional argument, the azimuth and elevation are now defined with respect to the local coordinate system.

You use local coordinates to specify

- the location and orientation of antenna or microphone elements of an array. The beam pattern of an antenna array depends upon the angle of arrival or emission of radiation with respect to the array local coordinates.
- the reflected energy from a target is a function of the incident and reflection angles with respect to the target local coordinate axes.

Two examples of local coordinate systems are

- an airplane may have a local coordinate system with the *x*-axis aligned along the fuselage axis of the body and the *y*-axis pointing along the port wing. Choose the *z*-axis to form a right-handed coordinate system.
- a vehicle-mounted planar phased array may have a local coordinate system adapted to the array. The *x*-axis of the coordinate system may point along the array normal vector.

The following figure illustrates the relationship of local and global coordinate systems in a bistatic radar scenario. The thick solid lines represent the coordinate axes of the global coordinate system. There are two phased arrays: a 5-by-5 transmitting uniform rectangular array (URA) and 5-by-5 receiving URA. Each of the phased arrays carries its own local coordinate system. The target, indicated by the red arrow, also carries a local coordinate system.
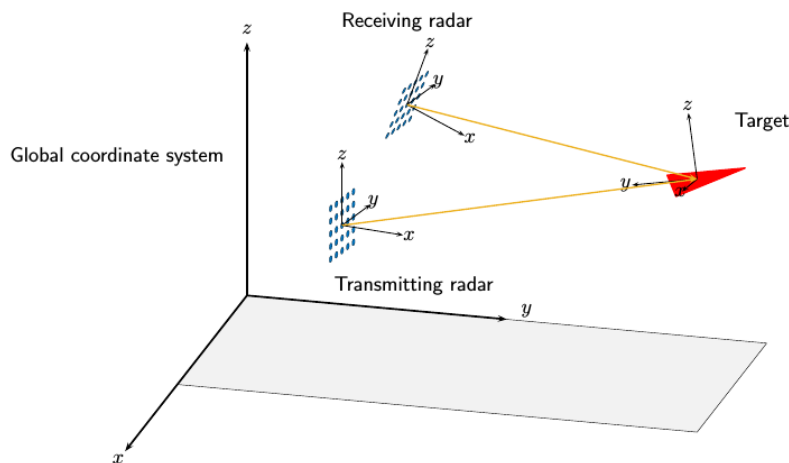


The next few sections review the local coordinate systems used by arrays.

**Local Coordinate Systems of Arrays**

The positions of the elements of any Phased Array System Toolbox array are always defined in a local coordinate system. When you use any of the System objects that create uniform arrays, the array element positions are defined automatically with respect to a predefined local coordinate system. The arrays for which this property holds are the `phased.ULA`, `phased.URA`, `phased.UCA`, `phased.HeterogeneousULA`, and `phased.HeterogeneousURA` System objects. For these System objects, the arrays are described using a few parameters such as element spacing and number of elements. The positions of the elements are then defined with respect to the array origin located at *(0,0,0)* which is the geometric center of the array. The geometric center is a good approximation to the array *phase center*. The *phase center* of an array is the point from which the radiating waves appear to emanate when observed in the far field. For example, for a ULA with an odd number of elements, the elements are located at distances (*-2d,-d,0,d,2d*) along the array axis.

There are array System objects for which you must explicitly specify the element coordinates. You can use these objects for creating arbitrary array shapes. These objects are the `phased.ConformalArray` and `phased.HeterogeneousConformalArray` System objects. For these arrays, the phase center of the array need not coincide with the array origin or geometric center.

**Element Boresight Directions**

In addition to element positions, you need to specify the element orientations, that is, the directions in which the elements point. Some elements are highly directional — most of their radiated energy flows in one direction, called the main response axis (MRA). Others are omnidirectional. Element orientation is the pointing direction of the MRA. You specify element orientation using azimuth and elevation in the array local coordinate system. The direction that an antenna or microphone MRA faces when transmitting or receiving a signal is also called the *boresight* or *look* direction. For the uniform arrays, all boresight directions of all elements are determined by array parameters. For conformal arrays, you specify the boresight direction of each element independently.

**Local Coordinate System of Uniform Linear Array**

**Array Origin and Phase Center**

A uniform linear array (ULA) is an array of antenna or microphone elements that are equidistantly spaced along a straight line. In the Phased Array System Toolbox, the `phased.ULA` System object creates a ULA array. The geometry of the ULA orientation of its elements is determined by three parameters: the number of elements, the distance between elements, and the `ArrayAxis` property. For the ULA, the local coordinate system is adapted to the array — the elements are automatically assigned positions in the local coordinate system.

The positions of elements in the array are determined by the `ArrayAxis` property which can take the values `'x'`, `'y'` or `'z'`. The array axis property determines the axis on which all elements are defined. For example, when the `ArrayAxis` property is set to `'x'`, the array elements lie along the *x*-axis. The elements are positioned symmetrically with respect to the origin. Therefore, the geometric center of the array lies at the origin of the coordinate system.

This figure shows a four-element ULA with directional elements in a local right-handed coordinate system. The elements lie on the *y*-axis with their boresight axes pointing in the *x*-direction. In this case, the `ArrayAxis` property is set to `'y`.

ULA Array
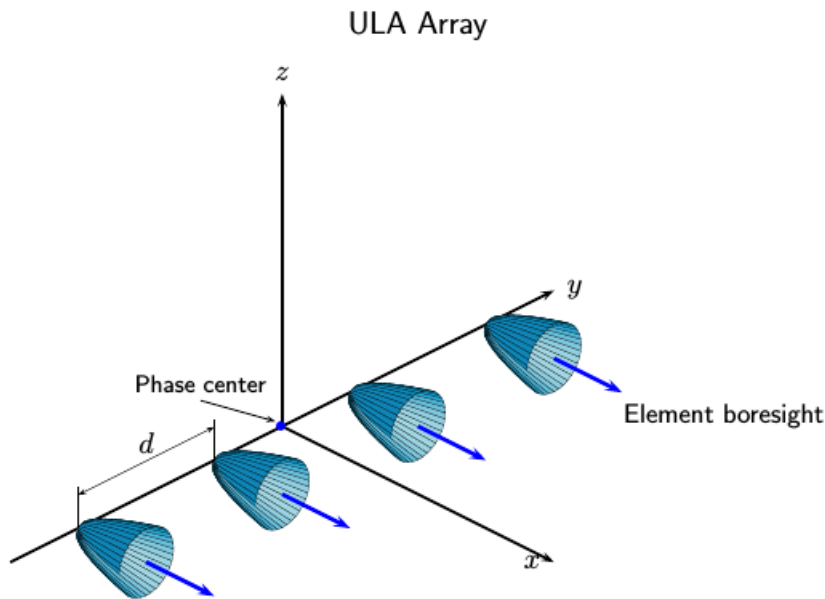
**ULA Element Boresight Direction**

In a ULA, the boresight directions of every element point in the same direction. The direction is orthogonal to the array axis. This direction depends upon the choice of `ArrayAxis` property.

| ArrayAxis Property Value | Element Positions and Boresight Directions |
|---|---|
| `'x'` | Array elements lie on the *x*-axis. Element boresight vectors point along the *y*-axis. |
| `'y'` | Array elements lie on the *y*-axis. Element boresight vectors point along the *x*-axis. |
| `'z'` | Array elements lie on the *z*-axis. Element boresight vectors point along the *x*-axis. |

**Local Coordinates Adapted to Uniform Linear Array**

Construct two examples of a uniform linear array and display the coordinates of the elements with respect to the local coordinate systems defined by the arrays.

First, construct a 4-element ULA with one-half meter element spacing.

```
sULA = phased.ULA('NumElements',4,'ElementSpacing',0.5);
ElementLocs = getElementPosition(sULA)
```

ElementLocs = *3×4*

```
         0           0           0           0
```

```
   -0.7500   -0.2500    0.2500    0.7500
         0         0         0         0
```

```
viewArray(sULA)
```

**Array Geometry**



```
Aperture Size:
Y axis = 2 m
Element Spacing:
Δ y = 500 mm
Array Axis: Y axis
```
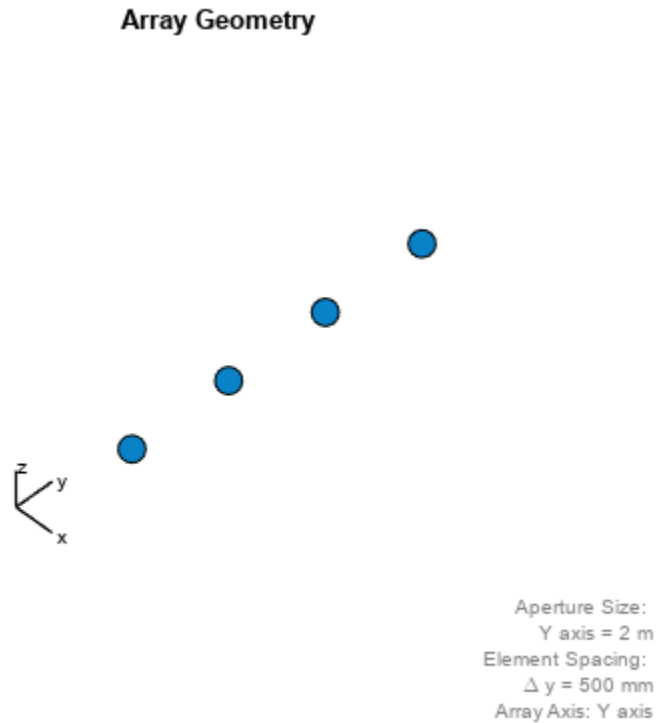
The origin of the array-centric local coordinate system is set to the phase center of the array. The phase center is the average value of the array element positions.

```
disp(mean(ElementLocs'))
```

```
     0     0     0
```

Because the array has an even number of elements, no element of the array actually lies at the phase center *(0,0,0)*.

Next construct a 5-element ULA with thirty-centimeter element spacing.

```
sULA1 = phased.ULA('NumElements',5,'ElementSpacing',0.3);
ElementLocs = getElementPosition(sULA1)
```

```
ElementLocs = 3×5
```

```
        0         0        0        0        0
  -0.6000   -0.3000        0   0.3000   0.6000
        0         0        0        0        0
```

```
viewArray(sULA1)
```
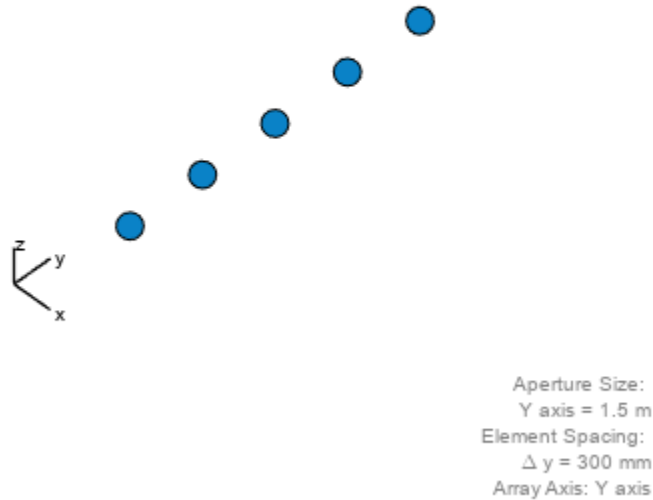
**10-21**

**Array Geometry**



Aperture Size:
Y axis = 1.5 m
Element Spacing:
△ y = 300 mm
Array Axis: Y axis

Because the array has an odd number of elements in each row and column, the center element of the array lies at the phase center.

### Local Coordinate System of Uniform Rectangular Array

**Array Origin and Phase Center**

A uniform rectangular array (URA) is an array of antenna or microphone elements placed on a regular two-dimensional grid. The geometry of a URA and the location and orientation of its elements are determined by several parameters: the dimensions of the array, the distance between elements, and the `ArrayNormal` property. For the URA, the local coordinate system is adapted to the array — the elements are automatically assigned positions in the local coordinate system. The origin of the local coordinate system is the geometric center of the array. The *phase center* of the array coincides with the geometric center. The elements are automatically assigned positions in this local coordinate system. The positions are determined by the `ArrayNormal` property which can take the values `'x'`, `'y'` or `'z'`. All elements lie in a plane passing through the origin and orthogonal to the axis specified in this property. For example, when the `ArrayNormal` property is set to `'x'`, the array elements lie in the *yz*-plane as shown in the figure. The figure shows a 2-by-4 element URA with elements spaced $d$ meters apart in both the $y$ and $z$ directions.
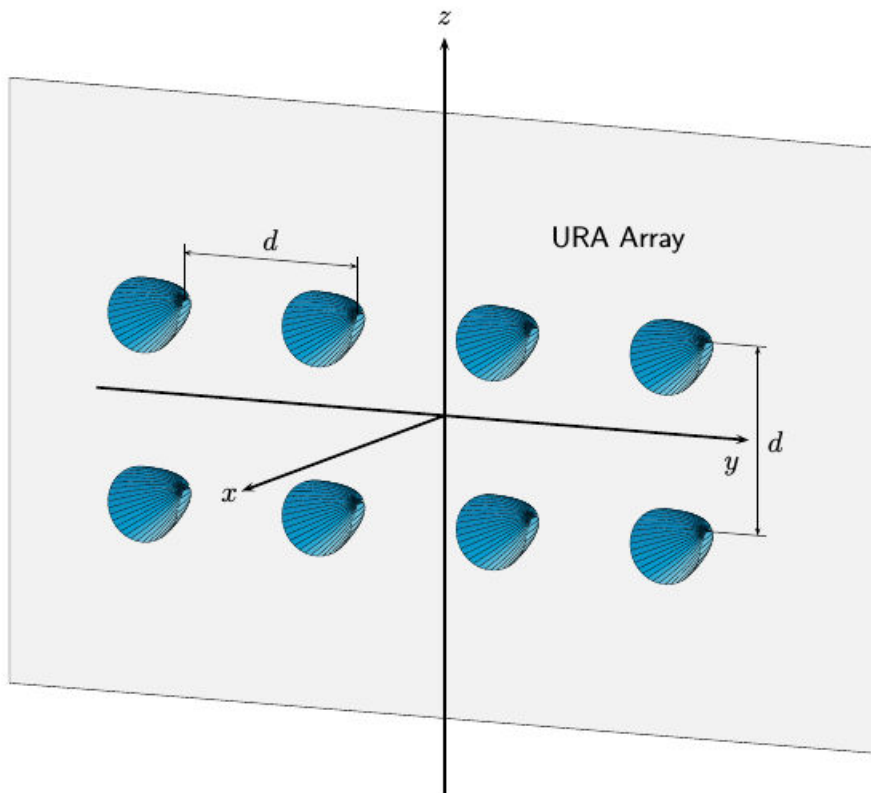
**Element Boresight Direction**

In a URA, like the ULA, the boresight directions of every element point in the same direction. You control this direction using the `ArrayNormal` property. For the URA shown in the preceding figure, the `ArrayNormal` property is set to `'x'`. Then, element boresights point along the x-axis.

| ArrayNormal Property Value | Element Positions and Boresight Directions |
|---|---|
| `'x'` | Array elements lie on the yz-plane. Element boresight vectors point along the x-axis. |
| `'y'` | Array elements lie on the zx-plane. Element boresight vectors point along the y-axis. |
| `'z'` | Array elements lie on the xy-plane. Element boresight vectors point along the z-axis. |

**Local Coordinates Adapted to Uniform Rectangular Array**

Construct two examples of uniform rectangular arrays and display the coordinates of the elements with respect to the local coordinate systems defined by the arrays.

First, construct a 2-by-4 URA with one-half meter element spacing.

```
sURA = phased.URA('Size',[2 4],'ElementSpacing',[0.5 0.5]);
ElementLocs = getElementPosition(sURA)
```

ElementLocs = *3×8*

```
        0         0         0         0         0         0         0         0
  -0.7500   -0.7500   -0.2500   -0.2500    0.2500    0.2500    0.7500    0.7500
   0.2500   -0.2500    0.2500   -0.2500    0.2500   -0.2500    0.2500   -0.2500
```

```
viewArray(sURA)
```
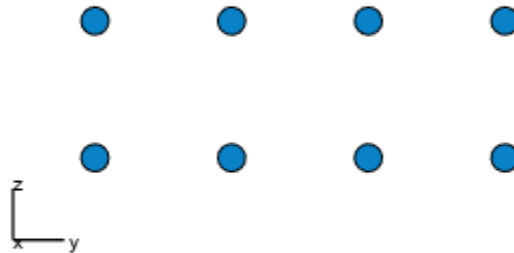


The phase center of the array is the mean value of the array element positions. The origin of the array local coordinate system is set to the phase center of the array.

```
disp(mean(ElementLocs'))
```

```
    0     0     0
```

Because the array has an even number of elements in each row and column, no element of the array actually lies at the phase center *(0,0,0)*.

Next construct a 5-by-3 URA with thirty-centimeter element spacing.

```
sURA1 = phased.URA([5 3],'ElementSpacing',[0.3 0.3]);
ElementLocs = getElementPosition(sURA1)
```

ElementLocs = *3×15*

```
       0          0          0          0          0          0          0          0          0
 -0.3000    -0.3000    -0.3000    -0.3000    -0.3000          0          0          0          0
  0.6000     0.3000          0    -0.3000    -0.6000     0.6000     0.3000          0    -0.3000    -0.6
```

viewArray(sURA1)



**Array Geometry**
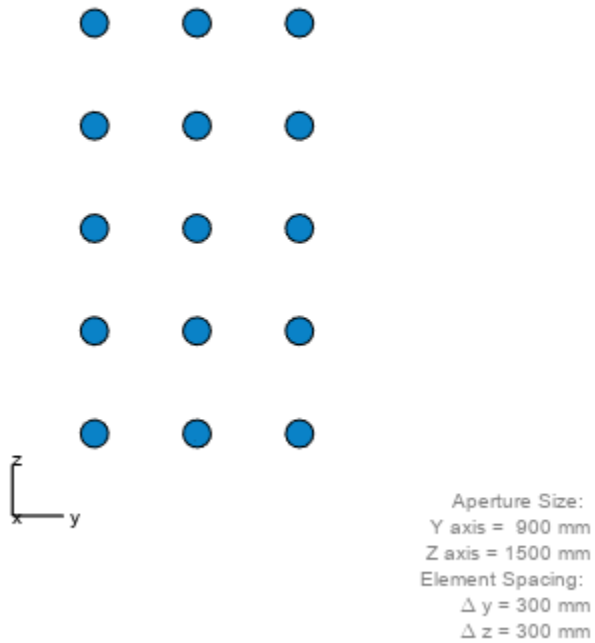
Aperture Size:
Y axis = 900 mm
Z axis = 1500 mm
Element Spacing:
Δ y = 300 mm
Δ z = 300 mm

Because the array has an odd number of elements in each row and column, the center element of the array lies at the phase center.

A signal arrives at the array from a point 1000 meters from along the +x-axis of the global coordinate system. The local URA array is rotated 30 degrees clockwise around the y-axis. Compute the angle of arrival of the signal in the local array axes.

```
laxes = roty(30);
[rng,ang] = rangeangle([1000,0,0]',[0,0,0]',laxes)
```

rng = 1.0000e+03

ang = *2×1*

```
       0
 30.0000
```

### Local Coordinate System of Uniform Circular Array

**Array Origin and Phase Center**

A uniform circular array (UCA) is an array of antenna or microphone elements spaced at equal angles around a circle. The `phased.UCA` System object creates a special case of a UCA. In this case, element boresight directions point away from the array origin like spokes of a wheel. The origin of the local coordinate system is the geometric center of the array. The geometry of the UCA and the location and orientation of its elements is determined by three parameters: the radius of the array, the number of elements, and the `ArrayNormal` property. The elements are automatically assigned positions in the local coordinate system. The positions are determined by the `ArrayNormal` property which can take the values `'x'`, `'y'` or `'z'`. All elements lie in a plane passing through the origin and orthogonal to the axis specified in this property. The *phase center* of the array coincides with the geometric center. For example, when the `ArrayNormal` property is set to `'x'`, the array elements lie in the *yz*-plane as shown in the figure. You can create a more general UCA with arbitrary boresight directions using the `phased.ConformalArray` System object.

This figure shows an 8-element UCA with elements lying in the *yz* plane.
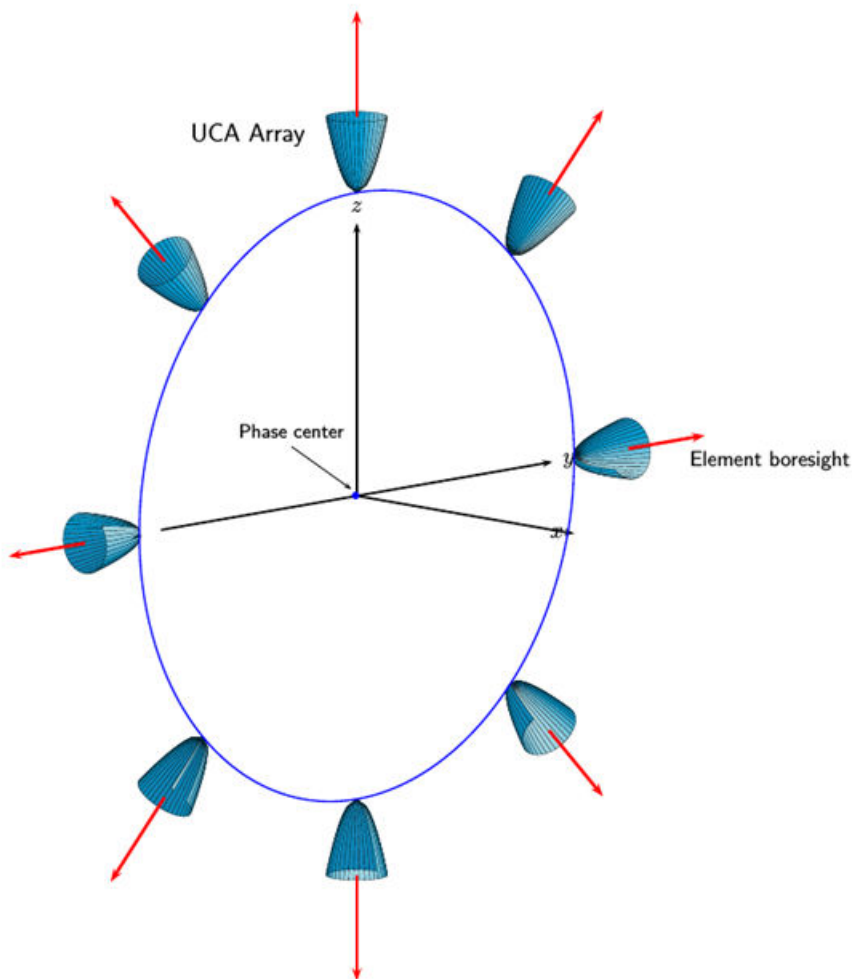
**Element Boresight Directions**

In a UCA defined by a `phased.UCA` System object, element boresight directions point radially outward from the array origin. In the UCA shown in the preceding figure, because the `ArrayNormal` property is set to `'x'`, the element boresight directions point radially outward in the *yz*-plane.

| ArrayNormal Property Value | Element Positions and Boresight Directions |
|---|---|
| `'x'` | Array elements lie on the *yz*-plane. All element boresight vectors lie in the *yz*-plane and point radially-outward from the origin. |
| `'y'` | Array elements lie on the *zx*-plane. All element boresight vectors lie in the *zx*-plane and point radially-outward from the origin. |
| `'z'` | Array elements lie on the *xy*-plane. All element boresight vectors lie in the *xy*-plane and point radially-outward from the origin. |

**Local Coordinate System of Conformal Arrays**

**Array Origin and Phase Center**

You can use `phased.ConformalArray` to create arrays of arbitrary shape. Unlike the case of uniform arrays, you must specify the element positions explicitly. An *N*-element array requires the specification of *N* 3-D coordinates in the array local coordinate system. The origin of a conformal array can be located at any arbitrary point. The boresight directions of the elements of a conformal array need not be parallel. The azimuth and elevation angles defining the boresight directions are with respect to the local coordinate system. The phase center of the array does not need to coincide with the geometric center. The same properties apply to the `phased.HeterogeneousConformalArray` array.

This illustration shows the positions and orientations of a 4-element conformal array.
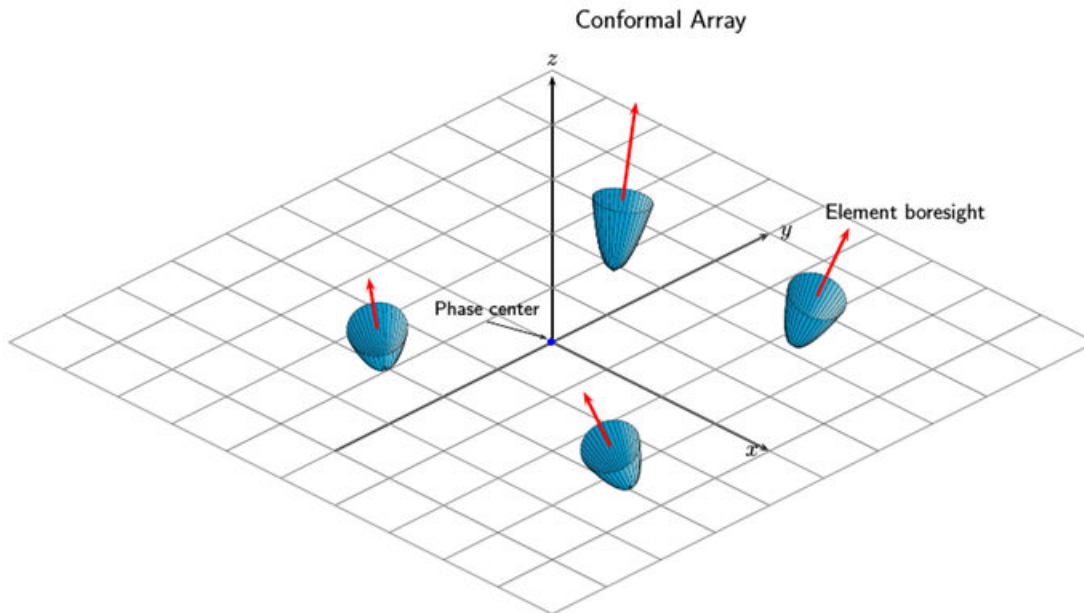
Conformal Array

**4-Element Conformal Array**

Construct a 4-element array using the ConformalArray System object™. Assume the operating frequency is 900 MHz. Display the array geometry and normal vectors.

```
fc = 900e6;
c = physconst('LightSpeed');
lam = c/fc;
x = [1.0,-.5,0,.8]*lam/2;
y = [-.4,-1,.5,1.5]*lam/2;
z = [-.3,.3,0.4,0]*lam/2;
sIso = phased.CosineAntennaElement(...
    'FrequencyRange',[0,1e9]);
nv = [-140,-140,90,90;80,80,80,80];
sConformArray = phased.ConformalArray('Element',sIso,...
    'ElementPosition',[x;y;z],...
    'ElementNormal',nv);
pos = getElementPosition(sConformArray)
```

pos = *3×4*

```
    0.1666   -0.0833        0    0.1332
   -0.0666   -0.1666   0.0833    0.2498
   -0.0500    0.0500   0.0666        0
```

```
normvec = getElementNormal(sConformArray)
```

```
normvec = 2×4

  -140  -140    90    90
    80    80    80    80
```

viewArray(sConformArray,'ShowIndex','All','ShowNormal',true)



**Array Geometry**

Array Span:
X axis = 249.827 mm
Y axis = 416.378 mm
Z axis = 116.586 mm
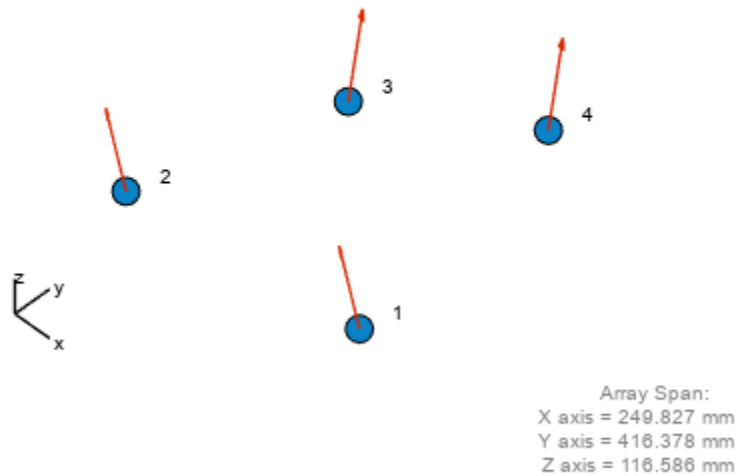
## Converting Between Global and Local Coordinate Systems

In many array processing applications, it is necessary to convert between global and local coordinates. Two utility functions, global2localcoord and local2globalcoord, perform these conversions.

## Convert Local Spherical Coordinates to Global Rectangular Coordinates

Determine the position of a target in rectangular coordinates in the global coordinate system. First, specify the local spherical coordinates of a target with respect to a URA. The center of the URA defines the origin of the local coordinate system. The target location in local spherical coordinates is 30° azimuth, 45° elevation, and 1000 m range. To convert to global rectangular coordinates, specify the position of the local coordinate system origin in global coordinates. The origin of the local coordinate system is (1000,500,100) m from the global origin.

Convert the coordinates of the point to global rectangular coordinates. To convert from local spherical coordinates to global rectangular coordinates, use the `'sr'` option in the call to the `local2globalcoord` function.

```
gCoord = local2globalcoord([30; 45; 1000],'sr',[1000; 500; 100])
```

```
gCoord = 3×1
10³ ×

    1.6124
    0.8536
    0.8071
```

The target is located at (1612,854,807) m in the global coordinate system.

## Convert Global Rectangular Coordinates to Local Spherical Coordinates

Determine the position of a target in local spherical coordinates centered at the phase center of a URA array. The center of the URA defines the origin of the local coordinate system and has the global rectangular coordinates *(5000,3000,50)*. The local coordinate axes of the URA are *(0,1,0)*, *(1,0,0)*, and *(0,0,-1)*. Specify the global rectangular coordinates of the target at *(1000,500,10)*.

Convert the coordinates of the target to local spherical rectangular coordinates. To convert from global rectangular coordinates to local spherical coordinates, use the `'rs'` option in the call to the `global2localcoord` function.

```
lCoord = global2localcoord([5000; 3000; 50],'rs',[1000; 500; 100], ...
    [0 1 0; 1 0 0; 0 0 -1])
```

```
lCoord = 3×1
10³ ×

    0.0580
    0.0006
    4.7173
```

The output has the form (az,el,rng). The target is located in local spherical coordinates at 58° azimuth, 0.6° elevation and 4717 m.

# Global and Local Coordinate Systems Radar Example

This example shows how several different coordinate systems come into play when modeling a typical radar scenario. The scenario considered here is a bistatic radar system consisting of a transmitting radar array, a target, and a receiving radar array. The transmitting radar antenna emits radar signals that propagate to the target, reflect off the target, and then propagate to the receiving radar.

Choose a signal frequency of 1 GHz.

```
fc = 1e9;
c = physconst('LightSpeed');
lam = c/fc;
```

**Create All Radar System Components**

First, set up the transmitting radar array. The transmitting array is a 5-by-5 uniform rectangular array (URA) composed of isotropic antenna elements. The array is stationary and is located at the position *(50,50,50)* meters in the global coordinate system. Although you position arrays in the global system, array element positions are always defined in the array local coordinate system. The transmitted signal strength in any direction is a function of the transmitting angle in the local array coordinate system. Specify the orientation of the array. Without any orientation, local array axes are aligned with the global coordinate system. Choose an array orientation so that the array normal vector points approximately in the direction of the target. Do this by rotating the array 90° around the *z*-axis. Then, rotate the array slightly by 2° around the *y*-axis and 1° around the *z*-axis again.

```
antenna = phased.IsotropicAntennaElement('BackBaffled',false);
txarray = phased.URA('Element',antenna','Size',[5,5],'ElementSpacing',0.4*lam*[1,1]);
txradarAx = rotz(1)*roty(2)*rotz(90);
txplatform = phased.Platform('InitialPosition',[50;50;50],...
    'Velocity',[0;0;0],'InitialOrientationAxes',txradarAx,...
    'OrientationAxesOutputPort',true);
radiator = phased.Radiator('Sensor',txarray,'PropagationSpeed',c,...
    'WeightsInputPort',true,'OperatingFrequency',fc);
steervec = phased.SteeringVector('SensorArray',txarray,'PropagationSpeed',c,...
    'IncludeElementResponse',true);
```

Next, position a target approximately 5 km from the transmitter along the global coordinate system *y*-axis and moving in the *x*-direction. Typically, you specify radar cross-section values as functions of the incident and reflected ray angles with respect to the local target axes. Choose any target orientation with respect to the global coordinate system.

Simulate a non-fluctuating target, but allow the RCS to change at each call to the `target` method. Set up a simple inline function, `rcsval`, to compute fictitious but reasonable values for RCS at different ray angles.

```
tgtAx = rotz(10)*roty(15)*rotx(20);
tgtplatform = phased.Platform('InitialPosition',[100; 10000; 100],...
    'MotionModel','Acceleration','InitialVelocity',[-50;0;0],'Acceleration',[.015;.015;0],...
    'InitialOrientationAxes',tgtAx,'OrientationAxesOutputPort',true);
target = phased.RadarTarget('OperatingFrequency',fc,...
    'Model','Nonfluctuating','MeanRCSSource','Input port');
rcsval = @(az1,el1,az2,el2) 2*abs(cosd((az1+az2)/2 - 90)*cosd((el1+el2)/2));
```

Finally, set up the receiving radar array. The receiving array is also a 5-by-5 URA composed of isotropic antenna elements. The array is stationary and is located 150 meters in the *z*-direction from

the transmitting array. The received signal strength in any direction is a function of the incident angle of the signal in the local array coordinate system. Specify an orientation of the array. Choose an orientation so that this array also points approximately in the *y*-direction towards the target but not quite aligned with the first array. Do this by rotating the array 92° around the *z*-axis and then 5° around the *x*-axis.

```
rxradarAx = rotx(5)*rotz(92);
rxradarAx = rotx(-.2)*rotz(92);

rxplatform = phased.Platform('InitialPosition',[50;50;200],...
    'Velocity',[0;0;0],'InitialOrientationAxes',rxradarAx,...
    'OrientationAxesOutputPort',true);
rxarray = phased.URA('Element',antenna','Size',[5,5],'ElementSpacing',0.4*lam*[1,1]);
```

In summary, four different coordinate systems are needed to describe the radar scenario. These are

**1**    The global coordinate system.

**2**    A local radar coordinate system defined by the transmitting radar axes.

**3**    A local coordinate system defined by the target axes.

**4**    A second local radar coordinate system defined by the receiving radar axes.

The figure here illustrates the four coordinate systems. It is not to scale and does not accurately represent the scenario in the example code.
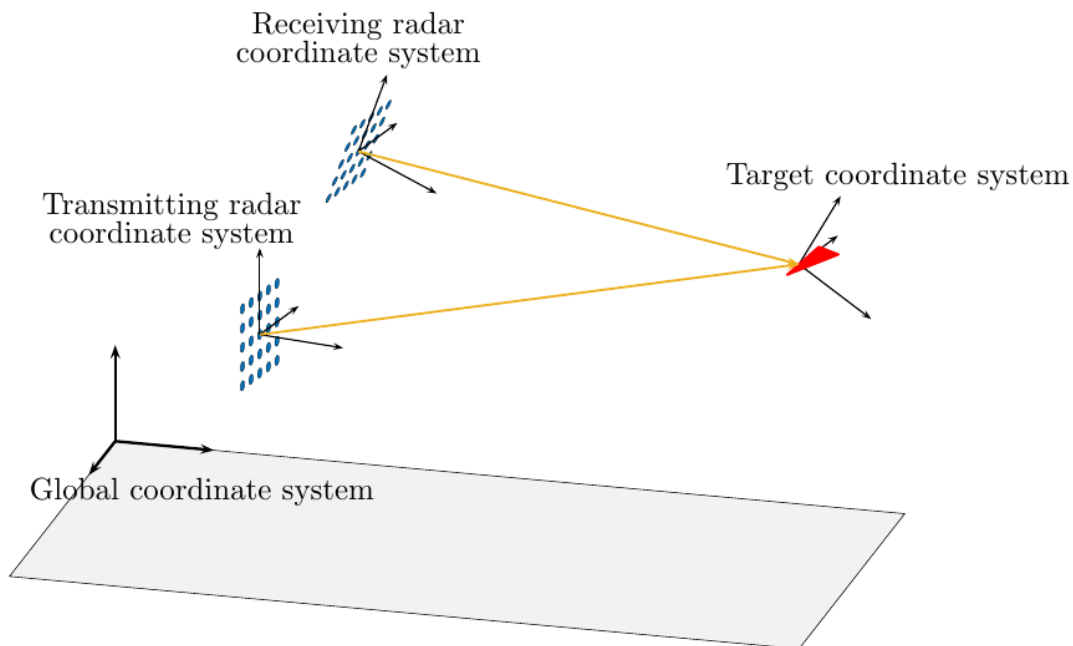
**Specify Transmitted Waveform and Transmitter Amplification**

Use a linear FM waveform as the transmitted signal. Assume a sampling frequency of 1 MHz, a pulse repetition frequency of 5 kHz, and a pulse length of 100 microseconds. Set the transmitter peak output power to 1000 W and the gain to 40.0.

```
tau = 100e-6;
prf = 5000;
fs = 1e6;
waveform = phased.LinearFMWaveform('PulseWidth',tau,...
    'OutputFormat','Pulses','NumPulses',1,'PRF',prf,'SampleRate',fs);
transmitter = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Create a matched filter from the transmitted waveform.

```
filter = phased.MatchedFilter('Coefficients',getMatchedFilter(waveform));
```

**Specify Propagation Channels**

Use free-space models for the propagation of the signal from the transmitting radar to the target and back to the receiving radar.

```
channel1 = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false);
channel2 = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false);
```

**Specify Phaseshift Beamformer**

Create a phase-shift beamformer. Point the mainlobe of the beamformer in a specific direction with respect to the local receiver coordinate system. This direction is chosen to be one through which the target passes at some time in its motion. This choice lets us demonstrate how the beamformer response changes as the target passes through the mainlobe.

```
rxangsteer = [22.2244;-5.0615];
rxangsteer = [10;-.07];
beamformer = phased.PhaseShiftBeamformer('SensorArray',rxarray,...
    'DirectionSource','Property','Direction',rxangsteer,...
    'PropagationSpeed',c,'OperatingFrequency',fc);
```

**Simulation loop**

Each iteration of the processing loop performs these operations:

1    Updates positions of the radars and target.
2    Generates the LFM waveform.
3    Amplifies the waveform.
4    Radiates the signal from the transmitting antenna array.
5    Propagates the signal to the target.
6    Reflects the signal from the target.
7    Propagates the signal from the target to the receiving antenna array.
8    Collects the received signal at the receiving antenna.
9    Beamforms the arriving signal at the receiving antenna.

**10**  Match-filters the beamformed signal and finds its peak value.

Transmit 100 pulses of the waveform. Transmit one pulse every 100 milliseconds.

```
t = 0;
Npulse = 100;
dt = 1;
```

Create storage for later plotting.

```
azes1 = zeros(Npulse,1);
elevs1 = zeros(Npulse,1);
azes2 = zeros(Npulse,1);
elevs2 = zeros(Npulse,1);
rxsig = zeros(Npulse,1);
```

Enter the simulation loop and generate the transmitted waveform.

```
for k = 1:Npulse

    t = t + dt;
    wav = waveform();
```

Update the positions of the radars and targets. All positions and velocities are defined with respect to the global coordinate system. Because the `OrientationAxesOutputPort` property of the target System object™ is set to `true`, you can obtain the instantaneous local target axes, `tgtAx1`, from the `target` method. These axes are needed to compute the target RCS. The array local axes are fixed so you do not need to update them.

```
    [txradarPos,txradarVel] = txplatform(dt);
    [rxradarPos,rxradarVel] = rxplatform(dt);
    [tgtPos,tgtVel,tgtAx1] = tgtplatform(dt);
```

Compute the instantaneous range and direction of the target from the transmitting radar. The strength of the transmitted wave depends upon the array gain pattern. This pattern is a function of direction angles with respect to the local radar axes. You can compute the direction of the target with respect to the transmitter local axes using the `rangeangle` function with an optional argument that specifies the local radar axes, `txradarAx`. (Without this additional argument, `rangeangle` returns the azimuth and elevation angles with respect to the global coordinate system).

```
    [~,tgtang_tlcs] = rangeangle(tgtPos,txradarPos,txradarAx);
```

An alternative way to compute the direction angles is to first compute them in the global coordinate system and then convert them using the `global2localcoord` function.

Create the transmitted waveform. The transmitted waveform is an amplified version of the generated waveform.

```
    txwaveform = transmitter(wav);
```

Radiate the signal in the instantaneous target direction. Recall that the radiator is not steered in this direction but in an angle defined by the steering vector, `txangsteer`. The steering angle is chosen because the target passes through this direction during its motion. A plot will let us see the improvement in the response as the target moves into the main lobe of the radar.

```
    txangsteer = [23.1203;-0.5357];
    txangsteer = [10;-.07];
```

```
sv1 = steervec(fc,txangsteer);
wavrad = radiator(txwaveform,tgtang_tlcs,conj(sv1));
```

Propagate the signal from the transmitting radar to the target. Propagation coordinates are in the global coordinate system.

```
wavprop1 = channel1(wavrad,txradarPos,tgtPos,txradarVel,tgtVel);
```

Reflect the waveform from target back to the receiving radar array. Use the simple angle-dependent RCS model defined previously. Inputs to the rcs-model are azimuth and elevation of the incoming and reflected rays with respect to the local target coordinate system.

```
[~,txang_tgtlcs] = rangeangle(txradarPos,tgtPos,tgtAx1);
[~,rxang_tgtlcs] = rangeangle(rxradarPos,tgtPos,tgtAx1);
rcs = rcsval(txang_tgtlcs(1),txang_tgtlcs(2),rxang_tgtlcs(1),rxang_tgtlcs(2));
wavreflect = target(wavprop1,rcs);
ns = size(wavreflect,1);
tm = [0:ns-1]/fs*1e6;
```

Propagate the signal from the target to the receiving radar. As before, all coordinates for signal propagation are expressed in the global coordinate system.

```
wavprop2 = channel2(wavreflect,tgtPos,rxradarPos,tgtVel,rxradarVel);
```

Compute the response of the receiving antenna array in the direction from which the radiation is coming. First, use the `rangeangle` function to compute the direction of the target with respect to the receiving array local axes, by specifying the receiver local coordinate system, `rxradarAx`.

```
[tgtrange_rlcs,tgtang_rlcs] = rangeangle(tgtPos,rxradarPos,rxradarAx);
```

Store the ranges and direction angles for later plotting.

```
azes1(k) = tgtang_tlcs(1);
elevs1(k) = tgtang_tlcs(2);
azes2(k) = tgtang_rlcs(1);
elevs2(k) = tgtang_rlcs(2);
```

Simulate an incoming plane wave at each element from the current direction of the target calculated in the receiver local coordinate system.

```
wavcoll = collectPlaneWave(rxarray,wavprop2,tgtang_rlcs,fc);
```

Beamform the arriving wave. In this scenario, the receiver beamformer points in the direction, `rxangsteer`, specified by the `Direction` property of the `phased.PhaseShiftBeamformer` System object. When the target actually lies in that direction, the response of the array maximized.

```
wavbf = beamformer(wavcoll);
```

Perform match filtering of the beamformed received wave and then find and store the maximum value of each pulse for display. This value will be plotted after the simulation loop ends.

```
y = filter(wavbf);
rxsig(k) = max(abs(y));
```

```
end
```

Plot the target track in azimuth and elevation with respect to the transmitter local coordinates. The red circle denotes the direction toward which the transmitter array points.

```
plot(azes1,elevs1,'.b')
grid
xlabel('Azimuth (degrees)')
ylabel('Elevation (degrees)')
title('Target Track in Transmitter Local Coordinates')
hold on
plot(txangsteer(1),txangsteer(2),'or')
hold off
```
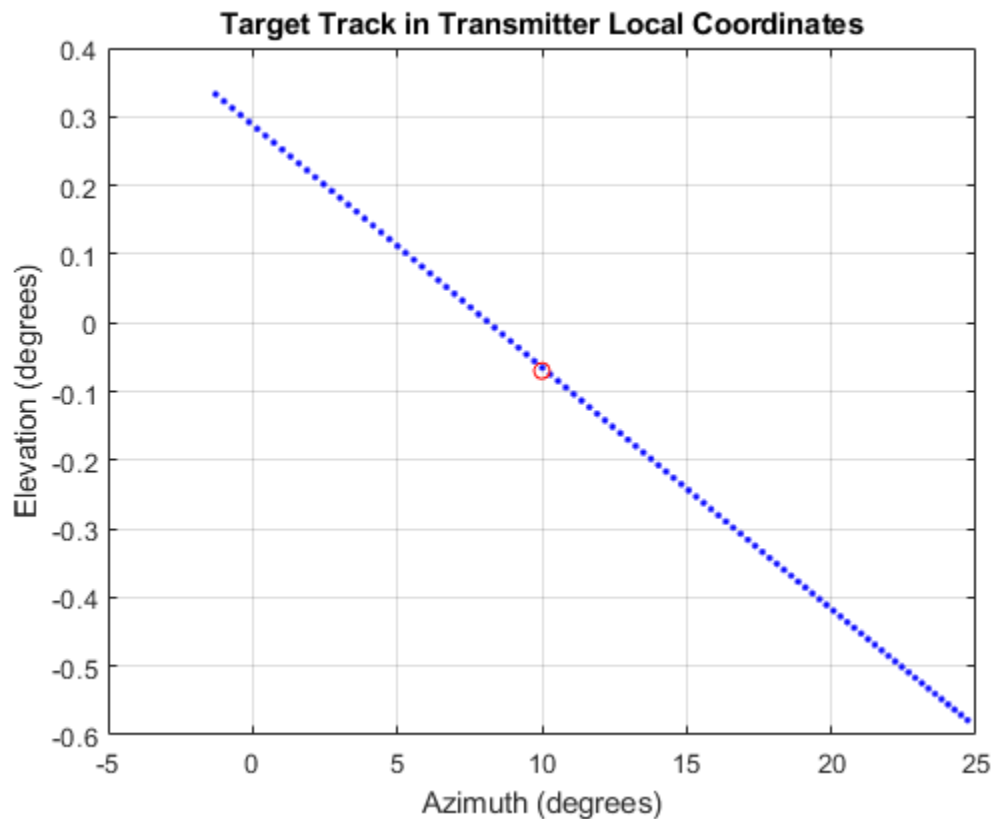


Plot the target track in azimuth and elevation with respect to the receiver local coordinates. The red circle denotes the direction toward which the beamformer points.

```
plot(azes2,elevs2,'.b')
axis([-5.0,25.0,-5.0,5])
grid
xlabel('Azimuth (degrees)')
ylabel('Elevation (degrees)')
title('Target Track in Receiver Local Coordinates')
hold on
plot(rxangsteer(1),rxangsteer(2),'or')
hold off
```
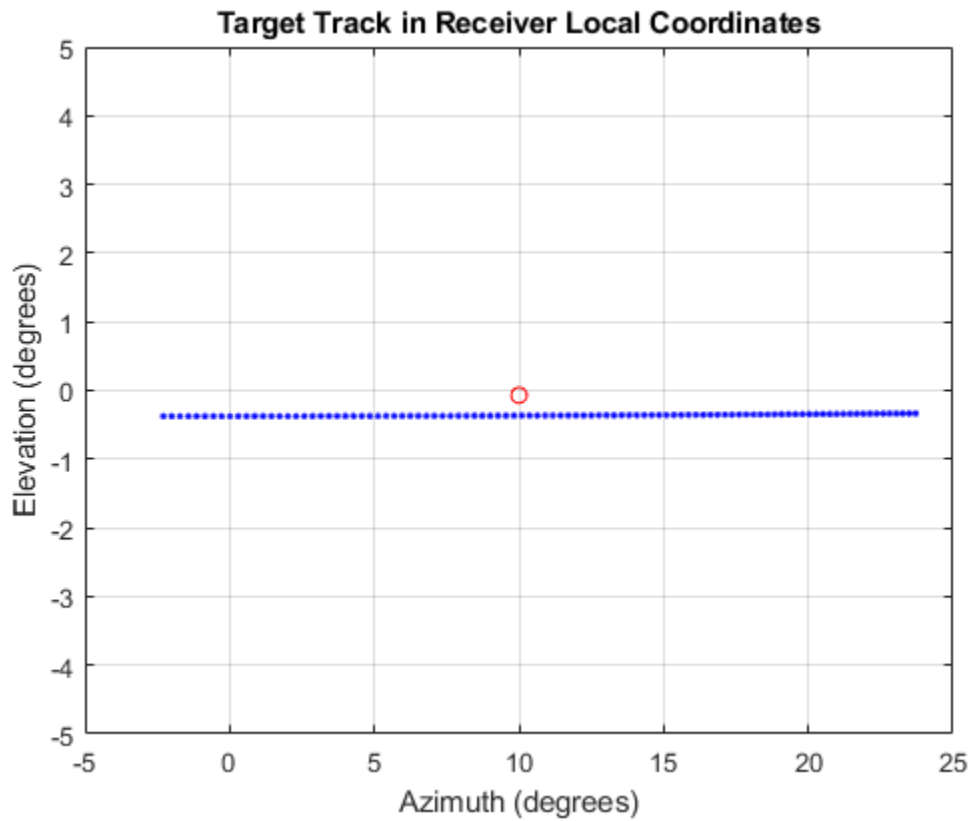
Plot the returned signal amplitude vs azimuth in the receiver local coordinates. The value of the amplitude depends on several factors.

```
plot(azes2,rxsig,'.')
grid
xlabel('Azimuth (degrees)')
ylabel('Amplitude')
title('Amplitude vs Azimuth in Receiver Local Coordinates')
```
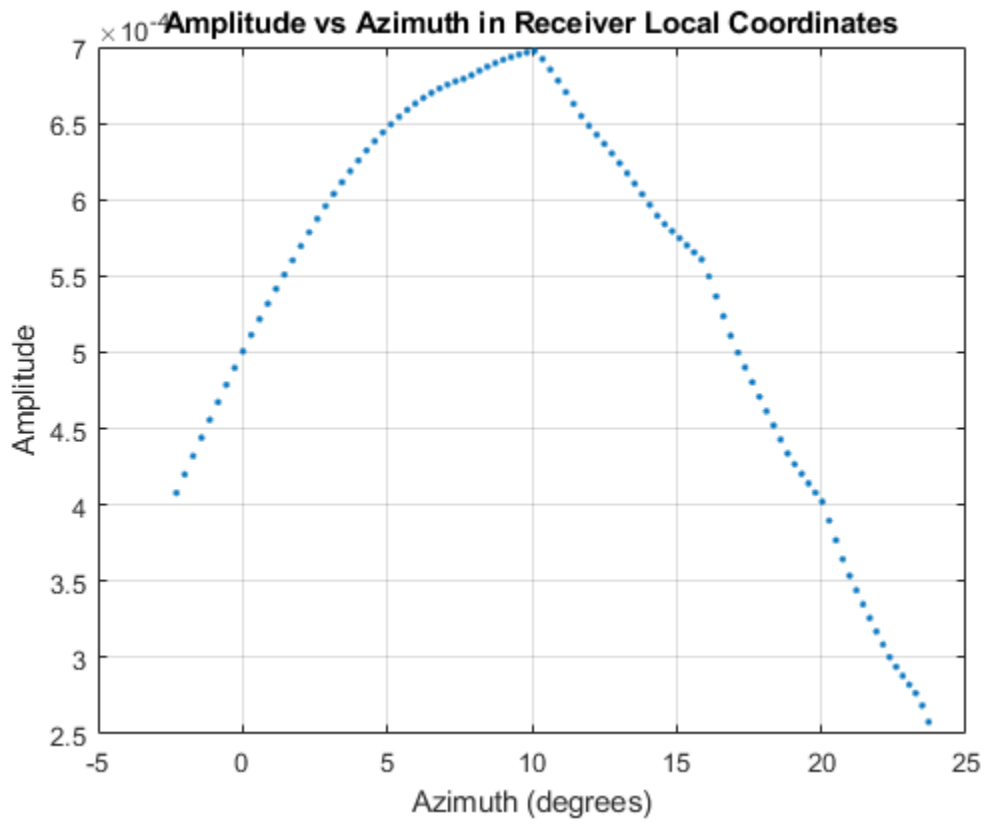
# Motion Modeling in Phased Array Systems

| **In this section...** |
| --- |
| "Support for Motion Modeling" on page 10-39 |
| "Platform Motion with Constant Velocity" on page 10-40 |
| "Platform Motion with Changing Velocity" on page 10-40 |
| "Track Range and Angle Changes Between Platforms" on page 10-41 |

## Support for Motion Modeling

A critical component in phased array system applications is the ability to model motion in space. Such modeling includes the motion of arrays, targets, and sources of interference. For convenience, you can ignore the distinction between these objects and collectively model the motion of a platform.

Extended bodies can undergo both translational and rotational motion in space. Phased Array System Toolbox software supports modeling of translational motion.

Modeling translational platform motion requires the specification of a position and velocity vector. Specification of a position vector implies a coordinate system. In the Phased Array System Toolbox, platform position and velocity are specified in a "Global Coordinate System" on page 10-17. You can think of the platform position as the displacement vector from the global origin or as the coordinates of a point with respect to the global origin.

Let $r_0$ denote the position vector at time 0 and $v$ denote the velocity vector. The position vector of a platform as a function of time, $r(t)$, is:

$$r(t) = r_0 + vt$$

The following figure depicts the vector interpretation of translational motion.
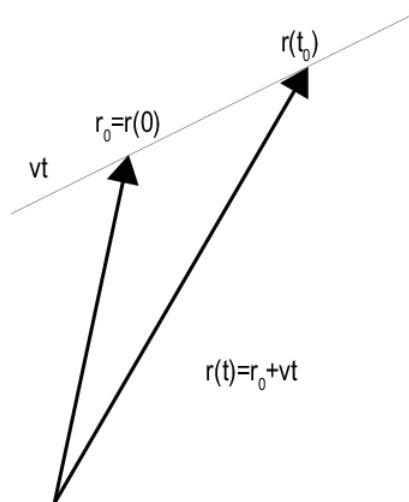


When the platform represents a sensor element or array, it is important to know the orientation of the element or array *local coordinate axes*. For example, the orientation of the local coordinate axes is necessary to extract angle information from incident waveforms. See "Global and Local Coordinate

Systems" on page 10-17 for a description of global and local coordinate systems in the software. Finally, for platforms with varying velocity, you must be able to update the velocity vector over time.

You can model platform position, velocity, and local axes orientation with the `phased.Platform` object.

## Platform Motion with Constant Velocity

Beginning with a simple example, model the motion of a platform over ten time steps. To determine the time step, assume that you have a pulse transmitter with a pulse repetition frequency (PRF) of 1 kilohertz. Accordingly, the time interval between each pulse is 1 millisecond. Set the time step equal to pulse repetition interval.

```
PRF = 1e3;
Tstep = 1/PRF;
Nsteps = 10;
```

Next, construct a platform object specifying the platform initial position and velocity. Assume that the initial position of the platform is 100 meters from the origin at *(60,80,0)*. Assume the speed is approximately 30 meters per second (m/s) with the constant velocity vector given by *(15,25.98,0)* .

```
platform = phased.Platform('InitialPosition',[60;80;0],'Velocity',[15;25.98;0]);
```

The orientation of the local coordinate axes of the platform is the value of the `InitialOrientationAxes` property. You can view the value of this property by entering `hplat.InitialOrientationAxes` at the MATLAB™ command prompt. Because the `InitialOrientationAxes` property is not specified in the construction of the `phased.Platform` System object™, the property is assigned its default value of `[1 0 0;0 1 0;0 0 1]`. To simulate the translational motion of the platform, call the `platform` object.

```
initialPos = platform.InitialPosition;
for k = 1:Nsteps
    pos = platform(Tstep);
end
finalPos = pos + platform.Velocity*Tstep;
distTravel = norm(finalPos - initialPos)
```

distTravel = 0.3000

The object returns the current position of the platform and then updates the platform position based on the time step and velocity. The first time you call the object, the output is the position of the platform at *t = 0*.

Recall that the platform is moving with a constant velocity of approximately 30 m/s. The total time elapsed is 0.01 seconds. Calling the object returns the current position of the platform and then updates that position. Accordingly, you expect the final position to differ from the initial position by 0.30 meters. Confirm this difference by examining the value of `distTravel`.

## Platform Motion with Changing Velocity

Most platforms in phased array applications do not move with constant velocity. If the time interval described by the number of time steps is small with respect to the platform speed, you can often

approximate the velocity as constant. However, there are situations where you must update the platform velocity over time. You can do so with the `phased.Platform` System Object™ because the `Velocity` property is *tunable*.

This example models a target initially at rest. The initial velocity vector is *(0,0,0)* . Assume the time step is 1 millisecond. After 500 milliseconds, the platform begins to move with a speed of approximately 10 m/s. The velocity vector is *(7.07,7.07,0)*. The platform continues at this velocity for an additional 500 milliseconds.

```
Tstep = 1e-3;
Nsteps = 1/Tstep;
platform = phased.Platform('InitialPosition',[100;100;0]);
for k = 1:Nsteps/2
    [tgtpos,tgtvel] = platform(Tstep);
end
platform.Velocity = [7.07; 7.07; 0];
for k = Nsteps/2+1:Nsteps
    [tgtpos,tgtvel] = platform(Tstep);
end
```

## Track Range and Angle Changes Between Platforms

This example uses the `phased.Platform` System object™ to model the change in range between a stationary radar and a moving target. The radar is located at *(1000,1000,0)* and has a velocity of *(0,0,0)* . The target has an initial position of *(5000,8000,0* ) and moves with a constant velocity of *(−30,−45,0)*. The pulse repetition frequency (PRF) is 1 kHz. Assume that the radar emits ten pulses.

```
PRF = 1e3;
Tstep = 1/PRF;
radar = phased.Platform('InitialPosition',[1000;1000;0]);
target = phased.Platform('InitialPosition',[5000;8000;0],...
    'Velocity',[-30;-45;0]);
```

Calculate initial target range and angle.

```
[initRng, initAng] = rangeangle(target.InitialPosition,radar.InitialPosition);
```

Calculate relative radial speed.

```
v = radialspeed(target.InitialPosition,target.Velocity,...
    radar.InitialPosition);
```

Simulate target motion.

```
Npulses = 10;
for num = 1:Npulses
    tgtpos = target(Tstep);
end
```

Compute last target position.

```
tgtpos = tgtpos + target.Velocity*Tstep;
```

Calculate final target range and angle.

```
[finalRng,finalAng] = rangeangle(tgtpos,radar.InitialPosition);
deltaRng = finalRng - initRng
```

```
deltaRng = -0.5396
```

The constant velocity of the target is approximately 54 m/s. The total time elapsed is 0.01 seconds. The range between the target and the radar should decrease by approximately 54 centimeters. Compare the initial range of the target, `initRng`, to the final range, `finalRng`, to confirm that this decrease occurs.

## See Also

## Related Examples

- "Introduction to Space-Time Adaptive Processing" on page 18-247

# Model Motion of Circling Airplane

Start with an airplane moving along a circular track with a radius of 10 km at a horizontal speed of 100 m/s and descending at a rate of 1 m/sec. To create circular motion, specify a radially-inward acceleration and constrain the acceleration to lie in the horizontal plane. The acceleration of a body moving in a circle is $\frac{v^2}{r}$. The rate of descent is constant. Set the initial orientation axes matrix of the platform to the identity matrix.

**Set up the initial conditions**

```
alt = 10000;
radcirc = 10000; % 10 km
phi = 60;
initPos = [cosd(phi)*radcirc;sind(phi)*radcirc;alt];
vs = 100.0;
vx = vs*sind(phi);
vy = -vs*cosd(phi);
vz = -1;
initVel = [vx,vy,vz]';
airplane = phased.Platform('MotionModel','Acceleration', ...
    'AccelerationSource','Input port','InitialPosition',initPos, ...
    'InitialVelocity',initVel,'OrientationAxesOutputPort',true, ...
    'InitialOrientationAxes',eye(3));
accelmag = vs^2/radcirc;
initPos1 = [cosd(phi)*radcirc;sind(phi)*radcirc;0];
unitvec = initPos1/radcirc;
accel = -accelmag*unitvec;
```

**Compute the trajectory**

Compute the trajectory for 20000 integration steps at T = 0.1 s intervals

```
N = 20000;
tstep = .10;
posmat = zeros(3,N);
for n = 1:N
    [pos,vel,oax] = airplane(tstep,accel);
    velcirc2 = vel(1)^2 + vel(2)^2;
    vmag = sqrt(velcirc2);
    pos1 = [pos(1),pos(2),0]';
    radcirc = sqrt(pos1'*pos1);
    unitvec = pos1/radcirc;
    accelmag = velcirc2/radcirc;
    accel = -accelmag*unitvec;
    posmat(:,n) = pos;
end
```

Display the final orientation of the local coordinate system.

```
disp(oax)
```

```
    0.1271    0.9919    0.0001
   -0.9919    0.1271    0.0003
    0.0003   -0.0001    1.0000
```

**Plot the trajectory**

```
plot3(posmat(1,:)/1000,posmat(2,:)/1000,posmat(3,:)/1000,'b.')
xlabel('X (km)')
ylabel('Y (km)')
zlabel('Z (km)')
axis equal
grid
```
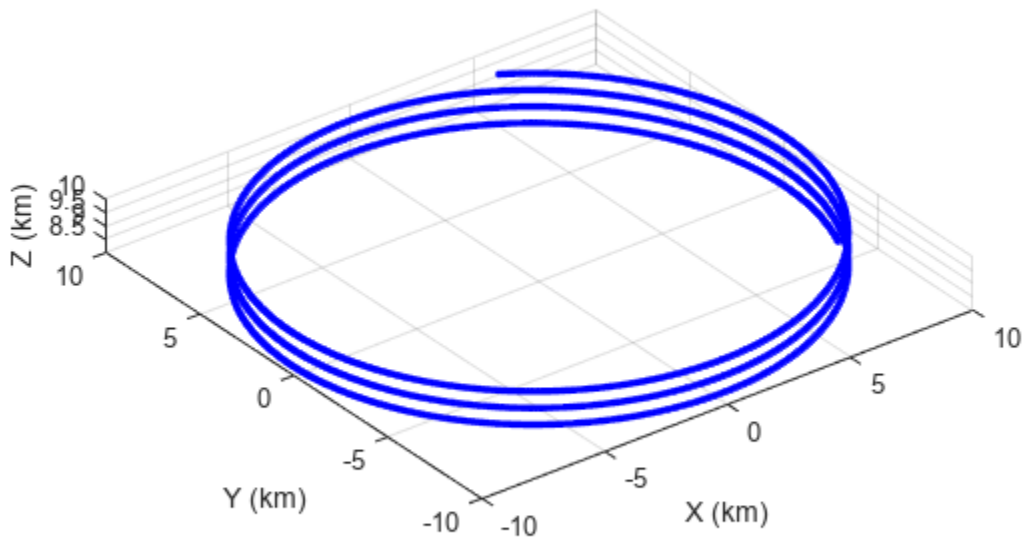
# Visualize Multiplatform Scenario

This example shows how to create and display a multiplatform scenario containing a ground-based stationary radar, a turning airplane, a constant-velocity airplane, and a moving ground vehicle. The turning airplane follows a parabolic flight path while descending at a rate of 20 m/s.

Specify the scenario refresh rate at 0.5 Hz. For 150 steps, the time duration of the scenario is 300 s.

```
updateRate = 0.5;
N = 150;
```

Set up the turning airplane using the `Acceleration` model of the `phased.Platform` System object™. Specify the initial position of the airplane by range and azimuth from the ground-based radar and its elevation. The airplane is 10 km from the radar at 60° azimuth and has an altitude of 6 km. The airplane is accelerating at 10 m/s$^2$ in the negative *x*-direction.

```
airplane1range = 10.0e3;
airplane1Azimuth = 60.0;
airplane1alt = 6.0e3;
airplane1Pos0 = [cosd(airplane1Azimuth)*airplane1range;...
    sind(airplane1Azimuth)*airplane1range;airplane1alt];
airplane1Vel0 = [400.0;-100.0;-20];
airplane1Accel = [-10.0;0.0;0.0];
airplane1platform = phased.Platform('MotionModel','Acceleration',...
    'AccelerationSource','Input port','InitialPosition',airplane1Pos0,...
    'InitialVelocity',airplane1Vel0,'OrientationAxesOutputPort',true,...
    'InitialOrientationAxes',eye(3));
```

Set up the stationary ground radar at the origin of the global coordinate system. To simulate a rotating radar, change the ground radar beam steering angle in the processing loop.

```
groundRadarPos = [0,0,0]';
groundRadarVel = [0,0,0]';
groundradarplatform = phased.Platform('MotionModel','Velocity',...
    'InitialPosition',groundRadarPos,'Velocity',groundRadarVel,...
    'InitialOrientationAxes',eye(3));
```

Set up the ground vehicle to move at a constant velocity.

```
groundVehiclePos = [5e3,2e3,0]';
groundVehicleVel = [50,50,0]';
groundvehicleplatform = phased.Platform('MotionModel','Velocity',...
    'InitialPosition',groundVehiclePos,'Velocity',groundVehicleVel,...
    'InitialOrientationAxes',eye(3));
```

Set up the second airplane to also move at constant velocity.

```
airplane2Pos = [8.5e3,1e3,6000]';
airplane2Vel = [-300,100,20]';
airplane2platform = phased.Platform('MotionModel','Velocity',...
    'InitialPosition',airplane2Pos,'Velocity',airplane2Vel,...
    'InitialOrientationAxes',eye(3));
```

Set up the scenario viewer. Specify the radar as having a beam range of 8 km, a vertical beam width of 30°, and a horizontal beam width of 2°. Annotate the tracks with position, speed, altitude, and range.

```
BeamSteering = [0;50];
viewer = phased.ScenarioViewer('BeamRange',8.0e3,'BeamWidth',[2;30],'UpdateRate',updateRate,...
    'PlatformNames',{'Ground Radar','Turning Airplane','Vehicle','Airplane 2'},'ShowPosition',tru
    'ShowSpeed',true,'ShowAltitude',true,'ShowLegend',true,'ShowRange',true,...
    'Title','Multiplatform Scenario','BeamSteering',BeamSteering);
```

Step through the display processing loop, updating radar and target positions. Rotate the ground-based radar steering angle by four degrees at each step.

```
for n = 1:N
    [groundRadarPos,groundRadarVel] = groundradarplatform(updateRate);
    [airplane1Pos,airplane1Vel,airplane1Axes] = airplane1platform(updateRate,airplane1Accel);
    [vehiclePos,vehicleVel] = groundvehicleplatform(updateRate);
    [airplane2Pos,airplane2Vel] = airplane2platform(updateRate);
    viewer(groundRadarPos,groundRadarVel,[airplane1Pos,vehiclePos,airplane2Pos],...
        [airplane1Vel,vehicleVel,airplane2Vel]);
    BeamSteering = viewer.BeamSteering(1);
    BeamSteering = mod(BeamSteering + 4,360.0);
    if BeamSteering > 180.0
        BeamSteering = BeamSteering - 360.0;
    end
    viewer.BeamSteering(1) = BeamSteering;
    pause(0.2);
end
```

# Doppler Shift and Pulse-Doppler Processing

| In this section... |
| --- |
| "Support for Pulse-Doppler Processing" on page 10-48 |
| "Convert Speed to Doppler Shift" on page 10-48 |
| "Convert Doppler Shift to Speed" on page 10-48 |
| "Pulse-Doppler Processing of Slow-Time Data" on page 10-49 |
| "Range and Speed Using Pulse-Doppler Processing" on page 10-49 |

## Support for Pulse-Doppler Processing

Relative motion between a signal source and a receiver produces shifts in the frequency of the received waveform. Measuring this *Doppler* shift provides an estimate of the relative radial velocity of a moving target.

For a narrowband signal propagating at the speed of light, the one-way Doppler shift in hertz is:

$$\Delta f = \pm \frac{v}{\lambda}$$

where $v$ is the relative radial speed of the target with respect to the transmitter. For a target approaching the receiver, the Doppler shift is positive. For a target receding from the transmitter, the Doppler shift is negative.

You can use `speed2dop` to convert the relative radial speed to the Doppler shift in hertz. You can use `dop2speed` to determine the radial speed of a target relative to a receiver based on the observed Doppler shift.

## Convert Speed to Doppler Shift

Assume a target approaches a stationary receiver with a radial speed of 23.0 m/s. The target reflects a narrowband electromagnetic wave with a frequency of 1 GHz. Estimate the one-way Doppler shift.

```
freq = 1e9;
v = 23.0;
lambda = physconst('LightSpeed')/freq;
dopplershift = speed2dop(v,lambda)
```

```
dopplershift = 76.7197
```

The one-way Doppler shift is approximately 76.72 Hz. Because the target approaches the receiver, the Doppler shift is positive.

## Convert Doppler Shift to Speed

Assume you observe a Doppler shift of 400.0 Hz for a waveform with a frequency of 9 GHz. Determine the radial velocity of the target.

```
freq = 9e9;
df = 400.0;
lambda = physconst('LightSpeed')/freq;
speed = dop2speed(df,lambda)
```

```
speed = 13.3241
```

The target speed is approximately 13.32 m/s.

## Pulse-Doppler Processing of Slow-Time Data

A common technique for estimating the radial velocity of a moving target is pulse-Doppler processing. In pulse-Doppler processing, you take the discrete Fourier transform (DFT) of the slow-time data from a range bin containing a target. If the pulse repetition frequency is sufficiently high with respect to the speed of the target, the target is located in the same range bin for a number of pulses. Accordingly, the slow-time data corresponding to that range bin contain information about the Doppler shift induced by the moving target, which you can use to estimate the target's radial velocity.

The slow-time data are sampled at the pulse repetition frequency (PRF) and therefore the DFT of the slow-time data for a given range bin yields an estimate of the Doppler spectrum from [-PRF/2, PRF/2] Hz. Because the slow-time data are complex-valued, the DFT magnitudes are not necessarily an even function of the Doppler frequency. This removes the ambiguity between a Doppler shift corresponding to an approaching (positive Doppler shift), or receding (negative Doppler shift) target. The resolution in the Doppler domain is PRF/N where N is the number of slow-time samples. You can pad the spectral estimate of the slow-time data with zeros to interpolate the DFT frequency grid and improve peak detection, but this does not improve the Doppler resolution.

The typical workflow in pulse-Doppler processing involves:

- Detecting a target in the range dimension (fast-time samples). This gives the range bin to analyze in the slow-time dimension.
- Computing the DFT of the slow-time samples corresponding to the specified range bin. Identify significant peaks in the magnitude spectrum and convert the corresponding Doppler frequencies to speeds.

## Range and Speed Using Pulse-Doppler Processing

This example illustrates pulse-Doppler processing using Phased Array System Toolbox™. Assume that you have a stationary monostatic radar located at the global origin, *(0,0,0)*. The radar consists of a single isotropic antenna element. There is a target with a nonfluctuating radar cross section (RCS) of 1 square meter initially located at *(1000,1000,0)* m and moving at a constant velocity of *(-100,-100,0)* m/s. The antenna operates at a frequency of 1 GHz and illuminates the target with 10 rectangular pulses at a PRF of 10 kHz.

Define the System objects needed for this example and set their properties. Seed the random number generator for the `phased.ReceiverPreamp` System object™ to produce repeatable results.

```
waveform = phased.RectangularWaveform('SampleRate',5e6,...
    'PulseWidth',6e-7,'OutputFormat','Pulses',...
    'NumPulses',1,'PRF',1e4);
target = phased.RadarTarget('Model','Nonfluctuating',...
    'MeanRCS',1,'OperatingFrequency',1e9);
```

```
targetpos = phased.Platform('InitialPosition',[1000; 1000; 0],...
    'Velocity',[-100; -100; 0]);
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[5e8 5e9]);
transmitter = phased.Transmitter('PeakPower',5e3,'Gain',20,...
'InUseOutputPort',true);
transpos = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0]);
radiator = phased.Radiator('OperatingFrequency',1e9,'Sensor',antenna);
collector = phased.Collector('OperatingFrequency',1e9,'Sensor',antenna);
channel = phased.FreeSpace('SampleRate',waveform.SampleRate,...
    'OperatingFrequency',1e9,'TwoWayPropagation',false);
receiver = phased.ReceiverPreamp('Gain',0,'LossFactor',0,...
    'SampleRate',5e6,'NoiseFigure',5,...
    'EnableInputPort',true,'SeedSource','Property','Seed',1e3);
```

This loop transmits ten successive rectangular pulses toward the target, reflects the pulses off the target, collects the reflected pulses at the receiver, and updates the target position with the specified constant velocity.

```
NumPulses = 10;
sig = waveform(); % get waveform
transpos = transpos.InitialPosition; % get transmitter position
rxsig = zeros(length(sig),NumPulses);
% transmit and receive ten pulses
for n = 1:NumPulses
    % update target position
    [tgtpos,tgtvel] = targetpos(1/waveform.PRF);
    [tgtrng,tgtang] = rangeangle(tgtpos,transpos);
    tpos(n) = tgtrng;
    [txsig,txstatus] = transmitter(sig); % transmit waveform
    txsig = radiator(txsig,tgtang); % radiate waveform toward target
    txsig = channel(txsig,transpos,tgtpos,[0;0;0],tgtvel); % propagate waveform to target
    txsig = target(txsig); % reflect the signal
    % propagate waveform from the target to the transmiter
    txsig = channel(txsig,tgtpos,transpos,tgtvel,[0;0;0]);
    txsig = collector(txsig,tgtang); % collect signal
    rxsig(:,n) = receiver(txsig,~txstatus); % receive the signal
end
```

The matrix `rxsig` contains the echo data in a 500-by-10 matrix where the row dimension contains the fast-time samples and the column dimension contains the slow-time samples. In other words, each row in the matrix contains the slow-time samples from a specific range bin.

Construct a linearly-spaced grid corresponding to the range bins from the fast-time samples. The range bins extend from 0 meters to the maximum unambiguous range.

```
prf = waveform.PRF;
fs = waveform.SampleRate;
fasttime = unigrid(0,1/fs,1/prf,'[)');
rangebins = (physconst('LightSpeed')*fasttime)/2;
```

Next, detect the range bins which contain targets. In this simple scenario, no matched filtering or time-varying gain compensation is utilized.

In this example, set the false-alarm probability to $10^{-9}$. Use noncoherent integration of the ten rectangular pulses and determine the corresponding threshold for detection in white Gaussian noise.

Because this scenario contains only one target, take the largest peak above the threshold. Display the estimated target range.

```
probfa = 1e-9;
NoiseBandwidth = 5e6/2;
npower = noisepow(NoiseBandwidth,...
    receiver.NoiseFigure,receiver.ReferenceTemperature);
thresh = npwgnthresh(probfa,NumPulses,'noncoherent');
thresh = sqrt(npower*db2pow(thresh));
[pks,range_detect] = findpeaks(pulsint(rxsig,'noncoherent'),...
    'MinPeakHeight',thresh,'SortStr','descend');
range_estimate = rangebins(range_detect(1));
```

Extract the slow-time samples corresponding to the range bin containing the detected target. Compute the power spectral density estimate of the slow-time samples using `periodogram` function and find the peak frequency. Convert the peak Doppler frequency to speed using the `dop2speed` function. A positive Doppler shift indicates that the target is approaching the transmitter. A negative Doppler shift indicates that the target is moving away from the transmitter.

```
ts = rxsig(range_detect(1),:).';
[Pxx,F] = periodogram(ts,[],256,prf,'centered');
plot(F,10*log10(Pxx))
grid
xlabel('Frequency (kHz)')
ylabel('Power (dB)')
title('Periodogram Spectrum Estimate')
```
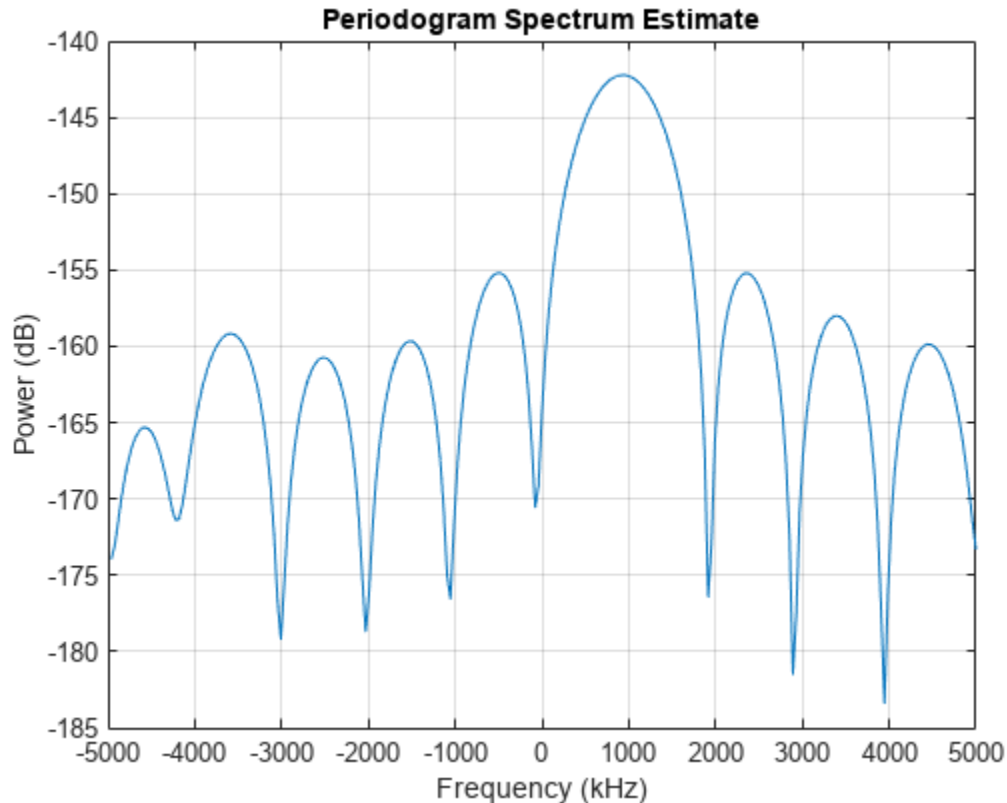
```
[Y,I] = max(Pxx);
lambda = physconst('LightSpeed')/1e9;
tgtspeed = dop2speed(F(I)/2,lambda);
fprintf('Estimated range of the target is %4.2f meters.\n',...
    range_estimate)
```

Estimated range of the target is 1439.00 meters.

```
fprintf('Estimated target speed is %3.1f m/sec.\n',tgtspeed)
```

Estimated target speed is 140.5 m/sec.

```
if F(I)>0
    fprintf('The target is approaching the radar.\n')
else
    fprintf('The target is moving away from the radar.\n')
end
```

The target is approaching the radar.

The true radial speed of the target is detected within the Doppler resolution and the range of the target is detected within the range resolution of the radar.

## See Also

### Related Examples

- "Doppler Estimation" on page 18-308
- "Electronic Scanning Using a Uniform Rectangular Array" on page 18-405

# Using Polarization

# Polarized Fields

| In this section... |
|---|

## Introduction to Polarization

You can use the Phased Array System Toolbox software to simulate radar systems that transmit, propagate, reflect, and receive polarized electromagnetic fields. By including this capability, the toolbox can realistically model the interaction of radar waves with targets and the environment.

It is a basic property of plane waves in free-space that the directions of the electric and magnetic field vectors are orthogonal to their direction of propagation. The direction of propagation of an electromagnetic wave is determined by the Poynting vector

$$\mathbf{S} = \mathbf{E} \times \mathbf{H}$$

In this equation, $\mathbf{E}$ represents the electric field and $\mathbf{H}$ represents the magnetic field. The quantity, $\mathbf{S}$, represents the magnitude and direction of the wave's energy flux. Maxwell's equations, when applied to plane waves, produce the result that the electric and magnetic fields are related by

$$\mathbf{E} = -Z\mathbf{s} \times \mathbf{H}$$
$$\mathbf{H} = \frac{1}{Z}\mathbf{s} \times \mathbf{E}$$

The vector $\mathbf{s}$, the unit vector in the $\mathbf{S}$ direction, represents the direction of propagation of the wave. The quantity, $Z$, is the wave impedance and is a function of the electric permittivity and the magnetic permeability of medium in which the wave travels.

After manipulating the two equations, you can see that the electric and magnetic fields are orthogonal to the direction of propagation

$$\mathbf{E} \cdot \mathbf{s} = \mathbf{H} \cdot \mathbf{s} = 0.$$

This last result proves that there are really only two independent components of the electric field, labeled $E_x$ and $E_y$. Similarly, the magnetic field can be expressed in terms of two independent components. Because of the orthogonality of the fields, the electric field can be represented in terms of two unit vectors orthogonal to the direction of propagation.

$$\mathbf{E} = E_x\widehat{\mathbf{e}}_x + E_y\widehat{\mathbf{e}}_y$$

The unit vectors together with the unit vector in direction of propagation

$$\left\{ \widehat{\mathbf{e}}_x, \widehat{\mathbf{e}}_y, \mathbf{s} \right\}.$$

form a right-handed orthonormal triad. Later, these vectors and the coordinates they define will be related to the coordinates of a specific radar system. Because the electric and magnetic fields are determined by each other, only the properties of the electric field need be consider.

For a radar system, the electric and magnetic field are actually spherical waves, rather than plane waves. However, in practice, these fields are usually measured in the far field region or radiation zone of the radar source and are approximately plane waves. In the far field, the waves are called quasi-plane waves. A point lies in the far field if its distance, $R$, from the source satisfies $R \gg D^2/\lambda$ where $D$ is a typical dimension of the source, whether it is a single antenna or an array of antennas.

Polarization applies to purely sinusoidal signals. The most general expression for a sinusoidal plane-wave has the form

$$\mathbf{E} = E_{x0}\cos(\omega t - \mathbf{k} \cdot \mathbf{x} + \phi_x)\widehat{\mathbf{e}}_x + E_{y0}\cos(\omega t - \mathbf{k} \cdot \mathbf{x} + \phi_y)\widehat{\mathbf{e}}_y = E_x\widehat{\mathbf{e}}_x + E_y\widehat{\mathbf{e}}_y$$

The quantities $E_{x0}$ and $E_{y0}$ are the real-valued, non-negative, amplitudes of the components of the electric field and $\phi_x$ and $\phi_y$ are field's phases. This expression is the most general one used for a *polarized* wave. An electromagnetic wave is polarized if the ratio of the amplitudes of its components and phase difference between it components do not change with time. The definition of polarization can be broadened to include narrowband signals, for which the bandwidth is small compared to the center or carrier frequency of the signal. The amplitude ratio and phases difference vary slowly with time when compared to the period of the wave and may be thought of as constant over many oscillations.

You can usually suppress the spatial dependence of the field and write the electric field vector as

$$\mathbf{E} = E_{x0}\cos(\omega t + \phi_x)\widehat{\mathbf{e}}_x + E_{y0}\cos(\omega t + \phi_y)\widehat{\mathbf{e}}_y = E_x\widehat{\mathbf{e}}_x + E_y\widehat{\mathbf{e}}_y$$

## Linear and Circular Polarization

The preceding equation for a polarized plane wave shows that the tip of the two-dimensional electric field vector moves along a path which lies in a plane orthogonal to field's direction of propagation. The shape of the path depends upon the magnitudes and phases of the components. For example, if $\phi_x = \phi_y$, you can remove the time dependence and write

$$E_y = \frac{E_{y0}}{E_{x0}}E_x$$

This equation represents a straight line through the origin with positive slope. Conversely, suppose $\phi_x = \phi_y + \pi$. Then, the tip of the electric field vector follows a straight line through the origin with negative slope

$$E_y = -\frac{E_{y0}}{E_{x0}}E_x$$

These two polarization cases are named linear polarized because the field always oscillates along a straight line in the orthogonal plane. If $E_{x0} = 0$, the field is vertically polarized, and if $E_{y0} = 0$ the field is horizontally polarized.

A different case occurs when the amplitudes are the same, $E_x = E_y$, but the phases differ by $\pm\pi/2$

$E_x = E_0 \cos(\omega t + \phi)$

$E_y = E_0 \cos(\omega t + \phi \pm \pi/2) = \mp E_0 \sin(\omega t + \phi)$

By squaring both sides, you can show that the tip of the electric field vector obeys the equation of a circle

$$E_x^2 + E_y^2 = E_0^2$$

While this equation gives the path the vector takes, it does not tell you in what direction the electric field vector travels around the circle. Does it rotate clockwise or counterclockwise? The rotation direction depends upon the sign of *π/2* in the phase. You can see this dependency by examining the motion of the tip of the vector field. Assume the common phase angle, *ϕ = 0*. This assumption is permissible because the common phase only determines starting position of the vector and does not change the shape of its path. First, look at the +*π/2* case for a wave travelling along the *s*-direction (out of the page). At *t=0*, the vector points along the *x*-axis. One quarter period later, the vector points along the negative *y*-axis. After another quarter period, it points along the negative *x*-axis.
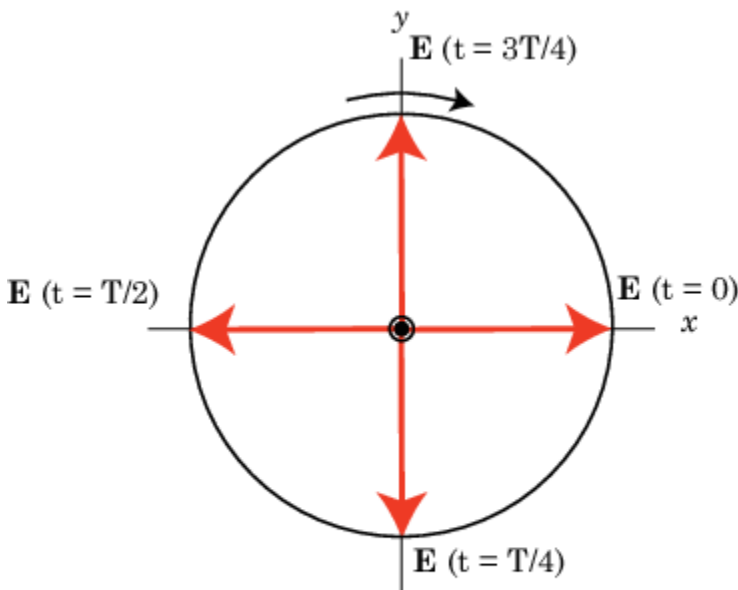


Left hand circular polarization. The direction of
the electric vector at 0, 1/4, 1/2, and 3/4 periods, T.
The z-axis points out of the page.

MATLAB uses the IEEE convention to assign the names *right-handed* or *left-handed* polarization to the direction of rotation of the electric vector, rather than *clockwise* or *counterclockwise*. When using this convention, left or right handedness is determined by pointing your left or right thumb along the direction of propagation of the wave. Then, align the curve of your fingers to the direction of rotation of the field at a given point in space. If the rotation follows the curve of your left hand, then the wave is left-handed polarized. If the rotation follows the curve of your right hand, then the wave is right-handed polarized. In the preceding scenario, the field is left-handed circularly polarized (LHCP). The phase difference –*π/2* corresponds to right-handed circularly polarized wave (RHCP). The following figure provides a three-dimensional view of what a LHCP electromagnetic wave looks like as it moves in the *s*-direction.

When the terms *clockwise* or *counterclockwise* are used they depend upon how you look at the wave. If you look along the direction of propagation, then the clockwise direction corresponds to right-handed polarization and counterclockwise corresponds to left-handed polarization. If you look toward where the wave is coming from, then clockwise corresponds to left-handed polarization and counterclockwise corresponds to right-handed polarization.
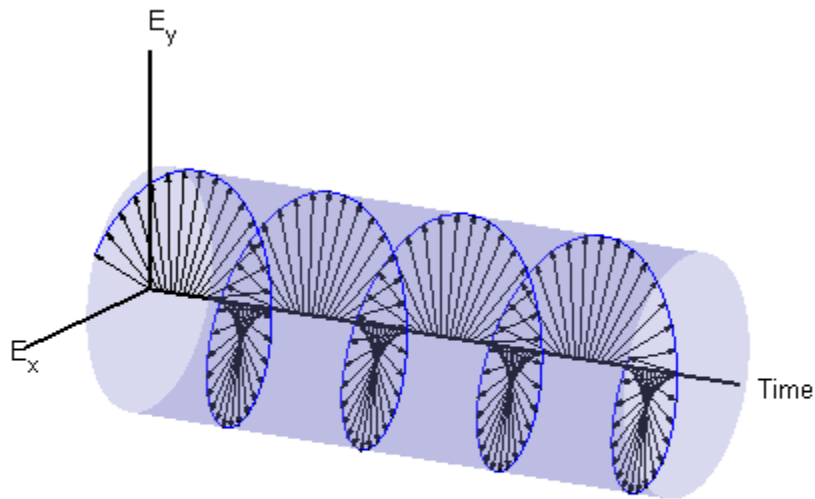


**Left-Handed Circular Polarization**

The figure below shows the appearance of linear and circularly polarized fields as they move towards you along the *s*-direction.
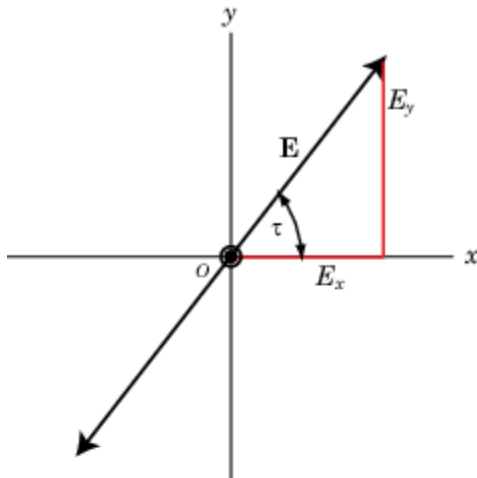
Linear polarization with positive slope
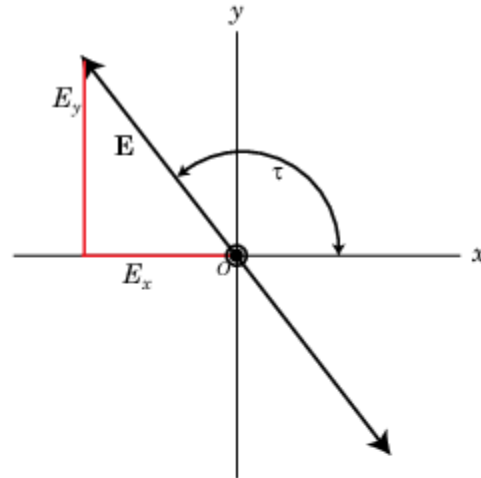
Linear polarization with negative slope
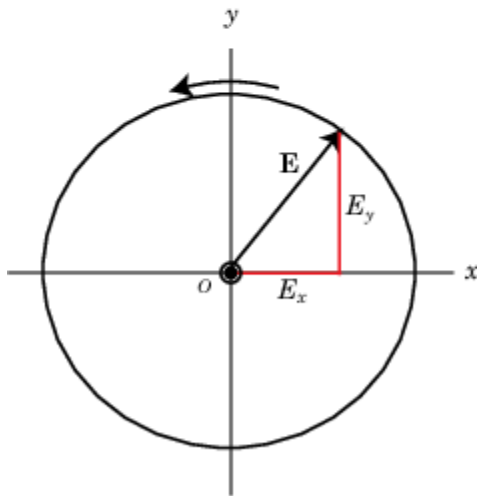
Right hand circular polarization
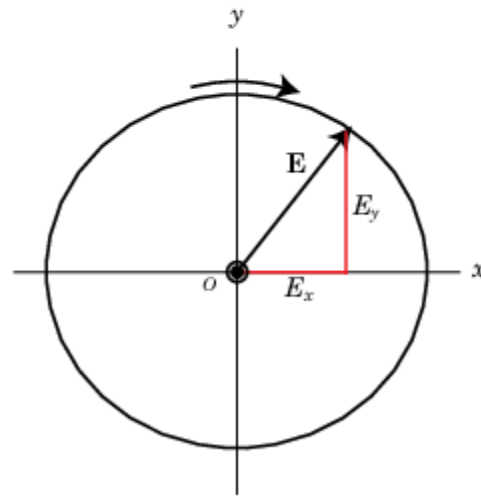
Left hand circular polarization

**Linear and Circular Polarization**

## Elliptic Polarization

Besides the linear and circular states of polarization, a third type of polarization is *elliptic polarization*. Elliptic polarization includes linear and circular polarization as special cases.

As with linear or circular polarization, you can remove the time dependence to obtain the locus of points that the tip of the electric field vector travels

$$\left(\frac{E_x}{E_{x0}}\right)^2 + \left(\frac{E_y}{E_{y0}}\right)^2 - 2\left(\frac{E_x}{E_{x0}}\right)\left(\frac{E_y}{E_{y0}}\right)\cos\phi = \sin^2\phi$$

In this case, $\varphi = \varphi_y - \varphi_x$. This equation represents a tilted two-dimensional ellipse. Its size and shape are determined by the component amplitudes and phase difference. The presence of the cross term

indicates that the ellipse is tilted. The equation does not, just as in the circularly polarized case, provide any information about the rotation direction. For example, the following figure shows the instantaneous state of the electric field but does not indicate the direction in which the field is rotating.

The size and shape of a two-dimensional ellipse can be defined by three parameters. These parameters are the lengths of its two axes, the semi-major axis, $a$, and semi-minor axis, $b$, and a tilt angle, $\tau$. The following figure illustrates the three parameters of a tilted ellipse. You can derive them from the two electric field amplitudes and phase difference.
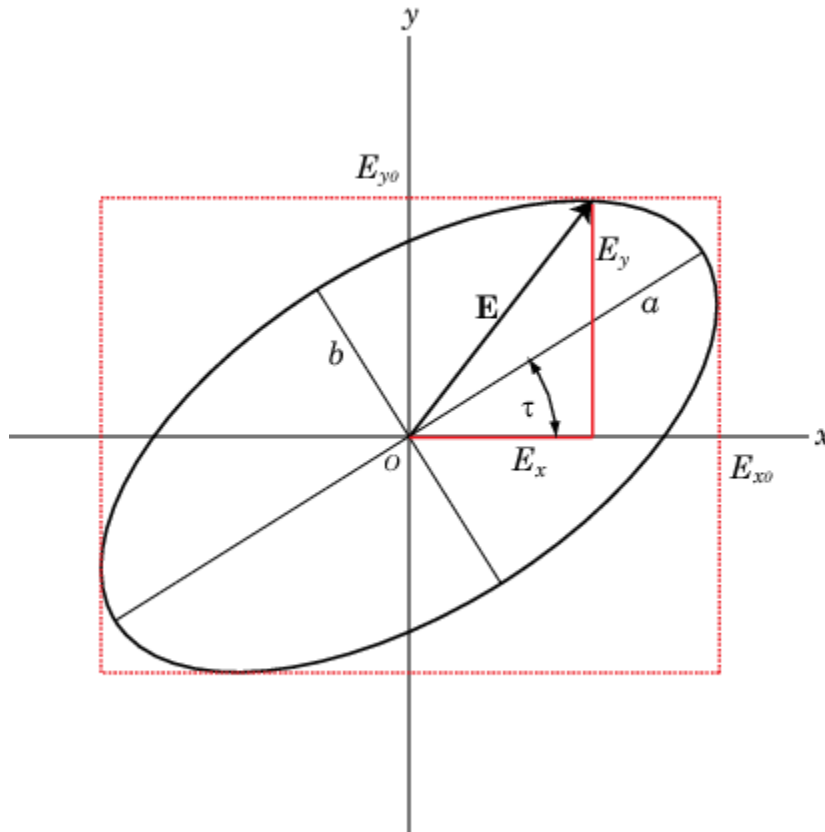


**Polarization Ellipse**

Polarization can best be understood in terms of complex signals. The complex representation of a polarized wave has the form

$$\mathbf{E} = E_{x0}e^{i\phi_x}e^{i\omega t}\widehat{\mathbf{e}}_x + E_{y0}e^{i\phi_y}e^{i\omega t}\widehat{\mathbf{e}}_y = \left(E_{x0}e^{i\phi_x}\widehat{\mathbf{e}}_x + E_{y0}e^{i\phi_y}\widehat{\mathbf{e}}_y\right)e^{i\omega t}$$

Define the complex polarization ratio as the ratio of the complex amplitudes

$$\rho = \frac{E_{y0}}{E_{x0}}e^{i(\phi_y - \phi_x)} = \frac{E_{y0}}{E_{x0}}e^{i\phi}$$

where $\phi = \phi_y - \phi_x$.

It is useful to introduce the polarization vector. For the complex polarized electric field above, the polarization vector, $\boldsymbol{P}$, is obtained by normalizing the electric field

$$\mathbf{P} = \frac{E_{x0}}{E_m}\widehat{\mathbf{e}}_x + \frac{E_{y0}}{E_m}e^{i(\phi_y - \phi_x)}\widehat{\mathbf{e}}_y = \frac{E_{x0}}{E_m}\widehat{\mathbf{e}}_x + \frac{E_{y0}}{E_m}e^{i\phi}\widehat{\mathbf{e}}_y$$

where $E_m{}^2 = E_{x0}{}^2 + E_{y0}{}^2$ is the magnitude of the wave.

The overall size of the polarization ellipse is not important because that can vary as the wave travels through space, especially through geometric attenuation. What is important is the shape of the ellipse. Thus, the significant ellipse parameters are the ratio of its axis dimensions, *a/b*, called the axial ratio, and the tilt angle, $\tau$. Both of these quantities can be determined from the ratio of the component amplitudes and the phase difference, or, equivalently, from the polarization ratio. Another quantity, equivalent to the axial ratio, is the ellipticity angle, $\varepsilon$.

In Phased Array System Toolbox, use the `polratio` function to convert the complex amplitudes `fv=[Eh;Ev]` to the polarization ratio.

```
p = polratio(fv)
```

In radar systems, it is common to use the subscripts, *H* and *V*, denoting the horizontal and vertical components, instead of *x* and *y*. Phased Array System Toolbox functions follow this convention for writing polarized fields where $E_H$ corresponds to $E_X$ and $E_V$ corresponds to $E_V$.

**Tilt Angle**

The tilt angle is defined as the positive (counterclockwise) rotation angle from the *x*-axis to the semi-major axis of the ellipse. Because of the symmetry properties of the ellipse, the tilt angle, $\tau$, need only be defined in the range $-\pi/2 \le \tau \le \pi/2$. You can find the tilt angle by determining the rotated coordinate system in which the semi-major and semi-minor axes align with the rotated coordinate axes. Then, the ellipse equation has no cross-terms. The solution takes the form

$$\tan 2\tau = \frac{2E_{H0}E_{V0}}{E_{H0}^2 - E_{V0}^2}\cos\phi$$

where $\varphi = \varphi_V - \varphi_H$. Notice that you can rewrite this equation strictly in terms of the amplitude ratio and the phase difference.

**Axial Ratio and Ellipticity Angle**

After solving for the tilt angle, you can determine the semi-major and semi-minor axis lengths. Conceptually, you rotate the ellipse clockwise by the tilt angle and measure the lengths of the intersections of the ellipse with the *H*- and *V*-axes. The point of intersection with the larger value is the semi-major axis, *a*, and the one with the smaller value is the semi-minor axis, *b*.

The *axial ratio* is defined as $AR = a/b$ and, by construction, is always greater than or equal to one. The *ellipticity angle* is defined by

$$\tan\varepsilon = \mp\frac{b}{a}$$

and always lies in the range $-\pi/4 \le \varepsilon \le \pi/4$.

If you define the *auxiliary angle*, $\alpha$, by

$$\tan\alpha = \frac{E_{V0}}{E_{H0}}$$

then, the *ellipticity angle* is given by

$$\sin 2\varepsilon = \sin 2\alpha \sin\phi$$

Both the axial ratio and ellipticity angle are defined from the amplitude ratio and phase difference and are independent of the overall magnitude of the field.

### Rotation Sense

For elliptic polarization, just as with circular polarization, you need another parameter to completely describe the ellipse. This parameter must provide the rotation sense or the direction that the tip of the electric (or magnetic vector) moves in time. The rate of change of the angle that the field vector makes with the *x*-axis is proportion to *–sin φ* where *φ* is the phase difference. If *sin φ* is positive, the rate of change is negative, indicating that the field has left-handed polarization. If *sin φ* is negative, the rate of change is positive or right-handed polarization.

The function `polellip` lets you find the values of the parameters of the polarization ellipse from either the field component vector `fv=[Eh;Ev]` or the polarization ratio, `p`.

```
fv=[Ey;Ex];
[tau,epsilon,ar,rs] = polellip(fv);
p = polratio(fv);
[tau,epsilon,ar,rs] = polellip(p);
```

The variables `tau`, `epsilon`, `ar` and `rs` represent the tilt angle, ellipticity angle, axial ratio and rotation sense, respectively. Both syntaxes give the same result.

### Polarization Value Summary

This table summaries several different common polarization states and the values of the amplitudes, phases, and polarization ratio that produce them:

| Polarization | Amplitudes | Phases | Polarization Ratio |
|---|---|---|---|
| Linear positive slope | Any non-negative real values for $E_H$, $E_V$. | $\varphi_V = \varphi_H$ | Any non-negative real number |
| Linear negative slope | Any non-negative real values for $E_H$, $E_V$ | $\varphi_V = \varphi_H + \pi$ | Any negative real number |
| Right-Handed Circular | $E_H = E_V$ | $\varphi_V = \varphi_H - \pi/2$ | $-i$ |
| Left-Handed Circular | $E_H = E_V$ | $\varphi_V = \varphi_H + \pi/2$ | $i$ |
| Right-Handed Elliptical | Any non-negative real values for $E_H$, $E_V$ | $\sin(\varphi_V - \varphi_H) < 0$ | $\sin(\arg \rho) < 0$ |
| Left-Handed Elliptical | Any non-negative real values for $E_H$, $E_V$ | $\sin(\varphi_V - \varphi_H) > 0$ | $\sin(\arg \rho) > 0$ |

## Linear and Circular Polarization Bases

As shown earlier, you can express a polarized electric field as a linear combination of basis vectors along the *H* and *V* directions. For example, the complex electric field vectors for the right-handed circularly polarized (RHCP) wave and the left-handed circularly polarized (LHCP) wave, take the form:

$$\mathbf{E} = \mathrm{Re}[E_0(\mathbf{e}_H \mp i\mathbf{e}_V)e^{i(\omega t + \phi)}]$$

In this equation, the positive sign is for the LHCP field and the negative sign is for the RHCP field. These two special combinations can be given a new name. Define a new basis vector set, called the circular basis set

$$\mathbf{e}_r = \frac{1}{\sqrt{2}}(\mathbf{e}_H - i\mathbf{e}_V)$$

$$\mathbf{e}_l = \frac{1}{\sqrt{2}}(\mathbf{e}_H + i\mathbf{e}_V)$$

You can express any polarized field in terms of the circular basis set instead of the linear basis set. Conversely, you can also write the linear polarization basis in terms of the circular polarization basis

$$\mathbf{e}_H = \frac{1}{\sqrt{2}}(\mathbf{e}_r + \mathbf{e}_l)$$

$$\mathbf{e}_V = \frac{1}{\sqrt{2i}}(\mathbf{e}_r - \mathbf{e}_l)$$

Any general elliptic field can be written as a combination of circular basis vectors

$$\mathbf{E} = E_l\mathbf{e}_l + E_r\mathbf{e}_r$$

### Jones Vector

The polarized field is orthogonal to the wave's direction of propagation. Thus, the field can be completely specified by the two complex components of the electric field vector in the plane of polarization. The formulation of a polarized wave in terms of two-component vectors is called the Jones vector formulation. The Jones vector formulation can be expressed in either a linear basis or a circular basis or any basis. This table shows the representation of common polarizations in a linear basis and circular basis.

| Common Polarizations | Jones Vector in Linear Basis | Jones Vector in Circular Basis |
|---|---|---|
| Vertical | [0;1] | 1/sqrt(2)*[-1;1] |
| Horizontal | [1;0] | 1/sqrt(2)*[1;1] |
| 45° Linear | 1/sqrt(2)*[1;1] | 1/sqrt(2)*[1-1i;1+1i] |
| 135° Linear | 1/sqrt(2)*[1;-1] | 1/sqrt(2)*[1+1i;1-1i] |
| Right Circular | 1/sqrt(2)*[1;-1i] | [0;1] |
| Left Circular | 1/sqrt(2)*[1;1i] | [1;0] |

### Stokes Parameters and the Poincaré Sphere

The polarization ellipse is an instantaneous representation of a polarized wave. However, its parameters, the tilt angle and the ellipticity angle, are often not directly measurable, particularly at very high frequencies such as light frequencies. However, you can determine the polarization from measurable intensities of the polarized field.

The measurable intensities are the Stokes parameters, $S_0$, $S_1$, $S_2$, and $S_3$. The first Stokes parameter, $S_0$, describes the total intensity of the field. The second parameter, $S_1$, describes the preponderance of linear horizontally polarized intensity over linear vertically polarized intensity. The third parameter, $S_2$, describes the preponderance of linearly +45° polarized intensity over linearly 135° polarized intensity. Finally, $S_3$ describes the preponderance of right circularly polarized intensity over left circularly polarized intensity. The Stokes parameters are defined as

$$S_0 = E_{H0}^2 + E_{V0}^2$$

$$S_1 = E_{H0}^2 - E_{V0}^2$$

$$S_2 = 2E_{H0}E_{V0}\cos\phi$$

$$S_3 = 2E_{H0}E_{V0}\sin\phi$$

For completely polarized fields, you can show by time averaging the polarization ellipse equation that

$$S_0^2 = S_1^2 + S_2^2 + S_3^2$$

Thus, there are only three independent Stokes' parameters.

For partially polarized fields, in contrast, the Stokes parameters satisfy the inequality

$$S_0^2 < S_1^2 + S_3^2 + S_3^2$$

The Stokes parameters are related to the tilt and ellipticity angles, $\tau$ and $\varepsilon$

$$S_1 = S_0\cos2\tau\cos2\varepsilon$$

$$S_2 = S_0\sin2\tau\cos2\varepsilon$$

$$S_3 = S_0\sin2\varepsilon$$

and inversely by

$$\tan2\tau = \frac{S_2}{S_1}$$

$$\sin2\varepsilon = \frac{S_3}{S_0}$$

After you measure the Stokes' parameters, the shape of the ellipse is completely determined by the preceding equations.

The two-dimensional Poincaré sphere can help you visualize the state of a polarized wave. Any point on or in the sphere represents a state of polarization determined by the four Stokes parameters, $S_0$, $S_1$, $S_2$, and $S_3$. On the Poincaré sphere, the angle from the $S_1$-$S_2$ plane to a point on the sphere is twice the ellipticity angle, $\varepsilon$. The angle from the $S_1$- axis to the projection of the point into the $S_1$-$S_2$ plane is twice the tilt angle, $\tau$.
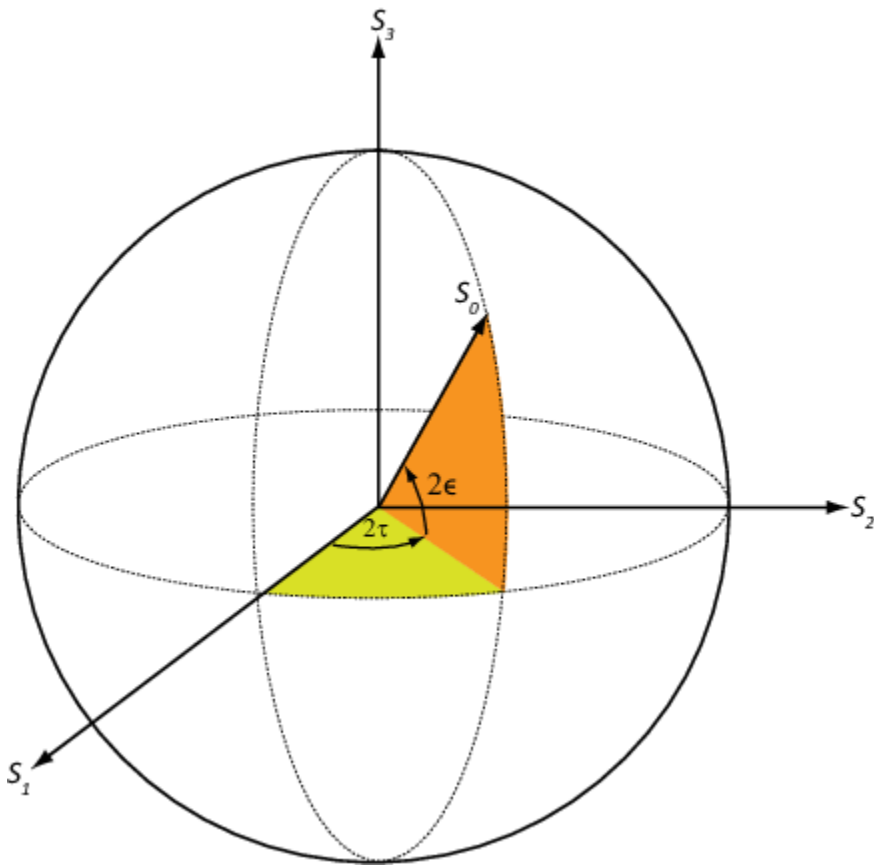
As an example, solve for the Stokes parameters of a RHCP field, `fv=[1,-i]`, using the `stokes` function.

```
S = stokes(fv)

S =

     2
     0
     0
    -2
```

## Sources of Polarized Fields

Antennas couple propagating electromagnetic radiation to electrical currents in wires, electromagnetic fields in waveguides or aperture fields. This coupling is a phenomenon common to both transmitting and receiving antennas. For some transmitting antennas, source currents in a wire produce electromagnetic waves that carrying power in all directions. Sometimes an antenna provides a means for a guided electromagnetic wave on a transmission line to transition to free-space waves such as a waveguide feeding a dish antennas. For receiving antennas, electromagnetic fields can induce currents in wires to generate signals to be then amplified and passed on to a detector.

For transmitting antennas, the shape of the antenna is chosen to enhance the power projected into a given direction. For receiving antennas, you choose the shape of the antenna to enhance the power

received from a particular direction. Often, many transmitting antennas or receiving antennas are formed into an array. Arrays increase the transmitted power for a transmitting system or the sensitivity for a receiving system. They improve directivity over a single antenna.

An antenna can be assigned a polarization. The polarization of a transmitting antenna is the polarization of its radiated wave in the far field. The polarization of a receiving antenna is actually the polarization of a plane wave, from a given direction, resulting in maximum power at the antenna terminals. By the reciprocity theorem, all transmitting antennas can serve as receiving antennas and vice versa.

Each antenna or array has an associated local Cartesian coordinate system *(x,y,z)* as shown in the following figure. See "Global and Local Coordinate Systems" on page 10-17 for more information. The local coordinate system can also be represented by a spherical coordinate system using azimuth, elevation and range coordinates, *az, el, r*, or alternately written, *(φ,θ,r)*, as shown. At each point in the far field, you can create a set of unit spherical basis vectors, $\{\widehat{\mathbf{e}}_H, \widehat{\mathbf{e}}_V, \widehat{\mathbf{r}}\}$. The basis vectors are aligned with the *(φ,θ,r)* directions, respectively. In the far field, the electric field is orthogonal to the unit vector $\widehat{\mathbf{r}}$. The components of a polarized field with respect to this basis, *(E_H,E_V)*, are called the horizontal and vertical components of the polarized field. In radar, it is common to use *(H,V)* instead of *(x,y)* to denote the components of a polarized field. In the far field, the polarized electric field takes the form

$$\mathbf{E} = \mathbf{F}(\phi,\theta)\frac{e^{ikr}}{r} = \left(F_H(\phi,\theta)\widehat{\mathbf{e}}_H + F_V(\phi,\theta)\widehat{\mathbf{e}}_V\right)\frac{e^{ikr}}{r}$$

In this equation, the quantity *F(φ,θ)* is called the vector radiation pattern of the source and contains the angular dependence of the field in the far-field region.



Radar Coordinate System

### Short Dipole Antenna Element

The simplest polarized antenna is the dipole antenna which consist of a split length of wire coupled at the middle to a coaxial cable. The simplest dipole, from a mathematical perspective, is the Hertzian dipole, in which the length of wire is much shorter than a wavelength. A diagram of the short dipole antenna of length *L* appears in the next figure. This antenna is fed by a coaxial feed which splits into two equal length wires of length *L/2*. The current, *I*, moves along the *z*-axis and is assumed to be the same at all points in the wire.

The electric field in the far field has the form

$$E_r = 0$$

$$E_H = 0$$

$$E_V = -\frac{iZ_0IL}{2\lambda}\text{cosel}\,\frac{e^{-ikr}}{r}$$

The next example computes the vertical and horizontal polarization components of the field. The vertical component is a function of elevation angle and is axially symmetric. The horizontal component vanishes everywhere.

The toolbox lets you model a short dipole antenna using the `phased.ShortDipoleAntennaElement` System object.

**Short-Dipole Polarization Components**

Compute the vertical and horizontal polarization components of the field created by a short-dipole antenna pointed along the z-direction. Plot the components as a function of elevation angle from 0° to 360°.

Create the `phased.ShortDipoleAntennaElement` System object™.

```
antenna = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[1,2]*1e9,'AxisDirection','Z');
```

Compute the antenna response. Because the elevation angle argument to `antenna` is restricted to ±90°, compute the responses for 0° azimuth and then for 180° azimuth. Combine the two responses in the plot. The operating frequency of the antenna is 1.5 GHz.

```
el = -90:90;
az = zeros(size(el));
fc = 1.5e9;
resp = antenna(fc,[az;el]);
az = 180.0*ones(size(el));
resp1 = antenna(fc,[az;el]);
```

Overlay the responses in the same figure.

```
figure(1)
subplot(121)
polarplot(el*pi/180.0,abs(resp.V.'),'b')
hold on
polarplot((el+180)*pi/180.0,abs(resp1.V.'),'b')
str = sprintf('%s\n%s','Vertical Polarization','vs Elevation Angle');
title(str)
hold off
subplot(122)
polarplot(el*pi/180.0,abs(resp.H.'),'b')
hold on
polarplot((el+180)*pi/180.0,abs(resp1.H.'),'b')
str = sprintf('%s\n%s','Horizontal Polarization','vs Elevation Angle');
title(str)
hold off
```

The plot shows that the horizontal component vanishes, as expected.

**Crossed Dipole Antenna Element**

You can use a cross-dipole antenna to generate circularly-polarized radiation. The crossed-dipole antenna consists of two identical but orthogonal short-dipole antennas that are phased 90° apart. A diagram of the crossed dipole antenna appears in the following figure. The electric field created by a crossed-dipole antenna constructed from a $y$-directed short dipole and a $z$-directed short dipole has the form

$$E_r = 0$$

$$E_H = -\frac{iZ_0IL}{2\lambda}\cos az\frac{e^{-ikr}}{r}$$

$$E_V = \frac{iZ_0IL}{2\lambda}(\sin el\sin az + i\cos el)\frac{e^{-ikr}}{r}$$

The polarization ratio $E_V/E_H$, when evaluated along the $x$-axis, is just $-i$ which means that the polarization is exactly RHCP along the $x$-axis. It is predominantly RHCP when the observation point is close to the $x$-axis. Moving away from the $x$-axis, the field becomes a mixture of LHCP and RHCP polarizations. Along the $-x$-axis, the field is LHCP polarized. The figure illustrates, for a point near the $x$, that the field is primarily RHCP.



The toolbox lets you model a crossed-dipole antenna using the `phased.CrossedDipoleAntennaElement` System object.

**LHCP and RHCP Polarization Components**

This example plots the right-hand and left-hand circular polarization components of fields generated by a crossed-dipole antenna at 1.5 GHz. You can see how the circular polarization changes from pure RHCP at 0 degrees azimuth angle to pure LHCP at 180 degrees azimuth angle, both at 0 degrees elevation angle.

Create the `phased.CrossedDipoleAntennaElement` object.

```
fc = 1.5e9;
antenna = phased.CrossedDipoleAntennaElement('FrequencyRange',[1,2]*1e9);
```

Compute the left-handed and right-handed circular polarization components from the antenna response.

```
az = [-180:180];
el = zeros(size(az));
resp = antenna(fc,[az;el]);
cfv = pol2circpol([resp.H.';resp.V.']);
clhp = cfv(1,:);
crhp = cfv(2,:);
```

Plot both circular polarization components at 0 degrees elevation.

```
polar(az*pi/180.0,abs(clhp))
hold on
polar(az*pi/180.0,abs(crhp))
title('LHCP and RHCP vs Azimuth Angle')
legend('LHCP','RHCP')
hold off
```

**Arrays Supporting Polarization**

You can create polarized fields from arrays by using polarized antenna elements as a value of the Elements property of an array System object. All Phased Array System Toolbox arrays support polarization.

## Scattering Cross-Section Matrix

After a polarized field is created by an antenna system, the field radiates to the far-field region. When the field propagates into free space, the polarization properties remain unchanged until the field interacts with a material substance which scatters the field into many directions. In such situations, the amplitude and polarization of the scattered wave can differ from the incident wave polarization. The scattered wave polarization may depend upon the direction in which the scattered wave is observed. The exact way that the polarization changes depends upon the properties of the scattering object. The quantity describing the response of an object to the incident field is called the radar scattering cross-section matrix (RSCM), $S$. You can measure the scattering matrix as follows. When a unit amplitude horizontally polarized wave is scattered, both a horizontal and a vertical scattered component are produced. Call these two components $S_{HH}$ and $S_{VH}$. These components are complex numbers containing the amplitude and phase changes from the incident wave. Similarly, when a unit amplitude vertically polarized wave is scattered, the horizontal and vertical scattered component produced are $S_{HV}$ and $S_{VV}$. Because, any incident field can be decomposed into horizontal and vertical components, you can arrange these quantities into a matrix and write the scattered field in terms of the incident field

$$\begin{bmatrix} E_H^{(scat)} \\ E_V^{(scat)} \end{bmatrix} = \sqrt{\frac{4\pi}{\lambda^2}} \begin{bmatrix} S_{HH} & S_{VH} \\ S_{HV} & S_{VV} \end{bmatrix} \begin{bmatrix} E_H^{(inc)} \\ E_V^{(inc)} \end{bmatrix} = \sqrt{\frac{4\pi}{\lambda^2}} [S] \begin{bmatrix} E_H^{(inc)} \\ E_V^{(inc)} \end{bmatrix}$$

In general, the scattering cross-section matrix depends upon the angles that the incident and scattered fields make with the object. When the incident field is scattered back to the transmitting antenna or, backscattered, the scattering matrix is symmetric.

**Polarization Signature**

To understand how the scattered wave depends upon the polarization of the incident wave, you need to examine all possible scattered field polarizations for each incident polarization. Because this amount of data is difficult to visualize, consider two cases:

- For the copolarization case, the scattered polarization has the same polarization as the incident field.

- For the cross-polarization case, the scattered polarization has an orthogonal polarization to the incident field.

You can represent the incident polarizations in terms of the tilt angle-ellipticity angle pair $(\tau, \varepsilon)$. Every unit incident polarization vector can be expressed as

$$\begin{bmatrix} E_H^{(inc)} \\ E_V^{(inc)} \end{bmatrix} = \begin{bmatrix} \cos\tau & -\sin\tau \\ \sin\tau & \cos\tau \end{bmatrix} \begin{bmatrix} \cos\varepsilon \\ j\sin\varepsilon \end{bmatrix}$$

while the orthogonal polarization vector is

$$\begin{bmatrix} E_H^{(inc)\,\perp} \\ E_V^{(inc)\,\perp} \end{bmatrix} = \begin{bmatrix} -\sin\tau & -\cos\tau \\ \cos\tau & -\sin\tau \end{bmatrix} \begin{bmatrix} \cos\varepsilon \\ -j\sin\varepsilon \end{bmatrix}$$

When you have an RSCM matrix, $S$, form the copolarization signature by computing

$$P^{(co)} = \begin{bmatrix} E_H^{(inc)} & E_V^{(inc)} \end{bmatrix} * S \begin{bmatrix} E_H^{(inc)} \\ E_V^{(inc)} \end{bmatrix}$$

where []* denotes complex conjugation. To obtain the cross-polarization signature, compute

$$P^{(cross)} = \begin{bmatrix} E_H^{(inc)\,\perp} & E_V^{(inc)\,\perp} \end{bmatrix} * S \begin{bmatrix} E_H^{(inc)} \\ E_V^{(inc)} \end{bmatrix}$$

You can compute both the copolarization and cross polarization signatures using the `polsignature` function. This function returns the absolute value of the scattered power (normalized by its maximum value). The next example shows how to plot the polarization signatures for the RSCM matrix

$$S = \begin{bmatrix} 2i & \dfrac{1}{2} \\ \dfrac{1}{2} & i \end{bmatrix}$$

for all possible incident polarizations. The range of values of the ellipticity angle and tilt span the entire possible range of polarizations.

**Plot Polarization Signatures**

Plot the copolarization and cross-polarization signatures of the scattering matrix

$$\begin{bmatrix} 2i & 0.5 \\ 0.5 & -i \end{bmatrix}.$$

Specify the scattering matrix. and specify the range of ellipticity angles and orientation (tilt) angles that define the polarization states. These angles cover all possible incident polarization states.

```
rscmat = [1i*2,0.5;0.5,-1i];
el = [-45:45];
tilt = [-90:90];
```

Plot the copolarization signatures for all incident polarizations.

```
polsignature(rscmat,'c',el,tilt)
```



Co-Pol Response
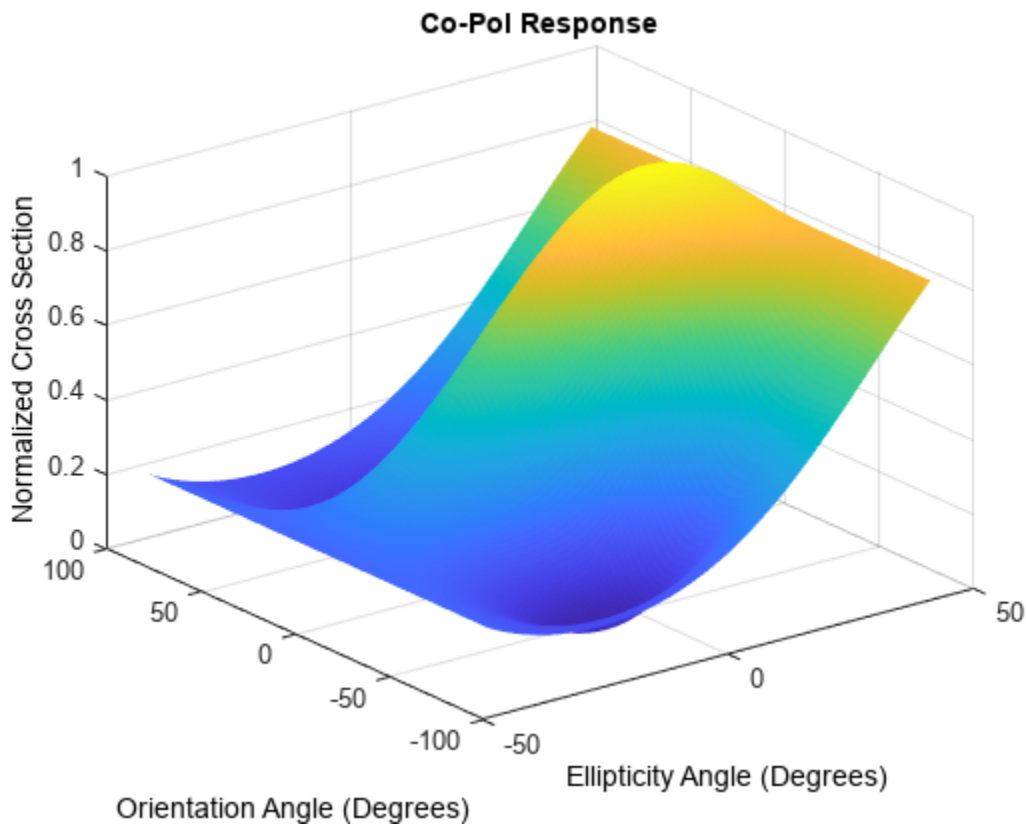
Plot the cross-polarizations signatures for all incident polarizations.

```
polsignature(rscmat,'x',el,tilt)
```

**Cross-Pol Response**
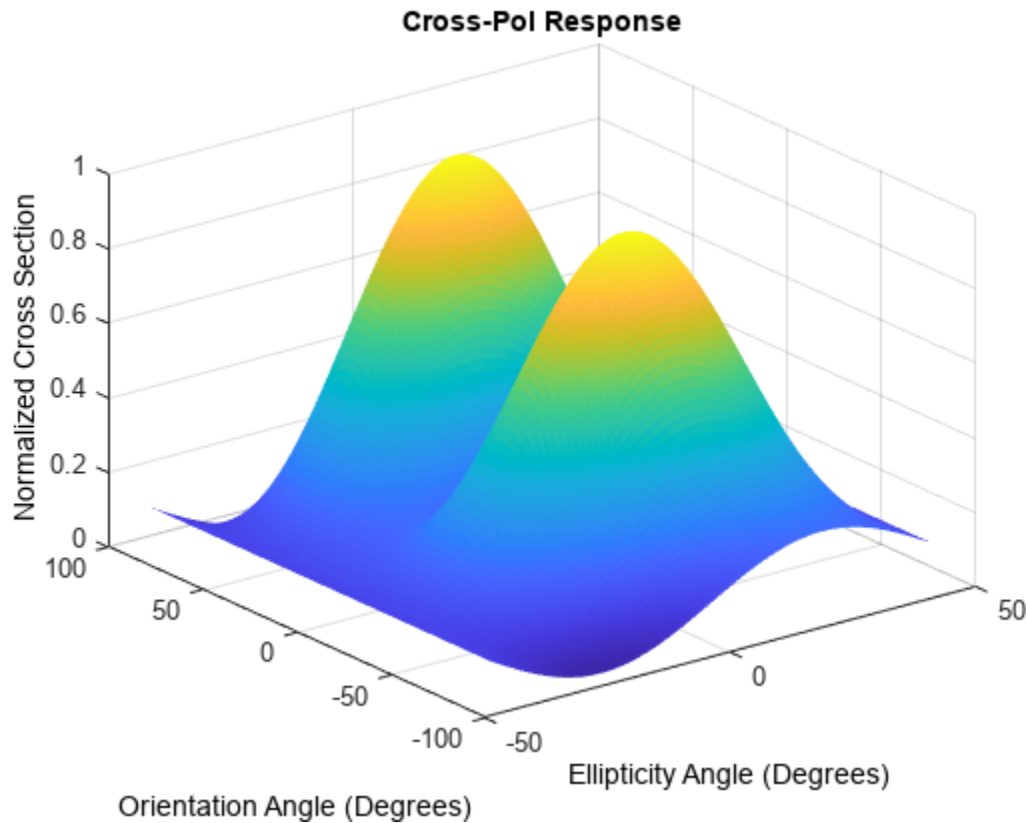
## Polarization Loss Due to Field and Receiver Mismatch

An antenna that is used to receive polarized electromagnetic waves achieves its maximum output power when the antenna polarization is matched to the polarization of the incident electromagnetic field. Otherwise, there is polarization loss:

- The polarization loss is computed from the projection (or dot product) of the transmitted field's electric field vector onto the receiver polarization vector.
- Loss occurs when there is a mismatch in direction of the two vectors, not in their magnitudes.
- The polarization loss factor describes the fraction of incident power that has the correct polarization for reception.

Using the transmitter's spherical basis at the receiver's position, you can represent the incident electric field, $(E_{iH}, E_{iV})$, by

$$\mathbf{E} = E_{iH}\widehat{\mathbf{e}}_H + E_{iV}\widehat{\mathbf{e}}_V = E_m\mathbf{P}_i$$

You can represent the receiver's polarization vector, $(P_H, P_V)$, in the receiver's local spherical basis by:

$$\mathbf{P} = P_H\widehat{\mathbf{e}}'_H + P_V\widehat{\mathbf{e}}'_V$$

The next figure shows the construction of the transmitter and receiver spherical basis vectors.

Transmitter Coordinate System
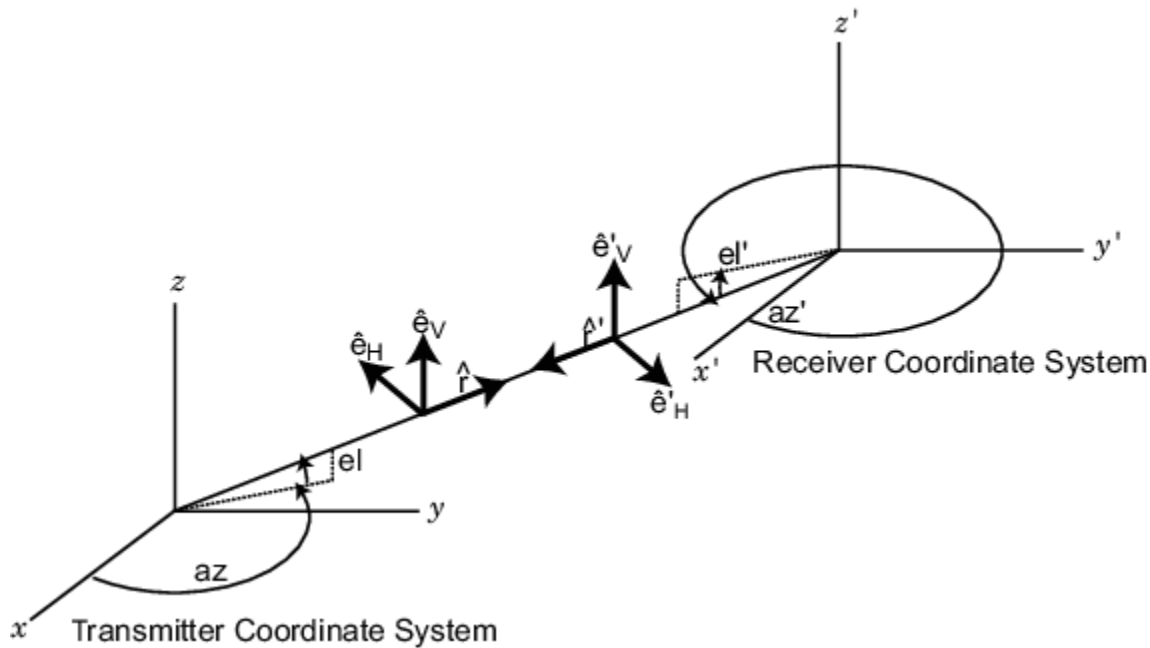
The polarization loss is defined by:

$$\rho = \frac{|\mathbf{E}_i \cdot \mathbf{P}|^2}{|\mathbf{E}_i|^2 \, |\mathbf{P}|^2}$$

and varies between 0 and 1. Because the vectors are defined with respect to different coordinate systems, they must be converted to the global coordinate system to form the projection. The toolbox function `polloss` computes the polarization mismatch between an incident field and a polarized antenna.

To achieve maximum output power from a receiving antenna, the matched antenna polarization vector must be the complex conjugate of the incoming field's polarization vector. As an example, if the incoming field is RHCP, with polarization vector given by $\mathbf{e}_r = \frac{1}{\sqrt{2}}(\mathbf{e}_x - i\mathbf{e}_y)$, the optimum receiver antenna polarization is LHCP. The introduction of the complex conjugate is needed because field polarizations are described with respect to its direction of propagation, whereas the polarization of a receive antenna is usually specified in terms of the direction of propagation towards the antenna. The complex conjugate corrects for the opposite sense of polarization when receiving.

As an example, if the transmitting antenna transmits an RHCP field, the polarization loss factors for various received antenna polarizations are

| Receive Antenna Polarization | Receive Antenna Polarization Vector | Polarization Loss Factor | Polarization Loss Factor (dB) |
|---|---|---|---|
| Horizontal linear | $\mathbf{e}_H$ | 1/2 | 3 dB |
| Vertical linear | $\mathbf{e}_V$ | 1/2 | 3 |
| RHCP | $\mathbf{e}_r = \frac{1}{\sqrt{2}}(\mathbf{e}_x - i\mathbf{e}_y)$ | 0 | $\infty$ |

| Receive Antenna Polarization | Receive Antenna Polarization Vector | Polarization Loss Factor | Polarization Loss Factor (dB) |
|---|---|---|---|
| LHCP | $\mathbf{e}_l = \dfrac{1}{\sqrt{2}}(\mathbf{e}_x + i\mathbf{e}_y)$ | 1 | 0 |

## Model Radar Transmitting Polarized Radiation

This example models a tracking radar based on a 31-by-31 (961-element) uniform rectangular array (URA). The radar is designed to follow a moving target. At each time instant, the radar points in the known direction of the target. The basic radar requirements are the probability of detection, `pd`, the probability of false alarm, `pfa`, the maximum unambiguous range, `max_range`, and the range resolution, `range_res`, (all distance units are in meters). The `range_gate` parameter limits the region of interest to a range smaller than the maximum range. The operating frequency is set in `fc`. The simulation lasts for `numpulses` pulses.

### Radar Definition

Set up the radar operating parameters. The existing radar design meets the following specifications.

```
pd = 0.9;               % Probability of detection
pfa = 1e-6;             % Probability of false alarm
max_range = 1500*1000;  % Maximum unambiguous range
range_res = 50.0;       % Range resolution
rangegate = 5*1000;     % Assume all objects are in this range
numpulses = 200;        % Number of pulses to integrate
fc = 8e9;               % Center frequency of pulse
c = physconst('LightSpeed');
tmax = 2*rangegate/c;   % Time of  echo from object at rangegate
```

### Pulse Repetition Interval

Set the pulse repetition interval, PRI, and pulse repetition frequency, PRF, based on the maximum unambiguous range.

```
PRI = 2*max_range/c;
PRF = 1/PRI;
```

### Transmitted Signal

Set up the transmitted rectangular waveform using the `phased.RectangularWaveform` System object™. The waveform pulse width, `pulse_width`, and pulse bandwidth, `pulse_bw`, are determined by the range resolution, `range_res`, that you select. Specify the sampling rate, `fs`, to be twice the pulse bandwidth. The sampling rate must be an integer multiple of the PRF. Therefore, modify the sampling rate to satisfy the requirement.

```
pulse_bw = c/(2*range_res);     % Pulse bandwidth
pulse_width = 1/pulse_bw;               % Pulse width
fs = 2*pulse_bw;                        % Sampling rate
n = ceil(fs/PRF);
fs = n*PRF;
waveform = phased.RectangularWaveform('PulseWidth',pulse_width,'PRF',PRF,...
    'SampleRate',fs);
```

**Antennas and URA Array**

The array consists of short-dipole antenna elements. Use the
`phased.ShortDipoleAntennaElement` System object to create a short-dipole antenna oriented
along the *z*-axis.

```
antenna = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[5e9,10e9],'AxisDirection','Z');
```

Define a 31-by-31 Taylor tapered uniform rectangular array using the `phased.URA` System object.
Set the size of the array using the number of rows, `numRows`, and the number of columns, `numCols`.
The distance between elements, `d`, is slightly smaller than one-half the wavelength, `lambda`. Compute
the array taper, `tw`, using separate Taylor windows for the row and column directions. Obtain the
Taylor weights using the `taylorwin` function. Plot the 3-D array response using the array `pattern`
method.

```
numCols = 31;
numRows = 31;
lambda = c/fc;
d = 0.9*lambda/2; % Nominal spacing
wc = taylorwin(numCols);
wr = taylorwin(numRows);
tw = wr*wc';
array = phased.URA('Element',antenna,'Size',[numCols,numRows],...
    'ElementSpacing',[d,d],'Taper',tw);
pattern(array,fc,-180:180,-90:90,'CoordinateSystem','polar','Type','powerdb',...
    'Polarization','V');
```
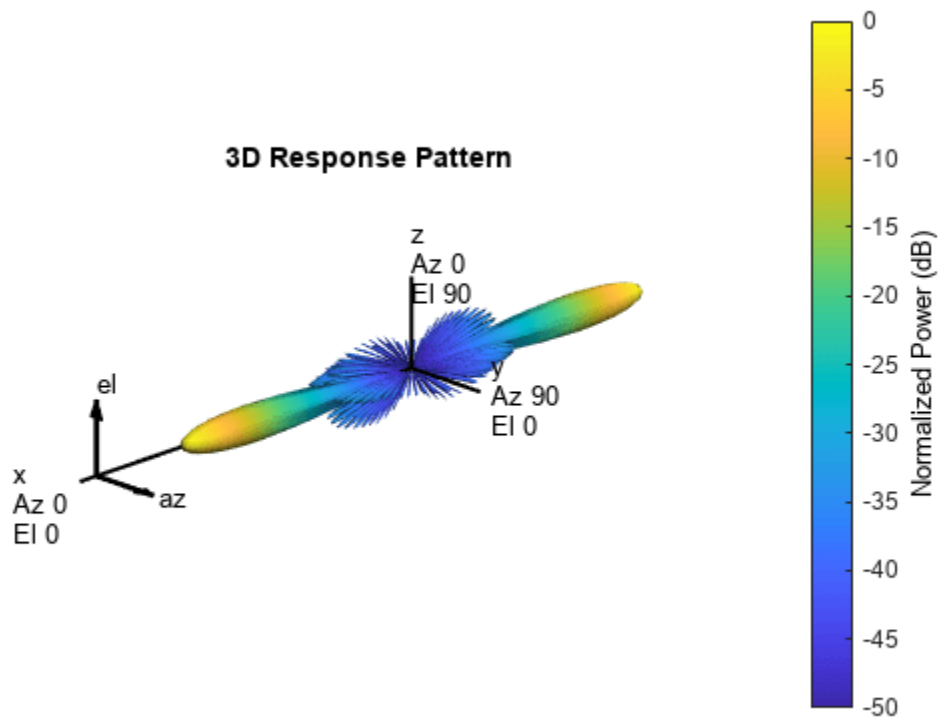
**Radar Platform Motion**

Next, set the position and motion of the radar platform in the `phased.Platform` System object. The radar is assumed to be stationary and positioned at the origin. Set the `Velocity` property to `[0,0,0]` and the `InitialPosition` property to `[0,0,0]`. Set the `InitialOrientationAxes` property to the identity matrix to align the radar platform coordinate axes with the global coordinate system.

```
radarPlatformAxes = [1 0 0;0 1 0;0 0 1];
radarplatform = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0],'OrientationAxes',radarPlatformAxes);
```

**Transmitters and Receivers**

In radar, the signal propagates in the form of an electromagnetic wave. The signal is radiated and collected by the antennas used in the radar system. Associate the array with a radiator System object, `phased.Radiator`, and two collector `System objects`, `phased.Collector`. Set the `WeightsInputPort` property of the radiator to `true` to enable dynamic steering of the transmitted signal at each execution of the radiator. Creating the two collectors allows for collection of both horizontal and vertical polarization components.

```
radiator = phased.Radiator('Sensor',array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'CombineRadiatedSignals',true,...
    'Polarization','Combined','WeightsInputPort',true);
collector1 = phased.Collector('Sensor',array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'Wavefront','Plane','Polarization','Combined',...
    'WeightsInputPort',false);
collector2 = phased.Collector('Sensor',array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'Wavefront','Plane','Polarization','Combined',...
    'WeightsInputPort',false);
```

Estimate the peak power needed in the `phased.Transmitter` System object to calculate the desired radiated power levels. The transmitted peak power is the power required to achieve a minimum-detection SNR, `snr_min`. You can determine the minimum SNR from the probability of detection, `pd`, and the probability of false alarm, `pfa`, using the `albersheim` function. Then, compute the peak power, `peak_power`, from the radar equation. The peak power depends on the overall signal gain, which is the sum of the transmitting element gain, `TransmitterGain` and the array gain, `AG`. The peak power also depends on the maximum detection range, `rangegate`. Finally, you need to supply a target cross-section value, `tgt_rcs`. A scalar radar cross section is used in this code section as an approximation even though the full polarization computation later uses a 2-by-2 radar cross section scattering matrix.

Using the radar equation formula, estimate the total transmitted power to achieve a required detection SNR using all the pulses.

The SNR has contributions from the transmitting element gain as well as the array gain. Compute first an estimate of the array gain, then add the array gain to the transmitter gain to get the peak power which achieves the desired SNR.

- Use an approximate target cross section of 1.0 for the radar equation even though the analysis calls for the full scattering matrix.
- Set the maximum range to be equal to the value of 'rangegate' since targets outside that range are of no interest.
- Compute the array gain as 10*log10(number of elements)

- Assume each element has a gain of 20 dB.

```
snr_min = albersheim(pd, pfa, numpulses);
AG = 10*log10(numCols*numRows);
tgt_rcs = 1;
TransmitterGain = 20;
tau = waveform.PulseWidth;
Ts = 290;
dbterm = db2pow(snr_min - 2*TransmitterGain + AG);
peak_power = (4*pi)^3*physconst('Boltzmann')*Ts/tau/tgt_rcs/lambda^2*rangegate^4*dbterm
```

```
peak_power = 5.1778e+05
```

```
transmitter = phased.Transmitter('PeakPower',peak_power,'Gain',TransmitterGain,...
    'LossFactor',0,'InUseOutputPort',true,'CoherentOnTransmit',true);
```

### Define Target

We want to simulate the pulse returns from a target that is rotating so that the scattering cross-section matrix changes from pulse to pulse. Create a rotating target object and a moving target platform. The rotating target is represented later as an angle-dependent scattering matrix. Rotation is in degrees per second.

```
targetSpeed = 1000;
targetVec = [-1;1;0]/sqrt(2);
target = phased.RadarTarget('EnablePolarization',true,...
    'Mode','Monostatic','ScatteringMatrixSource','Input port',...
    'OperatingFrequency',fc);
targetPlatformAxes = [1 0 0;0 1 0;0 0 1];
targetRotRate = 45;
targetplatform = phased.Platform('InitialPosition',[3500.0; 0; 0],...
    'Velocity', targetSpeed*targetVec);
```

### Other System Objects

- Steering vector defined by `phased.SteeringVector` System object.
- Beamformer defined by `phased.PhaseShiftBeamformer` System object. The `DirectionSource` property is set to `'Input Port'` to enable the beamformer to always points towards the known target direction at each execution.
- Free-space propagator using the `phased.FreeSpace` System object.
- Receiver preamp model using the `phased.ReceiverPreamp` system object.

### Signal Propagation

Because the reflected signals are received by an array, use a beamformer pointing to the steering direction to obtain the combined signal.

```
steeringvector = phased.SteeringVector('SensorArray',array,'PropagationSpeed',c,...
    'IncludeElementResponse',false);
beamformer = phased.PhaseShiftBeamformer('SensorArray',array,...
    'OperatingFrequency',fc,'PropagationSpeed',c,...
    'DirectionSource','Input port');
channel = phased.FreeSpace('SampleRate',fs,...
    'TwoWayPropagation',true,'OperatingFrequency',fc);
% Define a receiver with receiver noise
amplifier = phased.ReceiverPreamp('Gain',20,'LossFactor',0,'NoiseFigure',1,...
```

```
'ReferenceTemperature',290,'SampleRate',fs,'EnableInputPort',true,...
'PhaseNoiseInputPort',false,'SeedSource','Auto');
```

For such a large PRI and sampling rate, there will be too many samples per element. This will cause problems with the collector which has 961 channels. To keep the number of samples manageable, set a maximum range of 5 km. We know that the target is within this range.

This set of axes specifies the direction of the local coordinate axes with respect to the global coordinate system. This is the orientation of the target.

**Processing Loop**

Pre-allocate arrays for collecting data to be plotted.

```
sig_max_V  = zeros(1,numpulses);
sig_max_H  = zeros(1,numpulses);
tm_V = zeros(1,numpulses);
tm_H = zeros(1,numpulses);
```

After all the System objects are created, loop over the number of pulses to create the reflected signals.

```
maxsamp = ceil(tmax*fs);
fast_time_grid = (0:(maxsamp-1))/fs;
rotangle = 0.0;
for m = 1:numpulses
    x = waveform(); % Generate pulse
    % Capture only samples within range gated
    x = x(1:maxsamp);
    [s, tx_status] = transmitter(x);   % Create transmitted pulse
    % Move the radar platform and target platform.
    [radarPos,radarVel] = radarplatform(1/PRF);
    [targetPos,targetVel] = targetplatform(1/PRF);
    % Compute the known target angle
    [targetRng,targetAng] = rangeangle(targetPos,...
        radarPos,...
        radarPlatformAxes);
    % Compute the radar angle with respect to the target axes.
    [radarRng,radarAng] = rangeangle(radarPos,...
        targetPos,...
        targetPlatformAxes);
    % Calculate the steering vector designed to track the target
    sv = steeringvector(fc,targetAng);
    % Radiate the polarized signal toward the targat
    tsig1 = radiator(s,targetAng,radarPlatformAxes,conj(sv));
    % Compute the two-way propagation loss (4*pi*R/lambda)^2
    tsig2 = channel(tsig1,radarPos,targetPos,radarVel,targetVel);
    % Create a very simple model of a changing scattering matrix
    scatteringMatrix = [cosd(rotangle),0.5*sind(rotangle);...
        0.5*sind(rotangle),cosd(rotangle)];
    rsig1 = target(tsig2,radarAng,targetPlatformAxes,scatteringMatrix);  % Reflect off target
    % Collect the vertical component of the radiation.
    rsig3V = collector1(rsig1,targetAng,radarPlatformAxes);
    % Collect the horizontal component of the radiation. This
    % second collector is rotated around the x-axis to be more
    % sensitive to horizontal polarization
    rsig3H = collector2(rsig1,targetAng,rotx(90)*radarPlatformAxes);
    % Add receiver noise to both sets of signals
```

```matlab
        rsig4V = amplifier(rsig3V,~(tx_status>0)); % Receive signal
        rsig4H = amplifier(rsig3H,~(tx_status>0)); % Receive signal
        % Beamform the signal
        rsigV = beamformer(rsig4V,targetAng); % Beamforming
        rsigH = beamformer(rsig4H,targetAng); % Beamforming
        % Find the maximum returns for each pulse and store them in
        % a vector. Store the pulse received time as well.
        [sigmaxV,imaxV] = max(abs(rsigV));
        [sigmaxH,imaxH] = max(abs(rsigH));
        sig_max_V(m) = sigmaxV;
        sig_max_H(m) = sigmaxH;
        tm_V(m) = fast_time_grid(imaxV) + (m-1)*PRI;
        tm_H(m) = fast_time_grid(imaxH) + (m-1)*PRI;

        % Update the orientation of the target platform axes
        targetPlatformAxes = ...
            rotx(PRI*targetRotRate)*targetPlatformAxes;
        rotangle = rotangle + PRI*targetRotRate;
end
% Plot the vertical and horizontal polarization for each pulse as a
% function of time.
plot(tm_V,sig_max_V,'.')
hold on
plot(tm_H,sig_max_H,'r.')
hold off
xlabel('Time (sec)')
ylabel('Amplitude')
title('Vertical and Horizontal Polarization Components')
legend('Vertical','Horizontal')
grid on
```

Vertical and Horizontal Polarization Components
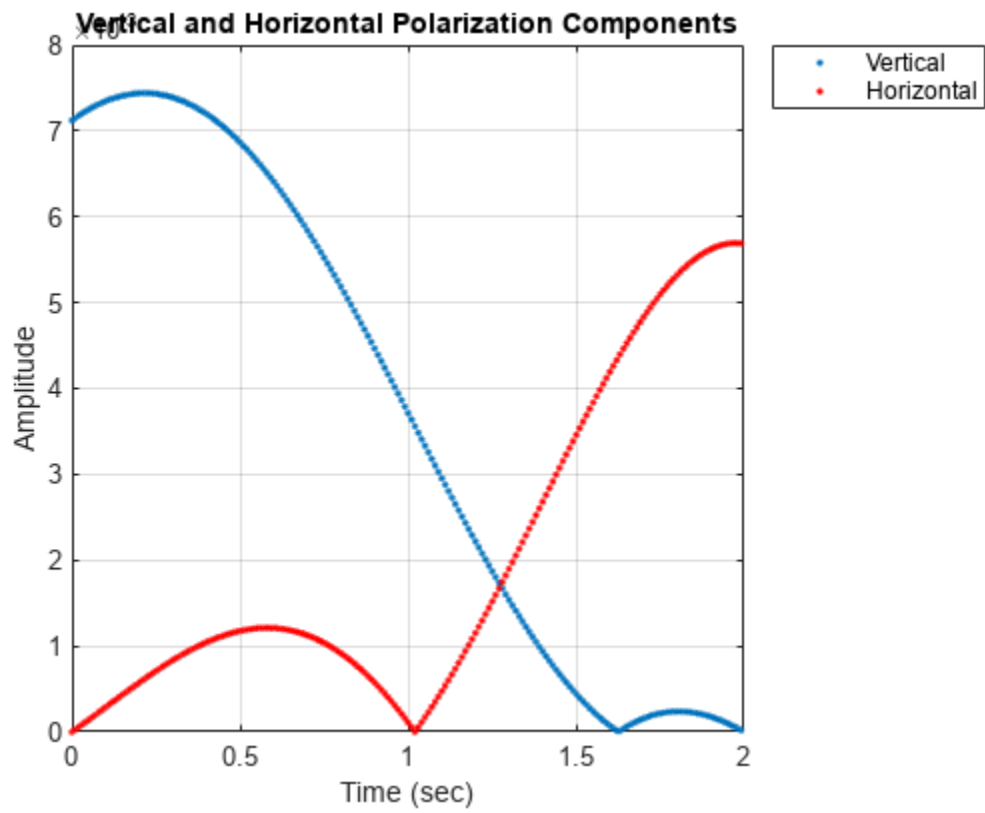
# Antenna and Array Definitions

# Element and Array Radiation and Response Patterns

| In this section... |
|---|
| "Element Response and Radiation Patterns" on page 12-2 |
| "Array Response and Radiation Patterns" on page 12-5 |

## Element Response and Radiation Patterns

Antennas and acoustic transducers create radiated fields which propagate outwards into space or into the air and water for acoustics. Conversely, antennas and transducers react to impinging fields to produce output voltages. Transducers are called microphones or speakers in speech acoustics or projectors or hydrophones in ocean acoustics. The electromagnetic fields created by an antenna, or the acoustic field created by a transducer, depend upon the distance and direction from the radiator. The terms *response pattern* and *radiation pattern* are often used interchangeably but the term *radiation pattern* is mostly used to describe the field radiated by an element and the term *response pattern* is mostly used to describe the output of the antenna with respect to impinging wave field as a function of wave direction. By the principle of reciprocity, these two patterns are identical. When discussing the generation of the patterns, it is conceptually easier to think in terms of radiation patterns.

In radar and sonar applications, the interactions between fields and targets take place in the far-field region, often called the Fraunhofer region. The far-field region is defined as the region for which

$$r \gg L^2/\lambda$$

where *L* represents the largest dimension of the source. In the far-field region, the fields take a special form: they can be written as the product of a function of direction (such as azimuth and elevation angles) and a geometric fall-off function, *1/r*. It is the angular function that is called the *radiation pattern*, *response pattern*, or simply *pattern*.

Radiation patterns can be viewed as field patterns or as power patterns. The terms "field" or "power" are often added to be more specific: contrast element field pattern versus element power pattern. The radiation power pattern describes the radiant intensity of a field, *U*, as a function of direction. Radiant intensity units are watts/steradian. Sometimes, radiant intensity is confused with power density. Power density, *I*, is the energy passing through a unit area in a unit time. Units for power density are Watts/square meter. Unfortunately, in some disciplines, power density is sometimes called intensity. This document always uses radiant intensity instead of intensity to avoid confusion. For a point source, the radiant intensity is the power density multiplied by the square of the distance from the source, $U = r^2 I$.

### Element Field Patterns

The *element field response* or *element field pattern* represents the angular distribution of the electromagnetic field create by an antenna, $E(\theta, \varphi)$, or the scalar acoustic field, $p(\theta, \varphi)$, generated by an acoustic transducer such as a speaker or hydrophone. Because the far field electromagnetic field consists of horizontal and vertical components orthogonal, $(E_H(\theta, \varphi), E_V(\theta, \varphi))$ there can be different patterns for each component. Acoustic fields are scalar fields so there is only one pattern. The general form of any field or field component is

$$A f(\theta, \phi) \frac{e^{-ikr}}{r}$$

where $A$ is a nominal field amplitude and $f(\theta,\varphi)$ is the normalized field pattern (normalized to unity). Because the field patterns are evaluated at some reference distance from the source, the fields returned by the element `step` method are represented simply as $A f(\theta,\varphi)$. You can display the nominal element field pattern by invoking the element `pattern` method and then choosing the `'Type'` parameter value as `'efield'` and setting the `'Normalize'` parameter to `false`.

```
pattern(elem,'Normalize',false,'Type','efield');
```

You can view the normalized field pattern by setting the `'Normalize'` parameter value to `true`. For example, if $E_H(\theta,\varphi)$ is the horizontal component of the complex electromagnetic field, the normalized field pattern has the form $|E_H(\theta,\varphi)/E_{H,max}|$.

```
pattern(elem,'Polarization','H','Normalize',true,'Type','efield');
```

**Element Power Patterns**

The *element power response* (or *element power radiation pattern*) is defined as the angular distribution of the radiant intensity in the far field, $U_{rad}(\theta,\varphi)$. When the elements are used for reception, the patterns are interpreted as the sensitivity of the element to radiation arriving from direction $(\theta,\varphi)$ and the power pattern represents the output voltage power of the element as a function of wave arrival direction.

Physically, the radiant intensity for the electromagnetic field produced by an antenna element is

$$U_{rad}(\theta,\phi) = \frac{r^2}{2Z_0}\left(|E_H|^2 + |E_V|^2\right)$$

where $Z_0$ is the characteristic impedance of free space. The radiant intensity of an acoustic field is

$$U_{rad}(\theta,\phi) = \frac{r^2}{2Z}\left|p\right|^2$$

where $Z$ is the characteristic impedance of the acoustic medium. For the fields produced by the Phased Array System Toolbox element System objects, the radial dependence, the impedances, and the field magnitudes are all collected in the nominal field amplitudes defined above. Then the radiant intensity can generally be written

$$U_{rad}(\theta,\phi) = |Af(\theta,\phi)|^2$$

The radiant intensity pattern is the quantity returned by the element `pattern` method when the `'Normalize'` parameter is set to `false` and the `'Type'` parameter is set to `'power'` (or `'powerdb'` for decibels).

```
pattern(elem,'Normalize',false,'Type','power');
```

The *normalized power pattern* is defined as the radiant intensity divided by its maximum value

$$U_{norm}(\theta,\phi) = \frac{U_{rad}(\theta,\phi)}{U_{rad,max}} = \left|f(\theta,\phi)\right|^2$$

The `pattern` method returns a normalized power pattern when the `'Normalize'` parameter is set to `true` and the `'Type'` parameter is set to `'power'` (or `'powerdb'` for decibels).

```
pattern(elem,'Normalize',true,'Type','power');
```

**Element Directivity**

*Element directivity* measures the capability of an antenna or acoustic transducer to radiate or receive power preferentially in a particular direction. Sometimes it is referred to as *directive gain*. Directivity is measured by comparing the transmitted radiant intensity in a given direction to the transmitted radiant intensity of an isotropic radiator having the same total transmitted power. An isotropic radiator radiates equal power in all directions. The radiant intensity of an isotropic radiator is just the total transmitted power divided by the solid angle of a sphere, *4π*,

$$U_{rad}^{iso}(\theta, \phi) = \frac{P_{total}}{4\pi}$$

The element directivity is defined to be

$$D(\theta, \phi) = \frac{U_{rad}(\theta, \phi)}{U_{rad}^{iso}} = 4\pi \frac{U_{rad}(\theta, \phi)}{P_{total}}$$

By this definition, the integral of the directivity over a sphere surrounding the element is exactly *4π*. Directivity is related to the effective *beamwidth* of an element. Start with an ideal antenna that has a uniform radiation field over a small solid angle (its beamwidth), *ΔΩ*, in a particular direction, and zero outside that angle. The directivity is

$$D(\theta, \phi) = 4\pi \frac{U_{rad}(\theta, \phi)}{P_{total}} = \frac{4\pi}{\Delta\Omega}$$

The greater the directivity, the smaller the beamwidth.

The radiant intensity can be expressed in terms of the directivity and the total power

$$U_{rad}(\theta, \phi) = \frac{1}{4\pi}D(\theta, \phi)P_{total}$$

As an example, the directivity of the electric field of a z-oriented short-dipole antenna element is

$$D(\theta, \phi) = \frac{3}{2}\cos^2\theta$$

with a peak value of 1.5. Often, the largest value of *D(θ,φ)* is specified as an antenna operating parameter. The direction in which *D(θ,φ)* is largest is the direction of maximum power radiation. This direction is often called the *boresight* direction. In some of the literature, the maximum value itself is called the *directivity*, reserving the phrase *directive gain* for what is called here *directivity*. For the short-dipole antenna, the maximum value of directivity occurs at *θ = 0*, independent of *φ*, and attains a value of *3/2*. The concept of directivity applies to receiving antennas as well. It describes the output power as a function of the arrival direction of a plane wave impinging upon the antenna. By reciprocity, the directivity of a receiving antenna is the same as the directivity when used as a transmitting antenna. A quantity closely related to directivity is *element gain*. The definition of directivity assumes that all the power fed to the element is radiated to space. In reality, system losses reduce the radiant intensity by some factor, the element efficiency, *η*. The term $P_{total}$ becomes the power supplied to the antenna and $P_{rad}$ becomes the power radiated into space. Then, $P_{rad} = \eta P_{total}$. The element gain is

$$G(\theta, \phi) = 4\pi \frac{U_{rad}(\theta, \phi)}{P_{total}} = 4\pi\eta \frac{U_{rad}(\theta, \phi)}{P_{rad}} = \eta D(\theta, \phi)$$

and represents the power radiated away from the element compared to the total power supplied to the element.

Using the element `pattern` method, you can plot the directivity of an element by setting the `'Type'` parameter to `'directivity'`,

```
pattern(elem,'Type','directivity');
```

## Array Response and Radiation Patterns

### Array Magnitude and Power Patterns

When individual antenna elements are aggregated into arrays of elements, new response/radiation patterns are created which depend upon both the element patterns and the geometry of the array. These patterns are called *beampatterns* to reflect the fact that the pattern can be constructed to have a narrow angular distribution, that is, a *beam*. This term is used for an array in transmitting or receiving modes. Most often, but not always, the array consists of identical antennas. The identical antenna case is interesting because it lets us partition the radiation pattern into two components: one component describes the element radiation pattern and the second describes the array radiation pattern.

Just as an array of transmitting elements has a radiation pattern, an array of receiving elements has a response pattern which describes how the output voltage of the array changes with the direction of arrival of a plane incident wave. By reciprocity, the response pattern is identical to the radiation pattern.

For transmitting arrays, the voltage driving the elements can be phase-adjusted to allow the maximum radiant intensity to be transmitted in a particular direction. For receiving arrays, the arriving signals can be phase-adjusted to maximize the sensitivity in a particular direction.

Start with a simple model of the radiation field produced by a single antenna which is given by

$$y(\theta, \phi, r) = A f(\theta, \phi) \frac{e^{-ikr}}{r}$$

where $A$ is the field amplitude and $f((\theta,\varphi)$ is the normalized element field pattern. This field can represent any of the components of the electric field, a scalar field, or an acoustic field. For an array of identical elements, the output of the array is the weighted sum of the individual elements, using the complex weights, $w_m$

$$z(\theta, \phi, r) = A \sum_{m=0}^{M-1} w_m^* f(\theta, \phi) \frac{e^{-ikr_m}}{r_m}$$

where $r_m$ is the distance from the m$^{\text{th}}$ element source point to the field point. In the far-field region, this equation takes the form

$$z(\theta, \phi, r) = A \frac{e^{-ikr}}{r} f(\theta, \phi) \sum_{m=0}^{M-1} w_m^* e^{-ik\mathbf{u} \cdot \mathbf{x}_m}$$

where $\mathbf{x}_m$ are the vector positions of the array elements with respect to the array origin. $\mathbf{u}$ is the unit vector from the array origin to the field point. This equation can be written compactly is the form

$$z(\theta, \phi, r) = A \frac{e^{-ikr}}{r} f(\theta, \phi) \mathbf{w}^H \mathbf{s}$$

The term $\mathbf{w}^H\mathbf{s}$ is called the *array factor*, $F_{array}(\theta,\varphi)$. The vector $\mathbf{s}$ is the *steering vector* (or *array manifold vector*) for directions of propagation for transmit arrays or directions of arrival for receiving arrays

$$\mathbf{s}(\theta, \phi) = \left\{..., e^{ik\mathbf{u} \cdot \mathbf{x}_m}, ...\right\}$$

The total *array pattern* consists of an amplitude term, an element pattern, $f(\theta,\varphi)$, and an array factor, $F_{array}(\theta,\varphi)$. The total angular behavior of the array pattern, $B(\theta,\varphi)$, is called the *beampattern* of the array

$$z(\theta, \phi, r) = A\frac{e^{-ikr}}{r}f(\theta, \phi)\mathbf{w}^H\mathbf{s} = A\frac{e^{-ikr}}{r}f(\theta, \phi)F_{array}(\theta, \phi) = A\frac{e^{-ikr}}{r}B(\theta, \phi)$$

When evaluated at the reference distance, the array field pattern has the form

$$Af(\theta, \phi)\mathbf{w}^H\mathbf{s} = Af(\theta, \phi)F_{array}(\theta, \phi) = AB(\theta, \phi)$$

The `pattern` method, when the `'Normalize'` parameter is set to `false` and the `'Type'` parameter is set to `'efield'`, returns the magnitude of the array field pattern at the reference distance.

```
pattern(array,'Normalize',false,'Type','efield');
```

When the `'Normalize'` parameter is set to `true`, the `pattern` method returns a pattern normalized to unity.

```
pattern(array,'Normalize',true,'Type','efield');
```

The array power pattern is given by

$$\left|Af(\theta, \phi)\mathbf{w}^H\mathbf{s}\right|^2 = \left|Af(\theta, \phi)F_{array}(\theta, \phi)\right|^2 = \left|AB(\theta, \phi)\right|^2$$

The `pattern` method, when the `'Normalize'` parameter is set to `false` and the `'Type'` parameter is set to `'power'` or `'powerdb'`, returns the array power pattern at the reference distance.

```
pattern(array,'Normalize',false,'Type','power');
```

When the `'Normalize'` parameter is set to `true`, the `pattern` method returns the power pattern normalized to unity.

```
pattern(array,'Normalize',true,'Type','power');
```

For the conventional beamformer, the weights are chosen to maximize the power transmitted towards a particular direction, or in the case of receiving arrays, to maximize the response of the array for a particular arrival direction. If $\mathbf{u_0}$ is the desired pointing direction, then the weights which maximize the power and response in this direction have the general form

$$\mathbf{w} = \left|w_m\right|e^{-ik\mathbf{u}_0 \cdot \mathbf{x}_m}$$

For these weights, the array factor becomes

$$F_{array}(\theta, \phi) = \sum_{m = 0}^{M - 1}\left|w_m\right|e^{-ik(\mathbf{u} - \mathbf{u}_0) \cdot \mathbf{x}_m}$$

which has a maximum at $\mathbf{u} = \mathbf{u}_0$.

**Array Directivity**

*Array directivity* is defined the same way as *element directivity*: the radiant intensity in a specific direction divided by the isotropic radiant intensity. The isotropic radiant intensity is the array total radiated power divided by *4π*. In terms of the arrays weights and steering vectors, the directivity can be written as

$$D(\theta, \phi) = 4\pi \frac{\left|Af(\theta, \phi)\mathbf{w}^H\mathbf{s}\right|^2}{P_{total}}$$

where $P_{total}$ is the total radiated power from the array. In a discrete implementation, the total radiated power can be computed by summing radiant intensity values over a uniform grid of angles that covers the full sphere surrounding the array

$$P_{total} = \frac{2\pi^2}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left|Af(\theta_m, \phi_n)\mathbf{w}^H\mathbf{s}(\theta_m, \phi_n)\right|^2 \cos\theta_m$$

where $M$ is the number of elevation grid points and $N$ is the number of azimuth grid points.

Because the radiant intensity is proportional to the beampattern, $B(\theta,\varphi)$, the directivity can also be written in terms of the beampattern

$$D(\theta, \phi) = 4\pi \frac{|B(\theta, \phi)|^2}{\int \left|B(\theta, \phi)\right|^2 \cos\theta d\theta d\phi}$$

You can plot the directivity of an array by setting the `'Type'` parameter of the `pattern` methods to `'directivity'`,

`pattern(array,'Type','directivity');`

**Array Gain**

In the Phased Array System Toolbox, *array gain* is defined to be the *array SNR gain*. Array gain measures the improvement in SNR of a receiving array over the SNR for a single element. Because an array is a spatial filter, the array SNR depends upon the spatial properties of the noise field. When the noise is spatially isotropic, the array gain takes a simple form

$$G = \frac{\text{SNR}_{array}}{\text{SNR}_{element}} = \frac{\left|\mathbf{w}^H\mathbf{s}\right|^2}{\mathbf{w}^H\mathbf{w}}$$

In addition, for an array with uniform weights, the array gain for an *N*-element array has a maximum value at boresight of *N* (or *10logN* in db).

# Grating Lobe Diagram for Microphone URA

Plot the grating lobe diagram for an 11-by-9-element uniform rectangular array having element spacing equal to one-half wavelength.

Assume the operating frequency of the array is 10 kHz. All elements are omnidirectional microphone elements. Steer the array in the direction 20 degrees in azimuth and 30 degrees in elevation. The speed of sound in air is 344.21 m/s at 21 deg C.

```
cair = 344.21;
f = 10.0e3;
lambda = cair/f;
microphone = phased.OmnidirectionalMicrophoneElement(...
    'FrequencyRange',[20 20000]);
array = phased.URA('Element',microphone,'Size',[11,9],...
    'ElementSpacing',0.5*lambda*[1,1]);
plotGratingLobeDiagram(array,f,[20;30],cair);
```
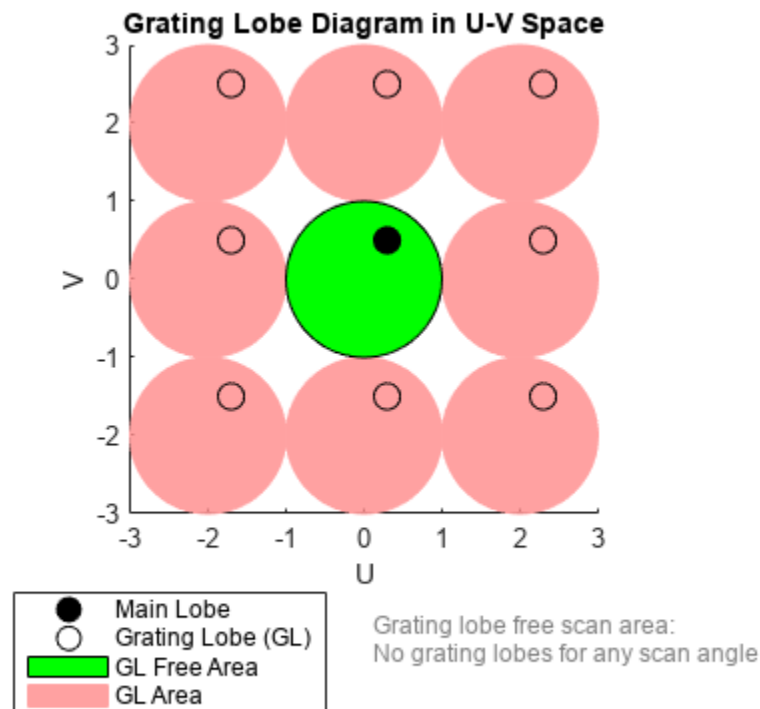


Plot the grating lobes. The main lobe of the array is indicated by a filled black circle. The grating lobes in visible and nonvisible regions are indicated by unfilled black circles. The visible region is the region in u-v coordinates for which $u^2 + v^2 \leq 1$. The visible region is shown as a unit circle centered at the origin. Because the array spacing is less than one-half wavelength, there are no grating lobes in the visible region of space. There are an infinite number of grating lobes in the nonvisible regions, but only those in the range [-3,3] are shown.

The grating-lobe free region, shown in green, is the range of directions of the main lobe for which there are no grating lobes in the visible region. In this case, it coincides with the visible region.

The white areas of the diagram indicate a region where no grating lobes are possible.

# Sonar System Models

# Sonar Equation

The sonar equation is used in underwater signal processing to relate received signal power to transmitted signal power for one-way or two-way sound propagation. The equation computes the received signal-to-noise ratio (SNR) from the transmitted signal level, taking into account transmission loss, noise level, sensor directivity, and target strength. The sonar equation serves the same purpose in sonar as the radar equation does in radar. The sonar equation has different forms for passive sonar and active sonar.

## Passive Sonar Equation

In a passive sonar system, sound propagates directly from a source to a receiver. The passive sonar equation is

$$SNR = SL - TL - (NL - DI)$$

where SNR is the received signal-to-noise ratio in dB.

### Source Level (SL)

The source level ($SL$) is the ratio of the transmitted intensity from the source to a reference intensity, converted to dB:

$$SL = 10\log\frac{I_s}{I_{ref}}$$

where $I_s$ is the intensity of the transmitted signal measured at 1 m distance from the source. The reference intensity, $I_{ref}$, is the intensity of a sound wave having a root mean square (rms) pressure of 1 μPa. Source level is sometimes written in dB// 1 μPa, but actually is referenced to the intensity of a 1 μPa signal. The relation between intensity and pressure is

$$I = \frac{p_{rms}^2}{\rho c}$$

where $\rho$ is the density of seawater, (approximately 1000 kg/m$^3$), $c$ is the speed of sound (approximately 1500 m/s). 1 μPa is equivalent to an intensity of $I_{ref} = 6.667 \times 10^{-19}$ W/m$^2$

Sometimes, it is useful to compute the source level from the transmitted power, $P$. Assuming a nondirectional (isotropic) source, the intensity at one meter from the source is

$$I = \frac{P}{4\pi}$$

Then, the source level as a function of transmitted power is

$$SL = 10\log_{10}\frac{I}{I_{ref}} = 10\log_{10}\frac{P}{4\pi I_{ref}} = 10\log_{10}P - 10\log_{10}4\pi I_{ref} = 10\log_{10}P + 170.8$$

When source level is defined at one yard instead of one meter, the final constant in this equation is 171.5.

When the source is directional, the source level becomes

$$SL = 10\log_{10}\frac{I}{I_{ref}} = 10\log_{10}P + 170.8 + DI_{src}$$

where $DI_{src}$ is the directivity of the source. Source directivity is not explicitly included in the sonar equation.

**Receiver Directivity Index (DI)**

The sonar equation includes the directivity index of the receiver (*DI*). Directivity is the ratio of the total noise power at the array to the noise received by the array along its main response axis. Directivity improves the signal-to-noise ratio by reducing the total noise. See "Element and Array Radiation and Response Patterns" on page 12-2 for discussions of directivity.

**Transmission Loss (TL)**

Transmission loss is the attenuation of sound intensity as the sound propagates through the underwater channel. Transmission loss (*TL*) is defined as the ratio of sound intensity at 1 m from a source to the sound intensity at distance *R*.

$$TL = 10\log\frac{I_s}{I(R)}$$

There are two major contributions to transmission loss. The larger contribution is geometrical spreading of the sound wavefront. The second contribution is absorption of the sound as it propagates. There are several absorption mechanisms.

In an infinite medium, the wavefront expands spherically with distance, and attenuation follows a *1/R²* law, where *R* is the propagation distance. However, the ocean channel has a surface and a bottom. Because of this, the wavefronts expand cylindrically when they are far from the source and follow a *1/R* law. Near the source, the wavefronts still expand spherically. There must be a transition region where the spreading changes from spherical to cylindrical. In Phased Array System Toolbox sonar models, the transition region as a single range and ensures that the transmission loss is continuous at that range. Authors define the transition range differently. Here, the transition range, $R_{trans}$, is one-half the depth, *D*, of the channel. The geometric transmission loss for ranges less than the transition range is

$$TL_{geom} = 20\log_{10}R$$

For ranges greater than the transition depth, the geometric transmission loss is

$$TL_{geom} = 10\log_{10}R + 10\log_{10}R_{trans}$$

In Phased Array System Toolbox, the transition range is one-half the channel depth, *H/2*.

The absorption loss model has three components: viscous absorption, the boric acid relaxation process, and the magnesium sulfate relaxation process. All absorption components are modeled by linear dependence on range, $\alpha R$.

Viscous absorption describes the loss of intensity due to molecular motion being converted to heat. Viscous absorption applies primarily to higher frequencies. The viscous absorption coefficient is a function of frequency, *f*, temperature in Celsius, *T*, and depth, *D*:

$$\alpha_{vis} = 4.9 \times 10^{-4}f^2e^{-(T/27 + D/17)}$$

in dB/km. This is the dominant absorption mechanism above 1 MHz. Viscous absorption increases with temperature and depth.

The second mechanism for absorption is the relaxation process of boric acid. Absorption depends upon the frequency in kHz, f, the salinity in parts per thousand (ppt), $S$, and temperature in Celsius,T. The absorption coefficient (measured in dB/km) is

$$\alpha_B = 0.106 \frac{f_1 f^2}{f_1^2 + f^2} e^{-(pH-8)/0.56}$$

$$f_1 = 0.78\sqrt{S/35} e^{T/26}$$

in dB/km. $f_1$ is the relaxation frequency of boric acid and is about 1.1 kHz at $T = 10$ °C and $S = 35$ ppt.

The third mechanism is the relaxation process of magnesium sulfate. Here, the absorption coefficient is

$$\alpha_M = 0.52\left(1 + \frac{T}{43}\right)\left(\frac{S}{35}\right)\frac{f_2 f^2}{f_2^2 + f^2} e^{-D/6}$$

$$f_2 = 42 e^{T/17}$$

in dB/km. $f_2$ is the relaxation frequency of magnesium sulfate and is about 75.6 kHz at T = 10°C and S = 35 ppt.

The total transmission loss modeled in the toolbox is

$$TL = TL_{geom}(R) + (\alpha_{vis} + \alpha_B + \alpha_M)R$$

where $R$ is the range in km. In Phased Array System Toolbox, all absorption models parameters are fixed at $T = 10$, $S = 35$, and $pH = 8$. The model is implemented in `range2tl`. Because $TL$ is a monotonically increasing function of $R$, you can use the Newton-Raphson method to solve for $R$ in terms of $TL$. This calculation is performed in `tl2range`.

### Noise Level (NL)

Noise level ($NL$) is the ratio of the noise intensity at the receiver to the same reference intensity used for source level.

## Active Sonar Equation

The active sonar equation describes a scenario where sound is transmitted from a source, reflects off a target, and returns to a receiver. When the receiver is collocated with the source, this sonar system is called monostatic. Otherwise, it is bistatic. Phased Array System Toolbox models monostatic sonar systems. The active sonar equation is

$$SNR = SL - 2TL - (NL - DI) + TS$$

where $2TL$ is the two-way transmission loss (in dB) and $TS$ is the target strength (in dB). The transmission loss is calculated by computing the outbound and inbound transmission losses (in dB) and adding them. In this toolbox, two-way transmission loss is twice the one-way transmission loss.

### Target Strength (TS)

Target strength is the sonar analog of radar cross section. Target strength is the ratio of the intensity of a reflected signal at 1 m from a target to the incident intensity, converted to dB. Using the

conservation of energy or, equivalently, power, the incident power on a target equals the reflected power. The incident power is the incident signal intensity multiplied by an effective cross-sectional area, $\sigma$. The reflected power is the reflected signal intensity multiplied by the area of a sphere of radius $R$ centered on the target. The ratio of the reflected power to the incident power is

$$I_{\text{inc}}\sigma = I_{\text{refl}}4\pi R^2$$

$$\frac{I_{\text{refl}}}{I_{\text{inc}}} = \frac{\sigma}{4\pi R^2}.$$

The reflected intensity is evaluated on a sphere of 1 m radius. The target strength coefficient ($\sigma$) is referenced to an area 1 m$^2$.

$$TS = 10\log_{10}\frac{I_{\text{refl}}(1 \text{ meter})}{I_{\text{inc}}} = 10\log_{10}\frac{\sigma}{4\pi}$$

## References

[1] Ainslie M. A. and J.G. McColm. "A simplified formula for viscous and chemical absorption in sea water." *Journal of the Acoustical Society of America*. Vol. 103, Number 3, 1998, pp. 1671--1672.

[2] Urick, Robert J. *Principles of Underwater Sound*, 3rd ed. Los Altos, CA: Peninsula Publishing, 1983.

# Doppler Effect for Sound

The Doppler effect is the change in the observed frequency of a source due to the motion of either the source or receiver or both. Only the component of motion along the line connecting the source and receiver contributes to the Doppler effect. Any arbitrary motion can be replaced by motion along the source-receiver axis with velocities consisting of the projections of the velocities along that axis. Therefore, without loss of generality, assume that the source and receiver move along the x-axis and that the receiver is positioned further out along the x-axis. The source emits a continuous tone of frequency, $f_0$, equally in all directions. First examine two important cases. The first case is where the source is stationary and the receiver is moving toward or away from the source. A receiver moving away from the source will have positive velocity. A receiver moving toward the source will have negative velocity. If the receiver moves towards the source, it will encounter wave crests more frequently and the received frequency will increase according to

$$f' = f_0\left(\frac{c - v_r}{c}\right)$$

Frequency will increase because $v_r$ is negative. If the receiver is moving away from the source, the $v_r$ is positive and the frequency decreases. A similar situation occurs when the source is moving and the receiver is stationary. Then the frequency at the receiver is

$$f' = f_0\left(\frac{c}{c - v_s}\right)$$

The frequency increases when $v_s$ is positive as the source moves toward the receiver. When $v_s$ is negative, the frequency decreases. Both effects can be combined into

$$f' = f_0\left(\frac{c - v_r}{c}\right)\left(\frac{c}{c - v_s}\right) = f_0\left(\frac{c - v_r}{c - v_s}\right) = f_0\left(\frac{1 - v_r/c}{1 - v_s/c}\right).$$

There is a difference in the Doppler formulas for sound versus electromagnetic waves. For sound, the Doppler shift depends on both the source and receiver velocities. For electromagnetic waves, the Doppler shift depends on the difference between the source and receiver velocities.

## References

[1] Halliday, David, R. Resnick, and J. Walker, *Fundamentals of Physics*, 10th ed. Wiley, New York, 2013.

# Code Generation

# Code Generation

## Code Generation Use and Benefits

You can use the Phased Array System Toolbox software together with the MATLAB Coder™ product to create C/C++ code that implements your MATLAB functions and models. With this software, you can

- Create a MEX file to speed up your own MATLAB application.
- Generate a stand-alone executable that runs independently of MATLAB on your own computer or another platform.
- Include System objects in the same way as any other element.

In general, the code you generate using the toolbox is portable ANSI® C code. In order to use code generation, you need a MATLAB Coder license. Using Phased Array System Toolbox software requires licenses for both the DSP System Toolbox™ and the Signal Processing Toolbox™. See the "Get Started with MATLAB Coder" (MATLAB Coder) page for more information.

Creating a MATLAB Coder MEX-file can lead to substantial acceleration of your MATLAB algorithms. It is also a convenient first step in a workflow that ultimately leads to completely standalone code. When you create a MEX-file, it runs in the MATLAB environment. Its inputs and outputs are available for inspection just like any other MATLAB variable. You can use MATLAB visualization and other tools for verification and analysis.

Within your code, you can run specific commands either as generated C code or by running using the MATLAB engine. In cases where an isolated command does not yet have code generation support, you can use the `coder.extrinsic` command to embed the command in your code. This means that the generated code reenters the MATLAB environment when it needs to run that particular command. This also useful if you wish to embed certain commands that cannot generate code (such as plotting functions).

The simplest way to generate MEX-files from your MATLAB code is by using the `codegen` function at the command line. Often, generating a MEX-files involves nothing more than invoking the `coder` command on one of your existing functions. For example, if you have an existing function, `myfunction.m`, you can type the commands at the command line to compile and run the MEX function. `codegen` adds a platform-specific extension to this name. In this case, the `"mex"` suffix is added.

```
codegen myfunction.m
myfunction_mex;
```

You can generate standalone executables that run independently of the MATLAB environment. You can do this by creating a MATLAB Coder project inside the MATLAB Coder Integrated Development Environment (IDE). Alternatively, you can issue the `codegen` command in the command line environment with appropriate configuration parameters. To create a standalone executable, you must

write your own `main.c` or `main.cpp` function. See "Generating Standalone C/C++ Executables from MATLAB Code" (MATLAB Coder) for more information.

**Set Up Your Compiler**

Before using `codegen` to compile your code, you must set up your C/C++ compiler. For 32-bit Windows platforms, MathWorks® supplies a default compiler with MATLAB. If your installation does not include a default compiler, you can supply your own compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site. Install a compiler that is suitable for your platform. Then, read "Setting Up the C or C++ Compiler" (MATLAB Coder). After installation, at the MATLAB command prompt, run `mex -setup`. You can then use the `codegen` function to compile your code.

**Functions and System Objects That Support Code Generation**

Almost all Phased Array System Toolbox functions and System objects are supported for code generation. For a list of supported functions and System objects, see Function List (C/C++ Code Generation).

## Limitations Specific to Phased Array System Toolbox

Code Generation has the following limitations when used with the Phased Array System Toolbox software:

- When you employ antennas and arrays that produce polarized fields, the `EnablePolarization` parameter for these System objects must be set to `true`:

  - `phased.Collector`
  - `phased.Radiator`
  - `phased.WidebandCollector`
  - `phased.WidebandRadiator`
  - `phased.RadarTarget`
  - `phased.BackscatterRadarTarget`
  - `phased.WidebandBackscatterRadarTarget`
  - `phased.ArrayResponse`
  - `phased.SteeringVector`

  This requirement differs from regular MATLAB usage where you can set `EnablePolarization` property to `false` even when you use a polarization-enabled antenna. For example, this code uses a polarized antenna, which requires that `EnablePolarization` property of the `phased.Radiator` System object be set to `true`.

  ```
  function [y] = codegen_radiator()
  sSD = phased.ShortDipoleAntennaElement(...
      'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
  c = physconst('LightSpeed');
  fc = 200e6;
  lambda = c/fc;
  d = lambda/2;
  sURA = phased.URA('Element',sSD,...
      'Size',[3,3],...
      'ElementSpacing',[d,d]);
  ```

```
sRad = phased.Radiator('Sensor',sURA,...
    'OperatingFrequency',150e6,...
    'CombineRadiatedSignals',true,...
    'EnablePolarization',true);
x = [1;2;1];
radiatingAngle = [10;0]; % One angle for one antenna
y = sRad(x,radiatingAngle,eye(3,3));
```

- Visualization methods for Phased Array System Toolbox System objects are not supported. These methods are `pattern`, `patternAzimuth`, `patternElevation`, `plot`, `plotResponse`, and `viewArray`.

- When a System object contains another System object as a property value, you must set the contained System object in the constructor. You cannot use Object Property notation to set the property. For example

```
antenna = phased.ShortDipoleAntennaElement( ...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
array = phased.URA('Element',antenna,'Size',[3,3], ...
    'ElementSpacing',[0.75,0.75]);
```

is valid for `codegen` but

```
antenna = phased.ShortDipoleAntennaElement( ...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
array = phased.URA('Size',[3,3],'ElementSpacing',[0.75,0.75]);
array.Element = antenna;
```

is not.

- Code generation of Phased Array System Toolbox arrays that contain Antenna Toolbox antennas is not supported.

- A list of the limitations on Phased Array System Toolbox functions and System objects is presented here:

| Function or System Object | Limitation |
| --- | --- |
| `plotResponse` | This System object method is not supported. |
| `pattern` | This System object method is not supported. |
| `patternAzimuth` | This System object method is not supported. |
| `patternElevation` | This System object method is not supported. |
| `plot` | This System object method is not supported. |
| `viewArray` | This System object method is not supported. |
| `ambgfun` | Supported only when output arguments are specified. |
| `pambgfun` | Supported only when output arguments are specified. |
| `polsignature` | Supported only when output arguments are specified. |
| `rocpfa` | <ul><li>Supported only when output arguments are specified.</li><li>The `NonfluctuatingNoncoherent` signal type is not supported.</li></ul> |

| Function or System Object | Limitation |
|---|---|
| `rocsnr` | • Supported only when output arguments are specified.<br>• The `NonfluctuatingNoncoherent` signal type is not supported. |
| `stokes` | Supported only when output arguments are specified. |
| `phased.ArrayGain` | This System object cannot be used with arrays that contain antenna elements that create polarized signals, that is, `phased.ShortDipoleAntennaElement` or `phased.CrossedDipoleAntennaElement` |
| `phased.IntensityScope` | This System object is not supported. |
| `phased.MatchedFilter` | The `CustomSpectrumWindow` property is not supported. |
| `phased.RangeDopplerResponse` | The `CustomRangeWindow` and the `CustomDopplerWindow` properties are not supported. |
| `phased.ScenarioViewer` | This System object is not supported. |

## General Limitations

Code Generation has some general limitations not specifically related to the Phased Array System Toolbox software. For a more complete discussion, see "System Objects in MATLAB Code Generation" (MATLAB Coder).

• The data type and complexity (i.e., real or complex) of any input argument to a function or System object must always remain the same.
• You cannot pass a System object to any method or function that you made extrinsic using `coder.extrinsic`.
• You cannot load a MAT-file using `coder.load` when it contains a System object. For example, if you construct a System object in the MATLAB environment and save it to a MAT-file

```
sSD = phased.ShortDipoleAntennaElement(...
          'FrequencyRange',[0.9e8,2e9],'AxisDirection','Y');
save x.mat sSD;
clear sSD;
```

then you cannot load the System object in your compiled MEX-file:

```
function codegen_load1()
W = coder.load('x.mat');
sSD = W.sSD;
```

The compilation

```
codegen codegen_load1
```

will produced an error message: `'Found unsupported class for variable using function 'coder.load'. MATLAB class 'phased.ShortDipoleAntennaElement' found at 'W.sSD' is unsupported.'`

To avoid this problem, you can save the object's properties to a MAT-file, then, use `coder.load` to load the object properties and re-create the object. For example, create and save a System object's properties in the MATLAB environment

```
sSD = phased.ShortDipoleAntennaElement(...
            'FrequencyRange',[0.9e8,2e9],'AxisDirection','Y');
FrequencyRange = sSD.FrequencyRange;
AxisDirection = sSD.AxisDirection;
save x.mat FrequencyRange AxisDirection;
```

Then, write a function `codegen_load2` to load the properties and create a System object.

```
function codegen_load2()
W = coder.load('x.mat');
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',W.FrequencyRange,...
    'AxisDirection',W.AxisDirection);
```

Then, issue the commands to create and execute the MEX-file, `codegen_load2_mex`.

```
codegen codegen_load2;
codegen_load2_mex
```

* System object properties are either tunable or nontunable. Unless otherwise specified, System object properties are nontunable. Nontunable properties must be constant. A constant is a value that can be evaluated at compile-time. You can change tunable properties even if the object is locked. Refer to the object's reference page to determine whether an individual property is tunable or not. If you try to set a nontunable System object property and the compiler determines that it is not constant, you will get an error. For example, the `phased.URA` System object has a nontunable property, `ElementSpacing`, which sets the distance between elements. You may want to create an array that is tuned to a frequency. You cannot pass in the frequency as an input argument because the frequency must be a constant.

```
function [resp] = codegen_const1(fc)
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
c = physconst('LightSpeed');
lambda = c/fc;
d = lambda/2;
sURA = phased.URA('Element',sSD,...
    'Size',[3,3],...
    'ElementSpacing',[d,d]);
ang = [30;0];
resp = sURA(fc,ang);
```

When you `codegen` this function

```
fc = 200e6;
codegen codegen_const1 -args {fc}
```

the compiler responds that the value of the `'ElementSpacing'` property, `d`, is not constant and generates the error message: `"Failed to compute constant value for nontunable property 'ElementSpacing'. In code generation, nontunable properties can only be assigned constant values."` It is not constant because it depends upon a non-constant variable, `fc`.

To correct this problem, set `fc` to a constant within the function:

```
function [resp] = codegen_const2()
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
c = physconst('LightSpeed');
fc = 200e6;
lambda = c/fc;
d = lambda/2;
sURA = phased.URA('Element',sSD,...
    'Size',[3,3],...
    'ElementSpacing',[d,d]);
ang = [30;0];
resp = sURA(fc,ang);
```

and then compile

```
codegen codegen_const2
```

- You can assign a nontunable System object property value only once before you call the object. This requirement differs from MATLAB usage where you can initialize these properties multiple times before calling the object.

  This example sets the `Size` property twice.

```
function codegen_property
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[0.9e8,2e9],'AxisDirection','Y');
sURA = phased.URA('Element',sSD,...
    'Size',[3,3],...
    'ElementSpacing',[0.15,0.15]);
sURA.Size = [4,4];
```

  When you issue the command

```
codegen codegen_property
```

  the following error message is produced: "`A nontunable property may only be assigned once.`"

- In certain cases, the compiler cannot determine the values of nontunable properties at compile time or the code may not even compile. Consider the following example that reads in the *x,y,z*-coordinates of a 5-element array from a file and then, creates a conformal array System object. The text file, `elempos.txt`, contains the element coordinates

```
-0.5000 -0.2588  0        0.2588  0.5000
-0.8660 -0.9659 -1.0000 -0.9659 -0.8660
 0       0        0        0       0
```

  The file `collectWave.m` contains reads the element coordinates and creates the object.

```
function y = collectWave(angle)
elPos = calcElPos;
cArr = phased.ConformalArray('ElementPosition',elPos);
y = collectPlaneWave(cArr,randn(4,2),angle,1e8);
end

function elPos = calcElPos
fid = fopen('elempos.txt','r');
el = textscan(fid, '%f');
n = length(el{1});
```

```
nelem = n/3;
fclose(fid);
elPos = reshape(el{1},nelem,3).';
end
```

Attempting to compile

```
codegen collectWave -args {[10 30]}
```

produces the error "`Permissions 'r' and 'r+' are not supported`".

The following example is a work-around that uses `coder.extrinsic` and `coder.const` to insure that the value for the nontunable property, `'ElementPosition'`, is a compile time constant. The function in the file, `collectWave1.m`, creates the object using the `calcElPos` function. This function runs inside the MATLAB interpreter at compile time.

```
function y = collectWave1(angle)
coder.extrinsic('calcElPos')
elPos = coder.const(calcElPos);
cArr = phased.ConformalArray('ElementPosition',elPos);
y = collectPlaneWave(cArr,randn(4,2),angle,1e8);
end
```

The file `calcElPos.m` loads the element positions from the text file

```
function elPos = calcElPos
fid = fopen('elempos.txt','r');
el = textscan(fid, '%f');
n = length(el{1});
nelem = n/3;
fclose(fid);
elPos = reshape(el{1},nelem,3).';
```

Only the `collectWave1.m` file is compiled with `codegen`. Compiling and running

```
codegen collectWave1 -args {[10 30]}
collectWave1_mex([10,30])
```

will succeed.

An alternate work-around uses `coder.load` to insure that the value of the nontunable property `'ElementPosition'` is compile-time constant. In the MATLAB environment, run `calcElPos2` to save the array coordinates contained in `elempos.txt` to a MAT-file. Then, load the contents of the MAT-file within the compiled code.

```
function calcElPos2
fid = fopen('elempos.txt');
el = textscan(fid, '%f');
fclose(fid);
elPos = reshape(el{1},[],3).';
save('positions', 'elPos');
end
```

The file `collectWave2.m` loads the coordinate positions and creates the conformal array object

```
function y = collectWave2(angle)
var = coder.load('positions');
cArr = phased.ConformalArray('ElementPosition',var.elPos);
```

```
y = collectPlaneWave(cArr,randn(4,2),angle,1e8);
end
```

Only the `collectWave2.m` file is compiled with `codegen`. Compiling and running `collectWave2.m`

```
codegen collectWave2 -args {[10 30]}
collectWave2_mex([10,30])
```

will succeed. This second approach is more general than the first since a MAT-file can contain any variables, except System objects.

- The System object `clone` method is not supported.

## Limitations for System Objects that Require Dynamic Memory Allocation

System objects that require dynamic memory allocation cannot be used for code generation in the following cases:

| Inside a MATLAB Function block in a Simulink® model. |
|---|
| Inside a MATLAB function in a Stateflow® chart. |
| When using MATLAB as the action language in a Stateflow chart. |
| Inside a Truth Table block in a Simulink model. |
| Inside a MATLAB System block (except for normal mode). |
| When using Simulink Coder for code generation. |
| When using MATLAB Coder for code generation and dynamic memory allocation is disabled. |

# Generate MEX Function to Estimate Directions of Arrival

Compile, using `codegen`, the function `EstimateDOA.m`. This function estimates the directions-of-arrival (DOA's) of two signals with added noise that are received by a standard 10-element Uniform Line Array (ULA). The antenna operating frequency is 150 MHz and the array elements are spaced one-half wavelength apart. The actual direction of arrival of the first signal is 10° azimuth, 20° elevation. The direction of arrival of the second signal is 45° azimuth, 60° elevation. Signals and noise are generated using the `sensorsig` function.

```matlab
function [az] = EstimateDOA()
% Example:
% Estimate the DOAs of two signals received by a standard
% 10-element ULA with element spacing one half-wavelength apart.
% The antenna operating frequency is 150 MHz.
% The actual direction of the first signal is 10 degrees in
% azimuth and 20 degrees in elevation. The direction of the
% second signal is 45 degrees in azimuth and 60 degrees in
% elevation.
c = physconst('LightSpeed');
fc = 150e6;
lambda = c/fc;
fs = 8000;
nsamp = 8000;
sigma = 0.1;
ang = [10 20; 45 60]';
antenna = phased.IsotropicAntennaElement( ...
    'FrequencyRange',[100e6,300e6]);
array = phased.ULA('Element',antenna,'NumElements',10, ...
    'ElementSpacing',lambda/2);
pos = getElementPosition(array)/lambda;
sig = sensorsig(pos,nsamp,ang,sigma^2);
estimator = phased.RootMUSICEstimator('SensorArray',array,...
    'OperatingFrequency',fc,...
    'NumSignalsSource','Property','NumSignals',2);
doas = estimator(sig);
az = broadside2az(sort(doas),[20,60]);
end
```

Run `codegen` at the command line to generate the mex function, `EstimateDOA_mex`, and then run the mex function:

```matlab
codegen EstimateDOA.m
EstimateDOA_mex
```

The estimated arrival angles are:

```
az =

  10.0036   45.0030
```

The program contains a fixed value for the noise variance. If you wanted to reuse the same code for different noise levels, you can pass the noise variance as an argument into the function. This is done in the function `EstimateDOA1.m`, shown here, which has the input argument `sigma`.

```matlab
function [az] = EstimateDOA1(sigma)
% Example:
% Estimate the DOAs of two signals received by a standard
```

```
% 10-element ULA with element spacing one half-wavelength apart.
% The antenna operating frequency is 150 MHz.
% The actual direction of the first signal is 10 degrees in
% azimuth and 20 degrees in elevation. The direction of the
% second signal is 45 degrees in azimuth and 60 degrees in
% elevation.
c = physconst('LightSpeed');
fc = 150e6;
lambda = c/fc;
fs = 8000;
nsamp = 8000;
ang = [10 20; 45 60]';
antenna = phased.IsotropicAntennaElement( ...
    'FrequencyRange',[100e6,300e6]);
array = phased.ULA('Element',antenna,'NumElements',10, ...
    'ElementSpacing',lambda/2);
pos = getElementPosition(array)/lambda;
sig = sensorsig(pos,nsamp,ang,sigma^2);
estimator = phased.RootMUSICEstimator('SensorArray',array, ...
    'OperatingFrequency',fc, ...
    'NumSignalsSource','Property','NumSignals',2);
doas = estimator(sig);
az = broadside2az(sort(doas),[20,60]);
end
```

Run `codegen` at the command line to generate the mex function, `EstimateDOA1_mex`, using the -`args` option to specify the type of input argument. Then run the mex function with several different input parameters:

```
codegen EstimateDOA1.m -args {1}
EstimateDOA1_mex(1)
az =
   10.0130   45.0613
EstimateDOA1_mex(10)
az =
   10.1882   44.3327
EstimateDOA1_mex(15)
az =
    8.1620   46.2440
```

Increasing the value of `sigma` degrades the estimates of the azimuth angles.

# Generate MEX Function Containing Persistent System Objects

Sometimes, it is convenient to put System objects inside a function that is to be called many times. This eliminates the overhead in creating new instances of a System object each time the function is called. You can write logic which creates the System object just once and declares it to be `persistent`. For example, suppose you require the response of an 11-element ULA for several different arrival angles and want to plot that response versus angle.

```matlab
function plot_ULA_response
azangles = [-90:90];
elangles = zeros(size(azangles));
fc = 100e9;
c = physconst('LightSpeed');
N = size(azangles,2);
lambda = c/fc;
d = 0.4*lambda;
numelements = 11;
resp = zeros(1,N);
sIso = phased.IsotropicAntennaElement(...
    'FrequencyRange',[1,200]*1e9,...
    'BackBaffled',false);
sULA = phased.ULA('Element',sIso,...
    'NumElements',numelements,...
    'ElementSpacing',d,...
    'Taper',taylorwin(numelements).');
for n = 1:N
    x = get_ULA_response(sULA,fc,azangles(n),elangles(n));
    resp(n) = abs(x);
end
plot(azangles,20*log10(resp));
title('ULA Response');
xlabel('Angle (deg)');
ylabel('Response (db)');
grid;
end

function resp = get_ULA_response(sULA,fc,az,el)
persistent sAR;
c = physconst('LightSpeed');
if isempty(sAR)
    sAR = phased.ArrayResponse('SensorArray',sULA,...
        'PropagationSpeed',c,...
        'WeightsInputPort',false,...
        'EnablePolarization',false);
end
resp = sAR(fc,[az;el]);
end
```

To create the code, run `codegen`to create the MEX-file `plot_ULA_response_mex`, and execute the mex-file at the command line:

```matlab
codegen plot_ULA_response
plot_ULA_response_mex;
```

which yields the plot
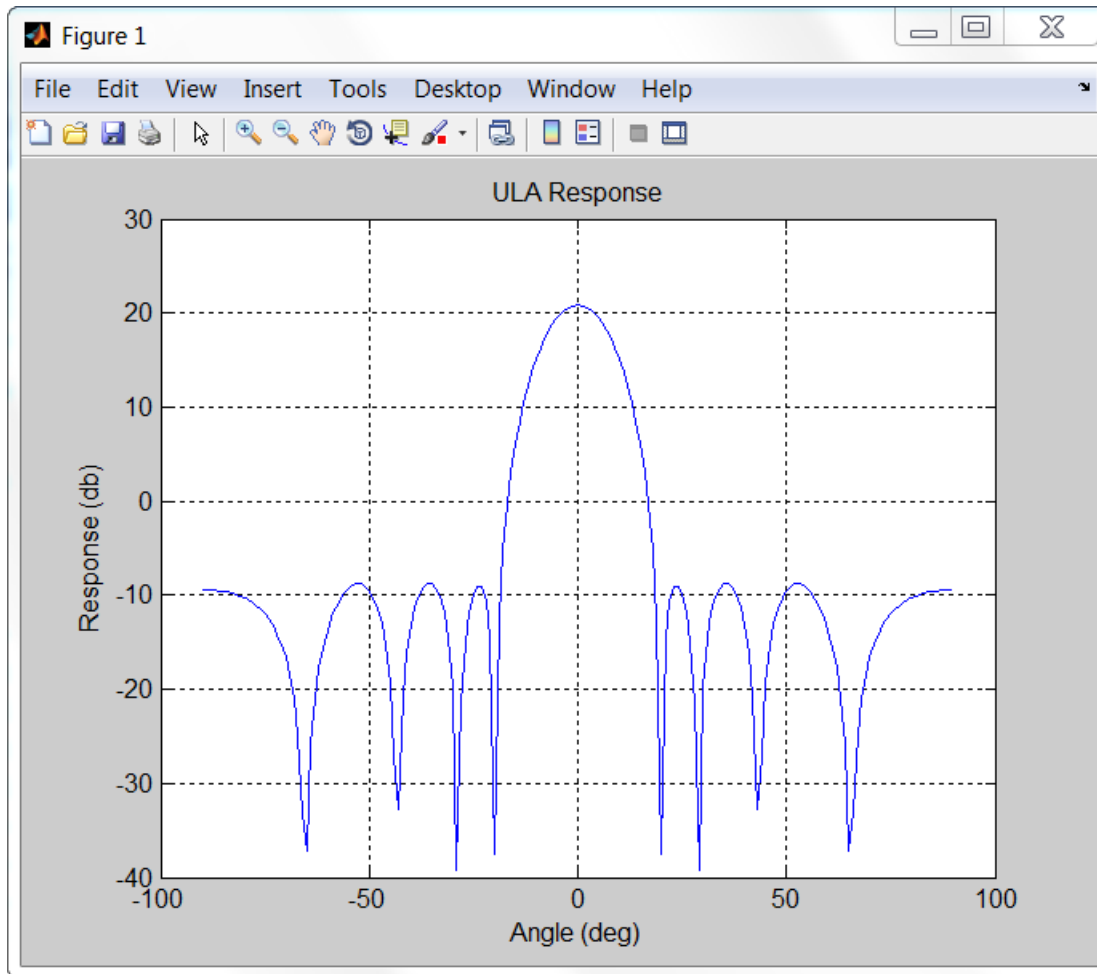
# Simulink Examples

# Convert Azimuth and Elevation to Broadside Angle

Convert a direction with an azimuth angle of 45 degrees and an elevation angle of 30 degrees into a broadside angle.

```
model = 'AzEl2Broadside1';
open_system(model);
sim(model);
```
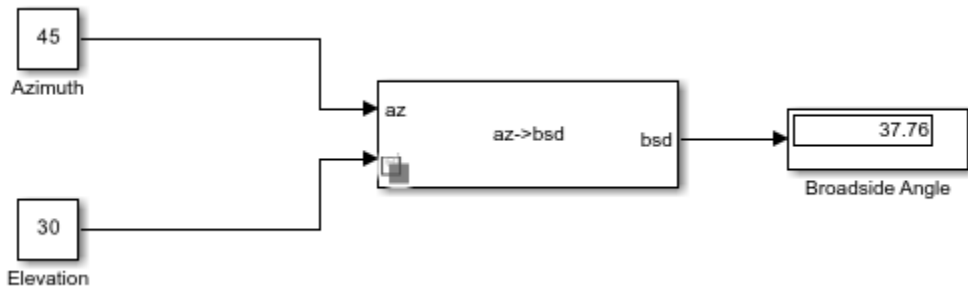


```
close_system(model);
```

# Phase-Shift Beamforming of Plane Wave Signal

Beamform a plane wave arriving at a 10 element ULA of isotropic antenna elements. All the signals are delayed by 100 samples. The operating frequency of the array is 100 MHz.

```
model = 'PhaseShiftBeamformer1Example';
open_system(model);
sim(model);
```
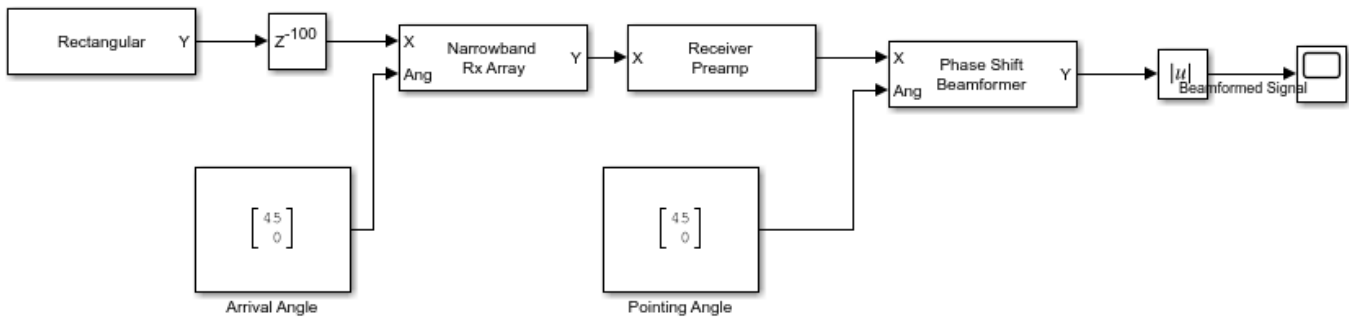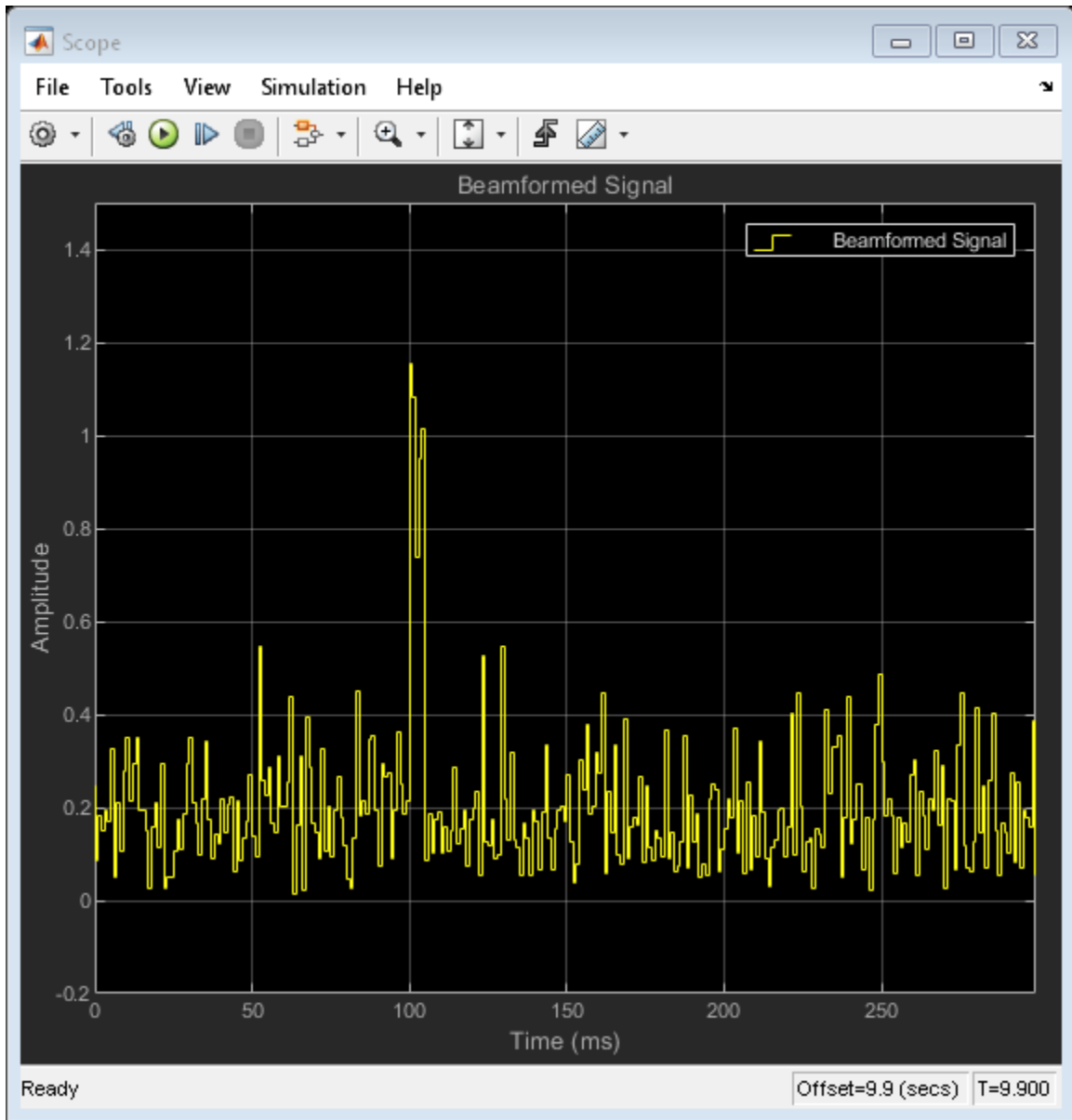
# DSP HDL Toolbox Examples

# FPGA-Based Beamforming in Simulink: Code Generation

This example shows the second half of a workflow to generate HDL code for a beamforming algorithm and verify that the generated code is functionally correct.

The first part of the example, "FPGA-Based Beamforming in Simulink: Algorithm Design" on page 16-9, shows how to develop an algorithm in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA), and how to compare the output of the fixed-point implementation model to that of the corresponding floating-point behavioral model.

This example uses HDL Coder™ to generate HDL code from the Simulink model developed in the first part and verifies the HDL code using HDL Verifier™. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the generated HDL code. The test bench uses ModelSim® for cosimulation to verify the automatically generated HDL code.

The Phased Array System Toolbox™ Simulink blocks model operations on frame-based, floating-point data and provides the behavioral reference model. The example uses this behavioral model to verify the results of the implementation of the algorithm for hardware and the generated HDL code.

HDL Coder generates portable synthesizable Verilog® and VHDL® code for over 300 Simulink blocks that support HDL code generation. Those Simulink blocks operate on serial data using fixed-point arithmetic with delays included to enable pipelining by the synthesis tool.

HDL Verifier helps you test and verify Verilog and VHDL designs for FPGAs, ASICs, and SoCs. This example verifies HDL generated from a Simulink model against a test bench running in Simulink using cosimulation with an HDL simulator.

**Implementation Model**

This tutorial assumes that you have a Simulink model that contains a subsystem with a beamforming algorithm designed using Simulink blocks that use fixed-point arithmetic and support HDL code generation. The "FPGA-Based Beamforming in Simulink: Algorithm Design" on page 16-9 example shows how to create such a model.

Alternatively, if you start with a new model, you can run `hdlsetup` (HDL Coder) to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, open the Model Settings, select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

**Comparing Results of Implementation Model to Behavioral Model**

Run the model created in the "FPGA-Based Beamforming in Simulink: Algorithm Design" on page 16-9 example to display the results. You can run the Simulink model by clicking the Play button or calling the sim command from the MATLAB® command line. Use the Time Scope blocks to compare the output frames visually.

```
modelname = 'SimulinkBeamformingHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```

```
sim(modelname);
```
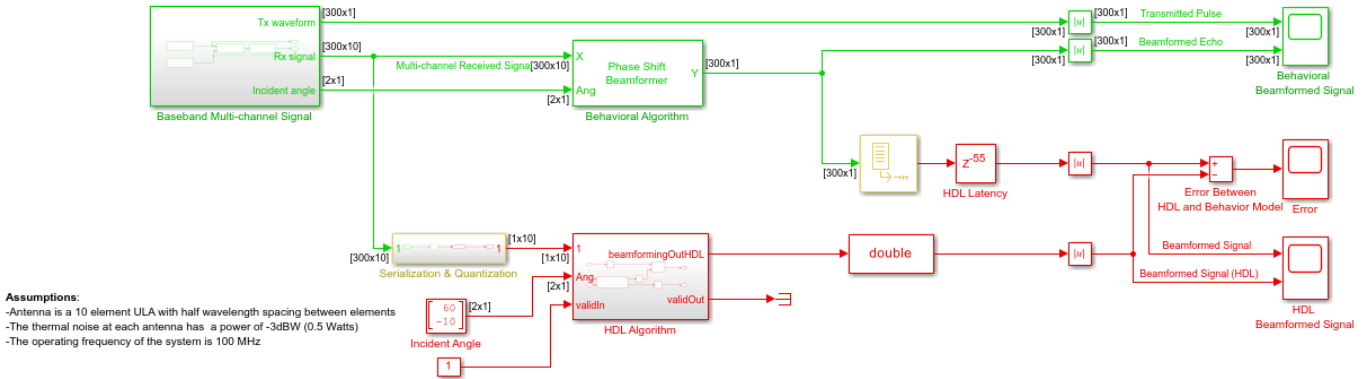
**Model Settings**

Once you verify that your fixed-point implementation model produces the same results as your floating-point behavioral model, you can generate HDL code and test bench. First, set the HDL Code Generation parameters in the Simulink Configuration Parameters dialog. For this example, set the following parameters in Model Settings under HDL Code Generation:

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations except Balance delays.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench and Cosimulation model.

The reason the optimizations are disabled in this model is because some blocks used in our implementation are already HDL-optimized blocks, which could conflict with the HDL Coder optimizations.

**HDL Code Generation and Test Bench Creation**

Next, use HDL Coder to generate HDL code for the `HDL Algorithm` subsystem. For an example of how to generate HDL code, see "Generate HDL Code from Simulink Model" (HDL Coder).

Use these commands to generate HDL code and test bench.

```
makehdl([modelname '/HDL Algorithm']);   % Generate HDL code
makehdltb([modelname '/HDL Algorithm']); % Generate Cosimulation test bench
```

When you call the `makehdl` command, the MATLAB command window shows the amount of delay added during the automatic code generation process. In this case, 24 delays are added which results in an extra delay of 24*1ms = 24ms. The final output is expected to be delayed this additional amount, for a total delay of 79ms.

Because of this extra delay added during code generation, the output of the floating-point behavioral model needs to be delay balanced by adding 24 delays to the original 55. This delay aligns the output of the behavioral model with the implementation model and the cosimulation output.

The process of generating the HDL code and test bench creates a new Simulink model in the working folder, named `gm_<modelname>_mq`. The model contains a ModelSim Cosimulation block.
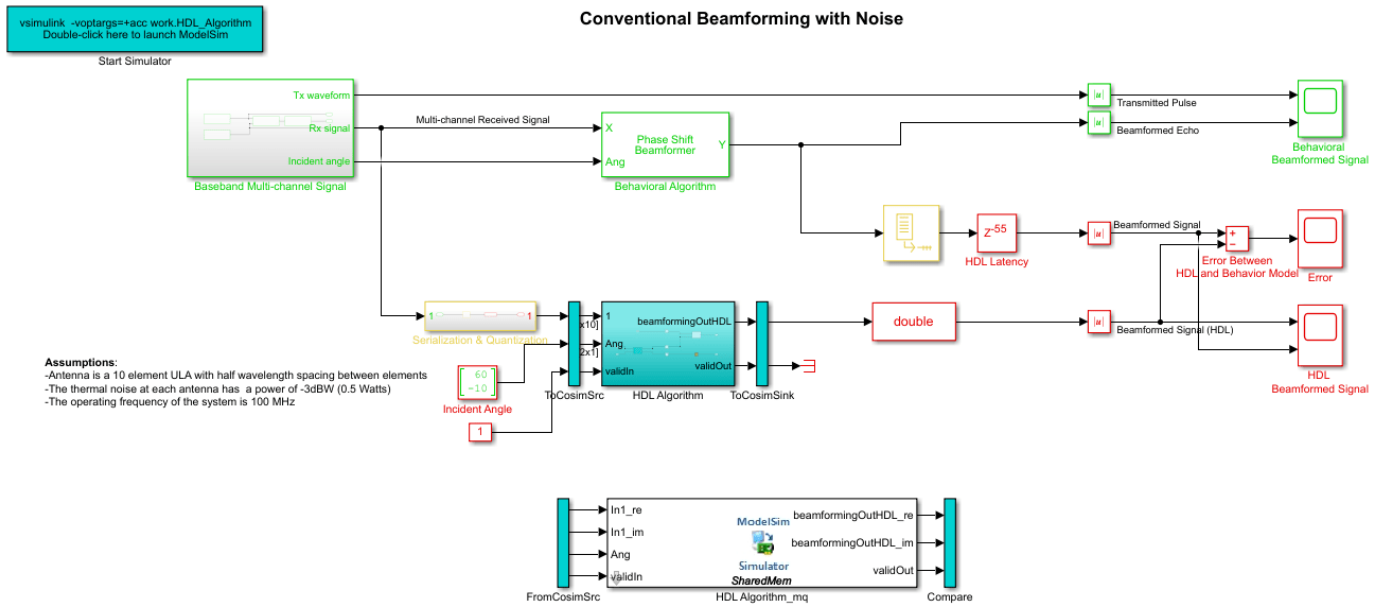
Use these commands to open the test bench model.

```
modelname = ['gm_',modelname,'_mq'];
open_system(modelname);
```

In this new model, change the delay setting in the HDL Latency block to 79 to account for the 24 delays added by the code generation process. Using a delay of 79 ensures that the behavioral model output is time-aligned with the output of the implementation and cosimulation output.

### HDL Code Verification via Cosimulation

Before opening ModelSim make sure that the command to start ModelSim, `vsim`, is on the path of your machine.

To run the cosimulation model, double-click the blue rectangular box in the upper-left corner of the Simulink test model to launch ModelSim.
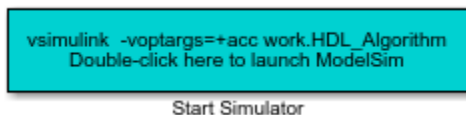


Run the Simulink test bench model to display the simulation results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB command line. The test bench model includes Time Scope blocks to compare the output of the cosimulation performed with ModelSim with the output of the HDL subsystem in Simulink.

If you have ModelSim on your path, use this command to run the test bench.

```
sim(modelname);
```

After starting ModelSim, running the Simulink test bench model will populate Questa Sim with the HDL model's waveforms and Time Scopes in Simulink. The figures show examples of the results in Questa Sim and Simulink scopes.

NOTE: You must restart Questa Sim each time you want to run the Simulink simulation. To restart, call `restart` at the Questa Sim command line. Alternatively, you can quit Questa Sim and re-launch it by double-clicking the blue box in the upper-left corner of the Simulink test bench model.
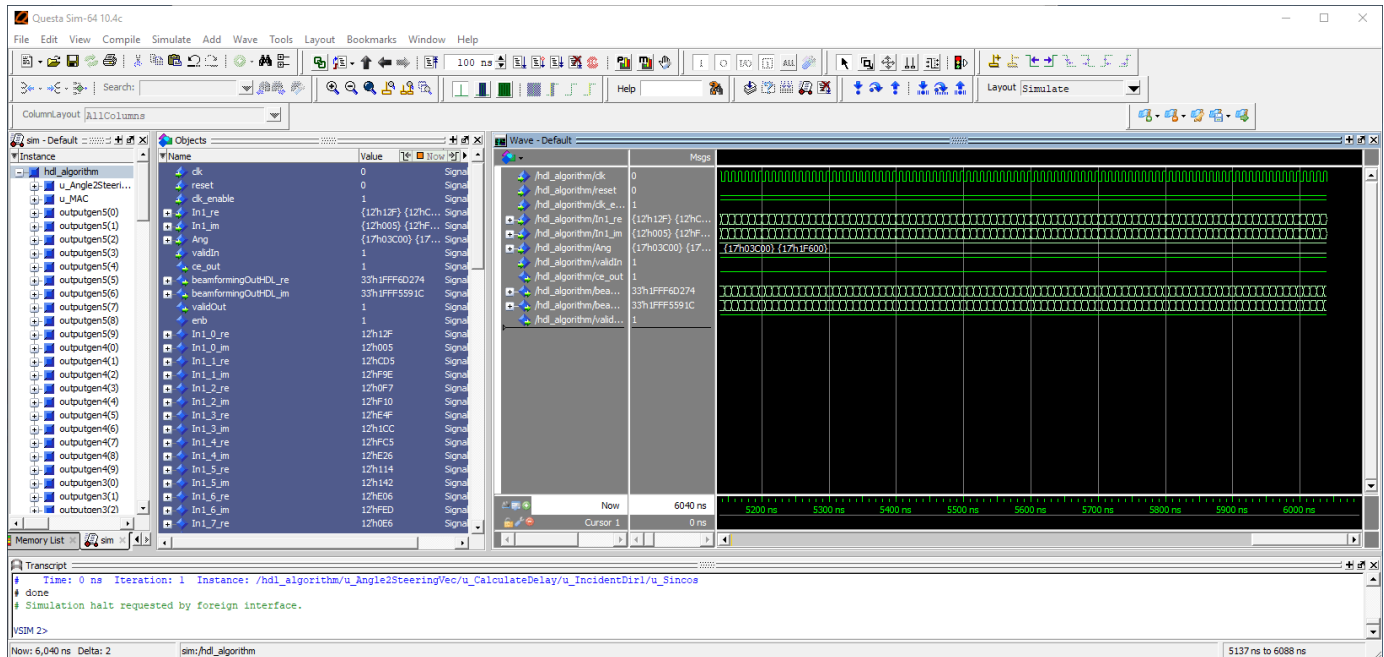


The Simulink scope shows both the cosimulation and HDL model (DUT) producing a 79ms delayed version of the original signal produced by the behavioral model, as expected, with no difference between the two waveforms. The 79ms delay is due to the original 55ms delay added in the `HDL Algorithm` subsystem to enable pipelining by the synthesis tool and an additional 24ms delay due to the delay balancing added by HDL code generation. The additional 24 delays are reported during the code generation step.
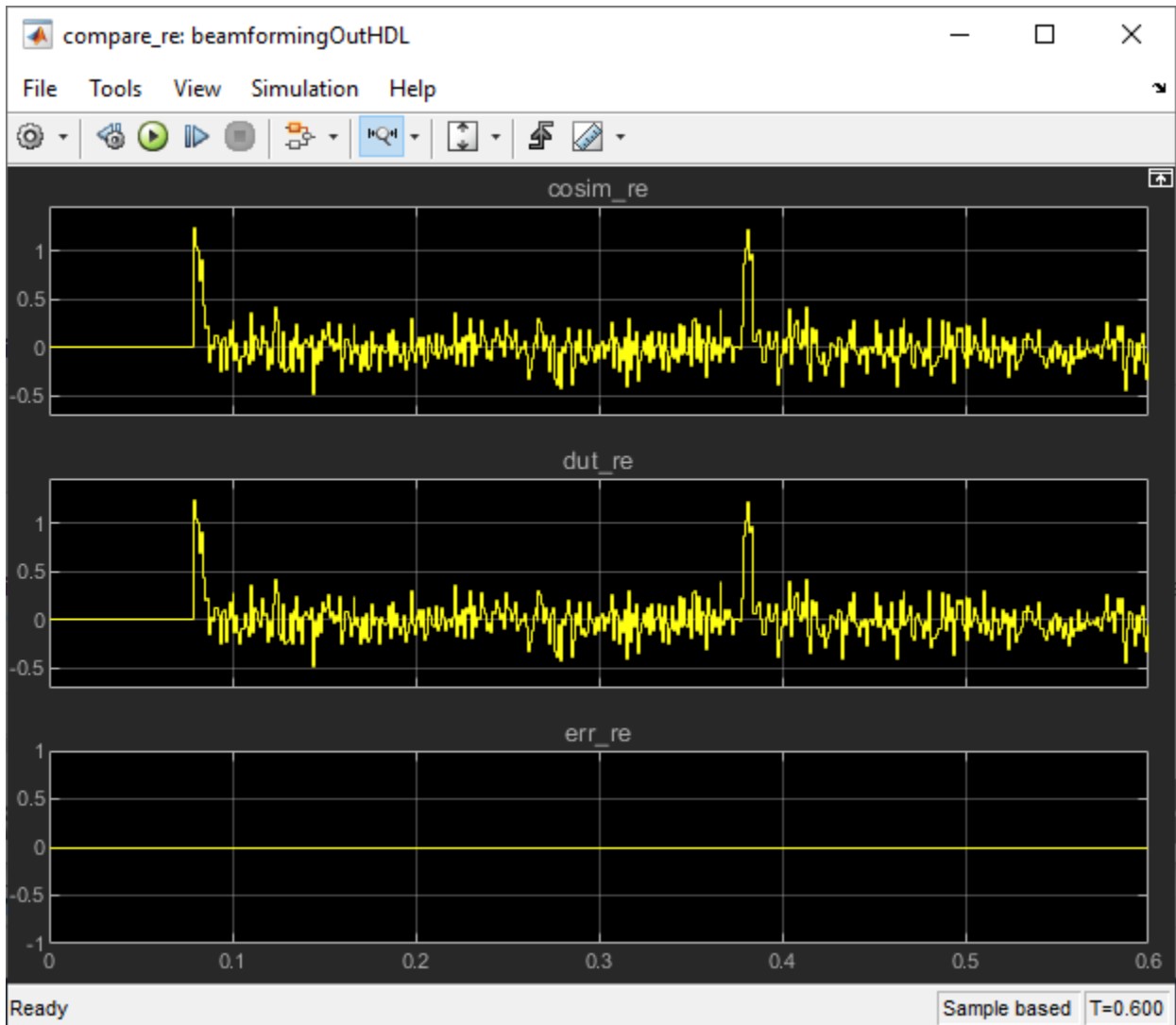
The Simulink scopes comparing the results of the cosimulation can be found in test bench model inside the `Compare` subsystem, which is at the output of the `HDL Algorithm_mq` subsystem.

Use this command to open the subsystem with the scopes.

```
open_system([modelname,'/Compare/Assert_beamformingOutHDL'])
```

**Summary**

This example is the second of a two-part tutorial series on how to automatically generate HDL code for a fixed-point, sample-based beamforming algorithm and verify the generated code in Simulink. The first part of the tutorial, "FPGA-Based Beamforming in Simulink: Algorithm Design" on page 16-9, shows how to develop an algorithm in Simulink suitable for implementation on an FPGA. This example showed how to set up a model to generate the HDL code and a cosimulation test bench for a Simulink subsystem created with blocks that support HDL code generation. It showed how to set up and launch ModelSim to cosimulate the HDL code and compare its output to the output generated by the HDL implementation model.

# FPGA-Based Beamforming in Simulink: Algorithm Design

This example shows the first half of a workflow to develop a beamformer in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA). It also shows how to compare the results of the implementation model with those of a behavioral model.

The second part of the example "FPGA-Based Beamforming in Simulink: Code Generation" on page 16-2 shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the correct results compared to the behavioral model.

This example shows how to implement an FPGA-ready beamformer to match a corresponding behavioral model in Simulink by using the Phased Array System Toolbox™, DSP System Toolbox™, and Fixed-Point Designer™ libraries. To verify the implementation model, this example compares the simulation output of the implementation model with the output of the behavioral model.

The Phased Array System Toolbox is used to design and verify the floating-point functional algorithm, which provides the behavioral reference model. The behavioral model is then used to verify the results of the fixed-point implementation model used to generate HDL code.

Fixed-Point Designer provides data types and tools for developing fixed-point and single-precision algorithms to optimize performance on embedded hardware. You can perform bit-true simulations to observe the impact of limited range and precision without implementing the design on hardware.

**Partitioning Model for FPGA**

There are three key modeling concepts to keep in mind when preparing a Simulink model to target FPGAs:

- **Sample-based processing:** Also commonly referred to as serial processing, sample-based processing is an efficient data processing technique in hardware designs that enables you to tradeoff between resources and throughput.
- **Subsystem targeted for HDL code generation:** In order to generate HDL code from a model, the implementation algorithm must be inside a Simulink subsystem.
- **Time-aligned outputs of behavioral and implementation models:** For comparing the outputs of the behavioral and FPGA implementation models, you must time align their outputs by adding latency to the behavioral model.

**Beamforming Algorithm**

This example shows a phase-shift beamformer as the behavioral algorithm, which is re-implemented in the HDL Algorithm subsystem by using Simulink blocks that support HDL code generation. The beamformer calculates the phase required between each of the ten channels to maximize the received signal power in the direction of the incident angle. The figure shows the Simulink model with the behavioral algorithm and its corresponding implementation algorithm for an FPGA.

```
modelname = 'SimulinkBeamformingHDLWorkflowExample';
open_system(modelname);
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```

**Conventional Beamforming with Noise**
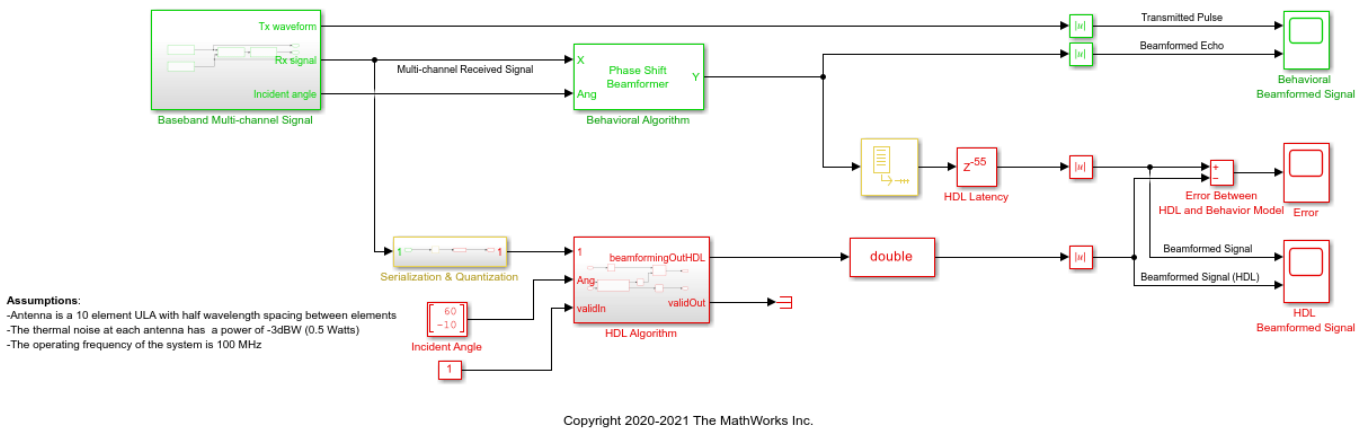


Copyright 2020-2021 The MathWorks Inc.

The Simulink model has two branches. The top branch is the behavioral, floating-point model of the algorithm and the bottom branch is the functionally-equivalent fixed-point version using blocks that support HDL code generation. Besides plotting the output of both branches to compare the two, this example also calculates and plots the difference, or error, between both outputs.

The model has a Delay ($Z^{-55}$) block at the output of the behavioral algorithm. This delay is necessary because the implementation algorithm uses 55 delays to implement pipelining which creates latency that needs to be accounted for. Accounting for this latency is called delay balancing and is necessary to time-align the output between the behavioral model and the implementation model to make it easier to compare the results.

**Multi-Channel Receive Signal**

To synthesize a received signal at the phased array antenna, the model includes a subsystem that generates a multi-channel signal. The `Baseband Multi-channel Signal` subsystem models a transmitted waveform and the received target echo at the incident angle captured via a 10-element antenna array. The subsystem also includes a receiver pre-amp model to account for receiver noise. This subsystem generates the input stimulus for our behavioral and implementation models.

```
open_system([modelname '/Baseband Multi-channel Signal']);
```
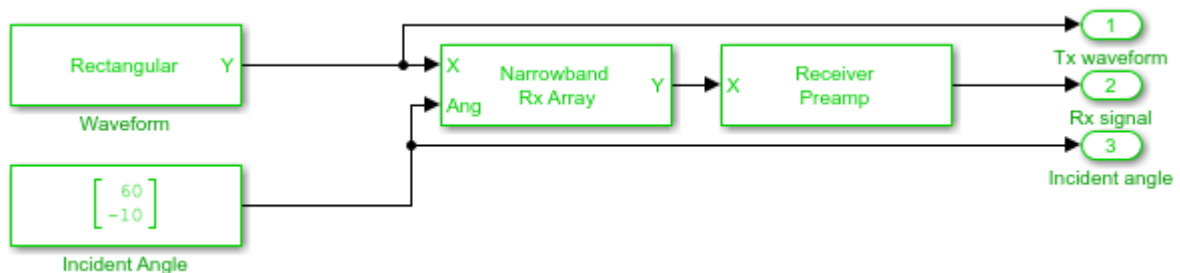


**Serialization and Quantization**

The model includes a `Serialization & Quantization` subsystem which converts floating-point, frame-based signals to fixed-point, sample-based signals necessary for modeling streaming data in

hardware. This model uses scalar streaming processing because this algorithm runs slower than 400 MHz. The hardware implementation can be optimized for resources rather than for higher throughput.

```
open_system([modelname '/Serialization & Quantization']);
set_param(modelname,'SimulationCommand','update')
```
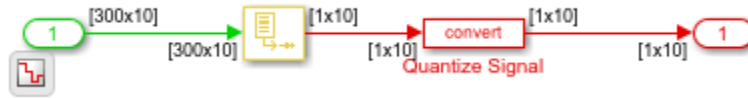


The input signal to the serialization subsystem has 10 channels with 300 samples per channel or a 300x10 size signal. The subsystem serializes, or unbuffers, the signal producing a sample-based signal that's 1x10, i.e., one sample per channel, which is then quantized to meet the requirements of the system.

The output data type of the Quantize Signal block is `fixdt(1,12,9)`, which is a signed value with 12-bit word length and 9-bit fraction length precision. The word length is 12 bits because the design is targeted to a Xilinx® Virtex®-7 FPGA which is connected to a 12-bit ADC. The fraction length accommodates the maximum range of the input signal.

**Designing the Implementation Subsystem**

The `HDL Algorithm` subsystem, which is targeted for HDL code generation, implements the beamformer, which was designed using Simulink blocks that support HDL code generation.

The `Angle2SteeringVec` subsystem calculates the signal delay at each antenna element of a Uniform Linear Array (ULA). The delay is then fed to a multiply and accumulate (MAC) subsystem to perform beamforming.

```
open_system([modelname '/HDL Algorithm']);
```
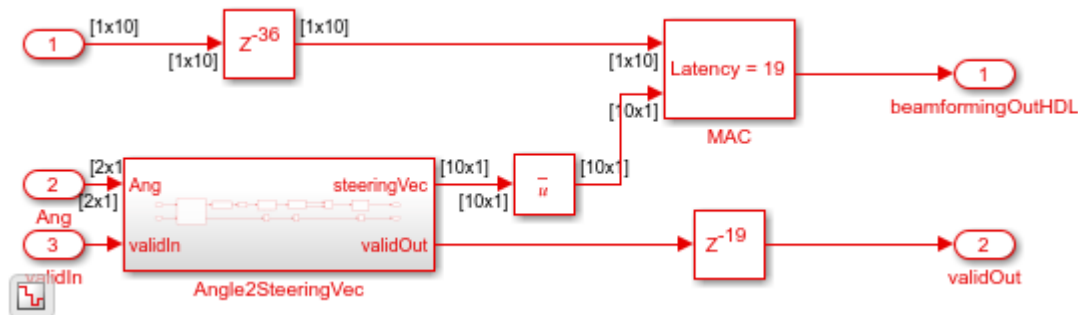


The algorithm in the `HDL Algorithm` subsystem is functionally equivalent to the phase-shift beamforming behavioral algorithm but can generate HDL code. There are three main differences that enable this subsystem to generate efficient HDL code.

1   The design processes inputs serially, i.e. using sample-based processing.
2   The subsystem uses fixed-point data types for any calculations.
3   The implementation includes Delays that enable pipelining by the HDL synthesis tool.

To ensure proper clock timing, any delay added to one branch of the implementation model must be matched to all other parallel branches as seen above. The `Angle2SteeringVec` subsystem, for

example, has 36 delays; therefore, the top branch of the `HDL Algorithm` subsystem includes a delay of 36 samples right before the `MAC` subsystem. Likewise, the `MAC` subsystem uses 19 delays, which must be balanced by adding 19 delays to the output of the `Angle2SteeringVec` subsystem. This figure shows the inside of the `MAC` subsystem to illustrate the 19 delays.

```
open_system([modelname '/HDL Algorithm/MAC']);
set_param(modelname,'SimulationCommand','update')
```
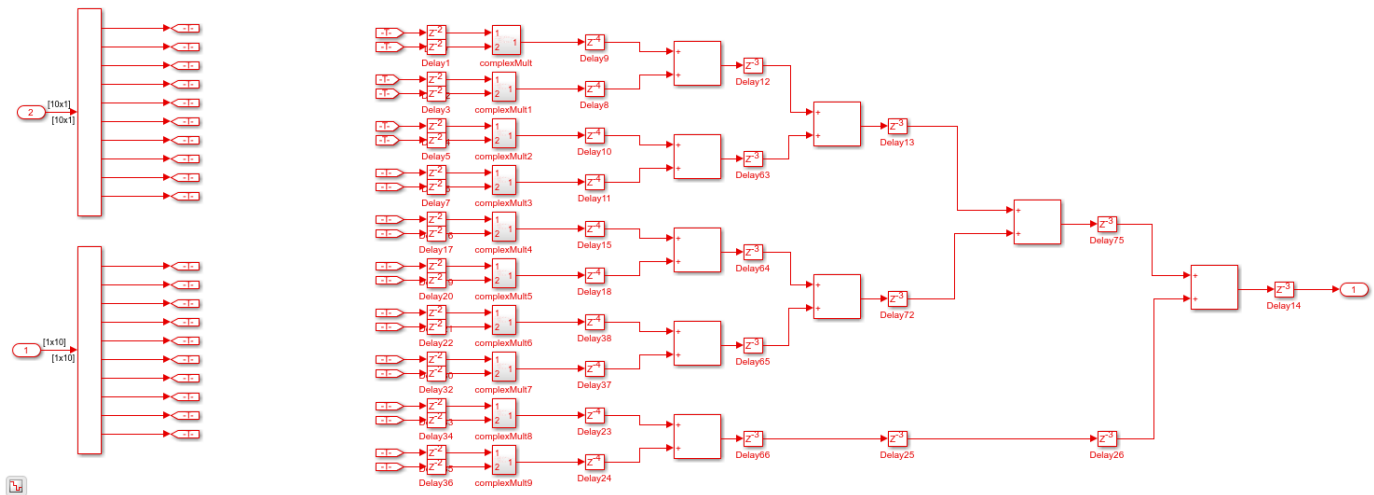


The bottom branch of the `MAC` subsystem has a $Z^{-2}$ block, followed by the complex multiply block which contains a $Z^{-1}$, then a $Z^{-4}$ block, followed by 4 delay blocks of $Z^{-3}$ for a total of 19 delays. The delay values are defined in the PreLoadFcn callback in Model Properties.

**Calculating the Steering Vector**

The `Angle2SteeringVec` subsystem calculates the steering vector from the signal's angle of arrival. It first calculates the signal's arrival delay at each sensor by matrix-multiplying the antenna element position in the array by the incident direction of the signal. The delays are then passed to the `SinCos` subsystem which calculates the sine and cosine trigonometric functions by using the simple and efficient CORDIC algorithm.

```
open_system([modelname '/HDL Algorithm/Angle2SteeringVec']);
```
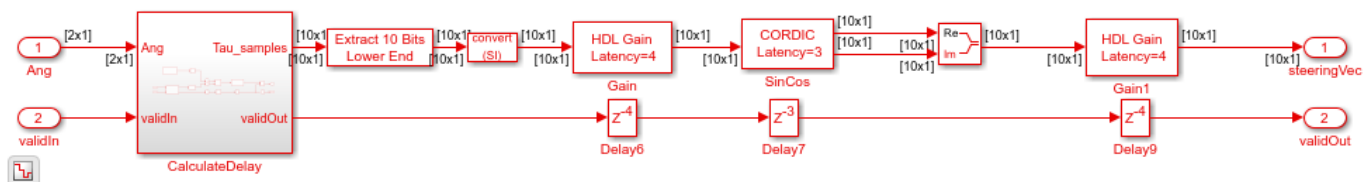


Because this design consists of a 10 element ULA spaced at half-wavelength, the antenna element position is based on the spacing between each antenna element measured outwardly from the center of the antenna array. Define the spacing between elements as a vector of 10 numbers ranging from -6.7453 to 6.7453, i.e., with a spacing of 1/2 wavelength, which is 2.99/2. In fixed-point arithmetic, the data type used for the element spacing vector is `fixdt(1,8,4)`, i.e., a signed value with 8-bit word length and 4-bit fraction length.

**Deserialization**

To compare your sample-based fixed-point implementation with the floating-point frame-based behavioral design you need to deserialize the output of the implementation subsystem and convert it to a floating-point data type. Alternatively, you can compare the results directly with sample-based signals but then you must unbuffer the output of the behavioral model to match the sample-based signal output from the implementation algorithm, as shown in this figure.
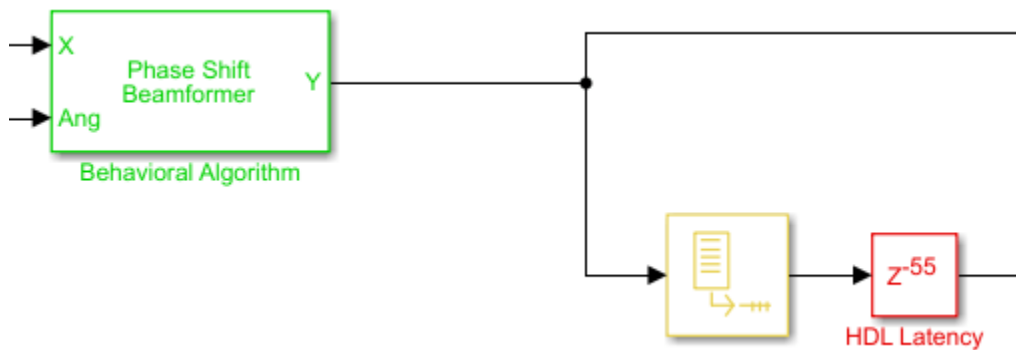


In this case, you only need to convert the output of the `HDL Algorithm` subsystem to floating-point by setting the output of the Data Type Conversion block to `double`.

**Comparing Output of HDL Model to Behavioral Model**

Run the model to display the results. You can run the Simulink model by clicking the Play button or calling the sim command from the MATLAB® command line. Use the scopes to compare the outputs visually.

```
sim(modelname);
```
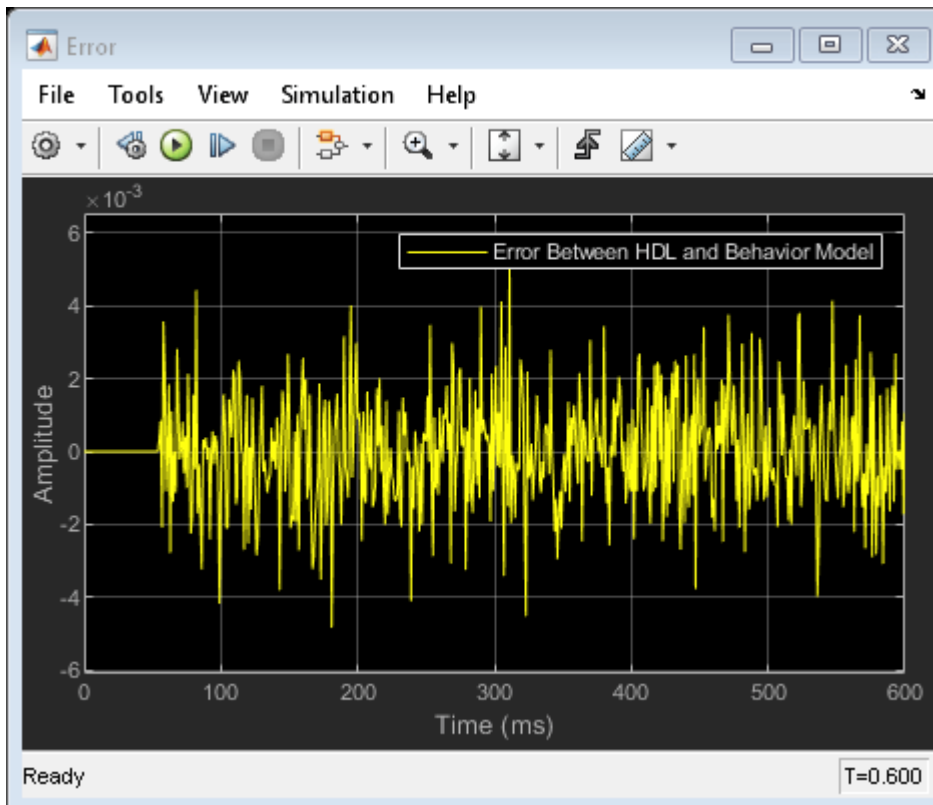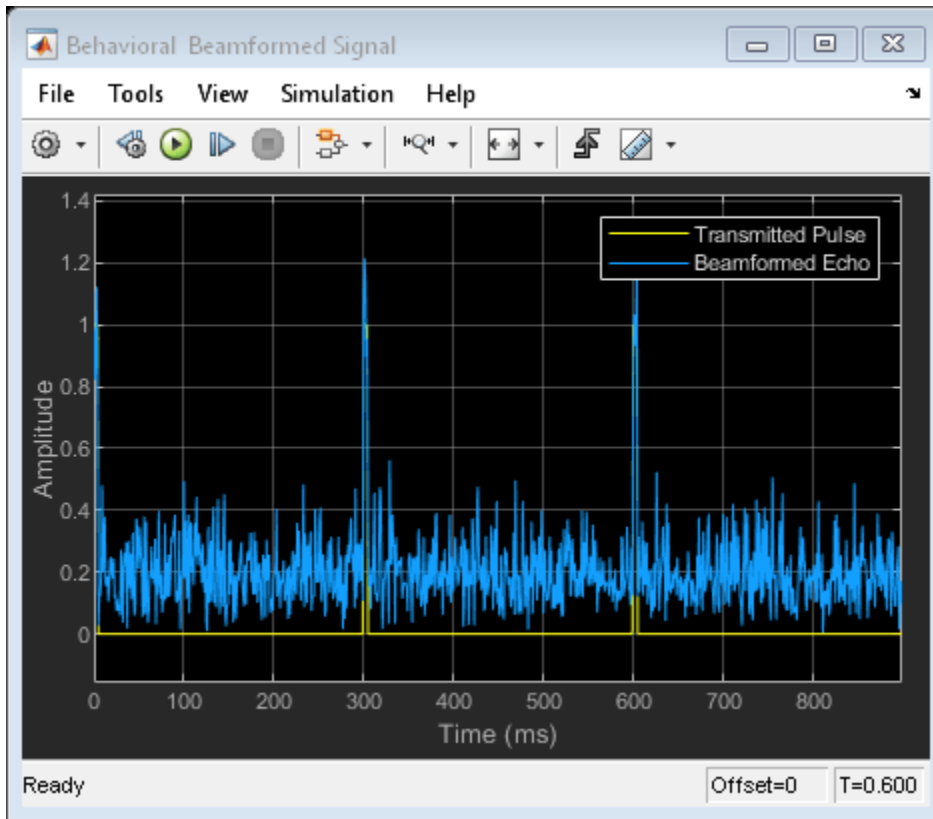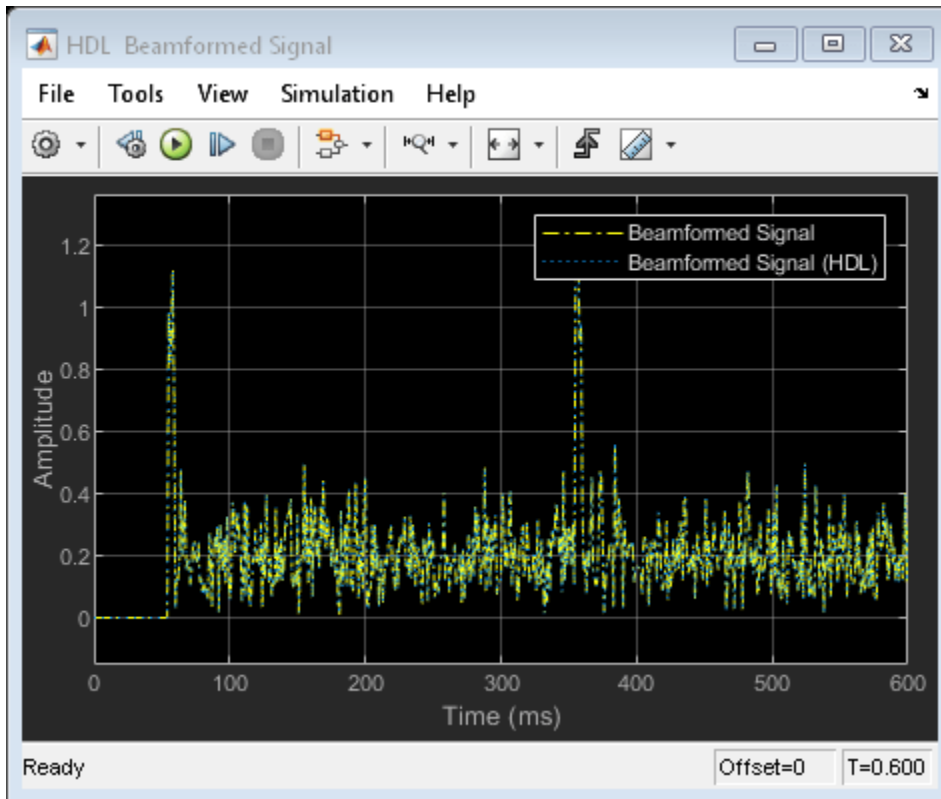
The Time Scope of the Beamformed Signal and Beamformed Signal (HDL) shows that the two signals are nearly identical. The error is on the order of 10^-3 in the Error scope. This result shows that the HDL Algorithm subsystem is producing the same output as the behavioral model within quantization error. This verification is an important first step before generating HDL code.

Because the HDL model used 55 delays, the scope titled HDL Beamformed Signal is delayed by 55ms when compared to the original transmitted or beamformed signal shown on the Behavioral Beamformed Signal scope.

**Summary**

This example is the first of a two-part tutorial series on how to design an FPGA-ready algorithm, automatically generate HDL code, and verify the HDL code in Simulink. This example showed how to use blocks from the Phased Array System Toolbox to create a behavioral model to serve as a golden reference, and how to create a subsystem for hardware implementation using Simulink blocks that support HDL code generation. It also compared the output of the implementation model to the output of the corresponding behavioral model to verify that the two algorithms are functionally equivalent.

Once you verify that your implementation algorithm is functionally equivalent to your golden reference, you can use HDL Coder™ for "HDL Code Generation from Simulink" (HDL Coder) and HDL Verifier™ to "Generate a Cosimulation Model" (HDL Coder) test bench.

The second part of this two-part tutorial series "FPGA-Based Beamforming in Simulink: Code Generation" on page 16-2 shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the same results as the floating-point behavioral model as well as the fixed-point implementation model.

# FPGA-Based Monopulse Technique: Algorithm Design

This example shows the first half part of a workflow to develop a monopulse technique where the signal is downconverted using digital downconversion (DDC). The model in this example is suitable for implementation on FPGA. This example focuses on the design of monopulse technique to estimate the azimuth and elevation of an object.

The second part of the example, "FPGA-Based Monopulse Technique: Code Generation" on page 16-27, shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the correct results compared to the behavioral model. The entire algorithm is designed using fixed-point data types.

The example shows how to design an FPGA-ready monopulse technique to match a corresponding behavioral model in Simulink® using the Phased Array System Toolbox™, DSP HDL Toolbox™, and Fixed-Point Designer™. To verify the implementation model, the example compares the simulation output of the implementation model with the output of the behavioral model.

The Phased Array System Toolbox provides the floating-point behavioral model for the monopulse technique as a `phased.MonopulseFeed` System object™. DSP HDL Toolbox provides the FIR filter essential for the downconversion filtering.

Fixed-Point Designer provides data types and tools for developing fixed-point and single-precision algorithms to optimize performance on embedded hardware. Bit-true simulations can be performed to observe the impact of limited range and precision without implementing the design on hardware.

Monopulse is a technique where the received echoes from different elements of an antenna are used to estimate the direction of arrival (DOA) of a signal. This direction helps estimate the location of an object. The example uses DSP HDL Toolbox and Fixed-Point Designer to design the algorithm. This technique uses four beams to measure the angular position of the target. All the four beams are generated simultaneously and the difference of azimuth and elevation is achieved in a single pulse, hence, the name monopulse.

**Designing the Subsystem**

The algorithm is implemented by using Simulink® blocks that support HDL code generation. The model assumes that the signal is received from the 4-element uniform rectangular array (URA), so, the model has 4 sinusoids as inputs. Assuming a 4-element URA, the model is comprised of 4 receive channels from each of the elements of the URA. Once the signals are converted to the digital domain, DDC blocks ensure that the frequency of the received signal is lowered to reduce the sample rate for processing. The block diagram shows the subsystem which consists of the following modules.

1  Multi-Channel Input Signal
2  Digital Down-Conversion
3  Monopulse Sum and Difference Channels

```
modelname = 'SimulinkDDCMonopulseHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```
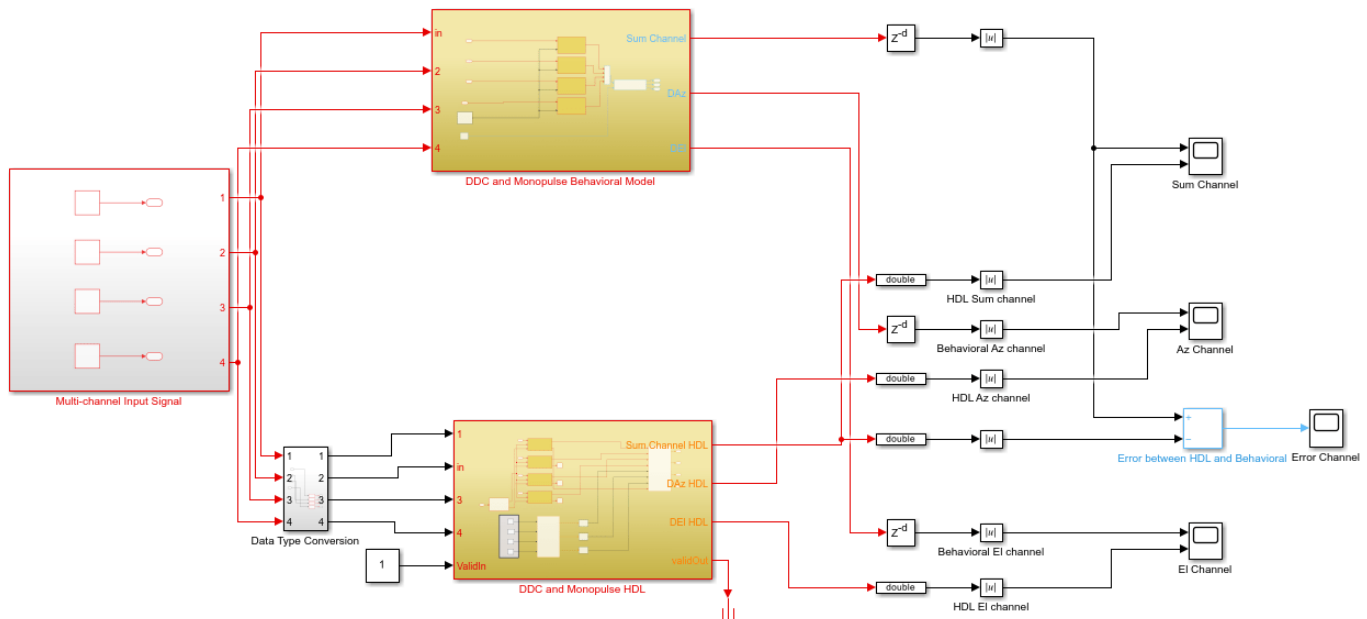
Copyright 2020-2021 The MathWorks Inc.

The Simulink model has two branches. The top branch is the behavioral floating-point model of the monopulse technique and digital downconversion algorithm and the bottom branch is the functionally equivalent fixed-point version using blocks that support HDL code generation. Apart from plotting the output of both branches to compare the two, the difference, or error, between sum channel of both the outputs has also been calculated and plotted.

The output of the behavioral model has a delay ($Z^{-220}$) block. This delay is needed because the implementation algorithm uses 220 delays to enable pipelining which creates latency that needs to be accounted for. This latency is necessary to time-align the output between the behavioral model and the implementation model.

### Digital DownConversion (DDC)

The `DDC and Monopulse HDL` subsystem shows how the received signal sampled at 80 MHz and nearly 15 MHz carrier frequency is downconverted to baseband via the DDC and then passed to the monopulse sum and difference subsystem. A DDC module is a combination of a numerically controlled oscillator (NCO) and a set of low-pass filters. The NCO block provides the signal to mix and demodulate the incoming signal.

```
open_system([modelname '/DDC and Monopulse HDL']);
```
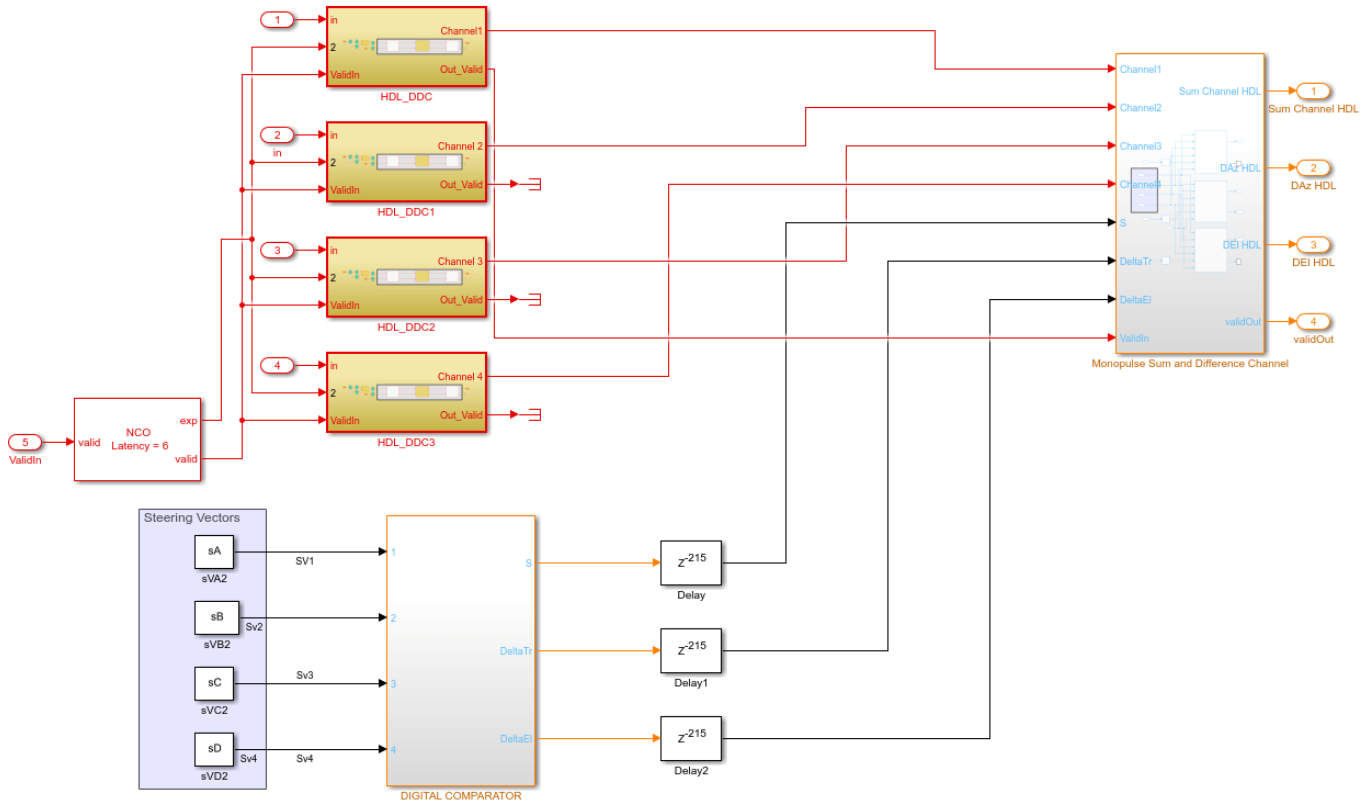
The delay of 215 ms at the sum and difference output of the steering vectors in the implementation subsystem compensates for the delay of the downconversion chain.

A DDC also contains a set of lowpass filters as shown in the figure. Once mixed, the lowpass filtering of the mixed signal is required to eliminate the high frequency components. This example uses a cascaded filter chain for lowpass filtering. The NCO generates the high-accuracy sinusoid for the mixer. The NCO block has a latency of 6 cycles. This signal is mixed with the incoming signal and is converted from a higher frequency to a relatively lower frequency as it progresses through the filter stages.

```
open_system([modelname '/DDC and Monopulse HDL/HDL_DDC']);
```
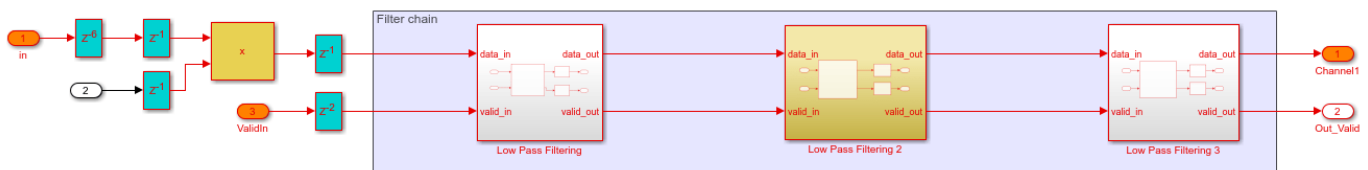


In this example, the incoming signal has a carrier frequency of 15 MHz and is sampled at 80 MHz. The downconversion process brings the sampled signal down to a few kHz. The coefficients for the lowpass FIR filters are designed using `filterBuilder`. The coefficient values must be chosen to satisfy the required passband criteria.
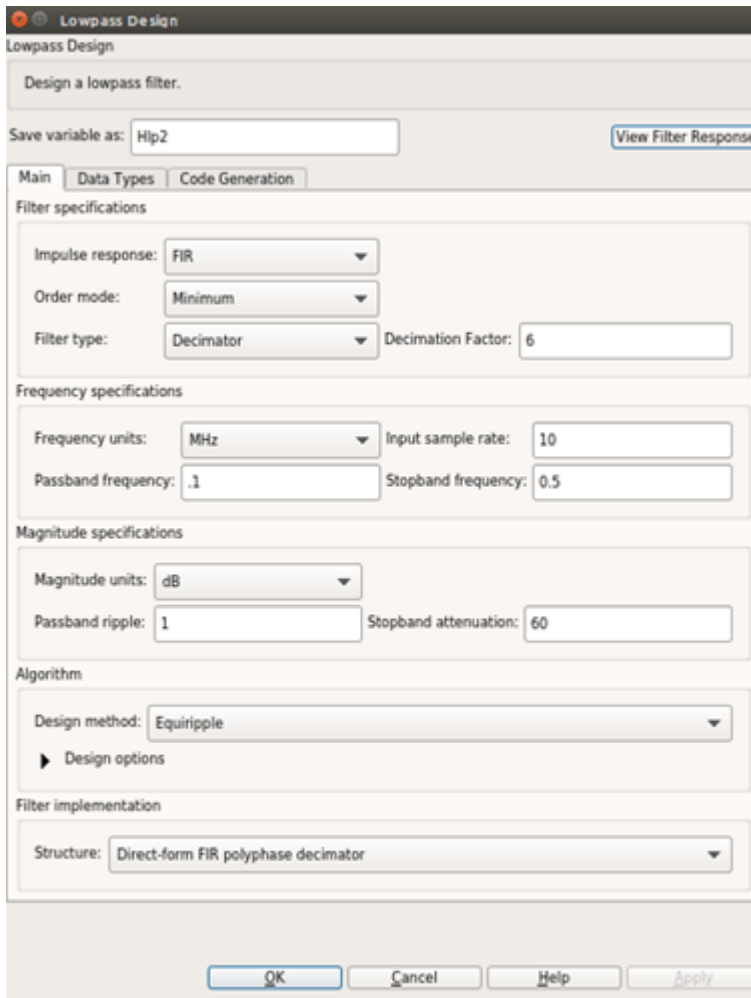
Once generated, the coefficients are used to configure the FIR Filter block.

**Monopulse Sum and Difference Channels**

The monopulse algorithm must also generate a steering vector for different elements. The steering vectors have been generated for an incident angle of azimuth 30 degrees and elevation 20 degrees. The steering vectors are passed to the digital comparator to provide the desired sum and difference channel outputs. The downconverted signal is then multiplied by the conjugate of these vectors as shown in the figure. By processing the sum and difference channels, the DOA of the received signal can be found. The digital comparator that compares the steering vectors for the different elements of the antenna array is shown.

```
open_system([modelname '/DDC and Monopulse HDL/DIGITAL COMPARATOR']);
```
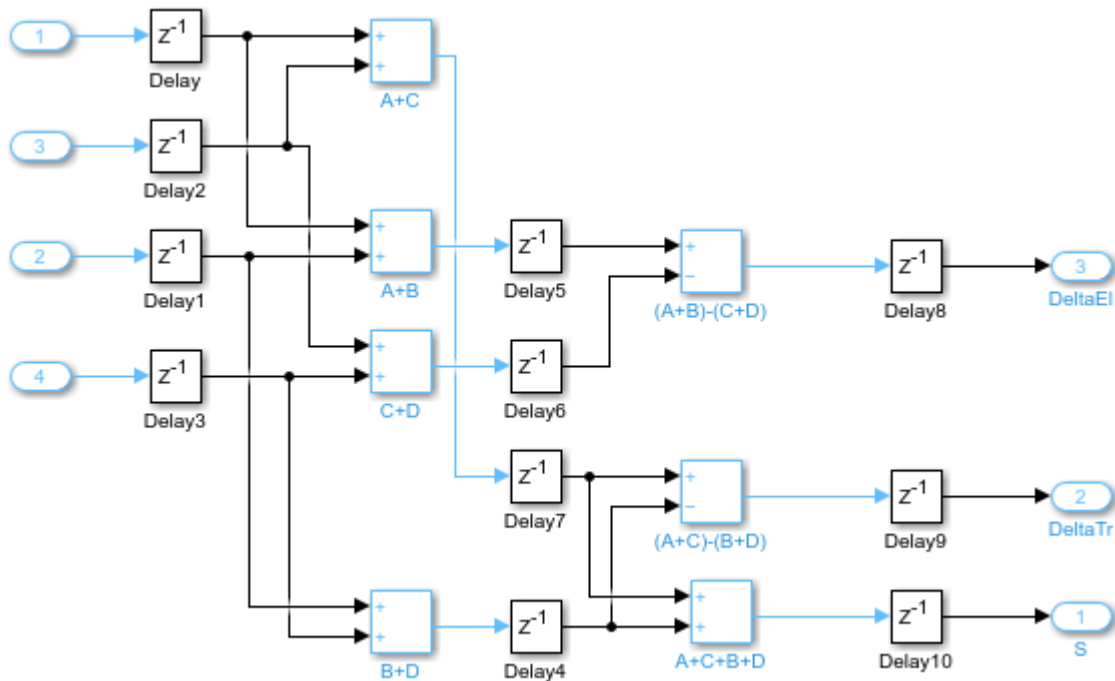
In the figure, the digital comparator takes the steering vectors and computes the sum and difference of the different steering vectors sVA, sVB, sVC and sVD respectively. You can also calculate the steering vectors by using the `phased.SteeringVector` System object or you can generate them using method similar to the one shown in the "FPGA-Based Beamforming in Simulink: Algorithm Design" on page 16-9. Once the sum and difference of various steering vectors corresponding to each element of the array has been done, the calculation of sum and difference channels for corresponding azimuth and elevation angles are performed. From the Sum and Difference Monopulse subsystem, 3 signals are obtained, namely the sum, the azimuth difference, and the elevation difference. The entire arithmetic is performed in fixed-point data types. Open the monopulse sum and difference channel subsystem by using this command.

```
open_system([modelname '/DDC and Monopulse HDL/Monopulse Sum and Difference Channel']);
```
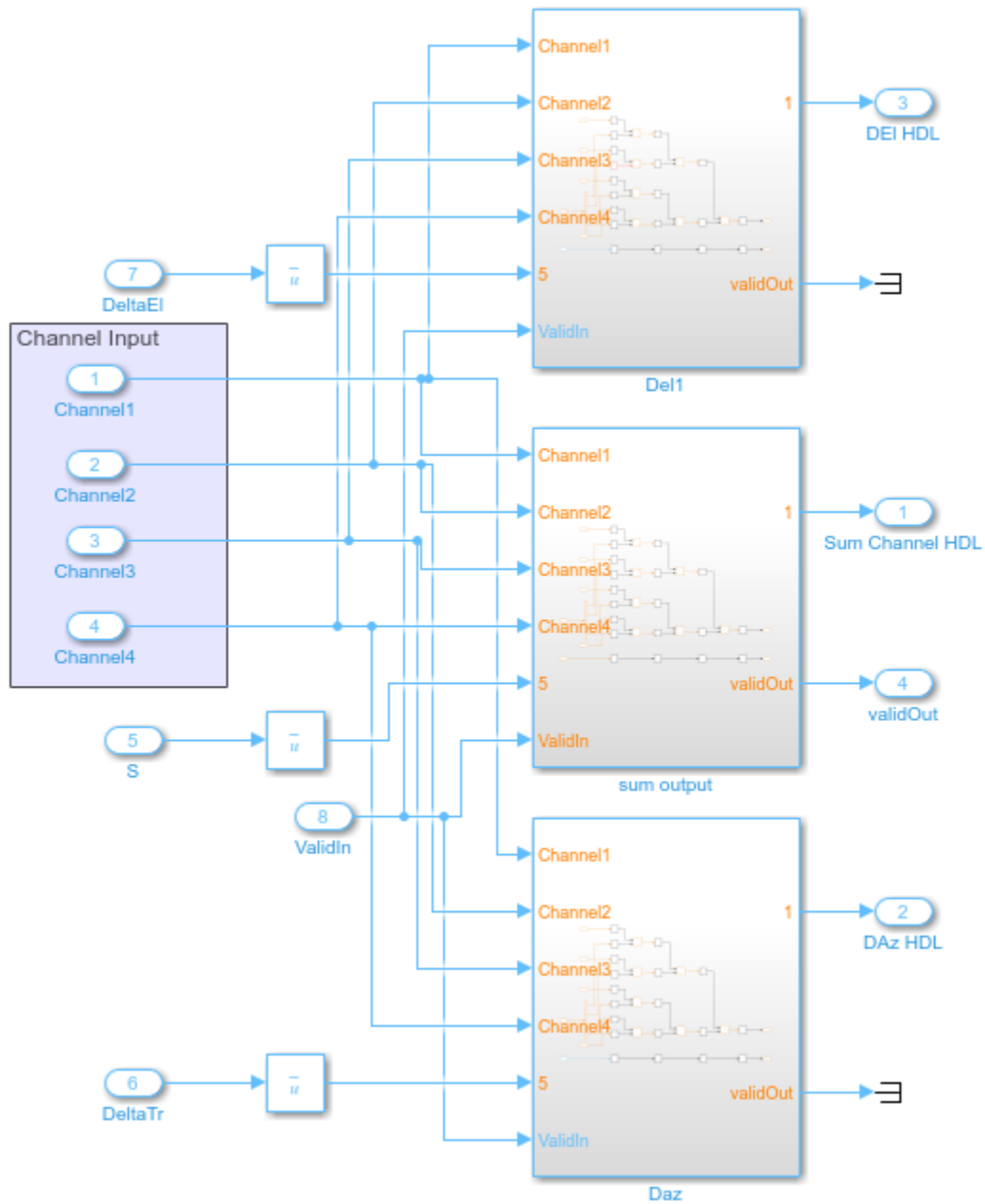
**Comparing Results of Implementation Model to Behavioral Model**

To compare results of the implementation model to the behavioral model, run the model created to display the results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB® command line. Use Scope blocks to compare the output frames.

```
sim(modelname);
```
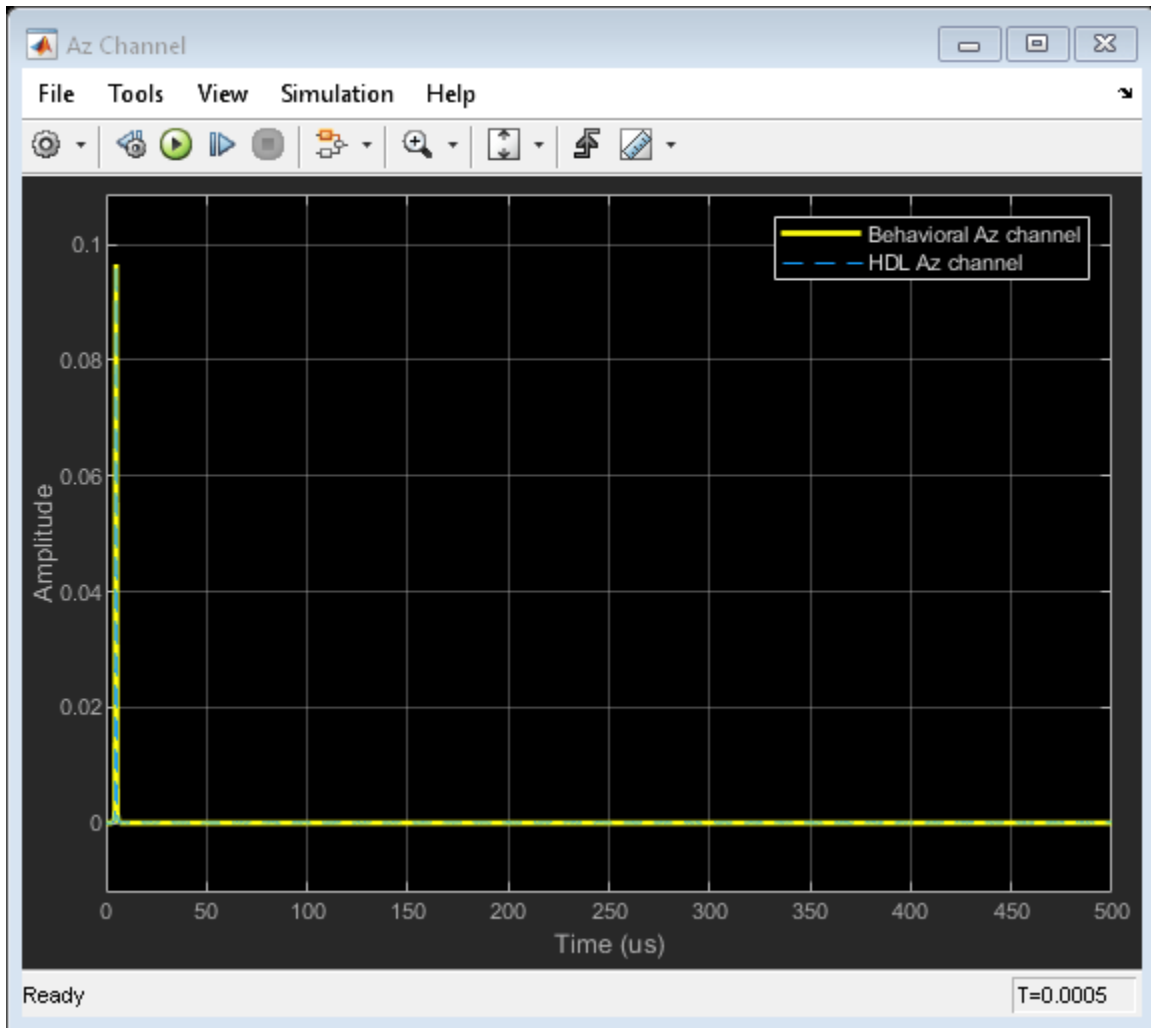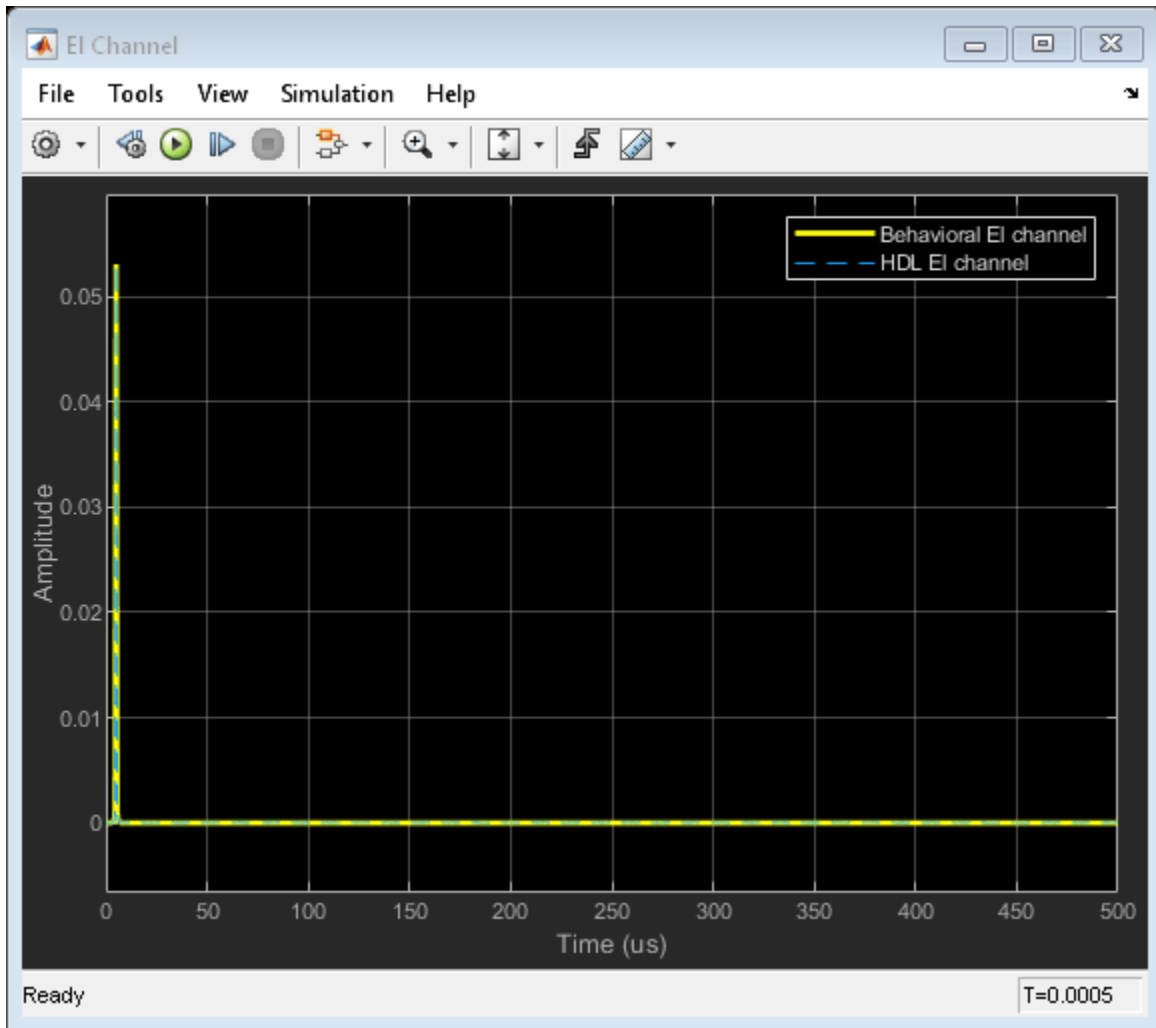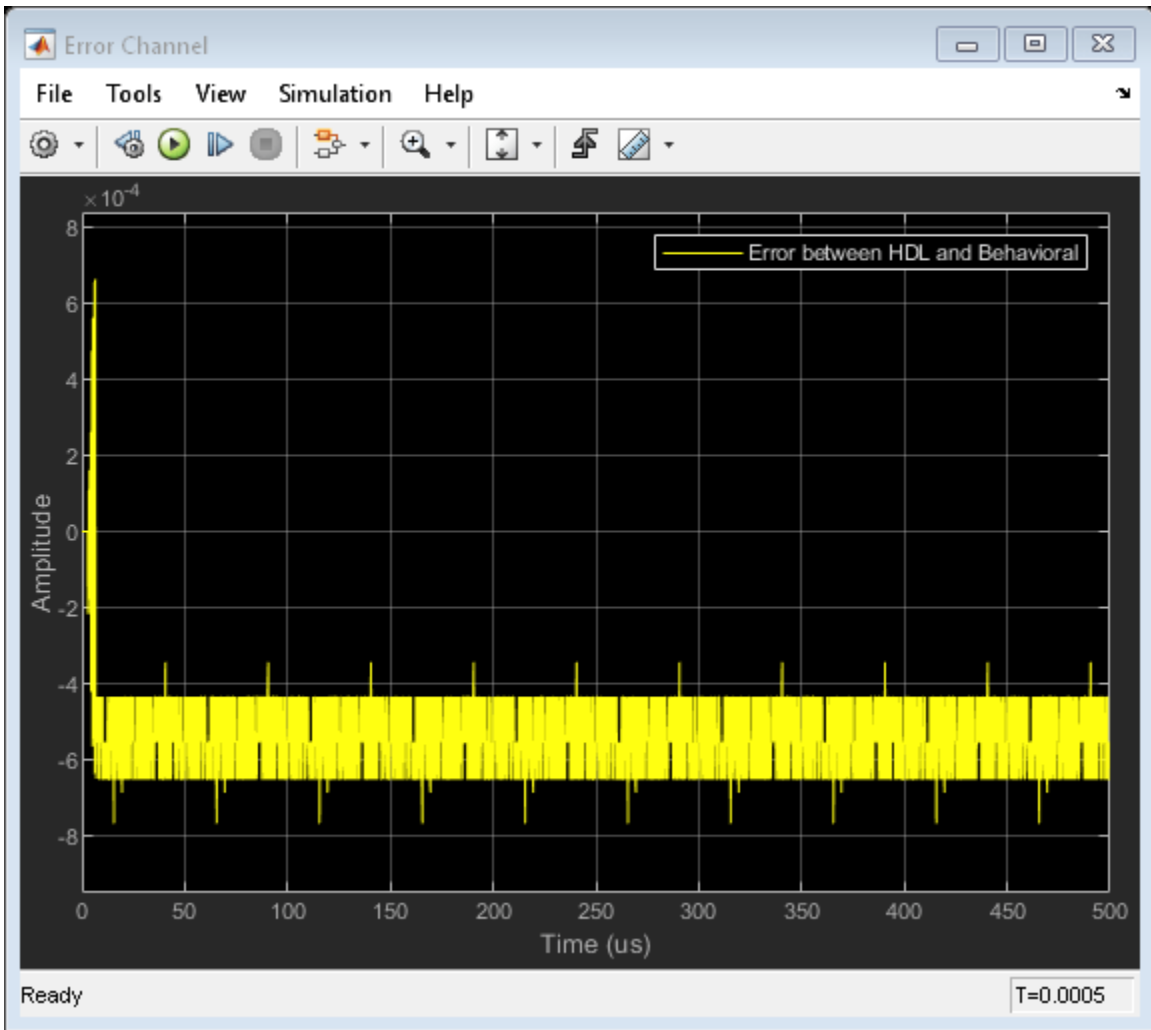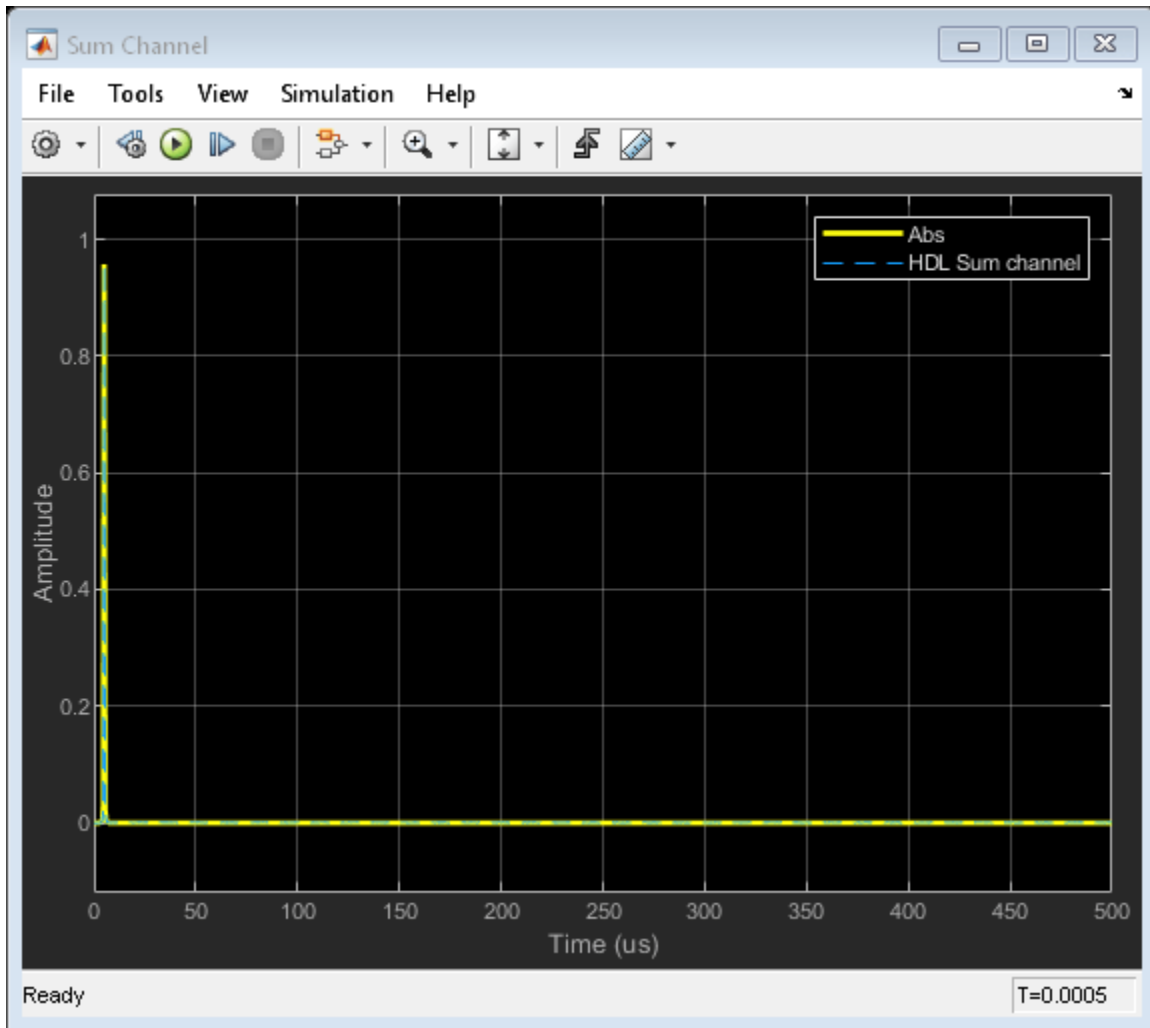
The plots show the output from sum and difference channels. These channels can be fed to an estimator to indicate the angle/direction of the object.

**Summary**

This example demonstrates how to design an FPGA-ready algorithm, automatically generate HDL code, and verify the HDL code in Simulink. The example illustrates the design of a Simulink model for a DDC and monopulse feed system, and verifies the results with an equivalent behavioral model from the Phased Array System Toolbox as the golden reference. Apart from the behavioral model, the example demonstrates how to create a subsystem for implementation using Simulink blocks that support HDL code generation. It also compared the output of the implementation model to the output of the corresponding behavioral model to verify that the two algorithms are functionally equivalent.

Once the implementation algorithm is functionally verified to be equivalent to golden reference, HDL Coder™ can be used for Simulink to HDL code generation and HDL Verifier™ can be used to "Generate a Cosimulation Model" (HDL Coder) test bench.

The second part of this two-part series shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the same results as the floating-point behavioral model as well as the fixed-point implementation model.

# FPGA-Based Monopulse Technique: Code Generation

This example shows the second half of a workflow to generate HDL code for a monopulse technique and verify that the generated code is functionally correct.

The first example in the workflow, "FPGA-Based Monopulse Technique: Algorithm Design" on page 16-16, shows how to develop an algorithm in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA), and how to compare the output of the fixed-point implementation model to that of the corresponding floating-point behavioral model.

This example uses HDL Coder™ to generate HDL code from the Simulink model developed in part one and verifies the HDL code using HDL Verifier™. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the automatically generated HDL code. The test bench uses ModelSim® for cosimulation to verify the automatically generated HDL code.

The Phased Array System Toolbox™ Simulink blocks model operations on floating-point data and provides the behavioral reference model. This behavioral model is used to verify the results of the implementation model and the automatically generated HDL code.

HDL Coder™ generates portable, synthesizable Verilog® and VHDL® code for Simulink blocks that support HDL code generation.

HDL Verifier lets you test and verify Verilog and VHDL designs for FPGAs, ASICs, and SoCs. This example verifies HDL generated from the Simulink model against a test bench running in Simulink using cosimulation with an HDL simulator.

**Implementation Model**

This example assumes that you have a Simulink model that contains a subsystem with a monopulse technique designed using Simulink blocks that use fixed-point arithmetic and support HDL code generation. "FPGA-Based Monopulse Technique: Algorithm Design" on page 16-16 shows how to create such a model.

To start with a new model, run the `hdlsetup` (HDL Coder) function to configure the Simulink model for HDL code generation. Open the Model Settings to configure the Simulink model for test bench creation needed for verification. Select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

**Comparing Results of Implementation Model to Behavioral Model**

Run the model created in the "FPGA-Based Monopulse Technique: Algorithm Design" on page 16-16 example to display the results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB® command line. Use the Time Scope blocks to compare the output frames visually.

```
modelname = 'SimulinkDDCMonopulseHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);

sim(modelname);
```
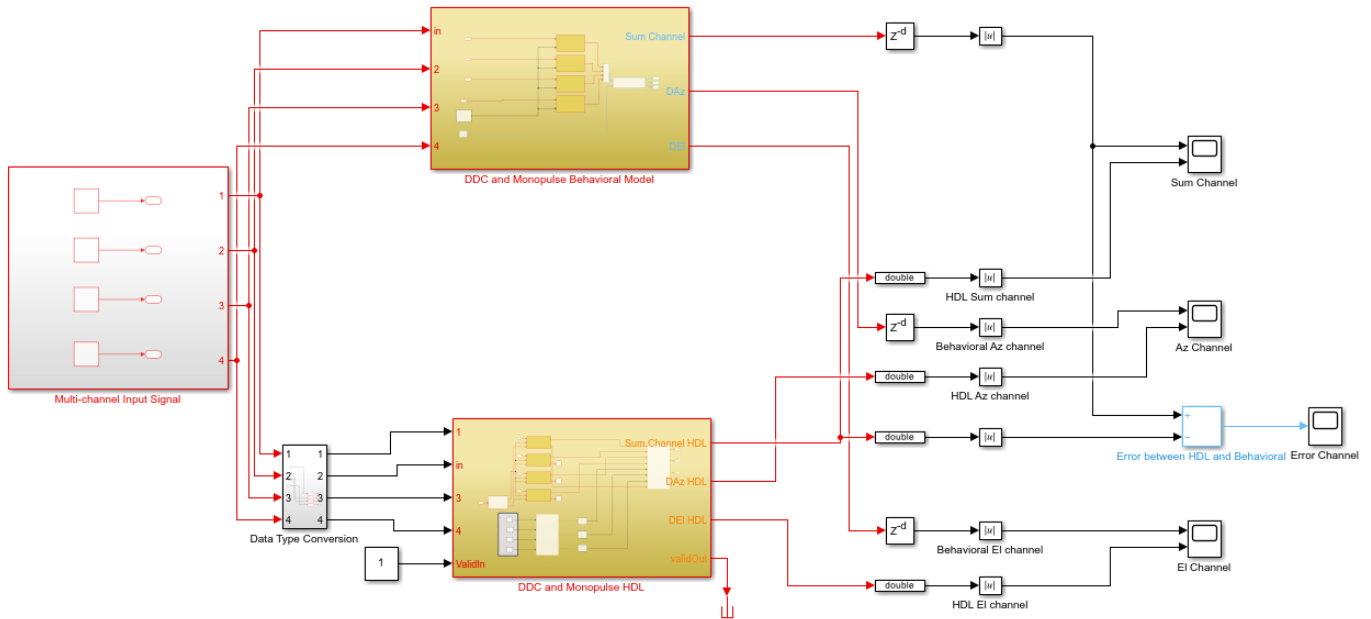
Copyright 2020-2021 The MathWorks Inc.
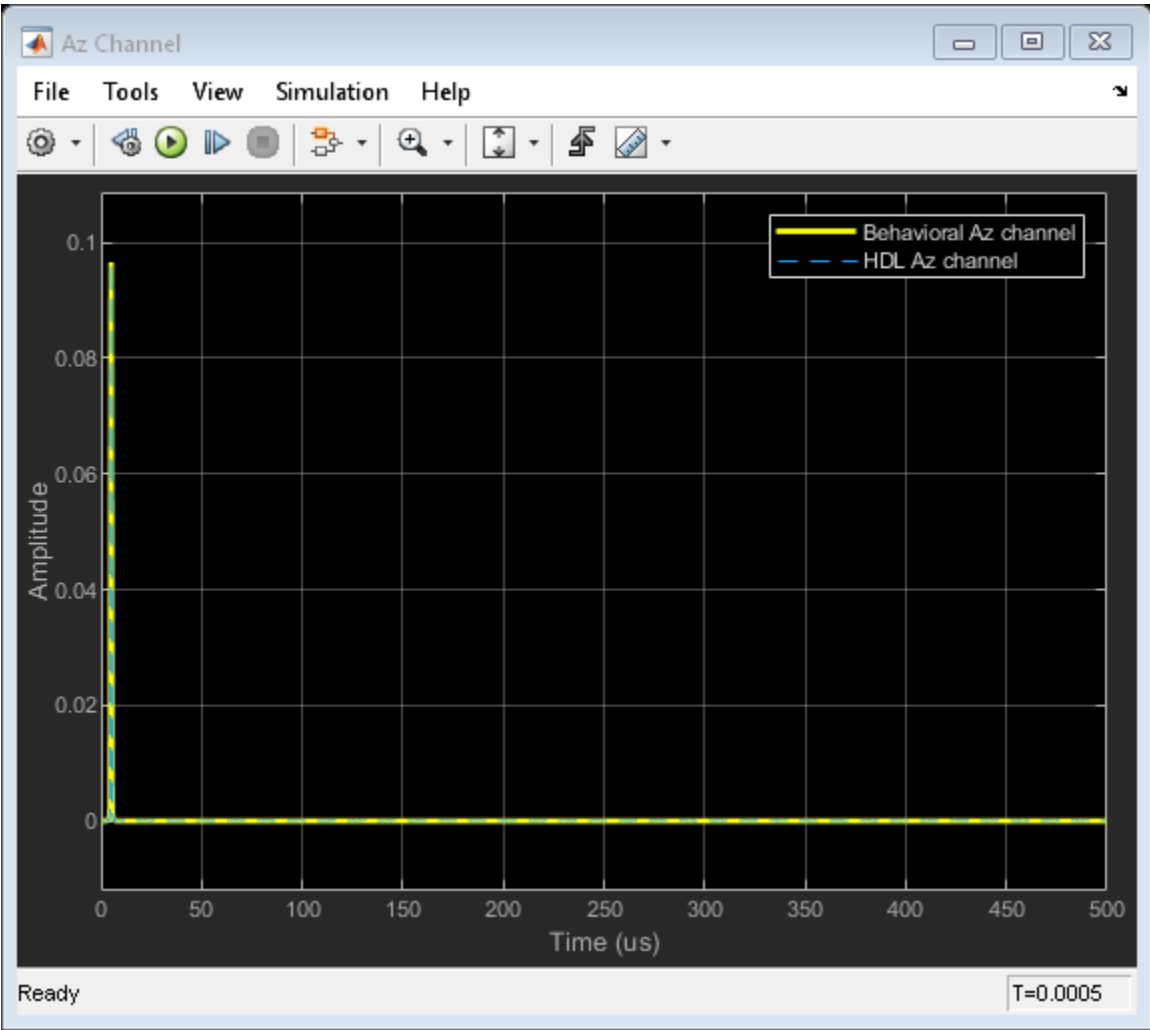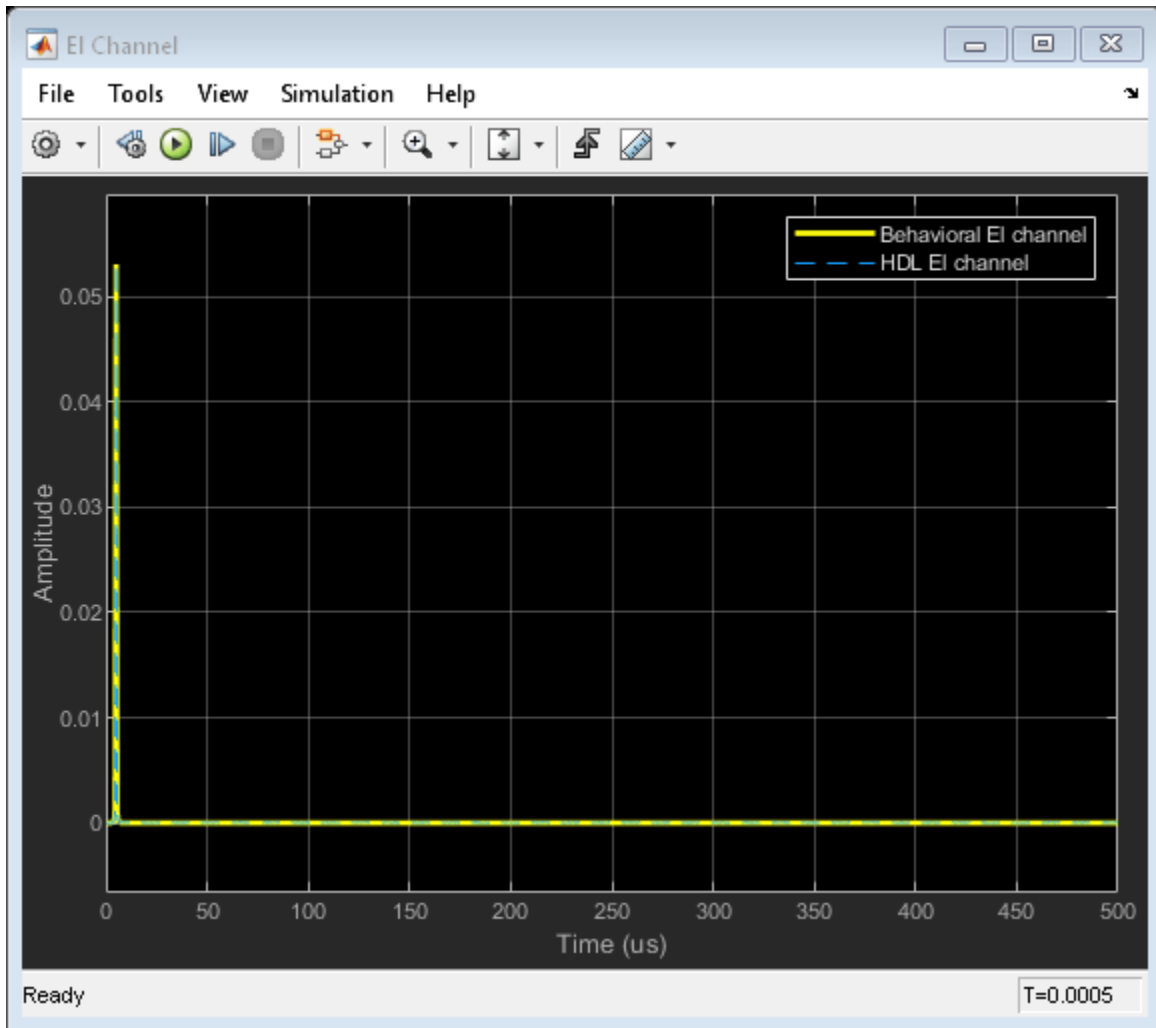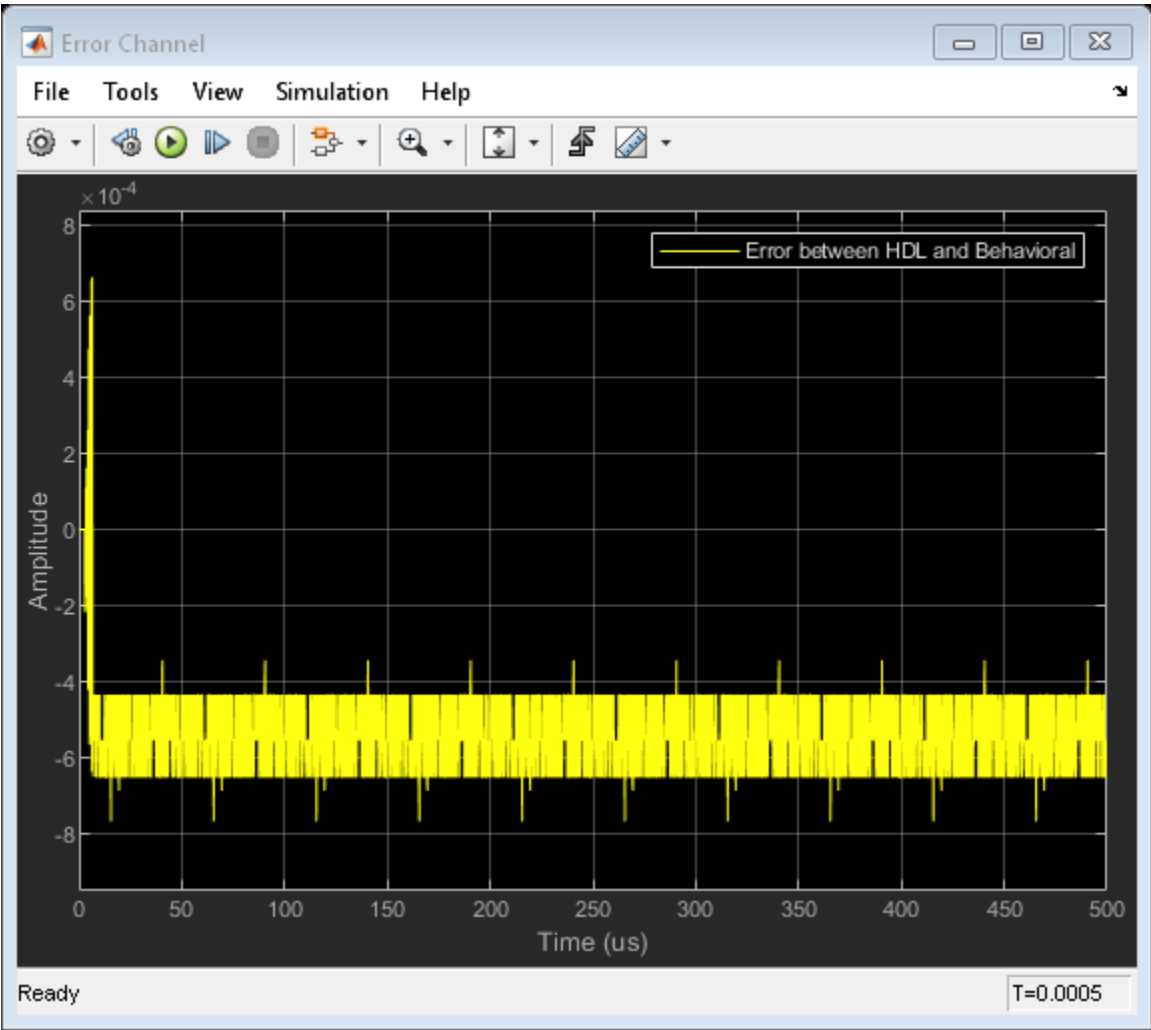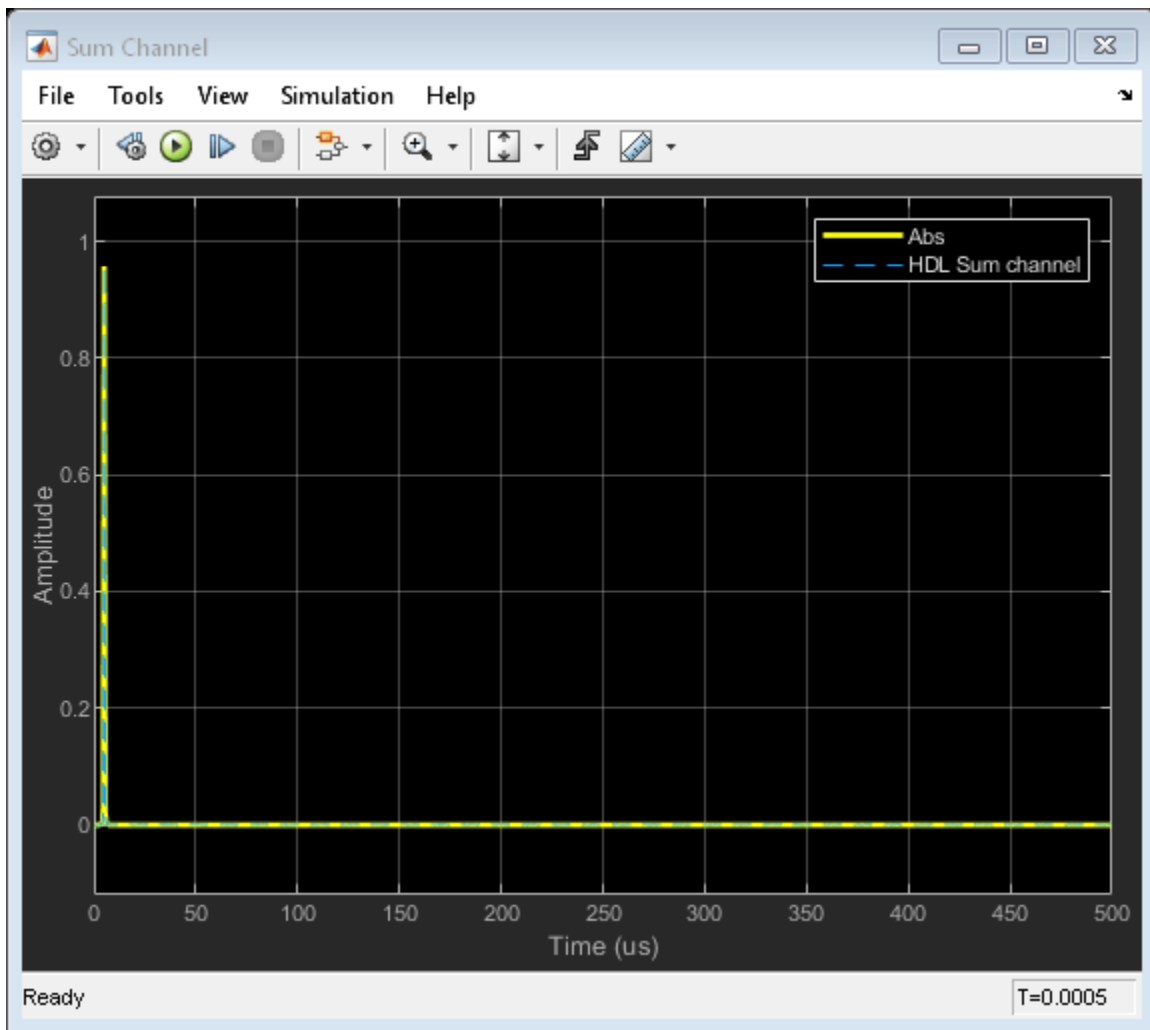
**Code Generation and Verification**

This section shows how to generate HDL code for a DDC and monopulse technique and verify that the generated code is functionally correct. The behavioral model provides the reference values to ensure that the output from HDL is within tolerance limits.

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point, behavioral model, you can generate HDL code and test bench. For code generation and test bench, set these HDL Code Generation parameters in the Configuration Parameters dialog.

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench, Cosimulation model and SystemVerilog DPI test bench.

## HDL Code Generation and Test Bench Creation

After Simulink Model Settings are updated, you can use HDL Coder to generate HDL Code for the `HDL Algorithm` subsystem and use HDL Verifier to generate test bench model.

Use these commands to generate HDL code and test bench.

```
makehdl([modelname '/DDC and Monopulse HDL']);   % Generate HDL code
makehdltb([modelname '/DDC and Monopulse HDL ']); % Generate Cosimulation test bench
```

Since the model has accounted for pipelining in the multiplications, and disabled code gen optimizations, there are no extra delays added to the model. To align the output of the behavioral model with the implementation model and the cosimulation, you must compensate for the pipeline delays. A delay of ($Z^{-215}$) is added to the output of the digital comparator. This delay is added to compensate for the latency in the DDC chain. Also, out of the 220 units delay, 215 unit delays compensates for the latency in the DDC chain and 5 units in the monopulse sum and difference subsystem.

When you generate HDL code and test bench, a new Simulink model named `gm_<modelname>_mq` that contains a ModelSim Cosimulation block is created in your working folder.
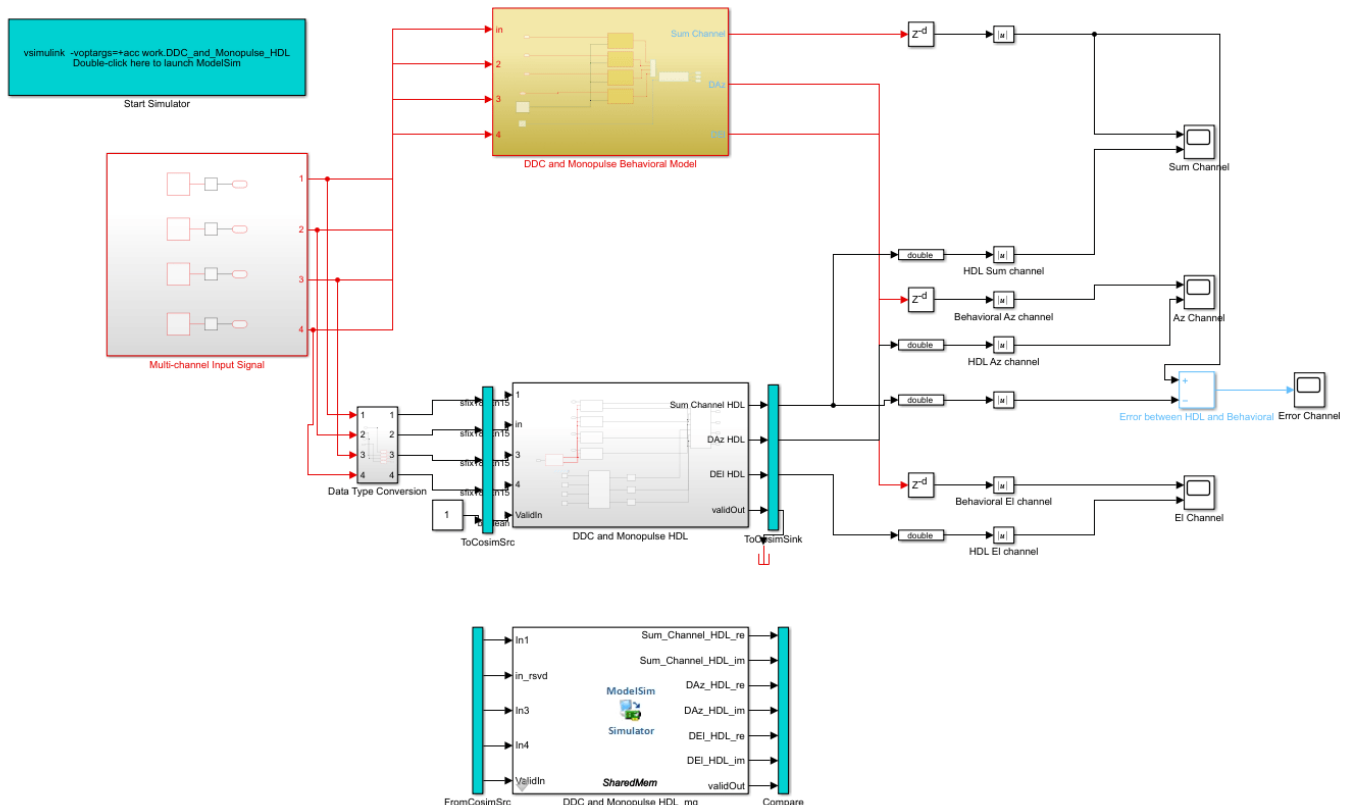


To open the test bench model, use this command.

```
modelname = ['gm_',modelname,'_mq'];
open_system(modelname);
```

Launch ModelSim and run the cosimulation model to display the simulation results. You can click on the Play button on the top of Simulink canvas to run the test bench or run it from the MATLAB® command line.

Use this command to run the test bench.

```
sim(modelname);
```

The Simulink test bench model populates the Questa® Sim waveform with the HDL model's signal and Time Scopes in Simulink. The figures show examples of the results in Questa Sim and Simulink scopes.
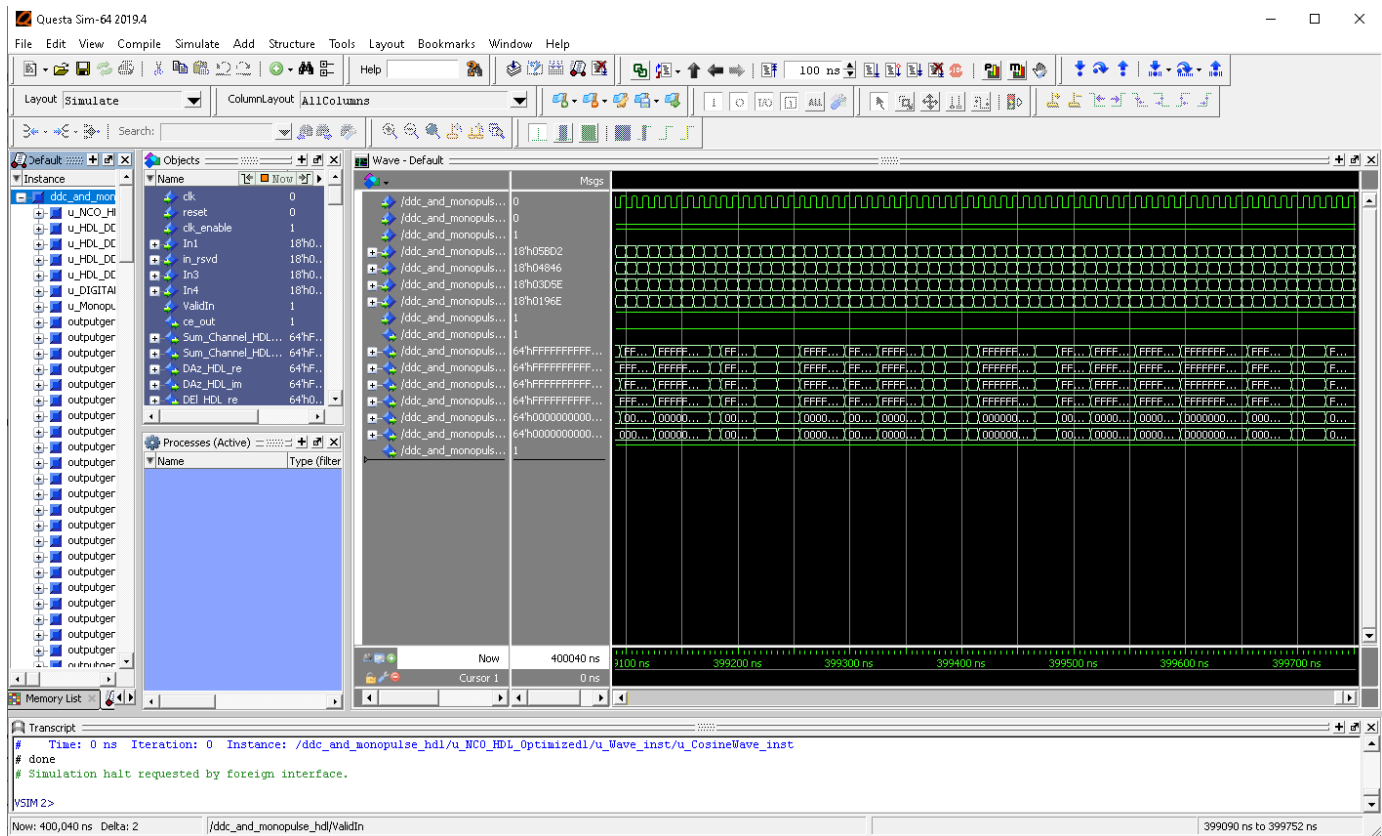


The Simulink scope shows real and imaginary parts for both the cosimulation and design under test (DUT) as well as the error between them.

The Simulink scopes comparing the results of the cosimulation can be found in test bench model inside the `Compare` subsystem, which is at the output of the `DDC and Monopulse HDL_mq` subsystem.

Use this command to open the subsystem with the scopes.

```
open_system([modelname,'/Compare/Assert_Sum Channel HDL'])
```
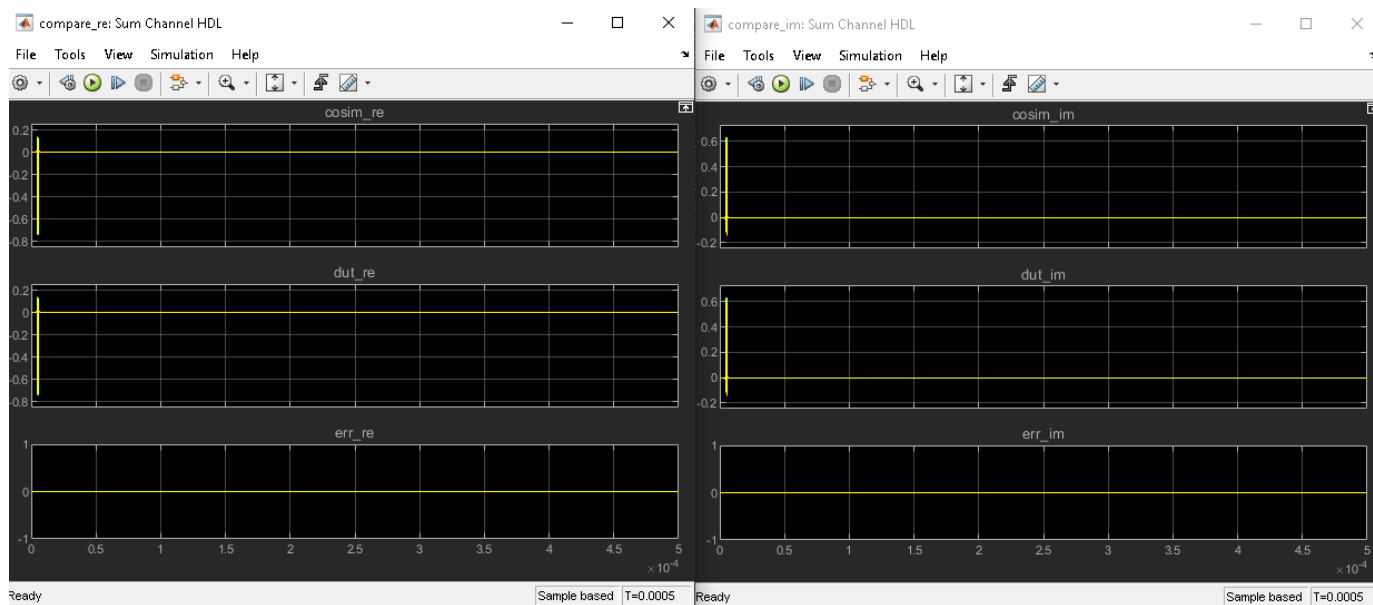
## Summary

The generated HDL code as well as a cosimulation test bench for the Simulink subsystem were created with blocks that support HDL code generation. The example showed how to setup and launch ModelSim to cosimulate the HDL code. The cosimulation is performed via ModelSim for the HDL code and comparison of results to the output generated by the HDL model. The example helped in automatically generating HDL code for a fixed-point monopulse technique and verify the generated code in Simulink.

# FPGA-Based Range-Doppler Processing - Algorithm Design and HDL Code Generation

This example shows how to design a range-Doppler response that is ready for FPGA (field programmable gate array) implementation. The example matches the FPGA implementation to a corresponding behavioral model in Simulink® using the Phased Array System Toolbox™.

To verify the functional correctness of the hardware implementation model, the example compares the simulation output of that model with the results of the behavioral model. The term deployment here implies designing a model that is suitable for implementation on an FPGA. The model is deployment-ready and this condition is verified in the example. The hardware implementation is designed using fixed-point data types.

The Phased Array System Toolbox provides the floating-point behavioral model for the range-Doppler response through the `phased.RangeDopplerResponse` System object™. This behavioral model is used to verify the correctness of the implementation model.

Fixed-Point Designer™ provides data types and tools for developing fixed-point and single precision algorithms to optimize performance on an embedded hardware. Bit-true simulations can be performed to observe the impact of limited range and precision without implementing the design in hardware.

This example uses HDL Coder™ to generate HDL code from the developed Simulink model and verifies the HDL code using HDL Verifier™ tools. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the automatically generated HDL code. The testbench uses ModelSim® for cosimulation of the generated HDL code.

**Range-Doppler Response Algorithm**

The `phased.RangeDopplerResponse` System object generates the range-Doppler Response using this algorithm.

1   Fast-time dimension: Filters the signal with a matched filter to generate the range response. The matched filter is an FIR filter with the coefficients set to the time-reverse replica of the transmitted signal.
2   Slow-time dimension: Computes the FFT to generate the Doppler response.

The input data is a matrix of M-by-N values, where M is the number of cells and N is the number of pulses. To calculate the range response, use an FIR filter across the rows (fast-time) and compute the FFT across the columns (slow-time).

**Hardware Implementation Model**

This example reuses the example input data and parameters from the `phased.RangeDopplerResponse` reference page example.

Serialize and deserialize the signal by using the Serializer1D and Deserializer1D blocks, respectively. These blocks have input and output constraints for code generation, so set the FFT length to 64 and use a subset of the input data cube.

The implementation model uses a word length of 32 bits and a fraction length of 31 bits.

Open the Simulink model by using this command.

```
modelname = 'SimulinkRangeDopplerProcessingHDLWorkflowExample';
open_system(modelname);
% Ensure model is visible and not obstructed by scopes
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```

**Range Doppler Processing**
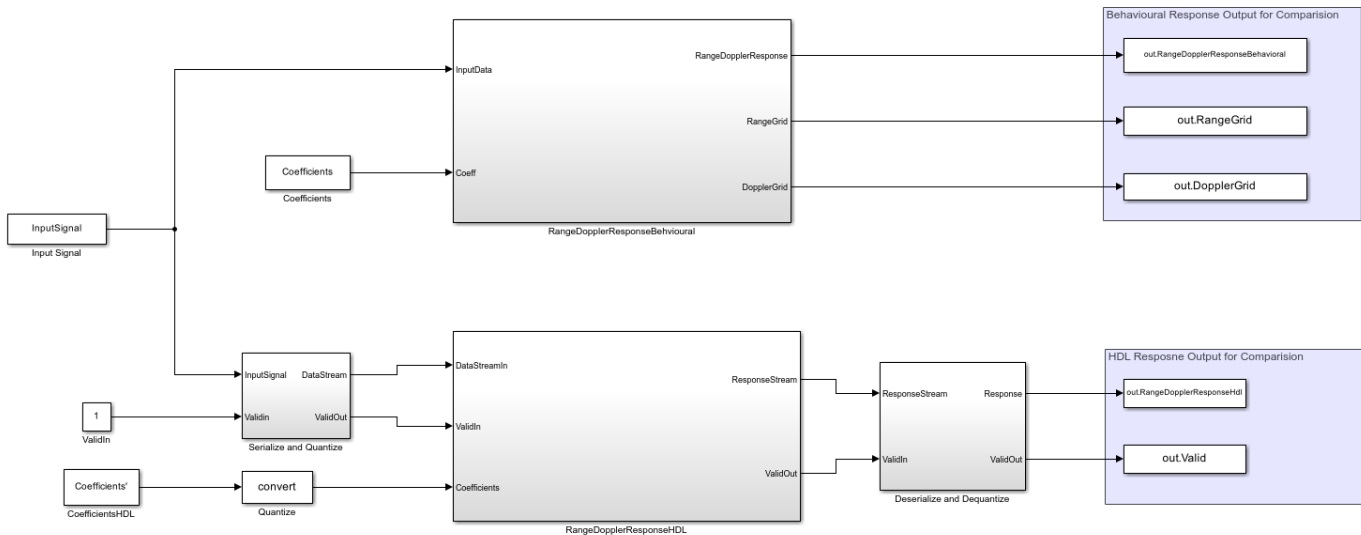


Copyright 2021 The MathWorks Inc.

The Simulink model consists of two branches from the Input block. The top branch is the behavioral model with floating-point operations of the `phased.RangeDopplerResponse` System object. The bottom branch is the functionally equivalent implementation model using fixed-point data types, designed with blocks that support HDL code generation from the Simulink HDL Coder Library.

The input and coefficients are generated from the range-Doppler example data. The input is a M-by-N matrix, where M is the number of range cells (fast-time dimension) and N is the number of pulses (slow-time dimension). The output of the `phased.RangeDopplerResponse` object is a M-by-L matrix, where M is the number of range cells and L is the FFT length. Since the input and output of the implementation model have to be data streams, the input is serialized and quantized in the `Serialize and Quantize` subsystem and the output is deserialized to form a range-Doppler matrix map at the output in the `Deserialize and Dequantize` subsystem.

**Preprocessing and Postprocessing Data**

Open the Serialize and Quantize subsystem.

```
open_system([modelname '/Serialize and Quantize'])
```

**16-37**
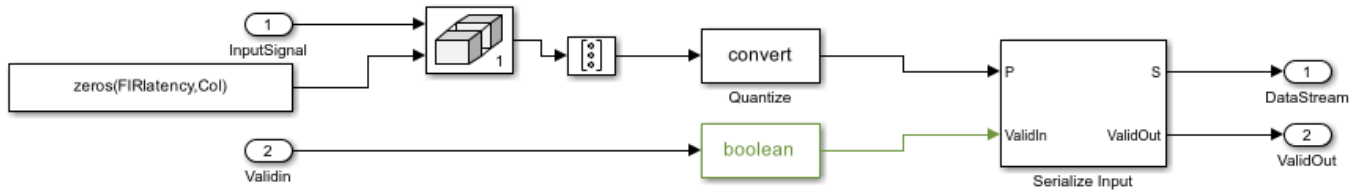
The input data is zero-padded to accommodate the FIR filter latency by using the Matrix Concatenate block. The data is then serialized by a Serializer1D block used in cascade with the reshape and data-type convert (quantize to fixed-point) block.

```
open_system([modelname '/Deserialize and Dequantize'])
```
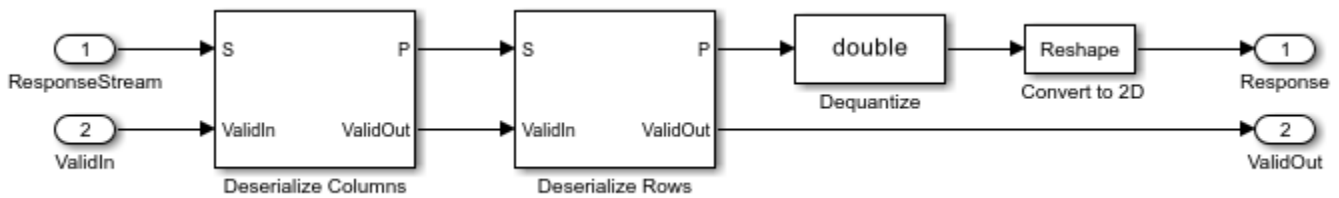


This subsystem uses a Deserializer1D block which uses the data-type convert and reshape blocks to convert the output stream to a range-Doppler map.

### Matched Filtering - Range Processing Subsystem

```
open_system([modelname '/RangeDopplerResponseHDL/Matched Filtering - Range Processing'])
```
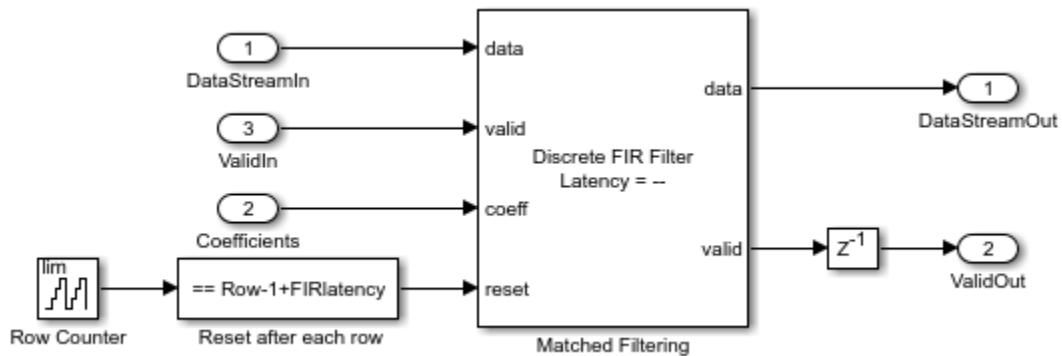


The input stream is processed with a Discrete FIR Filter block with the matching coefficients in the fast-time (row) dimension to get the range response. The block is suitable for hardware implementations and has a latency of seven cycles. The registers in the FIR filter have to be reset to an initial value of 0 after every row. This reset is performed by using a Boolean square wave implemented using a Counter and a Compare To Constant block.

### Buffer & Transpose-Column

Open the Buffer & Transpose-Column subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Buffer & Transpose - Column'])
```
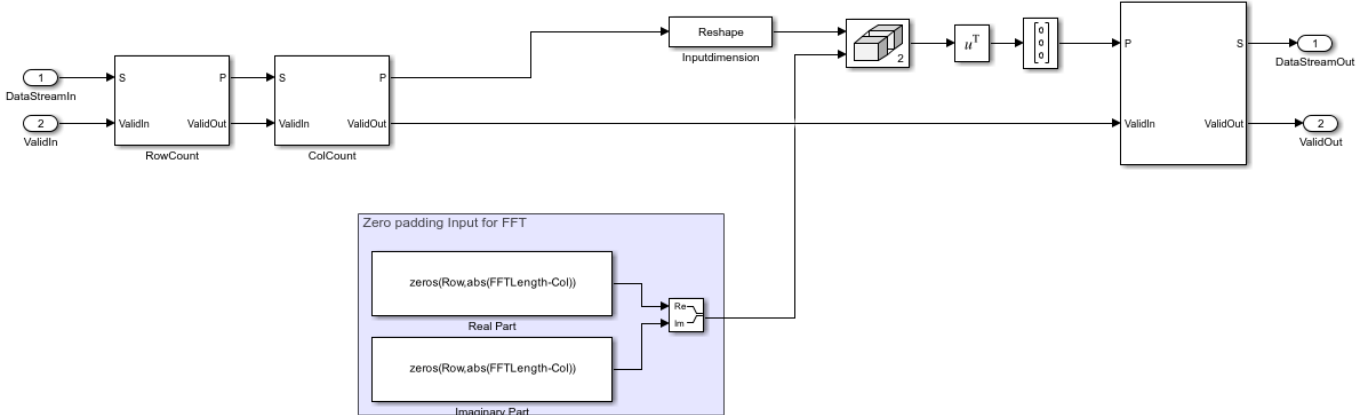
The range-processed data is deserialized using cascaded Deserializer1D blocks and is converted to matrix format with a reshape block. This data is now zero padded and transposed for computing the FFT across the slow-time (column) dimension. The range-processed and transposed data is now serialized again for FFT computation (Doppler processing).

**FFT-Doppler Processing**

Open the FFT-Doppler Processing subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/FFT - Doppler Processing'])
```
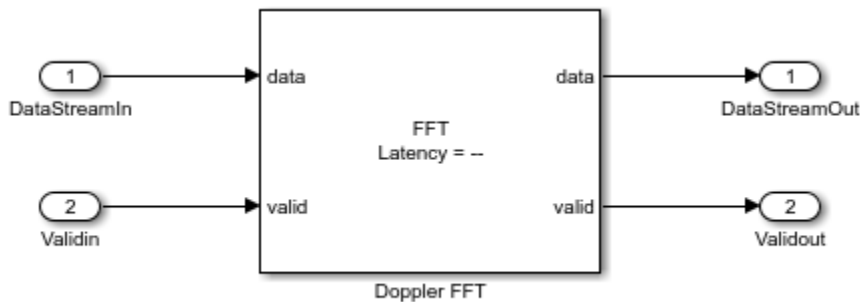


The FFT of the streamed (across column) data is calculated using an FFT block which has a latency of 173 cycles. A streaming radix 2^2 architecture is used with an FFT length of 64.

**Buffer-FFT Shift**

Open the Buffer-FFT Shift subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Buffer - FFT Shift'])
```

16-39
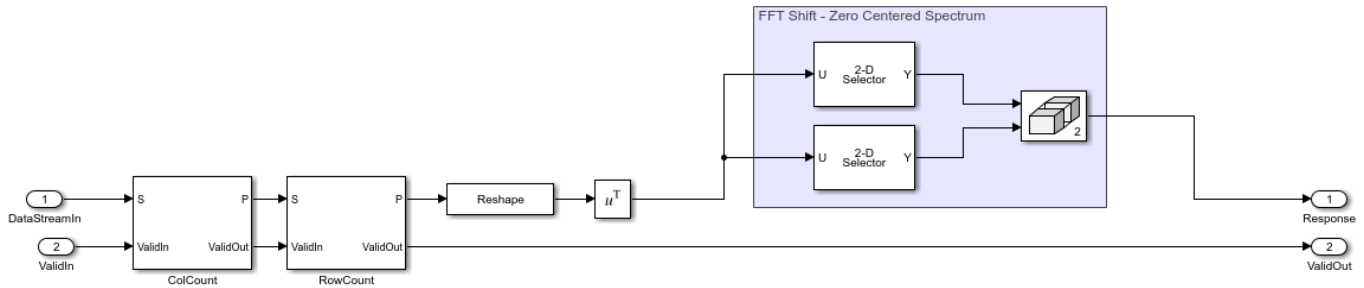
The Doppler-processed serial data is deserialized using Deserialized1D blocks and transposed. The FFT-processed data needs to be rearranged to have a zero-centric spectrum data, which is emulated by using a combination of selector blocks and a matrix concatenate.

**Serialize Output**

Open the Serialize Output subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Serialize Output'])
```
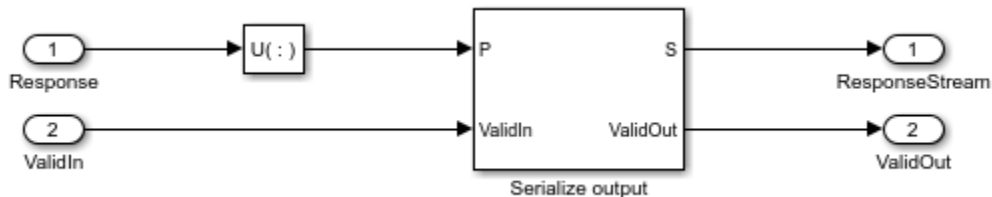


The range-Doppler map data after processing is in the form of a matrix, which is then serialized using a Reshape and a Serializer1D block for the output stream.

**Compare of Implementation Model and Behavioral Model Results**

Simulate the model by clicking the `Play` button or by using the `sim` command.

```
sim(modelname);
```

To verify the functional correctness of the implementation model, subtract the response matrix of the behavioral model from the response matrix of the implementation model and check that the difference (error) is close to zero (or, within the quantization in the implementation model).

Export the response data from Simulink to the MATLAB® workspace, in the array format. Subtract the behavioral response vector from the implementation model response, reshape the error matrix into a 1-D array, and plot the error with the element index in the x-axis and error in the y-axis. Use the `imagesc` function to display the range-Doppler map. Use the following script to plot the response and error.

```
% Uncomment the following lines of code to visualize the response
% Behavioral Output
behavioralResponse = out.RangeDopplerResponseBehavioral(:,:,1);   % Response from To Workspace Bl
behavioralResponsedB = mag2db(abs(behavioralResponse));           % Convert to dB
rangeGrid = out.RangeGrid(:,1,1);                                 % Range Grid
dopplerGrid = out.DopplerGrid(:,1,1);                             % Doppler Grid
```

```matlab
% Visualize the range-Doppler (Behavioral) map

f1 = figure(1);                                            % Figure handle
f1.Name = 'Behavioral';                                    % Figure Name
fax1 = axes;                                               % Axis Handle
imagesc(fax1,dopplerGrid,rangeGrid,behavioralResponsedB)

xlabel(fax1,'Doppler');
ylabel(fax1,'Range')
title(fax1,'Behavioral Response')

% HDL Output
idx = find(out.Valid);                                     % Search for valid output
HdlResponse = out.RangeDopplerResponseHdl(:,:,idx(1));     % Response from To Workspace Bl
HdlResponsedB = mag2db(abs(HdlResponse));                  % Convert to dB

% Visualize the range-Doppler (HDL) map

f2 = figure(2);                                            % Figure handle
f2.Name = 'HDL';                                           % Figure Name
fax2 = axes;                                               % Axis handle
imagesc(fax2,dopplerGrid,rangeGrid,HdlResponsedB);         % Use behavioral output of rang

xlabel(fax2,'Doppler')
ylabel(fax2,'Range')
title(fax2,'HDL Response')

% Error
% Subtract the Behavioral Response from the HDL response
errorMatrix = abs(HdlResponse - behavioralResponse);       % Matrix Subtract

% Convert the error matrix into a row vector which can be visualised on a 2D axis

errorStream = reshape(errorMatrix,1,[]);                   % Convert error matrix to 1D, ro

% Find the index and the maximum error between HDL and behavioral results

Ymax = max(errorStream);                                   % Find Maximum Error
Xmax = find(errorStream == Ymax);                          % Find index of maximum error

% Plot the error on a 2D plot and annotate the maximum error
% between HDL and behavioral response.
f3 = figure(3);                                            % Figure handle
f3.Name = 'Error';                                         % Figure Name
fax3 = axes;                                               % Axis handle
plot(fax3,errorStream)                                     % Plot

ylabel(fax3,'Error');
xlabel(fax3,'Data Point Index')
title(fax3,'Error between Behavioral and HDL Model');

textstr = strcat('     ErrorMax = ',num2str(Ymax));
text(fax3,Xmax,Ymax,textstr);
```

The figures show the range-Doppler map and the error between the behavioral and implementation models.

Figure 3: Error — Error between Behavioural and HDL Model

ErrorMax =3.1711e-08
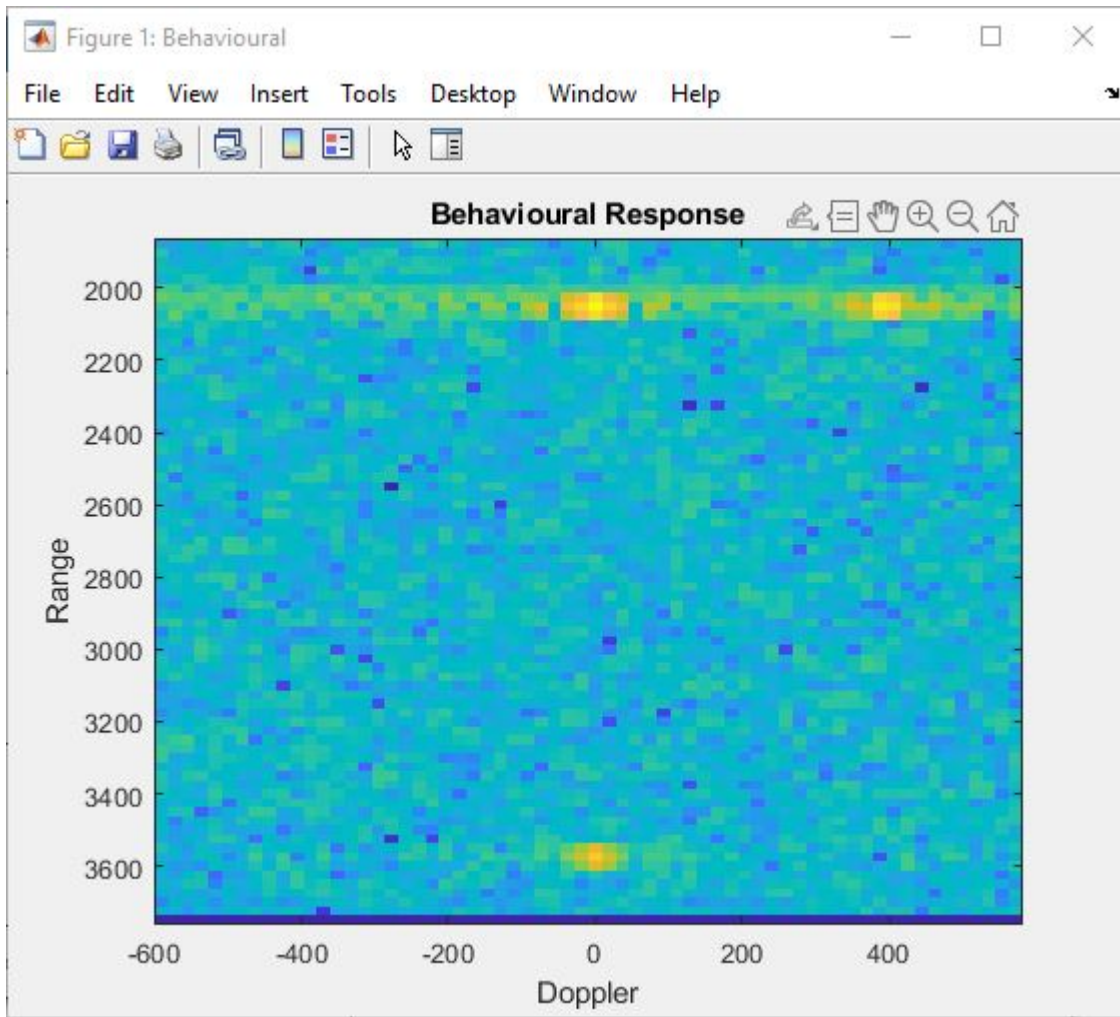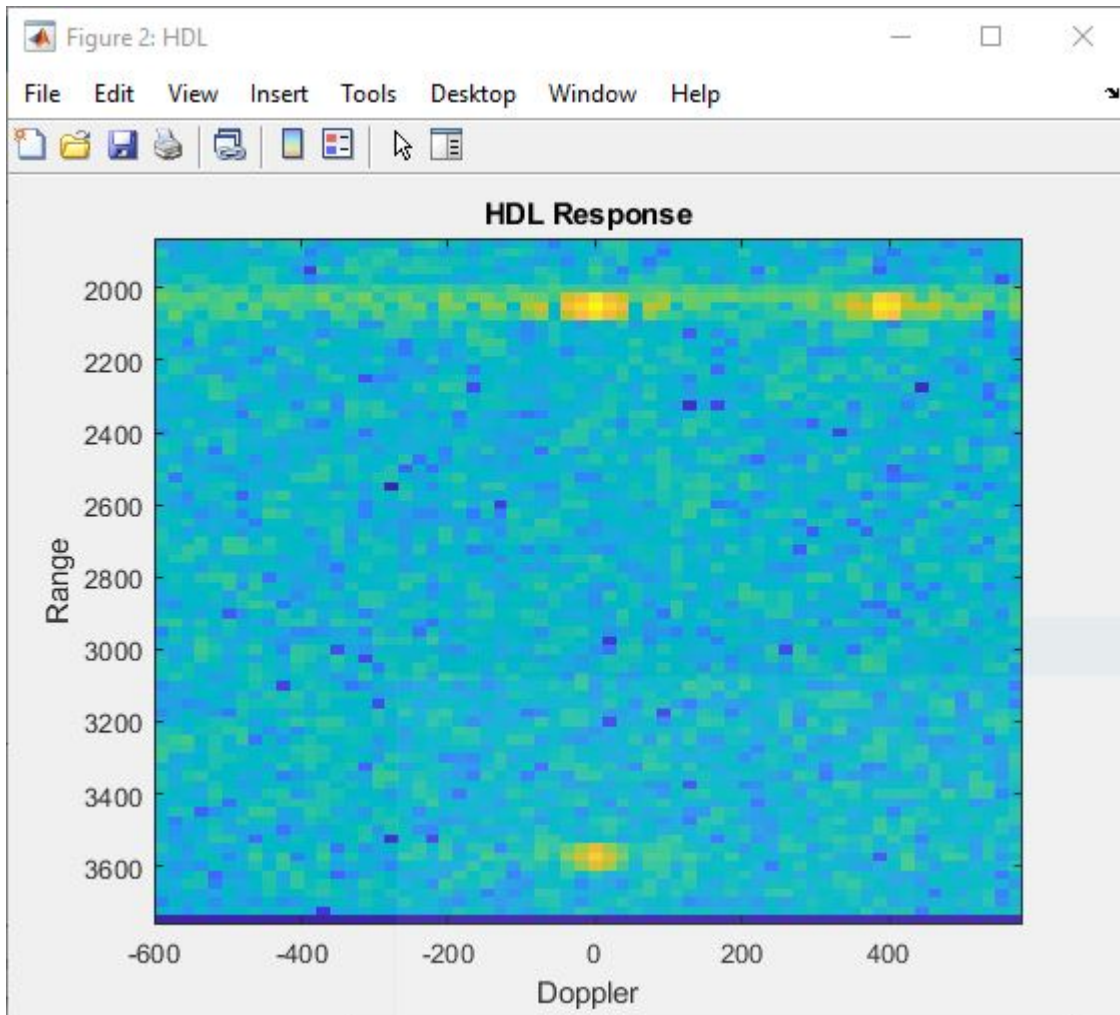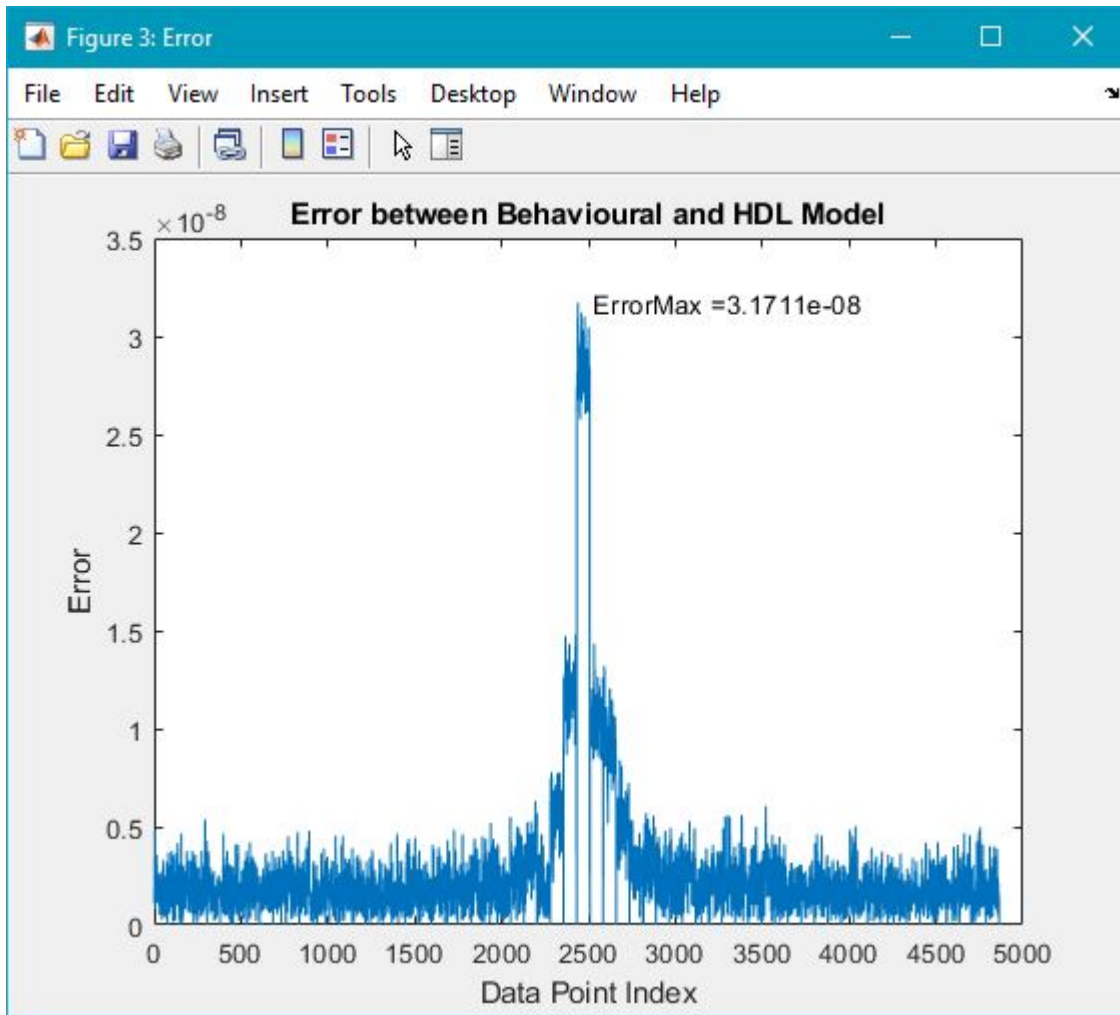
## Code Generation and Verification

This section covers the procedure to generate HDL code for the range-Doppler response implementation model and verifying the functional correctness. The behavioral model provides the reference values to ensure that the output from HDL model is within tolerance limits. Based on the Simulink model setup described in the earlier sections, the implementation model is designed using fixed-point arithmetic blocks that support HDL code generation. Alternatively, if you start with a new model, you can run the `hdlsetup` function to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, open **Model Settings**, select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

## Model Settings

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point behavioral model, you can generate HDL code and test bench. For code generation and test bench, set these HDL Code Generation parameters in the **Configuration Parameters** dialog. Set the following parameters in Model Settings under HDL Code Generation:

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Clear all optimizations.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench, Cosimulation model, and System Verilog DPI test bench.

### HDL Code Verification Using Cosimulation

After the Model is set up, use the **HDL Workflow Advisor** to generate the HDL code using the HDL Coder tools, and generate a System Verilog DPI test bench to test the model using HDL Verifier. To start the HDL Workflow Advisor, right-click on the `RangeDopplerResponseHDL` subsystem, navigate to **HDL Code**, and click **HDL Workflow Advisor**. Alternatively, you can use these commands to generate HDL code and System Verilog test bench.

```
% makehdl([modelname '/RangeDopplerResponseHDL']);   % Generate HDL code
% makehdltb([modelname '/RangeDopplerResponseHDL']); % Generate Cosimulation test bench
```

After generating HDL code and the test bench, a new Simulink model named `gm_<modelname>_mq` containing a ModelSim® Cosimulation block is created in your working directory. This figure shows the generated model.



Range Doppler Processing
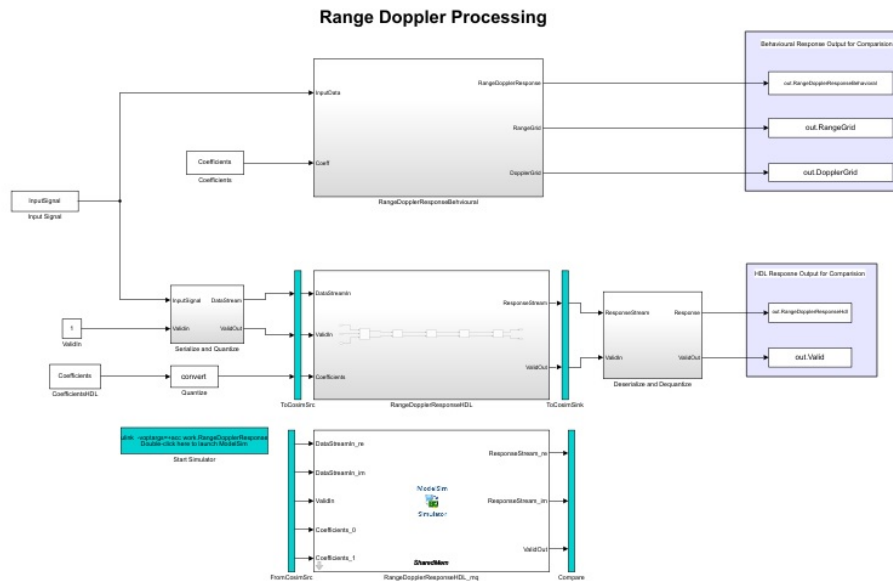
To open the test bench model use these commands.

```
% modelname = ['gm_',modelname,'_mq'];
% open_system(modelname);
```

Launch ModelSim and run the cosimulation model to display the simulation results. You can click the Play button to run the test bench, or you can run it via command window using the following command.

```
% sim(modelname);
```

The Simulink test bench model configures QuestaSim® with the signal from the HDL model and Time Scope in Simulink.

The test bench scopes shows the output of the complex-valued response vector from the implementation model and the cosimulation output as well as the error between the two outputs.
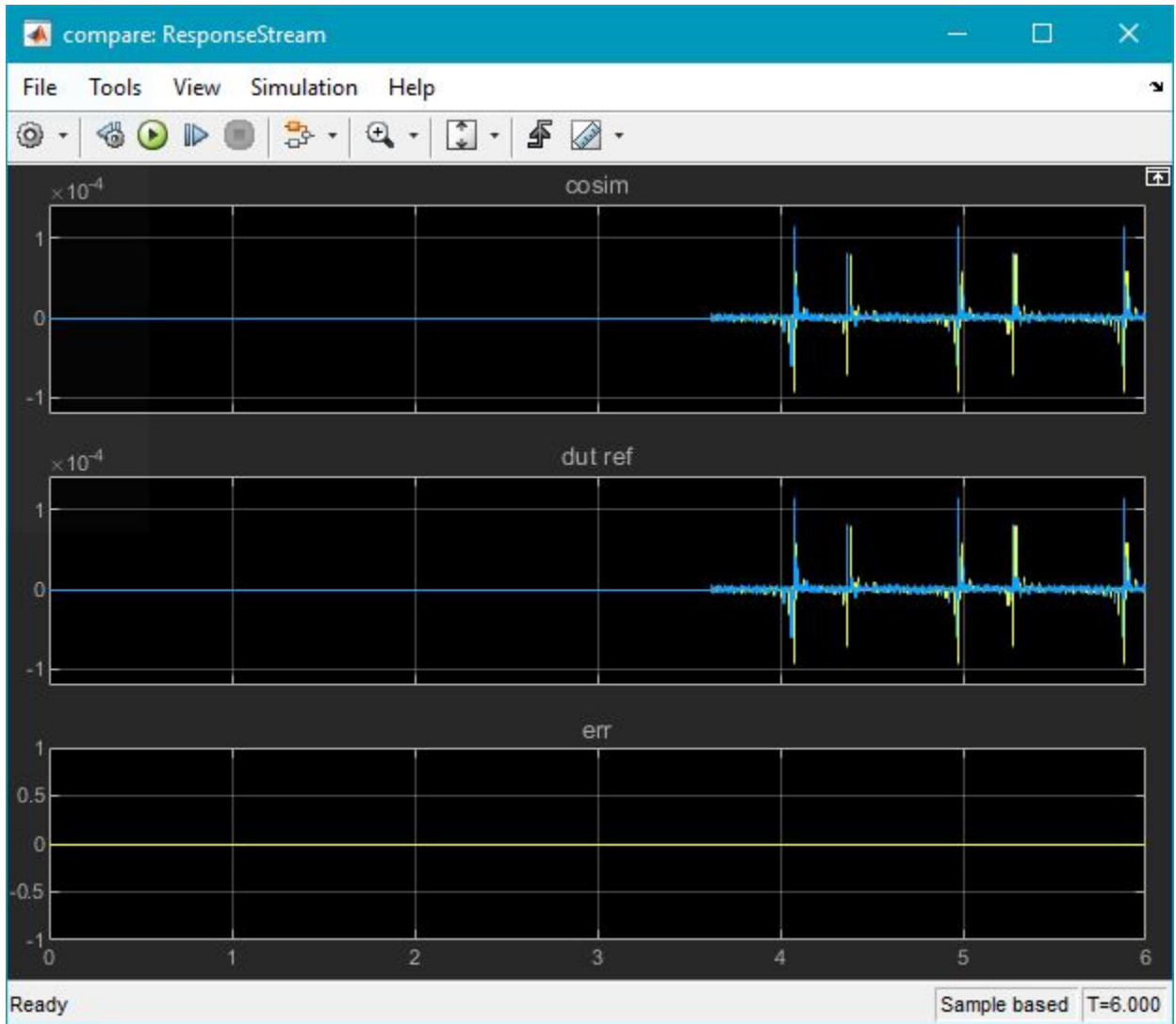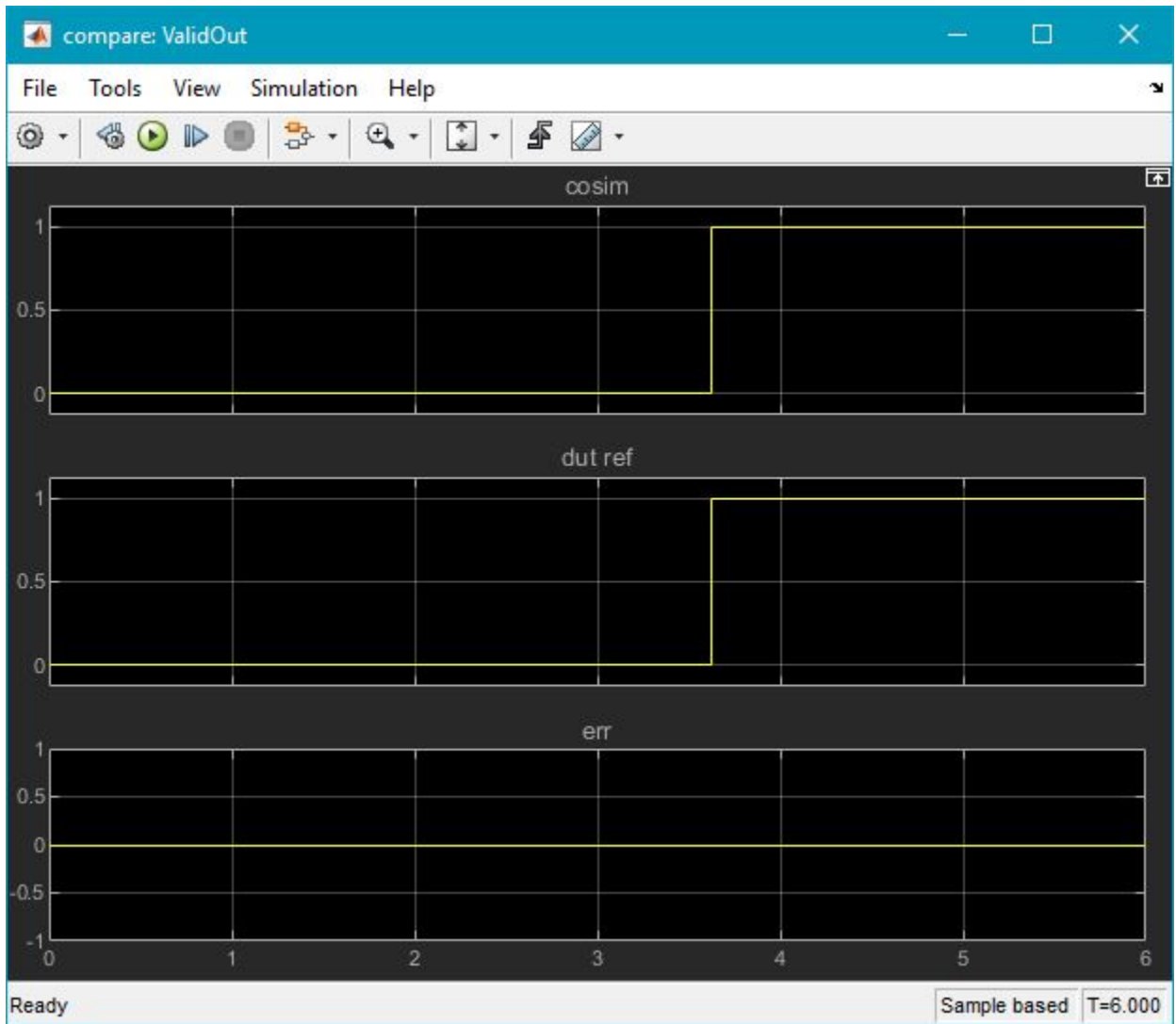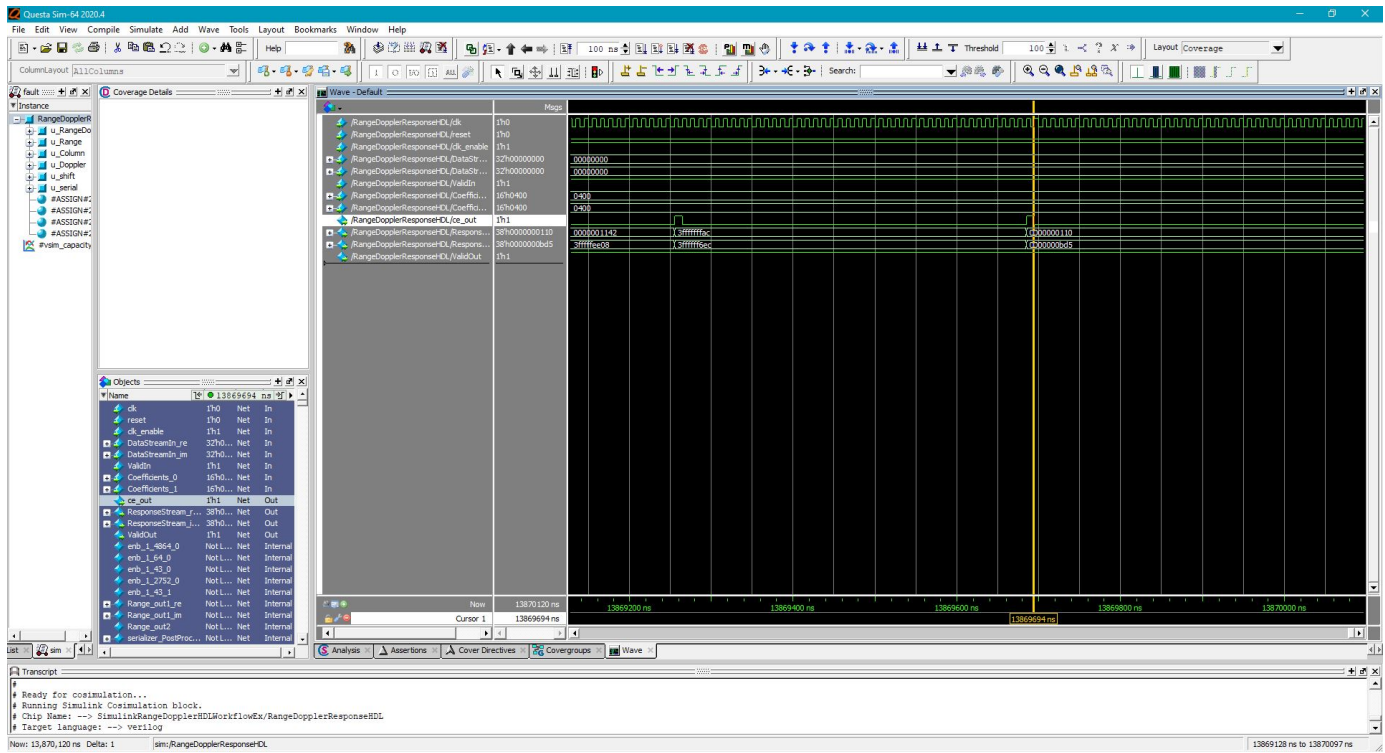
**Fixed-Point Word Length (Precision) and Resource Utilization Tradeoffs**

This example uses a word length of 32 bits and a fraction length of 31 bits for design, simulation, and implementation. There are tradeoffs associated with increasing the data precision with respect to resource utilization.

This figure shows the precision with respect to the word length.
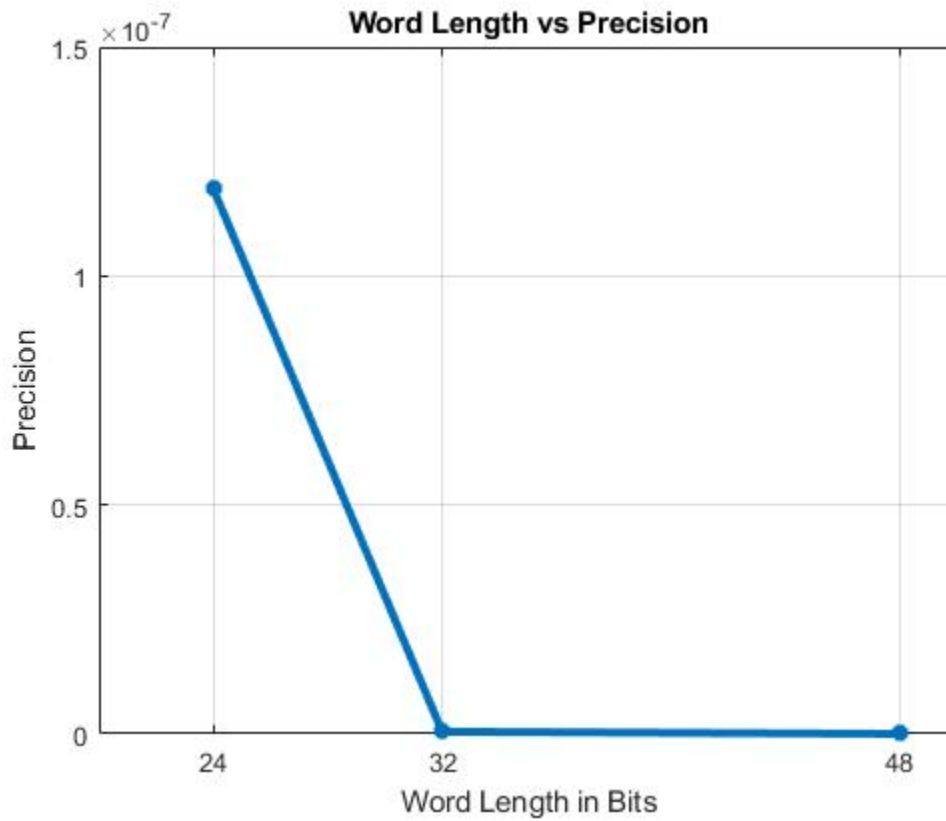
The next figure shows the slice LUT utilization with respect to the word length.
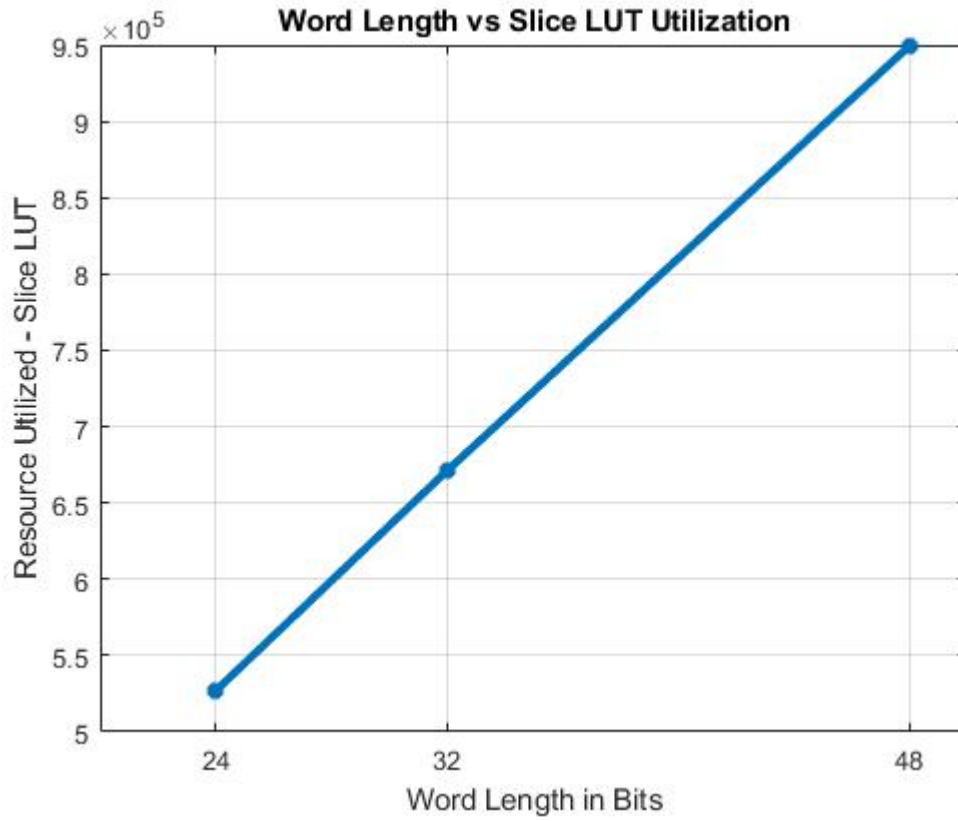
The next figure shows the slice registers utilization with respect to the word length.
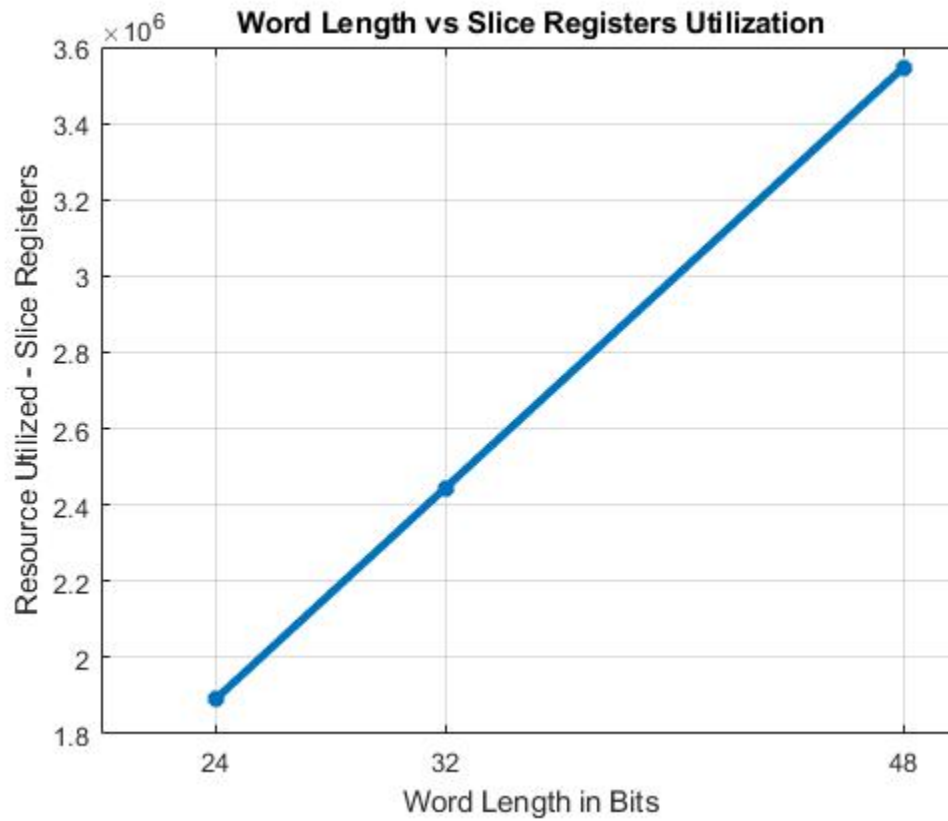
The next figure shows the DSP block utilization with respect to the word length.
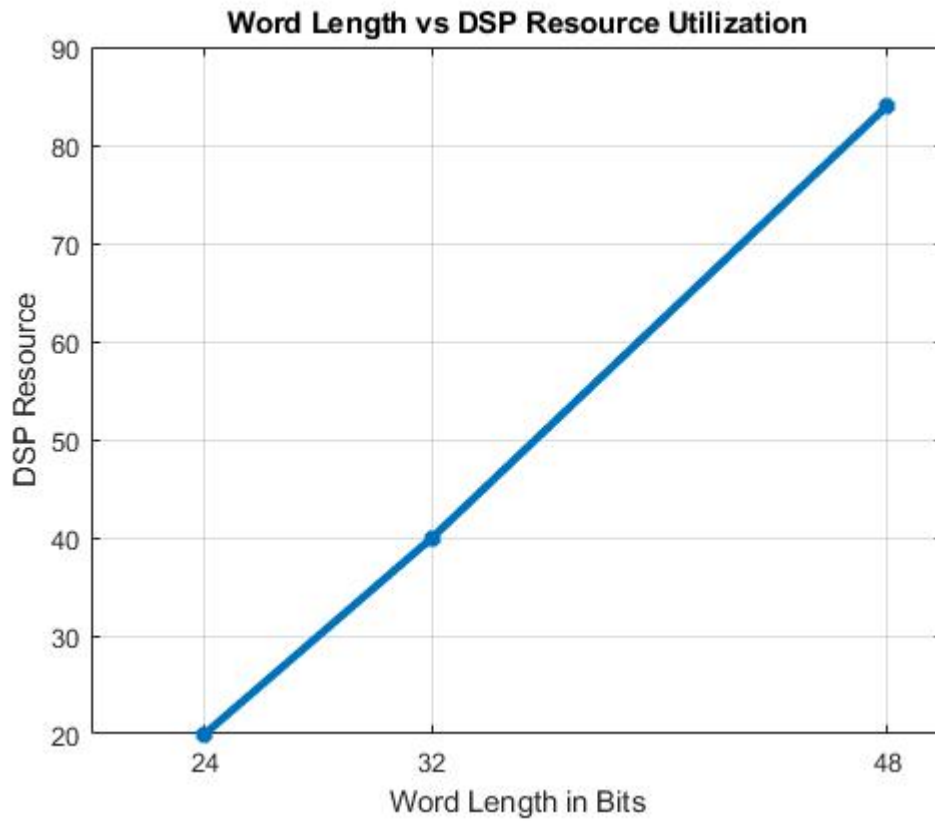
The next figure shows the block RAM tile utilization with respect to the word length.

## Word Length vs Block RAM Tile Utilization
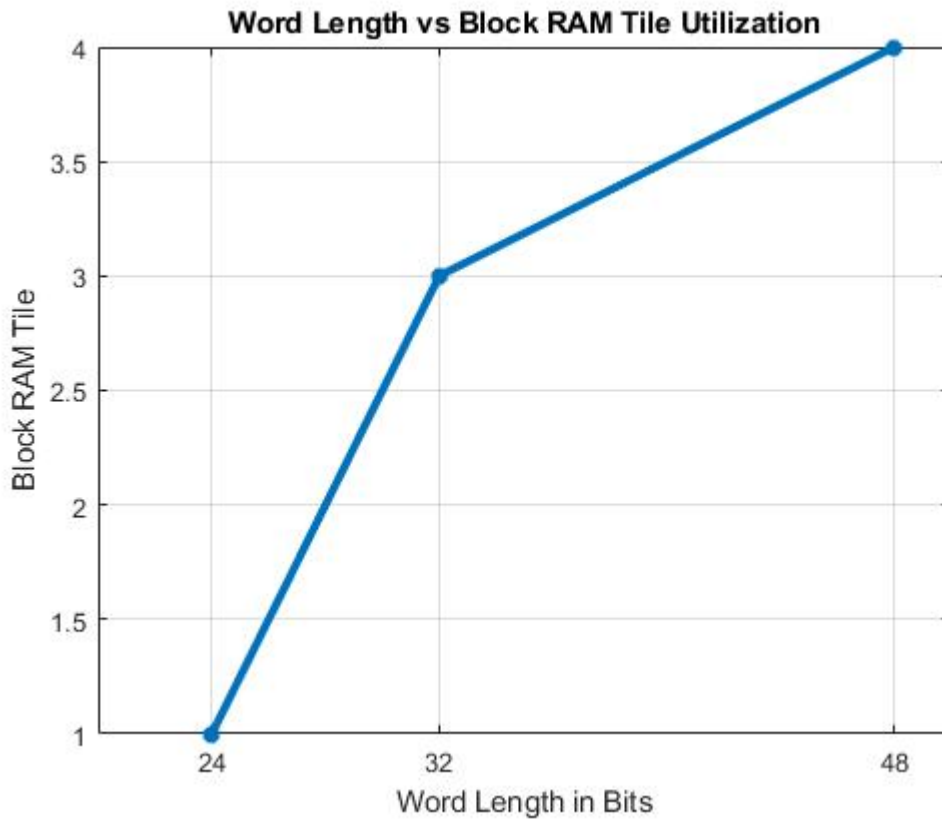


**Summary**

This example demonstrated a workflow for designing a Simulink model for a hardware-compatible range-Doppler response block, and verifying the results with an equivalent behavioral setup from the Phased Array System Toolbox. The example also shows how to generate HDL code for a fixed-point implementation and verifying the generated code in Simulink for functional correctness. This example showed how to set up and launch ModelSim to co-simulate the HDL code and compare its output to the output generated by the HDL implementation model. The cosimulation used ModelSim for the HDL code simulation and compared results to the output generated by the implementation model.

# FPGA-Based Cell-Averaging Constant False Alarm Rate (CA-CFAR) Detector

This example shows how to design a CA-CFAR detector suitable for hardware.

To verify the implementation model is functionally correct, we compare the simulation output of the implementation model with the output of a CFAR based behavioral model using Phased Array System Toolbox™. The term deployment here implies designing a model that is suitable for implementation on a FPGA. The model is implementation ready and this will be verified in the example. The HDL workflow is designed in fixed-point.

The Phased Array System Toolbox provides the floating point behavioral model for the CFAR detector through the phased.CFARDetector System object™. This behavioral model is used to verify the results of the implementation model and the automatically generated HDL code as well.

Fixed-Point Designer™ provides data types and tools for developing fixed-point and single precision algorithms to optimize performance on an embedded hardware. Bit true simulations can be performed to observe the impact of limited range and precision without implementing the design in hardware.

This example uses HDL Coder™ to generate HDL Code from the developed Simulink® model and verifies the HDL Code using the HDL Verifier™. HDL Verifier is used to generate a co-simulation test bench model to verify the behavior of the automatically generated HDL Code. The test bench uses ModelSim® for Co-simulation to verify the generated HDL code.

**Algorithm Design**

In a radar system, target detection is achieved by comparing the received signal power to a global threshold. If the received power is greater than the threshold, it marks the presence of a target else a target is said to be absent. This makes the choice of threshold a critical characteristic. The appropriate threshold value depends on maximizing the detection and minimizing the false alarm.

The threshold is chosen based on apriori knowledge (estimate) of the interferer power. The interferer power is affected by many external factors, hence, the variance will be a large value when measured globally. When the threshold is constant, the increase in interferer power can lead to an increase in false detections and at the same time, if the interferer power drops significantly, the target might not be detected.

The CFAR detector, as the name suggests, maintains the specified false alarm rate by means of an **Adaptive Thresholding**, wherein the threshold is calculated based on the locality of the Cell Under Test (CUT) and this defines the cell for which detection is required. The interference power of the neighboring cells is used to calculate the threshold for a CUT. The detection threshold is calculated as

$$T = -P_n^2 . ln(P_{FA}) = -P_n^2 \alpha$$

where, $P_{FA}$ is the Probability of False Alarm, $\alpha$ is the Threshold Factor, $P_n$ is the Interference power level.

Cell-Averaging(CA) CFAR Detector In a CA CFAR, the lead and lag cells are used to calculate the interferer estimate. The number of lead cells are the same as that of the number of lag cells. CA-CFAR assumes that, the neighboring cells to the CUT contains the same interference statistic -

Homogenous Interference and the target is present in only one CUT. To reinforce the second assumption, guard cells are placed immediately after the CUT.

For a CA-CFAR with an independent and identically distributed (i.i.d) Gaussian interference (standard normal), the average noise power is just the mean of output of square law detector of all the training cells, which is

$$\widehat{P_n^2} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

Here $x_i$ is the signal from the i-th training cell. For a given Probability of False Alarm, the threshold factor can be calculated as,

$$\alpha = N.(P_{FA}^{(\frac{-1}{N})} - 1)$$

The training cell and guard cell along with the CUT is called as the CFAR Window. The following figure shows a representation of a CFAR Window.
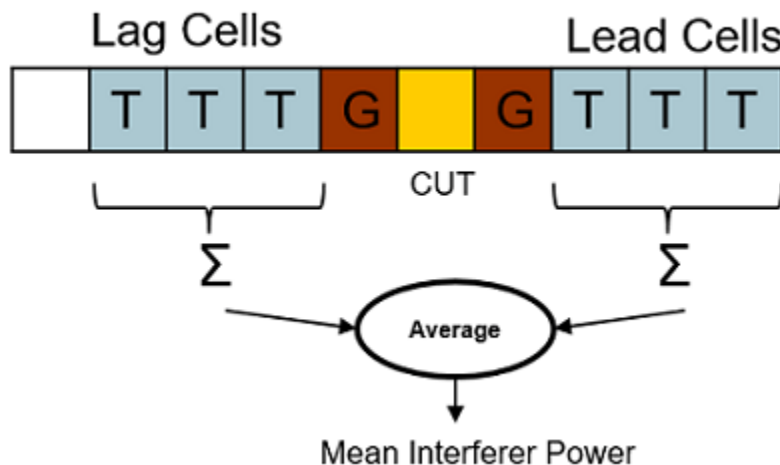


**Implementation Model**

The implementation model is designed using the HDL Coder compatible blocks from the Simulink HDL Coder library. For this example, we have chosen the following values for the parameters:

- No. of Training Cells = 50
- No. of Guard Cells = 2
- Probability of False Alarm = 0.005
- Total No. of Cells = 1000

The following command is used to open the Simulink model.

```
modelname = 'SimulinkCFARHDLWorkflowExample';
open_system(modelname);
%
% Ensure model is visible and not obstructed by scopes
```

```
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```
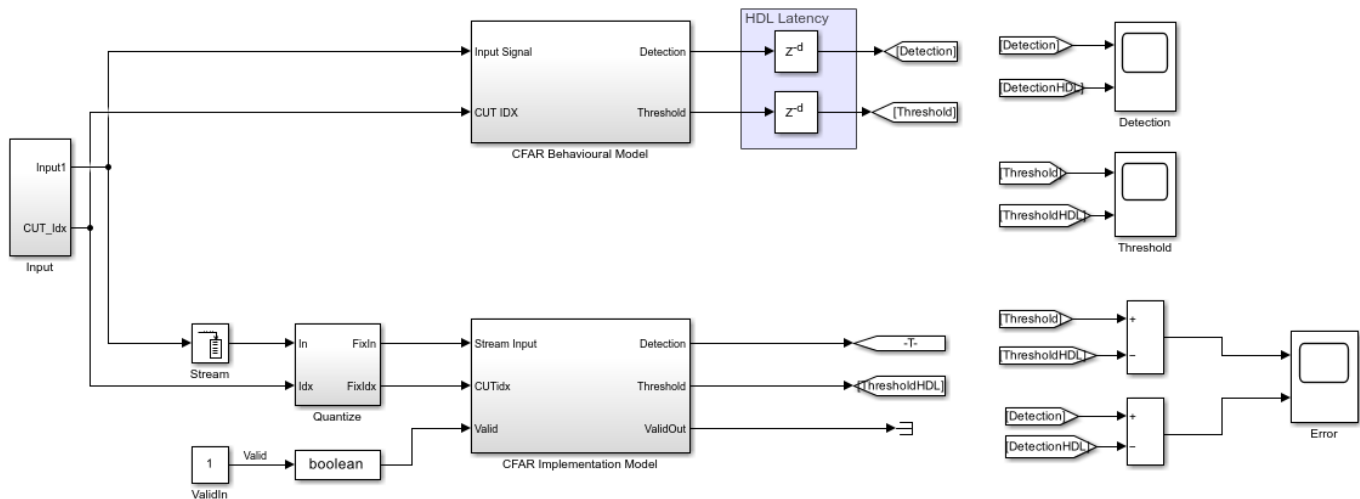


Copyright 2020 The MathWorks Inc.

The Simulink model consists of two branches from the input block. The top branch is the behavioral model with floating point operations of the phased.CFARdetector System object. The bottom branch is the functional equivalent implementation model with fixed-point version.

The input to the behavioral model is a (NCell x 1) 1000x1 matrix. The input is passed through the Square-Law sub-system which performs the square-law operation which is then forwarded into the CFAR Detector behavioral model.

The input to the implementation model is provided via buffer which converts a multi-dimensional signal into single dimensional data stream for a deployable model. The input data type is then converted to fixed-point using the Quantize block. The fixed-point has a word-length of 24 bits and a fraction length of 12 bits. The tradeoff between different fixed-point setting with resource utilization and accuracy is discussed later in this example. The input is then passed on to the CFAR implementation model sub-system which performs the CFAR Detection.

The output of the CFAR Detector behavioral model is passed through a delay of 125 cycles to compensate the delay for the output of the implementation model.

The scope block plots the threshold and detection outputs of the behavioral model and implementation model. In addition, the error between the threshold of implementation model and behavioral model, and the error between the detection of implementation model and behavioral model is also calculated and plotted.

The implementation model contains the following sub-systems:

1   Square-Law HDL
2   Alpha HDL
3   CFAR Core HDL
4   Validate

**Square-Law HDL**

The following command is used to open the Square-Law HDL subsystem model

```
open_system([modelname '/CFAR Implementation Model/Square Law HDL']);
```
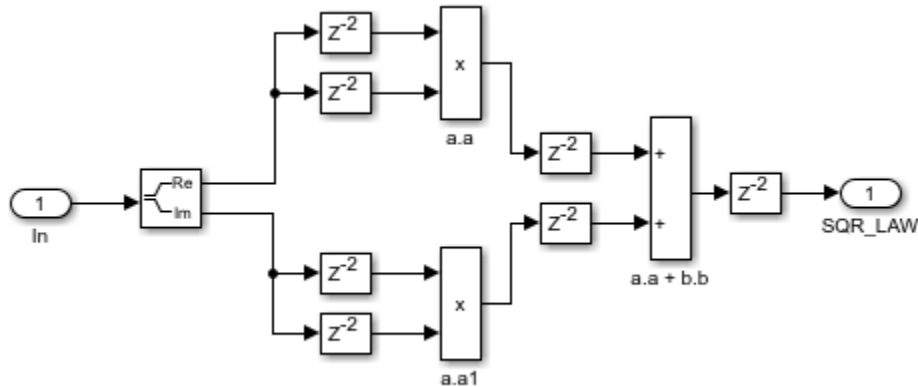


The model computes the square-law envelope of the complex input signal.

For the implementation model the square-law is designed as a deployable model, with additional pipelining registers. This model is implemented using adders and multipliers which account for a latency of 6 cycles.

**Alpha HDL**

The following command is used to open the Alpha HDL subsystem model

```
open_system([modelname '/CFAR Implementation Model/Alpha HDL']);
```
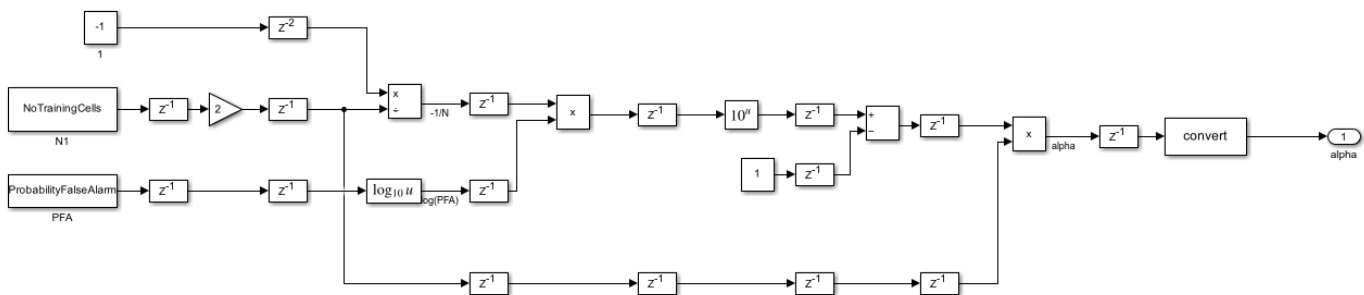


The Alpha HDL block utilizes the **No. of Training Cells** and **Probability of False Alarm** value to calculate the threshold factor ($\alpha$).

This subsystem uses Math HDL blocks and works in single precision for the native floating-point division operation which is then converted to fixed-point at the output. This block accounts for a pipeline latency of 7 Cycles.

**CFAR Core HDL**

```
%The following command is used to open the CFAR Core subsystem model
open_system([modelname '/CFAR Implementation Model/CFAR Core']);
```
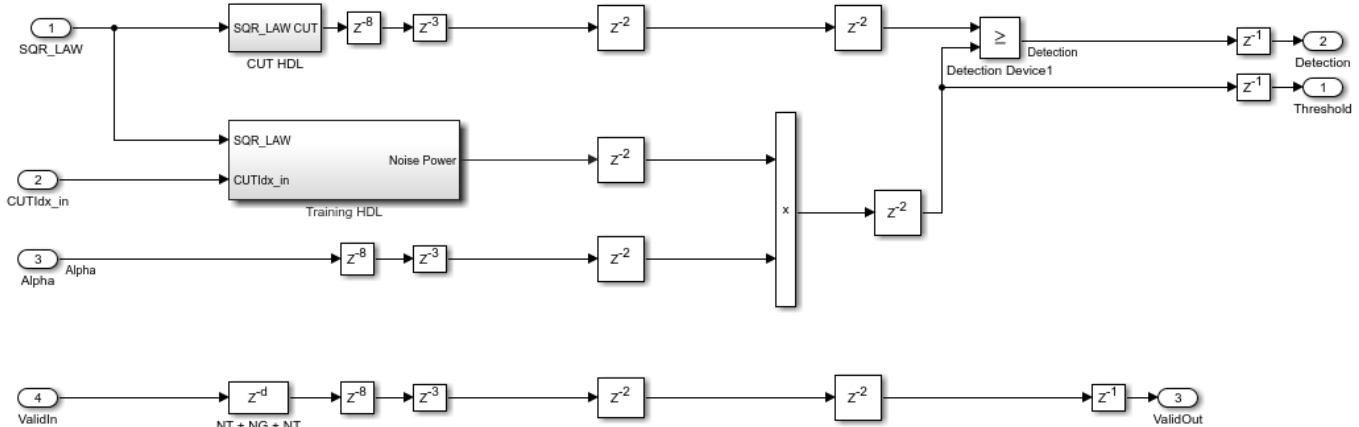
This subsystem extracts the training cells from the input stream and calculates the noise power. The noise power is then multiplied by alpha to generate the threshold which is later used in the detection process. There are two outputs from this block, namely, threshold and detection.

The threshold is the direct output of the product block whereas the detection output is provided through a comparator block which compares the threshold and signal value of CUT and returns true if the CUT signal is greater than the threshold.

This subsystem accounts for an input streaming delay of 102 (2*NumberTrainingCells + NoGuardCells) clock cycles with an additional pipelining latency of another 23 cycles. The total HDL latency is 125 cycles.

This block contains the following subsystems:

1    Training HDL
2    CUT HDL

**Training HDL**

The following command is used to open the Training HDL subsystem model.

```
open_system([modelname '/CFAR Implementation Model/CFAR Core/Training HDL']);
```
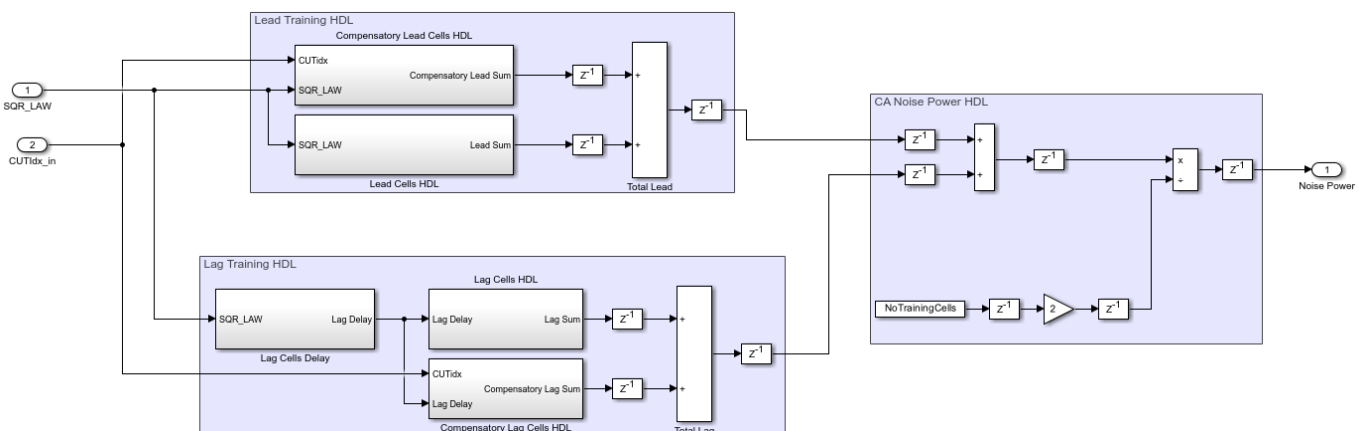
The lead training HDL subsystem extracts the lead cells of the CUT and performs a running sum. At the same time the lag training HDL subsystem pulls out the lag cells of the CUT and performs a running sum with a latency of 8 cycles each. The implementation of the lead training HDL and lag training HDL is very much analogous to a moving average filter the difference is that instead of the average we use the sum of window elements.

The CA noise power HDL subsystem sums up the lead power value and the lag power value and estimates the average noise power by dividing the sum by 100 (2*NoTrainingCells). This blocks accounts for a delay of 3 clock cycles.

The output of the training HDL subsystem is the noise power which is used to calculate the threshold.

**CUT HDL**

The following command is used to open the CUT HDL subsystem model

open_system([modelname '/CFAR Implementation Model/CFAR Core/CUT HDL']);
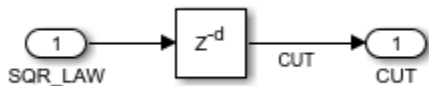


This subsystem uses a single delay block with a delay of 102 (2*NumberTrainingCells + NoGuardCells) cycles to time-align the CUT with the generated threshold value from the training HDL block. The above delay is the minimum delay required before which the CFAR Detector can detect the target at the first cell.

**Validate**

The following command is used to open the validate subsystem model

open_system([modelname '/CFAR Implementation Model/Validate']);
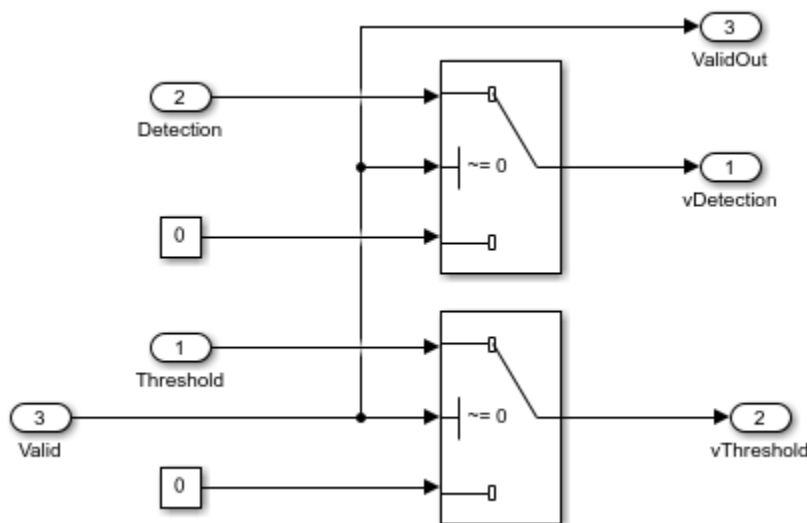


The valid input along with the latency is used to check the validity of the output. When the output is not valid this subsystem sends zero to the output.

**Comparing the Results of Implementation Model to the Behavioral Model**

The model can be simulated by clicking the Play button or using the sim command as shown below,

```
sim(modelname);
```
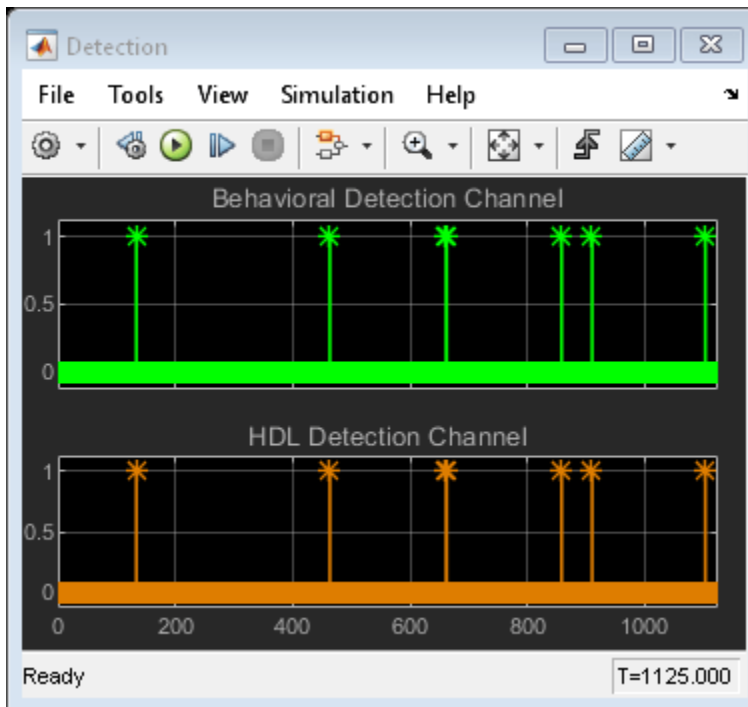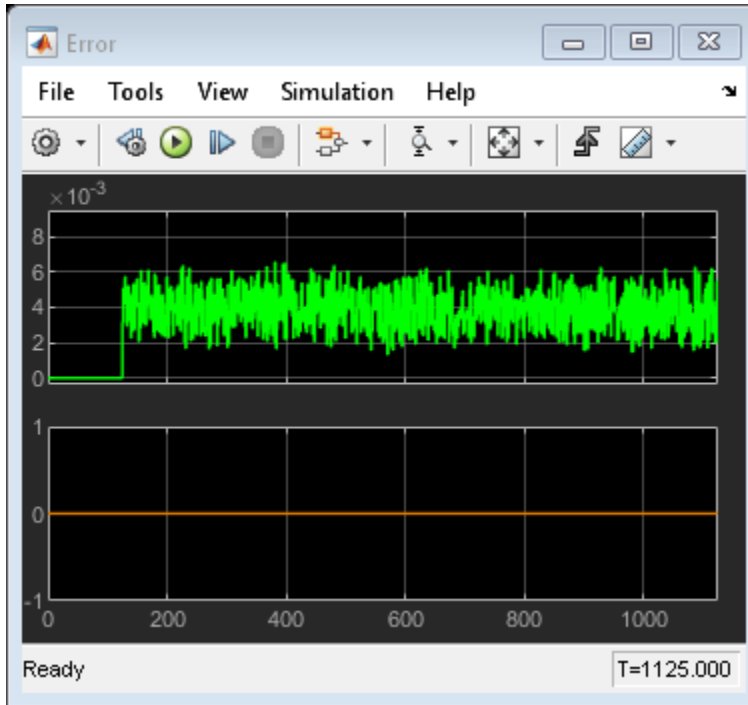
The Scope blocks are used to compare the output of implementation and behavioral model. The scope displays the detection and threshold from both the behavioral and implementation model and an additional scope displays the calculated the error.

The implementation model has a data streaming latency of 102 cycles and pipelining latency of 23 cycles. This in total accounts for an overall latency of 125 cycles. To time align the behavioral model with the implementation model we use an additional delay of 125 to the output of behavioral model.

With the 24 bit fixed-point of fraction length 12 bits, we have an error bounded by approximately 0.006 between the behavioral model and the implementation model threshold. Since the detection is Boolean we have no significant error in the detection output.

**Code Generation and Verification**

This section covers the procedure to perform HDL codegeneration for the implementation model. It also covers the verification that the generated code is functionally correct. The behavioral model provides the reference values to validate the output from HDL model.

If you start with a new model, you can run hdlsetup (HDL Coder) to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, *open* Simulink's **Model Settings** , *select* **Test Bench** under HDL Code Generation in the left panel, and *check* **HDL test bench** and **Co-simulation model** in the Test Bench Generation Output properties group.

**Model Settings**

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point, behavioral model, you can generate HDL code and test bench. For code generation and test bench, set the HDL Code Generation parameters in the **Configuration Parameters** dialog. The following parameters in Model Settings are set under HDL Code Generation:

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations
- **Global Settings:** Set the Reset type to Asynchronous
- **Test Bench:** Select HDL test bench, Co-simulation model and System Verilog DPI test bench

**HDL Code Verification via Co-Simulation**

After the model is set up, **HDL Workflow advisor** can be invoked to generate the HDL code using the HDL Coder also use the HDL Verifier to generate a System Verilog DPI Test Bench to test the model. To invoke HDL Workflow advisor *right-click* on the **CFAR Implementation model** subsystem and *navigate* to **HDL Code** and *left-click* **HDL Workflow advisor**. Instead of using HDL Workflow advisor the following lines of code can also be used to generate HDL code and System Verilog Test Bench.

```
% Uncomment the following two lines to generate HDL code and test bench.
% makehdl([modelname '/CFAR Implementation Model']);    % Generate HDL code
% makehdltb([modelname '/CFAR Implementation Model ']); % Generate Cosimulation test bench
```

Since all the optimizations are unchecked we do not have to add extra delays to the behavioral output other than the HDL latency previously added. (This is because all the critical paths are manually pipelined in the implementation model).

After generating HDL code and test bench a new Simulink model named gm_<modelname>_mq containing a ModelSim Simulator block is created in your working directory, which looks like this:
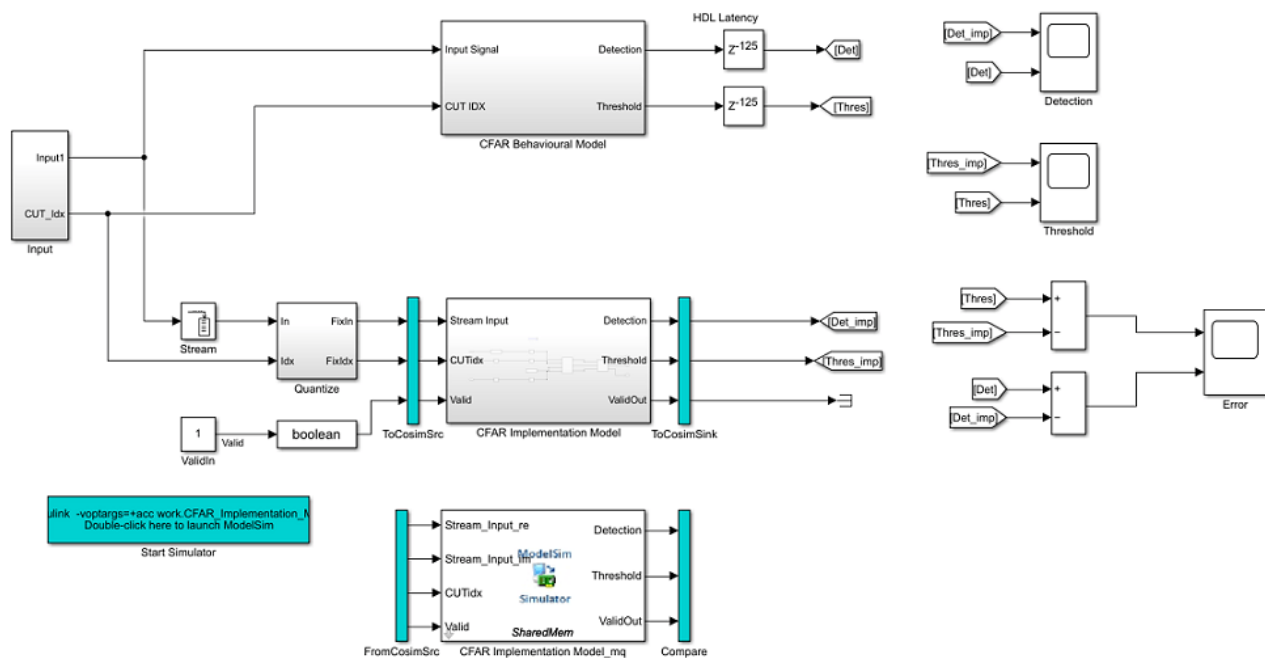


```
% To open the test bench model, uncomment the following lines of code
% modelname = ['gm_',modelname,'_mq'];
% open_system(modelname);
```

Launch ModelSim and run the co-simulation model to display the simulation results. You can click on the Play button on the top of Simulink canvas to run the test bench or you can do it via command window from the code below.

```
% Uncomment the following line, to run the test bench.
% sim(modelname);
```

The Simulink test bench model will populate the QuestaSim® with the HDL model's signal and Time Scopes in Simulink.
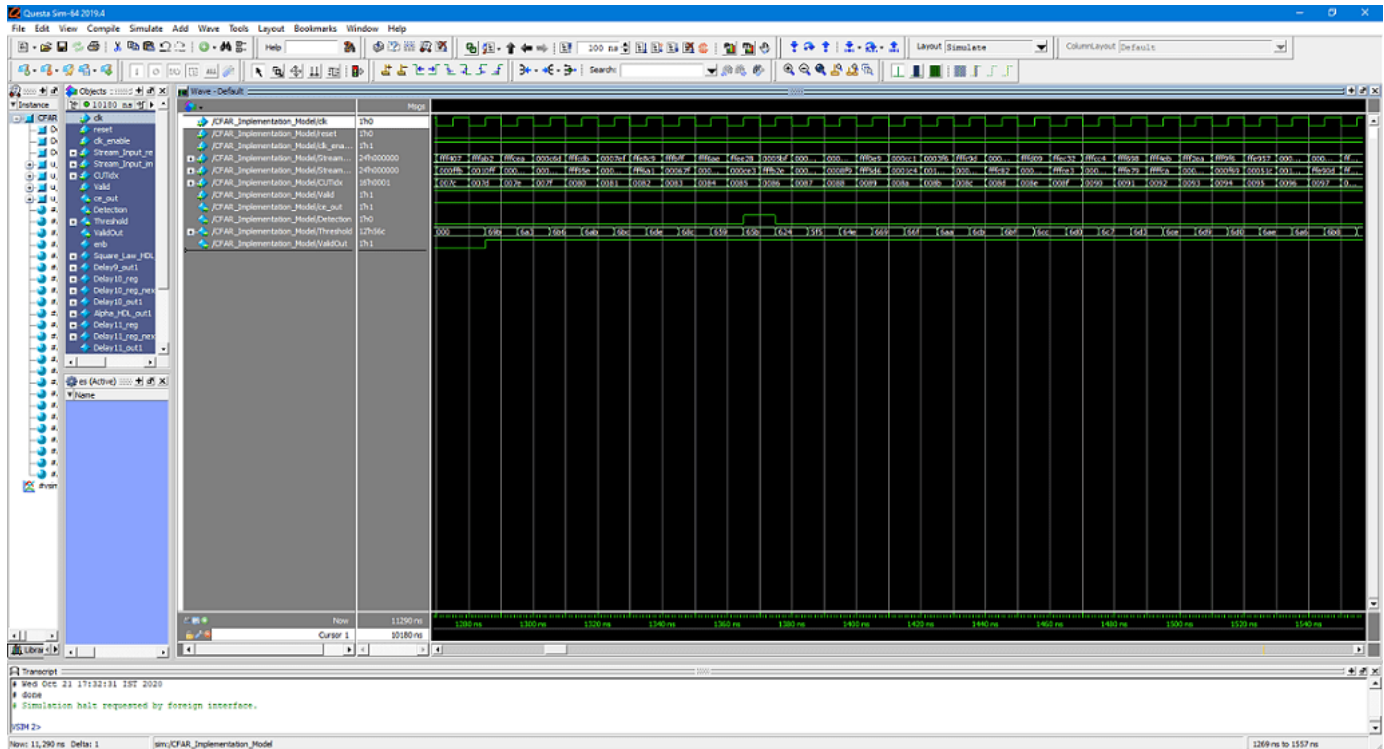


The Simulink scope shows detection output and threshold output for both the co-simulation and Design Under Test (DUT) as well as the error between them. The scopes comparing the results of the co-simulation can be found in test bench model inside the Compare subsystem, which is at the output of the CFAR_HDL_mq subsystem.
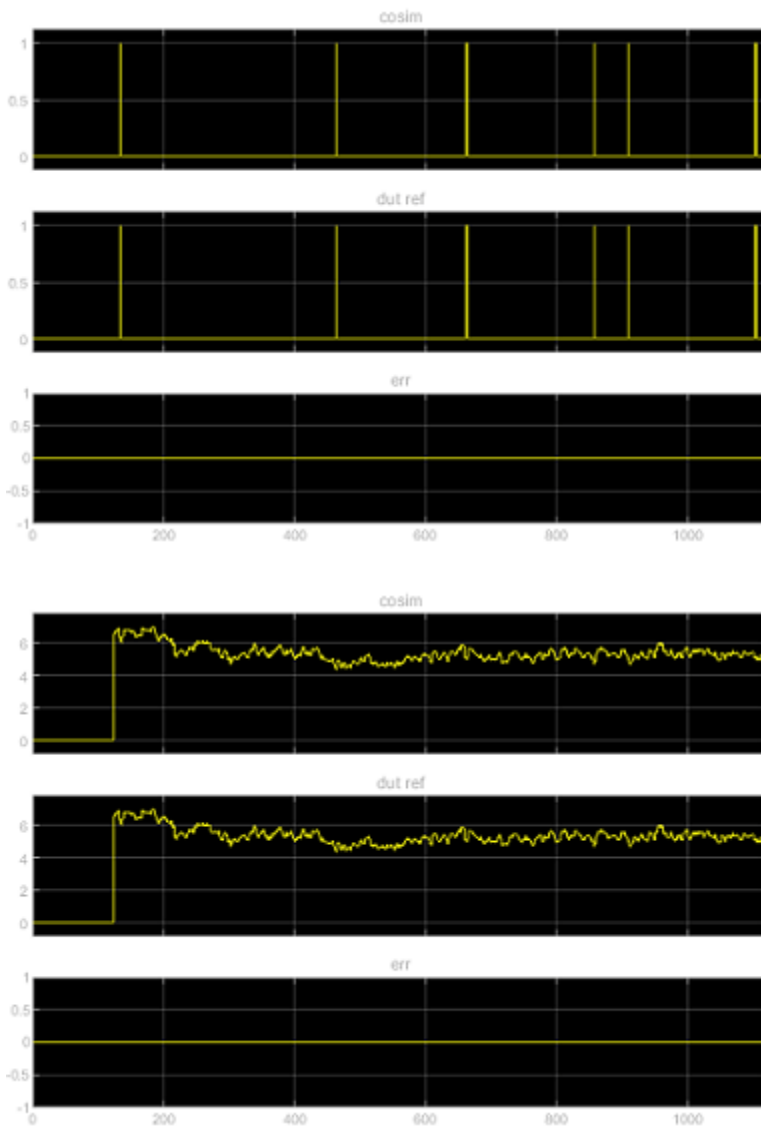
**Fixed-Point Word Length and Fraction Length Tradeoffs**

For this example a Fixed-Point word length of 24 bit and a fraction length of 12 bit is used for simulation, and implementation. The following figures show the trade-off with choosing a longer fraction length which would increase the precision (reduces the Error) but also increases the resource utilization.

The following plot shows fraction length associated with chosen word length.
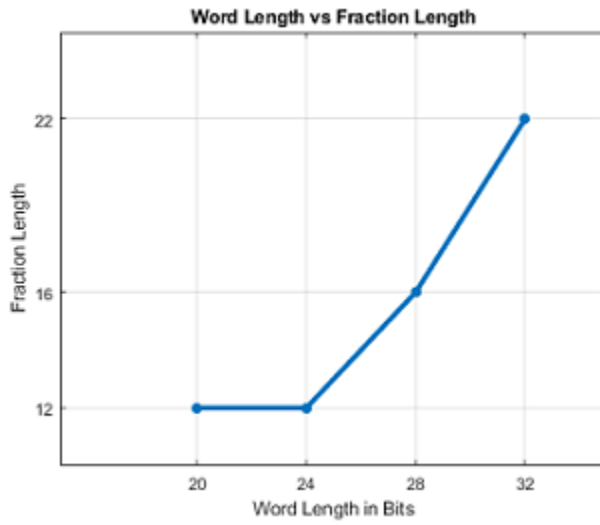
The following plot shows the Precision with respect to chosen word length.
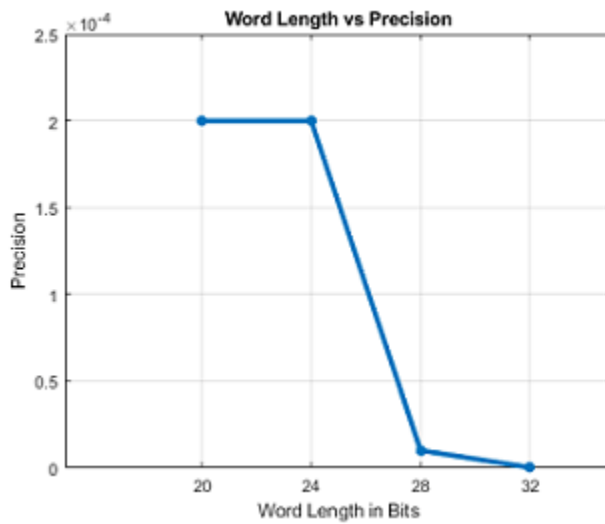


The following plot shows the Error with respect to chosen word length (Precision)
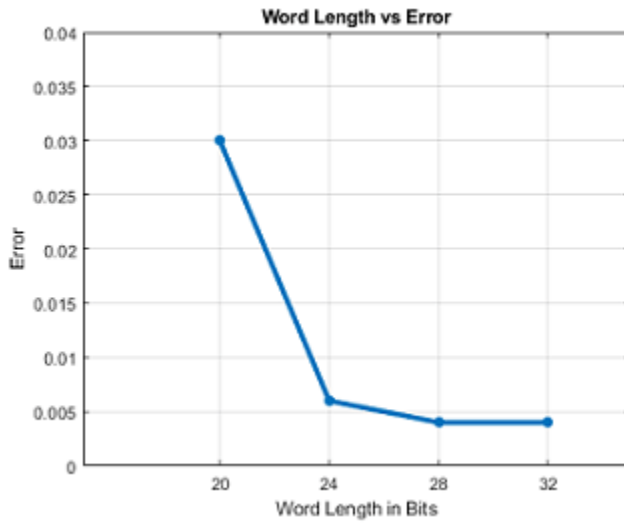
The following plots show the LUT/Registers/DSP Utilization with respect to chosen Word Length.

Word Length vs DSP Resource Utilization
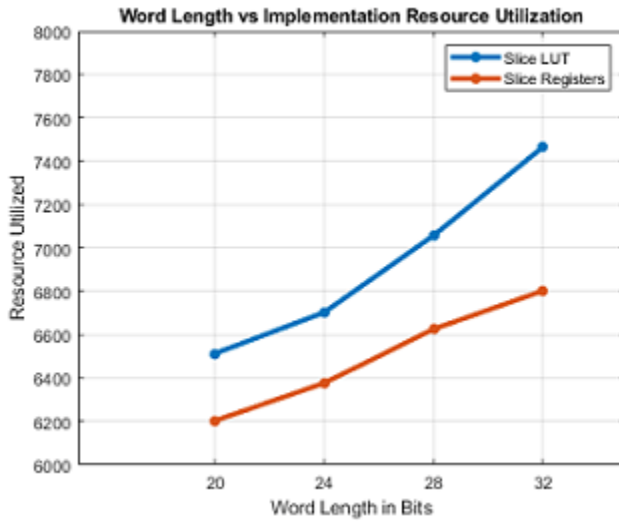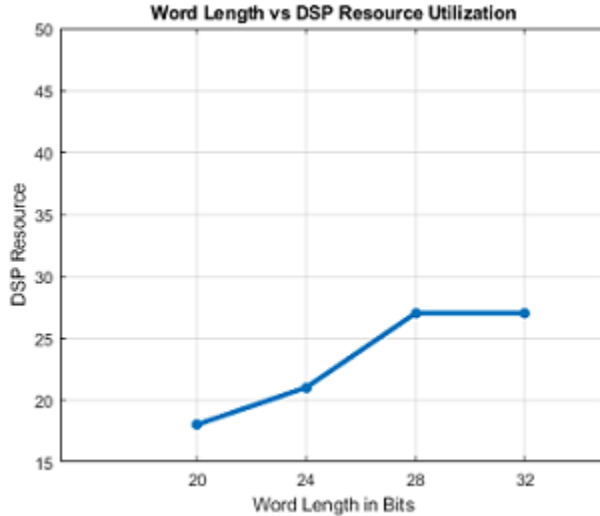
**Summary**

This example demonstrated how to design a Simulink model for a Cell Averaging Constant False Alarm Rate(CA CFAR) Detector, verify the results with an equivalent behavioral setup from the Phased Array System Toolbox. This example demonstrates how to automatically generate HDL code for a fixed-point equivalent algorithm and verify the generated code in Simulink. The generated HDL code as well as a co-simulation test bench for the Simulink subsystem was created with blocks that support HDL code generation. This example showed how to setup and launch ModelSim to cosimulate the HDL code and compare its output to the output generated by the HDL implementation model. The cosimulation is performed via ModelSim for the HDL code and compare results to the output generated by the HDL model.

# RF Propagation

# Access TIREM Software

The Terrain Integrated Rough Earth Model™ (TIREM™) is a propagation model for computing the path loss for irregular terrain and seawater scenarios. TIREM is developed, trademarked, and licensed by Huntington Ingalls Industries. To use TIREM, you need to acquire it from Huntington Ingalls Industries.

TIREM is designed to calculate the reference basic median propagation loss (path loss) based on the terrain profile along the great circle path between two antennas for example using digital terrain elevation data (DTED). You can use TIREM model to calculate the point-to-point path loss between sites over irregular terrain. The model combines physics with empirical data to provide path loss estimates. The TIREM propagation model can predict path loss at frequencies between 1 MHz and 1 THz.

Use `tiremSetup` to enable TIREM access from within MATLAB. The TIREM library folder contains the `tirem3` shared library. The full library name is platform dependent:

| Platform | Shared library name |
|---|---|
| Windows | `libtirem3.dll` or `tirem3.dll` |
| Linux | `libtirem3.so` |
| Mac | `libtirem3.dylib` |

# Featured Examples

# Antenna Array Analysis with Custom Radiation Pattern

This example shows how to create an antenna array with a custom antenna radiation pattern and then how to analyze the array's response pattern. Such a pattern can be obtained either from measurements or from simulations.

**Import the Radiation Pattern**

Depending on the application, practical phased antenna arrays sometimes use specially designed antenna elements whose radiation pattern cannot be represented by a closed-form equation. Even when the element pattern is well understood, as is the case with a dipole antenna, the mutual coupling among the elements can significantly change the individual element pattern when the element is put into an array. This makes the closed-form pattern less accurate. Therefore, for high fidelity pattern analysis, you often need to use a custom radiation pattern obtained from measurements or simulations.

There is no standard convention for the coordinate system used to specify the radiation pattern, so the result from one simulation package often cannot not be directly used in another software package. For example, in Phased Array System Toolbox™ (PST), the radiation pattern is expressed using azimuth (*az*) and elevation (*el*) angles, as depicted in Figure 1. It is assumed that the main beam of the antenna points toward $0°$ azimuth and $0°$ elevation, that is, the *x*-axis. The value of *az* lies between $-180°$ and $180°$ and the value of *el* lies between $-90°$ and $90°$. See "Spherical Coordinates" on page 10-10.
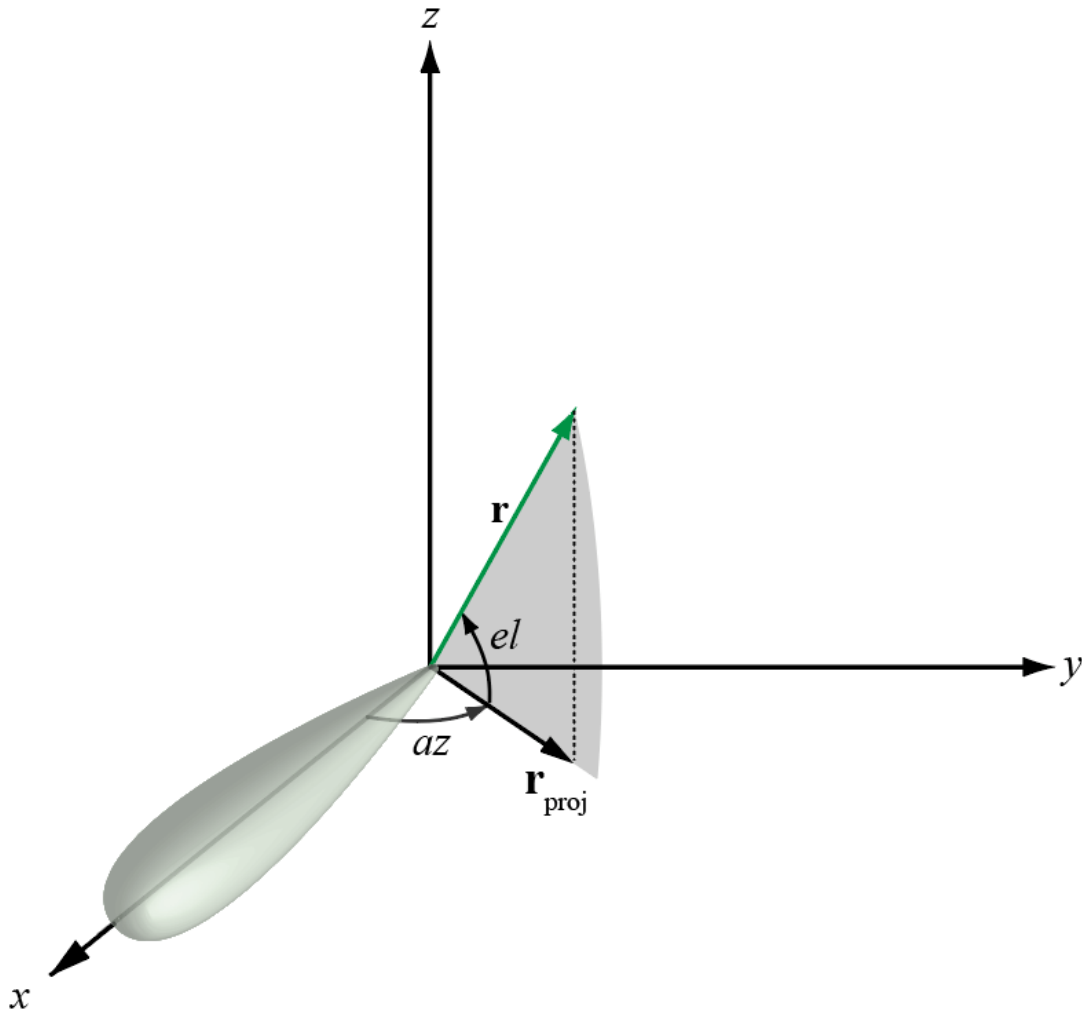
**Figure 1:** Spherical coordinate system convention used in Phased Array System Toolbox.

A frequently-used full-wave modeling tool for simulating antenna radiation patterns is HFSS™. In this tool, individual elements are modeled as if they were part of an infinite array. The simulated radiation pattern is represented as an $M$-by-3 matrix where the first column represents the azimuth angle $\phi$, the second column represents the elevation angle $\theta$, and the third column represents the radiation pattern in dB. The coordinate system and the definitions of $\theta$ and $\phi$ used in HFSS is shown in Figure 2. In this convention, the main beam of the antenna points along the $z$-axis which usually points vertically. The value of $\phi$ is between $0°$ and $360°$ and the value of $\theta$ is between $0°$ and $180°$.
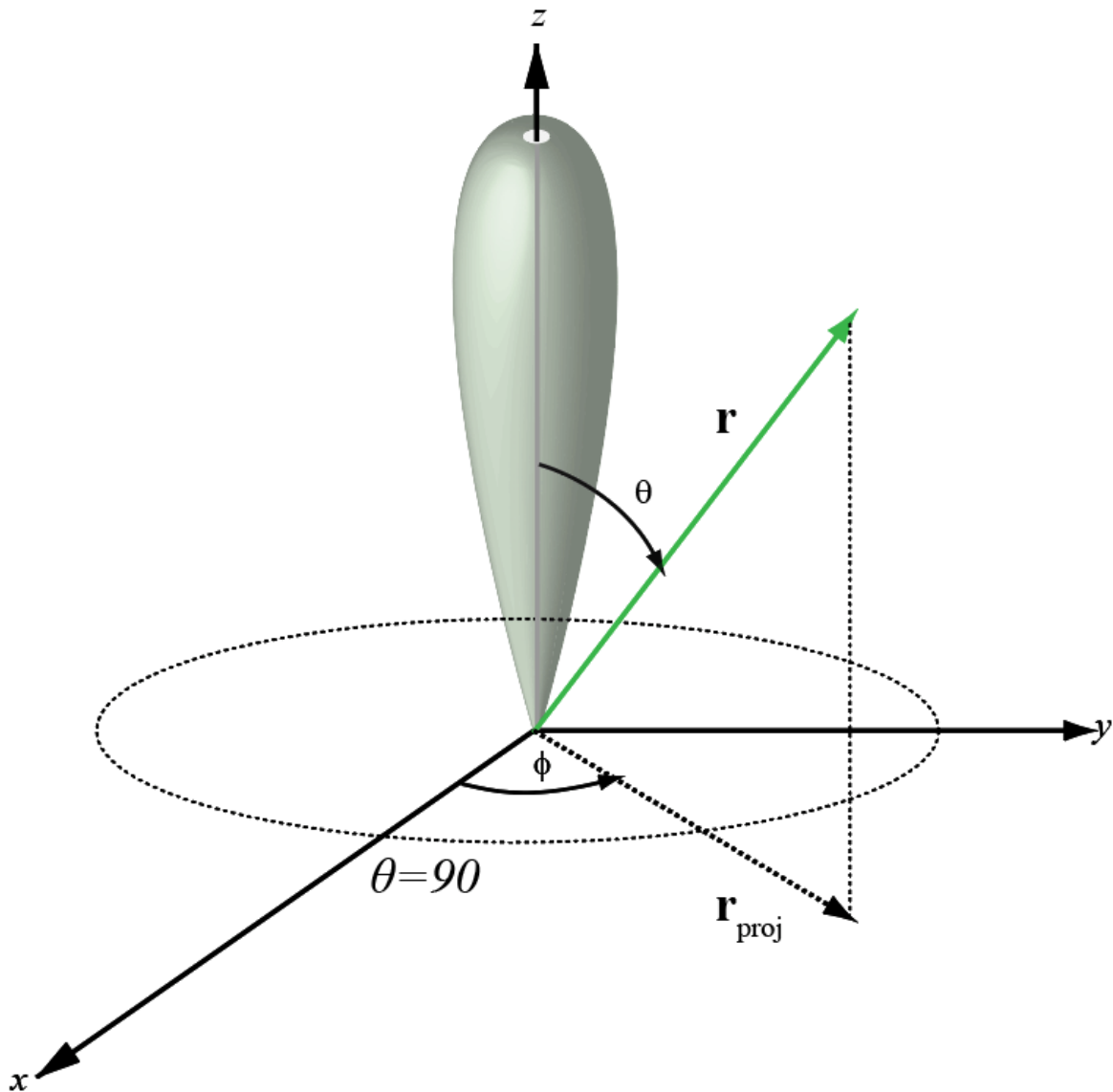
**Figure 2:** Spherical coordinate system convention used in HFSS.

Note that the HFSS coordinate system is not exactly the same as the $\phi - \theta$ coordinate system used in Phased Array System Toolbox™. In HFSS, the beam mainlobe points along the $z$-axis and the plane orthogonal to the beam is formed from the $x$- and $y$ axes. One possible approach to import a custom pattern in $\phi - \theta$ convention without any coordinate axes rotation is shown below.

For example, a cardioid-shaped antenna pattern is simulated in the $\phi$-$\theta$ convention and is saved in a `.csv` file. The helper function `helperPatternImport` reads the `.csv` file and reformats its contents into a two-dimensional matrix in $\phi$ and $\theta$.

```
[pattern_phitheta,phi,theta] = helperPatternImport;
```

The phi-theta pattern can now be used to form a custom antenna element. Assume that this antenna operates between 1 and 1.25 GHz.

```
freqVector  = [1 1.25].*1e9;          % Frequency range for element pattern
antenna     = phased.CustomAntennaElement('FrequencyVector',freqVector, ...
                        'PatternCoordinateSystem','phi-theta',...
                        'PhiAngles',phi,...
                        'ThetaAngles',theta,...
                        'MagnitudePattern',pattern_phitheta,...
                        'PhasePattern',zeros(size(pattern_phitheta)));
```

To verify that the pattern has been correctly imported, plot the response of the custom antenna element. Notice that the main beam points to $0°$ azimuth and $90°$ elevation, the custom pattern with main beam along z-axis is imported without any rotation.

```
fmax = freqVector(end);
pattern(antenna,fmax,'Type','powerdb')
```
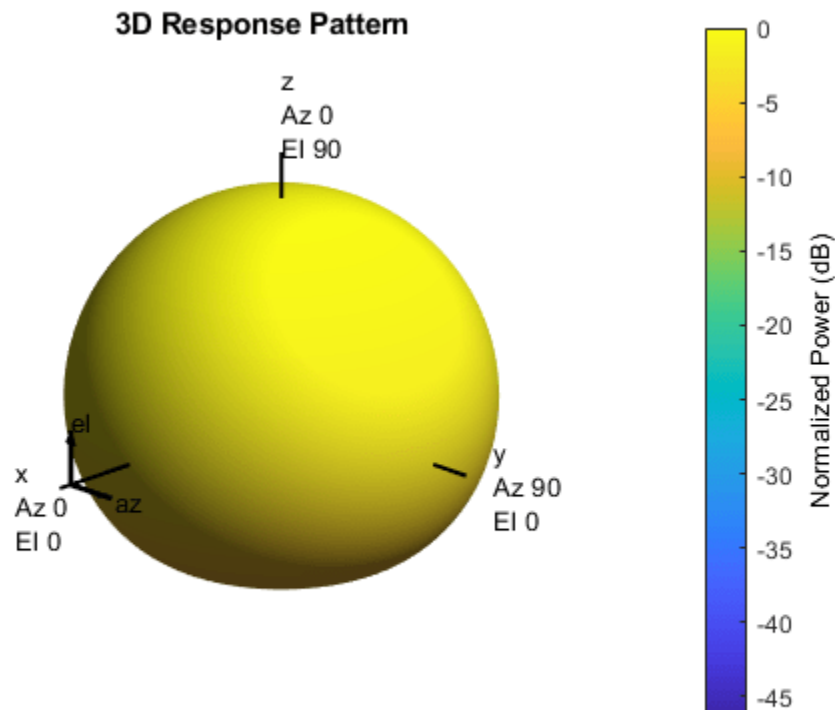
**Construct Antenna Array**

Consider a 100-element antenna array whose elements reside on a 10-by-10 rectangular grid, as shown in Figure 3. To ensure that no grating lobes appear, elements are spaced at one-half wavelength at the highest operating frequency. This rectangular array can be created using the following commands.
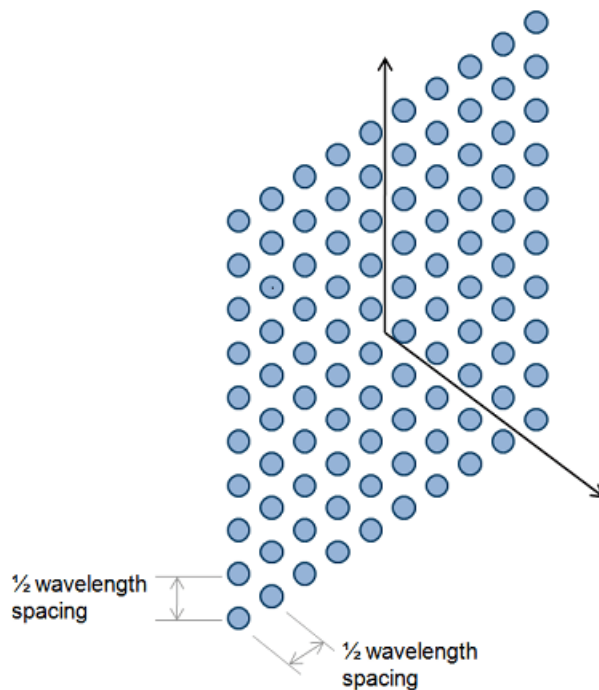


**Figure 3:** A 10-by-10 URA.

```
c = physconst('LightSpeed');
lambda = c/fmax;
array = phased.URA('Element',antenna,'Size',10,'ElementSpacing',lambda/2)
```

```
array =

  phased.URA with properties:

           Element: [1x1 phased.CustomAntennaElement]
              Size: [10 10]
    ElementSpacing: [0.1199 0.1199]
           Lattice: 'Rectangular'
       ArrayNormal: 'x'
             Taper: 1
```

The total radiation pattern of the resulting antenna array is plotted below in *u-v* space. The pattern is a combination of both the element pattern and the array factor.

```
pattern(array,fmax,'PropagationSpeed',c,'Type','powerdb',...
    'CoordinateSystem','UV');
```

3D Response Pattern in u-v space
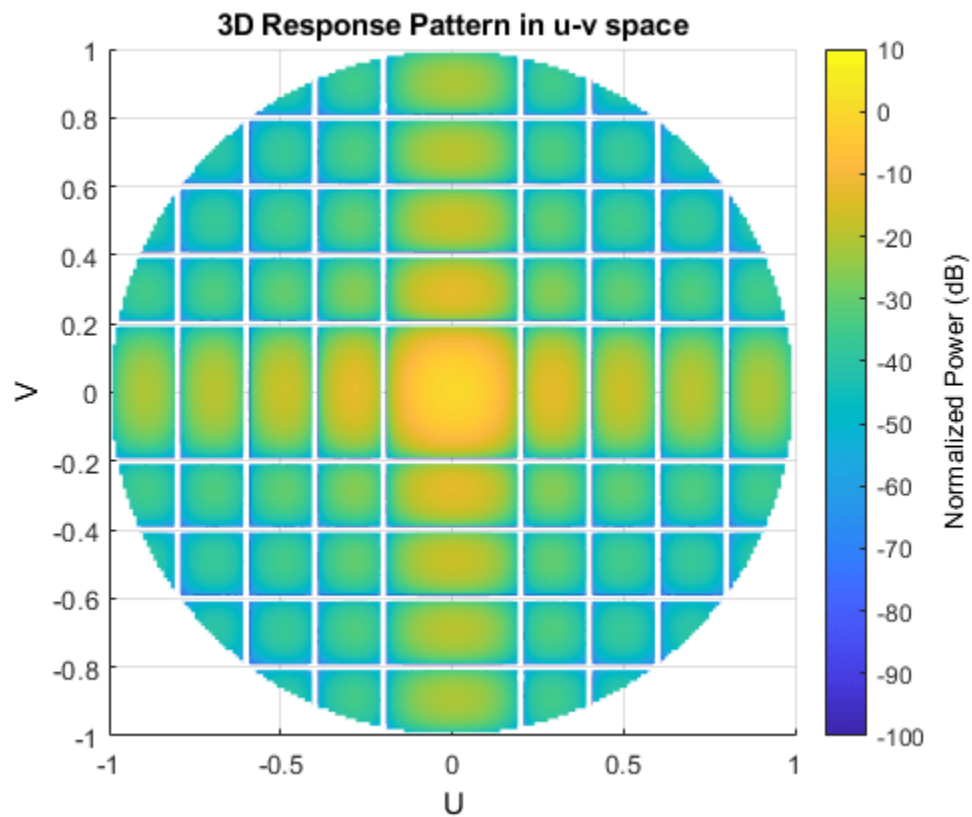
One can also easily examine the *u*-cut of the pattern as shown below.

```
pattern(array,fmax,-1:0.01:1,0,'PropagationSpeed',c, ...
    'CoordinateSystem','UV','Type','powerdb')
axis([-1 1 -50 0]);
```

Response in U Space
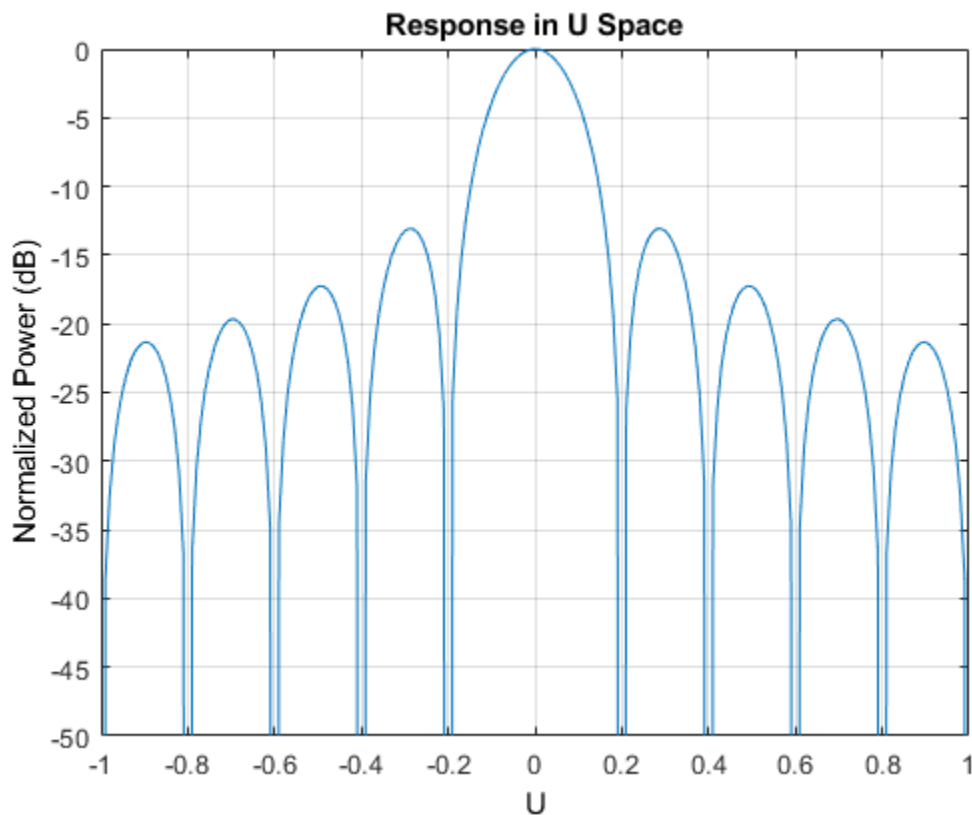
**Steer the Beam in Azimuth**

This section illustrates the idea of phase steering an array. An advantage of phased arrays over a single antenna element is that the main beam can be electronically steered to a given direction. Steering is accomplished by adjusting the weights assigned to each element. The set of weights is also called the steering vector. Each weight is a complex number whose magnitude controls the sidelobe characteristics of the array and whose phase steers the beam.

The example scans the main beam of the array from $-30°$ azimuth to $30°$ azimuth, with the elevation angle fixed at $0°$ during the scan.

```
helperPatternScan(array)
```
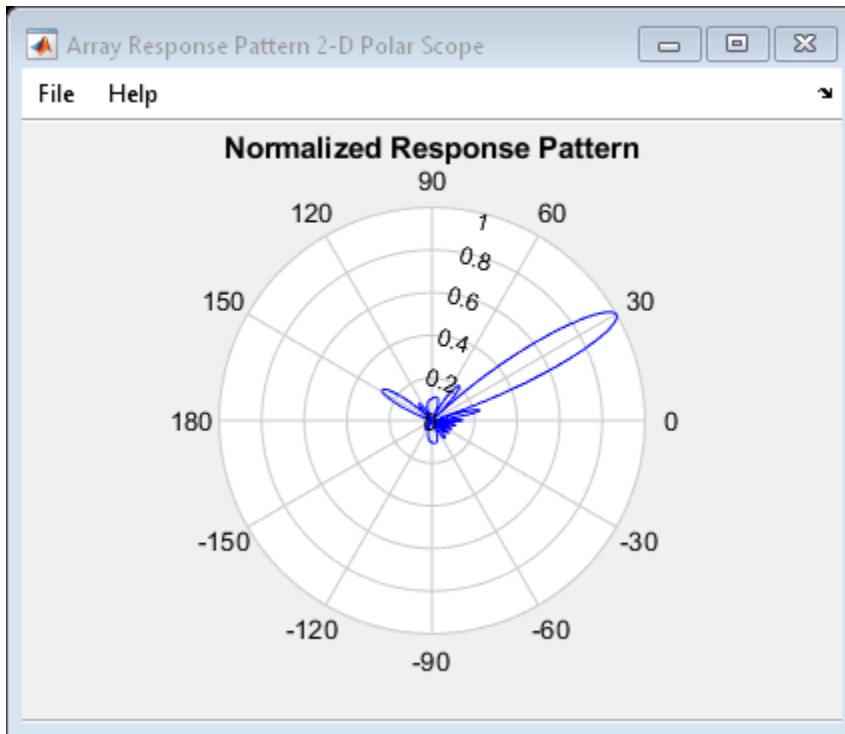
```
clear helperPatternScan
```

**Summary**

This example shows how to construct and analyze an antenna array using a custom antenna pattern. The pattern can be generated using full-wave modeling simulation software with the $\phi$ - $\theta$ convention. The pattern can then be used to form a custom antenna element. The resulting array is scanned from $-30°$ to $30°$ in the azimuth direction to illustrate the phase steering concept.

# Array Pattern Synthesis Part I: Nulling, Windowing, and Thinning

This example shows how to use Phased Array System Toolbox™ to solve some array synthesis problems.

In phased array design applications, it is often necessary to find a way to taper element responses so that the resulting array pattern satisfies certain performance criteria. Typical performance criteria include the mainlobe location, null location(s) and sidelobe levels.

**Interference Removal Using Sidelobe Canceller**

A common requirement when synthesizing beam patterns is pointing a null towards a given arrival direction. This helps suppress interference from that direction and improves the signal-to-interference ratio. The interference is not always malicious- an airport radar system may need to suppress interference from a nearby radio station. In this case, the position of the radio station is known and a sidelobe cancellation algorithm can be used to remove the interference.

Sidelobe cancellation is useful for suppressing interference that enters through the array's sidelobes. In this case, because the interference direction is known, the algorithm is simple. Form a beam that points towards the interference direction, then scale the beam weights and subtract scaled weights from the weights for the beam patterns that point towards any other look direction. This process always places a strong null in the interference direction.

The following example shows how to design the weights of the radar so that it scans between -30 and 30 degrees yet always keeps a null at 40 degrees. Assume that the radar uses a 10-element ULA that is parallel to the ground and that the known radio interference arrives from 40 degrees azimuth.

```
c = 3e8;          % signal propagation speed
fc = 1e9;         % signal carrier frequency
lambda = c/fc;    % wavelength

thetaad = -30:5:30;        % look directions
thetaan = 40;              % interference direction

ula = phased.ULA(10,lambda/2);
ula.Element.BackBaffled = true;

% Calculate the steering vector for null directions
wn = steervec(getElementPosition(ula)/lambda,thetaan);

% Calculate the steering vectors for lookout directions
wd = steervec(getElementPosition(ula)/lambda,thetaad);

% Compute the response of desired steering at null direction
rn = wn'*wd/(wn'*wn);

% Sidelobe canceler - remove the response at null direction
w = wd-wn*rn;

% Plot the pattern
pattern(ula,fc,-180:180,0,'PropagationSpeed',c,'Type','powerdb',...
    'CoordinateSystem','rectangular','Weights',w);
hold on; legend off;
```

```
plot([40 40],[-100 0],'r--','LineWidth',2)
text(40.5,-5,'\leftarrow Interference Direction','Interpreter','tex',...
    'Color','r','FontSize',10)
```
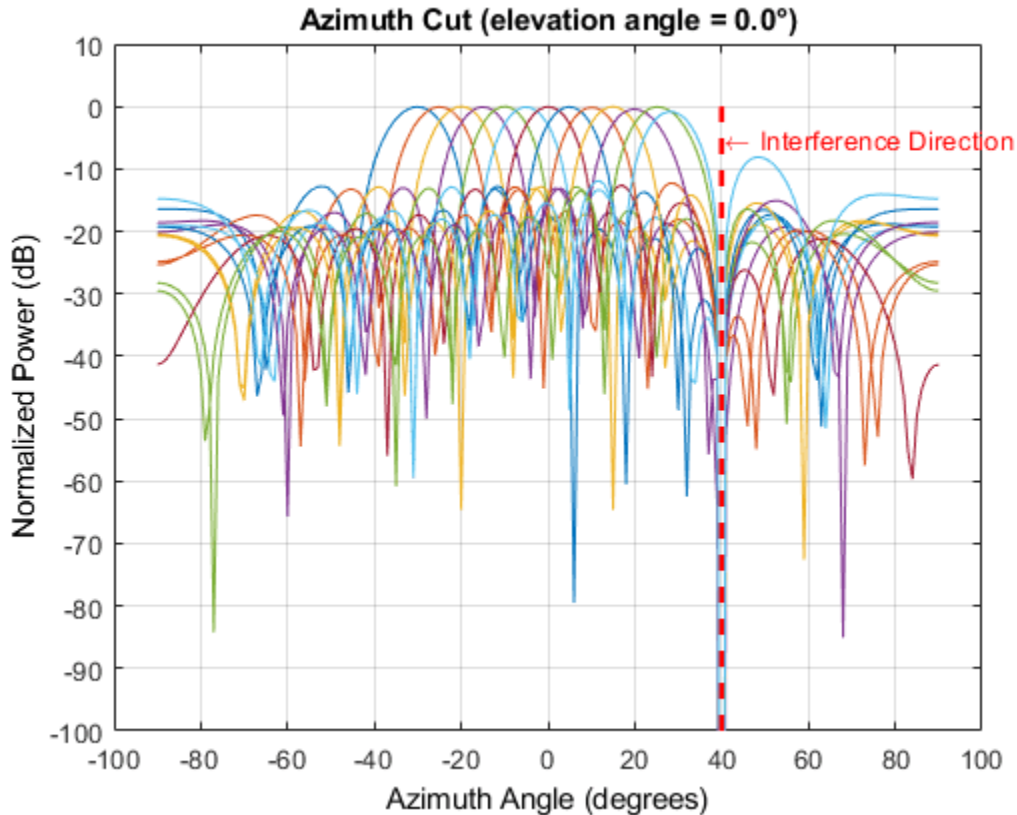


The figure above shows the resulting beam patterns for look directions from -30 degrees azimuth to 30 degrees azimuth, in 5 degrees increment. It is clear from the zoomed figure below that no matter where the look direction is, the radar beam pattern has a strong null at the interference direction.

```
% Zoom
xlim([30 50])
legend(arrayfun(@(k)sprintf('%d degrees',k),thetaad,...
    'UniformOutput',false),'Location','SouthEast');
```

**Azimuth Cut (elevation angle = 0.0°)**
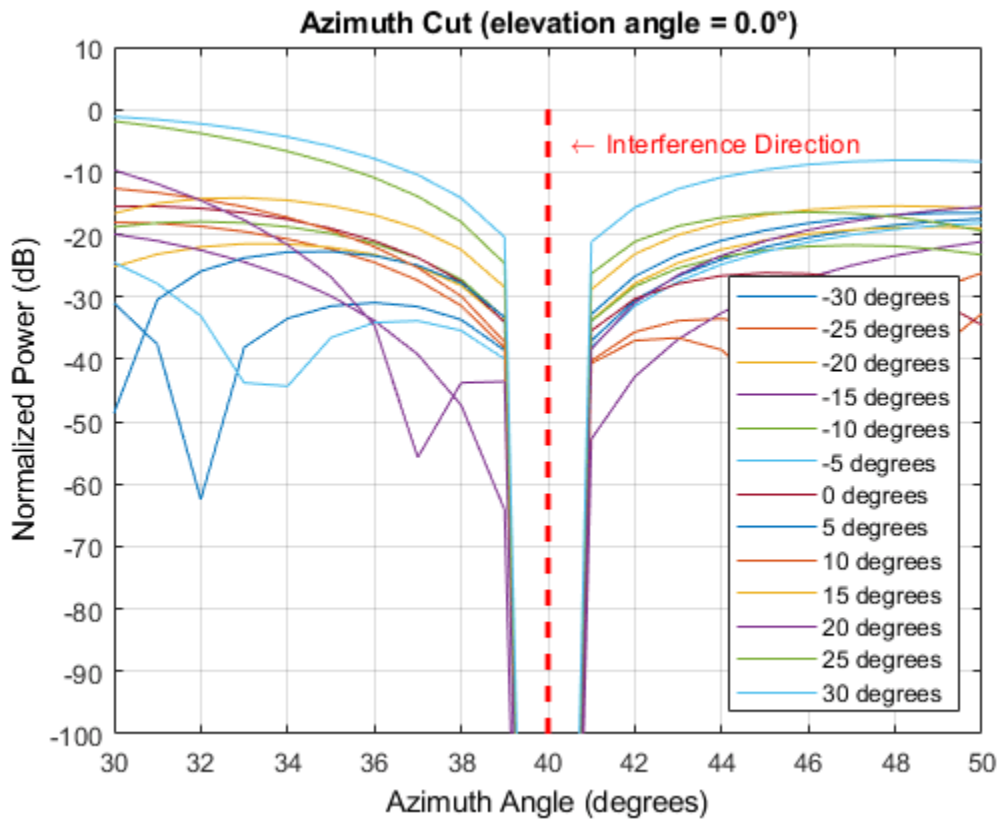
## Pattern Synthesis Using Windowing Function

Another frequent problem when designing a phased array is matching a desired beam pattern to a specification that is handed to you. Often, the requirements are expressed in terms of beamwidth and sidelobe level.

The process of addressing such a problem often includes these steps:

**1**  Observe the desired pattern and decide an array geometry;

**2**  Choose an array size based on the desired beamwidth;

**3**  Design tapers based on the desired sidelobe level;

**4**  Iterate on adjusting the parameter obtained step 2 and 3 to get a best match.

The following example illustrates these four steps. First, observe the desired pattern shown in the following figure.

```
load desiredSynthesizedAntenna;

clf;
pattern(mysteryAntenna,fc,'CoordinateSystem','polar','Type','powerdb');
view(50,20);
ax = gca;
ax.Position = [-0.15 0.1 0.9 0.8];
camva(4.5);
campos([520 -250 200]);
```
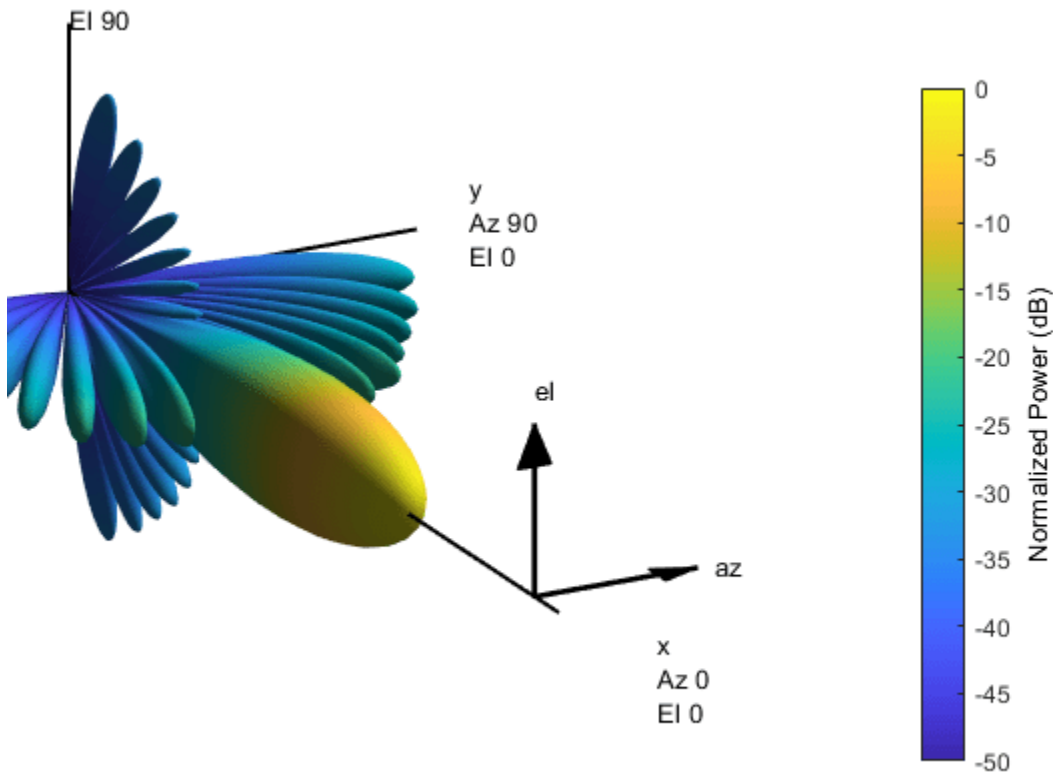
The 3D radiation patterns exhibits some symmetries in both azimuth and elevation cuts. Therefore, the pattern may be best obtained using a uniform rectangular array (URA). It is also clear from the plot that there is no energy radiated toward back of the array.

Next, determine the size of the array. To avoid grating lobes, the element spacing is set to half wavelength. For a URA, the sizes along the azimuth and elevation directions can be derived from the required beamwidths along azimuth and elevation directions, respectively. In the case of half wavelength spacing, the number of elements along a certain direction can be approximated by

$$N \simeq \frac{2}{\sin(\theta_b)}$$

where $\theta_b$ is the beamwidth along that direction. Hence, the aperture size of the URA can be computed as

```
[azpat,elpat,az,el] = helperExtractSynthesisPattern(mysteryAntenna,fc,c);

% Azimuth direction
idx = find(azpat>pow2db(1/2));
azco = [az(idx(1)) az(idx(end))]; % azimuth cutoff
N_col = round(2/sind(diff(azco)))

% Elevation direction
idx = find(elpat>pow2db(1/2));
elco = [el(idx(1)) el(idx(end))]; % elevation cutoff
N_row = round(2/sind(diff(elco)))
```

```
N_col =

    19


N_row =

    14
```

The estimation suggests to start with a 14x19 URA.

```matlab
% Form the URA
ura = phased.URA([N_row N_col],[lambda/2 lambda/2]);
ura.Element.BackBaffled = true;

helperArraySynthesisComparison(ura,mysteryAntenna,fc,c)
```
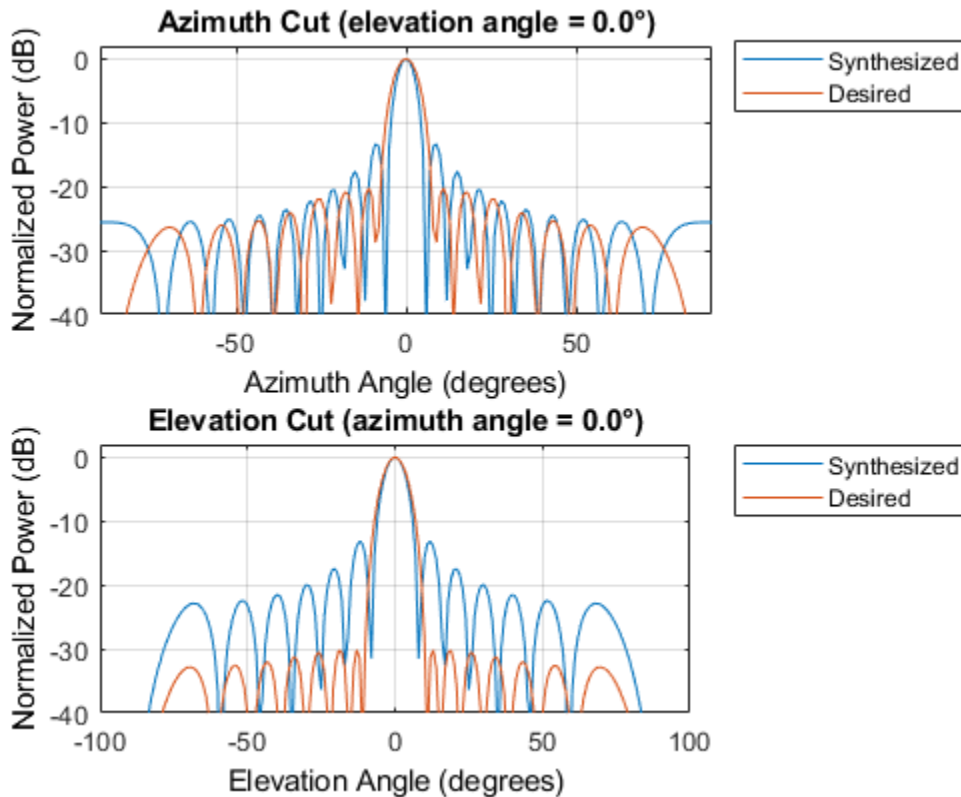


The figure shows that the synthesized array exceeds the beamwidth requirement of the desired pattern. However, the sidelobes are much larger than the desired pattern. You can reduce the sidelobes by applying a windowing operation to the array. Because the URA can be considered to be the combination of two separable uniform linear arrays (ULA), the window can be designed independently along both the azimuth and elevation directions using familiar filter design methods.

The code below shows how to obtain the windows for azimuth and elevation directions.

```
AzSidelobe = 20;
Ap = 0.1;                               % Passband ripples
AzWeights = designfilt('lowpassfir','FilterOrder',N_col-1,...
    'CutoffFrequency',azco(2)/90,'PassbandRipple',0.1,...
    'StopBandAttenuation',AzSidelobe);
azw = AzWeights.Coefficients;

ElSidelobe = 30;
ElWeights = designfilt('lowpassfir','FilterOrder',N_row-1,...
    'CutoffFrequency',elco(2)/90,'PassbandRipple',0.1,...
    'StopBandAttenuation',ElSidelobe);
elw = ElWeights.Coefficients;

% Assign the weights to the array
ura.Taper = elw(:)*azw(:).';

% Compare the pattern
helperArraySynthesisComparison(ura,mysteryAntenna,fc,c)
```
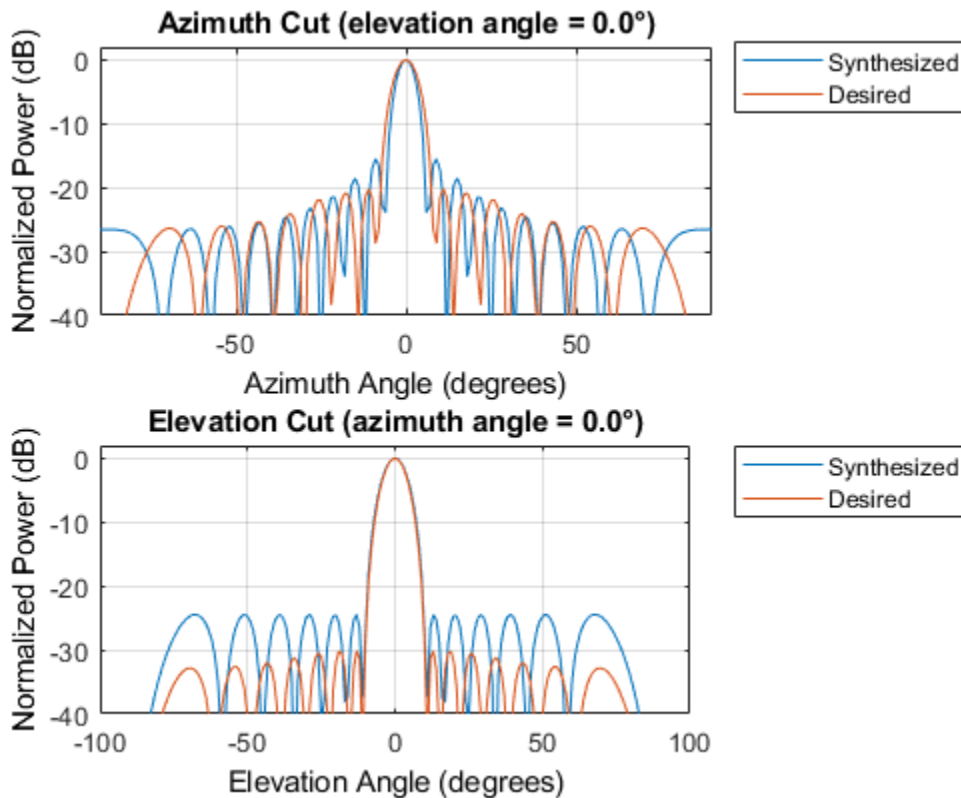


The figure shows that the resulting sidelobe level is lower compared to the previous design but still does not satisfy the requirement. By some trials and errors, the following parameters are used to create the final design:

```
N_row = N_row+2;        % trial and error
N_col = N_col-3;        % trial and error
AzSidelobe = 26;
ElSidelobe = 35;
```

```
AzWeights = designfilt('lowpassfir','FilterOrder',N_col-1,...
    'CutoffFrequency',azco(2)/90,'PassbandRipple',0.1,...
    'StopBandAttenuation',AzSidelobe);
azw = AzWeights.Coefficients;

ElWeights = designfilt('lowpassfir','FilterOrder',N_row-1,...
    'CutoffFrequency',elco(2)/90,'PassbandRipple',0.1,...
    'StopBandAttenuation',ElSidelobe);
elw = ElWeights.Coefficients;

ura = phased.URA([N_row N_col],[lambda/2 lambda/2]);
ura.Element.BackBaffled = true;

ura.Taper = elw(:)*azw(:).';

helperArraySynthesisComparison(ura,mysteryAntenna,fc,c)
```
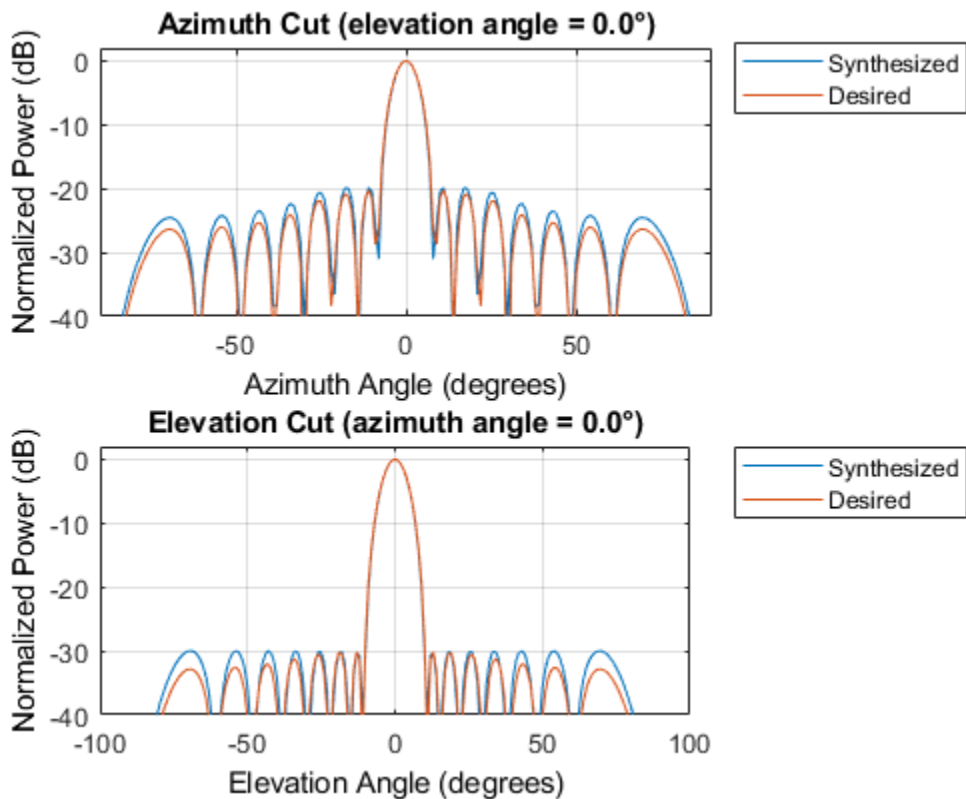


The figure shows that the beamwidth and sidelobe levels of the synthesized pattern match the desired specifications. The following figures show the desired 3D pattern, the synthesized 3D pattern, the resulting array geometry, and the taper.

```
helperArraySynthesisComparison(ura,mysteryAntenna,fc,c,'3d')
```
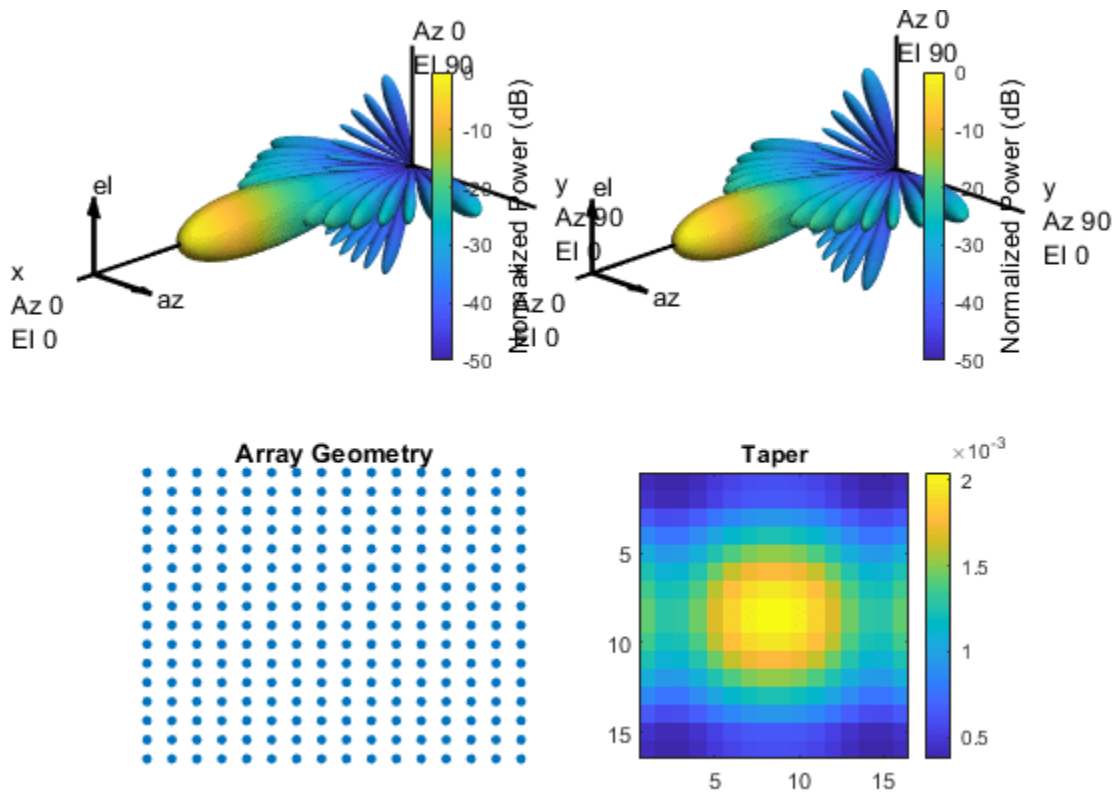
## Array Thinning Using Genetic Algorithm

Many array synthesis problems can be treated as optimization problems, especially for arrays with large apertures or complex geometries. In those situations, a closed form solution often does not exist and the solution space is very large. For example, for a large array, it is often necessary to thin the array to control the sidelobe levels to avoid wasting power delivered to each antenna element. In this case, an element can be turned on or off. If you were to try all possible solutions in a 400-element array, you would need to try $2^{400}$ combinations, which is unrealistic, and a 400-element array is not considered to be a big aperture at all. Optimization techniques are often adopted in this situation.

A frequently used optimization technique is the genetic algorithm. A genetic algorithm achieves the optimal solution by simulating the natural selection process. It starts with randomly selected candidates as the first generation. At each evolution cycle, the algorithm sorts the generation according to a predetermined performance measure (in the thinned array example, the performance measure would be the ratio of peak-to-sidelobe level), and then discards the ones with lower performance scores. The algorithm then mutates the remaining candidates to generate a newer generation and repeats the process, until it reaches a stop condition, such as the maximum number of generations.

The following example shows how to use a genetic algorithm to thin a 40x40 URA. The goal is to achieve maximum sidelobe suppression in both azimuth and elevation cut. The beam pattern of the full array is shown first.

```
Nside = 40;
geneticArray = phased.URA(Nside,lambda/2);
geneticArray.Element.BackBaffled = true;
```

**18-17**

```
clf;
wplot = helperThinnedArrayComparison(geneticArray,fc,c);
```
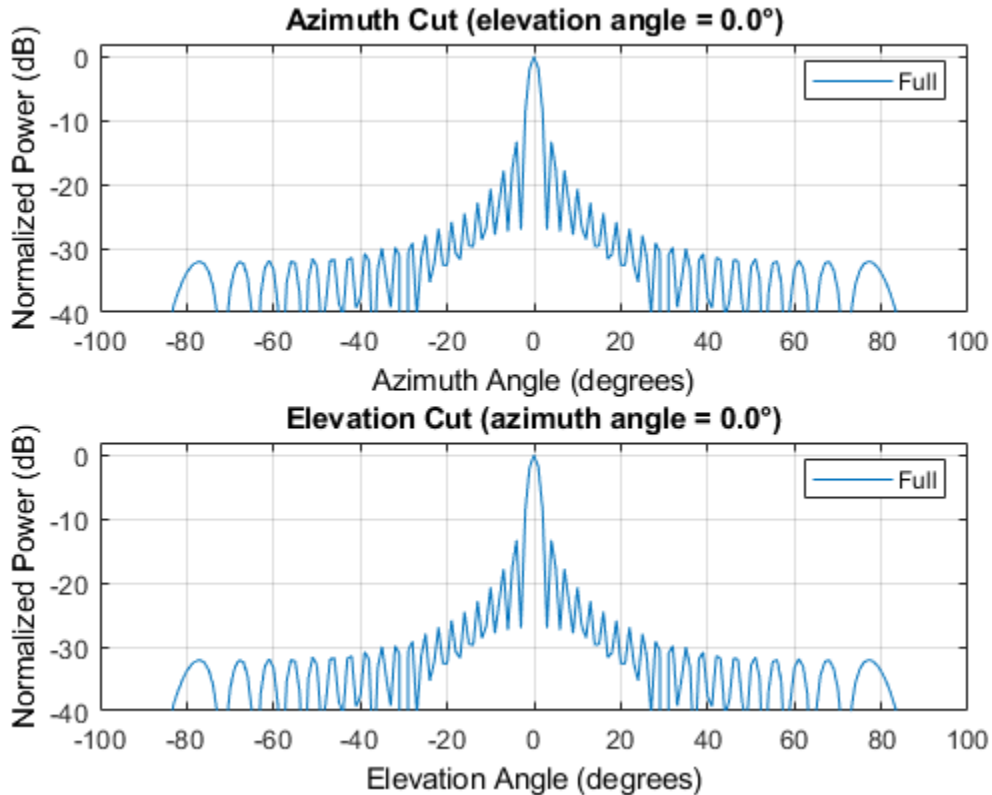


The sidelobe level can be computed as

```
% Compute beam pattern
[azpat,elpat,az,el] = helperExtractSynthesisPattern(geneticArray,fc,c);

% Compute relative sidelobe level
pks_az = findpeaks(azpat,'NPeaks',2,'SortStr','descend');
pks_el = findpeaks(elpat,'NPeaks',2,'SortStr','descend');

% Find the smaller sidelobe level between two cuts
sllopt = min(pks_az(1)-pks_az(2),pks_el(1)-pks_el(2))
```

```
sllopt =

   13.2981
```

As expected, the sidelobe level is about 13dB.

Now apply the genetic algorithm. Notice that the URA has symmetry in both rows and columns, thus one can take advantage of this symmetry so that each thinning coefficients candidate applies to only a quarter of the array. This reduces the search space of the algorithm.

```matlab
% Set random seed
prvS = rng(2013);

% Initial generation with 200 candidates. Initially, the elements toward
% the center are active and the dummy elements are toward the edge
w0 = double(rand(Nside/2,Nside/2,200)>0.5);
w0(1:14,1:14,:) = 1;

% Pick one candidate, plot the beam pattern, and compute the sidelobe
% level
wtemp = w0(:,:,100);
wo = [fliplr(wtemp) wtemp;rot90(wtemp,2) flipud(wtemp)];
wplot = helperThinnedArrayComparison(geneticArray,fc,c,[wplot wo(:)],...
    {'Full','Initial'});
```
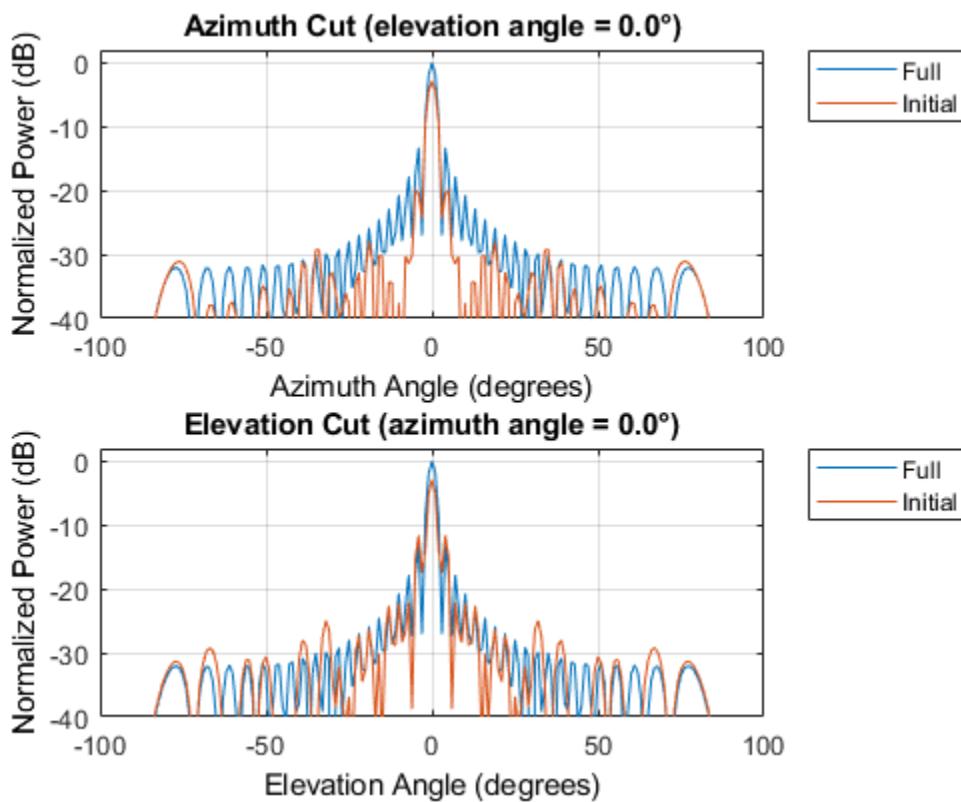


The figure shows the beam pattern resulted from one typical first generation candidate. The sidelobe level is lower in azimuth direction but higher in elevation direction compared to the full array. The exact sidelobe level and the fill rate of the array can be computed as

```matlab
[azpat,elpat] = helperExtractSynthesisPattern(geneticArray,fc,c,wo(:));

% Compute relative sidelobe level
pks_az = findpeaks(azpat,'NPeaks',2,'SortStr','descend');
pks_el = findpeaks(elpat,'NPeaks',2,'SortStr','descend');
sllopt = min(pks_az(1)-pks_az(2),pks_el(1)-pks_el(2))

fillrate = sum(wo(:))/Nside^2*100
```

```
sllopt =

    8.7013


fillrate =

   71.7500
```

This means that 71.75% of the array elements (1148 of them) are active and the sidelobe level is about 9 dB. It needs to be suppressed further. The code below applies genetic algorithm with 30 generations.

```matlab
% Max number of generations
Niter = 30;

% Number of candidates in each generation
numGene = size(w0,3);

w = w0;
pos = getElementPosition(geneticArray)/lambda;
angspan = -90:90;
for m = 1:Niter
    % Compute the beam pattern for the entire generation
    [azpat,elpat] = helperArraySynthesisBeamPattern(pos,angspan,w);

    % Compute the sidelobe level for each candidate
    sll = helperArraySynthesisRelativeSidelobeLevel(azpat,elpat);

    % Sort the resulting sidelobe level
    [~,idx] = sort(sll,2,'descend');

    % Discard half of the generation that gets the lower score
    w = w(:,:,[idx(1:numGene/2) idx(1:numGene/2)]);

    % Mutate rows and columns in the newly generated candidates
    mutIdx_row = randi(Nside/2,1,1);
    mutIdx_col = randi(Nside/2,1,1);
    w(mutIdx_row:end,:,numGene/2+1:numGene) = flipud(...
        w(mutIdx_row:end,:,numGene/2+1:numGene));
    w(mutIdx_col:end,:,numGene/2+1:numGene) = fliplr(...
        w(mutIdx_col:end,:,numGene/2+1:numGene));
end
wopt = w(:,:,1);

rng(prvS);

wo = [fliplr(wopt) wopt;rot90(wopt,2) flipud(wopt)];
wplot = helperThinnedArrayComparison(geneticArray,fc,c,[wplot wo(:)],...
    {'Full','Initial','Synthesized'});
```
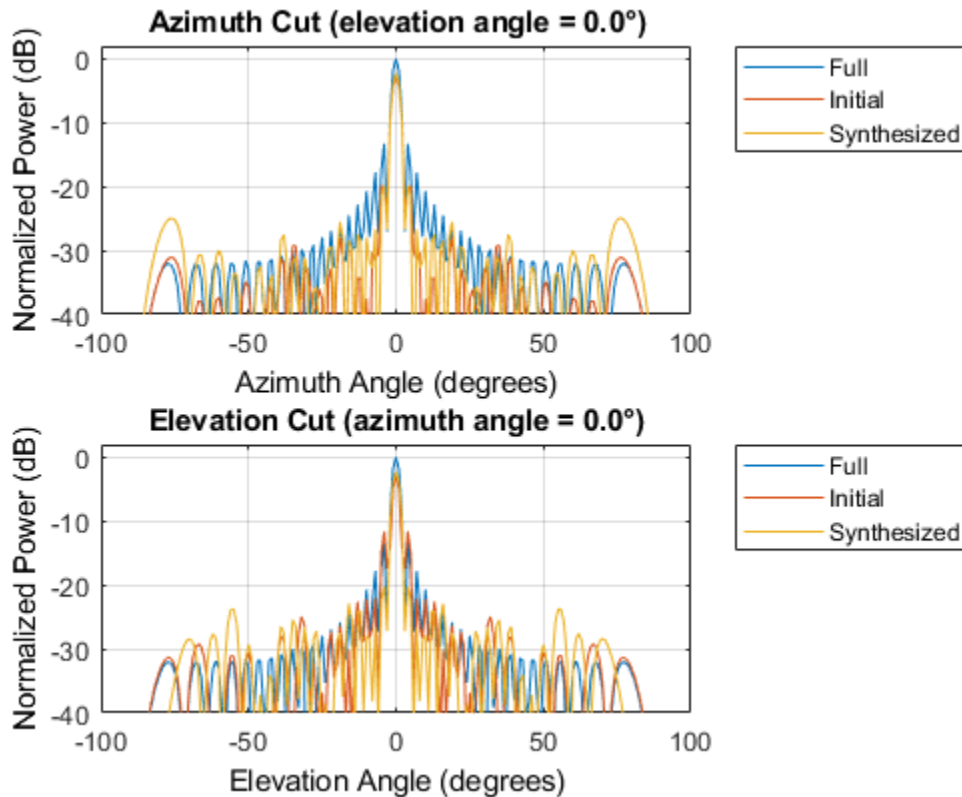
```
sllopt = sll(idx(1))

fillrate = sum(wo(:))/Nside^2*100


sllopt =

   17.5380


fillrate =

   76.5000
```

The figure shows the resulting beam pattern. It can be seen that the sidelobe level has been further improved to about 17.5 dB with a fill rate of 76.5% (1224 active elements). Compared to the first generation candidate, it uses 5% more active elements while achieving an additional 9 dB sidelobe suppression. Compared to the full array, the resulting thinned array can save the cost of implementing T/R switches behind dummy elements, which in turn leads to a roughly 25% saving on the consumed power. Also note that even though the thinned array uses less elements, the beamwidth is close to what could be achieved with a full array.

The final thinned array is shown below with black circles represents the dummy elements.

```
clf;
geneticArray.Taper = wo;
viewArray(geneticArray,'ShowTaper',true);
```

Array Geometry
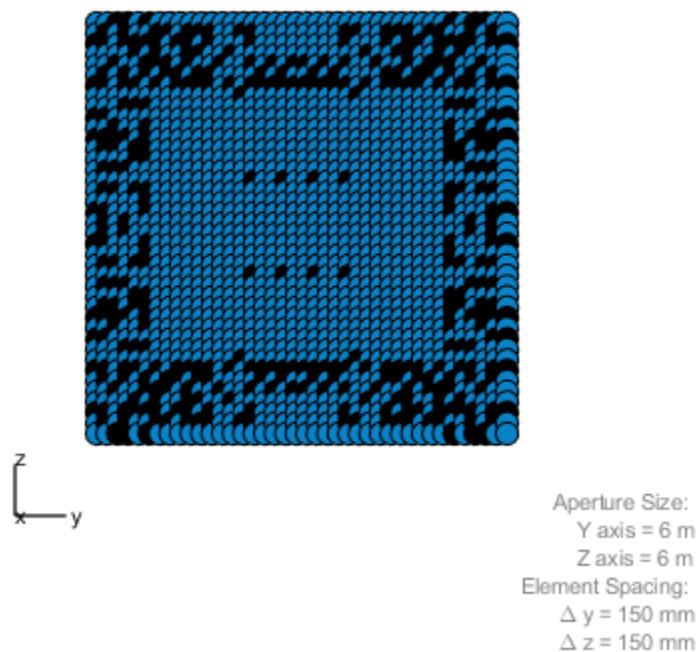


It is worth noting that the genetic algorithm does not always land on the same solution in each trial. However, in general the resulting beam patterns share a similar sidelobe level.

The script above shows a very simple genetic algorithm applied to the array synthesis problem. In real applications, the genetic algorithm is likely to be more complex. There are also other optimization algorithms used in array synthesis, such as the simulated annealing algorithm. Interested readers can find both genetic algorithm and simulated annealing algorithm solvers in the Global Optimization Toolbox.

**Summary**

This example shows several approaches to perform array synthesis on a phased array. In practice, one needs to choose the appropriate synthesis method according to the specific constraint of the application, such as the size of the array aperture, the shape of the array geometry, etc.

**Reference**

[1] Randy L. Haupt, Thinned Arrays Using Genetic Algorithms, IEEE Transactions on Antennas and Propagation, Vol 42, No 7, 1994

[2] Randy L. Haupt, An Introduction to Genetic Algorithms for Electromagnetics, IEEE antennas and Propagation Magazine, Vol 37, No 2, 1995

[3] Harry L. Van Trees, Optimum Array Processing, Wiley-Interscience, 2002

# Array Pattern Synthesis Part II: Optimization

This example shows how to use optimization techniques to perform pattern synthesis. Using minimum variance beamforming as an illustration, the example covers how to set up the pattern synthesis as an optimization problem and then solve it using optimization solvers.

Pattern synthesis can be achieved using a variety of techniques. The "Array Pattern Synthesis Part I: Nulling, Windowing, and Thinning" on page 18-10 example introduced nulling and windowing techniques.

This example requires Optimization Toolbox™.

**Minimum Variance Beamforming**

Compared to a single element, a major benefit of a phased array is the capability of forming beams. In general, an N-element phased array provides N degrees of freedom when synthesizing a pattern, meaning that one can adjust N weights, one for each element, to control the shape of the beam to satisfy some pre-defined constraints.

Many popular beamforming techniques can be expressed as optimization problems. For example, the popular minimum variance distortionless response (MVDR) beamformer is used to minimize the total noise output while preserving the signal from a given direction. Mathematically, the MVDR beamforming weights can be found by solving the optimization problem

$$w_o = \min_w w^H R \, w$$
$$\text{s.t.} \, B(\theta_0, w) = 1$$

where $R$ is the covariance matrix, $B(\theta_0, w)$ is the beam response towards the direction $\theta_0$ with the weights vector $w$, and $w_o$ is the vector containing the optimal weights. Because the beam pattern can be written as

$$B(\theta, w) = w^H v(\theta)$$

where $v(\theta)$ is the steering vector corresponding to the angle $\theta$, the constraints have a linear relation in the complex domain.

The distortionless constraint at $\theta_0$ only takes 1 degree of freedom in the design space. Since an N-element array can handle $N - 1$ constraints, the MVDR beamformer can be extended to include more constraints, which results in a linear constraints minimum variance (LCMV) beamformer. The extra constraints in an LCMV beamformer are often used to null out interferences from given directions. Therefore, such an LCMV beamformer is a solution to the following optimization problem

$$w_o = \min_w w^H R \, w$$
$$\text{s.t.} \, B(\theta_0, w) = 1$$
$$B(\theta_k, w) = 0 \quad k = 1, 2, ..., K$$

where $\theta_k$ represents the $k$th nulling direction and there are $K$ nulling directions in total. Because all constraints are linear in the complex domain, the LCMV beamforing weights can be derived analytically using the Lagrange multiplier method.

Next, a 32-element ULA with half wavelength spacing is used to demonstrate how to perform array synthesis using the LCMV approach. Note that even though this example uses a linear array, the technique applies to any array geometry. Assume that the signal of interest is at 0 degrees azimuth and the interferences are at –70, --40, and –20 degrees azimuth. The noise power is assumed to be at 40 dB below signal. The closed form solution for LCMV weights can be computed using the `lcmvweights` function.

```
N = 32;
pos = (0:N-1)*0.5;              % position of elements
ang_i = [-70 -40 -20];         % interference angles
ang_d = 0;                     % desired angle

Rn = sensorcov(pos,ang_i,db2pow(-40));   % Noise covariance matrix
sv_c = steervec(pos,[ang_d ang_i]);      % Linear constraints
r_c = [1 zeros(size(ang_i))]';           % Desired response
w_lcmv = lcmvweights(sv_c,r_c,Rn);       % LCMV weights

ang_plot = -90:0.1:90;
sv_plot = steervec(pos,ang_plot);
plcmv = plot(ang_plot,mag2db(abs(w_lcmv'*sv_plot)));
xline(ang_i,"--")
ylim([-100 0])
grid on
legend(plcmv,"LCMV - Analytic")
xlabel("Azimuth Angle (deg)")
ylabel("Beam Pattern (dB)")
```
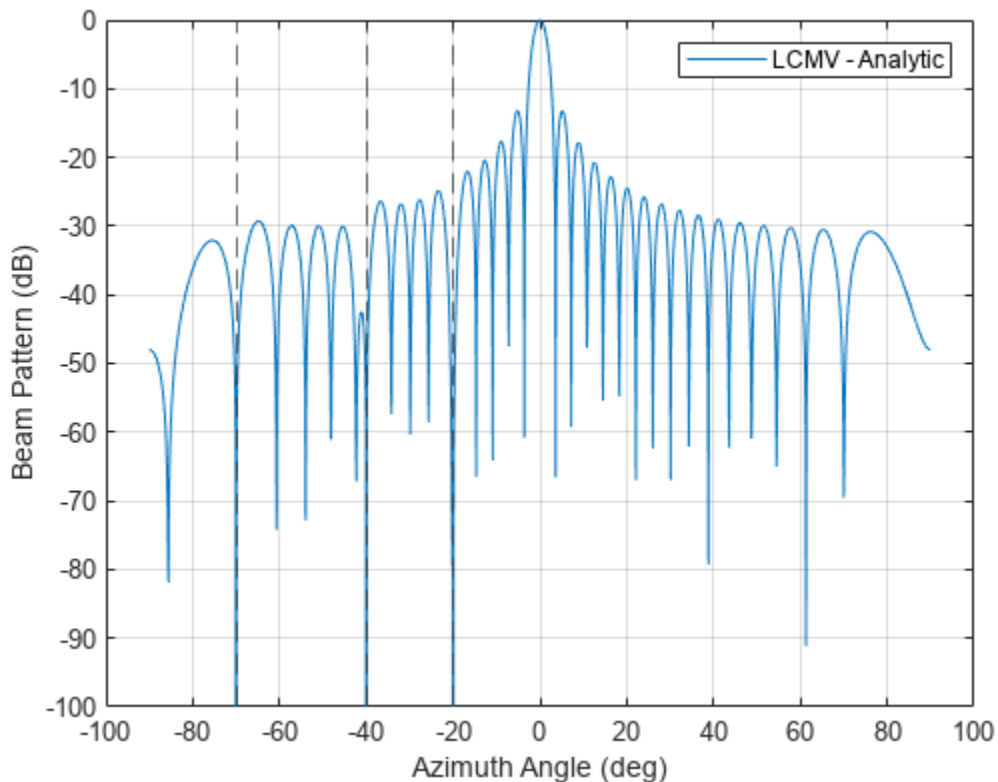
The figure shows that the computed weights satisfy all constraints.

**LCMV Beamforming Using Quadratic Programming**

Since LCMV beamforming can be described as an optimization problem, it would be useful to see how the problem can be solved using optimization techniques. When a closed form solution is not available, for example when constraints include inequalities, such workflow becomes important.

When using optimization techniques to perform pattern synthesis tasks, the first step is to pick a solver. Since solvers are often developed to address a particular problem formulation, it is of utmost importance to make the right choice.

The quadratic programming (QP) solver is often used to solve a quadratic objective function with linear constraints. The formulation is given by

$$w_o = \min_w \frac{1}{2} w^T H w + f^T w$$
$$s.t.\ A \cdot w \le b$$
$$A_{eq} \cdot w = b_{eq}$$
$$w_{lb} \le w \le w_{ub}$$

where $A$ and $A_{eq}$ are matrices specifying the inequality and equality constraints, respectively; $b$ and $b_{eq}$ are vectors specifying the desired bounds for inequality and equality constraints, respectively; and $w_{lb}$ and $w$ indicate lower and upper bounds for elements in $w$, respectively.

At first sight, the problem formulation of quadratic programming seems to be a perfect match for LCM beamforming because we know that the beam pattern can be written as $B(\theta, w) = w^H v(\theta)$. Unfortunately, this is not true because although the beam pattern is linear in complex domain, it is not so in real domain. On the other hand, most optimization solvers, including quadratic programming, work in real domain as the solution space becomes really complicated in the complex domain. In addition, the inequality is not well defined in the complex domain either. Thus, the question is if the LCMV beamforming problem can be formulated into real domain and still obtain the quadratic programming form.

Start with the objective function $w^H R w$. Note that $R$ is a covariance matrix in this context, thus $w^H R w$ is real. Therefore, the goal is really to minimize the real part of $w^H R w$, which can be written in the following form:

$$\Re\left\{w^H R w\right\} = \left[\ \Re\ \{w\}^T\ \ \Im\ \{w\}^T\right] \begin{bmatrix} \Re\ \{R\} & -\ \Im\ \{R\} \\ \Im\ \{R\} & \Re\ \{R\} \end{bmatrix} \begin{bmatrix} \Re\ \{w\} \\ \Im\ \{w\} \end{bmatrix} = \bar{w}^T \bar{R}\ \bar{w}$$

where $\bar{w} = \begin{bmatrix} \Re\ \{w\} \\ \Im\ \{w\} \end{bmatrix}$ and $\bar{R} = \begin{bmatrix} \Re\ \{R\} & -\ \Im\ \{R\} \\ \Im\ \{R\} & \Re\ \{R\} \end{bmatrix}$.

Next, look at the constraints. The fact that the constraints in LCMV beamforming are equalities makes the conversion a bit easier since one complex linear equality constraint can just be divided into two corresponding real linear equality constraints, one for the real part and the other for the imaginary part. For example, the following constraint

$$w^H v = r^H$$

can be rewritten as

$$v^H w = r$$

$$\begin{bmatrix} \Re\{v\} & \Im\{v\} \\ \Im\{v\} & -\Re\{v\} \end{bmatrix} \begin{bmatrix} \Re\{w\} \\ \Im\{w\} \end{bmatrix} = \begin{bmatrix} \Re\{v\} & \Im\{v\} \\ \Im\{v\} & -\Re\{v\} \end{bmatrix} \bar{w} = \begin{bmatrix} \Re\{r\} \\ \Im\{r\} \end{bmatrix}$$

With these relations, quadratic programming can then be used to obtain LCMV beamforming weights.

```
% Objective function
H = [real(Rn) -imag(Rn);imag(Rn) real(Rn)];
f = [];

% Equality constraints
Aeq = [real(sv_c).' imag(sv_c).';imag(sv_c).' -real(sv_c).'];
beq = [real(r_c);imag(r_c)];

% Compute weights
x_opt = quadprog(H,f,[],[],Aeq,beq);

Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

w_opt = x_opt(1:numel(x_opt)/2)+x_opt(numel(x_opt)/2+1:end)*1i;

hold on
plcmvq = plot(ang_plot,mag2db(abs(w_opt'*sv_plot)),"--");
hold off
legend([plcmv plcmvq],["LCMV - Lagrange" "LCMV - QP"])
```
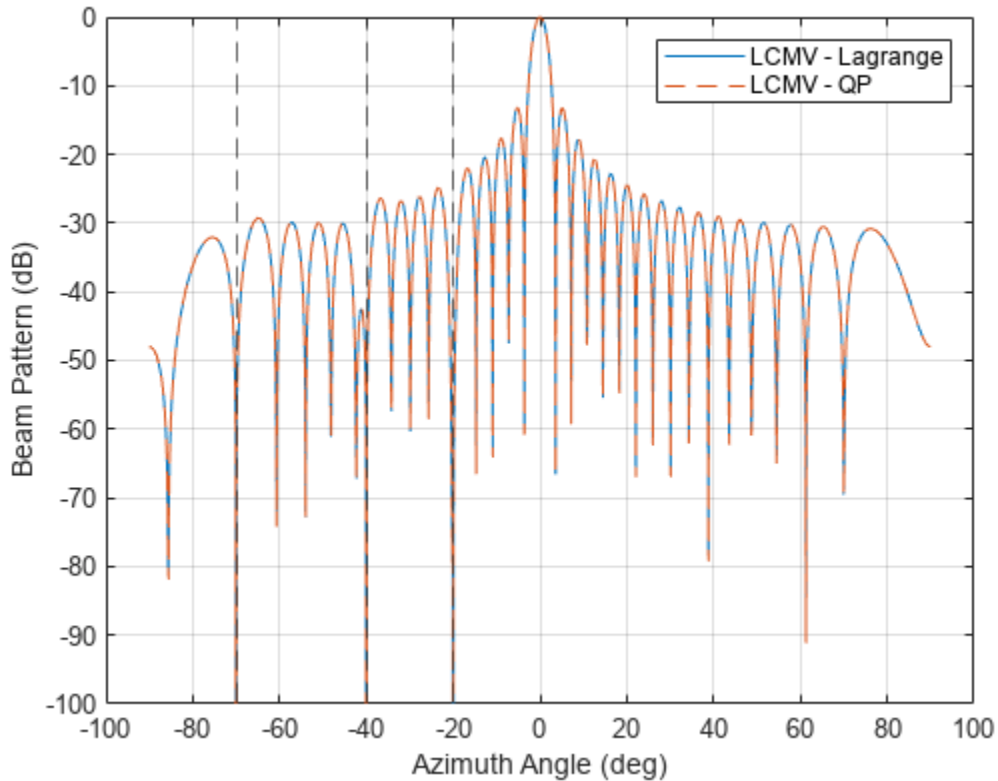
The figure shows that the QP technique can obtain the same LCMV beamforming weights.

**Minimum Variance Beamforming Using Quadratic Programming**

Another common requirement for pattern synthesis is to ensure the response is below a certain threshold for a set of given angles. These are inequality requirements and the overall optimization problem becomes

$$
\begin{aligned}
w_o = \ & \min_w w^H R\, w \\
& s.t.\, B(\theta_0, w) = 1 \\
& \quad B(\theta_k, w) = 0 \quad k = 1, 2, ..., K \\
& \quad |B(w, \theta_l)| \le r_l \quad l = 1, 2, ..., L
\end{aligned}
$$

where $\theta_l$ is the $l$th angle at which the sidelobe level should be less than the threshold $r_l$. There are $L$ inequality requirements in total.

As an example, add the following requirements to our sample problem:

**1**   The sidelobe should be below –40 dB between –30 and –10 degrees in azimuth.

**2**   Everywhere outside of the mainlobe, the sidelobe should be below −20 dB.

This code plots the mask of the desired pattern and returns the inequality constraints:

```
[rc,angc,sv_cineq,r_ineq,ang_ineq]=generatePatternMask(pos,r_c,[ang_d ang_i]);
```

```
pdp = plot(angc,rc,"k--");
ylim([-100 0])
grid on
xlabel("Azimuth Angle (deg)")
ylabel("Beam Pattern (dB)")
```
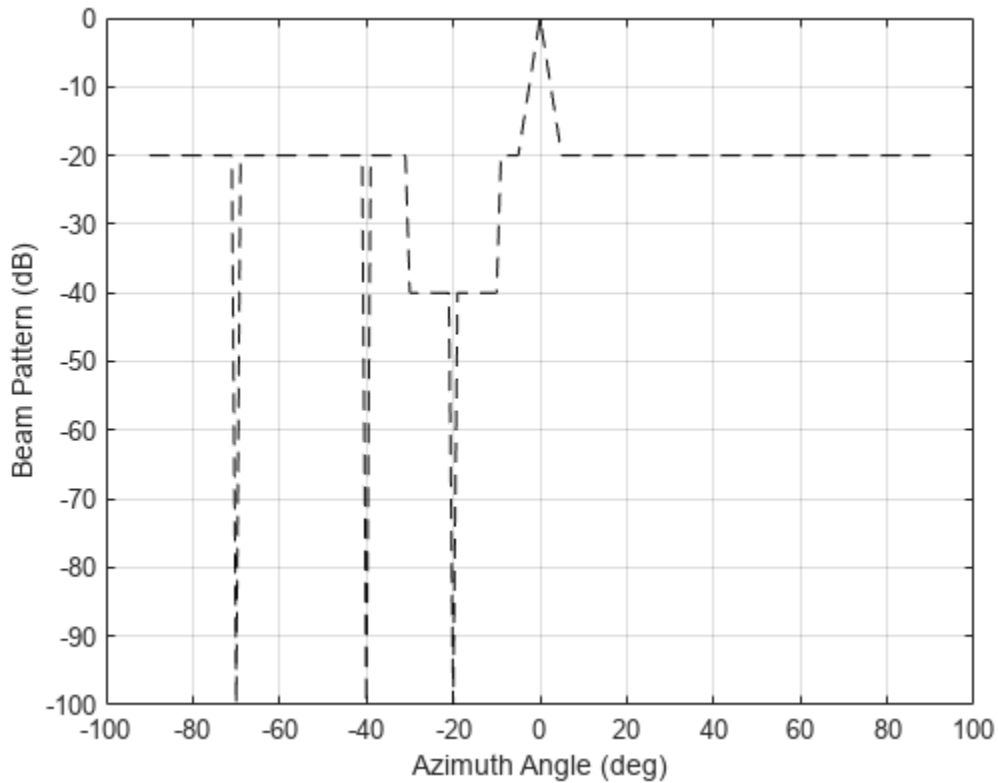


With the introduction of the new inequality constraint, the problem no longer has a closed-form solution. In addition, due to the presence of the norm operation, the aforementioned trick to translate complex constraints to real domain no longer works. Therefore, a new way to translate the complex inequality is needed to use the QP solver for this set of constraints,

Given a constraint of $|B(\theta, w)| < r$, one possible way to convert this is to assume that both real and imaginary parts contribute equally to the norm. In that case, the single complex inequality constraint can be divided into two real inequalities as

$$-\frac{r}{\sqrt{2}} \le \Re\{B(\theta, w)\} \le \frac{r}{\sqrt{2}}$$

$$-\frac{r}{\sqrt{2}} \le \Im\{B(\theta, w)\} \le \frac{r}{\sqrt{2}}$$

or

$$\Re\,\{B(\theta,w)\} \le \frac{r}{\sqrt{2}}$$

$$\Im\,\{B(\theta,w)\} \le \frac{r}{\sqrt{2}}$$

$$-\,\Re\,\{B(\theta,w)\} \le \frac{r}{\sqrt{2}}$$

$$-\,\Im\,\{B(\theta,w)\} \le \frac{r}{\sqrt{2}}$$

The real and imaginary part of the constraints can be extracted as described in the previous section. Hence, the following code sets up the constraints and solves the pattern synthesis task.

```
% Map complex inequality constraints to real constraints
sv_c_real = [real(sv_cineq).' imag(sv_cineq).'];
sv_c_imag = [imag(sv_cineq).' -real(sv_cineq).'];
A_ineq = [sv_c_real;sv_c_imag];
A_ineq = [A_ineq;-A_ineq];
b_ineq = [r_ineq;r_ineq]/sqrt(2);
b_ineq = [b_ineq;b_ineq];

% Compute weights
x_opt = quadprog(H,f,A_ineq,b_ineq,Aeq,beq);

Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

% Reassemble complex weights from real and imaginary parts
w_opt = x_opt(1:numel(x_opt)/2)+x_opt(numel(x_opt)/2+1:end)*1i;

hold on
pqp = plot(ang_plot,mag2db(abs(w_opt'*sv_plot)));
hold off
legend(pqp,"Pattern - QP")
```
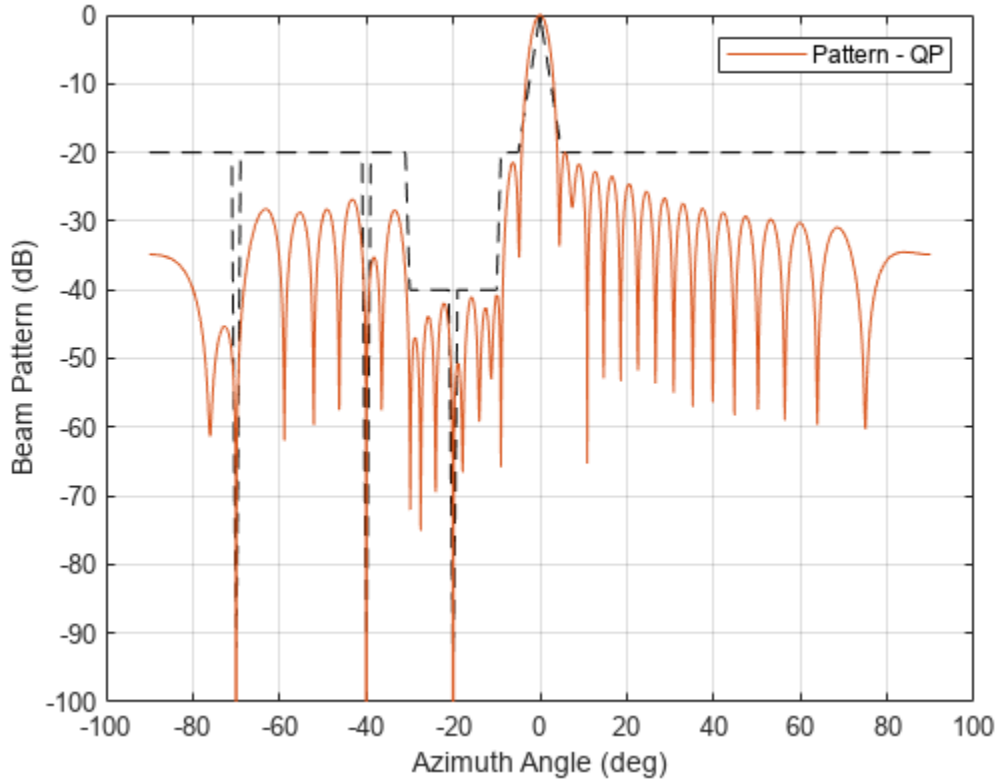
The resulting pattern satisfies the requirements.

**Minimum Variance Beamforming Using Second Order Cone Programming**

Even though the QP solver solves the minimum variance beamforming problem successfully, the conversion of the inequality constraints from complex to real is not ideal. For example, there is no reason that real and imaginary parts should always equally contribute to the norm. However, there is really no easy way to handle a constraint with norm in a QP problem setup.

Luckily, there is another optimization solver that can be used in this case, called Second order cone programming (SOCP). SOCP is developed to efficiently solve the following optimization problem:

$$w_o = \ \min_w f^T w$$
$$s.t. \, \| A_{\text{sc}} \cdot w - b_{\text{sc}} \| \le d^T w - \gamma$$
$$A \cdot w \le b$$
$$A_{\text{eq}} \cdot w = b_{\text{eq}}$$
$$w_{\text{lb}} \ \le w \le w_{\text{ub}}$$

where $A_{\text{sc}}$ is a matrix, $b_{\text{sc}}$ and $d$ are vectors, and $\gamma$ is a scalar. $A_{\text{sc}}$, $b_{\text{sc}}$, $d$, and $\gamma$ together define a second order cone constraint, which is a natural choice to specify a constraint on norms. The solver can handle multiple second order cone constraints simultaneously.

Comparing to the problem formulation for QP, SOCP can not only handle second order constraints, but also handle all constraints specified in QP. However, the objective function is quite different.

Instead of minimizing a variance, SOCP minimize a linear function of the optimization variable, $w$. Fortunately, there is a way to convert the minimum variance objective function to fit into this SOCP framework. The trick is to define an auxiliary variable, $t$, and ensure that the variance is bounded by $t$. This way, the problem becomes to minimize $t$ while the original quadratic objective becomes a cone constraint bounded by $t$.

Following this thought, amend $t$ to the bottom of $\bar{w}$. Now the optimization search space is defined by $\widetilde{w} = \begin{bmatrix} \bar{w} \\ t \end{bmatrix}$. Since the goal is to minimize $t$, $f$ is defined as $f^T = [0, ..., 0\ 1]$.

```
f = [zeros(2*N,1);1];
```

Next, explore how to convert the quadratic constraint to a second order cone constraint. Since the idea is to constraint the variance as

$$\bar{w}^T \bar{R} \bar{w} \leq t$$

it is equivalent to say

$$\| \bar{R}^{0.5} \bar{w} \|^2 \leq t + \frac{1}{4}$$

Thus,

$$\left\| \begin{bmatrix} \bar{R}^{0.5} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{w} \\ t \end{bmatrix} \right\|^2 \approx \| \bar{R}^{0.5} \bar{w} \|^2 + t^2 \leq t^2 + t + \frac{1}{4} = \left( t + \frac{1}{2} \right)^2$$

and the second cone constraint becomes

$$\left\| \begin{bmatrix} \bar{R}^{0.5} & 0 \\ 0 & 1 \end{bmatrix} \widetilde{w} \right\| \leq t + \frac{1}{2}$$

```
% The cone constraints are defined as ||A_c*u-b_c||<=d_c^T*u-gamma_c
Ac = [sqrtm(H) zeros(2*N,1);zeros(1,2*N) 1];
bc = zeros(2*N+1,1);
dc = [zeros(2*N,1);1];
gammac = -1/2;

M = numel(r_ineq);
socConstraints(M+1) = secondordercone(Ac,bc,dc,gammac);
```

The remaining second cone constraints are from the original inequality constraints and are fairly straightforward to be converted. Each inequality constraint in the form of

$$\| B(\theta, w) \| \leq r$$

can be written as a second order cone constraint as

$$\left\| \begin{bmatrix} \Re\ \{B(\theta, w)\} & 0 \\ \Im\ \{B(\theta, w)\} & 0 \\ 0 & 0 \end{bmatrix} \widetilde{w} \right\| \leq r$$

```
for m = 1:M
    Ac = [A_ineq([m m+M],:) zeros(2,1);zeros(1,2*N) 0];
```

```
    bc = zeros(3,1);
    dc = zeros(2*N+1,1);
    gammac = -r_ineq(m);
    socConstraints(m) = secondordercone(Ac,bc,dc,gammac);
end
```

The equality constraints also need to be augmented to accommodate the auxiliary variable, $t$.

```
% patch Aeq and beq
Aeqc = [Aeq zeros(size(Aeq,1),1);zeros(1,2*N) 0];
beqc = [beq;0];
```

The array weights can now be computed using the SOCP solver. Since the inequality constraints are more natural to be represented in cone constraints, naively one would expect that in this example, the resulting pattern from SOCP would be better compared to the pattern from QP.

```
[u_op,feval,exitflag] = coneprog(f,socConstraints,[],[],Aeqc,beqc);
```

```
Optimal solution found.
```

```
% Reassemble complex weights from real and imaginary parts
% The last entry in the optimized vector corresponds to the variable t
w_op = u_op(1:N)+u_op(N+1:2*N)*1i;
```

```
hold on
psocp = plot(ang_plot,mag2db(abs(w_op'*sv_plot)));
hold off
legend([pqp psocp],["Pattern - QP" "Pattern - SOCP"])
```
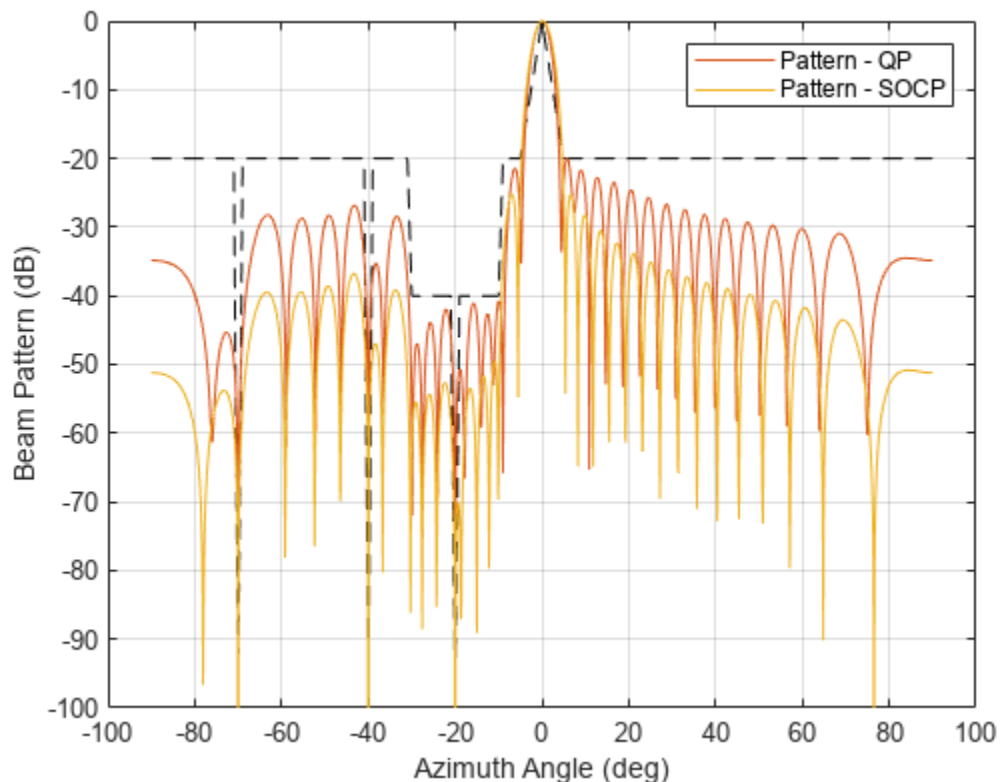
Indeed, the plot shows that the pattern obtained via SOCP has better sidelobe suppression compared to the pattern obtained using QP. As you can see, the process is quite involved when an optimization solver is used. We can simplify this process by calling the minvarweights function. The code below shows how to define the mask constraints as inputs for *minvarweights* function. The mask sidelobe level and mask angle represent the inequality constraints for the optimization problem, and the null angle represents the equality constraint for SOCP. For more details, refer to the `minvarweights` documentation.

```
W_opt = minvarweights(pos,ang_d,Rn, ...
    MaskSidelobeLevel=mag2db(r_ineq)', ...
    MaskAngle=ang_ineq,NullAngle=ang_i);

plot(ang_plot,mag2db(abs(W_opt'*sv_plot)))
hold on
plot(angc,rc,'k--')
hold off
legend(["Pattern - minvarweights" "Sidelobe Mask"], ...
    Location="best")
ylim([-100 0])
grid on
xlabel("Azimuth Angle (deg)")
ylabel("Beam Pattern (dB)")
```
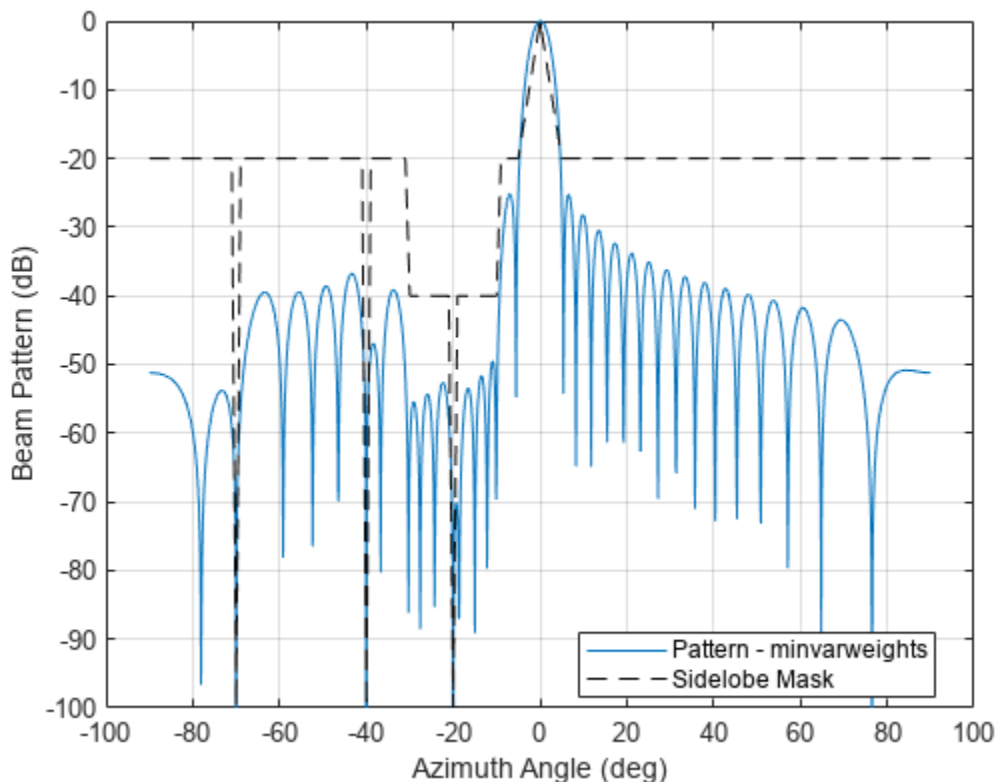


**Minimum Variance Beamforming Using `minvarweights` with Other Mask Constraints**

In this section, we use minvarweights function to solve pattern synthesis problems with different optimization constraints. Consider a linear array with 55 antenna elements and 0.5$\lambda$ spacing ULA in

the direction of 25-degree in azimuth. The goal is to optimize the beamforming weights with the following constraints:

- Sidelobe levels less than a tapered sidelobe mask decrease linearly from −18dB to −55dB in the range of 1st SLL to ±90 degrees in azimuth.
- Steer nulls at [−45 −35 40 60] degrees in azimuth.

```matlab
N = 55;                      % Number of elements
pos = (0:N-1)*0.5;           % Position of elements
ANGmainBeam = 25;            % Main Beam direction

% Define null directions
ANGn = [-35 -45 40 60];      % Null angles, degree

% Define the SLL mask
ANGml = -90:.2:22.5;                       % mask angles for left SLLs
SLLml = linspace(-55,-18,numel(ANGml));    % mask SLLs for ANGml ( decrease linearly from −18dB
ANGmr = 27.5:0.2:90;                       % mask angles for right SLLs
SLLmr = linspace(-18,-55,numel(ANGmr));    % mask SLLs for ANGmr ( decrease linearly from −18dB
ANGm = [ANGml ANGmr];                      % mask angles
SLLm = [SLLml SLLmr];                      % mask SLL

% Compute optimized weights
OptimizedWeights = minvarweights(pos, ANGmainBeam, ...
    MaskAngle=ANGm,MaskSidelobeLevel=SLLm,NullAngle=ANGn);

% Calculate array pattern
sv_plot = steervec(pos,-90:.25:90);
Pattern = OptimizedWeights'*sv_plot;

% Plot the pattern
plot(-90:.25:90,mag2db(abs(Pattern)))
xlabel("Azimuth Angle (deg)")
ylabel("Beam Pattern (dB)")
axis([-90 90 -80 0])
grid on

% Plot the mask with nulls
hold on
[AngMask,IA] = sort([ANGm ANGn]);
SLLmask = [SLLm -1000*ones(1,size(ANGn,2))];
plot(AngMask,SLLmask(IA),'k')
legend(["Pattern - minvarweights" "Sidelobe Mask"], ...
    Location="best")
hold off
```
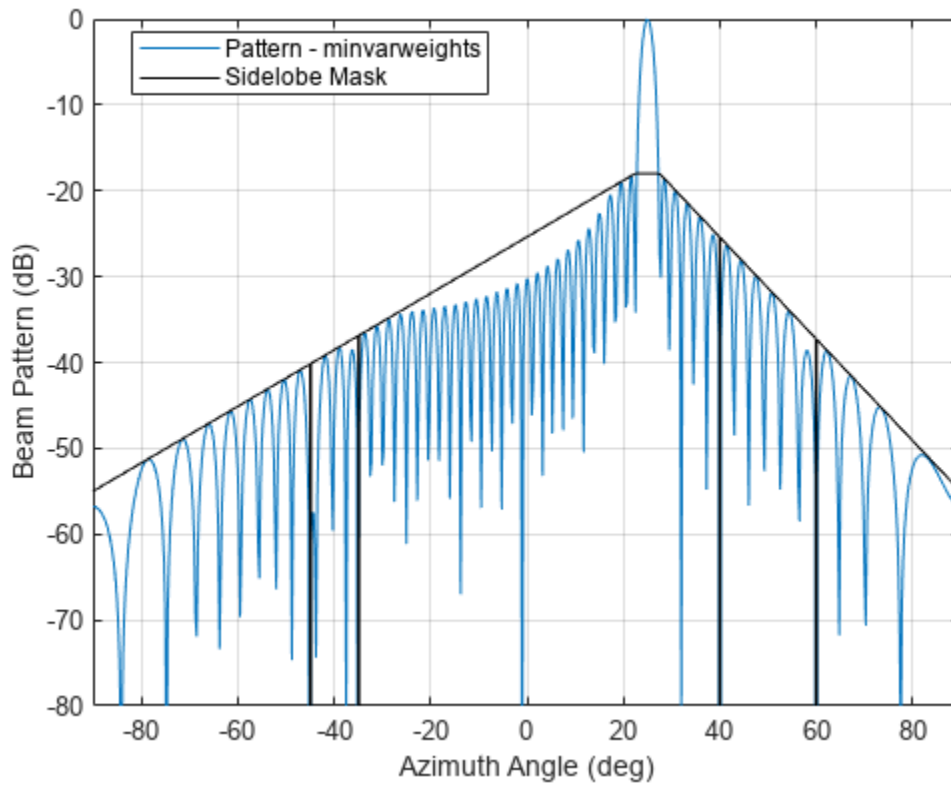
This animation shows the pattern synthesis results for the beam sweeping from −35 degree to +35 degree and SLLs decrease linearly from −18 dB to −55 dB in the range of 1st SLL to ±90 degrees in azimuth.
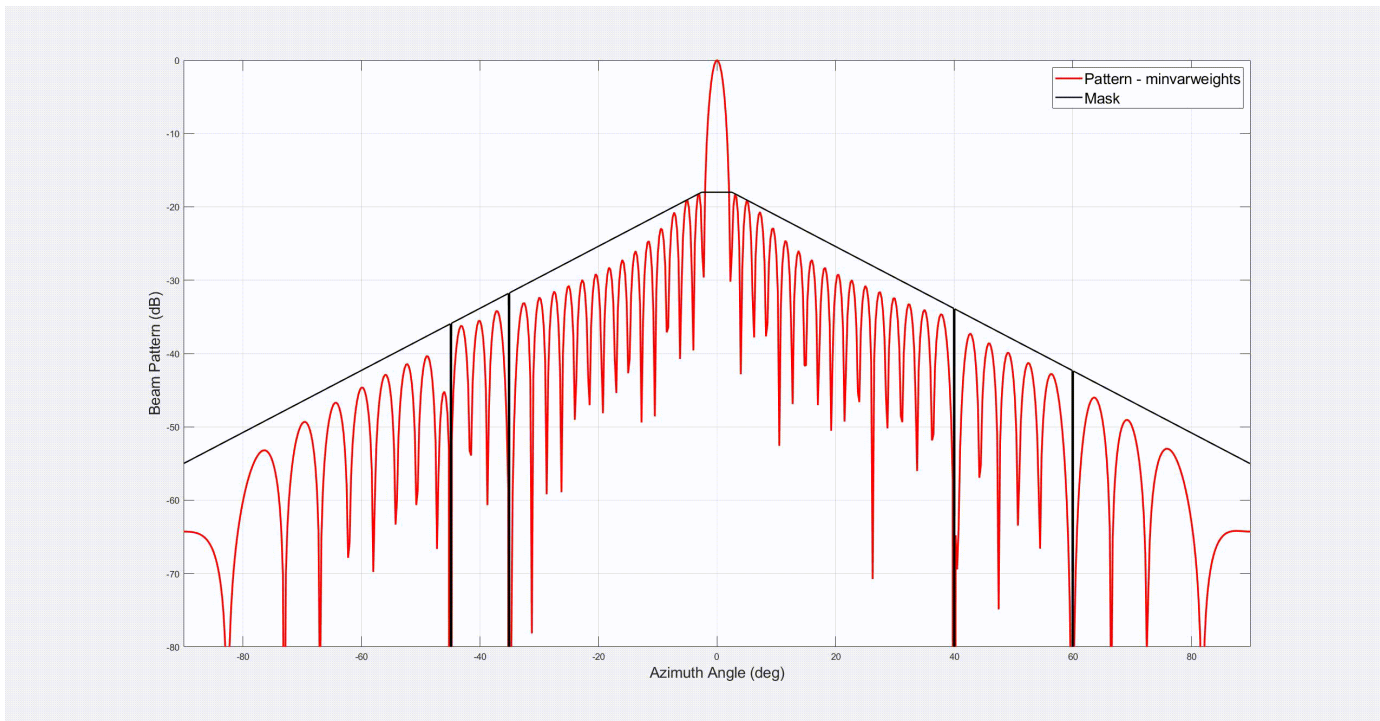
## Summary

This example shows how to use optimization techniques to perform pattern synthesis. The example started with the familiar LCMV problem and extended the desired pattern to include inequality constraints. Compared to the closed-form solution, optimization techniques are more flexible and can be applied to address more complex constraints. The example also showed how to address inequality constraints using either quadratic programming, second order cone programming, or minvarweights.

## Reference

[1] Lebret, Hervé, and Stephen Boyd. "Antenna Array Pattern Synthesis via Convex Optimization." *IEEE Transactions on Signal Processing,* Vol. 45, No. 3 (March 1997), pp. 526–532.

```
function [rcplot,angc,svcineq,rineq,angineq]=generatePatternMask(pos,rcexist,angexist)
% generatePatternMask  Generate mask for pattern synthesis

ang_c1 = [-30:-21 -19:-10];
ang_c2 = [-90:-71 -69:-41 -39:-31 -9:-5 5:90];
% Note the main beam is between -10 and 10 degrees
sv_c1 = steervec(pos,ang_c1);
sv_c2 = steervec(pos,ang_c2);

r1 = db2mag(-40)*ones(size(sv_c1,2),1);
r2 = db2mag(-20)*ones(size(sv_c2,2),1);

[angc,cidx] = sort([ang_c1 ang_c2 angexist]);
rc = [r1;r2;rcexist];
rcplot = mag2db(rc(cidx));
rcplot(rcplot<-100) = -100;

% Inequality constraints
```

```
sv_c1 = steervec(pos,ang_c1);
sv_c2 = steervec(pos,ang_c2);
svcineq = [sv_c1 sv_c2];
rineq = [r1;r2];
angineq = [ang_c1 ang_c2];

end
```

# Modeling and Analyzing Polarization

This example introduces the basic concept of polarization. It shows how to analyze the polarized field and model the signal transmission between polarized antennas and targets using Phased Array System Toolbox™.

**Polarization of an Electromagnetic Field**

The electromagnetic field generated by an antenna is orthogonal to the propagation direction in the far field. The field can be pointing to any direction in this plane, and therefore can be decomposed into two orthogonal components. Theoretically, there are an infinite number of ways to define these two components, but most often, one uses either (H,V) set or (L,R) set. (H,V) stands for horizontal and vertical, which can be easily pictured as x and y component; while (L,R) stands for left and right circular. It may be difficult to imagine that a vector in space can have a circular component in it, the secret lies in the fact that each component can be a complex number, which greatly increases the complexity of the trace of such a vector.

Let's look at several simple examples. The time varying field can be written as

$$E = \mathbf{u_h}|\mathbf{E_h}|\cos(\omega\mathbf{t} - \mathbf{kz} + \phi_\mathbf{h}) + \mathbf{u_v}|\mathbf{E_v}|\cos(\omega\mathbf{t} - \mathbf{kz} + \phi_\mathbf{v})$$

where

$$E_h = |E_h|e^{j\phi_h}, E_v = |E_v|e^{j\phi_v}$$

are the two components in phasor representation. $\mathbf{u_h}$ and $\mathbf{u_v}$ are the unit vector of h and v axes, respectively.

The simplest case is probably a linear polarization, which happens when the two components are always in phase. Assume

$$|E_h| = |E_v| = 1, \phi_h = \phi_v = 0,$$

the field can be represented by a vector of [1;1]. The polarization for such a field looks like

```
fv = [1;1];
helperPolarizationView(fv)
```

**H, V and Combined Fields**



**Polarization Ellipse**



**Poincare Sphere**
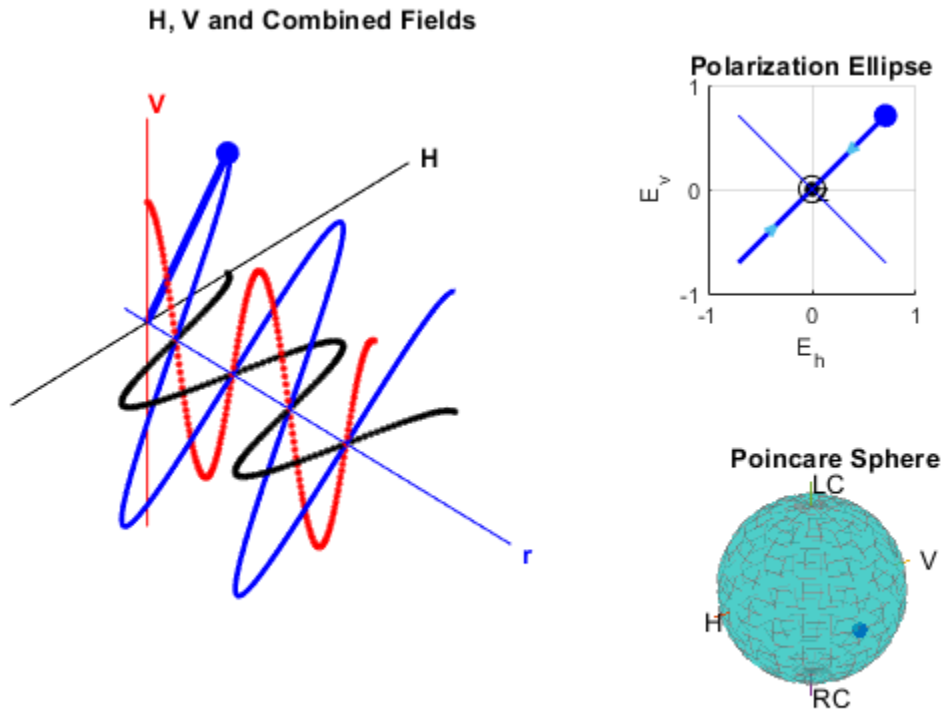


From the figure, it is clear that the combined polarization is along the 45 degrees diagonal.

The plot in the upper right portion of the figure is often referred to as the polarization ellipse. It is the projection of the combined field trace on the H-V plane. The polarization ellipse is often characterized by two angles, the tilt angle (also known as orientation angle) $\tau$ and the ellipticity angle $\epsilon$. In this case, the tilt angle is 45 degrees and the ellipticity angle is 0. The dot on the ellipse shows how the combined field moves along the trace on the H-V plane while time passes.

A polarized field can also be represented by Stokes vector, which is a length-4 vector. The corresponding Stokes vector of the linear polarization, [1;1], is given by

```
s = stokes(fv)
```

```
s =

    2
    0
    2
    0
```

Note that all 4 entries in the vector are real numbers. In fact, all these entries are measurable. In addition, it can be shown that the four quantities always satisfy the following equation

$$s(1)^2 = s(2)^2 + s(3)^2 + s(4)^2.$$

Therefore, each set of Stokes can be considered as a point on a sphere. Such a sphere is referred to as a Poincare sphere. The Poincare sphere for the above field is shown in the bottom right portion of the figure.

Next is a circular polarized field, where

$$|E_h| = |E_v| = 1, \phi_h = 0, \phi_v = \pi/2.$$

```
fv = [1;1i];
helperPolarizationView(fv)
```

**H, V and Combined Fields**

**Polarization Ellipse**
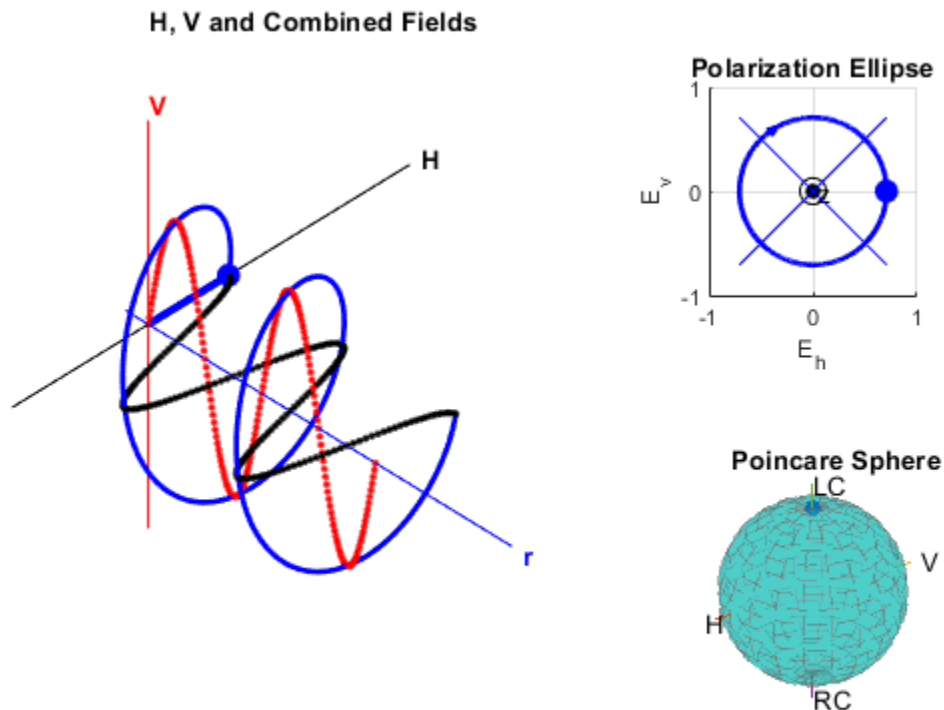
**Poincare Sphere**

The figure shows that the trace of the combined field is a circle. Both the polarization ellipse and the Poincare sphere shows that the field is left circularly polarized.

In general, the trace of a field is an ellipse, as shown below

```
fv = [2+1i;1-1i];
helperPolarizationView(fv)
```

**H, V and Combined Fields**
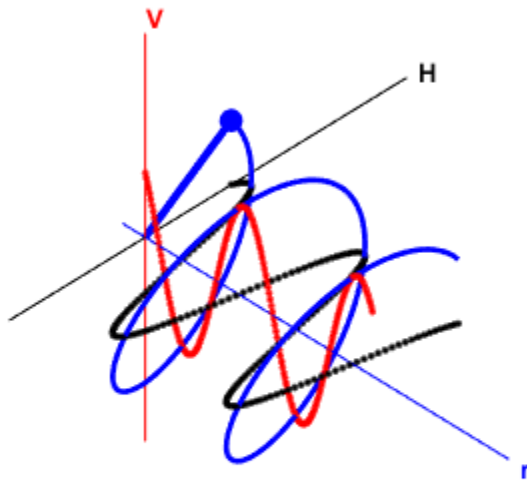


**Polarization Ellipse**
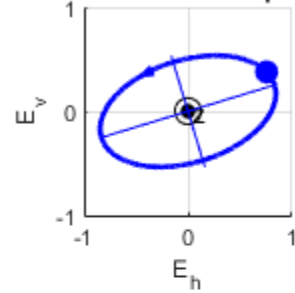


**Poincare Sphere**
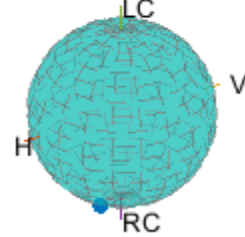


**Polarization of an Antenna**

The polarization of an antenna is defined as the polarization of the field transmitted by the antenna regardless whether it's in the transmitting or receiving mode. However, as mentioned earlier, the polarization is defined in the plane that is orthogonal to the propagation direction. Therefore, it is defined in the local coordinate system of each propagation direction, as shown in the following diagram.
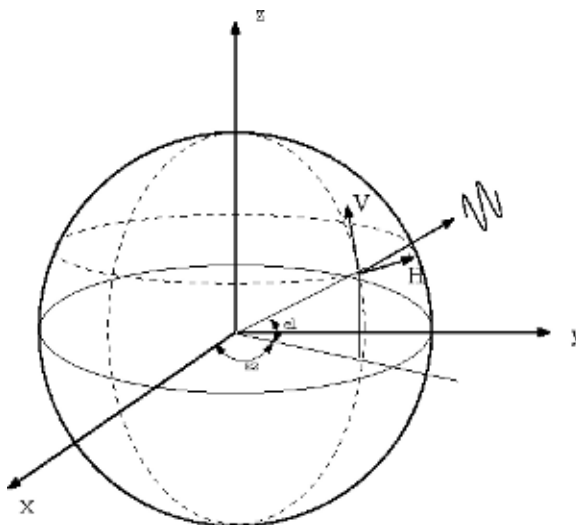
Some antennas have a structure that determines its polarization, such as a dipole. The dipole antenna has a polarization that is parallel to its orientation. Assuming the frequency is 300 MHz, for a vertical short dipole, the polarization response at boresight, i.e., 0 degrees azimuth and 0 degrees elevation, is given by

```
antenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');

fc = 3e8;
resp = antenna(fc,[0;0])
```

```
resp =

  struct with fields:

    H: 0
    V: -1.2247
```

Note that the horizontal component is 0. If we change the orientation of the dipole antenna to horizontal, the vertical component becomes 0.

```
antenna = phased.ShortDipoleAntennaElement('AxisDirection','Y');
resp = antenna(fc,[0;0])
```

```
resp =

  struct with fields:

    H: -1.2247
    V: 0
```

**Polarization Loss**

When two antennas form a transmit/receive pair, their polarizations could affect the received signal power. Therefore, to collect a signal with maximum power possible, the receive antenna's polarization has to match the transmit antenna's polarization. The polarization matching factor can be measured as

$$\rho = |p_t^T p_r|^2$$

where $p_t$ and $p_r$ represent the normalized polarization states of the transmit and receive antenna, respectively.

Assume both transmit and receive antennas are short dipoles. The transmit antenna sits at the origin and the receive antenna at location (100,0,0). First, consider the case where both antennas are along Y axis and face each other. This is the scenario where the two antennas are matched in polarization.

```
pos_r = [100;0;0];
lclaxes_t = azelaxes(0,0);      % transmitter coordinate system
lclaxes_r = azelaxes(180,0);    % receiver faces transmitter
ang_t = [0;0];                  % receiver at transmitter's boresight
ang_r = [0;0];                  % transmitter at receiver's boresight

txAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
```

```
resp_t = txAntenna(fc,ang_t);
rxAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
resp_r = rxAntenna(fc,ang_r);

ploss = polloss([resp_t.H;resp_t.V],[resp_r.H;resp_r.V],pos_r,lclaxes_r)
```

```
ploss =

     0
```

The loss is 0 dB, indicating that there is no loss due to polarization mismatch. The section below shows the effect with a simulated signal.

```
% Signal simulation
[x,t] = helperPolarizationSignal;

% Create radiator and collector
radiator = phased.Radiator('Sensor',txAntenna,'Polarization','Combined',...
    'OperatingFrequency',fc,'PropagationSpeed',3e8);
collector = ...
    phased.Collector('Sensor',rxAntenna,'Polarization','Combined',...
    'OperatingFrequency',fc,'PropagationSpeed',3e8);

% Signal transmission and reception
xt = radiator(x,ang_t,lclaxes_t);
y = collector(xt,ang_r,lclaxes_r);

helperPolarizationSignalPlot(t,x,y,'vertically')
```
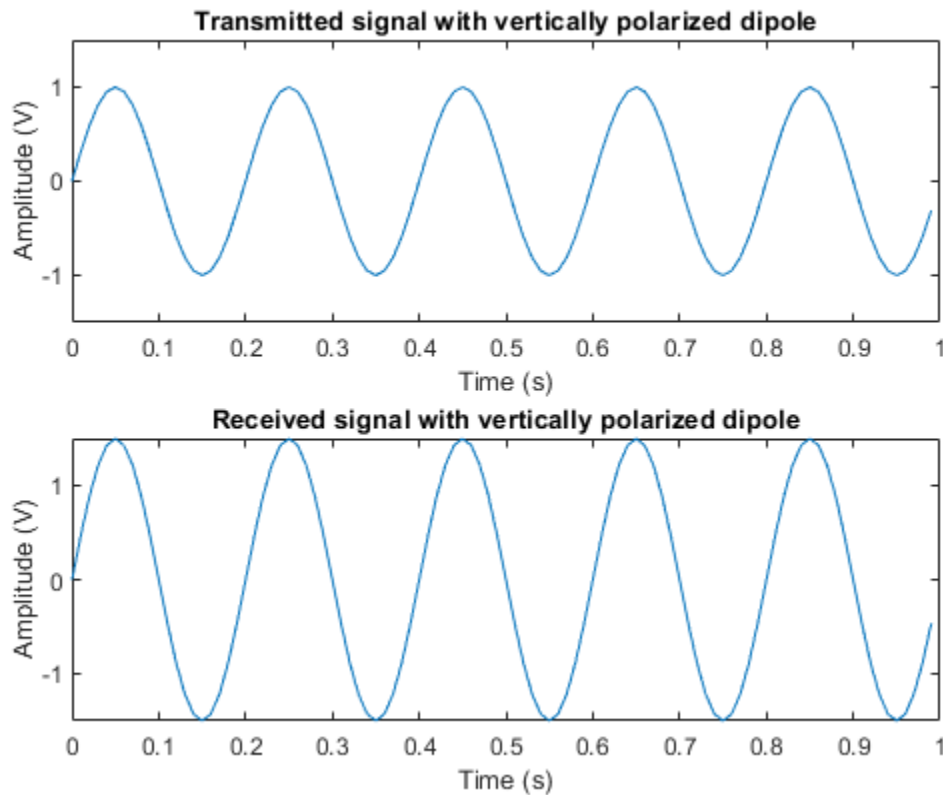
The figure shows that the signal is received with no loss. Each short dipole antenna provides a gain of 1.76 dB, so the received signal is 1.5 times stronger than the transmitted signal.

If instead a horizontally polarized antenna is used to receive the signal, the two antennas are now orthogonal in polarization and as a result, no power will be delivered to the received antenna. The polarization loss can be found by

```
rxAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Y');
resp_r = rxAntenna(fc,ang_r);

ploss = polloss([resp_t.H;resp_t.V],[resp_r.H;resp_r.V],pos_r,lclaxes_r)
```

```
ploss =

    Inf
```

This process can be better understood using the following diagram.
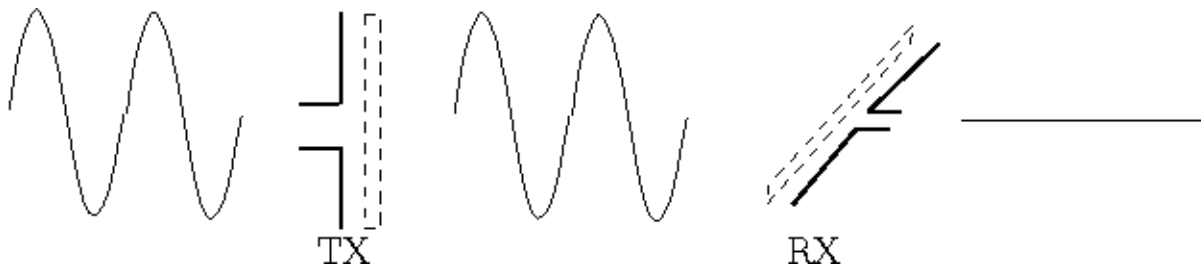
As the diagram shows, the polarization of an antenna can be seen as a filter blocking out any polarized wave that is orthogonal to the antenna's own polarization state.

As expected, the signal simulation shows that the received signal is 0.

```
collector = ...
    phased.Collector('Sensor',rxAntenna,'Polarization','Combined',...
    'OperatingFrequency',fc,'PropagationSpeed',3e8);

% Signal transmission and reception
xt = radiator(x,ang_t,lclaxes_t);
y = collector(xt,ang_r,lclaxes_r);

helperPolarizationSignalPlot(t,x,y,'horizontally')
```
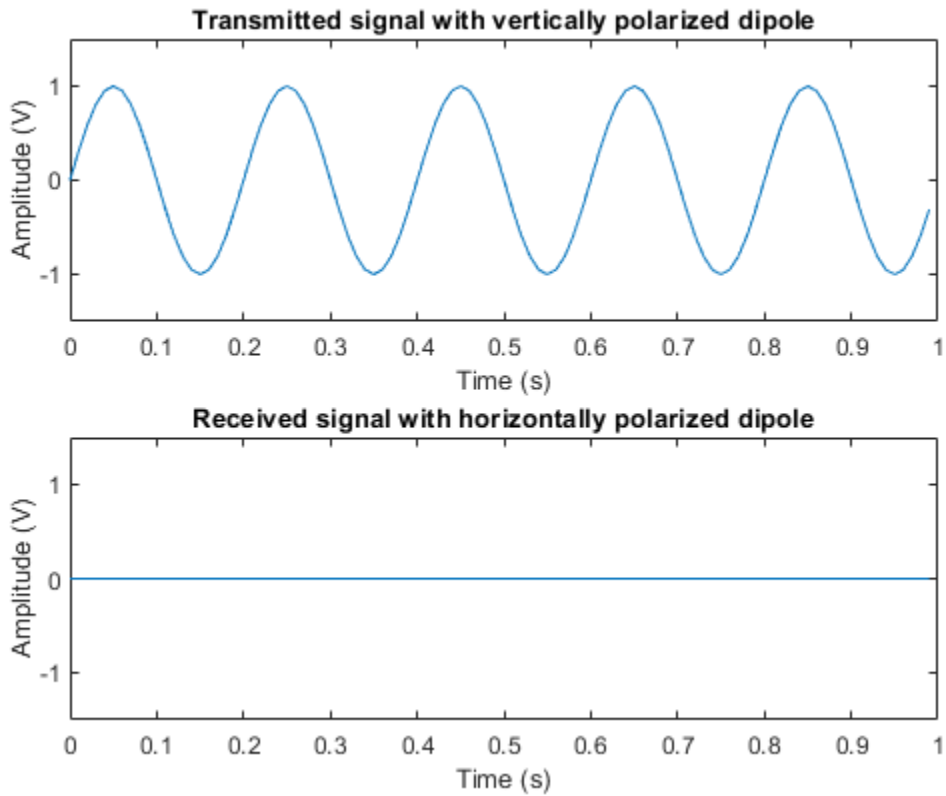
One can rotate the receive antenna to get a partial match in polarization. For instance, assume the receiving antenna in the previous example is rotated 45 degrees around x axis, then the received signal is no longer 0, although not as strong as when the polarizations are matched.

```
% Rotate axes
lclaxes_r = rotx(45)*azelaxes(180,0);

% Signal transmission and reception
xt = radiator(x,ang_t,lclaxes_t);
y = collector(xt,ang_r,lclaxes_r);

helperPolarizationSignalPlot(t,x,y,'45 degree')
```
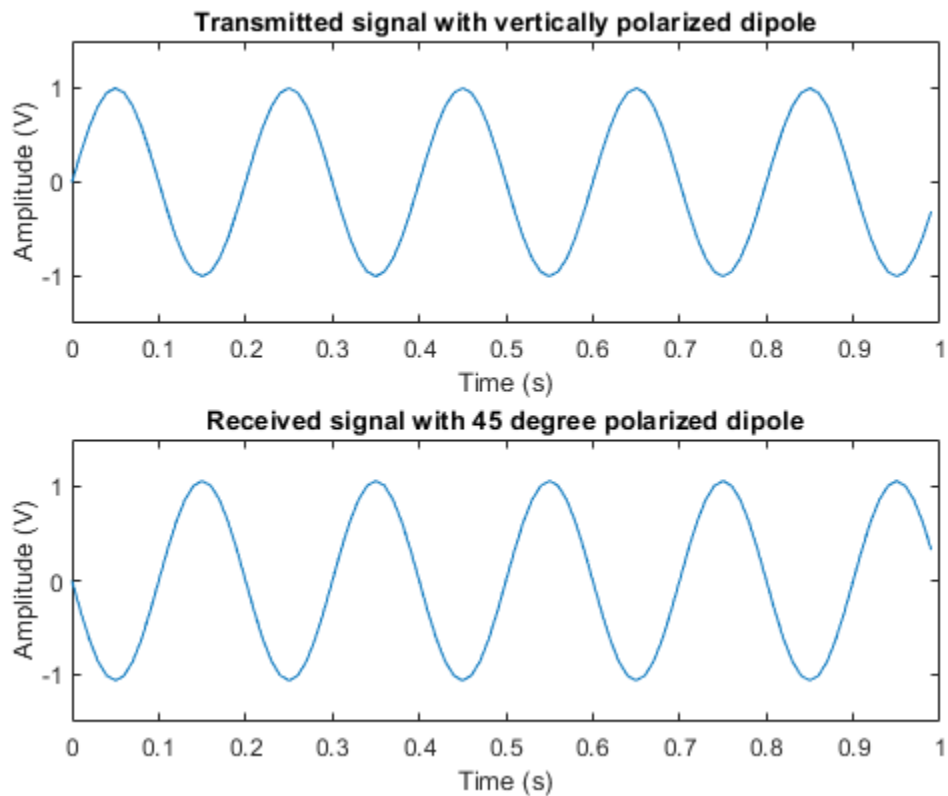


The corresponding polarization loss is

```
ploss = polloss([resp_t.H;resp_t.V],[resp_r.H;resp_r.V],pos_r,lclaxes_r)

% measured in dB.

ploss =

    3.0103
```

**Target Polarization Signature**

When an electromagnetic wave hits the target, the wave will be scattered off the target and some energy will be transferred between two orthogonal polarization components. Therefore, the target scattering mechanism is often modeled by a 2x2 radar cross section (RCS) matrix (also known as scattering matrix), whose diagonal terms specify how the target scatters the energy into the original H and V polarization component and off diagonal terms specify how the target scatters the energy into the opposite polarization component.

Because the transmit and receive antennas can have any combination of polarizations, it is often of interest to look at the polarization signature for a target for different polarization configurations. The signature plots the received power under different polarizations as a function of the tilt angle and the ellipticity angle of the transmit polarization ellipse. This can also be seen as a measure of the effective RCS. Two most widely used polarization signatures (also known as polarization responses), are co-polarization (co-pol) response and cross polarization (cross-pol) response. Co-pol response uses the same polarization for both transmit and receive while the cross-pol response uses the orthogonal polarization to receive.

The simplest target is a sphere, whose RCS matrix is given by [1 0;0 1], meaning that the reflected polarization is the same as the incident polarization. The polarization signature for a sphere is given by

```
s = eye(2);
subplot(211); polsignature(s,'c');
subplot(212); polsignature(s,'x');
```
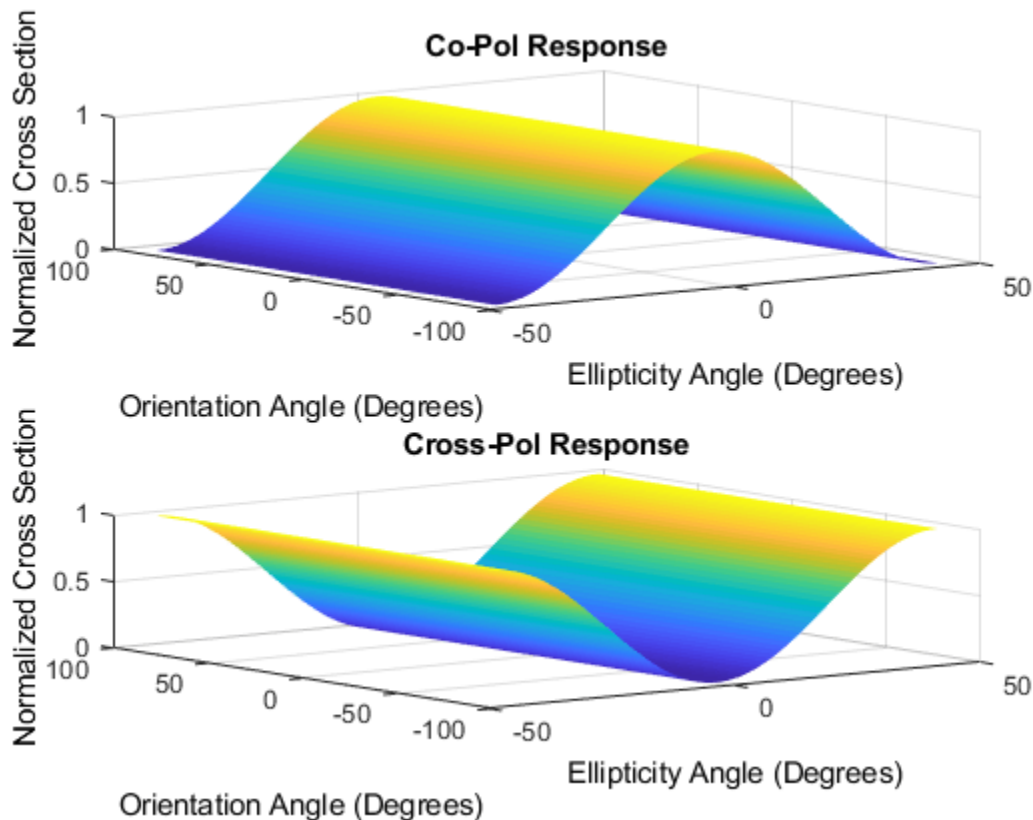
From the plot, it can be seen that for such a target, a linear polarization, where the ellipticity angle is 0, generates the maximum return in a co-pol setting while a circular polarization, where the ellipticity angle is either 45 or -45 degrees, generates the maximum return in a cross-pol configuration.

A more complicated target is a dihedral, which is essentially a corner that reflects the wave twice, as shown in left side of the following sketch:
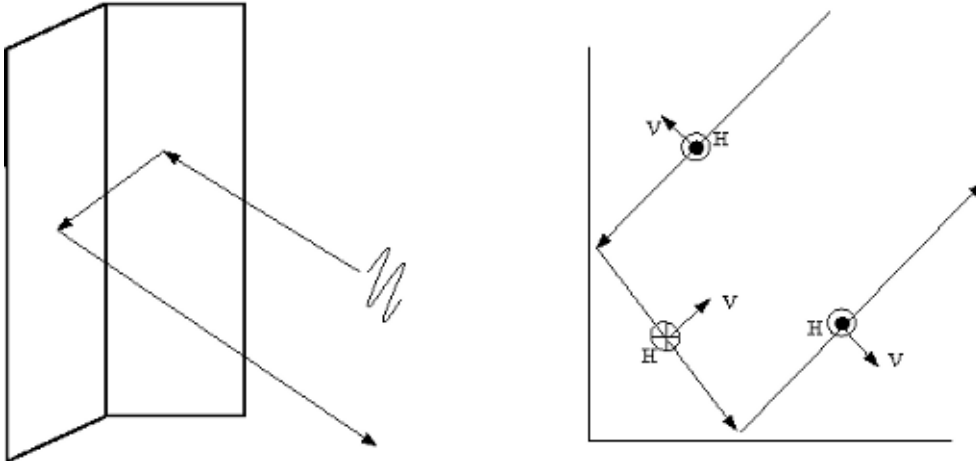


The right side of the above figure shows how the polarization field changes along the two reflections. After the two reflections, the horizontal polarization component remains the same while the vertical polarization component is reversed. Hence, its cross section matrix and polarization signature are give by

```
s = [1 0;0 -1];
subplot(211); polsignature(s,'c')
subplot(212); polsignature(s,'x')
```
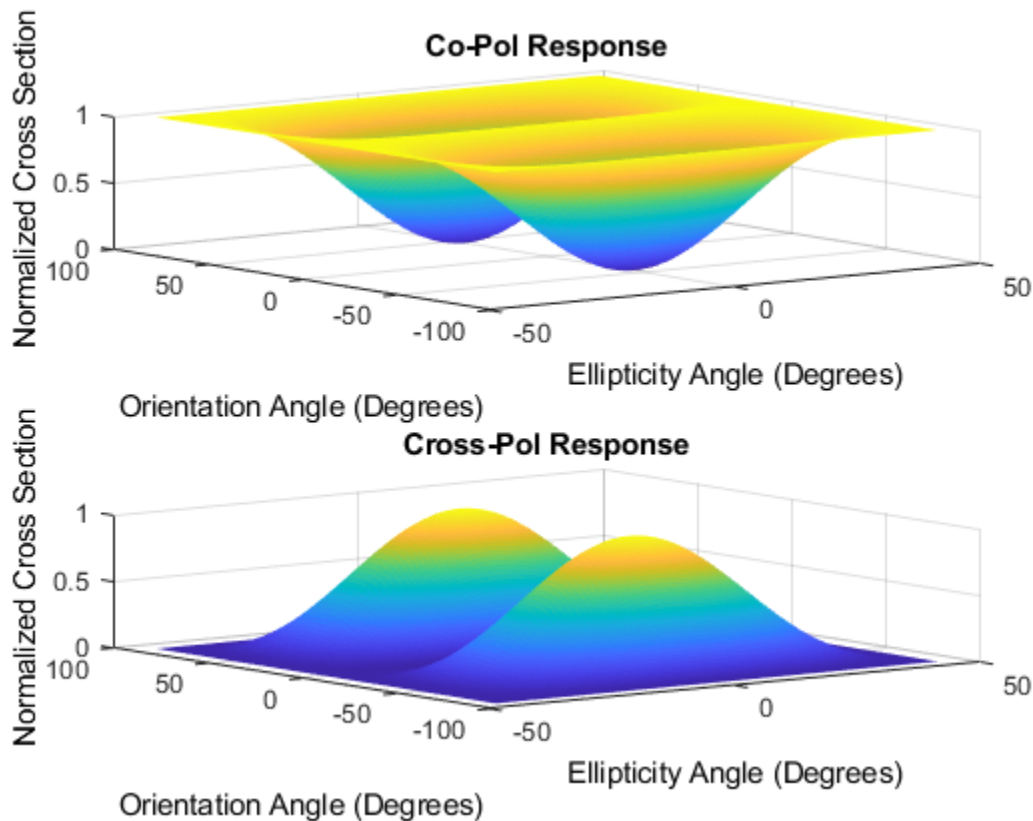
The signature shows that the circular polarization works best in a co-pol setting while 45-degree linear polarization works best in a cross-pol situation.

**Simulation of Polarized Signal Propagation Using Antenna and Target**

Putting everything together, the polarized signal is first transmitted by an antenna, then bounces off a target, and finally gets received at the receive antenna. Next is a simulation for this signal flow.

The simulation assumes a vertical dipole as the transmit antenna, a horizontal dipole as the receive antenna, and a target whose RCS matrix is [0 1;1 0], which flips the signal's polarization. For the illustration purpose, the propagation in free space is ignored because it does not affect the polarization. It is also assumed that the transmit antenna, the target, and the receive antenna are on a line along the transmit antenna's boresight. The local coordinate system is the same for the transmit antenna and the target. The receive antenna is facing the transmit antenna.

```
% Define transmit and antenna
txAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
rxAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Y');
radiator = phased.Radiator('Sensor',txAntenna,'Polarization','Combined');
collector = phased.Collector('Sensor',rxAntenna,'Polarization','Combined');

% Simulate signal
[x,t] = helperPolarizationSignal;

% Incident and arriving angles
ang_tx = [0;0];
ang_tgt_in = [180;0];
```

```
ang_tgt_out = [0;0];
ang_rx = [0;0];

% Local coordinate system
lclaxes_tx = azelaxes(0,0);
lclaxes_tgt = lclaxes_tx;
lclaxes_rx = azelaxes(180,0);

% Define target
target = phased.RadarTarget('EnablePolarization',true,...
    'Mode','Bistatic','ScatteringMatrix',[0 1;1 0]);

% Simulate received signal
xt = radiator(x,ang_tx,lclaxes_tx);                     % radiate
xr = target(xt,ang_tgt_in,ang_tgt_out,lclaxes_tgt);     % reflect
y = collector(xr,ang_rx,lclaxes_rx);                    % collect

helperPolarizationSignalPlot(t,x,y,'horizontally');
```
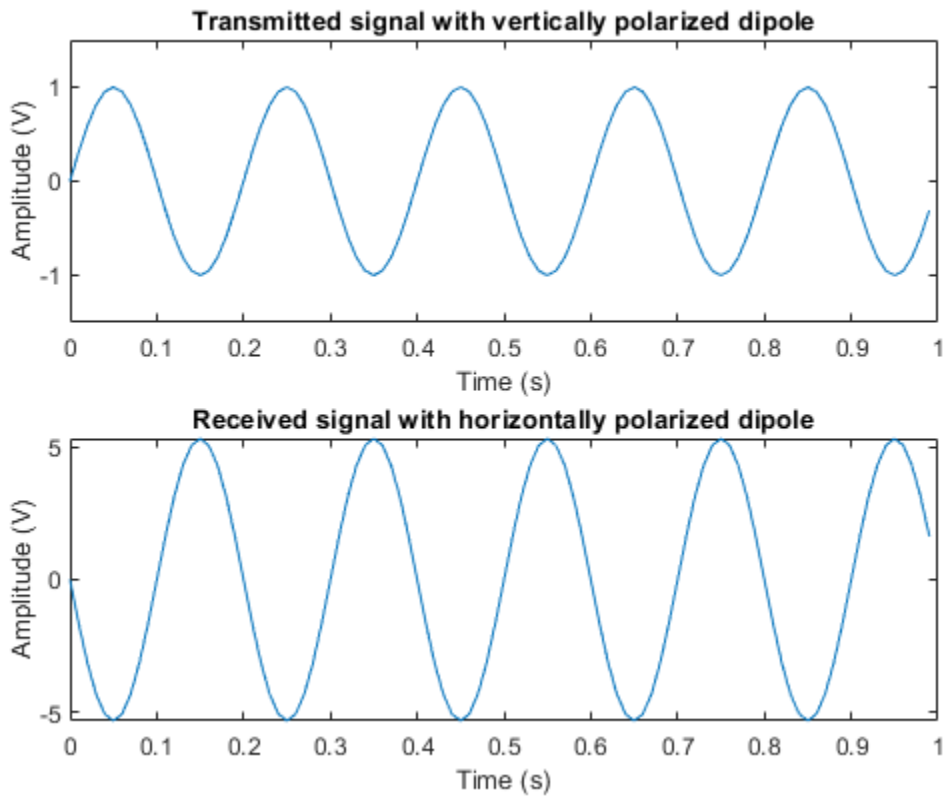
Note that because the target flips the polarization components, the horizontally polarized antenna is able to receive the signal sent with a vertically polarized antenna.

**Summary**

This example reviews the basic concepts of polarization and introduces how to analyze and model polarized antennas and targets using Phased Array System Toolbox.

# Modeling Mutual Coupling in Large Arrays Using Embedded Element Pattern

The pattern multiplication principle states that the radiation pattern of an array can be considered as the multiplication of the element pattern and the array factor. However, when an antenna gets deployed into an array, its radiation pattern is modified by its neighboring elements. This effect is often referred to as mutual coupling. Thus, to improve the fidelity of the analysis, one should use the element pattern with mutual coupling effect in the pattern multiplication instead of the isolated element (an element located in space by itself) pattern.

Unfortunately it is often very difficult to model the exact mutual coupling effect among elements. This example shows one possible approach to model the mutual coupling effects via an embedded pattern, which refers to the pattern of a single element embedded in a finite array. The element of choice is in general at the center of the array. The embedded pattern is calculated or measured by transmitting through the element itself while terminating all other elements in the array with a reference impedance [1]-[3]. This approach works well when the array is large so the edge effects may be ignored.

The example models two arrays: first using the pattern of the isolated element, second with the embedded element pattern and compare the results of the two with the full-wave Method of Moments (MoM) based solution of the array. The array performance for scanning at broadside, and for scanning off broadside is established. Finally, the array spacing is adjusted to investigate the occurrence of scan blindness and compare against reference results [3].

This example requires Antenna Toolbox™.

### Model an Array of Dipoles Using Isolated Element Pattern

First, we design an array with isolated element. For this example we choose the center of the X-band as our design frequency.

```
freq = 10e9;
vp = physconst('lightspeed');
lambda = vp/freq;
```

In [4], it was discussed that the central element of a 5 $\lambda$ X 5 $\lambda$ array, where $\lambda$ is the wavelength, starts to behave like it is in an infinite array. Such an aperture would correspond to a 10 X 10 array of half-wavelength spaced radiators. We choose to slightly exceed this limit and consider a 11 X 11 array of dipoles.

```
Nrow = 11;
Ncol = 11;
drow = 0.5*lambda;
dcol = 0.5*lambda;
```

The dipole of choice has a length slightly lower than $\lambda/2$ and a radius of approximately $\lambda/150$.

```
mydipole = dipole;
mydipole.Length = 0.47*lambda;
mydipole.Width = cylinder2strip(0.191e-3);
figure('Color','w');
show(mydipole);
```
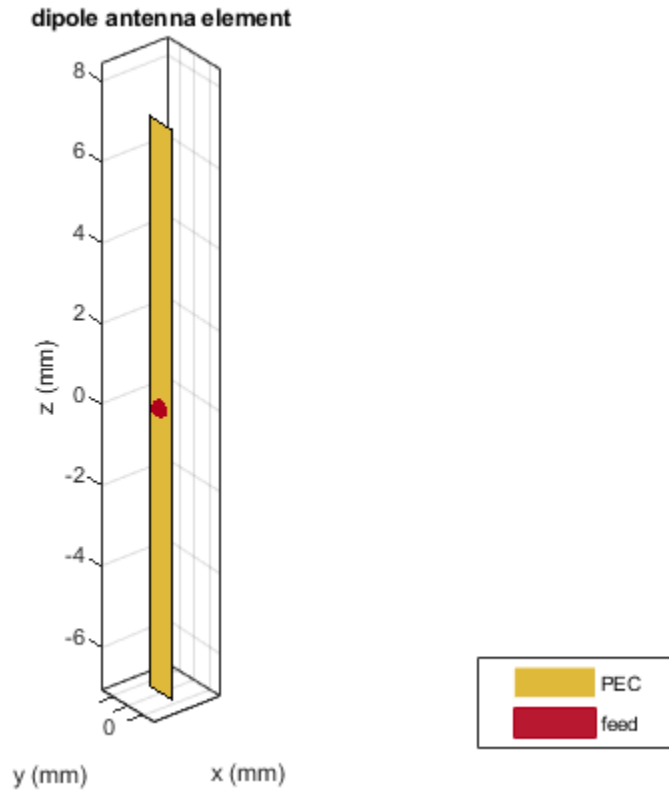
Now create an 11 X 11 URA and assign the isolated dipole as its element. Adjust the element spacing to be half-wavelength at 10 GHz. The dipole tilt is set to zero so its orientation matches the array geometry in the Y-Z plane.

```
isolatedURA = phased.URA;
isolatedURA.Element = mydipole;
isolatedURA.Size = [Nrow Ncol];
isolatedURA.ElementSpacing = [drow dcol];
viewArray(isolatedURA);
myFigure = gcf;
myFigure.Color = 'w';
```

Array Geometry



Aperture Size:
Y axis = 164.886 mm
Z axis = 164.886 mm
Element Spacing:
$\Delta$ y = 14.99 mm
$\Delta$ z = 14.99 mm
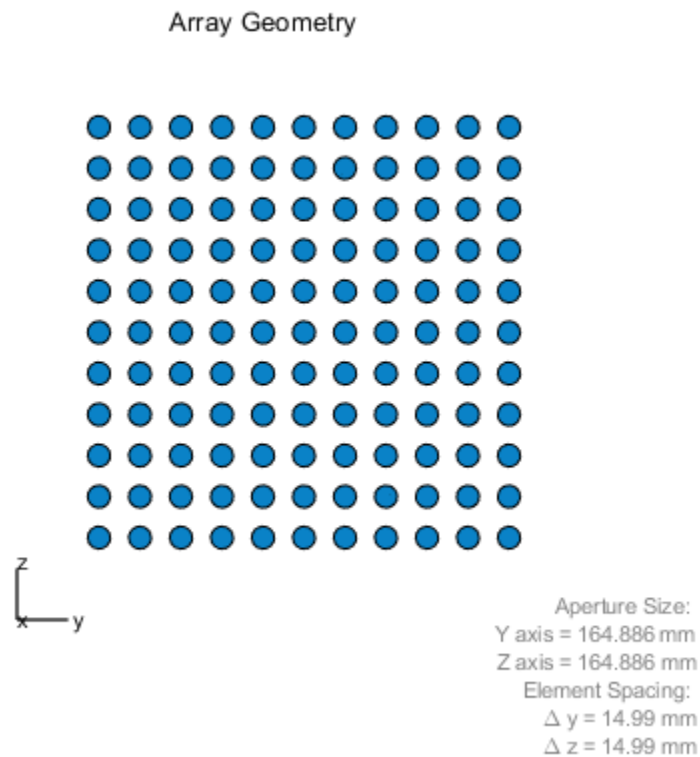
**Model Array of Dipoles Using Embedded Element Pattern**

To compute the embedded pattern of the center dipole element, we first create a full-wave model of the previous array. Since the default orientation of the dipole element in the library is along z-axis, we tilt it so that the array is formed in the X-Y plane.

```
fullWaveArray = rectangularArray(...
    'Size',[Nrow Ncol],...
    'RowSpacing',drow,...
    'ColumnSpacing',dcol);
fullWaveArray.Element = mydipole;
fullWaveArray.Element.Tilt = 90;
fullWaveArray.Element.TiltAxis = [0 1 0];
show(fullWaveArray)
title('Rectangular 11 X 11 Array of Dipole Antennas')
```
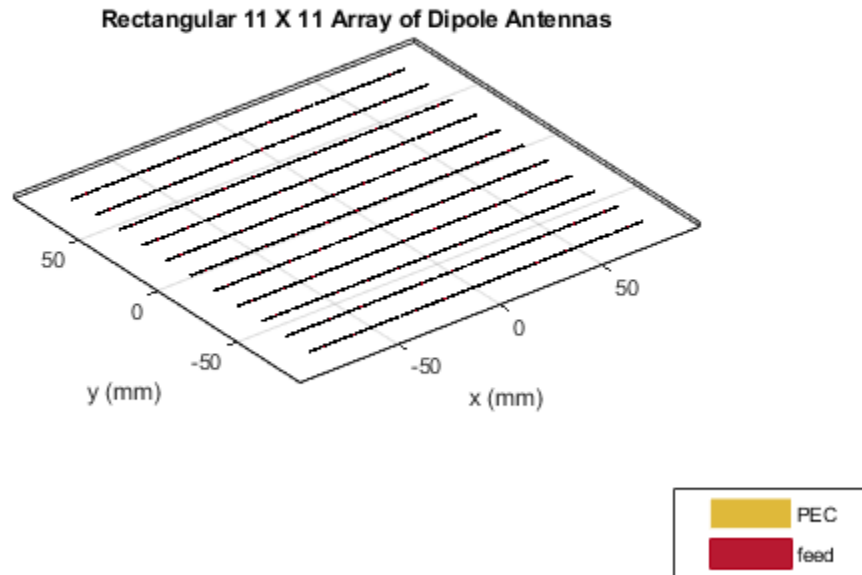
To calculate the embedded element pattern, use the `pattern` function and pass in additional input parameters of the element number (index of the center element) and termination resistance. The scan resistance and scan reactance for an infinite array of resonant dipoles spaced $\lambda/2$ apart is provided in [3] and we choose the resistance at broadside as the termination for all elements.

```
Zinf = 76 + 1i*31;
ElemCenter = (prod(fullWaveArray.Size)-1)/2 + 1;
az = -180:2:180;
el = -90:2:90;
EmbElFieldPatCenter = pattern(fullWaveArray,freq,az,el,...
    'ElementNumber',ElemCenter,'Termination',real(Zinf),'Type','efield');
```

Import this embedded element pattern into a custom antenna element and create the same rectangular array using that element. Since the array will be in the Y-Z plane, rotate the pattern to match the scan planes.

```
embpattern = helperRotatePattern(az,el,EmbElFieldPatCenter,[0 1 0],90);
embpattern = mag2db(embpattern);
fmin = freq - 0.1*freq;
fmax = freq + 0.1*freq;
freqVector = [fmin fmax];
embantenna = phased.CustomAntennaElement('FrequencyVector',freqVector,...
    'AzimuthAngles',az,'ElevationAngles',el,...
    'MagnitudePattern',embpattern,'PhasePattern',zeros(size(embpattern)));

embeddedURA = phased.URA;
embeddedURA.Element = embantenna;
```

```
embeddedURA.Size = [Nrow Ncol];
embeddedURA.ElementSpacing = [drow dcol];
```

**Compare Array Pattern in Elevation and Azimuth Plane**

Next, calculate and compare the patterns in different planes for the three arrays: the one using the isolated element pattern, the one using the embedded element pattern, and the full-wave model (used as the ground truth).

First, the pattern in the elevation plane (specified by azimuth = 0 deg and also called the E-plane)

```
Eplane_embedded = pattern(embeddedURA,freq,0,el);
Eplane_isolated = pattern(isolatedURA,freq,0,el);
[Eplane_fullwave,~,el3e] = pattern(fullWaveArray,freq,0,0:1:180);
el3e = el3e'-90;

helperATXPatternCompare([el(:) el(:) el3e(1:2:end)],...
    [Eplane_isolated Eplane_embedded Eplane_fullwave(1:2:end)],...
    'Elevation Angle (deg.)','Directivity (dBi)',...
    'E-plane Array Directivity Comparison',...
    {'With Isolated Pattern','With Embedded Pattern',...
    'Full Wave Solution'},[-60 30]);
```
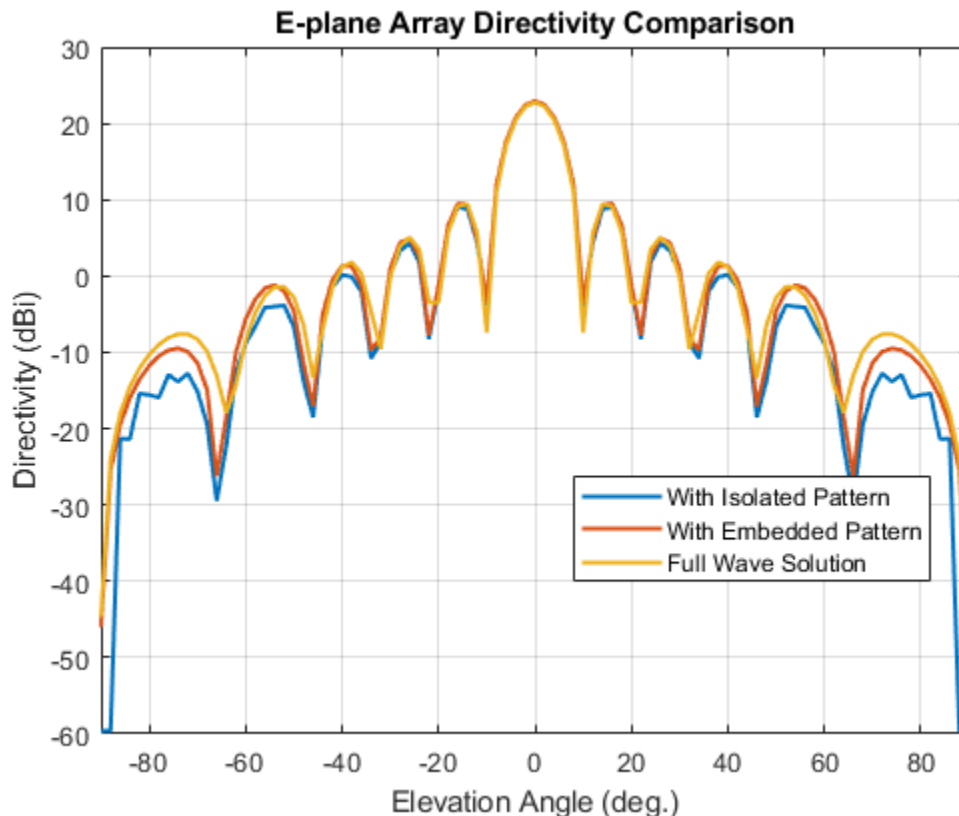


Now, the pattern in the azimuth plane (specified by elevation = 0 deg and called the H-plane).

```
Hplane_embedded = pattern(embeddedURA,freq,az/2,0);
Hplane_isolated = pattern(isolatedURA,freq,az/2,0);
Hplane_fullwave = pattern(fullWaveArray,freq,90,0:1:180);
```

```
helperATXPatternCompare([az(:)/2 az(:)/2 el3e],...
    [Hplane_isolated Hplane_embedded Hplane_fullwave],...
    'Azimuth Angle (deg.)','Directivity (dBi)',...
    'H-plane Array Directivity Comparison',...
    {'With Isolated Pattern','With Embedded Pattern',...
    'Full Wave Solution'},[-60 30]);
```
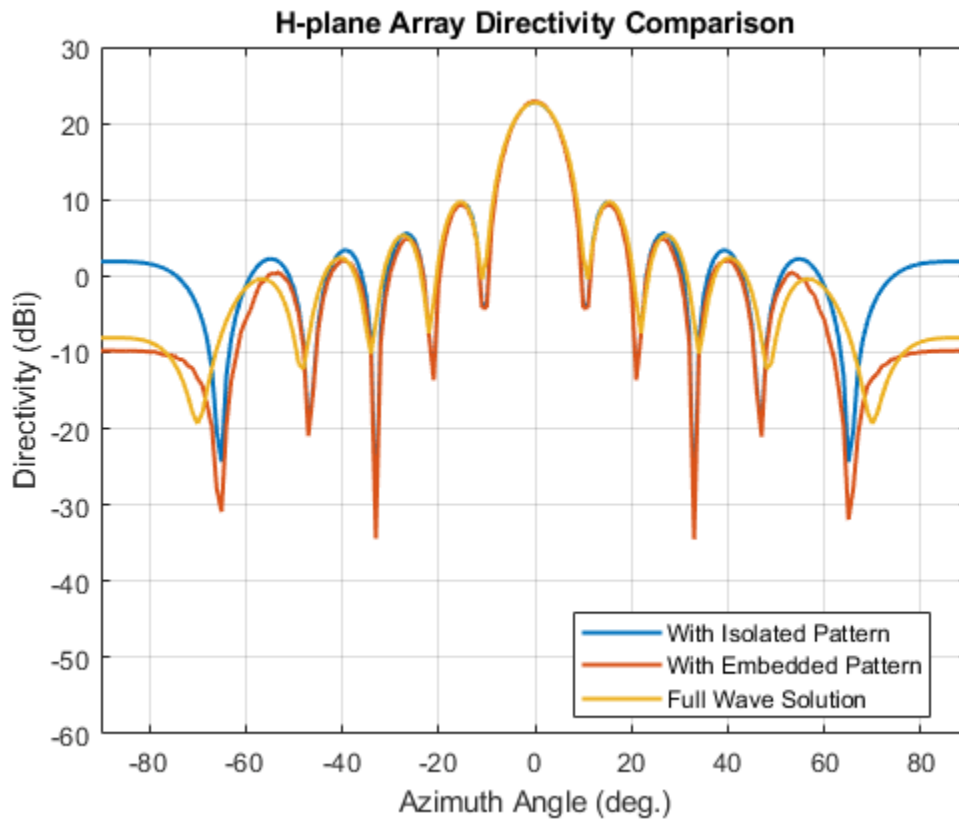


The array directivity is approximately 23 dBi. This result is close to the theoretical calculation for the peak directivity [5] after taking into account the absence of a reflector, $D = 4\,\pi\,A\,/\,\lambda^2\,Nrow Ncol$, $A = drow * dcol$.

The pattern comparison suggests that the main beam and the first sidelobes are aligned for all three cases. Moving away from the main beam shows the increasing effect of coupling on the sidelobe level. As expected, the embedded element pattern approach suggests a coupling level in between the full-wave simulation model and the isolated element pattern approach.

**Increase Array Size**

The behavior of the array pattern is intimately linked to the embedded element pattern. To understand how our choice of an 11 X 11 array impacts the center element behavior, we increase the array size to a 25 X 25 array (12.5 $\lambda$ X 12.5 $\lambda$ aperture size). Note that the triangular mesh size for the full wave Method of Moments (MoM) analysis with 625 elements increases to 25000 triangles (40 triangles per dipole) and the computation for the embedded element pattern takes approximately 12 minutes on a 2.4 GHz machine with 32 GB memory. This time can be reduced by lowering the mesh size per element by meshing manually using a maximum edge length of $\lambda/20$.

Below is the pattern plot for the E-plane,

```
load atexdipolearray
embpattern = helperRotatePattern(...
    DipoleArrayPatData.AzAngles,DipoleArrayPatData.ElAngles,...
    DipoleArrayPatData.ElemPat(:,:,3),[0 1 0],90);
embpattern = mag2db(embpattern);

embantenna2 = clone(embantenna);
embantenna2.AzimuthAngles = DipoleArrayPatData.AzAngles;
embantenna2.ElevationAngles = DipoleArrayPatData.ElAngles;
embantenna2.MagnitudePattern = embpattern;
embantenna2.PhasePattern = zeros(size(embpattern));

Eplane_embedded = pattern(embantenna2,freq,0,el);
Eplane_embedded = Eplane_embedded - max(Eplane_embedded); % normalize
Eplane_isolated = pattern(mydipole,freq,0,el);
Eplane_isolated = Eplane_isolated - max(Eplane_isolated); % normalize
embpatE = pattern(embantenna,freq,0,el);
embpatE = embpatE-max(embpatE);                           % normalize

helperATXPatternCompare([el(:) el(:) el(:)],...
    [Eplane_isolated embpatE Eplane_embedded],...
    'Elevation Angle (deg.)','Directivity (dBi)',...
    'Normalized E-plane Element Directivity Comparison',...
    {'Isolated Pattern','Embedded Pattern - 11 X 11',...
    'Embedded Pattern - 25 X 25'},[-50 5]);
```
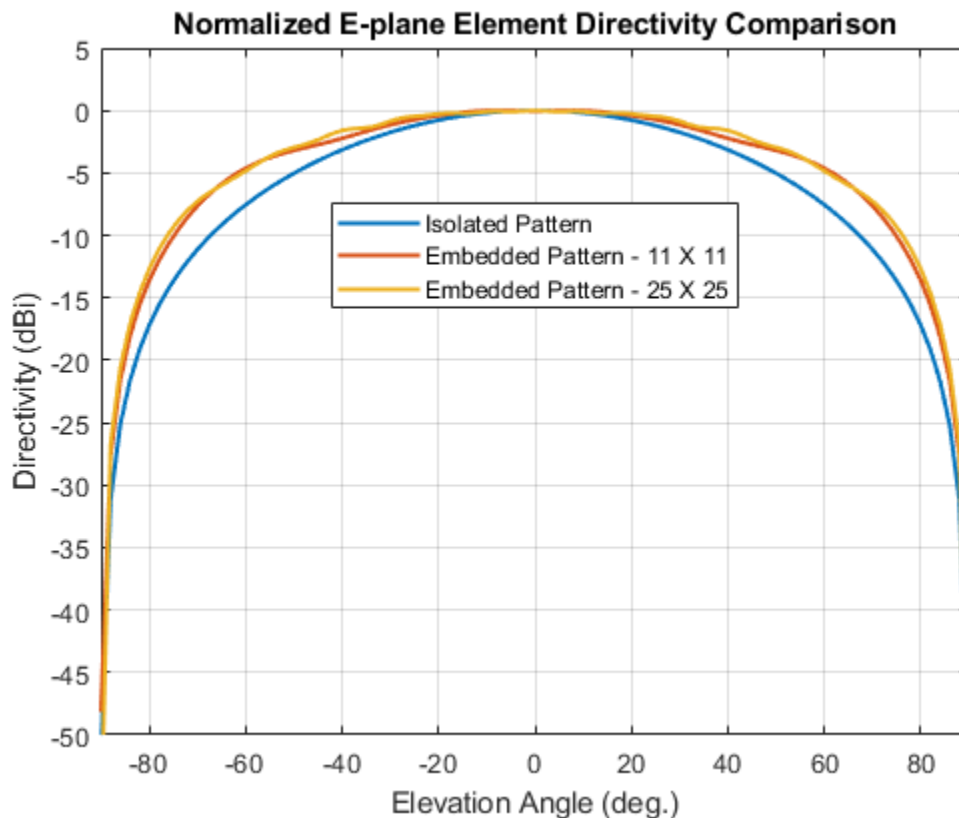
and the H-plane.

```
Hplane_embedded = pattern(embantenna2,freq,0,az/2);
Hplane_embedded = Hplane_embedded - max(Hplane_embedded);  % normalize
Hplane_isolated = pattern(mydipole,freq,0,az/2);
Hplane_isolated = Hplane_isolated - max(Hplane_isolated);  % normalize
embpatH = pattern(embantenna,freq,az/2,0);
embpatH = embpatH-max(embpatH);                            % normalize

helperATXPatternCompare([az(:)/2 az(:)/2 az(:)/2],...
    [Hplane_isolated embpatH Hplane_embedded],...
    'Azimuth Angle (deg.)','Directivity (dBi)',...
    'Normalized H-plane Element Directivity Comparison',...
    {'Isolated Pattern','Embedded Pattern - 11 X 11',...
    'Embedded Pattern - 25 X 25'},[-50 5]);
```
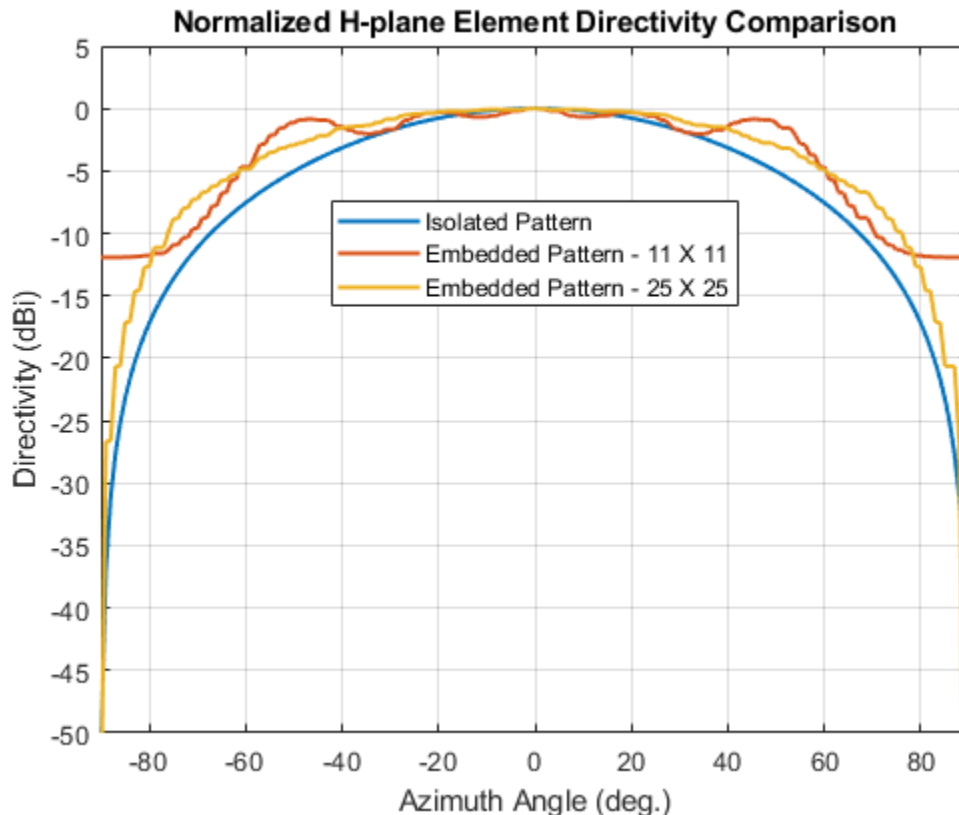


The plot above reveals that the difference between embedded element patterns of the 11 X 11 and the 25 X 25 array, respectively, is less than 0.5 dB, in the E-plane. However, the H-plane shows more variation for the 11 X 11 array as compared with the 25 X 25 array.

**Scan Behavior and Embedded Element Pattern**

This section scans the array based on the embedded element pattern in the elevation plane defined by azimuth = 0 deg and plot the normalized directivity. In addition, the normalized embedded element pattern is also plotted. Note the overall shape of the normalized array pattern approximately follows the normalized embedded element pattern, just as predicted by the pattern multiplication principle.

```
eplane_indx = find(az==0);
scan_el1 = -30:10:30;
scan_az1 = zeros(1,numel(scan_el1));
scanEplane = [scan_az1;scan_el1];

% compute array scanning weights
steeringvec = phased.SteeringVector('SensorArray',embeddedURA,...
    'IncludeElementResponse',true);
weights = steeringvec(freq,scanEplane);

% array scanning
legend_string1 = cell(1,numel(scan_el1));
scanEPat = nan(numel(el),numel(scan_el1));
for i = 1:numel(scan_el1)
    scanEPat(:,i) = pattern(embeddedURA,freq,scan_az1(i),el,...
        'Weights',weights(:,i));
    legend_string1{i} = strcat('scan = ',num2str(scan_el1(i)));
end
scanEPat = scanEPat - max(max(scanEPat));                % normalize

helperATXPatternCompare(el(:),scanEPat,...
    'Elevation Angle (deg.)','Directivity (dBi)',...
    'E-plane Scan Comparison',legend_string1(1:end-1),[-50 5]);
hold on;
plot(el(:),embpatE,'-.','LineWidth',1.5);
legend([legend_string1,{'Embedded element'}],'location','best')
hold off;
```
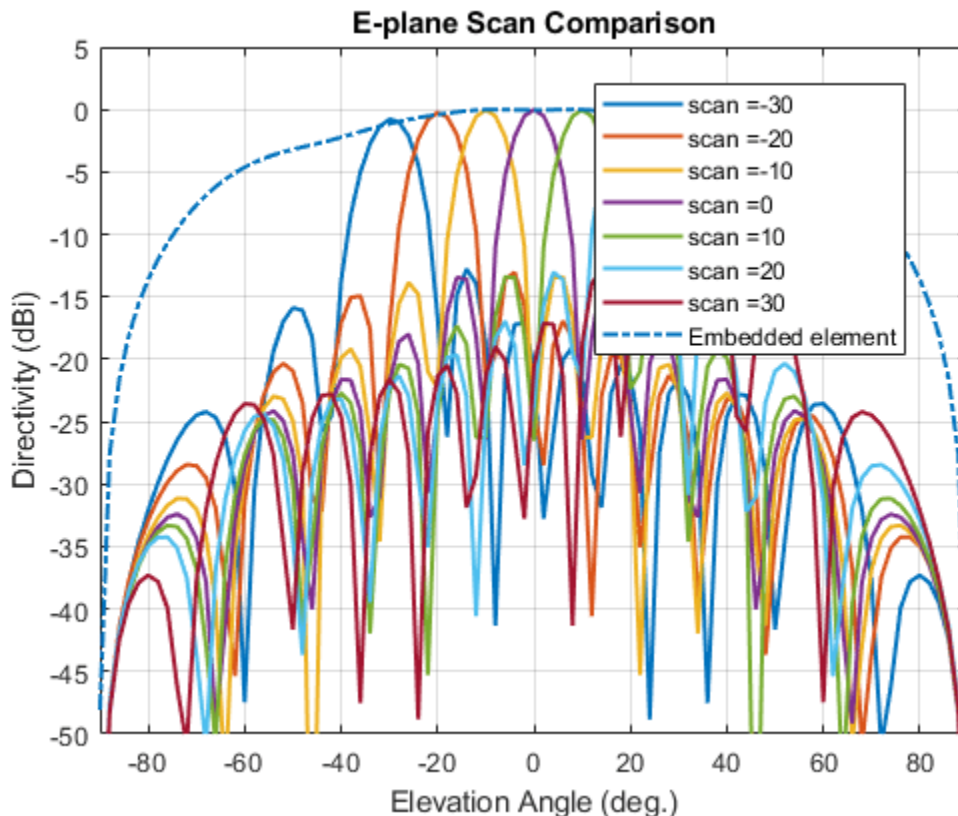
**Scan Blindness**

In large arrays, the array directivity can reduce drastically at certain scan angles under certain situations. At these scan angles, referred to as the blind angles, the array does not radiate the power supplied at its input terminals [3]. Two common mechanisms under which blindness conditions occur are

- Surface Wave Excitation
- Grating Lobe Excitation

It is possible to detect scan blindness in large finite arrays by studying the embedded element pattern (also known as array element pattern in the infinite array analysis). The array being investigated in this example does not have a dielectric substrate/ground plane, and therefore the surface waves are eliminated. However we can investigate the second mechanism, i.e. the grating lobe excitation. To do so, let us increase the spacing across rows and columns of the array to be $0.7\,\lambda$. Since this spacing is greater than the half-wavelength limit we should expect grating lobes in the visible space beyond a specific scan angle. As pointed out in [3], to accurately predict the depth of grating lobe blind angles in the finite array of dipoles, we need to have an array of the size 41 X 41 or higher. We will compare 3 cases, namely the 11 X 11, 25 X 25 and the 41 X 41 size arrays and check if the existence of blind angles can at least be observed in the 11 X 11 array. As mentioned earlier, the results were precomputed in Antenna Toolbox and saved in a MAT file. To reduce the computational time, the elements were meshed with maximum edge length of $\lambda/20$.

```
load atexdipolearrayblindness.mat
```
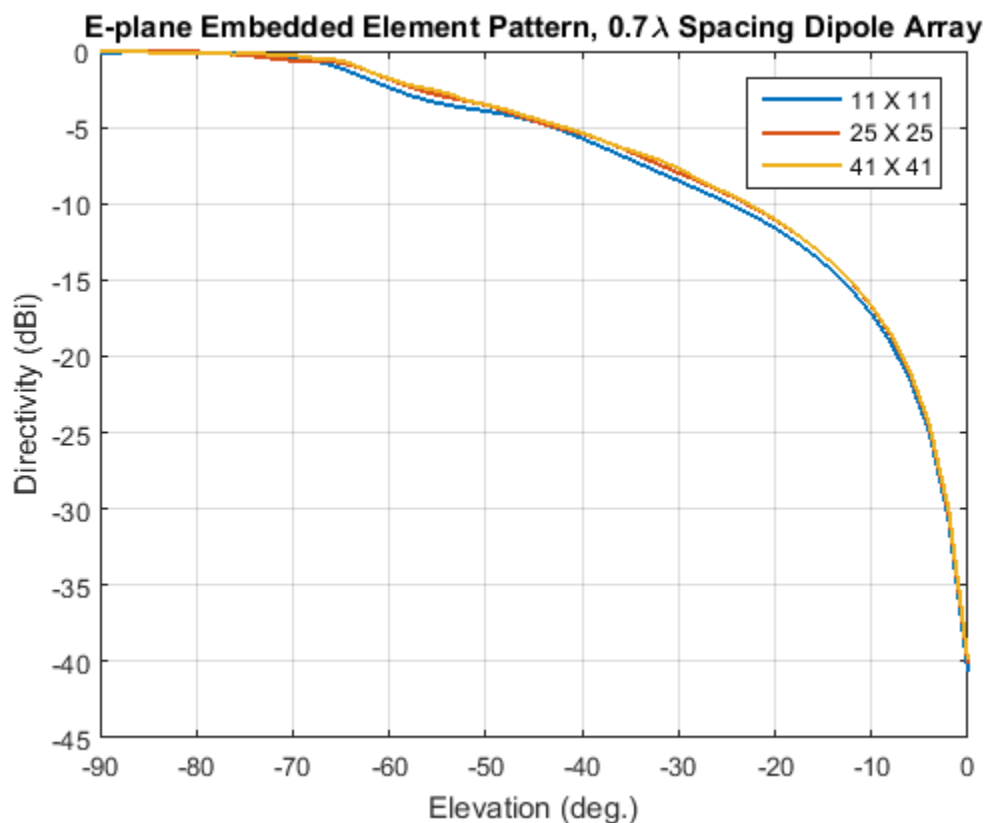


**The normalized E-plane embedded element pattern for arrays of three sizes**

**H - plane Embedded Element Pattern, 0.7λ Spacing Dipole Array**
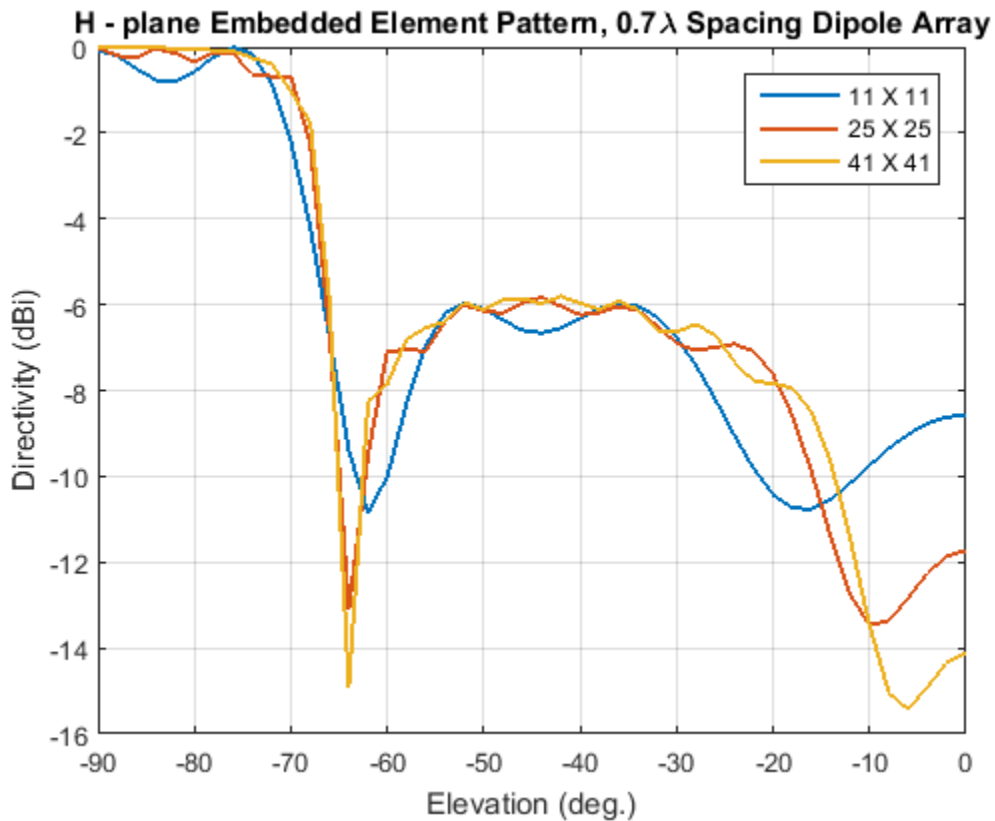
**The normalized H-plane embedded element pattern for arrays of three sizes. Notice the blind angle around -62 and -64 deg.**

**Conclusion**

The embedded element pattern approach is one possible way to perform the analysis of large finite arrays. They need to be large enough so that the edge effects can be ignored. The approach replaces the isolated element pattern with the embedded element pattern since the latter includes the effect of mutual coupling.

**Reference**

[1] R. J. Mailloux, 'Phased Array Antenna Handbook', Artech House,2nd edition, 2005

[2] W. Stutzman, G. Thiele, 'Antenna Theory and Design', John Wiley & Sons Inc., 3rd Edition, 2013.

[3] R. C. Hansen, Phased Array Antennas, Chapter 7 and 8, John Wiley & Sons Inc.,2nd Edition, 1998.

[4] H. Holter, H. Steyskal, "On the size requirement for finite phased-array models," IEEE Transactions on Antennas and Propagation, vol.50, no.6, pp.836-840, Jun 2002.

[5] P. W. Hannan, "The Element-Gain Paradox for a Phased-Array Antenna," IEEE Transactions on Antennas Propagation, vol. 12, no. 4, July 1964, pp. 423-433.

# Modeling Mutual Coupling in Large Arrays Using Infinite Array Analysis

This example uses infinite array analysis to model large finite arrays. The infinite array analysis on the unit cell reveals the scan impedance behavior at a particular frequency. This information is used with the knowledge of the isolated element pattern and impedance to calculate the scan element pattern. The large finite array is then modeled using the assumption that every element in the array possesses the same scan element pattern.

This example requires Antenna Toolbox™.

**Define Individual Element**

For this example we choose the center of the X-band as our design frequency.

```
freq = 10e9;
vp = physconst('lightspeed');
lambda = vp/freq;
ucdx = 0.5*lambda;
ucdy = 0.5*lambda;
```

Create a thin dipole of length slightly less than $\lambda/2$ and assign it as the exciter to a infinitely large reflector.

```
d = dipole;
d.Length = 0.495*lambda;
d.Width = lambda/160;
d.Tilt = 90;
d.TiltAxis = [0 1 0];

r = reflector;
r.Exciter = d;
r.Spacing = lambda/4;
r.GroundPlaneLength = inf;
r.GroundPlaneWidth = inf;
figure;
show(r);
```

**dipole over infinite ground plane**
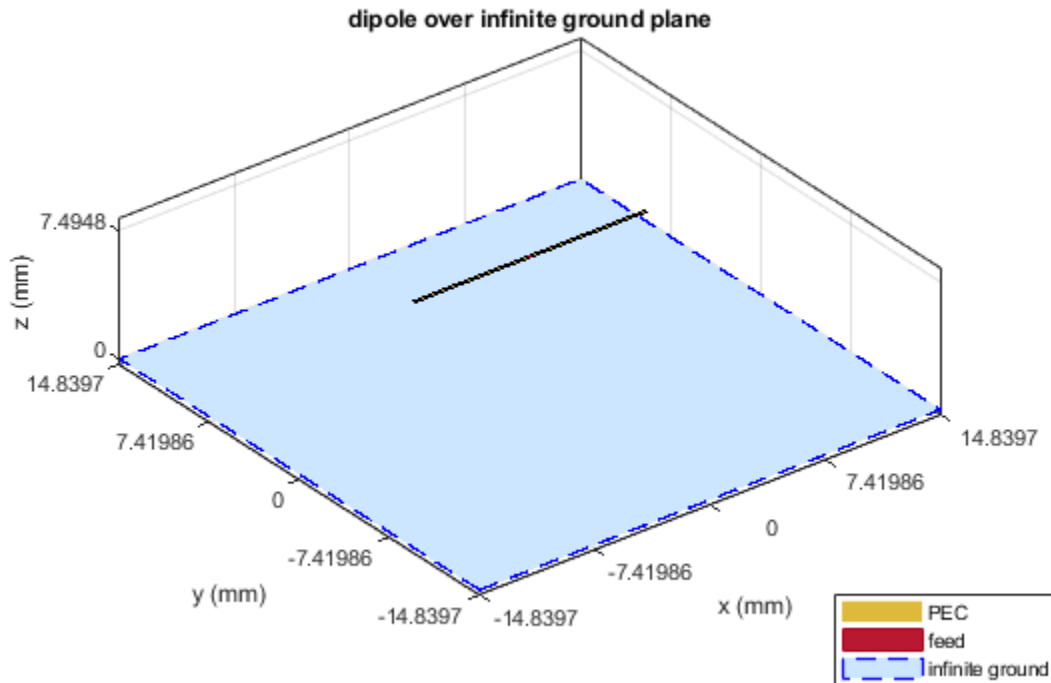


Calculate the isolated element pattern and the impedance of the above antenna. These results will be used to calculate the Scan Element Pattern(SEP). This term is also known as Array Element Pattern(AEP) or Embedded Element Pattern(EEP).

```
%Define az and el vectors
az = 0:2:360;
el = 90:-2:-90;

% Calculate power pattern
giso = pattern(r,freq,az,el,'Type','power');

% Calculated impedance
Ziso = impedance(r,freq);
```

**Calculate Infinite Array Scan Element Pattern**

**Unit Cell** In the infinite array analysis the term *unit cell* refers to a single element in an infinite array. The unit cell element needs a ground plane. Antennas that don't have a ground plane need to be backed by a reflector. A representative example for each case would be dipole backed by a reflector and a microstrip patch antenna. This example will use the dipole backed by a reflector and analyze the impedance behavior at 10 GHz as a function of scan angle. The unit cell will have a $\lambda/2 \times \lambda/2$ cross-section.

```
r.GroundPlaneLength = ucdx;
r.GroundPlaneWidth = ucdy;
infArray = infiniteArray;
infArray.Element = r;
```

```
infArray.ScanAzimuth = 30;
infArray.ScanElevation = 45;
figure;
show(infArray);
```

**Unit cell of dipole over a reflector in an infinite Array**
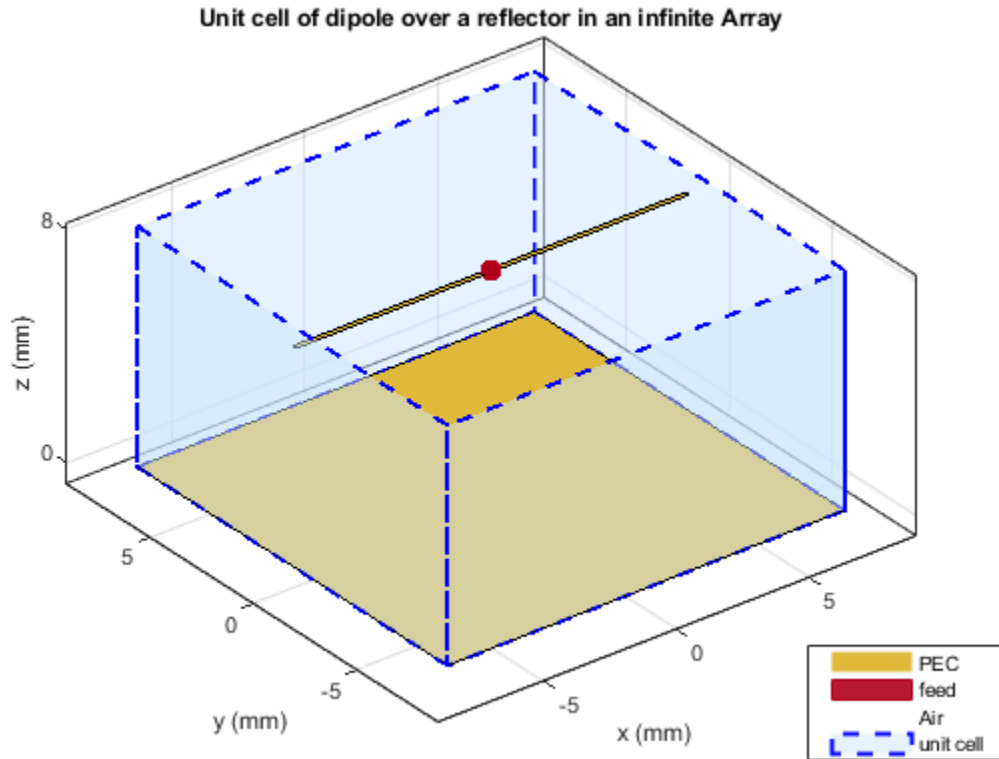


**Scan impedance** The scan impedance at a single frequency and single scan angle is shown below.

```
scanZ = impedance(infArray,freq)
```

```
scanZ =

   1.0860e+02 + 3.0442e+01i
```

For this example the scan impedance for the full volume of scan is calculated using 50 terms in the double summation for the periodic Greens function to improve convergence behavior.

**Scan Element Pattern /Array Element Pattern /Embedded Element Pattern** The scan element pattern (SEP) is calculated from the infinite array scan impedance, the isolated element pattern and the isolated element impedance. The expression used is shown here[1],[2]:

$$g_s(\theta) = \frac{4R_g R_i(g_i(\theta))}{|Z_s(\theta) + Z_g|^2}$$

```
load atexInfArrayScanZData
scanZ = scanZ.';
```

```
Rg = 185;
Xg = 0;
Zg = Rg + 1i*Xg;
gs = nan(numel(el),numel(az));
for i = 1:numel(el)
    for j = 1:numel(az)
        gs(i,j) = 4*Rg*real(Ziso).*giso(i,j)./(abs(scanZ(i,j) + Zg)).^2;
    end
end
```

**Build Custom Antenna Element**

The scan element pattern which represents a power pattern is used to build a custom antenna element.

```
fieldpattern = sqrt(gs);
bandwidth = 500e6;
customAntennaInf = helperATXBuildCustomAntenna(...
    fieldpattern,freq,bandwidth,az,el);
figure;
pattern(customAntennaInf,freq);
```
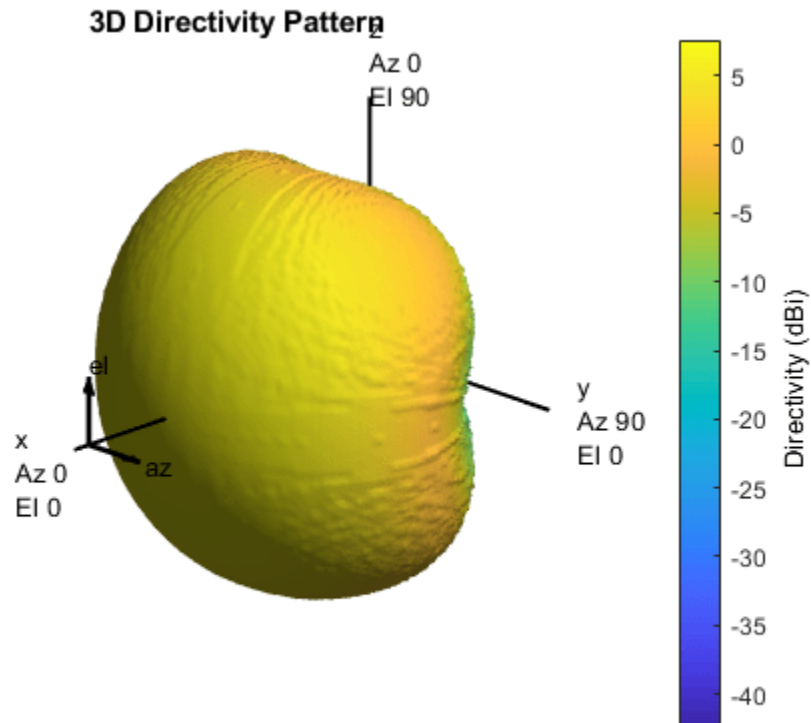


**Build 21 X 21 URA**

Create a uniform rectangular array (URA), with the custom antenna element, which has the scan element pattern.

```
N = 441;
Nrow = sqrt(N);
Ncol = sqrt(N);
drow = ucdx;
dcol = ucdy;
myURA1 = phased.URA;
myURA1.Element = customAntennaInf;
myURA1.Size = [Nrow Ncol];
myURA1.ElementSpacing = [drow dcol];
```

**Plot Slices in E and H Planes**

Calculate the pattern in the elevation plane (specified by azimuth = 0 deg and also called the E-plane) and azimuth plane (specified by elevation = 0 deg and called the H-plane) for the array built using infinite array analysis.

```
azang_plot = -90:0.5:90;
elang_plot = -90:0.5:90;
% E-plane
Darray1_E = pattern(myURA1,freq,0,elang_plot);
Darray1_Enormlz = Darray1_E - max(Darray1_E);
% H-plane
Darray1_H = pattern(myURA1,freq,azang_plot,0);
Darray1_Hnormlz = Darray1_H - max(Darray1_H);
% Scan element pattern in both planes
DSEP1_E = pattern(customAntennaInf,freq,0,elang_plot);
DSEP1_Enormlz = DSEP1_E - max(DSEP1_E);
DSEP1_H = pattern(customAntennaInf,freq,azang_plot,0);
DSEP1_Hnormlz = DSEP1_H - max(DSEP1_H);

figure
subplot(211)
plot(elang_plot,Darray1_Enormlz,elang_plot,DSEP1_Enormlz,'LineWidth',2)
grid on
axis([min(azang_plot) max(azang_plot) -40 0]);
legend('Array Pattern, az = 0 deg','Element Pattern')
xlabel('Elevation (deg)')
ylabel('Directivity (dB)')
title('Normalized Directivity')
subplot(212)
plot(azang_plot,Darray1_Hnormlz,azang_plot,DSEP1_Hnormlz,'LineWidth',2)
grid on
axis([min(azang_plot) max(azang_plot) -40 0]);
legend('Array Pattern, el = 0 deg','Element Pattern')
xlabel('Azimuth (deg)')
ylabel('Directivity (dB)')
```
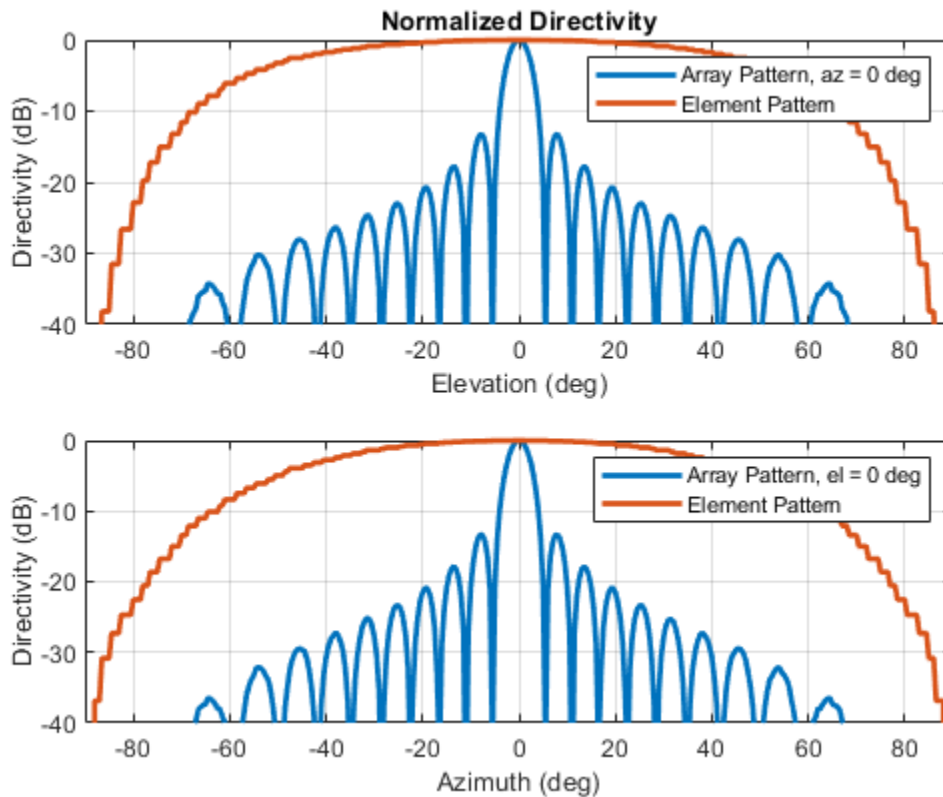
### Comparison with a Full Wave Finite Array Analysis

To understand the effect of the finite size of the array, we execute a full wave analysis of a 21 X 21 dipole array backed by an infinite reflector. The full wave array pattern slices in the E and H planes as well as the center element embedded element pattern is also calculated. This data is loaded from a MAT file. This analysis took approximately 630 seconds on a 2.4 GHz machine with 32 GB memory.

**Load Full Wave Data and Build Custom Antenna** Load the finite array analysis data, and use the embedded element pattern to build a custom antenna element. Note that the pattern from the full-wave analysis needs to be rotated by 90 degrees so that it lines up with the URA model built on the YZ plane.

```
load atexInfArrayDipoleRefArray
elemfieldpatternfinite = sqrt(FiniteArrayPatData.ElemPat);
arraypatternfinite = FiniteArrayPatData.ArrayPat;
bandwidth = 500e6;
customAntennaFinite = helperATXBuildCustomAntenna(...
    elemfieldpatternfinite,freq,bandwidth,az,el);
figure
pattern(customAntennaFinite,freq)
```
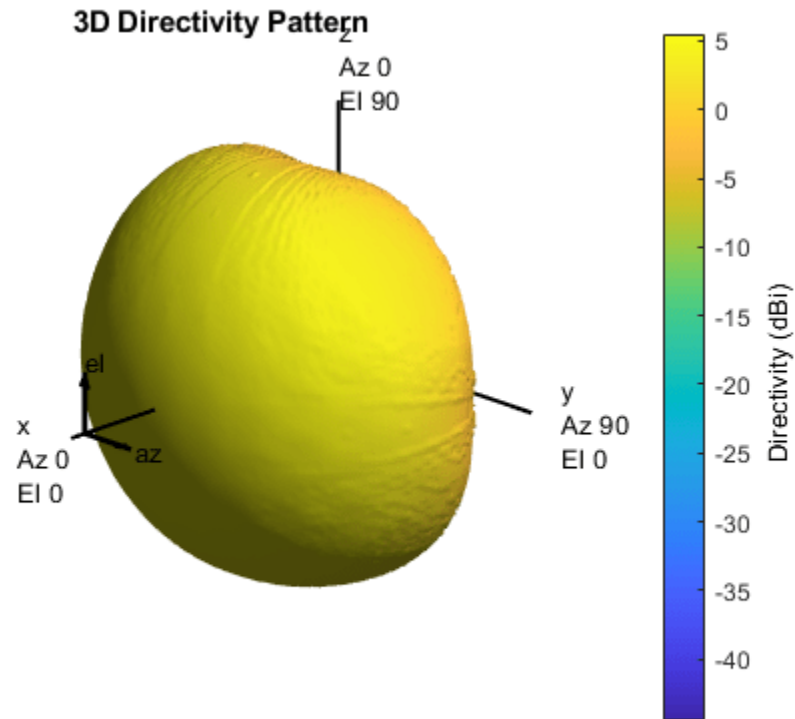
**Create Uniform Rectangular Array with Embedded Element Pattern** As done before create a uniform rectangular array with the custom antenna element.

```
myURA2 = phased.URA;
myURA2.Element = customAntennaFinite;
myURA2.Size = [Nrow Ncol];
myURA2.ElementSpacing = [drow dcol];
```

**E and H Plane Slice - Array With Embedded Element Pattern** Calculate the pattern slices in two orthogonal planes - E and H for the array with the embedded element pattern and the embedded element pattern itself. In addition since the full wave data for the array pattern is also available use this to compare results. E-plane

```
Darray2_E = pattern(myURA2,freq,0,elang_plot);
Darray2_Enormlz = Darray2_E - max(Darray2_E);
% H-plane
Darray2_H = pattern(myURA2,freq,azang_plot,0);
Darray2_Hnormlz = Darray2_H - max(Darray2_H);
```

**E and H Plane Slice - Embedded Element Pattern from Finite Array**

```
DSEP2_E = pattern(customAntennaFinite,freq,0,elang_plot);
DSEP2_Enormlz = DSEP2_E - max(DSEP2_E);
DSEP2_H = pattern(customAntennaFinite,freq,azang_plot,0);
DSEP2_Hnormlz = DSEP2_H - max(DSEP2_H);
```

**E and H Plane Slice - Full Wave Analysis of Finite Array**

```
azang_plot1 = -90:2:90;
elang_plot1 = -90:2:90;

Darray3_E = FiniteArrayPatData.EPlane;
Darray3_Enormlz = Darray3_E - max(Darray3_E);

Darray3_H = FiniteArrayPatData.HPlane;
Darray3_Hnormlz = Darray3_H - max(Darray3_H);
```

**Comparison of Array Patterns** The array patterns in the two orthogonal planes are plotted here.

```
figure
subplot(211)
plot(elang_plot,Darray1_Enormlz,elang_plot,Darray2_Enormlz,...
    elang_plot1,Darray3_Enormlz,'LineWidth',2)
grid on
axis([min(elang_plot) max(elang_plot) -40 0]);
legend('Infinite','Finite','Finite Full wave','location','best')
xlabel('Elevation (deg)')
ylabel('Directivity (dB)')
title('E-plane (az=0 deg) Normalized Array Directivity')
subplot(212)
plot(azang_plot,Darray1_Hnormlz,azang_plot,Darray2_Hnormlz,...
    azang_plot1,Darray3_Hnormlz,'LineWidth',2)
grid on
axis([min(azang_plot) max(azang_plot) -40 0]);
legend('Infinite','Finite','Finite Full wave','location','best')
xlabel('Azimuth (deg)')
ylabel('Directivity (dB)')
title('H-Plane (el = 0 deg) Normalized Array Directivity')
```
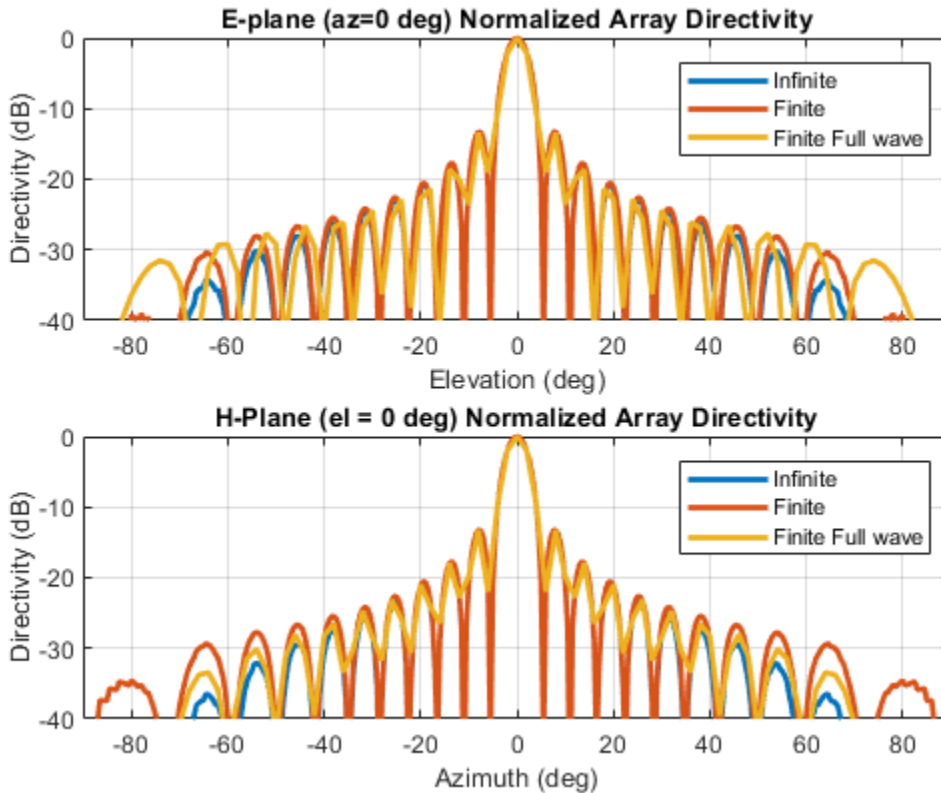
The pattern plots in the two planes reveals that all three analysis approaches suggest similar behavior out to +/-40 degree from boresight. Beyond this range, it appears that using the scan element pattern for all elements in a URA underestimates the sidelobe level compared to the full wave analysis of a finite array. The one possible reason for this could be the edge effect from the finite size array.

**Comparison of Element Patterns** The element patterns from the infinite array analysis and the finite array analysis are compared here.

```
figure
subplot(211)
plot(elang_plot,DSEP1_Enormlz,elang_plot,DSEP2_Enormlz,'LineWidth',2)
grid on
axis([min(azang_plot) max(azang_plot) -40 0]);
legend('Infinite','Finite','location','best')
xlabel('Elevation (deg)')
ylabel('Directivity (dB)')
title('E-plane (az=0 deg) Normalized Element Directivity')
subplot(212)
plot(azang_plot,DSEP1_Hnormlz,azang_plot,DSEP2_Hnormlz,'LineWidth',2)
grid on
axis([min(azang_plot) max(azang_plot) -40 0]);
legend('Infinite','Finite','location','best')
xlabel('Azimuth (deg)')
ylabel('Directivity (dB)')
title('H-Plane (el = 0 deg) Normalized Element Directivity')
```
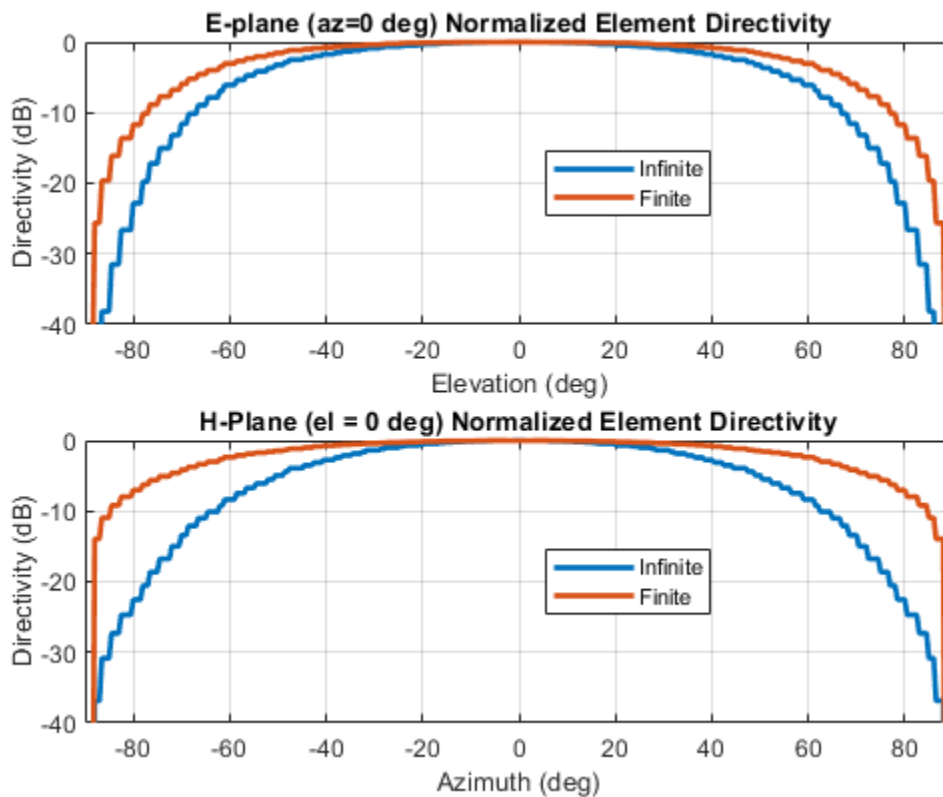
**Scan Behavior with Infinite Array Scan Element Pattern**

Scan the array based on the infinite array scan element pattern in the elevation plane defined by azimuth = 0 deg and plot the normalized directivity. Also, overlay the normalized scan element pattern.

```
helperATXScanURA(myURA1,freq,azang_plot,elang_plot,...
    DSEP1_Enormlz,DSEP1_Hnormlz);
```
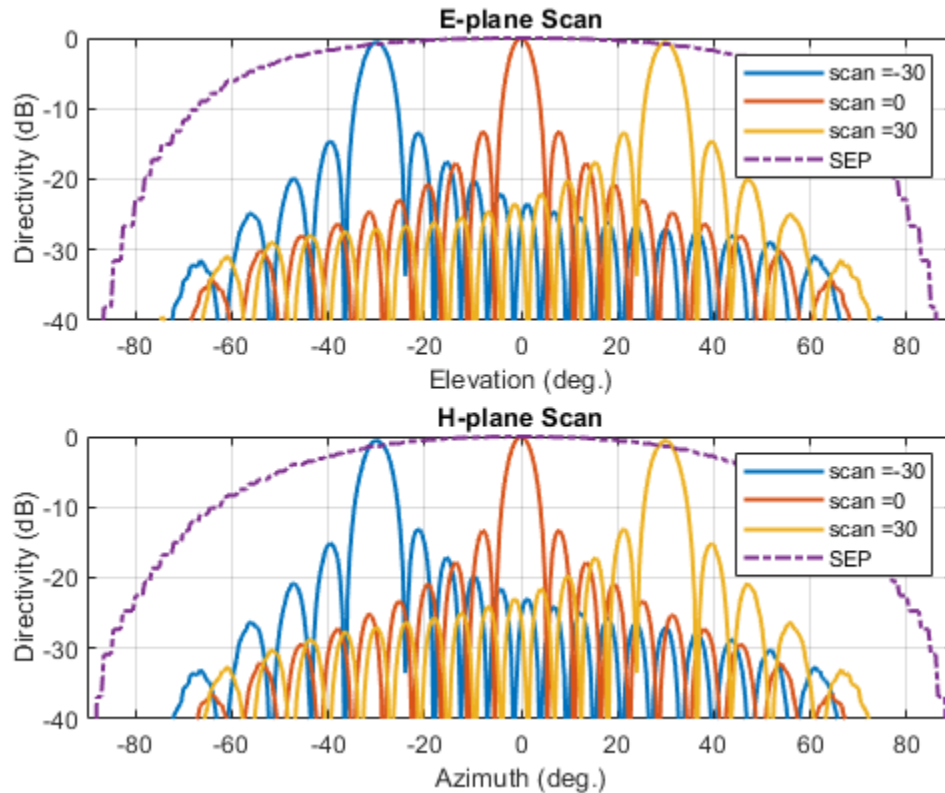
Note the overall shape of the normalized array pattern approximately follows the normalized scan element pattern. This is also predicted by the pattern multiplication principle.

**Conclusion**

The infinite array analysis is one of the tools deployed to analyze and design large finite arrays. The analysis assumes that all elements are identical, that edge effects can be ignored and have uniform excitation amplitude The isolated element pattern is replaced with the scan element pattern which includes the effect of mutual coupling.

**Reference**

[1] J. Allen, "Gain and impedance variation in scanned dipole arrays," IRE Transactions on Antennas and Propagation, vol.10, no.5, pp.566-572, September 1962.

[2] R. C. Hansen, Phased Array Antennas, Chapter 7 and 8, John Wiley & Sons Inc.,2nd Edition, 1998.

# Modeling Perturbations and Element Failures in a Sensor Array

This example shows how to model amplitude, phase, position and pattern perturbations as well as element failures in a sensor array.

**Amplitude Perturbation**

This section shows how to add gain or amplitude perturbations on a uniform linear array (ULA) of 10 elements. Consider the perturbations to be statistically independent zero-mean Gaussian random variables with standard deviation of 0.1.

Create a ULA antenna of 10 elements.

```
N = 10;
ula = phased.ULA(N);
```

Create amplitude or gain perturbations on the array

```
rs = rng(7);
pertStDev = 0.1;

perturbations(ula,'TaperMagnitude','Normal',1,pertStDev);
perturbedULA = perturbedArray(ula);
```

Compare the response of the perturbed to the ideal array.

```
% Overlay responses
c = 3e8; freq = c;

subplot(2,1,1);
helperCompareResponses(perturbedULA,ula,'Amplitude Perturbation', ...
                       {'Perturbed','Ideal'});

% Show the corresponding tapers
subplot(2,1,2);
stem(perturbedULA.Taper)
title('Amplitude Tapers');xlabel('Sensor');ylabel('Gain');
```
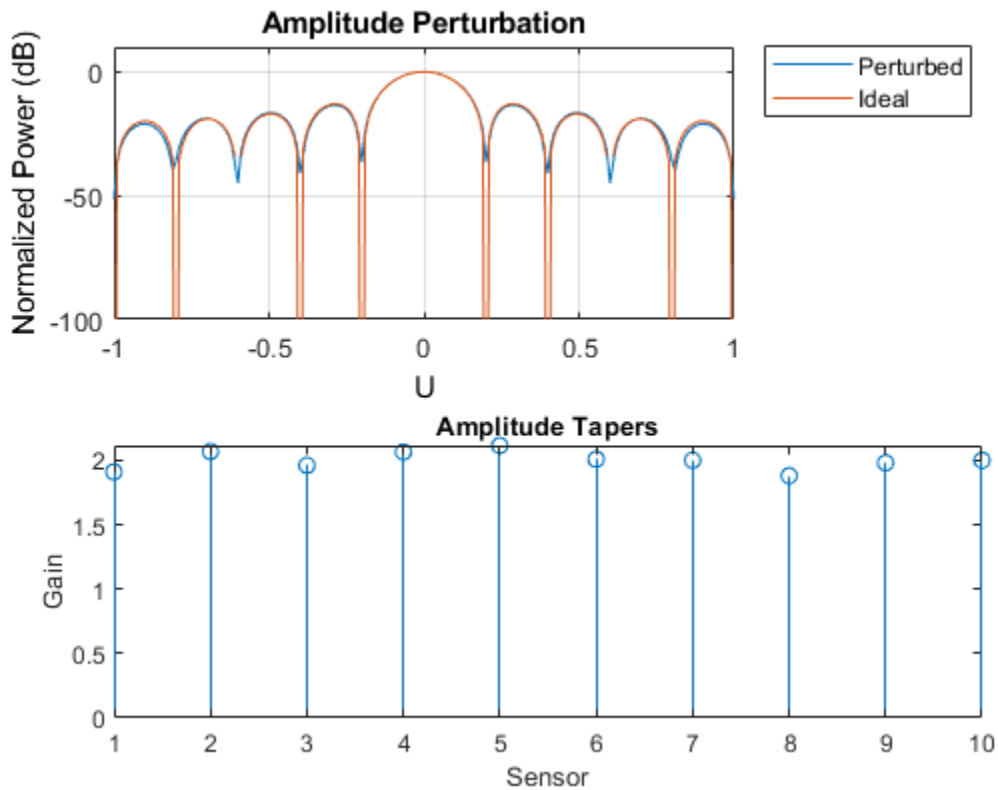
**Phase Perturbation**

This section shows how to add phase perturbations to the ULA antenna used in the previous section. Consider the perturbations distribution to be similar to the previous section.

```
perturbations(ula,'TaperMagnitude','none');
perturbations(ula,'TaperPhase','Normal',0,pertStDev);
perturbedULA = perturbedArray(ula);
```

Compare the response of the perturbed to the ideal array.

```
% Overlay responses
subplot(2,1,1);
helperCompareResponses(perturbedULA,ula,'Phase Perturbation', ...
                        {'Perturbed','Ideal'});

% Show the corresponding tapers
subplot(2,1,2);
stem(angle(perturbedULA.Taper))
title('Phase Tapers');xlabel('Sensor');ylabel('Phase (rad)');
```
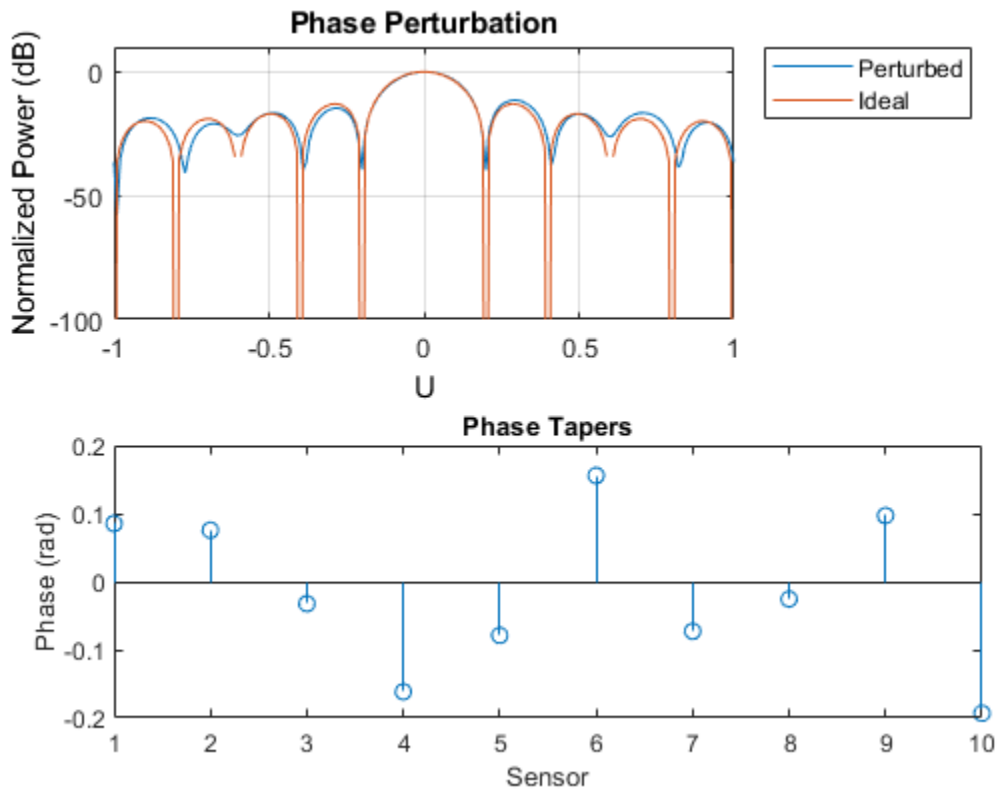
Notice how the perturbed response has shallower nulls.

**Position Perturbation**

This section shows how to perturb the positions of the ULA sensor along the three axes.

```
perturbations(ula,'TaperPhase','none');
perturbations(ula,'ElementPosition','Normal',0,pertStDev);
perturbedULA = perturbedArray(ula);
```

Compare the response of the perturbed to the ideal array.

```
% Overlay responses
clf;
helperCompareResponses(perturbedULA,phased.ULA(N), ...
    'Position Perturbation', {'Perturbed','Ideal'});
```
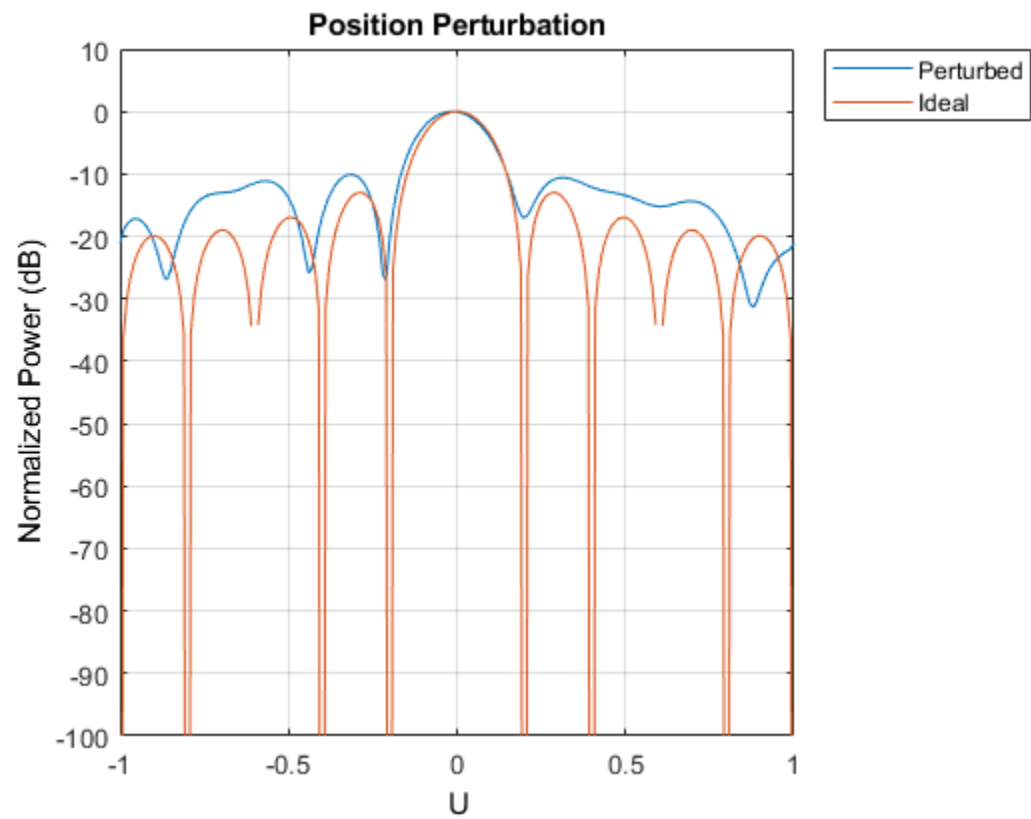
View the array.

```
viewArray(perturbedULA);
title('Element Positions');
```

Array Geometry
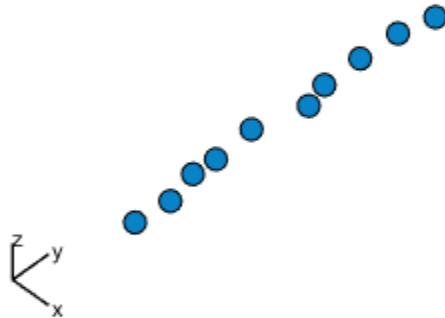
Array Span:
X axis = 246.411 mm
Y axis = 4211.028 mm
Z axis = 163.618 mm

**Pattern Perturbation**

This section will replace the isotropic antenna elements with perturbed patterns.

First create 10 custom antenna elements with perturbed isotropic patterns.

```
antenna = phased.CustomAntennaElement;
radpat = antenna.MagnitudePattern;
element = cell(1,N);
for i = 1:N
    perturbedAntenna = clone(antenna);
    perturbedAntenna.MagnitudePattern = ...
        pow2db(1+randn(size(radpat))*pertStDev);
    element{i} = perturbedAntenna;
end
```

Here, map the 10 patterns in the cell array 'element' to the 10 sensors using the ElementIndices property.

```
perturbedULA = phased.HeterogeneousULA('ElementSet',element, ...
                                    'ElementIndices',1:N);
```

Compare the response of the perturbed to the ideal array.

```
% Overlay responses
clf;
subplot(2,2,[1 2]);
helperCompareResponses(perturbedULA,phased.ULA(N),...
```

```
    'Pattern Perturbation', ...
    {'Perturbed','Ideal'});

% Show the perturbed pattern response next to the ideal isotropic pattern
subplot(2,2,3);
pattern(ula.Element,freq,'CoordinateSystem','polar','Type','power')
title('Isotropic pattern');
subplot(2,2,4);
pattern(element{1},freq,'CoordinateSystem','polar','Type','power')
title('Perturbed pattern');
```
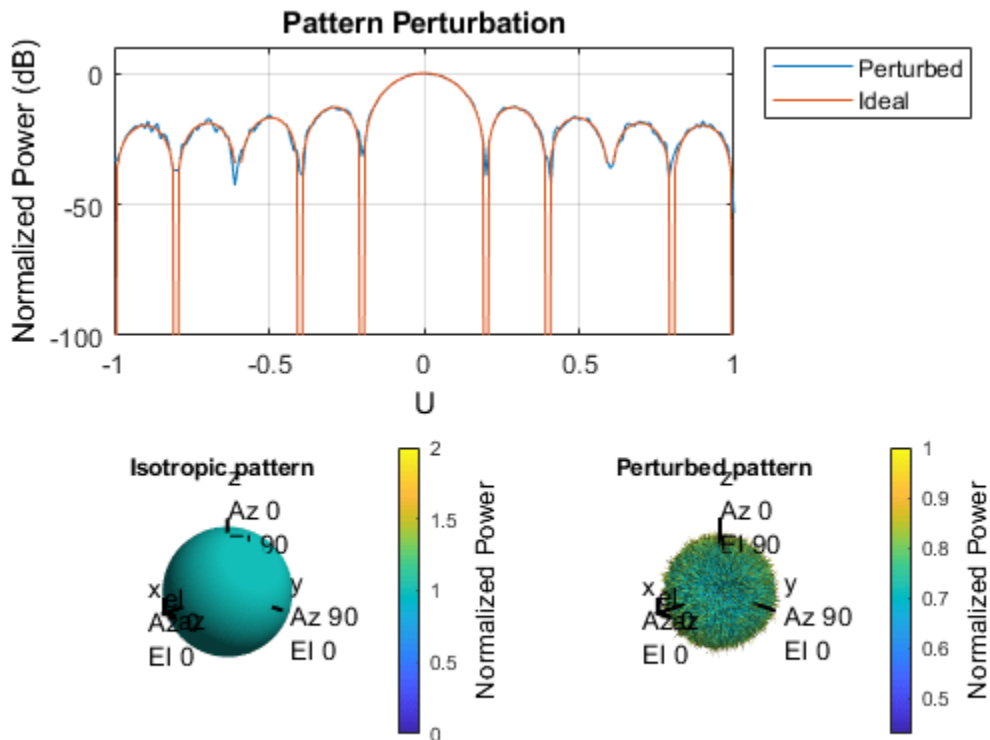


### Element Failures

This section will model element failures on an 8 by 10 uniformly rectangular array. Each element has 10 percent probability of failing.

Create a URA antenna of 8 by 10 elements.

```
ura = phased.URA([8 10]);
```

Failures can be modeled by setting the gain on the corresponding sensor to 0. Here a matrix is created in which each element has a 10 percent probability of being 0.

```
perturbations(ura,'ElementFailure','RandomFail',0.1);
perturbedURA = perturbedArray(ura);

% ura.Taper  = double(rand(8,10) > .1);
```

Compare the response of the array with failed elements to the ideal array.

```
% Overlay responses
clf;
helperCompareResponses(perturbedURA,ura,'Element Failures', ...
                        {'Failures','No Failures'});
```
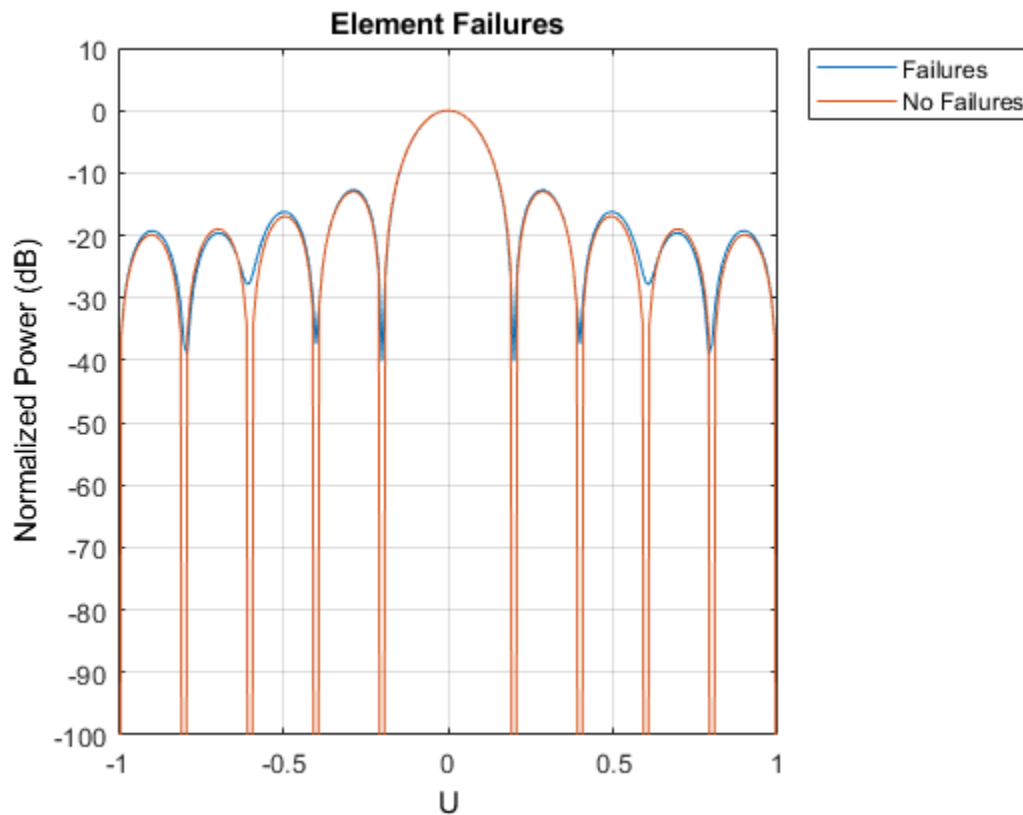


Notice how deep nulls are hard to attain in the response of the array with failed elements.

View the failed elements.

```
viewArray(perturbedURA,'ShowTaper',true);
title('Failed Elements');

% reset the random seed
rng(rs)
```

**Array Geometry**

**Failed Elements**
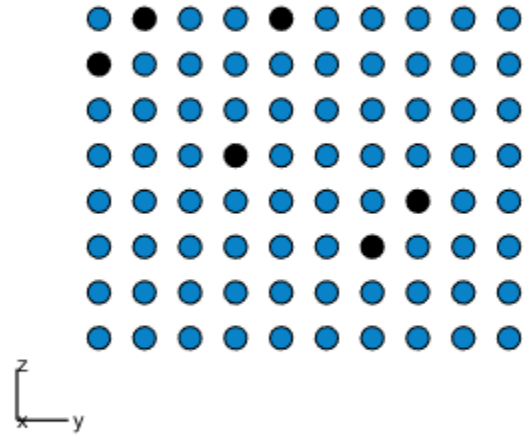
Array Span:
X axis = 0.0 m
Y axis = 4.5 m
Z axis = 3.5 m

**Summary**

This example showed how to model different kind of perturbations as well as element failures. It also demonstrated the effect on the array response for all the cases.

# Patch Antenna Array for FMCW Radar

This example shows how to model a 77 GHz antenna array for frequency-modulated continuous-wave (FMCW) radar applications. The presence of antennas and antenna arrays in and around vehicles has become commonplace with the introduction of wireless collision detection, collision avoidance, and lane departure warning systems. The two frequency bands considered for such systems are centered around 24 GHz and 77 GHz, respectively. In this example, we will investigate the microstrip patch antenna as a phased array radiator. The dielectric substrate is air.

This example requires the Antenna Toolbox™.

**Antenna Array Design**

The FMCW antenna array is intended for a forward radar system designed to look for and prevent a collision. Therefore, a cosine antenna pattern is an appropriate choice for the initial design since it does not radiate any energy backwards. Assume that the radar system operates at 77 GHz with a 700 MHz bandwidth.

```
fc = 77e9;
fmin = 73e9;
fmax = 80e9;
vp = physconst('lightspeed');
lambda = vp/fc;

cosineantenna = phased.CosineAntennaElement;
cosineantenna.FrequencyRange = [fmin fmax];
pattern(cosineantenna,fc)
```
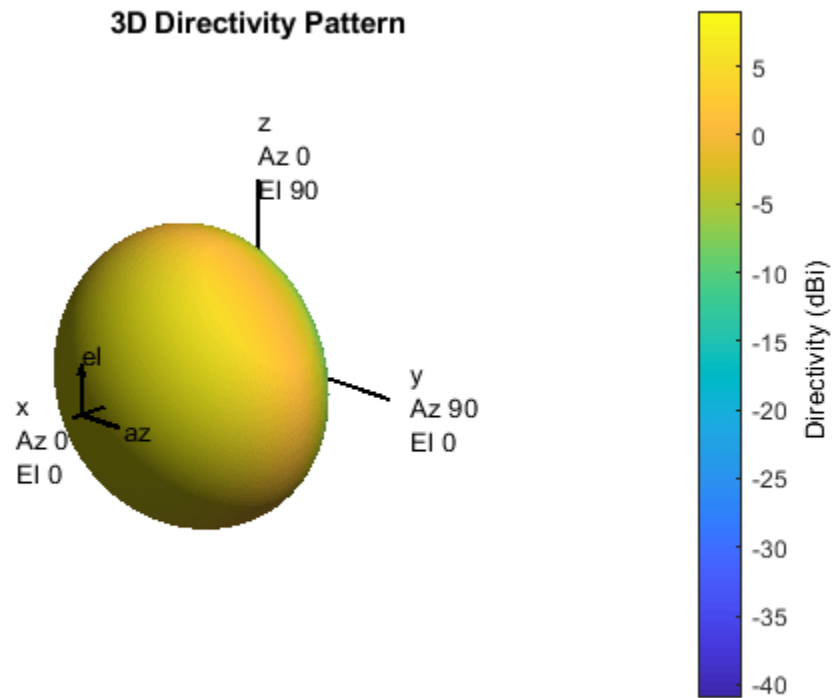
The array itself needs to be mounted on or around the front bumper. The array configuration we investigate is a 2 X 4 rectangular array, similar to what is mentioned in [1]. Such a design has bigger aperture along azimuth direction thus providing better azimuth resolution.

```
Nrow = 2;
Ncol = 4;
fmcwCosineArray = phased.URA;
fmcwCosineArray.Element = cosineantenna;
fmcwCosineArray.Size = [Nrow Ncol];
fmcwCosineArray.ElementSpacing = [0.5*lambda 0.5*lambda];
pattern(fmcwCosineArray,fc)
```

**3D Directivity Pattern**
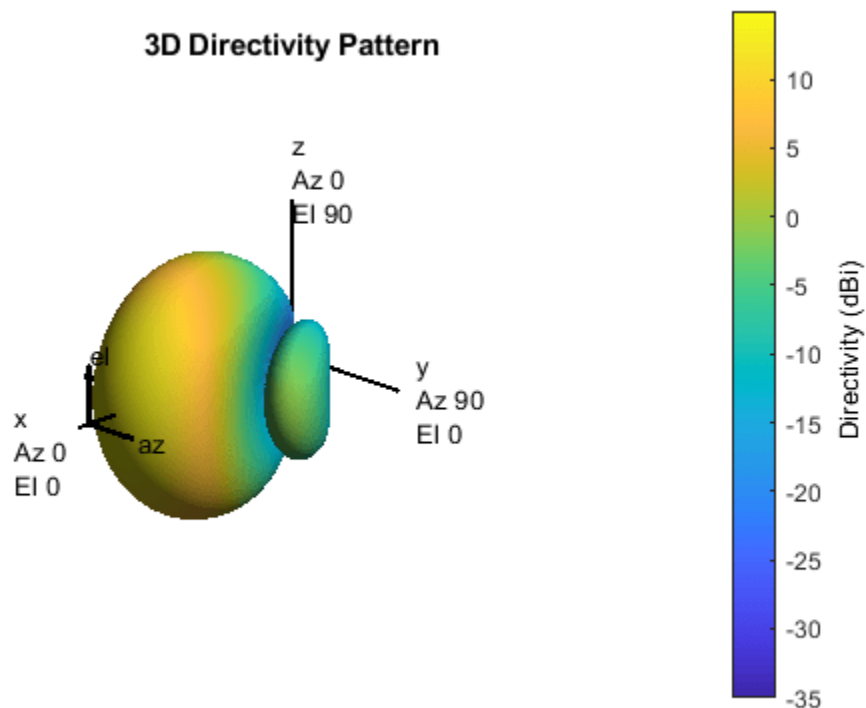
### Design Realistic Patch Antenna

The Antenna Toolbox has several antenna elements that could provide hemispherical coverage and resemble a pattern of cosine shape. Choose a patch antenna element with typical radiator dimensions. The patch length is approximately half-wavelength at 77 GHz and the width is 1.5 times the length to improving the bandwidth. The ground plane is $\lambda$ on each side and the feed offset from center in the direction of the patch length is about a quarter of the length.

```
patchElement = design(patchMicrostrip,fc);
```

Because the default patch antenna geometry has its maximum radiation directed towards zenith, rotate the patch antenna by 90 degrees about the y-axis so that the maximum is along the x-axis.

```
patchElement.Tilt = 90;
patchElement.TiltAxis = [0 1 0];
```

### Isolated Patch Antenna 3D Pattern and Resonance

Plot the pattern of the patch antenna at 77 GHz. The patch is a medium gain antenna with the peak directivity around 10 dBi.

```
myFigure = gcf;
myFigure.Color = 'w';
pattern(patchElement,fc)
```
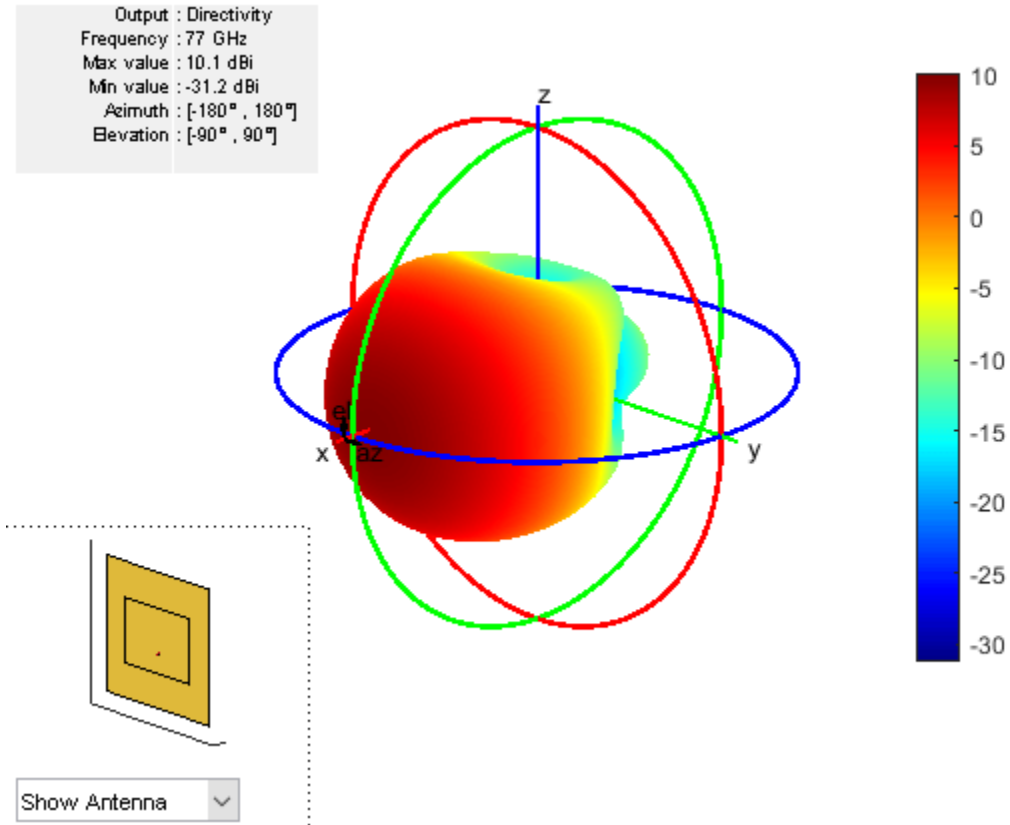
The patch is radiating in the correct mode with a pattern maximum at 0 degrees azimuth and 0 degrees elevation. Since the initial dimensions are approximations, it is important to verify the input impedance behavior.

```
Numfreqs = 21;
freqsweep = unique([linspace(fmin,fmax,Numfreqs) fc]);
impedance(patchElement,freqsweep);
```
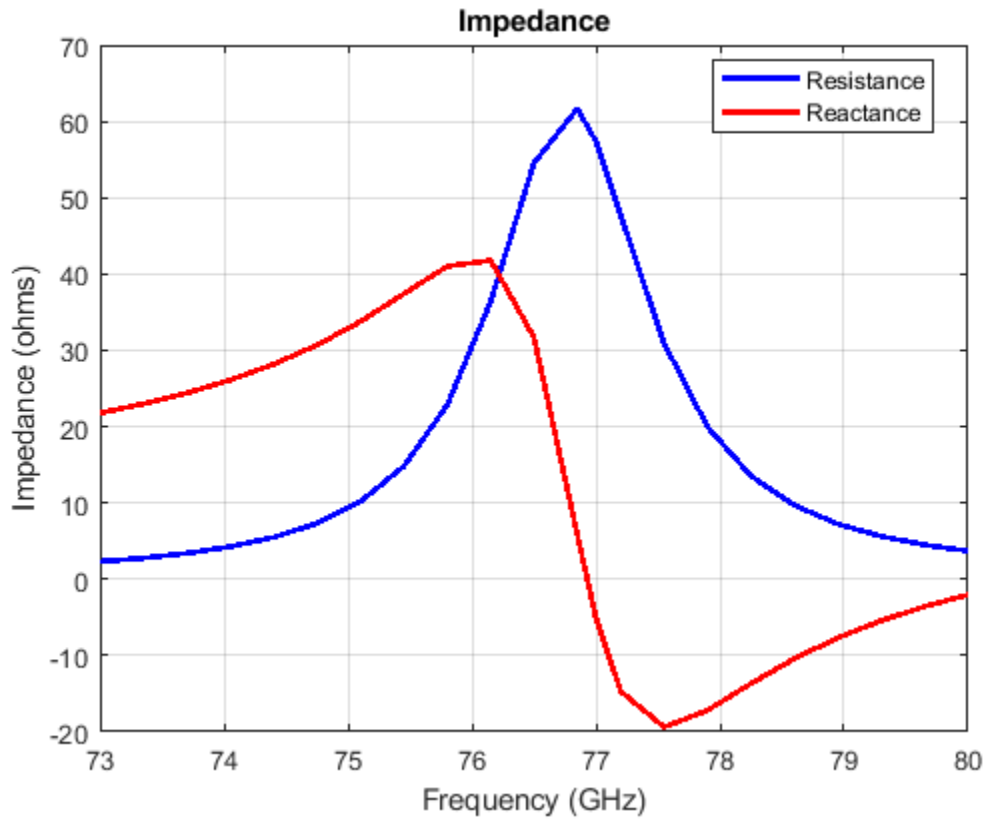
According to the figure, the patch antenna has its first resonance (parallel resonance) at 77 GHz.

Next is to check the reflection coefficient of the patch antenna to confirm a good impedance match. It is typical to consider the value $S_{11} = -10dB$ as a threshold value for determining the antenna bandwidth.

```
s = sparameters(patchElement,freqsweep);
rfplot(s,'m-.')
hold on
line(freqsweep/1e9,ones(1,numel(freqsweep))*-10,'LineWidth',1.5)
hold off
```
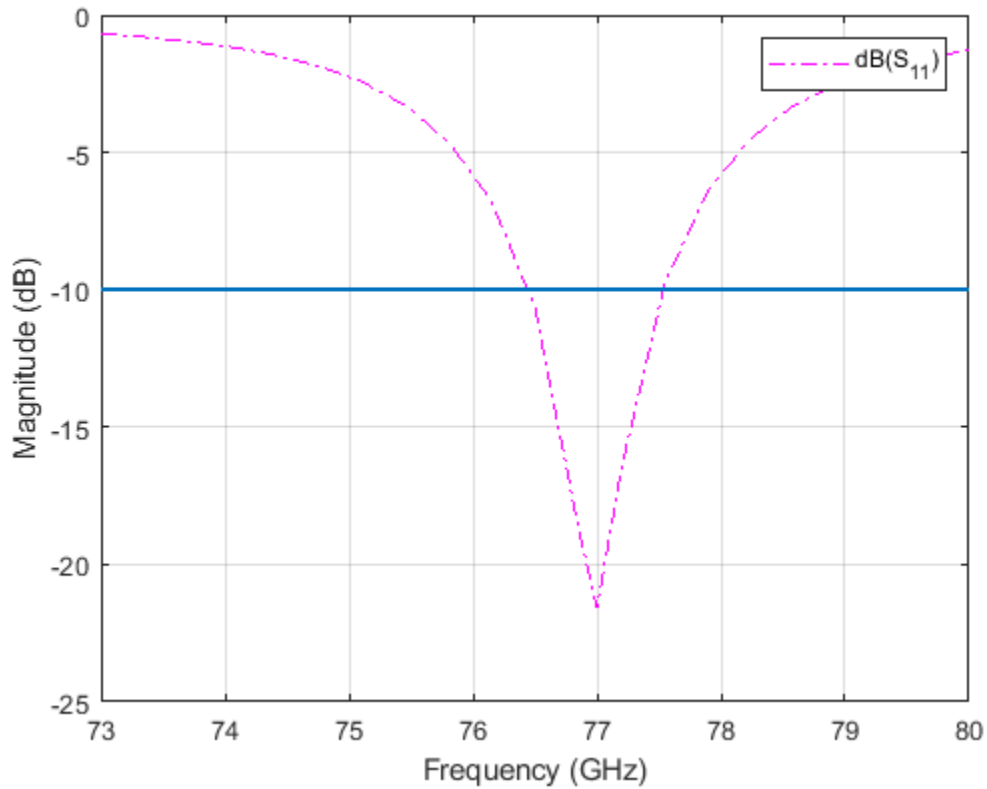
The deep minimum at 77 GHz indicates a good match to 50. The antenna bandwidth is slightly greater than 1 GHz. Thus, the frequency band is from 76.5 GHz to 77.5 GHz.

Finally, check if the pattern at the edge frequencies of the band meets the design. This is a good indication whether the pattern behaves the same across the band. The patterns at 76.5 GHz and 77.6 GHz are shown below.

```
pattern(patchElement,76.5e9)
```
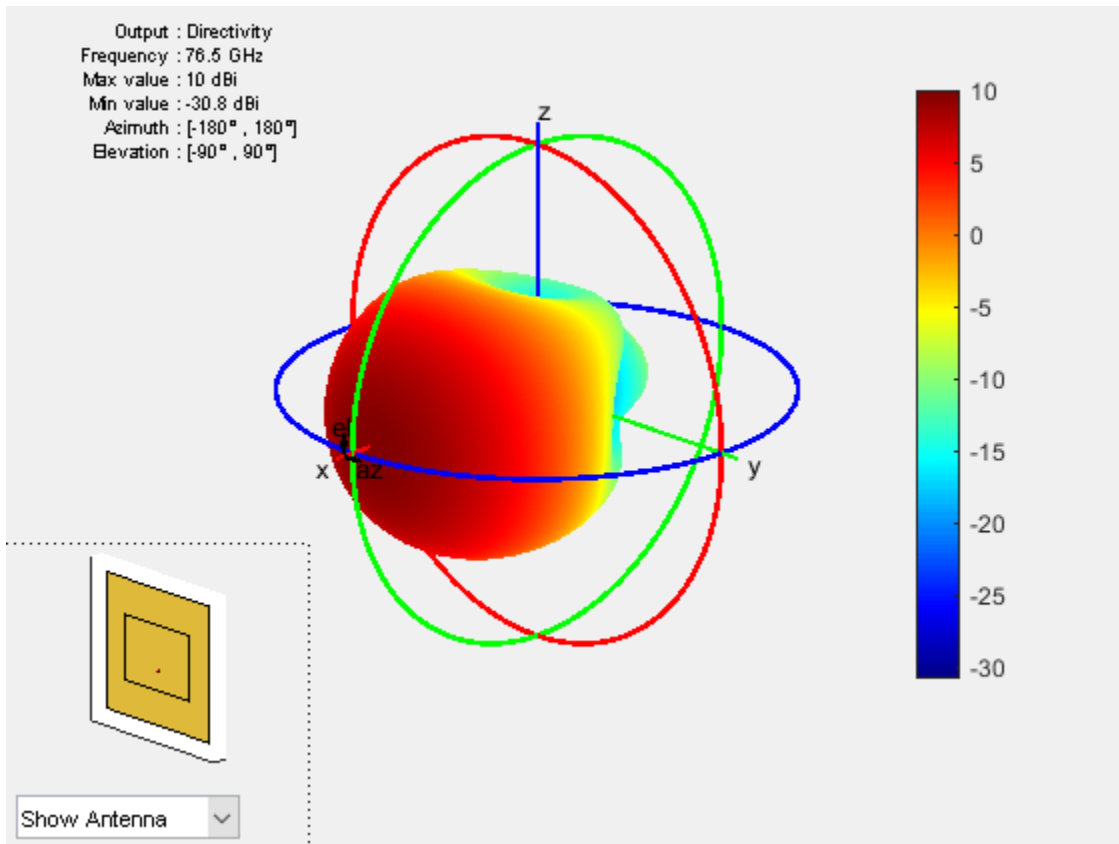
```
pattern(patchElement,77.6e9)
```
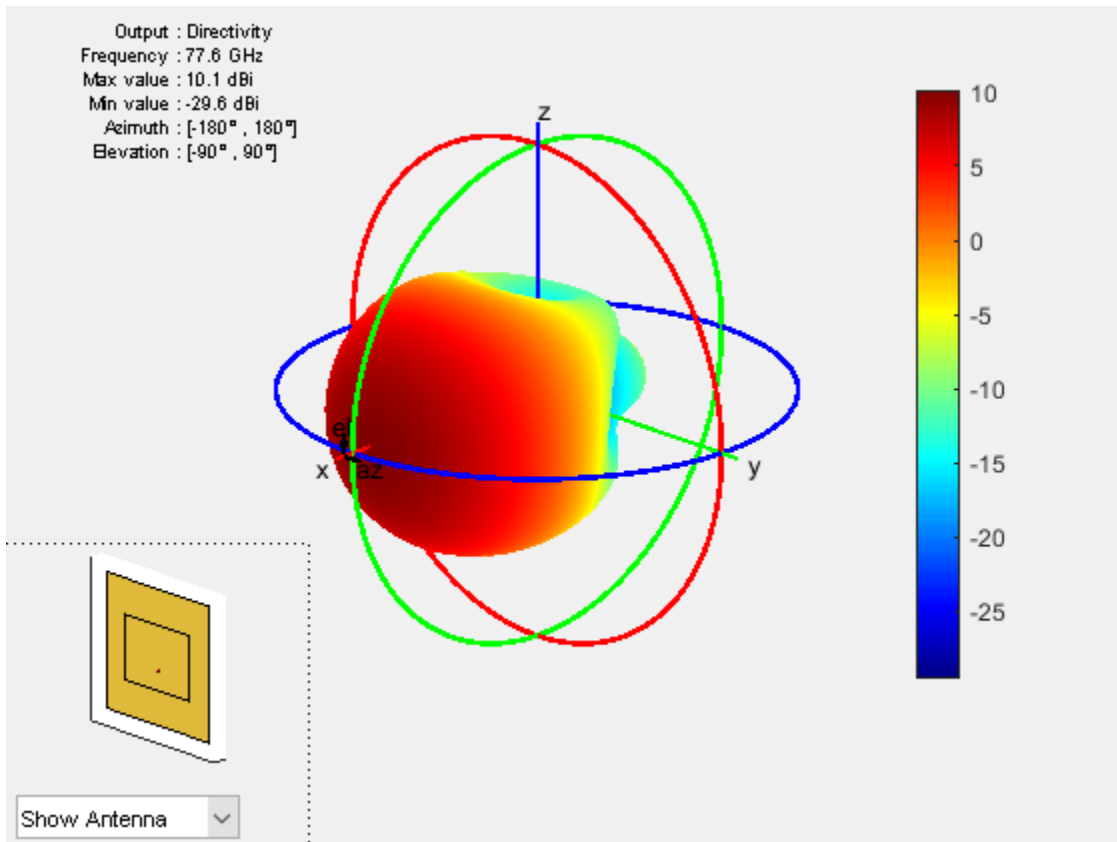
**Create Array from Isolated Radiators and Plot Pattern**

Next, create a uniform rectangular array (URA) with the patch antenna. The spacing is chosen to be $\lambda/2$, where $\lambda$ is the wavelength at the upper frequency of the band (77.6 GHz).

```
fc2 = 77.6e9;
lambda_fc2 = vp/77.6e9;
fmcwPatchArray = phased.URA;
fmcwPatchArray.Element = patchElement;
fmcwPatchArray.Size = [Nrow Ncol];
fmcwPatchArray.ElementSpacing = [0.5*lambda_fc2 0.5*lambda_fc2];
```

The following figure shows the pattern of the resulting patch antenna array. The pattern is computed using a 5 degree separation in both azimuth and elevation.

```
az = -180:5:180;
el = -90:5:90;
clf
pattern(fmcwPatchArray,fc,az,el)
```

**3D Directivity Pattern**
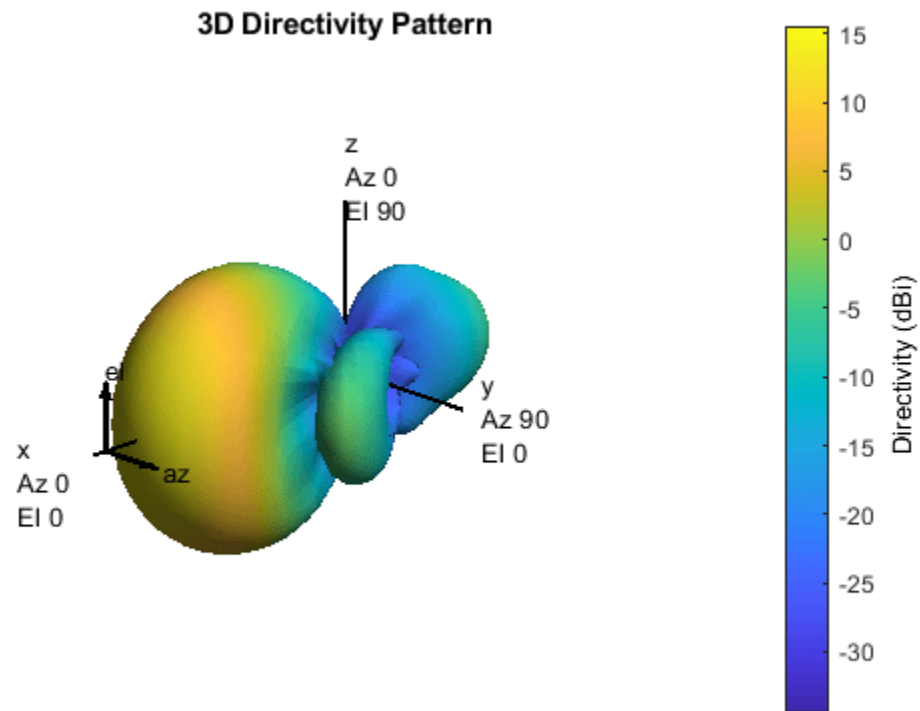


Plots below compare the pattern variation in orthogonal planes for the patch antenna array and the cosine element array. Note that both arrays ignore mutual coupling effect.

First, plot the patterns along the azimuth direction.

```
patternAzimuth(fmcwPatchArray,fc)
hold on
patternAzimuth(fmcwCosineArray,fc)
p = polarpattern('gco');
p.LegendLabels = {'Patch','Cosine'};
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
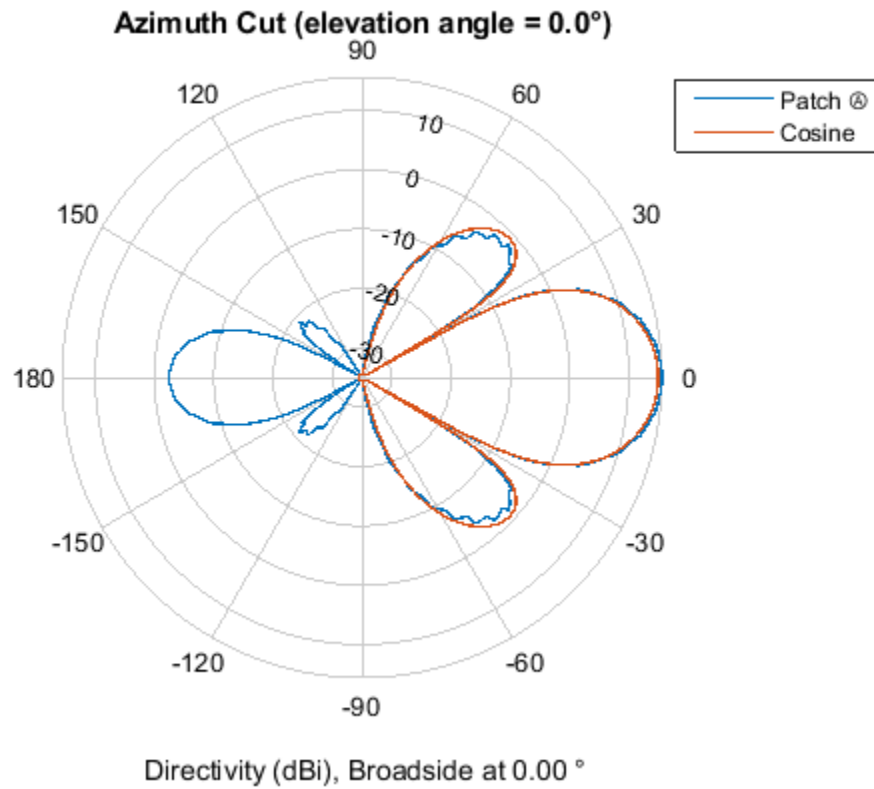
Then, plot the patterns along the elevation direction.

```
clf
patternElevation(fmcwPatchArray,fc)
hold on
patternElevation(fmcwCosineArray,fc)
p = polarpattern('gco');
p.LegendLabels = {'Patch','Cosine'};
```

Directivity (dBi), Broadside at 0.00 °
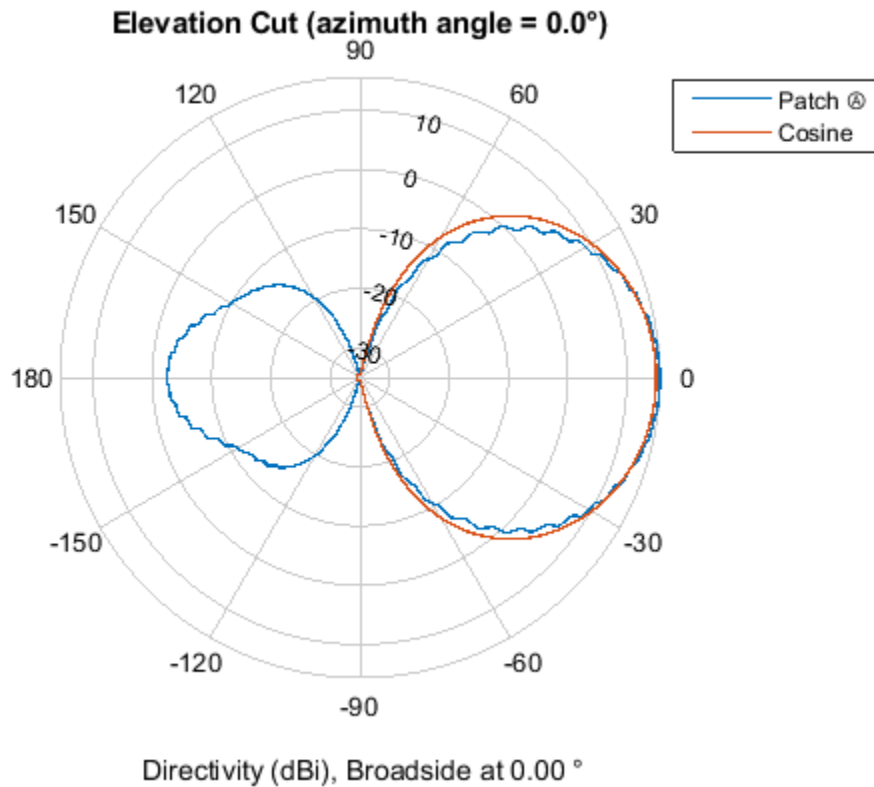
The figures show that both arrays have similar pattern behavior around the main beam in the elevation plane (azimuth = 0 deg). The patch-element array has a significant backlobe as compared to the cosine-element array.

**Conclusions**

This example starts the design of an antenna array for FMCW radar with an ideal cosine antenna and then uses a patch antenna to form the real array. The example compares the patterns from the two arrays to show the design tradeoff. From the comparison, it can be seen that using the isolated patch element is a useful first step in understanding the effect that a realistic antenna element will have on the array pattern.

However, analysis of realistic arrays must also consider mutual coupling effect. Since this is a small array, the individual element patterns in the array environment could be distorted significantly. As a result it is not possible to replace the isolated element pattern with an embedded element pattern, as shown in the "Modeling Mutual Coupling in Large Arrays Using Embedded Element Pattern" on page 18-52 example. A full-wave analysis must be performed to understand the effect of mutual coupling on the overall array performance.

**Reference**

[1] R. Kulke, et al. 24 GHz Radar Sensor Integrates Patch Antennas, EMPC 2005 `http://empire.de/main/Empire/pdf/publications/2005/26-doc-empc2005.pdf`

# Phased Array Gallery

This example shows how to model and visualize a variety of antenna array geometries with Phased Array System Toolbox™. These geometries can also be used to model other kinds of arrays, such hydrophone arrays and microphone arrays. You can view code for each plot, and use it in your own project.

**Linear Arrays**

Linear antenna arrays can have uniform or nonuniform spacing between elements. This most common linear antenna array is the Uniform Linear Array (ULA).

```
N = 20;   % Number of elements
D = 0.5; % Element spacing (m)
ula = phased.ULA(N,D);
viewArray(ula,'Title','Uniform Linear Array (ULA)')
set(gca,'CameraViewAngle',4.4);
```
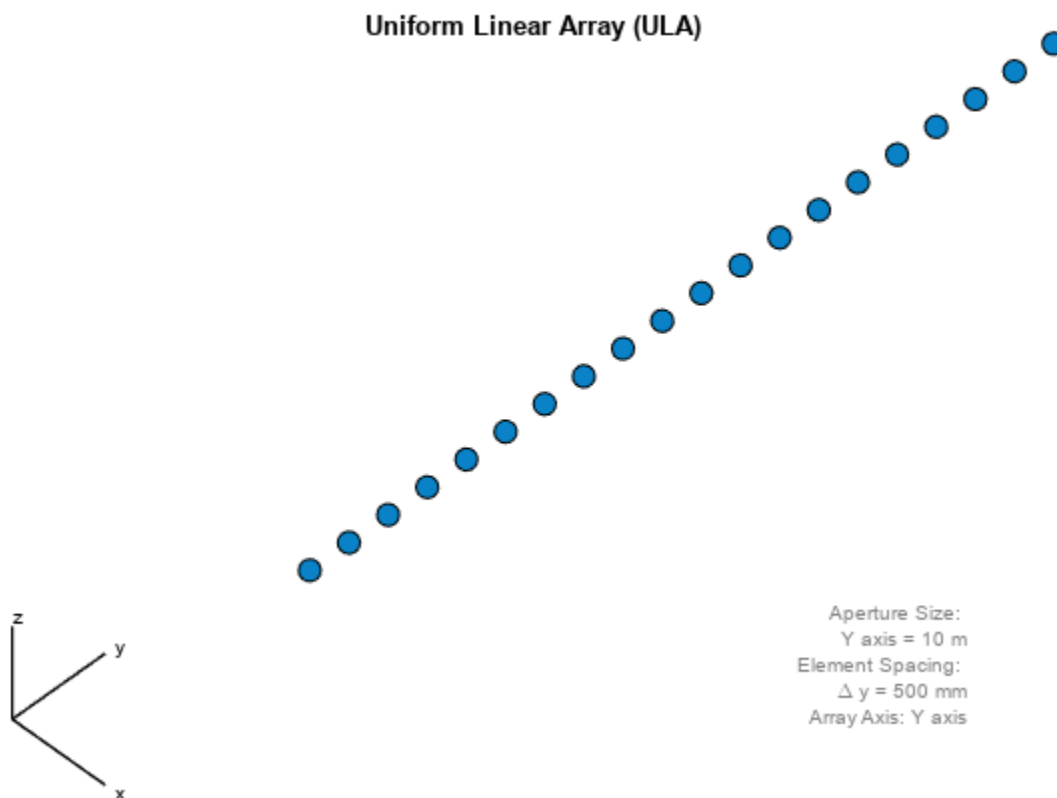


A Minimum Redundancy Linear Array (MRLA) is an example of a nonuniformly spaced linear array. The MRLA minimizes the number of element pairs that have the same spatial correlation lag. It is possible to design a 4-element array whose aperture is equivalent to 7-element ULA.

```
N = 4;                          % Number of elements
pos = zeros(3,N);
pos(2,:) = [-1.5 -1 0.5 1.5];  % Aperture equivalent to 7-element ULA
```

```
mrla = phased.ConformalArray('ElementPosition',pos,...
                             'ElementNormal',zeros(2,N));

viewArray(mrla,'Title','Minimum Redundancy Linear Array (MRLA)')
set(gca,'CameraViewAngle',4.85);
```



**Minimum Redundancy Linear Array (MRLA)**

Array Span:
X axis = 0 m
Y axis = 3 m
Z axis = 0 m
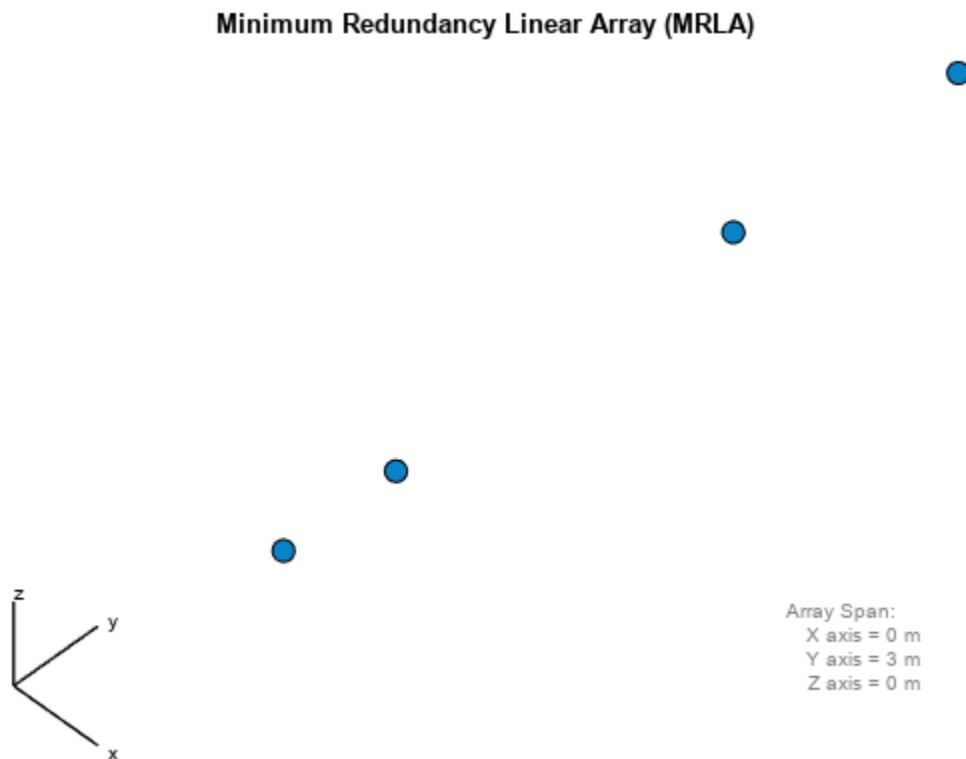
## Circular Arrays

Circular antenna arrays can also have uniform or nonuniform spacing between elements. Next is an example of a Uniform Circular Array (UCA).

```
N = 24;   % Number of elements
R = 1;    % Radius (m)
uca = phased.UCA(N,R);

viewArray(uca,'ShowNormals',true,'Title','Uniform Circular Array (UCA)')
view(0,90)
```

## Uniform Circular Array (UCA)



Radius = 1 m
Element Spacing:
Arc = 261.8 mm
Array Plane: XY plane
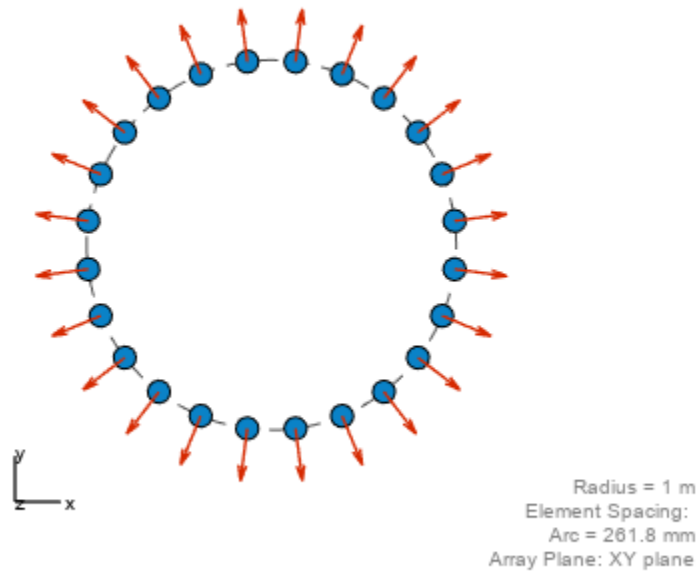
Multiple circular antenna arrays with the same number of elements and different radii form a concentric circular array.

```
N = 16;                     % Number of elements on each ring
R = [1 1.5 2];              % Radii (m)
azang = (0:N-1)*360/N-180;
pos = [zeros(1,N);cosd(azang);sind(azang)];
elNormal = zeros(2,N);
concentricCircularArray = phased.ConformalArray(...
    'ElementPosition',[R(1)*pos R(2)*pos R(3)*pos],...
    'ElementNormal',[elNormal elNormal elNormal]);

viewArray(concentricCircularArray,'Title','Concentric Circular Array');
```

**Concentric Circular Array**



Array Span:
X axis = 0 m
Y axis = 4 m
Z axis = 4 m
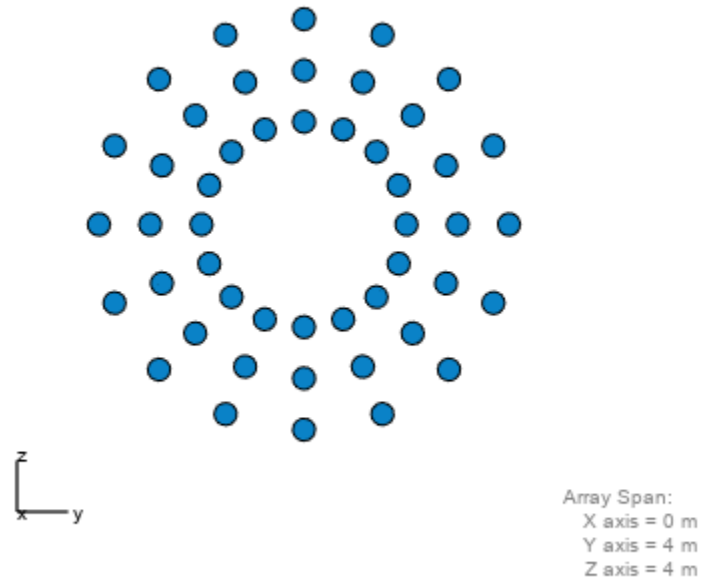
**Planar Arrays with Rectangular Grid**

Planar antenna arrays can have a uniform grid (or lattice) and different boundary shapes. Next is an example of a Uniform Rectangular Array (URA) with a rectangular grid and rectangular boundary.

```
M = 18;   % Number of elements on each row
N = 16;   % Number of elements on each column
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
ura = phased.URA([N M],[dz dy]);

viewArray(ura,'Title','Uniform Rectangular Array (URA)');
```

**Uniform Rectangular Array (URA)**



Aperture Size:
Y axis = 9 m
Z axis = 8 m
Element Spacing:
$\Delta$ y = 500 mm
$\Delta$ z = 500 mm
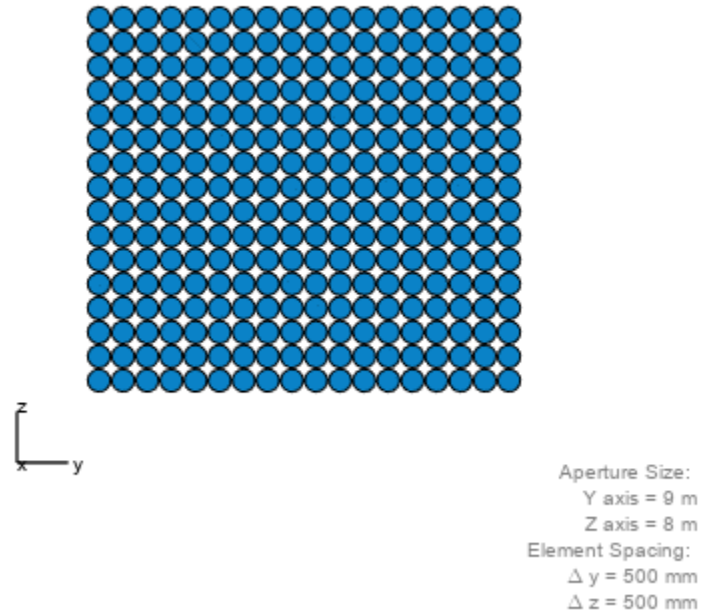
You can also model a planar antenna array with a circular boundary. The following code starts with a URA and removes elements outside a circle.

```
N = 20;   % Number of elements on each row/column of rectangular array
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
R = 4.5;  % Radius (m)
refArray = phased.URA(N,[dy,dz]);
pos = getElementPosition(refArray);
elemToRemove = sum(pos.^2)>R^2;
pos(:,elemToRemove) = []; % Exclude elements outside circle
circularPlanarArray = phased.ConformalArray('ElementPosition',pos,...
                          'ElementNormal',zeros(2,size(pos,2)));

viewArray(circularPlanarArray,'Title','Circular Planar Array');
```

Circular Planar Array

Array Span:
X axis = 0.0 m
Y axis = 8.5 m
Z axis = 8.5 m
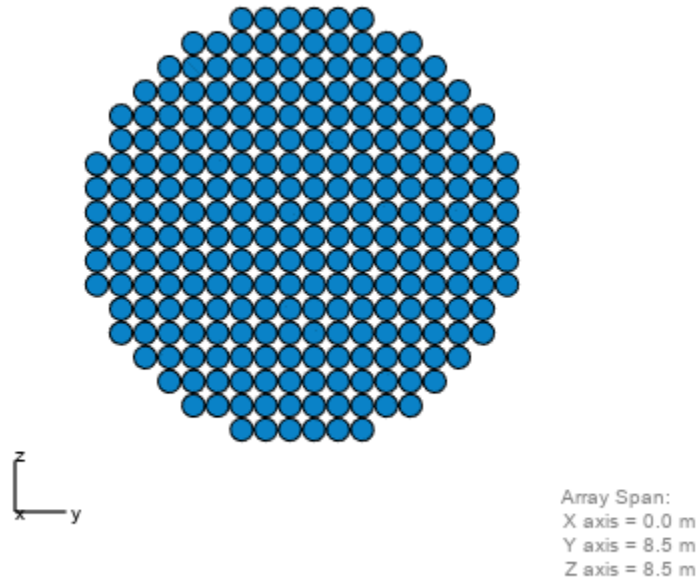
Next is an example of a planar antenna array with an elliptical boundary.

```
N = 20;   % Number of elements on each row/column of rectangular array
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
Ry = 4.5; % Major radius (m)
Rz = 2.8; % Minor radius (m)
refArray = phased.URA(N,[dy,dz]);
pos = getElementPosition(refArray);
elemToRemove = (pos(2,:)/Ry).^2+(pos(3,:)/Rz).^2>1;
pos(:,elemToRemove) = [];  % Exclude elements outside ellipse
ellipticalPlanarArray = phased.ConformalArray('ElementPosition',pos,...
                           'ElementNormal',zeros(2,size(pos,2)));

viewArray(ellipticalPlanarArray,'Title','Elliptical Planar Array');
```

**Elliptical Planar Array**
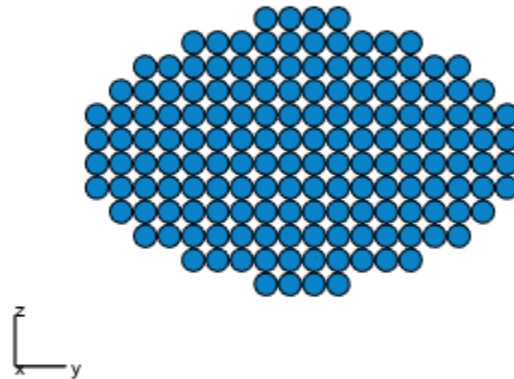


The next example is a hexagonal array with a rectangular grid.

```matlab
Nmin = 7;          % Number of elements on bottom row
Nmax = 19;         % Number of elements on widest row
dy = 0.5;          % Row spacing
dz = 0.5;          % Column spacing
rows = [Nmin:2:Nmax Nmax-2:-2:Nmin];
N = sum(rows); % Total number of elements
stop = cumsum(rows);
start = stop-rows+1;
pos = zeros(3,N);
count = 0;
for m = Nmin-Nmax:2:Nmax-Nmin
    count = count+1;
    idx = start(count):stop(count);
    pos(2,idx) = (-(rows(count)-1)/2:(rows(count)-1)/2)*dy;
    pos(3,idx) = m/2*dz;
end
hexagonalPlanarArray = phased.ConformalArray('ElementPosition',pos,...
                                'ElementNormal',zeros(2,N));

viewArray(hexagonalPlanarArray,...
    'Title','Hexagonal Planar Array with Rectangular Grid');
```

Hexagonal Planar Array with Rectangular Grid
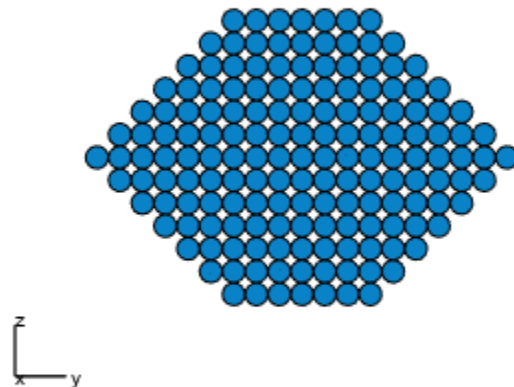


Array Span:
X axis = 0 m
Y axis = 9 m
Z axis = 6 m

**Planar Arrays with Triangular Grid**

Triangular grids provide an efficient spatial sampling and are widely used in practice. Here again, different boundary geometries can be applied. First is a rectangular array with a triangular lattice.

```
M = 18;   % Number of elements on each row
N = 16;   % Number of elements on each column
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
rectArrayTriGrid = phased.URA([N M],[dz dy],'Lattice','Triangular');

viewArray(rectArrayTriGrid,...
    'Title','Rectangular Array with Triangular Grid');
```

**Rectangular Array with Triangular Grid**
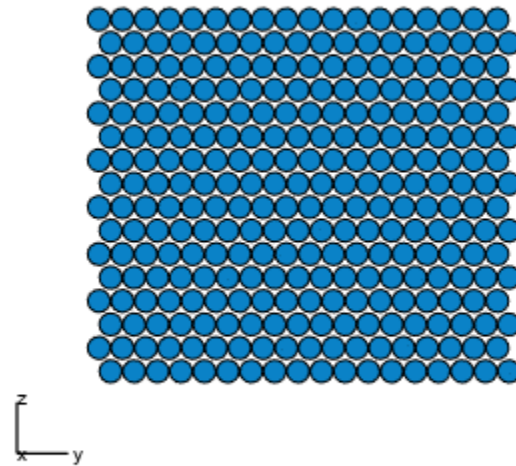


```
Aperture Size:
    Y axis = 9 m
    Z axis = 8 m
Element Spacing:
    Δ y = 500 mm
    Δ z = 500 mm
```

Next is a circular planar antenna array with a triangular lattice.

```
N = 18;   % Number of elements on each row/column of rectangular array
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
R = 4.5;  % Radius (m)
refArray = phased.URA(N,[dy,dz],'Lattice','Triangular');
pos = getElementPosition(refArray);
elemToRemove = sum(pos.^2)>R^2;
pos(:,elemToRemove) = []; % Exclude elements outside circle
circularPlanarArrayTriGrid = phased.ConformalArray(...
    'ElementPosition',pos,'ElementNormal',zeros(2,size(pos,2)));

viewArray(circularPlanarArrayTriGrid,...
    'Title','Circular Planar Array with Triangular Grid');
```

Circular Planar Array with Triangular Grid



Array Span:
X axis = 0.0 m
Y axis = 8.5 m
Z axis = 8.5 m
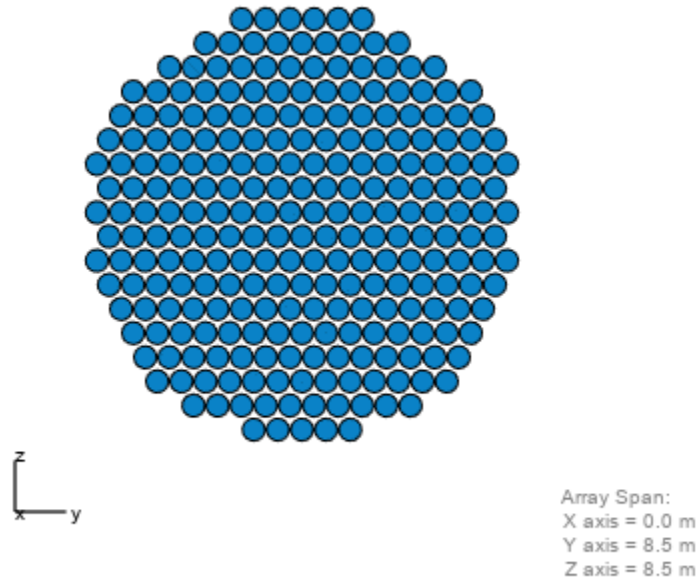
Next is an elliptical planar antenna array with a triangular lattice.

```
N = 18;    % Number of elements on each row/column of rectangular array
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
Ry = 4.5; % Major radius (m)
Rz = 2.8; % Minor radius (m)
refArray = phased.URA(N,[dy,dz],'Lattice','Triangular');
pos = getElementPosition(refArray);
elemToRemove = (pos(2,:)/Ry).^2+(pos(3,:)/Rz).^2>1;
pos(:,elemToRemove) = [];  % Exclude elements outside ellipse
ellipticalPlanarArrayTriGrid = ...
    phased.ConformalArray('ElementPosition',pos,...
    'ElementNormal',zeros(2,size(pos,2)));

viewArray(ellipticalPlanarArrayTriGrid,...
    'Title','Elliptical Planar Array with Triangular Grid');
```

**Elliptical Planar Array with Triangular Grid**
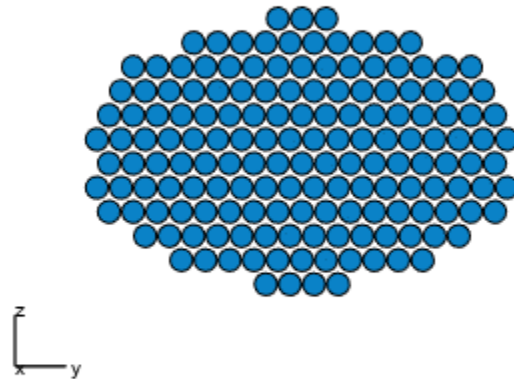


Array Span:
X axis = 0.0 m
Y axis = 8.5 m
Z axis = 5.5 m

Next is an example of a Uniform Hexagonal Array (UHA).

```
Nmin = 9;              % Number of elements on bottom row
Nmax = 17;             % Number of elements on mid row
dy = 0.5;              % Row spacing
dz = 0.5*sin(pi/3);    % Column spacing
rows = [Nmin:Nmax Nmax-1:-1:Nmin];
N = sum(rows);         % Total number of elements
stop = cumsum(rows);
start = stop-rows+1;
pos = zeros(3,N);
count = 0;
for m = Nmin-Nmax:Nmax-Nmin
    count = count+1;
    idx = start(count):stop(count);
    pos(2,idx) = (-(rows(count)-1)/2:(rows(count)-1)/2)*dy;
    pos(3,idx) = m*dz;
end
uha = phased.ConformalArray('ElementPosition',pos,...
    'ElementNormal',zeros(2,N));

viewArray(uha,'Title','Uniform Hexagonal Array (UHA)');
```

**18-103**

Uniform Hexagonal Array (UHA)



Array Span:
X axis = 0.000 m
Y axis = 8.000 m
Z axis = 6.928 m
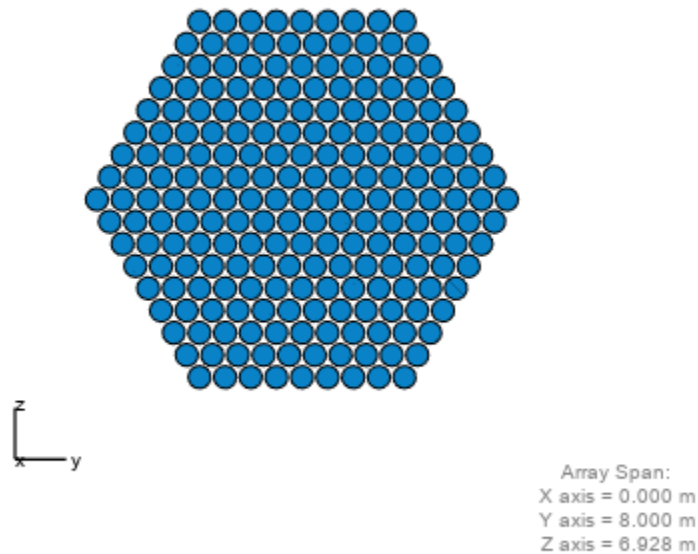
**Thinned Arrays**

You can also model planar antenna arrays with nonuniform grids. Next is an example of a thinned antenna array.

```
M = 19;   % Number of elements on each row
N = 17;   % Number of elements on each column
dy = 0.5; % Spacing between elements on each row (m)
dz = 0.5; % Spacing between elements on each column (m)
refArray = phased.URA([N M],[dz dy]);
pos = getElementPosition(refArray);
elemToRemove = [3:11:M*(N-1)/2 M*N-3:-11:(N+1)/2];
pos(:,elemToRemove) = [];
thinnedURA = phased.ConformalArray('ElementPosition',pos,...
                        'ElementNormal',zeros(2,size(pos,2)));

viewArray(thinnedURA,'Title','Thinned Array');
```

**Thinned Array**



```
Array Span:
 X axis = 0 m
 Y axis = 9 m
 Z axis = 8 m
```
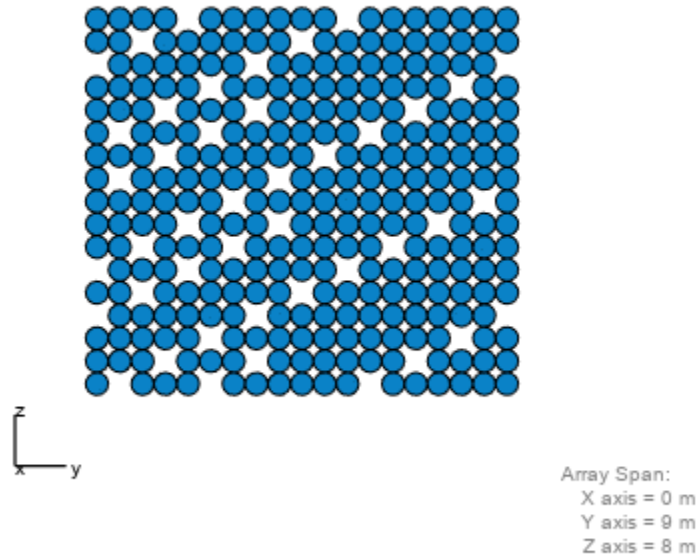
## Hemispherical Conformal Arrays

You can also model nonplanar arrays. In many applications, sensors must conform to the shape of the curved surface they are mounted on. Next is an example of an antenna array whose elements are uniformly distributed on a hemisphere.

```
R = 2;                          % Radius (m)
az = -90:10:90;                 % Azimuth angles
el = -80:10:80;                 % Elevation angles (excluding poles)
[az_grid, el_grid] = meshgrid(az,el);
poles = [0 0; -90 90];          % Add south and north poles
nDir = [poles [az_grid(:) el_grid(:)]']; % Element normal directions
N = size(nDir,2);               % Number of elements
[x, y, z] = sph2cart(degtorad(nDir(1,:)), degtorad(nDir(2,:)),R*ones(1,N));
hemisphericalConformalArray = phased.ConformalArray(...
    'ElementPosition',[x; y; z],'ElementNormal',nDir);

viewArray(hemisphericalConformalArray,...
    'Title','Hemispherical Conformal Array');
view(90,0)
```

**Hemispherical Conformal Array**



Array Span:
X axis = 2 m
Y axis = 4 m
Z axis = 4 m
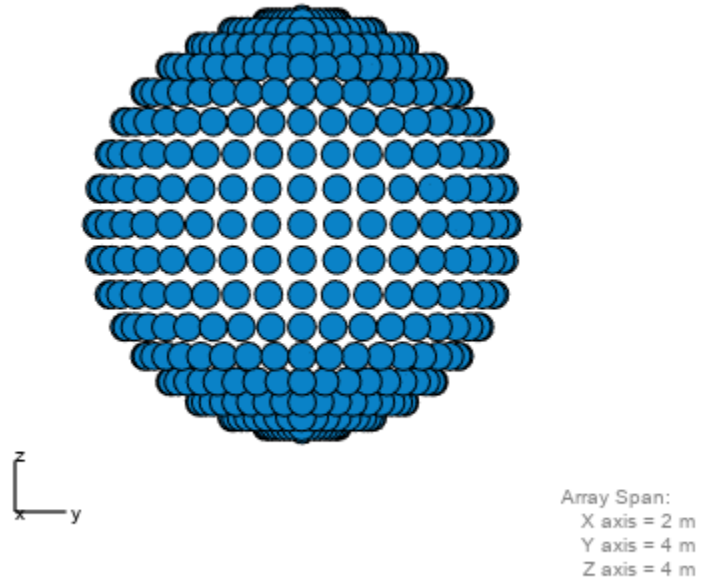
**Subarrays**

You can model and visualize subarrays. Next is an example of contiguous subarrays.

```matlab
replicatedURA = phased.ReplicatedSubarray('Subarray',phased.URA(5),...
                                'Layout','Rectangular',...
                                'GridSize',[3 3],'GridSpacing','Auto');

viewArray(replicatedURA,'Title','3x3 Subarrays Each Having 5x5 Elements');
```

3x3 Subarrays Each Having 5x5 Elements
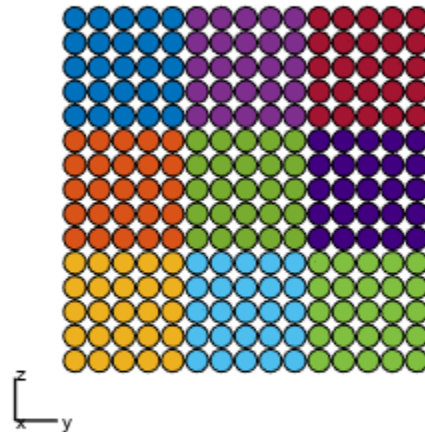


Array Span:
X axis = 0 m
Y axis = 7 m
Z axis = 7 m

You can lay out subarrays on a nonuniform grid. The next example models the failure of a T/R module for one subarray.

```
Ns = 6;                    % Number of subarrays
posc = zeros(3,Ns);
posc(2,:) = -5:2.5:7.5;    % Subarray phase centers
posc(:,3) = [];            % Take out 3rd subarray to model T/R failure
defectiveSubarray = phased.ReplicatedSubarray(...
    'Subarray',phased.URA([25 5]),...
    'Layout','Custom',...
    'SubarrayPosition',posc, ...
    'SubarrayNormal',zeros(2,Ns-1));

viewArray(defectiveSubarray,'Title','Defective Subarray');
view(90,0)
```

**Defective Subarray**
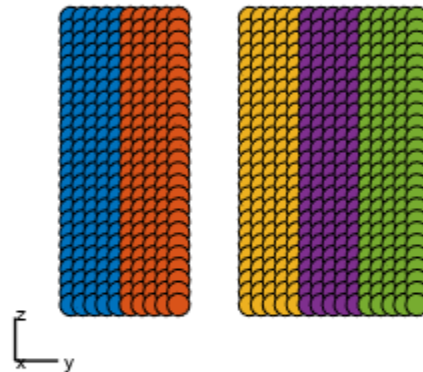


Array Span:
X axis = 0.0 m
Y axis = 14.5 m
Z axis = 12.0 m

Subarrays can be interlaced and overlapped to mitigate grating lobes.

```matlab
N = 40;  % Number of elements
Ns = 8;  % Number of subarrays
sel = zeros(Ns,N);
Nsec = N/Ns;
for m = 1:Ns
    if m==1
        sel(m,(m-1)*Nsec+1:m*Nsec+1) = 1;
    elseif m==Ns
        sel(m,(m-1)*Nsec:m*Nsec) = 1;
    else
        sel(m,(m-1)*Nsec:m*Nsec+1) = 1;
    end
end
overlappedSubarray = phased.PartitionedArray('Array',phased.ULA(N),...
    'SubarraySelection', sel);

viewArray(overlappedSubarray,'Title','Overlapped Subarrays');
set(gca,'CameraViewAngle',4.65);
```

**Overlapped Subarrays**
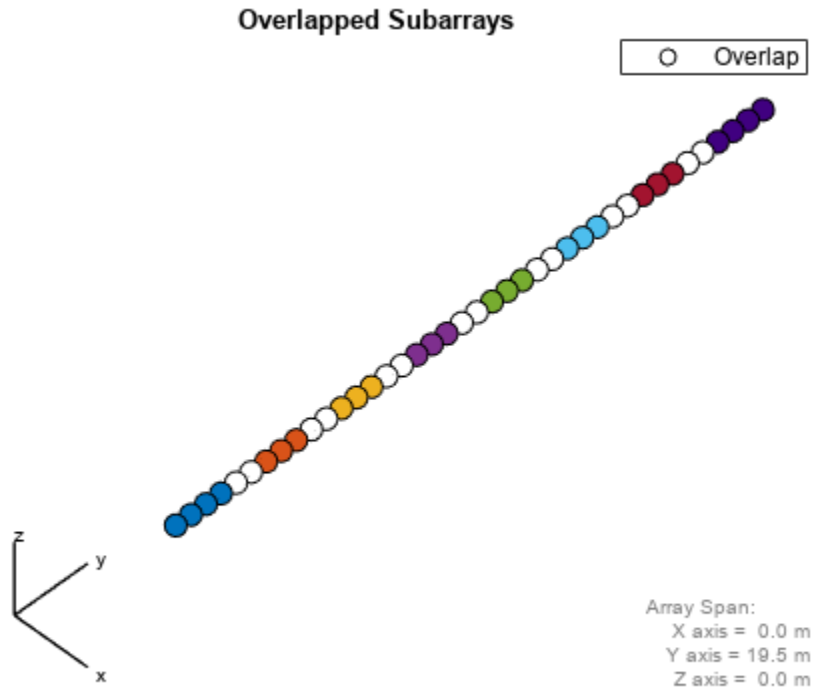


In certain space-constrained applications, such as on satellites, multiple antenna arrays must share the same space. Groups of elements are interleaved, interlaced or interspersed. The next example models interleaved, non-overlapped subarrays.

```
N = 20;
idx = reshape(randperm(N*N),N,N);
sel = zeros(N,N*N);
for i =1:N,
    sel(i,idx(i,:)) = 1;
end
interleavedArray = phased.PartitionedArray('Array',phased.URA(N),...
    'SubarraySelection',sel);

viewArray(interleavedArray,'Title','Interleaved Arrays');
```

**Interleaved Arrays**
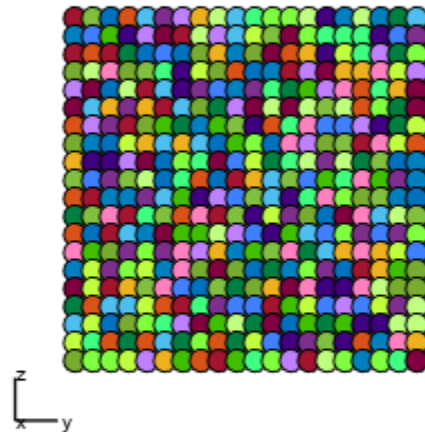


Array Span:
X axis = 0.0 m
Y axis = 9.5 m
Z axis = 9.5 m

Another type of nonplanar antenna array is an array with multiple planar faces. The next example shows uniform hexagonal arrays arranged as subarrays on a sphere.

```
R = 9;                             % Radius (m)
az = unigrid(-180,60,180,'[)');    % Azimuth angles
el = unigrid(-30,60,30);           % Elevation angles (excluding poles)
[az_grid, el_grid] = meshgrid(az,el);
poles = [0 0; -90 90];             % Add south and north poles
nDir = [poles [az_grid(:) el_grid(:)]']; % Subarray normal directions
N = size(nDir,2);                  % Number of subarrays
[x, y, z] = sph2cart(degtorad(nDir(1,:)), degtorad(nDir(2,:)),R*ones(1,N));
sphericalHexagonalSubarray = phased.ReplicatedSubarray('Subarray',uha,...
        'Layout','Custom',...
        'SubarrayPosition',[x; y; z], ...
        'SubarrayNormal',nDir);

viewArray(sphericalHexagonalSubarray,...
    'Title','Hexagonal Subarrays on a Sphere');
view(30,0)
```

## Hexagonal Subarrays on a Sphere
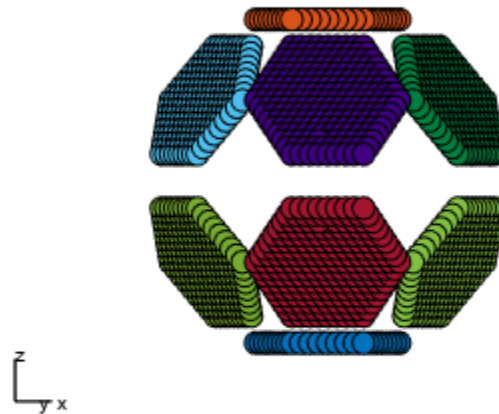


Array Span:
X axis = 19.053 m
Y axis = 18.500 m
Z axis = 18.000 m

You can also view the array from a different angle and interactively rotate it in 3-D.

```
view(0,90)
rotate3d on
```

**Hexagonal Subarrays on a Sphere**
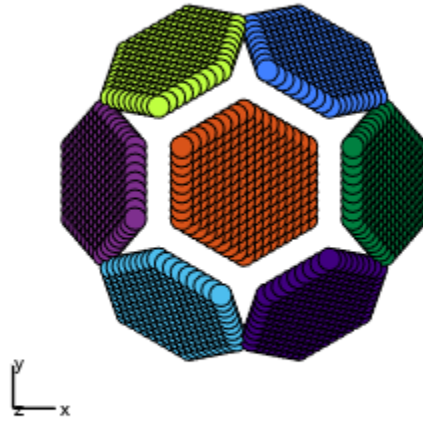


Array Span:
X axis = 19.053 m
Y axis = 18.500 m
Z axis = 18.000 m

# Using Pilot Calibration to Compensate for Array Uncertainties

This example shows how to use pilot calibration to improve the performance of an antenna array in the presence of unknown perturbations.

**Introduction**

In principle, one can easily design an ideal uniform linear array (ULA) to perform array processing tasks such as beamforming or direction of arrival estimation. In practice, there is no such thing as an ideal array. For example, there will always be some inevitable manufacturing tolerances among different elements within the array. Since in general it is impossible to obtain exact knowledge about those variations, they are often referred to as uncertainties or perturbations. Commonly observed uncertainties include element gain and element phase uncertainties (electrical uncertainties) as well as element location uncertainties (geometrical uncertainties).

The presence of uncertainties in an array system causes rapid degradation in the detection, resolution, and estimation performance of array processing algorithms. Therefore it is critical to calibrate the array before its deployment. In addition to the aforementioned factors, uncertainties can also arise due to other factors such as hardware aging and environmental effects. Calibration is therefore also performed on a regular basis in all deployed systems.

There are many array calibration algorithms. This example focuses on the pilot calibration approach [1], where the uncertainties are estimated from the response of the array to one or more known external sources at known locations. The example compares the effect of uncertainties on the array performance before and after the calibration.

**Modeling Electrical and Geometrical Uncertainties**

Consider an ideal 6-element ULA along y-axis operating with half-wavelength spacing and uniform tapering. For a ULA, the expected element positions and tapers can be computed.

```
N = 6;
designed_pos = [zeros(1,N);(0:N-1)*0.5;zeros(1,N)];
designed_taper = ones(N,1);
```

Next, model the perturbations that may exist in a real array. These are usually modeled as random variables. For example, assume that the taper's magnitude and phase are perturbed by normally distributed random variables with standard deviations of 0.1 and 0.05, respectively.

```
rng(2014);
taper = (designed_taper + 0.1*randn(N,1)).*exp(1i*0.05*randn(N,1));
```

The following figure shows the difference between the magnitude and phase of the perturbed taper and the designed taper.

```
helperCompareArrayProperties('Taper',taper,designed_taper,...
    {'Perturbed Array','Designed Array'});
```
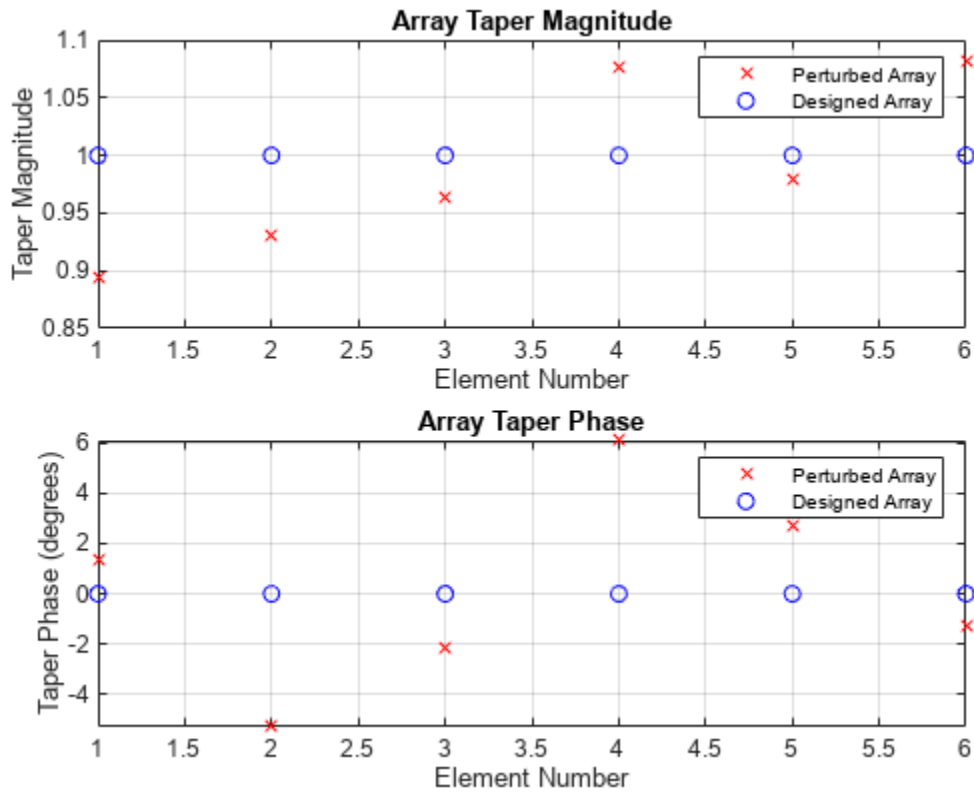
Perturbations in the sensor locations in the x, y, and z directions are generated similarly with a standard deviation of 0.05.

```
pos = designed_pos + 0.05*randn(3,N);
```

The figure below shows where the element positions of the perturbed array and the ideal array.

```
helperCompareArrayProperties('Position',pos,designed_pos,...
    {'Perturbed Array','Designed Array'});
```
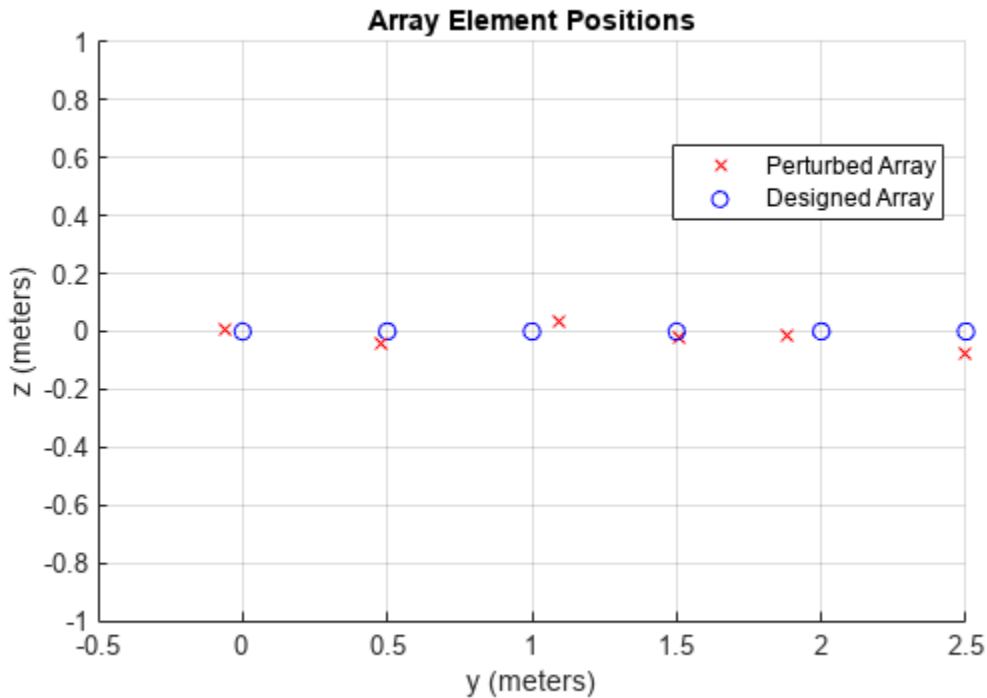
**Effect of Array Perturbation**

The previous section shows the difference between the designed, ideal array and the real, perturbed array. Because of these errors, if one blindly applies processing steps, such as beamforming weights, computed using the designed array, on the perturbed array, the performance degrades significantly.

Consider the case of an LCMV beamformer designed to steer the ideal array to a direction of 10 degrees azimuth with two interferences from two known directions of -10 degrees azimuth and 60 degrees azimuth. The goal is to preserve the signal of interest while suppressing the interferences.

If the precise knowledge of the array's taper and geometry is known, the beamforming weights can be computed as follows:

```
% Generate 10K samples from target and interferences with 30dB SNR
az = [-10 10 60];
Nsamp = 1e4;
ncov = db2pow(-30);
[~,~,rx_cov] = sensorsig(pos,Nsamp,az,ncov,'Taper',taper);

% Compute LCMV beamforming weights assuming the designed array
sv = steervec(pos,az);
w = lcmvweights(bsxfun(@times,taper,sv),[0;1;0],rx_cov);
```

However, since the array contains unknown perturbations, beamforming weights must be computed based on the positions and taper of the designed array.

```
designed_sv = steervec(designed_pos,az);
designed_w = lcmvweights(bsxfun(@times,designed_taper,designed_sv),...
    [0;1;0],rx_cov);
```

The following figure compares the expected beam pattern with the pattern resulted from applying the designed weights on the perturbed array.

```
helperCompareBeamPattern(pos,taper,w,designed_w,-90:90,az,...
    {'Expected Pattern','Uncalibrated Pattern'},...
    'Beam Pattern Before Calibration');
```
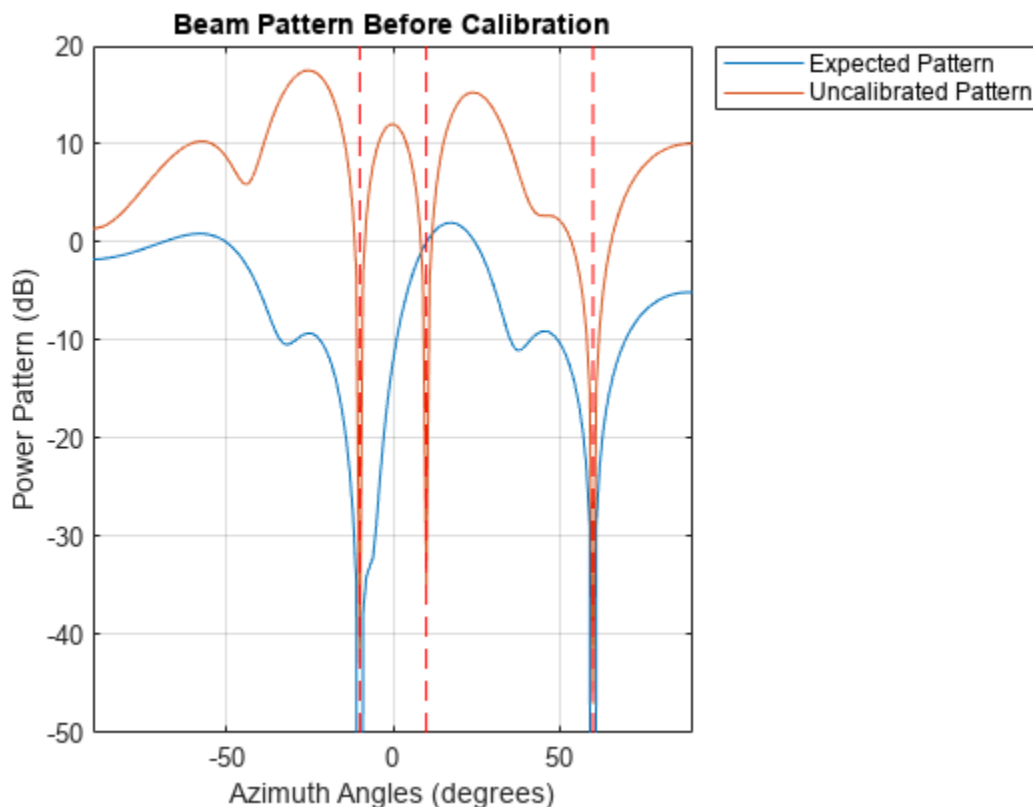


From the plotted patterns, it is clear that the pattern resulted from the uncalibrated weights does not satisfy the requirements. It puts a null around the desired 10 degrees azimuth direction. This means that the desired signal can no longer be retrieved. Fortunately, array calibration can help bring the pattern back to order.

**Pilot Calibration**

There are many algorithms available to perform array calibration. One class of commonly used algorithms is pilot calibration. The algorithm sets up several sources in known directions and then uses the array to receive the signal from those transmitters. Because these transmitters are at the known directions, the expected received signal of the ideal array can be computed. Comparing these with the actual received signal, it is possible to derive the difference due to the uncertainties and correct them.

The code below shows the process of array calibration. First, the pilot sources need to be chosen at different directions. Note that the number of pilot sources determines how many uncertainties the

algorithm can correct. In this example, to correct both sensor location uncertainties and taper uncertainty, a minimum of four external sources is required. If more sources are used, the estimation will improve.

```
pilot_ang = [-60, -5 ,5, 40; -10, 0, 0, 30];
```

The four pilot sources are located at the following azimuth and elevation angle pairs: (-60, -10), (-5, 0), (5, 0), and (40, 30). The received signal from these pilots can be simulated as

```
for m = size(pilot_ang,2):-1:1
    calib_sig(:,:,m) = sensorsig(pos,Nsamp,pilot_ang(:,m),...
        ncov,'Taper',taper);
end
```

Using the received signal from the pilots at the array, together with the element positions and tapers of the designed array, the calibration algorithm [1] estimates the element positions and tapers for the perturbed array.

```
[est_pos,est_taper] = pilotcalib(designed_pos,...
    calib_sig,pilot_ang,designed_taper);
```

Once the estimated positions and taper are available, these can be used in place of the designed array parameters when calculating beamformer weights. This results in the array pattern represented by the red line below.

```
corrected_w = lcmvweights(bsxfun(@times,est_taper,...
    steervec(est_pos,az)),[0;1;0],rx_cov);

helperCompareBeamPattern(pos,taper,...
    w,corrected_w,-90:90,az,...
    {'Expected Pattern','Calibrated Pattern'},...
    'Beam Pattern After Calibration');
```
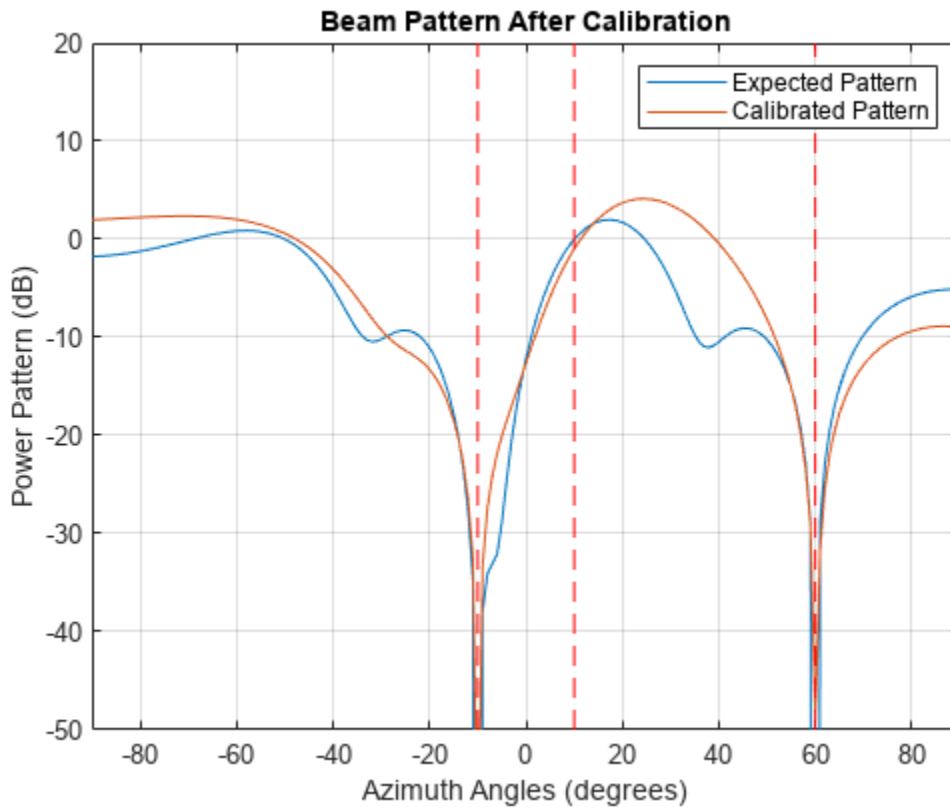
The figure above shows that the pattern resulting from the calibrated array is much better than the one from the uncalibrated array. In particular, signals from the desired direction are now preserved.

**Summary**

This example shows how uncertainties of an array can impact its response pattern and in turn degrade the array's performance. The example also illustrates how pilot calibration can be used to help restore the array performance.

**References**

[1] N. Fistas and A. Manikas, "A New General Global Array Calibration Method", IEEE Proceedings of ICASSP, Vol. IV, pp. 73-76, April 1994.

# Using Self Calibration to Accommodate Array Uncertainties

This example shows a self calibration procedure based on a constrained optimization process. Sources of opportunity are exploited to simultaneously estimate array shape uncertainties and source directions.

This example requires Optimization Toolbox™.

**Introduction**

In theory, one can design a perfect uniform linear array (ULA) to perform all sorts of processing such as beamforming or direction of arrival estimation. Typically this array will be calibrated in a controlled environment before being deployed. However, uncertainties may arise in the system during operation indicating that the array needs recalibrating. For instance, environmental effects may cause array element positions to become perturbed, introducing array shape uncertainties. The presence of these uncertainties causes rapid degradation in the detection, resolution and estimation performance of array processing algorithms. It is therefore critical to remove these array uncertainties as soon as possible.

There are many array calibration algorithms. This example focuses on one class of them, self calibration (also called auto-calibration), where uncertainties are estimated jointly with the positions of a number of external sources at unknown locations [1]. Unlike pilot calibration, this allows an array to be re-calibrated in a less known environment. However, in general, this results in a small number of signal observations with a large number of unknowns. There are a number of approaches to solving this problem as described in [2]. One is to construct and optimize against a cost function. These cost functions tend to be highly non-linear and contain local minima. In this example, a cost function based on the Multiple Signal Classification (MUSIC) algorithm [3] is formed and solved as an fmincon optimization problem using Optimization Toolbox. In the literature, many other combinations also exist [2].

**A Perfect Array**

Consider first a 5-element ULA operating with half wavelength spacing is deployed. In such an array, the element positions can be readily computed.

```
N = 5;
designed_pos = [zeros(1,N);-(N-1)/2:(N-1)/2;zeros(1,N)]*0.5;
```

**A Not So Perfect Array**

Next, assume the array is perturbed whilst in operation and so undergoes array shape uncertainties in the x and y dimensions. In order to fix the global axes, assume that the first sensor and the direction to the second sensor is known as prescribed in [4].

```
rng default
pos_std = 0.02;
perturbed_pos = designed_pos + pos_std*[randn(2,N);zeros(1,N)];
perturbed_pos(:,1) = designed_pos(:,1); % The reference sensor has no
                                        % uncertainties
perturbed_pos(1,2) = designed_pos(1,2); % The x axis is fixed down by
                                        % assuming the x-location of
                                        % another sensor is known
```

**Visualize the Array Imperfections**

The figure below shows the difference between the deployed and the perturbed array.

```
helperCompareArrayProperties('Position',perturbed_pos,designed_pos,...
    {'Perturbed Array','Deployed Array'});
view(90,90);
```



**Array Element Positions**
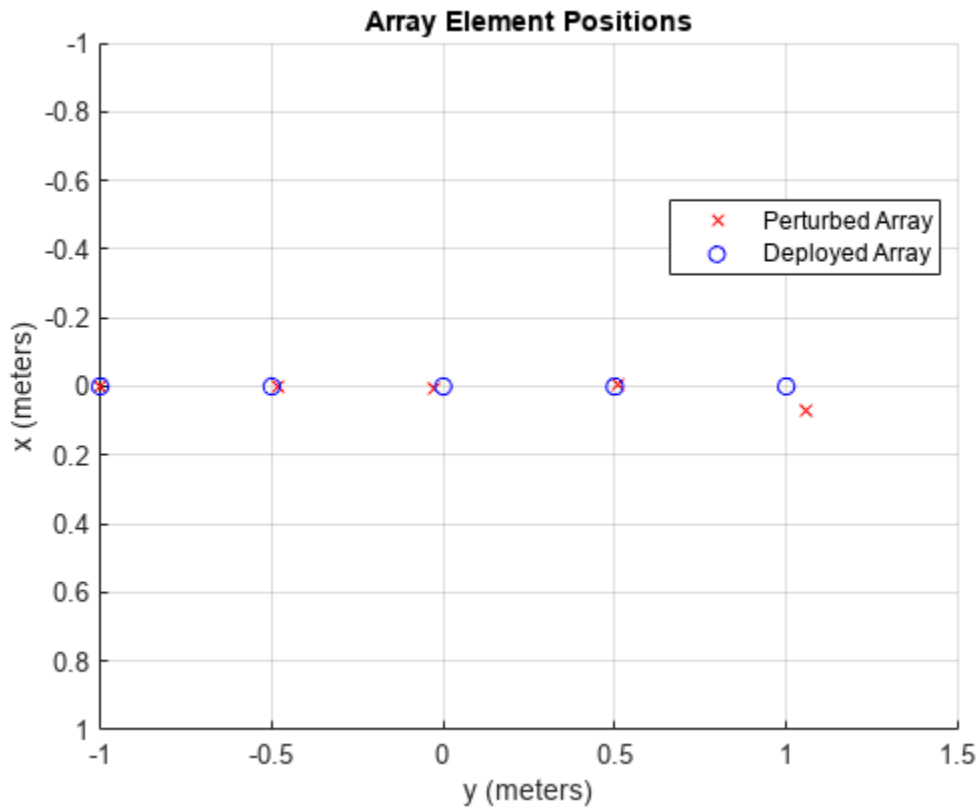
**Degradation of DOA Estimation**

The previous section shows the difference between the deployed array and an array which has undergone perturbations while in operation. If one blindly uses the processing designed for the deployed array, the performance of the array reduces. For example, consider a beamscan estimator is used to estimate the directions of 3 unknown sources at -20, 40 and 85 degrees azimuth.

```
% Generate 100K samples with 30dB SNR
ncov = db2pow(-30);
Nsamp = 1e5;                    % Number of snapshots (samples)
incoming_az = [-20,40,85]; % Unknown source locations to be estimated
M = length(incoming_az);
[x_pert,~,Rxx] = sensorsig(perturbed_pos,Nsamp,incoming_az,ncov);

% Estimate the directions of the sources
ula = phased.ULA('NumElements',N);
spatialspectrum = phased.BeamscanEstimator('SensorArray',ula,...
    'DOAOutputPort',true,'NumSignals',M);
[y,estimated_az] = spatialspectrum(x_pert);
```

```
incoming_az

incoming_az = 1×3

   -20    40    85


estimated_az

estimated_az = 1×3

   -19    48    75
```

These uncertainties degrade the array performance. Self calibration can allow the array to be re-calibrated using sources of opportunity, without needing to know their locations.

**Self Calibration**

A number of self calibration approaches are based on optimizing a cost function to jointly estimate unknown array and source parameters (such as array sensor and source locations). The cost function and optimization algorithm must be carefully chosen to encourage a global solution to be reached as easily and quickly as possible. In addition, parameters associated with the optimization algorithm must be tuned for the given scenario. A number of combinations of cost function and optimization algorithm exist in the literature. For this example scenario, a MUSIC cost function [3] is chosen alongside an fmincon optimization algorithm. As the scenario changes, it may be appropriate to adapt the approach used depending upon the robustness of the calibration algorithm. For instance, in this example, the performance of the calibration algorithm drops as sources move away from end-fire or the number of array elements increase. The initial estimates of the source locations estimated previously are used as the initialization criterion of the optimization procedure.

```
fun = @(x_in)helperMUSICIteration(x_in,Rxx,designed_pos);
nvars = 2*N - 3 + M;                        % Assuming 2D uncertainties
x0 = [0.1*randn(1,nvars-M),estimated_az];   % Initial value
locTol = 0.1;                               % Location tolerance
angTol = 20;                                % Angle tolerance
lb = [-locTol*ones(nvars-M,1);estimated_az.'-angTol]; % lower bound
ub = [locTol*ones(nvars-M,1);estimated_az.'+angTol];  % upper bound

options = optimoptions('fmincon','TolCon',1e-6,'DerivativeCheck','on',...
    'Display','off');
[x,fval,exitflag] = fmincon(fun,x0,[],[],[],[],lb,ub,[],options);

% Parse the final result
[~,perturbed_pos_est,postcal_estimated_az]=helperMUSICIteration(...
    x,Rxx,designed_pos);

helperCompareArrayProperties('Position',perturbed_pos,perturbed_pos_est,...
    {'Perturbed Array','Calibrated Array'});
view(90,90);
```

**Array Element Positions**
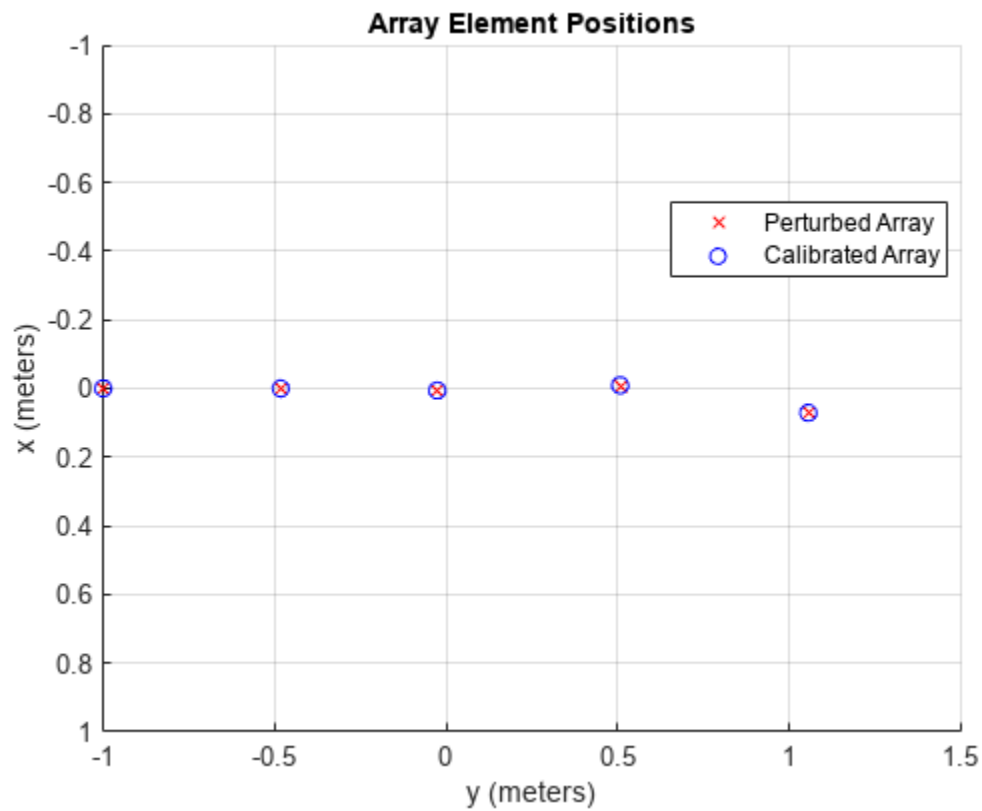


```
polarplot(deg2rad(incoming_az),[1 1 1],'s',...
    deg2rad(postcal_estimated_az(1,:)),[1 1 1],'+',...
    deg2rad(estimated_az),[1 1 1],'o','LineWidth',2,'MarkerSize',10)
legend('True directions','DOA after calibration',...
    'DOA before calibration','Location',[0.01 0.02 0.3 0.13])
rlim([0 1.3])
```
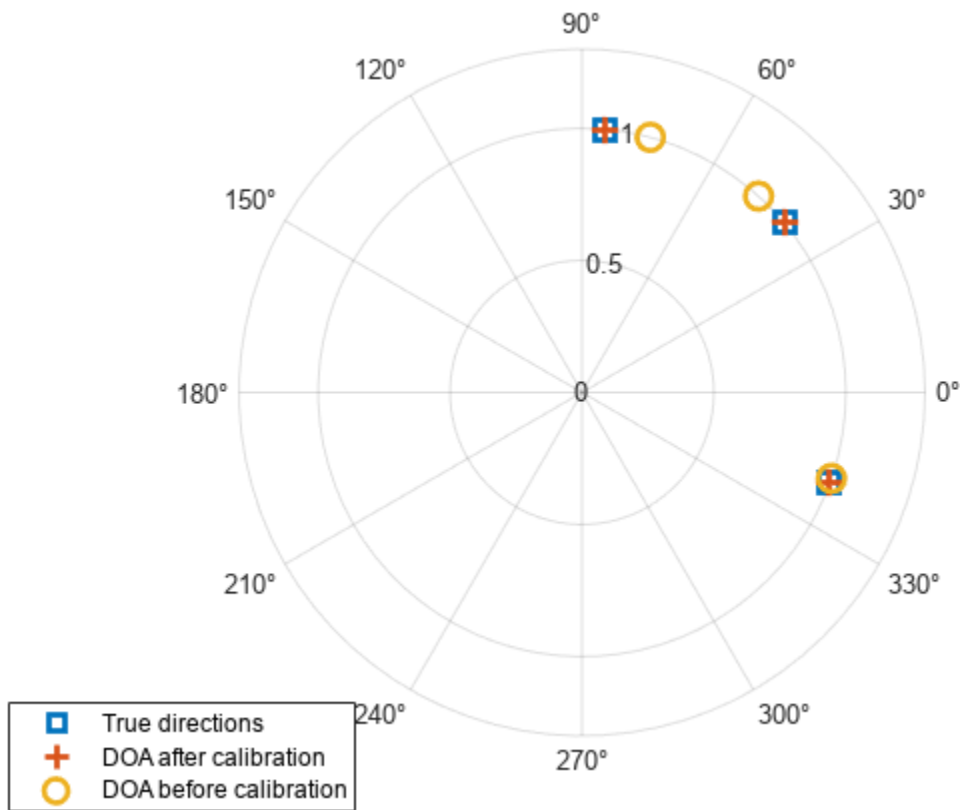
By performing this calibration process the accuracy of the source estimation has improved significantly. In addition, the positions of the perturbed sensors have also been estimated which can be used as the new array geometry in the future.

**Summary**

This example shows how array shape uncertainties can impact the ability to estimate the direction of arrival of unknown sources. The example also illustrates how self calibration can be used to overcome the effects of these perturbations and estimate these uncertainties simultaneously.

**References**

[1]Van Trees, H. Optimum Array Processing. New York: Wiley-Interscience, 2002.

[2] E Tuncer and B Friedlander. Classical and Modern Direction-of-Arrival Estimation. Elsevier, 2009.

[3] Schmidt, R. O. "Multiple Emitter Location and Signal Parameter Estimation." IEEE Transactions on Antennas and Propagation. Vol. AP-34, March, 1986, pp. 276-280.

[4] Y. Rockah and P. M. Schultheiss. Array shape calibration using sources in unknown locations- Part I: Farfield sources. IEEE Trans. ASSP, 35:286-299, 1987.

# Subarrays in Phased Array Antennas

This example shows how to model subarrays, commonly used in modern phased array systems, using Phased Array System Toolbox™ and perform analyses.

**Introduction**

Phased array antennas provide many benefits over traditional dish antennas. The elements of phased array antennas are easier to manufacture; the entire system suffers less from component failures; and best of all, can be electronically scanned toward different directions.

However, such flexibility does not come for free. Taking full advantage of a phased array requires placing steering circuitry and T/R switches behind each individual element. For applications that require large arrays with thousands or tens of thousands of elements, the cost of doing so is too high to be practical. In addition, in many such applications, the desired performance does not require full degree of freedom from the array. Hence, in practice, deployed systems often use a compromised approach. Elements are grouped into subarrays and then subarrays form the entire array. The elements are still easy to manufacture; the entire array is still robust with respect to component failures; in addition, T/R switches are only needed at each subarray, thus significantly reducing the cost.

The following sections show how to model a subarray network with different configurations for two specific applications: limited field of view (LFOV) arrays and wideband arrays.

**Limited Field of View (LFOV) Arrays**

LFOV arrays are commonly used in satellite applications. As the name suggests, an LFOV array only scans within a very limited window, normally less than 10 degrees. Because of that, it is possible to use subarrays and such subarrays can be placed at a spacing much larger than half of the wavelength.

The simplest way to construct an array with subarrays is to contiguously tile the subarray. The following code snippet constructs a 64-element ULA consists of eight 8-element ULAs. Within each subarray, the elements are spaced by half the wavelength. Note that there is no steering capability inside each subarray so the array can only be steered using subarrays.

The array geometry can be seen in the following figure.



```
fc = 3e8;
c = 3e8;
antenna = phased.IsotropicAntennaElement('BackBaffled',true);
N = 64;
Nsubarray = 8;
subula = phased.ULA(N/Nsubarray,0.5*c/fc,'Element',antenna);

replarray = phased.ReplicatedSubarray('Subarray',subula,...
    'GridSize',[1 Nsubarray])
```

```
replarray =
  phased.ReplicatedSubarray with properties:

            Subarray: [1x1 phased.ULA]
              Layout: 'Rectangular'
            GridSize: [1 8]
         GridSpacing: 'Auto'
    SubarraySteering: 'None'
```

Next, compare the radiation pattern of this array to the radiation pattern of a 64-element ULA with no subarrays.

```
refula = phased.ULA(N,0.5*c/fc,'Element',antenna);
subplot(2,1,1), pattern(replarray,fc,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c);
title('Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2), pattern(refula,fc,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c);
title('ULA Azimuth Cut'); axis([-90 90 -50 0]);
```
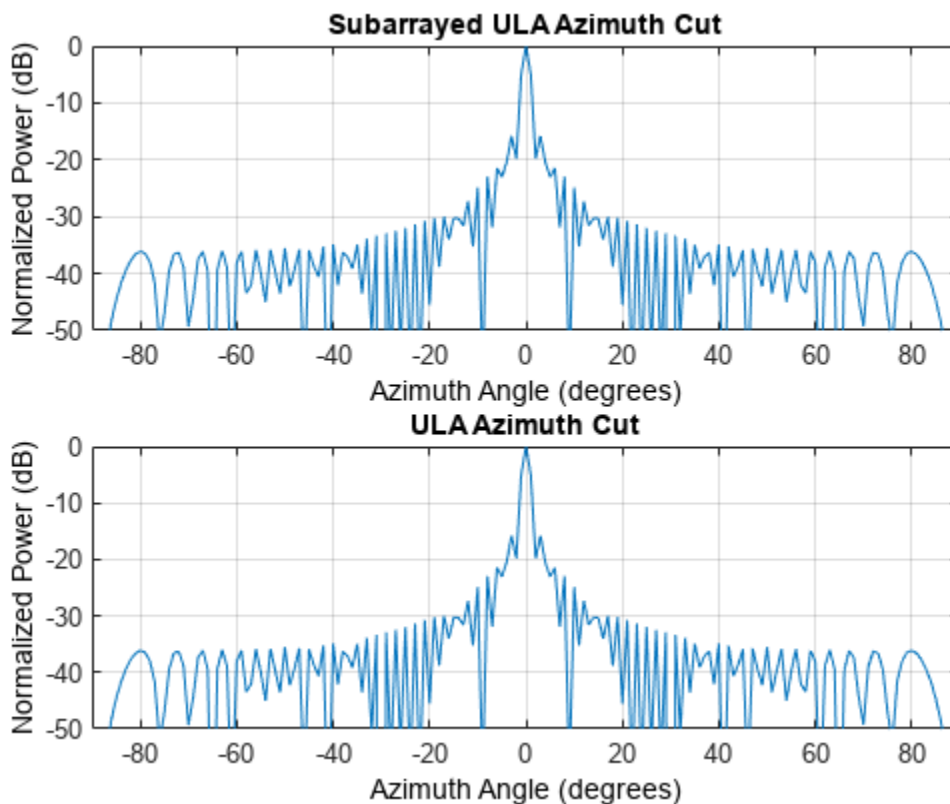


From the plot, it is clear that the two responses are identical at broadside. Note that even though the subarrays are widely spaced, there is no grating lobe in the response.

Next, steer both arrays to 2 degrees azimuth.

```
steerang = 2;
steeringvec_replarray = phased.SteeringVector('SensorArray',replarray,...
```

```
        'PropagationSpeed',c);
w = steeringvec_replarray(fc,steerang);
steeringvec_refula = phased.SteeringVector('SensorArray',refula,...
        'PropagationSpeed',c);
wref = steeringvec_refula(fc,steerang);

subplot(2,1,1), pattern(replarray,fc,-180:180,0,'Type','powerdb',...
        'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',w);
title('Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2), pattern(refula,fc,-180:180,0,'Type','powerdb',...
        'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',wref);
title('ULA Azimuth Cut'); axis([-90 90 -50 0]);
```
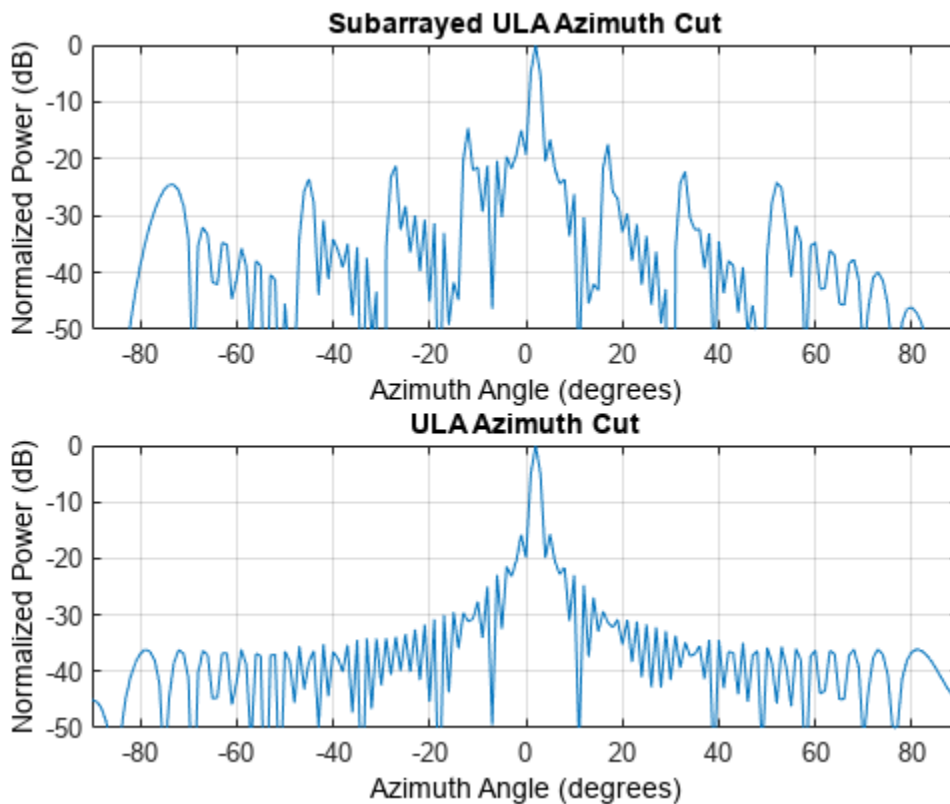


In this case, the response of the reference array still retains its original shape, but this is not the case for the subarrayed ULA. For the subarrayed ULA, although the mainlobe is correctly steered and stands well above the sidelobes, the response clearly shows what is often referred to as *quantization lobes*. The name comes from the fact that the steering is at the subarray level; hence, the required phase shift for each element is quantized at the subarray level. This effect gets worse when the array is steered further from the broadside. The following plots show the response after steering the arrays toward 6 degrees off broadside.

```
steerang = 6;
w = steeringvec_replarray(fc,steerang);
wref = steeringvec_refula(fc,steerang);

subplot(2,1,1), pattern(replarray,fc,-180:180,0,'Type','powerdb',...
        'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',w);
```

```
title('Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2), pattern(refula,fc,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',wref);
title('ULA Azimuth Cut'); axis([-90 90 -50 0]);
```
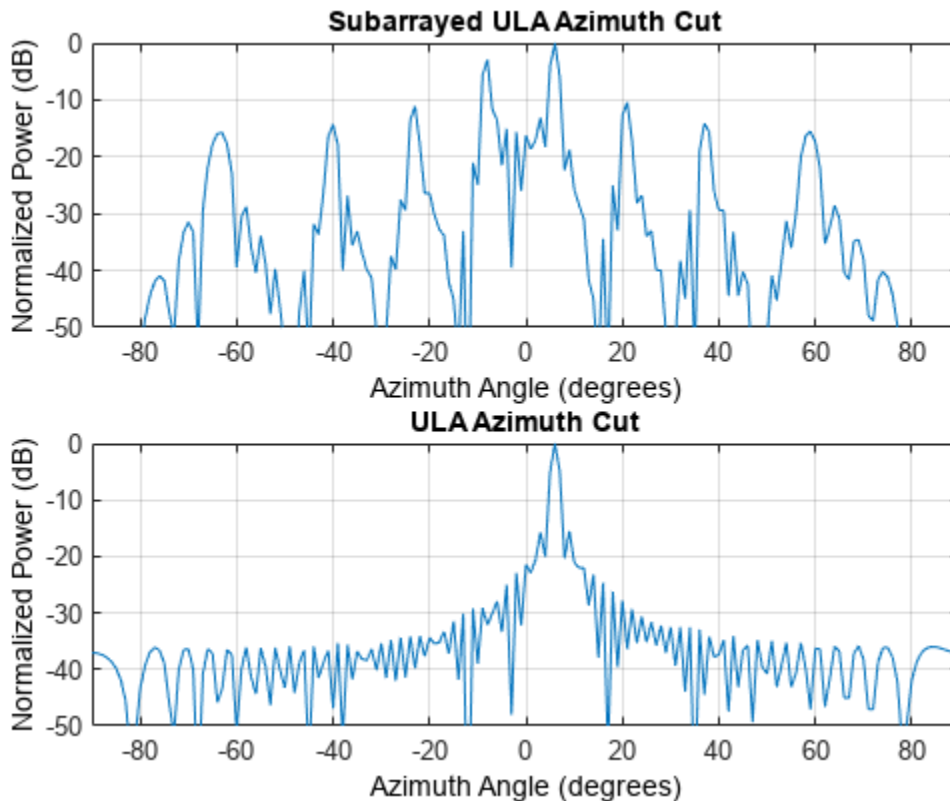


Therefore, when forming an LFOV, one needs to be cautious about using contiguous subarrays.

One way to compensate for quantization lobes is to add phase shifters behind each element. Although it increases the cost, it still provides a big saving compared to the full degree of freedom array because the T/R switches, which are the most expensive parts, only need to be implemented at the subarray level. If there is a phase shifter behind each element, then the response becomes much better, as shown in the following plots, assuming the phase shifters behind each element are also configured to point each subarray toward 6 degrees off the broadside.

```
release(replarray);
replarray.SubarraySteering = 'Phase';
replarray.PhaseShifterFrequency = fc;

subplot(2,1,1);
pattern(replarray,fc,-180:180,0,'Type','powerdb','Weights',w,...
    'CoordinateSystem','rectangular','PropagationSpeed',c,'SteerAngle',6);
title('Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2);
pattern(refula,fc,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',wref);
title('ULA Azimuth Cut'); axis([-90 90 -50 0]);
```
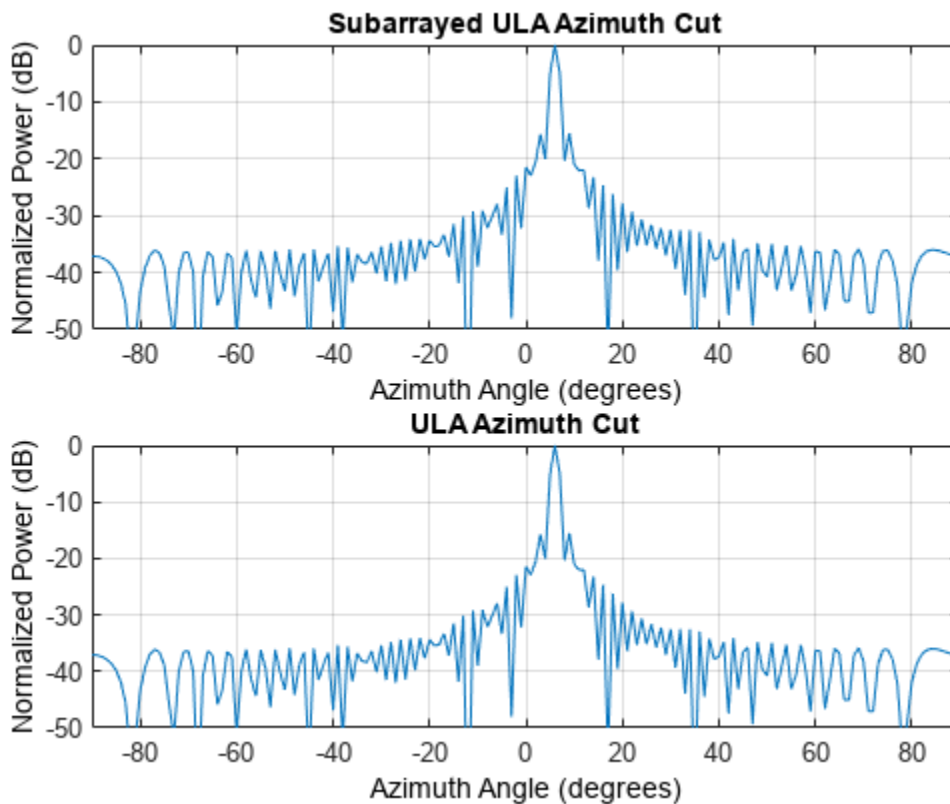
As a side note, the element and the subarrays do not necessarily steer to the same direction. In some applications, the elements inside the subarrays are steered toward a specific direction. The subarrays can then be steered to slightly different directions to search the vicinity.

**Wideband Scanning Arrays**

Although an electronically scanned array is often called a *phased* array, in reality, adjusting the phase is only one way to steer the array. The phase shifters are, by nature, narrowband devices so they only work well within a narrow band, especially for large arrays. The following figure shows the radiation patterns when the reference array is phase steered to 30 degrees, both at the carrier frequency and 3 percent above the carrier frequency.

```
fsteer = [1 1.03]*fc;
steerang = 30;
release(steeringvec_refula);
wref = squeeze(steeringvec_refula(fc,steerang));

subplot(2,1,1)
pattern(refula,fsteer,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',wref);
title('ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2)
pattern(refula,fsteer,-180:180,0,'Type','powerdb',...
    'CoordinateSystem','rectangular','PropagationSpeed',c,'Weights',wref);
title('ULA Azimuth Cut, Peak Zoom View'); axis([25 35 -5 0]);
```
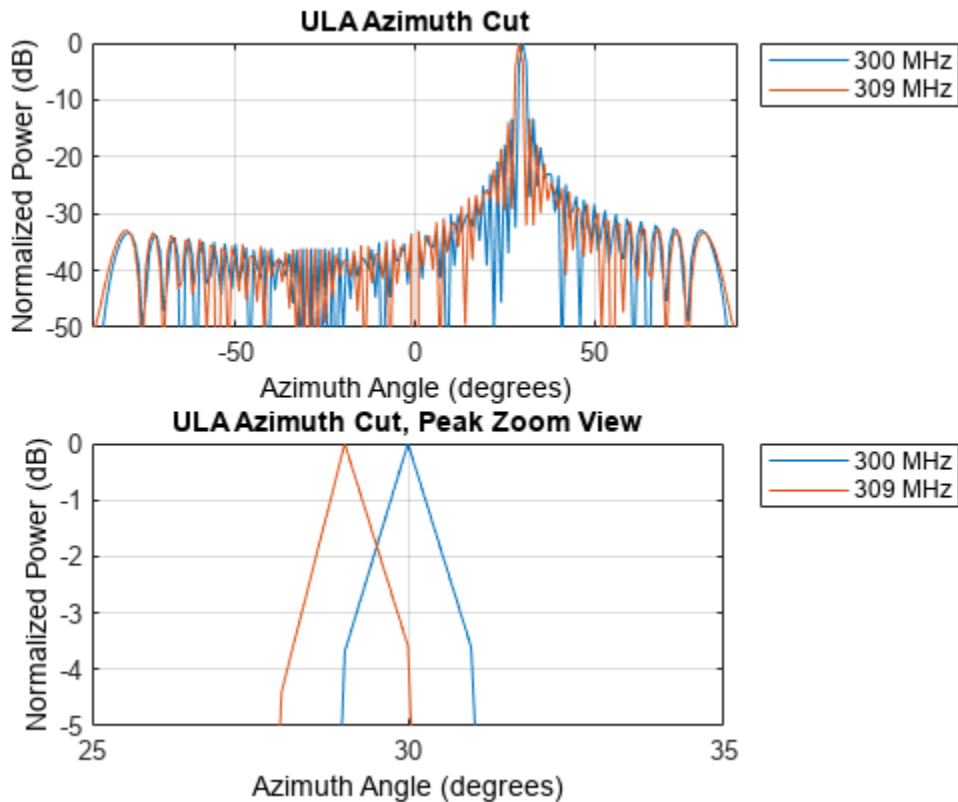
It is obvious from the figure that even though the frequency offset is a mere 3 percent, the peak location moved away from the desired direction. This is referred to as *squint* effect. Thus, to achieve steering across a wideband, one needs to steer using true time delays.

The most popular way to achieve true time delay is to use cables. However, in a big array aperture with thousands of elements, implementing the potentially huge time delay can require a lot of cables. Hence, this approach is not only expensive, but also cumbersome. Subarrays provide a compromise between the accuracy and feasibility. In summary, within each subarray, the steering is achieved by the phase; and among subarrays, the steering is done by true time delays.

The simplest way to build such an array is to contiguously group the subarrays, as in previous sections.

The following plots compare the radiation patterns at three frequencies for a subarrayed ULA. The array is steered toward 30 degrees azimuth at the subarray level using true time delay. Again, within each subarray, the elements are also steered toward 30 degrees azimuth. The radiation pattern is shown at the carrier frequency, 10 percent above the carrier frequency, and 15 percent above the carrier frequency.

```
steerang = 30;
fsteer = [1 1.1 1.15]*fc;
release(steeringvec_replarray);
release(steeringvec_refula);
w = squeeze(steeringvec_replarray(fsteer,steerang));
wref = squeeze(steeringvec_refula(fsteer,steerang));
```

```
subplot(2,1,1)
pattern(replarray,fsteer,-180:180,0,'Type','powerdb',...
    'PropagationSpeed',c,'CoordinateSystem','rectangular','Weights',w,...
    'SteerAngle',steerang);
title('Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
subplot(2,1,2)
pattern(replarray,fsteer,-180:180,0,'Type','powerdb',...
    'PropagationSpeed',c,'CoordinateSystem','rectangular','Weights',w,...
    'SteerAngle',steerang);
title('Subarrayed ULA Azimuth Cut, Peak Zoom View'); axis([25 35 -5 0]);
```
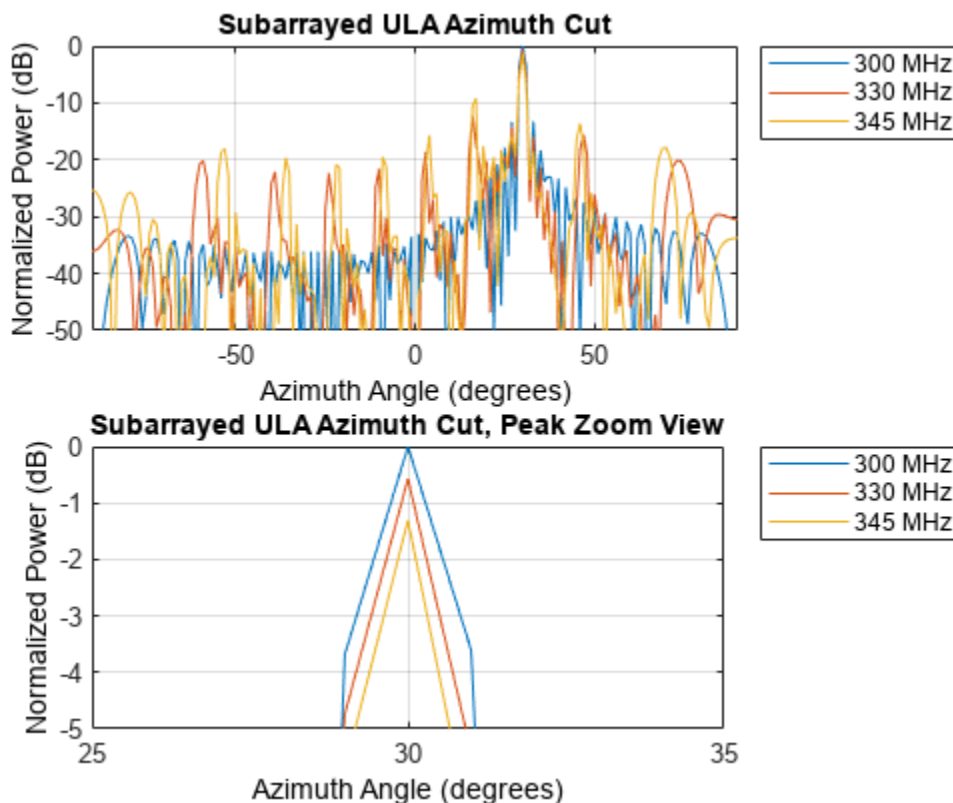
The plots show that the squint effect has been suppressed even though the bandwidth is much wider compared to the previous case. However, as in the LFOV case, if the required bandwidth extends to 15 percent above the carrier frequency, the radiation pattern becomes undesirable due to quantization lobes.

One way to address this issue is to use a configuration with aperiodic subarrays. Examples of such configurations are interlaced subarrays, overlapped subarrays, and even random subarrays. The next example shows an interlaced subarray, where the ends of the subarray are interlaced and overlapped. Because it is no longer formed by identical subarrays, one needs to start with a large array aperture and partition it to achieve such configuration.

The array geometry can be seen in the following figure.
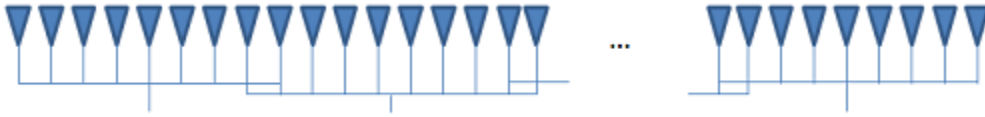
```matlab
partarray = ...
    phased.PartitionedArray('Array',phased.ULA(N,0.5,'Element',antenna),...
    'SubarraySteering','Phase');
sel = zeros(Nsubarray,N);
Nsec = N/Nsubarray;
for m = 1:Nsubarray
    if m==1
        sel(m,(m-1)*Nsec+1:m*Nsec+1) = 1;
    elseif m==Nsubarray
        sel(m,(m-1)*Nsec:m*Nsec) = 1;
    else
        sel(m,(m-1)*Nsec:m*Nsec+1) = 1;
    end
end
partarray.SubarraySelection = sel

partarray =
  phased.PartitionedArray with properties:

                    Array: [1x1 phased.ULA]
         SubarraySelection: [8x64 double]
          SubarraySteering: 'Phase'
     PhaseShifterFrequency: 300000000
       NumPhaseShifterBits: 0
```

The resulting radiation pattern can be seen in the following figures.

```matlab
steeringvec_partarray = ...
    phased.SteeringVector('SensorArray',partarray,'PropagationSpeed',c);
wwa = squeeze(steeringvec_partarray(fsteer,steerang));

subplot(2,1,1);
pattern(partarray,fsteer,-180:180,0,'Type','powerdb',...
    'PropagationSpeed',c,'CoordinateSystem','rectangular','Weights',wwa,...
    'SteerAngle',steerang);
title('Interlaced and Overlapped Subarrayed ULA Azimuth Cut');
axis([-90 90 -50 0]);
subplot(2,1,2);
pattern(replarray,fsteer,-180:180,0,'Type','powerdb',...
    'PropagationSpeed',c,'CoordinateSystem','rectangular','Weights',w,...
    'SteerAngle',steerang);
title('Contiguous Subarrayed ULA Azimuth Cut'); axis([-90 90 -50 0]);
```
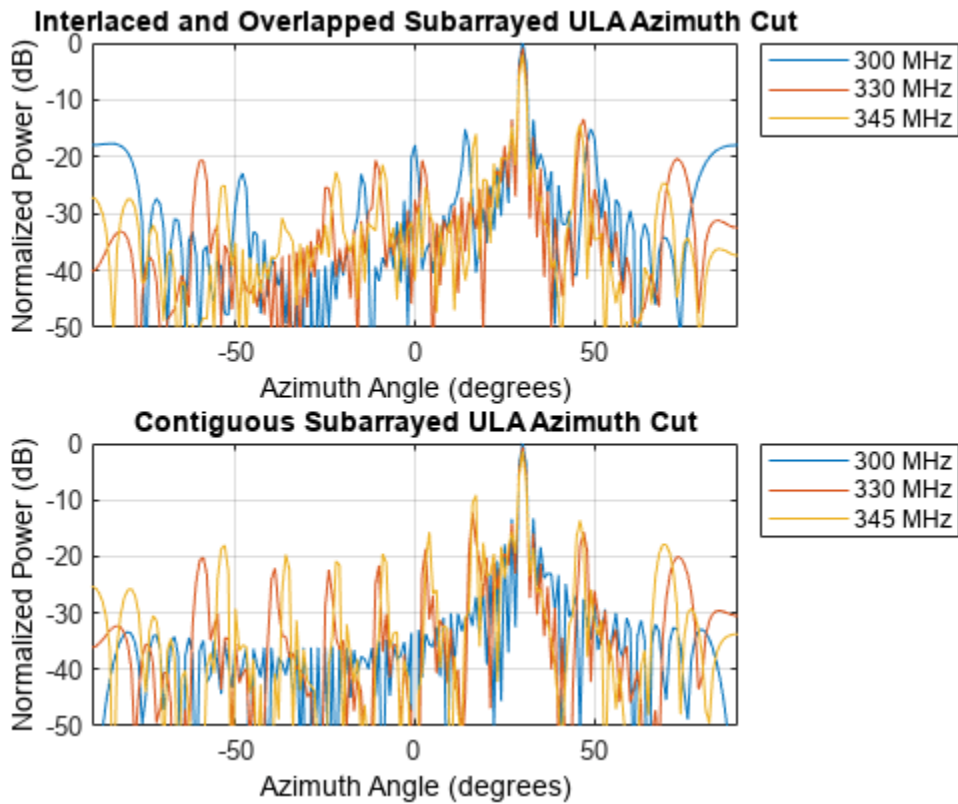
The new radiation pattern suppresses the largest quantization lobe, achieving a gain of around 5 dB. Higher gains can be achieved by designing a more sophisticated overlapped subarray network, but that is outside the scope of this example.

**Summary**

This example shows how to model a phased array with subarrays and illustrates several practical concerns when applying the subarray technique to applications such as LFOV arrays or wideband scanning arrays.

**Reference**

[1] Robert Mailloux, *Electronically Scanned Arrays*, Morgan & Claypool, 2007.

# Tapering, Thinning and Arrays with Different Sensor Patterns

This example shows how to apply tapering and model thinning on different array configurations. It also demonstrates how to create arrays with different element patterns.

**ULA Tapering**

This section shows how to apply a Taylor window on the elements of a uniform linear array (ULA) in order to reduce the sidelobe levels.

```
%set the random seed
rs = rng(6);

% Create a ULA antenna of 10 elements.
N = 20;
ula = phased.ULA(N);

% Clone the ideal ULA
taperedULA = clone(ula);

% Calculate and assign the taper
nbar = 5; sll = -20;
taperedULA.Taper = taylorwin(N,nbar,sll).';
```

Compare the response of the tapered to the untapered array. Notice how the sidelobes of the tapered ULA are lower.

```
helperCompareResponses(taperedULA,ula, ...
                       'Ideal ULA versus Tapered ULA response', ...
                       {'Tapered','Not tapered'});
```
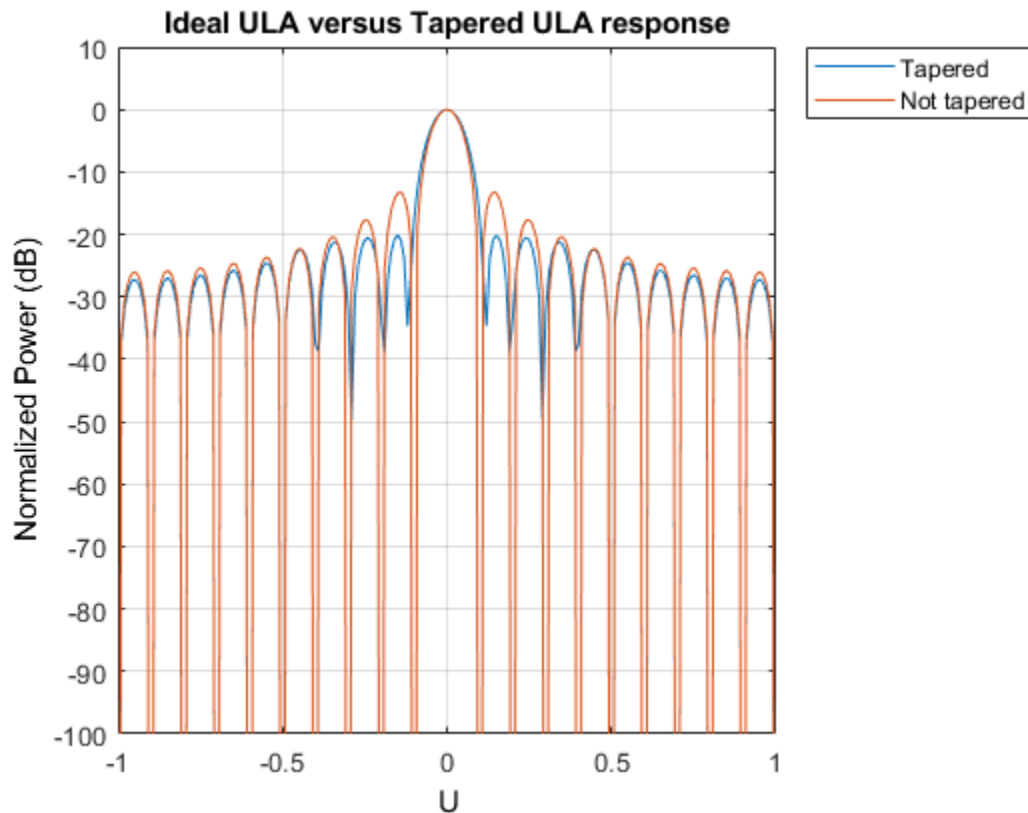
**ULA Thinning**

This section shows how to model thinning using tapering. When thinning, each element of the array has a certain probability of being deactivated or removed. The taper values can be either 0, for an inactive element, or 1 for an active element. Here, the probability of keeping the element is proportional to the value of the Taylor window at that element.

```
% Get that previously computed taper values corresponding to a Taylor
% window
taper = taperedULA.Taper;

% Create a random vector uniformly distributed between 0 and 1
randvect =  rand(size(taper));

% Compute the taper values whose probability of being 1 is equal to
% the value of the normalized Taylor window at the corresponding sensor.
thinningTaper = zeros(size(taper));
thinningTaper(randvect<taper/max(taper)) = 1;

% Apply thinning
thinnedULA = clone(ula);
thinnedULA.Taper = thinningTaper;
```

The following plot shows how the thinning taper values are distributed. Notice on the edges when the window level goes down, the number of inactive sensors is up.

```
plot(taper)
hold on
```

```
plot(thinningTaper,'o')
hold off
legend('Taylor window','Thinning taper')
title('Applied Thinning Taper');xlabel('Sensor Position');
ylabel('Taylor Window Values');
```
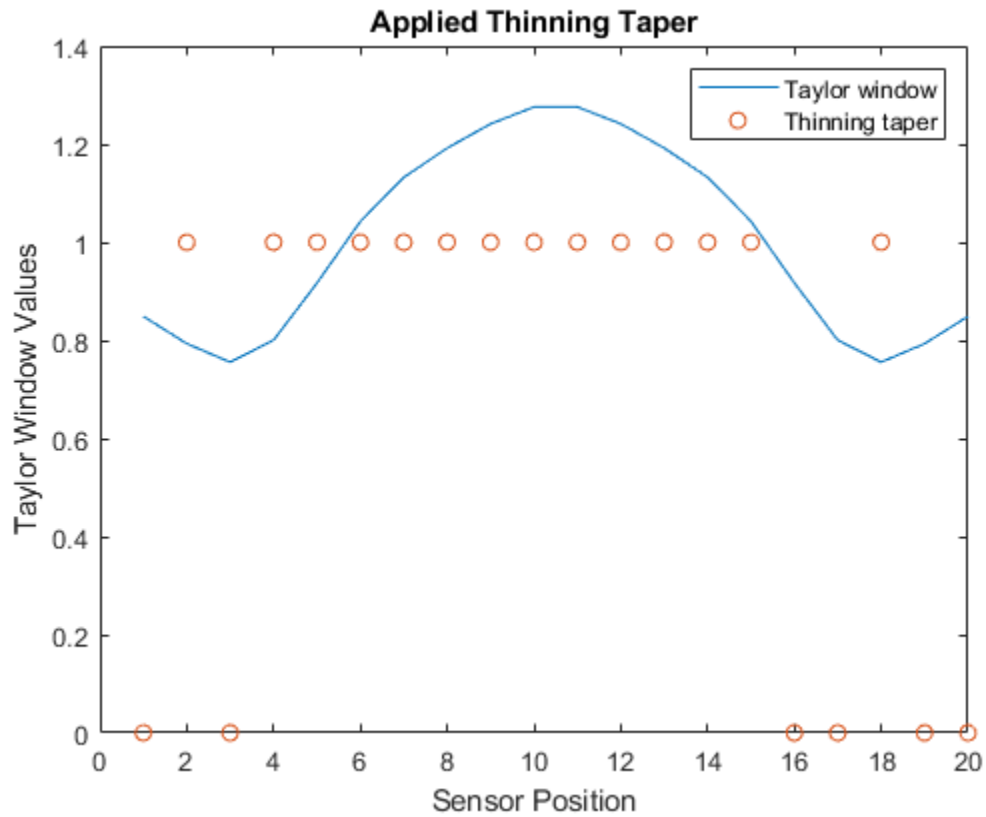


Compare the response of the thinned to the ideal array. Notice how the sidelobes of the thinned ULA are lower.

```
helperCompareResponses(thinnedULA,ula, ...
                        'Ideal ULA versus Thinned ULA response', ...
                        {'Thinned','Not thinned'});
```

**Ideal ULA versus Thinned ULA response**
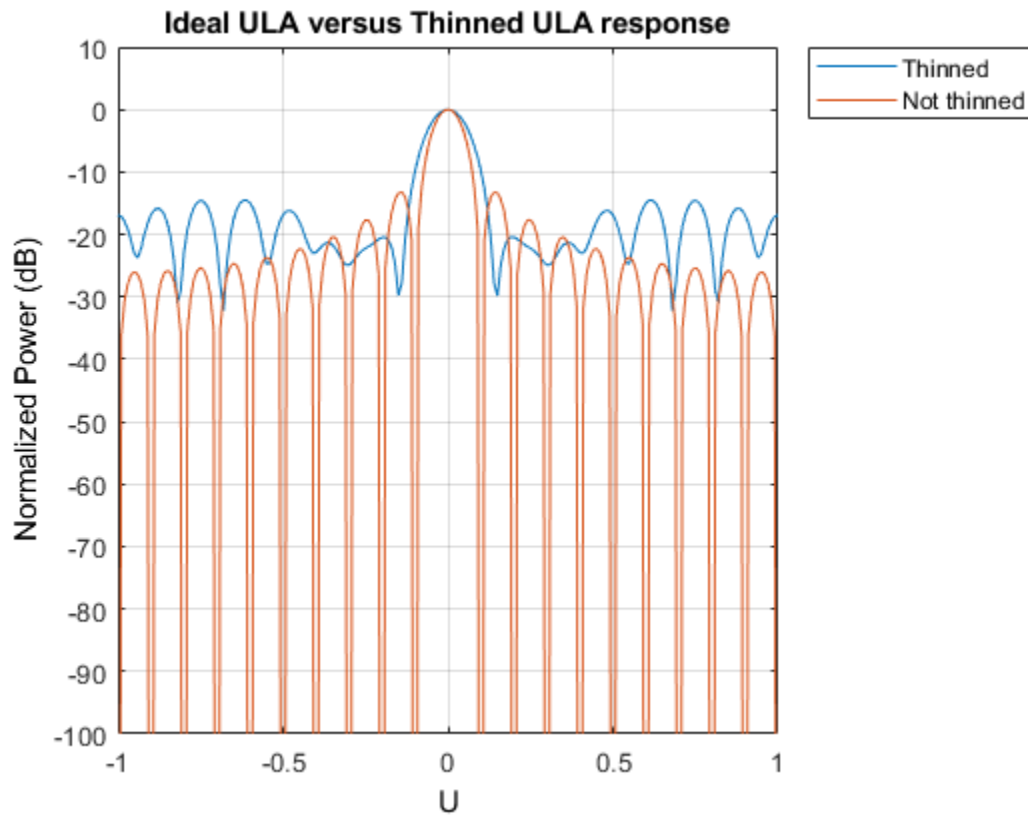


**URA Tapering**

This section shows how to apply a Taylor window along both dimensions of a 13 by 10 uniform rectangular array (URA).

```
uraSize = [13,10];
heterogeneousURA = phased.URA(uraSize);

nbar=2; sll = -20;

% along the z axis
twinz = taylorwin(uraSize(1),nbar,sll);

% along the y axis
twiny = taylorwin(uraSize(2),nbar,sll);

% Get the total taper values by multiplying the vectors of both dimensions
tap = twinz*twiny.';

% Apply the taper
taperedURA = clone(heterogeneousURA);
taperedURA.Taper = tap;
```

View the sensor's color brightness in proportion to the taper magnitudes.

```
viewArray(taperedURA,'Title','Tapered URA','ShowTaper',true);
```

Tapered URA

Aperture Size:
Y axis = 5.0 m
Z axis = 6.5 m
Element Spacing:
$\Delta$ y = 500 mm
$\Delta$ z = 500 mm
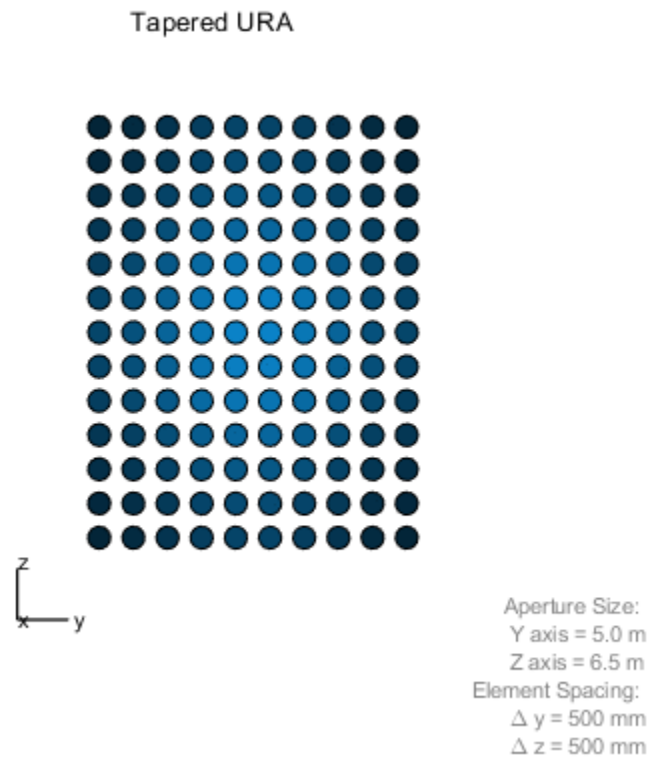
Plot the taper values at each sensor in 3d space.

```
clf
pos = getElementPosition(taperedURA);
plot3(pos(2,:),pos(3,:),taperedURA.Taper(:),'*');
title('Applied Taper');ylabel('Sensor Position');zlabel('Taper Values');
```
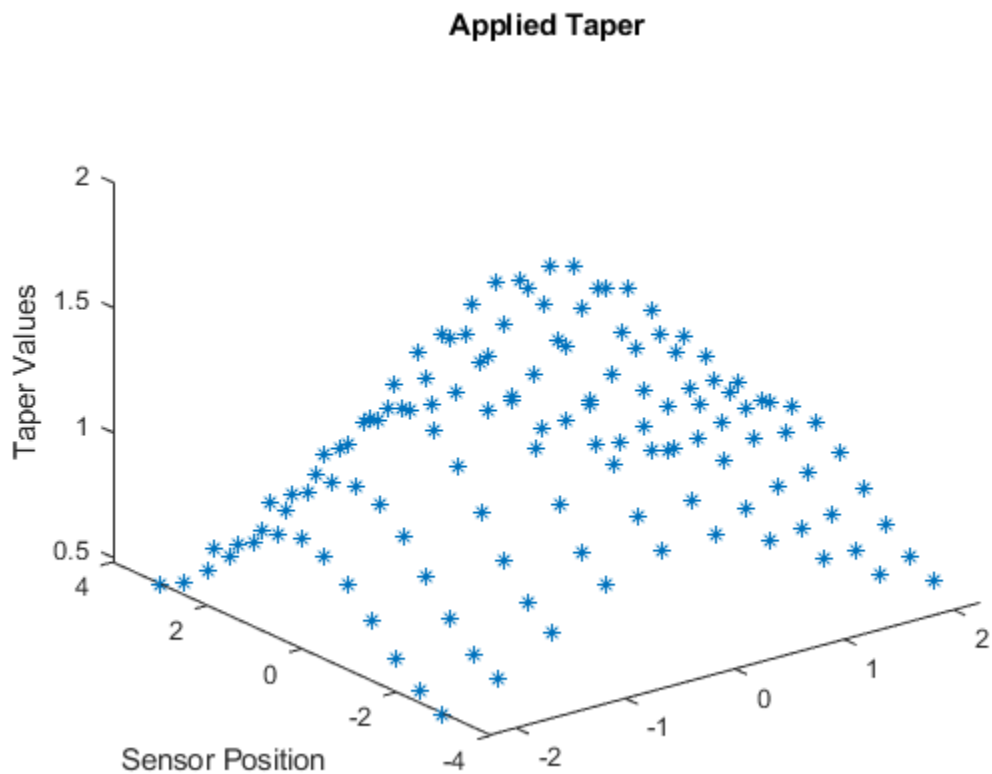
**Applied Taper**



Compare the response of the tapered to the untapered array. Notice how the sidelobes of the tapered URA are lower.

```
helperCompareResponses(heterogeneousURA,taperedURA, ...
                       'Ideal URA versus Tapered URA response', ...
                       {'Not tapered','Tapered'});
```

**Ideal URA versus Tapered URA response**
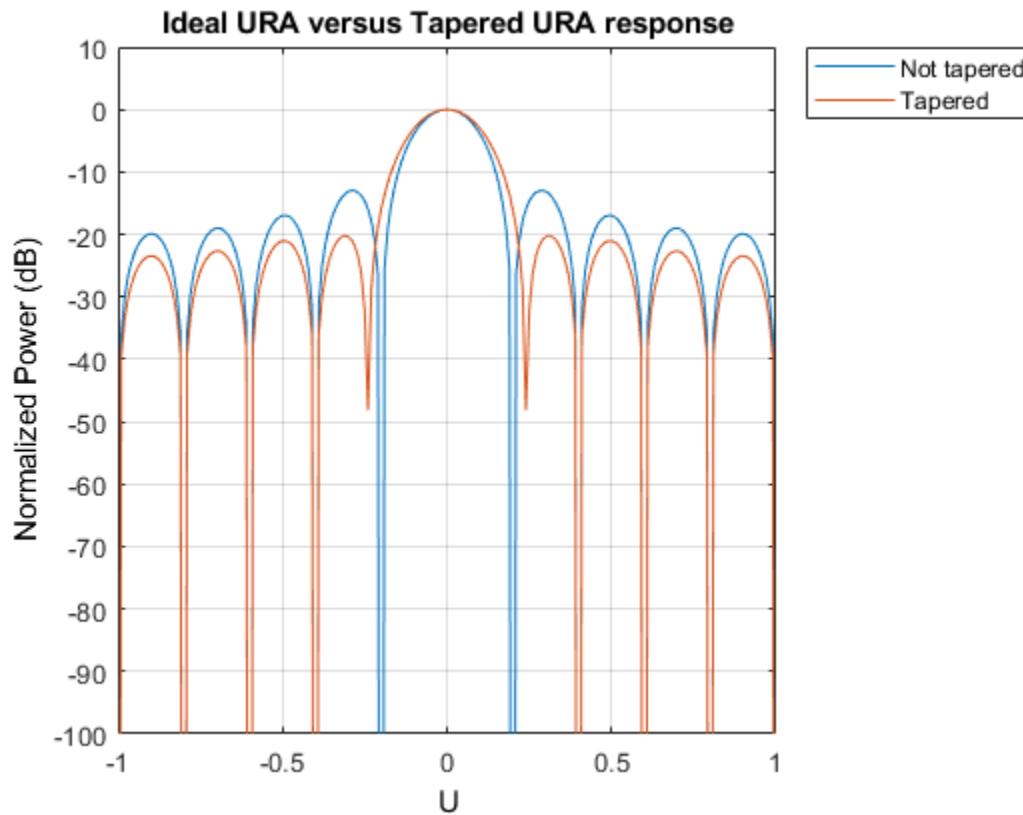
### Circular Planar Tapering

This section shows how to apply a taper on a circular planar array with a radius of 5 meters and distance between elements of 0.5 meters.

```
radius = 5; dist = 0.5;
numElPerSide = radius*2/dist;

% Get the positions of the smallest URA which could fit the circular planar
% array
pos = getElementPosition(phased.URA(numElPerSide,dist));

% Remove all elements in URA which are outside the circle
elemToRemove = sum(pos.^2) > radius^2;
pos(:,elemToRemove) = [];

% Create the circular planar array
circularPlanarArray = phased.ConformalArray('ElementPosition',pos,...
                          'ElementNormal',[0;0]*ones(1,size(pos,2)));
```

Apply a circular Taylor window.

```
taperedCircularPlanarArray = clone(circularPlanarArray);
nbar=3; sll = -25;

taperedCircularPlanarArray.Taper = taylortaperc(pos,2*radius,nbar,sll).';
```

View the array and plot the taper values at each sensor.

```
viewArray(taperedCircularPlanarArray,...
    'Title','Tapered Circular Planar Array','ShowTaper',true)
```



Tapered Circular Planar Array
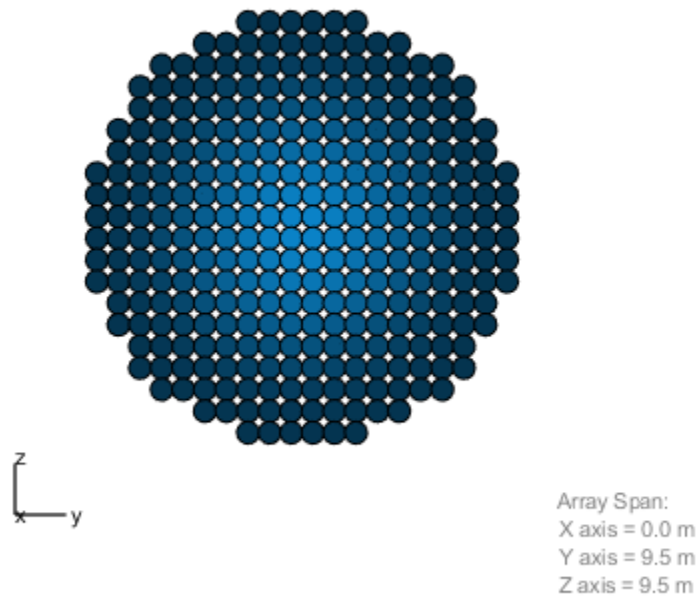
```
clf
plot3(pos(2,:),pos(3,:),taperedCircularPlanarArray.Taper,'*');
title('Applied Taper');ylabel('Sensor Position');zlabel('Taper Values');
```
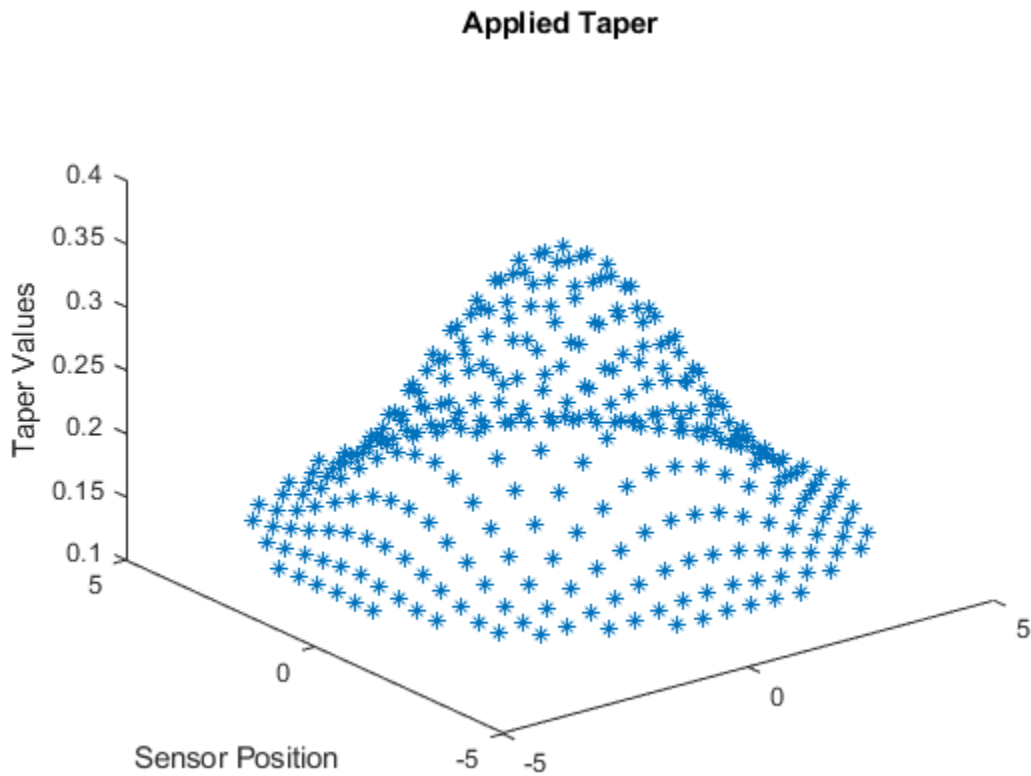
**Applied Taper**



Compare the response of the tapered to the untapered array. Notice how the sidelobes of the tapered array are lower.

```
helperCompareResponses(circularPlanarArray,taperedCircularPlanarArray, ...
                        'Ideal versus Tapered response', ...
                        {'Not tapered','Tapered'});
```
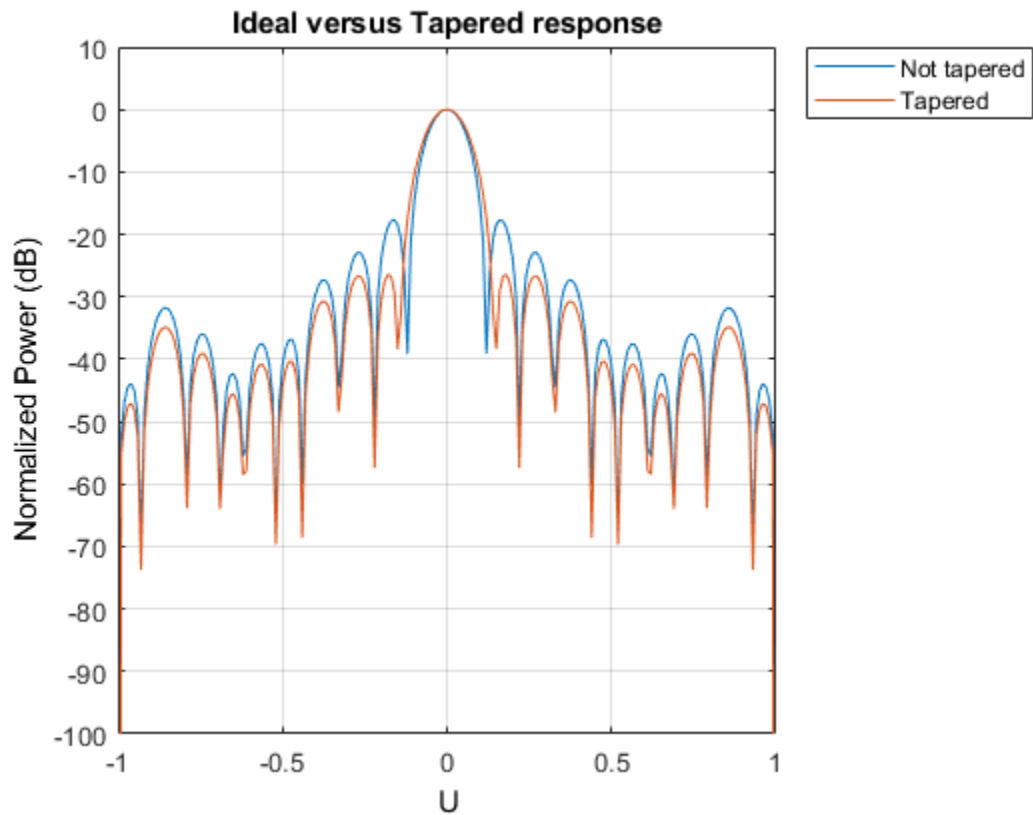
**Circular Planar Thinning**

Calculate the thinning taper values similar to the ULA section.

```
taper = taperedCircularPlanarArray.Taper;
randvect = rand(size(taper));
thinningTaper = zeros(size(taper));
thinningTaper(randvect<taper/max(max(taper))) = 1;

thinnedCircularPlanarArray = clone(circularPlanarArray);
thinnedCircularPlanarArray.Taper = thinningTaper;
```

View the array and compare the response of the thinned to the ideal array.

```
viewArray(thinnedCircularPlanarArray,'ShowTaper',true)
```
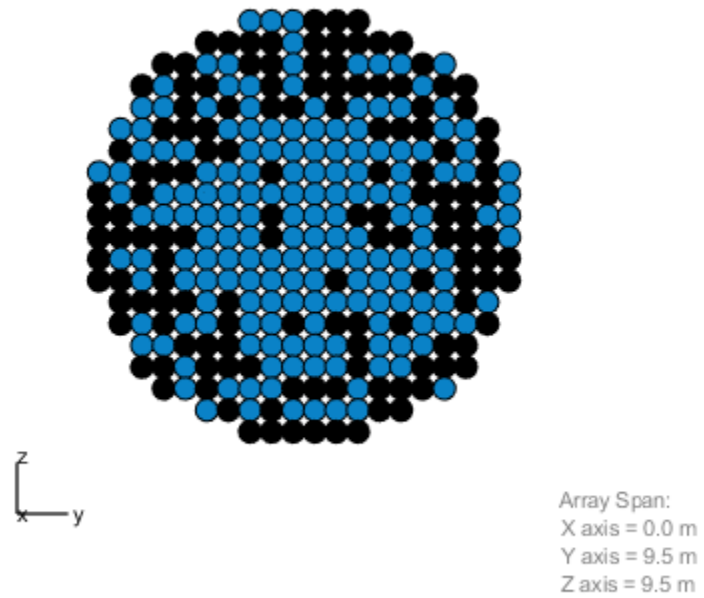
Array Geometry



Array Span:
X axis = 0.0 m
Y axis = 9.5 m
Z axis = 9.5 m

```
clf;
helperCompareResponses(circularPlanarArray,thinnedCircularPlanarArray, ...
                       'Ideal versus Thinned response', ...
                       {'Not thinned','Thinned'});
```

**Ideal versus Thinned response**
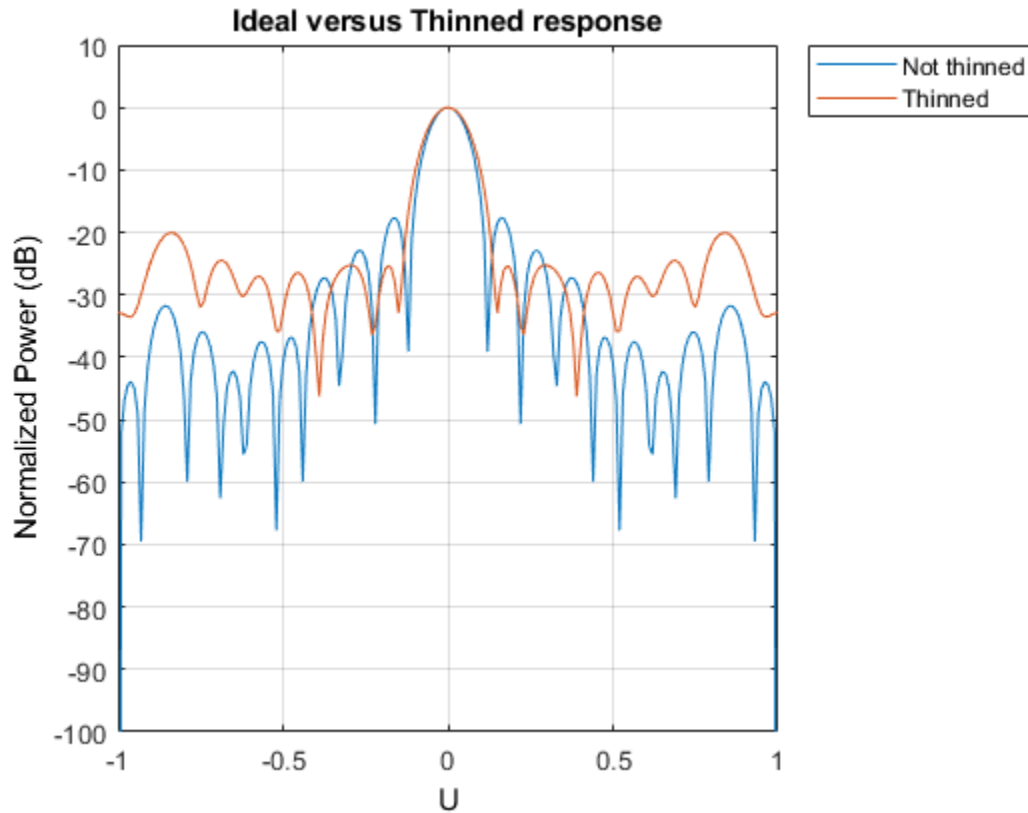


**Multiple Element Patterns in URA**

This section shows how to create a 13 by 10 URA with sensor patterns on the edges and corners different than the patterns of the remaining sensors. This ability could be used to model coupling effects.

Create three different cosine patterns with the following azimuth and elevation cosine exponents [azim exponent, elev exponent]: [2, 2] for the edges, [4, 4] for the corners, and [1.5, 1.5] for the main sensors.

```
mainAntenna = phased.CosineAntennaElement('CosinePower',[1.5 1.5]);
edgeAntenna = phased.CosineAntennaElement('CosinePower',[2 2]);
cornerAntenna = phased.CosineAntennaElement('CosinePower',[4 4]);
```

Map the sensors to the patterns.

```
uraSize = [13,10];

% Create a cell array which includes all the patterns
patterns = {mainAntenna, edgeAntenna, cornerAntenna};

% Initialize all sensors to first pattern.
patternMap = ones(uraSize);

% Set the edges to the second pattern.
patternMap([1 end],2:end-1) = 2;
patternMap(2:end-1,[1 end]) = 2;
```

```
% Set the corners to the third pattern.
patternMap([1 end],[1 end]) = 3;

% Create the URA

heterogeneousURA = phased.HeterogeneousURA('ElementSet' , patterns, ...
                              'ElementIndices', patternMap);
```

View the pattern layout in the array.

```
helperViewPatternArray(heterogeneousURA);
```
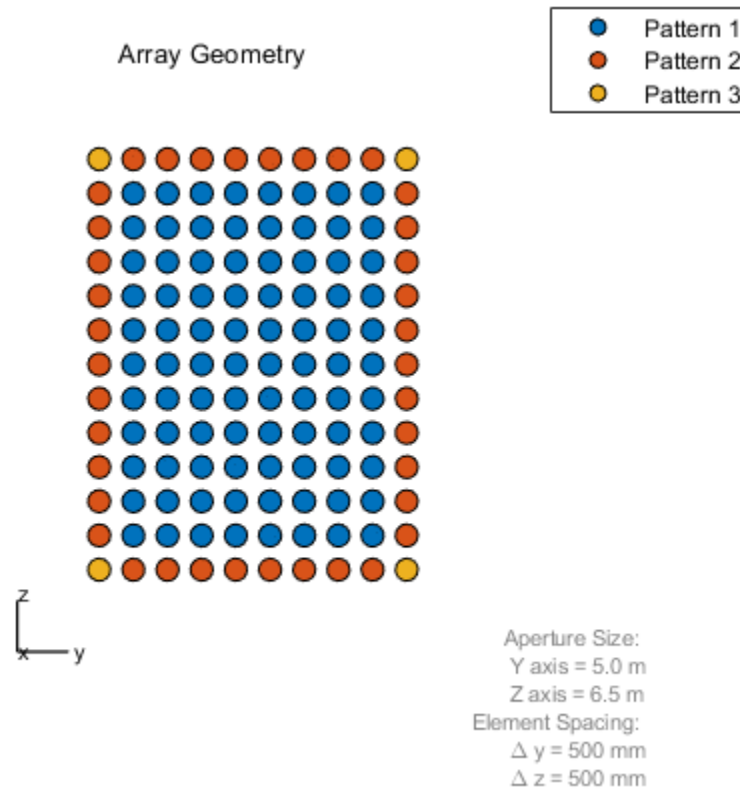


Compare the response of the multiple pattern array to the single pattern array.

```
clf;
helperCompareResponses( heterogeneousURA, ...
                        phased.URA(uraSize,'Element',mainAntenna), ...
                        'Multiple versus single pattern response', ...
                        {'Single Pattern','Multiple Patterns'});
```
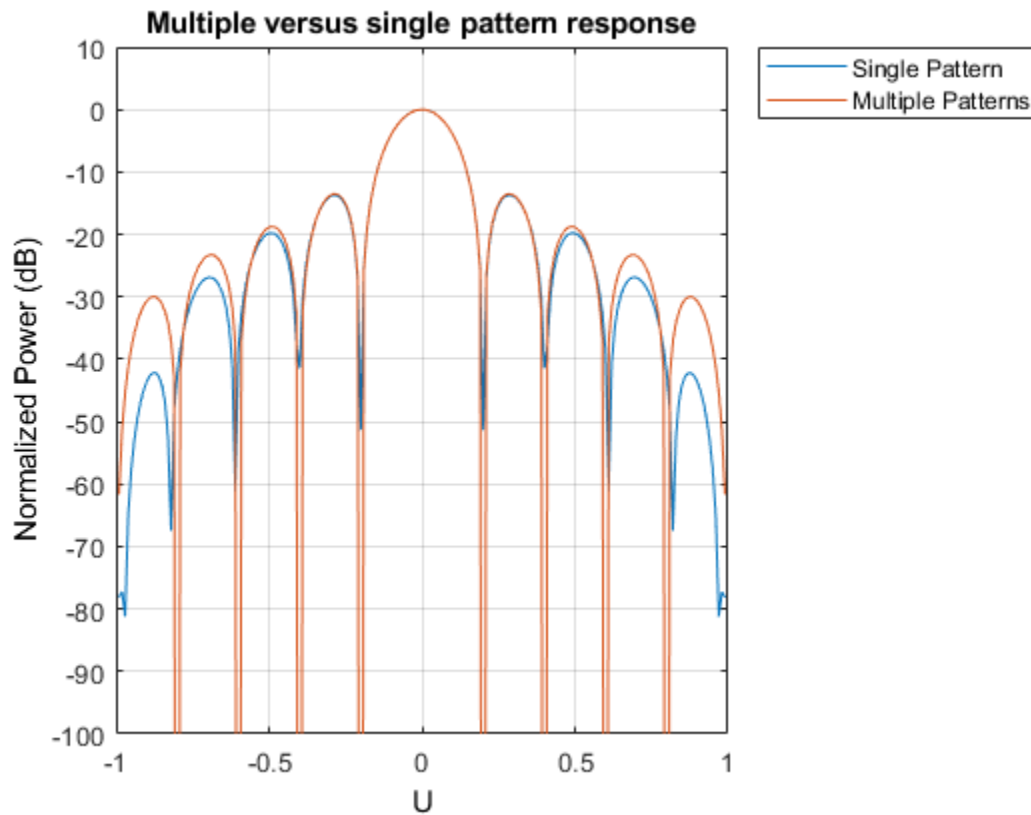
**Multiple Element Patterns in Circular Planar Arrays**

This section shows how to set the pattern of sensors located more than 4 meters from the center of the array.

```
% Create a cell array which includes all the patterns
patterns = {mainAntenna, edgeAntenna};

% Get positions
pos = getElementPosition(circularPlanarArray);

% Initialize all sensors to first pattern in sensorPatterns.
patternMap = ones(1,size(pos,2));

% Get the indexed of the sensors more than 4 meters away from the center.
sensorIdx = find(sum(pos.^2) > 4^2);

% Set the edges to the second pattern in sensorPatterns.
patternMap(sensorIdx) = 2;

% Set the corresponding properties
heterogeneousCircularPlanarArray = ...
    phased.HeterogeneousConformalArray('ElementPosition',pos,...
                            'ElementNormal',[1;0]*ones(1,size(pos,2)),...
                            'ElementSet' , patterns, ...
                            'ElementIndices', patternMap);
```

View the pattern layout in the array.

```
helperViewPatternArray(heterogeneousCircularPlanarArray);
```
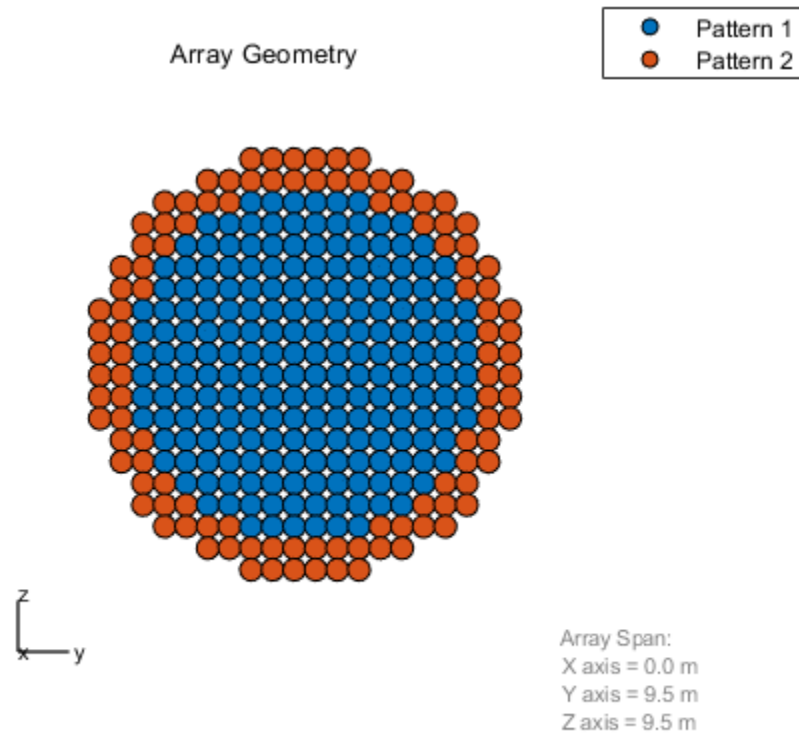


Compare the response of the multiple pattern array to the single pattern array.

```
clf;
helperCompareResponses(circularPlanarArray,...
    heterogeneousCircularPlanarArray,...
    'Multiple versus single pattern response',...
    {'Single Pattern','Multiple Patterns'});

% reset the random seed
rng(rs)
```

**Multiple versus single pattern response**
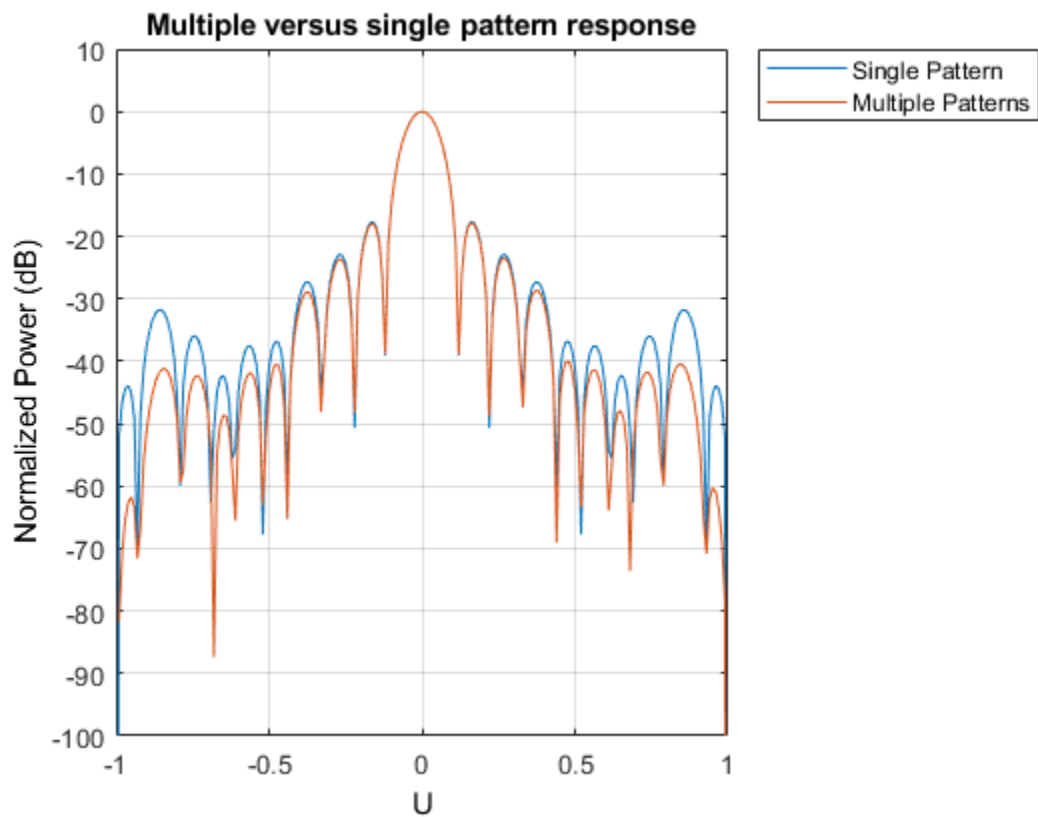


**Summary**

This example demonstrated how to apply taper values and model thinning using taper values for different array configurations. It also showed how to create arrays with different element patterns.

# Simultaneous Range and Speed Estimation Using MFSK Waveform

This example compares triangle sweep frequency-modulated continuous (FMCW) and multiple frequency-shift keying (MFSK) waveforms used for simultaneous range and speed estimation for multiple targets. The MFSK waveform is specifically designed for automotive radar systems used in advanced driver assistance systems (ADAS). It is particularly appealing in multitarget scenarios because it does not introduce ghost targets.

**Triangle Sweep FMCW Waveform**

In the "Automotive Adaptive Cruise Control Using FMCW Technology" (Radar Toolbox) example an automotive radar system is designed to perform range estimation for an automatic cruise control system. In the latter part of that example, a triangle sweep FMCW waveform is used to estimate the range and speed of the target vehicle simultaneously.

Although the triangle sweep FMCW waveform elegantly solves the range-Doppler coupling issue for a single target, its processing becomes complicated in multitarget situations. The next section shows how a triangle sweep FMCW waveform behaves when two targets are present.

The scene includes a car 50 m away from the radar, traveling at 96 km/h along the same direction as the radar, and a truck at 55 m away, traveling at 70 km/h in the opposite direction. The radar itself is traveling at 60 km/h.

```
rng(2015);
[fmcwwaveform,target,tgtmotion,channel,transmitter,receiver, ...
    sensormotion,c,fc,lambda,fs,maxbeatfreq] = helperMFSKSystemSetup;
```

Next, simulate the radar echo from the two vehicles. The FMCW waveform has a sweep bandwidth of 150 MHz so the range resolution is 1 meter. Each up or down sweep takes 1 millisecond, so each triangle sweep takes 2 milliseconds. Note that only one triangle sweep is needed to perform the joint range and speed estimation.

```
Nsweep = 2;
xr = helperFMCWSimulate(Nsweep,fmcwwaveform,sensormotion,tgtmotion, ...
    transmitter,channel,target,receiver);
```

Although the system needs a 150 MHz bandwidth, the maximum beat frequency is much less. This means that at the processing side, one can decimate the signal to a lower frequency to ease the hardware requirements. The beat frequencies are then estimated using the decimated signal.

```
dfactor = ceil(fs/maxbeatfreq)/2;
fs_d = fs/dfactor;
fbu_rng = rootmusic(decimate(xr(:,1),dfactor),2,fs_d);
fbd_rng = rootmusic(decimate(xr(:,2),dfactor),2,fs_d);
```

Now there are two beat frequencies from the up sweep and two beat frequencies from the down sweeps. Since any pair of beat frequencies from an up sweep and a down sweep can define a target, there are four possible combinations of range and Doppler estimates, yet only two of them are associated with the real targets.

```
sweep_slope = fmcwwaveform.SweepBandwidth/fmcwwaveform.SweepTime;
rng_est = beat2range([fbu_rng fbd_rng;fbu_rng flipud(fbd_rng)], ...
    sweep_slope,c)
```

```
rng_est = 4×1

    49.9802
    54.9406
    64.2998
    40.6210
```

The remaining two are what is often referred to as ghost targets. The relationship between real targets and ghost targets can be better explained using time-frequency representation.
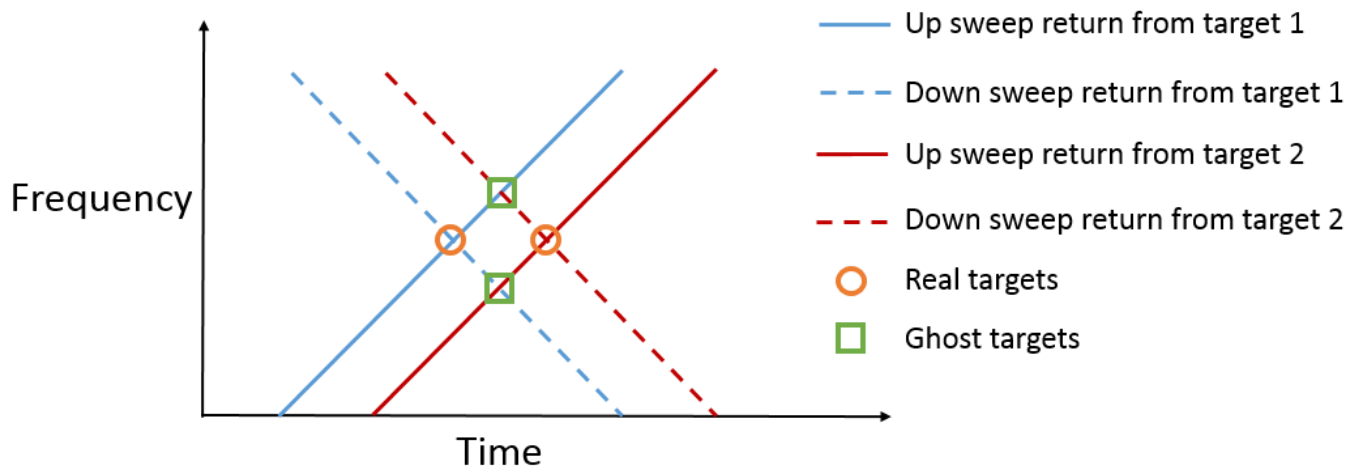


As shown in the figure, each intersection of an up sweep return and a down sweep return indicates a possible target. So it is critical to distinguish between the true targets and the ghost targets. To solve this ambiguity, one can transmit additional FMCW signals with different sweep slopes. Since only the true targets will occupy the same intersection in the time-frequency domain, the ambiguity is resolved. However, this approach significantly increases the processing complexity as well as the processing time needed to obtain the valid estimates.

**MFSK Waveform**

Multiple frequency shift keying (MFSK) waveform [1] on page 18-154 is designed for the automotive radar to achieve simultaneous range and Doppler estimation under the multiple targets situation without falling into the trap of ghost targets. Its time-frequency representation is shown in the following figure.
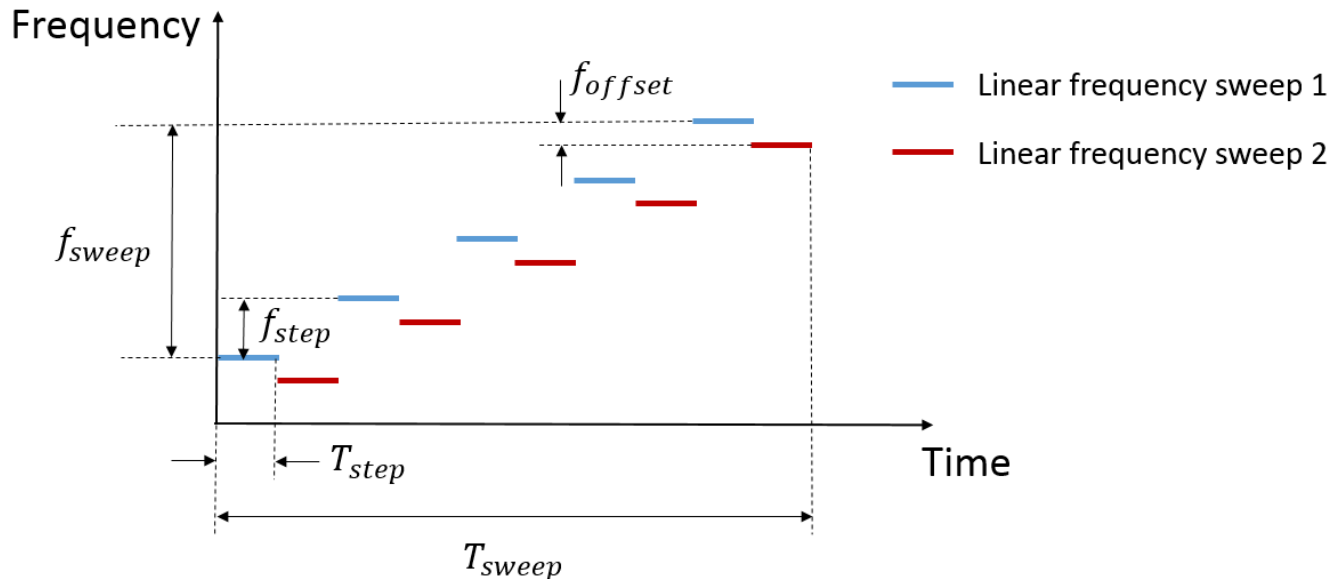
The figure indicates that the MFSK waveform is a combination of two linear FMCW waveforms with a fixed frequency offset. Unlike the regular FMCW waveforms, MFSK sweeps the entire bandwidth at discrete steps. Within each step, a single frequency continuous wave signal is transmitted. Because there are two tones within each step, it can be considered as a frequency shift keying (FSK) waveform. Thus, there is one set of range and Doppler relation from FMCW waveform and another set of range and Doppler relation from FSK. Combining two sets of relations together can help resolve the coupling between range and Doppler regardless of the number of targets present in the scene.

The following sections simulate the previous example, but use an MFSK waveform instead.

**End-to-end Radar System Simulation Using MFSK Waveform**

First, parameterize the MFSK waveform to satisfy the system requirement specified in [1] on page 18-154. Because the range resolution is 1 m, the sweep bandwidth is set at 150 MHz. In addition, the frequency offset is set at —294 kHz as specified in [1] on page 18-154. Each step lasts about 2 microseconds and the entire sweep has 1024 steps. Thus, each FMCW sweep takes 512 steps and the total sweep time is a little over 2 ms. Note that the sweep time is comparable to the FMCW signal used in previous sections.

```
mfskwaveform = phased.MFSKWaveform( ...
    'SampleRate',151e6, ...
    'SweepBandwidth',150e6, ...
    'StepTime',2e-6, ...
    'StepsPerSweep',1024, ...
    'FrequencyOffset',-294e3, ...
    'OutputFormat','Sweeps', ...
    'NumSweeps',1);
```

The figure below shows the spectrogram of the waveform. It is zoomed into a small interval to better reveal the time-frequency characteristics of the waveform.

```
numsamp_step = round(mfskwaveform.SampleRate*mfskwaveform.StepTime);
sig_display = mfskwaveform();
```

```
spectrogram(sig_display(1:8192),kaiser(3*numsamp_step,100), ...
    ceil(2*numsamp_step),linspace(0,4e6,2048),mfskwaveform.SampleRate, ...
    'yaxis','reassigned','minthreshold',-60)
```
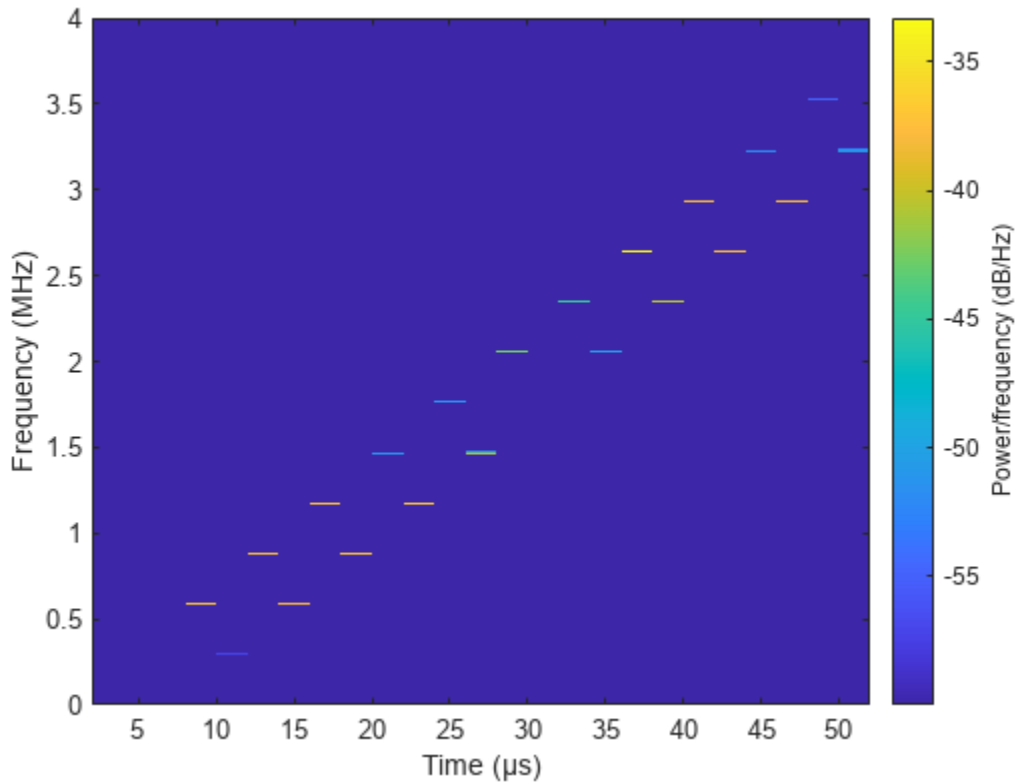


Next, simulate the return of the system. Again, only one sweep is needed to estimate the range and Doppler.

```
Nsweep = 1;
release(channel);
channel.SampleRate = mfskwaveform.SampleRate;
release(receiver);
receiver.SampleRate = mfskwaveform.SampleRate;
xr = helperFMCWSimulate(Nsweep,mfskwaveform,sensormotion,tgtmotion, ...
    transmitter,channel,target,receiver);
```

The subsequent processing samples the return echo at the end of each step and group the sampled signals into two sequences corresponding to two sweeps. Note that the sampling frequency of the resulting sequence is now proportional to the time at each step, which is much less compared the original sample rate.

```
x_dechirp = reshape(xr(numsamp_step:numsamp_step:end),2,[]).';
fs_dechirp = 1/(2*mfskwaveform.StepTime);
```

As in the case of FMCW signals, the MFSK waveform is processed in the frequency domain. The following figures show the frequency spectrums of the received echoes corresponding to the two sweeps.

```
xf_dechirp = fft(x_dechirp);
num_xf_samp = size(xf_dechirp,1);
beatfreq_vec = (0:num_xf_samp-1).'/num_xf_samp*fs_dechirp;
clf
subplot(211),plot(beatfreq_vec/1e3,abs(xf_dechirp(:,1)))
grid on
ylabel('Magnitude')
title('Frequency spectrum for sweep 1')
subplot(212),plot(beatfreq_vec/1e3,abs(xf_dechirp(:,2)))
grid on
ylabel('Magnitude')
title('Frequency spectrum for sweep 2')
xlabel('Frequency (kHz)')
```
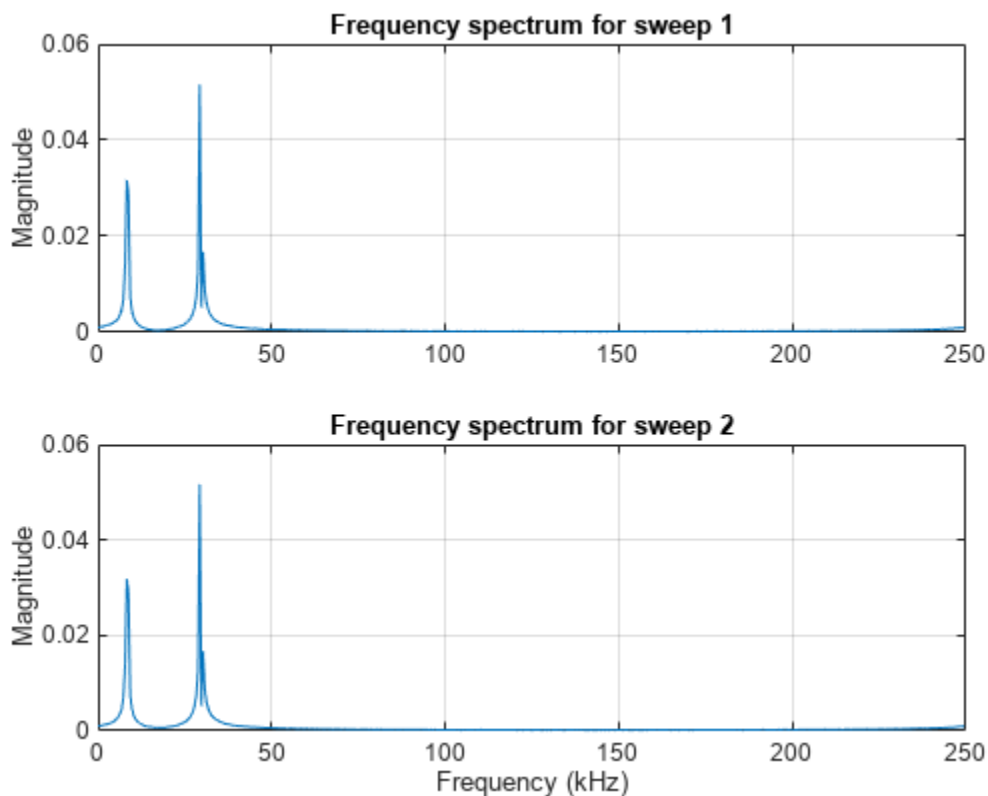


Note that there are two peaks in each frequency spectrum indicating two targets. In addition, the peaks are at the same locations in both returns so there are no ghost targets.

To detect the peaks, one can use a CFAR detector. Once detected, the beat frequencies as well as the phase differences between two spectra are computed at the peak locations.

```
cfar = phased.CFARDetector('ProbabilityFalseAlarm',1e-2, ...
    'NumTrainingCells',8);
peakidx = cfar(abs(xf_dechirp(:,1)),1:num_xf_samp);
Fbeat = beatfreq_vec(peakidx);
phi = angle(xf_dechirp(peakidx,2))-angle(xf_dechirp(peakidx,1));
```

Finally, the beat frequencies and phase differences are used to estimate the range and speed. Depending on how you construct the phase difference, the equations are slightly different. For the

approach shown in this example, it can be shown that the range and speed satisfy the following relation:

$$f_b = -\frac{2v}{\lambda} + \frac{2\beta R}{c}$$

$$\Delta\phi = -\frac{4\pi T_s v}{\lambda} + \frac{4\pi f_{offset} R}{c}$$

where $f_b$ is the beat frequency, $\Delta\phi$ is the phase difference, $\lambda$ is the wavelength, $c$ is the propagation speed, $T_s$ is the step time, $f_{offset}$ is the frequency offset, $\beta$ is the sweep slope, $R$ is the range, and $v$ is the speed. Based on the equation, the range and speed are estimated as follows.

```
sweep_slope = mfskwaveform.SweepBandwidth/ ...
    (mfskwaveform.StepsPerSweep*mfskwaveform.StepTime);
temp = ...
    [-2/lambda 2*sweep_slope/c;-2/lambda*mfskwaveform.StepTime 2*mfskwaveform.FrequencyOffset/c]\
    [Fbeat phi/(2*pi)].';
r_est = temp(2,:)

r_est = 1×2

   54.8564   49.6452


v_est = temp(1,:)

v_est = 1×2

   36.0089   -9.8495
```

The estimated range and speed match the true range and speed values very well.

- Car: r = 50 m, v = -10 m/s
- Truck: r = 55 m, v = 36 m/s

**Summary**

This example shows two simultaneous range and speed estimation approaches, using either a triangle sweep FMCW waveform or an MFSK waveform. This example shows that the MFSK waveform has an advantage over the FMCW waveform when multiple targets are present because it does not introduce ghost targets during the processing.

**References**

[1] Rohling, H. and M. Meinecke. *Waveform Design Principle for Automotive Radar Systems*, Proceedings of CIE International Conference on Radar, 2001.

# Waveform Analysis Using the Ambiguity Function

This example shows how to use the ambiguity function to analyze waveforms. It compares range and Doppler capabilities of several common pulsed waveforms such as the rectangular waveform, the linear frequency modulated (FM) waveform, and the stepped FM waveform. It also shows ambiguity functions for several nonlinear FM waveforms and a phase coded waveform.

In radar and sonar systems, the choice of a waveform plays an important role in enabling the system to separate two closely located targets, in either range or speed. Therefore, it is often necessary to examine a waveform and understand its resolution and ambiguity in both range and speed domains. The delay is used to measure range while the Doppler shift is used to measure speed. Thus, the range and the speed are used interchangeably with the delay and the Doppler.

**Introduction**

To improve the signal to noise ratio (SNR), modern radar and sonar systems often employ the matched filter in the receiver chain. The ambiguity function of a waveform represents exactly the output of the matched filter when the specified waveform is used as the filter input. This exact representation makes the ambiguity function a popular tool for designing and analyzing waveforms. This approach provides an insight into the resolution capability in both delay and Doppler domains for a given waveform. Based on this analysis, one can then determine whether a waveform is suitable for a particular application.

The following sections use the ambiguity function to explore the range-Doppler relationship for several popular waveforms. To establish a comparison baseline, assume that the design specification of a radar system requires a maximum unambiguous range of 15 km and a range resolution of 150 m.

```
Rmax = 15e3;
Rres = 150;
```

Based on these design specifications, the pulse repetition frequency (PRF) and the bandwidth of the waveform can be computed as follows.

```
prf = 1/range2time(Rmax)
```

```
prf = 9.9931e+03
```

```
bw = rangeres2bw(Rres)
```

```
bw = 9.9931e+05
```

**Rectangular Pulse Waveform**

The simplest waveform is a rectangular waveform, sometimes also referred to as *single frequency waveform*. For the rectangular waveform, the pulse width is the reciprocal of the bandwidth.

A rectangular waveform can be created as follows.

```
tau = 1/bw;      % Pulse width
fs = 4*bw;       % Set the sampling frequency to a high value to generate smoother plots
rectwav = phased.RectangularWaveform('SampleRate',fs,'PRF',prf,'PulseWidth',tau)
```

```
rectwav =
  phased.RectangularWaveform with properties:
```

```
              SampleRate: 3.9972e+06
    DurationSpecification: 'Pulse width'
               PulseWidth: 1.0007e-06
                      PRF: 9.9931e+03
    PRFSelectionInputPort: false
    FrequencyOffsetSource: 'Property'
          FrequencyOffset: 0
             OutputFormat: 'Pulses'
                NumPulses: 1
            PRFOutputPort: false
    CoefficientsOutputPort: false
```

Because the analysis of a waveform is always performed on full pulses, keep the `OutputFormat` property as `'Pulses'`. One can also check the bandwidth of the waveform using the bandwidth method.

```
rect_bw = bandwidth(rectwav)
```

```
rect_bw = 9.9931e+05
```

The resulting bandwidth matches the requirement. Now, generate one pulse of the waveform, and then examine it using the ambiguity function.

```
rectsamples = rectwav();
ambgfun(rectsamples,rectwav.SampleRate,rectwav.PRF)
xlim([-tau tau]*2e+6)
```
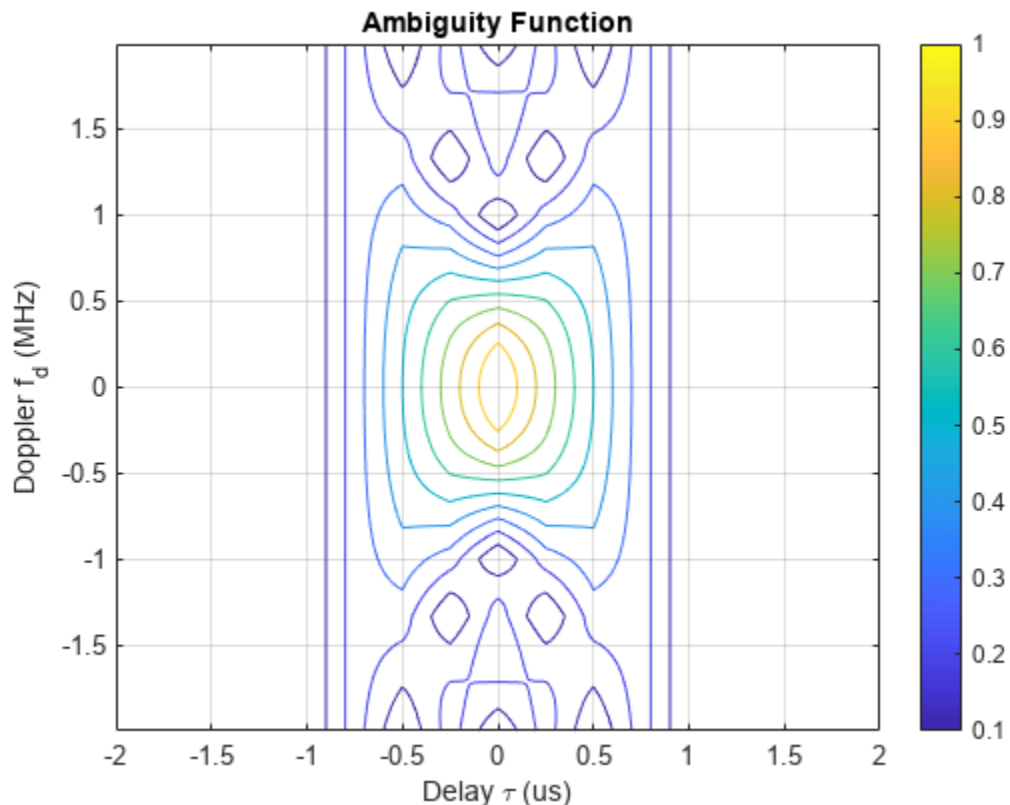
The nonzero response occupies only a small fraction of all delays focusing in a narrow strip around zero delay. This occurs because the waveform has a duty cycle of 0.01.

```
rect_dc = dutycycle(rectwav.PulseWidth,rectwav.PRF)
```

```
rect_dc = 0.0100
```

When investigating a waveform's resolution capability, the zero-delay cut and the zero-Doppler cut of the waveform ambiguity function are often of interest.

The zero-Doppler cut of the ambiguity function returns the auto-correlation function (ACF) of the rectangular waveform. The cut can be plotted using the following command.

```
ambgfun(rectsamples,rectwav.SampleRate,rectwav.PRF,'Cut','Doppler')
xlim([-tau tau]*2e+6)
```
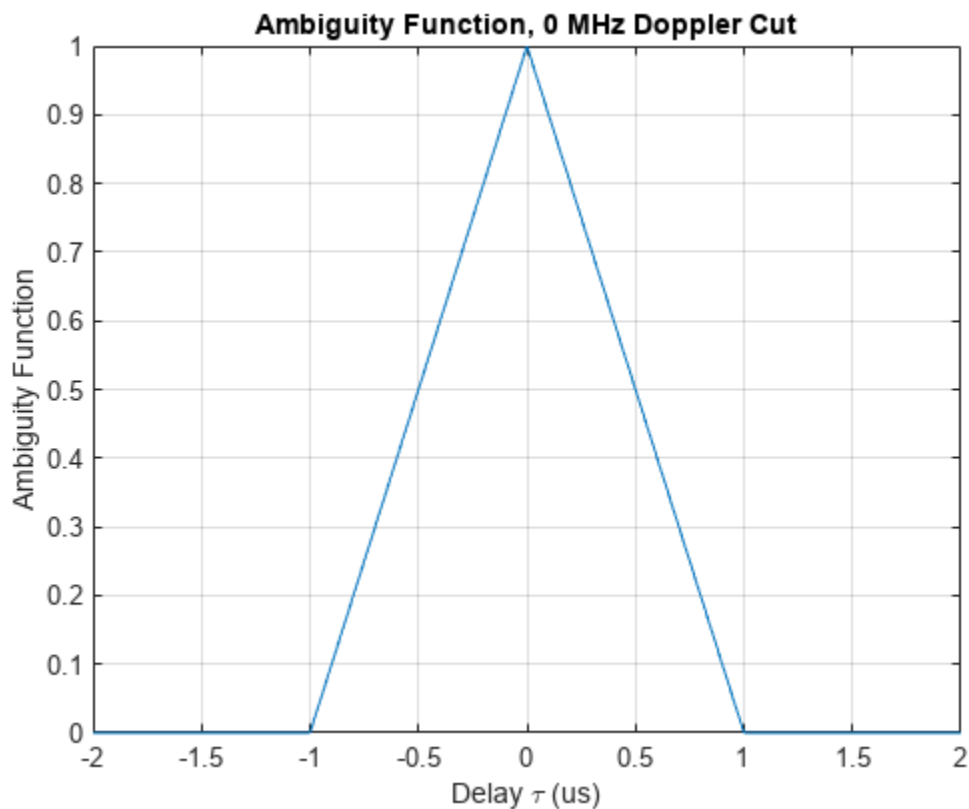


The zero-Doppler cut of the ambiguity function depicts the matched filter response of a target when the target is stationary. From the plot, one can see that the first null response appears at 1 microsecond, which means that this waveform is guaranteed to be able to resolve two targets that are at least 1 microsecond, or 150 m apart. Hence, the response matches the requirement in the design specification.

The zero-delay cut can be plotted using a similar syntax.

```
ambgfun(rectsamples,rectwav.SampleRate,rectwav.PRF,'Cut','Delay')
xlim([-bw bw]*2e-6)
```
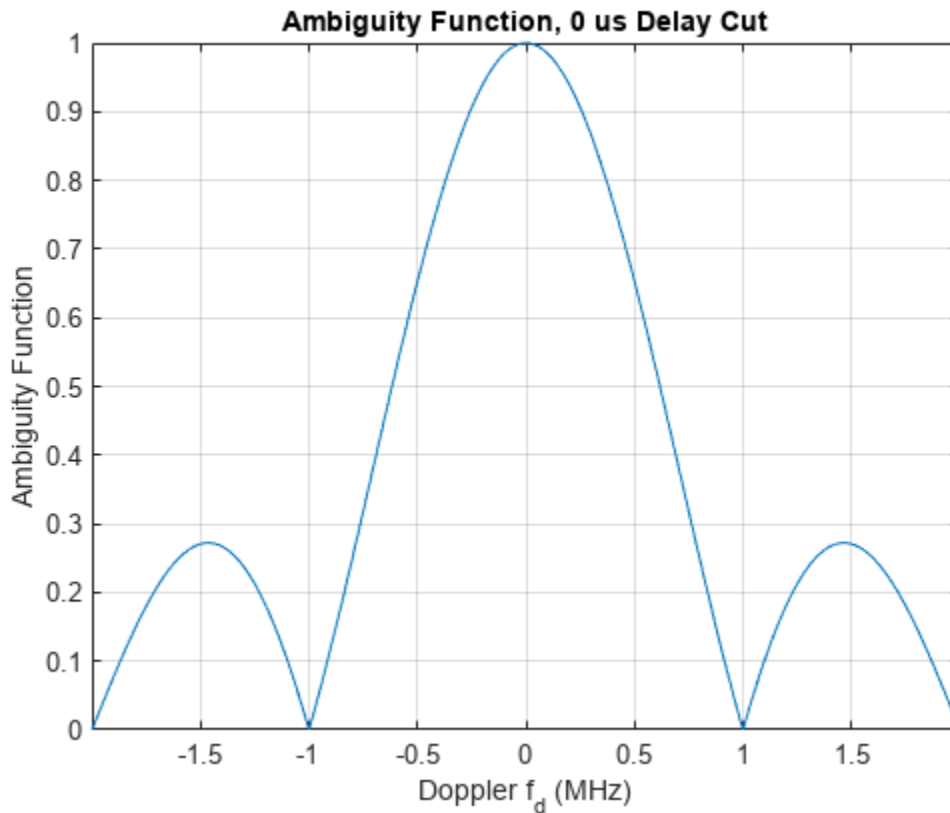
Notice that the returned zero-delay response is fairly broad. The first null appears at a Doppler shift of 1 MHz. Thus, if the two targets are at the same range, they need to have a difference of 1 MHz in the Doppler domain to be separated. Assuming the radar is working at 1 GHz, according to the computation below, such a separation corresponds to a speed difference of almost 150 km/s. Because this number is so large, essentially one cannot separate two targets in the Doppler domain using this system.

```
fc = 1e9;
rect_deltav = dop2speed(1e6,freq2wavelen(fc))/2
```

rect_deltav = 1.4990e+05

Another issue with the rectangular waveform is that the range resolution is determined by the pulse width. Thus, to achieve good range resolution, the system needs to adopt a very small pulse width. At the same time, the system also needs to be able to send out enough energy to the space so that the returned echo can be reliably detected. Hence, a narrow pulse width requires very high peak power at the transmitter. In practice, producing such power can be very costly.

**Linear FM Pulse Waveform**

One can see from the previous section that a single rectangular pulse has poor Doppler resolution. In fact, the Doppler resolution of a single rectangular pulse is given by the reciprocal of the pulse width. At the same time, the range resolution of a rectangular waveform is proportional to the pulse width. Apparently, there exists a conflict between range and Doppler resolutions of a rectangular waveform.

The root issue here is that both the delay and the Doppler resolution depend on the pulse width in opposite ways. Therefore, one way to solve this issue is to come up with a waveform that decouples this dependency. One can then improve the resolution in both domains simultaneously.

Linear FM waveform is just such a waveform. The range resolution of a linear FM waveform no longer depends on the pulse width. Instead, the range resolution is determined by the sweep bandwidth.

A system that transmits a linear FM waveform can afford a longer pulse width, which alleviates the power requirement. The longer pulse width also improves the Doppler resolution. This improvement occurs even though the Doppler resolution of a linear FM waveform is still given by the reciprocal of the pulse width.

Now, explore the linear FM waveform in detail. The linear FM waveform that provides the desired range resolution can be constructed as follows.

```
tau_fm = 50/bw;
lfmwav = phased.LinearFMWaveform('SampleRate',fs,'SweepBandwidth',bw,'PRF',prf,'PulseWidth',tau_
```
```
lfmwav =
  phased.LinearFMWaveform with properties:

               SampleRate: 3.9972e+06
       DurationSpecification: 'Pulse width'
               PulseWidth: 5.0035e-05
                      PRF: 9.9931e+03
       PRFSelectionInputPort: false
           SweepBandwidth: 9.9931e+05
           SweepDirection: 'Up'
            SweepInterval: 'Positive'
                 Envelope: 'Rectangular'
      FrequencyOffsetSource: 'Property'
          FrequencyOffset: 0
             OutputFormat: 'Pulses'
                 NumPulses: 1
            PRFOutputPort: false
       CoefficientsOutputPort: false
```

The pulse width of this linear FM waveform is 50 times longer than that of the rectangular waveform used in the earlier sections of this example. This means that the duty cycle is 50 times larger.

```
lfm_dc = dutycycle(lfmwav.PulseWidth,lfmwav.PRF)
```
```
lfm_dc = 0.5000
```

Notice that the bandwidth of the linear FM waveform is the same as the rectangular waveform.

```
lfm_bw = bandwidth(lfmwav)
```
```
lfm_bw = 9.9931e+05
```

The zero-Doppler cut of the ambiguity function for the linear FM waveform appears in the next plot. To better visualize the sidelobe behavior the ambiguity function is plotted in dB scale.

```
lfmsamples = lfmwav();
[lfmaf_delay,delay] = ambgfun(lfmsamples,lfmwav.SampleRate,lfmwav.PRF,'Cut','Doppler');
```

**18-159**

```
figure
hold on
plot(delay*1e6,mag2db(lfmaf_delay))
yline(-13.2,'Label','Sidelobe Level (-13.2 dB)')
grid on
xlabel('Delay \tau (us)')
ylabel('Ambiguity Function (dB)')
xlim([-tau_fm tau_fm]*3e5)
ylim([-50 0])
title('Ambiguity Function, 0 MHz Doppler Cut')
```
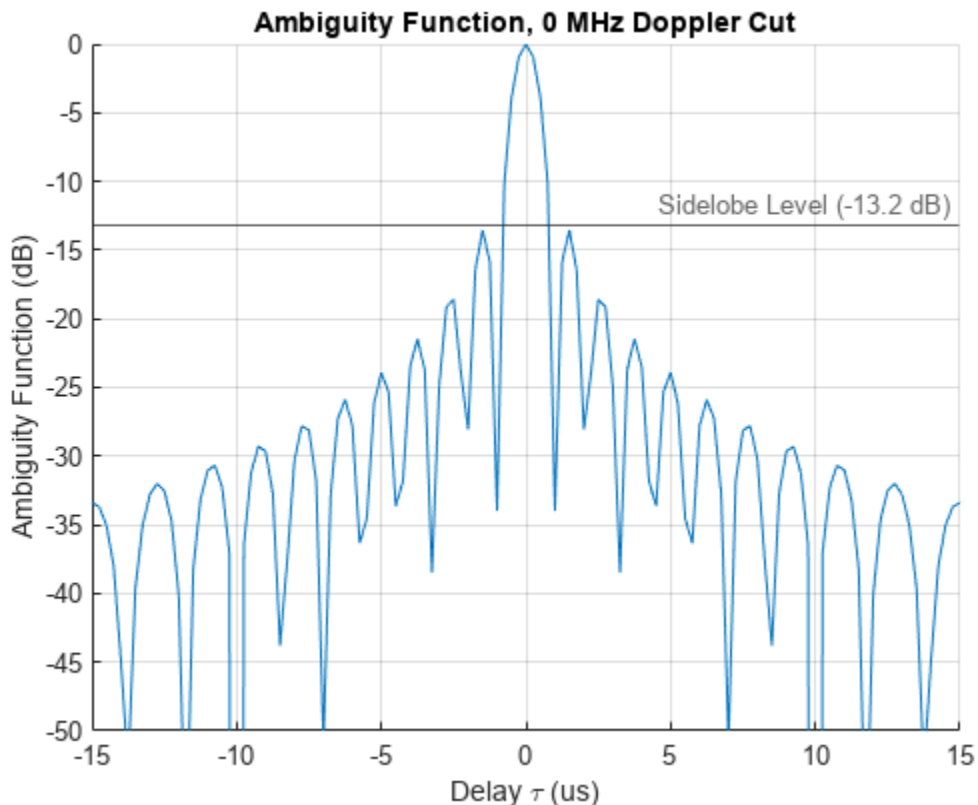


From the preceding figure, one can see that even though the response now has sidelobes, the first null still appears at 1 microsecond, so the range resolution is preserved. The sidelobe level for a waveform is often described using the peak-to-sidelobe ratio (PSR). Typically, the sidelobes near the main lobe are the highest. The sidelobes created by a stronger target can mask returns from a weaker target and make its detection impossible even if the two targets are resolvable in range. For a linear FM pulse with a rectangular amplitude the PSR is -13.2 dB.

One can also plot the zero-delay cut of the linear FM waveform. Observe that the first null in Doppler domain is now at around 20 kHz, which is 1/50 of the original rectangular waveform.

```
[lfmaf_doppler,doppler] = ambgfun(lfmsamples,lfmwav.SampleRate,lfmwav.PRF,'Cut','Delay');

figure
plot(doppler*1e-3,lfmaf_doppler)
grid on
xlabel('Doppler f_d (kHz)')
```

```
ylabel({'Ambiguity','Function'})
xlim([-200 200])
title('Ambiguity Function, 0 us Delay Cut')
```
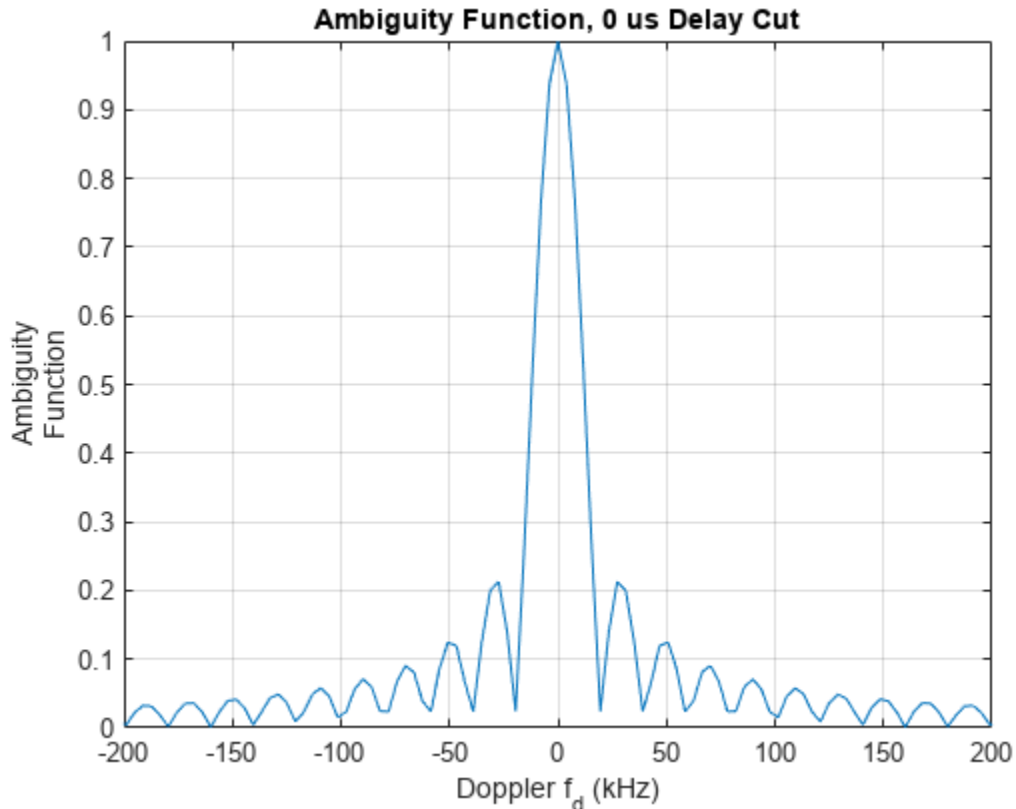


Following the same procedure as for the rectangular waveform in the earlier sections of this example, one can calculate that the 20 kHz Doppler separation translates to a speed difference of 3 km/s. This resolution is 50 times better than the rectangular waveform. Unfortunately, such resolution is still inadequate.

```
lfm_deltav = dop2speed(20e3,wavelen2freq(fc))/2
```

lfm_deltav = 2.9979e+03

It can be instructive to see a 3-D plot of the ambiguity function or to plot the top view of the ambiguity function as an image. To generate a plot in other than the contour format, you can just get the returned ambiguity function and then plot it using your favorite format. For example, the following snippet generates the surface plot and a top view image of a linear FM waveform ambiguity function.

```
[lfmaf,delay,doppler] = ambgfun(lfmsamples,lfmwav.SampleRate,lfmwav.PRF);

figure
tiledlayout(2, 1)

nexttile
surf(delay*1e6,doppler/1e3,lfmaf,'LineStyle','none')
xlim([-tau_fm tau_fm]*1e6)
```

```
ylim([-bw bw]*1e-3)
colorbar
view([-40,35])
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')

nexttile
imagesc(delay*1e6,doppler/1e3,mag2db(lfmaf))
xlim([-tau_fm tau_fm]*1e6)
ylim([-bw bw]*1e-3)
clim([-30 0])
cbar = colorbar;
cbar.Label.String = '(dB)';
axis xy
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')

sgtitle('Linear FM Pulse Waveform Ambiguity Function')
```
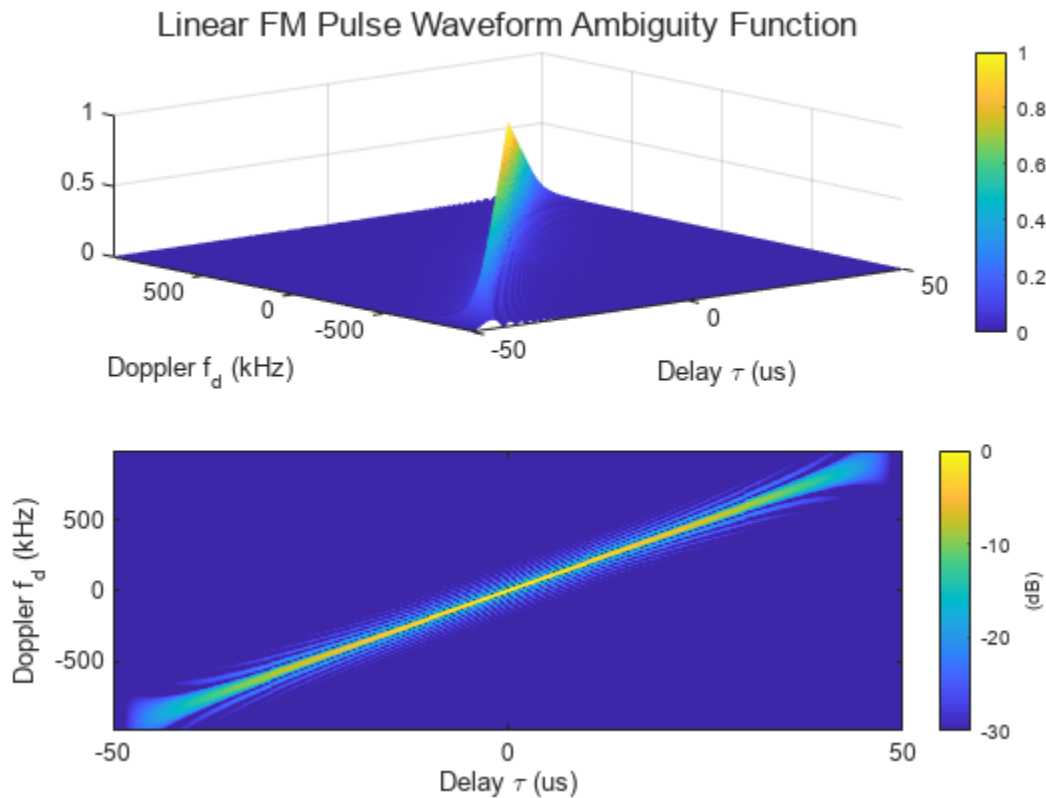


Notice that compared to the ambiguity function of the rectangular waveform, the ambiguity function of the linear FM waveform is tilted. This is due to delay-Doppler coupling. The tilt provides the improved resolution in the zero-delay cut.

Since typical Doppler shifts are much smaller compared to the waveform's bandwidth, the peak values of non-zero-Doppler cuts and hence the peak values of the matched filter outputs decrease only slightly when moving targets are observed. This behavior of the ambiguity function is what makes the linear FM waveform Doppler tolerant.

The ambiguity functions of both the rectangular waveform and the linear FM waveform have the shape of a long, narrow edge. This kind of ambiguity function is often termed as a *knife edge* ambiguity function.

Before proceeding to improve further the Doppler resolution, it is worth looking at an important figure of merit used in waveform analysis. The product of the pulse width and the bandwidth of a waveform is called the waveform's *time-bandwidth product*. For a rectangular waveform, the time-bandwidth product is always 1. For a linear FM waveform, because of the decoupling of the bandwidth and the pulse width, the time-bandwidth can be larger than 1. The waveform just used has a time-bandwidth product of 50. Recall that by preserving the same range resolution as the rectangular waveform, the linear FM waveform achieves a Doppler resolution that is 50 times better.

**Coherent Pulse Train**

As of the previous section, the Doppler resolution of the linear FM waveform is still fairly poor. One way to improve this resolution is to further extend the pulse width. However, this approach will not work for two reasons:

- The duty cycle of the waveform is already 50%, which is close to the practical limit. (Even if one could, say, use a 100% duty cycle, it is still only a factor of 2 improvement, which is far from being able to resolve the issue.)
- Longer pulse width means large minimum detectable range, which is also undesirable.

If one cannot extend the pulse width within one pulse, one has to look beyond this boundary. Indeed, in modern radar and sonar systems, the Doppler processing often uses a coherent pulse train. The more pulses in the pulse train, the finer the Doppler resolution.

To illustrate the idea, next, try a five-pulse burst.

```
release(lfmwav);
lfmwav.NumPulses = 5;
trainsamples = lfmwav();
```

First, plot the zero-Doppler cut of the ambiguity function.

```
[trainaf_delay,delay] = ambgfun(trainsamples,lfmwav.SampleRate,lfmwav.PRF,'Cut','Doppler');

figure
tiledlayout(2,1)

nexttile
plot(delay*1e6,trainaf_delay)
grid on
xlabel('Delay \tau (us)')
ylabel({'Ambiguity','Function'})
title('Ambiguity Function, 0 MHz Doppler Cut')

nexttile
plot(delay*1e6,trainaf_delay)
grid on
xlabel('Delay \tau (us)')
ylabel({'Ambiguity','Function'})
xlim([-2 2])
```
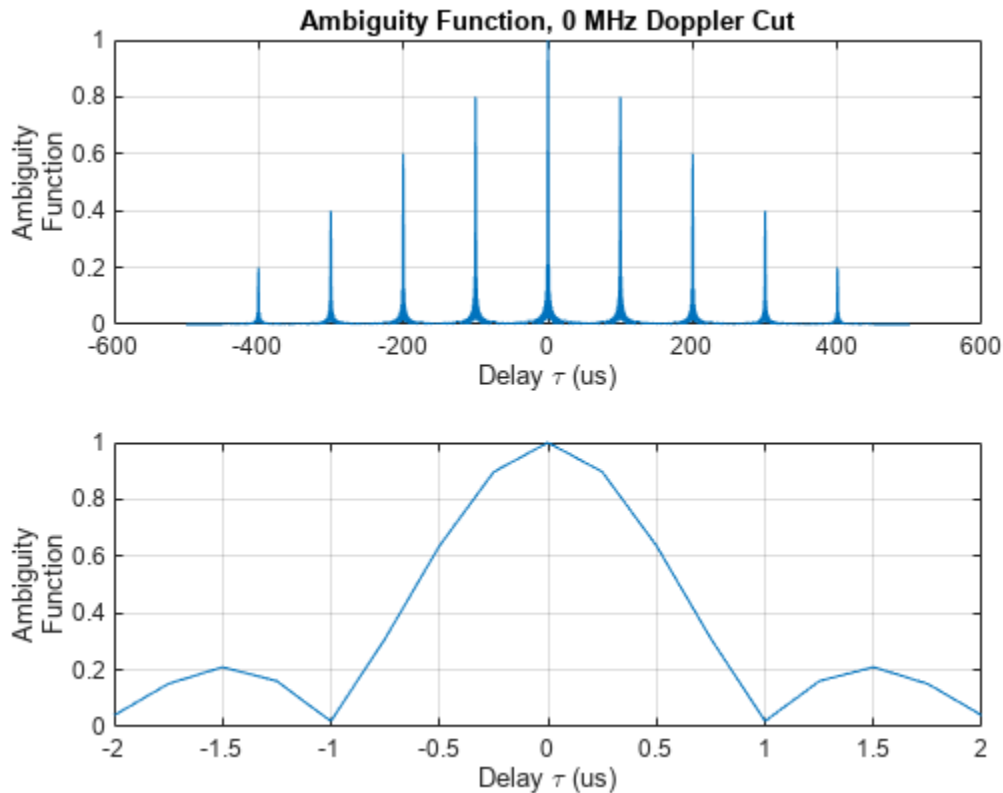
Notice that for the zero-Doppler cut, the first null is still around 1 microsecond, so the range resolution is the same. One should see immediately the presence of many range domain peaks. These peaks are the tradeoff for using a pulse train. The distance between the main lobe and the first peak is the length of one entire pulse, i.e., the reciprocal of the PRF.

```
T_max = 1/prf
```

```
T_max = 1.0007e-04
```

This time delay corresponds to the maximum unambiguous range.

The zero-delay cut also has peaks because of the pulse train.

```
[trainaf_doppler,doppler] = ambgfun(trainsamples,lfmwav.SampleRate,lfmwav.PRF,'Cut','Delay');
```

```
figure
tiledlayout(2,1)

nexttile
plot(doppler*1e-3,trainaf_doppler)
grid on
xlabel('Doppler f_d (kHz)')
ylabel({'Ambiguity','Function'})
title('Ambiguity Function, 0 us Delay Cut')
xlim([-bw bw]*1e-3)

nexttile
```

```
plot(doppler*1e-3,trainaf_doppler)
grid on
xlabel('Doppler f_d (kHz)')
ylabel({'Ambiguity','Function'})
xlim([-40 40])
```
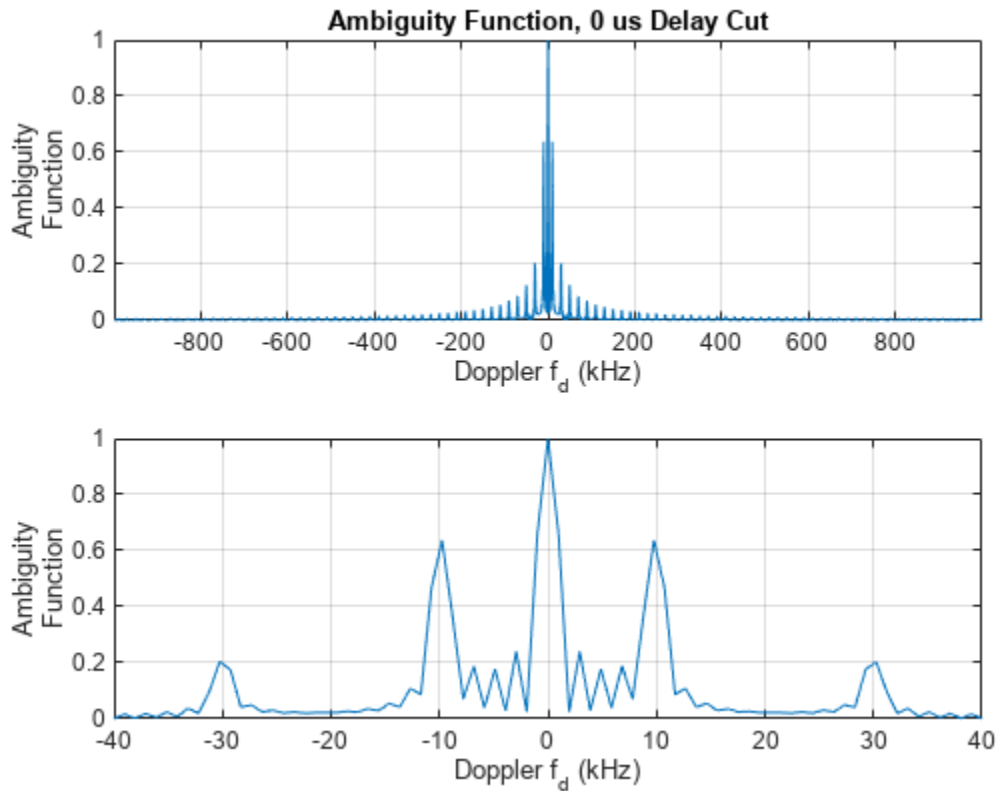


The distance between the adjacent peaks is equal to PRF. This value is the maximum unambiguous Doppler shift the system can measure. One can also calculate the corresponding maximum unambiguous speed.

```
V_max = dop2speed(lfmwav.PRF,freq2wavelen(fc))
```

```
V_max = 2.9959e+03
```

However, notice that the main lobe is now much sharper. Careful examination reveals that the first null is at about 2 kHz. This Doppler resolution can actually be obtained by the following equation,

```
train_deltaf = lfmwav.PRF/lfmwav.NumPulses
```

```
train_deltaf = 1.9986e+03
```

i.e., the resolution is now determined by the length of our entire pulse train, not the pulse width of a single pulse. The corresponding speed resolution is now

```
train_deltav = dop2speed(train_deltaf,freq2wavelen(fc))
```

```
train_deltav = 599.1701
```

which is significantly better. More importantly, to get even finer speed resolution, one can simply increase the number of pulses included in the pulse train. Of course, the number of pulses one can have in a burst depends on whether one can preserve the coherence for the entire duration, but that discussion is out of the scope of this example.

One may notice that in the zero-delay cut, the distance between the peaks is no longer constant, especially as the Doppler shift increases. This lack of constancy occurs because the linear FM waveform's ambiguity function is tilted. Hence, judging the separation of peaks in zero-delay cut can be misleading. The ambiguity caused by the pulse train is best observed from the top view, as the next code example shows. Notice that along the edge of the ambiguity function, those sidelobes are indeed evenly spaced.

```
[trainaf,delay,doppler] = ambgfun(trainsamples,lfmwav.SampleRate,lfmwav.PRF);

figure
tiledlayout(2, 1)

nexttile
imagesc(delay*1e6,doppler/1e3,mag2db(trainaf));
xlim([-1/prf 1/prf]*5e6)
ylim([-bw bw]*1e-3)
clim([-30 0])
cbar = colorbar;
cbar.Label.String = '(dB)';
axis xy
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')

nexttile
delay_idx = abs(delay) < 20e-6;
doppler_idx = abs(doppler) < 0.2e6;
contour(delay(delay_idx)*1e6,doppler(doppler_idx)/1e3,trainaf(doppler_idx,delay_idx))
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')
colorbar
grid on

sgtitle('Linear FM Pulse Train Waveform Ambiguity Function')
```
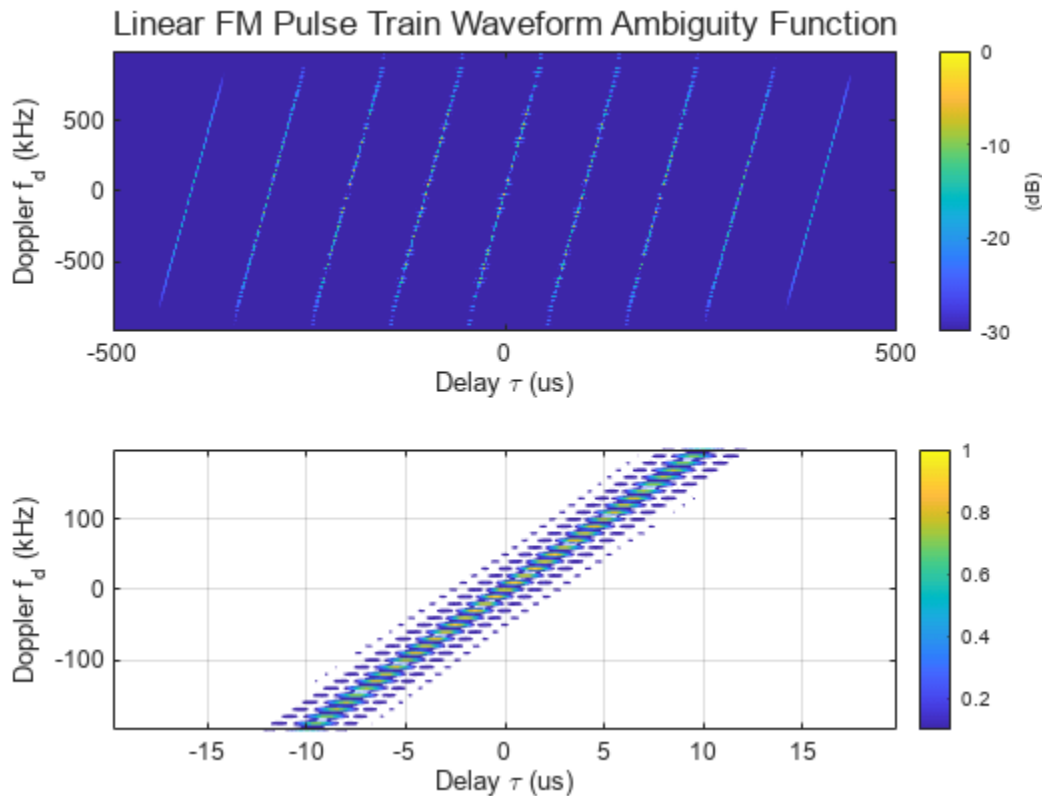
Because of all the peaks, this kind of ambiguity function is called a *bed of nails* ambiguity function.

**Stepped FM Waveform**

The linear FM waveform is very widely used in radar systems. However, it does present some challenges to the hardware. For one thing, the hardware has to be able to sweep the entire frequency range in one pulse. Using this waveform also makes it harder to build the receiver because it has to accommodate the entire bandwidth.

To avoid these issues, you can use a stepped FM waveform instead. A stepped FM waveform consists of multiple contiguous CW pulses. Each pulse has a different frequency and together, all pulses occupy the entire bandwidth. Hence, there is no more sweep within the pulse, and the receiver only needs to accommodate the bandwidth that is the reciprocal of the pulse width of a single pulse.

Next, set up such a stepped FM waveform.

```
stepfmwav = phased.SteppedFMWaveform('SampleRate', fs,'PulseWidth',5/bw,...
    'PRF',prf,'NumSteps',5,'FrequencyStep',bw/5,'NumPulses',5)

stepfmwav =
  phased.SteppedFMWaveform with properties:

                SampleRate: 3.9972e+06
     DurationSpecification: 'Pulse width'
                PulseWidth: 5.0035e-06
                       PRF: 9.9931e+03
     PRFSelectionInputPort: false
```

```
            FrequencyStep: 1.9986e+05
                 NumSteps: 5
     FrequencyOffsetSource: 'Property'
          FrequencyOffset: 0
             OutputFormat: 'Pulses'
                NumPulses: 5
            PRFOutputPort: false
    CoefficientsOutputPort: false
```

```
stepfmsamples = stepfmwav();
```

The zero-Doppler cut, zero-delay cut, and two-dimensional plot of the ambiguity function are shown below.

```
[stepaf_delay,delay] = ambgfun(stepfmsamples,stepfmwav.SampleRate,stepfmwav.PRF,'Cut','Doppler')

figure;
tiledlayout(2,1)

nexttile
plot(delay*1e6,stepaf_delay)
grid on
xlabel('Delay \tau (us)')
ylabel({'Ambiguity','Function'})
title('Ambiguity Function, 0 MHz Doppler Cut')

nexttile
plot(delay*1e6,stepaf_delay)
grid on
xlabel('Delay \tau (us)')
ylabel({'Ambiguity','Function'})
xlim([-2 2])
```
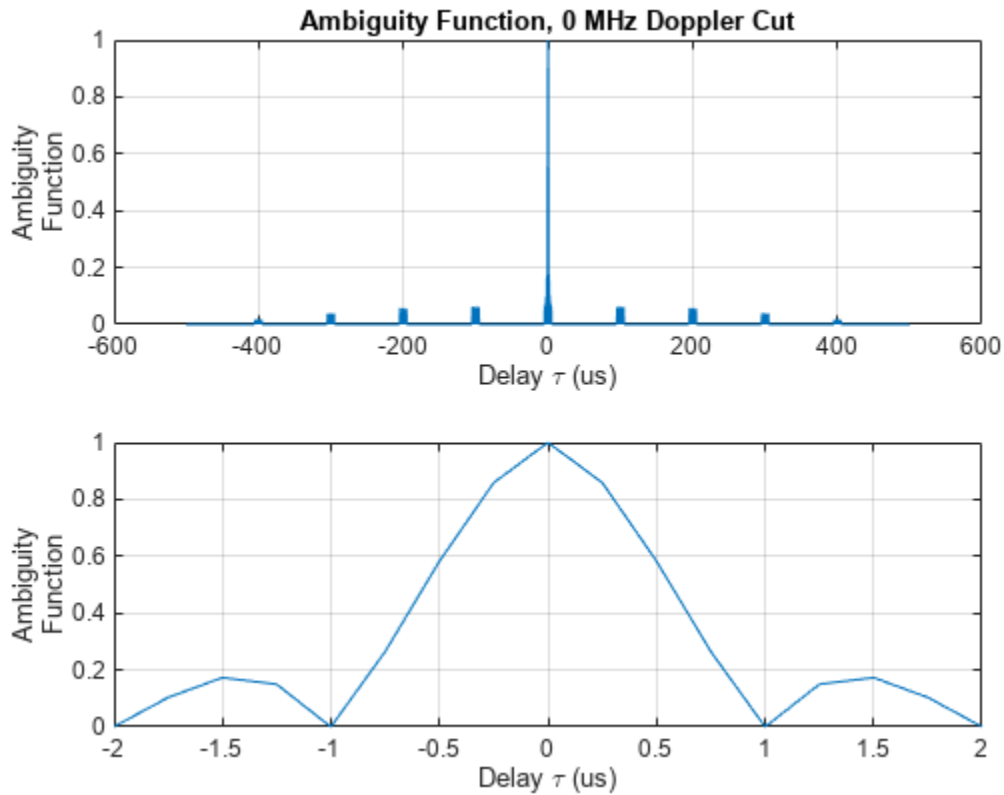
```
[stepaf_doppler,doppler] = ambgfun(stepfmsamples,stepfmwav.SampleRate,stepfmwav.PRF,'Cut','Delay

figure
tiledlayout(2,1)

nexttile
plot(doppler*1e-3,stepaf_doppler)
grid on
xlabel('Doppler f_d (kHz)')
ylabel({'Ambiguity','Function'})
title('Ambiguity Function, 0 us Delay Cut')
xlim([-bw bw]*1e-3)

nexttile
plot(doppler*1e-3,stepaf_doppler)
grid on
xlabel('Doppler f_d (kHz)')
ylabel({'Ambiguity','Function'})
xlim([-40 40])
```
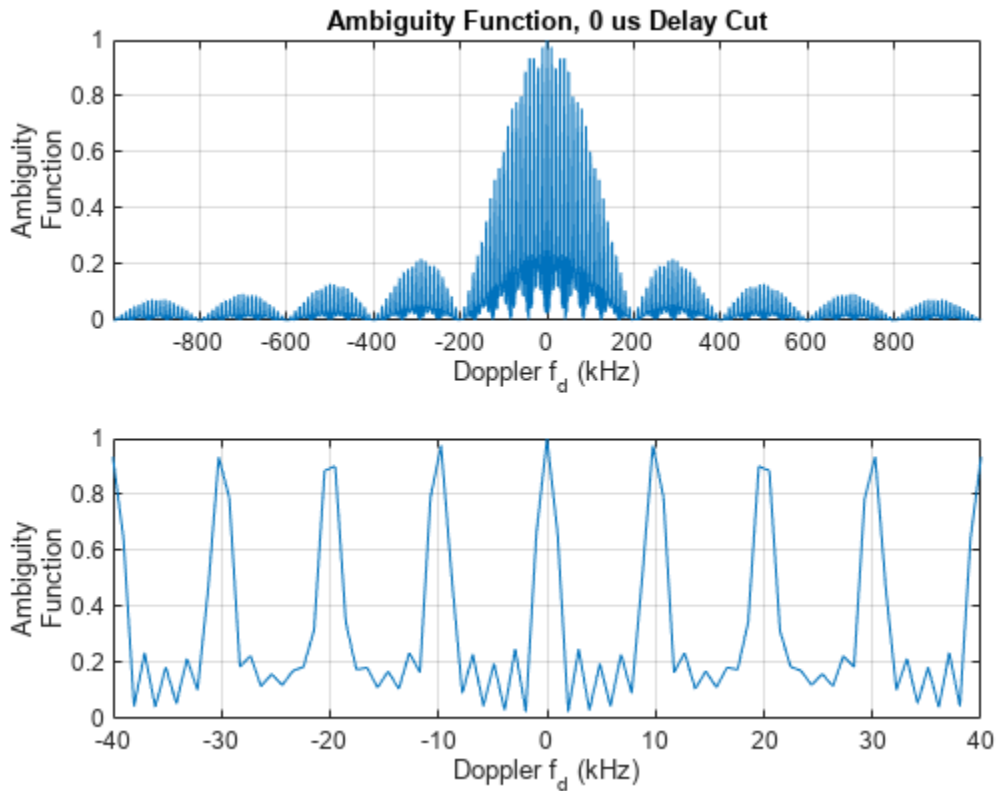
```
[stepaf,delay,doppler] = ambgfun(stepfmsamples,stepfmwav.SampleRate,stepfmwav.PRF);

figure
tiledlayout(2, 1)

nexttile
imagesc(delay*1e6,doppler/1e3,mag2db(stepaf))
xlim([-1/prf 1/prf]*5e6)
ylim([-bw bw]*1e-3)
clim([-30 0])
cbar = colorbar;
cbar.Label.String = '(dB)';
axis xy
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')

nexttile
delay_idx = abs(delay) < 20e-6;
doppler_idx = abs(doppler) < 0.2e6;
contour(delay(delay_idx)*1e6,doppler(doppler_idx)/1e3,stepaf(doppler_idx,delay_idx))
xlabel('Delay \tau (us)')
ylabel('Doppler f_d (kHz)')
colorbar
grid on

sgtitle('Stepped FM Waveform Ambiguity Function')
```

Stepped FM Waveform Ambiguity Function
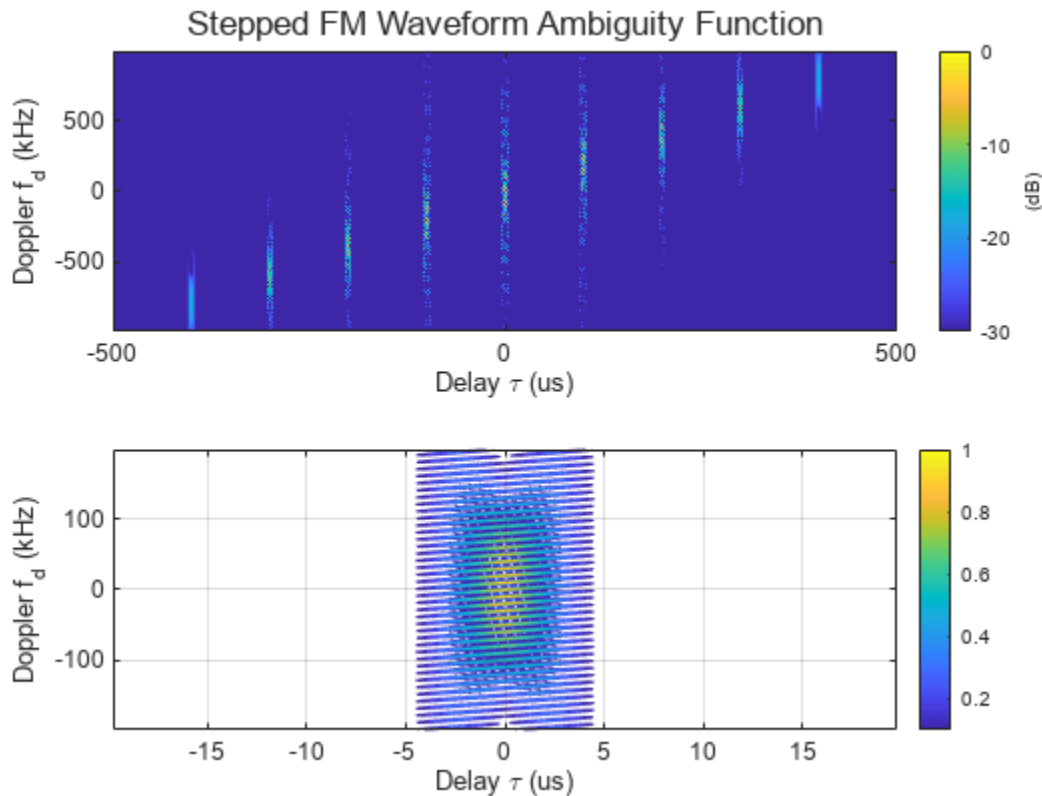
From these figures, one can make the following observations:

- The first null in delay is still at 1 microsecond, so the range resolution is preserved. Notice that because each pulse is different, the prominent peaks in the range domain disappear.
- The first null in Doppler is still at 2 kHz, so it has the same Doppler resolution as the 5-pulse linear FM pulse train. The sidelobes in the Doppler domain are still present as in the linear FM pulse train case.
- The ambiguity function of the stepped FM waveform also has a bed of nails type. Although the unambiguous range is greatly extended, the unambiguous Doppler is still confined by the waveform's PRF.

As to the disadvantage of a stepped FM waveform, the processing becomes more complicated.

**Nonlinear FM Waveforms**

Previous sections in this example showed that a linear FM pulse with a large time-bandwidth product can achieve a good resolution in range, while a coherent train of such pulses allows for achieving a good resolution in speed. Waveforms with large time-bandwidth products can also be generated by employing nonlinear FM. By selecting a particular nonlinear FM one can obtain a waveform with an ambiguity function that possesses some desired properties. For example, nonlinear FM can be used to obtain waveforms with low range sidelobes.

The shape of the waveform's spectrum determines the range sidelobe level. The spectrum of a linear FM pulse has an approximately rectangular shape. This is the reason the range sidelobes of a linear FM are 13.2 dB below the main lobe. Such relatively high sidelobe level might not be suited for some

radar and sonar applications. The sidelobe level reduction can be accomplished by applying a window (taper) on the received side. The windowing, however, perturbs the matched filter result. Since the cumulative filtering no longer matches the signal exactly, it also reduces the output SNR. In contrast to linear FM, nonlinear FM waveforms can achieve low range sidelobes by shaping the spectrum using frequency modulation. In this case the time domain envelope remains constant and the loss in SNR is not incurred.

Various nonlinear functions can be used to modulate the instantaneous frequency of nonlinear FM waveforms. Often a nonlinear FM waveform is constructed by combining linear and nonlinear frequency modulation. For example, in [3] a hybrid nonlinear FM waveform is presented that is a combination of a linear and a tangent FM.

To construct a hybrid linear-tangent FM waveform two additional parameters must be specified. The first parameter, `alpha`, determines the balance between the linear and the tangent components. If `alpha` is zero, the resulting waveform will have only linear frequency modulation. If `alpha` is one, the instantaneous frequency of the resulting waveform will follow the tangent curve. The second parameter, `gamma`, determines the portion of the tangent curve that is used. By adjusting `alpha` and `gamma` the range sidelobes of the hybrid linear-tangent FM waveform can be made significantly lower than -13.2 dB, while keeping the range resolution close to the range resolution of the linear FM.

Create a hybrid linear-tangent FM waveform with the same pulse width, bandwidth, and PRF as the linear FM discussed in the earlier sections.

```
alpha = 0.5;
gamma = 1.4;
nlfmwav = phased.NonlinearFMWaveform('SampleRate',fs,'PulseWidth',tau_fm, ...
    'SweepBandwidth',bw,'FrequencyModulation','Hybrid Linear-Tangent', ...
    'LinearTangentBalance',alpha,'TangentCurvePortion',gamma,'PRF',prf)
```

```
nlfmwav =
  phased.NonlinearFMWaveform with properties:

               SampleRate: 3.9972e+06
       DurationSpecification: 'Pulse width'
               PulseWidth: 5.0035e-05
                      PRF: 9.9931e+03
       PRFSelectionInputPort: false
         FrequencyModulation: 'Hybrid Linear-Tangent'
        LinearTangentBalance: 0.5000
         TangentCurvePortion: 1.4000
              SweepBandwidth: 9.9931e+05
              SweepDirection: 'Up'
               SweepInterval: 'Positive'
                    Envelope: 'Rectangular'
        FrequencyOffsetSource: 'Property'
              FrequencyOffset: 0
                 OutputFormat: 'Pulses'
                    NumPulses: 1
                PRFOutputPort: false
        CoefficientsOutputPort: false
```

Compute and plot in the dB scale the zero-Doppler cut of the ambiguity function for the hybrid linear-tangent FM. For comparison show the corresponding zero-Doppler cut for the linear FM waveform.

```
nlfmsamples = nlfmwav();
```

```
[nlfmaf_delay, delay] = ambgfun(nlfmsamples,fs,prf,'Cut','Doppler');

figure
hold on
plot(delay*1e6,mag2db(lfmaf_delay),'--')
plot(delay*1e6,mag2db(nlfmaf_delay))
yline(-23.5,'Label','Sidelobe Level (-23.5 dB)')
grid on
xlabel('Delay \tau (us)')
ylabel({'Ambiguity Function (dB)'})
xlim([-tau_fm tau_fm]*3e5)
ylim([-50 0])
title('Ambiguity Function, 0 MHz Doppler Cut');
legend('Linear FM', 'Nonlinear FM')
```
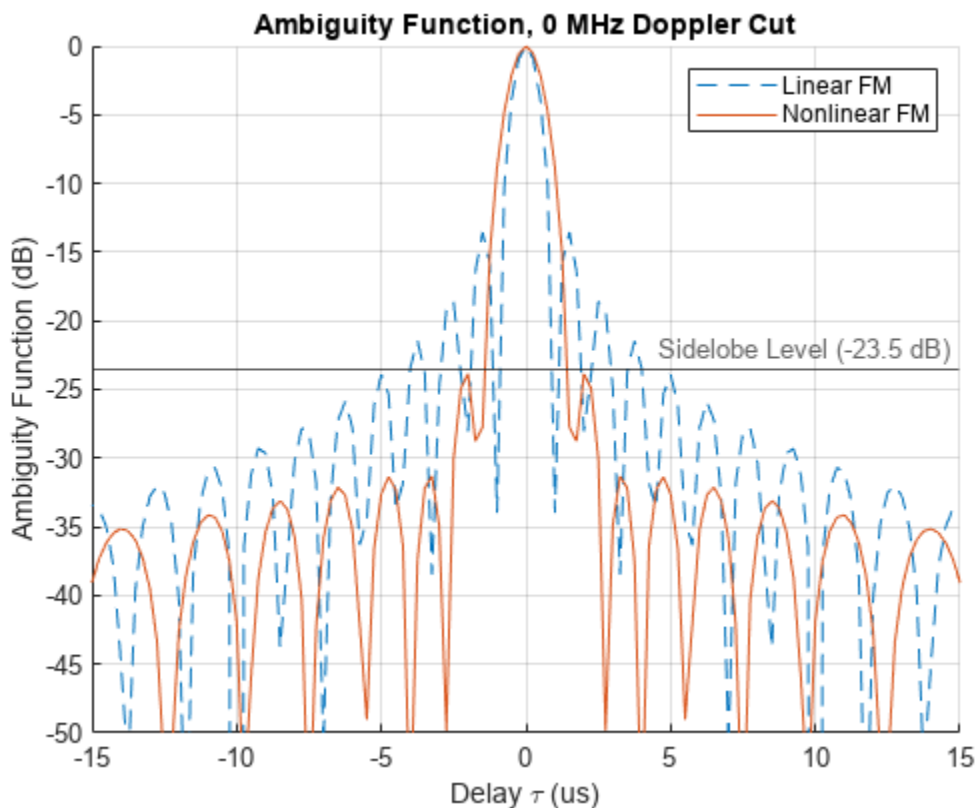


The zero-Doppler cut of the ambiguity function for the hybrid linear-tangent FM shows that the first null occurs at around 1.5 microseconds. This means that the range resolution is degraded to 225 m compared to 150 m for linear FM with the same bandwidth. The range sidelobe level, however, is more than 10 dB lower.

Nonlinear FM can also be used to generate a waveform with a *thumbtack* ambiguity function. An ideal thumbtack ambiguity function has a single sharp peak at the origin and a near-zero level everywhere else [1]. Waveforms with a thumbtack ambiguity are Doppler sensitive (Doppler intolerant) but can achieve high range and Doppler resolution. An example of a nonlinear FM that has a thumbtack ambiguity function is the generalized sinusoidal FM (GSFM) waveform [4]. The code below generates a GSFM waveform.

```
% Instantaneous frequency of a GSFM waveform
rho = 1.9;                    % Variable exponent parameter
alpha = 1e9;                  % Frequency modulation term
gsfm_finst = @(t)((bw/2)*sin(2*pi*alpha*t.^rho));

gsfmwav = phased.CustomFMWaveform('SampleRate',fs,...
    'PulseWidth',tau_fm,'FrequencyModulation',gsfm_finst,'PRF',prf)

gsfmwav =
  phased.CustomFMWaveform with properties:

                SampleRate: 3.9972e+06
     DurationSpecification: 'Pulse width'
                PulseWidth: 5.0035e-05
                       PRF: 9.9931e+03
     PRFSelectionInputPort: false
       FrequencyModulation: @(t)((bw/2)*sin(2*pi*alpha*t.^rho))
                  Envelope: 'Rectangular'
     FrequencyOffsetSource: 'Property'
           FrequencyOffset: 0
              OutputFormat: 'Pulses'
                 NumPulses: 1
             PRFOutputPort: false
      CoefficientsOutputPort: false
```

```
gsfmsamples = gsfmwav();
```

The thumbtack nature of the ambiguity function with a single peak at the origin surrounded by low sidelobes is clearly visible in the contour plot.

```
figure
ambgfun(gsfmsamples,fs,prf)
xlim([-tau_fm tau_fm]*0.5e6)
ylim([-bw bw]*0.5e-6)
```
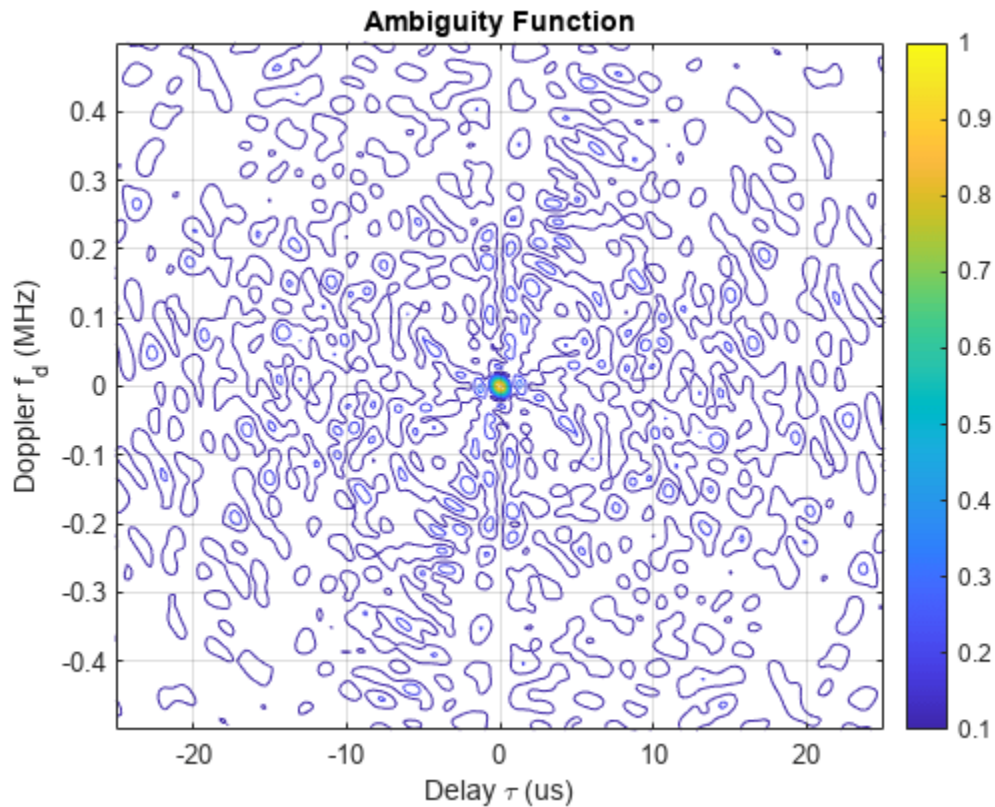
**Barker-Coded Waveform**

Another important group of waveforms is phase-coded waveforms, among which the popularly used ones are Barker codes, Frank codes, and Zadoff-Chu codes. In a phase-coded waveform, a pulse is divided into multiple subpulses, often referred to as chips, and each chip is modulated with a given phase. All phase-coded waveforms have good autocorrelation properties which make them good candidates for pulse compression. Thus, if a phase-coded waveform is adopted, it could lower the probability of interception as the energy is spread into chips. At the receiver, a properly configured matched filter could suppress the noise and achieve good range resolution.

Barker code is one of the most well-known phase-coded waveforms. A Barker-coded waveform can be constructed using the following command.

```
barkerwav = phased.PhaseCodedWaveform('Code','Barker','NumChips',7,...
    'SampleRate', fs,'ChipWidth',tau,'PRF',prf)

barkerwav =
  phased.PhaseCodedWaveform with properties:

                SampleRate: 3.9972e+06
                      Code: 'Barker'
                 ChipWidth: 1.0007e-06
                  NumChips: 7
                       PRF: 9.9931e+03
     PRFSelectionInputPort: false
      FrequencyOffsetSource: 'Property'
           FrequencyOffset: 0
```

**18-175**

```
            OutputFormat: 'Pulses'
               NumPulses: 1
           PRFOutputPort: false
  CoefficientsOutputPort: false
```

```
barkersamples = barkerwav();
```

This Barker code consists of 7 chips. Its zero-Doppler cut of the ambiguity function is given by

```
ambgfun(barkersamples,barkerwav.SampleRate,barkerwav.PRF,'Cut','Doppler')
xlim([-tau tau]*10e6)
```
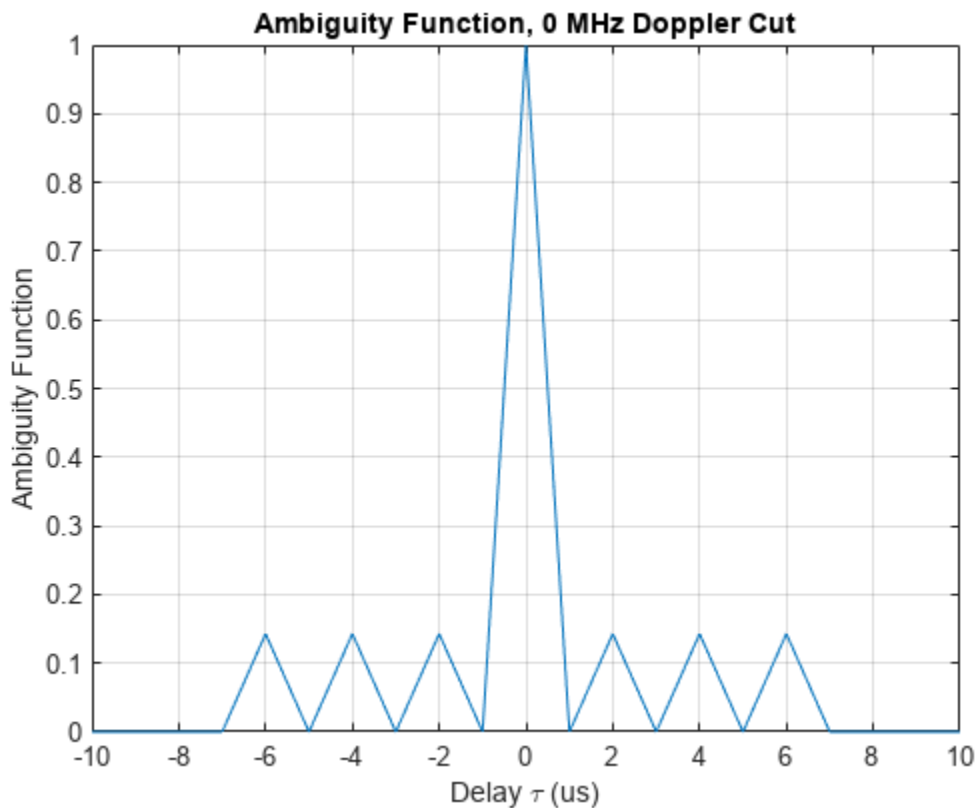


From the figure, one can see that the zero-Doppler cut of a Barker code's ambiguity function has an interesting property. All its sidelobes have the same height and are exactly 1/7 of the main lobe. In fact, a length-N Barker code can provide a peak-to-peak suppression of N, which helps distinguish closely located targets in range. This is the most important property of the Barker code. The range resolution is approximately 10 microseconds, the same as the chip width.

There are two issues associated with a Barker code. First, there are only seven known Barker codes. Their lengths are 2, 3, 4, 5, 7, 11 and 13. It is believed that there are no other Barker codes. Second, the Doppler performance of the Barker code is fairly poor. Although the ambiguity function has good shape at the zero-Doppler cut, once there is some Doppler shift, the sidelobe level increases significantly. Barker codes are Doppler sensitive with a thumbtack ambiguity function.

These properties of the Barker code are visible in the contour plot of the ambiguity function.

```
ambgfun(barkersamples,barkerwav.SampleRate,barkerwav.PRF)
xlim([-tau tau]*10e6)
ylim([-bw bw]*1e-6)
```



**Ambiguity Function**
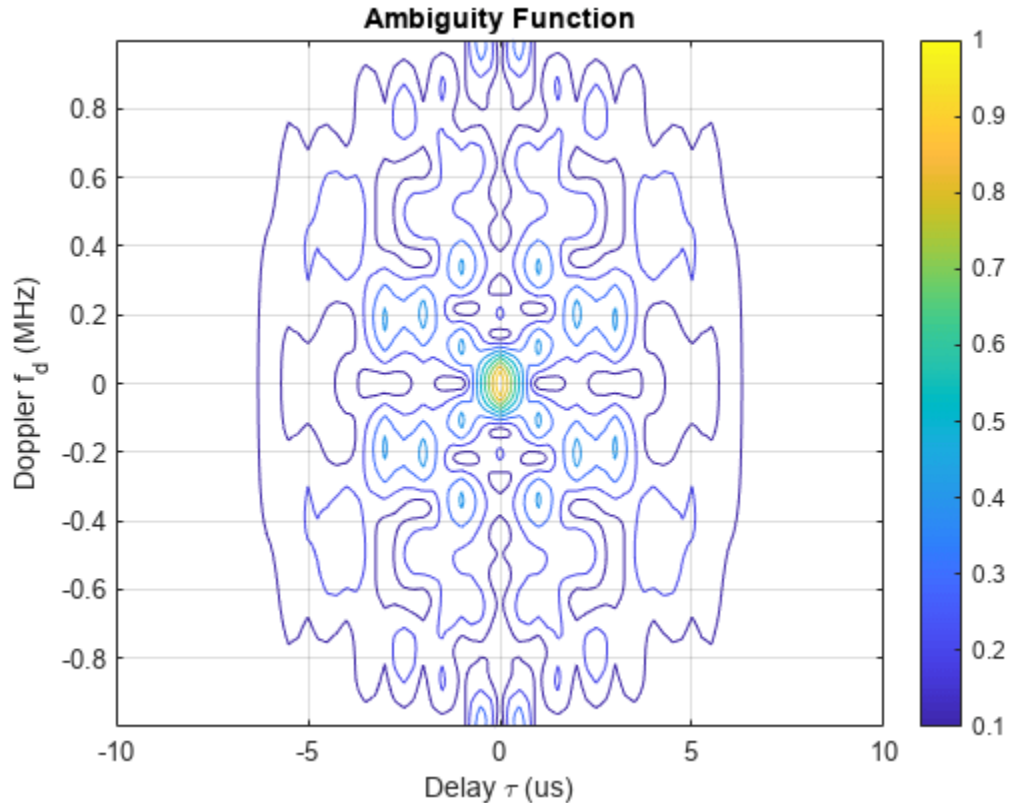
## Summary

This example shows how to use the ambiguity function to analyze and compare waveforms. It compares several common waveforms including the rectangular waveform, the linear FM waveform, the stepped FM waveform, the hybrid linear-tangent FM waveform, the generalized sinusoidal FM waveform, and the Barker-coded waveform. The example shows that the ambiguity function is a convenient tool for analyzing Doppler tolerance of a waveform, its delay and Doppler resolution capabilities, and the sidelobe level.

## Reference

[1] Nadav Levanon and Eli Mozeson, *Radar Signals*, Wiley-IEEE Press, 2004.

[2] Mark Richards, *Fundamentals of Radar Signal Processing*, McGraw Hill, 2005.

[3] Collins, T., and P. Atkins. "Nonlinear frequency modulation chirps for active sonar." *IEE Proceedings-Radar, Sonar and Navigation* 146.6 (1999): 312-316.

[4] Hague, David A., and John R. Buck. "The generalized sinusoidal frequency-modulated waveform for active sonar." *IEEE Journal of Oceanic Engineering* 42.1 (2016): 109-123.

# Waveform Design to Improve Range Performance of an Existing System

This example shows how waveform type affects a radar system's detection performance. The example considers the situation where a new performance goal is set for an existing radar system design. Since the old design can no longer achieve the desired performance, a new waveform is adopted. The example also shows how to model Swerling targets, simulate the return, and then detect the target ranges.

**Design Specification Change**

A monostatic pulse radar is designed in the example "Simulating Test Signals for a Radar Receiver" on page 18-393 to achieve the following goal:

1. minimum target radar cross section (RCS): 1 $m^2$, nonfluctuating;
2. maximum unambiguous range: 5 km;
3. probability of detection: 0.9;
4. probability of false alarm: 1e-6.

```
load BasicMonostaticRadarExampleData;
```

**New Performance Requirement**

After system deployment, two new requirements arise:

1. the maximum unambiguous range needs to be extended to 8 km.
2. the system also needs to be able to detect Swerling case 2 targets.

Can the existing design be modified to achieve the new performance goal? To answer this question, we need to recalculate the parameters affected by these new requirements.

The first affected parameter is the pulse repetition frequency (PRF). It needs to be recalculated based on the new maximum unambiguous range.

```
prop_speed = radiator.PropagationSpeed;
max_range = 8000;
prf = prop_speed/(2*max_range);
```

Compared to the 30 kHz PRF of the existing design, the new PRF, 18.737 kHz, is smaller. Hence the pulse interval is longer. Note that this is a trivial change in the radar software and is fairly cheap in hardware cost.

```
waveform.PRF = prf;
```

Next, because the target is described using a Swerling case 2 model, we need to use Shnidman's equation, instead of Albersheim's equation, to calculate the required SNR to achieve the designated Pd and Pfa. Shnidman's equation assumes noncoherent integration and a square law detector. The number of pulses to integrate is 10.

```
num_pulse_int = 10;
pfa = 1e-6;
snr_min = shnidman(0.9,pfa,num_pulse_int,2)
```

```
snr_min = 6.1583
```

**Waveform Selection**

If we were to use the same rectangular waveform in the existing design, the pulse width would remain the same because it is determined by the range resolution. However, because our maximum range has increased from 5 km to 8 km and the target model switched from nonfluctuating to Swerling case 2, we need to recalculate the required peak transmit power.

```
fc = radiator.OperatingFrequency;
lambda = prop_speed/fc;
peak_power = ((4*pi)^3*noisepow(1/waveform.PulseWidth)*max_range^4*...
    db2pow(snr_min))/(db2pow(2*transmitter.Gain)*1*lambda^2)
```

```
peak_power = 4.4821e+04
```

The peak power is roughly eight times larger than the previous requirement. This is no longer a trivial modification because (1) the existing radar hardware is designed to produce a pulse with peak power of about 5200 w. Although most designs will leave some margin above the required power, it is unlikely that an existing system can accommodate eight times more power; (2) it is very expensive to replace the hardware to output such high power. Therefore, the current design needs to be modified to accommodate the new goal by using more sophisticated signal processing techniques.

**Linear FM Waveform**

One approach to reduce the power requirement is to use a waveform other than the rectangular waveform. For example, a linear FM waveform can use a longer pulse than a rectangular waveform. As a result, the required peak transmit power drops.

Let us examine the details of the linear FM waveform's configuration.

The desired range resolution determines the waveform bandwidth. For a linear FM waveform, the bandwidth is equal to its sweep bandwidth. However, the pulse width is no longer restricted to the reciprocal of the pulse bandwidth, so a much longer pulse width can be used. We use a pulse width that is 20 times longer and set the sample rate to be twice the pulse bandwidth.

```
range_res = 50;
pulse_bw = prop_speed/(2*range_res);
pulse_width = 20/pulse_bw;
fs = 2*pulse_bw;

waveform = phased.LinearFMWaveform(...
    'SweepBandwidth',pulse_bw,...
    'PulseWidth',pulse_width,...
    'PRF',prf,...
    'SampleRate',fs);
```

We now determine the new required transmit power needed to achieve the design requirements.

```
peak_power = ((4*pi)^3*noisepow(1/waveform.PulseWidth)*max_range^4*...
    db2pow(snr_min))/(db2pow(2*transmitter.Gain)*1*lambda^2)
```

```
peak_power = 2.2411e+03
```

This transmit power is well within the capability of our existing radar system. We have achieved a peak transmit power that can meet the new requirements without modifying the existing hardware.

```
transmitter.PeakPower = peak_power;
```

### System Simulation

Now that we have defined the radar to meet the design specifications, we set up the targets and the environment to simulate the entire system.

### Targets

As in the case with the aforementioned example, we assume that there are 3 targets in a free space environment. However, now the target model is Swerling case 2, the target positions and the mean radar cross sections are specified as follows:

```
tgtpos = [[2024.66; 0; 0],[6518.63; 0; 0],[6845.04; 0; 0]];
tgtvel = [[0;0;0],[0;0;0],[0;0;0]];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [2.2 1.1 1.05];
fc = radiator.OperatingFrequency;
target = phased.RadarTarget(...
    'Model','Swerling2',...
    'MeanRCS',tgtrcs,...
    'OperatingFrequency',fc);
```

We set the seed for generating the rcs in the targets so that we can reproduce the same results.

```
target.SeedSource = 'Property';
target.Seed = 2007;
```

### Propagation Environments

We also set up the propagation channel between the radar and each target.

```
channel = phased.FreeSpace(...
    'SampleRate',waveform.SampleRate,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

### Signal Synthesis

We set the seed for the noise generation in the receiver so that we can reproduce the same results.

```
receiver.SeedSource = 'Property';
receiver.Seed = 2007;

fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
slow_time_grid = (0:num_pulse_int-1)/prf;

rxpulses = zeros(numel(fast_time_grid),num_pulse_int); % pre-allocate

for m = 1:num_pulse_int

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);

    % Simulate propagation of pulse in direction of targets
```

```
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    txsig = radiator(txsig,tgtang);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
    tgtsig = target(txsig,true);

    % Receive target returns at sensor
    rxsig = collector(tgtsig,tgtang);
    rxpulses(:,m) = receiver(rxsig,~(txstatus>0));
end
```

**Range Detection**

**Detection Threshold**

The detection threshold is calculated using the noise information, taking into consideration the pulse integration. Note that in the loaded system, as outlined in the aforementioned example, the noise bandwidth is half of the sample rate. We plot the threshold together with the first two pulses.

```
noise_bw = receiver.SampleRate/2;
npower = noisepow(noise_bw,...
    receiver.NoiseFigure,receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(pfa,num_pulse_int,'noncoherent'));

pulseplotnum = 2;
helperRadarPulsePlot(rxpulses,threshold,...
    fast_time_grid,slow_time_grid,pulseplotnum);
```
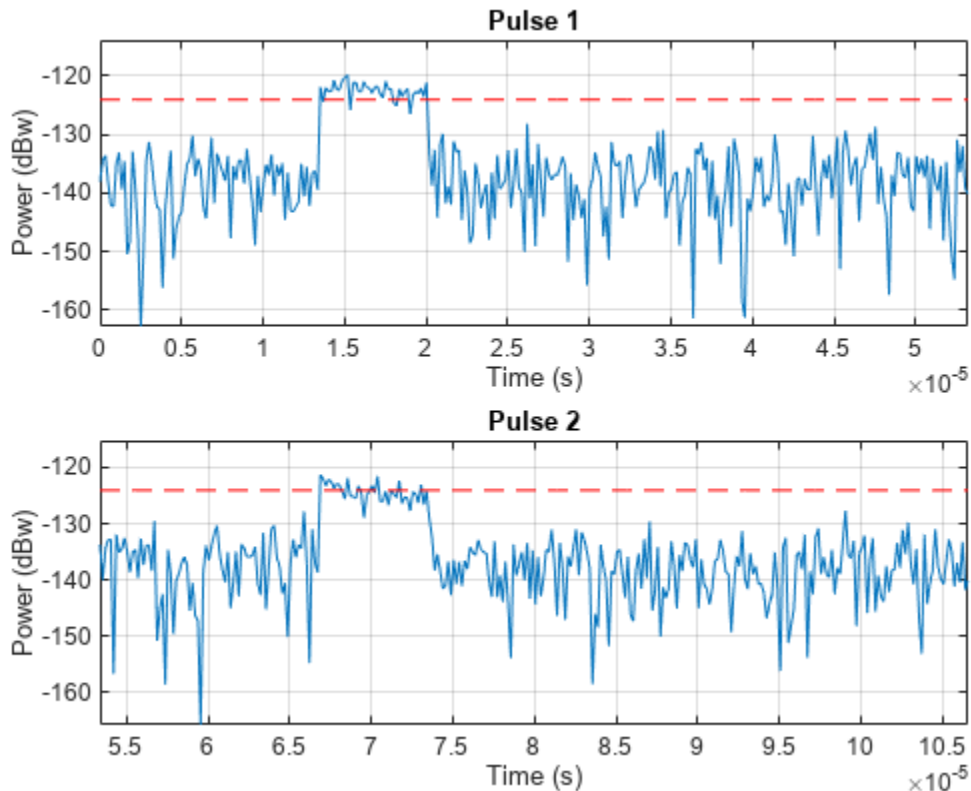
The figure shows that the pulses are very wide which may result in poor range resolution. In addition, the second and third targets are completely masked by the noise.

**Matched Filter**

As in the case of rectangular waveform, the received pulses are first passed through a matched filter to improve the SNR. The matched filter offers a processing gain which further improves the detection threshold. In addition, the added benefit of the matched filter of a linear FM waveform is that it compresses the waveform in the time domain so that the filtered pulse becomes much narrower, which translates to better range resolution.

```
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter(...
    'CoefficientsSource','Property',...
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);
[rxpulses, mfgain] = matchedfilter(rxpulses);
threshold = threshold * db2pow(mfgain);
```

Compensate for the matched filter delay.

```
matchingdelay = size(matchingcoeff,1)-1;
rxpulses = buffer(rxpulses(matchingdelay+1:end),size(rxpulses,1));
```

A time varying gain is then applied to the signal so that a constant threshold can be used across the entire detectable range.

```
range_gates = prop_speed*fast_time_grid/2;
lambda = prop_speed/fc;

tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(range_gates,lambda),...
    'ReferenceLoss',2*fspl(max_range,lambda));
rxpulses = tvg(rxpulses);
```

**Noncoherent Integration**

We now integrate the receive pulses noncoherently to further improve the SNR. This operation is also referred to as video integration.

```
rxpulses = pulsint(rxpulses,'noncoherent');
```

```
helperRadarPulsePlot(rxpulses,threshold,fast_time_grid,slow_time_grid,1);
```
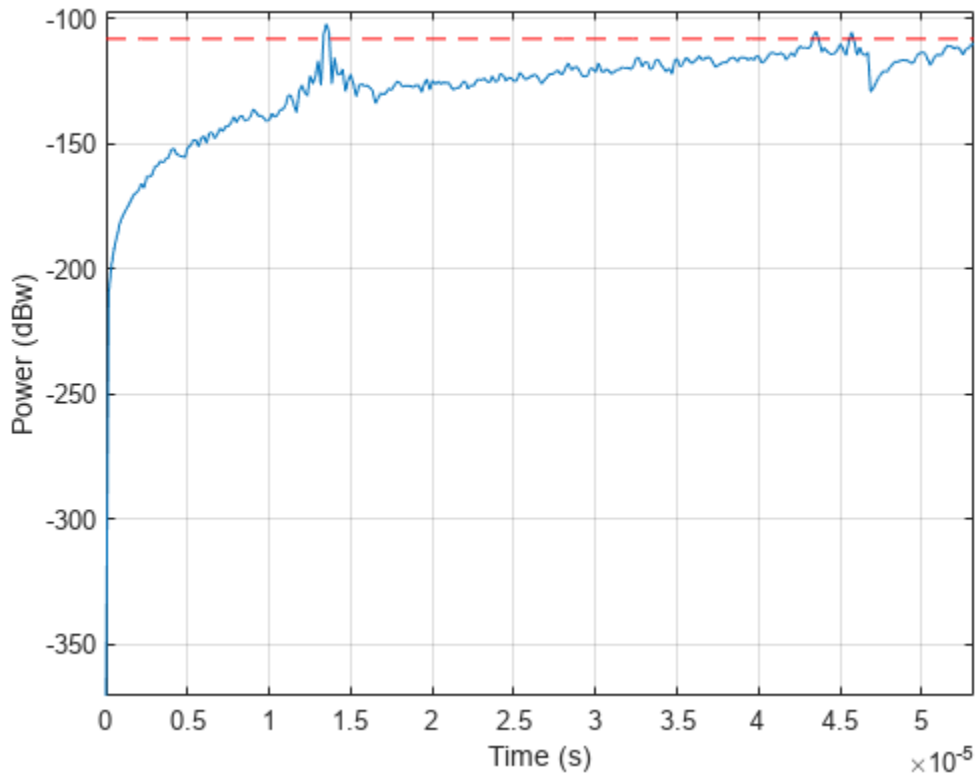


After video integration, the data is ready for the final detection stage. From the plot above, we see that there are no false alarms.

**Range Detection**

Finally, threshold detection is performed on the integrated pulses. The detection scheme identifies the peaks and then translates their positions into the ranges of the targets.

```
[~,range_detect] = findpeaks(rxpulses,'MinPeakHeight',sqrt(threshold));
```

The true ranges and the estimated ranges are

```
true_range = round(tgtrng)

true_range = 1×3

      2025        6519        6845


range_estimates = round(range_gates(range_detect))

range_estimates = 1×3

      2025        6525        6850
```

Note that these range estimates are only accurate up to the range resolution that can be achieved by the radar system, which is 50 m in this example.

**Summary**

In this example, we used the chirp waveform for range detection. By using the chirp waveform, we were able to reduce the required peak transmit power, thus achieving a larger detectable range of 8 km for Swerling case 2 targets.

# Acoustic Beamforming Using a Microphone Array

This example illustrates microphone array beamforming to extract desired speech signals in an interference-dominant, noisy environment. Such operations are useful to enhance speech signal quality for perception or further processing. For example, the noisy environment can be a trading room, and the microphone array can be mounted on the monitor of a trading computer. If the trading computer must accept speech commands from a trader, the beamformer operation is crucial to enhance the received speech quality and achieve the designed speech recognition accuracy.

This example shows two types of time domain beamformers: the time delay beamformer and the Frost beamformer. It illustrates how one can use diagonal loading to improve the robustness of the Frost beamformer. You can listen to the speech signals at each processing step if your system has sound support.

### Define a Uniform Linear Array

First, we define a uniform linear array (ULA) to receive the signal. The array contains 10 omnidirectional microphones and the element spacing is 5 cm.

```matlab
microphone = ...
    phased.OmnidirectionalMicrophoneElement('FrequencyRange',[20 20e3]);

Nele = 10;
ula = phased.ULA(Nele,0.05,'Element',microphone);
c = 340;                          % sound speed, in m/s
```

### Simulate the Received Signals

Next, we simulate the multichannel signals received by the microphone array. We begin by loading two recorded speeches and one laughter recording. We also load the laughter audio segment as interference. The sampling frequency of the audio signals is 8 kHz.

Because the audio signal is usually large, it is often not practical to read the entire signal into the memory. Therefore, in this example, we will simulate and process the signal in a streaming fashion, i.e., breaking the signal into small blocks at the input, processing each block, and then assembling them at the output.

The incident direction of the first speech signal is -30 degrees in azimuth and 0 degrees in elevation. The direction of the second speech signal is -10 degrees in azimuth and 10 degrees in elevation. The interference comes from 20 degrees in azimuth and 0 degrees in elevation.

```matlab
ang_dft = [-30; 0];
ang_cleanspeech = [-10; 10];
ang_laughter = [20; 0];
```

Now we can use a wideband collector to simulate a 3-second multichannel signal received by the array. Notice that this approach assumes that each input single-channel signal is received at the origin of the array by a single microphone.

```matlab
fs = 8000;
collector = phased.WidebandCollector('Sensor',ula,'PropagationSpeed',c,...
    'SampleRate',fs,'NumSubbands',1000,'ModulatedInput', false);

t_duration = 3;   % 3 seconds
t = 0:1/fs:t_duration-1/fs;
```

We generate a white noise signal with a power of 1e-4 watts to represent the thermal noise for each sensor. A local random number stream ensures reproducible results.

```
prevS = rng(2008);
noisePwr = 1e-4; % noise power
```

We now start the simulation. At the output, the received signal is stored in a 10-column matrix. Each column of the matrix represents the signal collected by one microphone. Note that we are also playing back the audio using the streaming approach during the simulation.

```
% preallocate
NSampPerFrame = 1000;
NTSample = t_duration*fs;
sigArray = zeros(NTSample,Nele);
voice_dft = zeros(NTSample,1);
voice_cleanspeech = zeros(NTSample,1);
voice_laugh = zeros(NTSample,1);

% set up audio device writer
audioWriter = audioDeviceWriter('SampleRate',fs, ...
        'SupportVariableSizeInput', true);
isAudioSupported = (length(getAudioDevices(audioWriter))>1);

dftFileReader = dsp.AudioFileReader('dft_voice_8kHz.wav',...
    'SamplesPerFrame',NSampPerFrame);
speechFileReader = dsp.AudioFileReader('cleanspeech_voice_8kHz.wav',...
    'SamplesPerFrame',NSampPerFrame);
laughterFileReader = dsp.AudioFileReader('laughter_8kHz.wav',...
    'SamplesPerFrame',NSampPerFrame);

% simulate
for m = 1:NSampPerFrame:NTSample
    sig_idx = m:m+NSampPerFrame-1;
    x1 = dftFileReader();
    x2 = speechFileReader();
    x3 = 2*laughterFileReader();
    temp = collector([x1 x2 x3],...
        [ang_dft ang_cleanspeech ang_laughter]) + ...
        sqrt(noisePwr)*randn(NSampPerFrame,Nele);
    if isAudioSupported
        play(audioWriter,0.5*temp(:,3));
    end
    sigArray(sig_idx,:) = temp;
    voice_dft(sig_idx) = x1;
    voice_cleanspeech(sig_idx) = x2;
    voice_laugh(sig_idx) = x3;
end
```

Notice that the laughter masks the speech signals, rendering them unintelligible. We can plot the signal in channel 3 as follows:

```
plot(t,sigArray(:,3));
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Signal Received at Channel 3'); ylim([-3 3]);
```
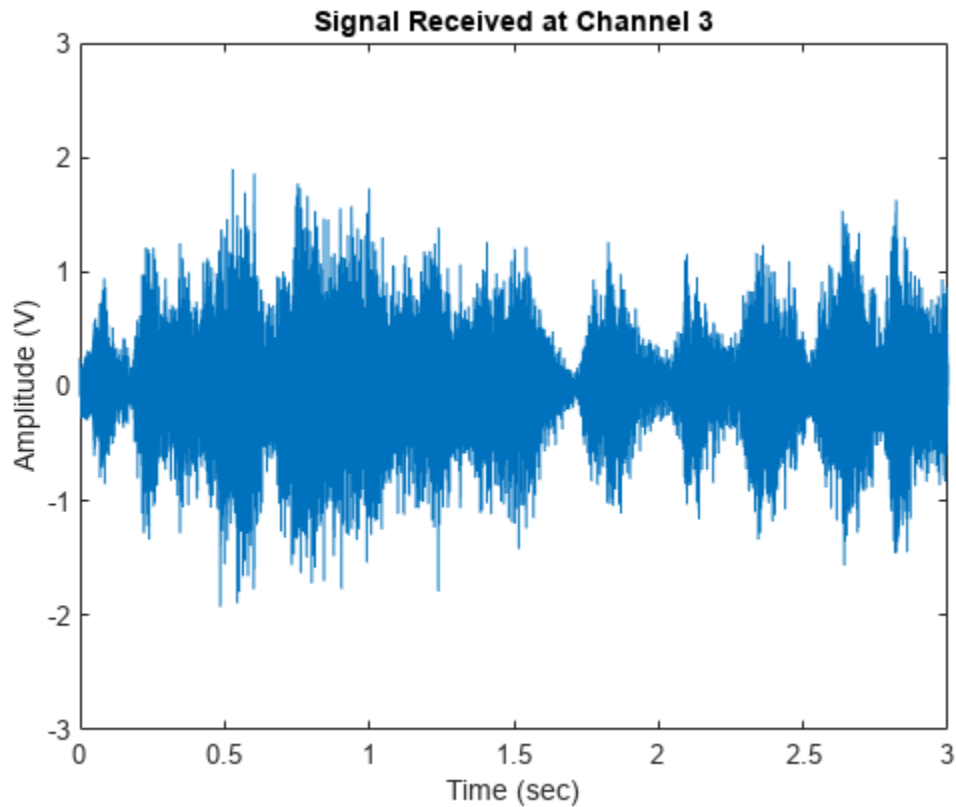
**Process with a Time Delay Beamformer**

The time delay beamformer compensates for the arrival time differences across the array for a signal coming from a specific direction. The time aligned multichannel signals are coherently averaged to improve the signal-to-noise ratio (SNR). Now, define a steering angle corresponding to the incident direction of the first speech signal and construct a time delay beamformer.

```
angSteer = ang_dft;
beamformer = phased.TimeDelayBeamformer('SensorArray',ula,...
    'SampleRate',fs,'Direction',angSteer,'PropagationSpeed',c)

beamformer =
  phased.TimeDelayBeamformer with properties:

          SensorArray: [1x1 phased.ULA]
     PropagationSpeed: 340
           SampleRate: 8000
      DirectionSource: 'Property'
            Direction: [2x1 double]
     WeightsOutputPort: false
```

Next, we process the synthesized signal, plot and listen to the output of the conventional beamformer. Again, we play back the beamformed audio signal during the processing.

```
signalsource = dsp.SignalSource('Signal',sigArray,...
    'SamplesPerFrame',NSampPerFrame);
```

```
cbfOut = zeros(NTSample,1);

for m = 1:NSampPerFrame:NTSample
    temp = beamformer(signalsource());
    if isAudioSupported
        play(audioWriter,temp);
    end
    cbfOut(m:m+NSampPerFrame-1,:) = temp;
end

plot(t,cbfOut);
xlabel('Time (Sec)'); ylabel ('Amplitude (V)');
title('Time Delay Beamformer Output'); ylim([-3 3]);
```
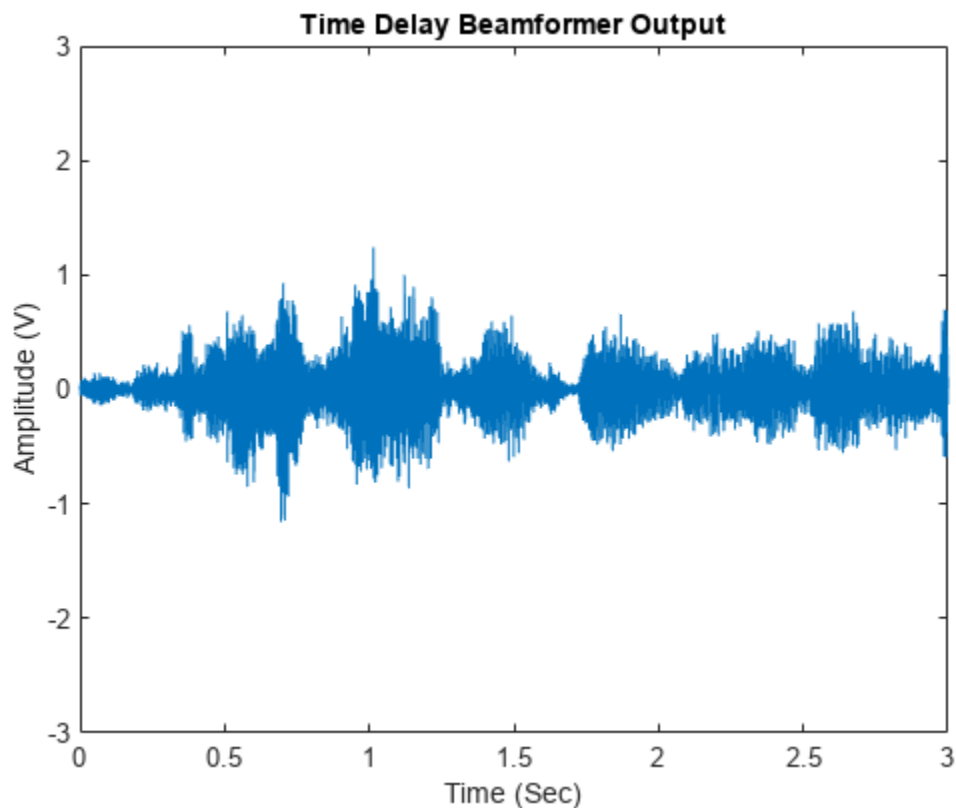


One can measure the speech enhancement by the array gain, which is the ratio of output signal-to-interference-plus-noise ratio (SINR) to input SINR.

```
agCbf = pow2db(mean((voice_cleanspeech+voice_laugh).^2+noisePwr)/...
    mean((cbfOut - voice_dft).^2))
```

agCbf = 9.5022

The first speech signal begins to emerge in the time delay beamformer output. We obtain an SINR improvement of 9.4 dB. However, the background laughter is still comparable to the speech. To obtain better beamformer performance, use a Frost beamformer.

**Process with a Frost Beamformer**

By attaching FIR filters to each sensor, the Frost beamformer has more beamforming weights to suppress the interference. It is an adaptive algorithm that places nulls at learned interference directions to better suppress the interference. In the steering direction, the Frost beamformer uses distortionless constraints to ensure desired signals are not suppressed. Let us create a Frost beamformer with a 20-tap FIR after each sensor.

```
frostbeamformer = ...
    phased.FrostBeamformer('SensorArray',ula,'SampleRate',fs,...
    'PropagationSpeed',c,'FilterLength',20,'DirectionSource','Input port');
```

Next, process the synthesized signal using the Frost beamformer.

```
reset(signalsource);
FrostOut = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    FrostOut(m:m+NSampPerFrame-1,:) = ...
        frostbeamformer(signalsource(),ang_dft);
end
```

We can play and plot the entire audio signal once it is processed.

```
if isAudioSupported
    play(audioWriter,FrostOut);
end

plot(t,FrostOut);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output'); ylim([-3 3]);
```

Frost Beamformer Output
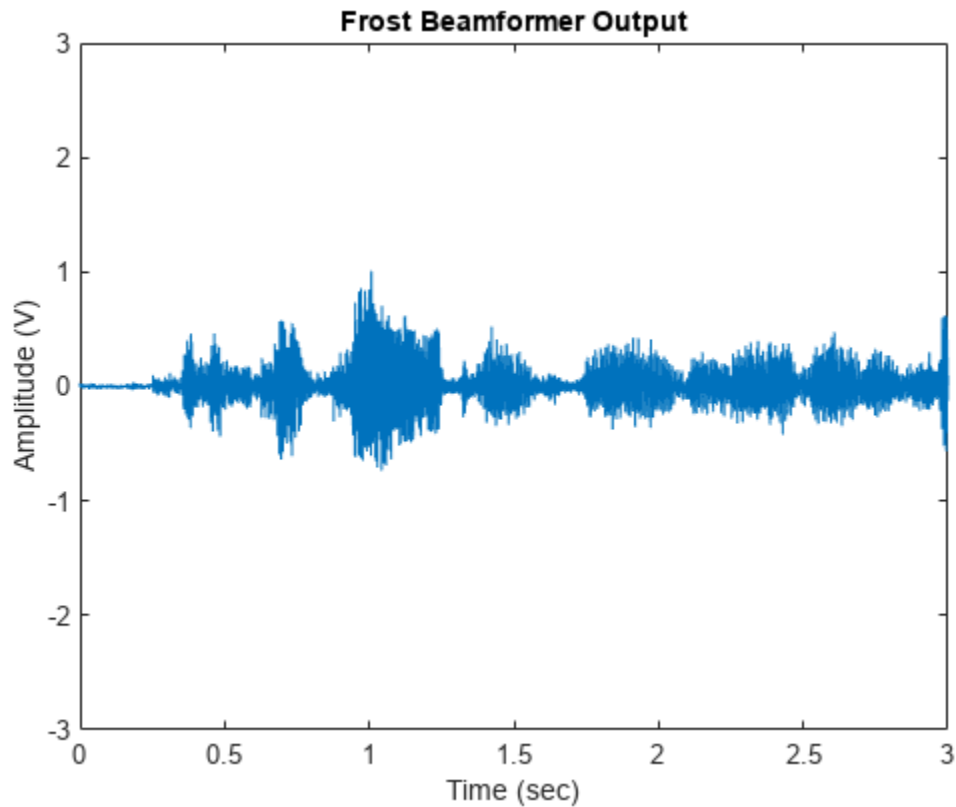
```
% Calculate the array gain
agFrost = pow2db(mean((voice_cleanspeech+voice_laugh).^2+noisePwr)/...
    mean((FrostOut - voice_dft).^2))
```

```
agFrost = 14.4385
```

Notice that the interference is now canceled. The Frost beamformer has an array gain of 14 dB, which is 4.5 dB higher than that of the time delay beamformer. The performance improvement is impressive, but has a high computational cost. In the preceding example, an FIR filter of order 20 is used for each microphone. With all 10 sensors, one needs to invert a 200-by-200 matrix, which may be expensive in real-time processing.

**Use Diagonal Loading to Improve Robustness of the Frost Beamformer**

Next, we want to steer the array in the direction of the second speech signal. Suppose we do not know the exact direction of the second speech signal except a rough estimate of azimuth -5 degrees and elevation 5 degrees.

```
release(frostbeamformer);
ang_cleanspeech_est = [-5; 5];  % Estimated steering direction

reset(signalsource);
FrostOut2 = zeros(NTSample,1);
for m = 1:NSampPerFrame:NTSample
    FrostOut2(m:m+NSampPerFrame-1,:) = frostbeamformer(signalsource(),...
        ang_cleanspeech_est);
```

```
end

if isAudioSupported
    play(audioWriter,FrostOut2);
end

plot(t,FrostOut2);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output');  ylim([-3 3]);
```
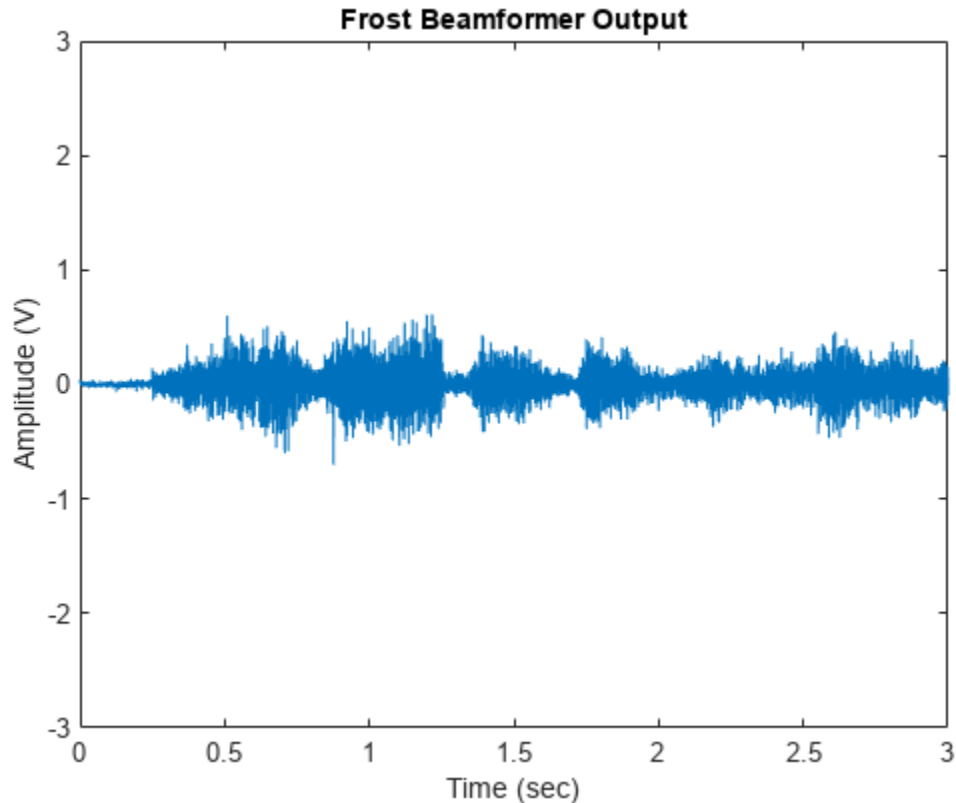


```
% Calculate the array gain
agFrost2 = pow2db(mean((voice_dft+voice_laugh).^2+noisePwr)/...
    mean((FrostOut2 - voice_cleanspeech).^2))
```

agFrost2 = 6.1927

The speech is barely audible. Despite the 6.1 dB gain from the beamformer, performance suffers from the inaccurate steering direction. One way to improve the robustness of the Frost beamformer is to use diagonal loading. This approach adds a small quantity to the diagonal elements of the estimated covariance matrix. Here we use a diagonal value of 1e-3.

```
% Specify diagonal loading value
release(frostbeamformer);
frostbeamformer.DiagonalLoadingFactor = 1e-3;

reset(signalsource);
FrostOut2_dl = zeros(NTSample,1);
```

**18-191**

```
for m = 1:NSampPerFrame:NTSample
    FrostOut2_dl(m:m+NSampPerFrame-1,:) = ...
        frostbeamformer(signalsource(),ang_cleanspeech_est);
end

if isAudioSupported
    play(audioWriter,FrostOut2_dl);
end

plot(t,FrostOut2_dl);
xlabel('Time (sec)'); ylabel ('Amplitude (V)');
title('Frost Beamformer Output');  ylim([-3 3]);
```
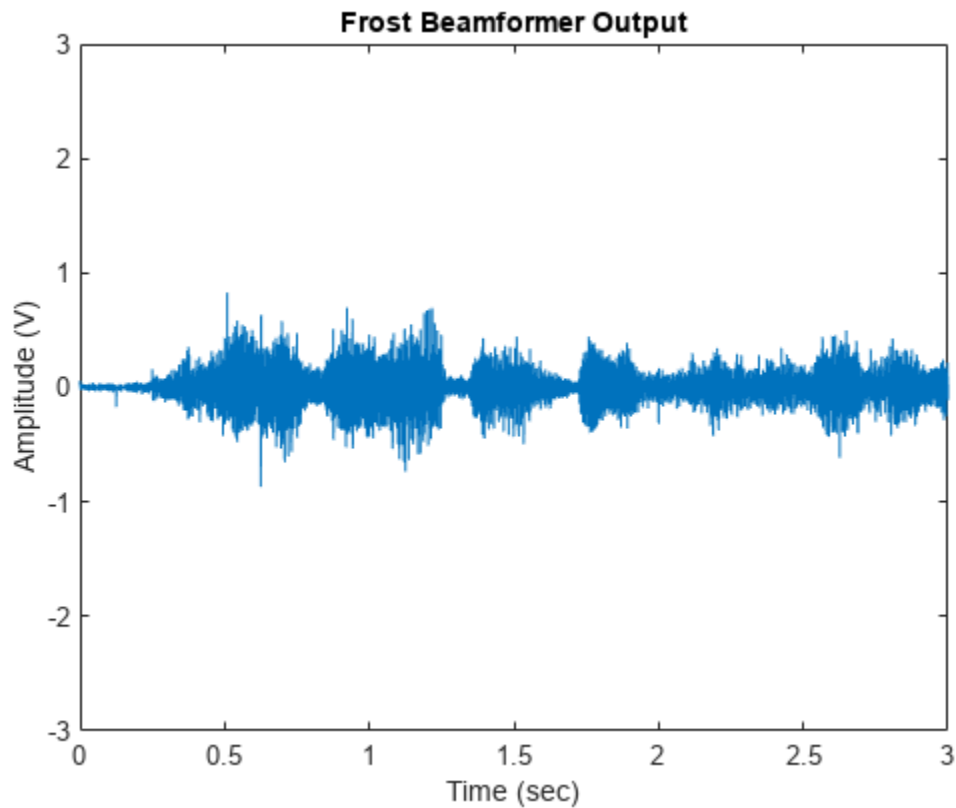


```
% Calculate the array gain
agFrost2_dl = pow2db(mean((voice_dft+voice_laugh).^2+noisePwr)/...
    mean((FrostOut2_dl - voice_cleanspeech).^2))
```

agFrost2_dl = 6.4788

Now the output speech signal is improved and we obtain a 0.3 dB gain improvement from the diagonal loading technique.

```
release(frostbeamformer);
release(signalsource);
if isAudioSupported
    pause(3); % flush out AudioPlayer buffer
    release(audioWriter);
```

```
end
rng(prevS);
```

**Summary**

This example shows how to use time domain beamformers to retrieve speech signals from noisy microphone array measurements. The example also shows how to simulate an interference-dominant signal received by a microphone array. The example used both time delay and the Frost beamformers and compared their performance. The Frost beamformer has a better interference suppression capability. The example also illustrates the use of diagonal loading to improve the robustness of the Frost beamformer.

**Reference**

[1] O. L. Frost III, An algorithm for linear constrained adaptive array processing, Proceedings of the IEEE, Vol. 60, Number 8, Aug. 1972, pp. 925-935.

# Beamforming for MIMO-OFDM Systems

This example shows how to model a point-to-point MIMO-OFDM system with beamforming. The combination of multiple-input-multiple-output (MIMO) and orthogonal frequency division multiplexing (OFDM) techniques have been adopted in recent wireless standards, such as 802.11x families, to provide higher data rate. Because MIMO uses antenna arrays, beamforming can be adopted to improve the received signal to noise ratio (SNR) which in turn reduces the bit error rate (BER).

This example requires Communications Toolbox™.

### Introduction

The term MIMO is used to describe a system where multiple transmitters or multiple receivers are present. In practice the system can take many different forms, such as single-input-multiple-output (SIMO) or multiple-input-single-output (MISO) system. This example illustrates a downlink MISO system. An 8-element ULA is deployed at the base station as the transmitter while the mobile unit is the receiver with a single antenna.

The rest of the system is configured as follows. The transmitter power is 9 watts and the transmit gain is -8 dB. The mobile receiver is stationary and located at 2750 meters away, and is 3 degrees off the transmitter's boresight. An interferer with a power of 1 watt and a gain of -20 dB is located at 9000 meters, 20 degrees off the transmitter's boresight.

```matlab
% Initialize system constants
rng(2014);
gc = helperGetDesignSpecsParameters();

% Tunable parameters
tp.txPower = 9;            % watt
tp.txGain = -8;           % dB
tp.mobileRange = 2750;    % m
tp.mobileAngle = 3;       % degrees
tp.interfPower = 1;       % watt
tp.interfGain = -20;      % dB
tp.interfRange = 9000;    % m
tp.interfAngle =   20;    % degrees
tp.numTXElements = 8;
tp.steeringAngle = 0;     % degrees
tp.rxGain = 108.8320 - tp.txGain; % dB

numTx= tp.numTXElements;
```

The entire scene can be depicted in the figure below.

```matlab
helperPlotMIMOEnvironment(gc, tp);
```
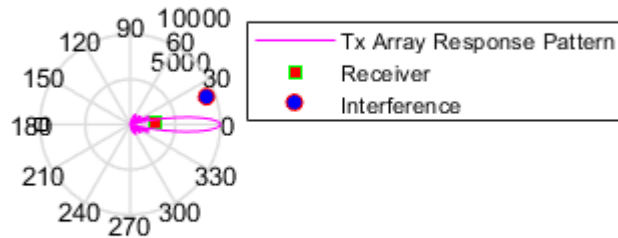
**Signal Transmission**

First, configure the system's transmitter.

```
[encoder,scrambler,modulatorOFDM,steeringvec,transmitter,...
    radiator,pilots,numDataSymbols,frmSz] = helperMIMOTxSetup(gc,tp);
```

There are many components in the transmitter subsystem, such as the convolutional encoder, the scrambler, the QAM modulator, the OFDM modulator, and so on. The message is first converted to an information bit stream and then passed through source coding and modulation stages to prepare for the radiation.

```
txBits = randi([0, 1], frmSz,1);
coded = encoder(txBits);
bitsS = scrambler(coded);
tx = qammod(bitsS,gc.modMode,'InputType','bit','UnitAveragePower',true);
```

In an OFDM system, the data is carried by multiple sub-carriers that are orthogonal to each other.

```
ofdm1 = reshape(tx, gc.numCarriers,numDataSymbols);
```

Then, the data stream is duplicated to all radiating elements in the transmitting array

```
ofdmData = repmat(ofdm1,[1, 1, numTx]);
txOFDM = modulatorOFDM(ofdmData, pilots);
%scale
txOFDM = txOFDM * ...
    (gc.FFTLength/sqrt(gc.FFTLength-sum(gc.NumGuardBandCarriers)-1));

% Amplify to achieve peak TX power for each channel
for n = 1:numTx
    txOFDM(:,n) = transmitter(txOFDM(:,n));
end
```

In a MIMO system, it is also possible to separate multiple users spatial division multiplexing (SDMA). In these situations, the data stream is often modulated by a weight corresponding to the desired direction so that once radiated, the signal is maximized in that direction. Because in a MIMO channel, the signal radiated from different elements in an array may go through different propagation

environments, the signal radiated from each antenna should be propagated individually. This can be achieved by setting CombineRadiatedSignals to false on the phased.Radiator component.

```
radiator.CombineRadiatedSignals = false;
```

To achieve precoding, the data stream radiated from each antenna in the array is modulated by a phase shift corresponding to its radiating direction. The goal of this precoding is to ensure these data streams add in phase if the array is steered toward that direction. Precoding can be specified as weights used at the radiator.

```
wR = steeringvec(gc.fc,[-tp.mobileAngle;0]);
```

Meanwhile, the array is also steered toward a given steering angle, so the total weights are a combination of both precoding and the steering weights.

```
wT = steeringvec(gc.fc,[tp.steeringAngle;0]);
weight = wT.* wR;
```

The transmitted signal is thus given by

```
txOFDM = radiator(txOFDM,repmat([tp.mobileAngle;0],1,numTx),conj(weight));
```

Note that the transmitted signal, txOFDM, is a matrix whose columns represent data streams radiated from the corresponding elements in the transmit array.

**Signal Propagation**

Next, the signal propagates through a MIMO channel. In general, there are two propagation effects on the received signal strength that are of interest: one of them is the spreading loss due to the propagation distance, often termed as the free space path loss; and the other is the fading due to multipath. This example models both effects.

```
[channel,interferenceTransmitter,toRxAng,spLoss] = ...
    helperMIMOEnvSetup(gc,tp);
[sigFade, chPathG] =  channel(txOFDM);
sigLoss = sigFade/sqrt(db2pow(spLoss(1)));
```

To simulate a more realistic mobile environment, the next section also inserts an interference source. Note that in a wireless communication system, the interference is often a different mobile user.

```
% Generate interference and apply gain and propagation loss
numBits = size(sigFade,1);
interfSymbols = wgn(numBits,1,1,'linear','complex');
interfSymbols = interferenceTransmitter(interfSymbols);
interfLoss = interfSymbols/sqrt(db2pow(spLoss(2)));
```

**Signal Reception**

The receiving antenna collects both the propagated signal as well as the interference and passes them to the receiver to recover the original information embedded in the signal. Just like the transmit end of the system, the receiver used in a MIMO-OFDM system also contains many stages, including OFDM demodulator, QAM demodulator, descrambler, equalizer, and Viterbi decoder.

```
[collector,receiver,demodulatorOFDM,descrambler,decoder] = ...
    helperMIMORxSetup(gc,tp,numDataSymbols);

rxSig = collector([sigLoss interfLoss],toRxAng);
```

```matlab
% Front-end amplifier gain and thermal noise
rxSig = receiver(rxSig);

rxOFDM = rxSig * ...
    (sqrt(gc.FFTLength-sum(gc.NumGuardBandCarriers)-1)) / (gc.FFTLength);

% OFDM Demodulation
rxOFDM = demodulatorOFDM(rxOFDM);

% Channel estimation
hD = helperIdealChannelEstimation(gc,  numDataSymbols, chPathG);

% Equalization
rxEq = helperEqualizer(rxOFDM, hD, numTx);

% Collapse OFDM matrix
rxSymbs = rxEq(:);

rxBitsS = qamdemod(rxSymbs,gc.modMode,'UnitAveragePower',true,...
    'OutputType','bit');
rxCoded = descrambler(rxBitsS);
rxDeCoded = decoder(rxCoded);
rxBits = rxDeCoded(1:frmSz);
```

A comparison of the decoded output with the original message stream suggests that the resulting BER is too high for a communication system. The constellation diagram is also shown below:

```matlab
ber = comm.ErrorRate;
measures = ber(txBits, rxBits);
fprintf('BER = %.2f%%; No. of Bits = %d; No. of errors = %d\n', ...
    measures(1)*100,measures(3), measures(2));
```

```
BER = 32.78%; No. of Bits = 30714; No. of errors = 10068
```

```matlab
constdiag = comm.ConstellationDiagram('SamplesPerSymbol', 1,...
    'ReferenceConstellation', [], 'ColorFading',true,...
    'Position', gc.constPlotPosition);
% Display received constellation
constdiag(rxSymbs);
```
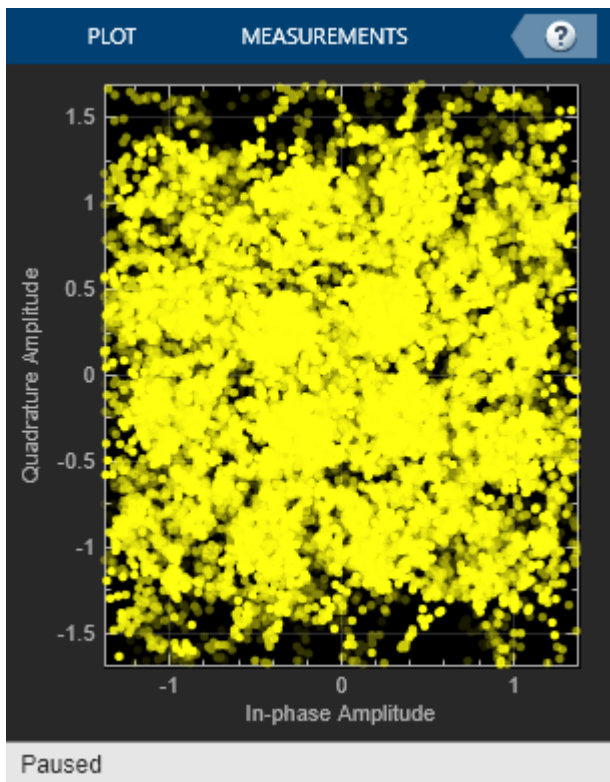
The high BER is mainly due to the mobile being off the steering direction of the base station array. If the mobile is aligned with the steering direction, the BER is greatly improved.

```
tp.steeringAngle = tp.mobileAngle;

% Steer the transmitter main lobe
wT = steeringvec(gc.fc,[tp.steeringAngle;0]);

[txBits, rxBits,rxSymbs] = helperRerunMIMOBeamformingExample(gc,tp,wT);

reset(ber);
measures = ber(txBits, rxBits);
fprintf('BER = %.2f%%; No. of Bits = %d; No. of errors = %d\n', ...
    measures(1)*100,measures(3), measures(2));

BER = 0.00%; No. of Bits = 30714; No. of errors = 0

constdiag(rxSymbs);
```
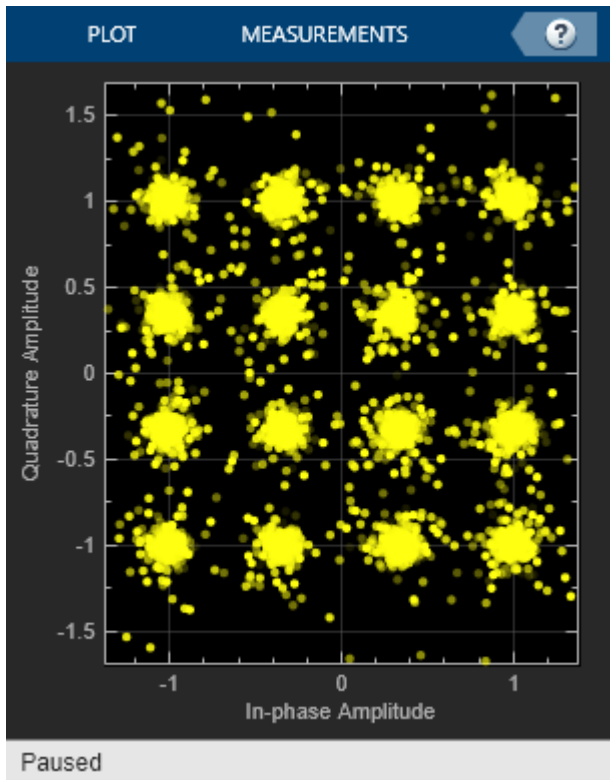
Therefore, the system is very sensitive to the steering error. On the other hand, it is this kind of spatial sensitivity that makes SDMA possible to distinguish multiple users in space.

**Phase Shifter Quantization Effect**

The discussion so far assumes that the beam can be steered toward the exact desired direction. In reality, however, this is often not true, especially when the analog phase shifters are used. Analog phase shifters have only limited precision and are categorized by the number of bits used in phase shifts. For example, a 3-bit phase shifter can only represent 8 different angles within 360 degrees. Thus, if such quantization is included in the simulation, the system performance degrades, which can be observed from the constellation plot.

```
% analog phase shifter with quantization effect
release(steeringvec);
steeringvec.NumPhaseShifterBits = 4;
wTq = steeringvec(gc.fc,[tp.steeringAngle;0]);

[txBits, rxBits,rxSymbs] = helperRerunMIMOBeamformingExample(gc,tp,wTq);

reset(ber);
measures = ber(txBits, rxBits);
fprintf('BER = %.2f%%; No. of Bits = %d; No. of errors = %d\n', ...
    measures(1)*100,measures(3), measures(2));

constdiag = comm.ConstellationDiagram('SamplesPerSymbol', 1,...
    'ReferenceConstellation', [], 'ColorFading',true,...
    'Position', gc.constPlotPosition);
constdiag(rxSymbs);

BER = 0.02%; No. of Bits = 30714; No. of errors = 5
```
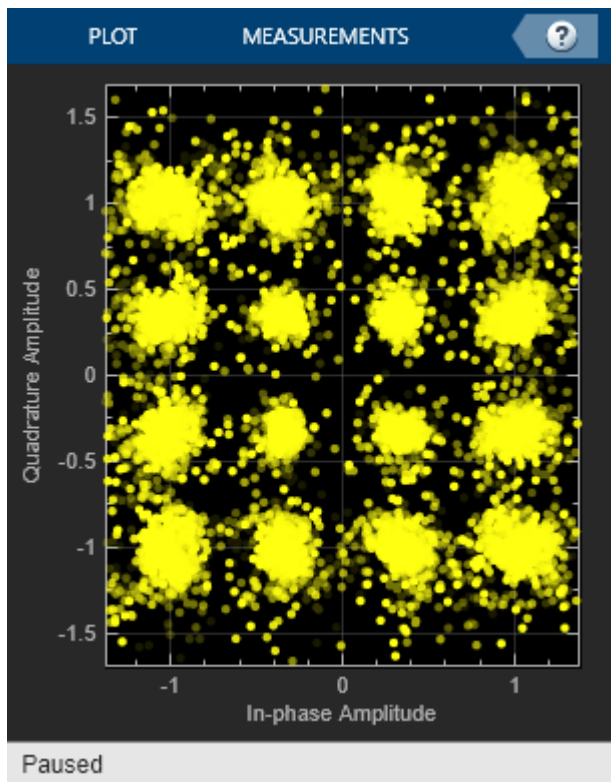
**Summary**

This example shows a system level simulation of a point-to-point MIMO-OFDM system employing beamforming. The simulation models many system components such as encoding, transmit beamforming, precoding, multipath fading, channel estimation, equalization, and decoding.

**Reference**

[1] Houman Zarrinkoub, Understanding LTE with MATLAB, Wiley, 2014

[2] Theodore S. Rappaport et al. Millimeter Wave Wireless Communications, Prentice Hall, 2014

## See Also
`phased.Radiator`

# Conventional and Adaptive Beamformers

This example illustrates how to apply digital beamforming to a narrowband signal received by an antenna array. Three beamforming algorithms are illustrated: the phase shift beamformer (PhaseShift), the minimum variance distortionless response (MVDR) beamformer, and the linearly constrained minimum variance (LCMV) beamformer.

**Simulating the Received Signal**

First, we define the incoming signal. The signal's baseband representation is a simple rectangular pulse as defined below:

```
t = 0:0.001:0.3;                % Time, sampling frequency is 1kHz
s = zeros(size(t));
s = s(:);                       % Signal in column vector
s(201:205) = s(201:205) + 1;    % Define the pulse
plot(t,s);
title('Pulse');xlabel('Time (s)');ylabel('Amplitude (V)');
```
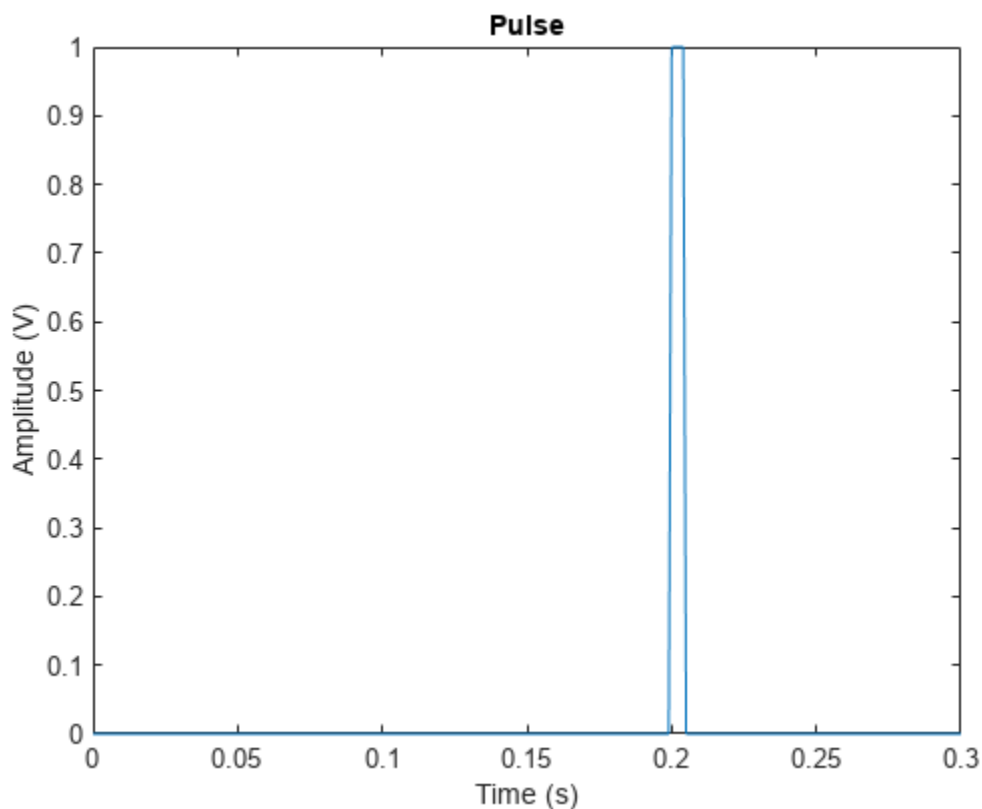


For this example, we also assume that the signal's carrier frequency is 100 MHz.

```
carrierFreq = 100e6;
wavelength = physconst('LightSpeed')/carrierFreq;
```

We now define the uniform linear array (ULA) used to receive the signal. The array contains 10 isotropic antennas. The element spacing is half of the incoming wave's wavelength.

```
ula = phased.ULA('NumElements',10,'ElementSpacing',wavelength/2);
ula.Element.FrequencyRange = [90e5 110e6];
```

Then we use the collectPlaneWave method of the array object to simulate the received signal at the array. Assume the signal arrives at the array from 45 degrees in azimuth and 0 degrees in elevation; the received signal can be modeled as

```
inputAngle = [45; 0];
x = collectPlaneWave(ula,s,inputAngle,carrierFreq);
```

The received signal often includes some thermal noise. The noise can be modeled as complex, Gaussian distributed random numbers. In this example, we assume that the noise power is 0.5 watts, which corresponds to a 3 dB signal-to-noise ratio (SNR) at each antenna element.

```
% Create and reset a local random number generator so the result is the
% same every time.
rs = RandStream.create('mt19937ar','Seed',2008);

noisePwr = .5;    % noise power
noise = sqrt(noisePwr/2)*(randn(rs,size(x))+1i*randn(rs,size(x)));
```

The total return is the received signal plus the thermal noise.

```
rxSignal = x + noise;
```

The total return has ten columns, where each column corresponds to one antenna element. The plot below shows the magnitude plot of the signal for the first two channels.

```
subplot(211);
plot(t,abs(rxSignal(:,1)));axis tight;
title('Pulse at Antenna 1');xlabel('Time (s)');ylabel('Magnitude (V)');
subplot(212);
plot(t,abs(rxSignal(:,2)));axis tight;
title('Pulse at Antenna 2');xlabel('Time (s)');ylabel('Magnitude (V)');
```
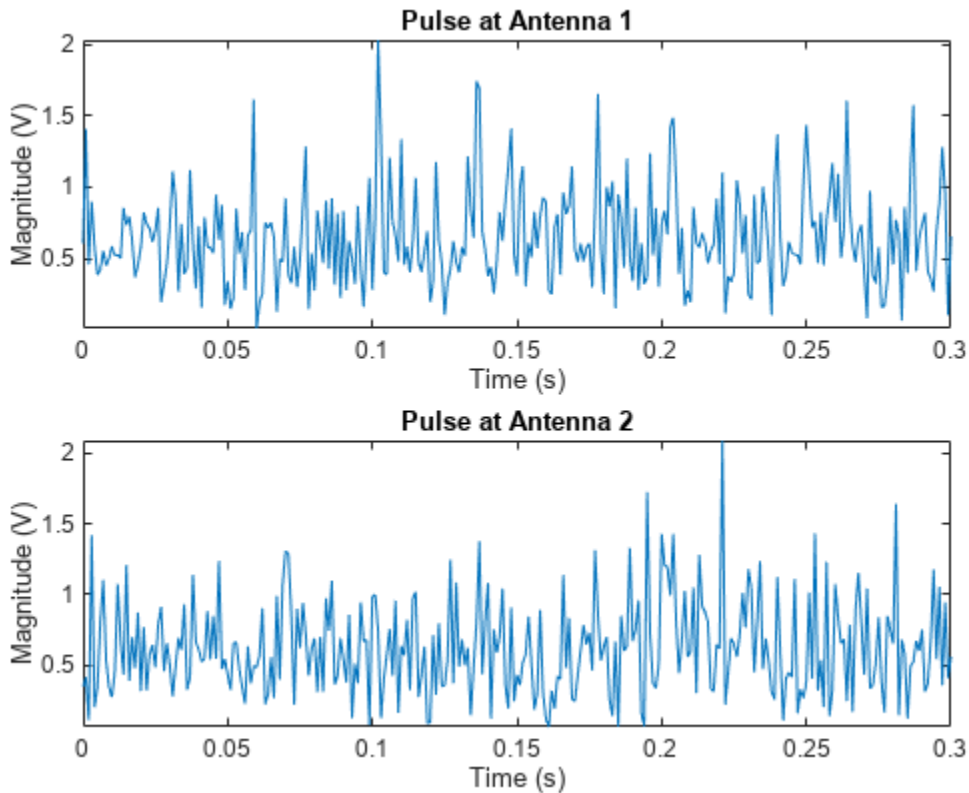
### Phase Shift Beamformer

A beamformer can be considered a spatial filter that suppresses the signal from all directions, except the desired ones. A conventional beamformer simply delays the received signal at each antenna so that the signals are aligned as if they arrive at all the antennas at the same time. In the narrowband case, this is equivalent to multiplying the signal received at each antenna by a phase factor. To define a phase shift beamformer pointing to the signal's incoming direction, we use

```
psbeamformer = phased.PhaseShiftBeamformer('SensorArray',ula,...
    'OperatingFrequency',carrierFreq,'Direction',inputAngle,...
    'WeightsOutputPort', true);
```

We can now obtain the output signal and weighting coefficients from the beamformer.

```
[yCbf,w] = psbeamformer(rxSignal);
% Plot the output
clf;
plot(t,abs(yCbf)); axis tight;
title('Output of Phase Shift Beamformer');
xlabel('Time (s)');ylabel('Magnitude (V)');
```

**Output of Phase Shift Beamformer**
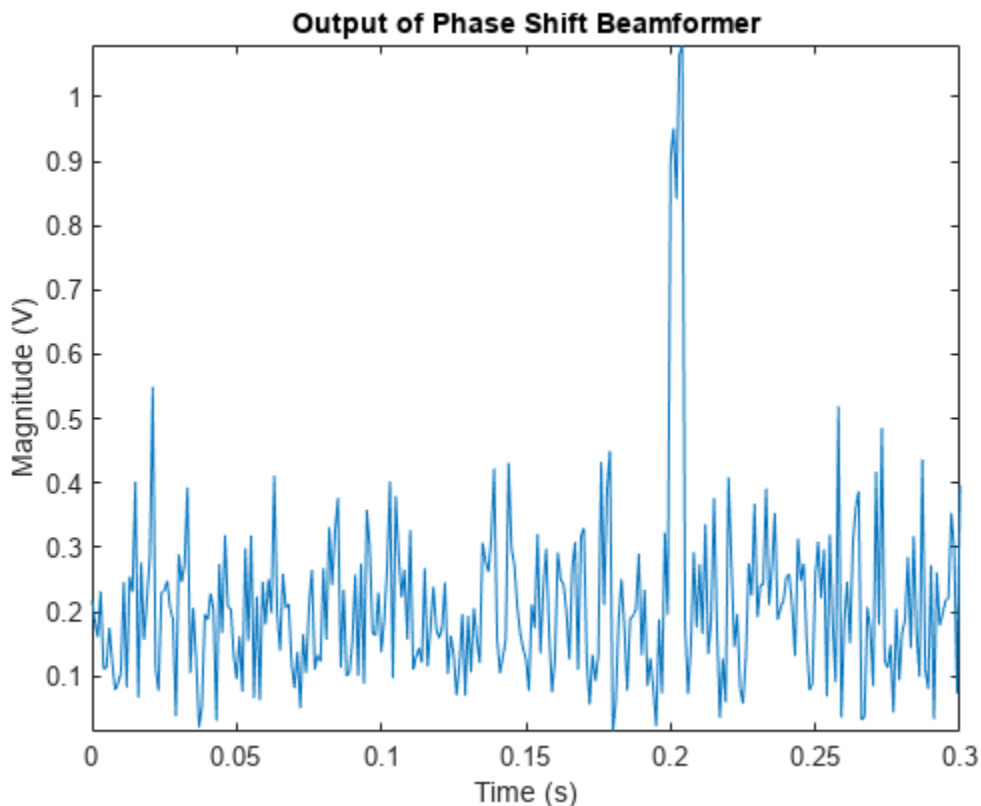


From the figure, we can see that the signal becomes much stronger compared to the noise. The output SNR is approximately 10 times stronger than that of the received signal on a single antenna, because a 10-element array produces an array gain of 10.

To see the beam pattern of the beamformer, we plot the array response along 0 degrees elevation with the weights applied. Since the array is a ULA with isotropic elements, it has ambiguity in front and back of the array. Therefore, we only plot between -90 and 90 degrees in azimuth.

```
% Plot array response with weighting
pattern(ula,carrierFreq,-180:180,0,'Weights',w,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
axis([-90 90 -60 0]);
```
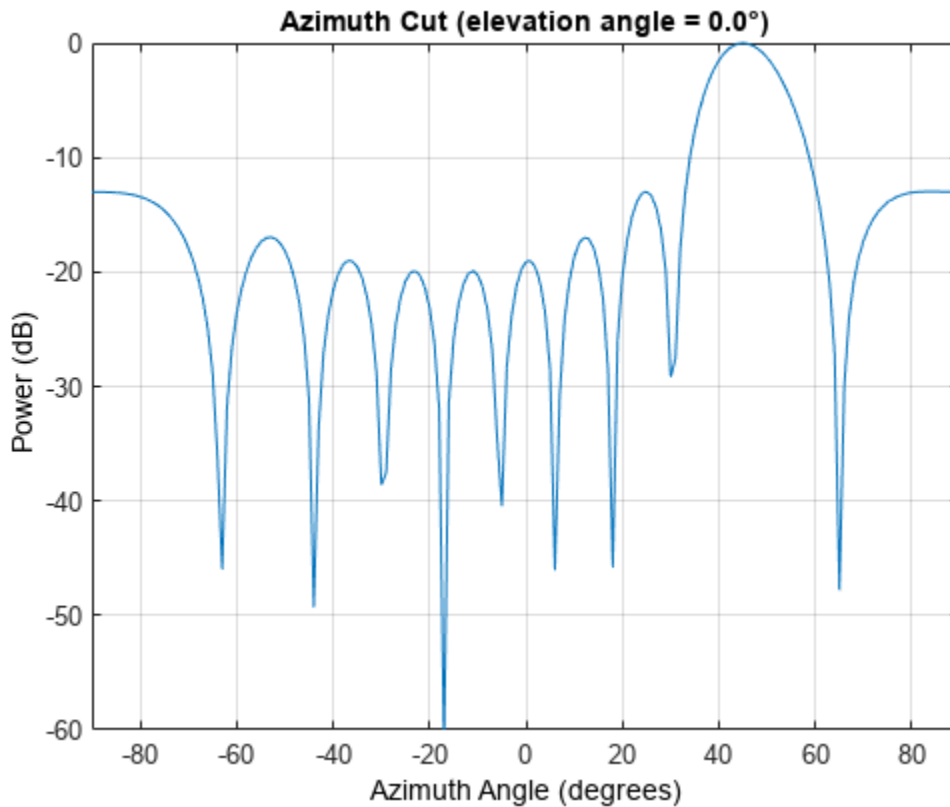
You can see that the main beam of the beamformer is pointing in the desired direction (45 degrees), as expected.

Next, we use the beamformer to enhance the received signal under interference conditions. In the presence of strong interference, the target signal may be masked by the interference signal. For example, interference from a nearby radio tower can blind the antenna array in that direction. If the radio signal is strong enough, it can blind the radar in multiple directions, especially when the desired signal is received by a sidelobe. Such scenarios are very challenging for a phase shift beamformer, and therefore, adaptive beamformers are introduced to address this problem.

**Modeling the Interference Signals**

We model two interference signals arriving from 30 degrees and 50 degrees in azimuth. The interference amplitudes are much higher than the desired signal shown in the previous scenario.

```
nSamp = length(t);
s1 = 10*randn(rs,nSamp,1);
s2 = 10*randn(rs,nSamp,1);
% interference at 30 degrees and 50 degrees
interference = collectPlaneWave(ula,[s1 s2],[30 50; 0 0],carrierFreq);
```

To illustrate the effect of interference, we'll reduce the noise level to a minimal level. For the rest of the example, let us assume a high SNR value of 50dB at each antenna. We'll see that even though there is almost no noise, the interference alone can make a phase shift beamformer fail.

```
noisePwr = 0.00001;    % noise power, 50dB SNR
noise = sqrt(noisePwr/2)*(randn(rs,size(x))+1i*randn(rs,size(x)));
```

```
rxInt = interference + noise;                    % total interference + noise
rxSignal = x + rxInt;                  % total received Signal
```

First, we'll try to apply the phase shift beamformer to retrieve the signal along the incoming direction.

```
yCbf = psbeamformer(rxSignal);
```

```
plot(t,abs(yCbf)); axis tight;
title('Output of Phase Shift Beamformer With Presence of Interference');
xlabel('Time (s)');ylabel('Magnitude (V)');
```
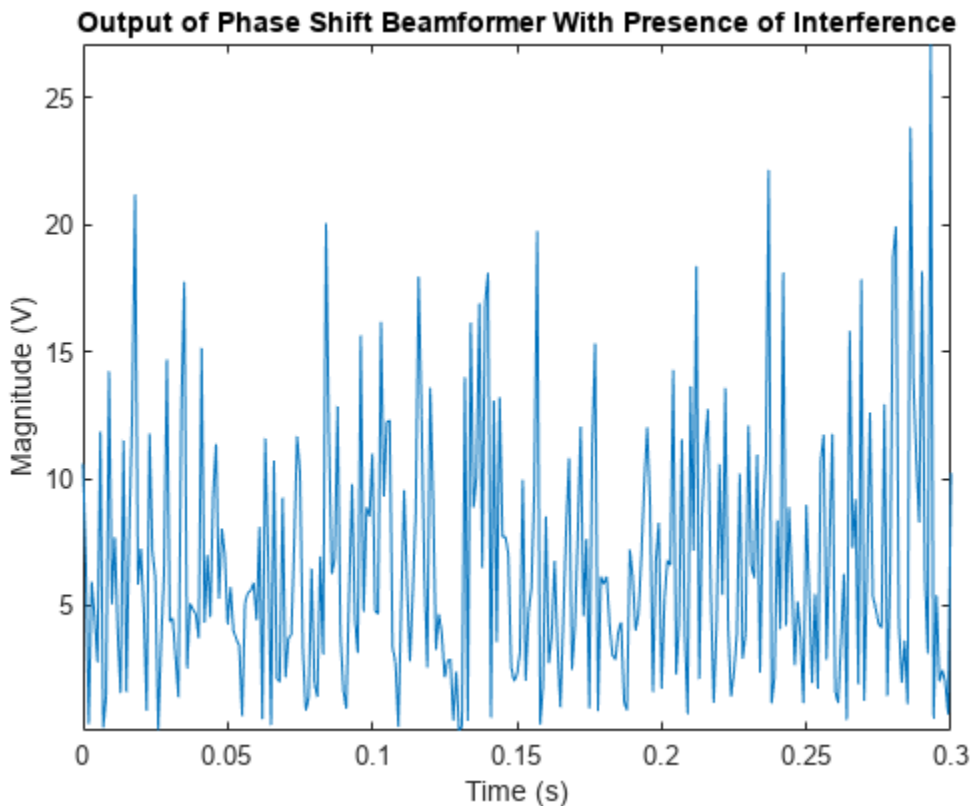


From the figure, we can see that, because the interference signals are much stronger than the target signal, we cannot extract the signal content.

**MVDR Beamformer**

To overcome the interference problem, we can use the MVDR beamformer, a popular adaptive beamformer. The MVDR beamformer preserves the signal arriving along a desired direction, while trying to suppress signals coming from other directions. In this case, the desired signal is at the direction 45 degrees in azimuth.

```
% Define the MVDR beamformer
mvdrbeamformer = phased.MVDRBeamformer('SensorArray',ula,...
    'Direction',inputAngle,'OperatingFrequency',carrierFreq,...
    'WeightsOutputPort',true);
```

When we have access to target-free data, we can provide such information to the MVDR beamformer by setting the TrainingInputPort property to true.

```
mvdrbeamformer.TrainingInputPort = true;
```

We apply the MVDR beamformer to the received signal. The plot shows the MVDR beamformer output signal. You can see that the target signal can now be recovered.

```
[yMVDR, wMVDR] = mvdrbeamformer(rxSignal,rxInt);

plot(t,abs(yMVDR)); axis tight;
title('Output of MVDR Beamformer With Presence of Interference');
xlabel('Time (s)');ylabel('Magnitude (V)');
```
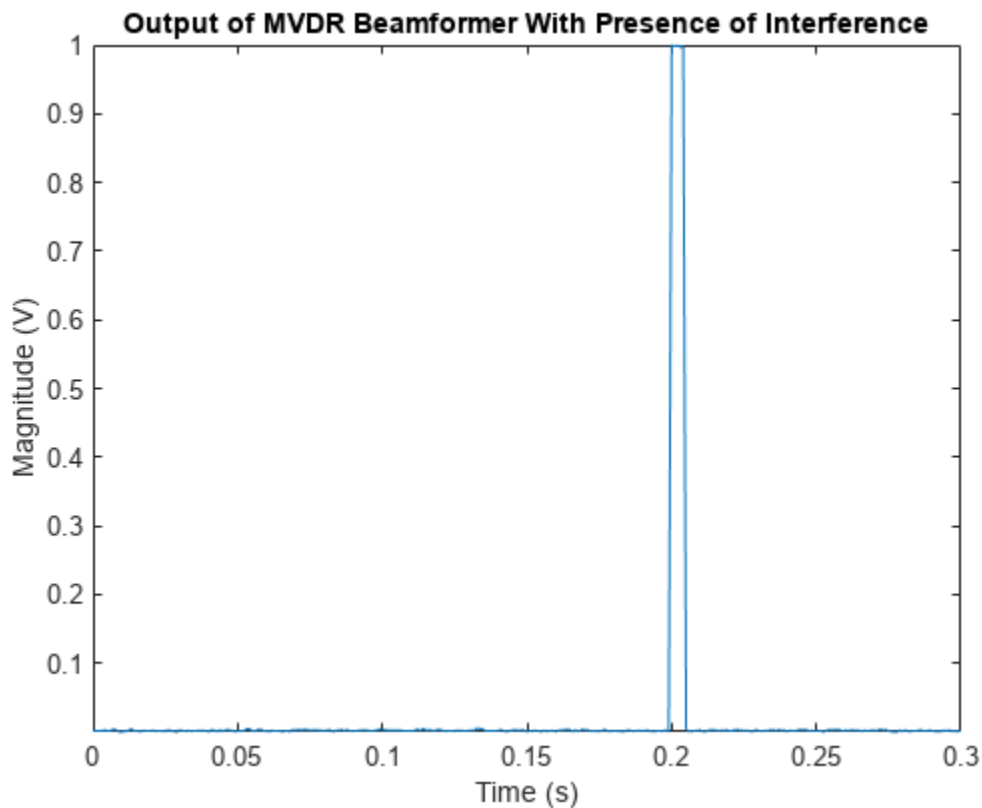


Looking at the response pattern of the beamformer, we see two deep nulls along the interference directions (30 and 50 degrees). The beamformer also has a gain of 0 dB along the target direction of 45 degrees. Thus, the MVDR beamformer preserves the target signal and suppresses the interference signals.

```
pattern(ula,carrierFreq,-180:180,0,'Weights',wMVDR,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
axis([-90 90 -80 20]);

hold on;   % compare to PhaseShift
pattern(ula,carrierFreq,-180:180,0,'Weights',w,...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
```

```
    'Type','powerdb','CoordinateSystem','rectangular');
hold off;
legend('MVDR','PhaseShift')
```
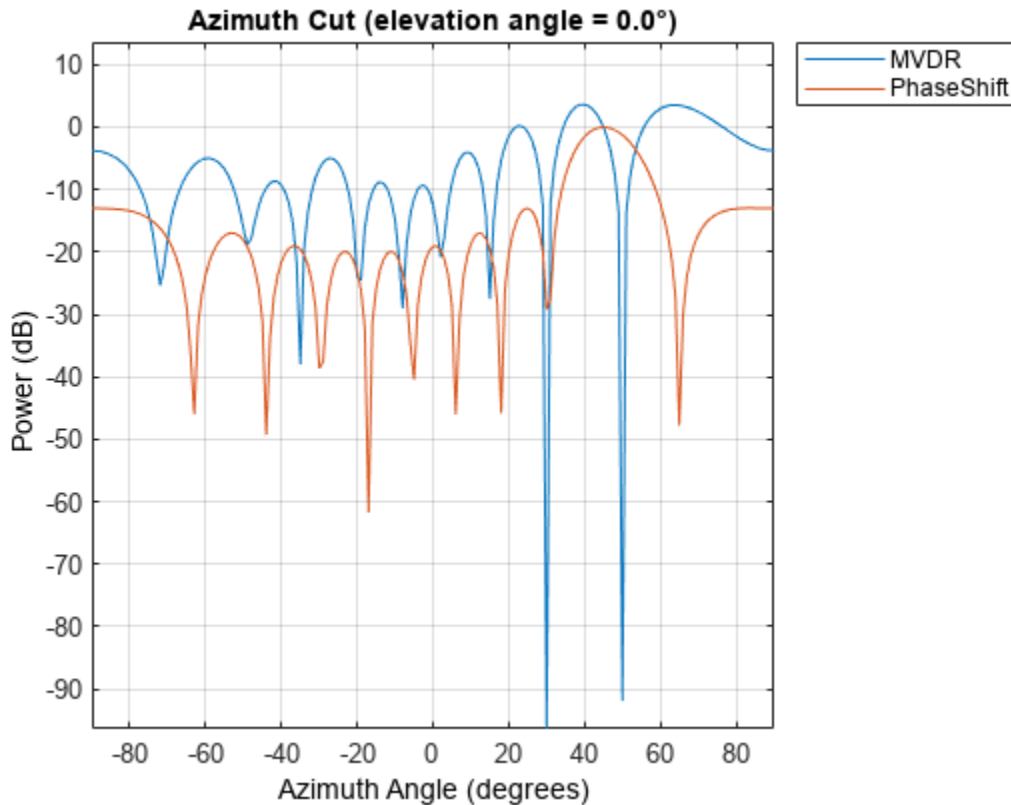


Also shown in the figure is the response pattern from PhaseShift. We can see that the PhaseShift pattern does not null out the interference at all.

**Self Nulling Issue in MVDR**

On many occasions, we may not be able to separate the interference from the target signal, and therefore, the MVDR beamformer has to calculate weights using data that includes the target signal. In this case, if the target signal is received along a direction slightly different from the desired one, the MVDR beamformer suppresses it. This occurs because the MVDR beamformer treats all the signals, except the one along the desired direction, as undesired interferences. This effect is sometimes referred to as "signal self nulling".

To illustrate this self nulling effect, we define an MVDR beamformer and set the TrainingInputPort property to false.

```
mvdrbeamformer_selfnull = phased.MVDRBeamformer('SensorArray',ula,...
    'Direction',inputAngle,'OperatingFrequency',carrierFreq,...
    'WeightsOutputPort',true,'TrainingInputPort',false);
```

We then create a direction mismatch between the incoming signal direction and the desired direction.

Recall that the signal is impinging from 45 degrees in azimuth. If, with some a priori information, we expect the signal to be arriving from 43 degrees in azimuth, then we use 43 degrees in azimuth as the

desired direction in the MVDR beamformer. However, since the real signal is arriving at 45 degrees in azimuth, there is a slight mismatch in the signal direction.

```
expDir = [43; 0];
mvdrbeamformer_selfnull.Direction = expDir;
```

When we apply the MVDR beamformer to the received signal, we see that the receiver cannot differentiate the target signal and the interference.

```
[ySn, wSn] = mvdrbeamformer_selfnull(rxSignal);
```

```
plot(t,abs(ySn)); axis tight;
title('Output of MVDR Beamformer With Signal Direction Mismatch');
xlabel('Time (s)');ylabel('Magnitude (V)');
```
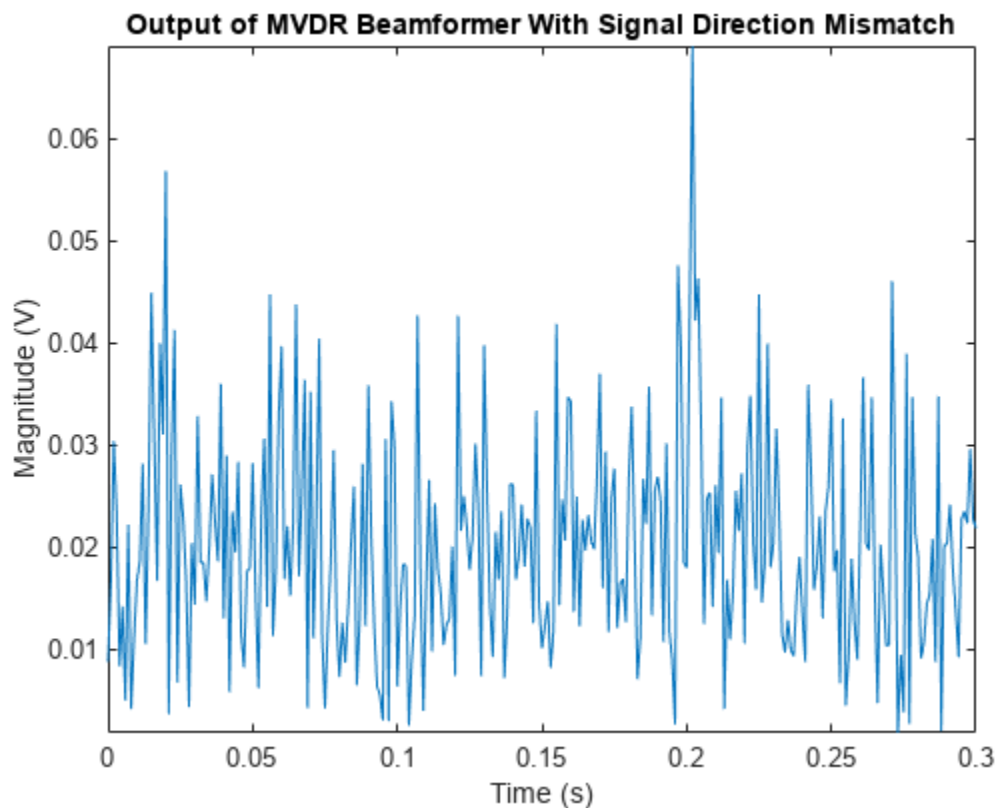


When we look at the beamformer response pattern, we see that the MVDR beamformer tries to suppress the signal arriving along 45 degrees because it is treated like an interference signal. The MVDR beamformer is very sensitive to signal-steering vector mismatch, especially when we cannot provide interference information.

```
pattern(ula,carrierFreq,-180:180,0,'Weights',wSn,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
axis([-90 90 -40 25]);
```

**Azimuth Cut (elevation angle = 0.0°)**
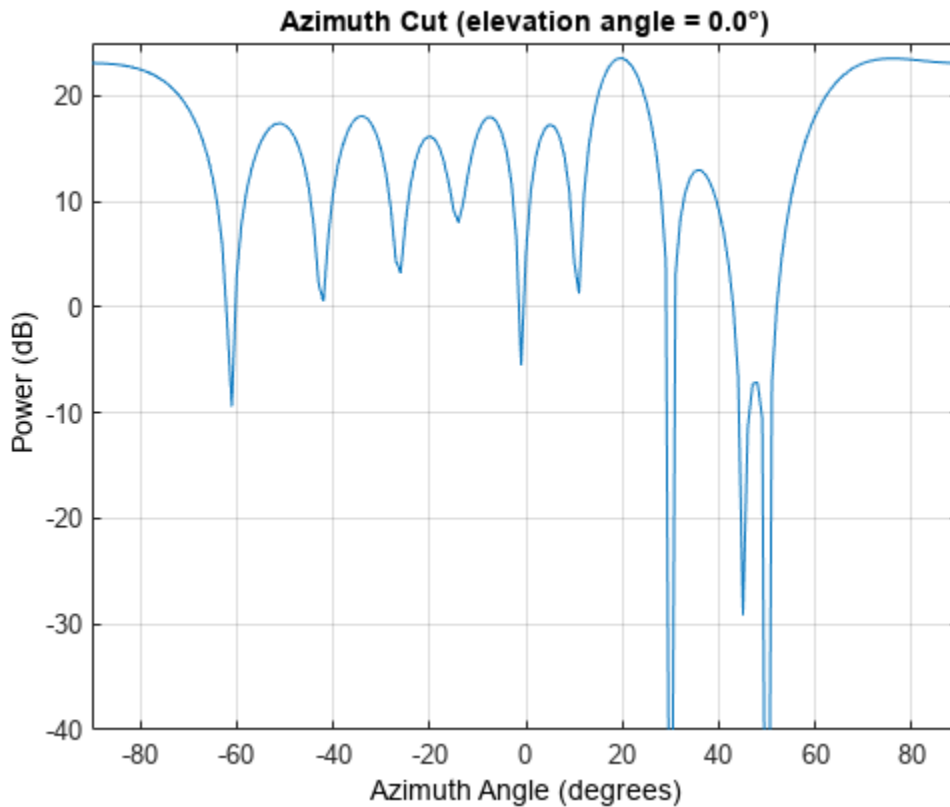


**LCMV Beamformer**

To prevent signal self-nulling, we can use an LCMV beamformer, which allows us to put multiple constraints along the target direction (steering vector). It reduces the chance that the target signal will be suppressed when it arrives at a slightly different angle from the desired direction. First we create an LCMV beamformer:

```
lcmvbeamformer = phased.LCMVBeamformer('WeightsOutputPort',true);
```

Now we need to create several constraints. To specify a constraint, we put corresponding entries in both the constraint matrix, Constraint, and the desired response vector, DesiredResponse. Each column in Constraint is a set of weights we can apply to the array and the corresponding entry in DesiredResponse is the desired response we want to achieve when the weights are applied. For example, to avoid self nulling in this example, we may want to add the following constraints to the beamformer:

- Preserve the incoming signal from the expected direction (43 degrees in azimuth).

- To avoid self nulling, ensure that the response of the beamformer will not decline at +/- 2 degrees of the expected direction.

For all the constraints, the weights are given by the steering vectors that steer the array toward those directions:

```
steeringvec = phased.SteeringVector('SensorArray',ula);
stv = steeringvec(carrierFreq,[43 41 45]);
```

The desired responses should be 1 for all three directions. The Constraint matrix and DesiredResponse are given by:

```
lcmvbeamformer.Constraint = stv;
lcmvbeamformer.DesiredResponse = [1; 1; 1];
```

Then we apply the beamformer to the received signal. The plot below shows that the target signal can be detected again even though there is the mismatch between the desired and the true signal arriving direction.

```
[yLCMV,wLCMV] = lcmvbeamformer(rxSignal);

plot(t,abs(yLCMV)); axis tight;
title('Output of LCMV Beamformer With Signal Direction Mismatch');
xlabel('Time (s)');ylabel('Magnitude (V)');
```
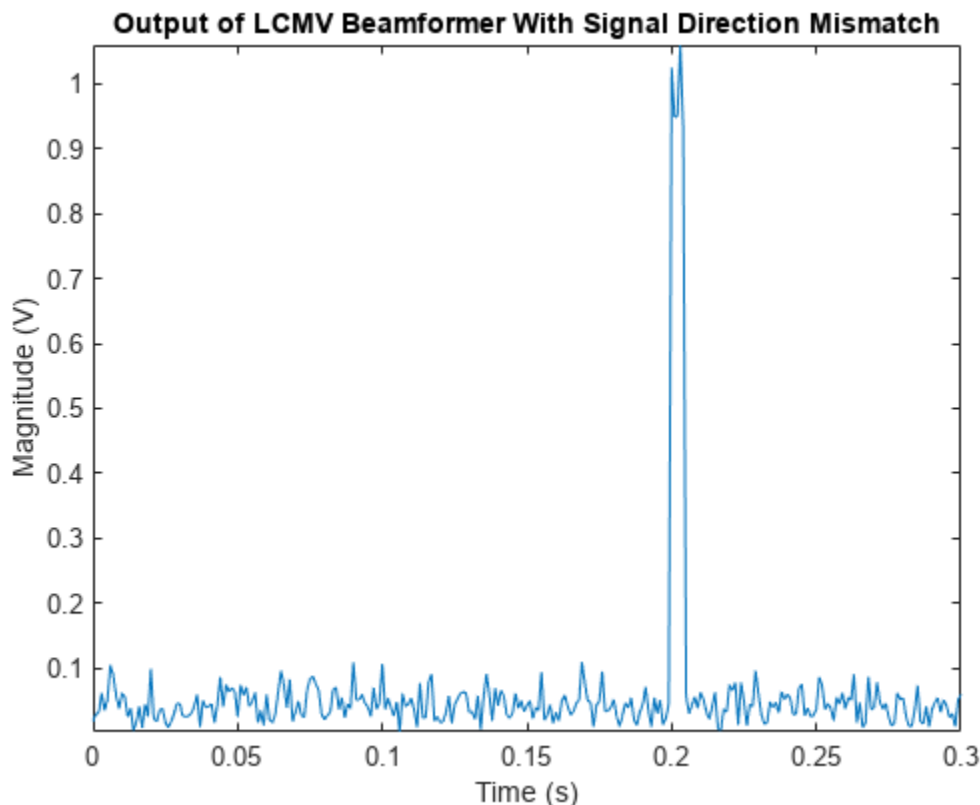


The LCMV response pattern shows that the beamformer puts the constraints along the specified directions, while nulling the interference signals along 30 and 50 degrees. Here we only show the pattern between 0 and 90 degrees in azimuth so that we can see the behavior of the response pattern at the signal and interference directions better.

```
pattern(ula,carrierFreq,-180:180,0,'Weights',wLCMV,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
axis([0 90 -40 35]);

hold on;  % compare to MVDR
```

**18-211**

```
pattern(ula,carrierFreq,-180:180,0,'Weights',wSn,...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'Type','powerdb','CoordinateSystem','rectangular');
hold off;
legend('LCMV','MVDR');
```
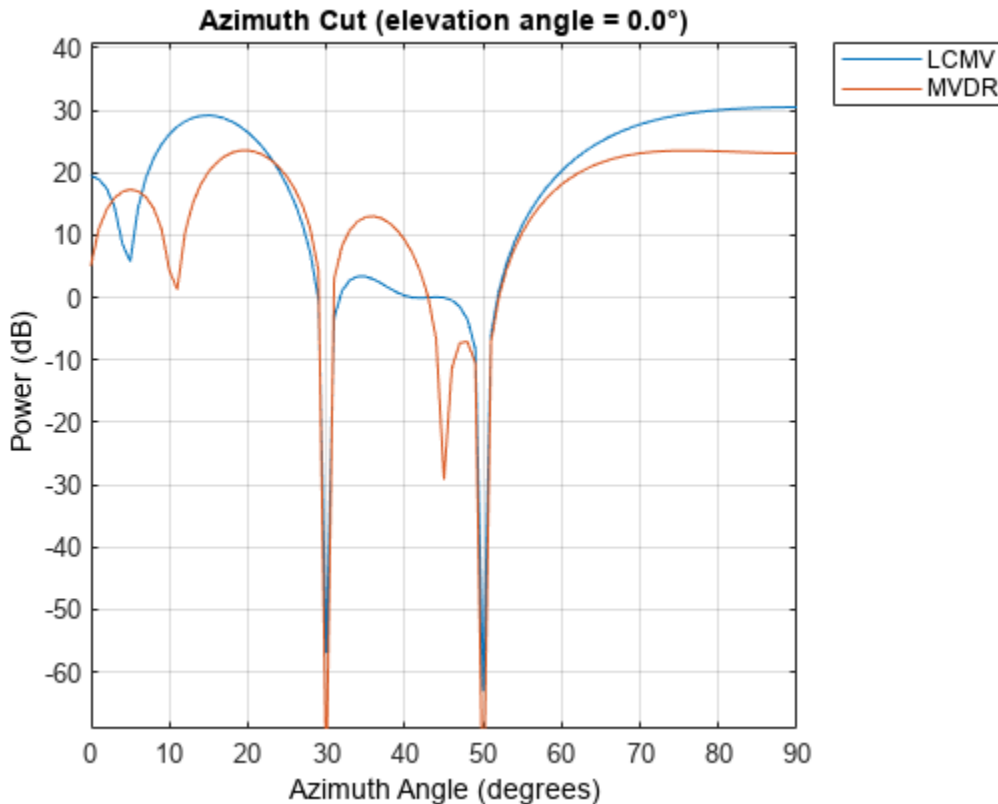


The effect of constraints can be better seen when comparing the LCMV beamformer's response pattern to the MVDR beamformer's response pattern. Notice how the LCMV beamformer is able to maintain a flat response region around the 45 degrees in azimuth, while the MVDR beamformer creates a null.

**2D Array Beamforming**

In this section, we illustrate the use of a beamformer with a uniform rectangular array (URA). The beamformer can be applied to a URA in the same way as to the ULA. We only illustrate the MVDR beamformer for the URA in this example. The usages of other algorithms are similar.

First, we define a URA. The URA consists of 10 rows and 5 columns of isotropic antenna elements. The spacing between the rows and the columns are 0.4 and 0.5 wavelength, respectively.

```
colSp = 0.5*wavelength;
rowSp = 0.4*wavelength;
ura = phased.URA('Size',[10 5],'ElementSpacing',[rowSp colSp]);
ura.Element.FrequencyRange = [90e5 110e6];
```

Consider the same source signal as was used in the previous sections. The source signal arrives at the URA from 45 degrees in azimuth and 0 degrees in elevation. The received signal, including the noise at the array, can be modeled as

```
x = collectPlaneWave(ura,s,inputAngle,carrierFreq);
noise = sqrt(noisePwr/2)*(randn(rs,size(x))+1i*randn(rs,size(x)));
```

Unlike a ULA, which can only differentiate the angles in azimuth direction, a URA can also differentiate angles in elevation direction. Therefore, we specify two interference signals arriving along the directions [30;10] and [50;-5] degrees, respectively.

```
s1 = 10*randn(rs,nSamp,1);
s2 = 10*randn(rs,nSamp,1);

%interference at [30; 10] and at [50; -5]
interference = collectPlaneWave(ura,[s1 s2],[30 50; 10 -5],carrierFreq);
rxInt = interference + noise;                    % total interference + noise
rxSignal = x + rxInt;                    % total received signal
```

We now create an MVDR beamformer pointing to the target signal direction.

```
mvdrbeamformer = phased.MVDRBeamformer('SensorArray',ura,...
    'Direction',inputAngle,'OperatingFrequency',carrierFreq,...
    'TrainingInputPort',true,'WeightsOutputPort',true);
```

Finally, we apply the MVDR beamformer to the received signal and plot its output.

```
[yURA,w]= mvdrbeamformer(rxSignal,rxInt);

plot(t,abs(yURA)); axis tight;
title('Output of MVDR Beamformer for URA');
xlabel('Time (s)');ylabel('Magnitude (V)');
```
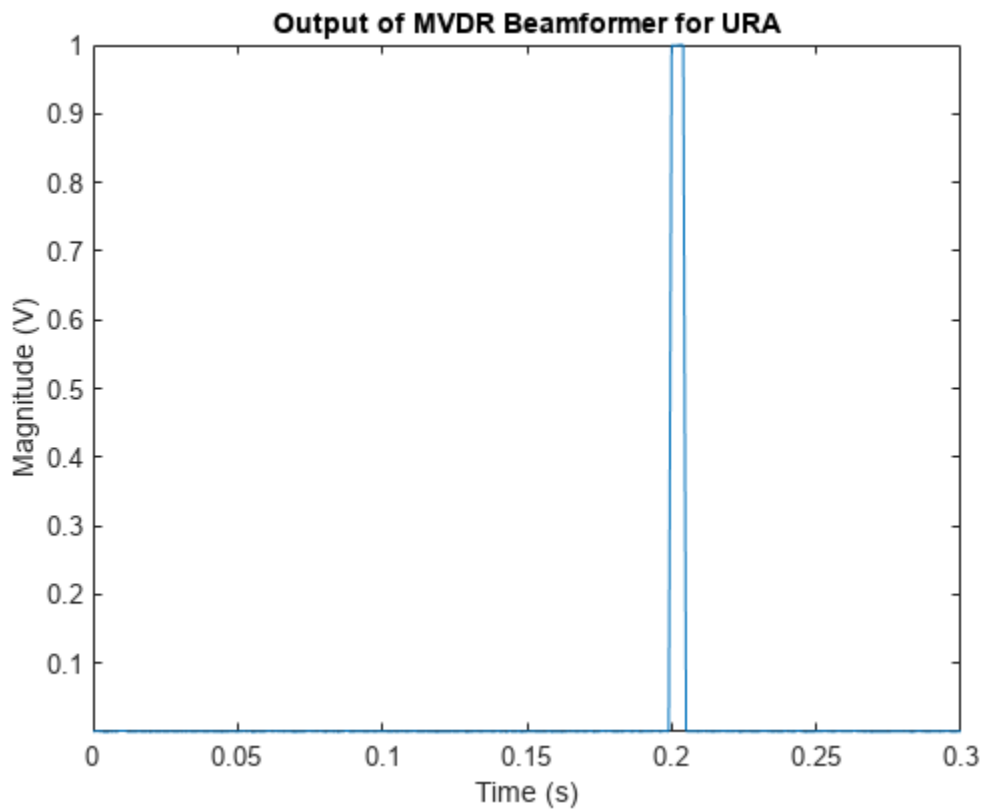
To see clearly that the beamformer puts the nulls along the interference directions, we'll plot the beamformer response pattern of the array along -5 degrees and 10 degrees in elevation, respectively. The figure shows that the beamformer suppresses the interference signals along [30 10] and [50 -5] directions.

```
subplot(2,1,1);
pattern(ura,carrierFreq,-180:180,-5,'Weights',w,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
title('Response Pattern at -5 Degrees Elevation');
axis([-90 90 -60 -5]);
subplot(2,1,2);
pattern(ura,carrierFreq,-180:180,10,'Weights',w,'Type','powerdb',...
    'PropagationSpeed',physconst('LightSpeed'),'Normalize',false,...
    'CoordinateSystem','rectangular');
axis([-90 90 -60 -5]);
title('Response Pattern at 10 Degrees Elevation');
```
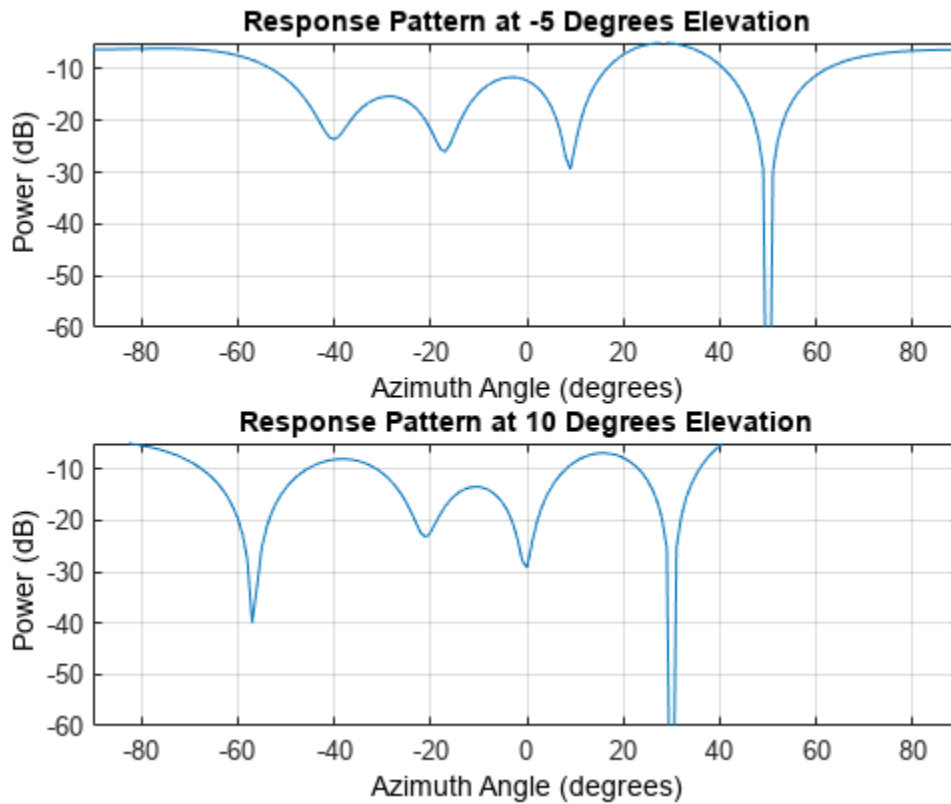
**Summary**

In this example, we illustrated how to use a beamformer to retrieve the signal from a particular direction using a ULA or a URA. The choice of the beamformer depends on the operating environment. Adaptive beamformers provide superior interference rejection compared to that offered by conventional beamformers. When the knowledge about the target direction is not accurate, the LCMV beamformer is preferable because it prevents signal self-nulling.

# Direction of Arrival Estimation with Beamscan, MVDR, and MUSIC

This example illustrates using beamscan, MVDR, and MUSIC for direction of arrival (DOA) estimation. Beamscan is a technique that forms a conventional beam and scans it over directions of interest to obtain a spatial spectrum. Minimum variance distortionless response (MVDR) is similar to beamscan but uses an MVDR beam. Multiple signal classification (MUSIC) is a subspace method that provides high resolution DOA estimates. For all three methods, the peaks of the output spatial spectrum indicate the DOAs of the received signals. In this example, we illustrate the use of beamscan, MVDR, and MUSIC to estimate broadside angles with a uniform linear array (ULA) and azimuth and elevation angles with a uniform rectangular array (URA).

**Modeling ULA's Received Signal**

First, model a uniform linear array (ULA) containing 10 isotropic antennas spaced 0.5 meters apart.

```
ula = phased.ULA('NumElements',10,'ElementSpacing',0.5);
```

Assume that two narrowband signals impinge on the array. The first signal arrives from 40° in azimuth and 0° in elevation, while the second signal arrives from -20° in azimuth and 0° in elevation. The operating frequency of the system is 300 MHz.

```
ang1 = [40; 0];           % First signal
ang2 = [-20; 0];          % Second signal
angs = [ang1 ang2];

c = physconst('LightSpeed');
fc = 300e6;               % Operating frequency
lambda = c/fc;
pos = getElementPosition(ula)/lambda;

Nsamp = 1000;
```

Also, assume that the thermal noise power at each antenna is 0.01 Watts.

```
nPower = 0.01;
```

Generate the multichannel signal received by the ULA.

```
rs = rng(2007);
signal = sensorsig(pos,Nsamp,angs,nPower);
```

**Beamscan DOA Estimation with a ULA**

We want to estimate the two DOAs using the received signal. Because the signal is received by a ULA, which is symmetric around its axis, we cannot obtain both azimuth and elevation at the same time. Instead, we can estimate the broadside angle, which is measured from the broadside of the ULA. The relationship between these angles is illustrated in the following figure:
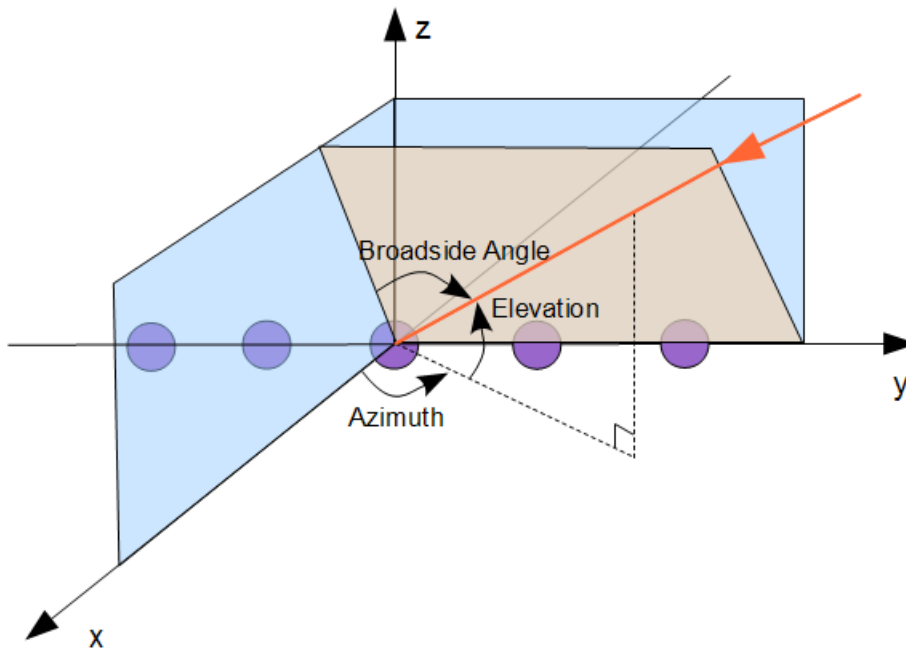
The broadside angles corresponding to the two incident directions are:

```
broadsideAngle = az2broadside(angs(1,:),angs(2,:))
```

```
broadsideAngle =

    40.0000   -20.0000
```

We can see that the two broadside angles are the same as the azimuth angles. In general, when the elevation angle is zero and the azimuth angle is within [-90 90], the broadside angle is the same as the azimuth angle. In the following we only perform the conversion when they are not equal.

The beamscan algorithm scans a conventional beam through a predefined scan region. Here we set the scanning region to [-90 90] to cover all 180 degrees.

```
spatialspectrum = phased.BeamscanEstimator('SensorArray',ula,...
            'OperatingFrequency',fc,'ScanAngles',-90:90);
```

By default, the beamscan estimator only produces a spatial spectrum across the scan region. Set the DOAOutputPort property to true to obtain DOA estimates. Set the NumSignals property to 2 to find the locations of the top two peaks.

```
spatialspectrum.DOAOutputPort = true;
spatialspectrum.NumSignals = 2;
```

We now obtain the spatial spectrum and the DOAs. The estimated DOAs show the correct values, which are 40° and -20°.

```
[~,ang] = spatialspectrum(signal)
```

```
ang =

    40    -20
```

Plot the spatial spectrum of the beamscan output.

```
plotSpectrum(spatialspectrum);
```
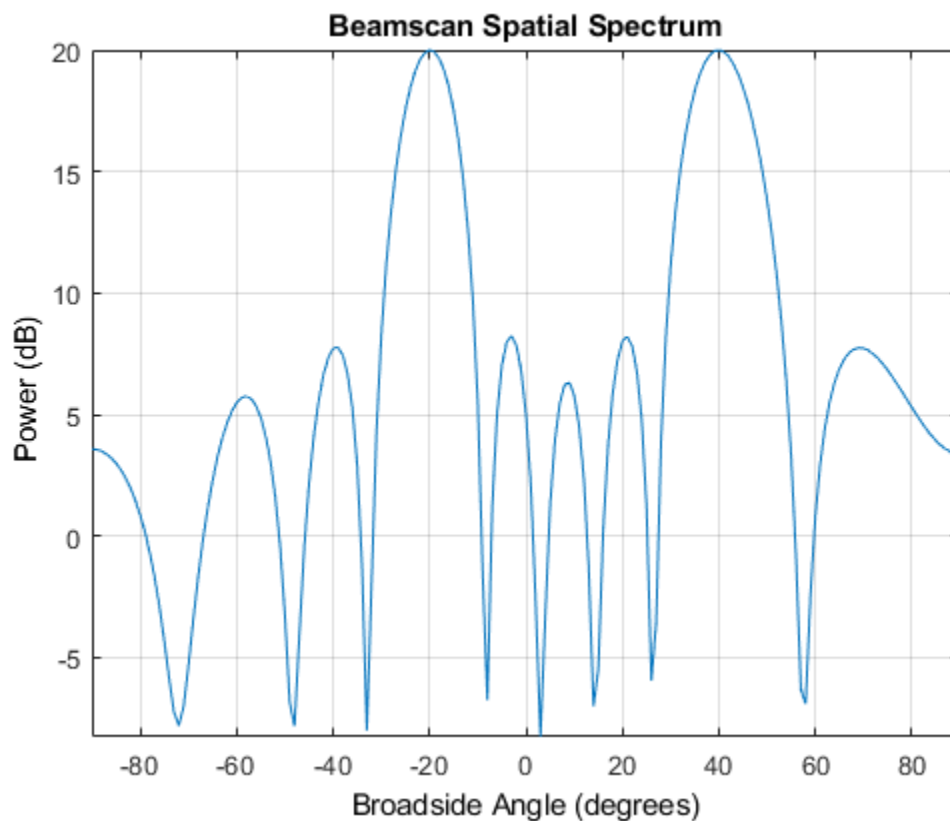


**Improving Resolution Using MVDR and MUSIC Estimators**

The conventional beam cannot resolve two closely-spaced signals. When two signals arrive from directions separated by less than the beamwidth, beamscan will fail to estimate the directions of the signals. To illustrate this limitation, we simulate two received signals from 30° and 40° in azimuth.

```
ang1 = [30; 0]; ang2 = [40; 0];
signal = sensorsig(pos,Nsamp,[ang1 ang2],nPower);
```

```
[~,ang] = spatialspectrum(signal)
```

```
ang =
```

```
        35      71
```

The results differ from the true azimuth angles. Let's take a look at the output spectrum.

```
plotSpectrum(spatialspectrum);
```
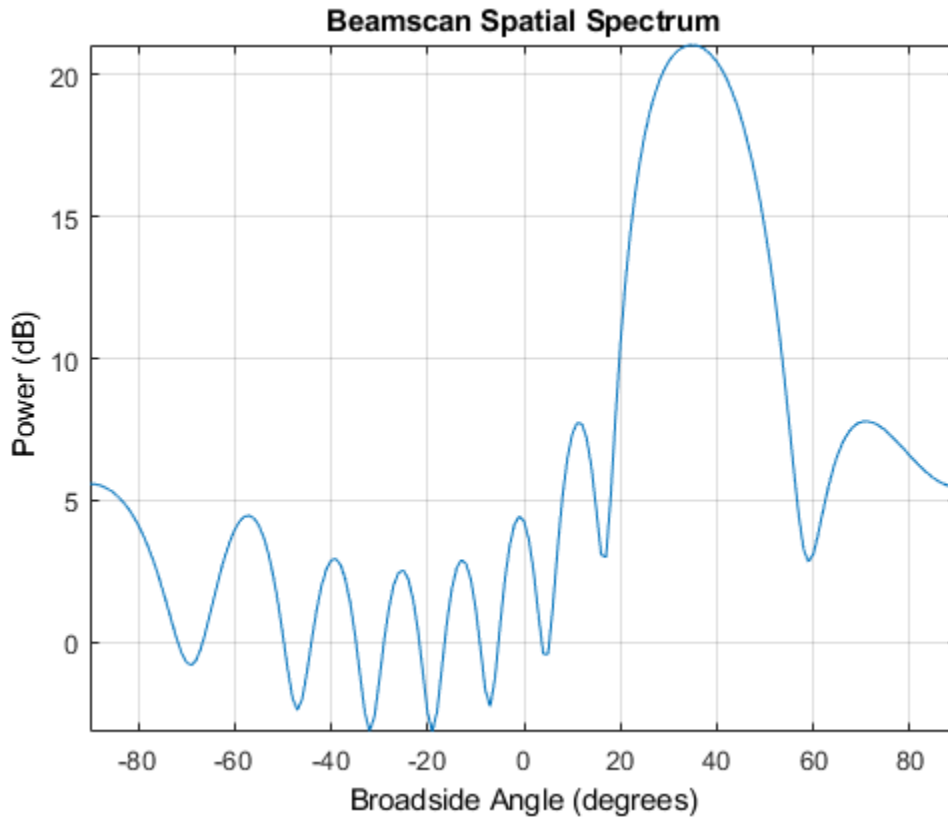


The output spatial spectrum has only one dominant peak. Therefore, it cannot resolve these two closely-spaced signals. When we try to estimate the DOA from the peaks of the beamscan output, we get incorrect estimates. The beamscan object returns two maximum peaks as the estimated DOAs no matter how different the peaks are. In this case, the beamscan returns the small peak at 71° as the second estimate.

To resolve closely-spaced signals, we can use the minimum variance distortionless response (MVDR) algorithm or the multiple signal classification (MUSIC) algorithm. First, we will examine the MVDR estimator, which scans an MVDR beam over the specified region. Because an MVDR beam has a smaller beamwidth, it has higher resolution.

```
mvdrspatialspect = phased.MVDREstimator('SensorArray',ula,...
        'OperatingFrequency',fc,'ScanAngles',-90:90,...
        'DOAOutputPort',true,'NumSignals',2);
[~,ang] = mvdrspatialspect(signal)
plotSpectrum(mvdrspatialspect);
```

```
ang =
```

30    40

**MVDR Spatial Spectrum**
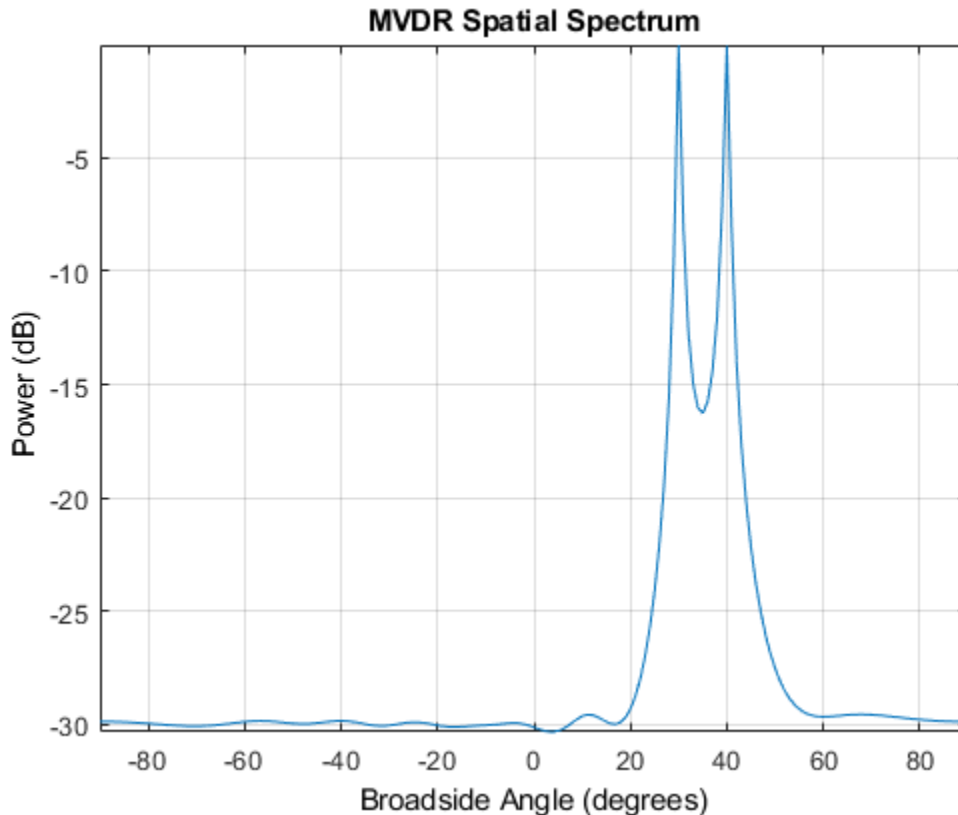


The MVDR algorithm correctly estimates the DOAs that are unresolvable by beamscan. The improved resolution comes with a price. The MVDR is more sensitive to sensor position errors. In circumstances where sensor positions are inaccurate, MVDR could produce a worse spatial spectrum than beamscan. Moreover, if we further reduce the difference of two signal directions to a level that is smaller than the beamwidth of an MVDR beam, the MVDR estimator will also fail.

The MUSIC algorithm can also be used to resolve these closely-spaced signals. Estimate the directions of arrival of the two sources and compare the spatial spectrum of MVDR to the spatial spectrum of MUSIC.

```
musicspatialspect = phased.MUSICEstimator('SensorArray',ula,...
        'OperatingFrequency',fc,'ScanAngles',-90:90,...
        'DOAOutputPort',true,'NumSignalsSource','Property','NumSignals',2);
[~,ang] = musicspatialspect(signal)
ymvdr = mvdrspatialspect(signal);
ymusic = musicspatialspect(signal);
helperPlotDOASpectra(mvdrspatialspect.ScanAngles,...
  musicspatialspect.ScanAngles,ymvdr,ymusic,'ULA')
```

ang =

30      40



DOA Spatial Spectra
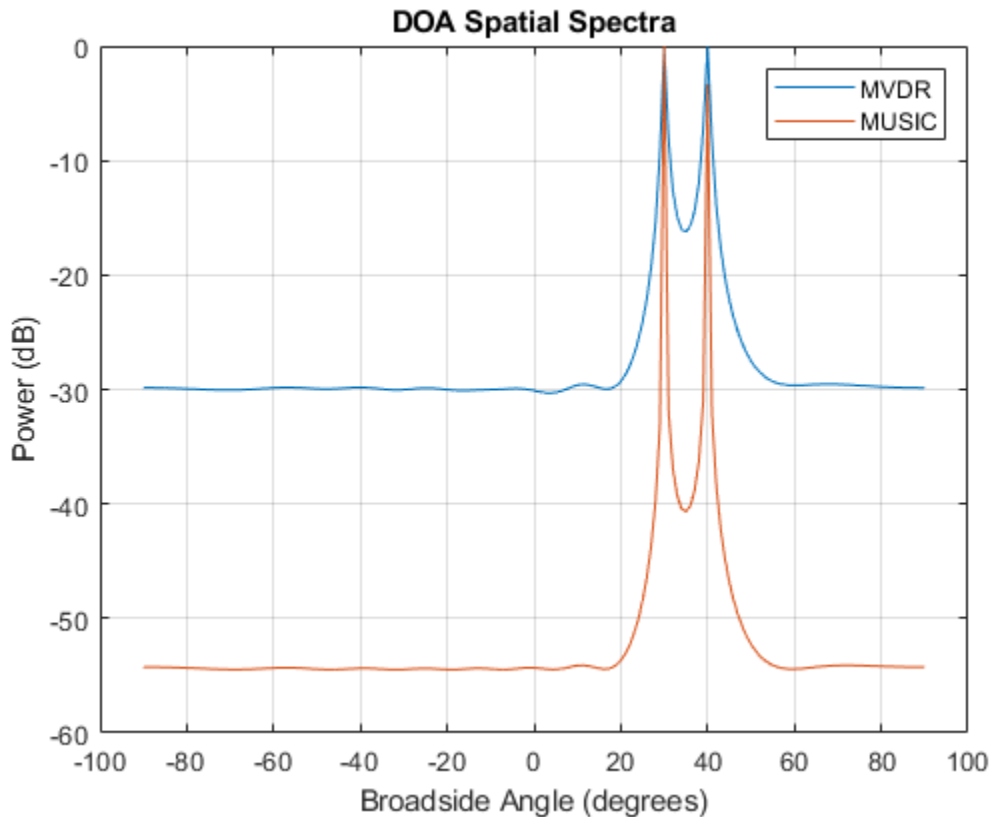
The directions of arrival using MUSIC are correct, and MUSIC provides even better spatial resolution than MVDR. MUSIC, like MVDR, is sensitive to sensor position errors. In addition, the number of sources must be known or accurately estimated. When the number of sources specified is incorrect, MVDR and Beamscan may simply return insignificant peaks from the correct spatial spectrum. In contrast, the MUSIC spatial spectrum itself may be inaccurate when the number of sources is not specified correctly. In addition, the amplitudes of MUSIC spectral peaks cannot be interpreted as the power of the sources.

For a ULA, additional high resolution algorithms can further exploit the special geometry of the ULA. See "High Resolution Direction of Arrival Estimation" on page 18-227.

**Converting Broadside Angles to Azimuth**

Although we can only estimate broadside angles using a ULA, we can convert the estimated broadside angles to azimuth angles if we know their incoming elevations. We now model two signals coming from 35° in elevation and estimate their corresponding broadside angles.

ang1 = [40; 35]; ang2 = [15; 35];

signal = sensorsig(pos,Nsamp,[ang1 ang2],nPower);
[~,ang] = mvdrspatialspect(signal)

ang =

```
    32    12
```

The resulting broadside angles are different from either the azimuth or elevation angles. We can convert the broadside angles to the azimuth angles if we know the elevation.

```
ang = broadside2az(ang,35)
```

```
ang =

  40.3092   14.7033
```

**Beamscan DOA Estimation with a URA**

Next, we illustrate DOA estimation using a 10-by-5 uniform rectangular array (URA). A URA can estimate both azimuth and elevation angles. The element spacing is 0.3 meters between each row, and 0.5 meters between each column.

```
ura = phased.URA('Size',[10 5],'ElementSpacing',[0.3 0.5]);
```

Assume that two signals impinge on the URA. The first signal arrives from 40° in azimuth and 45° in elevation, while the second signal arrives from -20° in azimuth and 20° in elevation.

```
ang1 = [40; 45];           % First signal
ang2 = [-20; 20];          % Second signal

signal = sensorsig(getElementPosition(ura)/lambda,Nsamp, ...
  [ang1 ang2],nPower);
rng(rs);                   % Restore random number generator
```

Create a 2-D beamscan estimator object from the URA. This object uses the same algorithm as the 1-D case except that it scans over both azimuth and elevation instead of broadside angles.

The scanning region is specified by the property 'AzimuthScanAngles' and 'ElevationScanAngles'. To reduce computational complexity, we assume some a priori knowledge about the incoming signal directions. We restrict the azimuth scan region to [-45 45] and the elevation scan region to [10 60].

```
azelspectrum = phased.BeamscanEstimator2D('SensorArray',ura,...
        'OperatingFrequency',fc,...
        'AzimuthScanAngles',-45:45,'ElevationScanAngles',10:60,...
        'DOAOutputPort',true,'NumSignals',2);
```

The DOA output is a 2-by-N matrix where N is the number of signal directions. The first row contains azimuth angles while the second row contains elevation angles.

```
[~,ang] = azelspectrum(signal)
```

```
ang =

    40   -20
    45    20
```

Plot a 3-D spectrum in azimuth and elevation.

```
plotSpectrum(azelspectrum);
```
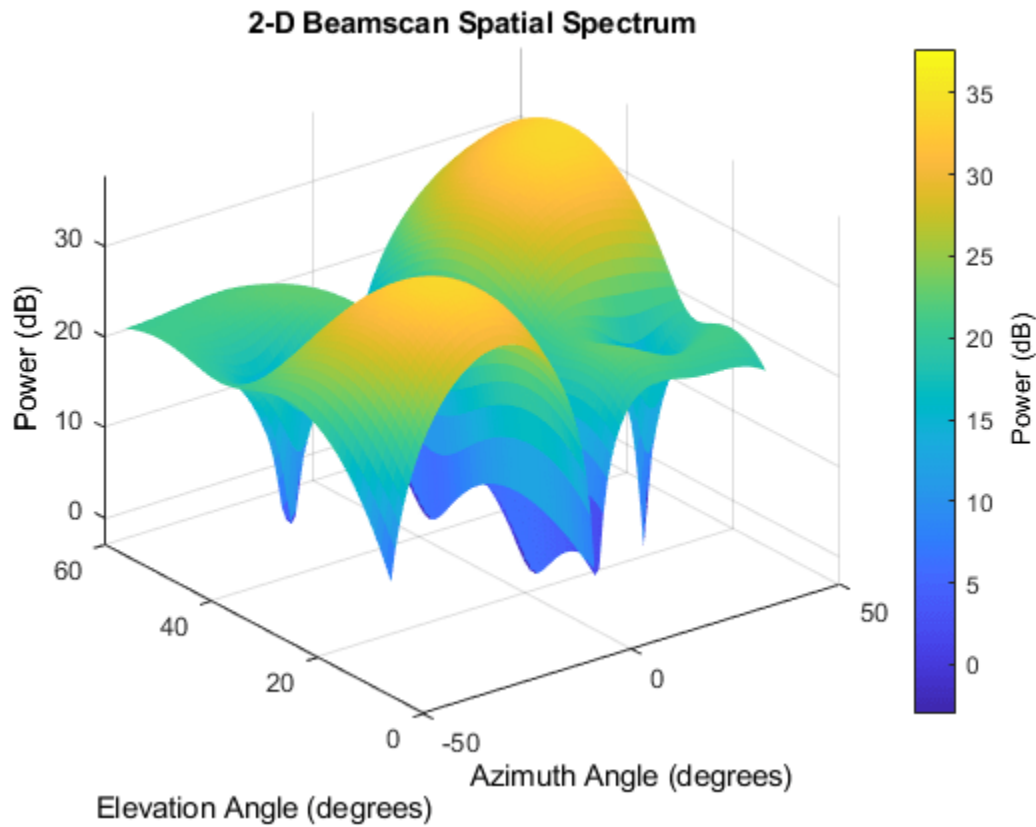


**MVDR DOA Estimation with a URA**

Similar to the ULA case, we use a 2-D version of the MVDR algorithm. Since our knowledge of the sensor positions is perfect, we expect the MVDR spectrum to have a better resolution than beamscan.

```
mvdrazelspectrum = phased.MVDREstimator2D('SensorArray',ura,...
        'OperatingFrequency',fc,...
        'AzimuthScanAngles',-45:45,'ElevationScanAngles',10:60,...
        'DOAOutputPort',true,'NumSignals',2);
[~,ang] = mvdrazelspectrum(signal)
plotSpectrum(mvdrazelspectrum);


ang =

    -20    40
     20    45
```

**2-D MVDR Spatial Spectrum**
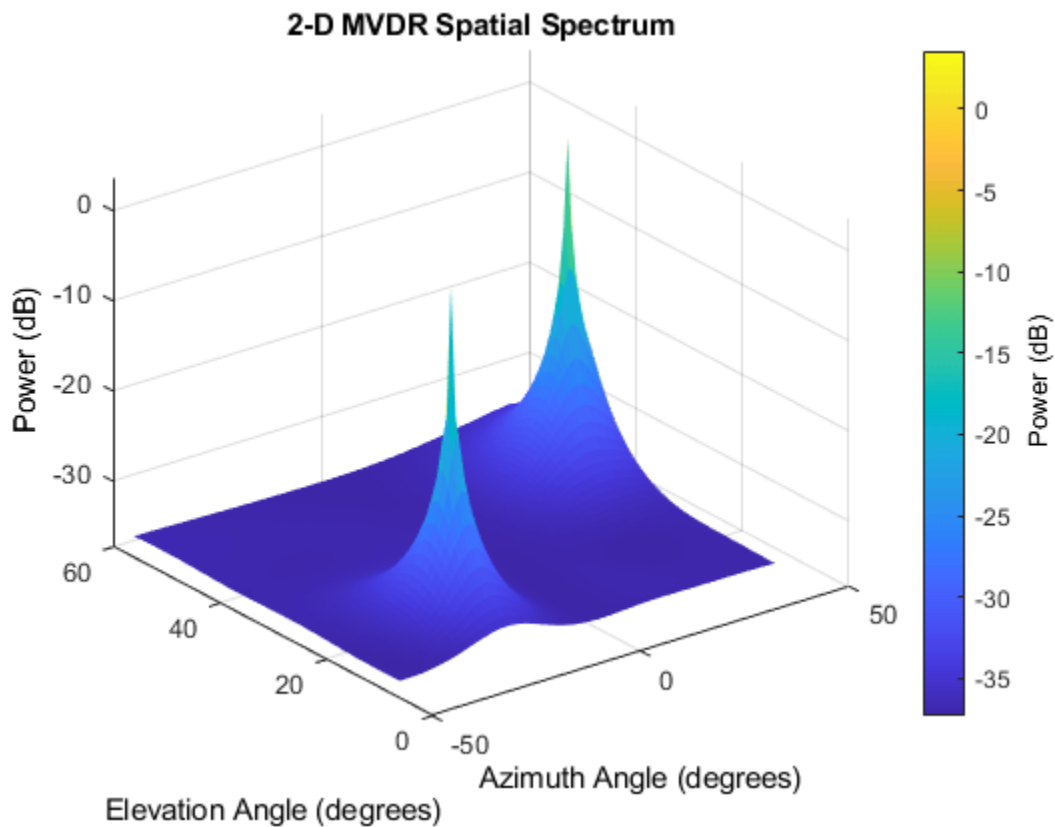
**MUSIC DOA Estimation with a URA**

We can also use the MUSIC algorithm to estimate the directions of arrival of the two sources.

```
musicazelspectrum = phased.MUSICEstimator2D('SensorArray',ura,...
        'OperatingFrequency',fc,...
        'AzimuthScanAngles',-45:45,'ElevationScanAngles',10:60,...
        'DOAOutputPort',true,'NumSignalsSource','Property','NumSignals',2);
[~,ang] = musicazelspectrum(signal)
plotSpectrum(musicazelspectrum);
```

```
ang =

    -20    40
     20    45
```

**2-D MUSIC Spatial Spectrum**
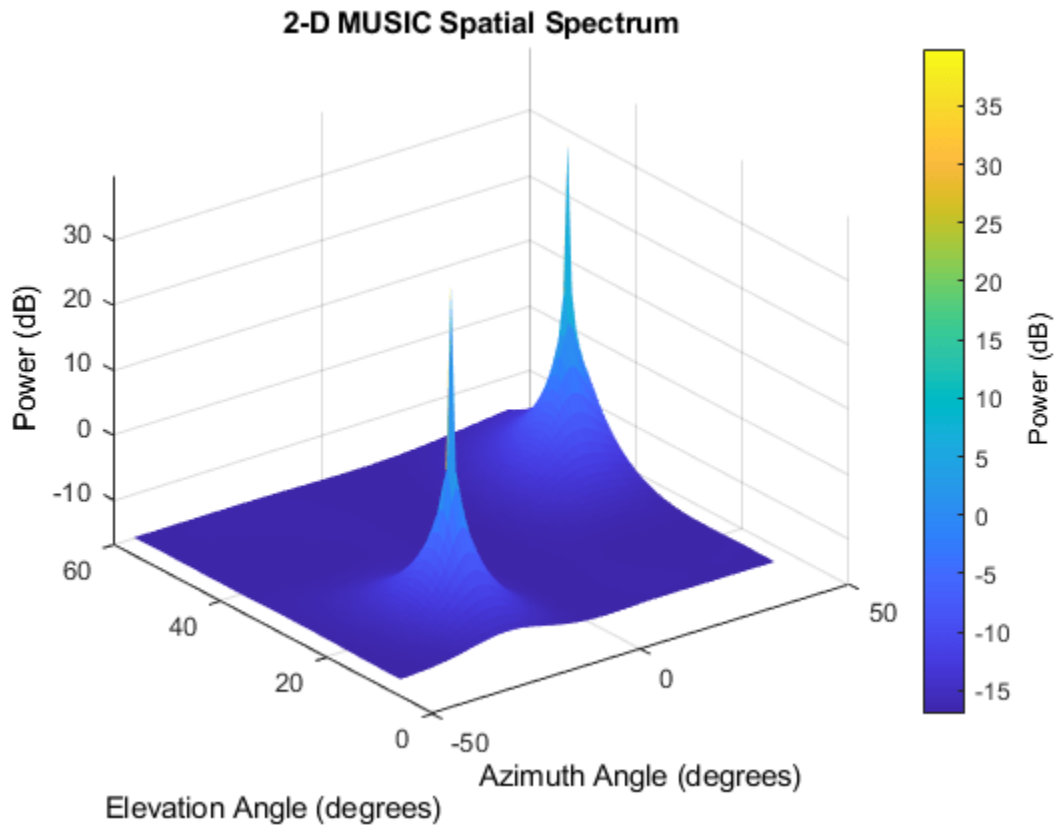


To compare MVDR and MUSIC estimators, let's consider sources located even closer together. Using MVDR and MUSIC, compute the spatial spectrum of two sources located at 10° in azimuth and separated by 3° in elevation.

```
ang1 = [10; 20];            % First signal
ang2 = [10; 23];            % Second signal

signal = sensorsig(getElementPosition(ura)/lambda,Nsamp, ...
  [ang1 ang2],nPower);
[~,angmvdr] = mvdrazelspectrum(signal)
[~,angmusic] = musicazelspectrum(signal)


angmvdr =

    10    -27
    22     21


angmusic =

    10     10
    23     20
```

In this case, only MUSIC correctly estimates to directions of arrival for the two sources. To see why, plot an elevation cut of each spatial spectrum at 10° azimuth.

```
ymvdr = mvdrazelspectrum(signal);
ymusic = musicazelspectrum(signal);
helperPlotDOASpectra(mvdrazelspectrum.ElevationScanAngles,...
    musicazelspectrum.ElevationScanAngles,ymvdr(:,56),ymusic(:,56),'URA')
```
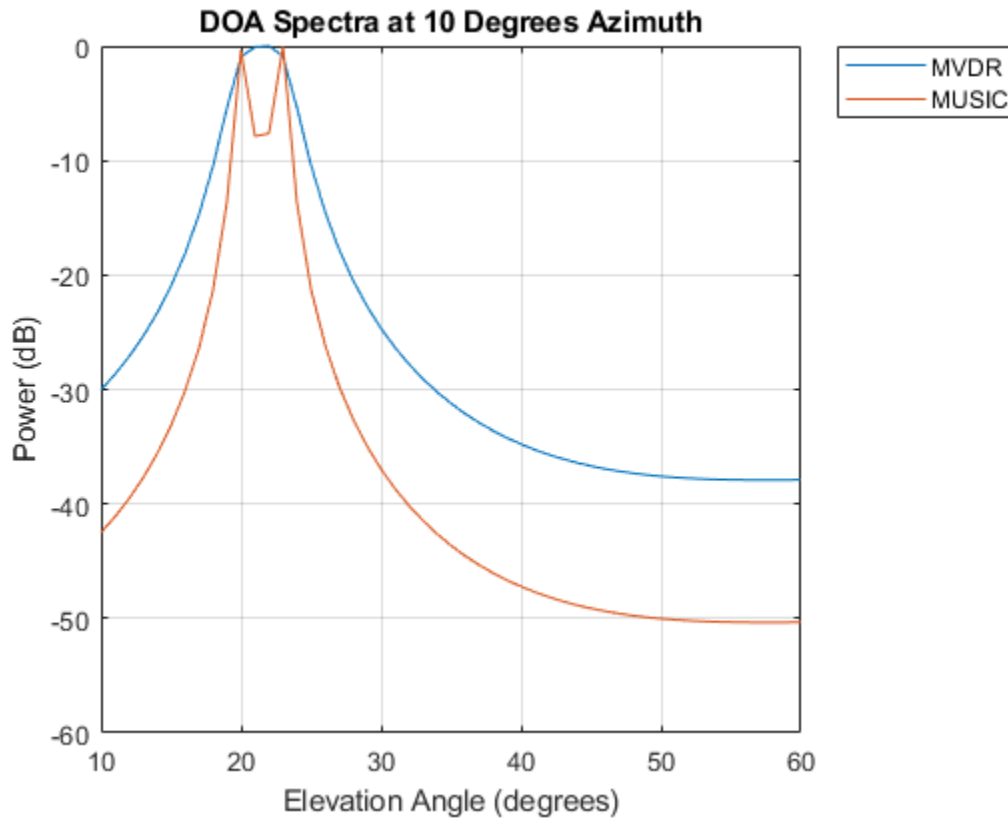


Since the MUSIC spectrum has better spatial resolution than MVDR, MUSIC correctly identifies the sources while MVDR fails to do so.

**Summary**

In this example, we showed how to apply the beamscan, MVDR, and MUSIC techniques to the DOA estimation problem. We used both techniques to estimate broadside angles for the signals received by a ULA. The MVDR algorithm has better resolution than beamscan when there is no sensor position error. MUSIC has even better resolution than MVDR, but the number of sources must be known. We also illustrated how to convert between azimuth and broadside angles. Next, we applied beamscan, MVDR, and MUSIC to estimate both azimuth and elevation angles using a URA. In all of these cases, we plotted the output spatial spectrum, and found again that MUSIC had the best spatial resolution. Beamscan, MVDR, and MUSIC are techniques that can be applied to any type of array, but for ULAs and URAs there are additional high resolution techniques that can further exploit the array geometry.

# High Resolution Direction of Arrival Estimation

This example illustrates several high-resolution direction of arrival (DOA) estimation techniques. It introduces variants of the MUSIC, root-MUSIC, ESPRIT and root-WSF algorithms and discusses their respective merits in the context of far-field, narrowband signal sources received by a uniform linear array (ULA) antenna.

**Modeling the Received Array Signals**

Define a uniform linear array (ULA) composed of 10 isotropic antennas. The array element spacing is 0.5 meters.

```
N = 10;
ula = phased.ULA('NumElements',N,'ElementSpacing',0.5)

ula =
  phased.ULA with properties:

          Element: [1x1 phased.IsotropicAntennaElement]
       NumElements: 10
    ElementSpacing: 0.5000
          ArrayAxis: 'y'
             Taper: 1
```

Simulate the array output for two incident signals. Both signals are incident from 90° in azimuth. Their elevation angles are 73° and 68° respectively. In this example, we assume that these two directions are unknown and need to be estimated. Simulate the baseband received signal at the array demodulated from an operating frequency of 300 MHz.

```
fc = 300e6;                            % Operating frequency
fs = 8192;                             % Sampling frequency
lambda = physconst('LightSpeed')/fc;   % Wavelength
pos = getElementPosition(ula)/lambda;  % Element position in wavelengths
ang1 = [90;73]; ang2 = [90;68];        % Direction of the signals
angs = [ang1 ang2];
Nsamp = 1024;                          % Number of snapshots
noisePwr = 0.01;                       % Noise power

rs = rng(2012);                        % Set random number generator
signal = sensorsig(pos,Nsamp,angs,noisePwr);
```

Because a ULA is symmetric around its axis, a DOA algorithm cannot uniquely determine azimuth and elevation. Therefore, the results returned by these high-resolution DOA estimators are in the form of broadside angles. An illustration of broadside angles can be found in the following figure.
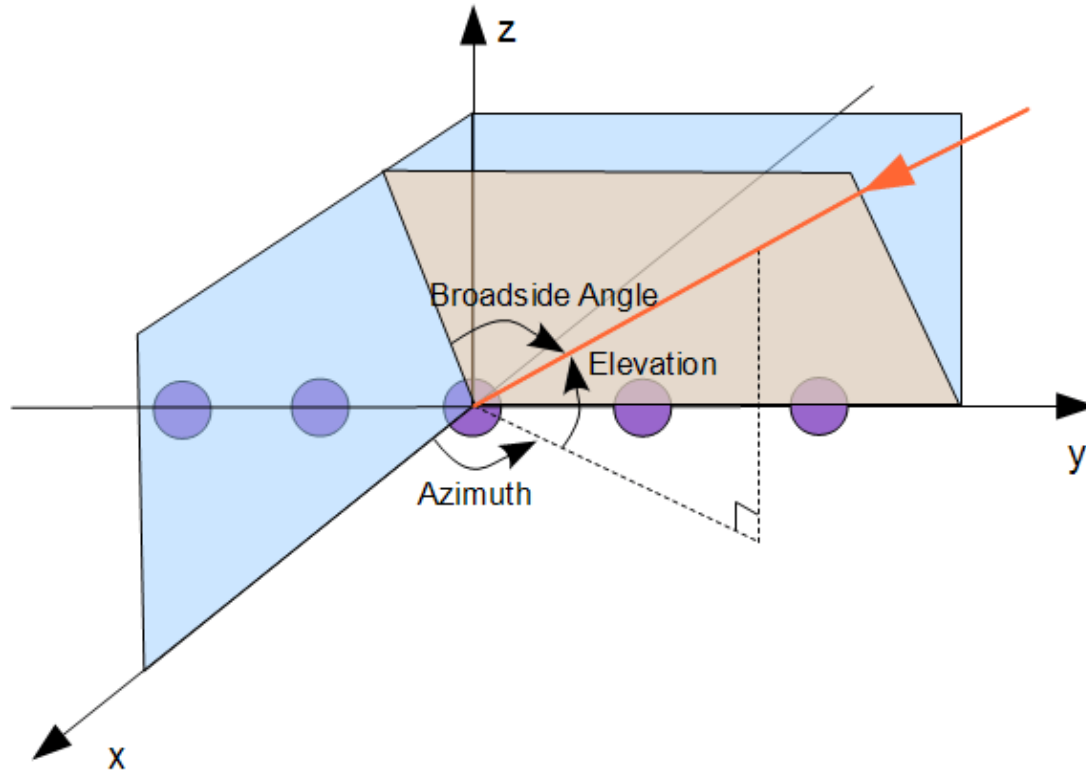
Calculate the broadside angles corresponding to the two incident angles.

```
ang_true = az2broadside(angs(1,:),angs(2,:))
```

```
ang_true = 1×2

   17.0000   22.0000
```

The broadside angles are 17° and 22°.

### Estimating the Direction of Arrival (DOA)

Assume that we know a priori that there are two sources. To estimate the DOA, use the root-MUSIC technique. Construct a DOA estimator using the root-MUSIC algorithm.

```
rootmusicangle = phased.RootMUSICEstimator('SensorArray',ula,...
          'OperatingFrequency',fc,...
          'NumSignalsSource','Property','NumSignals',2)
```

```
rootmusicangle =
  phased.RootMUSICEstimator with properties:
```

```
              SensorArray: [1x1 phased.ULA]
         PropagationSpeed: 299792458
       OperatingFrequency: 300000000
         NumSignalsSource: 'Property'
               NumSignals: 2
  ForwardBackwardAveraging: false
          SpatialSmoothing: 0
```

Because the array response vector of a ULA is conjugate symmetric, we can use forward-backward (FB) averaging to perform computations with real matrices and reduce computational complexity. FB-based estimators also have a lower variance and reduce the correlation between signals.

To apply forward-backward averaging, set the ForwardBackwardAveraging property of the root-MUSIC DOA estimator to true. In this case, the root-MUSIC algorithm is also called the unitary root-MUSIC algorithm.

```
rootmusicangle.ForwardBackwardAveraging = true;
```

Perform the DOA estimation:

```
ang = rootmusicangle(signal)
```

ang = *1×2*

```
   16.9960   21.9964
```

We can also use an ESPRIT DOA estimator. As in the case of root-MUSIC, set the ForwardBackwardAveraging property to true. This algorithm is also called unitary ESPRIT.

```
espritangle = phased.ESPRITEstimator('SensorArray',ula,...
            'OperatingFrequency',fc,'ForwardBackwardAveraging',true,...
            'NumSignalsSource','Property','NumSignals',2)

espritangle =
  phased.ESPRITEstimator with properties:

               SensorArray: [1x1 phased.ULA]
          PropagationSpeed: 299792458
        OperatingFrequency: 300000000
          NumSignalsSource: 'Property'
               NumSignals: 2
          SpatialSmoothing: 0
                   Method: 'TLS'
  ForwardBackwardAveraging: true
            RowWeighting: 1
```

```
ang = espritangle(signal)
```

ang = *1×2*

```
   21.9988   16.9748
```

Finally, use the MUSIC DOA estimator. MUSIC also supports forward-backward averaging. Unlike ESPRIT and root-MUSIC, MUSIC computes a spatial spectrum for specified broadside scan angles. Directions of arrival correspond to peaks in the MUSIC spatial spectrum.

```
musicangle = phased.MUSICEstimator('SensorArray',ula,...
             'OperatingFrequency',fc,'ForwardBackwardAveraging',true,...
             'NumSignalsSource','Property','NumSignals',2,...
             'DOAOutputPort',true)
```

```
musicangle =
  phased.MUSICEstimator with properties:

                SensorArray: [1x1 phased.ULA]
           PropagationSpeed: 299792458
         OperatingFrequency: 300000000
    ForwardBackwardAveraging: true
           SpatialSmoothing: 0
                 ScanAngles: [-90 -89 -88 -87 -86 -85 -84 -83 -82 -81 -80 -79 -78 -77 -76 -75 -
              DOAOutputPort: true
                 NumSignals: 2
           NumSignalsSource: 'Property'
```

```
[~,ang] = musicangle(signal)
```

```
ang = 1×2

    17    22
```

```
plotSpectrum(musicangle)
```

**MUSIC Spatial Spectrum**
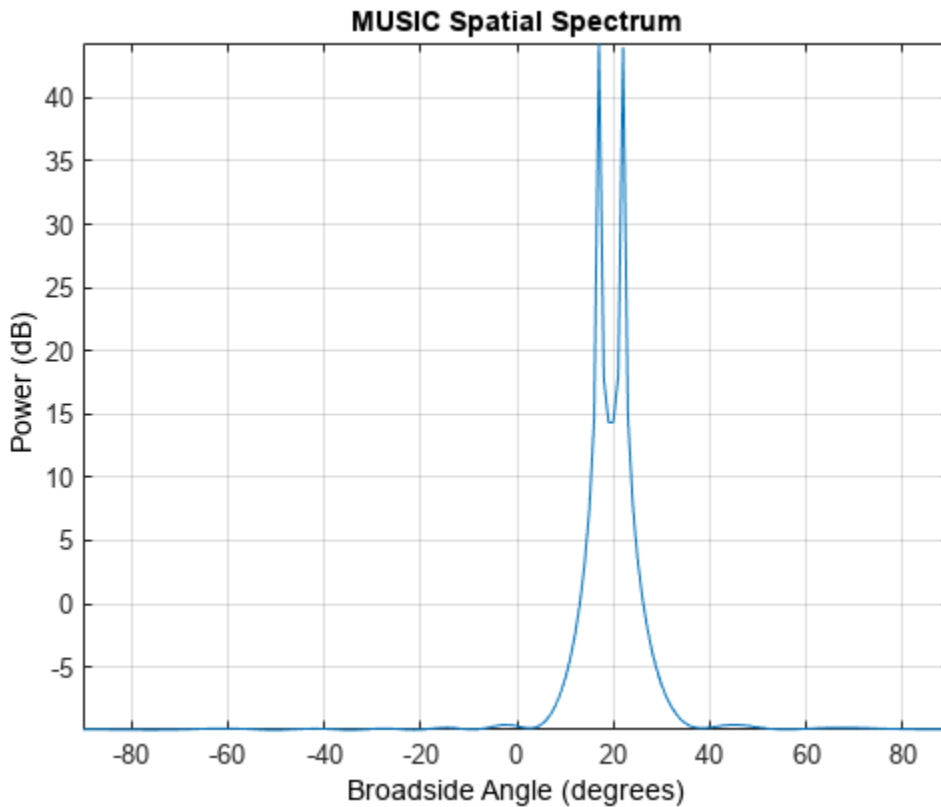


Directions of arrival for MUSIC are limited to the scan angles in the `ScanAngles` property. Because the true directions of arrival in this example coincide with the search angles in `ScanAngles`, MUSIC provides precise DOA angle estimates. In practice, root-MUSIC provides superior resolution to MUSIC. However, the MUSIC algorithm can also be used for DOA estimation in both azimuth and elevation using a 2-D array. See "Direction of Arrival Estimation with Beamscan, MVDR, and MUSIC" on page 18-216.

### Estimating the Number of Signal Sources

In practice, you generally do not know the number of signal sources, and need to estimate the number of sources from the received signal. You can estimate the number of signal sources by specifying 'Auto' for the NumSignalsSource property and choosing either 'AIC' or 'MDL' for the NumSignalsMethod property. For AIC, the Akaike information criterion (AIC) is used, and for MDL, the minimum description length (MDL) criterion is used.

Before you can set NumSignalsSource, you must release the DOA object because it is locked to improve efficiency during processing.

```
release(espritangle);
espritangle.NumSignalsSource = 'Auto';
espritangle.NumSignalsMethod = 'AIC';
ang = espritangle(signal)
```

ang = *1×2*

    21.9988   16.9748

**Reducing Computational Complexity**

In addition to forward-backward averaging, other methods can reduce computational complexity. One of these approaches is to solve an equivalent problem with reduced dimensions in beamspace. While the ESPRIT algorithm performs the eigenvalue decomposition (EVD) of a 10x10 real matrix in our example, the beamspace version can reduce the problem to the EVD of a 3x3 real matrix. This technique uses a priori knowledge of the sector where the signals are located to position the center of the beam fan. In this example, point the beam fan to 20° in azimuth.

```
bsespritangle = phased.BeamspaceESPRITEstimator('SensorArray',ula,...
                'OperatingFrequency',fc,...
                'NumBeamsSource','Property','NumBeams',3,...
                'BeamFanCenter',20);
ang = bsespritangle(signal)

ang = 1×2

   21.9875   16.9943
```

Another technique is the root-weighted subspace fitting (WSF) algorithm. This algorithm is iterative and the most demanding in terms of computational complexity. You can set the maximum number of iterations by specifying the MaximumIterationCount property to maintain the cost below a specific limit.

```
rootwsfangle = phased.RootWSFEstimator('SensorArray',ula,...
            'OperatingFrequency',fc,'MaximumIterationCount',2);
ang = rootwsfangle(signal)

ang = 1×2

   16.9961   21.9962
```

**Optimizing Performance**

In addition to FB averaging, you can use row weighting to improve the statistical performance of the element-space ESPRIT estimator. Row weighting is a technique that applies different weights to the rows of the signal subspace matrix. The row weighting parameter determines the maximum weight. In most cases, it is chosen to be as large as possible. However, its value can never be greater than (N-1)/2, where N is the number of elements of the array.

```
release(espritangle);
espritangle.RowWeighting = 4

espritangle =
  phased.ESPRITEstimator with properties:

                SensorArray: [1x1 phased.ULA]
           PropagationSpeed: 299792458
         OperatingFrequency: 300000000
           NumSignalsSource: 'Auto'
           NumSignalsMethod: 'AIC'
           SpatialSmoothing: 0
                     Method: 'TLS'
    ForwardBackwardAveraging: true
               RowWeighting: 4
```

```
ang = espritangle(signal)

ang = 1×2

   21.9884   17.0003
```

**Estimating Coherent Sources in Multipath Environments**

If several sources are correlated or coherent (as in multipath environments), the spatial covariance matrix becomes rank deficient and subspace-based DOA estimation methods may fail. To show this, model a received signal composed of 4 narrowband components. Assume that 2 of the first 3 signals are multipath reflections of the first source, having magnitudes equal to 1/4 and 1/2 that of the first source, respectively.

```
scov = eye(4);
magratio = [1;0.25;0.5];
scov(1:3,1:3) = magratio*magratio';
```

All signals are incident at 0° elevation, with azimuth incident angles of -23°, 0°, 12°, and 40°.

```
% Incident azimuth
az_ang = [-23 0 12 40];

% When the elevation is zero, the azimuth within [-90 90] is the same as
% the broadside angle.
el_ang = zeros(1,4);

% The received signals
signal = sensorsig(pos,Nsamp,[az_ang; el_ang],noisePwr,scov);
rng(rs);  % Restore random number generator
```

Compare the performance of the DOA algorithm when sources are coherent. To simplify the example, run only one trial per algorithm. Given the high SNR, the results will be a good indicator of estimation accuracy.

First, verify that the AIC criterion underestimates the number of sources, causing the unitary ESPRIT algorithm to give incorrect estimates. The AIC estimates the number of sources as two because three sources are correlated.

```
release(espritangle);
espritangle.NumSignalsSource = 'Auto';
espritangle.NumSignalsMethod = 'AIC';
ang = espritangle(signal)

ang = 1×2

  -15.3535   40.0024
```

The root-WSF algorithm is robust in the context of correlated signals. With the correct number of sources as an input, the algorithm correctly estimates the directions of arrival.

```
release(rootwsfangle);
rootwsfangle.NumSignalsSource = 'Property';
rootwsfangle.NumSignals = 4;
ang = rootwsfangle(signal)
```

```
ang = 1×4

   40.0016   -22.9919    12.0693     0.0737
```

ESPRIT, root-MUSIC, and MUSIC, however, fail to estimate the correct directions of arrival, even if we specify the number of sources and use the unitary implementations.

```
release(rootmusicangle);
rootmusicangle.NumSignalsSource = 'Property';
rootmusicangle.NumSignals = 4;
rootmusicangle.ForwardBackwardAveraging = true;
ang = rootmusicangle(signal)

ang = 1×4

   40.0077   -22.8313     4.4976   -11.9038
```

You can apply spatial smoothing to estimate the DOAs of correlated signals. Using spatial smoothing, however, decreases the effective aperture of the array. Therefore, the variance of the estimators increases because the subarrays are smaller than the original array.

```
release(rootmusicangle);
Nr = 2;     % Number of multipath reflections
rootmusicangle.SpatialSmoothing = Nr

rootmusicangle =
  phased.RootMUSICEstimator with properties:

                 SensorArray: [1x1 phased.ULA]
            PropagationSpeed: 299792458
           OperatingFrequency: 300000000
             NumSignalsSource: 'Property'
                   NumSignals: 4
     ForwardBackwardAveraging: true
             SpatialSmoothing: 2


ang = rootmusicangle(signal)

ang = 1×4

   40.0010   -22.9959    12.1376     0.1843
```

**Comparison of DOA Algorithms**

In summary, ESPRIT, MUSIC, root-MUSIC, and root-WSF are important DOA algorithms that provide good performance and reasonable computational complexity for ULAs. Unitary ESPRIT, unitary root-MUSIC, and beamspace unitary ESPRIT provide ways to significantly reduce the computational cost of the estimators, while also improving their performance. root-WSF is particularly attractive in the context of correlated sources because, contrary to the other methods, it does not require spatial smoothing to properly estimate the DOAs when the number of sources is known.

# Increasing Angular Resolution with Virtual Arrays

This example introduces how forming a virtual array in MIMO radars can help increase angular resolution. It shows how to simulate a coherent MIMO radar signal processing chain using Phased Array System Toolbox™.

**Introduction**

There are two categories of multiple input multiple output (MIMO) radars. Multistatic radars form the first category. They are often referred to as statistical MIMO radars. Coherent MIMO radars form the second category and are the focus of this example. A benefit of coherent MIMO radar signal processing is the ability to increase the angular resolution of the physical antenna array by forming a virtual array.

**Virtual Array**

A virtual array can be created by quasi-monostatic MIMO radars, where the transmit and receive arrays are closely located. To better understand the virtual array concept, first look at the two-way pattern of a conventional phased array radar. The two-way pattern of a phased array radar is the product of its transmit array pattern and receive array pattern. For example, consider a 77 GHz millimeter wave radar with a 2-element transmit array and a 4-element receive array.

```
fc = 77e9;
c = 3e8;
lambda = c/fc;
Nt = 2;
Nr = 4;
```

If both arrays have half-wavelength spacing, which are sometimes referred to as full arrays, then the two-way pattern is close to the receive array pattern.

```
dt = lambda/2;
dr = lambda/2;

txarray = phased.ULA(Nt,dt);
rxarray = phased.ULA(Nr,dr);

ang = -90:90;

pattx = pattern(txarray,fc,ang,0,'Type','powerdb');
patrx = pattern(rxarray,fc,ang,0,'Type','powerdb');
pat2way = pattx+patrx;

helperPlotMultipledBPattern(ang,[pat2way pattx patrx],[-30 0],...
    {'Two-way Pattern','Tx Pattern','Rx Pattern'},...
    'Patterns of full/full arrays - 2Tx, 4Rx',...
    {'-','--','-.'});
```
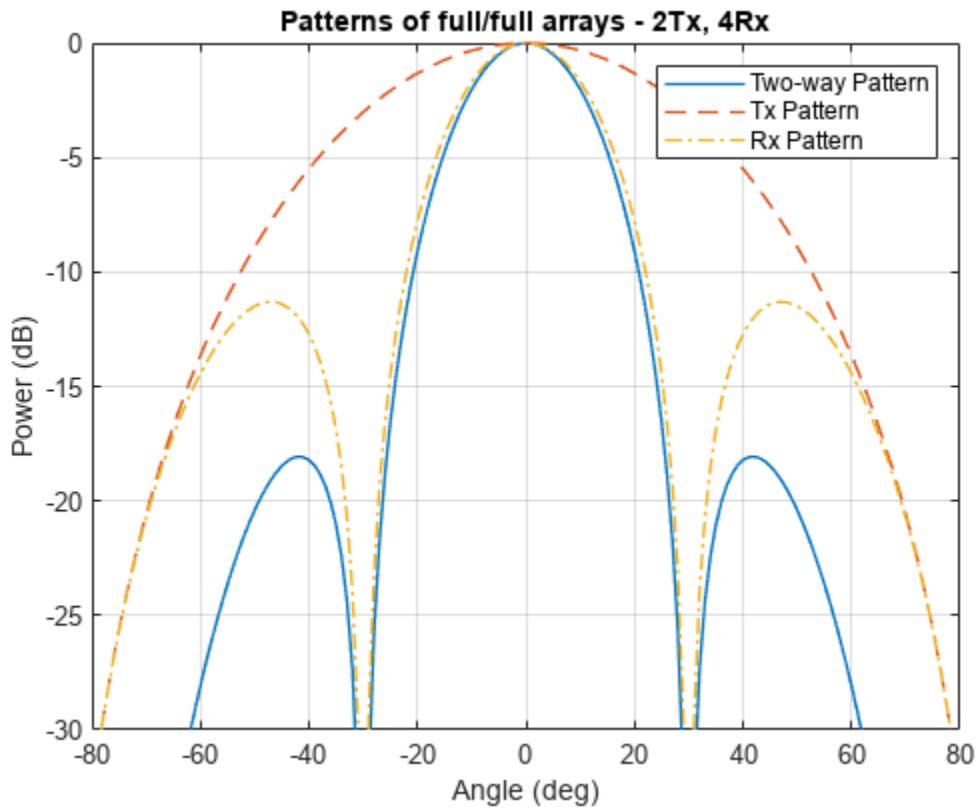
If the full transmit array is replaced with a thin array, meaning the element spacing is wider than half wavelength, then the two-way pattern has a narrower beamwidth. Notice that even though the thin transmit array has grating lobes, those grating lobes are not present in the two-way pattern.

```
dt = Nr*lambda/2;
txarray = phased.ULA(Nt,dt);
pattx = pattern(txarray,fc,ang,0,'Type','powerdb');
pat2way = pattx+patrx;
helperPlotMultipledBPattern(ang,[pat2way pattx patrx],[-30 0],...
    {'Two-way Pattern','Tx Pattern','Rx Pattern'},...
    'Patterns of thin/full arrays - 2Tx, 4Rx',...
    {'-','--','-.'});
```

Patterns of thin/full arrays - 2Tx, 4Rx
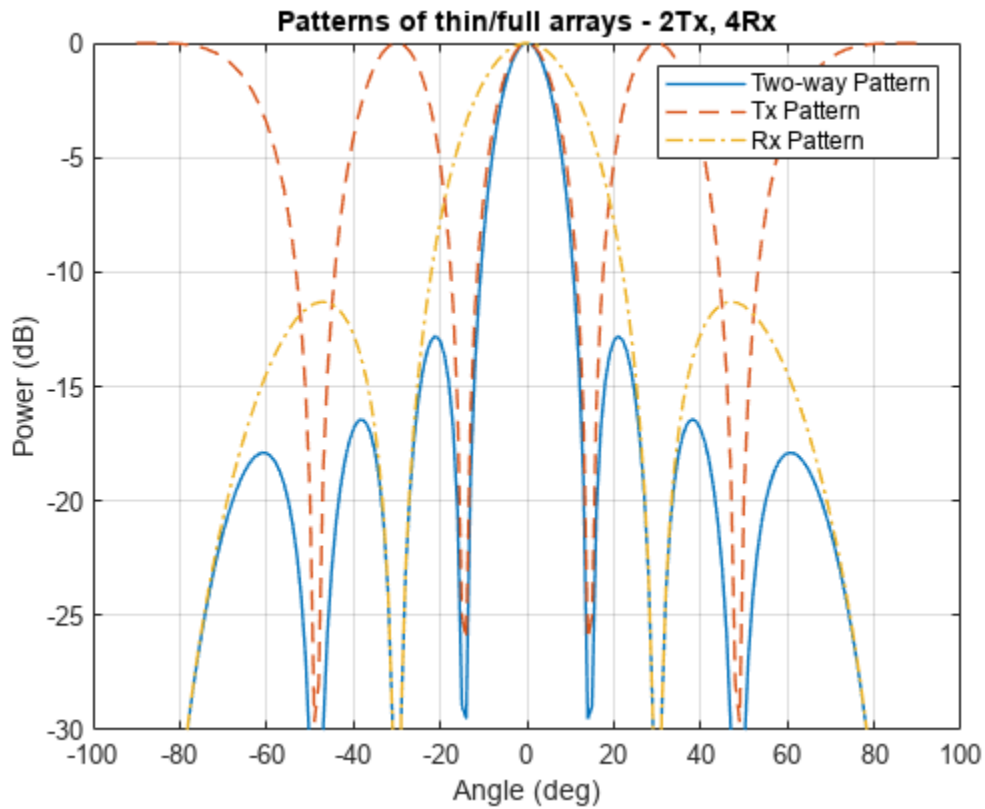
The two-way pattern of this system corresponds to the pattern of a virtual receive array with 2 x 4 = 8 elements. Thus, by carefully choosing the geometry of the transmit and the receive arrays, we can increase the angular resolution of the system without adding more antennas to the arrays.



TX Array 2λ spacing          RX Array 0.5λ spacing          Virtual Array 0.5λ spacing

```
varray = phased.ULA(Nt*Nr,dr);
patv = pattern(varray,fc,ang,0,'Type','powerdb');
helperPlotMultipledBPattern(ang,[pat2way patv],[-30 0],...
    {'Two-way Pattern','Virtual Array Pattern'},...
    'Patterns of thin/full arrays and virtual array',...
    {'-','--'},[1 2]);
```

Patterns of thin/full arrays and virtual array
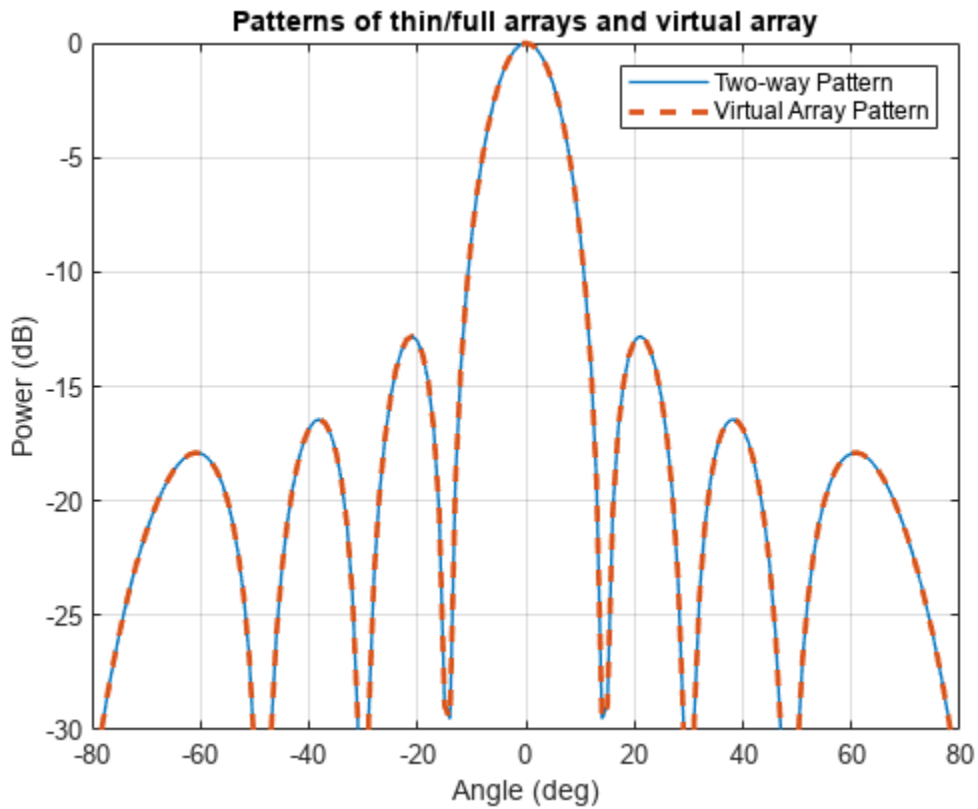
## Virtual Array in MIMO Radars

In a coherent MIMO radar system, each antenna of the transmit array transmits an orthogonal waveform. Because of this orthogonality, it is possible to recover the transmitted signals at the receive array. The measurements at the physical receive array corresponding to each orthogonal waveform can then be stacked to form the measurements of the virtual array.
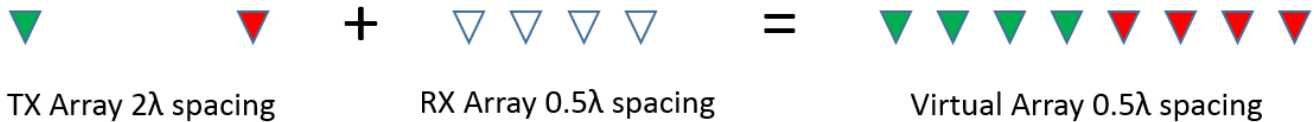


Note that since each element in the transmit array radiates independently, there is no transmit beamforming, so the transmit pattern is broad and covers a large field of view (FOV). This allows the simultaneous illumination of all targets in the FOV. The receive array can then generate multiple beams to process all target echoes. Compared to conventional phased array radars that need successive scans to cover the entire FOV, this is another advantage of MIMO radars for applications that require fast reaction time.

## TDM-MIMO Radar Simulation

Time division multiplexing (TDM) is one way to achieve orthogonality among transmit channels. The remainder of this example shows how to model and simulate a TDM-MIMO frequency-modulated

continuous wave (FMCW) automotive radar system. The waveform characteristics are adapted from the "Automotive Adaptive Cruise Control Using FMCW Technology" (Radar Toolbox) example.

```
waveform = helperDesignFMCWWaveform(c,lambda);
fs = waveform.SampleRate;
```

Imagine that there are two cars in the FOV with a separation of 20 degrees. As seen in the earlier array pattern plots of this example, the 3dB beamwidth of a 4-element receive array is around 30 degrees so conventional processing would not be able to separate the two targets in the angular domain. The radar sensor parameters are as follows:

```
transmitter = phased.Transmitter('PeakPower',0.001,'Gain',36);
receiver = phased.ReceiverPreamp('Gain',40,'NoiseFigure',4.5,'SampleRate',fs);

txradiator = phased.Radiator('Sensor',txarray,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'WeightsInputPort',true);

rxcollector = phased.Collector('Sensor',rxarray,'OperatingFrequency',fc,...
    'PropagationSpeed',c);
```

Define the position and motion of the ego vehicle and the two cars in the FOV.

```
radar_speed = 100*1000/3600;     % Ego vehicle speed 100 km/h
radarmotion = phased.Platform('InitialPosition',[0;0;0.5],'Velocity',[radar_speed;0;0]);

car_dist = [40 50];               % Distance between sensor and cars (meters)
car_speed = [-80 96]*1000/3600;  % km/h -> m/s
car_az = [-10 10];
car_rcs = [20 40];
car_pos = [car_dist.*cosd(car_az);car_dist.*sind(car_az);0.5 0.5];

cars = phased.RadarTarget('MeanRCS',car_rcs,'PropagationSpeed',c,'OperatingFrequency',fc);
carmotion = phased.Platform('InitialPosition',car_pos,'Velocity',[car_speed;0 0;0 0]);
```

The propagation model is assumed to be free space.

```
channel = phased.FreeSpace('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs,'TwoWayPropagation',true);
```

The raw data cube received by the physical array of the TDM MIMO radar can then be simulated as follows:

```
rng(2017);
Nsweep = 64;
Dn = 2;        % Decimation factor
fs = fs/Dn;
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w0 = [0;1];   % weights to enable/disable radiating elements

for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
```

```
        txsig = transmitter(sig);

        % Toggle transmit element
        w0 = 1-w0;
        txsig = txradiator(txsig,tgt_ang,w0);

        % Propagate the signal and reflect off the target
        txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
        txsig = cars(txsig);

        % Dechirp the received radar return
        rxsig = rxcollector(txsig,tgt_ang);
        rxsig = receiver(rxsig);
        dechirpsig = dechirp(rxsig,sig);

        % Decimate the return to reduce computation requirements
        for n = size(xr,2):-1:1
            xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
        end
    end
end
```

**Virtual Array Processing for TDM-MIMO Radar**

The data cube received by the physical array must be processed to form the virtual array data cube. For the TDM-MIMO radar system used in this example, the measurements corresponding to the two transmit antenna elements can be recovered from two consecutive sweeps by taking every other page of the data cube.

```
xr1 = xr(:,:,1:2:end);
xr2 = xr(:,:,2:2:end);
```

Now the data cube in `xr1` contains the return corresponding to the first transmit antenna element, and the data cube in `xr2` contains the return corresponding to the second transmit antenna element. Hence, the data cube from the virtual array can be formed as:

```
xrv = cat(2,xr1,xr2);
```

Next, perform range-Doppler processing on the virtual data cube. Because the range-Doppler processing is linear, the phase information is preserved. Therefore, the resulting response can be used later to perform further spatial processing on the virtual aperture.

```
nfft_r = 2^nextpow2(size(xrv,1));
nfft_d = 2^nextpow2(size(xrv,3));

rngdop = phased.RangeDopplerResponse('PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc,'SampleRate',fs,...
    'RangeMethod','FFT','PRFSource','Property',...
    'RangeWindow','Hann','PRF',1/(Nt*waveform.SweepTime),...
    'SweepSlope',waveform.SweepBandwidth/waveform.SweepTime,...
    'RangeFFTLengthSource','Property','RangeFFTLength',nfft_r,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',nfft_d,...
    'DopplerWindow','Hann');

[resp,r,sp] = rngdop(xrv);
```

The resulting `resp` is a data cube containing the range-Doppler response for each element in the virtual array. As an illustration, the range-Doppler map for the first element in the virtual array is shown.

```
plotResponse(rngdop,squeeze(xrv(:,1,:)));
```
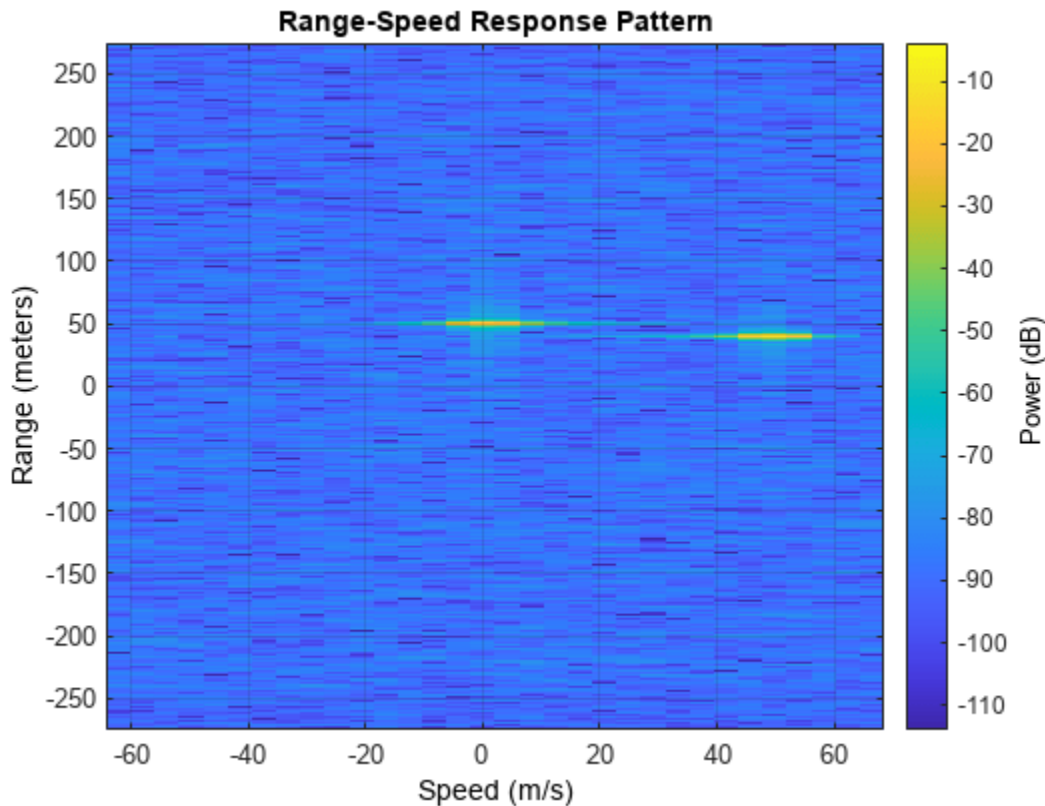


**Range-Speed Response Pattern**

The detection can be performed on the range-Doppler map from each pair of transmit and receive element to identify the targets in scene. In this example, a simple threshold-based detection is performed on the map obtained between the first transmit element and the first receive element, which corresponds to the measurement at the first element in the virtual array. Based on the range-Doppler map shown in the previous figure, the threshold is set at 10 dB below the maximum peak.

```
respmap = squeeze(mag2db(abs(resp(:,1,:))));
ridx = helperRDDetection(respmap,-10);
```

Based on the detected range of the targets, the corresponding range cuts can be extracted from the virtual array data cube to perform further spatial processing. To verify that the virtual array provides a higher resolution compared to the physical array, the code below extracts the range cuts for both targets and combines them into a single data matrix. The beamscan algorithm is then performed over these virtual array measurements to estimate the directions of the targets.

```
xv = squeeze(sum(resp(ridx,:,:),1))';

doa = phased.BeamscanEstimator('SensorArray',varray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'NumSignals',2,'ScanAngles',ang);
[Pdoav,target_az_est] = doa(xv);

fprintf('target_az_est = [%s]\n',num2str(target_az_est));

target_az_est = [-6  10]
```

The two targets are successfully separated. The actual angles for the two cars are -10 and 10 degrees.

The next figure compares the spatial spectrums from the virtual and the physical receive array.

```
doarx = phased.BeamscanEstimator('SensorArray',rxarray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'ScanAngles',ang);
Pdoarx = doarx(xr);

helperPlotMultipledBPattern(ang,mag2db(abs([Pdoav Pdoarx])),[-30 0],...
    {'Virtual Array','Physical Array'},...
    'Spatial spectrum for virtual array and physical array',{'-','--'});
```
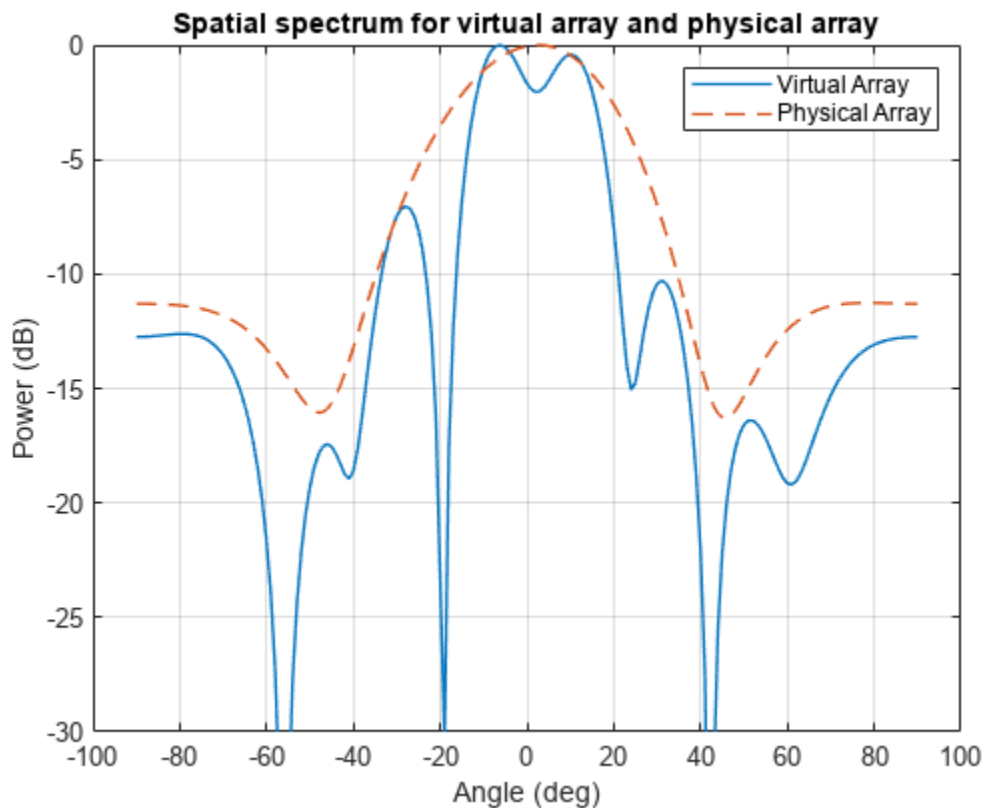


In this example, the detection is performed on the range-Doppler map without spatial processing of the virtual array data cube. This works because the SNR is high. If the SNR is low, it is also possible to process the virtual array blindly across the entire range-Doppler map to maximize the SNR before the detection.

**Phase-coded MIMO Radars**

Although a TDM-MIMO radar's processing chain is relatively simple, it uses only one transmit antenna at a time. Therefore, it does not take advantage of the full capacity of the transmit array. To improve the efficiency, there are other orthogonal waveforms that can be used in a MIMO radar.

For instance, a MIMO radars can also adopt a phase-coded MIMO waveform. In this case, each radiating element can transmit a uniquely coded waveform, and the receiver can then have a matched filter bank corresponding to each of those phase coded waveform. The signals can then be

recovered and processed to form the virtual array. Using the same system configuration as in previous sections, one phase-coded scheme to achieve orthogonality is to have one element always transmit the same FMCW waveform, while the second transmit element reverses the phase of the FMCW waveform for each sweep. This way both transmit elements are active in all sweeps. For the first sweep, the two elements transmit the same waveform; and for the second sweep, the two elements transmit the waveform with opposite phase, and so on. This is essentially coding the consecutive sweeps from different elements with a Hadamard code. It is similar to the Alamouti codes used in MIMO communication systems.

The raw data cube received by the physical array of the Phase-Coded MIMO radar can then be simulated as follows:

```
rng(2018);
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w =  hadamard(2); % weights to enable/disable radiating elements
for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
    txsig = transmitter(sig);

    % Toggle transmit element
    if mod(m,2) == 1
        w0 = w(:,1);
    else
        w0 = w(:,2);
    end

    txsig = txradiator(txsig,tgt_ang,w0);

    % Propagate the signal and reflect off the target
    txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
    txsig = cars(txsig);

    % Dechirp the received radar return
    rxsig = rxcollector(txsig,tgt_ang);
    rxsig = receiver(rxsig);
    dechirpsig = dechirp(rxsig,sig);

    % Decimate the return to reduce computation requirements
    for n = size(xr,2):-1:1
        xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
    end
end
```

**Virtual Array Processing for Phase-coded MIMO Radar**

For the Phase-Coded MIMO radar system used in this example, the measurements corresponding to the two sets of phase code can be recovered from two consecutive sweeps by taking every other page of the data cube.

```
Nvsweep = Nsweep/2;
xr1 = xr(:,:,1:2:end);
xr2 = xr(:,:,2:2:end);
```

Next, perform range-Doppler processing on the two data cubes separately.

```
nfft_r = 2^nextpow2(size(xr1,1));
nfft_d = 2^nextpow2(size(xr1,3));
```

```
rngdop = phased.RangeDopplerResponse('PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc,'SampleRate',fs,...
    'RangeMethod','FFT','PRFSource','Property',...
    'RangeWindow','Hann','PRF',1/(Nt*waveform.SweepTime),...
    'SweepSlope',waveform.SweepBandwidth/waveform.SweepTime,...
    'RangeFFTLengthSource','Property','RangeFFTLength',nfft_r,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',nfft_d,...
    'DopplerWindow','Hann');
```

As an illustration, the range-Doppler map of Phase-Coded MIMO radar is same as that of TDM MIMO radar.

```
plotResponse(rngdop,squeeze(xr1(:,1,:)));
```
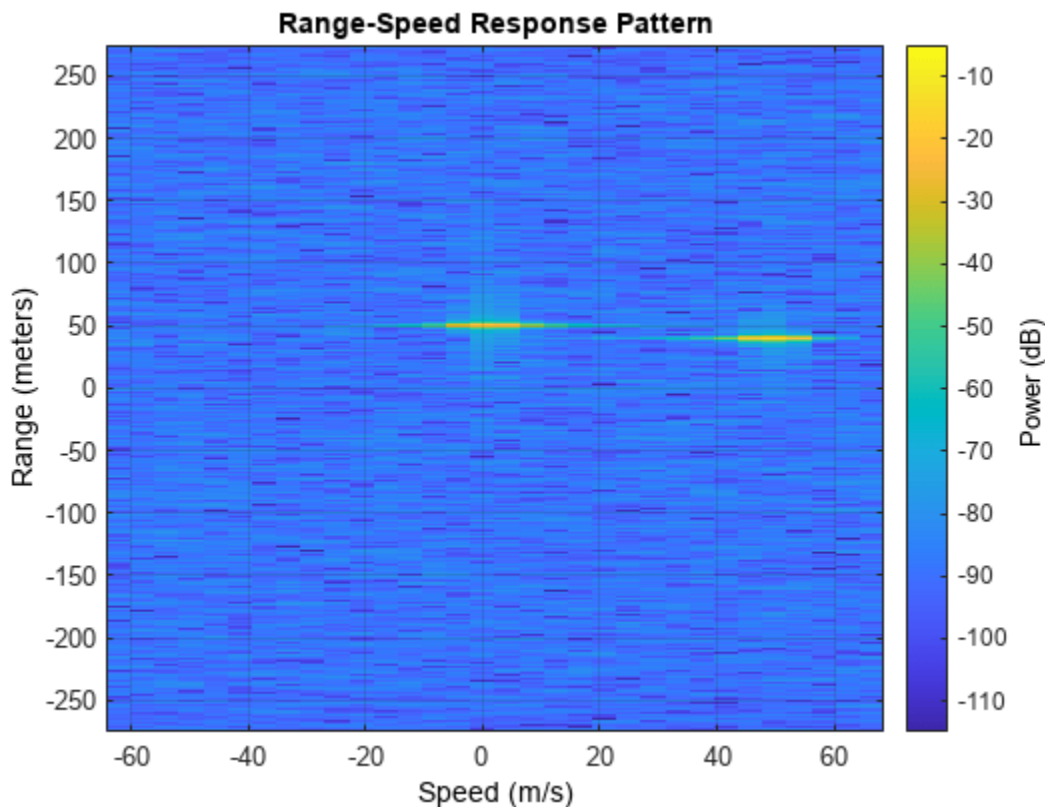


For a phase-coded MIMO radar, the virtual array can no longer be formed by simply cascading the signal from the two adjacent pulses together. Instead, before castcading, the phase code embedded in pulses has to be corrected. This example uses a Hadamard code based approach, thus the process to remove the embedded phases is referred to as Hadamard decode process. After the Hadamard

decode processing, range-Doppler responses of different transmitters are cascaded. The resulting `respvr` is a data cube containing the range-Doppler response for physical arrays from different transmit antenna elements.

```
resp_DDM(:,:,:,1) = rngdop(xr1);
[resp_DDM(:,:,:,2),r,sp] = rngdop(xr2);
respvr(:,:,:,1) = helperHadamardDecode(resp_DDM,w(:,1));
respvr(:,:,:,2) = helperHadamardDecode(resp_DDM,w(:,2));
% for i = 1:2
%     H = w(:,i);
%     respvr(:,:,:,i) = hadamard_decode(resp_DDM,H);
% end
```

Hence, the data cube from the virtual array can be formed as:

```
resp_vr = cat(2,respvr(:,:,:,1),respvr(:,:,:,2));
```

The detection can be performed on the range-Doppler map from each pair of transmit and receive element to identify the targets in scene. In this example, a simple threshold-based detection is performed on the map obtained between the first transmit element and the first receive element, which corresponds to the measurement at the first element in the virtual array. Based on the range-Doppler map shown in the previous figure, the threshold is set at 10 dB below the maximum peak.

```
respmap = squeeze(mag2db(abs(resp_vr(:,1,:))));
ridx = helperRDDetection(respmap,-10);
```

Based on the detected range of the targets, the corresponding range cuts can be extracted from the virtual array data cube to perform further spatial processing. To verify that the virtual array provides a higher resolution compared to the physical array, the code below extracts the range cuts for both targets and combines them into a single data matrix. The beamscan algorithm is then performed over these virtual array measurements to estimate the directions of the targets.

```
xv = squeeze(sum(resp_vr(ridx,:,:),1))';

doa = phased.BeamscanEstimator('SensorArray',varray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'NumSignals',2,'ScanAngles',ang);
[Pdoav,target_az_est] = doa(xv);

fprintf('target_az_est = [%s]\n',num2str(target_az_est));

target_az_est = [10 -11]
```

The two targets are successfully separated. The actual angles for the two cars are -10 and 10 degrees.

The next figure compares the spatial spectrums from the virtual and the physical receive array.

```
doarx = phased.BeamscanEstimator('SensorArray',rxarray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'ScanAngles',ang);
Pdoarx = doarx(xr);

helperPlotMultipledBPattern(ang,mag2db(abs([Pdoav Pdoarx])),[-30 0],...
    {'Virtual Array','Physical Array'},...
    'Spatial spectrum for virtual array and physical array',{'-','--'});
```

Spatial spectrum for virtual array and physical array
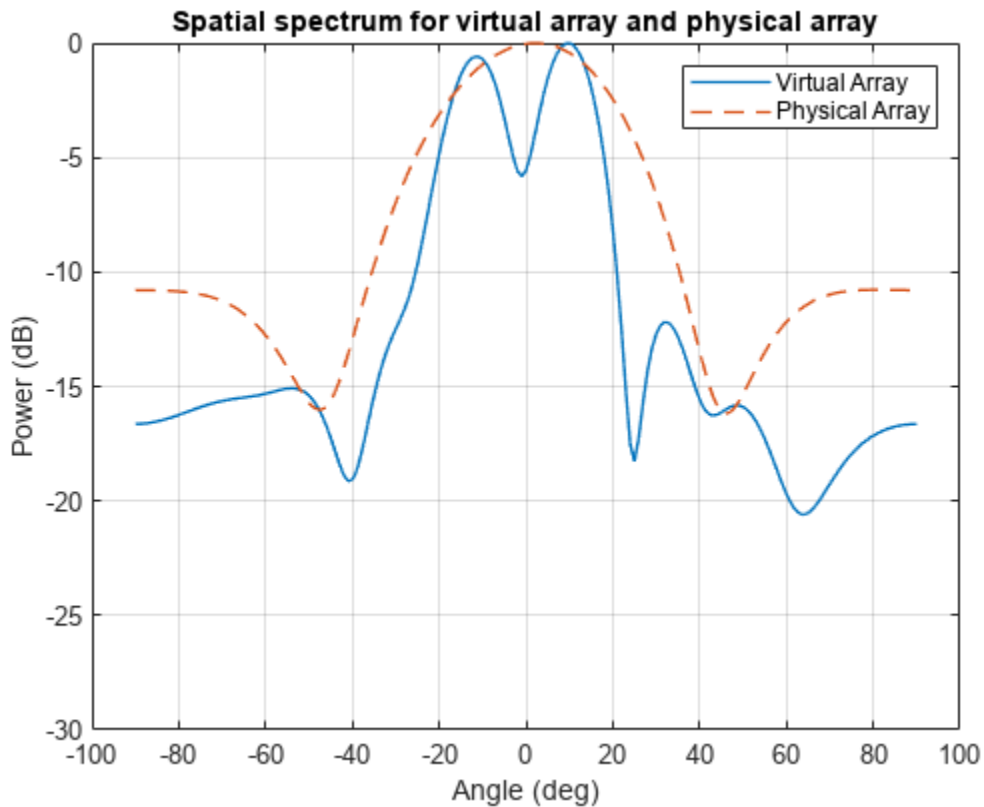
**Summary**

In this example, we gave a brief introduction to coherent MIMO radar and the virtual array concept. We simulated the return of a MIMO radar with a 2-element transmit array and a 4-element receive array and performed direction of arrival estimation of the simulated echos of two closely spaced targets using an 8-element virtual array.

**References**

[1] Frank Robey, et al. *MIMO Radar Theory and Experimental Results*, Conference Record of the Thirty Eighth Asilomar Conference on Signals, Systems and Computers, California, pp. 300-304, 2004.

[2] Eli Brookner, *MIMO Radars and Their Conventional Equivalents*, IEEE Radar Conference, 2015.

[3] Sandeep Rao, *MIMO Radar*, Texas Instruments Application Report SWRA554, May 2017.

[4] Jian Li and Peter Stoica, *MIMO Radar Signal Processing*, John Wiley & Sons, 2009.

# Introduction to Space-Time Adaptive Processing

This example gives a brief introduction to space-time adaptive processing (STAP) techniques and illustrates how to use Phased Array System Toolbox™ to apply STAP algorithms to the received pulses. STAP is a technique used in airborne radar systems to suppress clutter and jammer interference.

**Introduction**

In a ground moving target indicator (GMTI) system, an airborne radar collects the returned echo from the moving target on the ground. However, the received signal contains not only the reflected echo from the target, but also the returns from the illuminated ground surface. The return from the ground is generally referred to as *clutter*.

The clutter return comes from all the areas illuminated by the radar beam, so it occupies all range bins and all directions. The total clutter return is often much stronger than the returned signal echo, which poses a great challenge to target detection. Clutter filtering, therefore, is a critical part of a GMTI system.

In traditional MTI systems, clutter filtering often takes advantage of the fact that the ground does not move. Thus, the clutter occupies the zero Doppler bin in the Doppler spectrum. This principle leads to many Doppler-based clutter filtering techniques, such as pulse canceller. Interested readers can refer to "Ground Clutter Mitigation with Moving Target Indication (MTI) Radar" (Radar Toolbox) for a detailed example of the pulse canceller. When the radar platform itself is also moving, such as in a plane, the Doppler component from the ground return is no longer zero. In addition, the Doppler components of clutter returns are angle dependent. In this case, the clutter return is likely to have energy across the Doppler spectrum. Therefore, the clutter cannot be filtered only with respect to Doppler frequency.

Jamming is another significant interference source that is often present in the received signal. The simplest form of jamming is a barrage jammer, which is strong, continuous white noise directed toward the radar receiver so that the receiver cannot easily detect the target return. The jammer is usually at a specific location, and the jamming signal is therefore associated with a specific direction. However, because of the white noise nature of the jammer, the received jamming signal occupies the entire Doppler band.

STAP techniques filter the signal in both the angular and Doppler domains (thus, the name space-time adaptive processing) to suppress the clutter and jammer returns. In the following sections, you will simulate returns from target, clutter, and jammer and illustrate how STAP techniques filter the interference from the received signal.

**System Setup**

First define a radar system, starting from the system built in the example "Simulating Test Signals for a Radar Receiver" on page 18-393.

```
load BasicMonostaticRadarExampleData.mat;     % Load monostatic pulse radar
```

**Antenna Definition**

Assume that the antenna element has an isotropic response in the front hemisphere and all zeros in the back hemisphere. The operating frequency range is set to 8 to 12 GHz to match the 10 GHz operating frequency of the system.

```
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e9 12e9],'BackBaffled',true); % Baffled Isotropic
```

Define a 6-element uniform linear array (ULA) with a custom element pattern. Assume the element spacing to be one half the wavelength of the waveform.

```
fc = radiator.OperatingFrequency;
c = radiator.PropagationSpeed;
lambda = c/fc;
ula = phased.ULA('Element',antenna,'NumElements',6,...
    'ElementSpacing',lambda/2);

pattern(ula,fc,'PropagationSpeed',c,'Type','powerdb')
title('Six-Element Baffled ULA Response Pattern')
view(60,50)
```
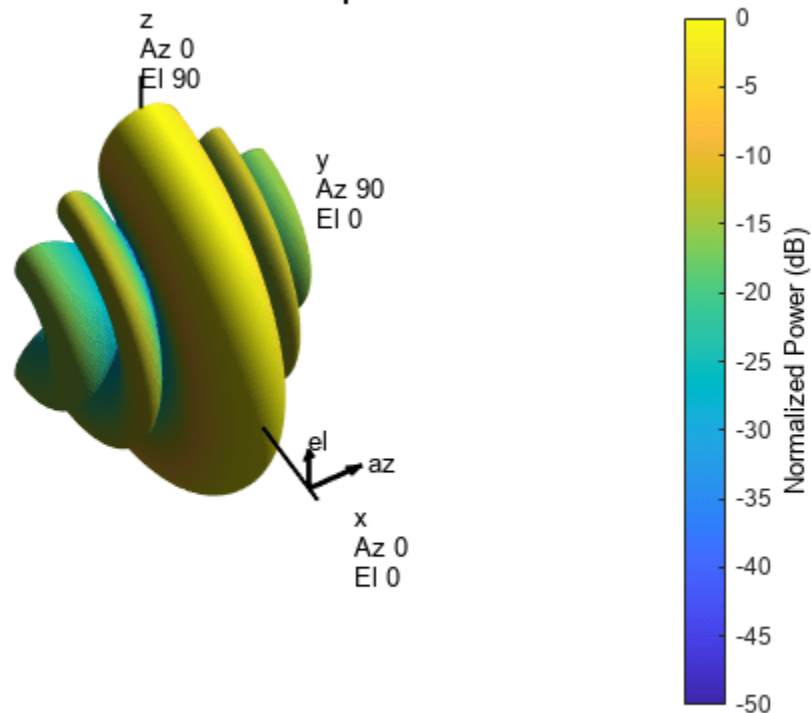


### Radar Setup

Next, mount the antenna array on the radiator and collector. Then, define the radar motion. The radar system is mounted on a plane that flies 1000 meters above the ground. The plane is flying along the array axis of the ULA at a speed such that it travels a half element spacing of the array during one pulse interval. (An explanation of such a setting is provided in the DPCA technique section that follows.)

```
radiator.Sensor = ula;
collector.Sensor = ula;
sensormotion = phased.Platform('InitialPosition',[0; 0; 1000]);
```

```
arrayAxis = [0; 1; 0];
prf = waveform.PRF;
vr = ula.ElementSpacing*prf;   % in [m/s]
sensormotion.Velocity = vr/2*arrayAxis;
```

**Target**

Next, define a nonfluctuating target with a radar cross section of 1 square meter moving on the ground.

```
target = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1, ...
    'OperatingFrequency', fc);
tgtmotion = phased.Platform('InitialPosition',[1000; 1000; 0],...
    'Velocity',[30; 30; 0]);
```

**Jammer**

The target returns the desired signal; however, several interferences are also present in the received signal. If you have a Radar Toolbox license, set the variable `hasRadarToolbox` to true to define a simple barrage jammer with an effective radiated power of 100 watts. Otherwise, the simulation uses a saved jammer signal.

```
hasRadarToolbox = false;
Fs = waveform.SampleRate;
rngbin = c/2*(0:1/Fs:1/prf-1/Fs).';
if hasRadarToolbox
    jammer = barrageJammer('ERP',100);
    jammer.SamplesPerFrame = numel(rngbin);
    jammermotion = phased.Platform('InitialPosition',[1000; 1732; 1000]);
end
```

**Clutter**

This example simulates clutter using the constant-gamma model with a gamma value of $-15$ dB. Literature shows that such a gamma value can be used to model terrain covered by woods. For each range, the clutter return can be thought of as a combination of the returns from many small clutter patches on that range ring. Since the antenna is back baffled, the clutter contribution is only from the front. To simplify the computation, use an azimuth width of 10 degrees for each patch. Again, if you do not have a Radar Toolbox license, the simulation uses a saved clutter signal.

```
if hasRadarToolbox
    Rmax = 5000;
    Azcov = 120;
    clutter = constantGammaClutter('Sensor',ula,'SampleRate',Fs,...
        'Gamma',-15,'PlatformHeight',1000,...
        'OperatingFrequency',fc,...
        'PropagationSpeed',c,...
        'PRF',prf,...
        'TransmitERP',transmitter.PeakPower*db2pow(transmitter.Gain),...
        'PlatformSpeed',norm(sensormotion.Velocity),...
        'PlatformDirection',[90;0],...
        'ClutterMaxRange', Rmax,...
        'ClutterAzimuthSpan',Azcov,...
        'PatchAzimuthSpan',10,...
        'OutputFormat','Pulses');
end
```

**Propagation Paths**

Finally, create a free space environment to represent the target and jammer paths. Because the example uses a monostatic radar system, the target channel is set to simulate two-way propagation delays. The jammer path computes only one-way propagation delays.

```
tgtchannel = phased.FreeSpace('TwoWayPropagation',true,'SampleRate',Fs,...
    'OperatingFrequency', fc);
jammerchannel = phased.FreeSpace('TwoWayPropagation',false,...
    'SampleRate',Fs,'OperatingFrequency', fc);
```

**Simulation Loop**

You can now simulate the returns. Collect 10 pulses. The seed of the random number generator from the jammer model is set to a constant to get reproducible results.

```
numpulse = 10; % Number of pulses
tsig = zeros(size(rngbin,1), ula.NumElements, numpulse);
jsig = tsig; tjcsig = tsig; tcsig = tsig; csig = tsig;

if hasRadarToolbox
    jammer.SeedSource = 'Property';
    jammer.Seed = 5;
    clutter.SeedSource = 'Property';
    clutter.Seed = 5;
else
    load STAPIntroExampleData;
end

for m = 1:numpulse

    % Update sensor, target and calculate target angle as seen by the sensor
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);
    [~,tgtang] = rangeangle(tgtpos,sensorpos);

    % Update jammer and calculate the jammer angles as seen by the sensor
    if hasRadarToolbox
        [jampos,jamvel] = jammermotion(1/prf);
        [~,jamang] = rangeangle(jampos,sensorpos);
    end

    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [pulse,txstatus] = transmitter(pulse);
    pulse = radiator(pulse,tgtang);
    pulse = tgtchannel(pulse,sensorpos,tgtpos,sensorvel,tgtvel);

    % Collect target returns at sensor
    pulse = target(pulse);
    tsig(:,:,m) = collector(pulse,tgtang);

    % Collect jammer and clutter signal at sensor
    if hasRadarToolbox
        jamsig = jammer();
        jamsig = jammerchannel(jamsig,jampos,sensorpos,jamvel,sensorvel);
        jsig(:,:,m) = collector(jamsig,jamang);

        csig(:,:,m) = clutter();
    end
```

```
    % Receive collected signals
    tjcsig(:,:,m) = receiver(tsig(:,:,m)+jsig(:,:,m)+csig(:,:,m),...
        ~(txstatus>0));                          % Target + jammer + clutter
    tcsig(:,:,m) = receiver(tsig(:,:,m)+csig(:,:,m),...
        ~(txstatus>0));                          % Target + clutter
    tsig(:,:,m) = receiver(tsig(:,:,m),...
        ~(txstatus>0));                          % Target echo only
end
```

**True Target Range, Angle and Doppler**

The target azimuth angle is 45 degrees, and the elevation angle is about -35.27 degrees.

```
tgtLocation = global2localcoord(tgtpos,'rs',sensorpos);
tgtAzAngle = tgtLocation(1)
```

```
tgtAzAngle = 44.9981
```

```
tgtElAngle = tgtLocation(2)
```

```
tgtElAngle = -35.2651
```

The target range is 1732 m.

```
tgtRng = tgtLocation(3)
```

```
tgtRng = 1.7320e+03
```

The target Doppler normalized frequency is about 0.21.

```
sp = radialspeed(tgtpos, tgtmotion.Velocity, ...
                 sensorpos, sensormotion.Velocity);
tgtDp = 2*speed2dop(sp,lambda);  % Round trip Doppler
tgtDp/prf
```

```
ans = 0.2116
```

The total received signal contains returns from the target, clutter and jammer combined. The signal is a data cube with three dimensions (range bins-by-number of elements-by-number of pulses). Notice that the clutter return dominates the total return and masks the target return. The target (blue vertical line) cannot be detected without further processing.

```
ReceivePulse = tjcsig;
plot([tgtRng tgtRng],[0 0.01],rngbin,abs(ReceivePulse(:,:,1)));
xlabel('Range (m)'), ylabel('Magnitude');
title('Signals Collected by the ULA Within the First Pulse Interval')
```
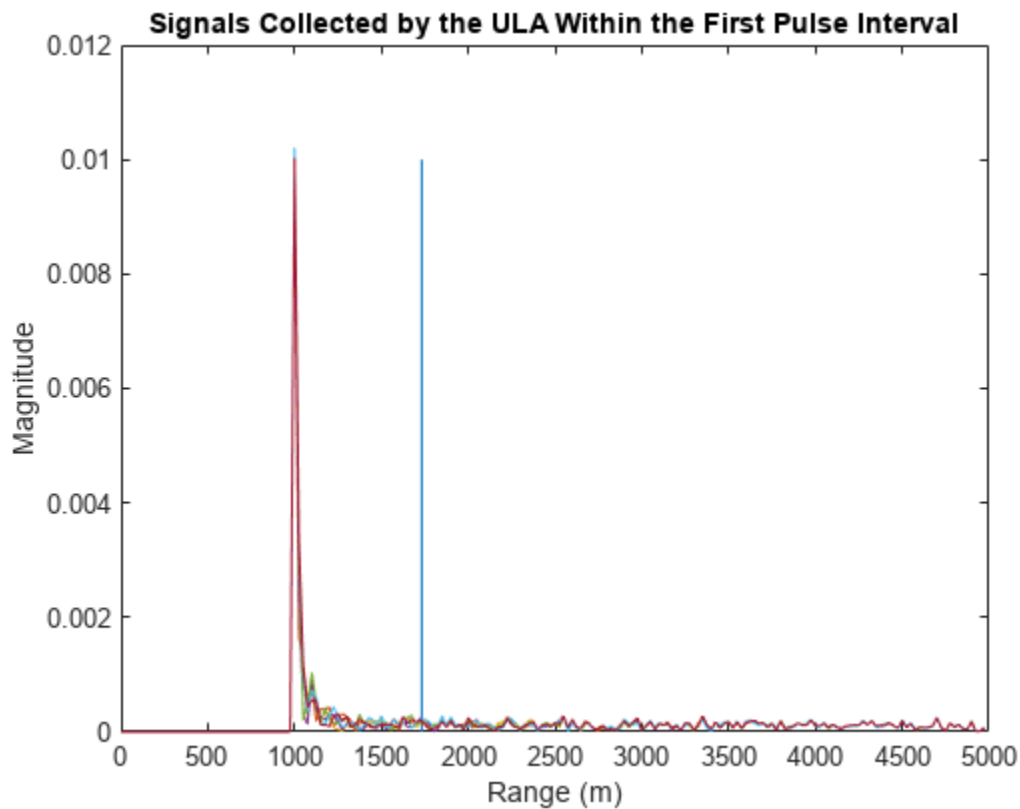
Now, examine the returns in 2-D angle Doppler (or space-time) domain. In general, the response is generated by scanning all ranges and azimuth angles for a given elevation angle. Because you know exactly where the target is, you can calculate its range and elevation angle with respect to the antenna array.

```
tgtCellIdx = val2ind(tgtRng,c/(2*Fs));
snapshot = shiftdim(ReceivePulse(tgtCellIdx,:,:));   % Remove singleton dim
angdopresp = phased.AngleDopplerResponse('SensorArray',ula,...
                'OperatingFrequency',fc, 'PropagationSpeed',c,...
                'PRF',prf, 'ElevationAngle',tgtElAngle);
plotResponse(angdopresp,snapshot,'NormalizeDoppler',true);
text(tgtAzAngle,tgtDp/prf,'+ Target')
```

**Angle-Doppler Response Pattern**
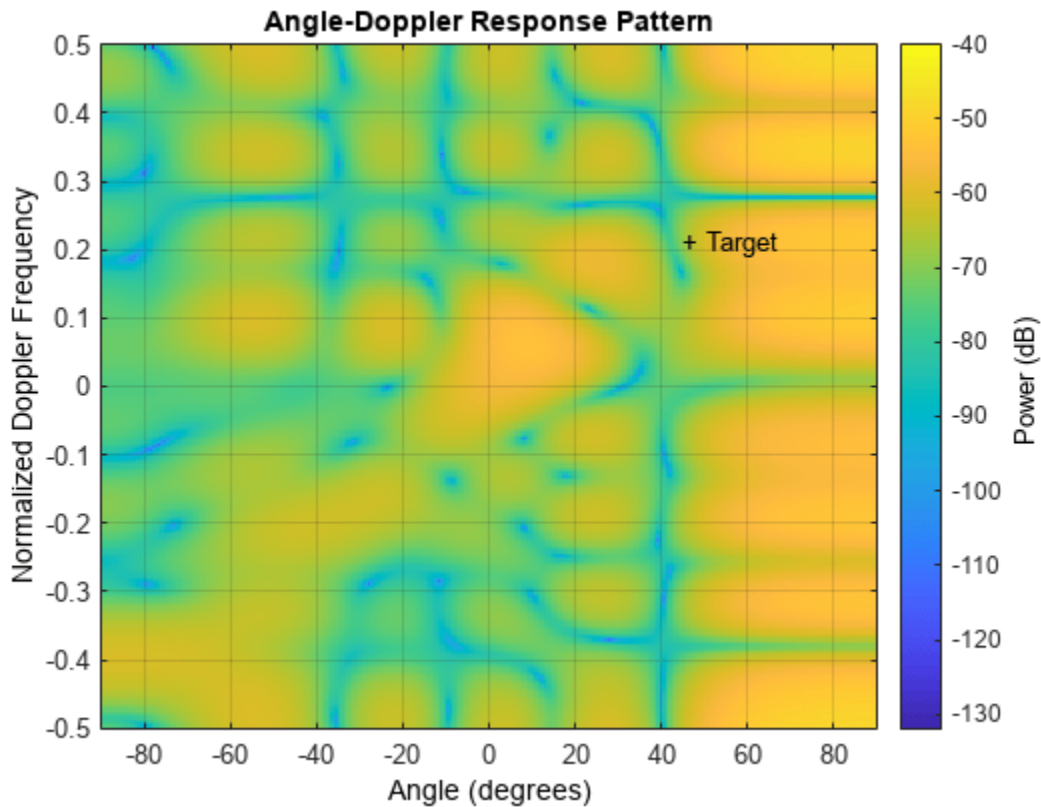
If you look at the angle-Doppler response, which is dominated by the clutter return, you see that the clutter return occupies not only the zero Doppler, but also other Doppler bins. The Doppler of the clutter return is also a function of the angle. The clutter return looks like a diagonal line in the entire angle Doppler space. Such a line is often referred to as *clutter ridge*. The received jammer signal is white noise, which spreads over the entire Doppler spectrum at a particular angle, around 60 degrees.

**Clutter Suppression with a DPCA Canceller**

The displaced phase center antenna (DPCA) algorithm is often considered to be the first STAP algorithm. This algorithm uses the shifted aperture to compensate for the platform motion, so that the clutter return does not change from pulse to pulse. Thus, this algorithm can remove the clutter via a simple subtraction of two consecutive pulses.

A DPCA canceller is often used on ULAs but requires special platform motion conditions. The platform must move along the array axis of the antenna and at a speed such that during one pulse interval, the platform travels exactly half of the element spacing. The system used here is set up, as described in earlier sections, to meet these conditions.

Assume that N is the number of ULA elements. The clutter return received at antenna 1 through antenna N-1 during the first pulse is the same as the clutter return received at antenna 2 through antenna N during the second pulse. By subtracting the pulses received at these two subarrays during the two pulse intervals, the clutter can be cancelled out. The cost of this method is an aperture that is one element smaller than the original array.

Now, define a DPCA canceller. The algorithm might need to search through all combinations of angle and Doppler to locate a target, but for the example, because you know exactly where the target is, you can direct the processor to that point.

```
rxmainlobedir = [0; 0];
stapdpca = phased.DPCACanceller('SensorArray',ula,'PRF',prf,...
    'PropagationSpeed',c,'OperatingFrequency',fc,...
    'Direction',rxmainlobedir,'Doppler',tgtDp,...
    'WeightsOutputPort',true)

stapdpca =
  phased.DPCACanceller with properties:

            SensorArray: [1x1 phased.ULA]
       PropagationSpeed: 299792458
     OperatingFrequency: 1.0000e+10
              PRFSource: 'Property'
                    PRF: 2.9979e+04
        DirectionSource: 'Property'
              Direction: [2x1 double]
     NumPhaseShifterBits: 0
          DopplerSource: 'Property'
                Doppler: 6.3429e+03
      WeightsOutputPort: true
       PreDopplerOutput: false
```

First, apply the DPCA canceller to both the target return and the clutter return.

```
ReceivePulse = tcsig;
[y,w] = stapdpca(ReceivePulse,tgtCellIdx);
```

The processed data combines all information in space and across the pulses to become a single pulse. Next, examine the processed signal in the time domain.

```
plot([tgtRng tgtRng],[0 1.2e-5],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('DPCA Canceller Output (no Jammer)')
```
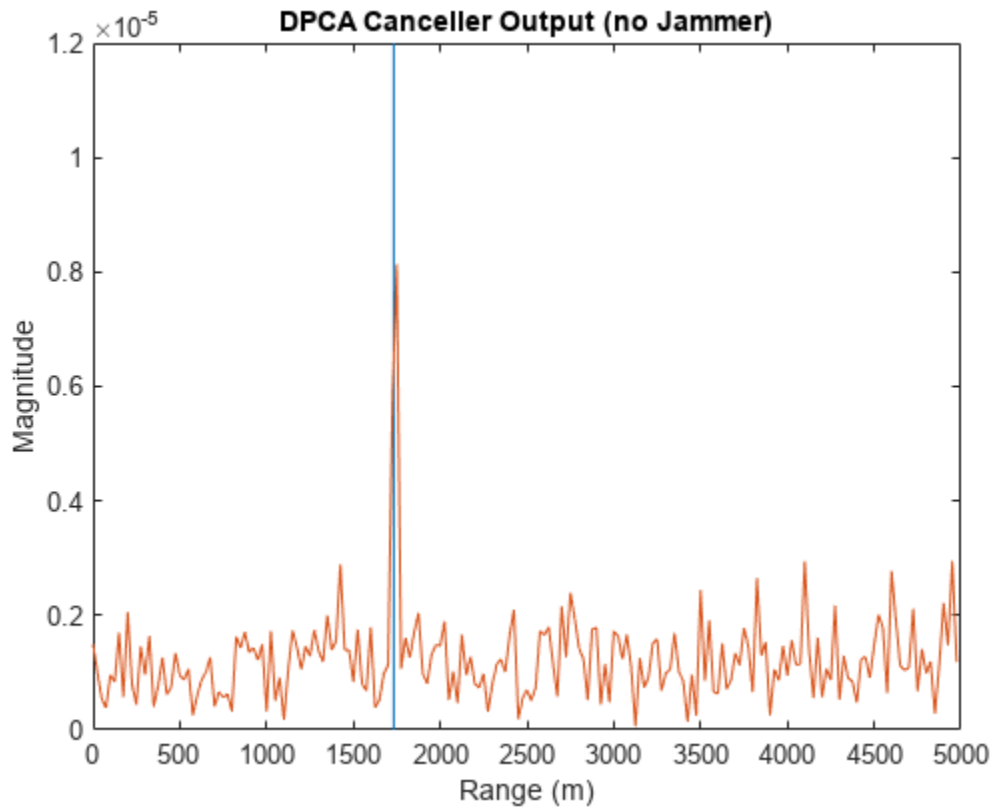
The signal now is clearly distinguishable from the noise and the clutter has been filtered out. From the angle-Doppler response of the DPCA processor weights below, you can also see that the weights produce a deep null along the clutter ridge.

```
angdopresp.ElevationAngle = 0;
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('DPCA Weights Angle Doppler Response at 0 degrees Elevation')
```

DPCA Weights Angle Doppler Response at 0 degrees Elevation
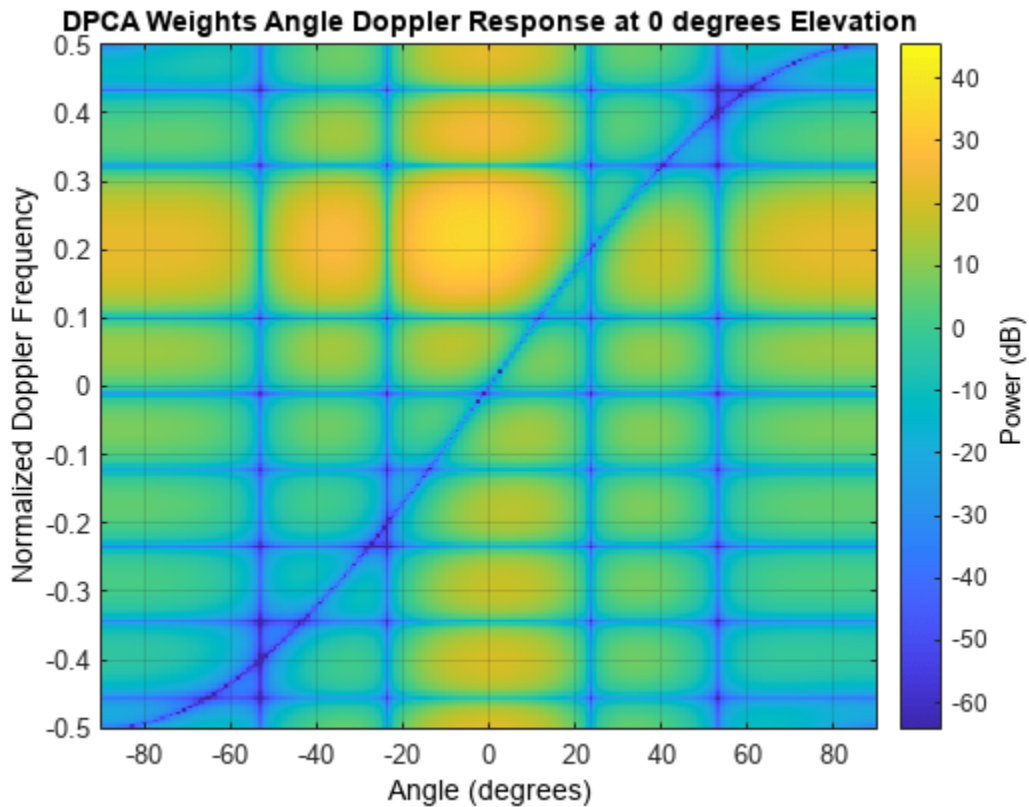
Although the results obtained by DPCA are very good, the radar platform has to satisfy very strict requirements in its movement to use this technique. Also, the DPCA technique cannot suppress the jammer interference.

Applying DPCA processing to the total signal produces the result shown in the following figure. You can see that DPCA cannot filter the jammer from the signal. The resulting angle Doppler pattern of the weights is the same as before. Thus, the processor cannot adapt to the newly added jammer interference.

```
ReceivePulse = tjcsig;
[y,w] = stapdpca(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 8e-4],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('DPCA Canceller Output (with Jammer)')
```
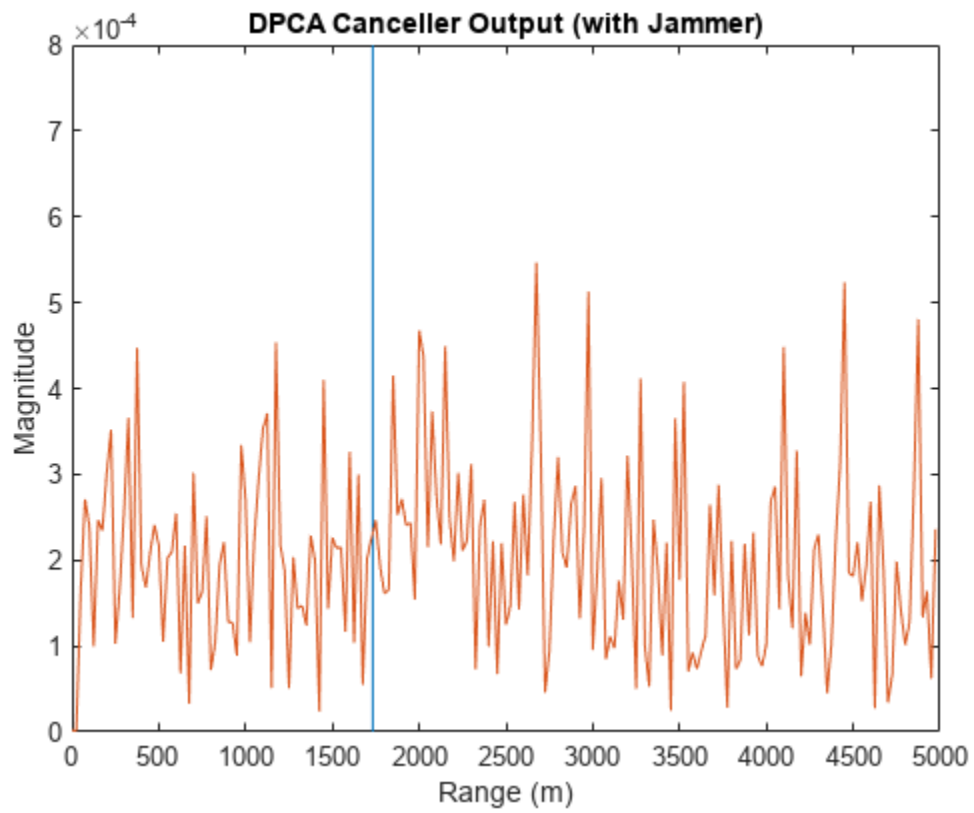
```
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('DPCA Weights Angle Doppler Response at 0 degrees Elevation')
```

DPCA Weights Angle Doppler Response at 0 degrees Elevation
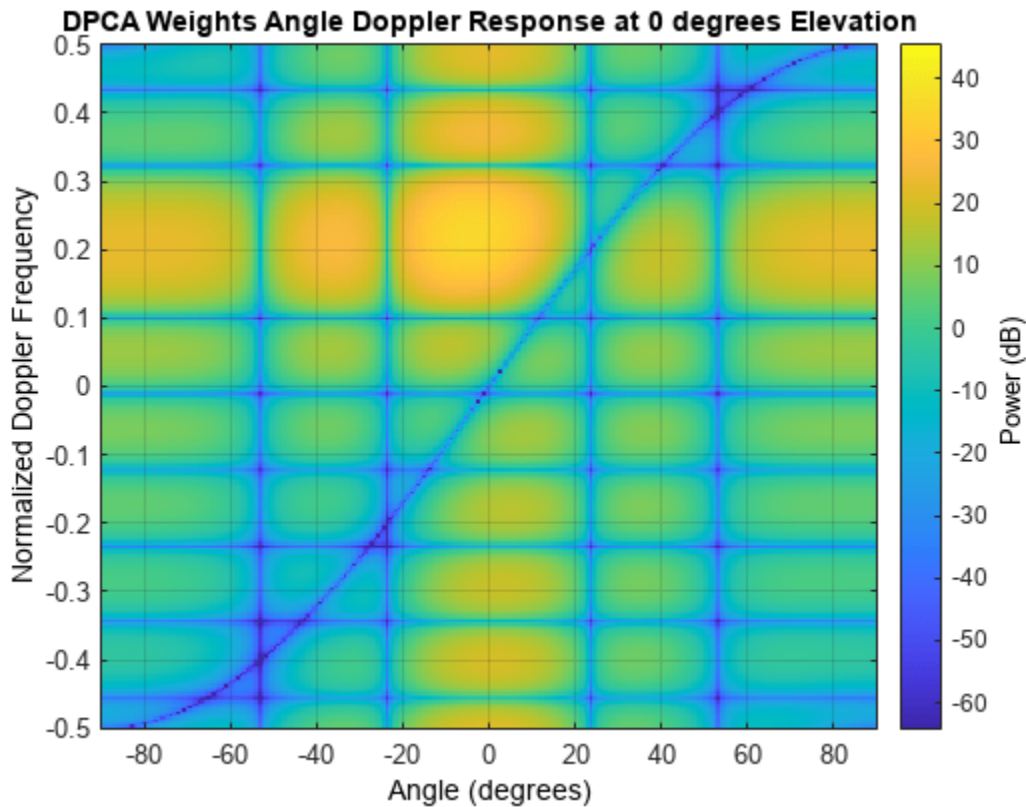
**Clutter and Jammer Suppression with an SMI Beamformer**

To suppress the clutter and jammer simultaneously, you need a more sophisticated algorithm. The optimum receiver weights, when the interference is Gaussian-distributed, are given by [1]

$$\mathbf{w} = k\mathbf{R}^{-1}\mathbf{s}$$

where $k$ is a scalar factor, $\mathbf{R}$ is the space-time covariance matrix of the interference signal, and $\mathbf{s}$ is the desired space-time steering vector. The exact information of $\mathbf{R}$ is often unavailable, so this example uses the sample matrix inversion (SMI) algorithm. The algorithm estimates $\mathbf{R}$ from training-cell samples and then uses it in the aforementioned equation.

Now, define an SMI beamformer and apply it to the signal. In addition to the information needed in DPCA, the SMI beamformer needs to know the number of guard cells and the number of training cells. The algorithm uses the samples in the training cells to estimate the interference. Hence do not use the cells that are close to the target cell for the estimates because they can contain some target information, that is, you need to define guard cells. The number of guard cells must be an even number to be split equally in front of and behind the target cell. The number of training cells also must be an even number and split equally in front of and behind the target. Normally, the larger the number of training cells, the better the interference estimate.

```
tgtAngle = [tgtAzAngle; tgtElAngle];
stapsmi = phased.STAPSMIBeamformer('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', tgtAngle, 'Doppler', tgtDp, ...
```

```
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)

stapsmi =
  phased.STAPSMIBeamformer with properties:

          SensorArray: [1x1 phased.ULA]
     PropagationSpeed: 299792458
    OperatingFrequency: 1.0000e+10
            PRFSource: 'Property'
                  PRF: 2.9979e+04
       DirectionSource: 'Property'
            Direction: [2x1 double]
    NumPhaseShifterBits: 0
        DopplerSource: 'Property'
              Doppler: 6.3429e+03
         NumGuardCells: 4
      NumTrainingCells: 100
     WeightsOutputPort: true


[y,w] = stapsmi(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 2e-6],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('SMI Beamformer Output (with Jammer)')
```
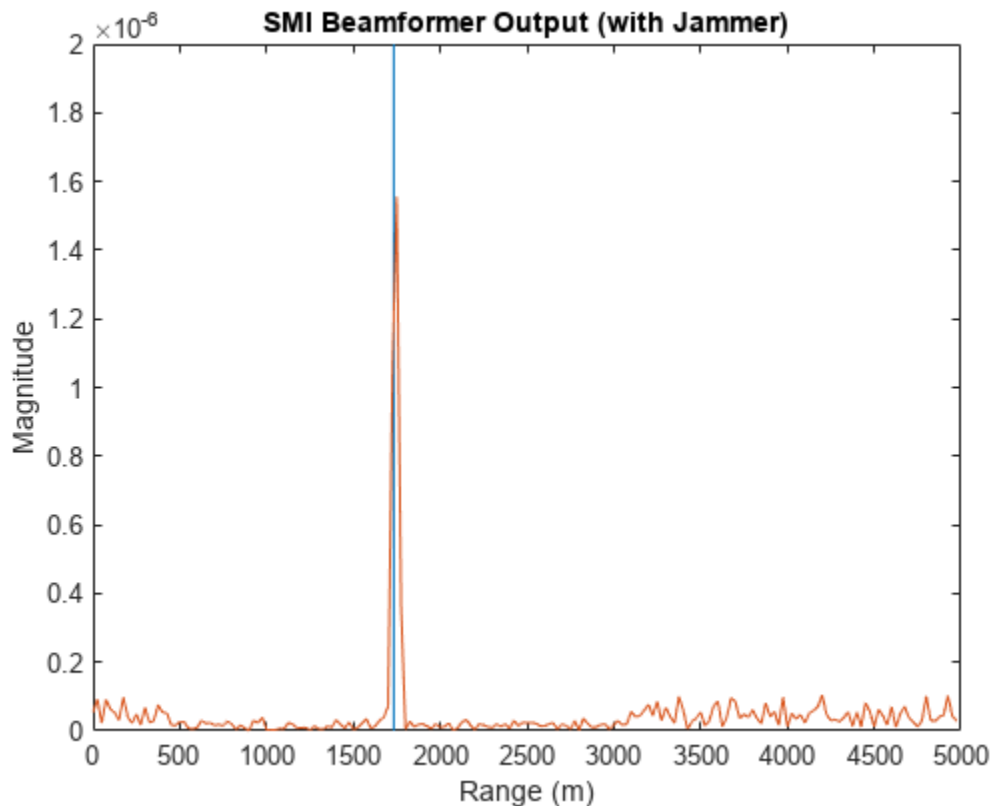


```
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('SMI Weights Angle Doppler Response at 0 degrees Elevation')
```

**SMI Weights Angle Doppler Response at 0 degrees Elevation**
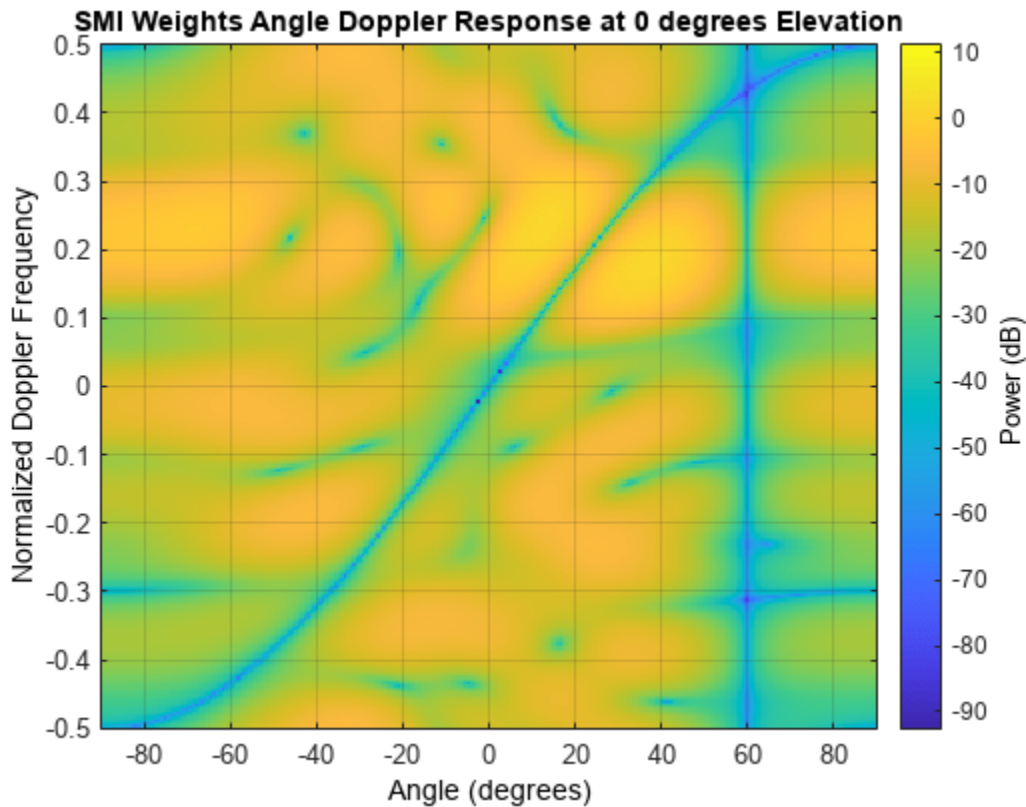


The result shows that an SMI beamformer can distinguish signals from both the clutter and the jammer. The angle Doppler pattern of the SMI weights shows a deep null along the jammer direction.

SMI provides the maximum degrees of freedom, and hence, the maximum gain among all STAP algorithms. It is often used as a baseline for comparing different STAP algorithms.

**Reducing the Computation Cost with an ADPCA Canceller**

Although SMI is the optimum STAP algorithm, it has several innate drawbacks, including a high computation cost because it uses the full dimension data of each cell. More importantly, SMI requires a stationary environment across many pulses. This kind of environment is not often found in real applications. Therefore, many reduced dimension STAP algorithms have been proposed.

An adaptive DPCA (ADPCA) canceller filters out the clutter in the same manner as DPCA, but it also has the capability to suppress the jammer as it estimates the interference covariance matrix using two consecutive pulses. Because there are only two pulses involved, the computation is greatly reduced. In addition, because the algorithm is adapted to the interference, it can also tolerate some motion disturbance.

Now, define an ADPCA canceller, and then apply it to the received signal.

```
stapadpca = phased.ADPCACanceller('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', rxmainlobedir, 'Doppler', tgtDp, ...
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)
```

```
stapadpca =
  phased.ADPCACanceller with properties:

            SensorArray: [1x1 phased.ULA]
       PropagationSpeed: 299792458
     OperatingFrequency: 1.0000e+10
              PRFSource: 'Property'
                    PRF: 2.9979e+04
         DirectionSource: 'Property'
              Direction: [2x1 double]
     NumPhaseShifterBits: 0
          DopplerSource: 'Property'
                Doppler: 6.3429e+03
          NumGuardCells: 4
       NumTrainingCells: 100
      WeightsOutputPort: true
        PreDopplerOutput: false
```

```matlab
[y,w] = stapadpca(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 2e-6],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('ADPCA Canceller Output (with Jammer)')
```
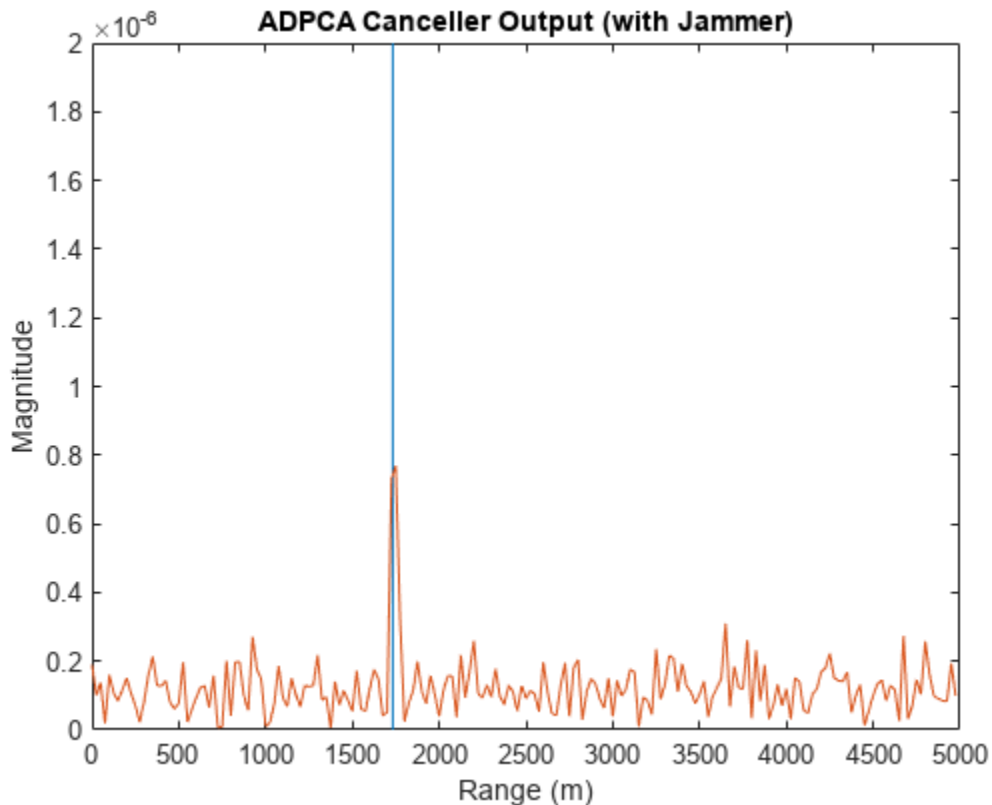


```matlab
plotResponse(angdoresp,w,'NormalizeDoppler',true);
title('ADPCA Weights Angle Doppler Response at 0 degrees Elevation')
```

ADPCA Weights Angle Doppler Response at 0 degrees Elevation
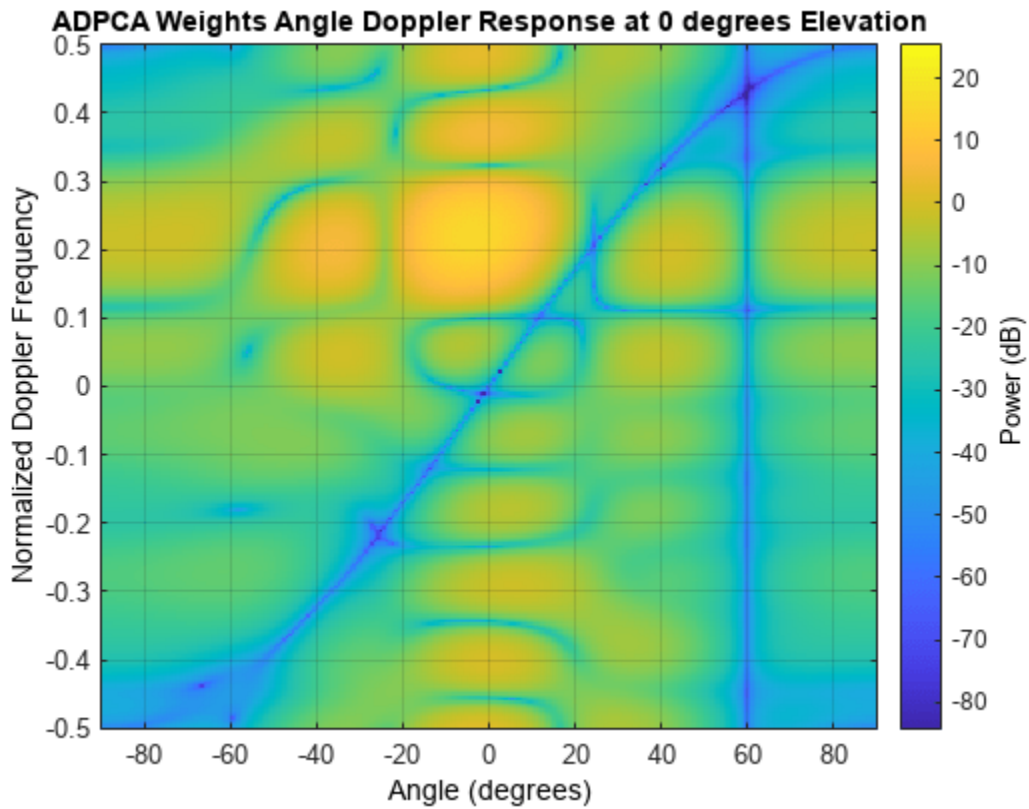
The time domain plot shows that the signal is successfully recovered. The angle Doppler response of the ADPCA weights is similar to the one produced by the SMI weights.

**Summary**

This example presented a brief introduction to STAP and illustrated how to use different STAP algorithms, namely, SMI, DPCA, and ADPCA, to suppress clutter and jammer interference in the received pulses.

**Reference**

[1] Guerci, J. R. *Space-Time Adaptive Processing for Radar*. Artech House Radar Library. Boston: Artech House, 2003.

# Source Localization Using Generalized Cross Correlation

This example shows how to determine the position of the source of a wideband signal using generalized cross-correlation (GCC) and triangulation. For simplicity, this example is confined to a two-dimensional scenario consisting of one source and two receiving sensor arrays. You can extend this approach to more than two sensors or sensor arrays and to three dimensions.

### Introduction

Source localization differs from direction-of-arrival (DOA) estimation. DOA estimation seeks to determine only the direction of a source from a sensor. Source localization determines its position. In this example, source localization consists of two steps, the first of which is DOA estimation.

1  Estimate the direction of the source from each sensor array using a DOA estimation algorithm. For wideband signals, many well-known direction of arrival estimation algorithms, such as Capon's method or MUSIC, cannot be applied because they employ the phase difference between elements, making them suitable only for narrowband signals. In the wideband case, instead of phase information, you can use the difference in the signal's time-of-arrival among elements. To compute the time-of-arrival differences, this example uses the *generalized cross-correlation with phase transformation* (GCC-PHAT) algorithm. From the differences in time-of-arrival, you can compute the DOA. (For another example of narrowband DOA estimation algorithms, see "High Resolution Direction of Arrival Estimation" on page 18-227).

2  Calculate the source position by triangulation. First, draw straight lines from the arrays along the directions-of-arrival. Then, compute the intersection of these two lines. This is the source location. Source localization requires knowledge of the position and orientation of the receiving sensors or sensor arrays.

### Triangulation Formula

The triangulation algorithm is based on simple trigonometric formulas. Assume that the sensor arrays are located at the 2-D coordinates *(0,0)* and *(L,0)* and the unknown source location is *(x,y)*. From knowledge of the sensor arrays positions and the two directions-of-arrival at the arrays, $\theta_1$ and $\theta_2$, you can compute the *(x,y)* coordinates from

$$L = y\tan\theta_1 + y\tan\theta_2$$

which you can solve for *y*

$$y = L/(\tan\theta_1 + \tan\theta_2)$$

and then for *x*

$$x = y\tan\theta_1$$

The remainder of this example shows how you can use the functions and System objects of the Phased Array System Toolbox™ to compute source position.

### Source and Sensor Geometry

Set up two receiving 4-element ULAs aligned along the *x*-axis of the global coordinate system and spaced 50 meters apart. The phase center of the first array is *(0,0,0)* . The phase center of the second array is *(50,0,0)* . The source is located at *(30,100)* meters. As indicated in the figure, the receiving array gains point in the +*y* direction. The source transmits in the -*y* direction.
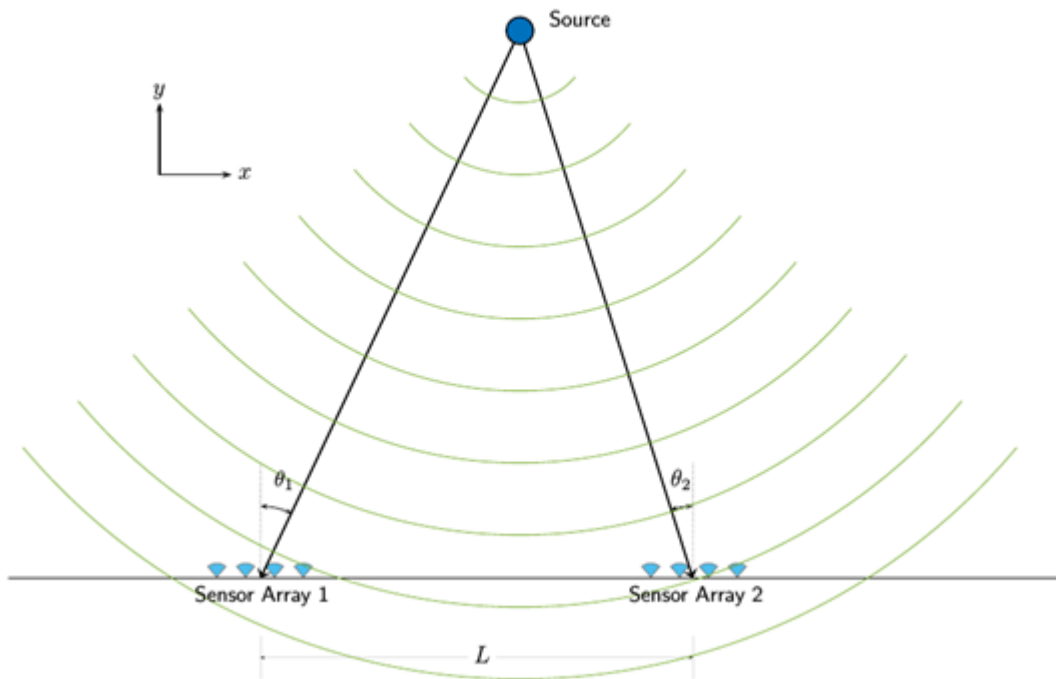
Specify the baseline between sensor arrays.

```
L = 50;
```

Create a 4-element receiver ULA of omnidirectional microphones. You can use the same `phased.ULA` System object™ for the `phased.WidebandCollector` and `phased.GCCEstimator` System objects for both arrays.

```
N = 4;
rxULA = phased.ULA('Element',phased.OmnidirectionalMicrophoneElement,...
    'NumElements',N);
```

Specify the position and orientation of the first sensor array. When you create a ULA, the array elements are automatically spaced along the y-axis. You must rotate the local axes of the array by 90° to align the elements along the x-axis of the global coordinate system.

```
rxpos1 = [0;0;0];
rxvel1 = [0;0;0];
rxax1 = azelaxes(90,0);
```

Specify the position and orientation of the second sensor array. Choose the local axes of the second array to align with the local axes of the first array.

```
rxpos2 = [L;0;0];
rxvel2 = [0;0;0];
rxax2 = rxax1;
```

Specify the signal source as a single omnidirectional transducer.

```
srcpos = [30;100;0];
srcvel = [0;0;0];
```

```
srcax = azelaxes(-90,0);
srcULA = phased.OmnidirectionalMicrophoneElement;
```

**Define Waveform**

Choose the source signal to be a wideband LFM waveform. Assume the operating frequency of the system is 300 kHz and set the bandwidth of the signal to 100 kHz. Assume a maximum operating range of 150 m. Then, you can set the pulse repetition interval (PRI) and the pulse repetition frequency (PRF). Assume a 10% duty cycle and set the pulse width. Finally, use a speed of sound in an underwater channel of 1500 m/s.

Set the LFM waveform parameters and create the `phased.LinearFMWaveform` System object.

```
fc = 300e3;               % 300 kHz
c = 1500;                 % 1500 m/s
dmax = 150;               % 150 m
pri = (2*dmax)/c;
prf = 1/pri;
bw = 100.0e3;             % 100 kHz
fs = 2*bw;
waveform = phased.LinearFMWaveform('SampleRate',fs,'SweepBandwidth',bw,...
    'PRF',prf,'PulseWidth',pri/10);
```

The transmit signal can then be generated as

```
signal = waveform();
```

**Radiate, Propagate, and Collect Signals**

Modeling the radiation and propagation for wideband systems is more complicated than modeling narrowband systems. For example, the attenuation depends on frequency. The Doppler shift as well as the phase shifts among elements due to the signal incoming direction also vary according to the frequency. Thus, it is critical to model those behaviors when dealing with wideband signals. This example uses a subband approach.

Set the number of subbands to 128.

```
nfft = 128;
```

Specify the source radiator and the sensor array collectors.

```
radiator = phased.WidebandRadiator('Sensor',srcULA,...
    'PropagationSpeed',c,'SampleRate',fs,...
    'CarrierFrequency',fc,'NumSubbands',nfft);
collector1 = phased.WidebandCollector('Sensor',rxULA,...
    'PropagationSpeed',c,'SampleRate',fs,...
    'CarrierFrequency',fc,'NumSubbands',nfft);
collector2 = phased.WidebandCollector('Sensor',rxULA,...
    'PropagationSpeed',c,'SampleRate',fs,...
    'CarrierFrequency',fc,'NumSubbands',nfft);
```

Create the wideband signal propagators for the paths from the source to the two sensor arrays.

```
channel1 = phased.WidebandFreeSpace('PropagationSpeed',c,...
    'SampleRate',fs,'OperatingFrequency',fc,'NumSubbands',nfft);
channel2 = phased.WidebandFreeSpace('PropagationSpeed',c,...
    'SampleRate',fs,'OperatingFrequency',fc,'NumSubbands',nfft);
```

Determine the propagation directions from the source to the sensor arrays. Propagation directions are with respect to the local coordinate system of the source.

```
[~,ang1t] = rangeangle(rxpos1,srcpos,srcax);
[~,ang2t] = rangeangle(rxpos2,srcpos,srcax);
```

Radiate the signal from the source in the directions of the sensor arrays.

```
sigt = radiator(signal,[ang1t ang2t]);
```

Then, propagate the signal to the sensor arrays.

```
sigp1 = channel1(sigt(:,1),srcpos,rxpos1,srcvel,rxvel1);
sigp2 = channel2(sigt(:,2),srcpos,rxpos2,srcvel,rxvel2);
```

Compute the arrival directions of the propagated signal at the sensor arrays. Because the collector response is a function of the directions of arrival in the sensor array local coordinate system, pass the local coordinate axes matrices to the `rangeangle` function.

```
[~,ang1r] = rangeangle(srcpos,rxpos1,rxax1);
[~,ang2r] = rangeangle(srcpos,rxpos2,rxax2);
```

Collect the signal at the receive sensor arrays.

```
sigr1 = collector1(sigp1,ang1r);
sigr2 = collector2(sigp2,ang2r);
```

**GCC Estimation and Triangulation**

Create the GCC-PHAT estimators.

```
doa1 = phased.GCCEstimator('SensorArray',rxULA,'SampleRate',fs,...
    'PropagationSpeed',c);
doa2 = phased.GCCEstimator('SensorArray',rxULA,'SampleRate',fs,...
    'PropagationSpeed',c);
```

Estimate the directions of arrival.

```
angest1 = doa1(sigr1);
angest2 = doa2(sigr2);
```

Triangulate the source position using the formulas established previously. Because the scenario is confined to the *x-y* plane, set the z-coordinate to zero.

```
yest = L/(abs(tand(angest1)) + abs(tand(angest2)));
xest = yest*abs(tand(angest1));
zest = 0;
srcpos_est = [xest;yest;zest]

srcpos_est = 3×1

   29.9881
  100.5743
        0
```

The estimated source location matches the true location to within 30 cm.

**Summary**

This example showed how to perform source localization using triangulation. In particular, the example showed how to simulate, propagate, and process wideband signals. The GCC-PHAT algorithm is used to estimate the direction of arrival of a wideband signal.

# Acoustic Beamforming Using Microphone Arrays

This example shows how to beamform signals received by an array of microphones to extract a desired speech signal in a noisy environment. This Simulink® example is based on the MATLAB® example "Acoustic Beamforming Using a Microphone Array" on page 18-185 for System objects.

**Structure of the Model**

The model simulates the reception of three audio signals from different directions on a 10-element uniformly linear microphone array (ULA). After the addition of thermal noise at the receiver, beamforming is applied and the result played on a sound device.
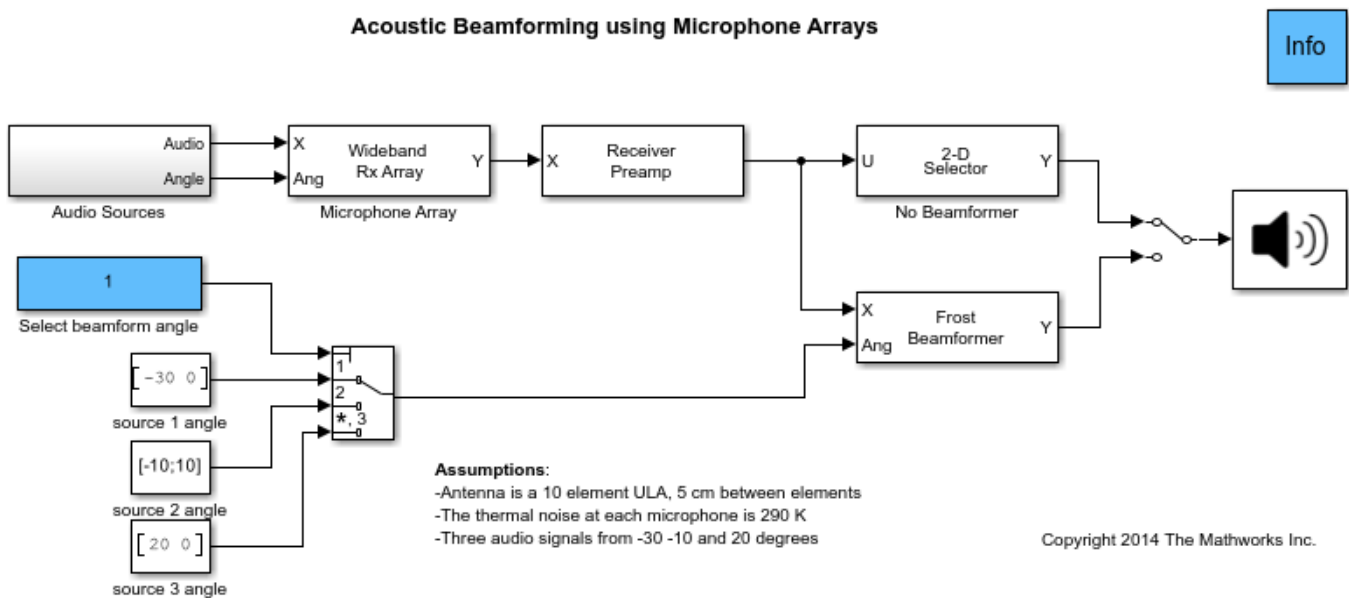


The model consists of two stages: simulate the received audio signals and beamform the result. The blocks that corresponds to each stage of the model are:

**Received audio simulation**

• `Audio Sources` - Subsystem reads the audio files and specifies their direction.
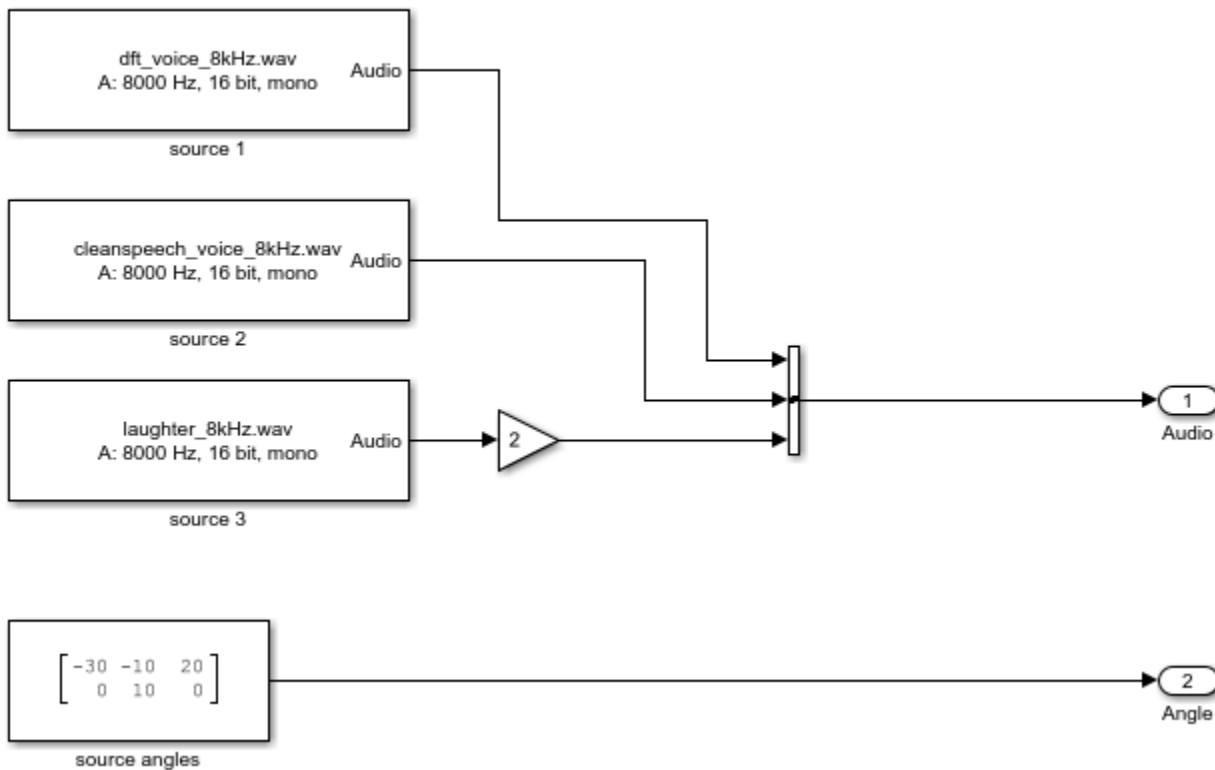
- `From Multimedia File` - Part of the `Audio Sources` subsystem, each block reads audio from a different wav file, 1000 samples at a time. Three blocks labeled `source1`, `source2` and `source3` correspond to the three sources.

- `Concatenate` - Concatenates the output of the three `From Multimedia File` blocks into a three column matrix, one column per audio signal.

- `source angles` - `Constant` block specifies the incident directions of the audio sources to the `Wideband Rx Array` block. The block outputs a 2x3 matrix. The two rows correspond to the azimuth and elevation angles in degrees of each source, the three columns correspond to the three audio signals.

- `Wideband Rx Array` - Simulates the audio signals received at the ULA. The first input port to this block is a 1000x3 matrix. Each column corresponds to the received samples of each audio signal. The second input port (Ang) specifies the incident direction of the pulses. The first row of Ang specifies the azimuth angle in degree for each signal and the second row specifies the elevation angle in degree for each signal. The second row is optional. If they are not specified, the elevation angles are assumed to be 0 degrees. The output of this block is a 1000x10 matrix. Each column corresponds to the audio recorded at each element of the microphone array. The microphone array's configuration is specified in the `Sensor Array` tab of the block dialog panel. This configuration should match the configuration specified on the block dialog panel of the `Frost Beamformer`. See the "Conventional and Adaptive Beamformers" on page 18-272 Simulink example to learn how to use sensor array configuration variables for conveniently sharing the same configuration across several blocks.
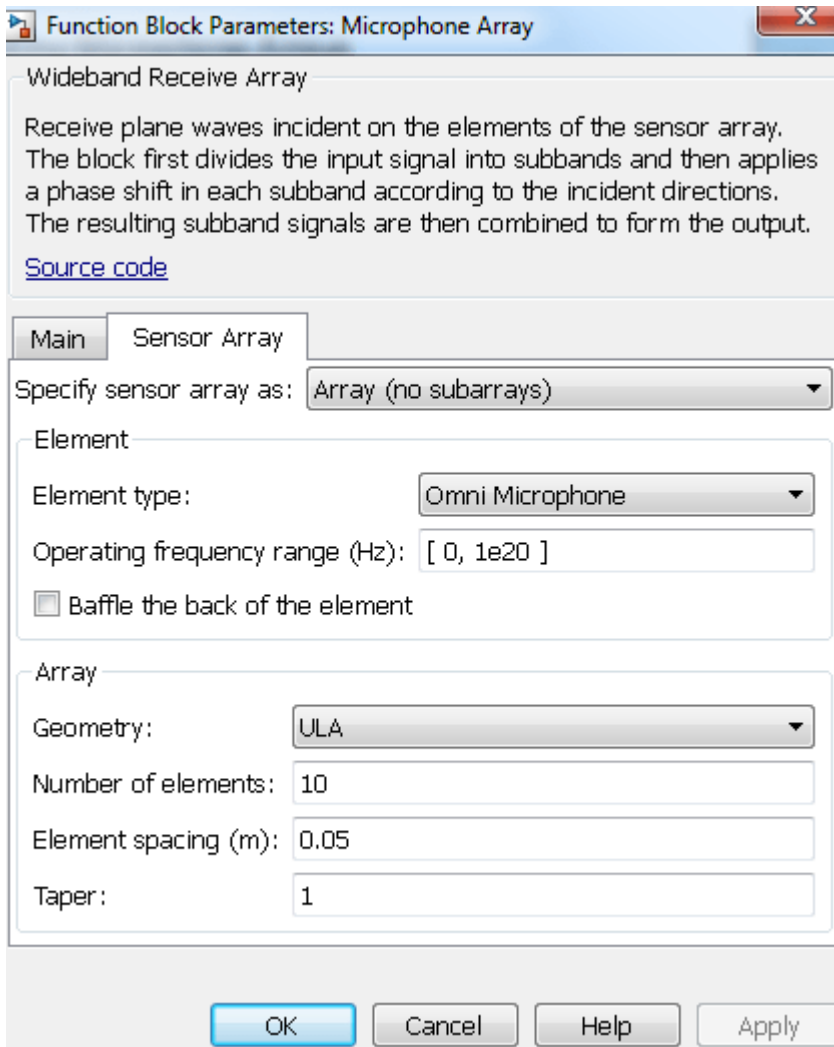
- `Receiver Preamp` - Adds white noise to the received signals.

**Beamforming**

- `Select beamform angle` - `Constant` block controls the `Multi-Port Switch` output and specifies which of the three source directions in which to beamform.
- `Frost Beamformer` - Performs Frost beamforming on the matrix passed via the input port X along the direction specified via the input port `Ang`.
- `2-D Selector` - Selects the received signal at one of the microphone elements.
- `Manual switch` - Switches between the non-beamformed and the beamformed audio stream sent to the audio device.

**Exploring the Example**

Click on the `Manual switch` while running the simulation to toggle between playing the non-beamformed audio stream and the beamformed stream. Setting a value of 1, 2, or 3 in the `Select beamform angle` block while running the simulation will beamform along one of the three audio

signals direction. You will notice that the non-beamformed audio sounds garbled while you can clearly hear any one of the selected audio streams after beamforming.

# Conventional and Adaptive Beamformers

This example shows how to apply conventional and adaptive beamforming in Simulink® to a narrowband signal received by an antenna array. The signal model includes noise and interference. This example is based on the "Conventional and Adaptive Beamformers" on page 18-201 example.

**Available Example Implementations**

This example includes two Simulink® models:

- Conventional Beamforming with Noise: slexBeamformerExample.slx
- Conventional and Adaptive Beamformers with Interference: slexBeamformerInterferenceExample.slx

**Conventional Beamforming with Noise**

The first model simulates the reception of a rectangular pulse with a delay offset on a 10-element uniformly linear antenna array (ULA). The source of the pulse is located at an azimuth of 45 degrees and an elevation of 0 degrees. Noise with power of 0.5 watts is added to the signal at each element of the array. A phase-shift beamformer is then applied. The example compares the output of the phase-shift beamformer with the signal received at one of the antenna elements.
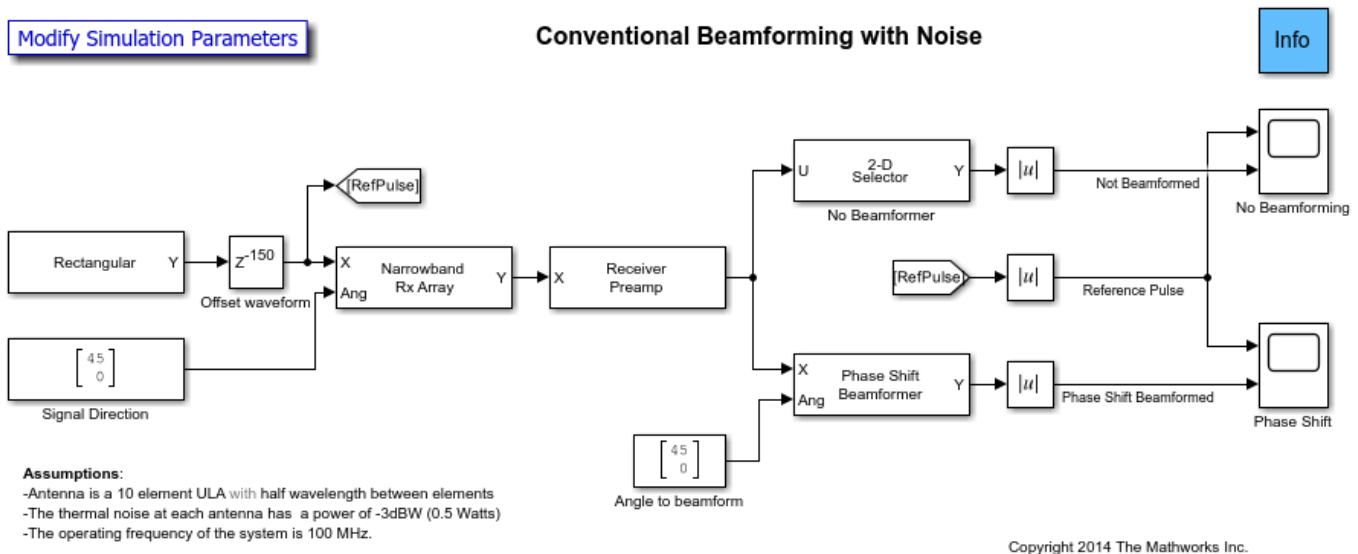


The model consists of a Signal Simulation stage and a Signal Processing stage. The blocks that corresponds to each stage of the model are:

**Signal simulation**

- `Rectangular` - Creates rectangular pulses.
- `Offset waveform` - `Delay` block delays each pulse by 150 samples.
- `Signal direction` - `Constant` block specifies the incident direction of the pulses to the `Narrowband Rx Array` block.

- **Narrowband Rx Array** - Simulates the signals received at the ULA. The first input to this block is a column vector which contains the received pulses. The pulses are assumed to be narrowband with a carrier frequency equal to the operating frequency specified in the block's dialog panel. The second input (Ang) specifies the incident direction of the pulses. The antenna array's configuration is created by a helper script as a variable in the MATLAB® workspace. This variable is referenced by the **Sensor Array** tab of the block's dialog panel. Using a variable makes it easier to share the antenna array's configuration across several blocks. Each column of the output corresponds to the signal received at each element of the antenna array.

- **Receiver Preamp** - Adds thermal noise to the received signal.

| Main | Sensor Array | |
|---|---|---|
| Specify sensor array as: | MATLAB expression | ▼ |
| Expression: | param.Antenna | |

OK   Cancel   Help   Apply
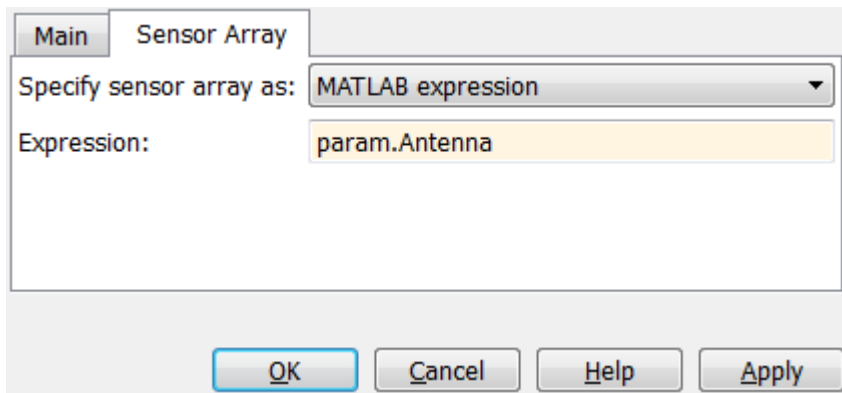
**Signal processing**

- **Angle to beamform** - **Constant** block specifies to the **Phase Shift Beamformer** the beamforming direction.
- **Phase Shift Beamformer** - Performs narrowband delay-and-sum beamforming on the matrix passed via the input port X along the direction specified via the input port Ang.
- **2-D Selector** - Selects the received signal at one of the antenna elements.

**Exploring the Example**

Several model parameters are calculated by the helper function helperslexBeamformerParam. To open the function from the model, click on **Modify Simulation Parameters** block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the model's dialog panels. To modify any parameters, either change the values in the structure from the command prompt or edit the helper function and rerun it to update the parameter structure.

**Results and Displays**

The displays below show the output of a single element (not beamformed) compared to the reference pulse and the output of the beamformer compared to the reference pulse. When the received signal is not beamformed, the pulse cannot be detected due to the noise. The display of the beamformer's output shows that the beamformed signal is much larger than the noise. The output SNR is approximately 10 times larger than that of the received signal on a single antenna, because a 10-element array produces an array gain of 10.
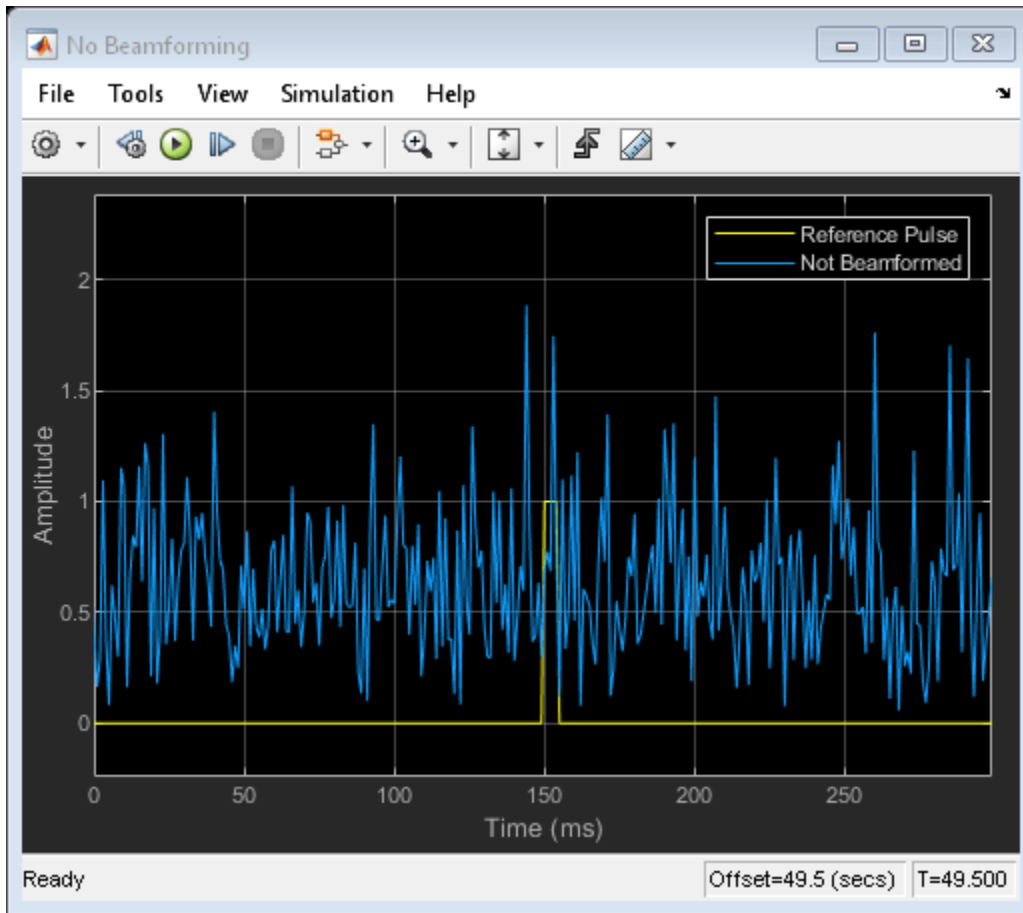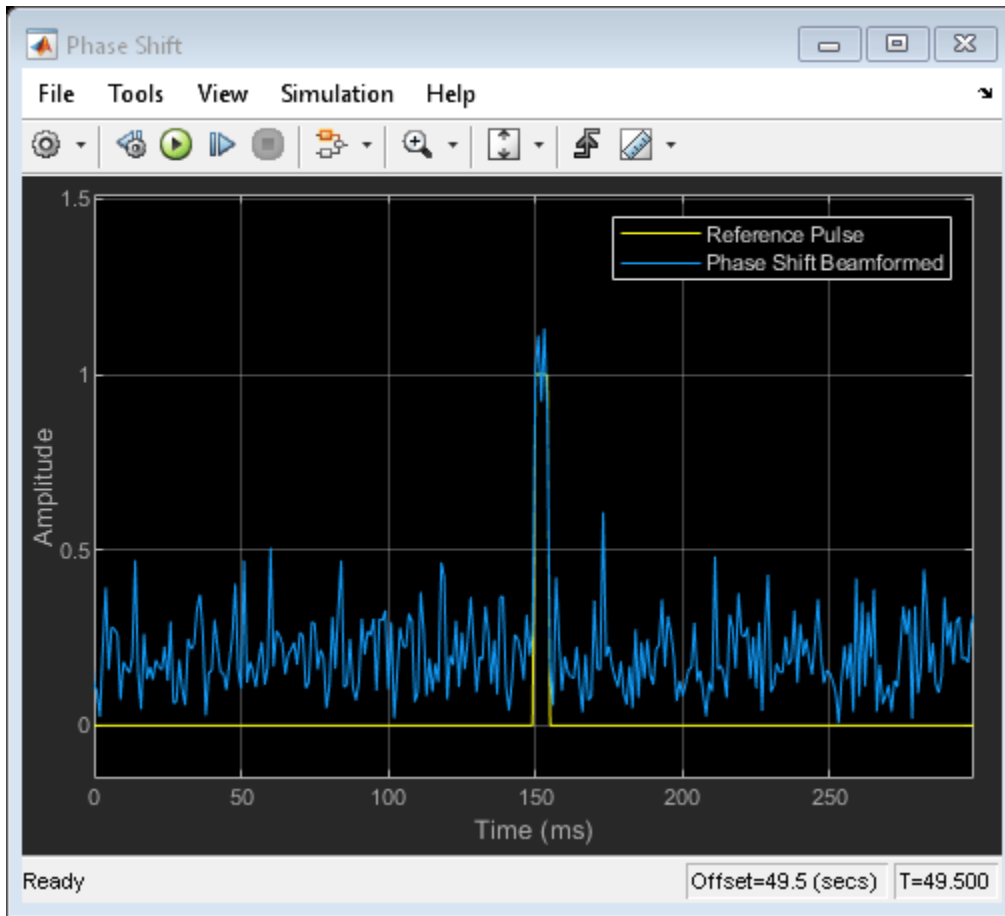
### Conventional and Adaptive Beamformers with Interference

The second model illustrates beamforming in the presence of two interference signals arriving from 30 degrees and 50 degrees in azimuth. The interference amplitudes are much larger than the pulse amplitude. The noise level is set to -50 dBW to highlight only the effect of interference. Phase shift, MVDR, and LCMV beamformers are applied to the received signal and their results are compared.
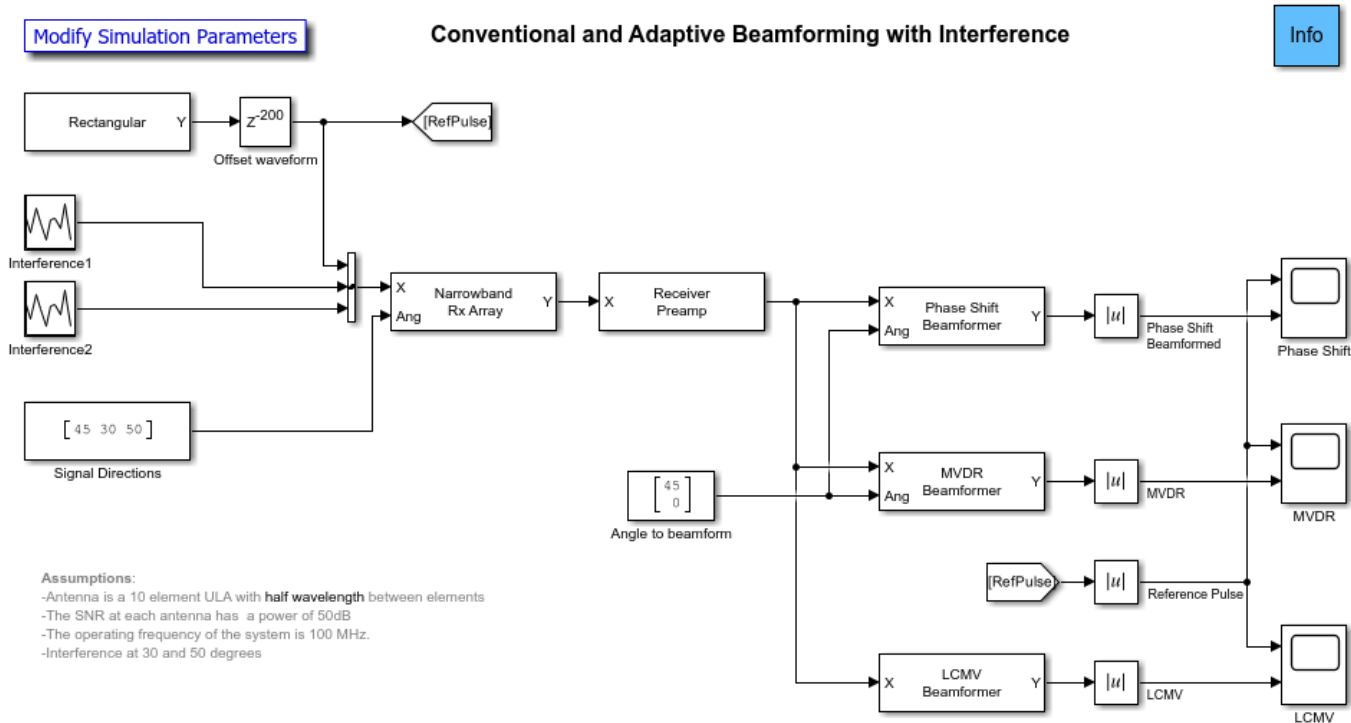
Several new blocks are added to the blocks used in the previous model:

- `Random Source` - Two blocks generate Gaussian vectors to simulate the interference signals (labeled `Interference1` and `Interference2`)
- `Concatenate` - Concatenates the outputs of the `Random Source` and the `Rectangular` blocks into a 3 column matrix.
- `Signal direction` - `Constant` block specifies the incident directions of the pulses and interference signals to the `Narrowband Rx Array` block.
- `MVDR Beamformer` - Performs MVDR beamforming along the specified direction.
- `LCMV Beamformer` - Performs LCMV beamforming with the specified constraint matrix and desired response.

**Exploring the Example**

The helper function used for this example is helperslexBeamformerParam. To open the function from the model, click on `Modify Simulation Parameters` block. The pulse, interference signal and beamforming directions can also be changed at runtime by changing the angles on the `Signal directions` and the `Angle to beamform` blocks without stopping the simulation.

**Results and Displays**

The figure below shows the output of the phased-shift beamformer. It is not able to detect the pulses because the interference signals are much stronger than the pulse signal.
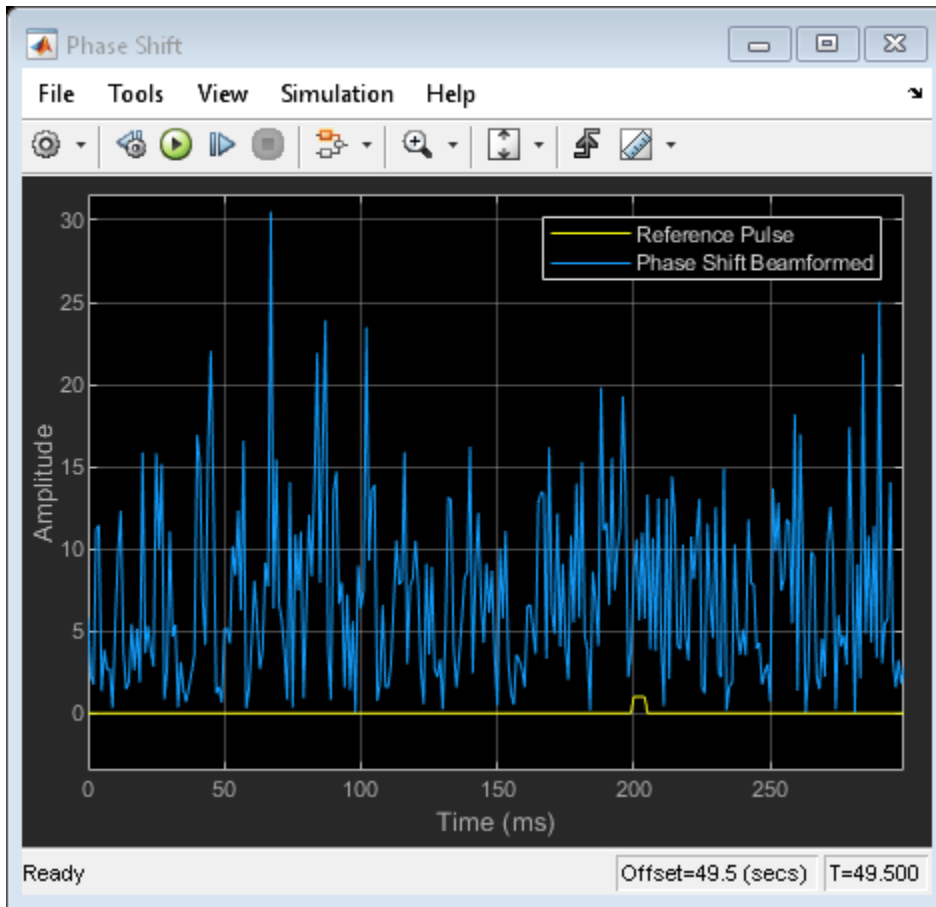
The next figure shows the output of the MVDR beamformer. An MVDR beamformer preserves the signal arriving along a desired direction, while trying to suppress signals coming from other directions. In this example, both interference signals were suppressed and the pulse at 45 degrees azimuth was preserved.
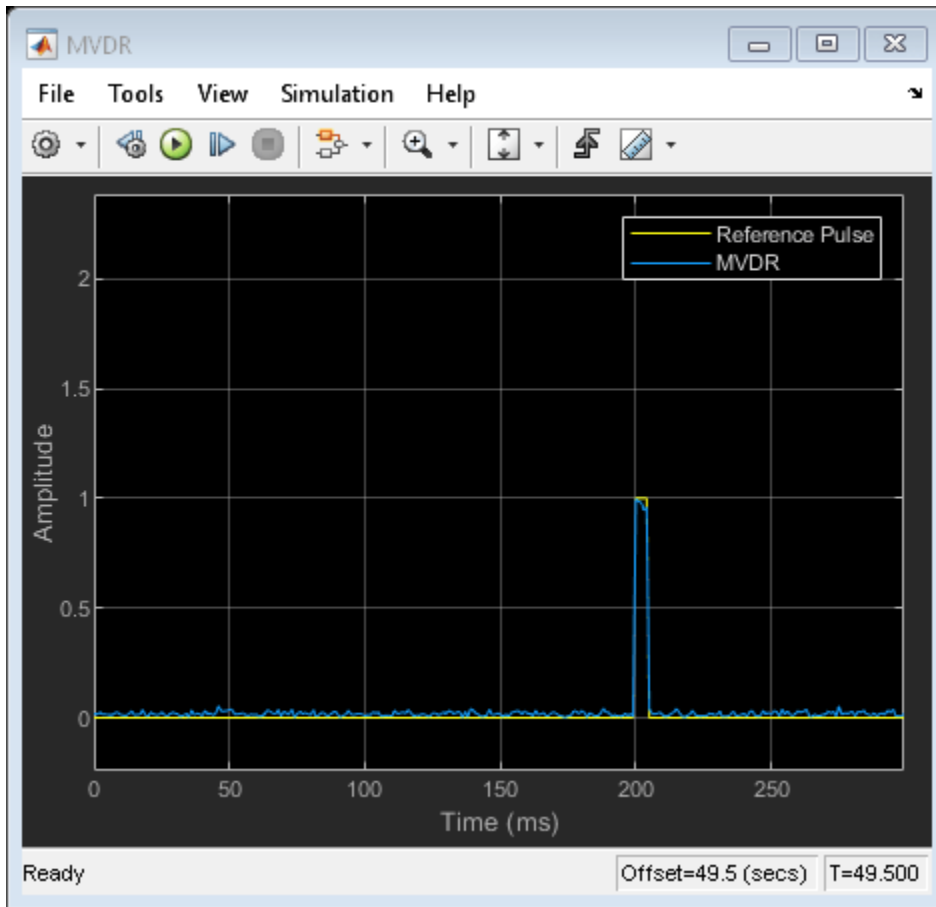
The MVDR beamformer is, however, very sensitive to the beamforming direction. If the target signal is received along a direction slightly different from the desired direction, the MVDR beamformer suppresses it. This occurs because the MVDR beamformer treats all the signals, except the one along the desired direction, as undesired interferences. This effect is sometimes referred to as "signal self-nulling". The following display shows what happens if we change the target signal's direction in the `Signal directions` block to 43 instead of 45. Notice how the received pulses have been suppressed as compared to the reference pulse.
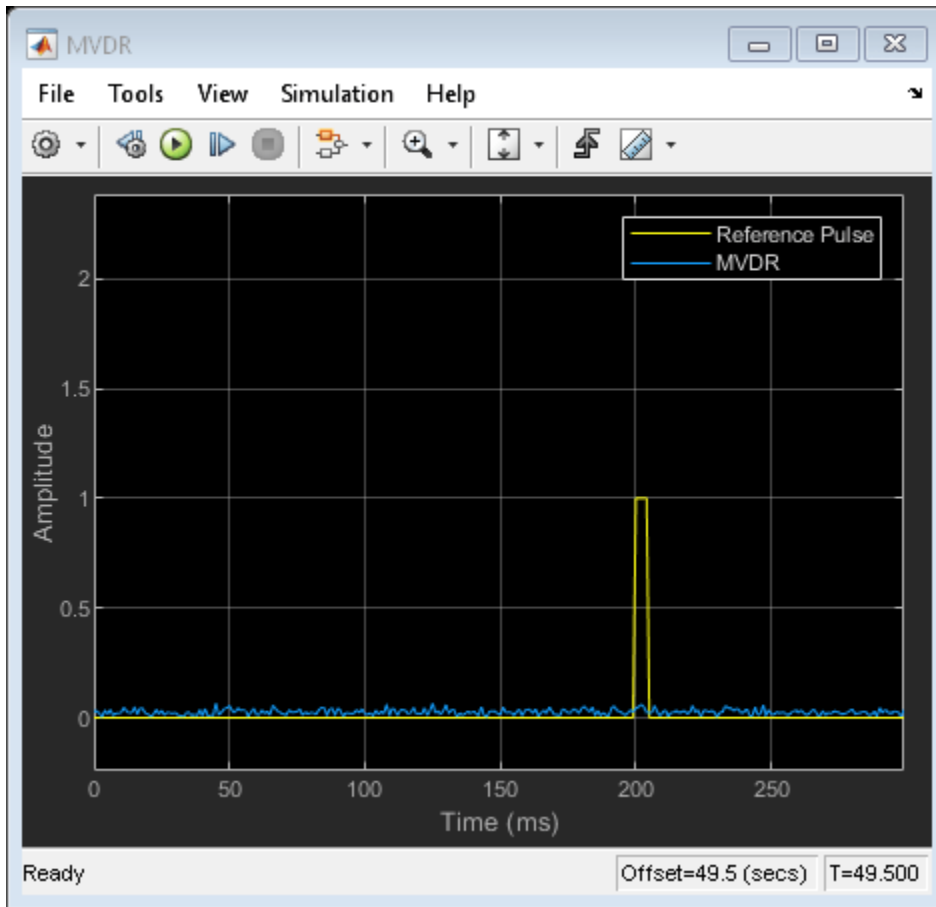
You can use an LCMV beamformer to prevent signal self-nulling by broadening the region surrounding the signal direction where you want to preserve the signal. In this example, three separate but closely-spaced constraints are imposed that preserve the response in directions corresponding to 43, 45, and 47 degrees in azimuth. The desired responses in these directions are all set to one. As shown in the figure below, the pulse is preserved.
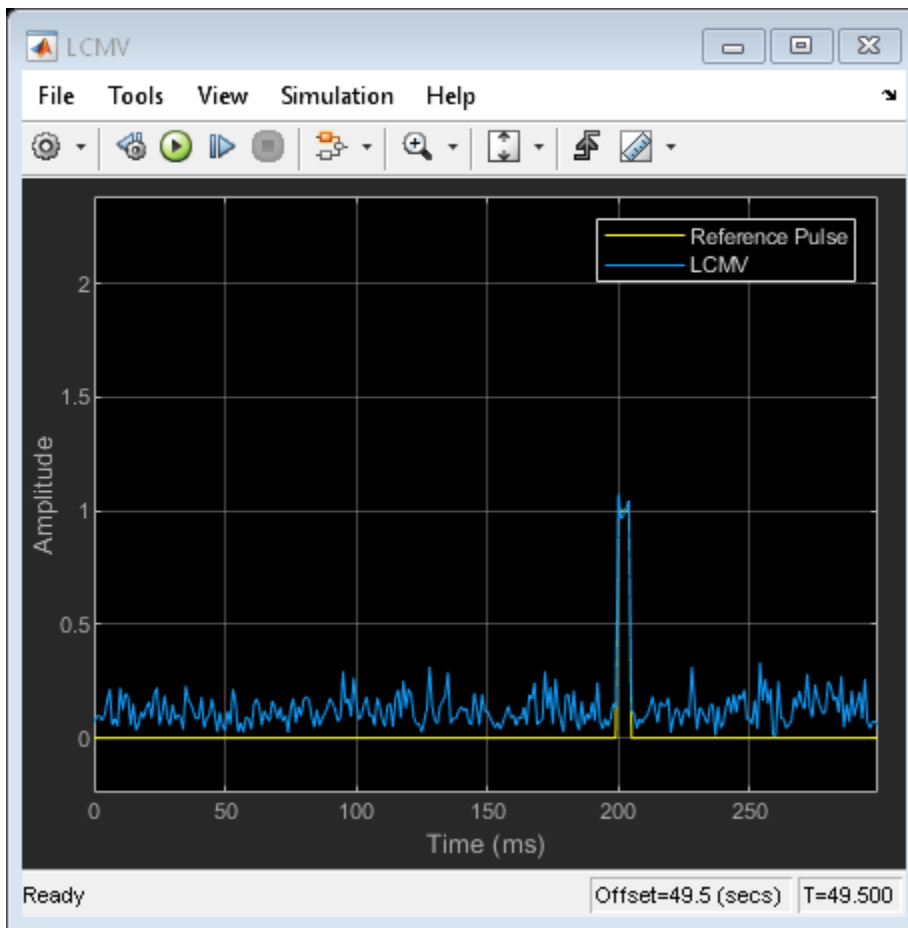
# Direction of Arrival with Beamscan and MVDR

This example shows how to use beamscan and minimum variance distortionless response (MVDR) techniques for direction of arrival (DOA) estimation in Simulink®. It is based on the MATLAB® example "Direction of Arrival Estimation with Beamscan, MVDR, and MUSIC" on page 18-216.

**Available Example Implementations**

This example includes two Simulink® models:

- Beamscan and MVDR Direction of Arrival Estimation on a ULA :
  slexBeamscanMVDRDOAExample.slx

- Beamscan and MVDR Direction of Arrival Estimation on a URA :
  slex2DBeamscanMVDRDOAExample.slx

**Beamscan and MVDR Direction of Arrival Estimation on a ULA**

This example simulates the reception of two narrowband incident signals on a 10-element uniformly linear antenna array (ULA). Both signal sources are located at 0 degrees elevation. One signal source moves from 30 degrees azimuth to 50 degrees and back. The other signal source, with 3 dB less power, moves in the opposite direction. After simulating the reception of the signals and adding noise, the beamscan and MVDR spectra are calculated. Because a ULA is symmetric around its axis, a DOA algorithm cannot uniquely determine azimuth and elevation. Therefore, the results returned by these DOA estimators are in the form of broadside angles. In this example because the elevation of the sources is at 0 degrees and the scanning region is between -90 to 90 degrees, the broadside and azimuth angles are the same. We use the Angle-Time Intensity scope to visualize the change in the direction of arrival of the signals over time.
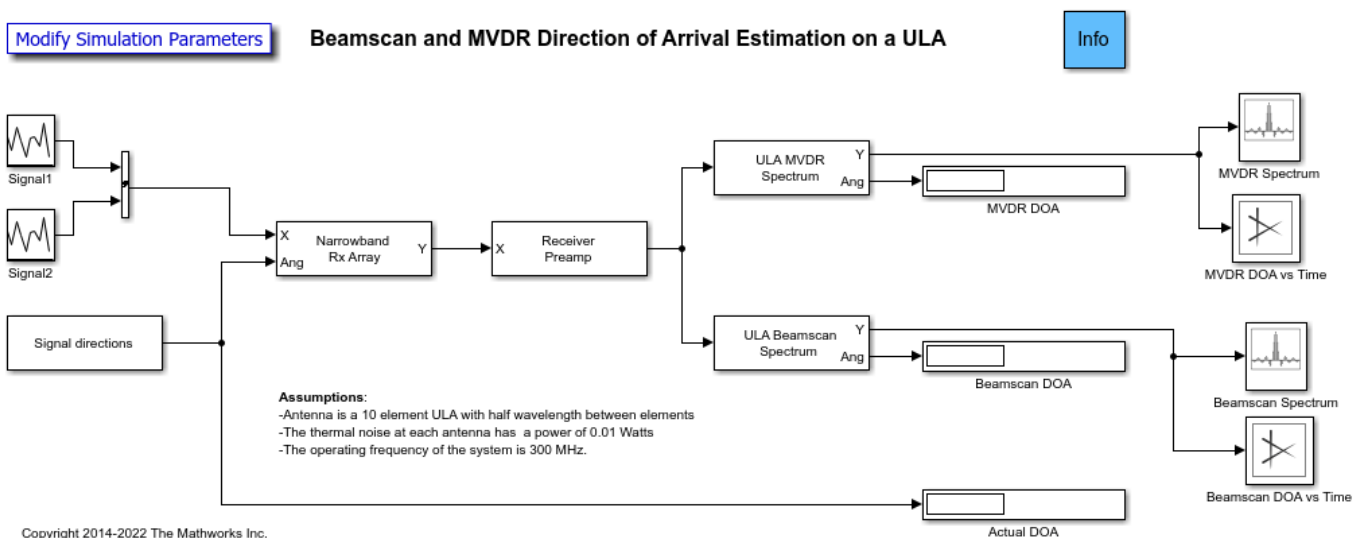


The model consists of signal simulation followed by DOA processing. The blocks used in the model are:

**Signal simulation**

- Random Source - The blocks labeled Signal1 and Signal2 generate Gaussian vectors to simulate the transmitted power of the narrowband plane waves. The signals are buffered at 300 samples per frame.

- Concatenate - Concatenates the outputs of the Random Source blocks into a 2 column matrix.

- Signal directions - Signal From Workspace block reads from the workspace, the arrival directions in degrees of each signals. The block outputs a vector of 2 angles, once per frame.

- Narrowband Rx Array - Simulates the signals received at the ULA. The first input to this block is a matrix with 2 columns. Each column corresponds to one of the received plane waves. The second input (Ang) is a 2-element vector that specifies the incident direction at the antenna array of the corresponding plane waves. The antenna array's configuration is contained in a MATLAB® workspace variable created by a helper script. This variable is used in the Sensor Array tab of the dialog. Using a variable makes it easier to share the antenna array's configuration across several blocks.
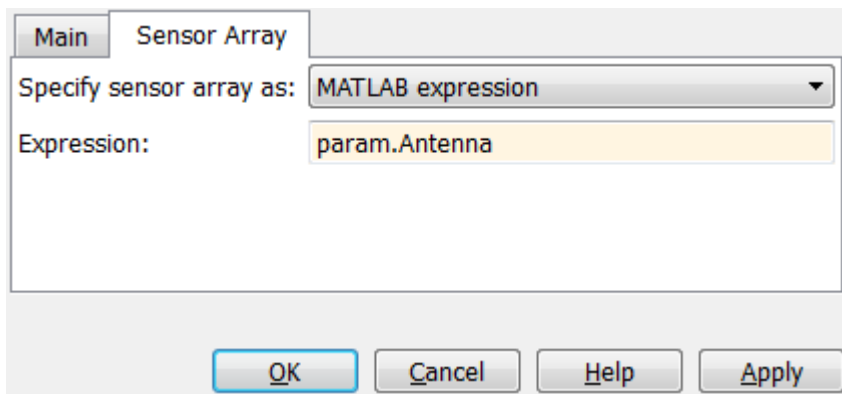


- Receiver Preamp - Adds thermal noise to the received signal.

**DOA processing**

- ULA MVDR Spectrum - Calculates the spatial spectrum of the incoming narrowband signals using the MVDR algorithm. This block also calculates the direction of arrivals of the incoming signals.

- ULA Beamscan Spectrum - Calculates the spatial spectrum of the incoming narrowband signals by scanning a region using a narrowband conventional beamformer. This block also calculates the direction of arrivals of the incoming signals.

**Exploring the Example**

Several dialog parameters of the model are calculated by the helper function helperslexBeamscanMVDRDOAParam. To open the function from the model, click on Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

**Results and Displays**

The beamscan spectrum is updated as the sources move towards each other. The spectrum shows two wide peaks with different magnitudes moving in opposite directions.
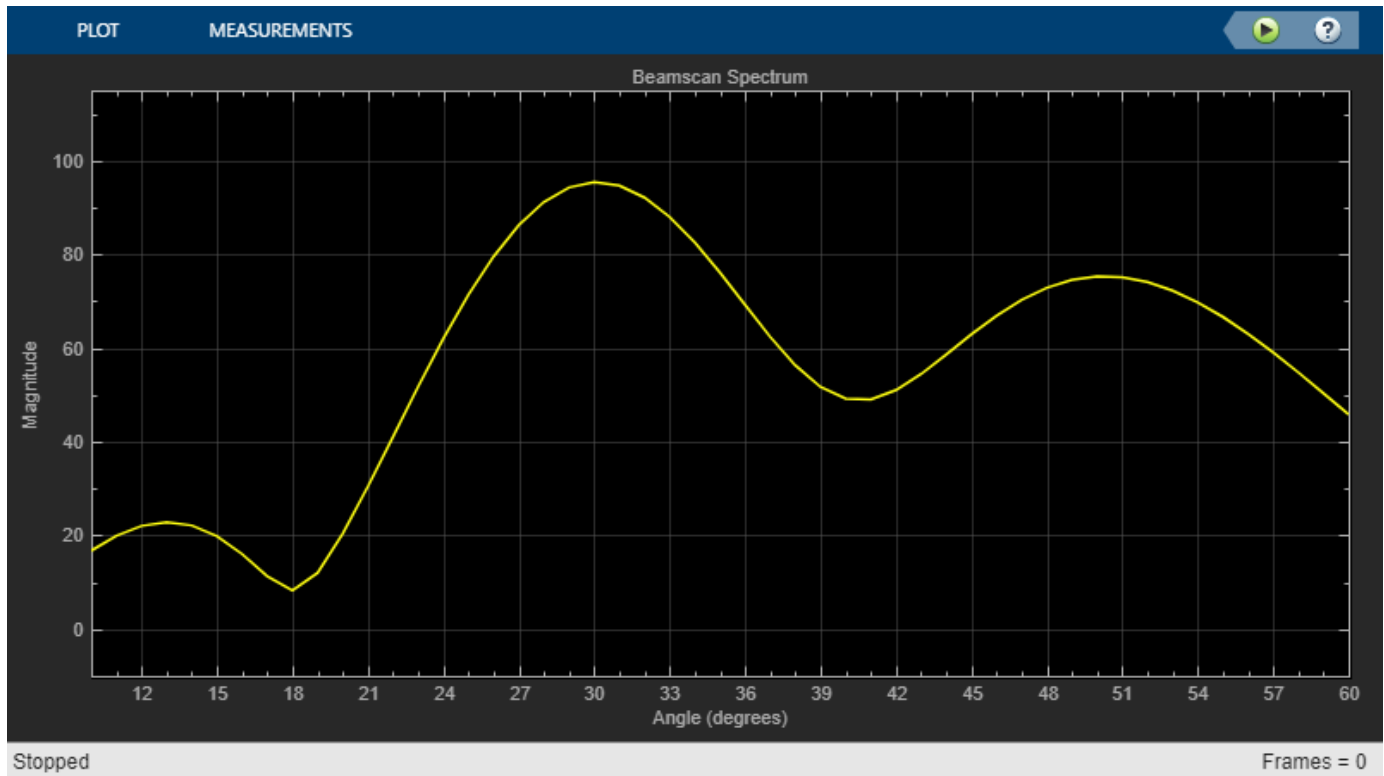
When the sources are approximately 10 degrees apart the peaks merge and the DOA of the signals are not clearly distinguished. The calculated DOA will start drifting from the actual values, as shown in the displays. When two signals arrive from directions separated by less than the beamwidth, their DOA's cannot be resolved accurately using the beamscan method.
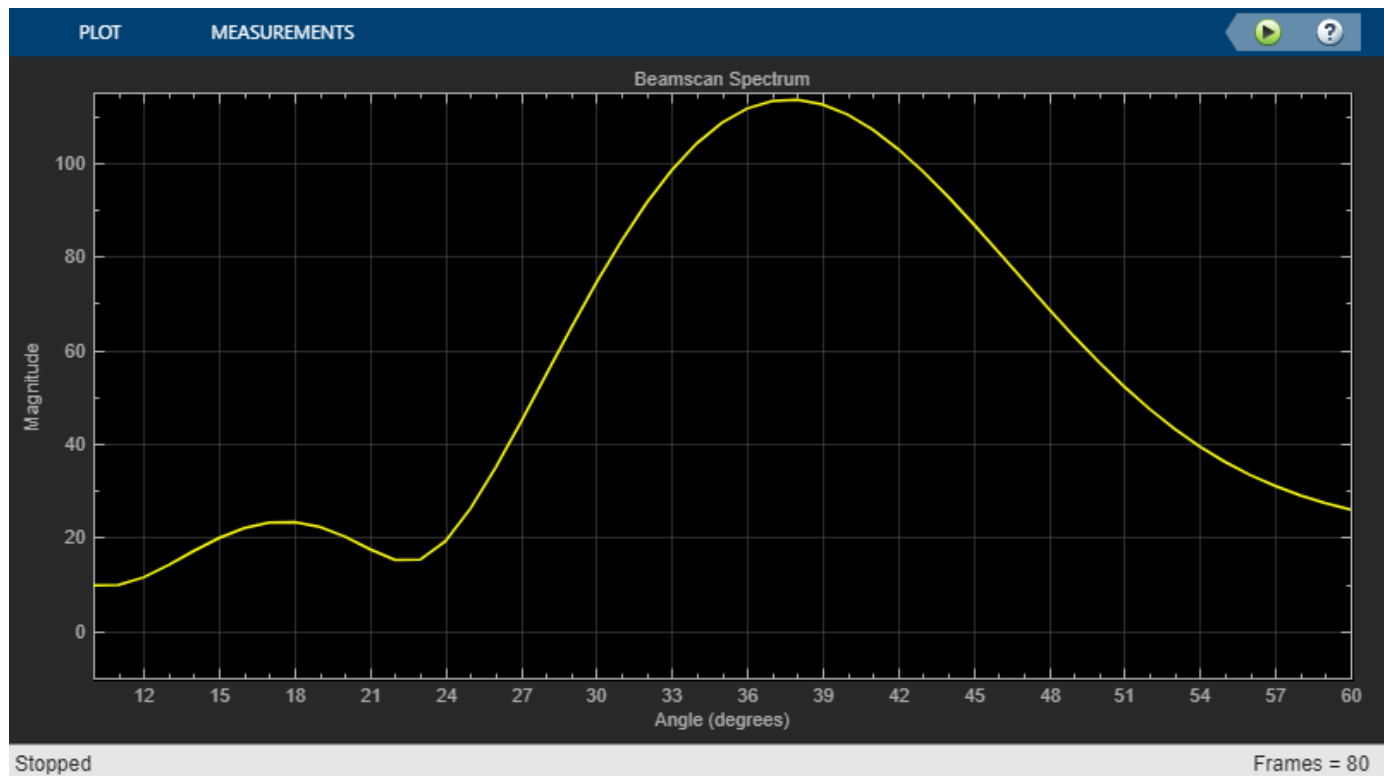
The MVDR spectrum, on the other hand, has a higher resolution. The peaks in the spectrum are narrower and can be distinguished even when the sources are very close to each other. The MVDR algorithm is very sensitive to the sources' locations. It tries to filter out signals that are not located precisely at one of the scan angles specified on the `ULA MVDR Spectrum` block. The peaks are greatest when the sources are located at one of the specified scan angles. They will pulsate as the sources move from one of the specified scan angles to another.
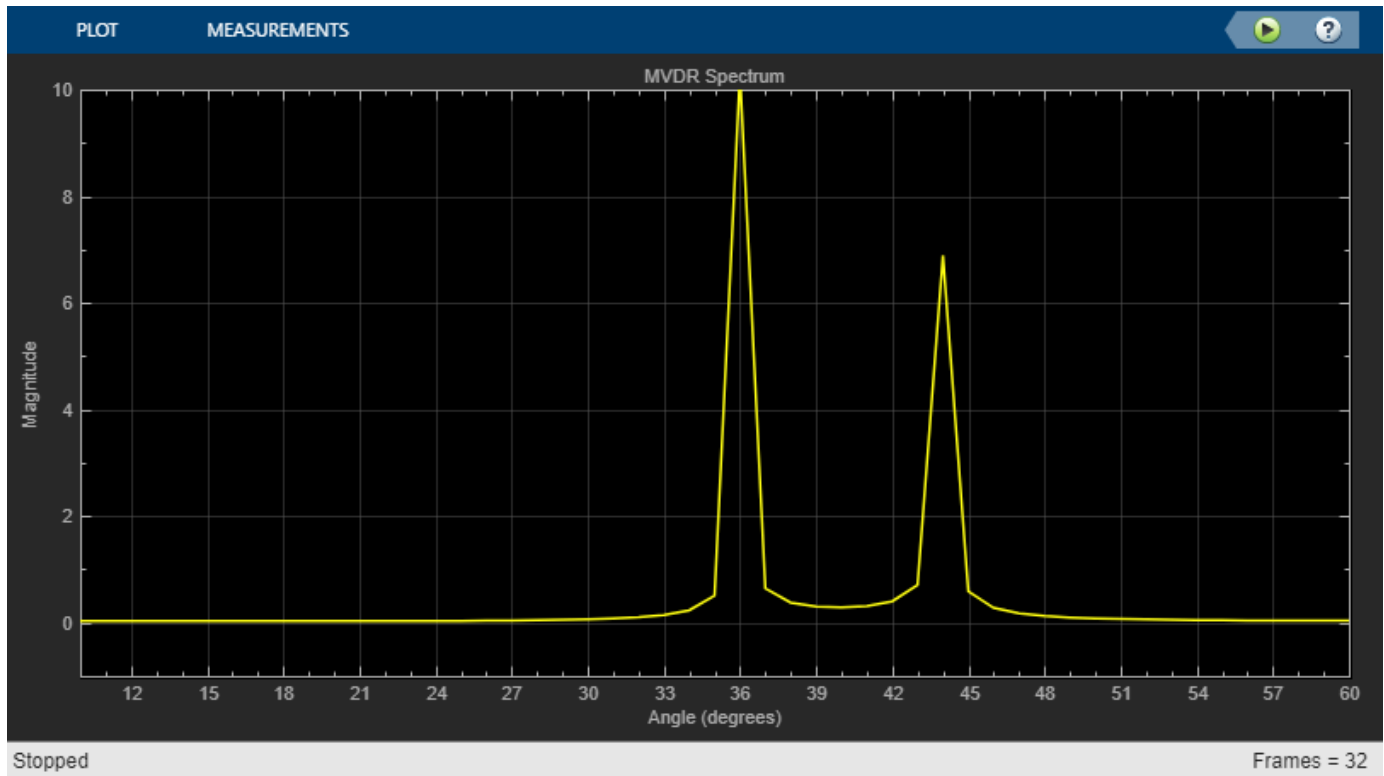
The Angle-Time Intensity scope help to visualize the change in spatial spectrum over time, the spatial spectrum helps to know the direction of the arriving signal. It evident that the spectrum estimated via MVDR algorithm helps to locate the location accurately compared to the spectrum estimated via the beamscan method.
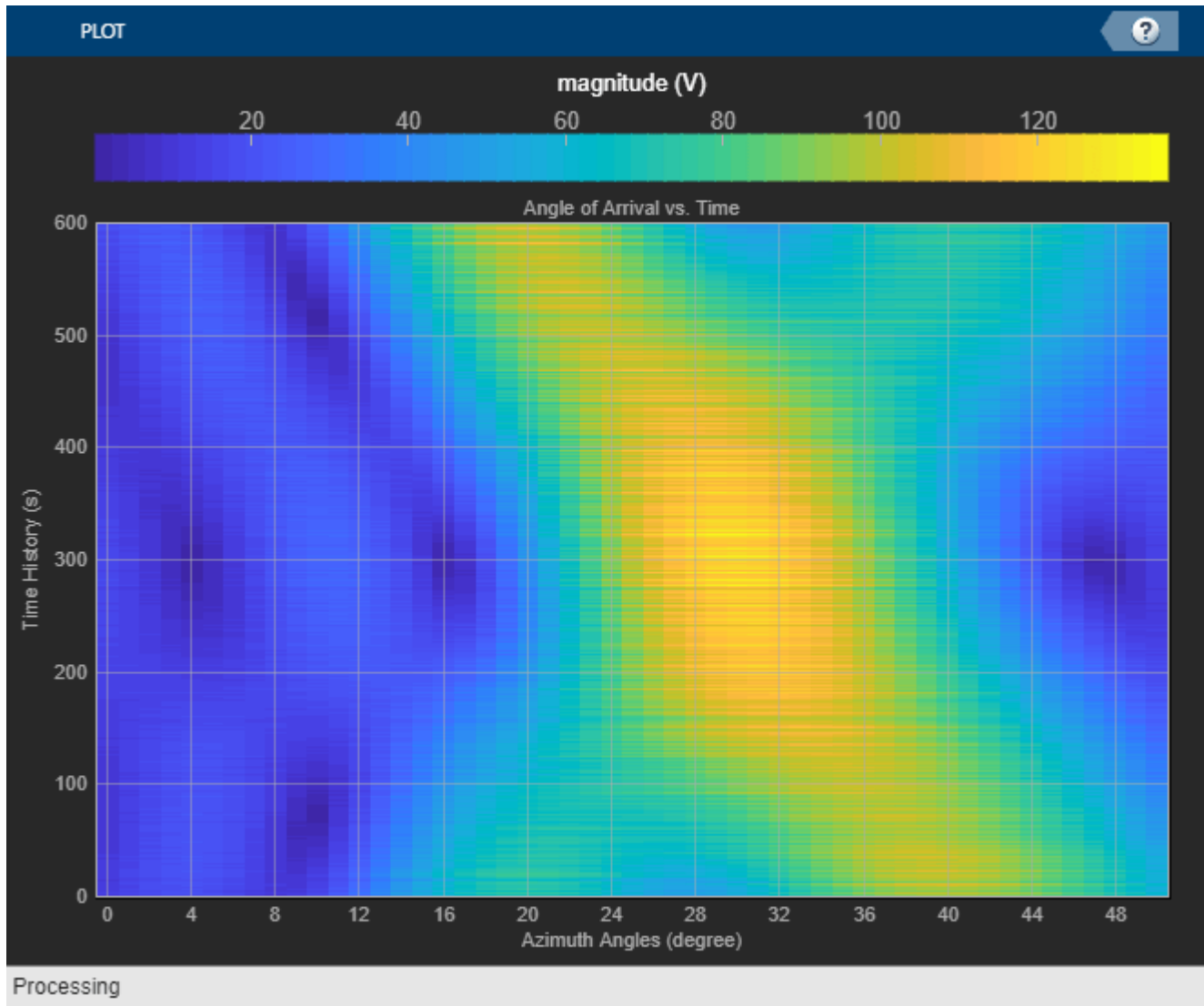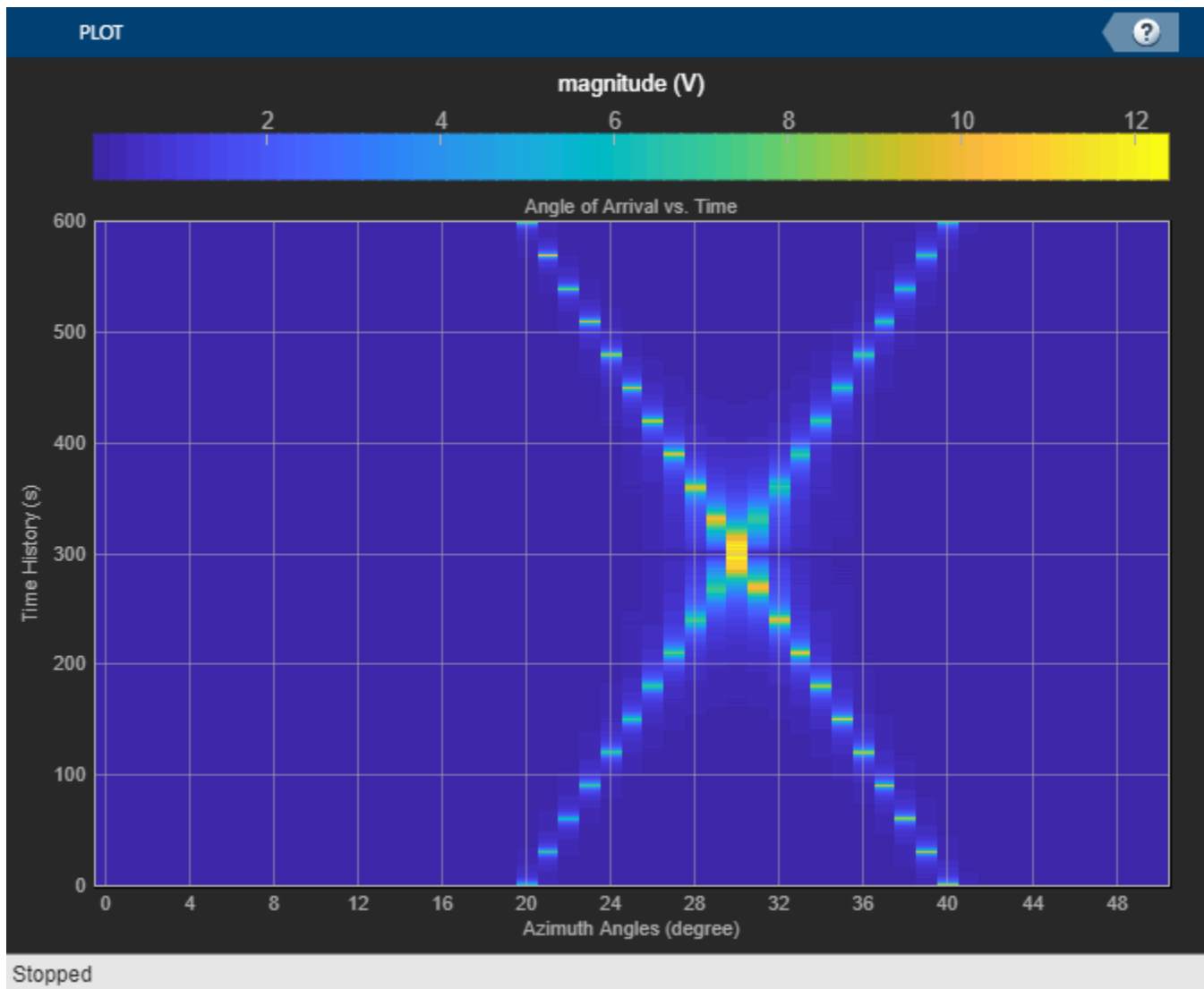
**Beamscan and MVDR Direction of Arrival Estimation on a URA**

This example replaces the ULA configuration of the previous example with a 10 by 5 uniformly rectangular antenna array (URA). One signal source moves from 30 degrees azimuth, 10 degrees elevation to 50 degrees azimuth, -5 degrees elevation. The other signal source, with 3 dB less power, moves in the opposite direction. Rectangular arrays allow the DOA estimators to determine both the azimuth and elevation. Matrix viewers are used instead of vector scopes to visualize the 2 dimensional spatial spectrum. Everything else is similar to the previous example.
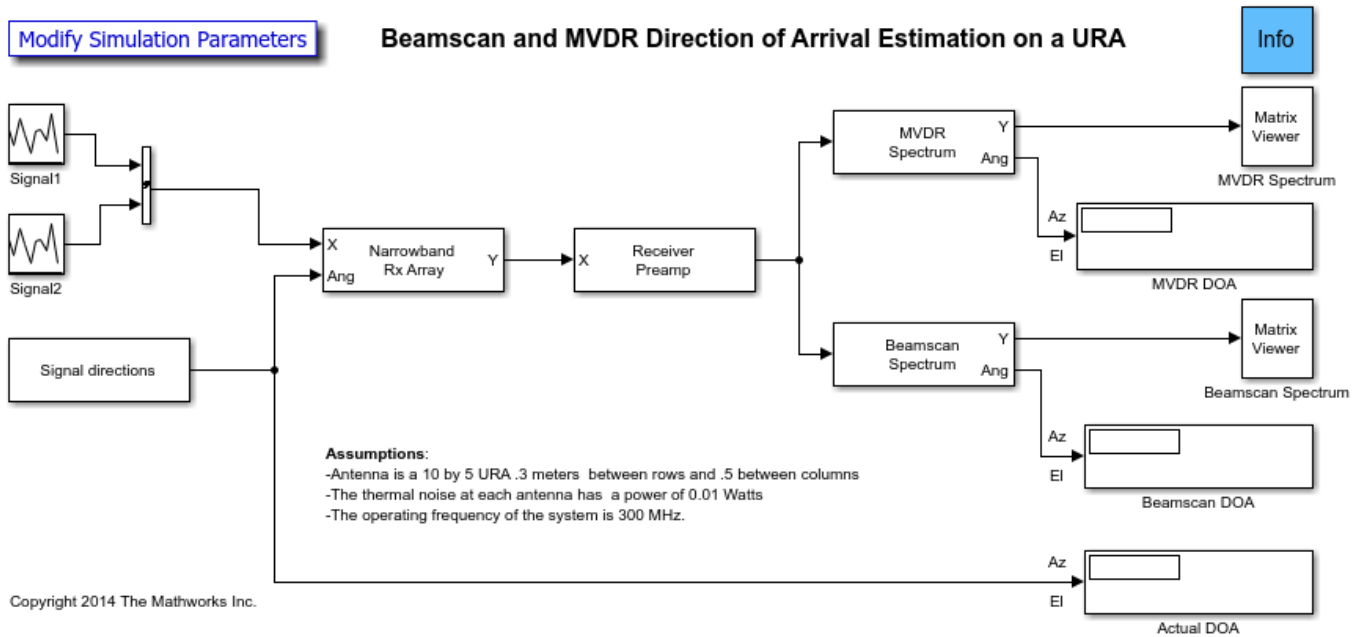
### Exploring the Example

The helper function used for this example is helperslex2DBeamscanMVDRDOAParam. To open the function from the model, click on `Modify Simulation Parameters` block.

### Results and Displays

The results are similar to the previous example. The beamscan spectrum is updated as the sources move towards each other. The spectrum shows two wide peaks with different magnitudes moving in opposite directions.
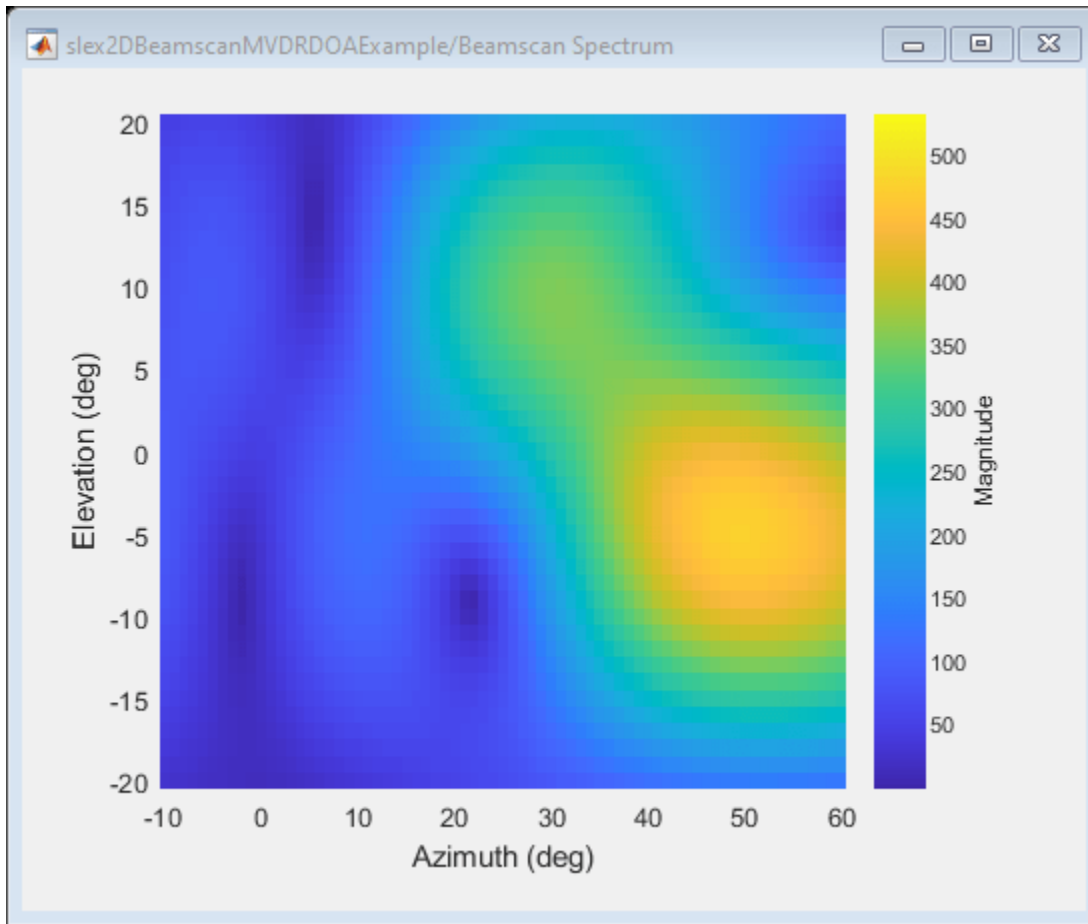
When the sources are approximately 10 degrees apart, the peaks merge and the DOA's of the signals are not clearly distinguished.
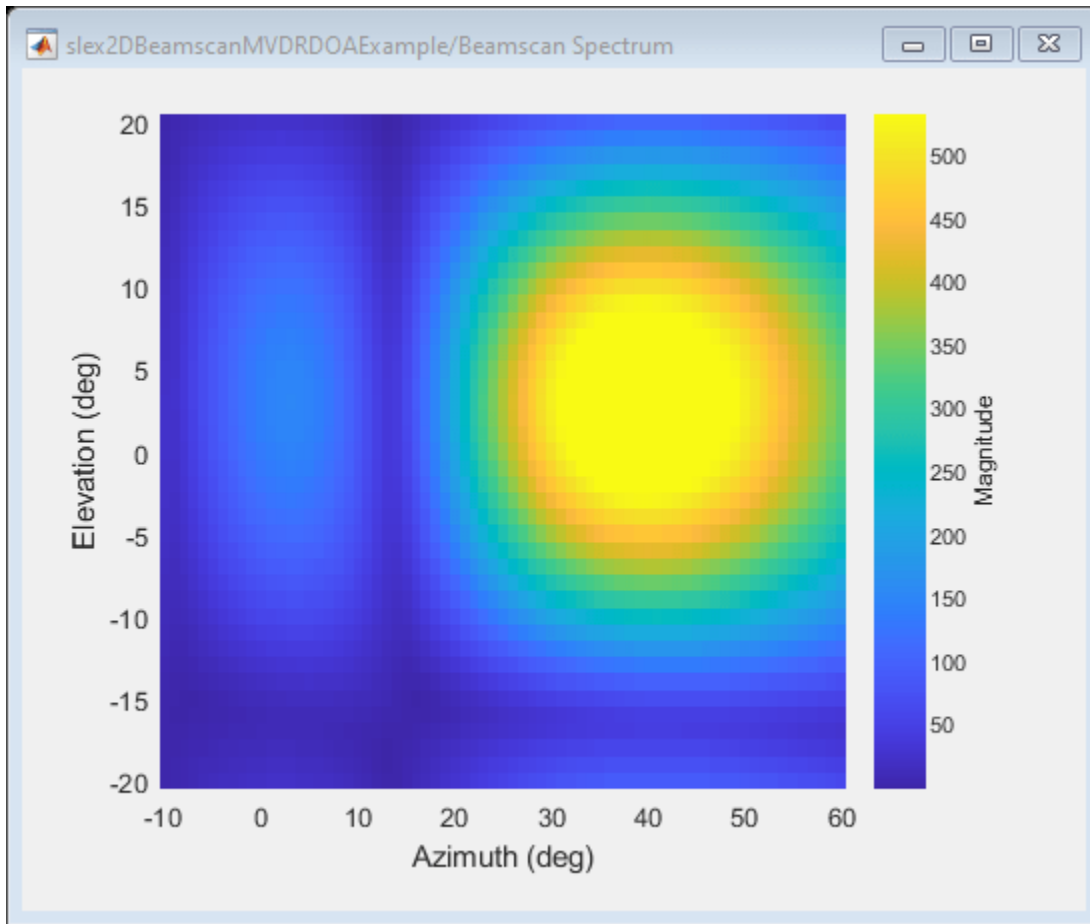
Here the MVDR spectrum can still distinguish both peaks.
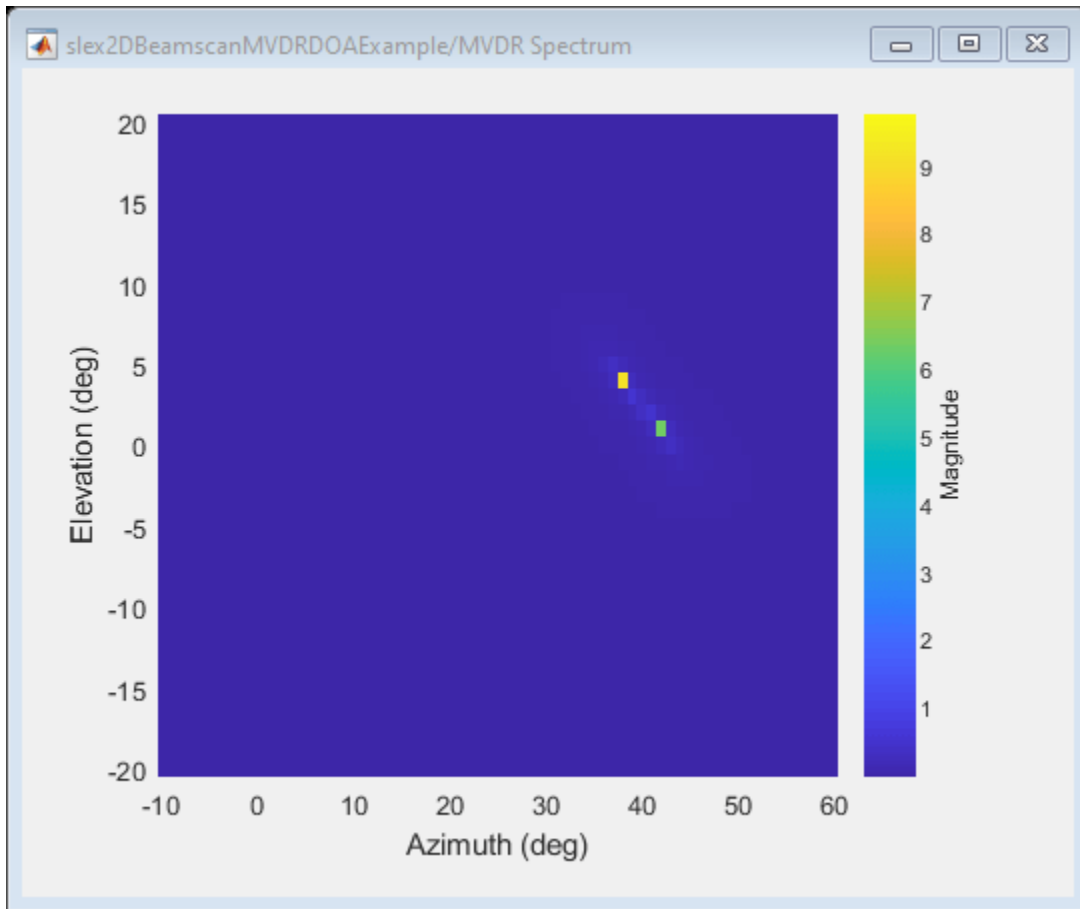
# Constant False Alarm Rate (CFAR) Detection

This example introduces constant false alarm rate (CFAR) detection and shows how to use CFARDetector and CFARDetector2D in the Phased Array System Toolbox™ to perform cell averaging CFAR detection.

### Introduction

One important task a radar system performs is target detection. The detection itself is fairly straightforward. It compares the signal to a threshold. Therefore, the real work on detection is coming up with an appropriate threshold. In general, the threshold is a function of both the probability of detection and the probability of false alarm.

In many phased array systems, because of the cost associated with a false detection, it is desirable to have a detection threshold that not only maximizes the probability of detection but also keeps the probability of false alarm below a preset level.

There is extensive literature on how to determine the detection threshold. Readers might be interested in the "Signal Detection in White Gaussian Noise" on page 18-320 and "Signal Detection Using Multiple Samples" on page 18-327 examples for some well known results. However, all these classical results are based on theoretical probabilities and are limited to white Gaussian noise with known variance (power). In real applications, the noise is often colored and its power is unknown.

CFAR technology addresses these issues. In CFAR, when the detection is needed for a given cell, often termed as the cell under test (CUT), the noise power is estimated from neighboring cells. Then the detection threshold, $T$, is given by

$$T = \alpha P_n$$

where $P_n$ is the noise power estimate and $\alpha$ is a scaling factor called the threshold factor.

From the equation, it is clear that the threshold adapts to the data. It can be shown that with the appropriate threshold factor, $\alpha$, the resulting probability of false alarm can be kept at a constant, hence the name CFAR.

### Cell Averaging CFAR Detection

The cell averaging CFAR detector is probably the most widely used CFAR detector. It is also used as a baseline comparison for other CFAR techniques. In a cell averaging CFAR detector, noise samples are extracted from both leading and lagging cells (called training cells) around the CUT. The noise estimate can be computed as [1]

$$P_n = \frac{1}{N} \sum_{m=1}^{N} x_m$$

where $N$ is the number of training cells and $x_m$ is the sample in each training cell. If $x_m$ happens to be the output of a square law detector, then $P_n$ represents the estimated noise power. In general, the number of leading and lagging training cells are the same. Guard cells are placed adjacent to the CUT, both leading and lagging it. The purpose of these guard cells is to avoid signal components from leaking into the training cell, which could adversely affect the noise estimate.

The following figure shows the relation among these cells for the 1-D case.
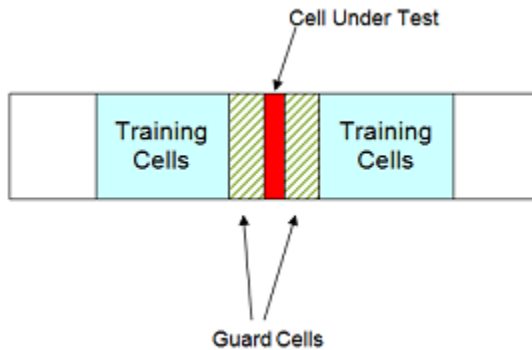
With the above cell averaging CFAR detector, assuming the data passed into the detector is from a single pulse, i.e., no pulse integration involved, the threshold factor can be written as [1]

$$\alpha = N(P_{fa}^{-1/N} - 1)$$

where $P_{fa}$ is the desired false alarm rate.

**CFAR Detection Using Automatic Threshold Factor**

In the rest of this example, we show how to use Phased Array System Toolbox to perform a cell averaging CFAR detection. For simplicity and without losing any generality, we still assume that the noise is white Gaussian. This enables the comparison between the CFAR and classical detection theory.

We can instantiate a CFAR detector using the following command:

```
cfar = phased.CFARDetector('NumTrainingCells',20,'NumGuardCells',2);
```

In this detector we use 20 training cells and 2 guard cells in total. This means that there are 10 training cells and 1 guard cell on each side of the CUT. As mentioned above, if we assume that the signal is from a square law detector with no pulse integration, the threshold can be calculated based on the number of training cells and the desired probability of false alarm. Assuming the desired false alarm rate is 0.001, we can configure the CFAR detector as follows so that this calculation can be carried out.

```
exp_pfa = 1e-3;
cfar.ThresholdFactor = 'Auto';
cfar.ProbabilityFalseAlarm = exp_pfa;
```

The configured CFAR detector is shown below.

```
cfar
```

```
cfar =
  phased.CFARDetector with properties:

                  Method: 'CA'
           NumGuardCells: 2
        NumTrainingCells: 20
          ThresholdFactor: 'Auto'
    ProbabilityFalseAlarm: 1.0000e-03
            OutputFormat: 'CUT result'
```

```
        ThresholdOutputPort: false
      NoisePowerOutputPort: false
```

We now simulate the input data. Since the focus is to show that the CFAR detector can keep the false alarm rate under a certain value, we just simulate the noise samples in those cells. Here are the settings:

- The data sequence is 23 samples long, and the CUT is cell 12. This leaves 10 training cells and 1 guard cell on each side of the CUT.

- The false alarm rate is calculated using 100 thousand Monte Carlo trials.

```
rs = RandStream('mt19937ar','Seed',2010);
npower = db2pow(-10);   % Assume 10dB SNR ratio

Ntrials = 1e5;
Ncells = 23;
CUTIdx = 12;

% Noise samples after a square law detector
rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
x = abs(sqrt(npower/2)*rsamp).^2;
```

To perform the detection, pass the data through the detector. In this example, there is only one CUT, so the output is a logical vector containing the detection result for all the trials. If the result is true, it means that a target is present in the corresponding trial. In our example, all detections are false alarms because we are only passing in noise. The resulting false alarm rate can then be calculated based on the number of false alarms and the number of trials.

```
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 9.4000e-04
```

The result shows that the resulting probability of false alarm is below 0.001, just as we specified.

**CFAR Detection Using Custom Threshold Factor**

As explained in the earlier part of this example, there are only a few cases in which the CFAR detector can automatically compute the appropriate threshold factor. For example, using the previous scenario, if we employ a 10-pulses noncoherent integration before the data goes into the detector, the automatic threshold can no longer provide the desired false alarm rate.

```
npower = db2pow(-10);   % Assume 10dB SNR ratio
xn = 0;
for m = 1:10
    rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
    xn = xn + abs(sqrt(npower/2)*rsamp).^2;   % noncoherent integration
end
x_detected = cfar(xn,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 0
```

One may be puzzled why we think a resulting false alarm rate of 0 is worse than a false alarm rate of 0.001. After all, isn't a false alarm rate of 0 a great thing? The answer to this question lies in the fact that when the probability of false alarm is decreased, so is the probability of detection. In this case,

because the true false alarm rate is far below the allowed value, the detection threshold is set too high. The same probability of detection can be achieved with our desired probability of false alarm at lower cost; for example, with lower transmitter power.

In most cases, the threshold factor needs to be estimated based on the specific environment and system configuration. We can configure the CFAR detector to use a custom threshold factor, as shown below.

```
release(cfar);
cfar.ThresholdFactor = 'Custom';
```

Continuing with the pulse integration example and using empirical data, we found that we can use a custom threshold factor of 2.35 to achieve the desired false alarm rate. Using this threshold, we see that the resulting false alarm rate matches the expected value.

```
cfar.CustomThresholdFactor = 2.35;
x_detected = cfar(xn,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 9.6000e-04
```

**CFAR Detection Threshold**

A CFAR detection occurs when the input signal level in a cell exceeds the threshold level. The threshold level for each cell depends on the threshold factor and the noise power in that derived from training cells. To maintain a constant false alarm rate, the detection threshold will increase or decrease in proportion to the noise power in the training cells. Configure the CFAR detector to output the threshold used for each detection using the `ThresholdOutputPort` property. Use an automatic threshold factor and 200 training cells.

```
release(cfar);
cfar.ThresholdOutputPort = true;
cfar.ThresholdFactor = 'Auto';
cfar.NumTrainingCells = 200;
```

Next, create a square-law input signal with increasing noise power.

```
rs = RandStream('mt19937ar','Seed',2010);
Npoints = 1e4;
rsamp = randn(rs,Npoints,1)+1i*randn(rs,Npoints,1);
ramp = linspace(1,10,Npoints)';
xRamp = abs(sqrt(npower*ramp./2).*rsamp).^2;
```

Compute detections and thresholds for all cells in the signal.

```
[x_detected,th] = cfar(xRamp,1:length(xRamp));
```

Next, compare the CFAR threshold to the input signal.

```
plot(1:length(xRamp),xRamp,1:length(xRamp),th,...
  find(x_detected),xRamp(x_detected),'o')
legend('Signal','Threshold','Detections','Location','Northwest')
xlabel('Time Index')
ylabel('Level')
```
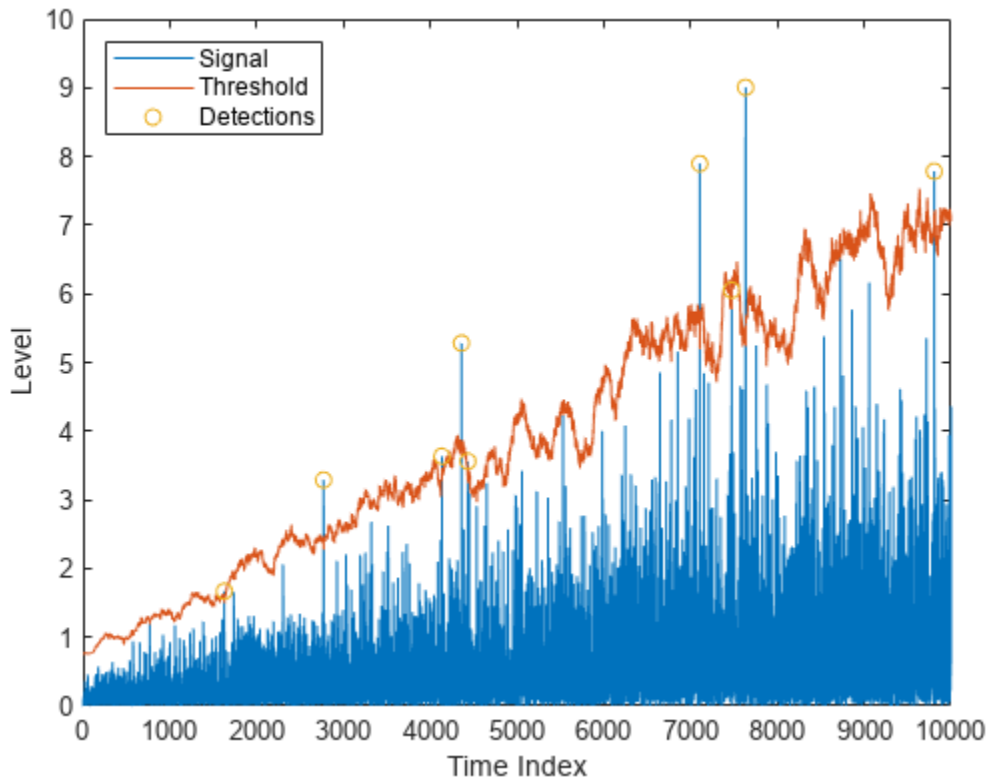
Here, the threshold increases with the noise power of the signal to maintain the constant false alarm rate. Detections occur where the signal level exceeds the threshold.

**Comparison Between CFAR and Classical Neyman-Pearson Detector**

In this section, we compare the performance of a CFAR detector with the classical detection theory using the Neyman-Pearson principle. Returning to the first example and assuming the true noise power is known, the theoretical threshold can be calculated as

```
T_ideal = npower*db2pow(npwgnthresh(exp_pfa));
```

The false alarm rate of this classical Neyman-Pearson detector can be calculated using this theoretical threshold.

```
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

act_Pfa_np = 9.5000e-04

Because we know the noise power, classical detection theory also produces the desired false alarm rate. The false alarm rate achieved by the CFAR detector is similar.

```
release(cfar);
cfar.ThresholdOutputPort = false;
cfar.NumTrainingCells = 20;
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

act_pfa = 9.4000e-04

Next, assume that both detectors are deployed to the field and that the noise power is 1 dB more than expected. In this case, if we use the theoretical threshold, the resulting probability of false alarm is four times more than what we desire.

```
npower = db2pow(-9);   % Assume 9dB SNR ratio
rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
x = abs(sqrt(npower/2)*rsamp).^2;
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

act_Pfa_np = 0.0041

On the contrary, the CFAR detector's performance is not affected.

```
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

act_pfa = 0.0011

Hence, the CFAR detector is robust to noise power uncertainty and better suited to field applications.

Finally, use a CFAR detection in the presence of colored noise. We first apply the classical detection threshold to the data.

```
npower = db2pow(-10);
fcoeff = maxflat(10,'sym',0.2);
x = abs(sqrt(npower/2)*filter(fcoeff,1,rsamp)).^2;   % colored noise
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

act_Pfa_np = 0

Note that the resulting false alarm rate cannot meet the requirement. However, using the CFAR detector with a custom threshold factor, we can obtain the desired false alarm rate.

```
release(cfar);
cfar.ThresholdFactor = 'Custom';
cfar.CustomThresholdFactor = 12.85;
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

act_pfa = 0.0010

**CFAR Detection for Range-Doppler Images**

In the previous sections, the noise estimate was computed from training cells leading and lagging the CUT in a single dimension. We can also perform CFAR detection on images. Cells correspond to pixels in the images, and guard cells and training cells are placed in bands around the CUT. The detection threshold is computed from cells in the rectangular training band around the CUT.
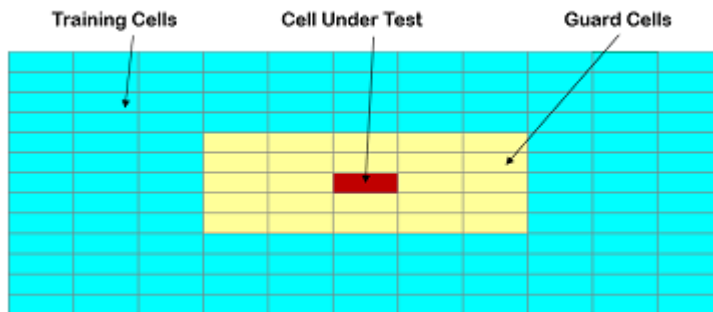
In the figure above, the guard band size is [2 2] and the training band size is [4 3]. The size indices refer to the number of cells on each side of the CUT in the row and columns dimensions, respectively. The guard band size can also be defined as 2, since the size is the same along row and column dimensions.

Next, create a two-dimensional CFAR detector. Use a probability of false alarm of 1e-5 and specify a guard band size of 5 cells and a training band size of 10 cells.

```
cfar2D = phased.CFARDetector2D('GuardBandSize',5,'TrainingBandSize',10,...
    'ProbabilityFalseAlarm',1e-5);
```

Next, load and plot a range-doppler image. The image includes returns from two stationary targets and one target moving away from the radar.

```
[resp,rngGrid,dopGrid] = helperRangeDoppler;
```
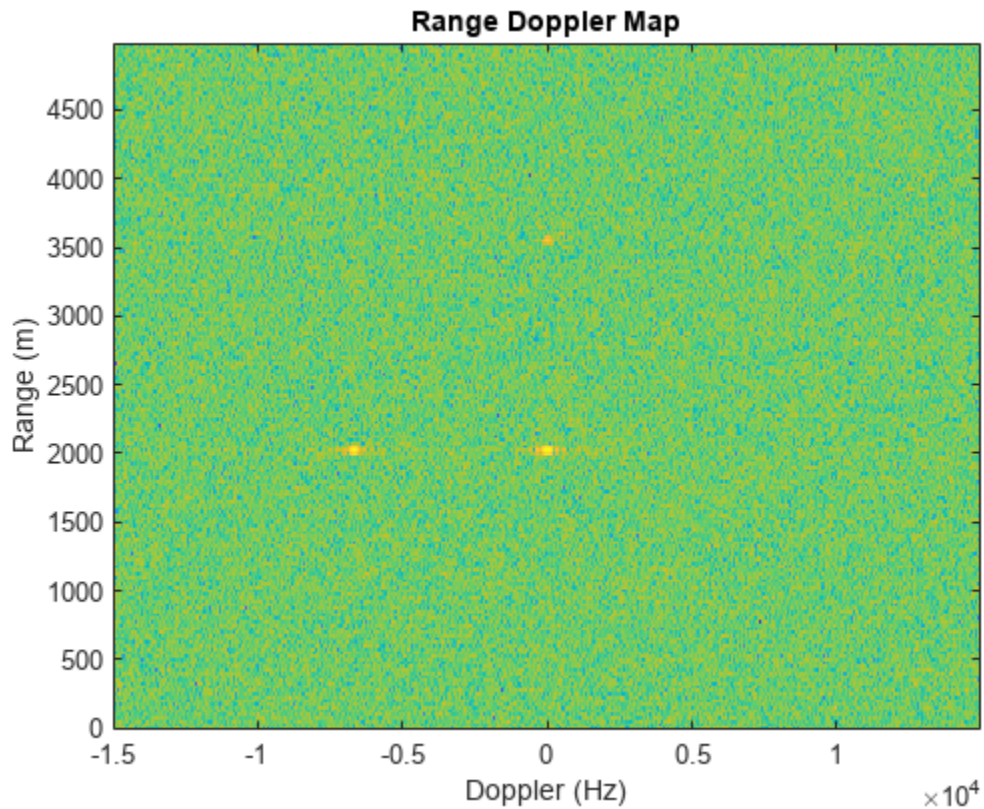
**Range Doppler Map**



Use CFAR to search the range-Doppler space for objects, and plot a map of the detections. Search from -10 to 10 kHz and from 1000 to 4000 m. First, define the cells under test for this region.

```
[~,rangeIndx] = min(abs(rngGrid-[1000 4000]));
[~,dopplerIndx] = min(abs(dopGrid-[-1e4 1e4]));
[columnInds,rowInds] = meshgrid(dopplerIndx(1):dopplerIndx(2),...
  rangeIndx(1):rangeIndx(2));
CUTIdx = [rowInds(:) columnInds(:)]';
```

Compute a detection result for each cell under test. Each pixel in the search region is a cell in this example. Plot a map of the detection results for the range-Doppler image.

```
detections = cfar2D(resp,CUTIdx);
helperDetectionsMap(resp,rngGrid,dopGrid,rangeIndx,dopplerIndx,detections)
```
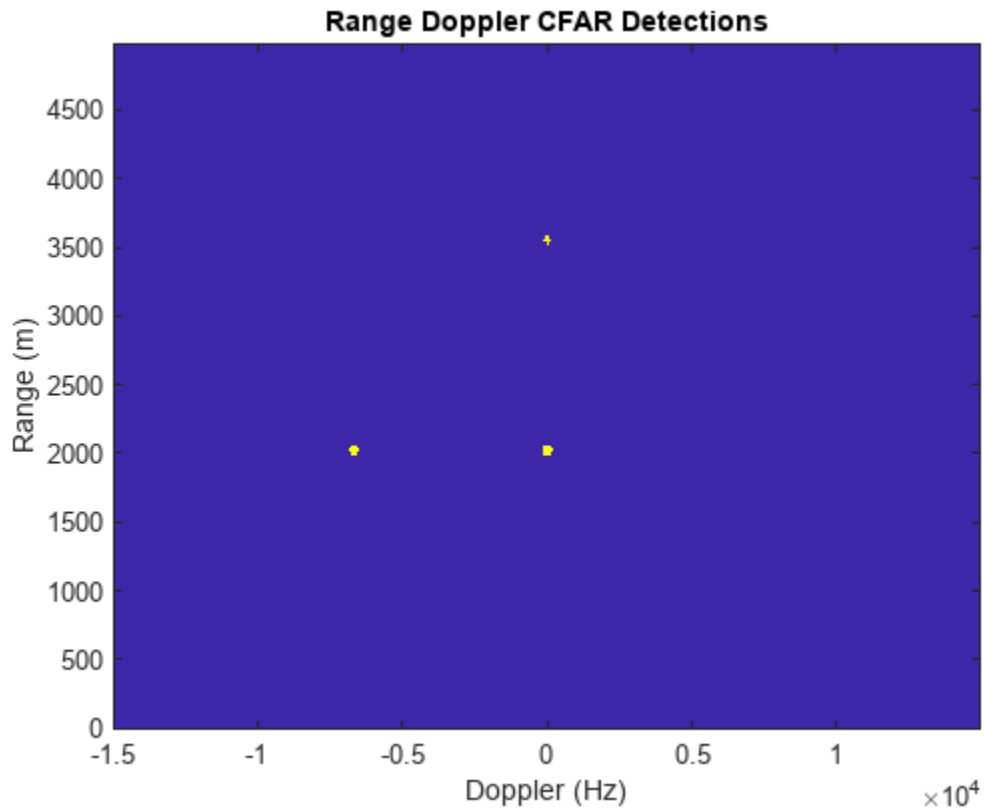
The three objects are detected. A data cube of range-Doppler images over time can likewise be provided as the input signal to `cfar2D`, and detections will be calculated in a single step.

**Summary**

In this example, we presented the basic concepts behind CFAR detectors. In particular, we explored how to use the Phased Array System Toolbox to perform cell averaging CFAR detection on signals and range-Doppler images. The comparison between the performance offered by a cell averaging CFAR detector and a detector equipped with the theoretically calculated threshold shows clearly that the CFAR detector is more suitable for real field applications.

**Reference**

[1] Mark Richards, *Fundamentals of Radar Signal Processing*, McGraw Hill, 2005

# Detector Performance Analysis Using ROC Curves

This example shows how you can assess the performance of both coherent and noncoherent systems using receiver operating characteristic (ROC) curves. The example assumes that the detector operates in an additive complex white Gaussian noise environment.

ROC curves are often used to assess the performance of a radar or sonar detector. ROC curves are plots of the probability of detection (Pd) versus the probability of false alarm (Pfa) for a given signal-to-noise ratio (SNR).

**Introduction**

Pd is the probability of saying that 1 is true given that event 1 occurred. Pfa is the probability of saying that 1 is true given that the 0 event occurred. In applications such as sonar and radar, the 1 event indicates that a target is present, and the 0 event indicates that a target is not present.

The performance of a detector is measured by its ability to achieve a certain Pd and Pfa for a given SNR. Examining the ROC curves provides insight into its performance. You can use the `rocsnr` function to calculate and plot the ROC curves.

**Single Pulse Detection**

Given an SNR value, you can calculate the Pd and Pfa values that a linear or square-law detector can achieve using a single pulse. Set the Pd value assuming that you have an SNR value of 8 dB and the requirements dictate a Pfa value of at most 1%. You can use the `rocsnr` function to calculate the Pd and Pfa values and then determine what value of Pd corresponds to Pfa = 0.01. Note that by default the `rocsnr` function assumes coherent detection.

```
[Pd,Pfa] = rocsnr(8);
idx = find(Pfa==0.01); % find index for Pfa=0.01
```

Using the index determined above you can find the Pd value that corresponds to Pfa = 0.01.

```
Pd(idx)
```

```
ans = 0.8899
```

One feature of the `rocsnr` function is that you can specify a vector of SNR values and the function calculates the ROC curve for each of these SNR values. Instead of individually calculating Pd and Pfa values for a given SNR, your can view the results in a plot of ROC curves. The `rocsnr` function plots the ROC curves by default if no output arguments are specified. Calling the `rocsnr` function with an input vector of four SNR values and no output arguments produces a plot of the ROC curves.

```
SNRvals = [2 4 8 9.4];
rocsnr(SNRvals);
```

Nonfluctuating Coherent Receiver Operating Characteristic (ROC) Curves
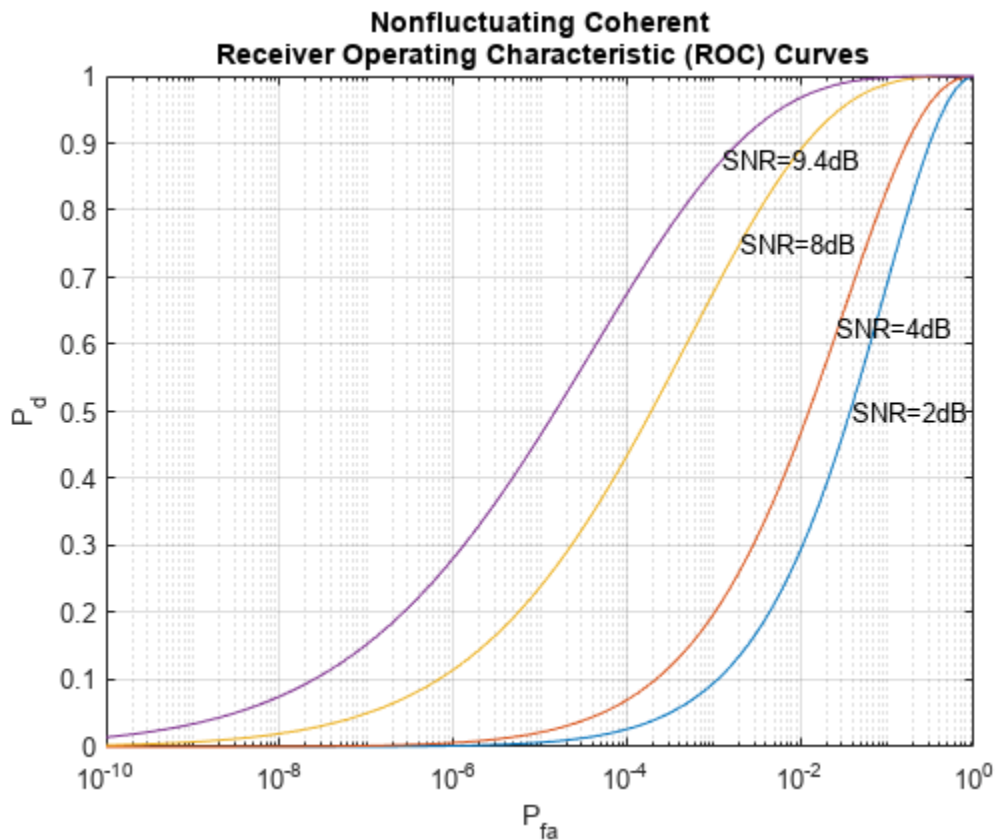
In the plot, select the data cursor button in the toolbar (or in the Tools menu) and then select the SNR = 8 dB curve at the point where Pd = 0.9 to verify that Pfa is approximately 0.01.

**Multiple Pulse Detection**

One way to improve detector performance is to average over several pulses. This is particularly useful in cases where the signal of interest is known and occurs in additive complex white noise. Although this still applies to both linear and square-law detectors, the result for square-law detectors could be off by about 0.2 dB. Analyze the performance by assuming an SNR of 8 dB and averaging over two pulses.

```
rocsnr(8,'NumPulses',2);
```
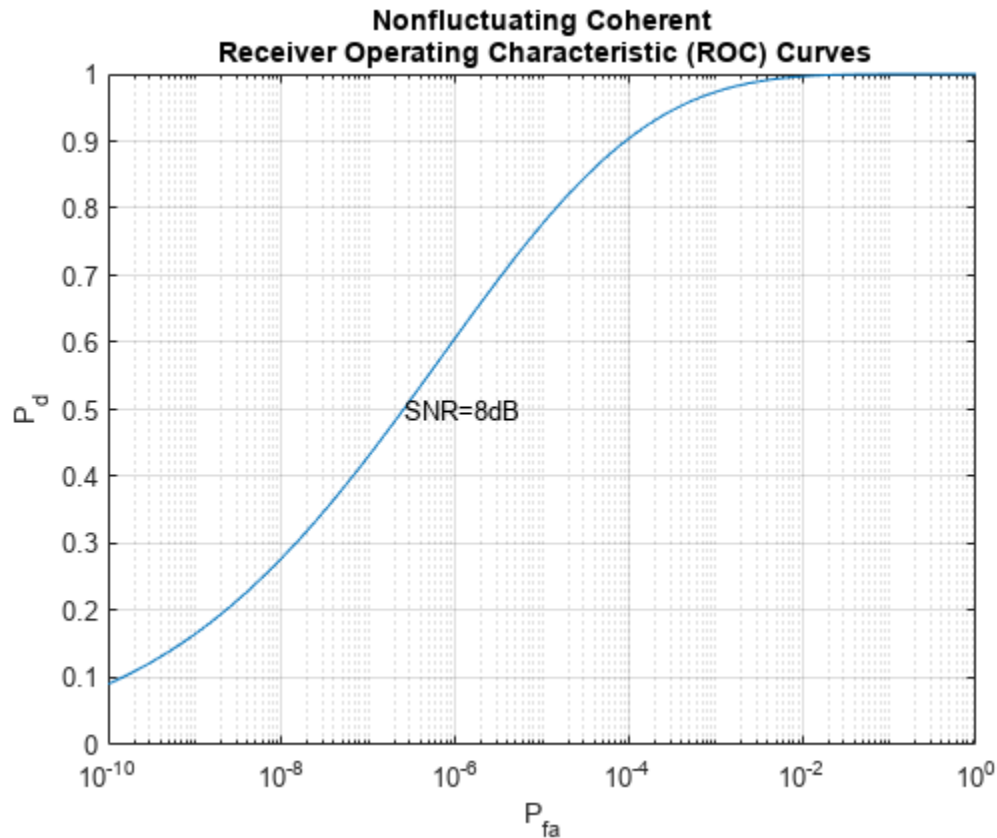
By inspecting the plot you can see that averaging over two pulses resulted in a higher probability of detection for a given false alarm rate. With an SNR of 8 dB and averaging over two pulses, you can constrain the probability of false alarm to be at most 0.0001 and achieve a probability of detection of 0.9. Recall that for a single pulse, you had to allow the probability of false alarm to be as much as 1% to achieve the same probability of detection.

**Noncoherent Detector**

To this point, the example deals with a known signal in complex white Gaussian noise. The `rocsnr` function by default assumes a coherent detector. To analyze the performance of a detector for the case where the signal is known except for the phase, you can specify a noncoherent detector. Using the same SNR values as before, analyze the performance of a noncoherent detector.

```
rocsnr(SNRvals,'SignalType','NonfluctuatingNoncoherent');
```

**Nonfluctuating Noncoherent Receiver Operating Characteristic (ROC) Curves**
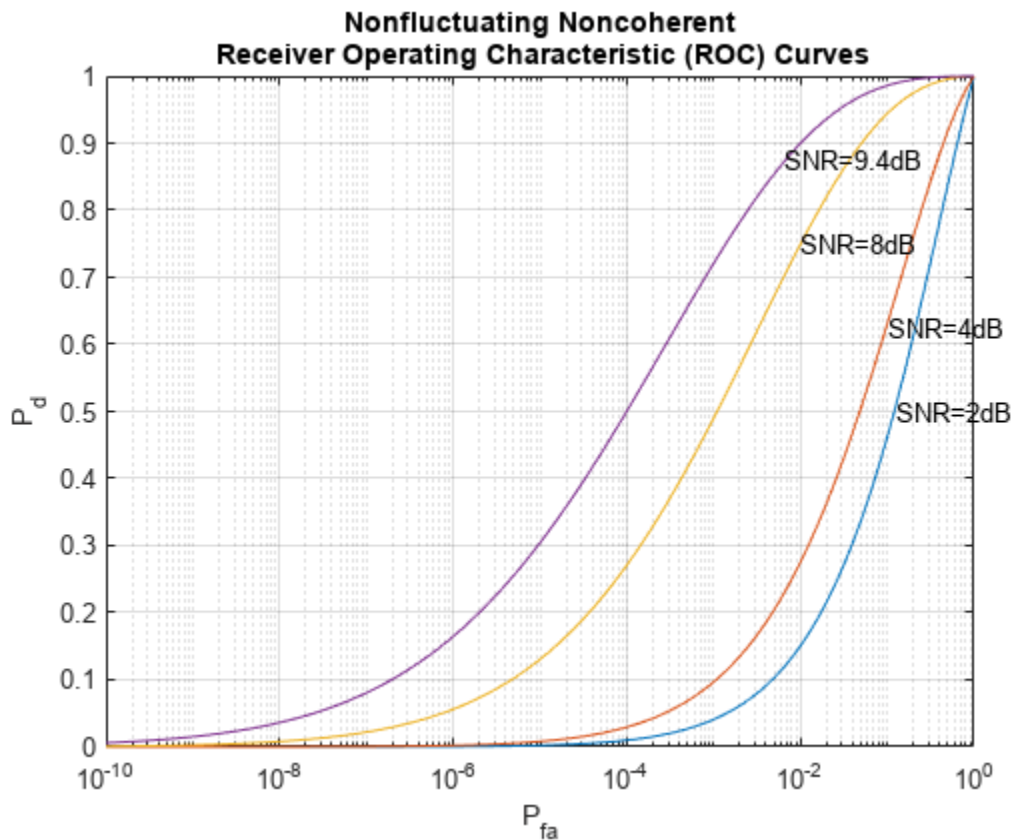
Focus on the ROC curve corresponding to an SNR of 8 dB. By inspecting the graph with the data cursor, you can see that to achieve a probability of detection of 0.9, you must tolerate a false-alarm probability of up to 0.05. Without using phase information, you will need a higher SNR to achieve the same Pd for a given Pfa. For noncoherent linear detectors, use Albersheim's equation to determine what value of SNR will achieve the desired Pd and Pfa.

```
SNR_valdB = albersheim(0.9,.01)   % Pd=0.9 and Pfa=0.01
```

SNR_valdB = 9.5027

Plotting the ROC curve for the SNR value approximated by Albersheim's equation, you can see that the detector will achieve Pd = 0.9 and Pfa = 0.01. Note that the Albersheim's technique applies only to noncoherent detectors.

```
rocsnr(SNR_valdB,'SignalType','NonfluctuatingNoncoherent');
```
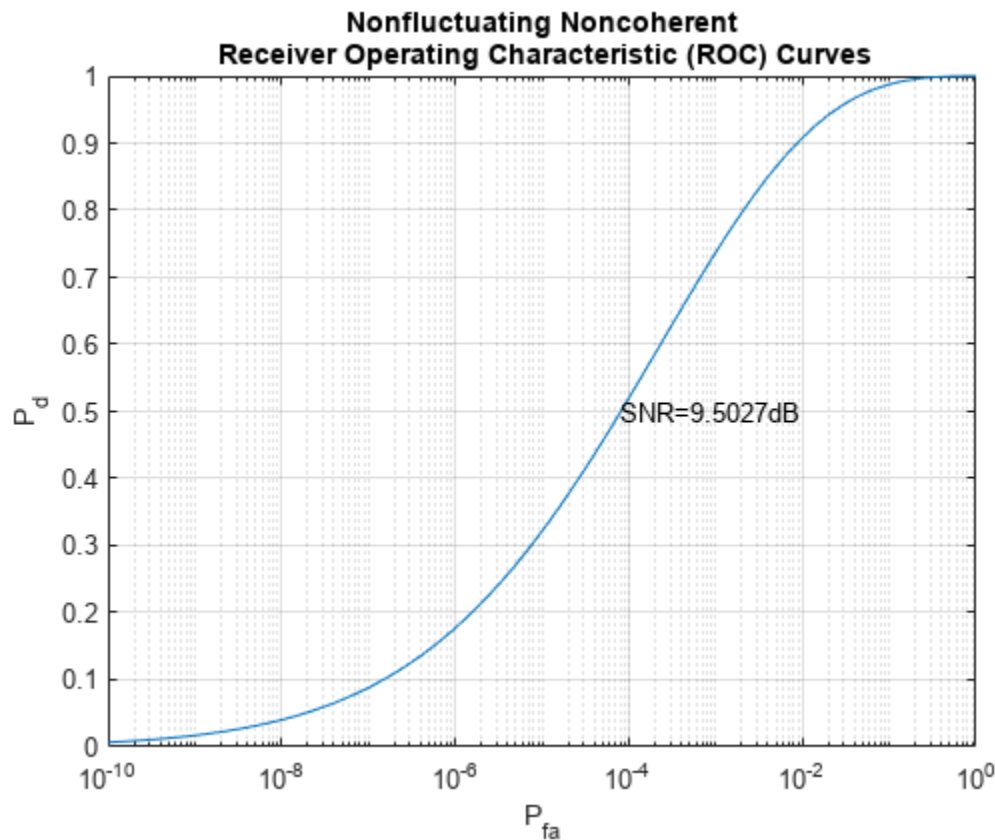
**Detection of Fluctuating Targets**

All the discussions above assume that the target is nonfluctuating, which means that the statistical characteristics of the target do not change over time. However, in real scenarios, targets can accelerate and decelerate as well as roll and pitch. These factors cause the radar cross section (RCS) of the target to vary over time. A set of statistical models called Swerling models are often used to describe the random variation in target RCS.

There are four Swerling models, namely Swerling 1 -- 4. The nonfluctuating target is often termed either Swerling 0 or Swerling 5. Each Swerling model describes how the RCS of a target varies over time and the probability distribution of the variation.

Because the target RCS is varying, the ROC curves for fluctuating targets are not the same as the nonfluctuating ones. In addition, because Swerling targets add random phase into the received signal, it is harder to use a coherent detector for a Swerling target. Therefore, noncoherent detection techniques are often used for Swerling targets.

Now compare the ROC curves for a nonfluctuating target and a Swerling 1 target. In particular, you want to explore what the SNR requirements are for both situations if you want to achieve the same Pd and Pfa. For such a comparison, it is often easy to plot the ROC curve as Pd against SNR with varying Pfa. We can use the `rocpfa` function to plot the ROC curve in this form.

Assume noncoherent detection with 10 integrated pulses, with the desired Pfa being at most 1e-8. First, plot the ROC curve for a nonfluctuating target.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','NonfluctuatingNoncoherent')
```
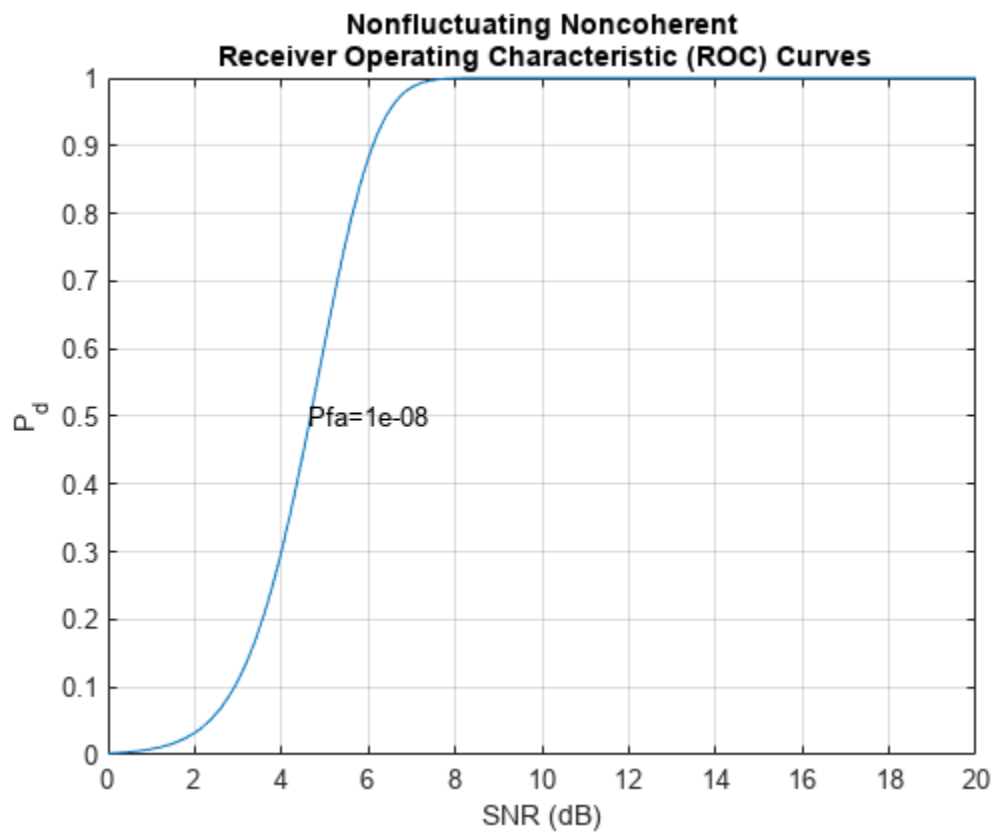
Then plot the ROC curve for a Swerling 1 target for comparison.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','Swerling1')
```
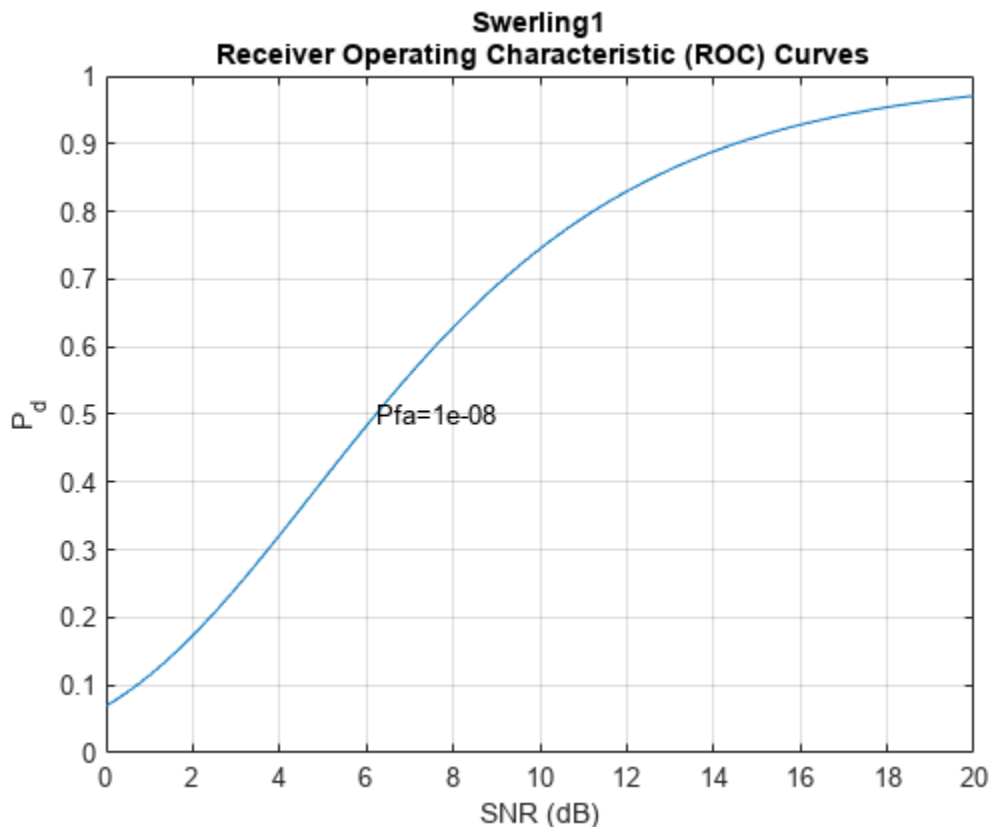
You can see from the figures, we can see that for a Pd of 0.9, you need an SNR of about 6 dB if the target is nonfluctuating. However, if the target is a Swerling case 1 model, the required SNR jumps to more than 14 dB, an 8 dB difference. This will greatly affect the design of the system.

As in the case of nonfluctuating targets, you have approximation equations to help determine the required SNR without having to plot all the curves. The equation used for fluctuating targets is Shnidman's equation. For the scenario used to plot the ROC curves, the SNR requirements can be derived using the `shnidman` function.

```
snr_sw1_db = shnidman(0.9,1e-8,10,1)   % Pd=0.9, Pfa=1e-8, 10 pulses,

snr_sw1_db = 14.7131

                                 % Swerling case 1
```

The calculated SNR requirement matches the value derived from the curve.

**Summary**

ROC curves are useful for analyzing detector performance, both for coherent and noncoherent systems. This example used the `rocsnr` function to analyze the effectiveness of a linear detector for various SNR values. It also reviewed the improvement in detector performance achieved by averaging multiple samples. Lastly, the example showed how you can use the `rocsnr` and `rocpfa` functions to analyze detector performance when using a noncoherent detector for both nonfluctuating and fluctuating targets.

# Doppler Estimation

This example shows a monostatic pulse radar detecting the radial velocity of moving targets at specific ranges. The speed is derived from the Doppler shift caused by the moving targets. We first identify the existence of a target at a given range and then use Doppler processing to determine the radial velocity of the target at that range.

**Radar System Setup**

First, we define a radar system. Since the focus of this example is on Doppler processing, we use the radar system built in the example "Simulating Test Signals for a Radar Receiver" on page 18-393. Readers are encouraged to explore the details of radar system design through that example.

```
load BasicMonostaticRadarExampleData;
```

**System Simulation**

**Targets**

Doppler processing exploits the Doppler shift caused by the moving target. We now define three targets by specifying their positions, radar cross sections (RCS), and velocities.

```
tgtpos = [[1200; 1600; 0],[3543.63; 0; 0],[1600; 0; 1200]];
tgtvel = [[60; 80; 0],[0;0;0],[0; 100; 0]];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [1.3 1.7 2.1];
fc = radiator.OperatingFrequency;
target = phased.RadarTarget('MeanRCS',tgtrcs,'OperatingFrequency',fc);
```

Note that the first and third targets are both located at a range of 2000 m and are both traveling at a speed of 100 m/s. The difference is that the first target is moving along the radial direction, while the third target is moving in the tangential direction. The second target is not moving.

**Environment**

We also need to setup the propagation environment for each target. Since we are using a monostatic radar, we use the two way propagation model.

```
fs = waveform.SampleRate;

channel = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

**Signal Synthesis**

With the radar system, the environment, and the targets defined, we can now simulate the received signal as echoes reflected from the targets. The simulated data is a data matrix with the fast time (time within each pulse) along each column and the slow time (time between pulses) along each row.

We need to set the seed for noise generation at the receiver so that we can reproduce the same results.

```
receiver.SeedSource = 'Property';
receiver.Seed = 2009;

prf = waveform.PRF;
num_pulse_int = 10;

fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
slow_time_grid = (0:num_pulse_int-1)/prf;

% Pre-allocate array for improved processing speed
rxpulses = zeros(numel(fast_time_grid),num_pulse_int);

for m = 1:num_pulse_int

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);

    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    txsig = radiator(txsig,tgtang);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
    tgtsig = target(txsig);

    % Receive target returns at sensor
    rxsig = collector(tgtsig,tgtang);
    rxpulses(:,m) = receiver(rxsig,~(txstatus>0));
end
```

**Doppler Estimation**

**Range Detection**

To be able to estimate the Doppler shift of the targets, we first need to locate the targets through range detection. Because the Doppler shift spreads the signal power into both I and Q channels, we need to rely on the signal energy to do the detection. This means that we use noncoherent detection schemes.

The detection process is described in detail in the aforementioned example so we simply perform the necessary steps here to estimate the target ranges.

```
% calculate initial threshold
pfa = 1e-6;
% in loaded system, noise bandwidth is half of the sample rate
noise_bw = receiver.SampleRate/2;
npower = noisepow(noise_bw,...
    receiver.NoiseFigure,receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(pfa,num_pulse_int,'noncoherent'));

% apply matched filter and update the threshold
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter(...
```

```
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);
[rxpulses, mfgain] = matchedfilter(rxpulses);
threshold = threshold * db2pow(mfgain);

% compensate the matched filter delay
matchingdelay = size(matchingcoeff,1)-1;
rxpulses = buffer(rxpulses(matchingdelay+1:end),size(rxpulses,1));

% apply time varying gain to compensate the range dependent loss
prop_speed = radiator.PropagationSpeed;
range_gates = prop_speed*fast_time_grid/2;
lambda = prop_speed/fc;

tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(range_gates,lambda),...
    'ReferenceLoss',2*fspl(prop_speed/(prf*2),lambda));

rxpulses = tvg(rxpulses);

% detect peaks from the integrated pulse
[~,range_detect] = findpeaks(pulsint(rxpulses,'noncoherent'),...
    'MinPeakHeight',sqrt(threshold));
range_estimates = round(range_gates(range_detect))
```

range_estimates = *1×2*

        2000        3550

These estimates suggest the presence of targets in the range of 2000 m and 3550 m.

**Doppler Spectrum**

Once we successfully estimated the ranges of the targets, we can then estimate the Doppler information for each target.

Doppler estimation is essentially a spectrum estimation process. Therefore, the first step in Doppler processing is to generate the Doppler spectrum from the received signal.

The received signal after the matched filter is a matrix whose columns correspond to received pulses. Unlike range estimation, Doppler processing processes the data across the pulses (slow time), which is along the rows of the data matrix. Since we are using 10 pulses, there are 10 samples available for Doppler processing. Because there is one sample from each pulse, the sampling frequency for the Doppler samples is the pulse repetition frequency (PRF).

As predicted by the Fourier theory, the maximum unambiguous Doppler shift a pulse radar system can detect is half of its PRF. This also translates to the maximum unambiguous speed a radar system can detect. In addition, the number of pulses determines the resolution in the Doppler spectrum, which determines the resolution of the speed estimates.

```
max_speed = dop2speed(prf/2,lambda)/2
```

max_speed = 224.6888

```
speed_res = 2*max_speed/num_pulse_int
```

speed_res = 44.9378

As shown in the calculation above, in this example, the maximum detectable speed is 225m/s, either approaching (-225) or departing (+225). The resulting Doppler resolution is about 45 m/s, which means that the two speeds must be at least 45 m/s apart to be separable in the Doppler spectrum. To improve the ability to discriminate between different target speeds, more pulses are needed. However, the number of pulses available is also limited by the radial velocity of the target. Since the Doppler processing is limited to a given range, all pulses used in the processing have to be collected before the target moves from one range bin to the next.

Because the number of Doppler samples are in general limited, it is common to zero pad the sequence to interpolate the resulting spectrum. This will not improve the resolution of the resulting spectrum, but can improve the estimation of the locations of the peaks in the spectrum.

The Doppler spectrum can be generated using a periodogram. We zero pad the slow time sequence to 256 points.

```
num_range_detected = numel(range_estimates);
[p1, f1] = periodogram(rxpulses(range_detect(1),:).',[],256,prf, ...
                'power','centered');
[p2, f2] = periodogram(rxpulses(range_detect(2),:).',[],256,prf, ...
                'power','centered');
```

The speed corresponding to each sample in the spectrum can then be calculated. Note that we need to take into consideration of the round trip effect.

```
speed_vec = dop2speed(f1,lambda)/2;
```

**Doppler Estimation**

To estimate the Doppler shift associated with each target, we need to find the locations of the peaks in each Doppler spectrum. In this example, the targets are present at two different ranges, so the estimation process needs to be repeated for each range.

Let's first plot the Doppler spectrum corresponding to the range of 2000 meters.

```
periodogram(rxpulses(range_detect(1),:).',[],256,prf,'power','centered');
```
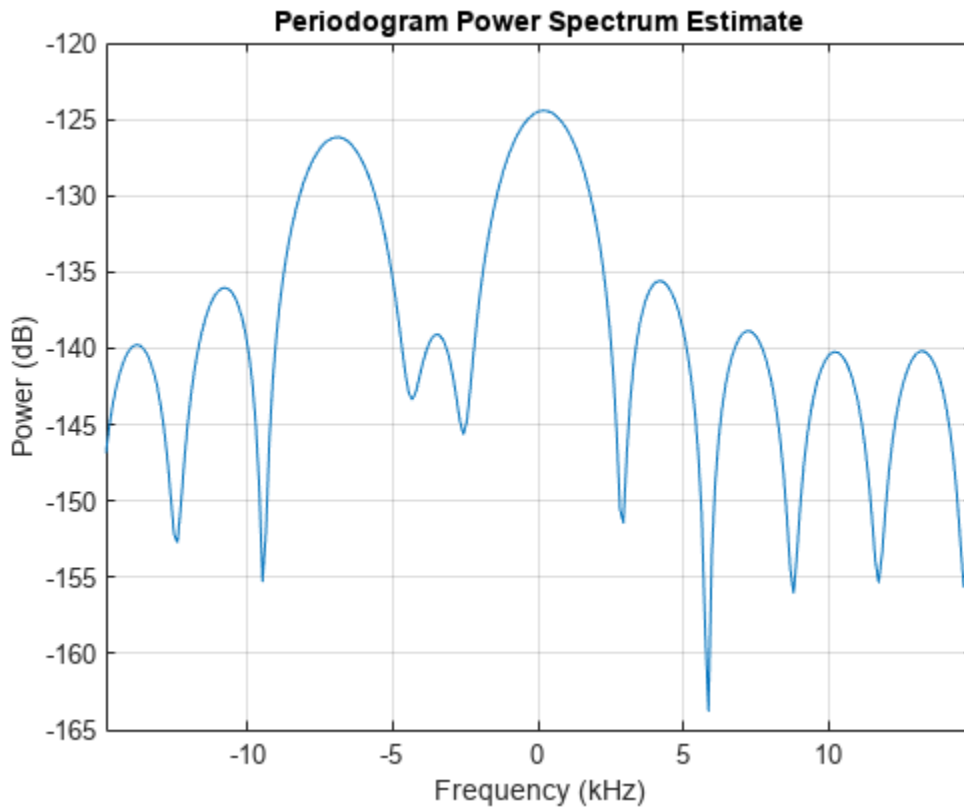
Note that we are only interested in detecting the peaks, so the spectrum values themselves are not critical. From the plot of Doppler spectrum, we notice that 5 dB below the maximum peak is a good threshold. Therefore, we use -5 as our threshold on the normalized Doppler spectrum.

```
spectrum_data = p1/max(p1);
[~,dop_detect1] = findpeaks(pow2db(spectrum_data),'MinPeakHeight',-5);
sp1 = speed_vec(dop_detect1)
```

sp1 = *2×1*

```
 -103.5675
    3.5108
```

The results show that there are two targets at the 2000 m range: one with a velocity of 3.5 m/s and the other with -104 m/s. The value -104 m/s can be easily associated with the first target, since the first target is departing at a radial velocity of 100 m/s, which, given the Doppler resolution of this example, is very close to the estimated value. The value 3.5 m/s requires more explanation. Since the third target is moving along the tangential direction, there is no velocity component in the radial direction. Therefore, the radar cannot detect the Doppler shift of the third target. The true radial velocity of the third target, hence, is 0 m/s and the estimate of 3.5 m/s is very close to the true value.

Note that these two targets cannot be discerned using only range estimation because their range values are the same.

The same operations are then applied to the data corresponding to the range of 3550 meters.

```
periodogram(rxpulses(range_detect(2),:).',[],256,prf,'power','centered');
```



**Periodogram Power Spectrum Estimate**
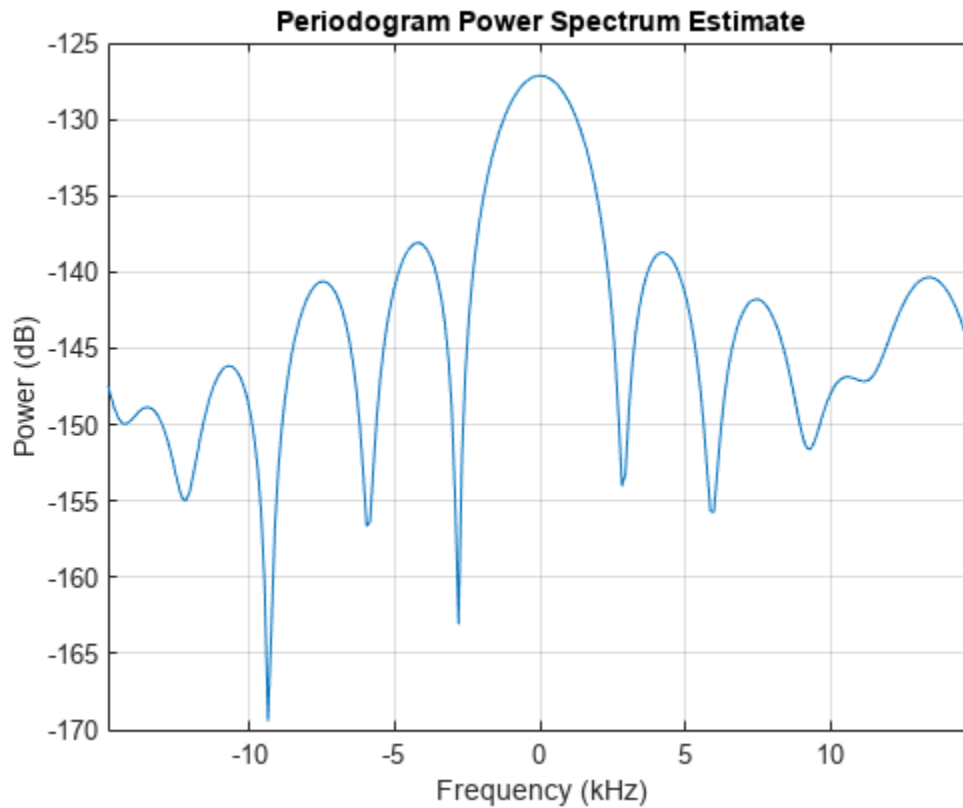
```
spectrum_data = p2/max(p2);
[~,dop_detect2] = findpeaks(pow2db(spectrum_data),'MinPeakHeight',-5);
sp2 = speed_vec(dop_detect2)
```

```
sp2 = 0
```

This result shows an estimated speed of 0 m/s, which matches the fact that the target at this range is not moving.

**Summary**

This example showed a simple way to estimate the radial speed of moving targets using a pulse radar system. We generated the Doppler spectrum from the received signal and estimated the peak locations from the spectrum. These peak locations correspond to the target's radial speed. The limitations of the Doppler processing are also discussed in the example.

# Range Estimation Using Stretch Processing

This example shows how to estimate the range of a target using stretch processing in a radar system that uses a linear FM pulse waveform.

**Introduction**

Linear FM waveform is a popular choice in modern radar systems because it can achieve high range resolution by sweeping through a wide bandwidth. However, when the bandwidth is on the order of hundreds of megahertz, or even gigahertz, it becomes difficult to perform matched filtering or pulse compression in the digital domain because high-quality A/D converters are hard to find at such data rates.

Stretch processing, sometimes also referred to as *deramp*, is a technique that can be used in such situations. Stretch processing is performed in the analog domain.

The received signal is first mixed with a replica of the transmitted pulse. Note that the replica matches the return from the reference range. Once mixed, the resulting signal contains a frequency component that corresponds to the range offset measured from this reference range. Hence, the exact range can be estimated by performing a spectral analysis on the signal at the output of the mixer.

In addition, instead of processing the entire range span covered by the pulse, the processing focuses on a small window around a predefined reference range. Because of the limited range span, the output data of the stretch processor can be sampled at a lower rate, relaxing the bandwidth requirement for A/D converters

The following sections show an example of range estimation using stretch processing.

**Simulation Setup**

The radar system in this example uses a linear FM waveform with a 3 MHz sweeping bandwidth. The waveform can be used to achieve a range resolution of 50 m and a maximum unambiguous range of 8 km. The sample rate is set to 6 MHz, i.e., twice the sweeping bandwidth. For more information about the radar system, see "Waveform Design to Improve Range Performance of an Existing System" on page 18-178.

Three targets are located at 2000.66, 6532.63, and 6845.04 meters from the radar, respectively. Ten pulses are simulated at the receiver. These pulses contain echoes from the targets.

```
[rx_pulses, waveform] = helperStretchSimulate;
fs = waveform.SampleRate;
```

A time frequency plot of the received pulse is shown below. A coherent pulse integration is done before the plot to improve the signal-to-noise ratio (SNR). In the figure, the return from the first target can be clearly seen between 14 and 21 ms while the return from the second and third targets are much weaker, appearing after 45 ms.

```
helperStretchSignalSpectrogram(pulsint(rx_pulses,'coherent'),fs,...
    8,4,'Received Signal');
```

Spectrogram for Received Signal
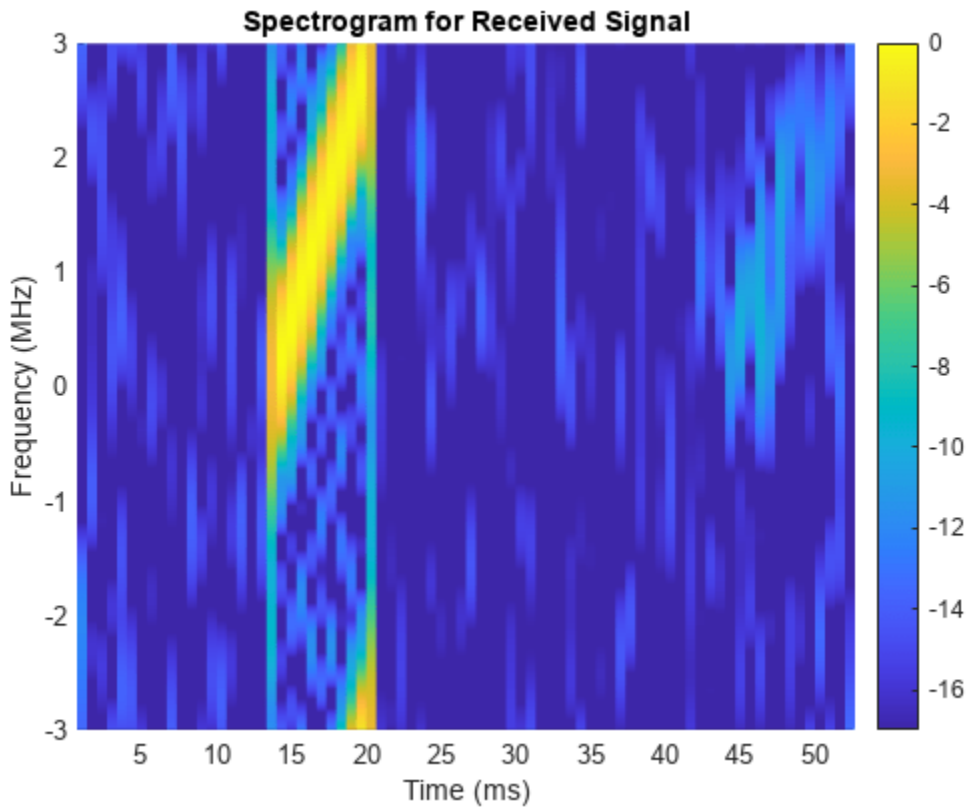
### Stretch Processing

To perform stretch processing, first determine a reference range. In this example, the goal is to search targets around 6700 m away from the radar, in a 500-meter window. A stretch processor can be formed using the waveform, the desired reference range and the range span.

```
refrng = 6700;
rngspan = 500;
prop_speed = physconst('lightspeed');
stretchproc = getStretchProcessor(waveform,refrng,rngspan,prop_speed)

stretchproc =
  phased.StretchProcessor with properties:

          SampleRate: 5.9958e+06
          PulseWidth: 6.6713e-06
           PRFSource: 'Property'
                 PRF: 1.8737e+04
          SweepSlope: 4.4938e+11
       SweepInterval: 'Positive'
    PropagationSpeed: 299792458
      ReferenceRange: 6700
           RangeSpan: 500
```

Next, pass the received pulses through the stretch processor.

```
y_stretch = stretchproc(rx_pulses);
```

Now, coherently integrate the pulses to improve the SNR.

```
y = pulsint(y_stretch,'coherent');
```

The spectrogram of the signal after stretch processing is shown below. Note that the second and third target echoes no longer appear as a ramp in the plot. Instead, their time-frequency signatures appear at constant frequencies, around 0.5 and -0.5 MHz. Hence, the signal is deramped. In addition, there is no return present from the first target. In fact, any signal outside the ranges of interest has been suppressed. This is because the stretch processor only allows target returns within the range window to pass. This process is often referred to as *range gating* in a real system.

```
helperStretchSignalSpectrogram(y,fs,16,12,'Deramped Signal');
```
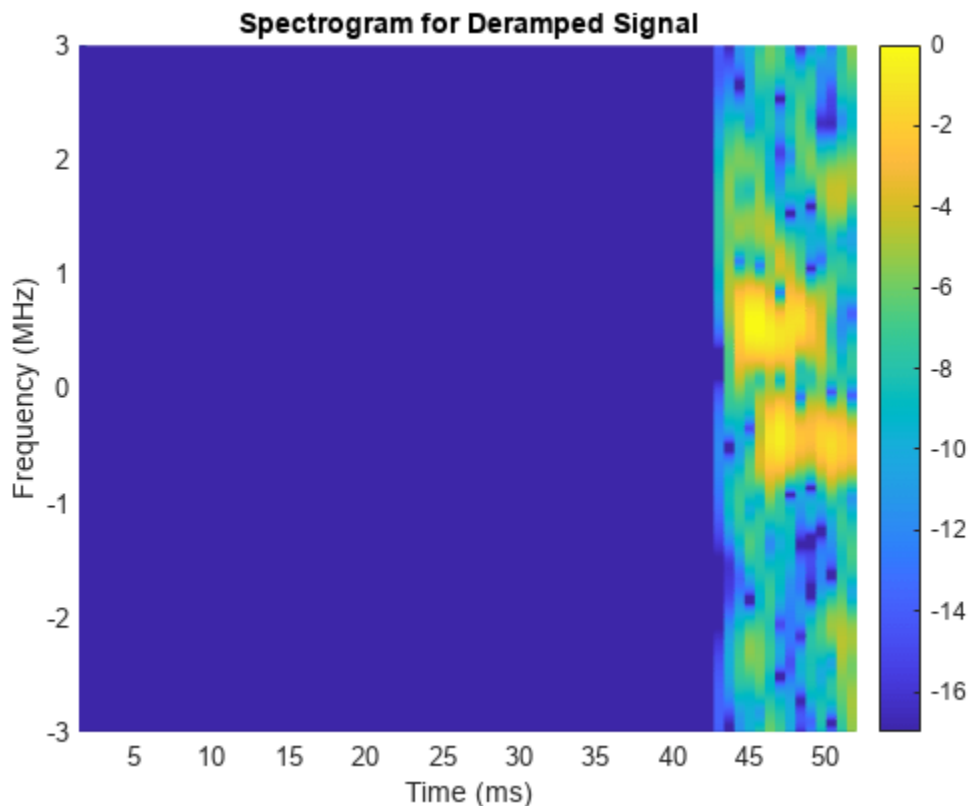


**Range Estimation**

To estimate the target range, plot the spectrum of the signal.

```
periodogram(y,[],2048,stretchproc.SampleRate,'centered');
```
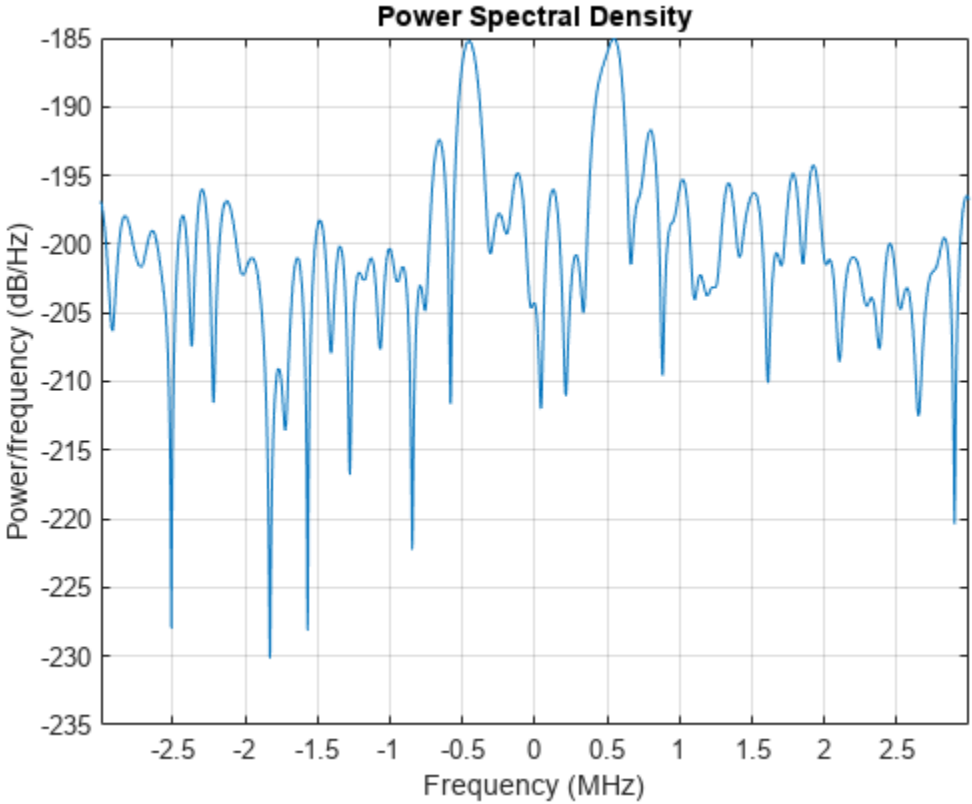
From the figure, it is clear that there are two dominant frequency components in the deramped signal, which correspond to two targets. The frequencies of these peaks can be used to determine the true range values of these targets.

```
[p, f] = periodogram(y,[],2048,stretchproc.SampleRate,'centered');
```

```
[~,rngidx] = findpeaks(pow2db(p/max(p)),'MinPeakHeight',-5);
rngfreq = f(rngidx);
re = stretchfreq2rng(rngfreq,...
    stretchproc.SweepSlope,stretchproc.ReferenceRange,prop_speed)
```

```
re = 2×1
10³ ×

    6.8514
    6.5174
```

The estimated ranges are 6518 and 6852 meters, matching the true ranges of 6533 and 6845 meters.

## Reduced Sample Rate

As mentioned in the introduction section, an attractive feature of stretch processing is that it reduces the bandwidth requirement for successive processing stages. In this example, the range span of interest is 500 meters. The required bandwidth for the successive processing stages can be computed as

```
rngspan_bw = ...
    2*rngspan/prop_speed*waveform.SweepBandwidth/waveform.PulseWidth
```

```
rngspan_bw = 1.4990e+06
```

Following the same design rule as in the original system, where twice the bandwidth is used as the sampling frequency, the new required sampling frequency becomes

```
fs_required = 2*rngspan_bw
```

```
fs_required = 2.9979e+06
```

```
dec_factor = round(fs/fs_required)
```

```
dec_factor = 2
```

The resulting decimator factor is 2. This means that after performing stretch processing in the analog domain, signals can be sampled at only half of the sampling frequency compared to the case where the stretch processing is not used. Thus, the requirement on the A/D converter has been relaxed.

To verify this benefit in simulation, the next section shows that the same ranges can be estimated with the signal decimated after stretch processing.

```
% Design a decimation filter
decimator = design(fdesign.decimator(dec_factor,'lowpass',...
    'N,F3dB',10,1/dec_factor),'SystemObject',true);

% Decimate
y_stretch = decimator(y_stretch);
```

This time, the power spectral density is plotted against ranges.

```
y = pulsint(y_stretch,'coherent');
[p, f] = periodogram(y,[],2048,fs_required,'centered');
rng_bin = stretchfreq2rng(f,...
    stretchproc.SweepSlope,stretchproc.ReferenceRange,prop_speed);
plot(rng_bin,pow2db(p));
xlabel('Range (m)'); ylabel('Power/frequency (dB/Hz)'); grid on;
title('Periodogram Power Spectral Density Estimate');
```

**Periodogram Power Spectral Density Estimate**
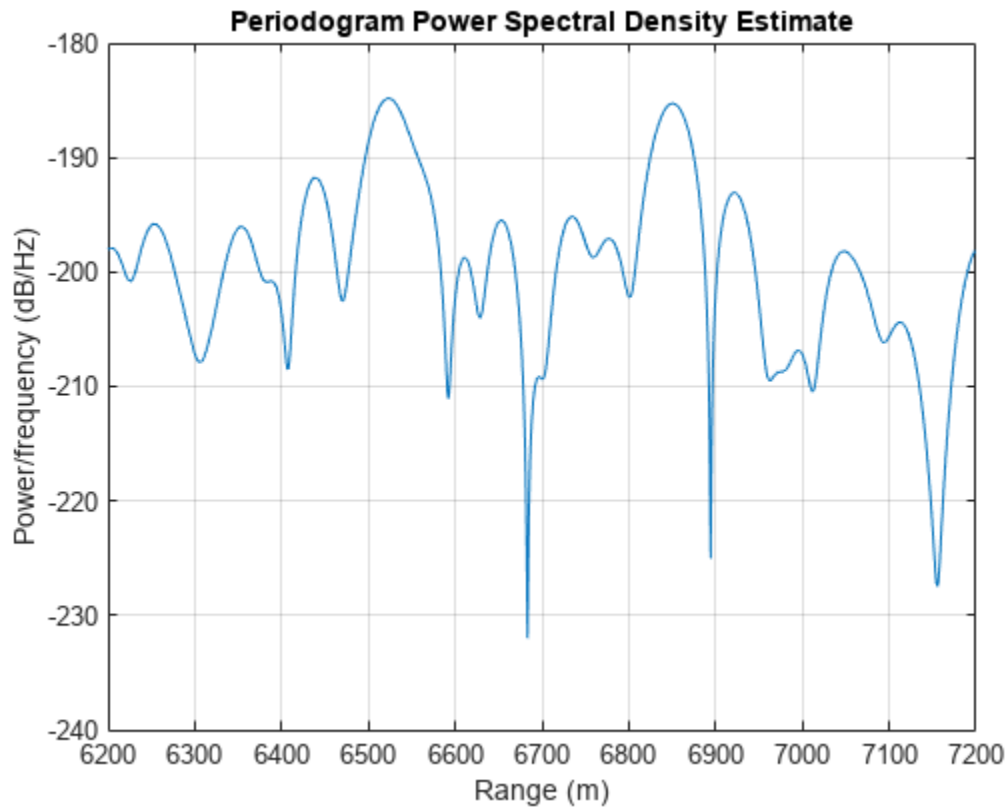
```
[~,rngidx] = findpeaks(pow2db(p/max(p)),'MinPeakHeight',-5);
re = rng_bin(rngidx)
```

```
re = 2×1
10³ ×

    6.8504
    6.5232
```

The true range values are 6533 and 6845 meters. Without decimation, the range estimates are 6518 and 6852 meters. With decimation, the range estimates are 6523 and 6851 meters. Therefore, the range estimation yields the same result with roughly only half of the computations compared to the nondecimated case.

## Summary

This example shows how to use stretch processing to estimate the target range when a linear FM waveform is used. It also shows that stretch processing reduces the bandwidth requirement.

## Reference

[1] Mark Richards, *Fundamentals of Radar Signal Processing*, McGraw-Hill, 2005.

# Signal Detection in White Gaussian Noise

This example discusses the detection of a deterministic signal in complex, white, Gaussian noise. This situation is frequently encountered in radar, sonar and communication applications.

**Overview**

There are many different kinds of detectors available for use in different applications. A few of the most popular ones are the Bayesian detector, maximum likelihood (ML) detector and Neyman-Pearson (NP) detector. In radar and sonar applications, NP is the most popular choice since it can ensure the probability of false alarm (*Pfa*) to be at a certain level.

In this example, we limit our discussion to the scenario where the signal is deterministic and the noise is white and Gaussian distributed. Both signal and noise are complex.

The example discusses the following topics and their interrelations: coherent detection, noncoherent detection, matched filtering and receiver operating characteristic (ROC) curves.

**Signal and Noise Model**

The received signal is assumed to follow the model

$$x(t) = s(t) + n(t)$$

where *s(t)* is the signal and *n(t)* is the noise. Without losing the generality, we assume that the signal power is equal to 1 watt and the noise power is determined accordingly based on the signal to noise ratio (SNR). For example, for an SNR of 10 dB, the noise power, i.e., noise variance will be 0.1 watt.

**Matched Filter**

A matched filter is often used at the receiver front end to enhance the SNR. From the discrete signal point of view, matched filter coefficients are simply given by the complex conjugated reversed signal samples.

When dealing with complex signals and noises, there are two types of receivers. The first kind is a coherent receiver, which assumes that both the amplitude and phase of the received signal are known. This results in a perfect match between the matched filter coefficients and the signal *s*. Therefore, the matched filter coefficients can be considered as the conjugate of *s*. The matched filter operation can then be modeled as

$$y = s*x = s*(s + n) = |s|^2 + s*n \ .$$

Note that although the general output *y* is still a complex quantity, the signal is completely characterized by $|s|^2$ , which is a real number and contained in the real part of *y*. Hence, the detector following the matched filter in a coherent receiver normally uses only the real part of the received signal. Such a receiver can normally provide the best performance. However, the coherent receiver is vulnerable to phase errors. In addition, a coherent receiver also requires additional hardware to perform the phase detection. For a noncoherent receiver, the received signal is modeled as a copy of the original signal with a random phase error. With a noncoherent received signal, the detection after the matched filter is normally based on the power or magnitude of the signal since you need both real and imaginary parts to completely define the signal.

**Detector**

The objective function of the NP decision rule can be written as

$$J = P_d + g(P_{fa} - a),$$

i.e., to maximize the probability of the detection, *Pd*, while limiting the probability of false alarm, *Pfa* at a specified level *a*. The variable *g* in the equation is the Lagrange multiplier. The NP detector can be formed as a likelihood ratio test (LRT) as follows:

$$\frac{p_y(y|H_1)}{p_y(y|H_0)} \underset{H_0}{\overset{H_1}{\underset{<}{\gtrless}}} Th \quad .$$

In this particular NP situation, since the false alarm is caused by the noise alone, the threshold *Th* is determined by the noise to ensure the fixed *Pfa*. The general form of the LRT shown above is often difficult to evaluate. In real applications, we often use an easy to compute quantity from the signal, i.e., sufficient statistic, to replace the ratio of two probability density functions. For example, the sufficient statistics, *z* may be as simple as

$$z = |y| \quad ,$$

then the simplified detector becomes

$$z \underset{H_0}{\overset{H_1}{\underset{<}{\gtrless}}} T \quad .$$

*T* is the threshold to the sufficient statistic *z*, acting just like the threshold *Th* to the LRT. Therefore, the threshold is not only related to the probability distributions, but also depends on the choice of sufficient statistic.

**Single Sample Detection Using Coherent Receiver**

We will first explore an example of detecting a signal in noise using just one sample.

Assume the signal is a unit power sample and the SNR is 3 dB. Using a 100000-trial Monte-Carlo simulation, we generate the signal and noise as

```
% fix the random number generator
rstream = RandStream.create('mt19937ar','seed',2009);

Ntrial = 1e5;              % number of Monte-Carlo trials
snrdb = 3;                 % SNR in dB
snr = db2pow(snrdb);       % SNR in linear scale
spower = 1;                % signal power is 1
npower = spower/snr;           % noise power
namp = sqrt(npower/2);     % noise amplitude in each channel
s = ones(1,Ntrial);        % signal
n = namp*(randn(rstream,1,Ntrial)+1i*randn(rstream,1,Ntrial));  % noise
```

Note that the noise is complex, white and Gaussian distributed.

If the received signal contains the target, it is given by

```
x = s + n;
```

The matched filter in this case is trivial, since the signal itself is a unit sample.

```
mf = 1;
```

In this case, the matched filter gain is 1, therefore, there is no SNR gain.

Now we do the detection and examine the performance of the detector. For a coherent receiver, the received signal after the matched filter is given by

```
y = mf'*x;  % apply the matched filter
```

The sufficient statistic, i.e., the value used to compare to the detection threshold, for a coherent detector is the real part of the received signal after the matched filter, i.e.,

```
z = real(y);
```

Let's assume that we want to fix the Pfa to 1e-3. Given the sufficient statistic, $z$, the decision rule becomes

$$z \underset{H_0}{\overset{H_1}{\gtrless}} T$$

where the threshold $T$ is related to *Pfa* as

$$P_{fa} = \frac{1}{2}\left[1 - \text{erf}\left(\frac{T}{\sqrt{NM}}\right)\right]$$

In the equation, $N$ is the signal power and $M$ is the matched filter gain. Note that $T$ is the threshold of the signal after the matched filter and $NM$ represents the noise power after the matched filter, so $\frac{T}{\sqrt{NM}}$ can be considered as the ratio between the signal and noise magnitude, i.e., it is related to the signal to noise ratio, SNR. Since SNR is normally referred to as the ratio between the signal and noise power, considering the units of each quantity in this expression, we can see that

$$\frac{T}{\sqrt{NM}} = \sqrt{\text{SNR}} \quad .$$

Since $N$ and $M$ are fixed once the noise and signal waveform are chosen, there is a correspondence between $T$ and SNR. Given T is a threshold of the signal, SNR can be considered as a threshold of the signal to noise ratio. Therefore, the threshold equation can then be rewritten in the form of

$$P_{fa} = \frac{1}{2}[1 - \text{erf}(\sqrt{\text{SNR}})] .$$

The required SNR threshold given a complex, white Gaussian noise for the NP detector can be calculated using the npwgnthresh function as follows:

```
Pfa = 1e-3;
snrthreshold = db2pow(npwgnthresh(Pfa, 1,'coherent'));
```

Note that this threshold, although also in the form of an SNR value, is different to the SNR of the received signal. The threshold SNR is a calculated value based on the desired detection performance, in this case the *Pfa*; while the received signal SNR is the physical characteristic of the signal determined by the propagation environment, the waveform, the transmit power, etc.

The true threshold $T$ can then be derived from this SNR threshold as

$$T = \sqrt{NM} \cdot \sqrt{\text{SNR}}.$$

```
mfgain = mf'*mf;
% To match the equation in the text above
% npower - N
% mfgain - M
% snrthreshold - SNR
threshold = sqrt(npower*mfgain*snrthreshold);
```

The detection is performed by comparing the signal to the threshold. Since the original signal, *s*, is presented in the received signal, a successful detection occurs when the received signal passes the threshold, i.e. *z>T*. The capability of the detector to detect a target is often measured by the *Pd*. In a Monte-Carlo simulation, *Pd* can be calculated as the ratio between the number of times the signal passes the threshold and the number of total trials.

```
Pd = sum(z>threshold)/Ntrial
```

```
Pd = 0.1390
```

On the other hand, a false alarm occurs when the detection shows that there is a target but there actually isn't one, i.e., the received signal passes the threshold when there is only noise present. The error probability of the detector to detect a target when there isn't one is given by *Pfa*.

```
x = n;
y = mf'*x;
z = real(y);
Pfa = sum(z>threshold)/Ntrial
```

```
Pfa = 9.0000e-04
```

which meets our requirement.

To see the relation among SNR, *Pd* and *Pfa* in a graph, we can plot the theoretical ROC curve using the rocsnr function for a SNR value of 3 dB as

```
rocsnr(snrdb,'SignalType','NonfluctuatingCoherent','MinPfa',1e-4);
```

**Nonfluctuating Coherent Receiver Operating Characteristic (ROC) Curves**
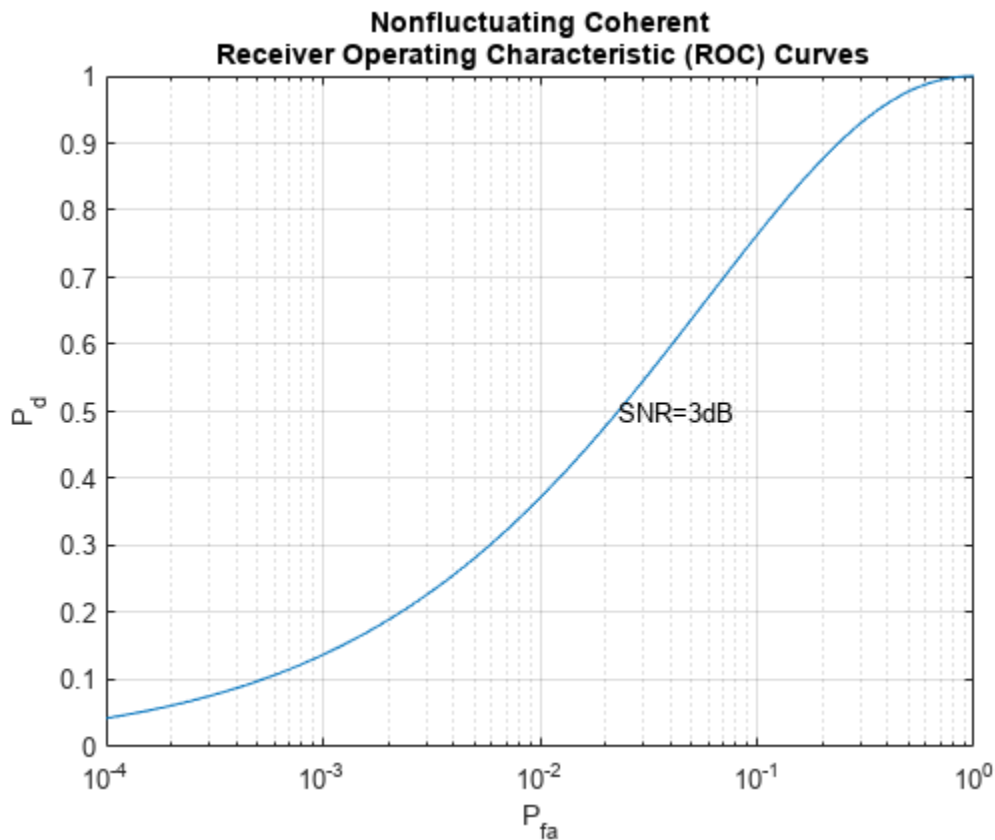
It can be seen from the figure that the measured *Pd*=0.1390 and *Pfa*=0.0009 obtained above for the SNR value of 3 dB match a theoretical point on the ROC curve.

**Single Sample Detection Using Noncoherent Receiver**

A noncoherent receiver does not know the phase of the received signal, therefore, for target present case, the signal x contains a phase term and is defined as

```
% simulate the signal
x = s.*exp(1i*2*pi*rand(rstream,1,Ntrial)) + n;
y = mf'*x;
```

When the noncoherent receiver is used, the quantity used to compare with the threshold is the power (or magnitude) of the received signal after the matched filter. In this simulation, we choose the magnitude as the sufficient statistic.

```
z = abs(y);
```

Given our choice of the sufficient statistic $z$, the threshold is related to *Pfa* by the equation

$$P_{fa} = \exp\left(-\frac{T^2}{NM}\right) = \exp(-\text{SNR}) \ .$$

The signal to noise ratio threshold SNR for an NP detector can be calculated using npwgnthresh as follows:

```
snrthreshold = db2pow(npwgnthresh(Pfa, 1,'noncoherent'));
```

The threshold, *T*, is derived from SNR as before

```
mfgain = mf'*mf;
threshold = sqrt(npower*mfgain*snrthreshold);
```

Again, *Pd* can then be obtained using

```
Pd = sum(z>threshold)/Ntrial
```

```
Pd = 0.0583
```

Note that this resulting *Pd* is inferior to the performance we get from a coherent receiver.

For the target absent case, the received signal contains only noise. We can calculate the *Pfa* using Monte-Carlo simulation as

```
x = n;
y = mf'*x;
z = abs(y);
Pfa = sum(z>threshold)/Ntrial
```

```
Pfa = 9.5000e-04
```

The ROC curve for a noncoherent receiver is plotted as

```
rocsnr(snrdb,'SignalType','NonfluctuatingNoncoherent','MinPfa',1e-4);
```
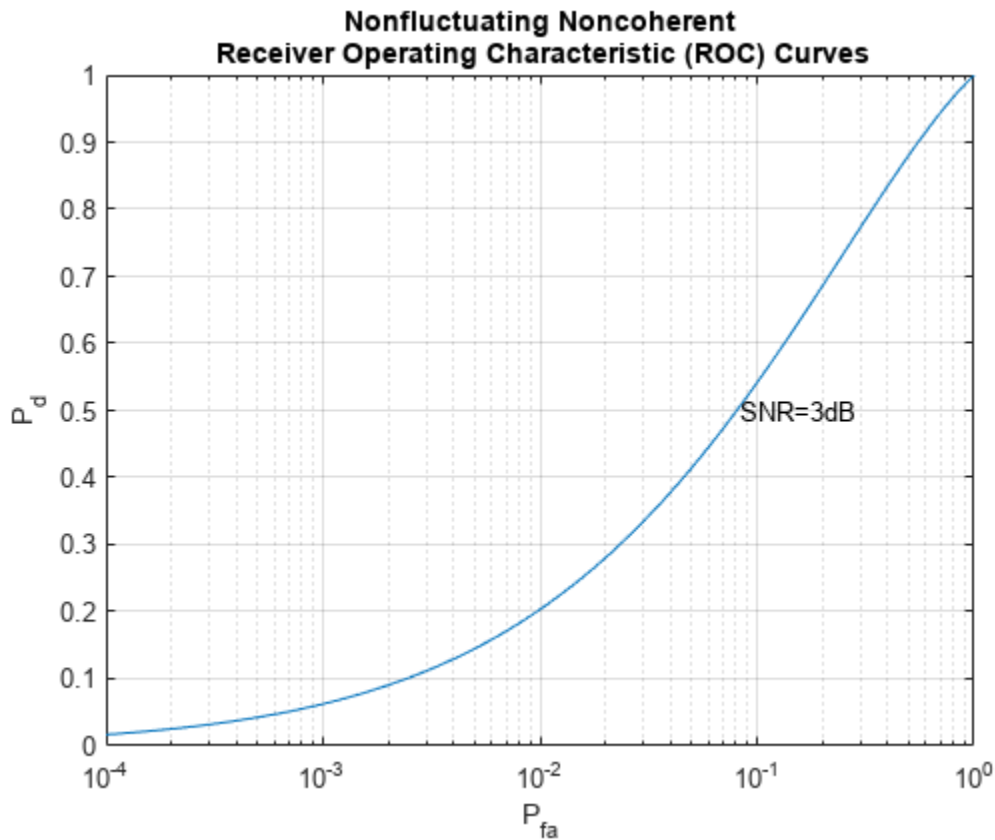


We can see that the performance of the noncoherent receiver detector is inferior to that of the coherent receiver.

**Summary**

This example shows how to simulate and perform different detection techniques using MATLAB®. The example illustrates the relationship among several frequently encountered variables in signal detection, namely, probability of detection (*Pd*), probability of false alarm (*Pfa*) and signal to noise ratio (SNR). In particular, the example calculates the performance of the detector using Monte-Carlo simulations and verifies the results of the metrics with the receiver operating characteristic (ROC) curves.

There are two SNR values we encounter in detecting a signal. The first one is the SNR of a single data sample. This is the SNR value appeared in a ROC curve plot. A point on ROC gives the required single sample SNR necessary to achieve the corresponding *Pd* and *Pfa*. However, it is NOT the SNR threshold used for detection. Using the Neyman-Pearson decision rule, the SNR threshold, the second SNR value we see in the detection, is determined by the noise distribution and the desired *Pfa* level. Therefore, such an SNR threshold indeed corresponds to the *Pfa* axis in a ROC curve. If we fix the SNR of a single sample, as depicted in the above ROC curve plots, each point on the curve will correspond to a *Pfa* value, which in turn translates to an SNR threshold value. Using this particular SNR threshold to perform the detection will then result in the corresponding *Pd*.

Note that an SNR threshold may not be the threshold used directly in the actual detector. The actual detector normally uses an easy to compute sufficient statistic quantity to perform the detection. Thus, the true threshold has to be derived from the aforementioned SNR threshold accordingly so that it is consistent with the choice of sufficient statistics.

This example performs the detection using only one received signal sample. Hence, the resulting *Pd* is fairly low and there is no processing gain achieved by the matched filter. To improve *Pd* and to take advantage of the processing gain of the matched filter, we can use multiple samples, or even multiple pulses, of the received signal. For more information about how to detect a signal using multiple samples or pulses, please refer to the example "Signal Detection Using Multiple Samples" on page 18-327.

# Signal Detection Using Multiple Samples

This example shows how to detect a signal in complex, white Gaussian noise using multiple received signal samples. A matched filter is used to take advantage of the processing gain.

**Introduction**

The example, "Signal Detection in White Gaussian Noise" on page 18-320, introduces a basic signal detection problem. In that example, only one sample of the received signal is used to perform the detection. This example involves more samples in the detection process to improve the detection performance.

As in the previous example, assume that the signal power is 1 and the single sample signal to noise ratio (SNR) is 3 dB. The number of Monte Carlo trials is 100000. The desired probability of false alarm (*Pfa*) level is 0.001.

```
Ntrial = 1e5;              % number of Monte Carlo trials
Pfa = 1e-3;                % Pfa

snrdb = 3;                 % SNR in dB
snr = db2pow(snrdb);       % SNR in linear scale
npower = 1/snr;            % noise power
namp = sqrt(npower/2);     % noise amplitude in each channel
```

**Signal Detection Using Longer Waveform**

As discussed in the previous example, the threshold is determined based on *Pfa*. Therefore, as long as the threshold is chosen, the *Pfa* is fixed, and vice versa. Meanwhile, one certainly prefers to have a higher probability of detection (*Pd*). One way to achieve that is to use multiple samples to perform the detection. For example, in the previous case, the SNR at a single sample is 3 dB. If one can use multiple samples, then the matched filter can produce an extra gain in SNR and thus improve the performance. In practice, one can use a longer waveform to achieve this gain. In the case of discrete time signal processing, multiple samples can also be obtained by increasing the sampling frequency.

Assume that the waveform is now formed with two samples:

```
Nsamp = 2;
wf = ones(Nsamp,1);
mf = conj(wf(end:-1:1));   % matched filter
```

For a coherent receiver, the signal, noise and threshold are given by

```
% fix the random number generator
rstream = RandStream.create('mt19937ar','seed',2009);

s = wf*ones(1,Ntrial);
n = namp*(randn(rstream,Nsamp,Ntrial)+1i*randn(rstream,Nsamp,Ntrial));
snrthreshold = db2pow(npwgnthresh(Pfa, 1,'coherent'));
mfgain = mf'*mf;
threshold = sqrt(npower*mfgain*snrthreshold);   % Final threshold T
```

If the target is present

```
x = s + n;
y = mf'*x;
```

```
z = real(y);
Pd = sum(z>threshold)/Ntrial
```

Pd = 0.3947

If the target is absent

```
x = n;
y = mf'*x;
z = real(y);
Pfa = sum(z>threshold)/Ntrial
```

Pfa = 0.0011

Notice that the SNR is improved by the matched filter.

```
snr_new = snr*mf'*mf;
snrdb_new = pow2db(snr_new)
```

snrdb_new = 6.0103

Plot the ROC curve with this new SNR value.

```
rocsnr(snrdb_new,'SignalType','NonfluctuatingCoherent','MinPfa',1e-4);
```
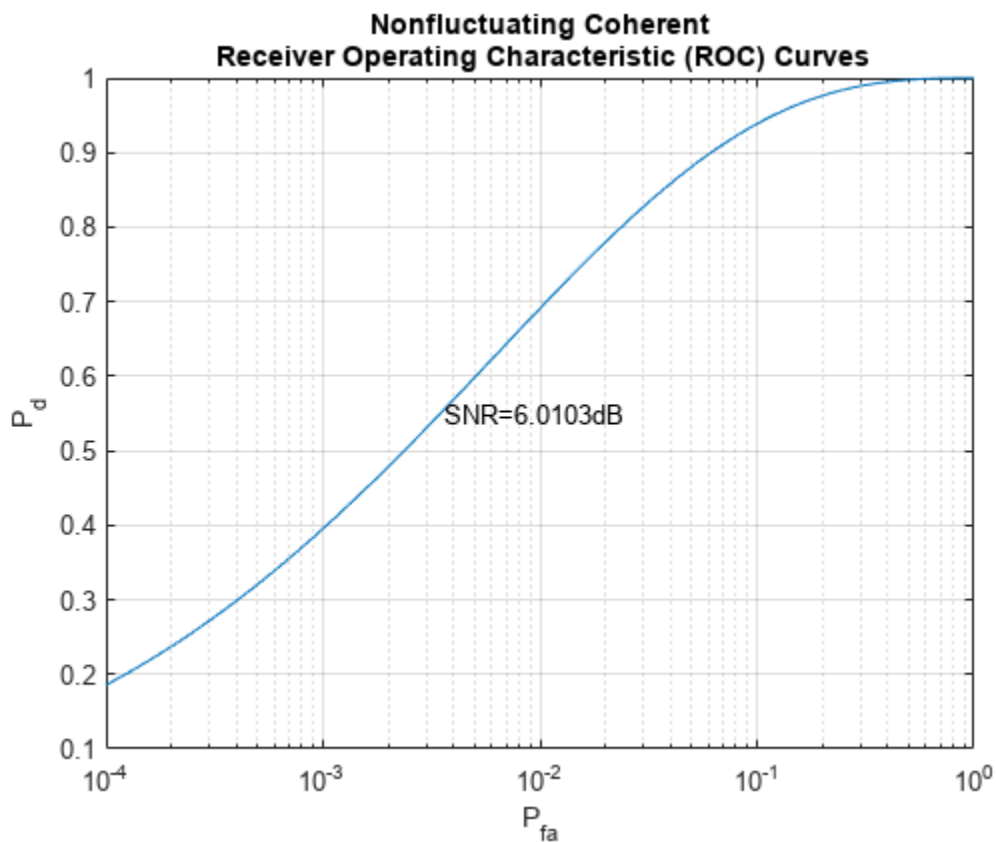


One can see from the figure that the point given by *Pfa* and *Pd* falls right on the curve. Therefore, the SNR corresponding to the ROC curve is the SNR of a single sample at the output of the matched filter. This shows that, although one can use multiple samples to perform the detection, the single

sample threshold in SNR (snrthreshold in the program) does not change compared to the simple sample case. There is no change because the threshold value is essentially determined by *Pfa*. However, the final threshold, *T,* does change because of the extra matched filter gain. The resulting *Pfa* remains the same compared to the case where only one sample is used to do the detection. However, the extra matched gain improved the *Pd* from 0.1390 to 0.3947.

One can run similar cases for the noncoherent receiver to verify the relation among *Pd*, *Pfa* and SNR.

### Signal Detection Using Pulse Integration

Radar and sonar applications frequently use pulse integration to further improve the detection performance. If the receiver is coherent, the pulse integration is just adding real parts of the matched filtered pulses. Thus, the SNR improvement is linear when one uses the coherent receiver. If one integrates 10 pulses, then the SNR is improved 10 times. For a noncoherent receiver, the relationship is not that simple. The following example shows the use of pulse integration with a noncoherent receiver.

Assume an integration of 2 pulses. Then, construct the received signal and apply the matched filter to it.

```
PulseIntNum = 2;
Ntotal = PulseIntNum*Ntrial;
s = wf*exp(1i*2*pi*rand(rstream,1,Ntotal));   % noncoherent
n = sqrt(npower/2)*...
    (randn(rstream,Nsamp,Ntotal)+1i*randn(rstream,Nsamp,Ntotal));
```

If the target is present

```
x = s + n;
y = mf'*x;
y = reshape(y,Ntrial,PulseIntNum);   % reshape to align pulses in columns
```

One can integrate the pulses using either of two possible approaches. Both approaches are related to the approximation of the modified Bessel function of the first kind, which is encountered in modeling the likelihood ratio test (LRT) of the noncoherent detection process using multiple pulses. The first approach is to sum abs(y)^2 across the pulses, which is often referred to as a *square law detector*. The second approach is to sum together abs(y) from all pulses, which is often referred to as a *linear detector*. For small SNR, square law detector is preferred while for large SNR, using linear detector is advantageous. We use square law detector in this simulation. However, the difference between the two kinds of detectors is normally within 0.2 dB.

For this example, choose the square law detector, which is more popular than the linear detector. To perform the square law detector, one can use the pulsint function. The function treats each column of the input data matrix as an individual pulse. The pulsint function performs the operation of

$$y = \sqrt{|x_1|^2 + \cdots + |x_n|^2} \ \ .$$

```
z = pulsint(y,'noncoherent');
```

The relation between the threshold *T* and the *Pfa*, given this new sufficient statistics, *z*, is given by

$$P_{fa} = 1 - I\left(\frac{T^2/(NM)}{\sqrt{L}}, L - 1\right) = 1 - I\left(\frac{\text{SNR}}{\sqrt{L}}, L - 1\right) \ \ .$$

where

$$I(u, K) = \int_0^{u\sqrt{K+1}} \frac{e^{-\tau}\tau^K}{K!}d\tau$$

is Pearson's form of the incomplete gamma function and $L$ is the number of pulses used for pulse integration. Using a square law detector, one can calculate the SNR threshold involving the pulse integration using the npwgnthresh function as before.

```
snrthreshold = db2pow(npwgnthresh(Pfa,PulseIntNum,'noncoherent'));
```

The resulting threshold for the sufficient statistics, $z$, is given by

```
mfgain = mf'*mf;
threshold = sqrt(npower*mfgain*snrthreshold);
```

The probability of detection is obtained by

```
Pd = sum(z>threshold)/Ntrial
```

```
Pd = 0.5343
```

Then, calculate the *Pfa* when the received signal is noise only using the noncoherent detector with 2 pulses integrated.

```
x = n;
y = mf'*x;
y = reshape(y,Ntrial,PulseIntNum);
z = pulsint(y,'noncoherent');
Pfa = sum(z>threshold)/Ntrial
```

```
Pfa = 0.0011
```

To plot the ROC curve with pulse integration, one has to specify the number of pulses used in integration in rocsnr function:

```
rocsnr(snrdb_new,'SignalType','NonfluctuatingNoncoherent',...
    'MinPfa',1e-4,'NumPulses',PulseIntNum);
```
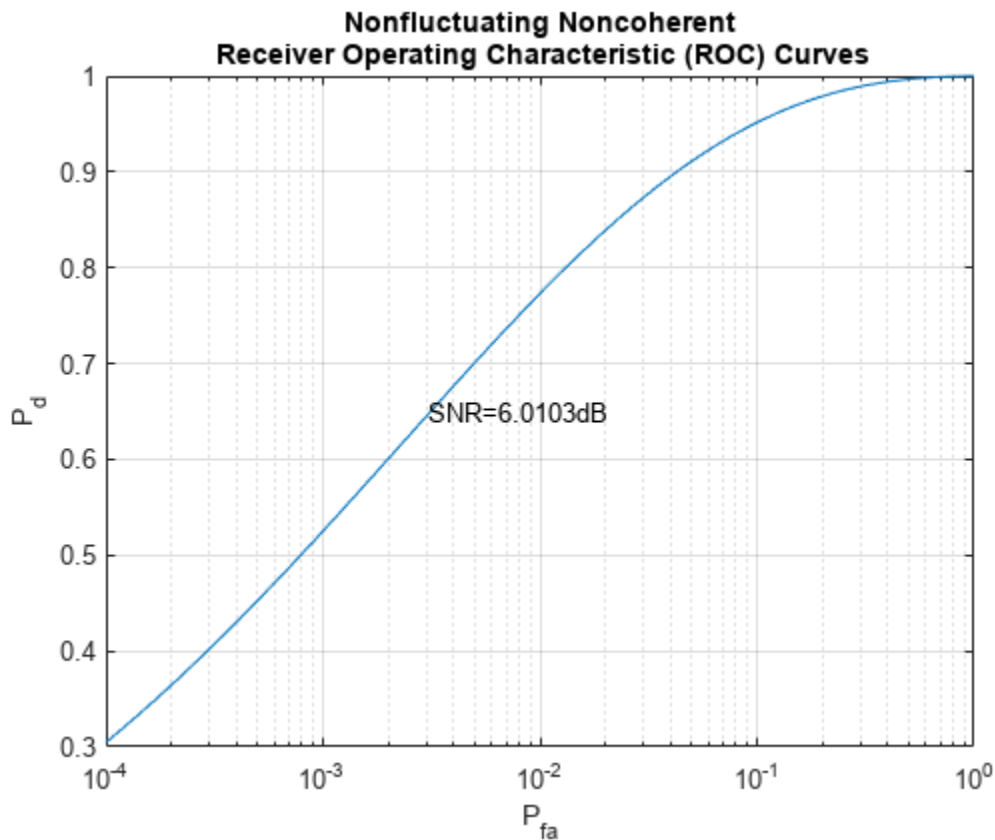
Again, the point given by *Pfa* and *Pd* falls on the curve. Thus, the SNR in the ROC curve specifies the SNR of a single sample used for the detection from one pulse.

Such an SNR value can also be obtained from Pd and Pfa using Albersheim's equation. The result obtained from Albersheim's equation is just an approximation, but is fairly good over frequently used *Pfa*, *Pd* and pulse integration range.

**Note:** Albersheim's equation has many assumptions, such as the target is nonfluctuating (Swirling case 0 or 5), the noise is complex, white Gaussian, the receiver is noncoherent and the linear detector is used for detection (square law detector for nonfluctuating target is also ok).

To calculate the necessary single sample SNR to achieve a certain *Pd* and *Pfa*, use the Albersheim function as

```
snr_required = albersheim(Pd,Pfa,PulseIntNum)
```

```
snr_required = 6.0009
```

This calculated required SNR value matches the new SNR value of 6 dB.

To see the improvement achieved in *Pd* by pulse integration, plot the ROC curve when there is no pulse integration used.

```
rocsnr(snrdb_new,'SignalType','NonfluctuatingNoncoherent',...
    'MinPfa',1e-4,'NumPulses',1);
```
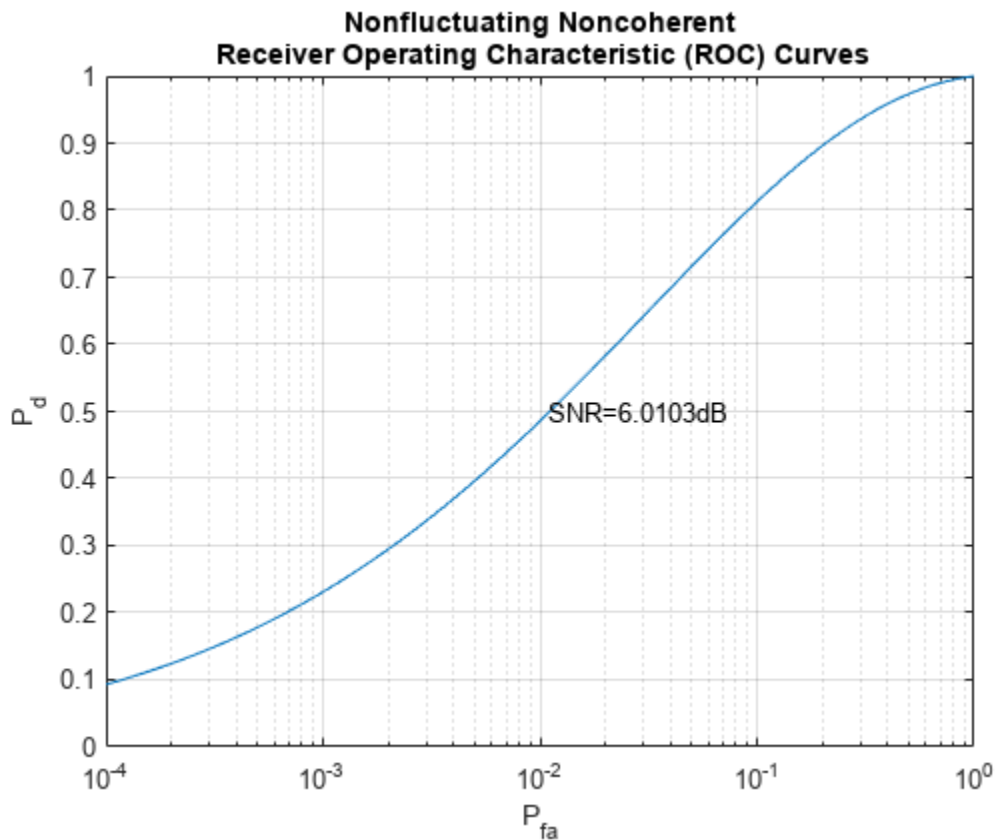
From the figure, one can see that without pulse integration, *Pd* can only be around 0.24 with *Pfa* at 1e-3. With 2-pulse integration, as illustrated in the above Monte Carlo simulation, for the same *Pfa*, the *Pd* is around 0.53.

**Summary**

This example showed how using multiple signal sample in detection can improve the probability of detection while maintaining a desired probability of false alarm level. In particular, it showed using either longer waveform or pulse integration technique to improve *Pd*. The example illustrates the relation among *Pd*, *Pfa*, ROC curve and Albersheim's equation. The performance is calculated using Monte Carlo simulations.

# Antenna Array Beam Scanning Visualization on a Map

This example shows how to visualize the changing pattern and coverage map of an antenna array as it scans a sweep of angles. The antenna array is created using Antenna Toolbox™ and Phased Array System Toolbox™. The array is designed to be directional and radiate in the xy-plane to generate a maximum coverage region in the geographic azimuth. Transmitter and receiver sites are created and shown on a map, and the pattern and coverage map are displayed as the antenna array is steered.

**Design a Reflector-Backed Dipole Antenna Element**

Use Antenna Toolbox to design a reflector-backed dipole antenna element. Design the element and its exciter for 10 GHz, and specify tilt to direct radiation in the xy-plane, which corresponds to the geographic azimuth.

```matlab
% Design reflector-backed dipole antenna element
fq = 10e9; % 10 GHz
myelement = design(reflector,fq);
myelement.Exciter = design(myelement.Exciter,fq);

% Tilt antenna element to radiate in xy-plane, with boresight along x-axis
myelement.Tilt = 90;
myelement.TiltAxis = "y";
myelement.Exciter.Tilt = 90;
myelement.Exciter.TiltAxis = "y";
```

**Create a 7-by-7 Rectangular Antenna Array**

Use Phased Array System Toolbox to create a 7-by-7 rectangular array from the antenna element. Specify the array normal to direct radiation in the x-axis direction.

```matlab
% Create 7-by-7 antenna array
nrow = 7;
ncol = 7;
myarray = phased.URA("Size",[nrow ncol],"Element",myelement);

% Define element spacing to be half-wavelength at 10 GHz, and specify
% array plane as yz-plane, which directs radiation in x-axis direction
lambda = physconst("lightspeed")/fq;
drow = lambda/2;
dcol = lambda/2;
myarray.ElementSpacing = [drow dcol];
myarray.ArrayNormal = "x";

% Display radiation pattern
f = figure;
az = -180:1:180;
el = -90:1:90;
pattern(myarray,fq,az,el)
```

**3D Directivity Pattern**
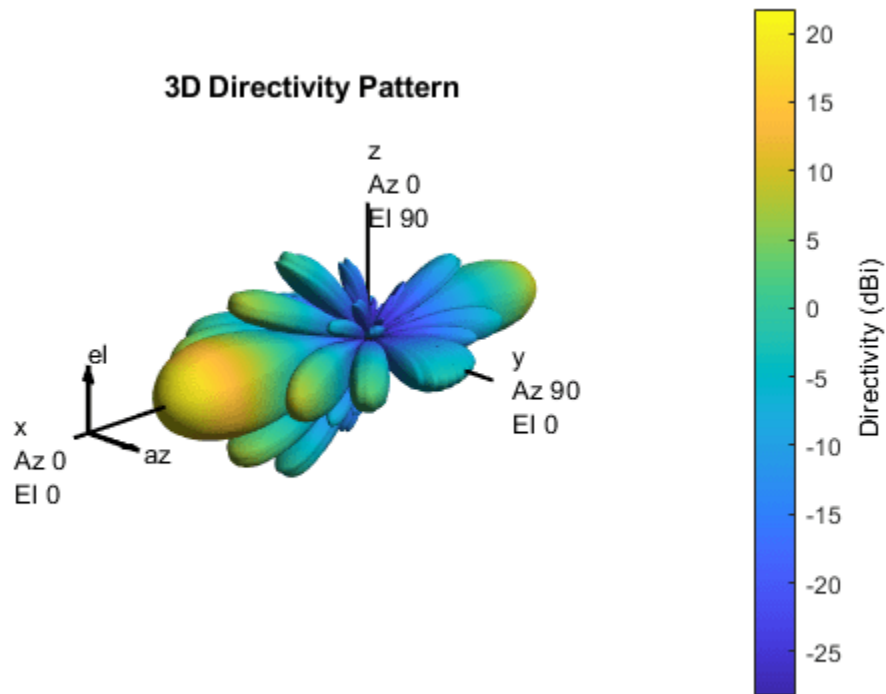


**Create Transmitter Site at Washington Monument**

Create a transmitter site at the Washington Monument in Washington, DC, using the antenna array. The transmitter frequency matches the antenna"s design frequency, and the transmitter output power is 1 W. Set antenna height to 169 m, which is the height of the monument.

```
tx = txsite("Name","Washington Monument",...
    "Latitude",38.88949, ...
    "Longitude",-77.03523, ...
    "Antenna",myarray,...
    "AntennaHeight",169', ...
    "TransmitterFrequency",fq,...
    "TransmitterPower",1);
```

**Show Transmitter Site on a Map**

Launch Site Viewer and show the transmitter site, which centers the view at the Washington Monument. The default map shows satellite imagery, and the site marker is shown at the site"s antenna height.

```
if isvalid(f)
    close(f)
end
viewer = siteviewer;
show(tx)
```
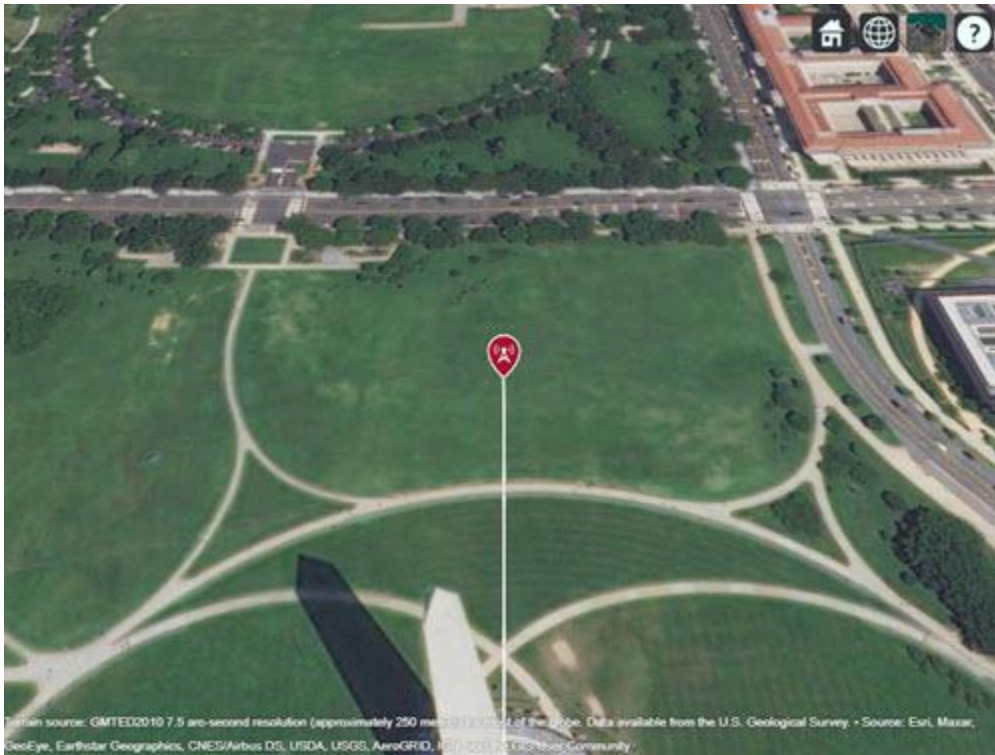
**Show Antenna Radiation Pattern on a Map**

Visualize the orientation of the antenna by showing the radiation pattern in Site Viewer.

```
pattern(tx)
```

Select the site marker to view the color legend of the pattern.
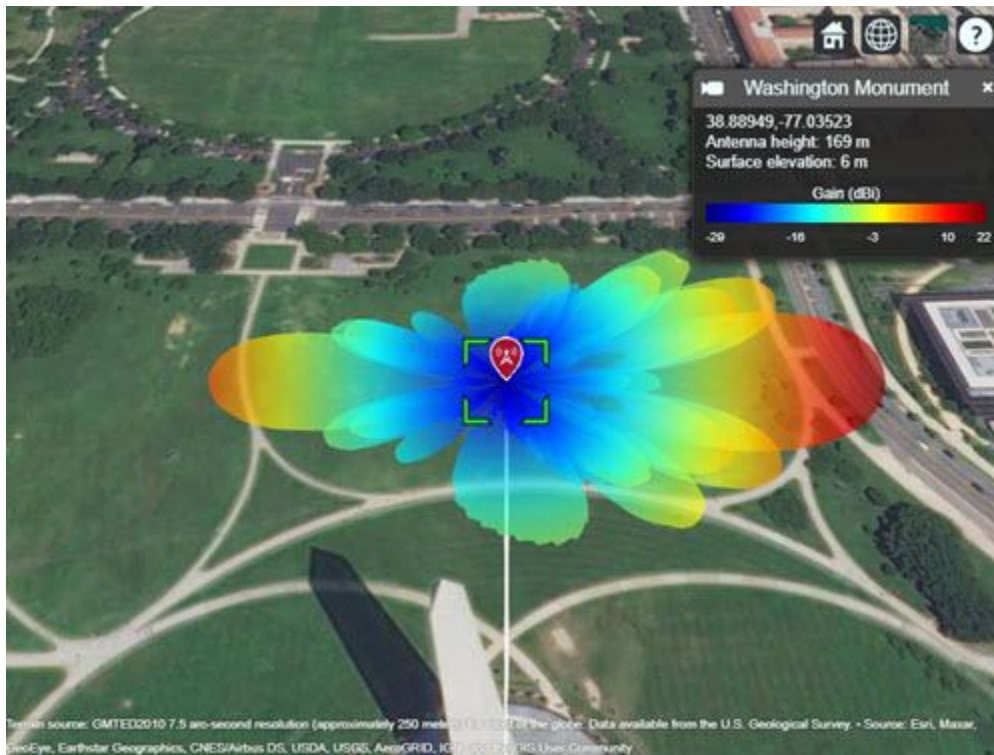
### Create Receiver Sites

Create an array of receiver sites in the Washington, DC, area. These are used as place markers for sites of interest to assess the coverage of the transmitter site.

```
% Define names for receiver sites
rxNames = [...
    "Brentwood Hamilton Field" ...
    "Nationals Park" ...
    "Union Station" ...
    "Georgetown University" ...
    "Arlington Cemetery"];
```

```
% Define coordinates for receiver sites
rxLocations = [...
    38.9080 -76.9958; ...
    38.8731 -77.0075; ...
    38.8976 -77.0062; ...
    38.9076 -77.0722; ...
    38.8783 -77.0685];
```

```
% Create array of receiver sites. Each receiver has a sensitivity of -75 dBm.
rxs = rxsite("Name",rxNames, ...
    "Latitude",rxLocations(:,1), ...
    "Longitude",rxLocations(:,2), ...
    "ReceiverSensitivity",-75);
```

Show receiver sites on a map.

```
show(rxs)
```

Set the map imagery using the `Basemap` property. Alternatively, open the map imagery picker in Site Viewer by clicking the second button from the right. Select "Streets" to see streets and labels on the map.

```
viewer.Basemap = "streets";
```
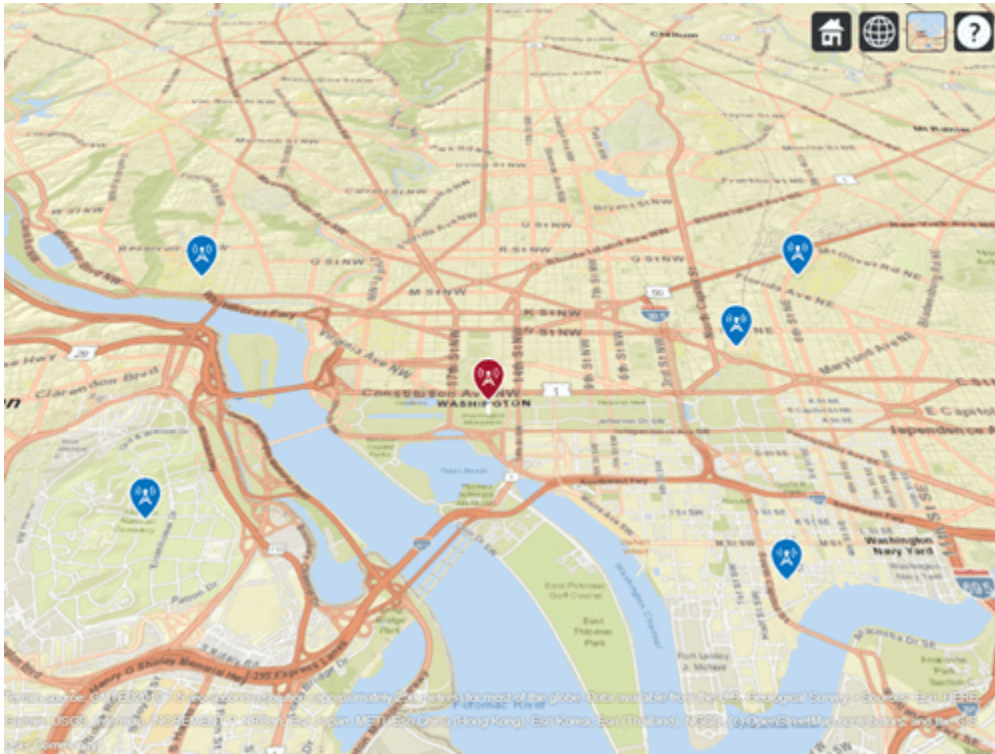


**Scan the Array and Update the Radiation Pattern**

Scan the antenna beam by applying a taper for a range of angles. For each angle, update the radiation pattern in Site Viewer. This approach of scanning the beam produces different patterns than physically rotating the antenna, as could be achieved by setting `AntennaAngle` of the transmitter site. This step is used to validate the orientation of the antenna"s main beam.

```
% Get the starting array taper
startTaper = myarray.Taper;

% Define angles over which to perform sweep
azsweep = -30:10:30;

% Set up tapering window and steering vector
N = nrow*ncol;
nbar = 5;
sll = -20;
sltaper = taylorwin(N,nbar,sll)';
steeringVector = phased.SteeringVector("SensorArray",myarray);

% Sweep the angles and show the antenna pattern for each
for az = azsweep
    sv = steeringVector(fq,[az; 0]);
    myarray.Taper = sltaper.*sv';
```

```
      % Update the radiation pattern. Use a larger size so the pattern is visible among the antenna
      pattern(tx, "Size", 2500,"Transparency",1);
end
```
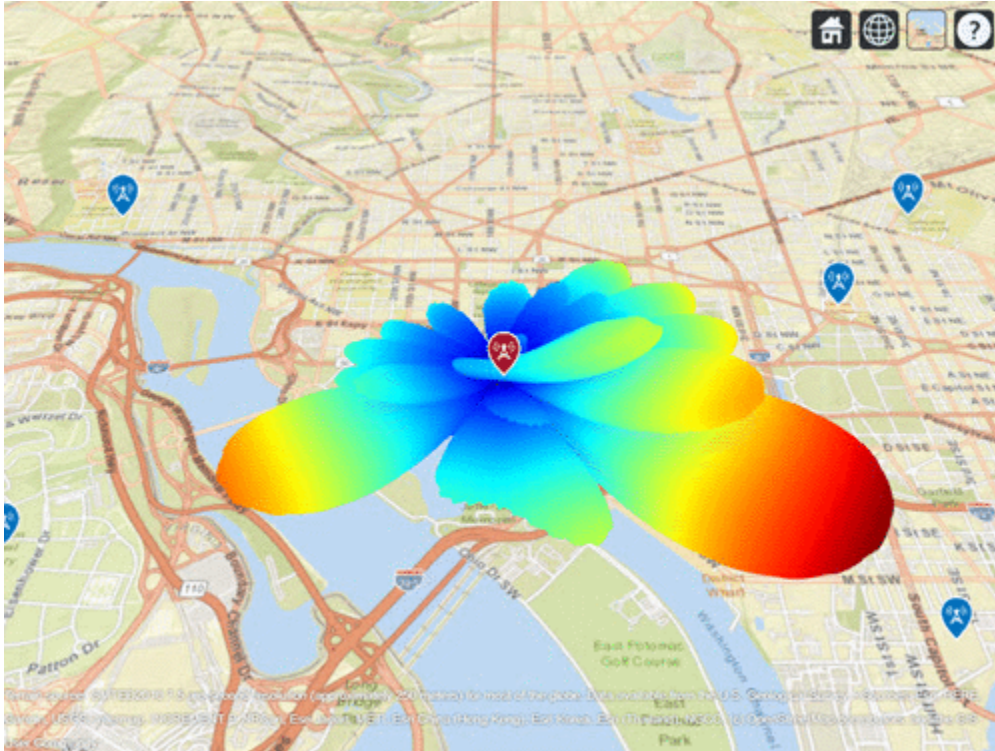


**Display Transmitter Coverage Map**

Define three signal strength levels and corresponding colors to display on the coverage map. Each color is visible where the received power for a mobile receiver meets the corresponding signal strength. The received power includes the total power transmitted from the rectangular antenna array.

The default orientation of the transmitter site points the antenna x-axis east, so that is the direction of maximum coverage.

```
% Reset the taper to the starting taper
myarray.Taper = startTaper;

% Define signal strength levels (dBm) and corresponding colors
strongSignal = -65;
mediumSignal = -70;
weakSignal = -75;
sigstrengths = [strongSignal mediumSignal weakSignal];
sigcolors = ["red" "yellow" "green"];

% Show the tx pattern
pattern(tx,"Size",500)

% Display coverage map out to 6 km
maxRange = 6000;
```

```
coverage(tx, ...
    "SignalStrengths",sigstrengths, ...
    "Colors",sigcolors, ...
    "MaxRange",maxRange)
```



The coverage map shows no coverage at the transmitter site and a couple of pockets of coverage along the boresight direction before the main coverage area. The radiation pattern provides insight into the coverage map by showing how the antenna power projects onto the map locations around the transmitter.

**Scan the Array and Update the Coverage Display**

Scan the antenna beam by applying a taper for a range of angles. For each angle, update the coverage map. This method of beamscanning is the same method used above. The final map includes two receiver sites of interest within the coverage region.

```
% Repeat the sweep but show the pattern and coverage map
for az = azsweep
    % Calculate and assign taper from steering vector
    sv = steeringVector(fq,[az; 0]);
    myarray.Taper = sltaper.*sv';

    % Update the tx pattern
    pattern(tx,"Size",500)

    % Update coverage map
    coverage(tx, ...
        "SignalStrengths",sigstrengths, ...
        "Colors",sigcolors, ...
        "MaxRange",maxRange)
end
```
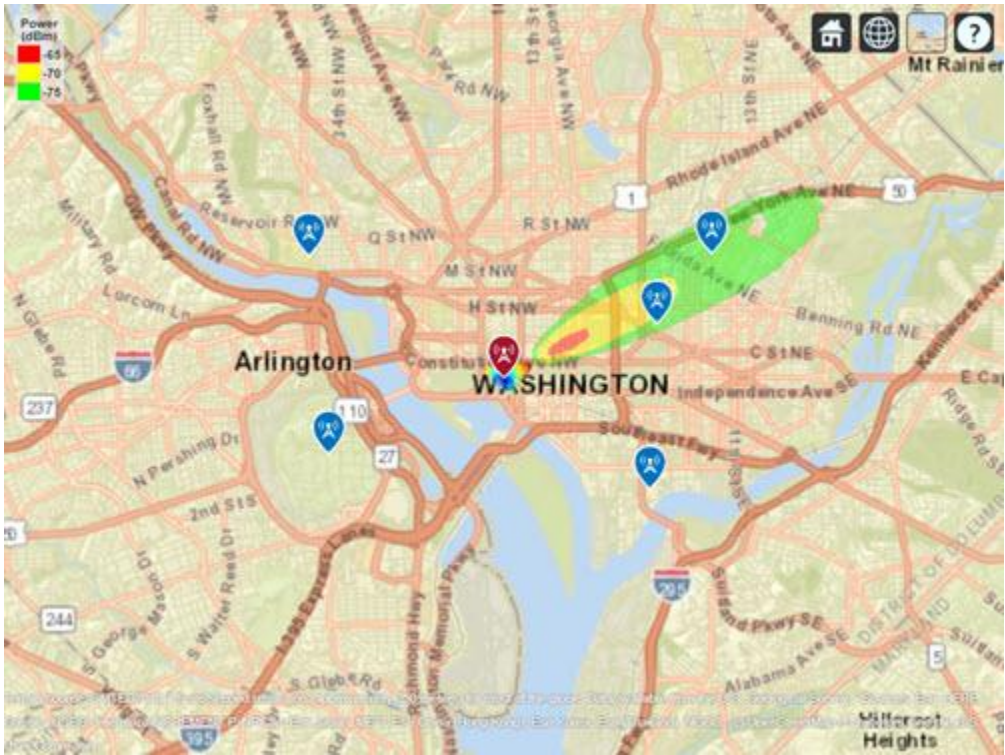
# SINR Map for a 5G Urban Macro-Cell Test Environment

This example shows how to construct a 5G urban macro-cell test environment and visualize the signal-to-interference-plus-noise ratio (SINR) on a map. The test environment is based on the guidelines defined in Report ITU-R M.[IMT-2020.EVAL] [1 on page 18-353] for evaluating 5G radio technologies. This report defines several test environments and usage scenarios in Section 8.2. The test environment in this example is based on the urban environment with high user density and traffic loads focusing on pedestrian and vehicular users (Dense Urban-eMBB). The test environment includes a hexagonal cell network as well as a custom antenna array that is implemented using Phased Array System Toolbox™.

**Define Network Layout**

The test environment guidelines for 5G technologies reuse the test network layout for 4G technologies defined in Section 8.3 of Report ITU-R M.2135-1 [2 on page 18-353], which is shown below. The layout consists of 19 sites placed in a hexagonal layout, each with 3 cells. The distance between adjacent sites is the inter-site distance (ISD) and depends on the test usage scenario. For the Dense Urban-eMBB test environment, the ISD is 200 m.
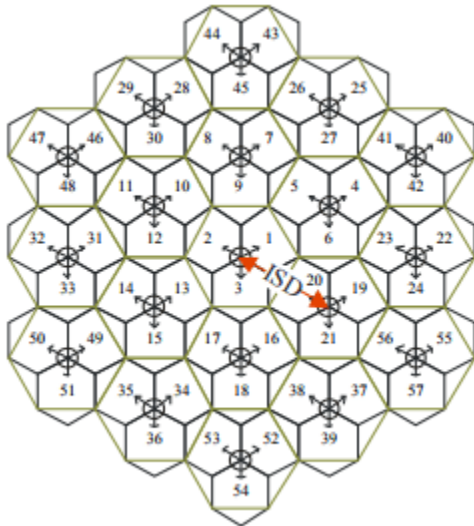


Create the locations corresponding to cell sites in the network layout, using MathWorks® Glasgow as the center location.

```
% Define center location site (cells 1-3)
centerSite = txsite('Name','MathWorks Glasgow', ...
    'Latitude',55.862787,...
    'Longitude',-4.258523);

% Initialize arrays for distance and angle from center location to each cell site, where
% each site has 3 cells
numCellSites = 19;
siteDistances = zeros(1,numCellSites);
siteAngles = zeros(1,numCellSites);

% Define distance and angle for inner ring of 6 sites (cells 4-21)
isd = 200; % Inter-site distance
```

```
siteDistances(2:7) = isd;
siteAngles(2:7) = 30:60:360;

% Define distance and angle for middle ring of 6 sites (cells 22-39)
siteDistances(8:13) = 2*isd*cosd(30);
siteAngles(8:13) = 0:60:300;

% Define distance and angle for outer ring of 6 sites (cells 40-57)
siteDistances(14:19) = 2*isd;
siteAngles(14:19) = 30:60:360;
```

**Define Cell Parameters**

Each cell site has three transmitters corresponding to each cell. Create arrays to define the names, latitudes, longitudes, and antenna angles of each cell transmitter.

```
% Initialize arrays for cell transmitter parameters
numCells = numCellSites*3;
cellLats = zeros(1,numCells);
cellLons = zeros(1,numCells);
cellNames = strings(1,numCells);
cellAngles = zeros(1,numCells);

% Define cell sector angles
cellSectorAngles = [30 150 270];

% For each cell site location, populate data for each cell transmitter
cellInd = 1;
for siteInd = 1:numCellSites
    % Compute site location using distance and angle from center site
    [cellLat,cellLon] = location(centerSite, siteDistances(siteInd), siteAngles(siteInd));

    % Assign values for each cell
    for cellSectorAngle = cellSectorAngles
        cellNames(cellInd) = "Cell " + cellInd;
        cellLats(cellInd) = cellLat;
        cellLons(cellInd) = cellLon;
        cellAngles(cellInd) = cellSectorAngle;
        cellInd = cellInd + 1;
    end
end
```

**Create Transmitter Sites**

Create transmitter sites using parameters defined above as well as configuration parameters defined for Dense Urban-eMBB. Launch Site Viewer and set the map imagery using the `Basemap` property. Alternatively, open the basemap picker in Site Viewer by clicking the second button from the right. Select "Topographic" to choose a basemap with topography, streets, and labels.

```
% Define transmitter parameters using Table 8-2 (b) of Report ITU-R M.[IMT-2020.EVAL]
fq = 4e9; % Carrier frequency (4 GHz) for Dense Urban-eMBB
antHeight = 25; % m
txPowerDBm = 44; % Total transmit power in dBm
txPower = 10.^((txPowerDBm-30)/10); % Convert dBm to W

% Create cell transmitter sites
txs = txsite('Name',cellNames, ...
    'Latitude',cellLats, ...
```

```
        'Longitude',cellLons, ...
        'AntennaAngle',cellAngles, ...
        'AntennaHeight',antHeight, ...
        'TransmitterFrequency',fq, ...
        'TransmitterPower',txPower);

% Launch Site Viewer
viewer = siteviewer;

% Show sites on a map
show(txs);
viewer.Basemap = 'topographic';
```
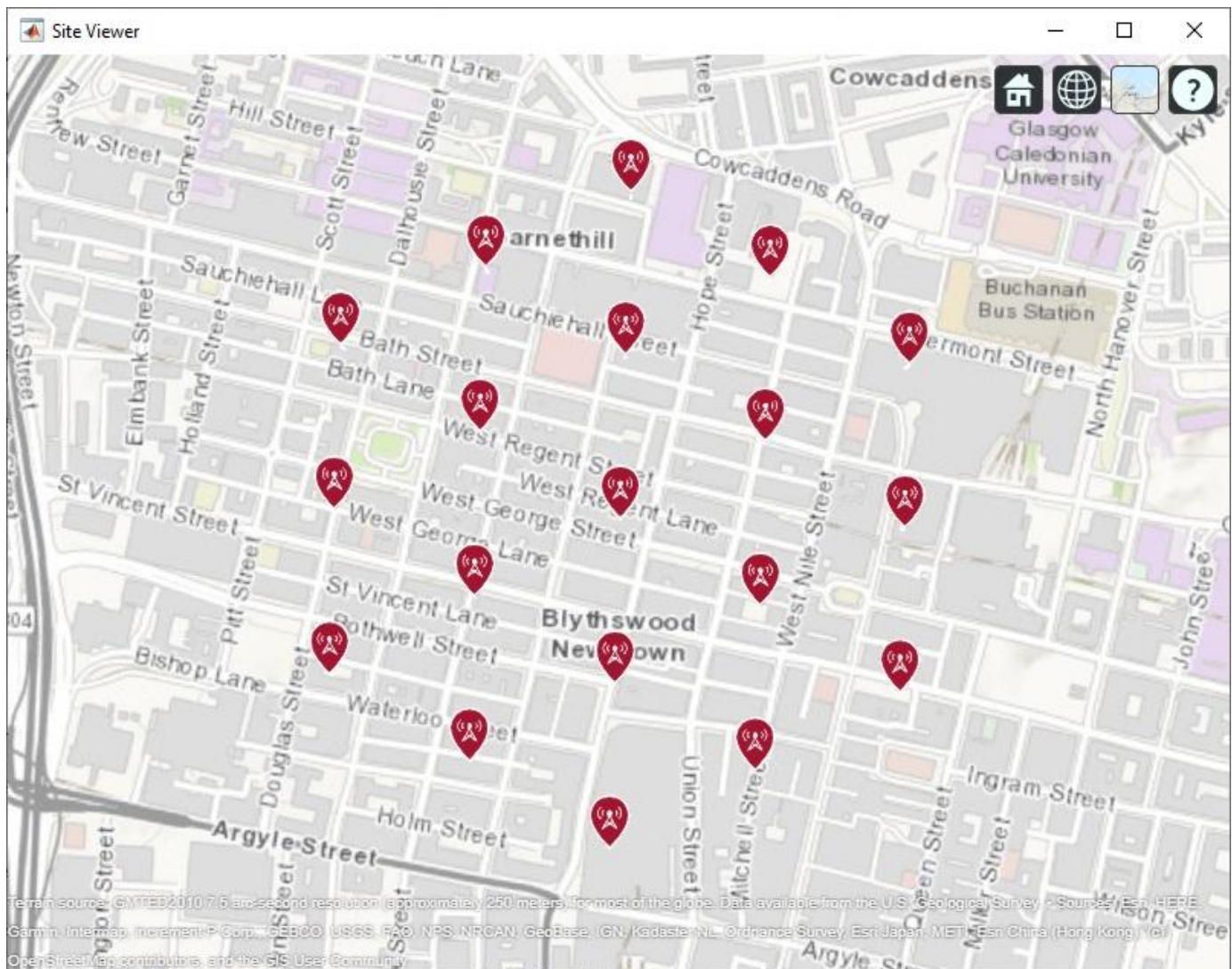


### Create Antenna Element

Section 8.5 of ITU-R report [1 on page 18-353] defines antenna characteristics for base station antennas. The antenna is modeled as having one or more antenna panels, where each panel has one or more antenna elements. Use Phased Array System Toolbox to implement the antenna element pattern defined in the report.

```
% Define pattern parameters
azvec = -180:180;
elvec = -90:90;
Am = 30; % Maximum attenuation (dB)
tilt = 0; % Tilt angle
az3dB = 65; % 3 dB bandwidth in azimuth
el3dB = 65; % 3 dB bandwidth in elevation

% Define antenna pattern
[az,el] = meshgrid(azvec,elvec);
azMagPattern = -12*(az/az3dB).^2;
elMagPattern = -12*((el-tilt)/el3dB).^2;
combinedMagPattern = azMagPattern + elMagPattern;
combinedMagPattern(combinedMagPattern<-Am) = -Am; % Saturate at max attenuation
phasepattern = zeros(size(combinedMagPattern));

% Create antenna element
antennaElement = phased.CustomAntennaElement(...
    'AzimuthAngles',azvec, ...
    'ElevationAngles',elvec, ...
    'MagnitudePattern',combinedMagPattern, ...
    'PhasePattern',phasepattern);

% Display radiation pattern
f = figure;
pattern(antennaElement,fq);
```
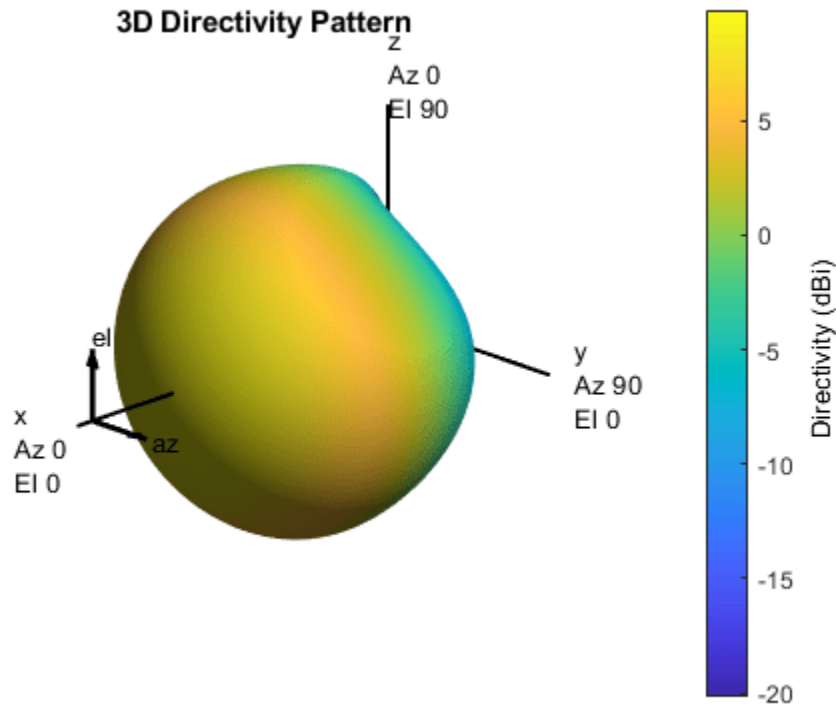
**Display SINR Map for Single Antenna Element**

Visualize SINR for the test scenario using a single antenna element and the free space propagation model. For each location on the map within the range of the transmitter sites, the signal source is the cell with the greatest signal strength, and all other cells are sources of interference. Areas with no color within the network indicate areas where the SINR is below the default threshold of -5 dB.

```
% Assign the antenna element for each cell transmitter
for tx = txs
    tx.Antenna = antennaElement;
end

% Define receiver parameters using Table 8-2 (b) of Report ITU-R M.[IMT-2020.EVAL]
bw = 20e6; % 20 MHz bandwidth
rxNoiseFigure = 7; % dB
rxNoisePower = -174 + 10*log10(bw) + rxNoiseFigure;
rxGain = 0; % dBi
rxAntennaHeight = 1.5; % m

% Display SINR map
if isvalid(f)
    close(f)
end
sinr(txs,'freespace', ...
    'ReceiverGain',rxGain, ...
    'ReceiverAntennaHeight',rxAntennaHeight, ...
    'ReceiverNoisePower',rxNoisePower, ...
    'MaxRange',isd, ...
    'Resolution',isd/20)
```
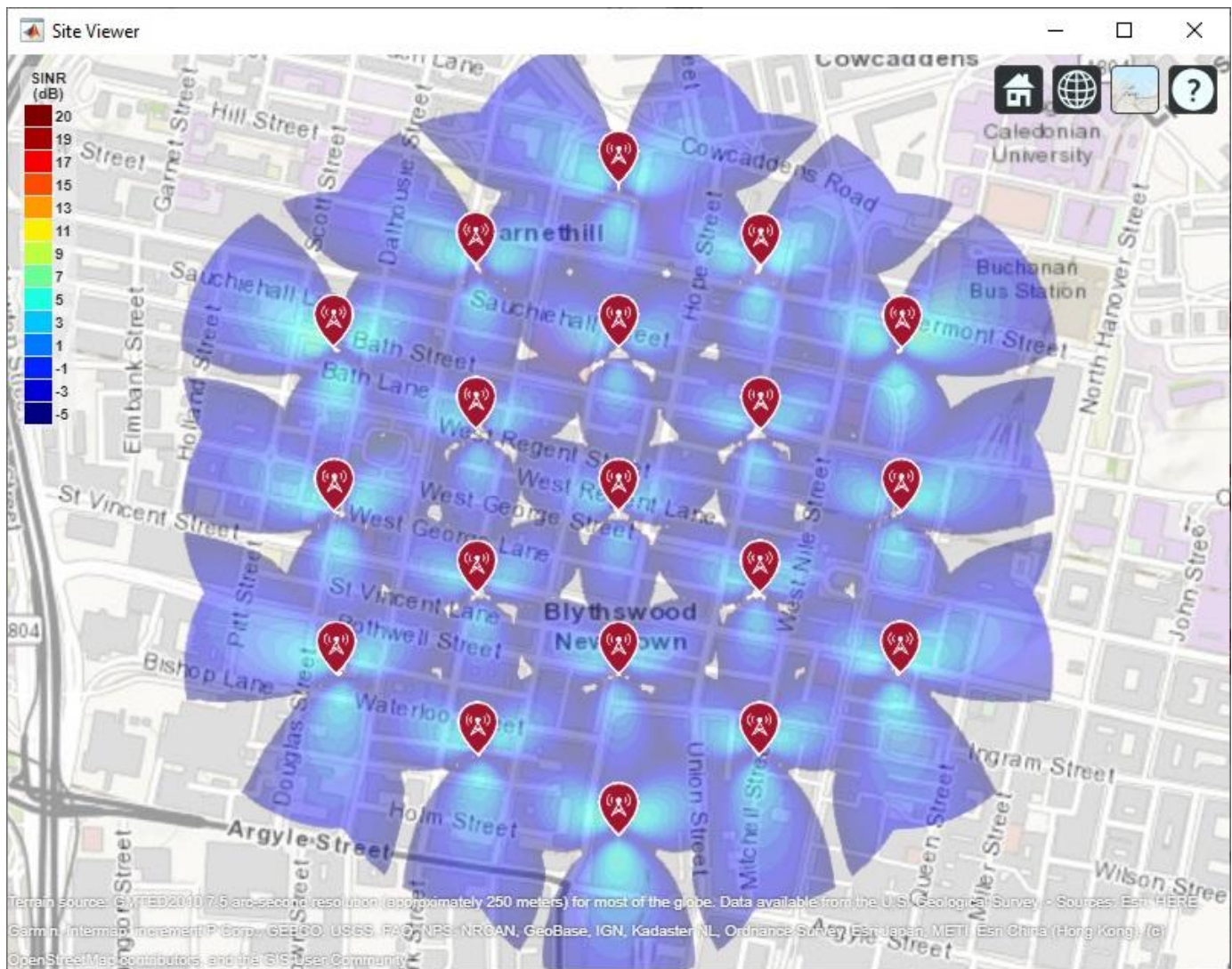
### Create 8-by-8 Rectangular Antenna Array

Define an antenna array to increase directional gain and increase peak SINR values. Use Phased Array System Toolbox to create an 8-by-8 uniform rectangular array.

```
% Define array size
nrow = 8;
ncol = 8;

% Define element spacing
lambda = physconst('lightspeed')/fq;
drow = lambda/2;
dcol = lambda/2;

% Define taper to reduce sidelobes
dBdown = 30;
taperz = chebwin(nrow,dBdown);
tapery = chebwin(ncol,dBdown);
tap = taperz*tapery.'; % Multiply vector tapers to get 8-by-8 taper values
```

```
% Create 8-by-8 antenna array
cellAntenna = phased.URA('Size',[nrow ncol], ...
    'Element',antennaElement, ...
    'ElementSpacing',[drow dcol], ...
    'Taper',tap, ...
    'ArrayNormal','x');

% Display radiation pattern
f = figure;
pattern(cellAntenna,fq);
```
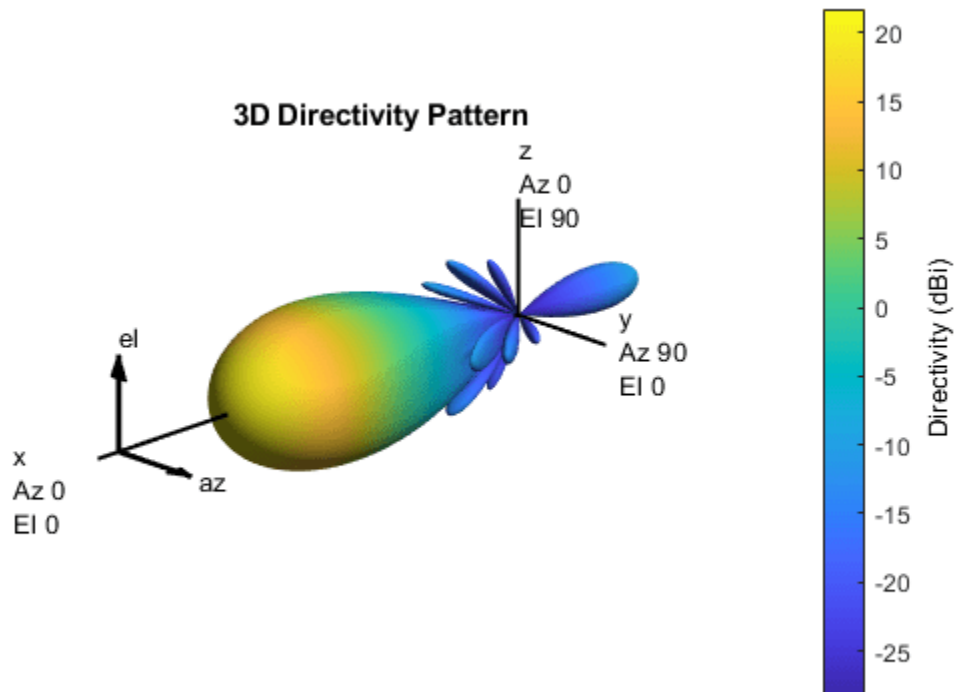


**Display SINR Map for 8-by-8 Antenna Array**

Visualize SINR for the test scenario using a uniform rectangular antenna array and the free space propagation model. Apply a mechanical downtilt to illuminate the intended ground area around each transmitter.

```
% Assign the antenna array for each cell transmitter, and apply downtilt.
% Without downtilt, pattern is too narrow for transmitter vicinity.
downtilt = 15;
for tx = txs
    tx.Antenna = cellAntenna;
    tx.AntennaAngle = [tx.AntennaAngle; -downtilt];
end

% Display SINR map
if isvalid(f)
```

```
        close(f)
end
sinr(txs,'freespace', ...
        'ReceiverGain',rxGain, ...
        'ReceiverAntennaHeight',rxAntennaHeight, ...
        'ReceiverNoisePower',rxNoisePower, ...
        'MaxRange',isd, ...
        'Resolution',isd/20)
```
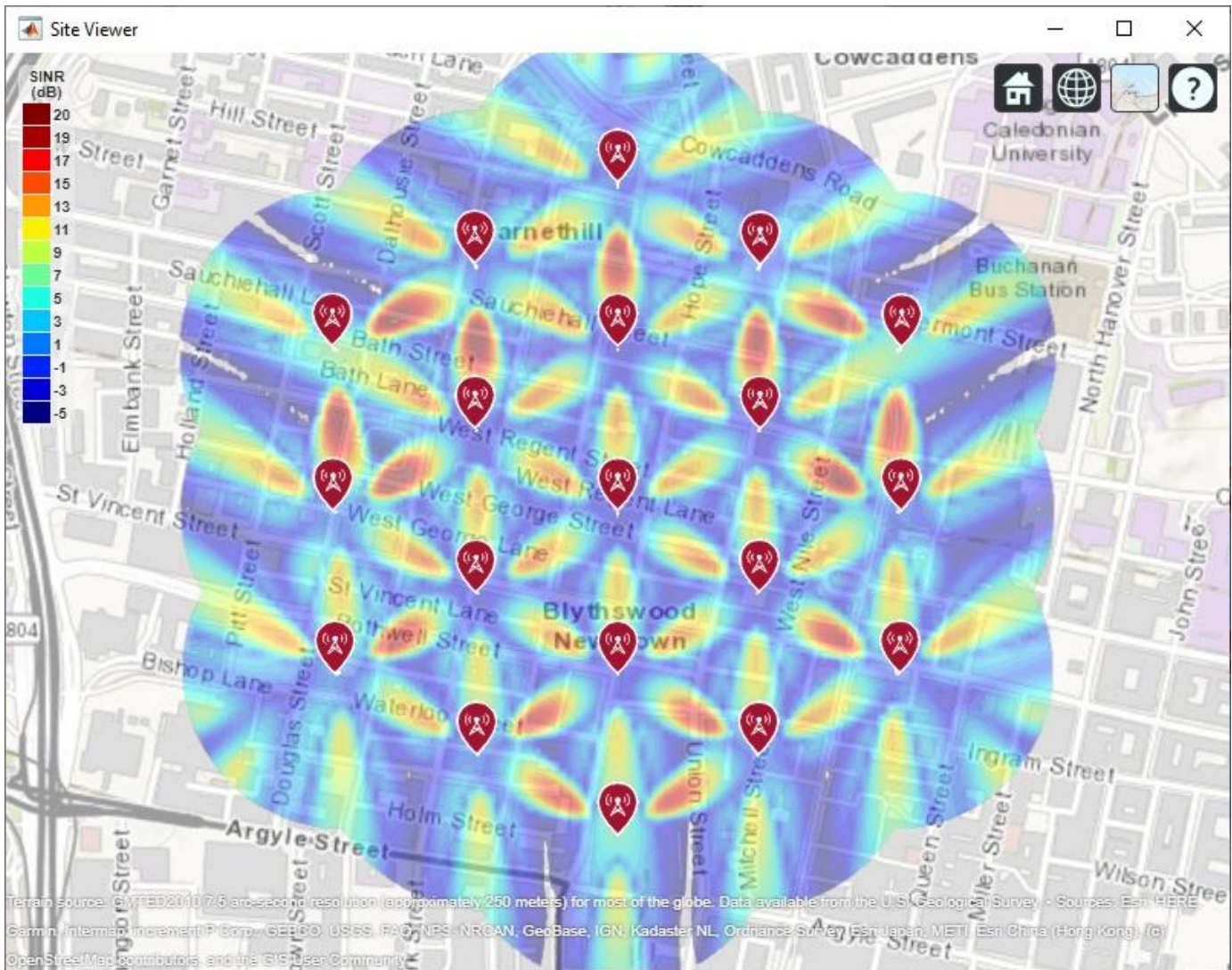


**Display SINR Map Using Close-In Propagation Model**

Visualize SINR for the test scenario using the Close-In propagation model [3 on page 18-353], which models path loss for 5G urban micro-cell and macro-cell scenarios. This model produces an SINR map that shows reduced interference effects compared to the free space propagation model.

```
sinr(txs,'close-in', ...
        'ReceiverGain',rxGain, ...
        'ReceiverAntennaHeight',rxAntennaHeight, ...
        'ReceiverNoisePower',rxNoisePower, ...
```

```
                'MaxRange',isd, ...
                'Resolution',isd/20)
```
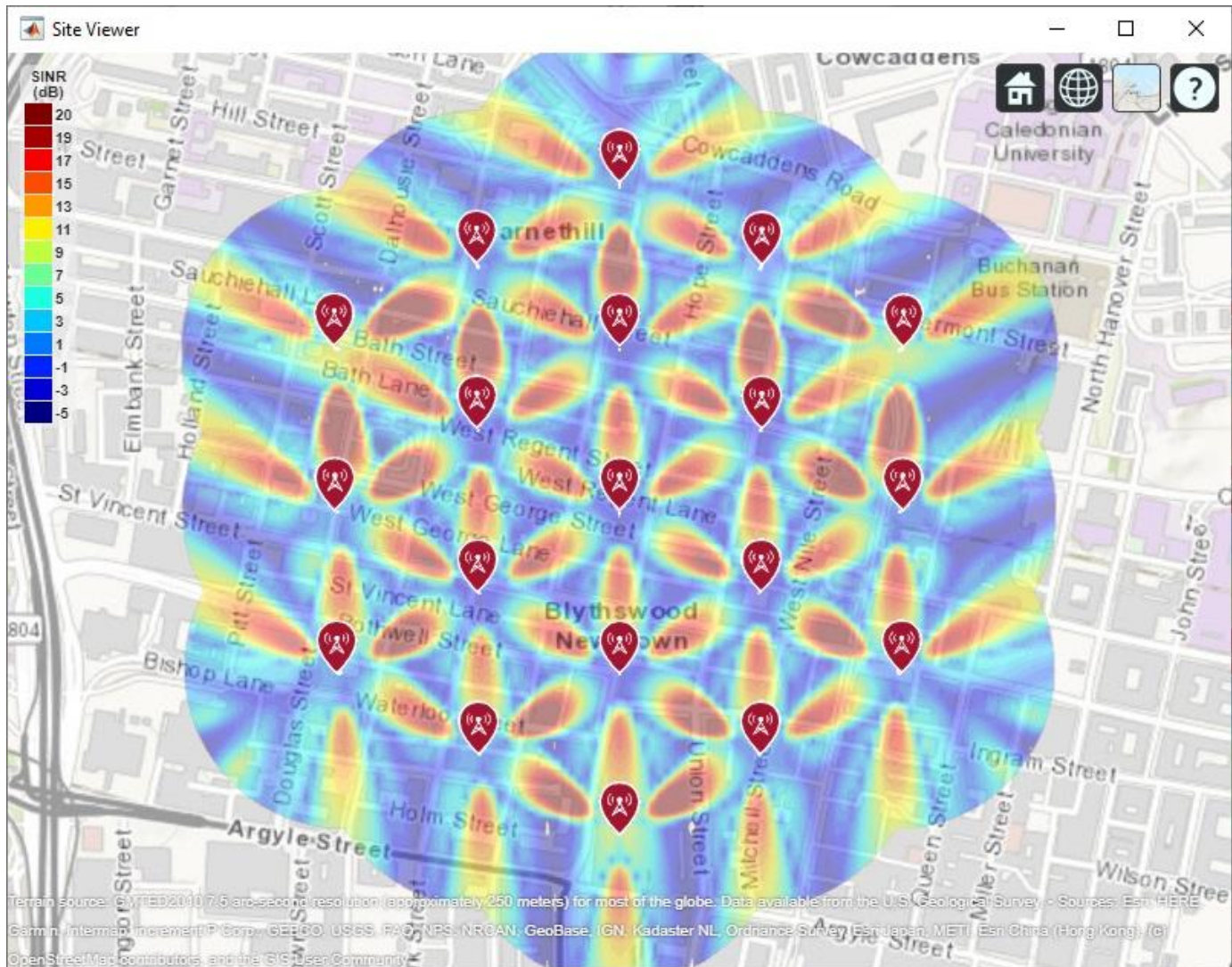


### Use Rectangular Patch Antenna as Array Element

The analysis above used an antenna element that was defined using the equations specified in the ITU-R report [1 on page 18-353]. The antenna element needs to provide a maximum gain of 9.5 dBi and a front-to-back ratio of approximately 30 dB. Now replace the equation-based antenna element definition with a real antenna model using a standard half-wavelength rectangular microstrip patch antenna. The antenna element provides a gain of about 9 dBi, although with a lower front-to-back ratio.

```
% Design half-wavelength rectangular microstrip patch antenna
patchElement = design(patchMicrostrip,fq);
patchElement.Width = patchElement.Length;
patchElement.Tilt = 90;
patchElement.TiltAxis = [0 1 0];

% Display radiation pattern
```

```
f = figure;
pattern(patchElement,fq)
```



```
Output : Directivity
Frequency : 4 GHz
Max value : 9.73 dBi
Min value : -30.1 dBi
Azimuth : [-180° , 180°]
Elevation : [-90° , 90°]
```
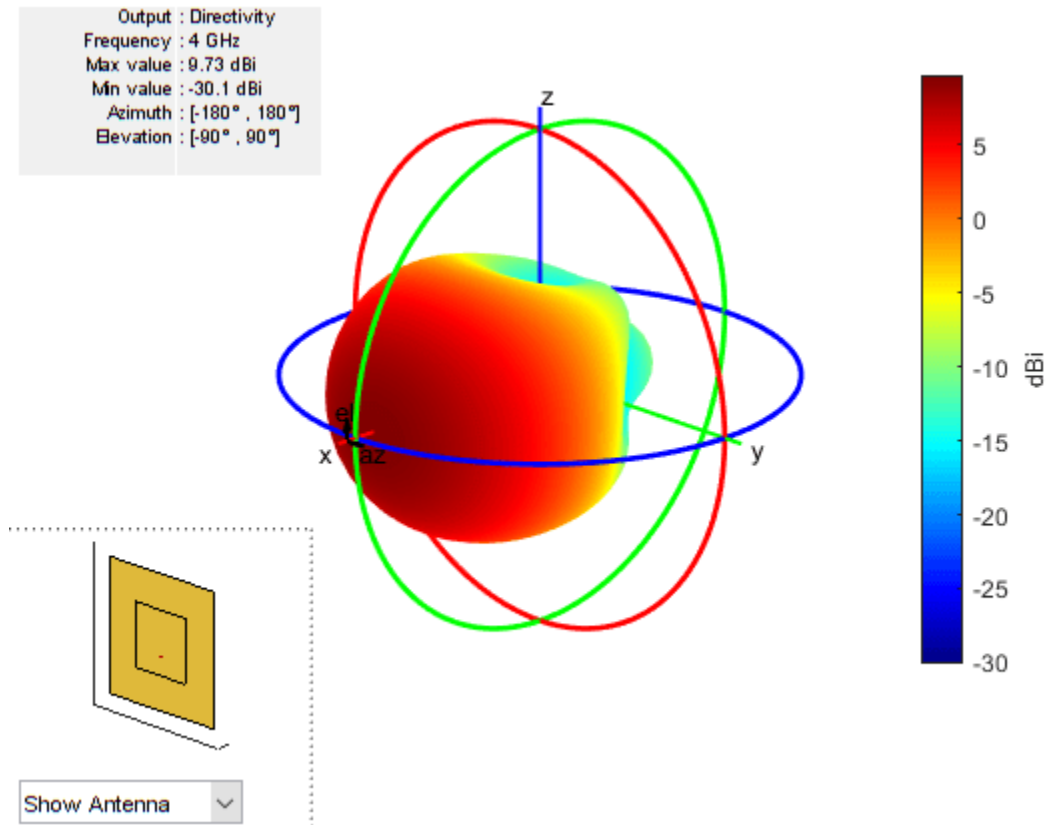
**Display SINR Map Using the Patch Antenna Element in the 8-by-8 Array**

Update the SINR map for the Close-In propagation model [3 on page 18-353] using the patch antenna as the array element. This analysis should capture the effect of deviations from an equation-based antenna specification as per the ITU-R report [1 on page 18-353], including:

- Variations in peak gain
- Variations in pattern symmetry with spatial angles
- Variations in front-to-back ratios

```
% Assign the patch antenna as the array element
cellAntenna.Element = patchElement;

% Display SINR map
if isvalid(f)
    close(f)
end
sinr(txs,'close-in',...
    'ReceiverGain',rxGain, ...
    'ReceiverAntennaHeight',rxAntennaHeight, ...
    'ReceiverNoisePower',rxNoisePower, ...
    'MaxRange',isd, ...
    'Resolution',isd/20)
```
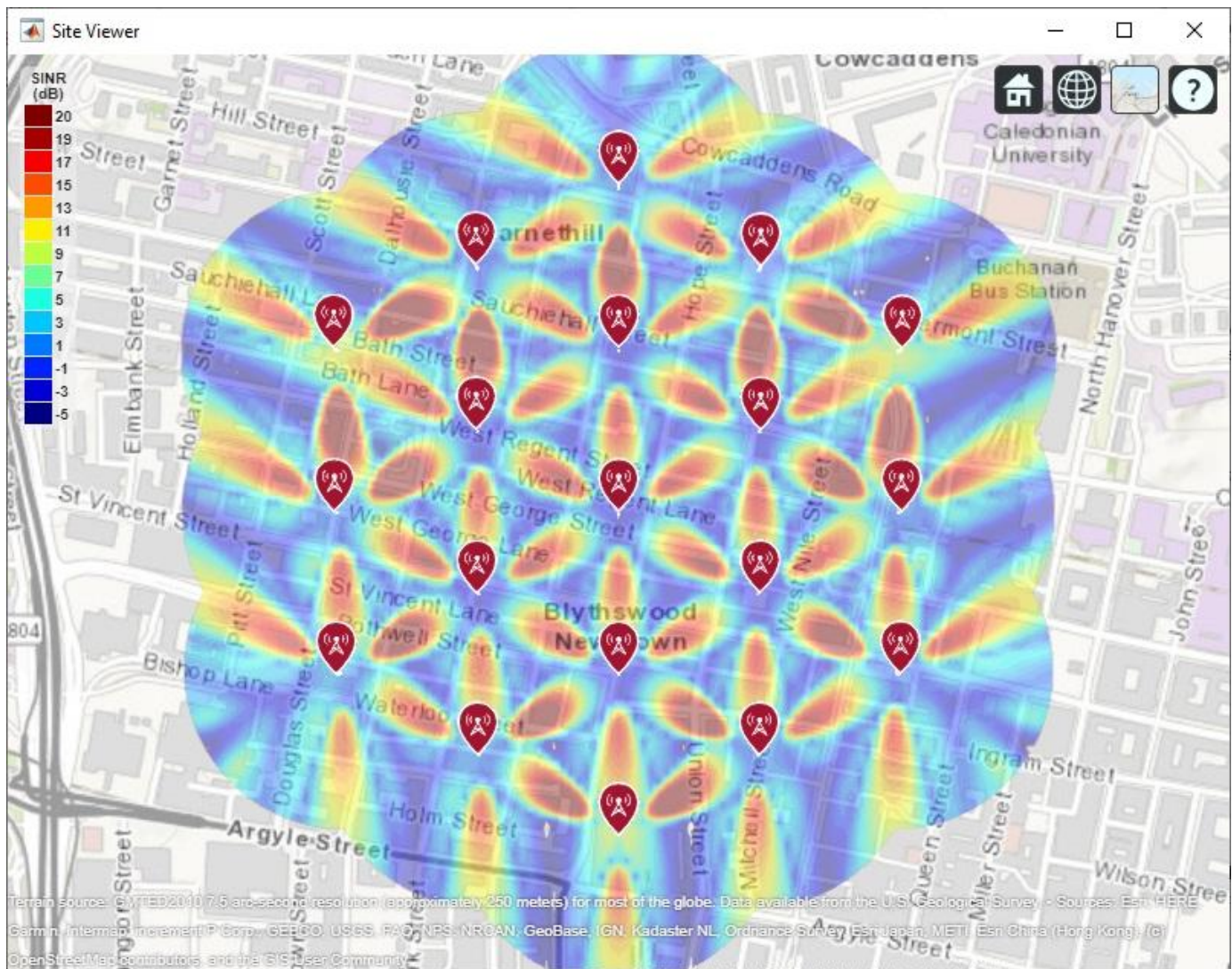
**Summary**

This example shows how to construct a 5G urban macro-cell test environment consisting of a hexagonal network of 19 cell sites, each containing 3 sectored cells. The signal-to-interference-plus-noise ratio (SINR) is visualized on a map for different antennas. The following observations are made:

- A rectangular antenna array can provide greater directionality and therefore peak SINR values than use of a single antenna element.

- The outward-facing lobes on the perimeter of the SINR map represent areas where less interference occurs. A more realistic modelling technique would be to replicate, or wrap around, cell sites to expand the geometry so that perimeter areas experience similar interference as interior areas.

- Using a rectangular antenna array, a propagation model that estimates increased path loss also results in higher SINR values due to less interference.

- Two antenna elements are tried in the antenna array: an equation-based element using Phased Array System Toolbox and a patch antenna element using Antenna Toolbox™. These produce similar SINR maps.

**References**

[1] Report ITU-R M.[IMT-2020.EVAL], "Guidelines for evaluation of radio interface technologies for IMT-2020", 2017. https://www.itu.int/md/R15-SG05-C-0057

[2] Report ITU-R M.2135-1, "Guidelines for evaluation of radio interface technologies for IMT-Advanced", 2009. https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2135-1-2009-PDF-E.pdf

[3] Sun, S.,Rapport, T.S., Thomas, T., Ghosh, A., Nguyen, H., Kovacs, I., Rodriguez, I., Koymen, O.,and Prartyka, A. "Investigation of prediction accuracy, sensitivity, and parameter stability of large-scale propagation path loss models for 5G wireless communications." *IEEE Transactions on Vehicular Technology*, Vol 65, No.5, pp.2843-2860, May 2016.

# Improve SNR and Capacity of Wireless Communication Using Antenna Arrays

The goal of a wireless communication system is to serve as many users with the highest possible data rate given constraints such as radiation power limit and operating budget. To improve the data rate, the key is to improve the signal to noise ratio (SNR). To serve more users, the key is to reuse the resources. Over the last several decades, numerous algorithms have been adopted to improve the SNR and reuse the resources in time, frequency, and coding spaces. This example shows how the adoption of antenna arrays can help improve the SNR and capacity of a wireless link.

**Introduction**

Antenna arrays have become part of the standard configuration in 5G wireless communication systems. Because there are multiple elements in an antenna array, such wireless communications systems are often referred to as multiple input multiple output (MIMO) systems. Antenna arrays can help improve the SNR by exploring the redundancy across the multiple transmit and receive channels. They also make it possible to reuse the spatial information in the system to improve the coverage.

For this example, assume the system is deployed at 60 GHz, which is a frequency being considered for the 5G system.

```
c = 3e8;          % propagation speed
fc = 60e9;        % carrier frequency
lambda = c/fc;    % wavelength
```

```
rng(6466);
```

With no loss in generality, place the transmitter at the origin and place the receiver approximately 1.6 km away.

```
txcenter = [0;0;0];
rxcenter = [1500;500;0];
```

Throughout this example, the `scatteringchanmtx` function will be used to create a channel matrix for different transmit and receive array configurations. The function simulates multiple scatterers between the transmit and receive arrays. The signal travels from the transmit array to all the scatterers first and then bounces off the scatterers and arrives at the receive array. Therefore, each scatterer defines a signal path between the transmit and the receive array and the resulting channel matrix describes a multipath environment. The function works with antenna arrays of arbitrary size at any designated frequency band.

**Improve SNR by Array Gain for Line of Sight Propagation**

The simplest wireless channel is a line of sight (LOS) propagation. Although simple, such channels can often be found in rural areas. Adopting an antenna array under such situations can improve the signal to noise ratio at the receiver and in turn improve the communication link's bit error rate (BER).

**SISO LOS Channel**

Before discussing the performance of a MIMO system, it is useful to build a baseline with a single input single output (SISO) communication system. A SISO LOS channel has a direct path from the transmitter to the receiver. Such a channel can be modeled as a special case of the multipath channel.

```
[~,txang] = rangeangle(rxcenter,txcenter);
[~,rxang] = rangeangle(txcenter,rxcenter);

txsipos = [0;0;0];
rxsopos = [0;0;0];

g = 1;  % gain for the path
sisochan = scatteringchanmtx(txsipos,rxsopos,txang,rxang,g);
```

Using BPSK modulation, the bit error rate (BER) for such a SISO channel can be plotted as

```
Nsamp = 1e6;
x = randi([0 1],Nsamp,1);

ebn0_param = -10:2:10;
Nsnr = numel(ebn0_param);

ber_siso = helperMIMOBER(sisochan,x,ebn0_param)/Nsamp;
helperBERPlot(ebn0_param,ber_siso);
legend('SISO')
```
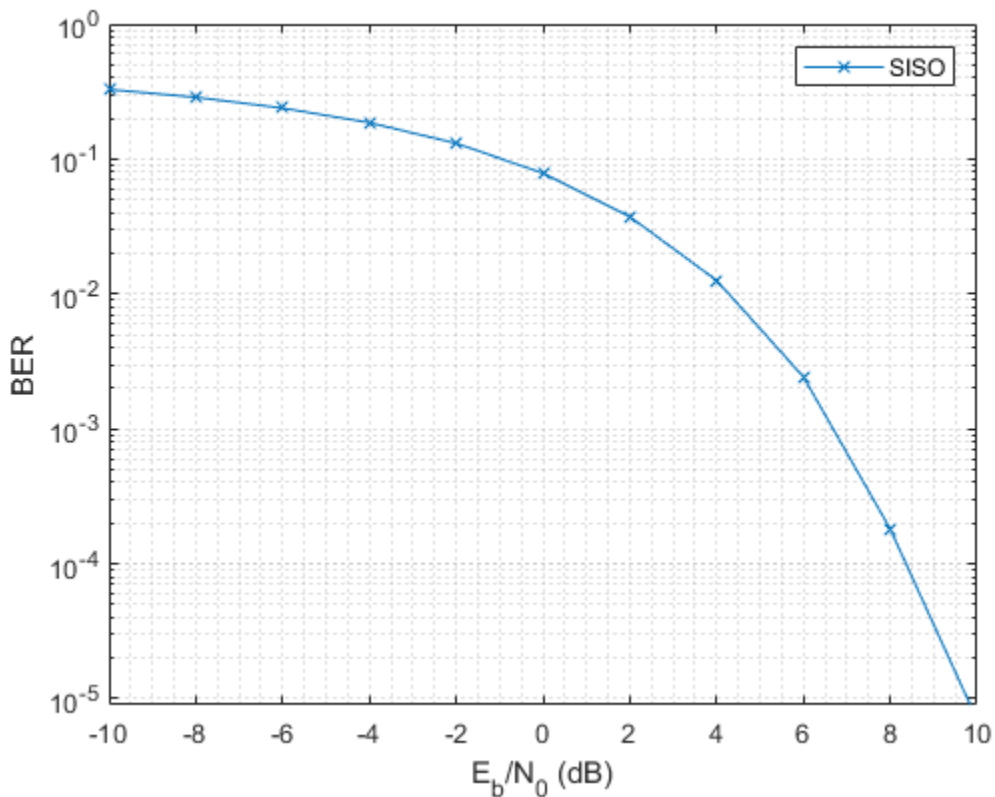


**SIMO LOS Channel**

With the baseline established for a SISO system, this section focuses on the single input multiple output (SIMO) system. In such a system, there is one transmit antenna but multiple receive antennas. Again, it is assumed that there is a direct path between the transmitter and the receiver.

Assume the receive array is a 4-element ULA with half-wavelength spacing, then the SIMO channel can be modeled as

```
rxarray = phased.ULA('NumElements',4,'ElementSpacing',lambda/2);
rxmopos = getElementPosition(rxarray)/lambda;

simochan = scatteringchanmtx(txsipos,rxmopos,txang,rxang,g);
```

In the SIMO system, because the received signals across receive array elements are coherent, it is possible to steer the receive array toward the transmitter to improve the SNR. Note that this assumes that the signal incoming direction is known to the receiver. In reality, the angle is often obtained using direction of arrival estimation algorithms.

```
rxarraystv = phased.SteeringVector('SensorArray',rxarray,...
    'PropagationSpeed',c);
wr = conj(rxarraystv(fc,rxang));
ber_simo = helperMIMOBER(simochan,x,ebn0_param,1,wr)/Nsamp;
helperBERPlot(ebn0_param,[ber_siso(:) ber_simo(:)]);
legend('SISO','SIMO')
```
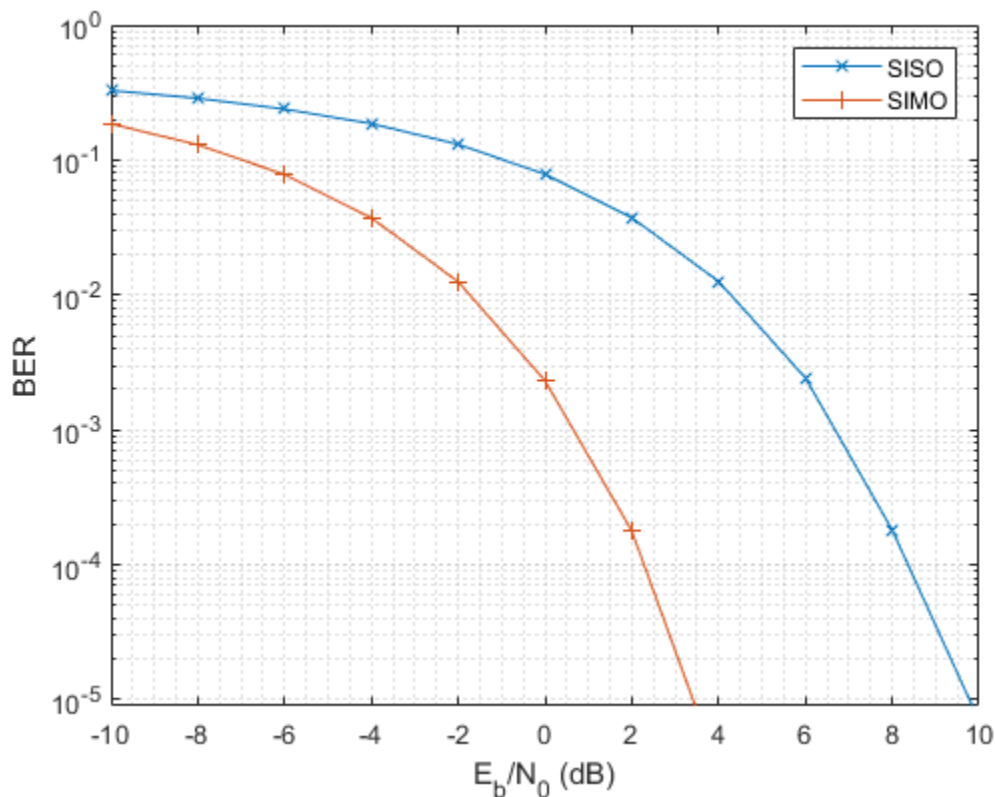


The BER curve shows a gain of 6 dB provided by the receive array.

**MISO LOS Channel**

The multiple input single output (MISO) system works in a similar way. In this case, the transmitter is a 4-element ULA with half-wavelength spacing.

```
txarray = phased.ULA('NumElements',4,'ElementSpacing',lambda/2);
txmipos = getElementPosition(txarray)/lambda;

misochan = scatteringchanmtx(txmipos,rxsopos,txang,rxang,g);
```

A line of sight MISO system achieves best SNR when the transmitter has the knowledge of the receiver and steers the beam toward the receiver. In addition, to do a fair comparison with the SISO system, the total transmitter power should be the same under both situations.

```
txarraystv = phased.SteeringVector('SensorArray',txarray,...
    'PropagationSpeed',c);
wt = txarraystv(fc,txang)';
ber_miso = helperMIMOBER(misochan,x,ebn0_param,wt,1)/Nsamp;
helperBERPlot(ebn0_param,[ber_siso(:) ber_simo(:) ber_miso(:)]);
legend('SISO','SIMO','MISO')
```
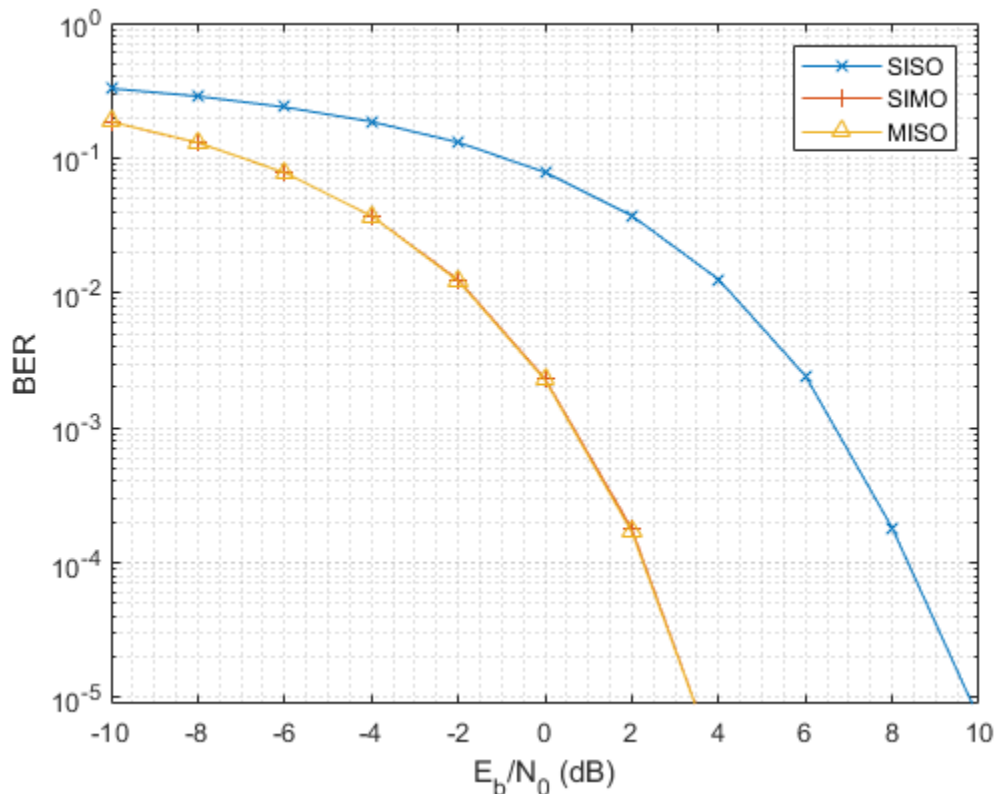


Note that with the pre-steering, the performance of MISO matches the performance of a SIMO system, gaining 6 dB in SNR. It may not be as intuitive compared to the SIMO case because the total transmit power does not increase. However, by replacing a single isotropic antenna with a 4-element transmit array, a 6 dB gain is achieved.

**MIMO LOS Channel**

Because a SIMO system provides an array gain from the received array and a MISO system provides an array gain from the transmit array, a MIMO system with an LOS propagation can benefit from both the transmit and receive array gain.

Assume a MIMO system with a 4-element transmit array and a 4-element receive array.

```
mimochan = scatteringchanmtx(txmipos,rxmopos,txang,rxang,g);
```

To achieve the best SNR, the transmit array and the receive array need to be steered toward each other. With this configuration, the BER curve can be computed as

```
wt = txarraystv(fc,txang)';
wr = conj(rxarraystv(fc,rxang));
ber_mimo = helperMIMOBER(mimochan,x,ebn0_param,wt,wr)/Nsamp;
helperBERPlot(ebn0_param,[ber_siso(:) ber_simo(:) ber_mimo(:)]);
legend('SISO','SIMO','MIMO')
```
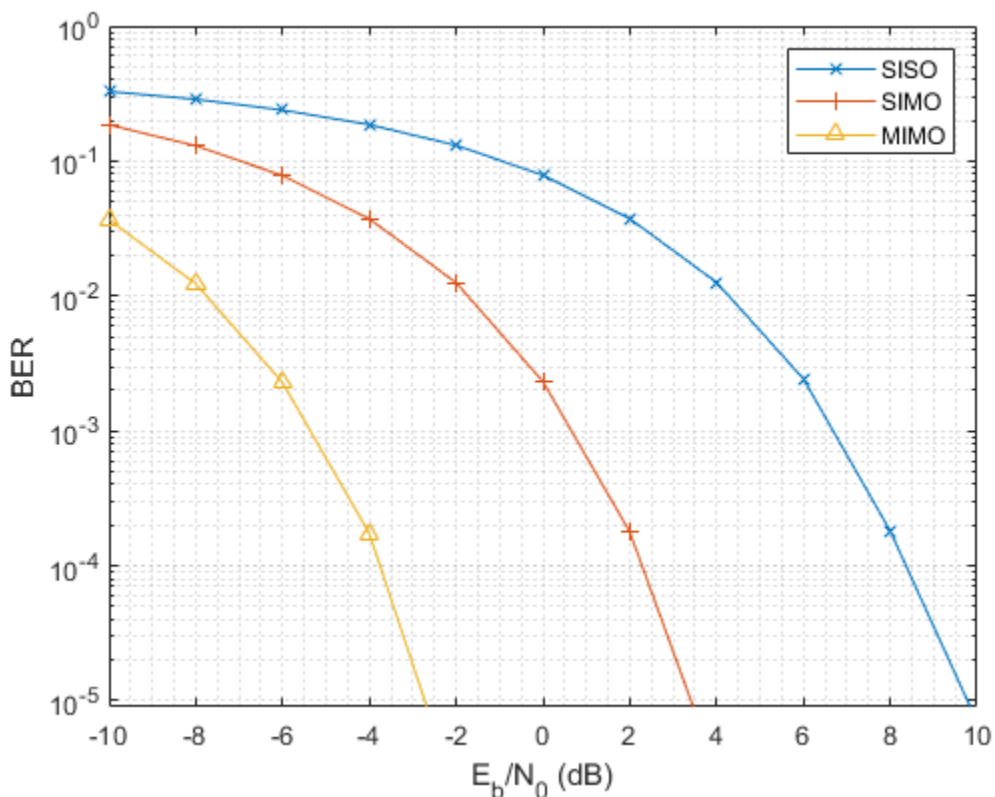


As expected, the BER curve shows that both the transmit array and the receive array contributes a 6 dB array gain, resulting in a total gain of 12 dB over the SISO case.

### Improve SNR by Diversity Gain for Multipath Channel

All the channels in the previous sections are line-of-sight channels. Although such channels are found in some wireless communication systems, in general wireless communications occurs in multipath fading environments. The rest of this example explores how using arrays can help in a multipath environment.

### SISO Multipath Channel

Assume there are 10 randomly placed scatterers in the channel, then there will be 10 paths from the transmitter to the receiver, as illustrated in the following figure.

```
Nscat = 10;

[~,~,~,scatpos] = ...
    helperComputeRandomScatterer(txcenter,rxcenter,Nscat);
helperPlotSpatialMIMOScene(txsipos,rxsopos,...
    txcenter,rxcenter,scatpos);
```
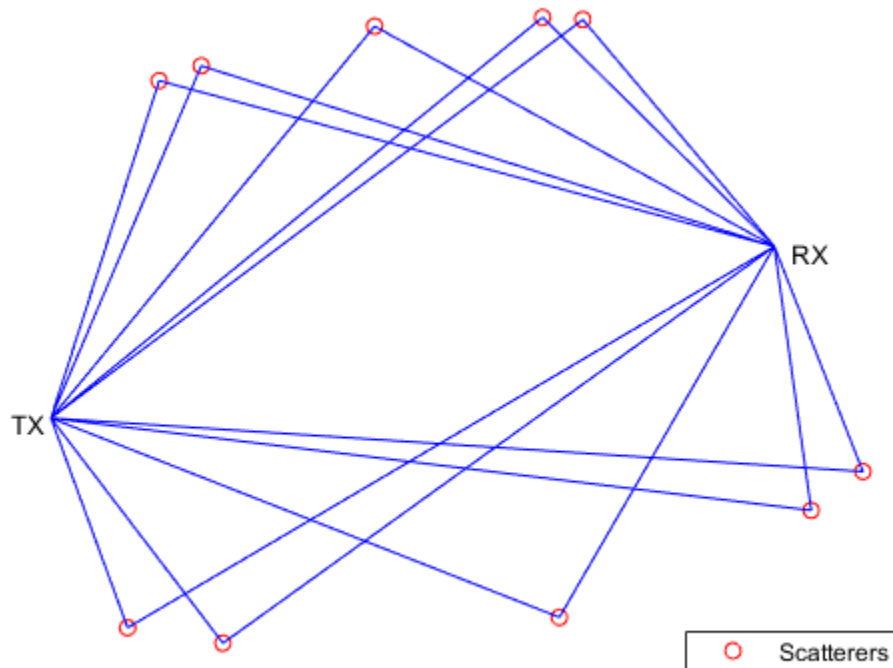


For simplicity, assume that signals traveling along all paths arrive within the same symbol period so the channel is frequency flat.

To simulate the BER curve for a fading channel, the channel needs to change over time. Assume we have 1000 frames and each frame has 10000 bits. The baseline SISO multipath channel BER curve is constructed as

```
Nframe = 1e3;
Nbitperframe = 1e4;
Nsamp = Nframe*Nbitperframe;

x = randi([0 1],Nbitperframe,1);

nerr = zeros(1,Nsnr);

for m = 1:Nframe
    sisompchan = scatteringchanmtx(txsipos,rxsopos,Nscat);
    wr = sisompchan'/norm(sisompchan);
    nerr = nerr+helperMIMOBER(sisompchan,x,ebn0_param,1,wr);
end
```

```
ber_sisomp = nerr/Nsamp;
helperBERPlot(ebn0_param,[ber_siso(:) ber_sisomp(:)]);
legend('SISO LOS','SISO Multipath');
```
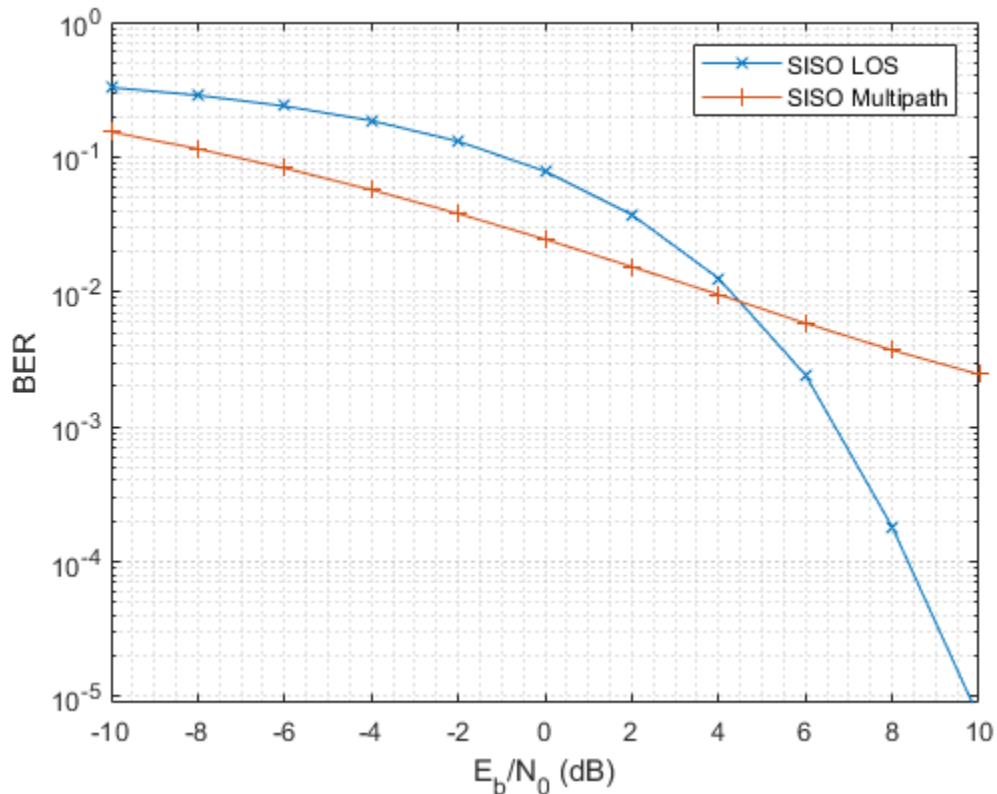


Compared to the BER curve derived from an LOS channel, the BER falls off much slower with the increase of energy per bit to noise power spectral density ratio (Eb/N0) due to the fading caused by the multipath propagation.

**SIMO Multipath Channel**

As more receive antennas are used in the receive array, more copies of the received signals are available at the receiver. Again, assume a 4-element ULA at the receiver.

The optimal combining weights can be derived by matching the channel response. Such a combining scheme is often termed as maximum ratio combining (MRC). Although in theory such scheme requires the knowledge of the channel, in practice the channel response can often be estimated at the receive array.

```
nerr = zeros(1,Nsnr);

for m = 1:Nframe
    simompchan = scatteringchanmtx(txsipos,rxmopos,Nscat);
    wr = simompchan'/norm(simompchan);
    nerr = nerr+helperMIMOBER(simompchan,x,ebn0_param,1,wr);
end
ber_simomp = nerr/Nsamp;
```

```
helperBERPlot(ebn0_param,[ber_sisomp(:) ber_simomp(:)]);
legend('SISO Multipath','SIMO Multipath');
```
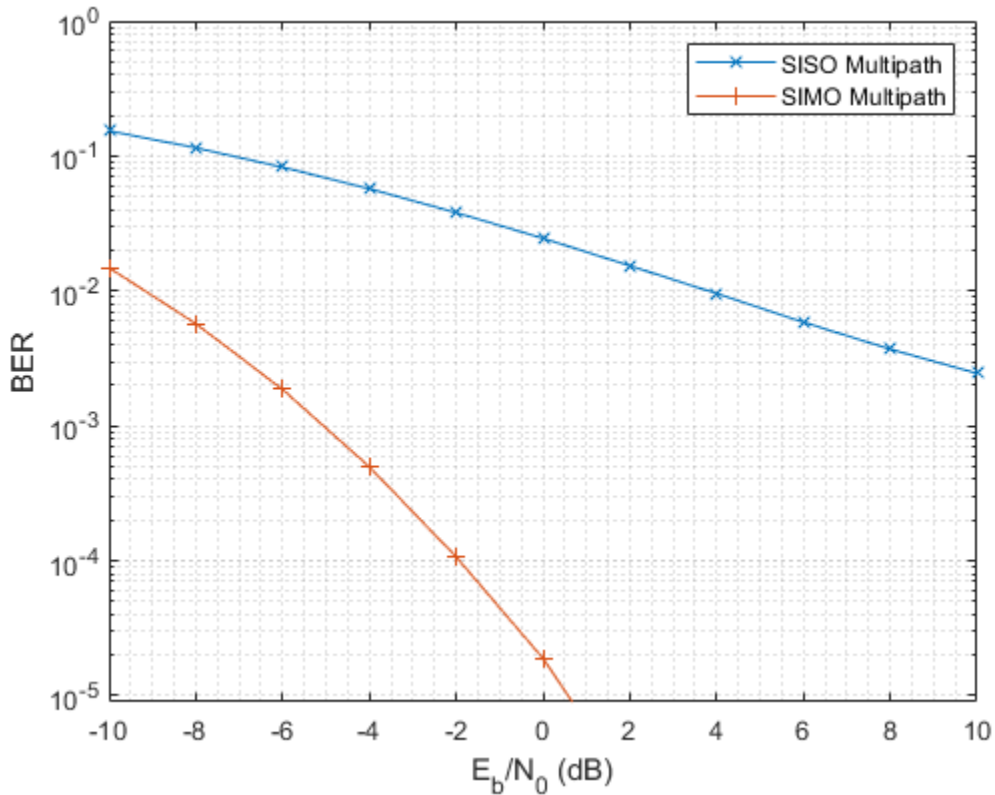


Note that the received signal is no longer weighted by a steering vector toward a specific direction. Instead, the receiving array weights in this case are given by the complex conjugate of the channel response. Otherwise it is possible that multipath could make the received signal out of phase with the transmitted signal. This assumes that the channel response is known to the receiver. If the channel response is unknown, pilot signals can be used to estimate the channel response.

It can be seen from the BER curve that not only the SIMO system provides some SNR gains compared to the SISO system, but the slope of the BER curve of the SIMO system is also steeper compared to the BER curve of the SISO system. The gain resulted from the slope change is often referred to as the diversity gain.

**MISO Multipath Channel**

Things get more interesting when there is multipath propagation in a MISO system. First, if the channel is known to the transmitter, then the strategy to improve the SNR is similar to maximum ratio combining. The signal radiated from each element of the transmit array should be weighted so that the propagated signal can be added coherently at the receiver.

```
nerr = zeros(1,Nsnr);

for m = 1:Nframe
    misompchan = scatteringchanmtx(txmipos,rxsopos,Nscat);
    wt = misompchan'/norm(misompchan);
    nerr = nerr+helperMIMOBER(misompchan,x,ebn0_param,wt,1);
```

```
end
ber_misomp = nerr/Nsamp;

helperBERPlot(ebn0_param,[ber_sisomp(:) ber_simomp(:) ber_misomp(:)]);
legend('SISO Multipath','SIMO Multipath','MISO Multipath');
```
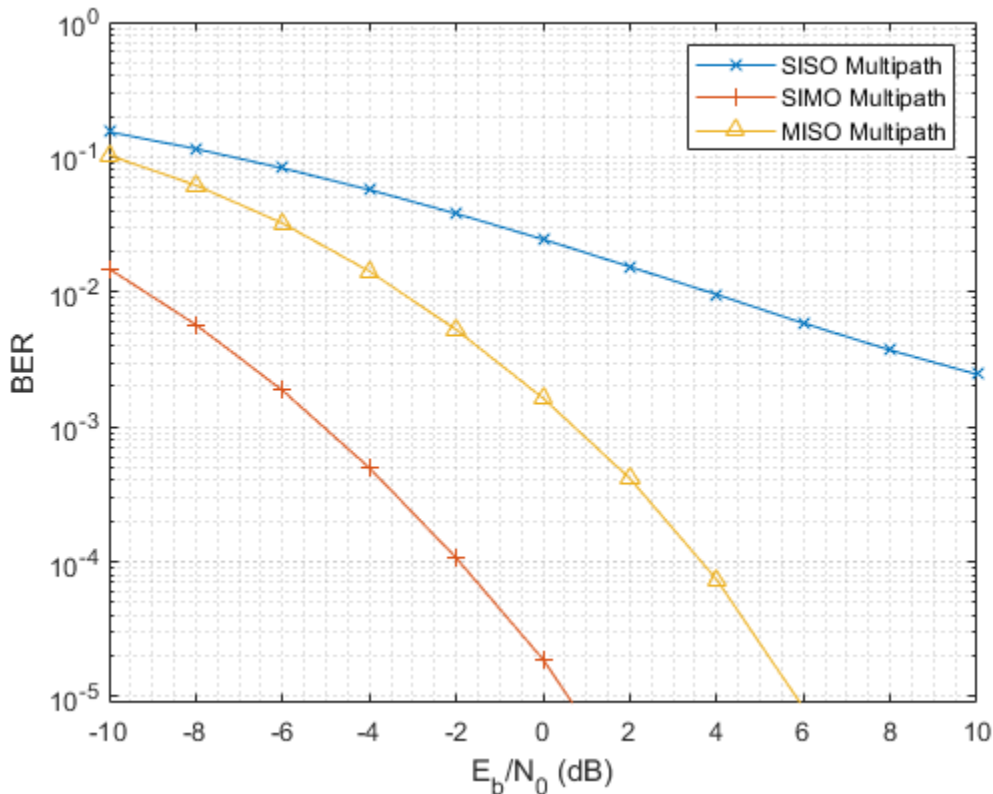


Note the transmit diversity gain shown in the BER curve. Compared to the SIMO multipath channel case, the performance of a MISO multipath system is not as good. This is because there is only one copy of the received signal yet the transmit power gets spread among multiple paths. It is certainly possible to amplify the signal at the transmit side to achieve an equivalent gain, but that introduces additional cost.

If the channel is not known to the transmitter, there are still ways to explore the diversity via space time coding. For example, Alamouti code is a well known coding scheme that can be used to achieve diversity gain when the channel is not known. Interested readers are encouraged to explore the Introduction to MIMO Systems example in Communications Toolbox™.

**MIMO Multipath Channel**

The rest of this example focuses on a multipath MIMO channel. In particular, this section illustrates the case where the number of scatterers in the environment is larger than the number of elements in the transmit and receive arrays. Such an environment is often termed as a rich scattering environment.

Before diving into the specific performance measures, it is helpful to get a quick illustration of what the channel looks like. The following helper function creates a 4x4 MIMO channel where both transmitter and receiver are 4-element ULAs.

```
[txang,rxang,scatg,scatpos] = ...
    helperComputeRandomScatterer(txcenter,rxcenter,Nscat);
mimompchan = scatteringchanmtx(txmipos,rxmopos,txang,rxang,scatg);
```

There are multiple paths available between the transmit array and the receive array because of the existence of the scatterers. Each path consists of a single bounce off the corresponding scatterer.

```
helperPlotSpatialMIMOScene(txmipos,rxmopos,txcenter,rxcenter,scatpos);
```
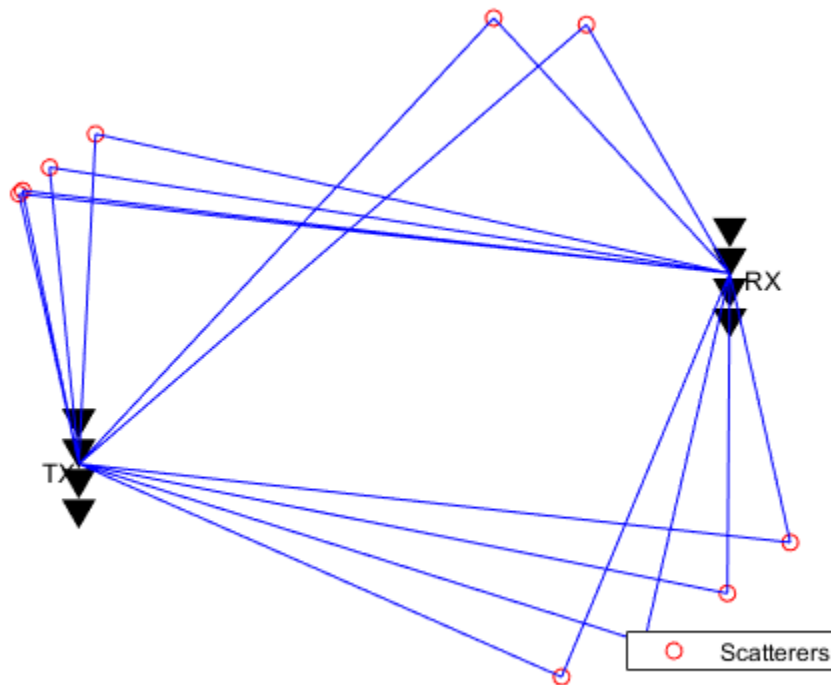


There are two ways to take advantage of a MIMO channel. The first way is to explore the diversity gain offered by a MIMO channel. Assuming the channel is known, the following figure shows the diversity gain with the BER curve.

```
nerr = zeros(1,Nsnr);

for m = 1:Nframe
    mimompchan = scatteringchanmtx(txmipos,rxmopos,Nscat);
    [u,s,v] = svd(mimompchan);
    wt = u(:,1)';
    wr = v(:,1);
    nerr = nerr+helperMIMOBER(mimompchan,x,ebn0_param,wt,wr);
end
ber_mimomp = nerr/Nsamp;

helperBERPlot(ebn0_param,[ber_sisomp(:) ber_simomp(:) ber_mimomp(:)]);
legend('SISO Multipath','SIMO Multipath','MIMO Multipath');
```
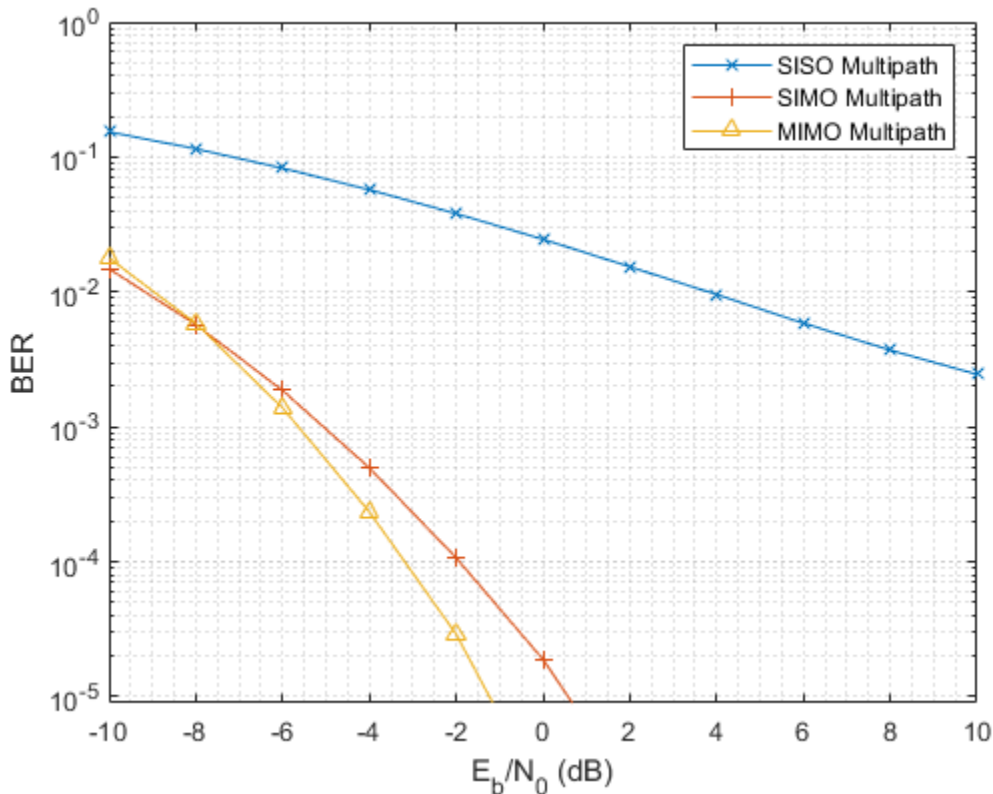
Compare the BER curve from a MIMO channel with the BER curve obtained from a SIMO system. In the multipath case, the diversity gain from a MIMO channel is not necessarily better than the diversity gain provided by a SIMO channel. This is because to obtain the best diversity gain, only the dominant mode in a MIMO channel is used yet there are other modes in the channel that are not used. So is there an alternative way to utilize the channel?

**Improve Capacity by Spatial Multiplexing for MIMO Multipath Channel**

The answer to the previous question lies in a scheme called spatial multiplexing. The idea behind spatial multiplexing is that a MIMO multipath channel with a rich scatterer environment can send multiple data streams simultaneously across the channel. For example, the channel matrix of a 4x4 MIMO channel becomes full rank because of the scatterers. This means that it is possible to send as many as 4 data streams at once. The goal of spatial multiplexing is less about increasing the SNR but more about increasing the information throughput.

The idea of spatial multiplexing is to separate the channel matrix to multiple mode so that the data stream sent from different elements in the transmit array can be independently recovered from the received signal. To achieve this, the data stream is precoded before the transmission and then combined after the reception. The precoding and combining weights can be computed from the channel matrix by

```
[wp,wc] = diagbfweights(mimompchan);
```

To see why the combination of the precoding and combining weights can help transmit multiple data streams at the same time, examine the product of the weights and the channel matrix.

```
wp*mimompchan*wc
```

```
ans =

  10.3543 + 0.0000i   -0.0000 + 0.0000i    0.0000 - 0.0000i   -0.0000 - 0.0000i
   0.0000 + 0.0000i    6.0693 + 0.0000i    0.0000 + 0.0000i    0.0000 - 0.0000i
  -0.0000 + 0.0000i   -0.0000 + 0.0000i    2.4446 + 0.0000i   -0.0000 + 0.0000i
   0.0000 + 0.0000i    0.0000 + 0.0000i    0.0000 + 0.0000i    1.1049 - 0.0000i
```

Note that the product is a diagonal matrix, which means that the information received by each receive array element is simply a scaled version of the transmit array element. So it behaves like multiple orthogonal subchannels within the original channel. The first subchannel corresponds to the dominant transmit and receive directions so there is no loss in the diversity gain. In addition, it is now possible to use other subchannels to carry information too, as shown in the BER curve for the first two subchannels.

```
Ntx = 4;
Nrx = 4;

x = randi([0 1],Nbitperframe,Ntx);

nerr = zeros(Nrx,Nsnr);

for m = 1:Nframe
    mimompchan = scatteringchanmtx(txmipos,rxmopos,Nscat);
    [wp,wc] = diagbfweights(mimompchan);
    nerr = nerr+helperMIMOMultistreamBER(mimompchan,x,ebn0_param,wp,wc);
end
ber_mimompdiag = nerr/Nsamp;
helperBERPlot(ebn0_param,[ber_sisomp(:) ber_mimomp(:)...
    ber_mimompdiag(1,:).' ber_mimompdiag(2,:).']);
legend('SISO LOS','MIMO Multipath','MIMO Multipath Stream 1',...
    'MIMO Multipath Stream 2');
```
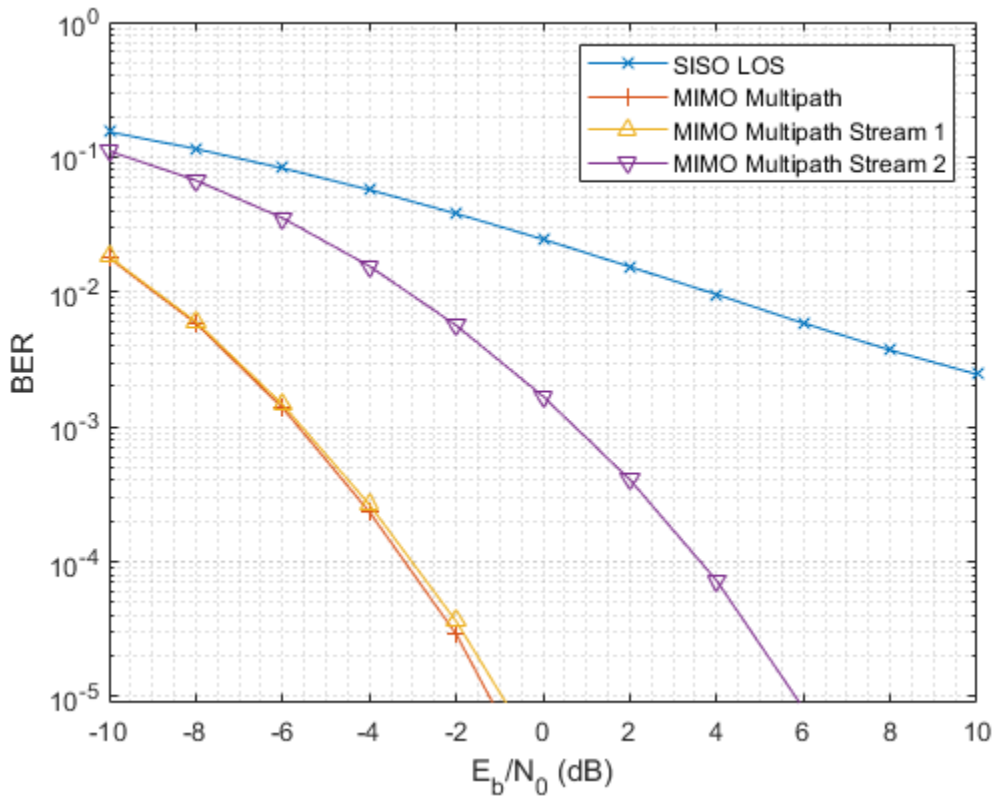
Although the second stream cannot provide a gain as high as the first stream as it uses a less dominant subchannel, the overall information throughput is improved. Therefore, next section measures the performance by the channel capacity instead of the BER curve.

The most intuitive way to transmit data in a MIMO system is to uniformly split the power among transmit elements. However, the capacity of the channel can be further improved if the channel is known at the transmitter. In this case, the transmitter could use the waterfill algorithm to make the choice of transmitting only in the subchannels where a satisfying SNR can be obtained. The following figure shows the comparison of the system capacity between the two power distribution schemes. The result confirms that the waterfill algorithm provides a better system capacity compared to the uniform power distribution. The difference gets smaller when the system level SNR improves.

```
C_mimo_cu = zeros(1,Nsnr);
C_mimo_ck = zeros(1,Nsnr);
Ntrial = 1000;
for m = 1:Nsnr
    for n = 1:Ntrial
        mimompchan = scatteringchanmtx(txmipos,rxmopos,Nscat);
        N0 = db2pow(-ebn0_param(m));
        [~,~,~,~,cu] = diagbfweights(mimompchan,1,N0,'uniform');
        [~,~,~,~,ck] = diagbfweights(mimompchan,1,N0,'waterfill');
        C_mimo_cu(m) = C_mimo_cu(m)+cu;
        C_mimo_ck(m) = C_mimo_ck(m)+ck;
    end
end
C_mimo_cu = C_mimo_cu/Ntrial;
C_mimo_ck = C_mimo_ck/Ntrial;
```

```
plot(ebn0_param,C_mimo_cu(:),'-*',ebn0_param,C_mimo_ck(:),'-^');
xlabel('SNR (dB)');
ylabel('Capacity (bps/Hz)');
legend('Uniform Power Distribution','Waterfill Power Distribution');
grid on;
```
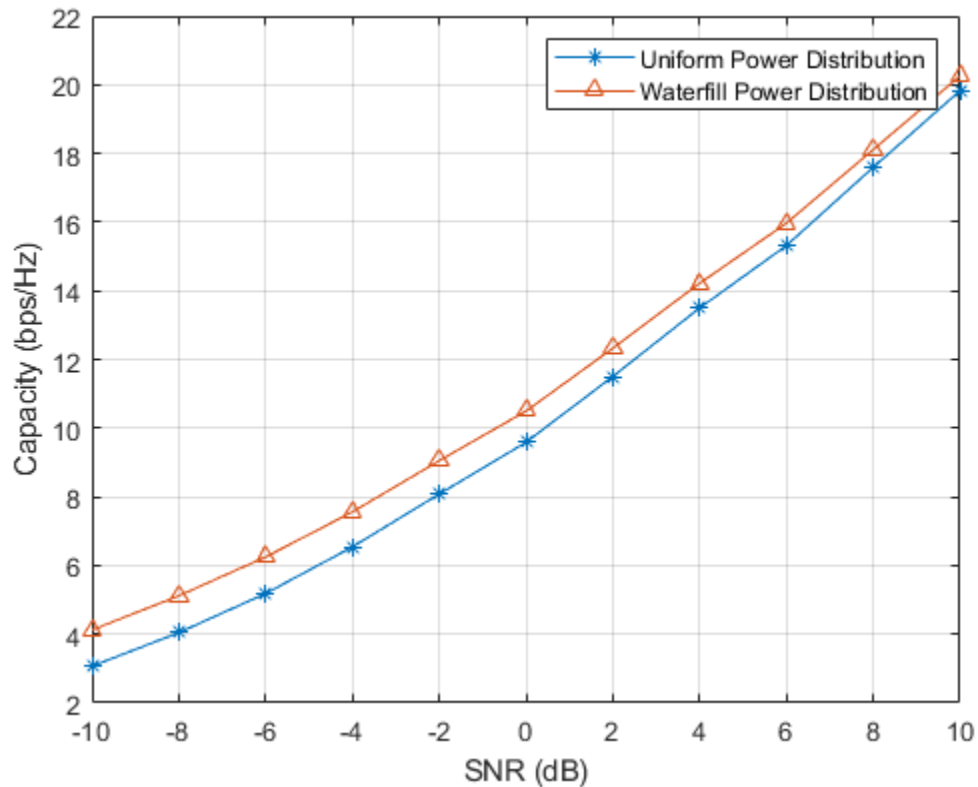


For more details on spatial multiplexing and its detection techniques, refer to the Spatial Multiplexing example in Communications Toolbox.

**From Beamforming to Precoding**

Finally, it is worth looking at how these different ways of using arrays relate to each other. Starting from the LOS channel, as mentioned in the previous sections, the benefit provided by the array is an improvement in the SNR.

```
[~,txang] = rangeangle(rxcenter,txcenter);
[~,rxang] = rangeangle(txcenter,rxcenter);

mimochan = scatteringchanmtx(txmipos,rxmopos,txang,rxang,1);

wt = txarraystv(fc,txang)';
wr = conj(rxarraystv(fc,rxang));

helperPlotSpatialMIMOScene(txmipos,rxmopos,txcenter,rxcenter,...
    (txcenter+rxcenter)/2,wt,wr)
```
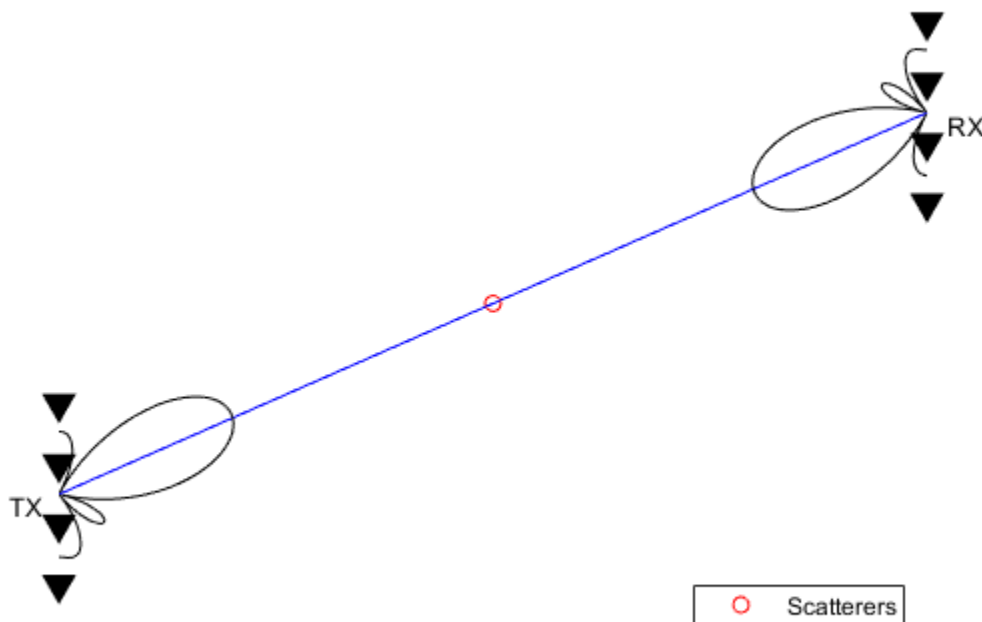
It is clear from the sketch that in this case, the transmit and receive weights form two beams that point to each other. Thus, the array gain is achieved by the beamforming technique. On the other hand, if one tries to create a similar sketch for a MIMO channel, it looks like the following figure.

```
[txang,rxang,scatg,scatpos] = ...
    helperComputeRandomScatterer(txcenter,rxcenter,Nscat);
mimompchan = scatteringchanmtx(txmipos,rxmopos,txang,rxang,scatg);

[wp,wc] = diagbfweights(mimompchan);
helperPlotSpatialMIMOScene(txmipos,rxmopos,txcenter,rxcenter,...
    scatpos,wp(1,:),wc(:,1))
```
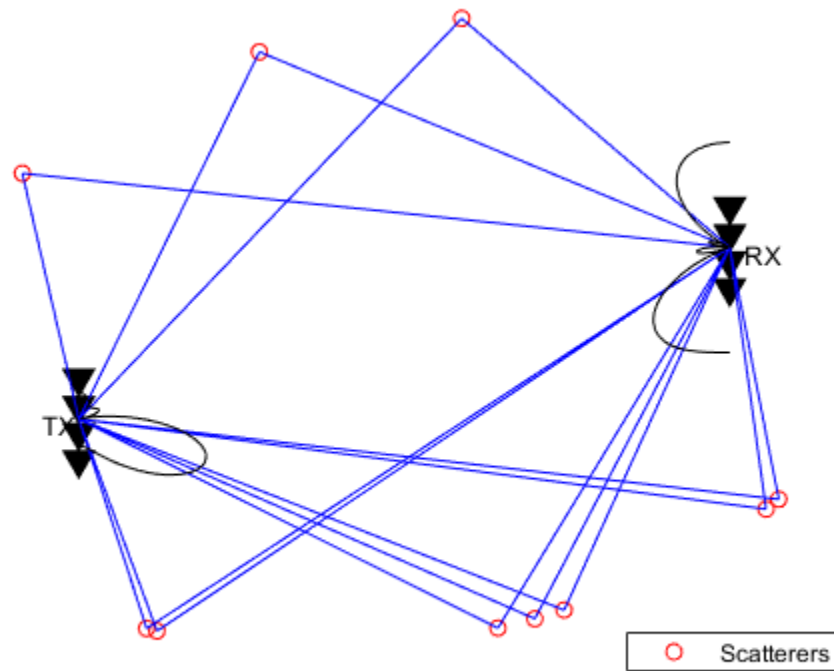
Note that the figure only depicts the pattern for the first data stream but nevertheless it is clear that the pattern no longer necessarily has a dominant main beam. However, if the number of scatterers is reduced to one, then the scene becomes

```
[txang,rxang,scatg,scatpos] = ...
    helperComputeRandomScatterer(txcenter,rxcenter,1);
mimompchan = scatteringchanmtx(txmipos,rxmopos,txang,rxang,scatg);

[wp,wc] = diagbfweights(mimompchan);
helperPlotSpatialMIMOScene(txmipos,rxmopos,txcenter,rxcenter,...
    scatpos,wp(1,:),wc(:,1))
```
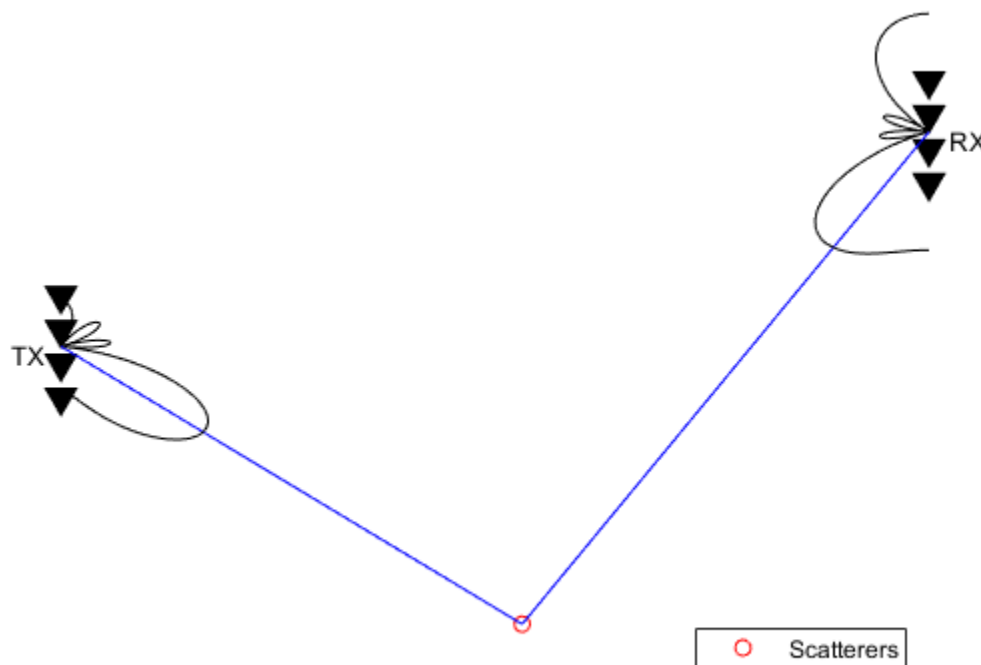
Therefore, the LOS channel case, or more precisely, the one scatterer case, can be considered as a special case of the precoding. When there is only one path available between the transmit and receive arrays, the precoding degenerates to a beamforming scheme.

**Summary**

This example explains how array processing can be used to improve the quality of a MIMO wireless communication system. Depending on the nature of the channel, the arrays can be used to either improve the SNR via array gain or diversity gain, or improve the capacity via spatial multiplexing. The example also shows how to use functions like `scatteringchanmtx` and `diagbfweights` to simulate those scenarios. For more information on MIMO systems modeling, interested readers may refer to examples provided in Communications Toolbox.

**Reference**

[1] David Tse and Pramod Viswanath, Fundamentals of Wireless Communications, Cambridge, 2005

[2] Arogyswami Paulraj, Introduction to Space-Time Wireless Communication, Cambridge, 2003

# Introduction to Hybrid Beamforming

This example introduces the basic concept of hybrid beamforming and shows how to simulate such a system.

**Introduction**

Modern wireless communication systems use spatial multiplexing to improve the data throughput within the system in a scatterer rich environment. In order to send multiple data streams through the channel, a set of precoding and combining weights are derived from the channel matrix. Then each data stream can be independently recovered. Those weights contain both magnitude and phase terms and are normally applied in the digital domain. One example of simulating such a system can be found in the "Improve SNR and Capacity of Wireless Communication Using Antenna Arrays" on page 18-354 example. In the system diagram shown below, each antenna is connected to a unique transmit and receive (TR) module.
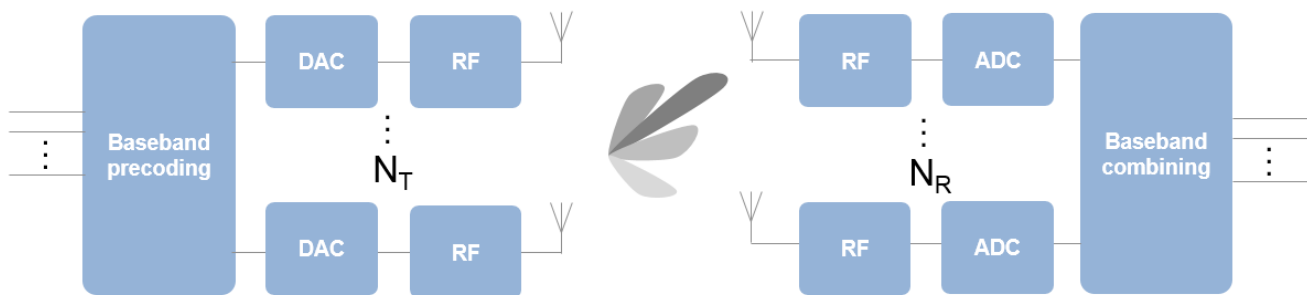


The ever growing demand for high data rate and more user capacity increases the need to use the spectrum more efficiently. As a result, the next generation, 5G, wireless systems will use millimeter wave (mmWave) band to take advantage of its wider bandwidth. In addition, 5G systems deploy large scale antenna arrays to mitigate severe propagation loss in the mmWave band. However, these configurations bring their unique technical challenges.

Compared to current wireless systems, the wavelength in the mmWave band is much smaller. Although this allows an array to contain more elements with the same physical dimension, it becomes much more expensive to provide one TR module for each antenna element. Hence, as a compromise, a TR switch is often used to supply multiple antenna elements. This is the same concept as the subarray configuration used in the radar community. One such configuration is shown in the following figure.

phase shifter
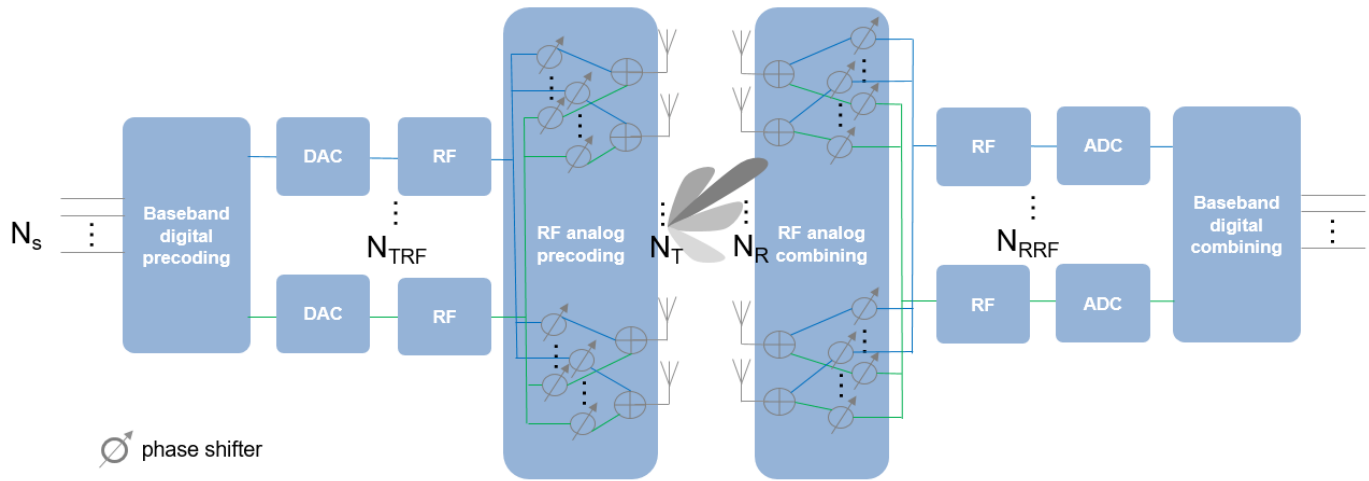
The figure above shows that on the transmit side, the number of TR switches, $N_{TRF}$, is smaller than the number of antenna elements, $N_T$. To provide more flexibility, each antenna element can be connected to one or more TR modules. In addition, analog phase shifters can be inserted between each TR module and antenna to provide some limited steering capability.

The configuration on the receiver side is similar, as shown in the figure. The maximum number of data streams, $N_s$, that can be supported by this system is the smaller of $N_{TRF}$ and $N_{RRF}$.

In this configuration, it is no longer possible to apply digital weights on each antenna element. Instead, the digital weights can only be applied at each RF chain. At the element level, the signal is adjusted by analog phase shifters, which only changes the phase of the signal. Thus, the precoding or combining are actually done in two stages. Because this approach performs beamforming in both digital and analog domains, it is referred to as hybrid beamforming.

### System Setup

This section simulates a 64 x 16 MIMO hybrid beamforming system, with a 64-element square array with 4 RF chains on the transmitter side and a 16-element square array with 4 RF chains on the receiver side.

```
Nt = 64;
NtRF = 4;

Nr = 16;
NrRF = 4;
```

In this simulation, it is assumed that each antenna is connected to all RF chains. Thus, each antenna is connected to 4 phase shifters. Such an array can be modeled by partitioning the array aperture into 4 completely connected subarrays.

```
rng(4096);
c = 3e8;
fc = 28e9;
lambda = c/fc;
txarray = phased.PartitionedArray(...
    'Array',phased.URA([sqrt(Nt) sqrt(Nt)],lambda/2),...
    'SubarraySelection',ones(NtRF,Nt),'SubarraySteering','Custom');
rxarray = phased.PartitionedArray(...
```

```
        'Array',phased.URA([sqrt(Nr) sqrt(Nr)],lambda/2),...
        'SubarraySelection',ones(NrRF,Nr),'SubarraySteering','Custom');
```

To maximize the spectral efficiency, each RF chain can be used to send an independent data stream. In this case, the system can support up to 4 streams.

Next, assume a scattering environment with 6 scattering clusters randomly distributed in space. Within each cluster, there are 8 closely located scatterers with an angle spread of 5 degrees, for a total of 48 scatterers. The path gain for each scatterer is obtained from a complex circular symmetric Gaussian distribution.

```
Ncl = 6;
Nray = 8;
Nscatter = Nray*Ncl;
angspread = 5;
% compute randomly placed scatterer clusters
txclang = [rand(1,Ncl)*120-60;rand(1,Ncl)*60-30];
rxclang = [rand(1,Ncl)*120-60;rand(1,Ncl)*60-30];
txang = zeros(2,Nscatter);
rxang = zeros(2,Nscatter);
% compute the rays within each cluster
for m = 1:Ncl
    txang(:,(m-1)*Nray+(1:Nray)) = randn(2,Nray)*sqrt(angspread)+txclang(:,m);
    rxang(:,(m-1)*Nray+(1:Nray)) = randn(2,Nray)*sqrt(angspread)+rxclang(:,m);
end

g = (randn(1,Nscatter)+1i*randn(1,Nscatter))/sqrt(Nscatter);
```

The channel matrix can be formed as

```
txpos = getElementPosition(txarray)/lambda;
rxpos = getElementPosition(rxarray)/lambda;
H = scatteringchanmtx(txpos,rxpos,txang,rxang,g);
```

**Hybrid Weights Computation**

In a spatial multiplexing system with all digital beamforming, the signal is modulated by a set of precoding weights, propagated through the channel, and recovered by a set of combining weights. Mathematically, this process can be described by Y = (X*F*H+N)*W where X is an Ns-column matrix whose columns are data streams, F is an Ns × Nt matrix representing the precoding weights, W is an Nr × Ns matrix representing the combining weights, N is an Nr-column matrix whose columns are the receiver noise at each element, and Y is an Ns-column matrix whose columns are recovered data streams. Since the goal of the system is to achieve better spectral efficiency, obtaining the precoding and combining weights can be considered as an optimization problem where the optimal precoding and combining weights make the product of F*H*W' a diagonal matrix so each data stream can be recovered independently.

In a hybrid beamforming system, the signal flow is similar. Both the precoding weights and the combining weights are combinations of baseband digital weights and RF band analog weights. The baseband digital weights convert the incoming data streams to input signals at each RF chain and the analog weights then convert the signal at each RF chain to the signal radiated or collected at each antenna element. Note that the analog weights can only contain phase shifts.

Mathematically, it can be written as F=Fbb*Frf and W=Wbb*Wrf, where Fbb is an Ns × NtRF matrix, Frf an NtRF × Nt matrix, Wbb an NrRF × Ns matrix, and Wrf an Nr × NrRF matrix. Since both Frf and Wrf can only be used to modify the signal phase, there are extra constraints in the optimization

process to identify the optimal precoding and combining weights. Ideally, the resulting combination of `Fbb*Frf` and `Wrf*Wbb` are close approximations of `F` and `W` that are obtained without those constraints.

Unfortunately, optimizing all four matrix variables simultaneously is quite difficult. Therefore, many algorithms are proposed to arrive at suboptimal weights with a reasonable computational load. This example uses the approach proposed in [1] which decouples the optimizations for the precoding and combining weights. It first uses the orthogonal matching pursuit algorithm to derive the precoding weights. Once the precoding weights are computed, the result is then used to obtain the corresponding combining weights.

Assuming the channel is known, the unconstrained optimal precoding weights can be obtained by diagonalizing the channel matrix and extracting the first `NtRF` dominating modes. The transmit beam pattern can be plotted as

```
F = diagbfweights(H);
F = F(1:NtRF,:);
pattern(txarray,fc,-90:90,-90:90,'Type','efield',...
    'ElementWeights',F','PropagationSpeed',c);
```
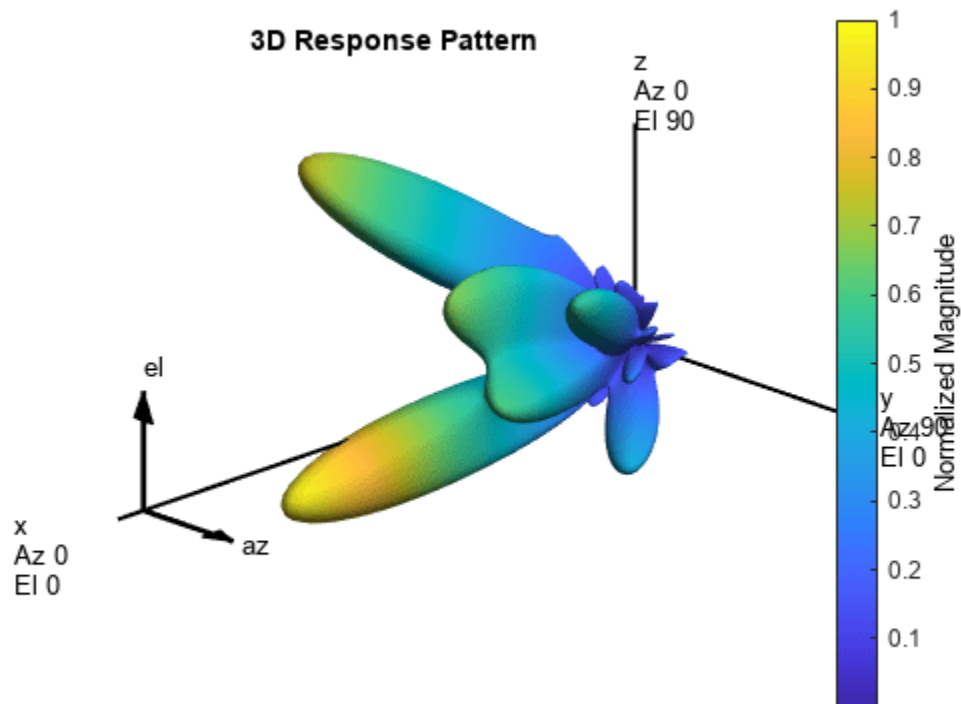


The response pattern above shows that even in a multipath environment, there are limited number of dominant directions.

The hybrid weights, on the other hand, can be computed as

```
At = steervec(txpos,txang);
Ar = steervec(rxpos,rxang);
```

```
Ns = NtRF;
[Fbb,Frf] = omphybweights(H,Ns,NtRF,At);
```

The beam pattern of the hybrid weights is shown below:

```
pattern(txarray,fc,-90:90,-90:90,'Type','efield',...
    'ElementWeights',Frf'*Fbb','PropagationSpeed',c);
```
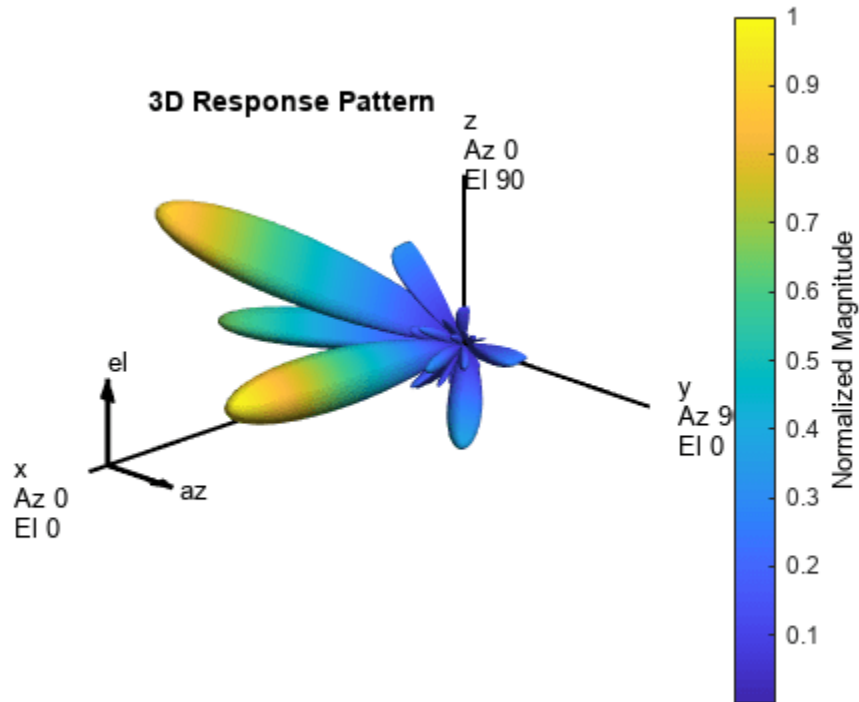


Compared to the beam pattern obtained using the optimal weights, the beam pattern using the hybrid weights is similar, especially for dominant beams. This means that the data streams can be successfully transmitted through those beams using hybrid weights.

**Spectral Efficiency Comparison**

One of the system level performance metrics of a 5G system is the spectral efficiency. The next section compares the spectral efficiency achieved using the optimal weights with that of the proposed hybrid beamforming weights. The simulation assumes 1 or 2 data streams as outlined in [1]. The transmit antenna array is assumed to be at a base station, with a focused beamwidth of 60 degrees in azimuth and 20 degrees in elevation. The signal can arrive at the receive array from any direction. The resulting spectral efficiency curve is obtained from 50 Monte-Carlo trials for each SNR.

```
snr_param = -40:5:0;
Nsnr = numel(snr_param);
Ns_param = [1 2];
NNs = numel(Ns_param);
```

```matlab
NtRF = 4;
NrRF = 4;

Ropt = zeros(Nsnr,NNs);
Rhyb = zeros(Nsnr,NNs);
Niter = 50;

for m = 1:Nsnr
    snr = db2pow(snr_param(m));
    for n = 1:Niter
        % Channel realization
        txang = [rand(1,Nscatter)*60-30;rand(1,Nscatter)*20-10];
        rxang = [rand(1,Nscatter)*180-90;rand(1,Nscatter)*90-45];
        At = steervec(txpos,txang);
        Ar = steervec(rxpos,rxang);
        g = (randn(1,Nscatter)+1i*randn(1,Nscatter))/sqrt(Nscatter);
        H = scatteringchanmtx(txpos,rxpos,txang,rxang,g);

        for k = 1:NNs
            Ns = Ns_param(k);
            % Compute optimal weights and its spectral efficiency
            [Fopt,Wopt] = helperOptimalHybridWeights(H,Ns,1/snr);
            Ropt(m,k) = Ropt(m,k)+helperComputeSpectralEfficiency(H,Fopt,Wopt,Ns,snr);

            % Compute hybrid weights and its spectral efficiency
            [Fbb,Frf,Wbb,Wrf] = omphybweights(H,Ns,NtRF,At,NrRF,Ar,1/snr);
            Rhyb(m,k) = Rhyb(m,k)+helperComputeSpectralEfficiency(H,Fbb*Frf,Wrf*Wbb,Ns,snr);
        end
    end
end
Ropt = Ropt/Niter;
Rhyb = Rhyb/Niter;

plot(snr_param,Ropt(:,1),'--sr',...
    snr_param,Ropt(:,2),'--b',...
    snr_param,Rhyb(:,1),'-sr',...
    snr_param,Rhyb(:,2),'-b');
xlabel('SNR (dB)');
ylabel('Spectral Efficiency (bits/s/Hz');
legend('Ns=1 optimal','Ns=2 optimal','Ns=1 hybrid', 'Ns=2 hybrid',...
    'Location','best');
grid on;
```
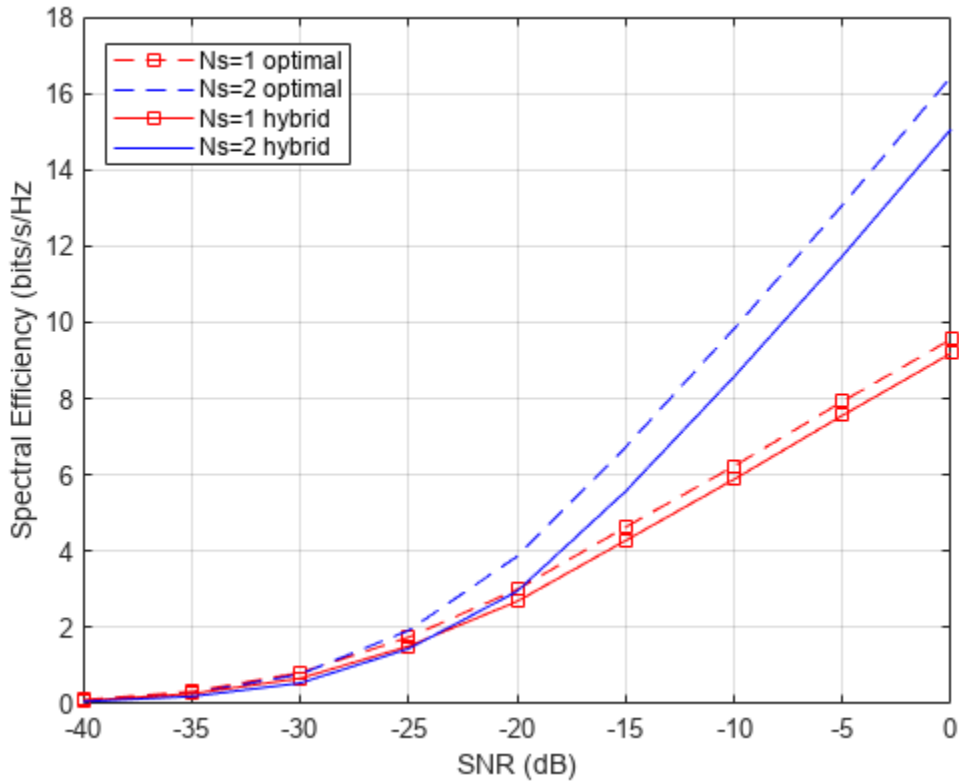
This figure shows that the spectral efficiency improves significantly when we increase the number of data streams. In addition, the hybrid beamforming can perform close to what optimal weights can offer using less hardware.

**Summary**

This example introduces the basic concept of hybrid beamforming and shows how to split the precoding and combining weights using orthogonal matching pursuit algorithm. It shows that hybrid beamforming can closely match the performance offered by optimal digital weights.

**References**

[1] Omar El Ayach, et al. Spatially Sparse Precoding in Millimeter wave MIMO Systems, IEEE Transactions on Wireless Communications, Vol. 13, No. 3, March 2014.

# MIMO-OFDM Precoding with Phased Arrays

This example shows how phased arrays are used in a MIMO-OFDM communication system employing beamforming. Using components from Communications Toolbox™ and Phased Array System Toolbox™, it models the radiating elements that comprise a transmitter and the front-end receiver components, for a MIMO-OFDM communication system. With user-specified parameters, you can validate the performance of the system in terms of bit error rate and constellations for different spatial locations and array sizes.

The example uses functions and System objects™ from Communications Toolbox and Phased Array System Toolbox and requires

• WINNER II Channel Model for Communications Toolbox

**Introduction**

MIMO-OFDM systems are the norm in current wireless systems (e.g. 5G NR, LTE, WLAN) due to their robustness to frequency-selective channels and high data rates enabled. With ever-increasing demands on data rates supported, these systems are getting more complex and larger in configurations with increasing number of antenna elements, and resources (subcarriers) allocated.

With antenna arrays and spatial multiplexing, efficient techniques to realize the transmissions are necessary [ 6 ]. Beamforming is one such technique, that is employed to improve the signal to noise ratio (SNR) which ultimately improves the system performance, as measured here in terms of bit error rate (BER) [ 1 ].

This example illustrates an asymmetric MIMO-OFDM single-user system where the maximum number of antenna elements on transmit and receive ends can be 1024 and 32 respectively, with up to 16 independent data streams. It models a spatial channel where the array locations and antenna patterns are incorporated into the overall system design. For simplicity, a single point-to-point link (one base station communicating with one mobile user) is modeled. The link uses channel sounding to provide the transmitter with the channel information it needs for beamforming.

The example offers the choice of a few spatially defined channel models, specifically a WINNER II Channel model and a scattering-based model, both of which account for the transmit/receive spatial locations and antenna patterns.

```
s = rng(61);          % Set RNG state for repeatability
```

**System Parameters**

Define parameters for the system. These parameters can be modified to explore their impact on the system.

```
% Single-user system with multiple streams
prm.numUsers = 1;            % Number of users
prm.numSTS = 16;             % Number of independent data streams, 4/8/16/32/64
prm.numTx = 32;              % Number of transmit antennas
prm.numRx = 16;              % Number of receive antennas
prm.bitsPerSubCarrier = 6;   % 2: QPSK, 4: 16QAM, 6: 64QAM, 8: 256QAM
prm.numDataSymbols = 10;     % Number of OFDM data symbols

prm.fc = 4e9;                  % 4 GHz system
prm.chanSRate = 100e6;         % Channel sampling rate, 100 Msps
```

```
prm.ChanType = 'Scattering';    % Channel options: 'WINNER', 'Scattering',
                                %              'ScatteringFcn', 'StaticFlat'
prm.NFig = 5;                   % Noise figure, dB

% Array locations and angles
prm.posTx = [0;0;0];           % BS/Transmit array position, [x;y;z], meters
prm.mobileRange = 300;          % meters
% Angles specified as [azimuth;elevation], az=[-90 90], el=[-90 90]
prm.mobileAngle = [33; 0];     % degrees
prm.steeringAngle = [30; -20]; % Transmit steering angle (close to mobileAngle)
prm.enSteering = true;          % Enable/disable steering
```

Parameters to define the OFDM modulation employed for the system are specified below.

```
prm.FFTLength = 256;
prm.CyclicPrefixLength = 64;
prm.numCarriers = 234;
prm.NumGuardBandCarriers = [7 6];
prm.PilotCarrierIndices = [26 54 90 118 140 168 204 232];
nonDataIdx = [(1:prm.NumGuardBandCarriers(1))'; prm.FFTLength/2+1; ...
              (prm.FFTLength-prm.NumGuardBandCarriers(2)+1:prm.FFTLength)'; ...
              prm.PilotCarrierIndices.';];
prm.CarriersLocations = setdiff((1:prm.FFTLength)',sort(nonDataIdx));

numTx = prm.numTx;
numRx = prm.numRx;
numSTS = prm.numSTS;
prm.numFrmBits = numSTS*prm.numDataSymbols*prm.numCarriers* ...
                 prm.bitsPerSubCarrier*1/3-6; % Account for termination bits

prm.modMode = 2^prm.bitsPerSubCarrier; % Modulation order
% Account for channel filter delay
prm.numPadZeros = 3*(prm.FFTLength+prm.CyclicPrefixLength);

% Get transmit and receive array information
prm.numSTSVec = numSTS;
[isTxURA,expFactorTx,isRxURA,expFactorRx] = helperArrayInfo(prm,true);
```

The processing for channel sounding, data transmission and reception modeled in the example are shown in the following block diagrams.



**Channel Sounding**

**Data Transmission & Reception**
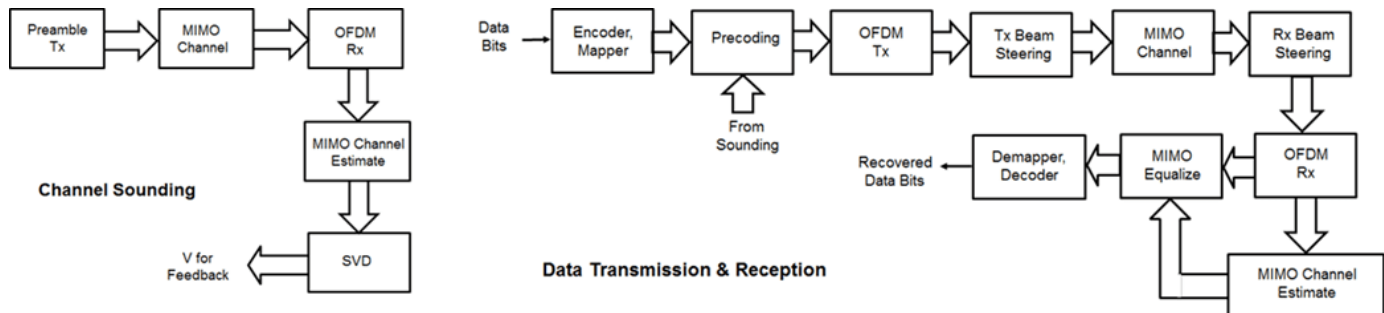
The free space path loss is calculated based on the base station and mobile station positions for the spatially-aware system modeled.

```
prm.cLight = physconst('LightSpeed');
prm.lambda = prm.cLight/prm.fc;
```

**18-379**

```
% Mobile position
[xRx,yRx,zRx] = sph2cart(deg2rad(prm.mobileAngle(1)),...
                         deg2rad(prm.mobileAngle(2)),prm.mobileRange);
prm.posRx = [xRx;yRx;zRx];
[toRxRange,toRxAng] = rangeangle(prm.posTx,prm.posRx);
spLoss = fspl(toRxRange,prm.lambda);
gainFactor = 1;
```

**Channel Sounding**

For a spatially multiplexed system, availability of channel information at the transmitter allows for precoding to be applied to maximize the signal energy in the direction and channel of interest. Under the assumption of a slowly varying channel, this is facilitated by sounding the channel first, wherein for a reference transmission, the receiver estimates the channel and feeds this information back to the transmitter.

For the chosen system, a preamble signal is sent over all transmitting antenna elements, and processed at the receiver accounting for the channel. The receiver components perform pre-amplification, OFDM demodulation, frequency domain channel estimation, and calculation of the feedback weights based on channel diagonalization using singular value decomposition (SVD) per data subcarrier.

```
% Generate the preamble signal
preambleSigSTS = helperGenPreamble(prm);
%   repeat over numTx
preambleSig = zeros(size(preambleSigSTS,1),numTx);
for i = 1:numSTS
    preambleSig(:,(i-1)*expFactorTx+(1:expFactorTx)) = ...
        repmat(preambleSigSTS(:,i),1,expFactorTx);
end

% Transmit preamble over channel
[rxPreSig,chanDelay] = helperApplyChannel(preambleSig,prm,spLoss);

% Front-end amplifier gain and thermal noise
rxPreAmp = phased.ReceiverPreamp( ...
    'Gain',gainFactor*spLoss, ... % account for path loss
    'NoiseFigure',prm.NFig, ...
    'ReferenceTemperature',290, ...
    'SampleRate',prm.chanSRate);
rxPreSigAmp = rxPreAmp(rxPreSig);
rxPreSigAmp = rxPreSigAmp * ...          % scale power
    (sqrt(prm.FFTLength-sum(prm.NumGuardBandCarriers)-1)/(prm.FFTLength));

% OFDM Demodulation
demodulatorOFDM = comm.OFDMDemodulator( ...
     'FFTLength',prm.FFTLength, ...
     'NumGuardBandCarriers',prm.NumGuardBandCarriers.', ...
     'RemoveDCCarrier',true, ...
     'PilotOutputPort',true, ...
     'PilotCarrierIndices',prm.PilotCarrierIndices.', ...
     'CyclicPrefixLength',prm.CyclicPrefixLength, ...
     'NumSymbols',numSTS, ... % preamble symbols alone
     'NumReceiveAntennas',numRx);

rxOFDM = demodulatorOFDM( ...
    rxPreSigAmp(chanDelay+1:end-(prm.numPadZeros-chanDelay),:));
```

```
% Channel estimation from preamble
%       numCarr, numSTS, numRx
hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

% Calculate the feedback weights
v = diagbfweights(hD);
```

For conciseness in presentation, front-end synchronization including carrier and timing recovery are assumed. The weights computed using `diagbfweights` are hence fed back to the transmitter, for subsequent application for the actual data transmission.

**Data Transmission**

Next, we configure the system's data transmitter. This processing includes channel coding, bit mapping to complex symbols, splitting of the individual data stream to multiple transmit streams, precoding of the transmit streams, OFDM modulation with pilot mapping and replication for the transmit antennas employed.

```
% Convolutional encoder
encoder = comm.ConvolutionalEncoder( ...
    'TrellisStructure',poly2trellis(7,[133 171 165]), ...
    'TerminationMethod','Terminated');

% Generate mapped symbols from bits
txBits = randi([0, 1],prm.numFrmBits,1);
encodedBits = encoder(txBits);

% Bits to QAM symbol mapping
mappedSym = qammod(encodedBits,prm.modMode,'InputType','Bit', ...
    'UnitAveragePower',true);

% Map to layers: per symbol, per data stream
gridData = reshape(mappedSym,prm.numCarriers,prm.numDataSymbols,numSTS);

% Apply precoding weights to the subcarriers, assuming perfect feedback
preData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for symIdx = 1:prm.numDataSymbols
    for carrIdx = 1:prm.numCarriers
        Q = squeeze(v(carrIdx,:,:));
        normQ = Q * sqrt(numTx)/norm(Q,'fro');
        preData(carrIdx,symIdx,:) = ...
            squeeze(gridData(carrIdx,symIdx,:)).' * normQ;
    end
end

% OFDM modulation of the data
modulatorOFDM = comm.OFDMModulator( ...
    'FFTLength',prm.FFTLength,...
    'NumGuardBandCarriers',prm.NumGuardBandCarriers.',...
    'InsertDCNull',true, ...
    'PilotInputPort',true,...
    'PilotCarrierIndices',prm.PilotCarrierIndices.',...
    'CyclicPrefixLength',prm.CyclicPrefixLength,...
    'NumSymbols',prm.numDataSymbols,...
    'NumTransmitAntennas',numSTS);

% Multi-antenna pilots
```

```
pilots = helperGenPilots(prm.numDataSymbols,numSTS);

txOFDM = modulatorOFDM(preData,pilots);
txOFDM = txOFDM * (prm.FFTLength/ ...
    sqrt(prm.FFTLength-sum(prm.NumGuardBandCarriers)-1)); % scale power

% Generate preamble with the feedback weights and prepend to data
preambleSigD = helperGenPreamble(prm,v);
txSigSTS = [preambleSigD;txOFDM];

% Repeat over numTx
txSig = zeros(size(txSigSTS,1),numTx);
for i = 1:numSTS
    txSig(:,(i-1)*expFactorTx+(1:expFactorTx)) = ...
        repmat(txSigSTS(:,i),1,expFactorTx);
end
```

For precoding, the preamble signal is regenerated to enable channel estimation. It is prepended to the data portion to form the transmission packet which is then replicated over the transmit antennas.

**Transmit Beam Steering**

Phased Array System Toolbox offers components appropriate for the design and simulation of phased arrays used in wireless communications systems.

For the spatially aware system, the signal transmitted from the base station is steered towards the direction of the mobile, so as to focus the radiated energy in the desired direction. This is achieved by applying a phase shift to each antenna element to steer the transmission.

The example uses a linear or rectangular array at the transmitter, depending on the number of data streams and number of transmit antennas selected.

```
% Gain per antenna element
amplifier = phased.Transmitter('PeakPower',1/numTx,'Gain',0);

% Amplify to achieve peak transmit power for each element
for n = 1:numTx
    txSig(:,n) = amplifier(txSig(:,n));
end

% Transmit antenna array definition
if isTxURA
    % Uniform Rectangular array
    arrayTx = phased.URA([expFactorTx,numSTS],[0.5 0.5]*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform Linear array
    arrayTx = phased.ULA(numTx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
end

% For evaluating weights for steering
SteerVecTx = phased.SteeringVector('SensorArray',arrayTx, ...
    'PropagationSpeed',prm.cLight);

% Generate weights for steered direction
wT = SteerVecTx(prm.fc,prm.steeringAngle);
```

```matlab
% Radiate along the steered direction, without signal combining
radiatorTx = phased.Radiator('Sensor',arrayTx, ...
    'WeightsInputPort',true, ...
    'PropagationSpeed',prm.cLight, ...
    'OperatingFrequency',prm.fc, ...
    'CombineRadiatedSignals',false);

if prm.enSteering
    txSteerSig = radiatorTx(txSig,repmat(prm.mobileAngle,1,numTx), ...
        conj(wT));
else
    txSteerSig = txSig;
end

% Visualize the array
h = figure('Position',figposition([10 55 22 35]),'MenuBar','none');
h.Name = 'Transmit Array Geometry';
viewArray(arrayTx);

% Visualize the transmit pattern and steering
h = figure('Position',figposition([32 55 22 30]),'MenuBar','none');
h.Name = 'Transmit Array Response Pattern';
pattern(arrayTx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wT);
h = figure('Position',figposition([54 55 22 35]),'MenuBar','none');
h.Name = 'Transmit Array Azimuth Pattern';
patternAzimuth(arrayTx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wT);
if isTxURA
    h = figure('Position',figposition([76 55 22 35]),'MenuBar','none');
    h.Name = 'Transmit Array Elevation Pattern';
    patternElevation(arrayTx,prm.fc,'PropagationSpeed',prm.cLight, ...
        'Weights',wT);
end
```

Array Geometry
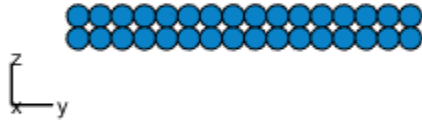
Aperture Size:
Y axis = 599.585 mm
Z axis =  74.948 mm
Element Spacing:
△ y = 37.474 mm
△ z = 37.474 mm

3D Directivity Pattern

**Azimuth Cut (elevation angle = 0.0°)**



Directivity (dBi), Broadside at 0.00 °

**Elevation Cut (azimuth angle = 0.0°)**



Directivity (dBi), Broadside at 0.00 °
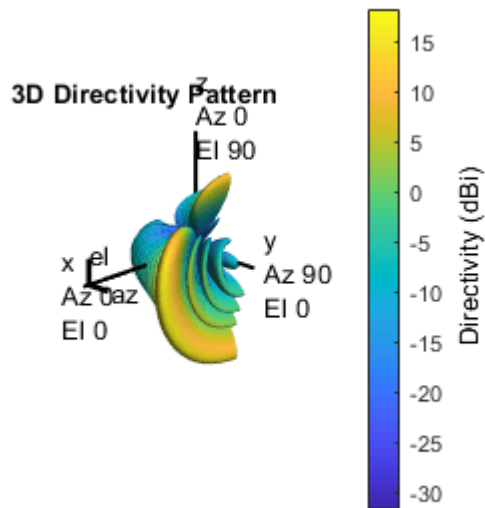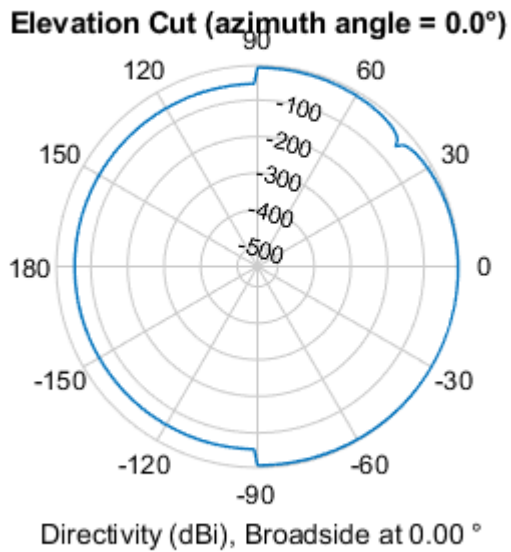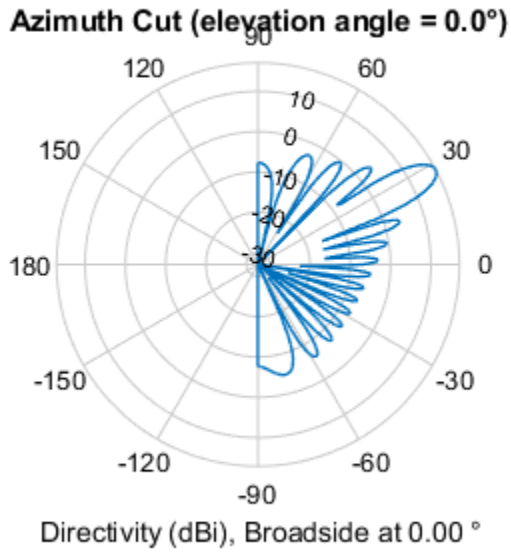
The plots indicate the array geometry and the transmit array response in multiple views. The response shows the transmission direction as specified by the steering angle.

The example assumes the steering angle known and close to the mobile angle. In actual systems, this would be estimated from angle-of-arrival estimation at the receiver as a part of the channel sounding or initial beam tracking procedures.

**Signal Propagation**

The example offers three options for spatial MIMO channels and a simpler static-flat MIMO channel for evaluation purposes.

The WINNER II channel model [ 5 ] is a spatially defined MIMO channel that allows you to specify the array geometry and location information. It is configured to use the typical urban microcell indoor scenario with very low mobile speeds.

The two scattering based channels use a single-bounce path through each scatterer where the number of scatterers is user-specified. For this example, the number of scatterers is set to 100. The 'Scattering' option models the scatterers placed randomly within a circle in between the transmitter and receiver, while the 'ScatteringFcn' models their placement completely randomly.

The models allow path loss modeling and both line-of-sight (LOS) and non-LOS propagation conditions. The example assumes non-LOS propagation and isotropic antenna element patterns with linear geometry.

```
% Apply a spatially defined channel to the steered signal
[rxSig,chanDelay] = helperApplyChannel(txSteerSig,prm,spLoss,preambleSig);
```

The same channel is used for both sounding and data transmission, with the data transmission having a longer duration controlled by the number of data symbols parameter, `prm.numDataSymbols`.

**Receive Beam Steering**

The receiver steers the incident signals to align with the transmit end steering, per receive element. Thermal noise and receiver gain are applied. Uniform linear or rectangular arrays with isotropic responses are modeled to match the channel and transmitter arrays.

```
rxPreAmp = phased.ReceiverPreamp( ...
    'Gain',gainFactor*spLoss, ... % accounts for path loss
    'NoiseFigure',prm.NFig, ...
    'ReferenceTemperature',290, ...
    'SampleRate',prm.chanSRate);

% Front-end amplifier gain and thermal noise
rxSigAmp = rxPreAmp(rxSig);
rxSigAmp = rxSigAmp * ...            % scale power
    (sqrt(prm.FFTLength - sum(prm.NumGuardBandCarriers)-1)/(prm.FFTLength));

% Receive array
if isRxURA
    % Uniform Rectangular array
    arrayRx = phased.URA([expFactorRx,numSTS],0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform Linear array
    arrayRx = phased.ULA(numRx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement);
end
```

```matlab
% For evaluating receive-side steering weights
SteerVecRx = phased.SteeringVector('SensorArray',arrayRx, ...
    'PropagationSpeed',prm.cLight);

% Generate weights for steered direction towards mobile
wR = SteerVecRx(prm.fc,toRxAng);

% Steer along the mobile receive direction
if prm.enSteering
    rxSteerSig = rxSigAmp.*(wR');
else
    rxSteerSig = rxSigAmp;
end

% Visualize the array
h = figure('Position',figposition([10 20 22 35]),'MenuBar','none');
h.Name = 'Receive Array Geometry';
viewArray(arrayRx);

% Visualize the receive pattern and steering
h = figure('Position',figposition([32 20 22 30]));
h.Name = 'Receive Array Response Pattern';
pattern(arrayRx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wR);
h = figure('Position',figposition([54 20 22 35]),'MenuBar','none');
h.Name = 'Receive Array Azimuth Pattern';
patternAzimuth(arrayRx,prm.fc,'PropagationSpeed',prm.cLight,'Weights',wR);
if isRxURA
    figure('Position',figposition([76 20 22 35]),'MenuBar','none');
    h.Name = 'Receive Array Elevation Pattern';
    patternElevation(arrayRx,prm.fc,'PropagationSpeed',prm.cLight, ...
        'Weights',wR);
end
```

Array Geometry
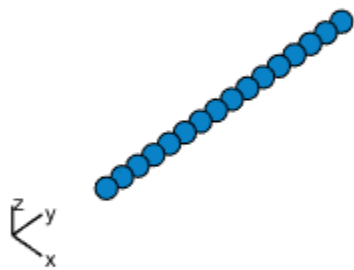


Aperture Size:
Y axis = 599.585 mm
Element Spacing:
$\Delta y$ = 37.474 mm
Array Axis: Y axis

**3D Directivity Pattern**

Directivity (dBi), Broadside at 0.00 °
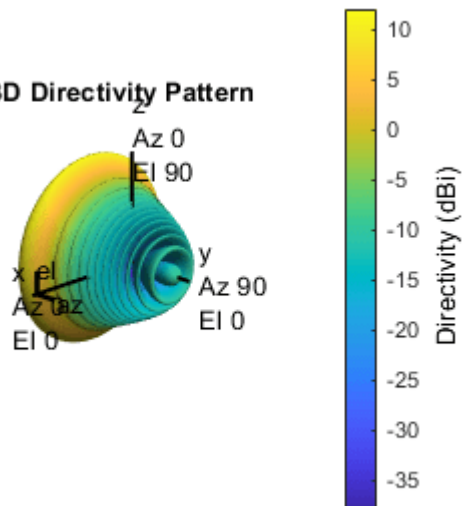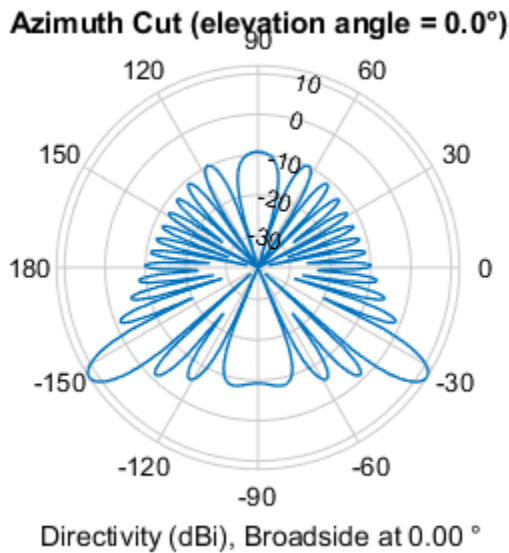
The receive antenna pattern mirrors the transmission steering.

**Signal Recovery**

The receive antenna array passes the propagated signal to the receiver to recover the original information embedded in the signal. Similar to the transmitter, the receiver used in a MIMO-OFDM system contains many components, including OFDM demodulator, MIMO equalizer, QAM demodulator, and channel decoder.

```
demodulatorOFDM = comm.OFDMDemodulator( ...
    'FFTLength',prm.FFTLength, ...
    'NumGuardBandCarriers',prm.NumGuardBandCarriers.', ...
    'RemoveDCCarrier',true, ...
    'PilotOutputPort',true, ...
    'PilotCarrierIndices',prm.PilotCarrierIndices.', ...
    'CyclicPrefixLength',prm.CyclicPrefixLength, ...
    'NumSymbols',numSTS+prm.numDataSymbols, ... % preamble & data
    'NumReceiveAntennas',numRx);

% OFDM Demodulation
rxOFDM = demodulatorOFDM( ...
    rxSteerSig(chanDelay+1:end-(prm.numPadZeros-chanDelay),:));

% Channel estimation from the mapped preamble
hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

% MIMO Equalization
[rxEq,CSI] = helperMIMOEqualize(rxOFDM(:,numSTS+1:end,:),hD);

% Soft demodulation
scFact = ((prm.FFTLength-sum(prm.NumGuardBandCarriers)-1) ...
        /prm.FFTLength^2)/numTx;
```

```
nVar = noisepow(prm.chanSRate,prm.NFig,290)/scFact;
rxSymbs = rxEq(:)/sqrt(numTx);
rxLLRBits = qamdemod(rxSymbs,prm.modMode,'UnitAveragePower',true, ...
    'OutputType','approxllr','NoiseVariance',nVar);

% Apply CSI prior to decoding
rxLLRtmp = reshape(rxLLRBits,prm.bitsPerSubCarrier,[], ...
                   prm.numDataSymbols,numSTS);
csitmp = reshape(CSI,1,[],1,numSTS);
rxScaledLLR = rxLLRtmp.*csitmp;

% Soft-input channel decoding
decoder = comm.ViterbiDecoder(...
    'InputFormat','Unquantized', ...
    'TrellisStructure',poly2trellis(7, [133 171 165]), ...
    'TerminationMethod','Terminated', ...
    'OutputDataType','double');
rxDecoded = decoder(rxScaledLLR(:));

% Decoded received bits
rxBits = rxDecoded(1:prm.numFrmBits);
```

For the MIMO system modeled, the displayed receive constellation of the equalized symbols offers a qualitative assessment of the reception. The actual bit error rate offers the quantitative figure by comparing the actual transmitted bits with the received decoded bits.

```
% Display received constellation
constDiag = comm.ConstellationDiagram( ...
    'SamplesPerSymbol',1, ...
    'ShowReferenceConstellation',true, ...
    'ReferenceConstellation', ...
    qammod((0:prm.modMode-1)',prm.modMode,'UnitAveragePower',true), ...
    'ColorFading',false, ...
    'Position',figposition([20 20 35 40]), ...
    'Title','Equalized Symbols', ...
    'EnableMeasurements',true, ...
    'MeasurementInterval',length(rxSymbs));
constDiag(rxSymbs);

% Compute and display bit error rate
ber = comm.ErrorRate;
measures = ber(txBits,rxBits);
fprintf('BER = %.5f; No. of Bits = %d; No. of errors = %d\n', ...
    measures(1),measures(3),measures(2));

rng(s); % Restore RNG state

BER = 0.00000; No. of Bits = 74874; No. of errors = 0
```
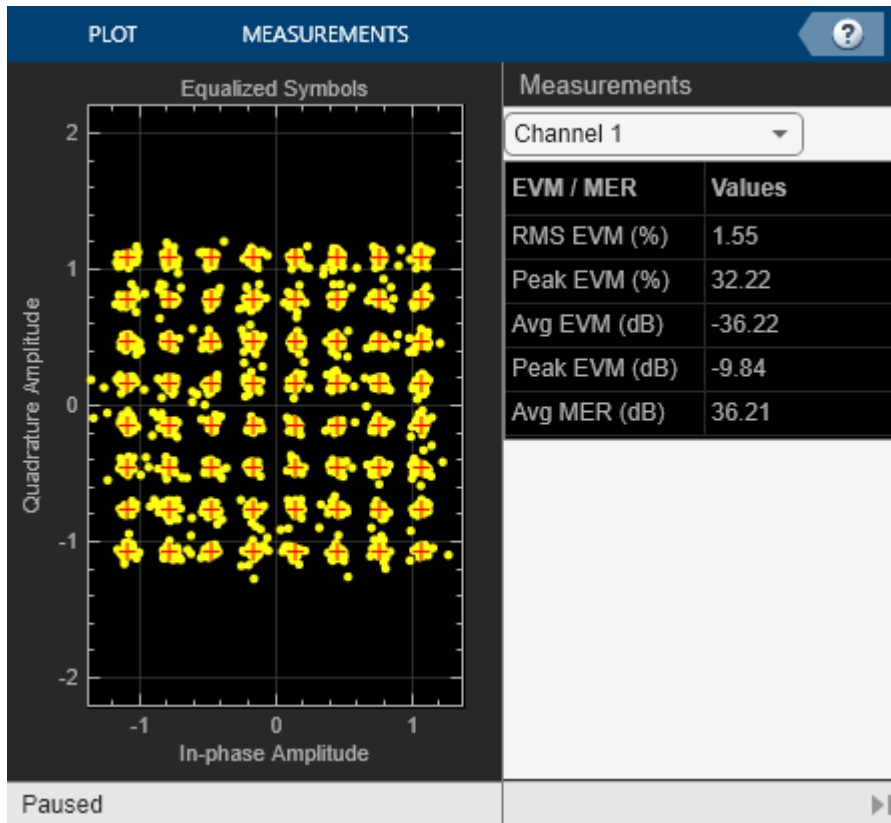
## Conclusion and Further Exploration

The example highlighted the use of phased antenna arrays for a beamformed MIMO-OFDM system. It accounted for the spatial geometry and location of the arrays at the base station and mobile station for a single user system. Using channel sounding, it illustrated how precoding is realized in current wireless systems and how steering of antenna arrays is modeled.

Within the set of configurable parameters, you can vary the number of data streams, transmit/receive antenna elements, station or array locations and geometry, channel models and their configurations to study the parameters' individual or combined effects on the system. E.g. vary just the number of transmit antennas to see the effect on the main lobe of the steered beam and the resulting system performance.

The example also made simplifying assumptions for front-end synchronization, channel feedback, user velocity and path loss models, which need to be further considered for a practical system. Individual systems also have their own procedures which must be folded in to the modeling [ 2, 3, 4 ].

Explore the following helper functions used:

- helperApplyChannel.m
- helperArrayInfo.m
- helperGenPilots.m
- helperGenPreamble.m
- helperGetP.m

- helperMIMOChannelEstimate.m
- helperMIMOEqualize.m

**Selected Bibliography**

**1** Perahia, Eldad, and Robert Stacey. Next Generation Wireless LANS: 802.11n and 802.11ac. Cambridge University Press, 2013.

**2** IEEE® Std 802.11™-2012 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

**3** 3GPP TS 36.213. "Physical layer procedures." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA). URL: https://www.3gpp.org.

**4** 3GPP TS 36.101. "User Equipment (UE) Radio Transmission and Reception." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA). URL: https://www.3gpp.org.

**5** Kyosti, Pekka, Juha Meinila, et al. WINNER II Channel Models. D1.1.2, V1.2. IST-4-027756 WINNER II, September 2007.

**6** George Tsoulos, Ed., "MIMO System Technology for Wireless Communications", CRC Press, Boca Raton, FL, 2006.

# Simulating Test Signals for a Radar Receiver

This example shows how to simulate received signal of a monostatic pulse radar to estimate the target range. A monostatic radar has the transmitter collocated with the receiver. The transmitter generates a pulse which hits the target and produces an echo received by the receiver. By measuring the location of the echoes in time, we can estimate the range of a target.

This example focuses on a pulse radar system design which can achieve a set of design specifications. It outlines the steps to translate design specifications, such as the probability of detection and the range resolution, into radar system parameters, such as the transmit power and the pulse width. It also models the environment and targets to synthesize the received signal. Finally, signal processing techniques are applied to the received signal to detect the ranges of the targets.

**Design Specifications**

The design goal of this pulse radar system is to detect non-fluctuating targets with at least one square meter radar cross section (RCS) at a distance up to 5000 meters from the radar with a range resolution of 50 meters. The desired performance index is a probability of detection (Pd) of 0.9 and probability of false alarm (Pfa) below 1e-6. Since coherent detection requires phase information and, therefore is more computationally expensive, we adopt a noncoherent detection scheme. In addition, this example assumes a free space environment.

```
pd = 0.9;              % Probability of detection
pfa = 1e-6;            % Probability of false alarm
max_range = 5000;      % Maximum unambiguous range
range_res = 50;        % Required range resolution
tgt_rcs = 1;           % Required target radar cross section
```

**Monostatic Radar System Design**

We need to define several characteristics of the radar system such as the waveform, the receiver, the transmitter, and the antenna used to radiate and collect the signal.

**Waveform**

We choose a rectangular waveform in this example. The desired range resolution determines the bandwidth of the waveform, which, in the case of a rectangular waveform, determines the pulse width.

Another important parameter of a pulse waveform is the pulse repetition frequency (PRF). The PRF is determined by the maximum unambiguous range.

```
prop_speed = physconst('LightSpeed');   % Propagation speed
pulse_bw = prop_speed/(2*range_res);    % Pulse bandwidth
pulse_width = 1/pulse_bw;               % Pulse width
prf = prop_speed/(2*max_range);         % Pulse repetition frequency
fs = 2*pulse_bw;                        % Sampling rate
waveform = phased.RectangularWaveform(...
    'PulseWidth',1/pulse_bw,...
    'PRF',prf,...
    'SampleRate',fs);
```

Note that we set the sampling rate as twice the bandwidth.

**Receiver Noise Characteristics**

We assume that the only noise present at the receiver is the thermal noise, so there is no clutter involved in this simulation. The power of the thermal noise is related to the receiver bandwidth. The receiver's noise bandwidth is set to be the same as the bandwidth of the waveform. This is often the case in real systems. We also assume that the receiver has a 20 dB gain and a 0 dB noise figure.

```
noise_bw = pulse_bw;

receiver = phased.ReceiverPreamp(...
    'Gain',20,...
    'NoiseFigure',0,...
    'SampleRate',fs,...
    'EnableInputPort',true);
```

Note that because we are modeling a monostatic radar, the receiver cannot be turned on until the transmitter is off. Therefore, we set the EnableInputPort property to true so that a synchronization signal can be passed from the transmitter to the receiver.

**Transmitter**

The most critical parameter of a transmitter is the peak transmit power. The required peak power is related to many factors including the maximum unambiguous range, the required SNR at the receiver, and the pulse width of the waveform. Among these factors, the required SNR at the receiver is determined by the design goal of Pd and Pfa, as well as the detection scheme implemented at the receiver.

The relation between Pd, Pfa and SNR can be best represented by a receiver operating characteristics (ROC) curve. We can generate the curve where Pd is a function of Pfa for varying SNRs using the following command:

```
snr_db = [-inf, 0, 3, 10, 13];
rocsnr(snr_db,'SignalType','NonfluctuatingNoncoherent');
```
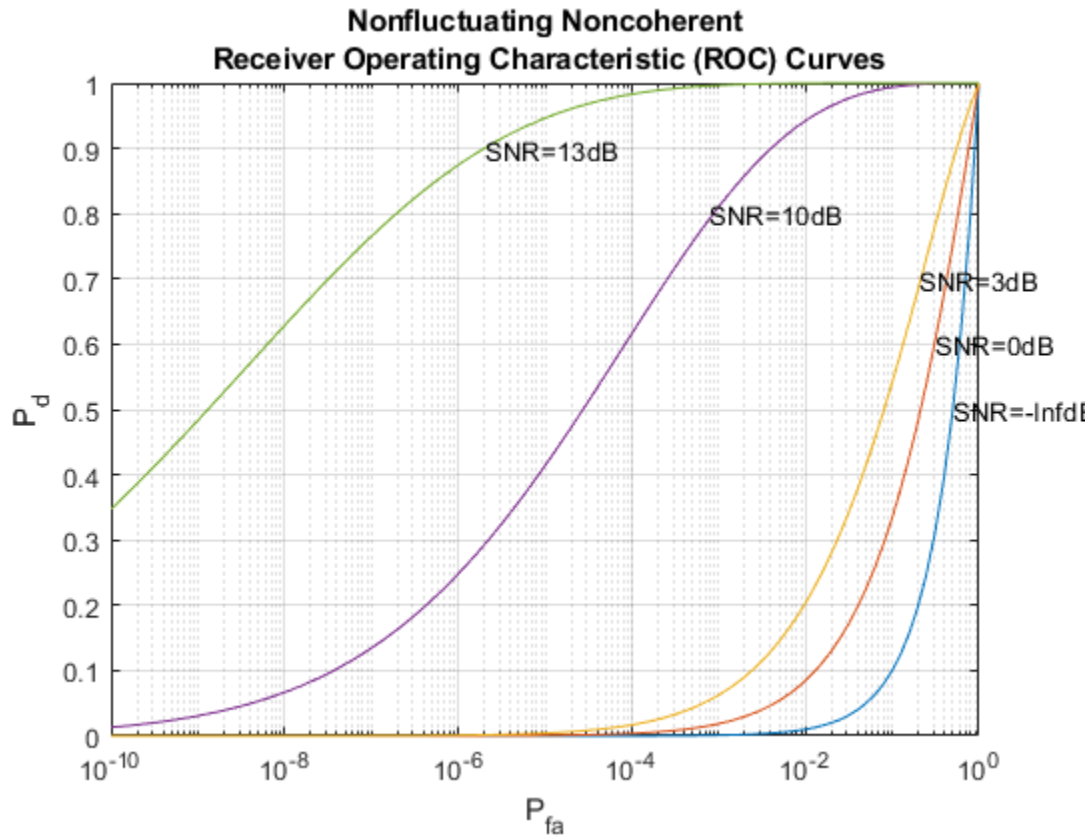
The ROC curves show that to satisfy the design goals of Pfa = 1e-6 and Pd = 0.9, the received signal's SNR must exceed 13 dB. This is a fairly high requirement and is not very practical. To make the radar system more feasible, we can use a pulse integration technique to reduce the required SNR. If we choose to integrate 10 pulses, the curve can be generated as

```
num_pulse_int = 10;
rocsnr([0 3 5],'SignalType','NonfluctuatingNoncoherent',...
    'NumPulses',num_pulse_int);
```

**Nonfluctuating Noncoherent Receiver Operating Characteristic (ROC) Curves**
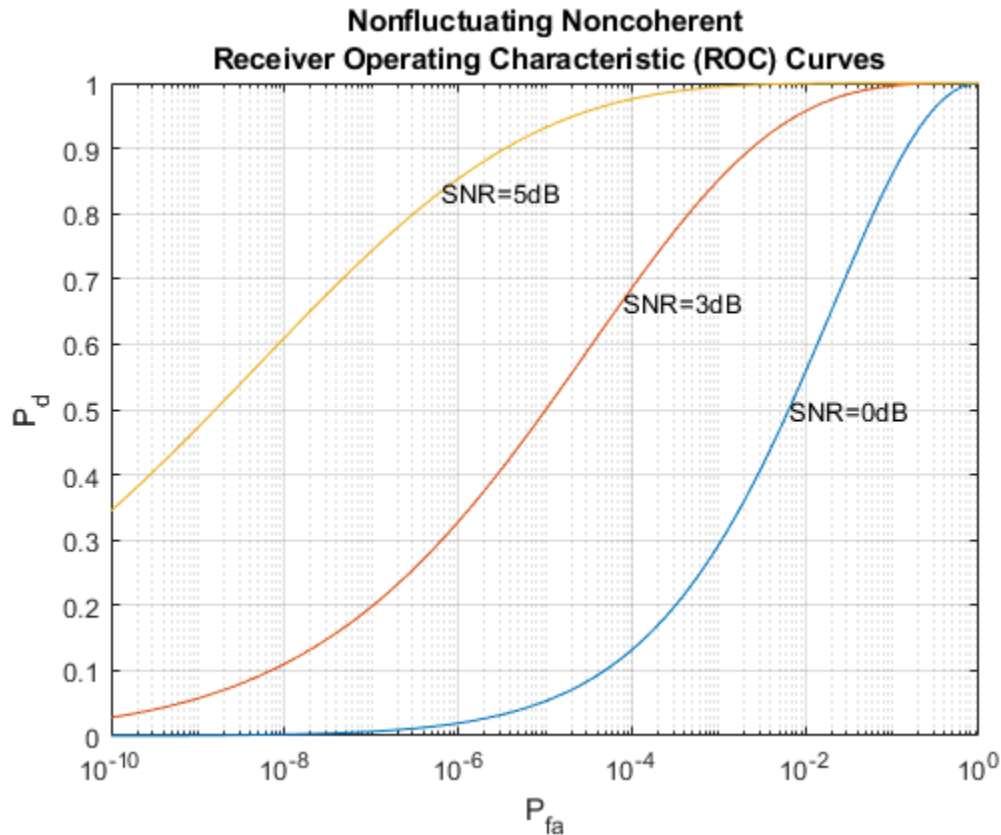


We can see that the required power has dropped to around 5 dB. Further reduction of SNR can be achieved by integrating more pulses, but the number of pulses available for integration is normally limited due to the motion of the target or the heterogeneity of the environment.

The approach above reads out the SNR value from the curve, but it is often desirable to calculate only the required value. For the noncoherent detection scheme, the calculation of the required SNR is, in theory, quite complex. Fortunately, there are good approximations available, such as Albersheim's equation. Using Albersheim's equation, the required SNR can be derived as

```
snr_min = albersheim(pd, pfa, num_pulse_int)
```

```
snr_min =

    4.9904
```

Once we obtain the required SNR at the receiver, the peak power at the transmitter can be calculated using the radar equation. Here we assume that the transmitter has a gain of 20 dB.

To calculate the peak power using the radar equation, we also need to know the wavelength of the propagating signal, which is related to the operating frequency of the system. Here we set the operating frequency to 10 GHz.

```
tx_gain = 20;
```

```
fc = 10e9;
```

```
lambda = prop_speed/fc;

peak_power = ((4*pi)^3*noisepow(1/pulse_width)*max_range^4*...
    db2pow(snr_min))/(db2pow(2*tx_gain)*tgt_rcs*lambda^2)

peak_power =

   5.2265e+03
```

Note that the resulting power is about 5 kW, which is very reasonable. In comparison, if we had not used the pulse integration technique, the resulting peak power would have been 33 kW, which is huge.

With all this information, we can configure the transmitter.

```
transmitter = phased.Transmitter(...
    'Gain',tx_gain,...
    'PeakPower',peak_power,...
    'InUseOutputPort',true);
```

Again, since this example models a monostatic radar system, the InUseOutputPort is set to true to output the status of the transmitter. This status signal can then be used to enable the receiver.

**Radiator and Collector**

In a radar system, the signal propagates in the form of an electromagnetic wave. Therefore, the signal needs to be radiated and collected by the antenna used in the radar system. This is where the radiator and the collector come into the picture.

In a monostatic radar system, the radiator and the collector share the same antenna, so we will first define the antenna. To simplify the design, we choose an isotropic antenna. Note that the antenna needs to be able to work at the operating frequency of the system (10 GHz), so we set the antenna's frequency range to 5-15 GHz.

We assume that the antenna is stationary.

```
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[5e9 15e9]);

sensormotion = phased.Platform(...
    'InitialPosition',[0; 0; 0],...
    'Velocity',[0; 0; 0]);
```

With the antenna and the operating frequency, we define both the radiator and the collector.

```
radiator = phased.Radiator(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);

collector = phased.Collector(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);
```

This completes the configuration of the radar system. In the following sections, we will define other entities, such as the target and the environment that are needed for the simulation. We will then simulate the signal return and perform range detection on the simulated signal.

**System Simulation**

**Targets**

To test our radar's ability to detect targets, we must define the targets first. Let us assume that there are 3 stationary, non-fluctuating targets in space. Their positions and radar cross sections are given below.

```matlab
tgtpos = [[2024.66;0;0],[3518.63;0;0],[3845.04;0;0]];
tgtvel = [[0;0;0],[0;0;0],[0;0;0]];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [1.6 2.2 1.05];
target = phased.RadarTarget('MeanRCS',tgtrcs,'OperatingFrequency',fc);
```

**Propagation Environment**

To simulate the signal, we also need to define the propagation channel between the radar system and each target.

```matlab
channel = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

Because this example uses a monostatic radar system, the channels are set to simulate two way propagation delays.

**Signal Synthesis**

We are now ready to simulate the entire system.

The synthesized signal is a data matrix with the fast time (time within each pulse) along each column and the slow time (time between pulses) along each row. To visualize the signal, it is helpful to define both the fast time grid and slow time grid.

```matlab
fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
slow_time_grid = (0:num_pulse_int-1)/prf;
```

The following loop simulates 10 pulses of the receive signal.

We set the seed for the noise generation in the receiver so that we can reproduce the same results.

```matlab
receiver.SeedSource = 'Property';
receiver.Seed = 2007;

% Pre-allocate array for improved processing speed
rxpulses = zeros(numel(fast_time_grid),num_pulse_int);

for m = 1:num_pulse_int

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);
```

```
    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    txsig = radiator(txsig,tgtang);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
    tgtsig = target(txsig);

    % Receive target returns at sensor
    rxsig = collector(tgtsig,tgtang);
    rxpulses(:,m) = receiver(rxsig,~(txstatus>0));
end
```

**Range Detection**

**Detection Threshold**

The detector compares the signal power to a given threshold. In radar applications, the threshold is often chosen so that the Pfa is below a certain level. In this case, we assume the noise is white Gaussian and the detection is noncoherent. Since we are also using 10 pulses to do the pulse integration, the signal power threshold is given by

```
npower = noisepow(noise_bw,receiver.NoiseFigure,...
    receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(pfa,num_pulse_int,'noncoherent'));
```

We plot the first two received pulses with the threshold

```
num_pulse_plot = 2;
helperRadarPulsePlot(rxpulses,threshold,...
    fast_time_grid,slow_time_grid,num_pulse_plot);
```
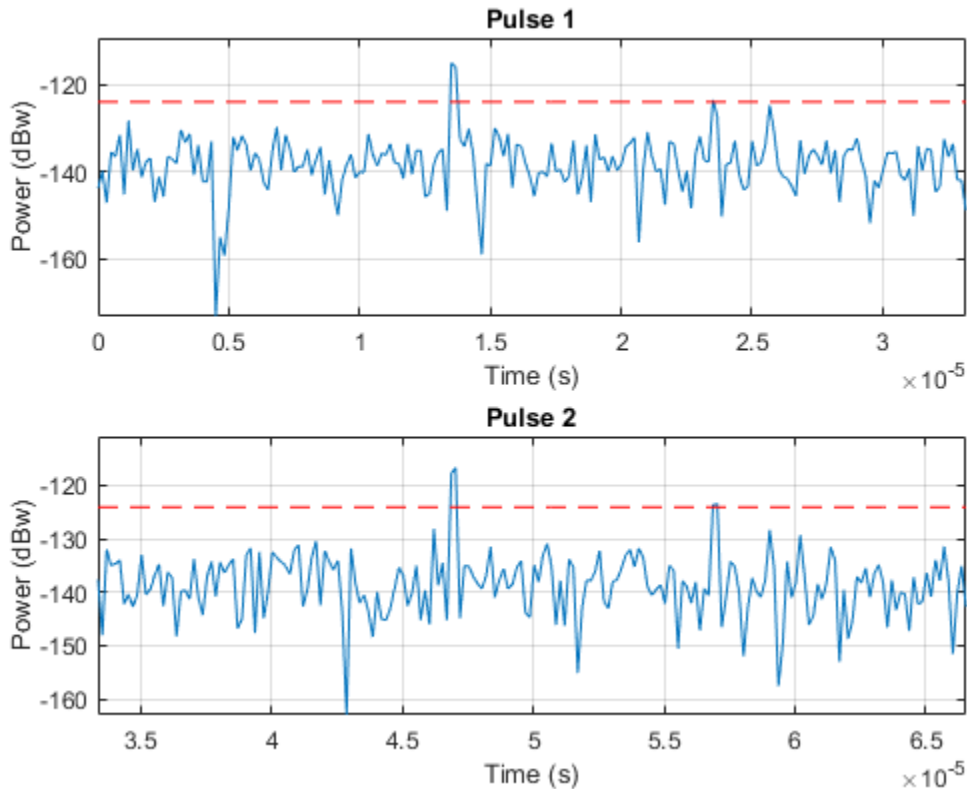
The threshold in these figures is for display purpose only. Note that the second and third target returns are much weaker than the first return because they are farther away from the radar. Therefore, the received signal power is range dependent and the threshold is unfair to targets located at different ranges.

**Matched Filter**

The matched filter offers a processing gain which improves the detection threshold. It convolves the received signal with a local, time-reversed, and conjugated copy of transmitted waveform. Therefore, we must specify the transmitted waveform when creating our matched filter. The received pulses are first passed through a matched filter to improve the SNR before doing pulse integration, threshold detection, etc.

```
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter(...
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);
[rxpulses, mfgain] = matchedfilter(rxpulses);
```

The matched filter introduces an intrinsic filter delay so that the locations of the peak (the maximum SNR output sample) are no longer aligned with the true target locations. To compensate for this delay, in this example, we will move the output of the matched filter forward and pad the zeros at the end. Note that in real systems, because the data is collected continuously, there is really no end of it.

```
matchingdelay = size(matchingcoeff,1)-1;
rxpulses = buffer(rxpulses(matchingdelay+1:end),size(rxpulses,1));
```

The threshold is then increased by the matched filter processing gain.

```
threshold = threshold * db2pow(mfgain);
```

The following plot shows the same two pulses after they pass through the matched filter.

```
helperRadarPulsePlot(rxpulses,threshold,...
    fast_time_grid,slow_time_grid,num_pulse_plot);
```
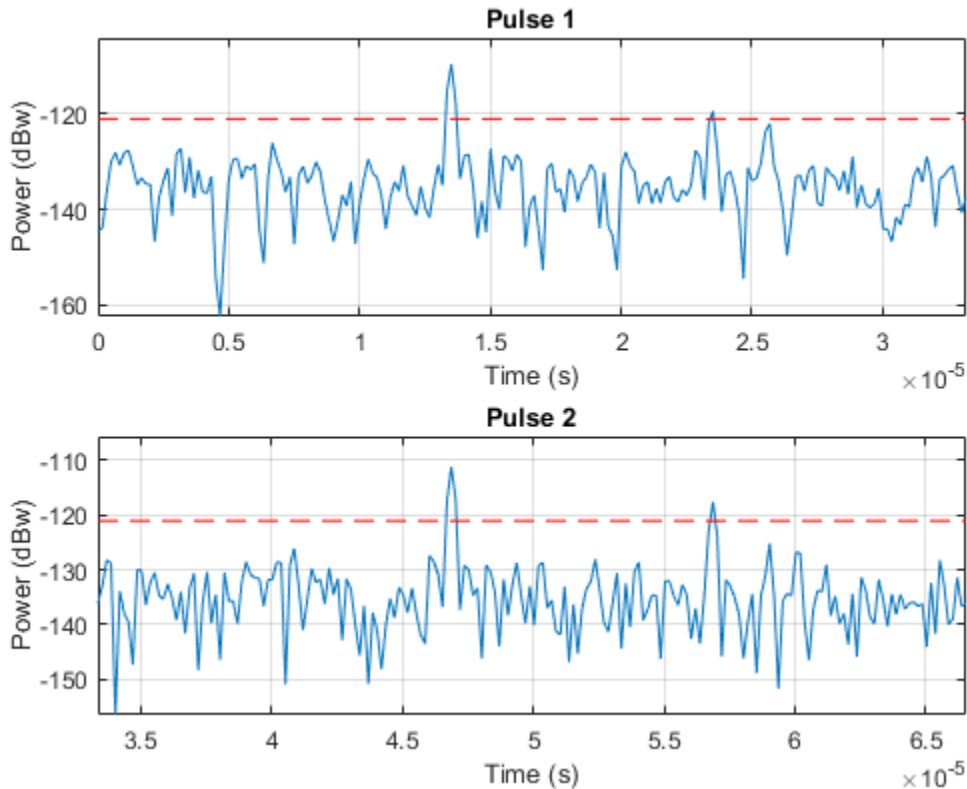


After the matched filter stage, the SNR is improved. However, because the received signal power is dependent on the range, the return of a close target is still much stronger than the return of a target farther away. Therefore, as the above figure shows, the noise from a close range bin also has a significant chance of surpassing the threshold and shadowing a target farther away. To ensure the threshold is fair to all the targets within the detectable range, we can use a time varying gain to compensate for the range dependent loss in the received echo.

To compensate for the range dependent loss, we first calculate the range gates corresponding to each signal sample and then calculate the free space path loss corresponding to each range gate. Once that information is obtained, we apply a time varying gain to the received pulse so that the returns are as if from the same reference range (the maximum detectable range).

```
range_gates = prop_speed*fast_time_grid/2;

tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(range_gates,lambda),...
    'ReferenceLoss',2*fspl(max_range,lambda));
```

```
rxpulses = tvg(rxpulses);
```

Now let's plot the same two pulses after the range normalization

```
helperRadarPulsePlot(rxpulses,threshold,...
    fast_time_grid,slow_time_grid,num_pulse_plot);
```
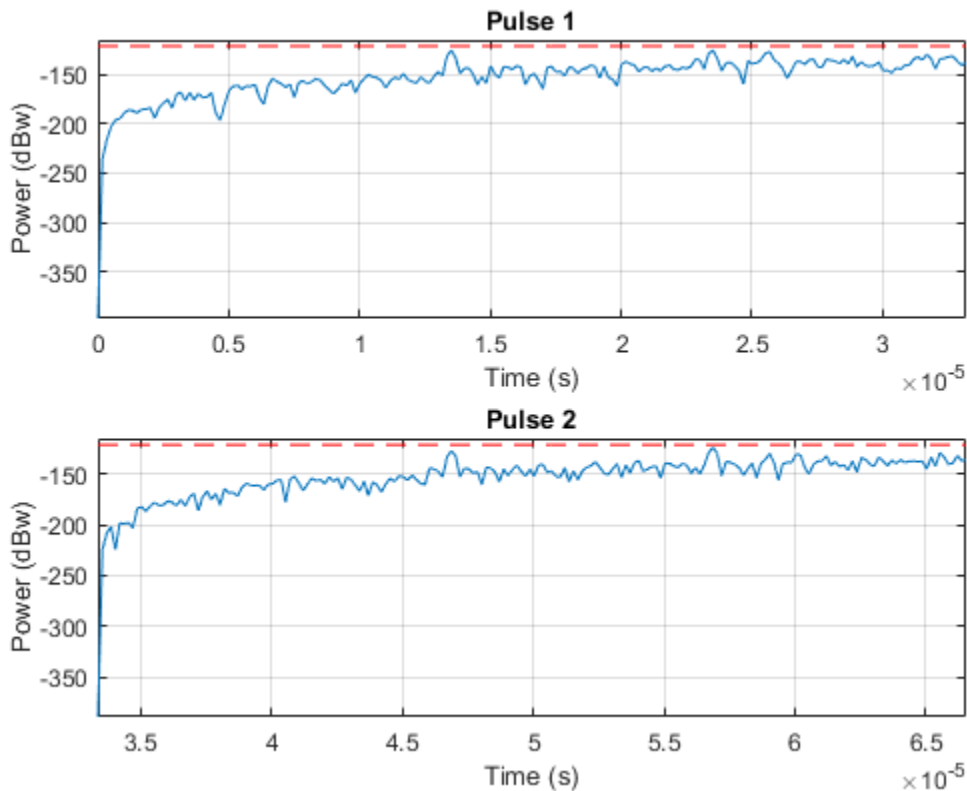


The time varying gain operation results in a ramp in the noise floor. However, the target return is now range independent. A constant threshold can now be used for detection across the entire detectable range.

Notice that at this stage, the threshold is above the maximum power level contained in each pulse. Therefore, nothing can be detected at this stage yet. We need to perform pulse integration to ensure the power of returned echoes from the targets can surpass the threshold while leaving the noise floor below the bar. This is expected since it is the pulse integration which allows us to use the lower power pulse train.

**Noncoherent Integration**

We can further improve the SNR by noncoherently integrating (video integration) the received pulses.

```
rxpulses = pulsint(rxpulses,'noncoherent');

helperRadarPulsePlot(rxpulses,threshold,...
    fast_time_grid,slow_time_grid,1);
```
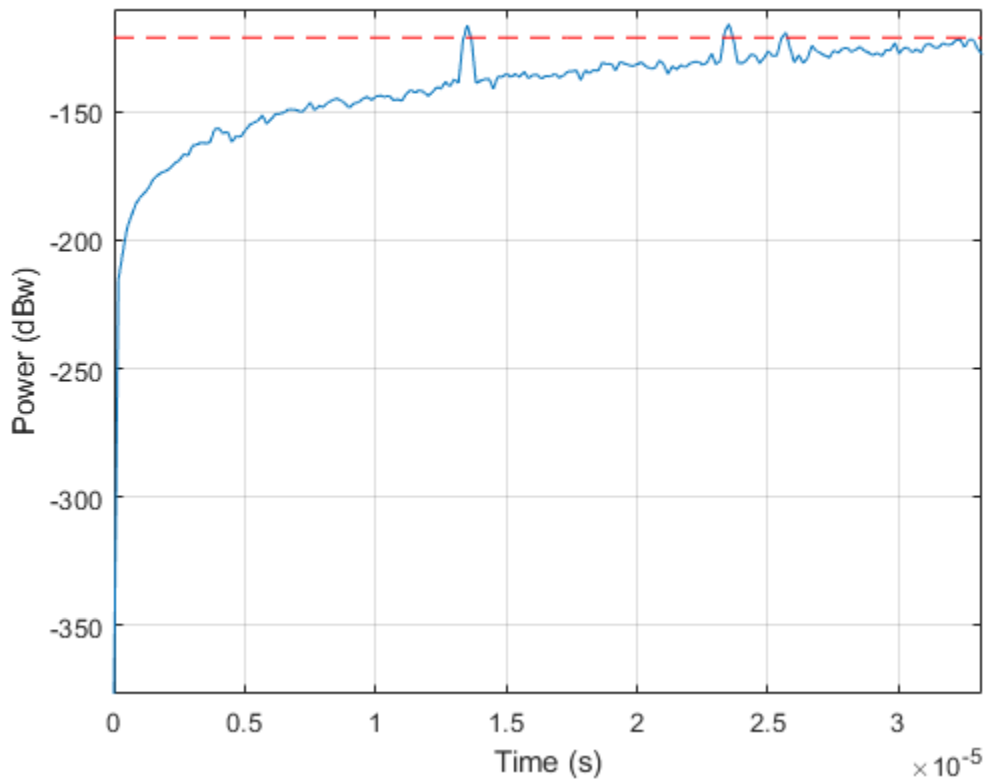
After the video integration stage, the data is ready for the final detection stage. It can be seen from the figure that all three echoes from the targets are above the threshold, and therefore can be detected.

**Range Detection**

Finally, the threshold detection is performed on the integrated pulses. The detection scheme identifies the peaks and then translates their positions into the ranges of the targets.

```
[~,range_detect] = findpeaks(rxpulses,'MinPeakHeight',sqrt(threshold));
```

The true ranges and the detected ranges of the targets are shown below:

```
true_range = round(tgtrng)
range_estimates = round(range_gates(range_detect))


true_range =

        2025        3519        3845


range_estimates =

        2025        3525        3850
```

Note that these range estimates are only accurate up to the range resolution (50 m) that can be achieved by the radar system.

**Summary**

In this example, we designed a radar system based on a set of given performance goals. From these performance goals, many design parameters of the radar system were calculated. The example also showed how to use the designed radar to perform a range detection task. In this example, the radar used a rectangular waveform. Interested readers can refer to "Waveform Design to Improve Range Performance of an Existing System" on page 18-178 for an example using a chirp waveform.

# Electronic Scanning Using a Uniform Rectangular Array

This example simulates a phased array radar that periodically scans a predefined surveillance region. A 900-element rectangular array is used in this monostatic radar. Steps are introduced to derive the radar parameters according to specifications. After synthesizing the received pulses, detection and range estimation are performed. Finally, Doppler estimation is used to obtain the speed of each target.

**Radar Definition**

First we create a phased array radar. We reuse most of the subsystems built in the example "Simulating Test Signals for a Radar Receiver" on page 18-393. Readers are encouraged to explore the details of radar system design through that example. A major difference is that we use a 30-by-30 uniform rectangular array (URA) in place of the original single antenna.

The existing radar design meets the following specifications.

```
pd = 0.9;              % Probability of detection
pfa = 1e-6;            % Probability of false alarm
max_range = 5000;      % Maximum unambiguous range
tgt_rcs = 1;           % Required target radar cross section
int_pulsenum = 10;     % Number of pulses to integrate
```

Load the radar system and retrieve system parameters.

```
load BasicMonostaticRadarExampleData;
```

```
fc = radiator.OperatingFrequency;        % Operating frequency (Hz)
v = radiator.PropagationSpeed;           % Wave propagation speed (m/s)
lambda = v/fc;                           % Wavelength (m)
fs = waveform.SampleRate;                % Sampling frequency (Hz)
prf = waveform.PRF;                      % Pulse repetition frequency (Hz)
```

Next, we define a 30-by-30 uniform rectangular array.

```
ura = phased.URA('Element',antenna,...
    'Size',[30 30],'ElementSpacing',[lambda/2, lambda/2]);
% Configure the antenna elements such that they only transmit forward
ura.Element.BackBaffled = true;

% Visualize the response pattern.
pattern(ura,fc,'PropagationSpeed',physconst('LightSpeed'),...
    'Type','powerdb');
```

3D Response Pattern
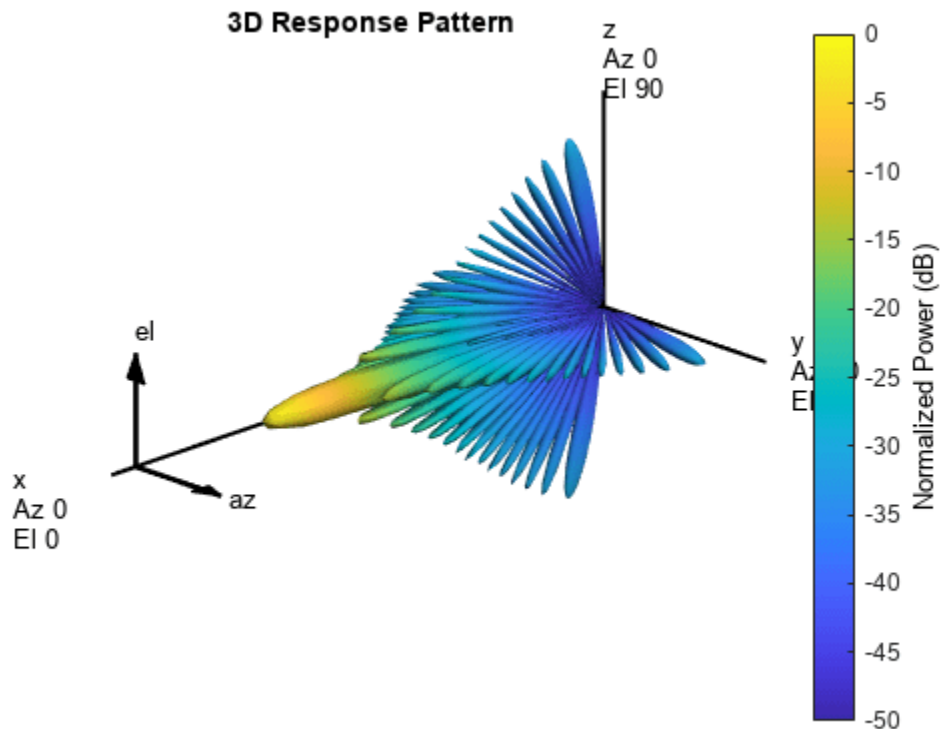
Associate the array with the radiator and collector.

```
radiator.Sensor = ura;
collector.Sensor = ura;

% We need to set the WeightsInputPort property to true to enable it to
% accept transmit beamforming weights
radiator.WeightsInputPort = true;
```

Now we need to recalculate the transmit power. The original transmit power was calculated based on a single antenna. For a 900-element array, the power required for each element is much less.

```
% Calculate the array gain
arraygain = phased.ArrayGain('SensorArray',ura,'PropagationSpeed',v);
ag = arraygain(fc,[0;0]);

% Calculate the peak power
snr_min = albersheim(pd, pfa, int_pulsenum);
peak_power = ((4*pi)^3*noisepow(1/waveform.PulseWidth)*max_range^4*...
    db2pow(snr_min))/(db2pow(2*(transmitter.Gain+ag))*tgt_rcs*lambda^2)
```

```
peak_power = 0.0065
```

The new peak power is 0.0065 Watts.

```
% Set the peak power of the transmitter
transmitter.PeakPower = peak_power;
```

We also need to design the scanning schedule of the phased array. To simplify the example, we only search in the azimuth dimension. We require the radar to search from 45 degrees to -45 degrees in azimuth. The revisit time should be less than 1 second, meaning that the radar should revisit the same azimuth angle within 1 second.

```
initialAz = 45; endAz = -45;
volumnAz = initialAz - endAz;
```

To determine the required number of scans, we need to know the beamwidth of the array response. We use an empirical formula to estimate the 3-dB beamwidth.

$$G = \frac{4\pi}{\theta^2}$$

where $G$ is the array gain and $\theta$ is the 3-dB beamwidth.

```
% Calculate 3-dB beamwidth
theta = radtodeg(sqrt(4*pi/db2pow(ag)))
```

```
theta = 6.7703
```

The 3-dB beamwidth is 6.77 degrees. To allow for some beam overlap in space, we choose the scan step to be 6 degrees.

```
scanstep = -6;
scangrid = initialAz+scanstep/2:scanstep:endAz;
numscans = length(scangrid);
pulsenum = int_pulsenum*numscans;
```

```
% Calculate revisit time
revisitTime = pulsenum/prf
```

```
revisitTime = 0.0050
```

The resulting revisit time is 0.005 second, well below the prescribed upper limit of 1 second.

**Target Definition**

We want to simulate the pulse returns from two non-fluctuating targets, both at 0 degrees elevation. The first target is approaching to the radar, while the second target is moving away from the radar.

```
tgtpos = [[3532.63; 800; 0],[2020.66; 0; 0]];
tgtvel = [[-100; 50; 0],[60; 80; 0]];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [1.6 2.2];
target = phased.RadarTarget('MeanRCS',tgtrcs,'OperatingFrequency',fc);

% Calculate the range, angle, and speed of the targets
[tgtrng,tgtang] = rangeangle(tgtmotion.InitialPosition,...
    sensormotion.InitialPosition);

numtargets = length(target.MeanRCS);
```

**Pulse Synthesis**

Now that all subsystems are defined, we can proceed to simulate the received signals. The total simulation time corresponds to one pass through the surveillance region. Because the reflected

signals are received by an array, we use a beamformer pointing to the steering direction to obtain the combined signal.

```matlab
% Create the steering vector for transmit beamforming
steeringvec = phased.SteeringVector('SensorArray',ura,...
    'PropagationSpeed',v);

% Create the receiving beamformer
beamformer = phased.PhaseShiftBeamformer('SensorArray',ura,...
    'OperatingFrequency',fc,'PropagationSpeed',v,...
    'DirectionSource','Input port');

% Define propagation channel for each target
channel = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);

fast_time_grid = unigrid(0, 1/fs, 1/prf, '[)');

% Pre-allocate array for improved processing speed
rxpulses = zeros(numel(fast_time_grid),pulsenum);

for m = 1:pulsenum

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);

    % Calculate steering vector for current scan angle
    scanid = floor((m-1)/int_pulsenum) + 1;
    sv = steeringvec(fc,scangrid(scanid));
    w = conj(sv);

    % Form transmit beam for this scan angle and simulate propagation
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    txsig = radiator(txsig,tgtang,w);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
    tgtsig = target(txsig);

    % Beamform the target returns received at the sensor
    rxsig = collector(tgtsig,tgtang);
    rxsig = receiver(rxsig,~(txstatus>0));
    rxpulses(:,m) = beamformer(rxsig,[scangrid(scanid);0]);
end
```

**Matched Filter**

To process the received signal, we first pass it through a matched filter, then integrate all pulses for each scan angle.

```matlab
% Matched filtering
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter(...
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);
[mf_pulses, mfgain] = matchedfilter(rxpulses);
mf_pulses = reshape(mf_pulses,[],int_pulsenum,numscans);

matchingdelay = size(matchingcoeff,1)-1;
sz_mfpulses = size(mf_pulses);
mf_pulses = [mf_pulses(matchingdelay+1:end) zeros(1,matchingdelay)];
mf_pulses = reshape(mf_pulses,sz_mfpulses);

% Pulse integration
int_pulses = pulsint(mf_pulses,'noncoherent');
int_pulses = squeeze(int_pulses);

% Visualize
r = v*fast_time_grid/2;
X = r'*cosd(scangrid); Y = r'*sind(scangrid);
clf;
pcolor(X,Y,pow2db(abs(int_pulses).^2));
axis equal tight
shading interp
axis off
text(-800,0,'Array');
text((max(r)+10)*cosd(initialAz),(max(r)+10)*sind(initialAz),...
    [num2str(initialAz) '^o']);
text((max(r)+10)*cosd(endAz),(max(r)+10)*sind(endAz),...
    [num2str(endAz) '^o']);
text((max(r)+10)*cosd(0),(max(r)+10)*sind(0),[num2str(0) '^o']);
colorbar;
```
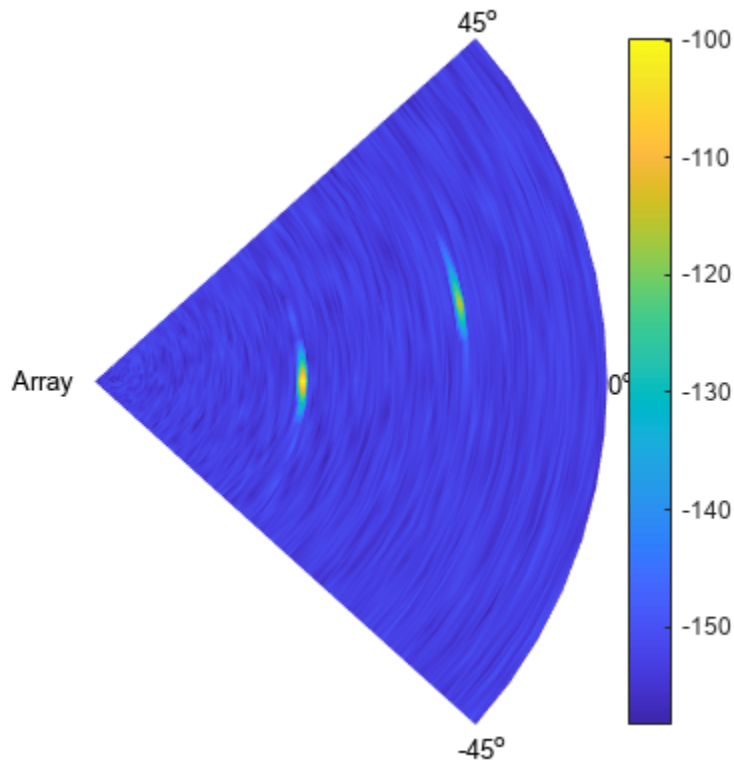
From the scan map, we can clearly see two peaks. The close one is at around 0 degrees azimuth, the remote one at around 10 degrees in azimuth.

**Detection and Range Estimation**

To obtain an accurate estimation of the target parameters, we apply threshold detection on the scan map. First we need to compensate for signal power loss due to range by applying time varying gains to the received signal.

```matlab
range_gates = v*fast_time_grid/2;
tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(range_gates,lambda),...
    'ReferenceLoss',2*fspl(max(range_gates),lambda));
tvg_pulses = tvg(mf_pulses);

% Pulse integration
int_pulses = pulsint(tvg_pulses,'noncoherent');
int_pulses = squeeze(int_pulses);

% Calculate the detection threshold

% sample rate is twice the noise bandwidth in the design system
noise_bw = receiver.SampleRate/2;
npower = noisepow(noise_bw,...
    receiver.NoiseFigure,receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(pfa,int_pulsenum,'noncoherent'));
```

```
% Increase the threshold by the matched filter processing gain
threshold = threshold * db2pow(mfgain);
```

We now visualize the detection process. To better represent the data, we only plot range samples beyond 50.

```
N = 51;
clf;
surf(X(N:end,:),Y(N:end,:),...
    pow2db(abs(int_pulses(N:end,:)).^2));
hold on;
mesh(X(N:end,:),Y(N:end,:),...
    pow2db(threshold*ones(size(X(N:end,:)))),'FaceAlpha',0.8);
view(0,56);
axis off
```
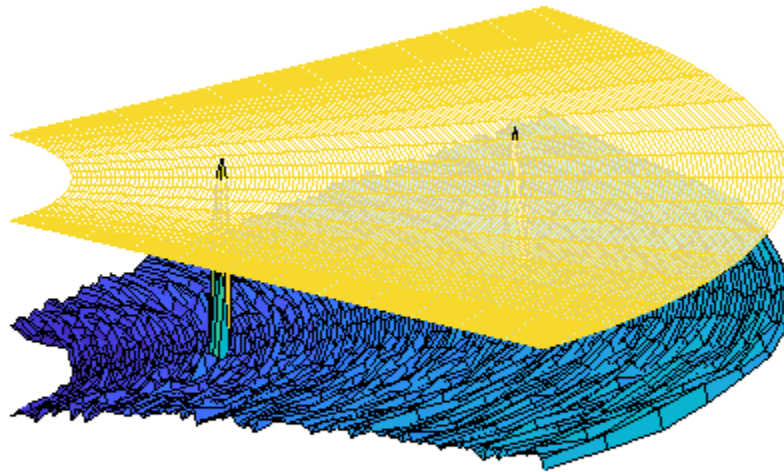


There are two peaks visible above the detection threshold, corresponding to the two targets we defined earlier. We can find the locations of these peaks and estimate the range and angle of each target.

```
[~,peakInd] = findpeaks(int_pulses(:),'MinPeakHeight',sqrt(threshold));
[rngInd,angInd] = ind2sub(size(int_pulses),peakInd);
est_range = range_gates(rngInd); % Estimated range
est_angle = scangrid(angInd);    % Estimated direction
```

**Doppler Estimation**

Next, we want to estimate the Doppler speed of each target. For details on Doppler estimation, refer to the example "Doppler Estimation" on page 18-308.

```
for m = numtargets:-1:1
    [p, f] = periodogram(mf_pulses(rngInd(m),:,angInd(m)),[],256,prf, ...
                'power','centered');
    speed_vec = dop2speed(f,lambda)/2;

    spectrum_data = p/max(p);
    [~,dop_detect1] = findpeaks(pow2db(spectrum_data),'MinPeakHeight',-5);
    sp(m) = speed_vec(dop_detect1); % Estimated Doppler speed
end
```

Finally, we have estimated all the parameters of both detected targets. Below is a comparison of the estimated and true parameter values.

```
--------------------------------------------------------------------------
              Estimated (true) target parameters
--------------------------------------------------------------------------
                Range (m)       Azimuth (deg)    Speed (m/s)
Target 1:    3625.00 (3622.08)   12.00 (12.76)    86.01 (86.49)
Target 2:    2025.00 (2020.66)    0.00 (0.00)    -59.68 (-60.00)
```

**Summary**

In this example, we showed how to simulate a phased array radar to scan a predefined surveillance region. We illustrated how to design the scanning schedule. A conventional beamformer was used to process the received multi-channel signal. The range, angle, and Doppler information of each target are extracted from the reflected pulses. This information can be used in further tasks such as high resolution direction-of-arrival estimation, or target tracking.

# Simulating Bistatic Polarimetric Radar

This example shows how to simulate a polarimetric bistatic radar system to estimate the range and speed of targets. Transmitter, receiver, and target kinematics are taken into account. For more information regarding polarization modeling capabilities, see "Modeling and Analyzing Polarization" on page 18-39.

**System Setup**

The system operates at 300 MHz, using a linear FM waveform whose maximum unambiguous range is 48 km. The range resolution is 50 meters and the time-bandwidth product is 20.

```
maxrng = 48e3;          % Maximum range
rngres = 50;            % Range resolution
tbprod = 20;            % Time-bandwidth product
```

The transmitter has a peak power of 2 kw and a gain of 20 dB. The receiver also provides a gain of 20 dB and the noise bandwidth is the same as the sweep bandwidth of the waveform.

The transmit antenna array is a stationary four-element Uniform Linear Arra (ULA) located at the origin. The array is made of vertical dipoles.

```
txAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
[waveform,transmitter,txmotion,radiator] = ...
    helperBistatTxSetup(maxrng,rngres,tbprod,txAntenna);
```

The receive antenna array is also a four-element ULA. It is located [20000;1000;100] meters away from the transmit antenna and is moving at a velocity of [0;20;0] m/s. Assume the elements in the receive array are also vertical dipoles. The received antenna array is oriented so that its broadside points back to the transmit antenna.

```
rxAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
[collector,receiver,rxmotion,rngdopresp,beamformer] = ...
    helperBistatRxSetup(rngres,rxAntenna);
```

There are two targets present in space. The first one is a point target modeled as a sphere; it preserves the polarization state of the incident signal. It is located at [15000;1000;500] meters away from the transmit array and is moving at a velocity of [100;100;0] m/s.

The second target is located [35000;-1000;1000] meters away from the transmit array and is approaching at a velocity of [-160;0;-50] m/s. Unlike the first target, the second target flips the polarization state of the incident signal, which means that the horizontal polarization component of the input signal becomes the vertical polarization components of the output signal and vice versa.

```
[target,tgtmotion,txchannel,rxchannel] = ...
    helperBistatTargetSetup(waveform.SampleRate);
```

A single scattering matrix is a fairly simple polarimetric model for a target. It assumes that no matter what the incident and reflecting directions are, the power distribution between the H and V components is fixed. However, even such a simple model can reveal complicated target behavior in the simulation because, firstly the H and V directions vary for different incident and reflecting directions; and secondly, the orientation of the targets, defined by the local coordinate system, also affects the polarization matching.

**System Simulation**

The next section simulates 256 received pulses. The receiving array is beamformed toward the two targets. The first figure shows the system setting and how the receive array and the targets move. The second figure shows a range-Doppler map generated for every 64 pulses received at the receiver array.

```
Nblock = 64; % Burst size
dt = 1/waveform.PRF;
y = complex(zeros(round(waveform.SampleRate*dt),Nblock));

hPlots = helperBistatViewSetup(txmotion,rxmotion,tgtmotion,waveform,...
    rngdopresp,y);
Npulse = Nblock*4;
for m = 1:Npulse

    % Update positions of transmitter, receiver, and targets
    [tpos,tvel,txax] = txmotion(dt);
    [rpos,rvel,rxax] = rxmotion(dt);
    [tgtp,tgtv,tgtax] = tgtmotion(dt);

    % Calculate the target angles as seen by the transmitter
    [txrng,radang] = rangeangle(tgtp,tpos,txax);

    % Simulate propagation of pulse in direction of targets
    wav = waveform();
    wav = transmitter(wav);
    sigtx = radiator(wav,radang,txax);
    sigtx = txchannel(sigtx,tpos,tgtp,tvel,tgtv);

    % Reflect pulse off of targets
    for n = 2:-1:1
        % Calculate bistatic forward and backward angles for each target
        [~,fwang] = rangeangle(tpos,tgtp(:,n),tgtax(:,:,n));
        [rxrng(n),bckang] = rangeangle(rpos,tgtp(:,n),tgtax(:,:,n));

        sigtgt(n) = target{n}(sigtx(n),fwang,bckang,tgtax(:,:,n));
    end

    % Receive path propagation
    sigrx = rxchannel(sigtgt,tgtp,rpos,tgtv,rvel);
    [~,inang] = rangeangle(tgtp,rpos,rxax);

    rspeed_t = radialspeed(tgtp,tgtv,tpos,tvel);
    rspeed_r = radialspeed(tgtp,tgtv,rpos,rvel);

    % Receive target returns at bistatic receiver
    sigrx = collector(sigrx,inang,rxax);
    yc = beamformer(sigrx,inang);
    y(:,mod(m-1,Nblock)+1) = receiver(sum(yc,2));

    helperBistatViewTrajectory(hPlots,tpos,rpos,tgtp);

    if ~rem(m,Nblock)
        rd_rng = (txrng+rxrng)/2;
        rd_speed = rspeed_t+rspeed_r;
        helperBistatViewSignal(hPlots,waveform,rngdopresp,y,rd_rng,...
            rd_speed)
```

```
        end
end
```

**System Dynamics**
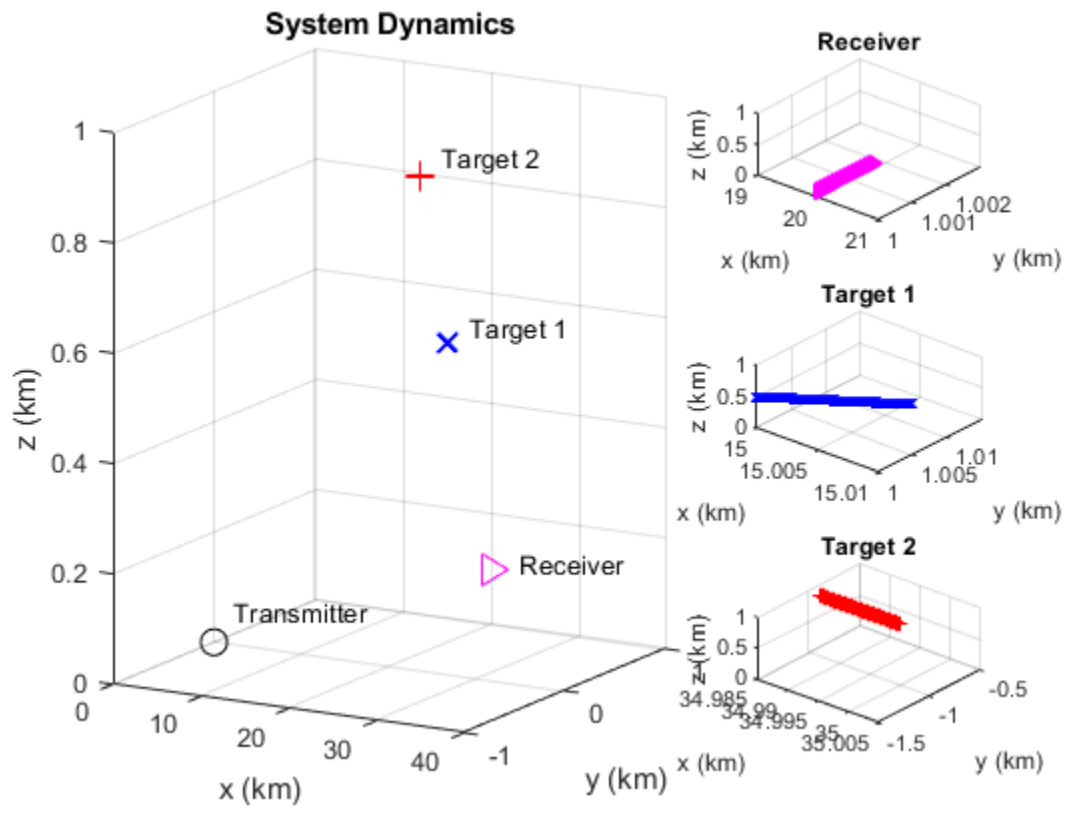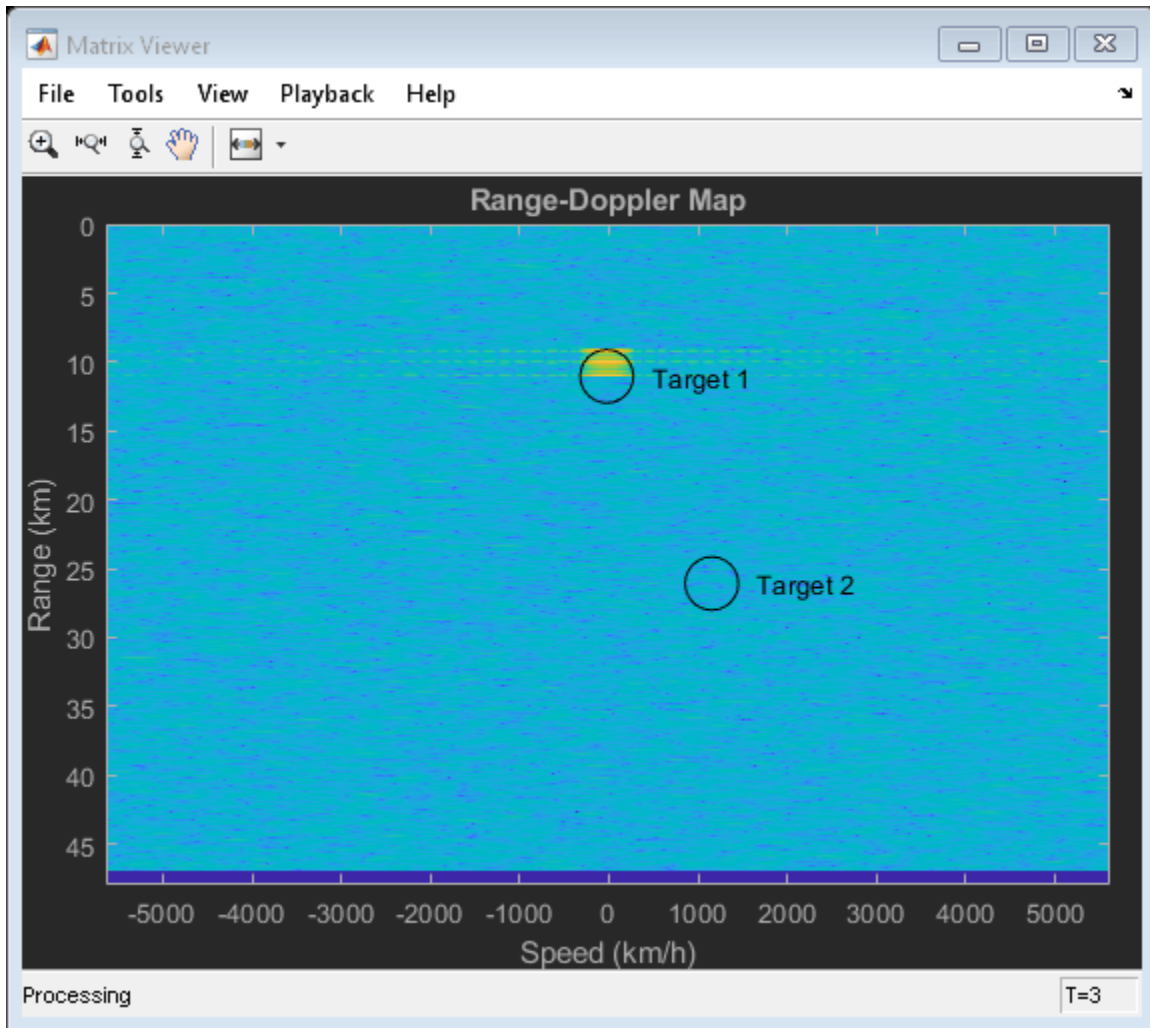
The range-Doppler map only shows the return from the first target. This is probably no surprise since both the transmit and receive array are vertically polarized and the second target maps the vertically polarized wave to horizontally polarized wave. The received signal from the second target is mostly orthogonal to the polarization of the receive array resulting in significant polarization loss.

You might also notice that the resulting range and radial speed do not agree with the range and radial speed of the target relative to the transmitter. This is because in a bistatic configuration, the estimated range is actually the geometric mean of the target range relative to the transmitter and the receiver. Similarly, the estimated radial speed is the sum of the target radial speed relative to the transmitter and the receiver. The circle in the map shows where the targets should appear in the range-Doppler map. Further processing is required to identify the exact location of the target, but that is beyond the scope of this example.

### Using Circularly Polarized Receive Array

Vertical dipole is a very popular choice of transmit antenna in real applications because it is low cost and has a omnidirectional pattern. However, the previous simulation shows that if the same antenna is used in the receiver, there is a risk that the system will miss certain targets. Therefore, a linear polarized antenna is often not the best choice as the receive antenna in such a configuration because no matter how the linear polarization is aligned, there always exists an orthogonal polarization. In

case the reflected signal bears a polarization state close to that direction, the polarization loss becomes large.

One way to solve this issue is to use a circularly polarized antenna at the receive end. A circularly polarized antenna cannot fully match any linear polarization. But on the other hand, the polarization loss between a circular polarized antenna and a linearly polarized signal is 3 dB, regardless what direction the linear polarization is in. Therefore, although it never gives the maximum return, it never misses a target. A frequently used antenna with circular polarization is a crossed dipole antenna.

The next section shows what happens when crossed dipole antennas are used to form the receive array.

```
rxAntenna = phased.CrossedDipoleAntennaElement;
collector = clone(collector);
collector.Sensor.Element = rxAntenna;
```

```
helperBistatSystemRun(waveform,transmitter,txmotion,radiator,collector,...
    receiver,rxmotion,rngdopresp,beamformer,target,tgtmotion,txchannel,...
    rxchannel,hPlots,Nblock,Npulse);
```
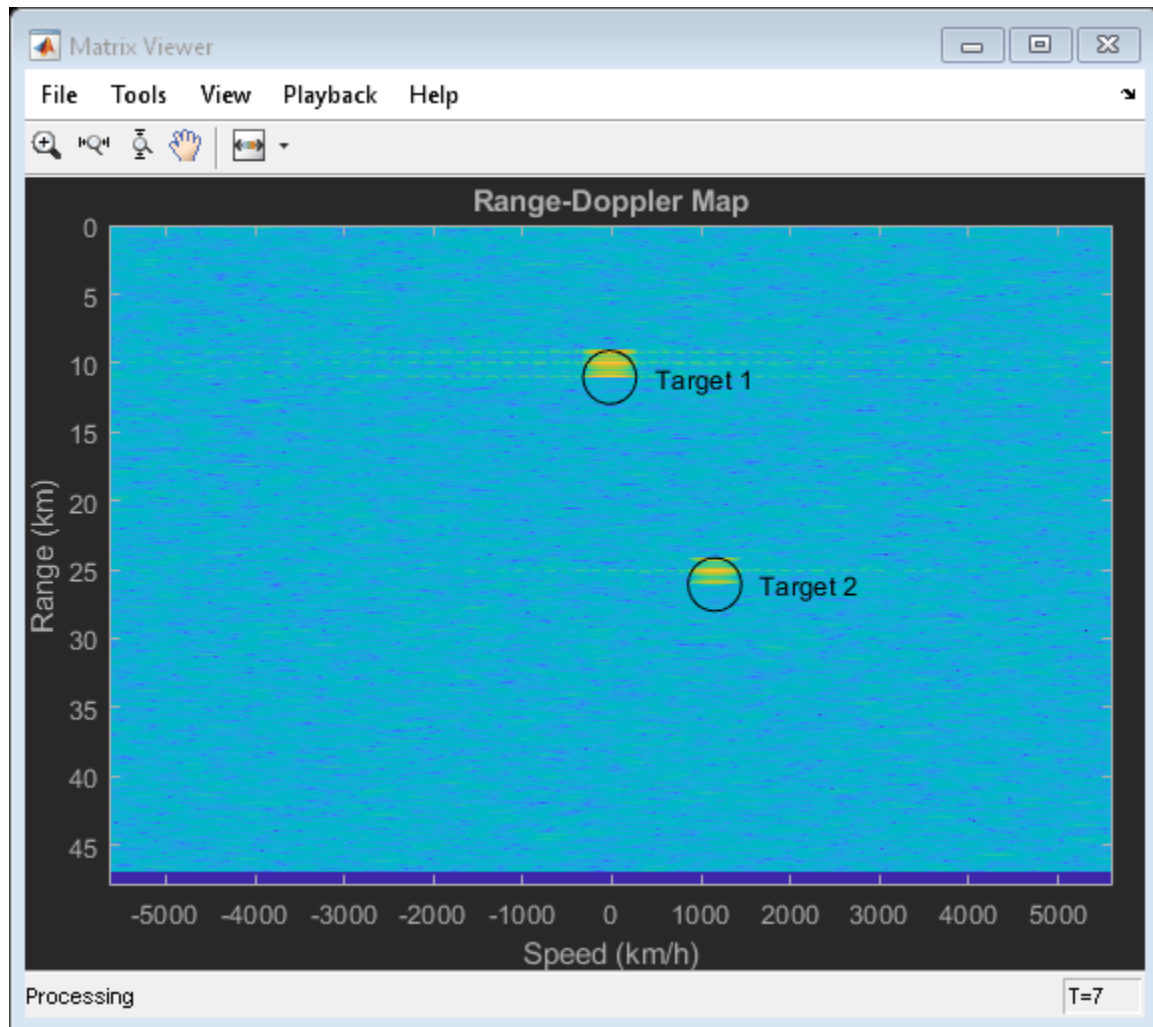


The range-Doppler map now shows both targets at their correct locations.

**Summary**

This example shows a system level simulation of a bistatic polarimetric radar. The example generates range-Doppler maps of the received signal for different transmit and receive array polarization configurations and shows how a circularly polarized antenna can be used to avoid losing linear polarized signals due to the polarization scattering property of a target.

# Stream and Accelerate System Simulation

Phased Array System Toolbox™ can be used to model an end-to-end phased array system - generate a transmitted waveform, simulate the target return, and then process the received signal to detect the target. This is shown in the examples "Simulating Test Signals for a Radar Receiver" on page 18-393 and "Waveform Design to Improve Range Performance of an Existing System" on page 18-178. This example shows how to simulate such a system in streaming mode so you can run the simulation for a long time and observe the system dynamics.

**Simulation Setup**

First, set up the radar system with some basic parameters. The entire radar system is similar to the one shown in the "Waveform Design to Improve Range Performance of an Existing System" on page 18-178 example.

```
fs = 6e6;
bw = 3e6;
c = 3e8;
fc = 10e9;
prf = 18750;
num_pulse_int = 10;

[waveform,transmitter,radiator,collector,receiver,sensormotion,...
    target,tgtmotion,channel,matchedfilter,tvg,threshold] = ...
    helperRadarStreamExampleSystemSetup(fs,bw,prf,fc,c);
```

**System Simulation**

Next, run the simulation for 100 pulses. During this simulation, four time-scopes are used to observe the signals at various stages. The first three scopes display the transmitted signal, received signal, and the post-matched-filter and gain-adjusted signal for 10 pulses. Although the transmitted signal is a high-power pulse train, scope 2 shows a much weaker received signal due to propagation loss. This signal cannot be detected using the preset detection threshold. Even after matched-filtering and gain compensation, it is still challenging to detect all three targets.

```
% pre-allocation
fast_time_grid = 0:1/fs:1/prf-1/fs;
num_pulse_samples = numel(fast_time_grid);
rx_pulses = complex(zeros(num_pulse_samples,num_pulse_int));
mf_pulses = complex(zeros(num_pulse_samples,num_pulse_int));
detect_pulse = zeros(num_pulse_samples,1);

% simulation loop
for m = 1:10*num_pulse_int

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);

    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [pulse,txstatus] = transmitter(pulse);
```

```
        txsig = radiator(pulse,tgtang);
        txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

        % Reflect pulse off of targets
        tgtsig = target(txsig);

        % Receive target returns at sensor
        rxsig = collector(tgtsig,tgtang);
        nn = mod(m-1,num_pulse_int)+1;
        rx_pulses(:,nn) = receiver(rxsig,~(txstatus>0));

        % Detection processing
        mf_pulses(:,nn) = matchedfilter(rx_pulses(:,nn));
        mf_pulses(:,nn) = tvg(mf_pulses(:,nn));

        % Perform pulse integration every 'num_pulse_int' pulses
        if nn == num_pulse_int
            detect_pulse = pulsint(mf_pulses,'noncoherent');
        end

        helperRadarStreamDisplay(pulse,abs(rx_pulses(:,nn)),...
            abs(mf_pulses(:,nn)),detect_pulse,...
            sqrt(threshold)*ones(num_pulse_samples,1));
    end
```
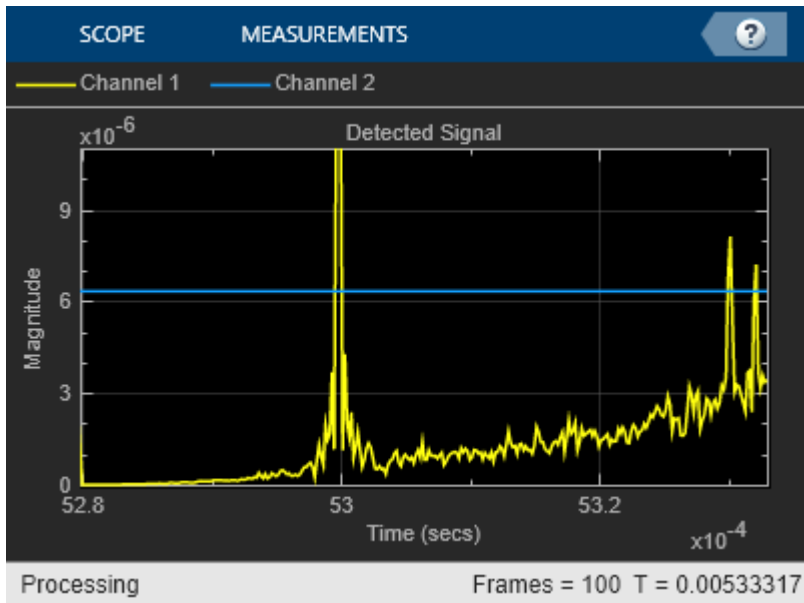
**Improve Simulation Speed Using Code Generation**

Because radar systems require intensive processing, simulation speed is a major concern. After you have run 100 pulse to check out your code, you may want to run 1000 pulses. When you run the simulation in interpreted MATLAB® mode, you can measure the elapsed time using:

```
tic;
helperRadarStreamRun;
time_interpreted = toc
```

```
time_interpreted =

    4.4584
```
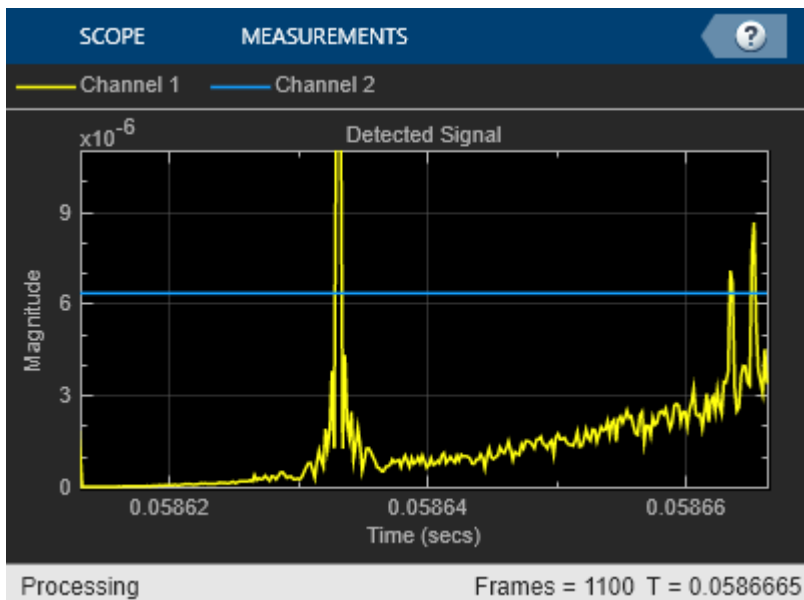
If the simulation is too slow, you can speed it up using MATLAB Coder™. MATLAB Coder can generate compiled MATLAB code resulting in significant improvement in processing speed. In this example, MATLAB Coder generates a helperRadarStreamRun_mex function from the helperRadarStreamRun function.

```
codegen helperRadarStreamRun.m
```

```
Code generation successful.
```

When the mex version is invoked, the simulation speed is improved.

```
tic;
helperRadarStreamRun_mex;
time_compiled = toc
```

```
time_compiled =

    1.0633
```
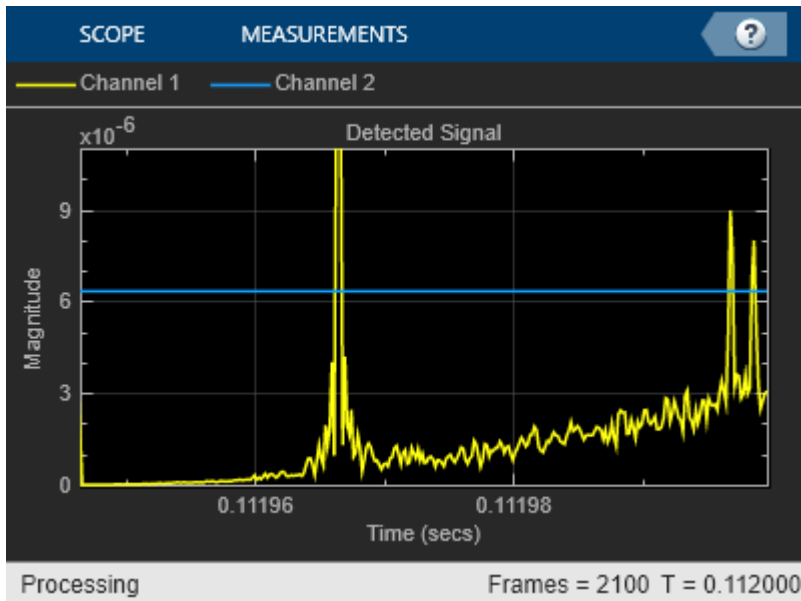
Speedup improvement depends on several factors such as machine CPU speed and available memory but is typically increased 3-4 times. Note that the visualization of data using scopes is not sped up by MATLAB Coder and is still handled by the MATLAB interpreter. If visualizations are not critical to your simulation, then you can remove them for further speed improvement.

Below are several trade-offs to consider when adopting this approach:

1   The visualization capability in generated code is very limited compared to what is available in MATLAB. If you need to keep the visualization in the simulation, use the `coder.extrinsic` trick; but this slows down the simulation.

2   The generated code does not allow dynamic changes of variable type and size compared to the original MATLAB code. The generated code is often optimized for a particular variable type and size; therefore, any change in a variable type and size, which can be caused, for example, by a change in PRF, requires a recompile.

3   The simulation speed benefit becomes more important when the MATLAB simulation time is long. If MATLAB simulation finishes in a few seconds, you do not gain much by generating the code from the original MATLAB simulation. As mentioned in the previous bullet, it is often necessary to recompile the code when parameters change. Therefore, it might be better to first use MATLAB simulation to identify appropriate parameter values and then use generated code to run long simulations.

**Summary**

This example shows how to perform the radar system simulation in streaming mode. It also shows how the code generation can be used to speed up the simulation. The tradeoff between using the generated code and MATLAB code is discussed at the end.

# Scene Visualization for Phased Array System Simulation

This example shows how to use scenario viewer to visualize a system level simulation.

**Introduction**

A phased array system simulation often includes many moving objects. For example, both the array and the targets can be in motion. In addition, each moving object may have its own orientations, so the bookkeeping becomes more and more challenging when more players present in the simulation.

Phased Array System Toolbox™ provides a scenario viewer to help visualize how radars and targets are moving in space. Through the scenario viewer, one can follow the trajectory of each moving platform and examine the relative motion between the radar and the target.

**Visualize Trajectories**

In the first example, the scenario viewer is used to visualize the trajectories of a radar and a target. Assume that the radar is circling around the origin at 3 km away. The plane with radar flies at 250 m/s (about 560 mph), and makes a circle approximately every 60 seconds.

```
v = 250;
deltaPhi = 360/60;
sensormotion = phased.Platform(...
    'InitialPosition',[0;-3000;500],...
    'VelocitySource','Input port',...
    'InitialVelocity',[0;v;0]);
```

The target is traveling along a straight road with a velocity of 30 m/s along the x-axis, which is approximately 67 mph.

```
tgtmotion = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[30;0;0]);
```

The viewer is set to update at every 0.1 second. For the simplest case, the beam is not shown in the viewer.

```
tau = 0.5;
sceneview = phased.ScenarioViewer('ShowBeam','None');
```

This code simulates and displays radar and target's trajectories.

```
for m = 1:tau:60
    [sensorpos,sensorvel] = sensormotion(tau,...
        v*[cosd(m*deltaPhi);sind(m*deltaPhi);0]);
    [tgtpos,tgtvel] = tgtmotion(tau);
    sceneview(sensorpos,sensorvel,tgtpos,tgtvel);
    drawnow;
end
```
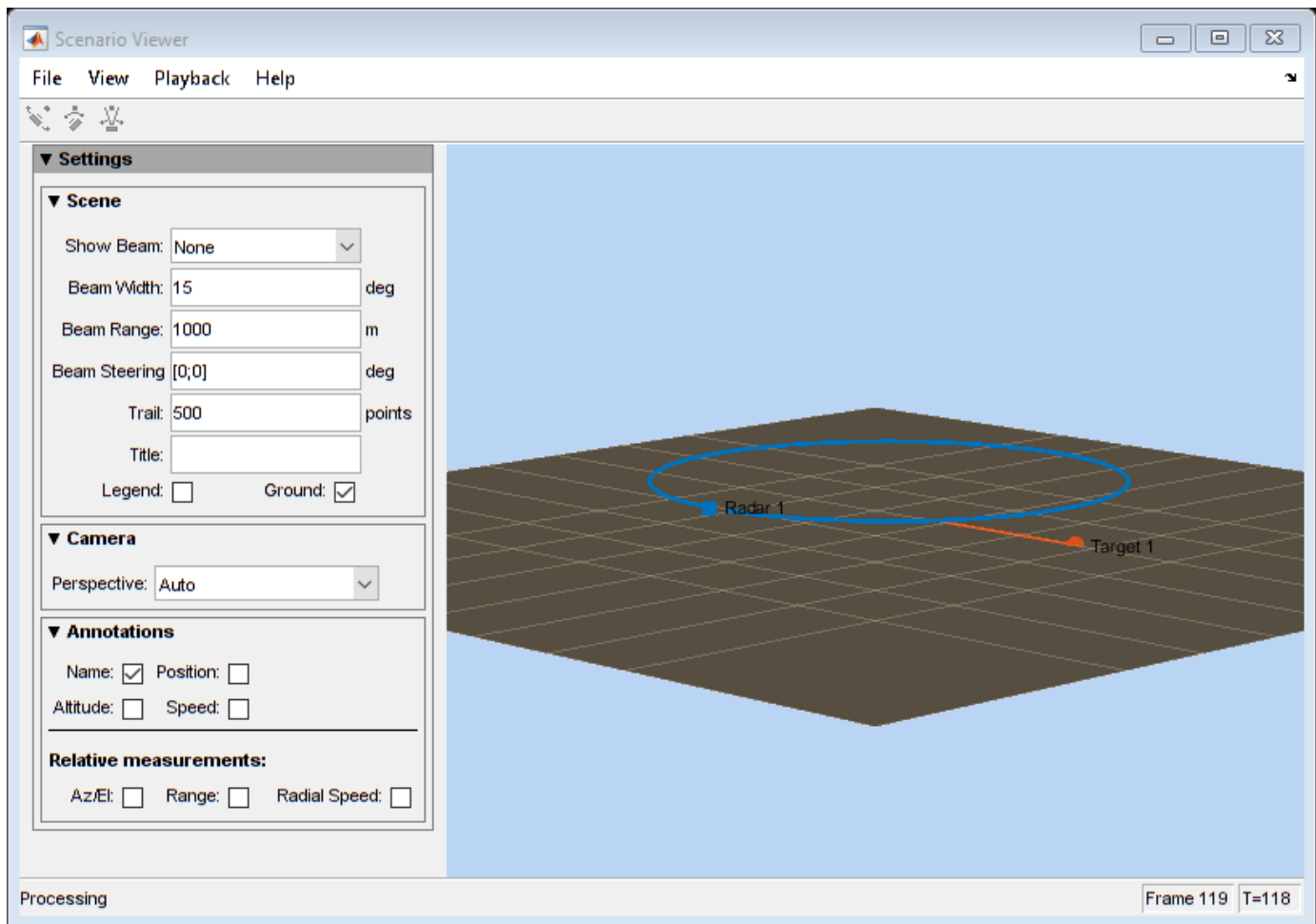
### Visualize Trajectories and Radar Beams

The next natural step is to visualize the antenna array beams together with the trajectories in the viewer. The following example shows how to visualize two radars and three targets moving in space. In particular, the first radar has a beam tracking the first target.

First, setup radars and targets. Note the first radar and the first target match what used in the previous section.

```
sensormotion = phased.Platform(...
    'InitialPosition',[0 0;-3000 500;500 1], ...
    'VelocitySource','Input port', ...
    'InitialVelocity',[0 100;v 0;0 0], ...
    'OrientationAxesOutputPort', true);
tgtmotion = phased.Platform(...
    'InitialPosition',[0 2000.66 3532.63;0 0 500;0 500 500],...
    'Velocity',[30 120 -120; 0 0 -20; 0 0 60],...
    'OrientationAxesOutputPort', true);
```

To properly point the beam, the scenario viewer needs to know the orientation information of radars and targets. Such information can be obtained from these moving platforms by setting the OrientationAxesOutputPort property to true at each simulation step, as shown in the code above. To pass this information to the viewer, set the scenario viewer's OrientationInputPort property to true.

```matlab
sceneview = phased.ScenarioViewer('BeamRange',[3e3 3e3],...
    'BeamWidth',[5 5], ...
    'ShowBeam', 'All', ...
    'CameraPerspective', 'Custom', ...
    'CameraPosition', [-15453.85 -19716.96 13539], ...
    'CameraOrientation', [-47 -27 0]', ...
    'CameraViewAngle', 11.28, ...
    'OrientationInputPort', true, ...
    'UpdateRate',1/tau);
```

Note the displayed beam has a beamwidth of 5 degrees and a length of 3 km. The camera perspective is also adjusted to visualize all trajectories more clearly.

```matlab
for m = 1:60
    [sensorpos,sensorvel,sensoraxis] = sensormotion(tau,...
        [v*[cosd(m*deltaPhi);sind(m*deltaPhi);0] [100; 0; 0]]);
    [tgtpos,tgtvel,tgtaxis] = tgtmotion(tau);
    % Radar 1 tracks Target 1
    [lclrng, lclang] = rangeangle(tgtpos(:,1),sensorpos(:,1),...
        sensoraxis(:,:,1));
    % Update beam direction
    sceneview.BeamSteering = [lclang [0;0]];
    sceneview(sensorpos,sensorvel,sensoraxis,tgtpos,tgtvel,tgtaxis);
    drawnow;
end
```
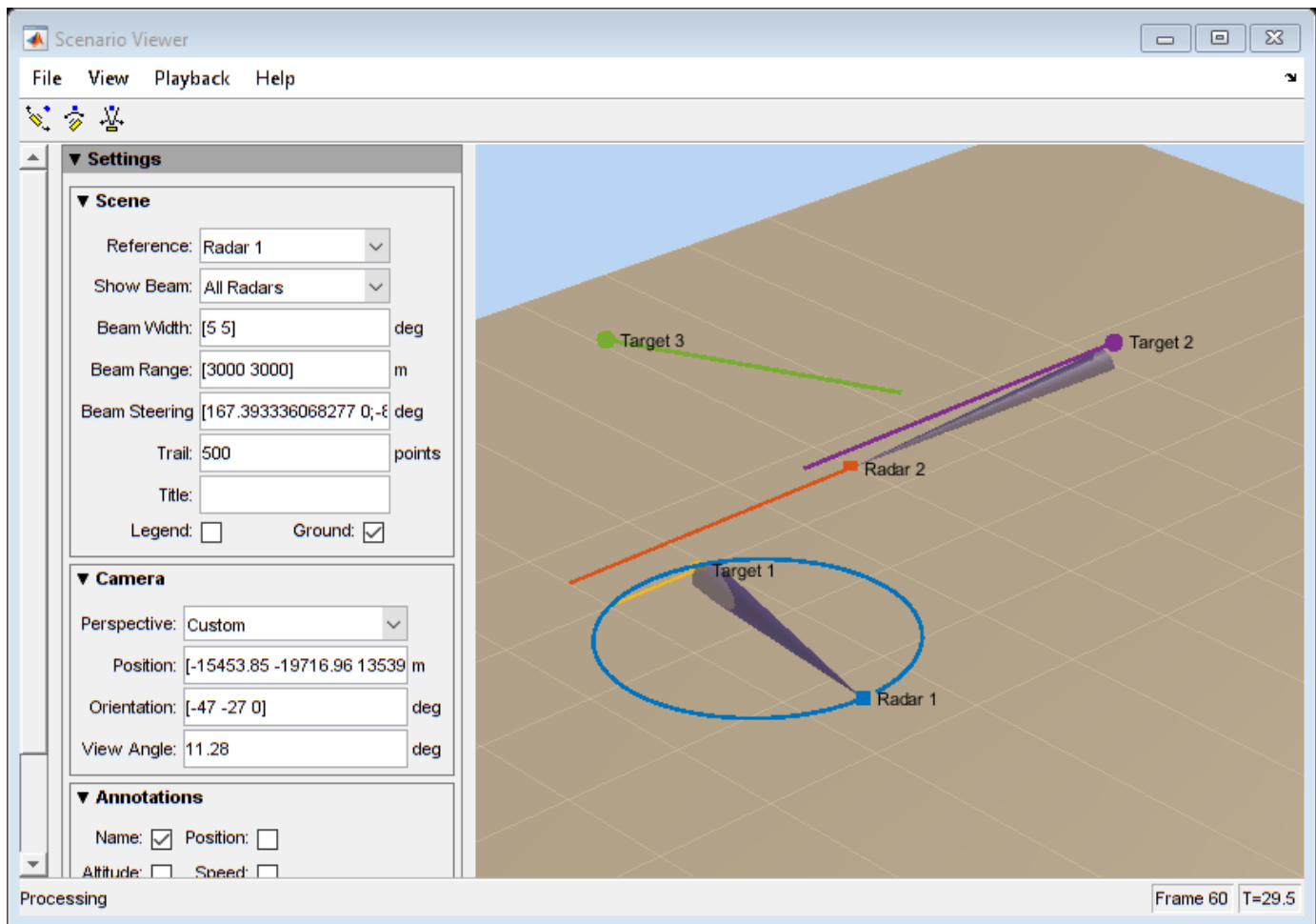
**System Simulation Visualization**

The scenario viewer can also be combined together with other visualizations to provide more information of the system under simulation. Next example uses a scenario viewer together with a range time intensity (RTI) scope and a Doppler time intensity (DTI) scope so an engineer can examine whether the estimated ranges and range rates of targets match the ground truth.

The example uses the radar system created in the "Simulating Test Signals for a Radar Receiver" on page 18-393 example.

```
load BasicMonostaticRadarExampleData.mat
```

Consider the scene where there are three targets.

```
fc = radiator.OperatingFrequency;
fs = waveform.SampleRate;
c = radiator.PropagationSpeed;

sensormotion = phased.Platform(...
    'InitialPosition',[0; 0; 10],...
    'Velocity',[0; 0; 0]);

target = phased.RadarTarget(...
```

```
    'MeanRCS',[1.6 2.2 1.05],...
    'OperatingFrequency',fc);
tgtmotion = phased.Platform(...
    'InitialPosition',[2000.66 3532.63 3845.04; 0 0 0;10 10 10], ...
    'Velocity',[120 -120 0; 0 0 0; 0 0 0]);

channel = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

Once the echo arrives at the receiver, a matched filter and a pulse integrator are used to perform the range estimation.

```
matchingcoeff = getMatchedFilter(waveform);
matchingdelay = size(matchingcoeff,1)-1;
matchedfilter = phased.MatchedFilter(...
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);

prf = waveform.PRF;
fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
rangeGates = c*fast_time_grid/2;

lambda = c/fc;
max_range = 5000;
tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(rangeGates,lambda),...
    'ReferenceLoss',2*fspl(max_range,lambda));

num_pulse_int = 10;
```

Because it is unnecessary to monitor the trajectory at the pulse repetition rate, this example assumes that the system reads the radar measurement at a rate of 20 Hz. The example uses a scenario viewer to monitor the scene and a range time intensity (RTI) plot as well as a Doppler time intensity (DTI) plot to examine the estimated range and range rate value.

```
r_update = 20;

sceneview = phased.ScenarioViewer('UpdateRate',r_update,...
    'Title','Monostatic Radar with Three Targets');

rtiscope = phased.IntensityScope('Name','Range-Time Intensity Scope',...
    'XLabel','Range (m)', ...
    'XResolution',c/(2*fs), ...
    'XOffset',-(matchingdelay-1)*c/(2*fs), ...
    'TimeResolution',1/r_update,'TimeSpan',5,'IntensityUnits','dB');

nfft = 128;
df = prf/nfft;
dtiscope = phased.IntensityScope(...
    'Name','Doppler-Time Intensity Scope',...
    'XLabel','Velocity (m/sec)', ...
    'XResolution',dop2speed(df,lambda)/2, ...
    'XOffset', dop2speed(-prf/2,lambda)/2, ...
    'TimeResolution',1/r_update,'TimeSpan',5,'IntensityUnits','dB');
```

The next section performs the system simulation and produces the visualizations.

```matlab
% Pre-allocate array for improved processing speed
rxpulses = zeros(numel(rangeGates),num_pulse_int);

for k = 1:100
    for m = 1:num_pulse_int
        % Update sensor and target positions
        [sensorpos,sensorvel] = sensormotion(1/prf);
        [tgtpos,tgtvel] = tgtmotion(1/prf);

        % Calculate the target angles as seen by the sensor
        [~,tgtang] = rangeangle(tgtpos,sensorpos);

        % Simulate propagation of pulse in direction of targets
        pulse = waveform();
        [txsig,txstatus] = transmitter(pulse);
        txsig = radiator(txsig,tgtang);
        txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

        % Reflect pulse off of targets
        tgtsig = target(txsig);

        % Receive target returns at sensor
        rxsig = collector(tgtsig,tgtang);
        rxpulses(:,m) = receiver(rxsig,~(txstatus>0));
    end

    rxpulses = matchedfilter(rxpulses);

    % Correct for matched filter delay
    rxpulses = buffer(...
        rxpulses(matchingdelay+1:end),...
        size(rxpulses,1));

    rxpulses = tvg(rxpulses);

    rx_int = pulsint(rxpulses,'noncoherent');

    % display RTI
    rtiscope(rx_int);

    % display DTI
    rx_dop = mean(fftshift(...
        abs(fft(rxpulses,nfft,2)),2));
    dtiscope(rx_dop.');

    % display scene
    sceneview(sensorpos,sensorvel,tgtpos,tgtvel);

    % perform next detection when next update is needed
    sensormotion(1/r_update);
    tgtmotion(1/r_update);
end

hide(dtiscope);
hide(rtiscope);
```
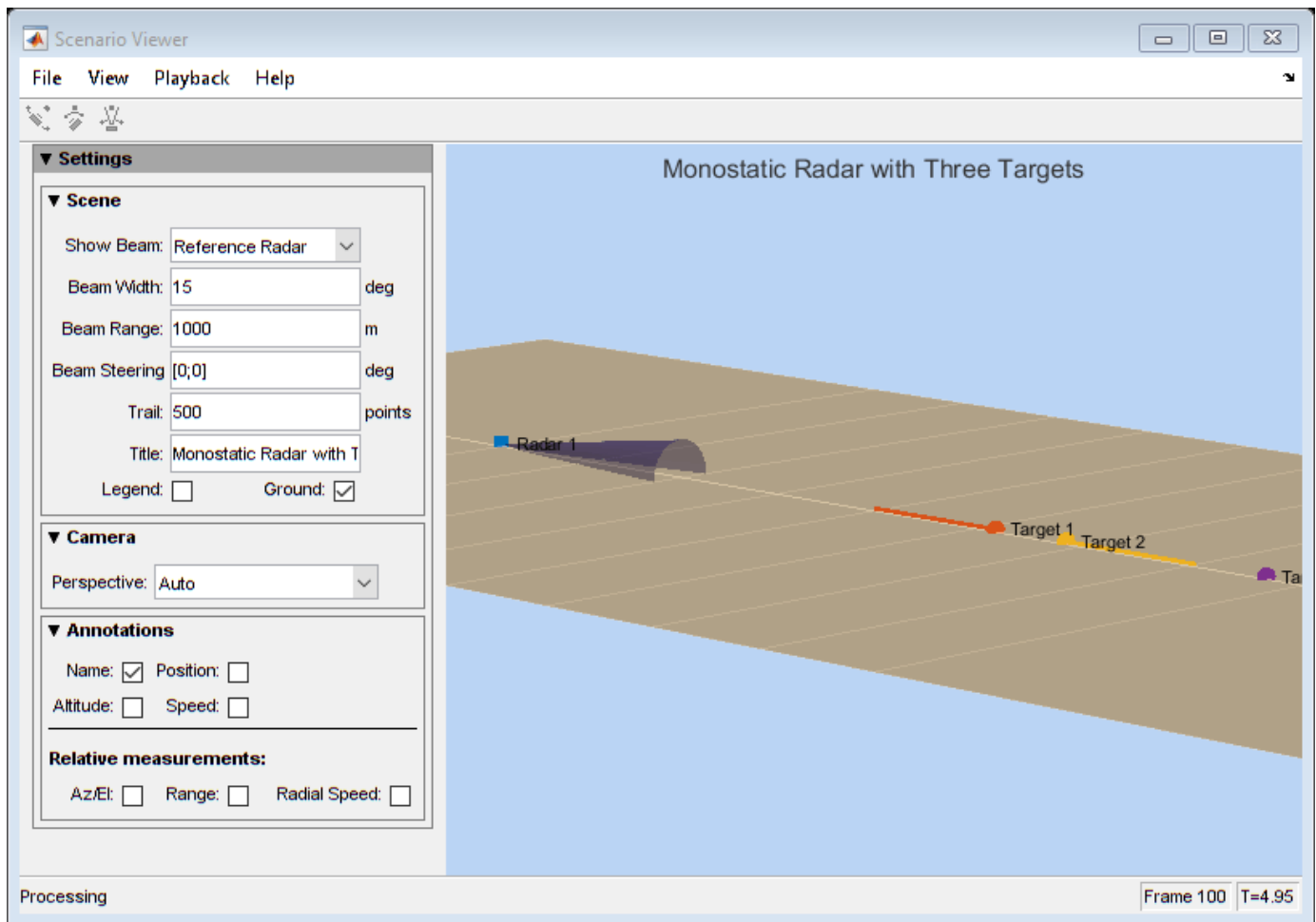
```
hide(sceneview);
show(rtiscope);
```
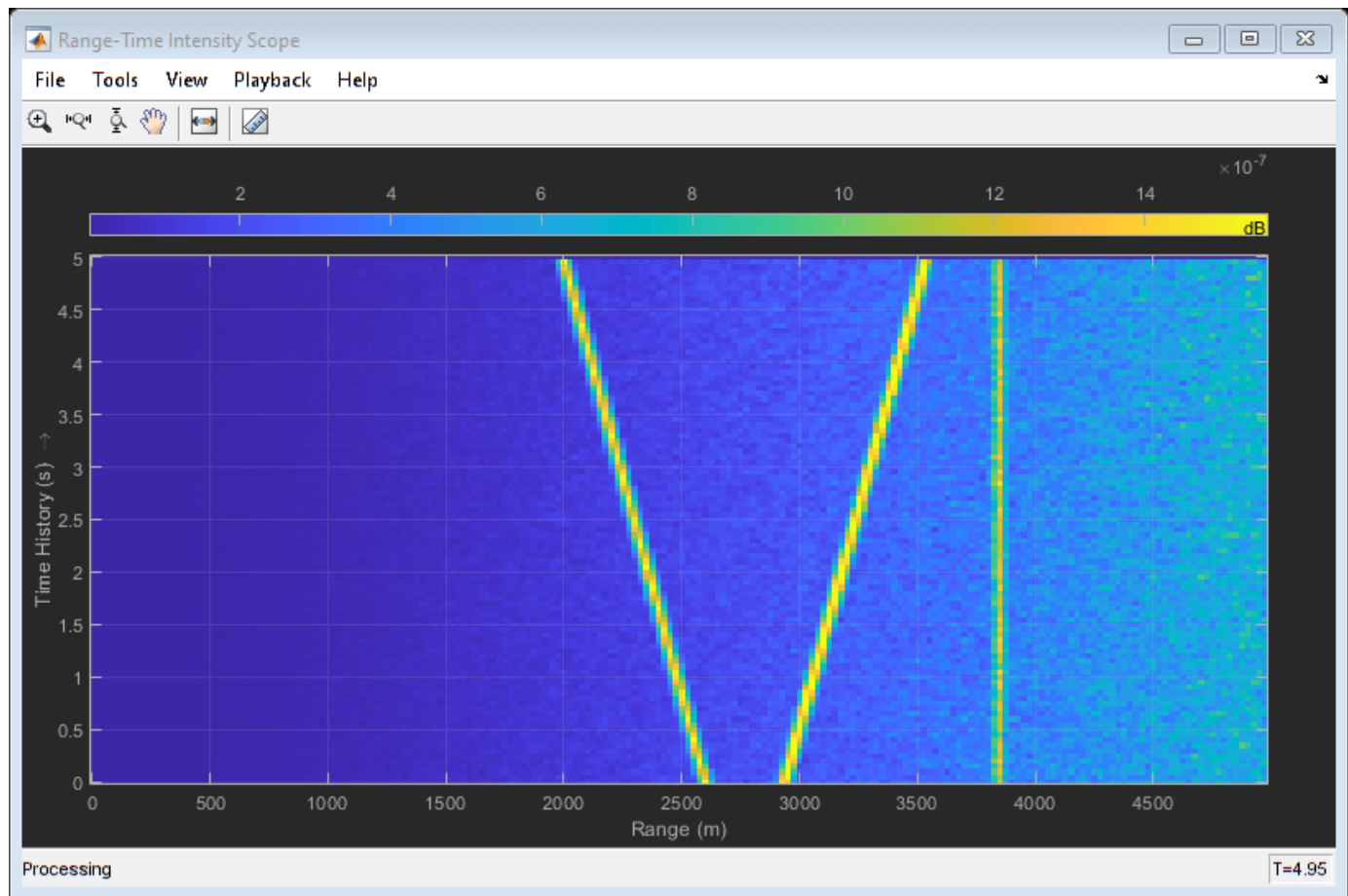
Both the scenario viewer and the RTI are updated during the simulation so one can easily verify whether the simulation is running as expected and whether the range estimation matches the ground truth while the simulation is running.

```
hide(rtiscope);
show(dtiscope);
```
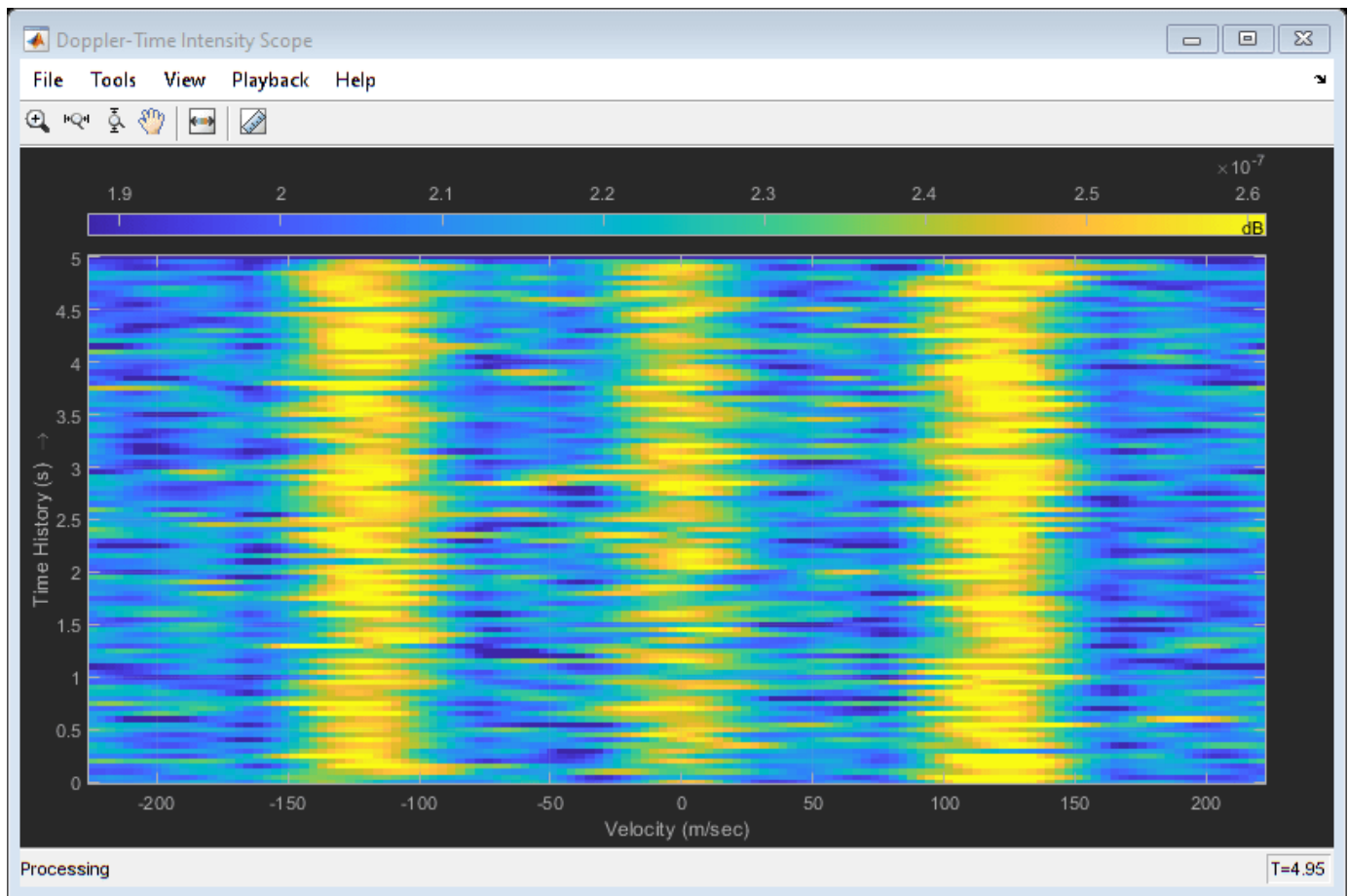
Similarly, the DTI provides the range rate estimates of each target.

**Conclusions**

This example describes different ways to visualize the radar and target's trajectories. Such visualizations help provide an overall picture of the system dynamics.

# Simulating Test Signals for a Radar Receiver in Simulink

This example shows how to model an end-to-end monostatic radar using Simulink®. A monostatic radar consists of a transmitter colocated with a receiver. The transmitter generates a pulse which hits the target and produces an echo received by the receiver. By measuring the time location of the echoes, you can estimate the range of the target. The first part of this example demonstrates how to detect the range of a single target using the equivalent of a single element antenna. The second part of the example will show how to build a monostatic radar with a 4-element uniform linear array (ULA) that detects the range of 4 targets.

**Available Example Implementations**

This example includes two Simulink® models:

- Monostatic Radar with One Target: slexMonostaticRadarExample.slx
- Monostatic ULA Radar with Four Targets: slexMonostaticRadarMultipleTargetsExample.slx

**Monostatic Radar with One Target**

This model simulates a simple end-to-end monostatic radar. Using the transmitter block without the narrowband transmit array block is equivalent to modeling a single isotropic antenna element. Rectangular pulses are amplified by the transmitter block then propagated to and from a target in free-space. Noise and amplification are then applied in the receiver preamp block to the return signal, followed by a matched filter. Range losses are compensated for and the pulses are noncoherently integrated. Most of the design specifications are derived from the "Simulating Test Signals for a Radar Receiver" on page 18-393 example provided for System objects.
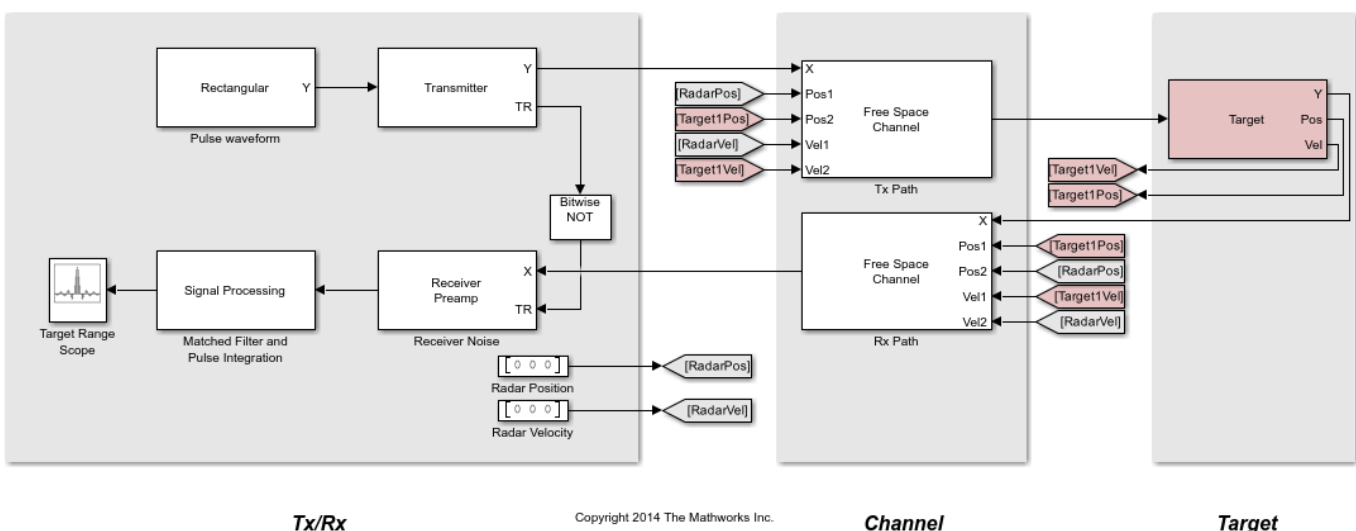


The model consists of a transceiver, a channel, and a target. The blocks that corresponds to each section of the model are:

**Transceiver**

- **Rectangular** - Creates rectangular pulses.
- **Transmitter** - Amplifies the pulses and sends a Transmit/Receive status to the **Receiver Preamp** block to indicate if it is transmitting.
- **Receiver Preamp** - Receives the pulses from free space when the transmitter is off. This block also adds noise to the signal.
- **Constant** - Used to set the position and velocity of the radar. Their values are received by the **Freespace** blocks using the **Goto** and **From**.
- **Signal Processing** - Subsystem performs match filtering and pulse integration.
- **Target Range Scope** - Displays the integrated pulse as a function of the range.

**Signal Processing Subsystem**



- **Matched Filter** - Performs match filtering to improve SNR.
- **TVG** - Time varying gain to compensate for range loss.
- **Pulse Integrator** - Integrates several pulses noncoherently.

**Channel**

- **Freespace** - Applies propagation delays, losses and Doppler shifts to the pulses. One block is used for the transmitted pulses and another one for the reflected pulses. The **Freespace** blocks require the positions and velocities of the radar and the target. Those are supplied using the **Goto** and **From** blocks.

**Target**

- **Target** - Subsystem reflects the pulses according to the specified RCS. This subsystem includes a **Platform** block that models the speed and position of the target which are supplied to the **Freespace** blocks using the **Goto** and **From** blocks. In this example the target is stationary and positioned 1998 meters from the radar.

**Exploring the Example**

Several dialog parameters of the model are calculated by the helper function helperslexMonostaticRadarParam. To open the function from the model, click on **Modify Simulation Parameters** block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

**Results and Displays**

The figure below shows the range of the target. Target range is computed from the round-trip delay of the reflected pulse. The delay is measured from the peak of the matched filter output. We can see that the target is approximately 2000 meters from the radar. This range is within the radar's 50-meter range resolution from the actual range.
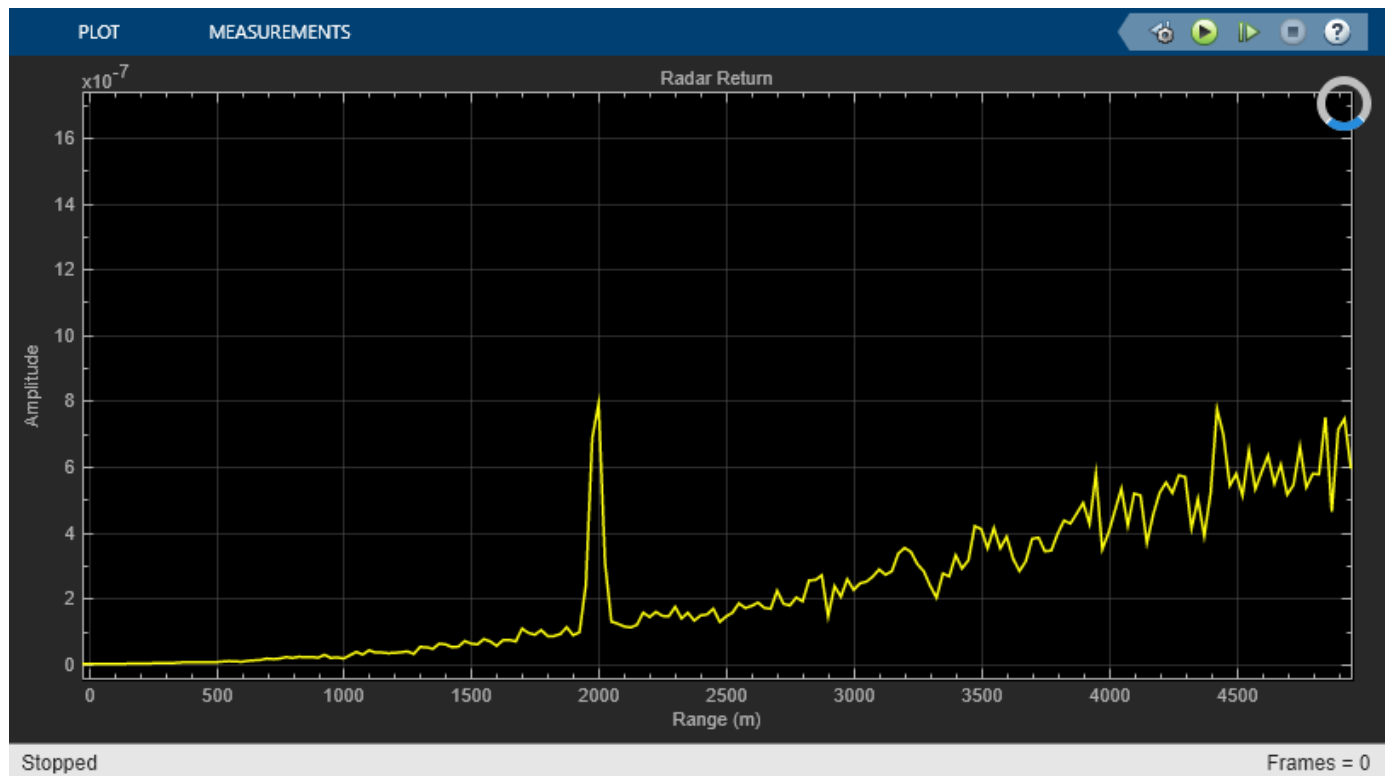
**Monostatic Radar with Multiple Targets**

This model estimates the range of four stationary targets using a monostatic radar. The radar transceiver uses a 4-element uniform linear antenna array (ULA) for improved directionality and gain. A beamformer is also included in the receiver. The targets are positioned at 1988, 3532, 3845 and 1045 meters from the radar.
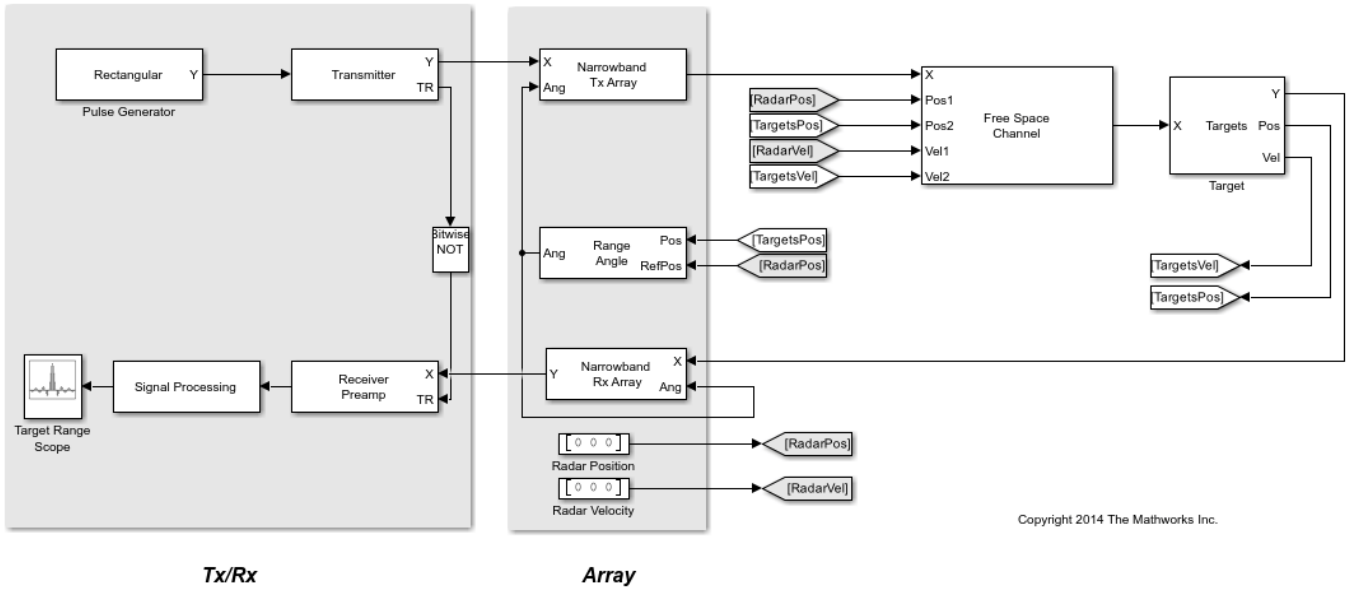
The blocks added to the previous example are:

- **Narrowband Tx Array** - Models an antenna array for transmitting narrowband signals. The antenna array is configured using the "Sensor Array" tab of the block's dialog panel. The **Narrowband Tx Array** block models the transmission of the pulses through the antenna array in the four directions specified using the **Ang** port. The output of this block is a matrix of four columns. Each column corresponds to the pulses propagated towards the directions of the four targets.
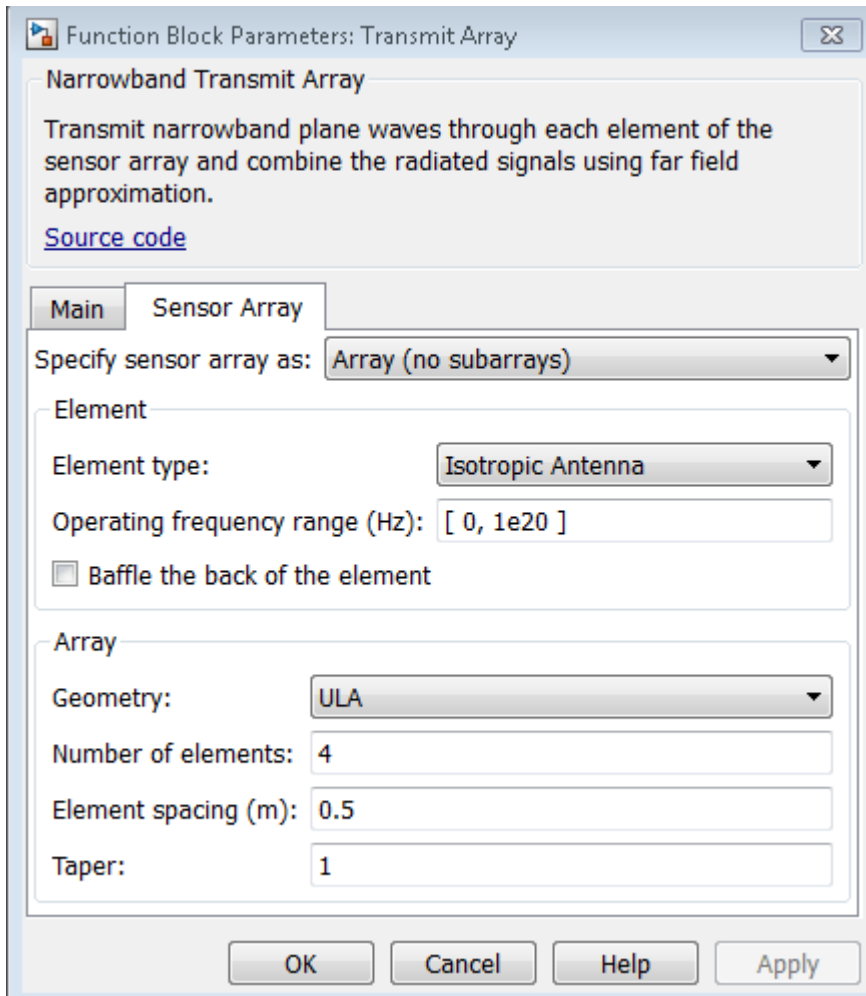
- **Narrowband Rx Array** - Models an antenna array for receiving narrowband signals. The array is configured using the "Sensor Array" tab of the block's dialog panel. The block receives pulses from the four directions specified using the Ang port. The input of this block is a matrix of four columns. Each column corresponds to the pulses propagated from the direction of each target. The output of the block is a matrix of 4 columns. Each column corresponds to the signal received at each antenna element.

- **Range Angle** - Calculates the angles between the radar and the targets. The angles are used by the **Narrowband Tx Array** and the **Narrowband Rx Array** blocks to determine in which directions to model the pulses' transmission or reception.

- **Phase Shift Beamformer** - Beamforms the output of the **Receiver Preamp**. The input to the beamformer is a matrix of 4 columns, one column for the signal received at each antenna element. The output is a beamformed vector of the received signal.

This example illustrates how to use single **Platform**, **Freespace** and **Target** blocks to model all four round-trip propagation paths. In the **Platform** block, the initial positions and velocity parameters are specified as three-by-four matrices. Each matrix column corresponds to a different target. Position and velocity inputs to the **Freespace** block come from the outputs of the **Platform** block as three-by-four matrices. Again, each matrix column corresponds to a different target. The signal inputs and outputs of the **Freespace** block have four columns, one column for the propagation

path to each target. The `Freespace` block has two-way propagation setting enabled. The "Mean radar cross section" (RCS) parameter of the `Target` block is specified as a vector of four elements representing the RCS of each target.

**Exploring the Example**

Several dialog parameters of the model are calculated by the helper function helperslexMonostaticRadarMultipleTargetsParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

**Results and Displays**

The figure below shows the detected ranges of the targets. Target ranges are computed from the round-trip time delay of the reflected signals from the targets. We can see that the targets are approximately 2000, 3550, and 3850 meters from the radar. These results are within the radar's 50-meter range resolution from the actual range.
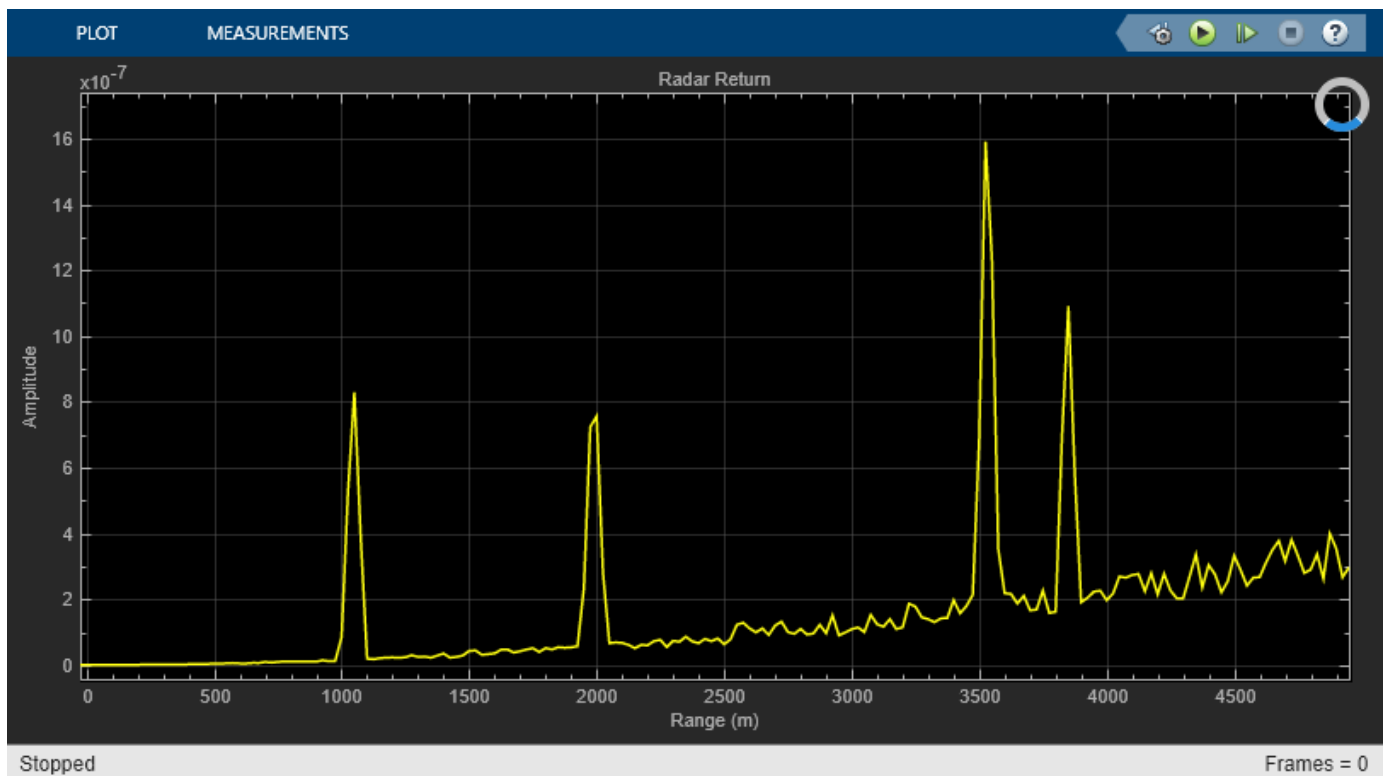
# Modeling RF Front End in Radar System Simulation

In a radar system, the RF front end often plays an important role in defining the system performance. For example, because the RF front end is the first section in the receiver chain, the design of its low noise amplifier is critical to achieving the desired signal to noise ratio (SNR). This example shows how to incorporate RF front end behavior into an existing radar system design.

This example requires RF Blockset™.

**Available Example Implementations**

This example includes two Simulink® models:

- Monostatic Radar with One Target: slexMonostaticRadarRFExample.slx
- FMCW Radar Range and Speed Estimation: slexFMCWRFExample.slx

**Introduction**

Several examples, such as "Simulating Test Signals for a Radar Receiver in Simulink" on page 18-434 and "Automotive Adaptive Cruise Control Using FMCW and MFSK Technology" (Radar Toolbox) have shown that one can build end-to-end radar systems in Simulink using Phased Array System Toolbox™. In many cases, once the system model is built, the next step could be adding more fidelity in different components. A popular candidate for such a component is the RF front end. One advantage of modeling the system in Simulink is the capability of performing multidomain simulations.

The following sections show two examples of incorporating RF Blockset modeling capability in radar systems built with Phased Array System Toolbox.

**Monostatic Radar with One Target**

The first model is adapted from example "Simulating Test Signals for a Radar Receiver in Simulink" on page 18-434 which simulates a monostatic pulse radar with one target. From the diagram itself, the model below looks identical to the model shown in that example.

Monostatic Radar with One Target
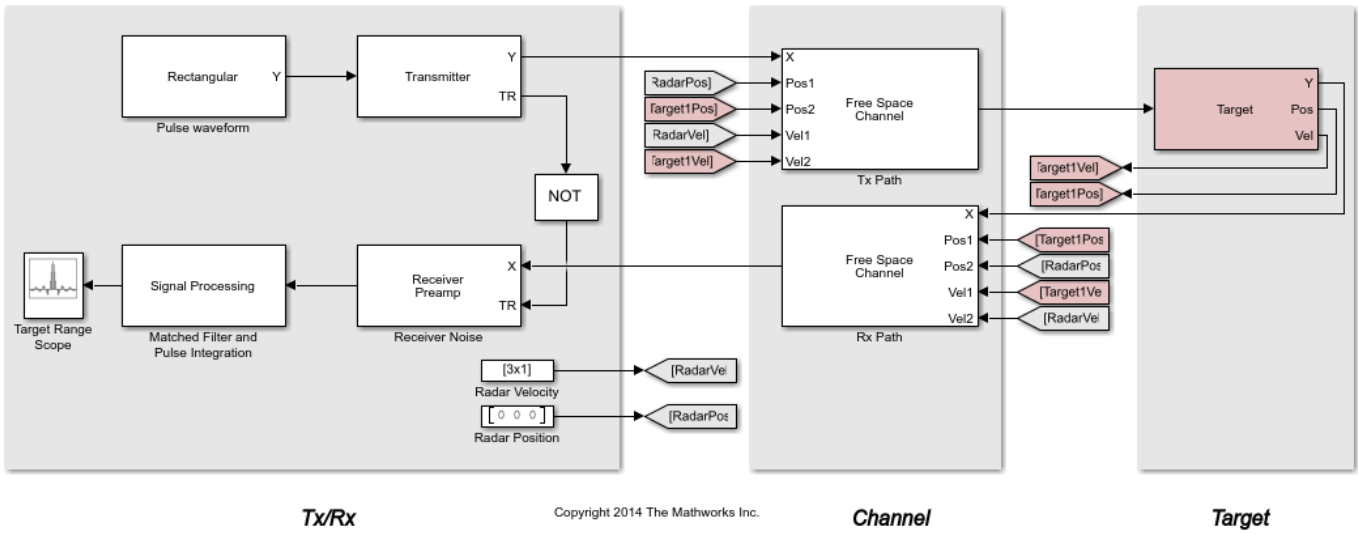
When the model is executed, the resulting plot is also the same.

However, a deeper look in the transmitter subsystem shows that now the transmitter is modeled by power amplifiers from RF Blockset.
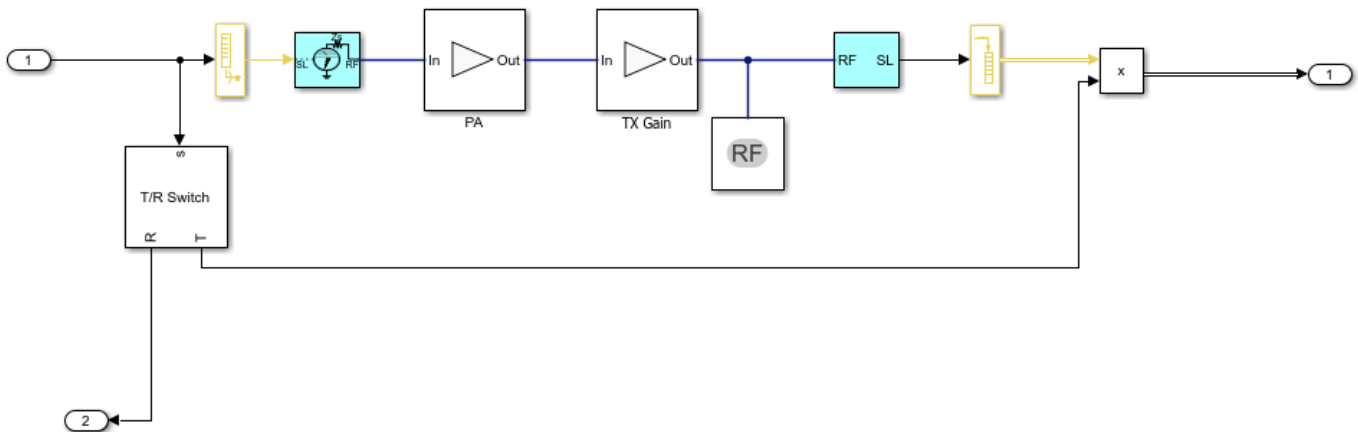


Similar changes are also implemented in the receiver side.
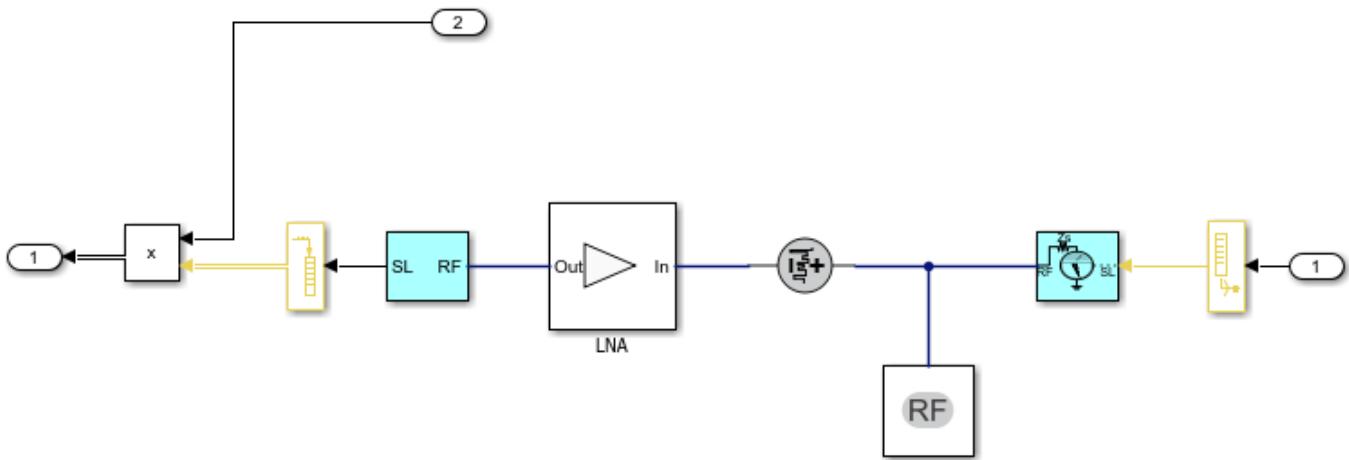
With these changes, the model is capable of simulating RF behaviors. For example, the simulation result shown above assumes a perfect power amplifier. In real applications, the amplifier will suffer many nonlinearities. If one sets the IP3 of the transmitter to 70 dB and runs the simulation again, the peak corresponding to the target is no longer as dominant. This gives the engineer some knowledge regarding the system's performance under different situations.

**FMCW Radar Range and Speed Estimation**

The second example is adapted from "Automotive Adaptive Cruise Control Using FMCW and MFSK Technology" (Radar Toolbox). However, this model uses a triangle sweep waveform instead so the system can estimate range and speed simultaneously. At the top level, the model is similar to what gets built from Phased Array System Toolbox. Once executed, the model shows the estimated range and speed values that matches the distance and relative speed of the target car.

However, similar to the first example, the transmitter and receiver subsystems are now built with RF Blockset blocks.

The following figure shows the transmitter subsystem.
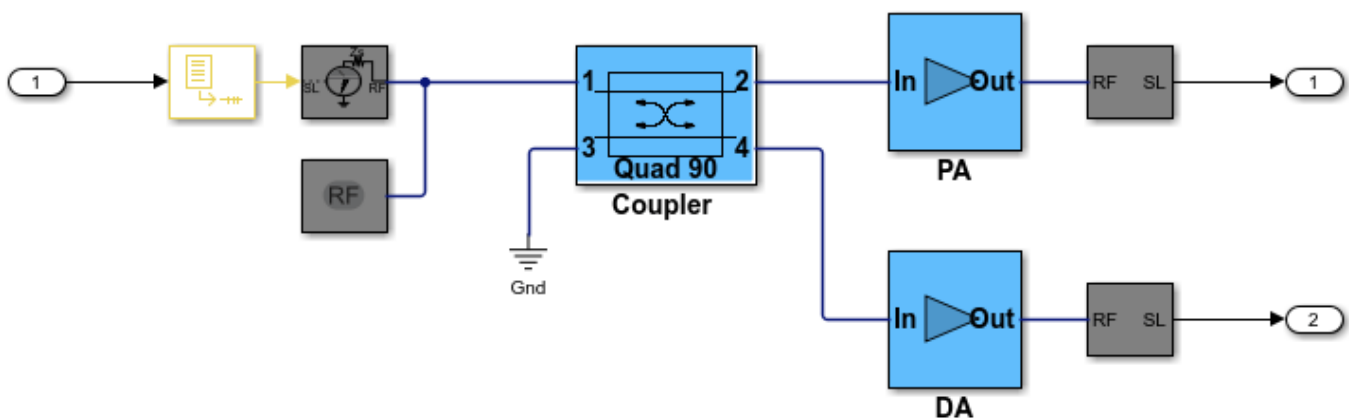


The following figure shows the receiver subsystem.
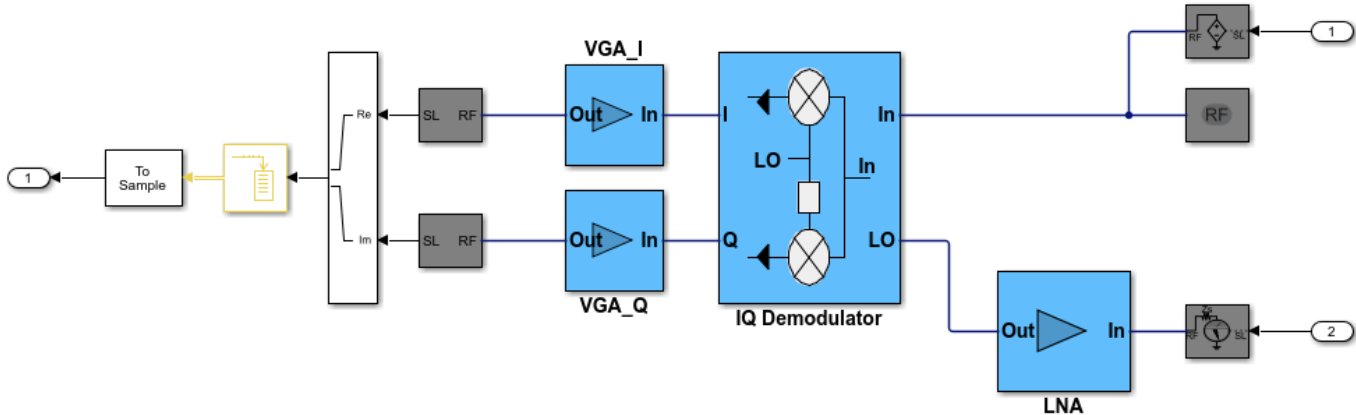
In a continuous wave radar system, part of the transmitted waveform is used as a reference to dechirp the received target echo. From the diagrams above, one can see that the transmitted waveform is sent to the receiver via a coupler and the dechirp is performed via an I/Q demodulator. Therefore, by adjusting parameters in those RF components, higher simulation fidelity can be achieved.

**Summary**

This example shows two radar models that are originally built with Phased Array System Toolbox and later incorporated RF models from RF Blockset. The simulation fidelity is greatly improved by combining the two products together.

# Simulating a Bistatic Radar with Two Targets

This example shows how to simulate a bistatic radar system with two targets. The transmitter and the receiver of a bistatic radar are not co-located and move along different paths.

**Exploring the Example**

The following model shows an end-to-end simulation of a bistatic radar system. The system is divided into three parts: the transmitter subsystem, the receiver subsystem, and the targets and their propagation channels. The model shows the signal flowing from the transmitter, through the channels to the targets and reflected back to the receiver. Range-Doppler processing is then performed at the receiver to generate the range-Doppler map of the received echoes.
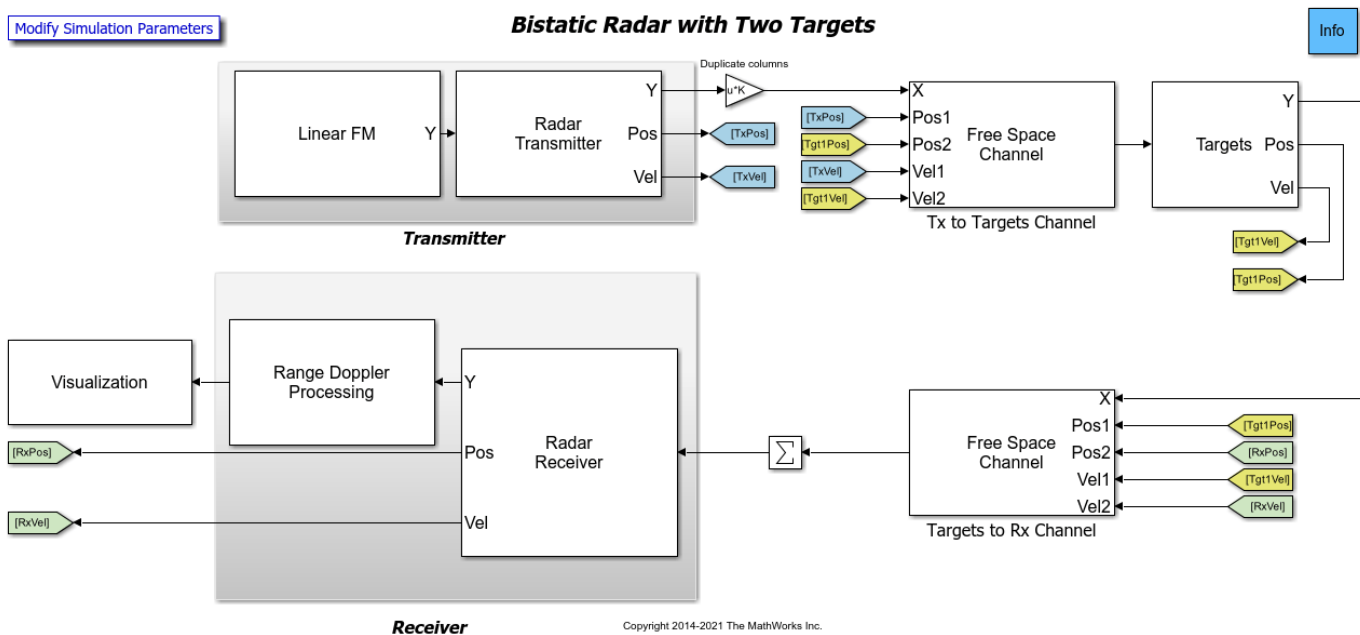


**Transmitter**

- `Linear FM` - Creates linear FM pulse as the transmitter waveform. The signal sweeps a 3 MHz bandwidth, corresponding to a 50-meter range resolution.
- `Radar Transmitter` - Amplifies the pulse and simulates the transmitter motion. In this case, the transmitter is mounted on a stationary platform located at the origin. The operating frequency of the transmitter is 300 MHz.
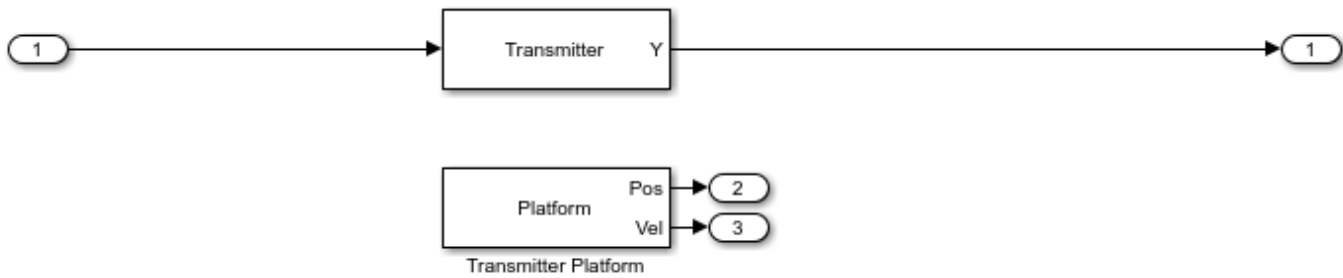
**Targets**

This example includes two targets with similar configurations. The targets are mounted on the moving platforms.

- `Tx to Targets Channel` - Propagates signal from the transmitter to the targets. The signal inputs and outputs of the channel block have two columns, one column for the propagation path to each target.

- `Targets to Rx Channel` - Propagates signal from the targets to the receiver. The signal inputs and outputs of the channel block have two columns, one column for the propagation path from each target.

- `Targets` - Reflects the incident signal and simulates both targets motion. This first target with an RCS of 2.5 square meters is approximately 15 km from the transmitter and is moving at a speed of 141 m/s. The second target with an RCS of 4 square meters is approximately 35 km from the transmitter and is moving at a speed of 168 m/s. The RCS of both targets are specified as a vector of two elements in the Mean radar cross section parameter of the underlying Target block.
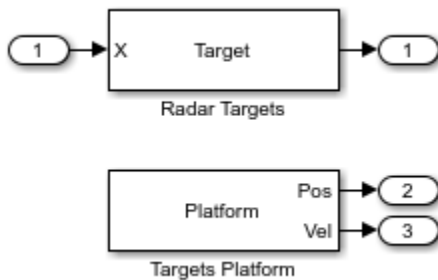


**Receiver**

- `Radar Receiver` - Receives the target echo, adds receiver noise, and simulates the receiver motion. The distance between the transmitter and the receiver is 20 km, and the receiver is moving at a speed of 20 m/s. The distance between the receiver and the two targets are approximately 5 km and 15 km, respectively.

Receiver Platform
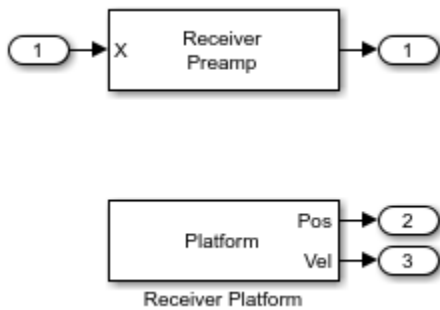
- `Range-Doppler Processing` - Computes the range-Doppler map of the received signal. The received signal is buffered to form a 64-pulse burst which is then passed to a range-Doppler processor. The processor performs a matched filter operation along the range dimension and an FFT along the Doppler dimension.
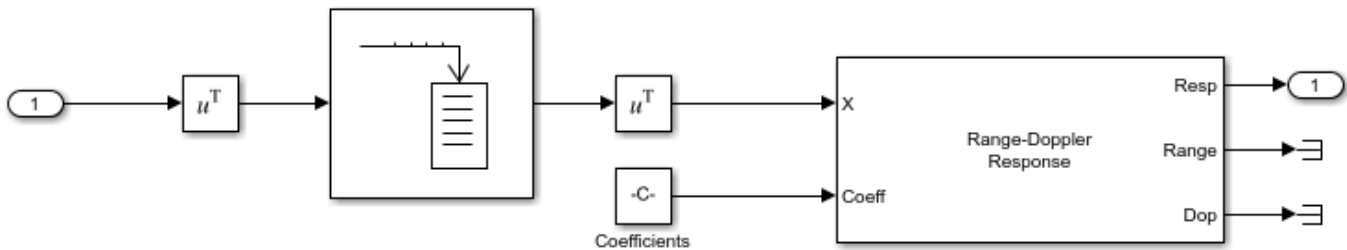


### Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexBistaticParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

### Results and Displays

The figure below shows the two targets in the range-Doppler map, range-time intensity plot and Doppler-time intensity plot.
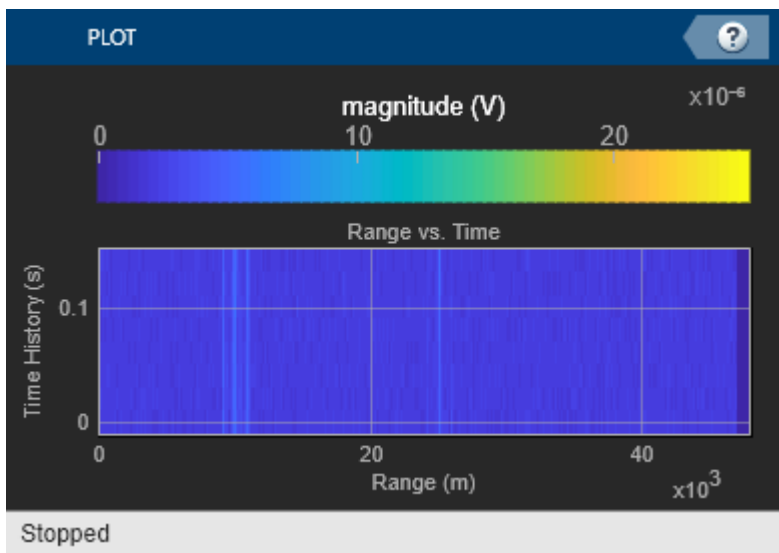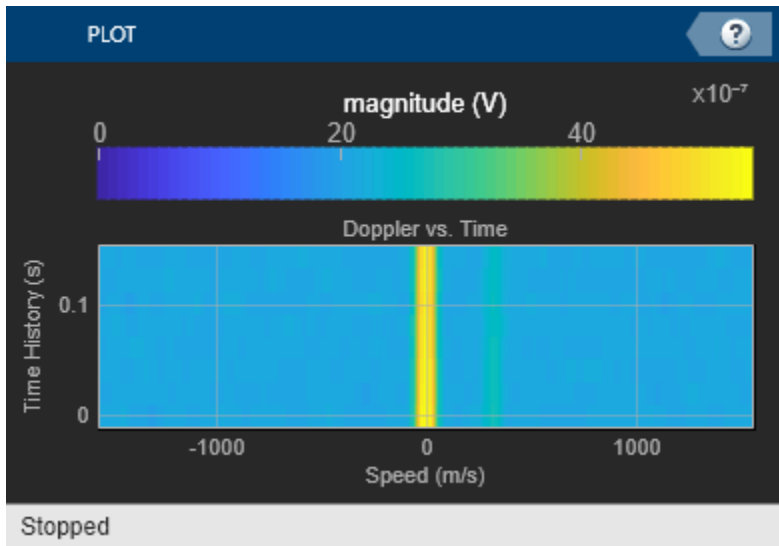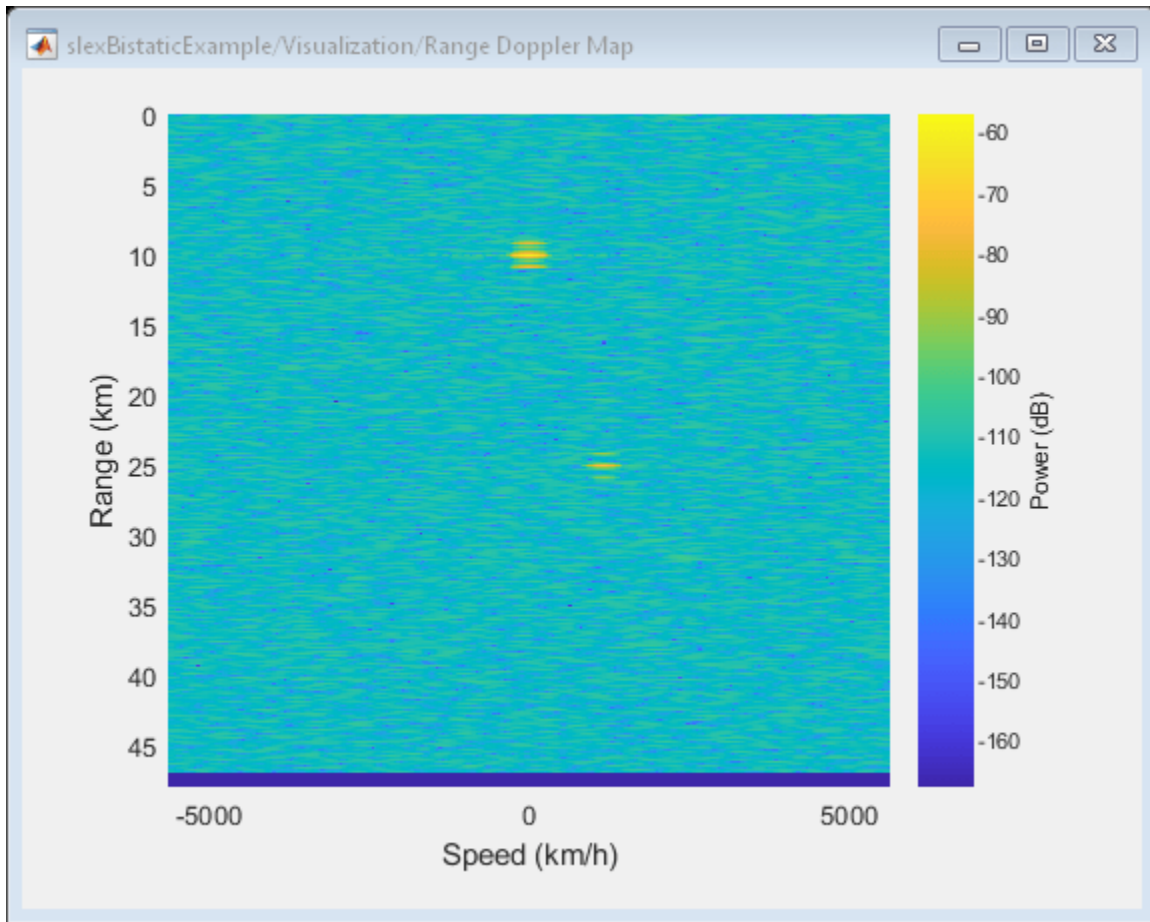
Because this is a bistatic radar, the range-Doppler map above actually shows the target range as the arithmetic mean of the distances from the transmitter to the target and from the target to the receiver. Therefore, the expected range of the first target is approximately 10 km, ((15+5)/2) and for second target approximately 25 km, ((35+15)/2). The range-Doppler map shows these two values as the measured values.

Similarly, the Doppler shift of a target in a bistatic configuration is the sum of the target's Doppler shifts relative to the transmitter and the receiver. The relative speeds to the transmitter are -106.4 m/s for the first target and 161.3 m/s for the second target while the relative speeds to the receiver are 99.7 m/s for the first target and 158.6 m/s for second target. Thus, the range-Doppler map shows the overall relative speeds as -6.7 m/s (-24 km/h) and 319.9 m/s (1152 km/h) for the first target and the second target, respectively, which agree with the expected sum values.

The Range-Time Intensity scope and Doppler-Time Intensity scope shows the scrolling range and Doppler response intensities over time which matches the measured values range and speed values.
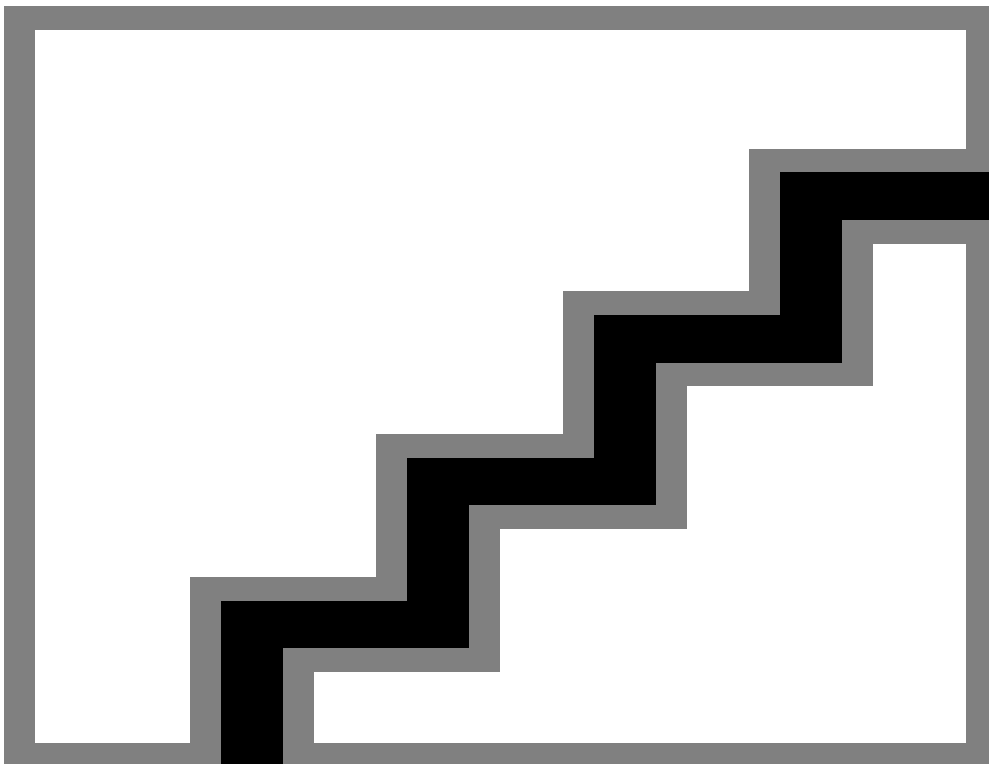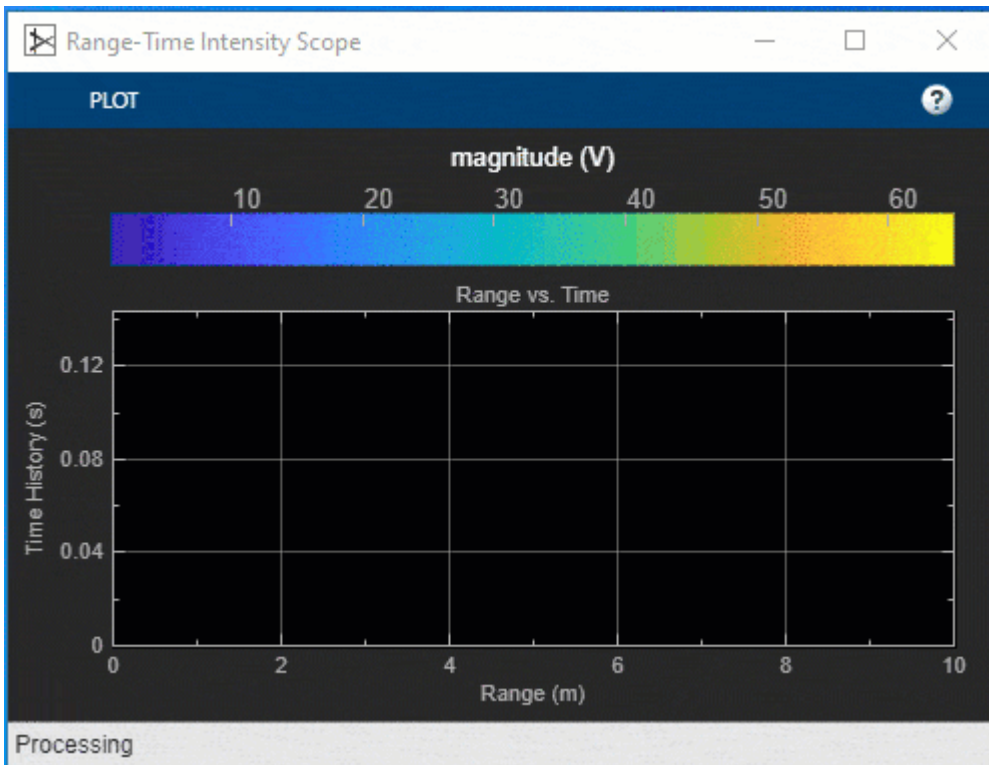
**Summary**

This example shows an end-to-end bistatic radar system simulation with two targets. It explains how to analyze the target return by plotting a range-Doppler map.

# Waveform Scheduling Based on Target Detection

In radar operation, it is often necessary to adjust the operation mode based on the target return. This example shows how to model a radar that changes its pulse repetition frequency (PRF) based on the radar detection.

This example requires SimEvents®.

**Available Example Implementations**

This example includes one Simulink® model:

- Dynamic PRF Selection Based on Radar Detection: slexPRFSelectionSEExample.slx

**Dynamic PRF Selection Based on Radar Detection**

This model simulates a monostatic radar that searches for targets with an unambiguous range of 5 km. If the radar detects a target within 2 km, then it will switch to a higher PRF to only look for targets with 2 km range and enhance its capability to detect high speed targets.
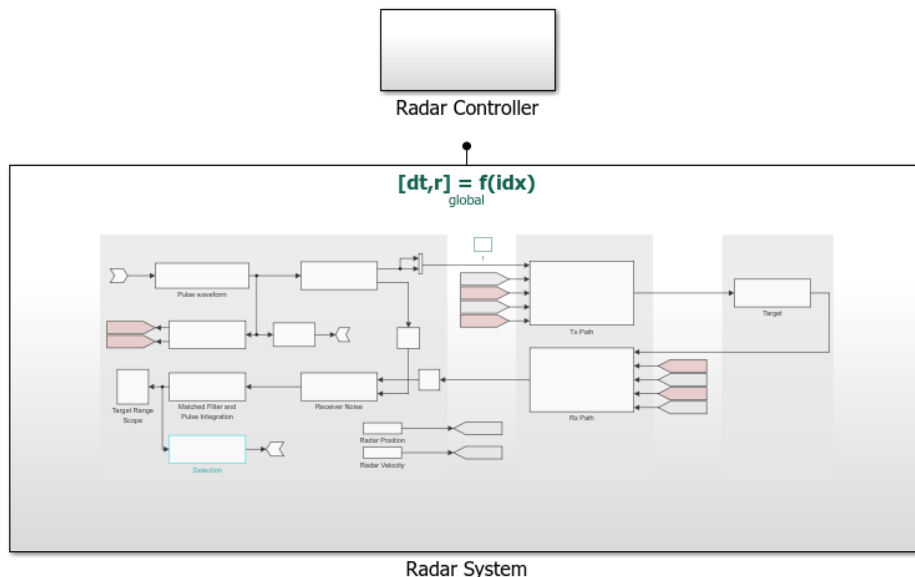


The model consists of two main subsystem, a radar system and its corresponding controller. From the top level, the radar system resides in a Simulink function block. Note that the underlying function is specified in the figure as $[dt,r] = f(idx)$. This means that the radar takes one input, $idx$, which specifies the index of selected PRF of the transmitted signal and returns two outputs: $dt$, the time the next pulse should be transmitted and $r$, the detected target range of the radar system. The radar controller, shown in the following figure, uses the detection and the time to schedule when and what to transmit next.
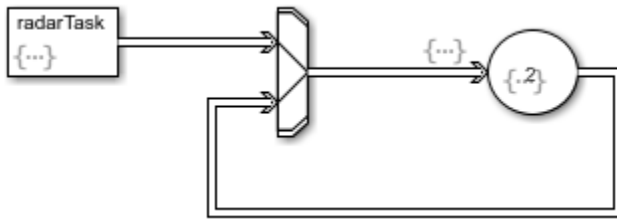
**Radar System**

The radar system resides in a Simulink function block and is shown in the following figure.
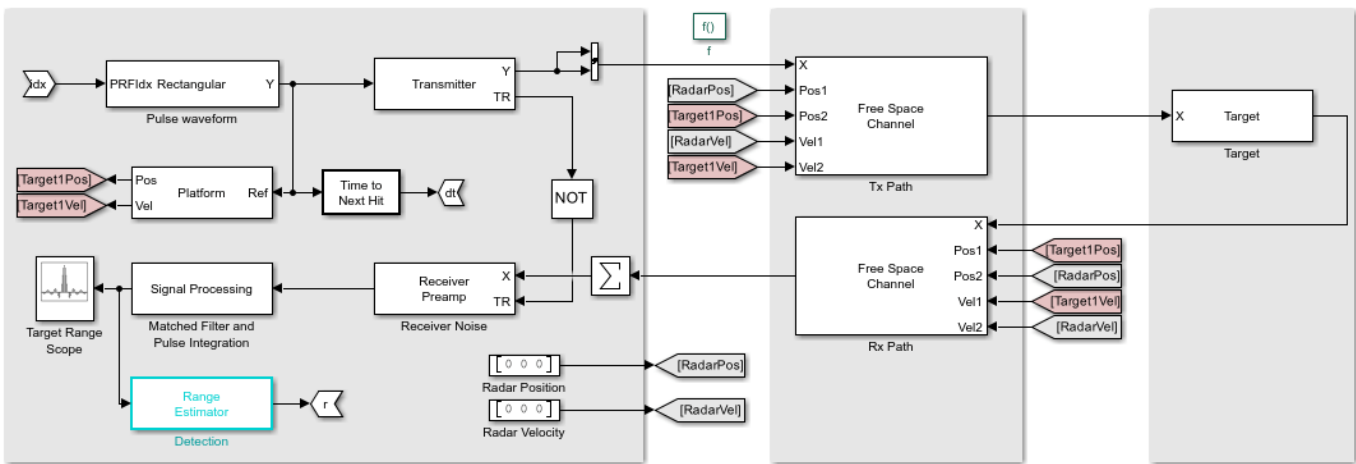


The system is very similar to what is used in the "Waveform Scheduling Based on Target Detection" on page 18-451 example with the following notable difference:

1  The waveform block is no longer a source block. Instead, it takes an input, `idx`, to select which PRF to use. The available PRF values are specified in the PRF parameter of the waveform dialog.

2  The output of the waveform is also used to compute the time, `dt`, that the next pulse should be transmitted. Note that in this case, the time interval is proportional to the length of the transmitted signal.

3  At the end of the signal processing chain, the target range is estimated and returned in `r`. The controller will use this information to decide which PRF to choose for next transmission.

4  Once the model is compiled, notice that the signal passing through the system can vary in length because of a possible change of the waveform PRF. In addition, because the sample rate cannot be derived inside a Simulink function subsystem, the sample rate is specified in the block diagrams, such as the Tx and Rx paths, the receiver preamp, and other blocks.
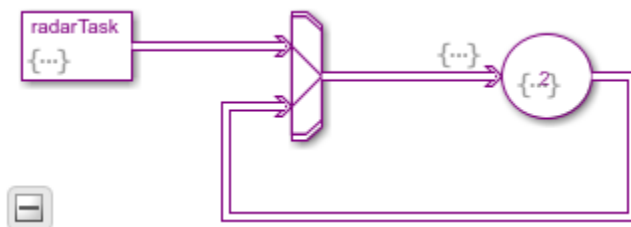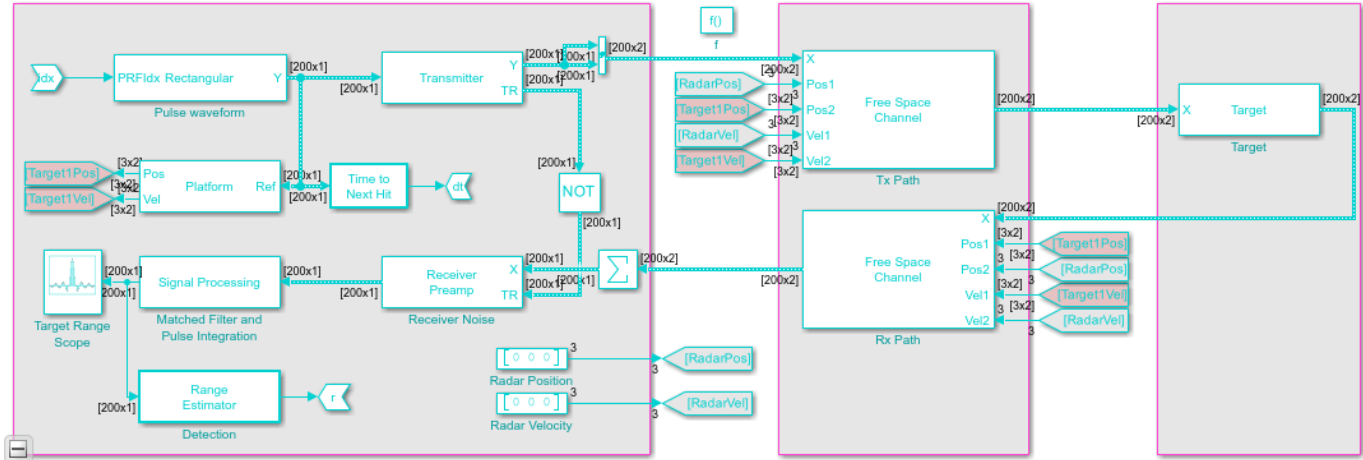
## Exploring the Example

Several dialog parameters of the model are calculated by the helper function helperslexPRFSelectionParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

## Results and Displays

The figure below shows the detected ranges of the targets. Target ranges are computed from the round-trip time delay of the reflected signals from the targets. At the simulation start, the radar detects two targets, one is slightly over 2 km away and the other one is at approximately 3.5 km away.
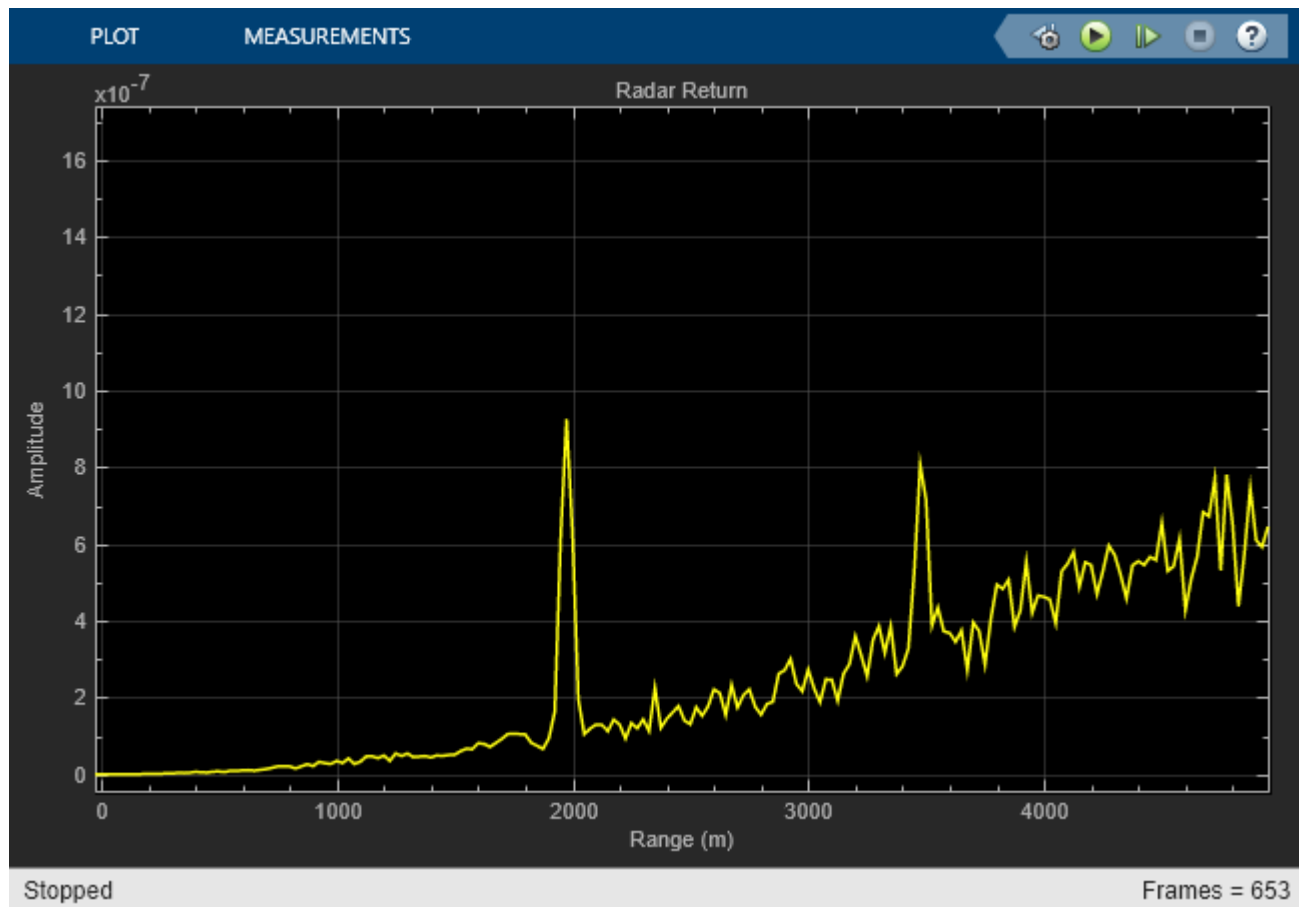
After some time, the first target moves into the 2 km zone and triggers a change of PRF. Then the received signal only covers the range up to 2 km. The display is zero padded to ensure that the plot limits do not change. Notice that the target at 3.5 km gets folded to the 1.5 km range due to range ambiguity.
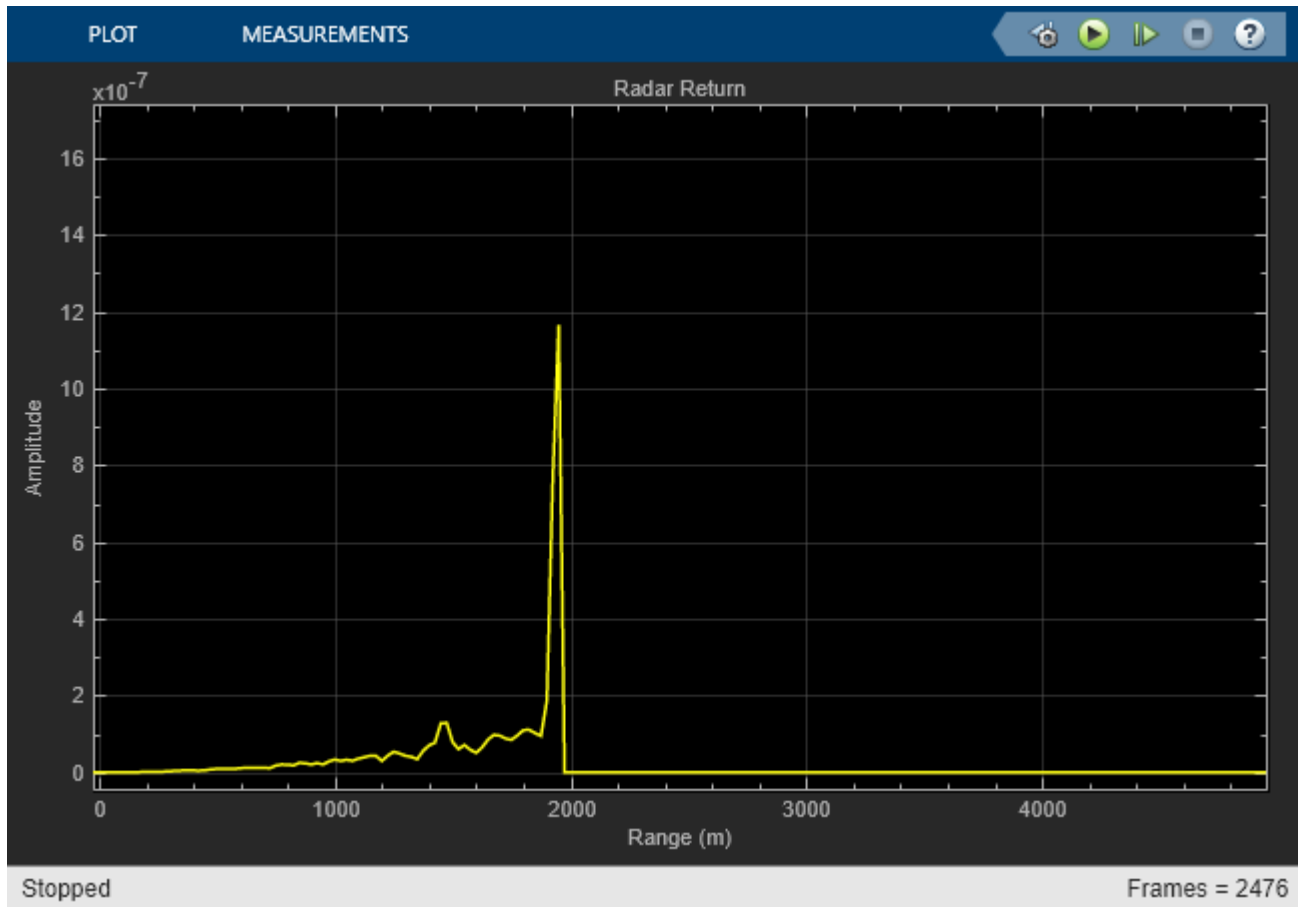
**Summary**

This example shows how to build a radar system in Simulink® that dynamically changes its PRF based on the target detection range. A staggered PRF system can be modeled similarly.

# Underwater Target Detection with an Active Sonar System

This example shows how to simulate an active monostatic sonar scenario with two targets. The sonar system consists of an isotropic projector array and a single hydrophone element. The projector array is spherical in shape. The backscattered signals are received by the hydrophone. The received signals include both direct and multipath contributions.

**Underwater Environment**

Multiple propagation paths are present between the sound source and target in a shallow water environment. In this example, five paths are assumed in a channel with a depth of 100 meters and a constant sound speed of 1520 m/s. Use a bottom loss of 0.5 dB in order to highlight the effects of the multiple paths.

Define the properties of the underwater environment, including the channel depth, the number of propagation paths, the propagation speed, and the bottom loss.

```
numPaths = 5;
propSpeed = 1520;
channelDepth = 100;

isopath{1} = phased.IsoSpeedUnderwaterPaths(...
        'ChannelDepth',channelDepth,...
        'NumPathsSource','Property',...
        'NumPaths',numPaths,...
        'PropagationSpeed',propSpeed,...
        'BottomLoss',0.5,...
        'TwoWayPropagation',true);

isopath{2} = phased.IsoSpeedUnderwaterPaths(...
        'ChannelDepth',channelDepth,...
        'NumPathsSource','Property',...
        'NumPaths',numPaths,...
        'PropagationSpeed',propSpeed,...
        'BottomLoss',0.5,...
        'TwoWayPropagation',true);
```

Next, create a multipath channel for each target. The multipath channel propagates the waveform along the multiple paths. This two-step process is analogous to designing a filter and using the resulting coefficients to filter a signal.

```
fc = 20e3;    % Operating frequency (Hz)

channel{1} = phased.MultipathChannel(...
        'OperatingFrequency',fc);

channel{2} = phased.MultipathChannel(...
        'OperatingFrequency',fc);
```

**Sonar Targets**

The scenario has two targets. The first target is more distant but has a larger target strength, and the second is closer but has a smaller target strength. Both targets are isotropic and stationary with respect to the sonar system.

```matlab
tgt{1} = phased.BackscatterSonarTarget(...
    'TSPattern',-5*ones(181,361));

tgt{2} = phased.BackscatterSonarTarget(...
    'TSPattern',-15*ones(181,361));

tgtplat{1} = phased.Platform(...
    'InitialPosition',[500; 1000; -70],'Velocity',[0; 0; 0]);

tgtplat{2} = phased.Platform(...
    'InitialPosition',[500; 0; -40],'Velocity',[0; 0; 0]);
```

The target positions, along with the channel properties, determine the underwater paths along which the signals propagate. Plot the paths between the sonar system and each target. Note that the z-coordinate determines depth, with zero corresponding to the top surface of the channel, and the distance in the x-y plane is plotted as the range between the source and target.

```matlab
helperPlotPaths([0;0;-60],[500 500; 1000 0; -70 -40], ...
    channelDepth,numPaths)
```
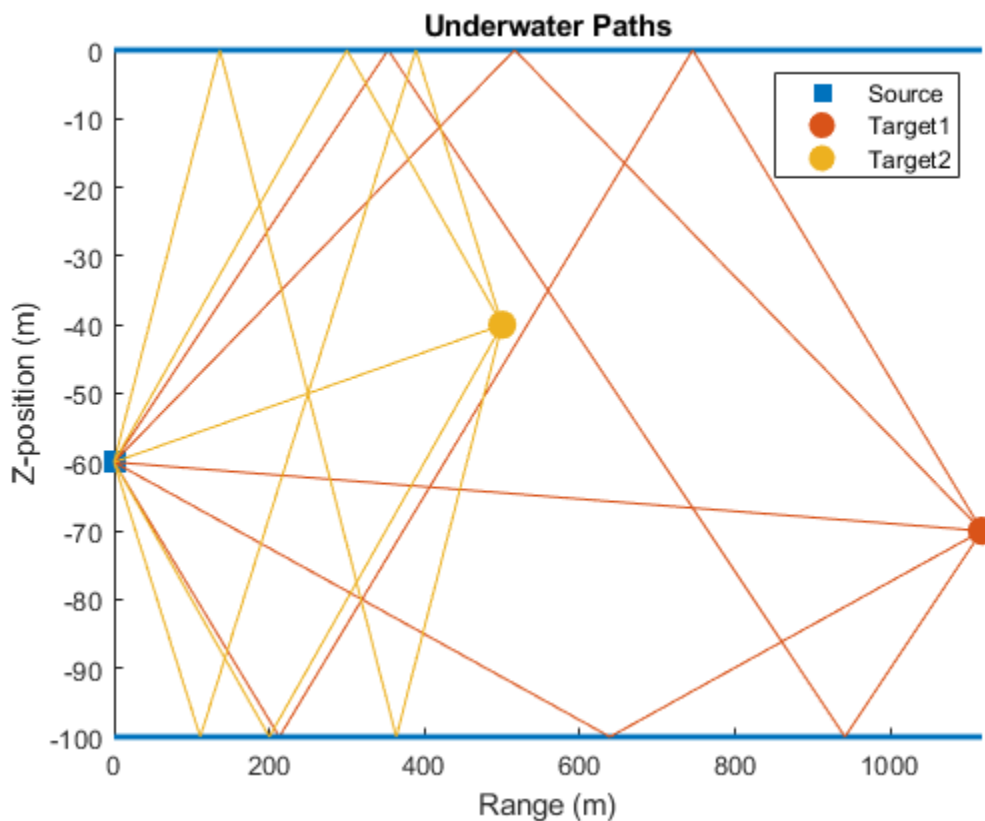


**Transmitter and Receiver**

**Transmitted Waveform**

Next, specify a rectangular waveform to transmit to the targets. The maximum target range and desired range resolution define the properties of the waveform.

```
maxRange = 5000;                          % Maximum unambiguous range
rangeRes = 10;                            % Required range resolution
prf = propSpeed/(2*maxRange);            % Pulse repetition frequency
pulse_width = 2*rangeRes/propSpeed;      % Pulse width
pulse_bw = 1/pulse_width;                % Pulse bandwidth
fs = 2*pulse_bw;                         % Sampling rate
wav = phased.RectangularWaveform(...
    'PulseWidth',pulse_width,...
    'PRF',prf,...
    'SampleRate',fs);
```

Update the sample rate of the multipath channel with the transmitted waveform sample rate.

```
channel{1}.SampleRate = fs;
channel{2}.SampleRate = fs;
```

**Transmitter**

The transmitter consists of a hemispherical array of back-baffled isotropic projector elements. The transmitter is located 60 meters below the surface. Create the array and view the array geometry.

```
plat = phased.Platform(...
    'InitialPosition',[0; 0; -60],...
    'Velocity',[0; 0; 0]);

proj = phased.IsotropicProjector(...
    'FrequencyRange',[0 30e3],'VoltageResponse',80,'BackBaffled',true);

[ElementPosition,ElementNormal] = helperSphericalProjector(8,fc,propSpeed);

projArray = phased.ConformalArray(...
    'ElementPosition',ElementPosition,...
    'ElementNormal',ElementNormal,'Element',proj);

viewArray(projArray,'ShowNormals',true);
```

Array Geometry

Array Span:
X axis = 25.656 m
Y axis = 25.656 m
Z axis = 26.316 m
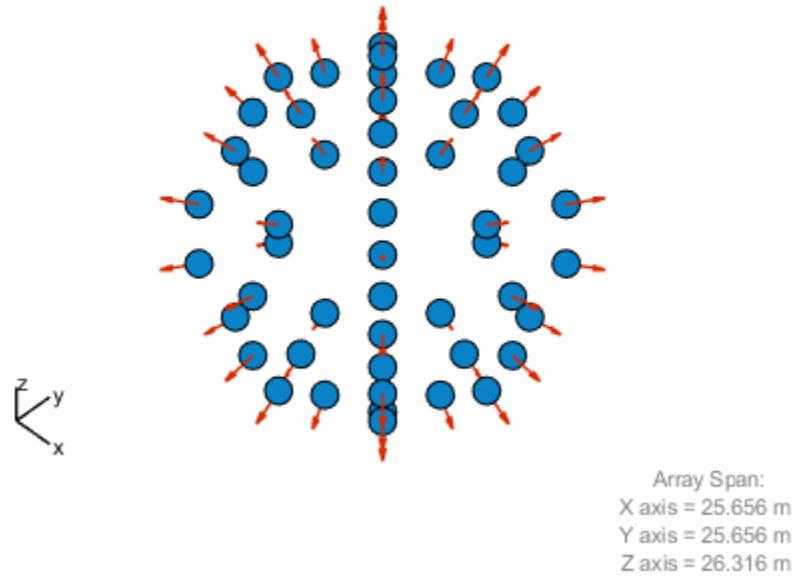
View the pattern of the array at zero degrees in elevation. The directivity shows peaks in azimuth corresponding to the azimuth position of the array elements.

```
pattern(projArray,fc,-180:180,0,'CoordinateSystem','polar',...
    'PropagationSpeed',propSpeed);
```

Directivity (dBi), Broadside at 0.00 °
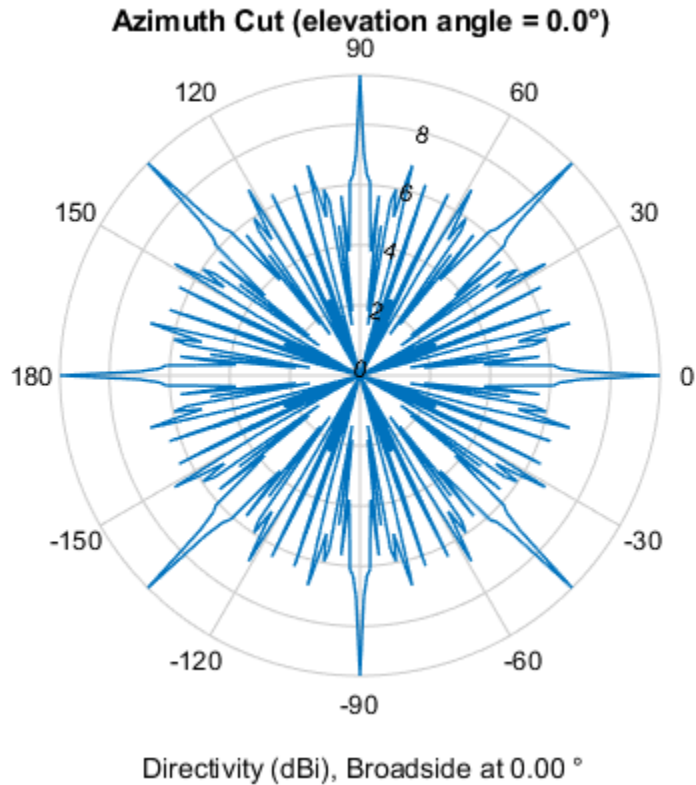
### Receiver

The receiver consists of a hydrophone and an amplifier. The hydrophone is a single isotropic element and has a frequency range from 0 to 30 kHz, which contains the operating frequency of the multipath channel. Specify the hydrophone voltage sensitivity as -140 dB.

```
hydro = phased.IsotropicHydrophone(...
    'FrequencyRange',[0 30e3],'VoltageSensitivity',-140);
```

Thermal noise is present in the received signal. Assume that the receiver has 20 dB of gain and a noise figure of 10 dB.

```
rx = phased.ReceiverPreamp(...
    'Gain',20,...
    'NoiseFigure',10,...
    'SampleRate',fs,...
    'SeedSource','Property',...
    'Seed',2007);
```

### Radiator and Collector

In an active sonar system, an acoustic wave is propagated to the target, scattered by the target, and received by a hydrophone. The radiator generates the spatial dependence of the propagated wave due to the array geometry. Likewise, the collector combines the backscattered signals received by the hydrophone element from the far-field target.

```
radiator = phased.Radiator('Sensor',projArray,'OperatingFrequency',...
 fc,'PropagationSpeed',propSpeed);
```

```
collector = phased.Collector('Sensor',hydro,'OperatingFrequency',fc,...
    'PropagationSpeed',propSpeed);
```

**Sonar System Simulation**

Next, transmit the rectangular waveform over ten repetition intervals and simulate the signal received at the hydrophone for each transmission.

```
x = wav();      % Generate pulse
xmits = 10;
rx_pulses = zeros(size(x,1),xmits);
t = (0:size(x,1)-1)/fs;

for j = 1:xmits

    % Update target and sonar position
    [sonar_pos,sonar_vel] = plat(1/prf);

    for i = 1:2 %Loop over targets
      [tgt_pos,tgt_vel] = tgtplat{i}(1/prf);

      % Compute transmission paths using the method of images. Paths are
      % updated according to the CoherenceTime property.
      [paths,dop,aloss,tgtAng,srcAng] = isopath{i}(...
            sonar_pos,tgt_pos,...
            sonar_vel,tgt_vel,1/prf);

      % Compute the radiated signals. Steer the array towards the target.
      tsig = radiator(x,srcAng);

      % Propagate radiated signals through the channel.
      tsig = channel{i}(tsig,paths,dop,aloss);

      % Target
      tsig = tgt{i}(tsig,tgtAng);

      % Collector
      rsig = collector(tsig,srcAng);
      rx_pulses(:,j) = rx_pulses(:,j) + ...
            rx(rsig);
    end
end
```

Plot the magnitude of non-coherent integration of the received signals to locate the returns of the two targets.

```
figure
rx_pulses = pulsint(rx_pulses,'noncoherent');
plot(t,abs(rx_pulses))
grid on
xlabel('Time (s)')
ylabel('Amplitude (V)')
title('Integrated Received Pulses')
```

**18-461**
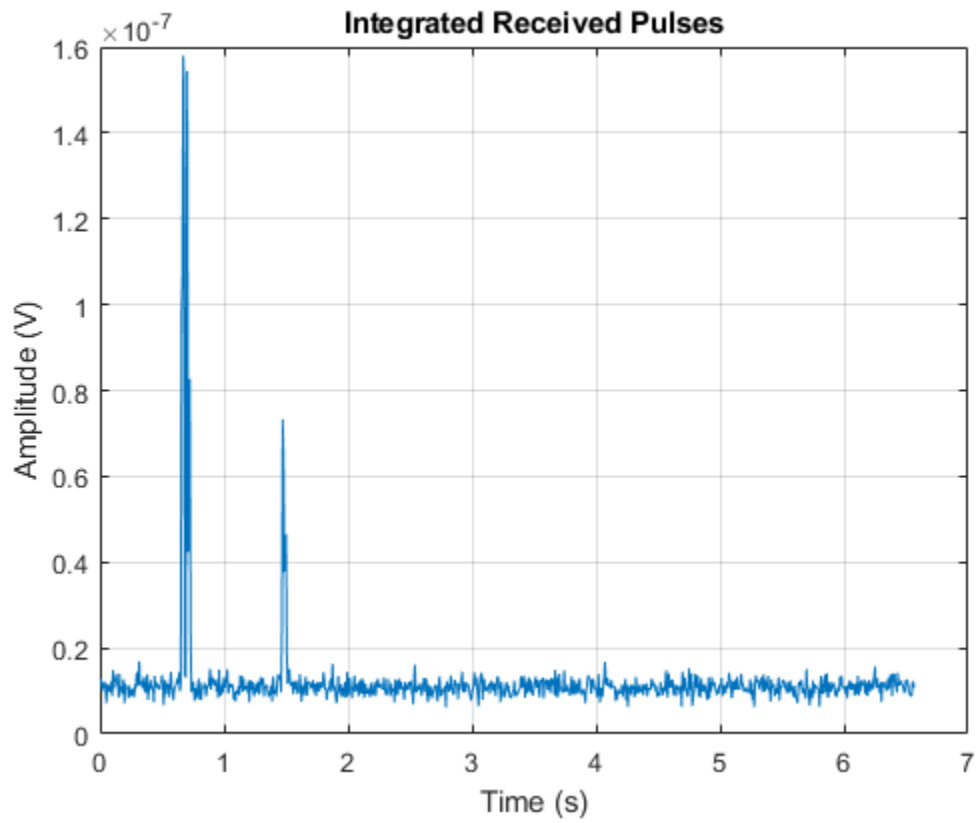
The targets, which are separated a relatively large distance, appear as distinct returns. Zoom in on the first return.

```
xlim([0.55 0.85])
```
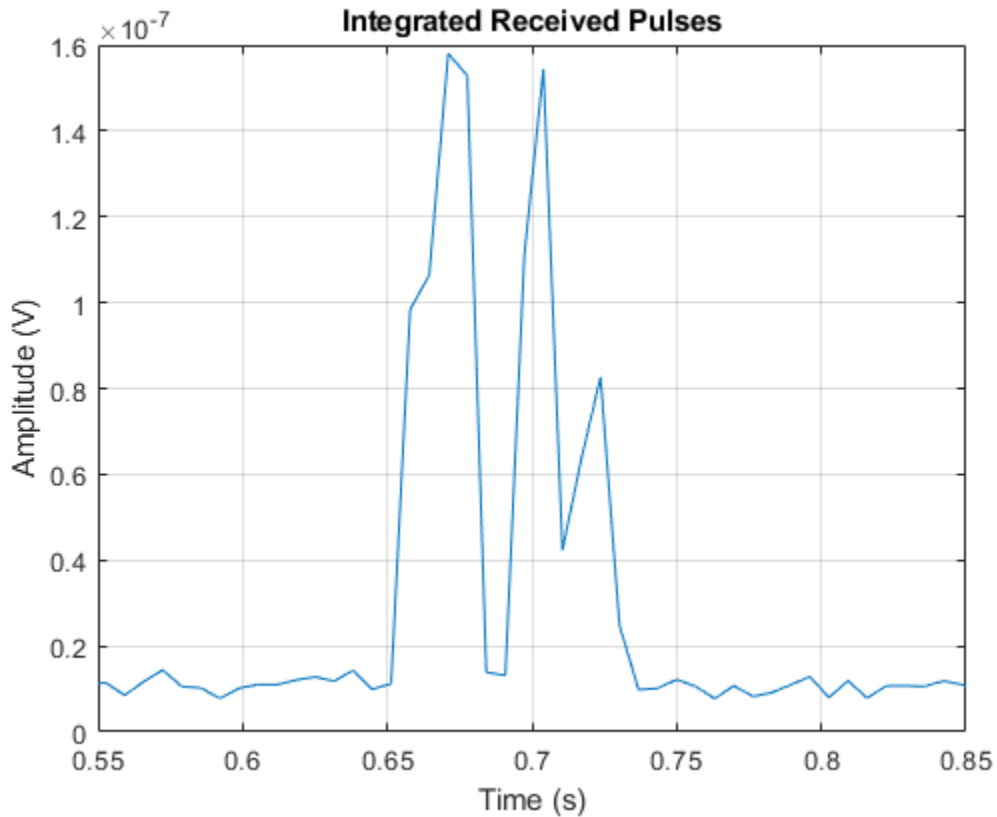
The target return is the superposition of pulses from multiple propagation paths, resulting in multiple peaks for each target. The resulting peaks could be misinterpreted as additional targets.

**Active Sonar with Bellhop**

In the previous section, the sound speed was constant as a function of channel depth. In contrast, a ray tracing program like Bellhop can generate acoustic paths for spatially-varying sound speed profiles. You can use the path information generated by Bellhop to propagate signals via the multipath channel. Simulate transmission between an isotropic projector and isotropic hydrophone in a target-free environment with the 'Munk' sound speed profile. The path information is contained in a Bellhop arrival file (MunkB_eigenray_Arr.arr).

**Bellhop Configuration**

In this example, the channel is 5000 meters in depth. The source is located at a depth of 1000 meters and the receiver is located at a depth of 800 meters. They are separated by 100 kilometers in range. Import and plot the paths computed by Bellhop.

```
[paths,dop,aloss,rcvAng,srcAng] = helperBellhopArrivals(fc,6,false);
helperPlotPaths('MunkB_eigenray')
```
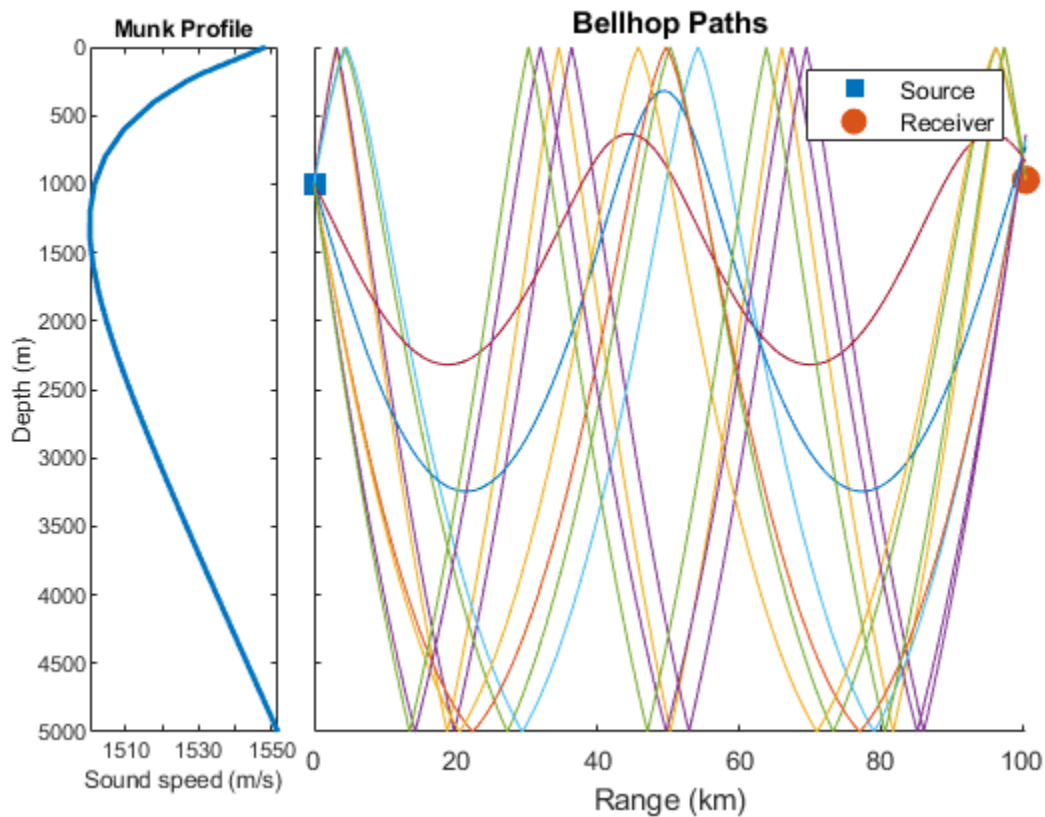
For this scenario, there are two direct paths with no interface reflections, and eight paths with reflections at both the top and bottom surfaces. The sound speed in the channel is lowest at approximately 1250 meters in depth, and increases towards the top and bottom of the channel, to a maximum of 1550 meters/second.

Create a new channel and receiver to use with data from Bellhop.

```
release(collector)
channelBellhop = phased.MultipathChannel(...
        'SampleRate',fs,...
        'OperatingFrequency',fc);

rx = phased.ReceiverPreamp(...
    'Gain',10,...
    'NoiseFigure',10,...
    'SampleRate',fs,...
    'SeedSource','Property',...
    'Seed',2007);
```

Specify a pulse for the new problem configuration.

```
maxRange = 150000;                    % Maximum unambiguous range
prf = propSpeed/(maxRange);           % Pulse repetition frequency
pulse_width = 0.02;
wav = phased.RectangularWaveform(...
    'PulseWidth',pulse_width,...
    'PRF',prf,...
    'SampleRate',fs);
```

**Bellhop Simulation**

Next, simulate the transmission of ten pulses from transmitter to receiver.

```
x = repmat(wav(),1,size(paths,2));
xmits = 10;
rx_pulses = zeros(size(x,1),xmits);
t = (0:size(x,1)-1)/fs;

for j = 1:xmits

    % Projector
    tsig = x.*proj(fc,srcAng)';

    % Propagate radiated signals through the channel.
    tsig = channelBellhop(tsig,paths,dop,aloss);

    % Collector
    rsig = collector(tsig,rcvAng);
    rx_pulses(:,j) = rx_pulses(:,j) + ...
            rx(rsig);
end
```

Plot the non-coherent integration of the transmitted pulses.

```
figure
rx_pulses = pulsint(rx_pulses,'noncoherent');
plot(t,abs(rx_pulses))
grid on
xlim([66 70])
xlabel('Time (s)')
ylabel('Amplitude (V)')
title('Integrated Received Pulses')
```
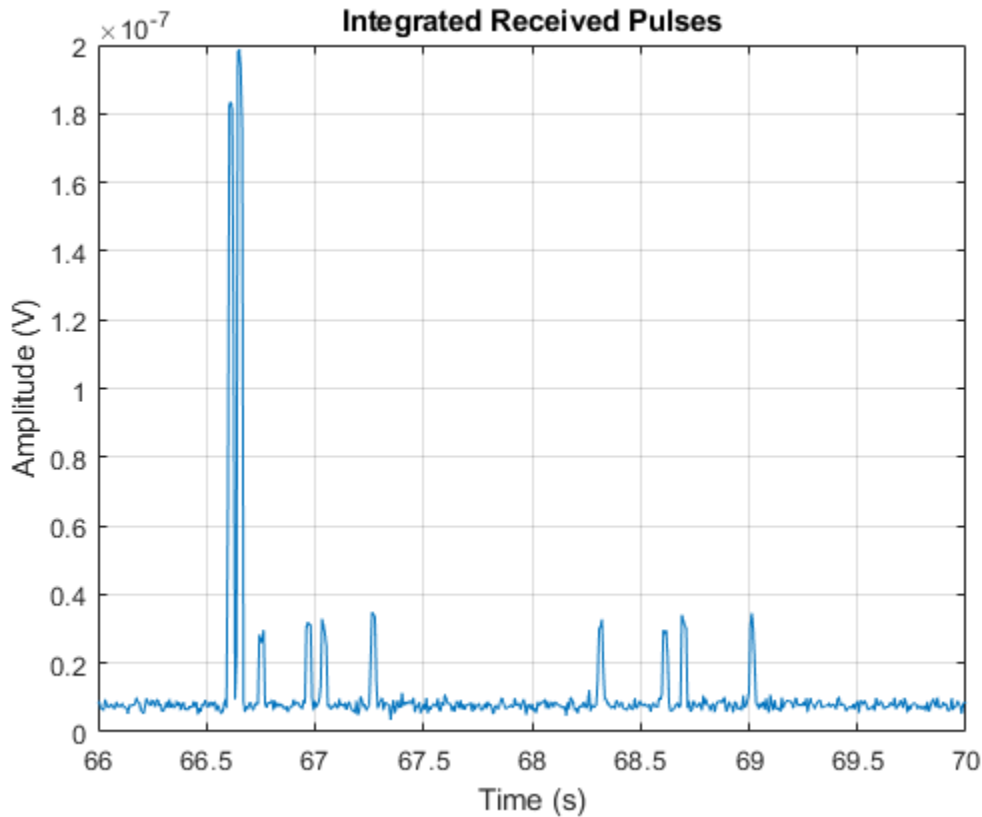
The transmitted pulses appear as peaks in the response. Note that the two direct paths, which have no interface reflections, arrive first and have the highest amplitude. In comparing the direct path received pulses, the second pulse to arrive has the higher amplitude of the two, indicating a shorter propagation distance. The longer delay time for the shorter path can be explained by the fact that it propagates through the slowest part of the channel. The remaining pulses have reduced amplitude compared to the direct paths due to multiple reflections at the channel bottom, each contributing to the loss.

**Summary**

In this example, acoustic pulses were transmitted and received in shallow-water and deep-water environments. Using a rectangular waveform, an active sonar system detected two well-separated targets in shallow water. The presence of multiple paths was apparent in the received signal. Next, pulses were transmitted between a projector and hydrophone in deep water with the 'Munk' sound speed profile using paths generated by Bellhop. The impact of spatially-varying sound speed was noted.

**Reference**

Urick, Robert. *Principles of Underwater Sound*. Los Altos, California: Peninsula Publishing, 1983.

# Locating an Acoustic Beacon with a Passive Sonar System

This example shows how to simulate a passive sonar system. A stationary underwater acoustic beacon is detected and localized by a towed passive array in a shallow-water channel. The acoustic beacon transmits a 10 millisecond pulse at 37.5 kilohertz every second, and is modeled as an isotropic projector. The locator system tows a passive array beneath the surface, which is modeled as a uniform linear array. Once the acoustic beacon signal is detected, a direction of arrival estimator is used to locate the beacon.

**Define the Underwater Channel**

In this example, the acoustic beacon is located at the bottom of a shallow water channel, which is 200 meters deep. A passive array is towed beneath the surface to locate the beacon.
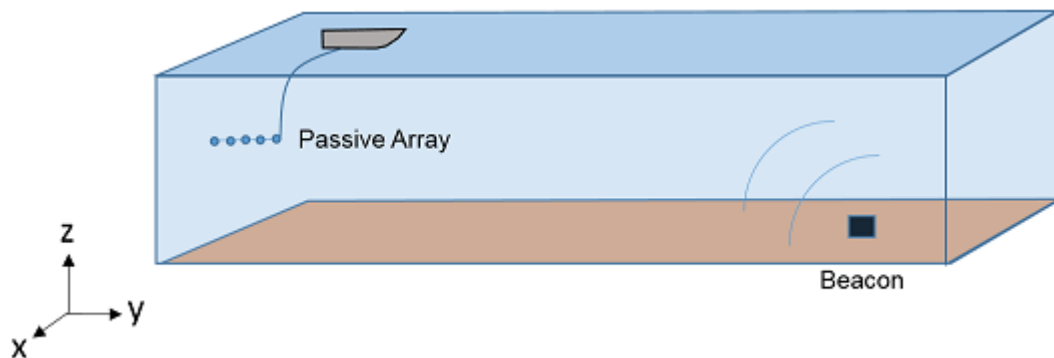


First, create a multipath channel to transmit the signal between the beacon and passive array. Consider ten propagation paths including the direct path and reflections from the top and bottom surfaces. The paths generated by `isopaths` will be used by the multipath channel, `channel`, to simulate the signal propagation.

```
propSpeed = 1520;
channelDepth = 200;
OperatingFrequency = 37.5e3;

isopaths = phased.IsoSpeedUnderwaterPaths('ChannelDepth',channelDepth,...
    'NumPathsSource','Property','NumPaths',10,'PropagationSpeed',propSpeed);

channel = phased.MultipathChannel('OperatingFrequency',OperatingFrequency);
```

**Define the Acoustic Beacon and Passive Array**

**Acoustic Beacon Waveform**

Define the waveform emitted by the acoustic beacon. The waveform is a rectangular pulse having a 1 second repetition interval and 10 millisecond width.

```
prf = 1;
pulseWidth = 10e-3;
pulseBandwidth = 1/pulseWidth;
fs = 2*pulseBandwidth;
wav = phased.RectangularWaveform('PRF',prf,'PulseWidth',pulseWidth,...
```

```
                     'SampleRate',fs);
channel.SampleRate = fs;
```

**Acoustic Beacon**

Next, define the acoustic beacon, which is located 1 meter above the bottom of the channel. The acoustic beacon is modeled as an isotropic projector. The acoustic beacon waveform will be radiated to the far field.

```
projector = phased.IsotropicProjector('VoltageResponse',120);

projRadiator = phased.Radiator('Sensor',projector,...
  'PropagationSpeed',propSpeed,'OperatingFrequency',OperatingFrequency);

beaconPlat = phased.Platform('InitialPosition',[5000; 2000; -199],...
  'Velocity',[0; 0; 0]);
```

**Passive Towed Array**

A passive towed array will detect and localize the source of the pings, and is modeled as a five-element linear array with half-wavelength spacing. The passive array has velocity of 1 m/s in the y-direction. The array axis is oriented parallel to the direction of travel.

```
hydrophone = phased.IsotropicHydrophone('VoltageSensitivity',-150);
array = phased.ULA('Element',hydrophone,...
  'NumElements',5,'ElementSpacing',propSpeed/OperatingFrequency/2,...
  'ArrayAxis','y');

arrayCollector = phased.Collector('Sensor',array,...
  'PropagationSpeed',propSpeed,'OperatingFrequency',OperatingFrequency);

arrayPlat = phased.Platform('InitialPosition',[0; 0; -10],...
  'Velocity',[0; 1; 0]);
```

Define the receiver amplifier for each hydrophone element. Choose a gain of 20 dB and noise figure of 10 dB.

```
rx = phased.ReceiverPreamp(...
    'Gain',20,...
    'NoiseFigure',10,...
    'SampleRate',fs,...
    'SeedSource','Property',...
    'Seed',2007);
```

**Simulate the Passive Sonar System**

Activate the acoustic beacon and transmit ten pings. After the propagation delay, the pings appear as peaks in the received signals of the array.

```
x = wav();
numTransmits = 10;
rxsig = zeros(size(x,1),5,numTransmits);
for i = 1:numTransmits

  % Update array and acoustic beacon positions
  [pos_tx,vel_tx] = beaconPlat(1/prf);
  [pos_rx,vel_rx] = arrayPlat(1/prf);
```

```
% Compute paths between the acoustic beacon and array
[paths,dop,aloss,rcvang,srcang] = ...
    isopaths(pos_tx,pos_rx,vel_tx,vel_rx,1/prf);

% Propagate the acoustic beacon waveform
tsig = projRadiator(x,srcang);
rsig = channel(tsig,paths,dop,aloss);

% Collect the propagated signal
rsig = arrayCollector(rsig,rcvang);

% Store the received pulses
rxsig(:,:,i) = abs(rx(rsig));

end
```

Plot the last received pulse. Because of the multiple propagation paths, each ping is a superposition of multiple pulses.

```
t = (0:length(x)-1)'/fs;
plot(t,rxsig(:,end))
xlabel('Time (s)');
ylabel('Signal Amplitude (V)')
```
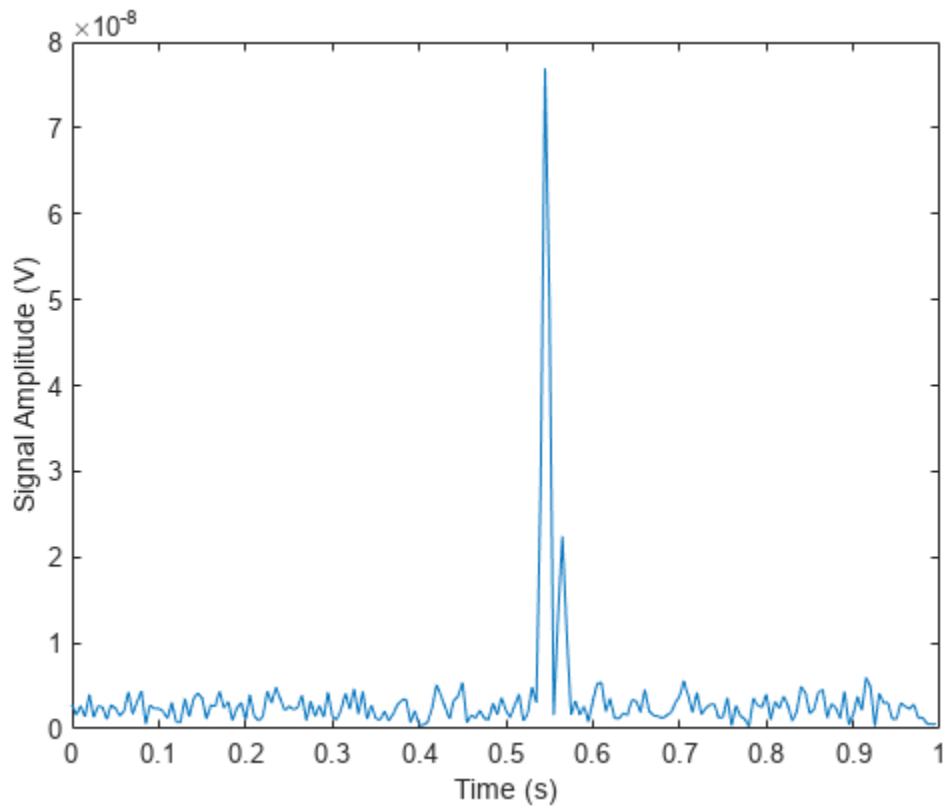
**Estimate the Direction of Arrival**

Estimate the direction of arrival of the acoustic beacon with respect to the array. Create a MUSIC estimator object, specifying a single source signal and the direction of arrival as an output. Use a scan angle grid with 0.1 degree spacing.

```
musicspatialspect = phased.MUSICEstimator('SensorArray',array,...
        'PropagationSpeed',propSpeed,'OperatingFrequency',...
        OperatingFrequency,'ScanAngles',-90:0.1:90,'DOAOutputPort',true,...
        'NumSignalsSource','Property','NumSignals',1);
```

Next, collect pings for 500 more repetition intervals. Estimate the direction of arrival for each repetition interval, and compare the estimates to the true direction of arrival.

```
numTransmits = 500;
angPassive = zeros(numTransmits,1);
angAct = zeros(numTransmits,1);

for i = 1:numTransmits

  % Update array and acoustic beacon positions
  [pos_tx,vel_tx] = beaconPlat(1/prf);
  [pos_rx,vel_rx] = arrayPlat(1/prf);

  % Compute paths between acoustic beacon and the array
  [paths,dop,aloss,rcvang,srcang] = ...
      isopaths(pos_tx,pos_rx,vel_tx,vel_rx,1/prf);
  angAct(i) = rcvang(1,1);

  % Propagate the acoustic beacon waveform
  tsig = projRadiator(x,srcang);
  rsig = channel(tsig,paths,dop,aloss);

  % Collect the propagated signal
  rsig = arrayCollector(rsig,rcvang);

  rxsig = rx(rsig);

  % Estimate the direction of arrival
  [~,angPassive(i)] = musicspatialspect(rxsig);

end
```

Plot the estimated arrival angles and the true directions of arrival for each pulse repetition interval.

```
plot([angPassive angAct])
xlabel('Pulse Number')
ylabel('Arrival angle (degrees)')
legend('Estimated DOA','Actual DOA')
```
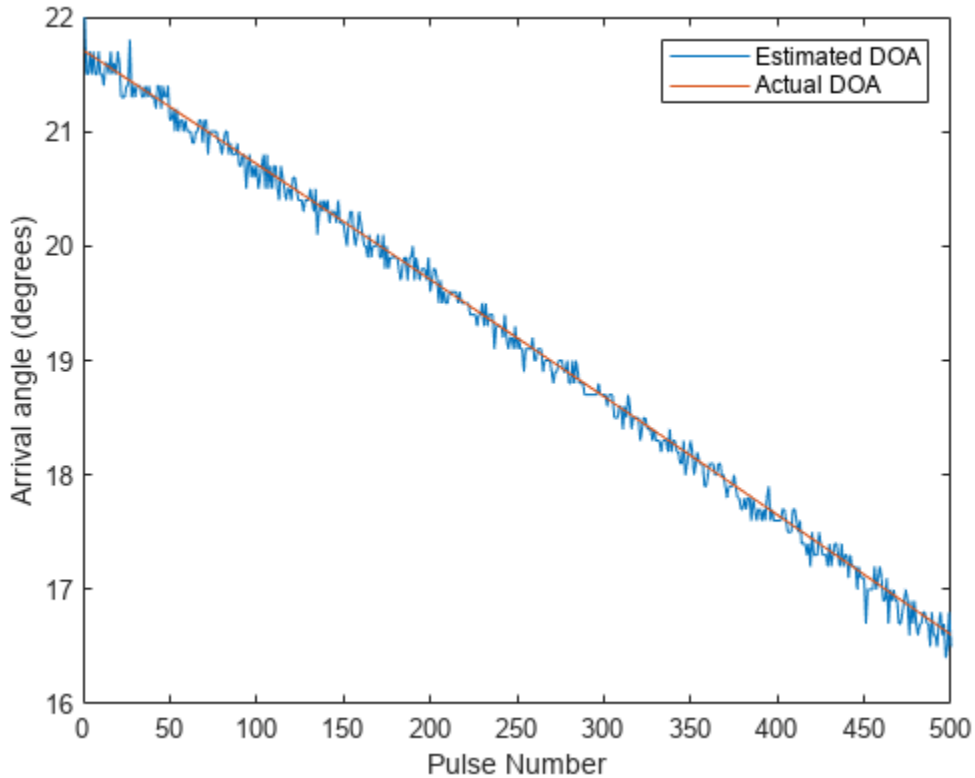
The estimated and actual directions of arrival agree to within less than one degree.

**Summary**

In this example, the transmission of acoustic pings between a beacon and passive array was simulated in a shallow-water channel. Each ping was received along ten acoustic paths. The direction of arrival of the beacon was estimated with respect to the passive array for each received ping and compared to the true direction of arrival. The direction of arrival could be used to locate and recover the beacon.

**Reference**

Urick, Robert. *Principles of Underwater Sound*. Los Altos, California: Peninsula Publishing, 1983.

# Waveform Parameter Extraction from Received Pulse

Modern aircraft often carry a radar warning receiver (RWR) with them. The RWR detects the radar emission and warns the pilot when the radar signal shines on the aircraft. An RWR can not only detect the radar emission, but also analyze the intercepted signal and catalog what kind of radar is the signal coming from. This example shows how an RWR can estimate the parameters of the intercepted pulse. The example simulates a scenario with a ground surveillance radar (emitter) and a flying aircraft (target) equipped with an RWR. The RWR intercepts radar signals, extracts the waveform parameters from the intercepted pulse, and estimates the location of the emitter. The extracted parameters can be utilized by the aircraft to take counter-measures.

This example requires Image Processing Toolbox™.

**Introduction**

An RWR is a passive electronic warfare support system [1] that provides timely information to the pilot about its RF signal environment. The RWR intercepts an impinging signal, and uses signal processing techniques to extract information about the intercepted waveform characteristics, as well as the location of the emitter. This information can be used to invoke counter-measures, such as jamming to avoid being detected by the radar. The interaction between the radar and the aircraft is depicted in the following diagram.
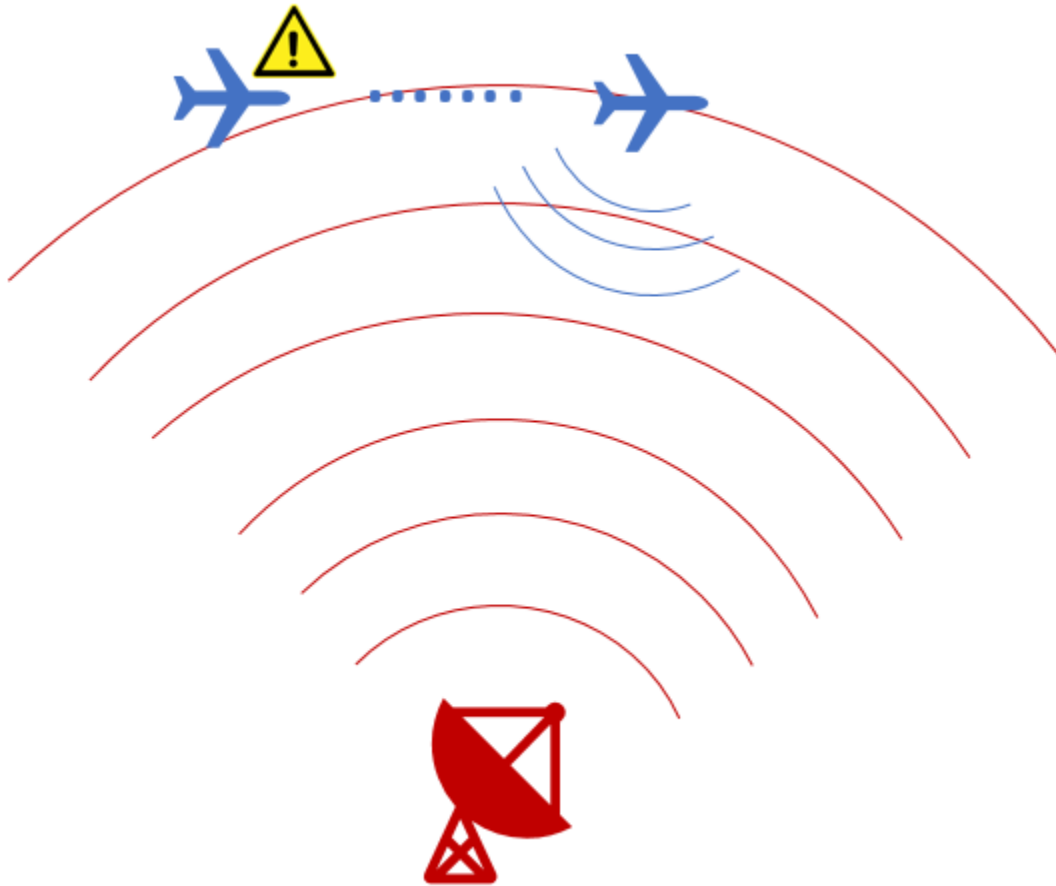
This example simulates a scenario with a ground surveillance radar and an airplane with an RWR. The RWR detects the radar signal and extracts the following waveform parameters from the intercepted signal:

1. Pulse repetition interval
2. Center frequency
3. Bandwidth
4. Pulse duration
5. Direction of arrival
6. Position of the emitter

The RWR chain consists of a phased array antenna, a channelized receiver, an envelope detector, and a signal processor. The frequency band of the intercepted signal is estimated by the channelized receiver and the envelope detector, following which the detected subbanded signal is fed to the signal processor. Beam steering is applied towards the direction of arrival of this subbanded signal, and the waveform parameters are estimated using pseudo Wigner-Ville transform in conjunction with Hough transform. Using angle of arrival and single-baseline approach, the location of the emitter is also estimated.

**Scenario Setup**

Assume the ground based surveillance radar operates in the L band, and transmits chirp signals of 3 μs duration at a pulse repetition interval of 15 μs. Bandwidth of the transmitted chirp is 30 MHz, and the carrier frequency is 1.8 GHz. The surveillance radar is located at the origin and is stationary, and the aircraft is flying at a constant speed of 200 m/s (~0.6 Mach).

```
% Define the transmitted waveform parameters
fs = 4e9;                         % Sampling frequency for the systems (Hz)
fc = 1.8e9;                       % Operating frequency of the surveillance radar (Hz)
T = 3e-6;                         % Chirp duration (s)
PRF = 1/(15e-6);                  % Pulse repetition frequency (Hz)
BW = 30e6;                        % Chirp bandwidth (Hz)
c= physconst('LightSpeed');       % Speed of light in air (m/s)

% Assume the surveillance radar is at the origin and is stationary
radarPos= [0;0;0];           % Radar position (m)
radarVel= [0;0;0];           % Radar speed (m/s)

% Assume aircraft is moving with constant velocity
rwrPos= [-3000;1000;1000];   % Aircraft position (m)
rwrVel= [200; 0; 0];         % Aircraft speed (m/s)

% Configure objects to model ground radar and aircraft's relative motion
rwrPose = phased.Platform(rwrPos, rwrVel);
radarPose = phased.Platform(radarPos, radarVel);
```

The transmit antenna of the radar is a 8-by-8 uniform rectangular phased array, having a spacing of λ/2 between its elements. The signal propagates from the radar to the aircraft and is intercepted and analyzed by the RWR. For simplicity, the waveform is chosen as a linear FM waveform with a peak power of 100 W.

```
% Configure the LFM waveform using the waveform parameters defined above
wavGen= phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',T,'SweepBandwidth',BW,'PRF',PRF);

% Configure the Uniform Rectangular Array
antennaTx = phased.URA('ElementSpacing',repmat((c/fc)/2, 1, 2), 'Size', [8,8]);

% Configure objects for transmitting and propagating the radar signal
tx = phased.Transmitter('Gain', 5, 'PeakPower',100);
radiator = phased.Radiator( 'Sensor', antennaTx, 'OperatingFrequency', fc);
envIn = phased.FreeSpace('TwoWayPropagation',false,'SampleRate', fs,'OperatingFrequency',fc);
```

The ground surveillance radar is unaware of the direction of the target, therefore, it needs to scan the entire space to look for the aircraft. In general, the radar will transmit a series of pulses at each direction before moving to the next direction. Therefore, without losing generality, this example assumes that the radar is transmitting toward zero degrees azimuth and elevation. The following figure shows the time frequency representation of the arrival of a four-pulse train at the aircraft. Note that although the pulse train arrives at a specific delay, the time delay of the arrival of the first pulse is irrelevant for the RWR because it has no knowledge transmit time and has to constantly monitor its environment.

```
% Transmit a train of pulses
numPulses = 4;
txPulseTrain = helperRWR('simulateTransmission',numPulses, wavGen, rwrPos, ...
    radarPos, rwrVel, radarVel, rwrPose, radarPose, tx, radiator, envIn,fs,fc,PRF);
```

```
% Observe the signal arriving at the RWR
pspectrum(txPulseTrain,fs,'spectrogram','FrequencyLimits',[1.7e9 1.9e9], 'Leakage',0.65)
title('Transmitted Pulse Train Spectrogram')
caxis([-110 -90])
```
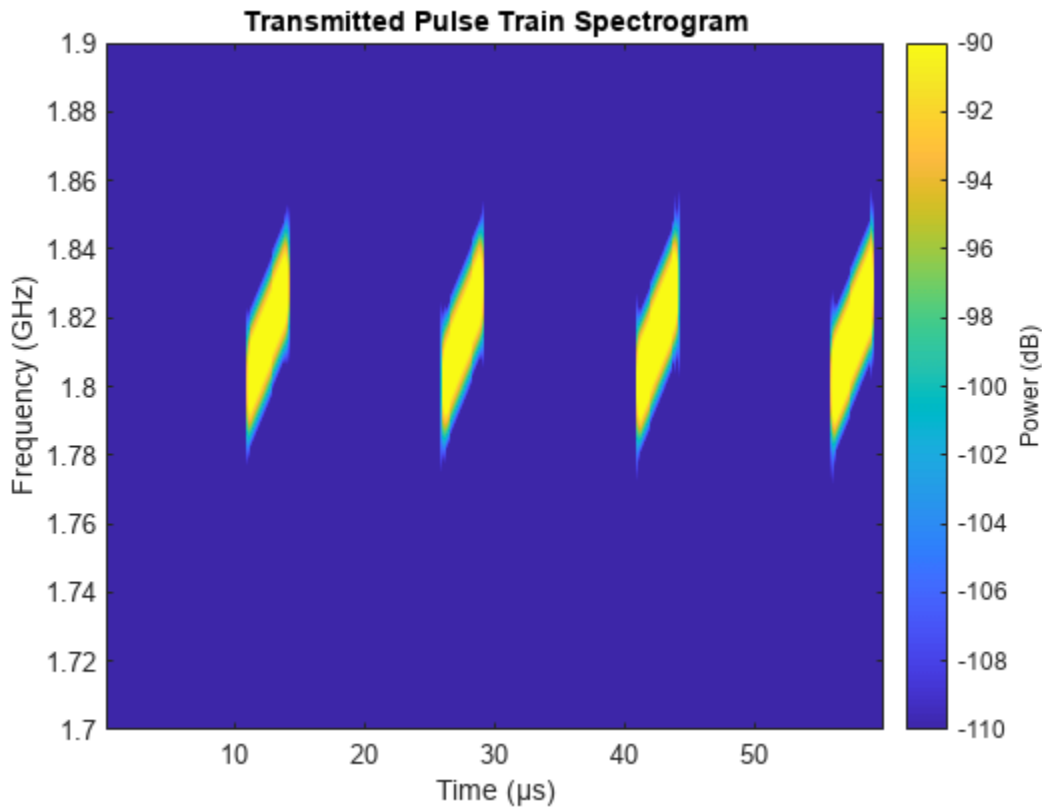


The RWR is equipped with a 10-by-10 uniform rectangular array with a spacing of $\lambda/2$ between its elements. It operates in the entire L-band, with a center frequency of 2 GHz. The RWR listens to the environment, and continuously feeds the collected data into the processing chain.

```
% Configure the receive antenna
dip = phased.IsotropicAntennaElement('BackBaffled',true);
antennaRx = phased.URA('ElementSpacing',repmat((c/2e9)/2,1,2),'Size', [10,10],'Element',dip);

% Model the radar receiver chain
collector = phased.Collector('Sensor', antennaRx,'OperatingFrequency',fc);
rx = phased.ReceiverPreamp('Gain',0,'NoiseMethod','Noise power', 'NoisePower',2.5e-6,'SeedSource

% Collect the waves at the receiver
[~, tgtAng] = rangeangle(radarPos,rwrPos);
yr = collector(txPulseTrain,tgtAng);
yr = rx(yr);
```

**RWR Envelope Detector**

The envelope detector in the RWR is responsible for detecting the presence of any signal. As the RWR is continuously receiving data, the receiver chain buffers and truncates the received data into 50 $\mu s$ segments.

```
% Truncate the received data
truncTime = 50e-6;
truncInd = round(truncTime*fs);
yr = yr(1:truncInd,:);
```

Since the RWR has no knowledge about the exact center frequency used in the transmit waveform, it first uses a bank of filters, each tuned to a slightly different RF center frequency, to divide the received data into subbands. Then the envelope detector is applied in each band to check whether a signal is present. In this example, the signal is divided into subbands of 100 MHz bandwidth. An added benefit for such an operation is that instead of sampling the entire bandwidth covered by the RWR, the signal in each subband can be downsampled to a sampling frequency of 100 MHz.

```
% Define the bandwidth of each frequency subband
stepFreq = 100e6;
```

```
% Calculate number of subbands and configure dsp.Channelizer
numChan = fs/stepFreq;
channelizer = dsp.Channelizer('NumFrequencyBands', numChan, 'StopbandAttenuation', 80);
```

This plot shows the first four band created by the filter bank.

```
% Visualize the first four filters created in the filter bank of the
% channelizer
freqz(channelizer, 1:4)
title('Zoomed Channelizer Response for First Four Filters')
xlim([0 0.2])
```
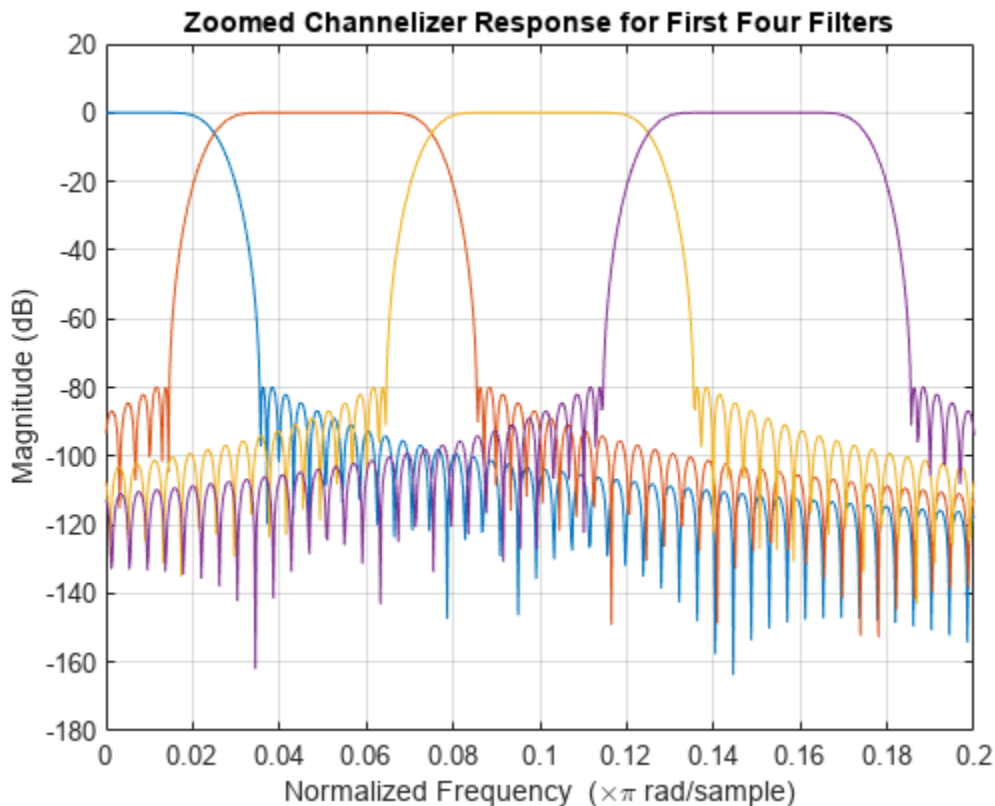
```
% Pass the received data through the channelizer
subData = channelizer(yr);
```

The received data, `subData`, has three dimensions. The first dimension represents the fast-time, the second dimension represents the subbands, and the third dimension corresponds to the receiving elements of the receiving array. For the RWR's 10-by-10 antenna configuration used in this example, you have 100 receiving elements. Because the transmit power is low and the receiver noise is high, the radar signal is indistinguishable from the noise. Therefore the received powers are summed across these elements to enhance the signal-to-nose ratio (SNR) and get a better estimate of the power in each subband. The band that has the maximum power is the one used by the radar.

```
% Rearrange the subData to combine the antenna array channels only
incohsubData = pulsint(permute(subData,[1,3,2]),'noncoherent');
incohsubData = squeeze(incohsubData);

% Plot power distribution
subbandPow = pow2db(rms(incohsubData,1).^2)+30;
plot(subbandPow);
xlabel('Band Index');
ylabel('Power (dBm)');
```
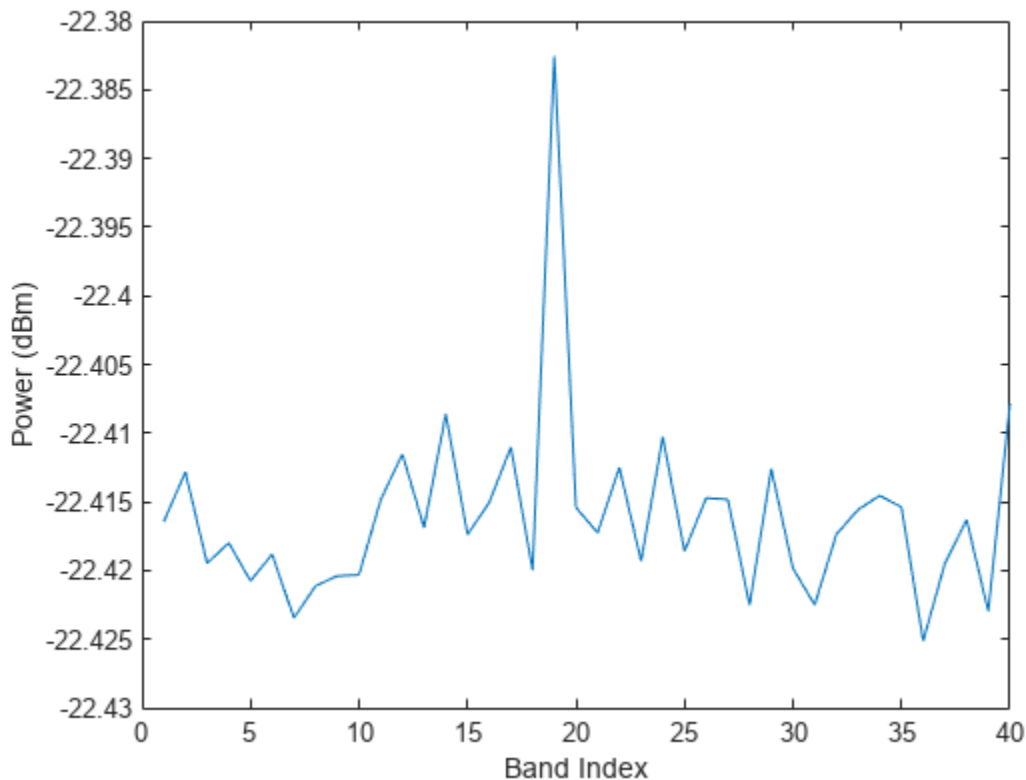


```
% Find the subband with maximum power
[~,detInd] = max(subbandPow);
```

**RWR Signal Processor**

Although the power in the selected band is higher compared to the neighboring band, the SNR within the band is still low, as shown in the following figure.

```
subData = (subData(:,detInd,:));
subData = squeeze(subData);  % adjust the data to 2-D matrix

% Visualize the detected sub-band data
plot(mag2db(abs(sum(subData,2)))+30)
ylabel('Power (dBm)')
title('Detected Subband from 100 Channels Combined Incoherently')
```
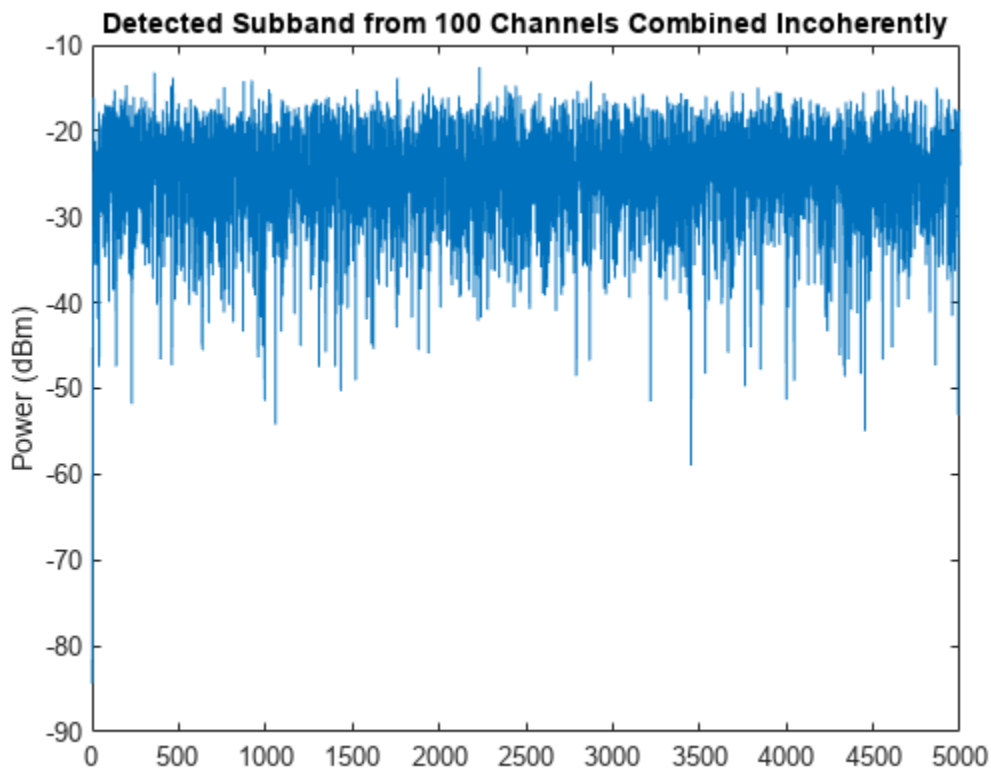


```
% Find the original starting frequency of the sub-band having the detected
% signal
detfBand = fs*(detInd-1)/(fs/stepFreq);

% Update the sampling frequency to the decimated frequency
fs = stepFreq;
```

The subData is now a two-dimensional matrix. The first dimension represents fast-time samples and the second dimension is the data across 100 receiving antenna channels. The detected subband starting frequency is calculated to find the carrier frequency of the detected signal.

The next step for the RWR is to find the direction from which the radio waves are arriving. This angle of arrival information will be used to steer the receive antenna beam in the direction of the emitter and locate the emitter on the ground using single baseline approach. The RWR estimates the

direction of arrival using a two-dimensional MUSIC estimator. Beam steering is done using phase-shift beamformer to achieve maximum SNR of the signal, and thus help in waveform parameter extraction.

Assume that the ground plane is flat and parallel to the xy-plane of the coordinate system. Then, the RWR can use the altitude information from its altimeter readings of the aircraft along with the direction of arrival to triangulate the location of the emitter.

```
% Configure the MUSIC Estimator to find the direction of arrival of the
% signal
doaEst = phased.MUSICEstimator2D('OperatingFrequency',fc,'PropagationSpeed',c,...
    'SensorArray',antennaRx,'DOAOutputPort',true,'AzimuthScanAngles',-50:.5:50,...
    'ElevationScanAngles',-50:.5:50, 'NumSignalsSource', 'Property','NumSignals', 1);

[mSpec,doa] = doaEst(subData);
plotSpectrum(doaEst,'Title','2-D MUSIC Spatial Spectrum Top View');
view(0,90); axis([-30 0 -30 0]);
```
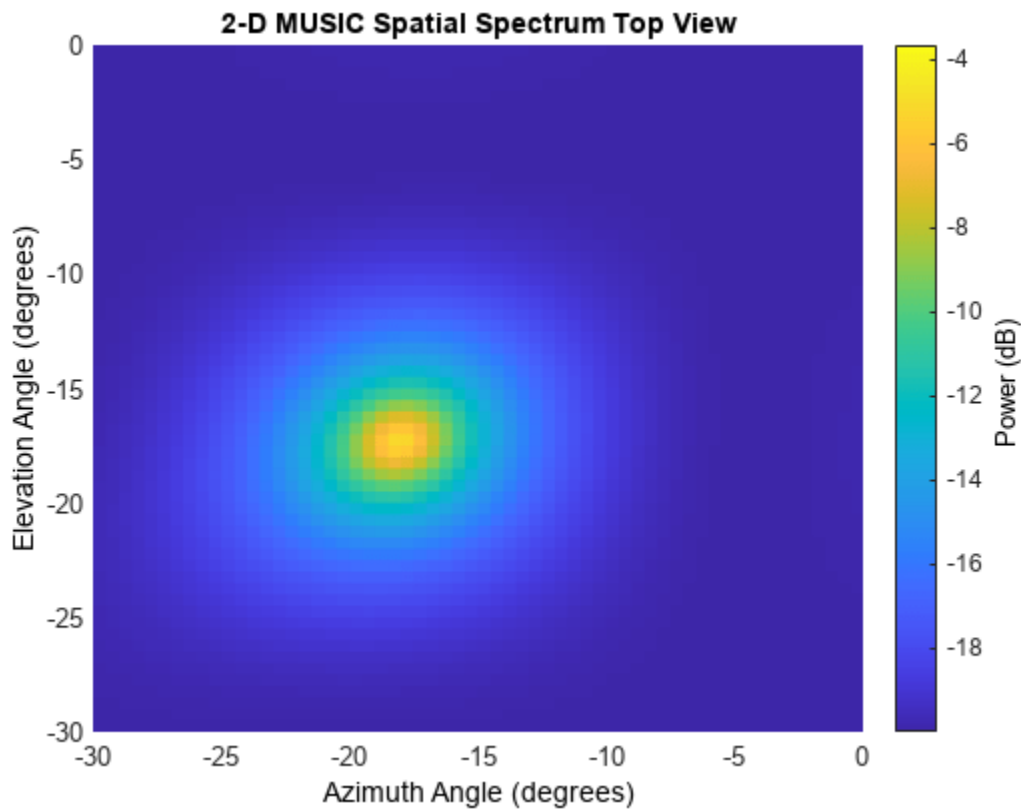


The figure clearly shows the location of the emitter.

```
% Configure the beamformer object to steer the beam before combining the
% channels
beamformer = phased.PhaseShiftBeamformer('SensorArray',antennaRx,...
    'OperatingFrequency',fc,'DirectionSource','Input port');

% Apply the beamforming, and visualize the beam steered radiation
% pattern
```

```
mBeamf = beamformer(subData, doa);

% Find the location of the emitter
altimeterElev = rwrPos(3);
d = abs(altimeterElev/sind(doa(2)));
```

After applying the beam steering, the antenna has the maximum gain in the azimuth and elevation angle of the arrival of the signal. This further improves the SNR of the intercepted signal. Next, the signal parameters are extracted in the signal processor using one of the time-frequency analysis techniques known as pseudo Wigner-Ville transform coupled with Hough transform as described in [2].

First, derive the time frequency representation of the intercepted signal using the Wigner-Ville transform.

```
% Compute the pseudo Wigner-Ville transform
[tpwv,t,f] = helperRWR('pWignerVille',mBeamf,fs);
```

```
% Plot the pseudo Wigner-Ville transform
imagesc(f*1e-6,t*1e6,pow2db(abs(tpwv./max(tpwv(:)))));
xlabel('Frequency (MHz)'); ylabel('Time(\mus)');
caxis([-50 0]); clb = colorbar; clb.Label.String = 'Normalized Power (dB)';
title ('Pseudo Wigner-Ville Transform')
```
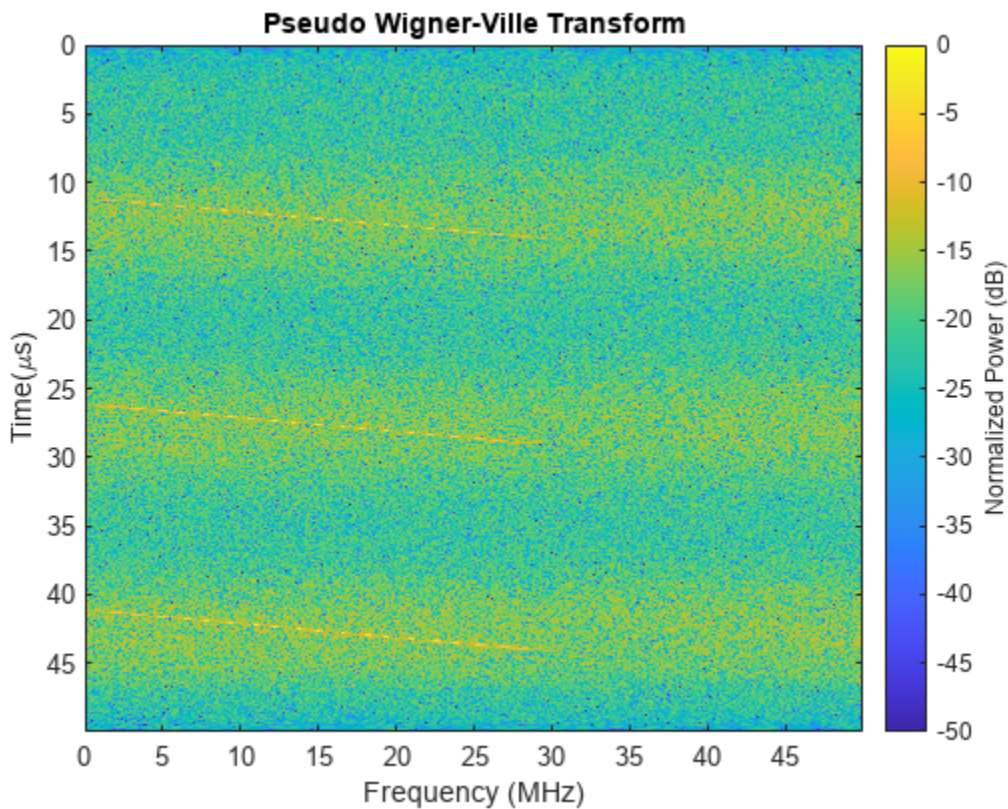


Using human eyes, even though the resulting time frequency representation is noisy, it is not too hard to separate the signal from the background. Each pulse appears as a line in the time frequency plane. Thus, using beginning and end of the time-frequency lines, you can derive the pulse width and the

bandwidth of the pulse. Similarly, the time between lines from different pulses gives you the pulse repetition interval.

To do this automatically without relying on human eyes, use Hough transform to identify those lines from the image. The Hough transform can perform well in the presence of noise, and the transform is an enhancement to the time-frequency signal analysis method.

To use the Hough transform, it is necessary to convert the time frequency image into a binary image. The following performs some data smoothing on the image and then uses the `imbinarize` function to do the conversion. The conversion threshold can be modified based on the signal-noise characteristics of the receiver and the operating environment.

```matlab
% Normalize the pseudo Wigner-Ville image
twvNorm = abs(tpwv)./max(abs(tpwv(:)));

% Implement a median filter to clear the noise
filImag = medfilt2(twvNorm,[7 7]);

% Use threshold to convert filtered image into binary image
BW = imbinarize(filImag./max(filImag(:)), 0.15);
imagesc(f*1e-6,t*1e6,BW); colormap('gray');
xlabel('Frequency (MHz)'); ylabel('Time(\mus)');
title ('Pseudo Wigner-Ville Transform - BW')
```
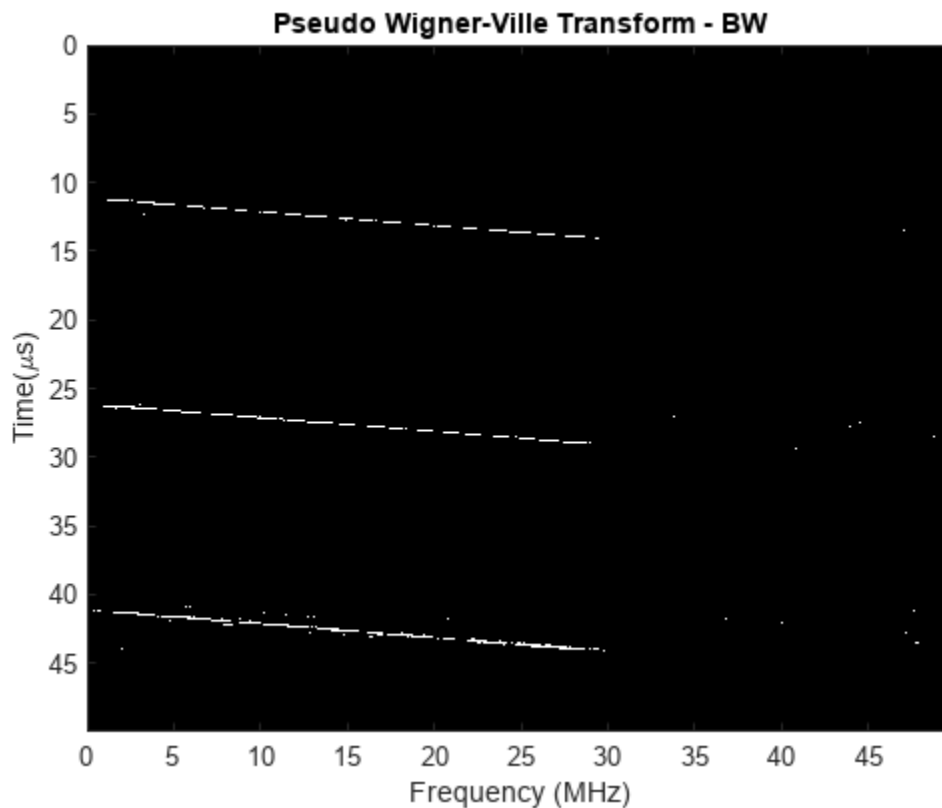


Using the Hough transform, the binary pseudo Wigner-Ville image is first transformed to peaks. This way, instead of detecting the line in an image, you just need to detect a peak in an image.

```
% Compute the Hough transform of the image and plot
[H,T,R] = hough(BW);
imshow(H,[],'XData',T,'YData',R,'InitialMagnification','fit');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
title('Hough Transform of the Image')
```

The peak positions are extracted using the `houghpeaks` function.

```
% Compute peaks in the transform, up to 5 peaks
P  = houghpeaks(H,5);
x = T(P(:,2)); y = R(P(:,1));
plot(x,y,'s','color','g'); xlim([-90 -50]); ylim([-5000 0])
```
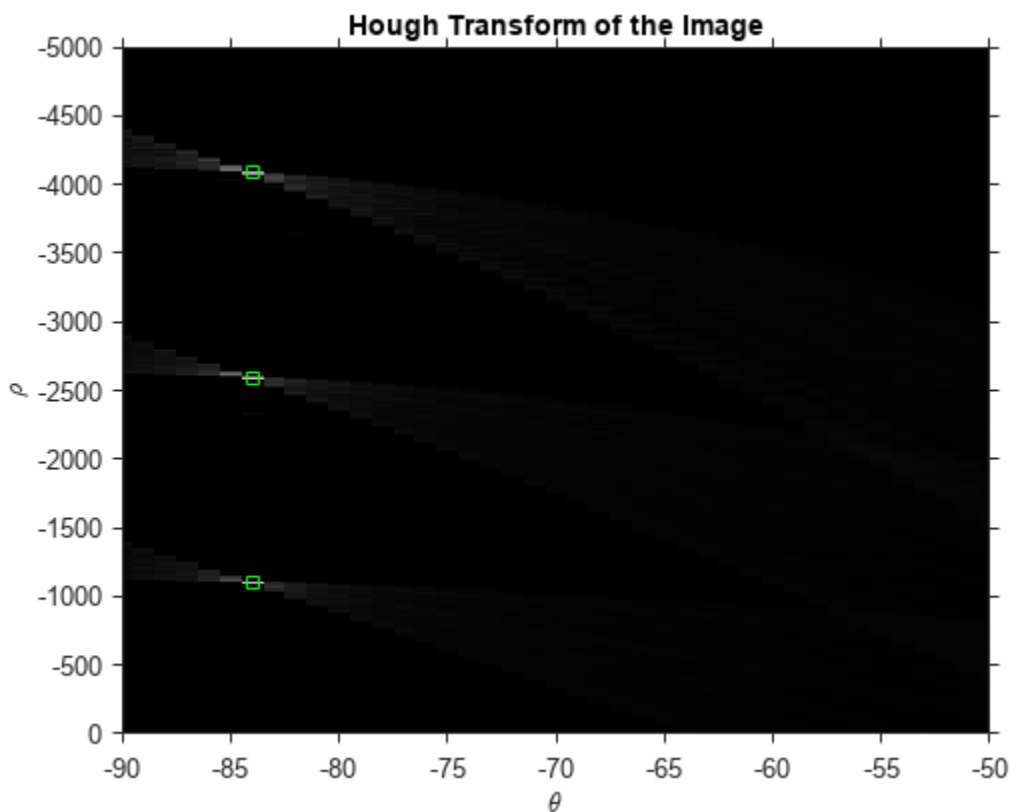


Using these positions, the `houghlines` function can reconstruct the lines in the original binary image. Then as discussed earlier, the beginning and the end of these lines help you estimate the waveform parameters.

```
lines = houghlines(BW,T,R,P,'FillGap',3e-6*fs,'MinLength',1e-6*fs);
coord = [lines(:).point1; lines(:).point2];

% Plot the detected lines superimposed on the binary image
clf;
imagesc(f*1e-6, t*1e6, BW); colormap(gray); hold on
xlabel('Frequency (MHz)')
ylabel('Time(\mus)')
title('Hough Transform – Detected Lines')
for ii = 1:2:2*size(lines,2)
```

```
    plot(f(coord(:,ii))*1e-6, t(coord(:,ii+1))*1e6,'LineWidth',2,'Color','green');
end
```
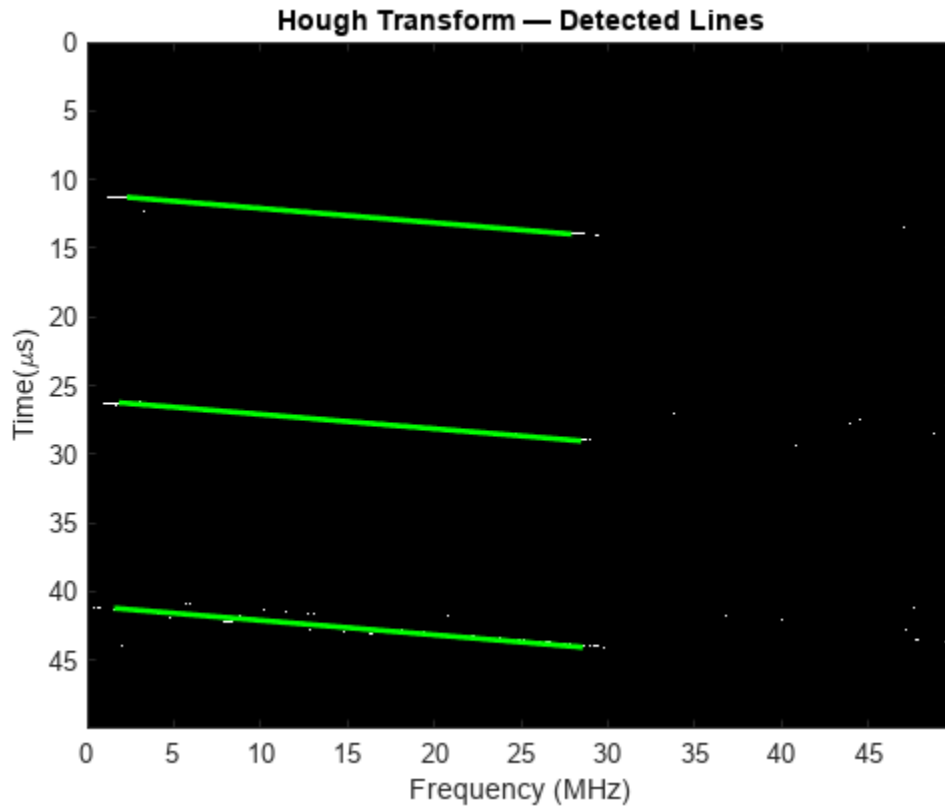


```
% Calculate the parameters using the line co-ordinates
pulDur = t(coord(2,2)) - t(coord(1,2));          % Pulse duration
bWidth = f(coord(2,1)) - f(coord(1,1));          % Pulse Bandwidth
pulRI = abs(t(coord(1,4)) - t(coord(1,2)));      % Pulse repetition interval
detFc = detfBand + f(coord(2,1));                % Center frequency
```

The extracted waveform characteristics are listed next. They match the truth very well. These estimates can then be used to catalog the radar and prepare for counter measures if necessary.

```
helperRWR('displayParameters',pulRI, pulDur, bWidth,detFc, doa,d);
```

```
Pulse repetition interval = 14.97 microseconds
Pulse duration = 2.84 microseconds
Pulse bandwidth = 27 MHz
Center frequency = 1.8286 GHz
Azimuth angle of emitter = -18.5 degrees
Elevation angle of emitter = -17.5 degrees
Distance of the emitter = 3325.5095 m
```

**Summary**

This example shows how an RWR can estimate the parameters of the intercepted radar pulse using signal processing and image processing techniques.

**References**

[1] *Electronic Warfare and Radar Systems Engineering Handbook* 2013, Naval Air Warfare Center Weapons Division, Point Mugu, California.

[2] Stevens, Daniel L., and Stephanie A. Schuckers. "Detection and Parameter Extraction of Low Probability of Intercept Radar Signals Using the Hough Transform." *Global Journal of Researches in Engineering*, Jan. 2016, pp. 9–25. *DOI.org (Crossref)*, https://doi.org/10.34257/GJREJVOL15IS6PG9.

# Massive MIMO Hybrid Beamforming

This example shows how to use hybrid beamforming at the transmit end of a massive MIMO communications system, using multi-user and single-user systems techniques. The example determines the channel state information at the transmitter by using full channel sounding. It partitions the required precoding into digital baseband and analog RF components, using different techniques for multi-user and single-user systems. The example computes EVM and BER communications system figures of merit to compare the signal recovered by simplified all-digital receivers to the multiple transmitted data streams.

In the example a scattering-based spatial channel technique models the transmit/receive spatial locations and antenna patterns. A simpler static-flat MIMO channel is also offered for link validation purposes.

**Introduction**

The ever-growing demand for high data rate and more user capacity increases the need to use the available spectrum more efficiently. Multi-user MIMO (MU-MIMO) improves the spectrum efficiency by allowing a base station (BS) transmitter to communicate simultaneously with multiple mobile stations (MS) receivers using the same time-frequency resources. Massive MIMO allows the number of BS antenna elements to be on the order of tens or hundreds, thereby also increasing the number of data streams in a cell to a large value.

5G wireless systems use millimeter wave (mmWave) bands to take advantage of their wider bandwidth. The 5G systems also deploy large scale antenna arrays to mitigate severe propagation loss in the mmWave band.

The small mmWave band wavelengths allow an array to contain more elements within the same physical dimension. It becomes much more expensive to provide one transmit-receive (TR) module, or an RF chain, for each antenna element. Hybrid transceivers are a practical solution as they use a combination of analog beamformers in the RF and digital beamformers in the baseband domains, with fewer RF chains than the number of transmit elements [ 1 ].

**Hybrid Beamforming**

Hybrid beamforming (also known as hybrid precoding) is a method that enables the use of massive MIMO antenna arrays in a lower power and cost-efficient manner [ 1 ]. In a traditional antenna array, each antenna requires a dedicated RF chain to transmit and receive each data stream; with hybrid precoding, each stream requires a dedicated RF chain. This greatly reduces the number of RF chains, thus reducing cost and power. The analog outputs of each chain are combined into a network of analog RF gains and phase shifters (the analog RF beamformer, denoted `Frf`) that are connected to a large antenna array, where the number of antennas >> number of streams. These analog units cannot change weights quickly; however the computed RF weights change slowly over time since they are primarily determined by spatial positions of receivers. The digital baseband precoding weights (denoted `Fbb`) may change from symbol to symbol due to smaller-scale multipath effects, and may also be different from subcarrier to subcarrier to account for frequency-selective fading.

Massive Antenna Array
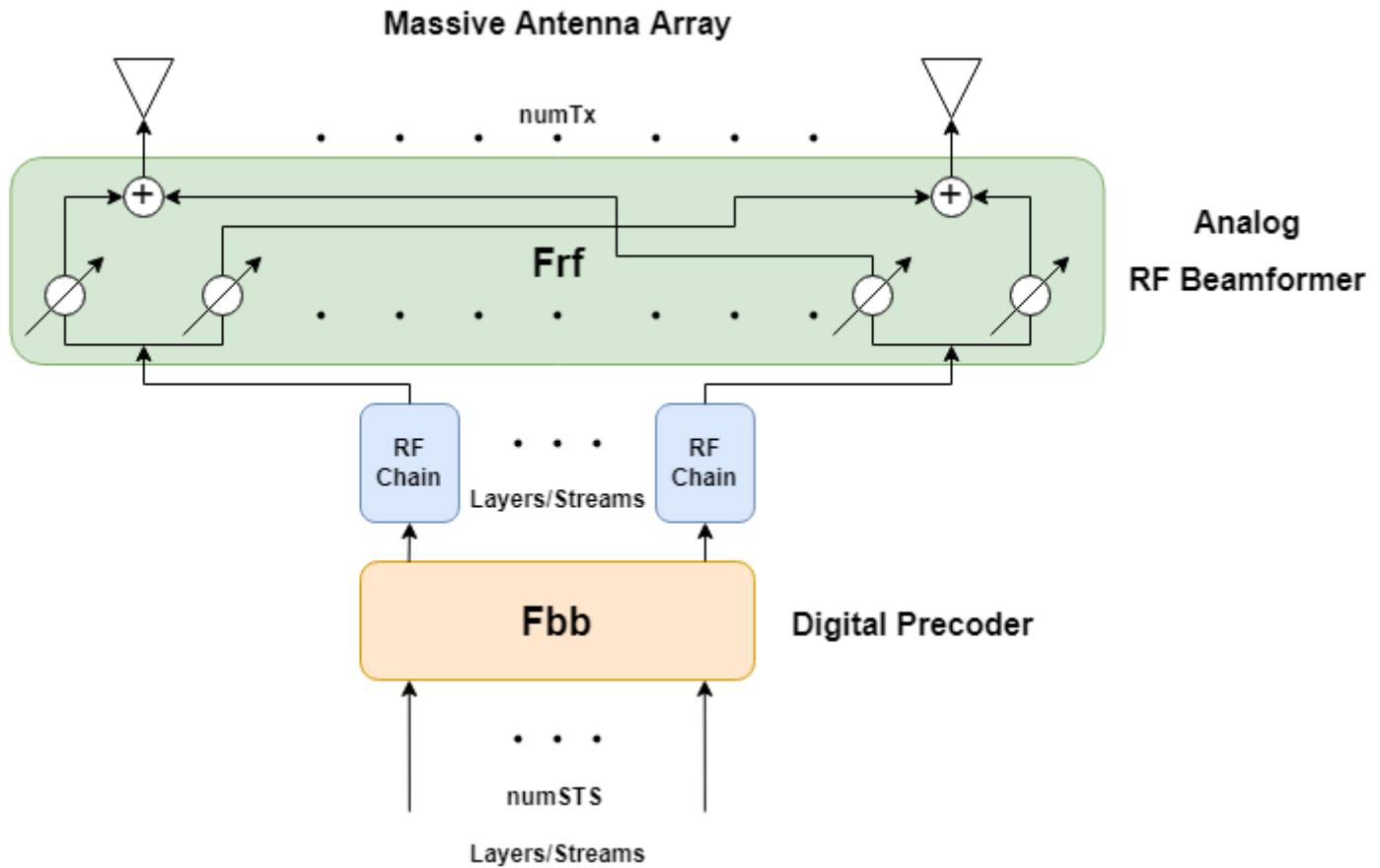
This example uses a single-user or multi-user MIMO-OFDM system to highlight the partitioning of the required precoding into its digital baseband and RF analog components at the transmitter end. The example uses the orthogonal matching pursuit (OMP) algorithm [ 3 ] for a single-user system and the joint spatial division multiplexing (JSDM) technique [ 2, 4 ] for a multi-user system, to determine the digital baseband `Fbb` and RF analog `Frf` precoding weights for the selected system configuration. Building on the system highlighted in the "MIMO-OFDM Precoding with Phased Arrays" on page 18-378 example, this example shows the formulation of the transmit-end precoding matrices and their application to a MIMO-OFDM system.

**Virtual Sectorization Using Joint Spatial Division Multiplexing**

JSDM takes advantage of spatial clustering of users within the cell and groups these clusters to create virtual sectors via RF analog beamforming to these groups. The spatial covariance matrices of the groups are related to the spatial direction of the groups with respect to the base station antenna array. These matrices are computed using the channel estimates produced from the sounding signals, and the analog RF beams are derived from these matrices. These beams focus transmitted energy to the respective groups and minimize intergroup interference.
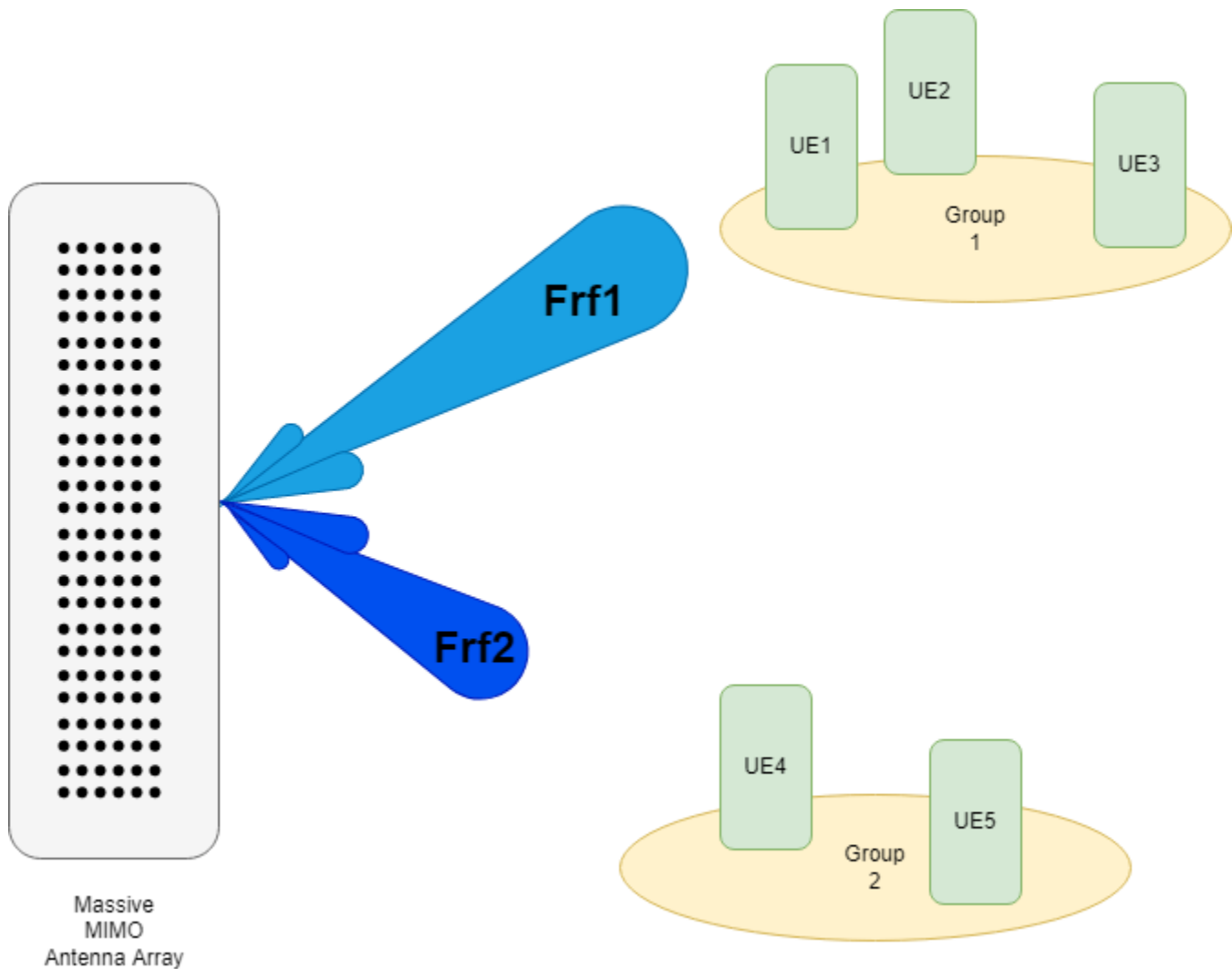
Once the beams are formed, the digital precoding weights are computed and used to orthogonalize users within each of the groups. These precoding weights are based on the user's channel state information (CSI) of the "effective channel" as seen from the output of the RF chains (not the output of the antenna array). Such beam-based measurements are done in 5G systems.

Because the analog RF gains and phase shifters are done at the antennas, it is implied that the RF weights apply across all subcarriers. However, the digital precoding weights may be applied per subcarrier. In this example, each subcarrier applies digital precoding weights as computed from the subcarrier's CSI.

```
s = rng(67);                      % Set RNG state for repeatability
```

**System Parameters**

Define system parameters for the example. Modify these parameters to explore their impact on the system.

```
prm.fc = 28e9;                    % 28 GHz system
prm.chanSRate = 100e6;            % Channel sampling rate, 100 Msps
```

```
prm.ChanType = 'Scattering'; % Channel options: 'Scattering', 'MIMO'
prm.NFig = 8;                % Noise figure (increase to worsen, 5-10 dB)
prm.nRays = 500;             % Number of rays for Frf, Fbb partitioning

% Each user has the same modulation
prm.bitsPerSubCarrier = 4;   % 2: QPSK, 4: 16QAM, 6: 64QAM, 8: 256QAM
prm.numDataSymbols = 10;     % Number of OFDM data symbols

% Create a vector of users where each element is a user and the value of
% that element describes the number of independent data streams. For a
% single user case, make this a scalar value.
prm.numSTSVec = [4 3 2 4 3];         % Number of independent data streams per user
prm.numSTS    = sum(prm.numSTSVec);  % Must be a power of 2
prm.numTx     = prm.numSTS*8;        % Number of BS transmit antennas (power of 2)
```

Assign each user to a group. Users are randomly arranged around the center of each group. For a single user use case, the concept of grouping is not applicable, and the user is randomly placed in the cell.

```
% The group assignments for the active users must be in ascending
% order. The default setting matches the group diagram above; there are two
% groups of users with the first three users assigned to group one and the
% last two users assigned to group two.
prm.groups    = [1 1 1 2 2];  % Describes which users are in which groups
```

Two JSDM options are available for simulation. This option determines the method for computing the digital precoding weights; the RF analog weights are calculated independent of which option is chosen.

- 'JGP': Joint Spatial Division Multiplexing with Joint Group Processing (JGP). MIMO streams are precoded based on the CSI feedback of the channel estimates for all users.

- 'PGP': Joint Spatial Division Multiplexing with Per-Group Processing (PGP). This is a special case where user CSI feedback is reduced to only send back the CSI of the streams transmitted to the user's group, thus increasing available bandwidth that was reserved for CSI feedback. Performance is slightly worse than JGP since precoding is done only on the streams for each group, and is particularly sensitive to the rank of the channel relative to the number of transmitted streams.

```
prm.jsdmType = 'JGP'; % JSDM option: 'JGP' or 'PGP' (for multi-user simulation)
```

Note that selecting the MIMO channel via the `ChanType` parameter field will cause sub-optimal performance for JSDM, since the MIMO channel is not location-aware. If using a scattering channel, add in an additional receiver in each group to improve the channel rank of the group's observed covariance matrix and support the number of streams being transmitted to each group. These receivers are not actively receiving data, but are considered connected to the base station as part of a group.

```
prm.numSTSVecAll = prm.numSTSVec;         % Number of independent data streams per user
prm.numUsers  = length(prm.numSTSVec);  % Number of active users
prm.numGroups = max(prm.groups);  % Number of groups
numGroups = prm.numGroups;
if strcmp(prm.ChanType,'Scattering')
    for g = 1:numGroups
        prm.groups(prm.numUsers+g) = g;
        prm.numSTSVecAll(prm.numUsers+g) = 2;
    end
```

```
    end

% Define the center of each group's range, azimuth, and elevation,
% assuming the BS is at the origin.
%   Angles specified as [azimuth;elevation] degrees
% Note: the number of columns must equal the number of groups
maxRange = 700;              % all MSs within 700 meters of BS
groupRanges = randi([1 maxRange],1,numGroups);
groupAzimuth = -60 + 60/(numGroups+1) + ...
    (120-(120/(numGroups+1)))/(numGroups-1) * (0:numGroups-1);
groupElevations = randi([-10 10],1,numGroups);

% Position mobile units
prm.numConnectedUsers = length(prm.numSTSVecAll); % Number of connected users
if prm.numUsers == 1
    % For a single user, randomly place the user
    % MS positions: assumes BS at origin
    %   Angles specified as [azimuth;elevation] degrees
    %   az in range [-180 180], el in range [-90 90], e.g. [45;0]
    prm.mobileRanges = randi([1 maxRange],1,prm.numConnectedUsers);
    prm.mobileAngles = [rand(1,prm.numConnectedUsers)*360-180; ...
                        rand(1,prm.numConnectedUsers)*180-90];
else
    % For multiple users, randomly place the users within their assigned
    % group locations
    prm.mobileRanges = zeros(prm.numConnectedUsers,1);
    prm.mobileAngles = zeros(2,prm.numConnectedUsers);
    for uIdx = 1:prm.numConnectedUsers
        g = prm.groups(uIdx);
        prm.mobileRanges(uIdx)  = groupRanges(g) + 30*rand(1);
        prm.mobileAngles(:,uIdx) = ...
            [groupAzimuth(g)+3*rand(1); groupElevations(g)+3*rand(1)];
    end
end
```

Define OFDM modulation parameters used for the system.

```
prm.FFTLength = 256;
prm.CyclicPrefixLength = 64;
prm.numCarriers = 234;
prm.NullCarrierIndices = [1:7 129 256-5:256]'; % Guards and DC
prm.PilotCarrierIndices = [26 54 90 118 140 168 204 232]';
nonDataIdx = [prm.NullCarrierIndices; prm.PilotCarrierIndices];
prm.CarriersLocations = setdiff((1:prm.FFTLength)', sort(nonDataIdx));
prm.numRx = prm.numSTSVecAll;   % Number of receive antennas, per connected user

numSTS = prm.numSTS;
numTx = prm.numTx;
numRx = prm.numRx;
numSTSVec = prm.numSTSVec;
numUsers = prm.numUsers;
codeRate = 1/3;                 % same code rate per user
numTails = 6;                   % number of termination tail bits
prm.numFrmBits = numSTSVec.*(prm.numDataSymbols*prm.numCarriers* ...
                 prm.bitsPerSubCarrier*codeRate)-numTails;
prm.modMode = 2^prm.bitsPerSubCarrier; % Modulation order
% Account for channel filter delay
```

```matlab
numPadSym = 3;              % number of symbols to zero pad
prm.numPadZeros = numPadSym*(prm.FFTLength+prm.CyclicPrefixLength);
```

Define transmit and receive arrays and positional parameters for the system.

```matlab
prm.cLight = physconst('LightSpeed');
prm.lambda = prm.cLight/prm.fc;

% Get transmit and receive array information
prm.numSTSVec = prm.numSTSVecAll; % get array info for all connected users
prm.numUsers = prm.numConnectedUsers;
[isTxURA,expFactorTx,isRxURA,expFactorRx] = helperArrayInfo(prm,true);
prm.numSTSVec = numSTSVec;                  % restore parameters to active users
prm.numUsers = numUsers;

% Transmit antenna array definition
%   Array locations and angles
prm.posTx = [0;0;0];          % BS/Transmit array position, [x;y;z], meters
if isTxURA
    % Uniform Rectangular array
    txarray = phased.PartitionedArray(...
        'Array',phased.URA([expFactorTx numSTS],0.5*prm.lambda),...
        'SubarraySelection',ones(numSTS,numTx),'SubarraySteering','Custom');
else
    % Uniform Linear array
    txarray = phased.ULA(numTx, 'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',false));
end
prm.posTxElem = getElementPosition(txarray)/prm.lambda;

spLoss = zeros(prm.numConnectedUsers,1);
prm.posRx = zeros(3,prm.numConnectedUsers);
for uIdx = 1:prm.numConnectedUsers

    % Receive arrays
    if isRxURA(uIdx)
        % Uniform Rectangular array
        rxarray = phased.PartitionedArray(...
            'Array',phased.URA([expFactorRx(uIdx) numRx(uIdx)], ...
            0.5*prm.lambda),'SubarraySelection',ones(numRx(uIdx), ...
            numRx(uIdx)),'SubarraySteering','Custom');
        prm.posRxElem = getElementPosition(rxarray)/prm.lambda;
    else
        if numRx(uIdx)>1
            % Uniform Linear array
            rxarray = phased.ULA(numRx(uIdx), ...
                'ElementSpacing',0.5*prm.lambda, ...
                'Element',phased.IsotropicAntennaElement);
            prm.posRxElem = getElementPosition(rxarray)/prm.lambda;
        else
            rxarray = phased.IsotropicAntennaElement;
            prm.posRxElem = [0; 0; 0]; % LCS
        end
    end

    % Mobile positions
    [xRx,yRx,zRx] = sph2cart(deg2rad(prm.mobileAngles(1,uIdx)), ...
                             deg2rad(prm.mobileAngles(2,uIdx)), ...
```

```
                                prm.mobileRanges(uIdx));
    prm.posRx(:,uIdx) = [xRx;yRx;zRx];
    [toRxRange,toRxAng] = rangeangle(prm.posTx,prm.posRx(:,uIdx));
    spLoss(uIdx) = fspl(toRxRange,prm.lambda);
end
```

**Channel State Information**

For a spatially multiplexed system, availability of channel information at the transmitter allows for precoding to be applied to maximize the signal energy in the direction and channel of interest. Under the assumption of a slowly varying channel, this is facilitated by sounding the channel first. The BS sounds the channel by using a reference transmission, that the MS receiver uses to estimate the channel. The MS transmits the channel estimate information back to the BS for calculation of the precoding needed for the subsequent data transmission.

The following schematic shows the processing for the channel sounding modeled.



**Channel Sounding**
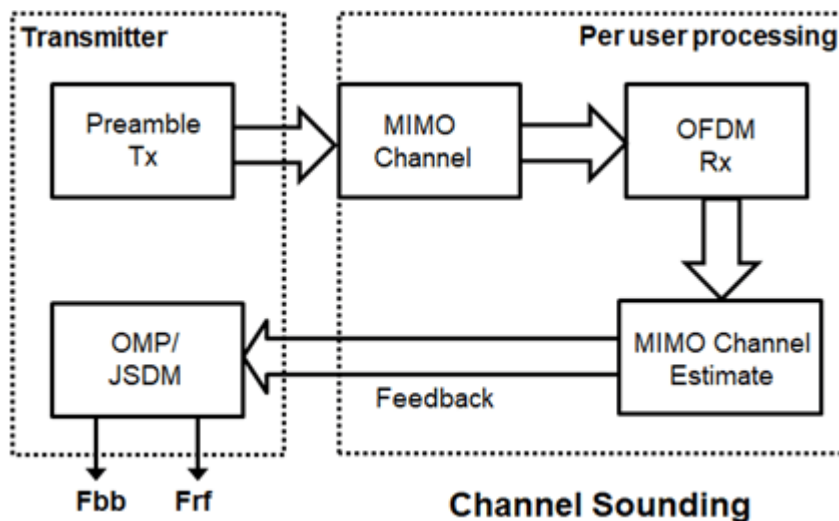
For the chosen MIMO system, a preamble signal is sent over all transmitting antenna elements, and processed at the receiver accounting for the channel. The receiver antenna elements perform pre-amplification, OFDM demodulation, and frequency domain channel estimation for all links.

```
% Generate the preamble signal
prm.numSTS = numTx;                  % set to numTx to sound out all channels
preambleSig = helperGenPreamble(prm);

% Transmit preamble over channel
prm.numSTS = numSTS;                 % keep same array config for channel
prm.numUsers = prm.numConnectedUsers; % transmit sounding to all connected users
prm.numSTSVec = prm.numSTSVecAll; % transmit sounding to all connected users
[rxPreSig,chanDelay] = helperApplyMUChannel(preambleSig,prm,spLoss);
prm.numUsers = numUsers;
prm.numSTSVec = numSTSVec;

% Channel state information feedback
hDp = cell(prm.numConnectedUsers,1);
prm.numSTS = numTx;                  % set to numTx to estimate all links
for uIdx = 1:prm.numConnectedUsers
```

```
    % Front-end amplifier gain and thermal noise
    rxPreAmp = phased.ReceiverPreamp( ...
        'Gain',spLoss(uIdx), ...      % account for path loss
        'NoiseFigure',prm.NFig,'ReferenceTemperature',290, ...
        'SampleRate',prm.chanSRate);
    rxPreSigAmp = rxPreAmp(rxPreSig{uIdx});
    %    scale power for used sub-carriers
    rxPreSigAmp = rxPreSigAmp * (sqrt(prm.FFTLength - ...
        length(prm.NullCarrierIndices))/prm.FFTLength);

    % OFDM demodulation
    rxOFDM = ofdmdemod(rxPreSigAmp(chanDelay(uIdx)+1: ...
        end-(prm.numPadZeros-chanDelay(uIdx)),:),prm.FFTLength, ...
        prm.CyclicPrefixLength,prm.CyclicPrefixLength, ...
        prm.NullCarrierIndices,prm.PilotCarrierIndices);

    % Channel estimation from preamble
    %        numCarr, numTx, numRx
    hDp{uIdx} = helperMIMOChannelEstimate(rxOFDM(:,1:numTx,:),prm);

end
prm.numSTS = numSTS;                        % revert back for data transmission
prm.numRx  = prm.numRx(1:numUsers);  % set numRx to number of active users
```

For a single-user system, the OMP partitioning algorithm is sensitive to the array response vectors At. Ideally, these response vectors account for all the scatterers seen by the channel, but these are unknown for an actual system and channel realization, so a random set of rays within a 3-dimensional space to cover as many scatterers as possible is used. The prm.nRays parameter specifies the number of rays.

For a multi-user system, the JSDM algorithm requires the calculation of the spatial covariance from each group to calculate the RF beamformer weights based on the block diagonalization method 5. The channel estimate is fed back from each MS, and used by the BS to determine the precoding weights. The example assumes perfect feedback with no quantization or implementation delays. The effective channel is calculated from the RF beamformer weights and the CSI, and the digital baseband precoder weights are then calculated.

```
% Calculate the hybrid weights on the transmit side
if numUsers==1
    % Single-user OMP
    %    Spread rays in [az;el]=[-180:180;-90:90] 3D space, equal spacing
    %    txang = [-180:360/prm.nRays:180; -90:180/prm.nRays:90];
    txang = [rand(1,prm.nRays)*360-180;rand(1,prm.nRays)*180-90]; % random
    At = steervec(prm.posTxElem,txang);
    AtExp = complex(zeros(prm.numCarriers,size(At,1),size(At,2)));
    for carrIdx = 1:prm.numCarriers
        AtExp(carrIdx,:,:) = At; % same for all sub-carriers
    end

    % Orthogonal matching pursuit hybrid weights
    [Fbb,Frf] = omphybweights(hDp{1},numSTS,numSTS,AtExp);

    v = Fbb;      % set the baseband precoder (Fbb)
    % Frf is same across subcarriers for flat channels
    mFrf = permute(mean(Frf,1),[2 3 1]);
```

```matlab
        else
            % Multi-user Joint Spatial Division Multiplexing

            % Get number of streams per group
            prm.numSTSGroupVec = zeros(1,numGroups);
            for g = 1:numGroups
                prm.numSTSGroupVec(g) = sum(prm.numSTSVecAll(prm.groups(1:prm.numUsers)==g));
            end

            % Get Rg (group covariances)
            % Note that there is only one R per group (not per user). It is up to the
            % caller to determine how this autocorrelation matrix is obtained from the
            % group.
            hcov = helperGetCovariance(hDp,prm);

            % Calculate the analog beamformer weights
            Bg = jsdmrfweights(hcov,prm.numSTSGroupVec);

            % Pack beamformer weights into a precoding array
            mFrf = cat(1,Bg{:});

            % Calculate (predict) the effective channel response for each user from
            % a beamformed sounding signal
            hEff = helperGetHeff(hDp,mFrf,prm);

            % Get the digital (baseband) weights
            Fbb = jsdmbbweights(hEff,prm.numSTSGroupVec,Bg);

            if strcmp(prm.jsdmType,'PGP')
                % Pack the per group CSI into a matrix (block diagonal) for PGP
                v = zeros(prm.numCarriers,sum(numSTSVec),sum(numSTSVec));
                for g = 1:prm.numGroups
                    stsIdx = sum(prm.numSTSGroupVec(1:g-1))+(1:prm.numSTSGroupVec(g));
                    v(:,stsIdx,stsIdx) = Fbb{g};  % Nst-by-Nsts-by-Nsts
                end
            else
                % JGP
                v = Fbb;
            end

    end
```

Plot the transmit array pattern. For a scattering channel, the beam directions should correlate closely with the spatial positions of the users, and the beam magnitude should correlate with the number of streams being sent in that direction.

```matlab
% Transmit array pattern plots
figure;
if isTxURA
    % URA element response for the first subcarrier
    pattern(txarray,prm.fc,-180:180,-90:90,'Type','efield', ...
            'ElementWeights',mFrf.'*squeeze(v(1,:,:)), ...
            'PropagationSpeed',prm.cLight);
else % ULA
    % Array response for first subcarrier
    wts = mFrf.'*squeeze(v(1,:,:));
    pattern(txarray,prm.fc,-180:180,-90:90,'Type','efield', ...
```

```
                'Weights',wts(:,1),'PropagationSpeed',prm.cLight);
end
```



**3D Response Pattern**
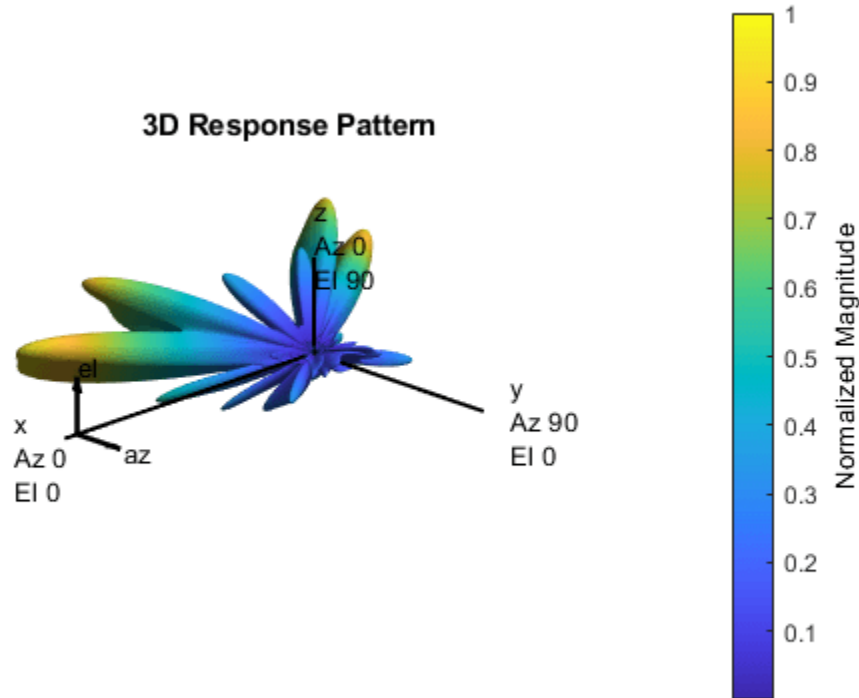
For the wideband OFDM system modeled, the analog weights, `mFrf`, are the averaged weights over the multiple subcarriers. The array response pattern shows distinct data streams represented by the stronger lobes. These lobes indicate the spread or separability achieved by beamforming. The "Introduction to Hybrid Beamforming" on page 18-371 example compares the patterns realized by the optimal, fully digital approach, with those realized from the selected hybrid approach, for a single-user system.

**Data Transmission**

Next, we configure the system's data transmitter. This processing includes channel coding, bit mapping to complex symbols, splitting of the individual data stream to multiple transmit streams, baseband precoding of the transmit streams, OFDM modulation with pilot mapping and RF analog beamforming for all the transmit antennas employed.

```
% Convolutional encoder
encoder = comm.ConvolutionalEncoder( ...
    'TrellisStructure',poly2trellis(7,[133 171 165]), ...
    'TerminationMethod','Terminated');

txDataBits = cell(numUsers, 1);
gridData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for uIdx = 1:numUsers
    % Generate mapped symbols from bits per user
    txDataBits{uIdx} = randi([0,1],prm.numFrmBits(uIdx),1);
```

```
    encodedBits = encoder(txDataBits{uIdx});

    % Bits to QAM symbol mapping
    mappedSym = qammod(encodedBits,prm.modMode,'InputType','bit', ...
    'UnitAveragePower',true);

    % Map to layers: per user, per symbol, per data stream
    stsIdx = sum(numSTSVec(1:(uIdx-1)))+(1:numSTSVec(uIdx));
    gridData(:,:,stsIdx) = reshape(mappedSym,prm.numCarriers, ...
        prm.numDataSymbols,numSTSVec(uIdx));
end

% Apply precoding weights to the subcarriers, assuming perfect feedback
preData = complex(zeros(prm.numCarriers,prm.numDataSymbols,numSTS));
for symIdx = 1:prm.numDataSymbols
    for carrIdx = 1:prm.numCarriers
        Q = squeeze(v(carrIdx,:,:));
        normQ = Q * sqrt(numTx)/norm(Q,'fro');
        preData(carrIdx,symIdx,:) = squeeze(gridData(carrIdx,symIdx,:)).' ...
            * normQ;
    end
end

% Multi-antenna pilots
pilots = helperGenPilots(prm.numDataSymbols,numSTS);

% OFDM modulation of the data
txOFDM = ofdmmod(preData,prm.FFTLength,prm.CyclicPrefixLength,...
                 prm.NullCarrierIndices,prm.PilotCarrierIndices,pilots);
%   scale power for used sub-carriers
txOFDM = txOFDM * (prm.FFTLength/ ...
    sqrt((prm.FFTLength-length(prm.NullCarrierIndices))));

% Generate preamble with the feedback weights and prepend to data
preambleSigD = helperGenPreamble(prm,v);
txSigSTS = [preambleSigD;txOFDM];

% RF beamforming: Apply Frf to the digital signal
%   Each antenna element is connected to each data stream
txSig = txSigSTS*mFrf;
```

For the selected, fully connected RF architecture, each antenna element uses `prm.numSTS` analog gains, as given by the individual columns of the `mFrf` matrix.

The processing for the data transmission and reception modeled is shown below.

**Data Transmission & Reception**
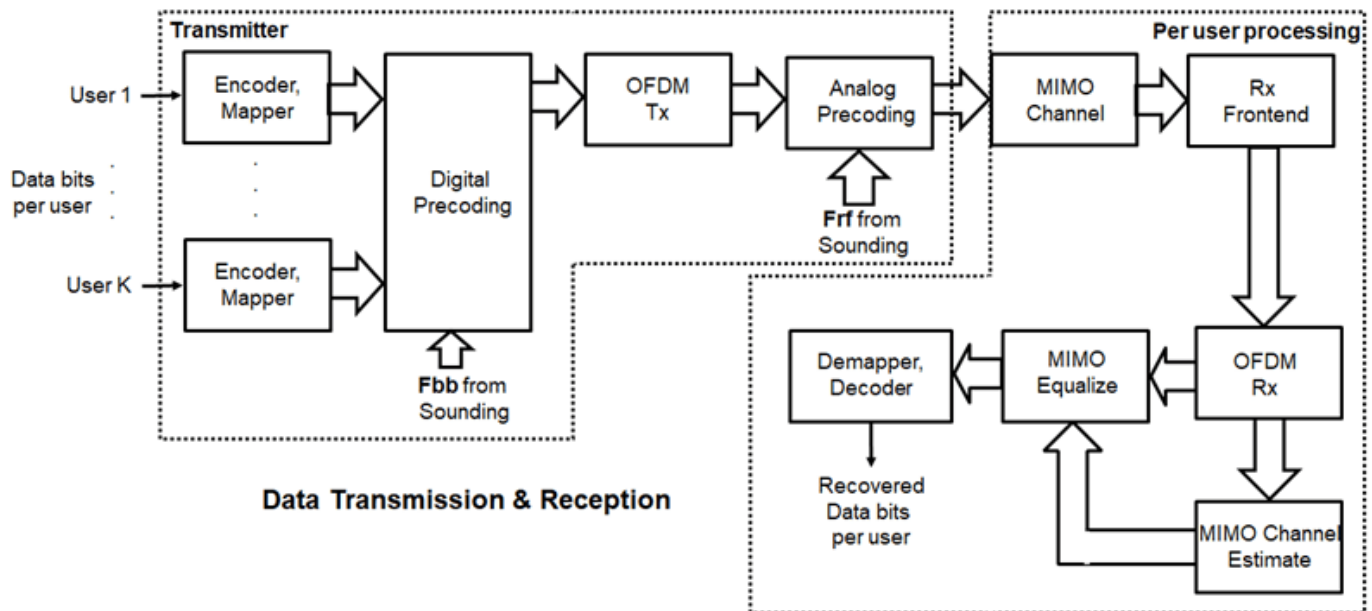
### Signal Propagation

The example offers an option for spatial MIMO channel and a simpler static-flat MIMO channel for validation purposes.

The scattering model uses a single-bounce ray tracing approximation with a parametrized number of scatterers. For this example, the number of scatterers is set to 50. The 'Scattering' option models the scatterers placed randomly within a sphere around the receiver, similar to the one-ring model [ 6 ].

The channel models allow path-loss modeling and both line-of-sight (LOS) and non-LOS propagation conditions. The example assumes non-LOS propagation and isotropic antenna element patterns with linear or rectangular geometry.

```
% Apply a spatially defined channel to the transmit signal
[rxSig,chanDelay] = helperApplyMUChannel(txSig,prm,spLoss,preambleSig);
```

The same channel is used for both sounding and data transmission. The data transmission has a longer duration and is controlled by the number of data symbols parameter, `prm.numDataSymbols`. The channel evolution between the sounding and transmission stages is modeled by prepending the preamble signal to the data signal. The preamble primes the channel to a valid state for the data transmission, and is ignored from the channel output.

For a multi-user system, independent channels per user are modeled.

### Receive Amplification and Signal Recovery

The receiver modeled per user compensates for the path loss by amplification and adds thermal noise. Like the transmitter, the receiver used in a MIMO-OFDM system contains many stages including OFDM demodulation, MIMO equalization, QAM demapping, and channel decoding.

```
hfig = figure('Name','Equalized symbol constellation per stream');
scFact = ((prm.FFTLength-length(prm.NullCarrierIndices))...
        /prm.FFTLength^2)/numTx;
nVar = noisepow(prm.chanSRate,prm.NFig,290)/scFact;
```

```matlab
decoder = comm.ViterbiDecoder('InputFormat','Unquantized', ...
    'TrellisStructure',poly2trellis(7, [133 171 165]), ...
    'TerminationMethod','Terminated','OutputDataType','double');

for uIdx = 1:numUsers
    stsU = numSTSVec(uIdx);
    stsIdx = sum(numSTSVec(1:(uIdx-1)))+(1:stsU);

    % Front-end amplifier gain and thermal noise
    rxPreAmp = phased.ReceiverPreamp( ...
        'Gain',spLoss(uIdx), ...          % account for path loss
        'NoiseFigure',prm.NFig,'ReferenceTemperature',290, ...
        'SampleRate',prm.chanSRate);
    rxSigAmp = rxPreAmp(rxSig{uIdx});

    % Scale power for occupied sub-carriers
    rxSigAmp = rxSigAmp*(sqrt(prm.FFTLength-length(prm.NullCarrierIndices)) ...
        /prm.FFTLength);

    % OFDM demodulation
    rxOFDM = ofdmdemod(rxSigAmp(chanDelay(uIdx)+1: ...
        end-(prm.numPadZeros-chanDelay(uIdx)),:),prm.FFTLength, ...
        prm.CyclicPrefixLength,prm.CyclicPrefixLength, ...
        prm.NullCarrierIndices,prm.PilotCarrierIndices);

    % Channel estimation from the mapped preamble
    hD = helperMIMOChannelEstimate(rxOFDM(:,1:numSTS,:),prm);

    % MIMO equalization
    %   Index into streams for the user of interest
    [rxEq,CSI] = ofdmEqualize(rxOFDM(:,numSTS+1:end,:),hD(:,stsIdx,:),'Algorithm','zf');

    % Soft demodulation
    rxSymbs = rxEq(:)/sqrt(numTx);
    rxLLRBits = qamdemod(rxSymbs,prm.modMode,'UnitAveragePower',true, ...
        'OutputType','approxllr','NoiseVariance',nVar);

    % Apply CSI prior to decoding
    rxLLRtmp = reshape(rxLLRBits,prm.bitsPerSubCarrier,[], ...
        prm.numDataSymbols,stsU);
    csitmp = reshape(CSI,1,[],1,numSTSVec(uIdx));
    rxScaledLLR = rxLLRtmp.*csitmp;

    % Soft-input channel decoding
    rxDecoded = decoder(rxScaledLLR(:));

    % Decoded received bits
    rxBits = rxDecoded(1:prm.numFrmBits(uIdx));

    % Plot equalized symbols for all streams per user
    scaler = 2;
    for i = 1:stsU
        subplot(numUsers, max(numSTSVec), (uIdx-1)*max(numSTSVec)+i);
        plot(reshape(rxEq(:,:,i)/sqrt(numTx), [], 1), '.');
        axis square
        xlim(gca,[-scaler scaler]);
        ylim(gca,[-scaler scaler]);
        title(['U' num2str(uIdx) ', G' num2str(prm.groups(uIdx)) ', DS' num2str(i)]);
```

```
        grid on;
    end

    % Compute and display the EVM
    evm = comm.EVM('Normalization','Average constellation power', ...
        'ReferenceSignalSource','Estimated from reference constellation', ...
        'ReferenceConstellation', ...
        qammod((0:prm.modMode-1)',prm.modMode,'UnitAveragePower',1));
    rmsEVM = evm(rxSymbs);
    disp(['User ' num2str(uIdx) ' (Group ' num2str(prm.groups(uIdx)) ')']);
    disp(['   RMS EVM (%) = ' num2str(rmsEVM)]);

    % Compute and display bit error rate
    ber = comm.ErrorRate;
    measures = ber(txDataBits{uIdx},rxBits);
    fprintf('   BER = %.5f; No. of Bits = %d; No. of errors = %d\n', ...
        measures(1),measures(3),measures(2));
end

User 1 (Group 1)
  RMS EVM (%) = 0.00051611
  BER = 0.00000; No. of Bits = 12474; No. of errors = 0
User 2 (Group 1)
  RMS EVM (%) = 0.00053232
  BER = 0.00000; No. of Bits = 9354; No. of errors = 0
User 3 (Group 1)
  RMS EVM (%) = 0.00070922
  BER = 0.00000; No. of Bits = 6234; No. of errors = 0
User 4 (Group 2)
  RMS EVM (%) = 0.00081068
  BER = 0.00000; No. of Bits = 12474; No. of errors = 0
User 5 (Group 2)
  RMS EVM (%) = 0.00055681
  BER = 0.00000; No. of Bits = 9354; No. of errors = 0
```
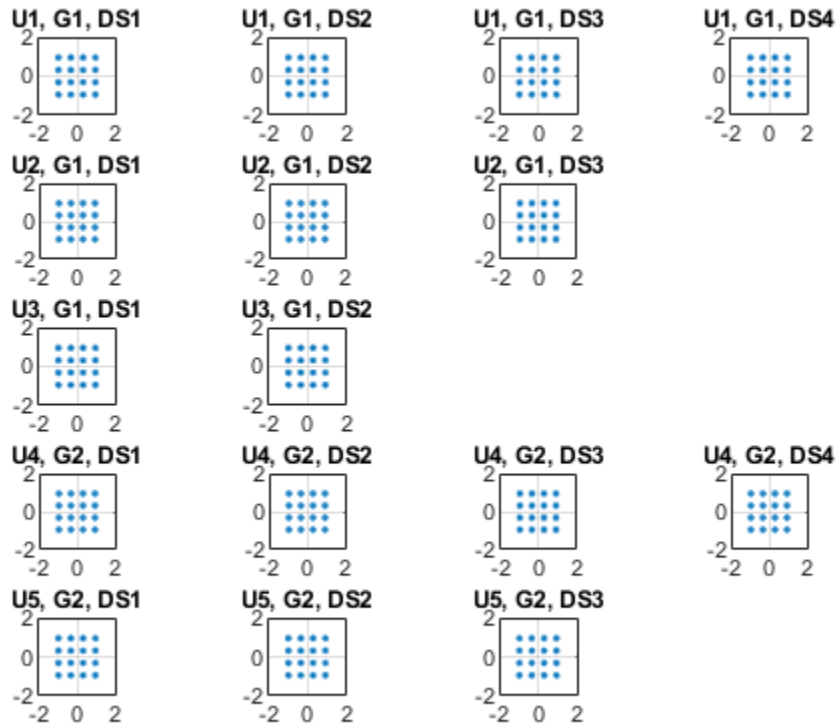
For the MIMO system modeled, the displayed receive constellation of the equalized symbols offers a qualitative assessment of the reception. The actual bit error rate offers the quantitative figure by comparing the actual transmitted bits with the received decoded bits per user.

```
rng(s);            % restore RNG state
```

**Conclusion and Further Exploration**

The example highlights the use of hybrid beamforming for multi-user MIMO-OFDM systems. It allows you to explore different system configurations for a variety of channel models by changing a few system-wide parameters.

The set of configurable parameters includes the number of users, number of data streams per user, number of transmit/receive antenna elements, array locations, and channel models. Adjusting these parameters, you can study the parameters' individual or combined effects on the overall system. As examples, vary:

- the number of users and corresponding data streams, `prm.numSTSVec`, and their corresponding group, `prm.groups`, to switch between multi-user and single-user systems, or
- the channel type, `prm.ChanType`, or
- the number of rays, `prm.nRays`, used for a single-user system, or
- the JSDM feedback type, `prm.jsdmType`, and the group locations, `groupAngles` and `groupRanges`, used in the multi-user system.

Explore the following helper functions used by the example:

- helperApplyMUChannel.m
- helperArrayInfo.m
- helperGetCovariance.m
- helperGenPreamble.m
- helperGenPilots.m
- helperGetHeff.m
- helperGetP.m
- helperMIMOChannelEstimate.m

To see JSDM used in a 5G communications system, see the "TDD Reciprocity-Based PDSCH MU-MIMO Using SRS" (5G Toolbox) example.

**References**

**1**   Molisch, A. F., et al. "Hybrid Beamforming for Massive MIMO: A Survey." IEEE® Communications Magazine, Vol. 55, No. 9, September 2017, pp. 134-141.

**2**   Li Z., S. Han, and A. F. Molisch. "Hybrid Beamforming Design for Millimeter-Wave Multi-User Massive MIMO Downlink." IEEE ICC 2016, Signal Processing for Communications Symposium.

**3**   El Ayach, Oma, et al. "Spatially Sparse Precoding in Millimeter Wave MIMO Systems." IEEE Transactions on Wireless Communications, Vol. 13, No. 3, March 2014, pp. 1499-1513.

**4**   Adhikary A., J. Nam, J-Y Ahn, and G. Caire. "Joint Spatial Division and Multiplexing - The Large-Scale Array Regime." IEEE Transactions on Information Theory, Vol. 59, No. 10, October 2013, pp. 6441-6463.

**5**   Spencer Q., A. Swindlehurst, M. Haardt, "Zero-Forcing Methods for Downlink Spatial Multiplexing in Multiuser MIMO Channels." IEEE Transactions on Signal Processing, Vol. 52, No. 2, February 2004, pp. 461-471.

**6**   Shui, D. S., G. J. Foschini, M. J. Gans and J. M. Kahn. "Fading Correlation and its Effect on the Capacity of Multielement Antenna Systems." IEEE Transactions on Communications, Vol. 48, No. 3, March 2000, pp. 502-513.

# Multicore Simulation of Test Signals for a Radar Receiver

This example simulates a monostatic radar system. It uses dataflow domain in Simulink® to automatically partition the data-driven portions of the radar system into multiple threads and thereby improving the performance of the simulation by executing it on your desktop's multiple cores.

**Introduction**

The dataflow execution domain allows you to make use of multiple cores in the simulation of computationally intensive systems. This example shows how dataflow as the execution domain of a subsystem improves simulation performance of the model. To learn more about dataflow and how to run Simulink models using multiple threads, see "Multicore Execution using Dataflow Domain".

**Monostatic Radar with One Target**

This example simulates a simple end-to-end monostatic radar. Rectangular pulses are amplified by the transmitter block then propagated to and from a target in free-space. Noise and amplification are then applied in the receiver preamp block to the return signal, followed by a matched filter. Range losses are compensated for and the pulses are noncoherently integrated.
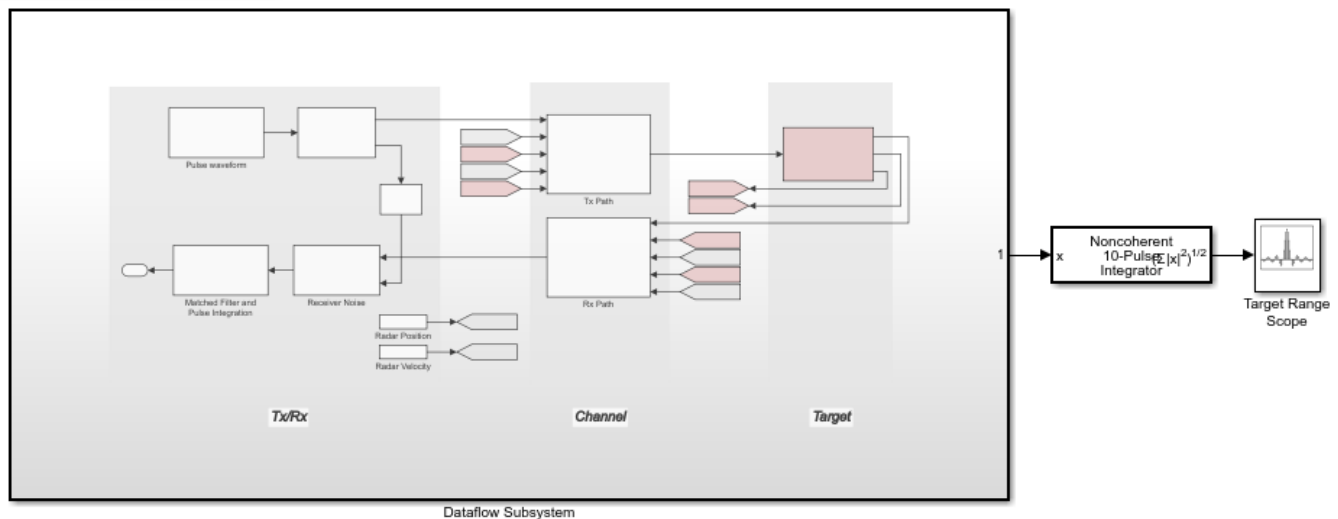


Copyright 2014-2022 The Mathworks Inc.

**Setting up the Dataflow Subsystem**

This example uses dataflow domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. The **Domain** parameter of the dataflow subsystem in this model is set as `Dataflow`. You can view this by selecting the subsystem and then accessing Property Inspector. To access Property Inspector, in the Simulink Toolstrip, on the Modeling tab, in the Design gallery select Property Inspector or on the Simulation tab, Prepare gallery, select Property Inspector.

Dataflow domains automatically partition your model into multiple threads for better performance. Once you set the **Domain** parameter to `Dataflow`, you can use the **Multicore** tab analysis to analyze your model to get better performance. The **Multicore** tab is available in the toolstrip when there is a dataflow domain in the model. To learn more about the **Multicore** tab, see "Perform Multicore Analysis for Dataflow".



### Analyzing Concurrency in Dataflow Subsystem

For this example the **Multicore** tab mode is set to `Simulation Profiling` for simulation performance analysis.

It is recommended to optimize model settings for optimal simulation performance. To accept the proposed model settings, on the **Multicore** tab, click **Optimize**. Alternatively, you can use the drop menu below the **Optimize** button to change the settings individually. In this example the model settings are already optimal.



On the **Multicore** tab, click the **Run Analysis** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Analysis Report and Suggestions window shows how many threads the dataflow subsystem uses during simulation.

After analyzing the model, the Analysis Report and Suggestions window shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the dataflow subsystem can increase concurrency for higher data throughput. The Analysis Report and Suggestions window shows the recommended number of pipeline delays as Suggested for Increasing Concurrency. The suggested latency value is computed to give the best performance.

The following diagram shows the Analysis Report and Suggestions window where the suggested latency is 5 for the Dataflow Subsystem.

Click the **Accept** button to use the recommended latency for the dataflow subsystem. This value can also be entered directly in the Property Inspector for **Latency** parameter. Simulink shows the latency parameter value using $Z^{-n}$ tags at the output ports of the dataflow subsystem.

The Analysis Report and Suggestions window now shows the number of threads as 4 meaning that the blocks inside the dataflow subsystem simulate in parallel using 4 threads. **Highlight threads** highlights the blocks with colors based on their thread allocation as shown in the **Thread Highlighting Legend**. **Show pipeline delays** shows where pipelining delays were inserted within the dataflow subsystem using $Z^{-n}$ tags.



## Multicore Simulation Performance

We measure the performance improvement of using dataflow domain by comparing the execution time taken for running the model with and without using dataflow. Execution time is measured using the sim command, which returns the simulation execution time of the model. These numbers and analysis were published on a Windows® desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor.

```
Simulation execution time for multithreaded model = 18.90s
Simulation execution time for single-threaded model = 36.01s
Actual speedup with dataflow: 1.9x
```

**Summary**

This example shows how dataflow execution domain can improve performance in simulation of a radar system by using multiple cores on the desktop.

# Multicore Simulation of Audio Beamforming System

This example shows how an audio beamforming system simulation model in Simulink® can have improved performance using dataflow domain. It uses the dataflow domain in Simulink to automatically partition the data-driven portions of the communications system into multiple threads and thereby improving the performance of the simulation by executing it on your desktop's multiple cores.

### Introduction

The dataflow execution domain allows you to make use of multiple cores in the simulation of computationally intensive systems. This example shows how dataflow as the execution domain of a subsystem improves simulation performance of the model. To learn more about dataflow and how to run Simulink models using multiple threads, see "Multicore Execution using Dataflow Domain".

### Acoustic Beamforming

This example shows acoustic beamforming using a uniform linear array (ULA) of microphones. The model simulates the reception of three audio signals from different directions on a 10-element uniformly spaced linear microphone array. After the addition of thermal noise at the receiver, beamforming is applied for different source angles and the result is played on a sound device. The audio source that needs to be played in the audio player can be selected using the dialog from the Select Source block.



### Setting up the Dataflow Subsystem

This example uses dataflow domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. The **Domain** parameter of the dataflow subsystem in this model is set as `Dataflow`. You can view this by selecting the subsystem and then accessing Property Inspector. To access Property Inspector, in the Simulink Toolstrip, on the Modeling tab, in the Design gallery select Property Inspector or on the Simulation tab, Prepare gallery, select Property Inspector.

Dataflow domains automatically partition your model into multiple threads for better performance. Once you set the **Domain** parameter to `Dataflow`, you can use the **Multicore** tab analysis to analyze your model to get better performance. The **Multicore** tab is available in the toolstrip when there is a dataflow domain in the model. To learn more about the **Multicore** tab, see "Perform Multicore Analysis for Dataflow".



### Analyzing Concurrency in Dataflow Subsystem

For this example the **Multicore** tab mode is set to `Simulation Profiling` for simulation performance analysis.

It is recommended to optimize model settings for optimal simulation performance. To accept the proposed model settings, on the **Multicore** tab, click **Optimize**. Alternatively, you can use the drop menu below the **Optimize** button to change the settings individually. In this example the model settings are already optimal.



On the **Multicore** tab, click the **Run Analysis** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Analysis Report and Suggestions window shows how many threads the dataflow subsystem uses during simulation.

After analyzing the model, the Analysis Report and Suggestions window shows 3 threads. This is because the three beamformer blocks are computationally intensive and can run in parallel. The three beamformer blocks however, depend on the Microphone Array and the Receiver blocks. Pipeline delays can be used to break this dependency and increase concurrency. The Analysis Report and Suggestions window shows the recommended number of pipeline delays as Suggested for Increasing Concurrency. The suggested latency value is computed to give the best performance.

The following diagram shows the Analysis Report and Suggestions window where the suggested latency is 1 for the dataflow subsystem.

Click the **Accept** button to use the recommended latency for the dataflow subsystem. This value can also be entered directly in the Property Inspector for **Latency** parameter. Simulink shows the latency parameter value using $Z^{-n}$ tags at the output ports of the dataflow subsystem.

The Analysis Report and Suggestions window now shows the number of threads as 4 meaning that the blocks inside the dataflow subsystem simulate in parallel using 4 threads. **Highlight threads** highlights the blocks with colors based on their thread allocation as shown in the **Thread Highlighting Legend**. **Show pipeline delays** shows where pipelining delays were inserted within the dataflow subsystem using $Z^{-n}$ tags.



### Multicore Simulation Performance

We measure the performance improvement of using dataflow domain by comparing the execution time taken for running model with and without using dataflow. Execution time is measured using the sim command, which returns the simulation execution time of the model. These numbers and analysis were published on a Windows® desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor.

```
Simulation execution time for multithreaded model = 4.03s
Simulation execution time for single-threaded model = 6.26s
Actual speedup with dataflow: 1.6x
```

**Summary**

This example shows how multithreading using dataflow domain can improve performance in a audio beamforming simulation model using multiple cores on the desktop.

# 802.11ad Single Carrier Link with RF Beamforming in Simulink

This example shows how to model an IEEE 802.11ad™ single carrier link in Simulink® which includes a phased array antenna with RF beamforming. This example requires the following products:

- WLAN Toolbox™ for baseband transmitter and receiver

- Phased Array System Toolbox™ for receive antenna array

- RF Blockset™ for RF receiver

**Introduction**

This model simulates an 802.11ad single carrier (SC) [ 1 ] link with RF beamforming. Multiple packets are transmitted through free space, then RF beamformed, demodulated and the PLCP service data units (PSDU) are recovered. The PSDUs are compared with those transmitted to determine the packet error rate. The receiver performs packet detection, timing synchronization, carrier frequency offset correction and unique word based phase tracking.

The MATLAB function block allows Simulink models to use MATLAB® functions. In this example, an 802.11ad SC link modeled in Simulink uses WLAN Toolbox functions called using MATLAB function blocks. For an 802.11ad baseband simulation in MATLAB, see the example "802.11ad Packet Error Rate Single Carrier PHY Simulation with TGay Channel" (WLAN Toolbox).
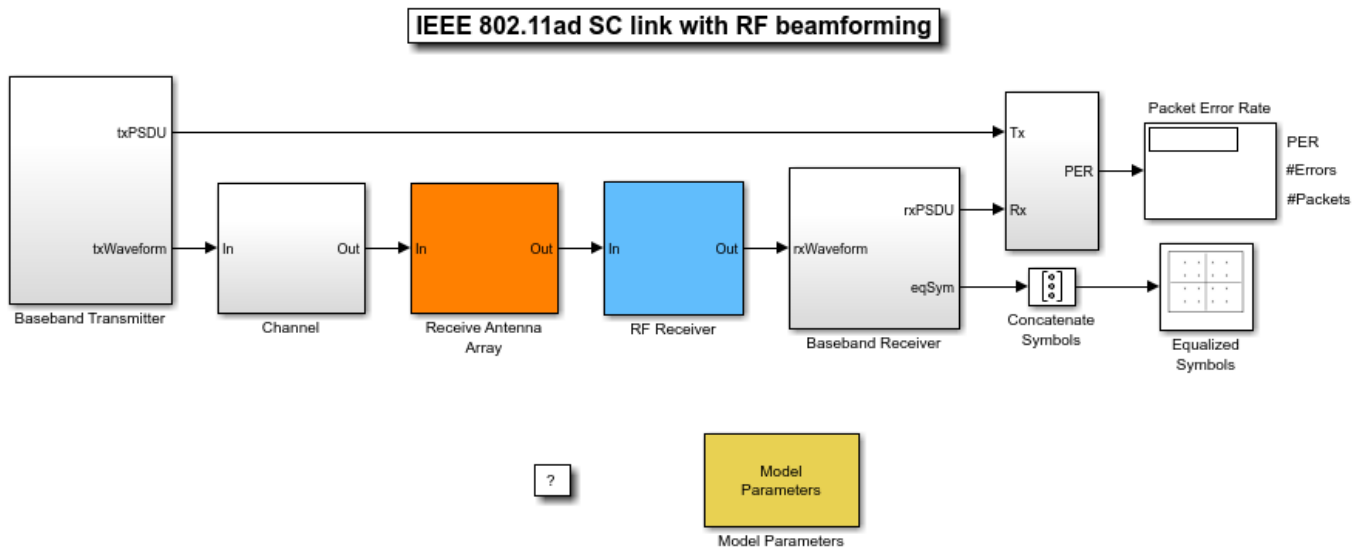


Copyright 2018-2021 The MathWorks, Inc.

**System Architecture**

The system consists of:

- A baseband transmitter which generates a random PSDU and an 802.11ad SC packet.
- A free space channel.

- A receive antenna array which supports up to 16 elements. This module allows control of the array geometry, number of elements in an array, operating frequency, and receiver direction.
- A 16 channel RF receiver module to process the RF signals. This receiver module includes low noise amplifiers, phase shifters, Wilkinson 16:1 combiner, and a down-converter. This module allows control of the beamforming direction used to calculate the corresponding phase shifts.
- A baseband receiver which recovers the transmitted PSDU by performing packet detection, time and frequency synchronization, channel estimation, PSDU demodulation, and decoding.

The system diagnostics includes the display of equalized constellation and the obtained packet error rate.

The following sections describe the transmitter and receiver in more detail.

### Baseband Transmitter

The baseband transmitter block creates a random PSDU and encodes the bits to create a single packet waveform based on the MCS and PSDU length values in the Model Parameters block. The packet generator block uses the function `wlanWaveformGenerator` (WLAN Toolbox) to encode a packet.
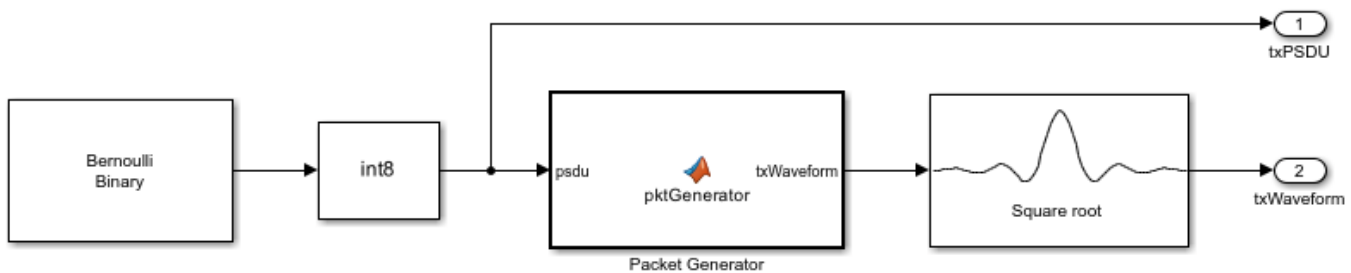


### RF Receiver

The RF receiver consists of amplifiers, phase shifters, Wilkinson 16:1 combiner and is implemented in superheterodyne fashion.

**RF RECEIVE MODULE FOR A 16 CHANNEL PHASED ARRAY with Ideal 16:1 Combiner**
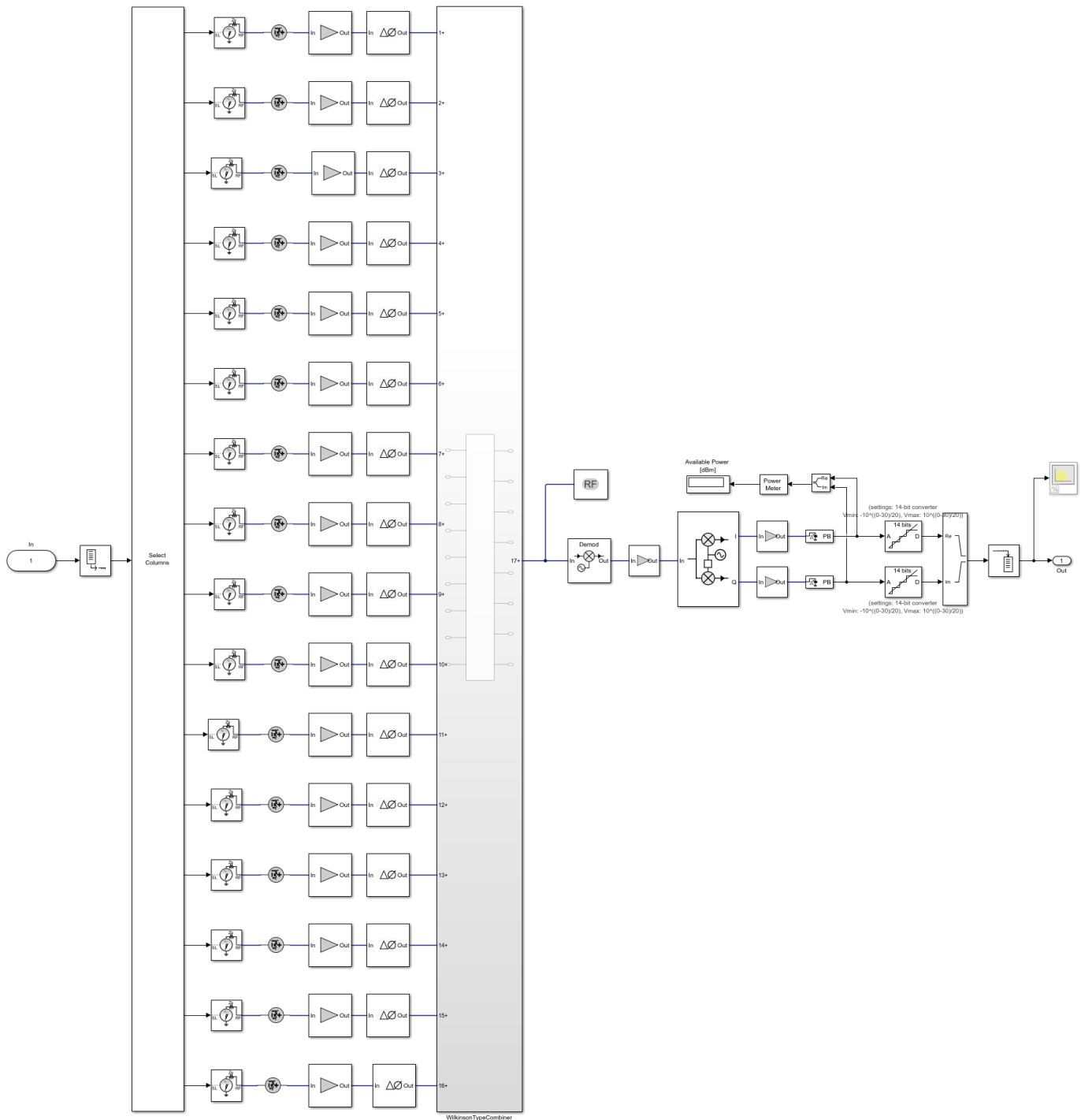


The phase shift applied to each element is calculated based on the beamforming direction. This is provided by the user and indicates the direction of the main beam. The receiver maximizes the SNR when the receiver's main beam points to the transmitter. Transmitter is omnidirectional and the

receiver direction (az,el) indicates the direction of incident signal. The scenario where the receiver direction and the beamforming direction are different is shown. In this case, there will be a reduction in the received signal power leading to high packet error rate (PER) and error vector magnitude (EVM). The results section shows these values.
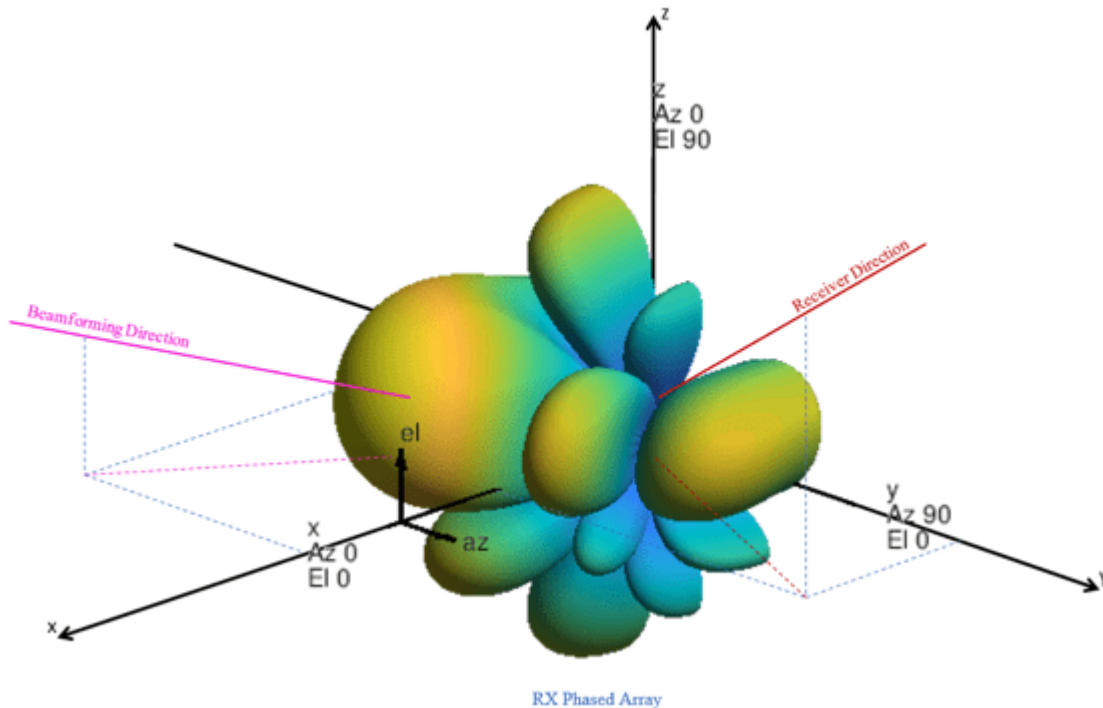


**Baseband Receiver**

The baseband receiver has two components: packet detection and packet recovery.
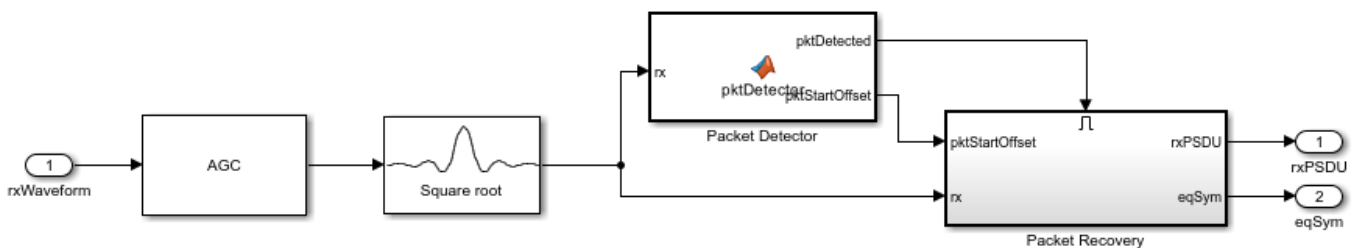


If a packet is detected, the packet recovery subsystem is enabled to process the detected packet.

The packet recovery subsystem processing consists of the following steps:

1  Frequency offset estimation and correction.
2  Symbol timing and channel frequency response estimation.
3  Noise power estimation.
4  Synchronization error checking. This determines whether the packet can be decoded or not.
5  Packet decoding.
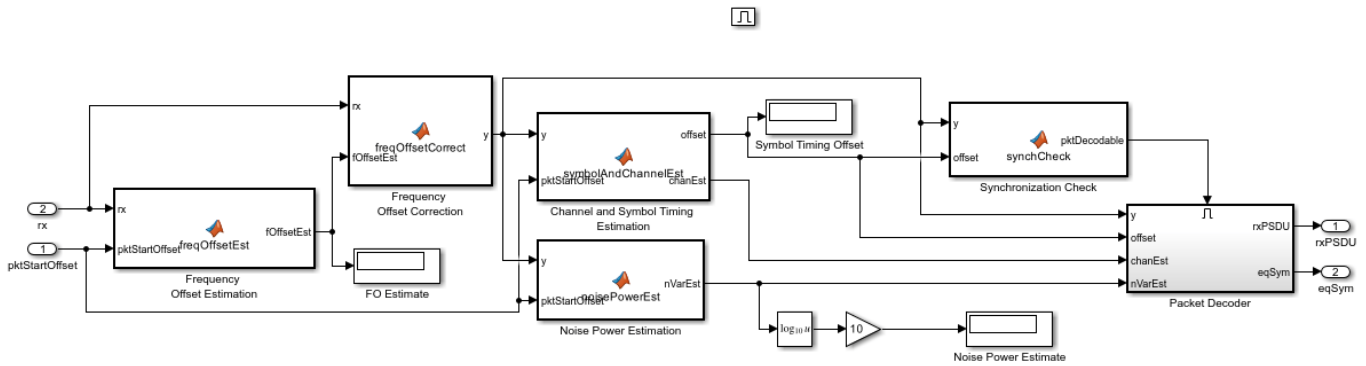
In the packet decoder subsystem, the SC data field is extracted from the synchronized received waveform. Then, the PSDU is recovered using the extracted field, channel, and noise power estimates.
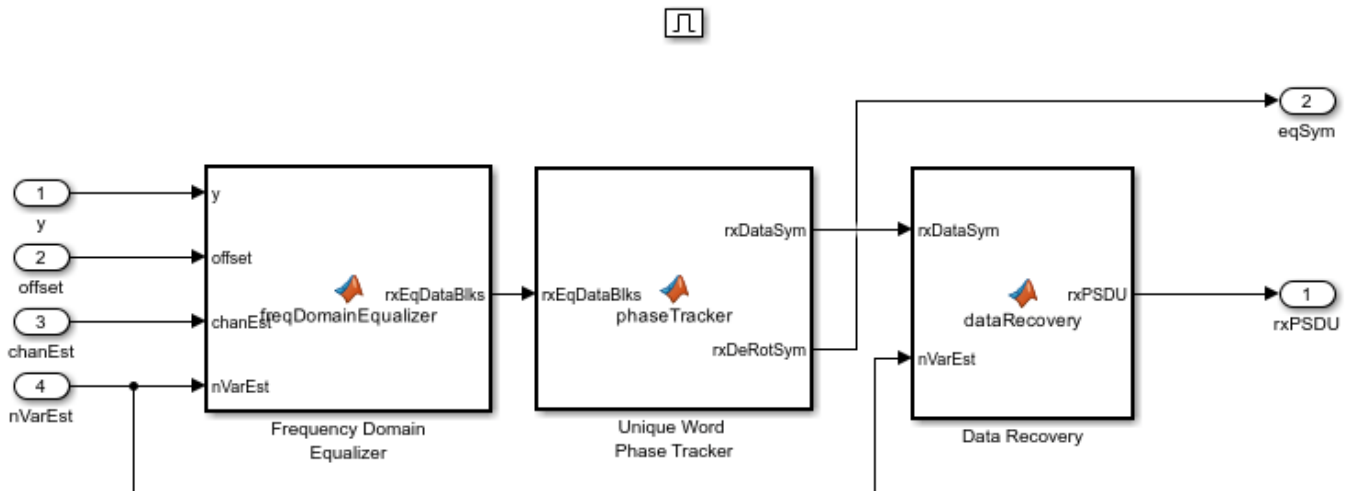


### Results

Running the simulation displays the packet error rate. The model updates the PER after processing each packet. The model also displays the equalized symbol constellation along with the EVM measurement. Note that for statistically valid results, long simulation times are required.

By default, the main beam of the receive antenna array points towards the direction: azimuth = 0 deg. and elevation = 0 deg.
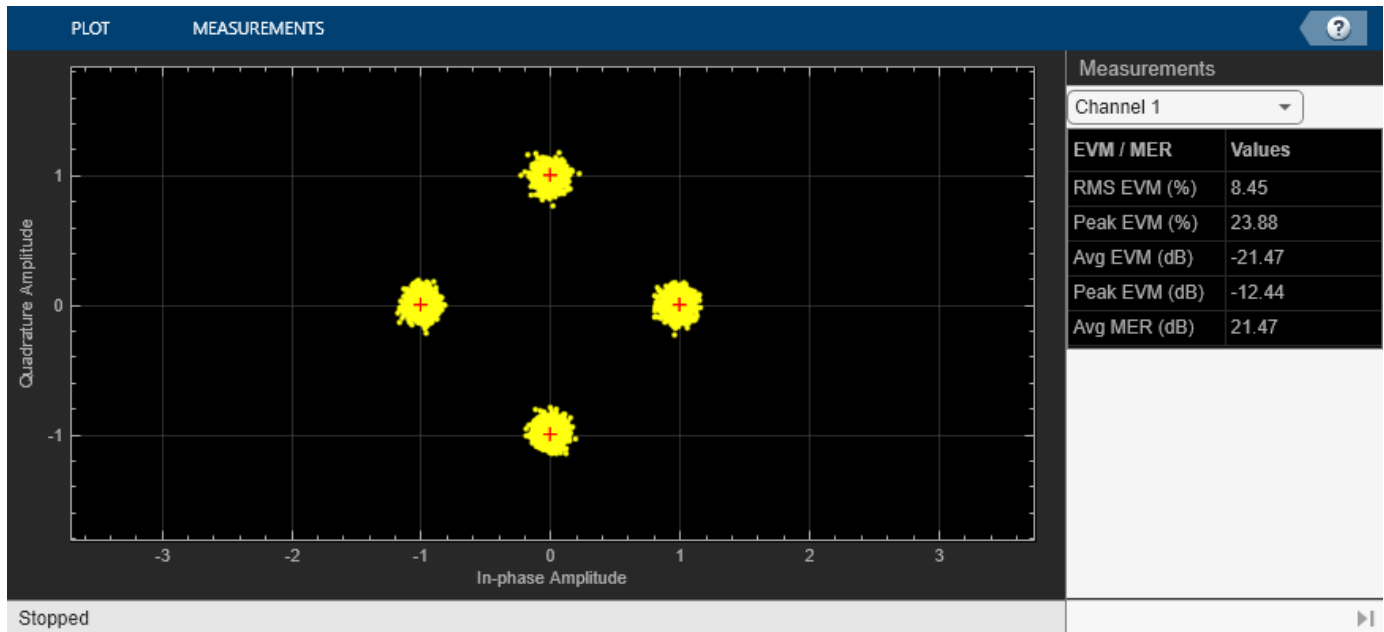
If you change the `Receiver direction` value in the receive antenna array towards a proximity null in the array radiation, the EVM increases and the packets cannot be successfully decoded.
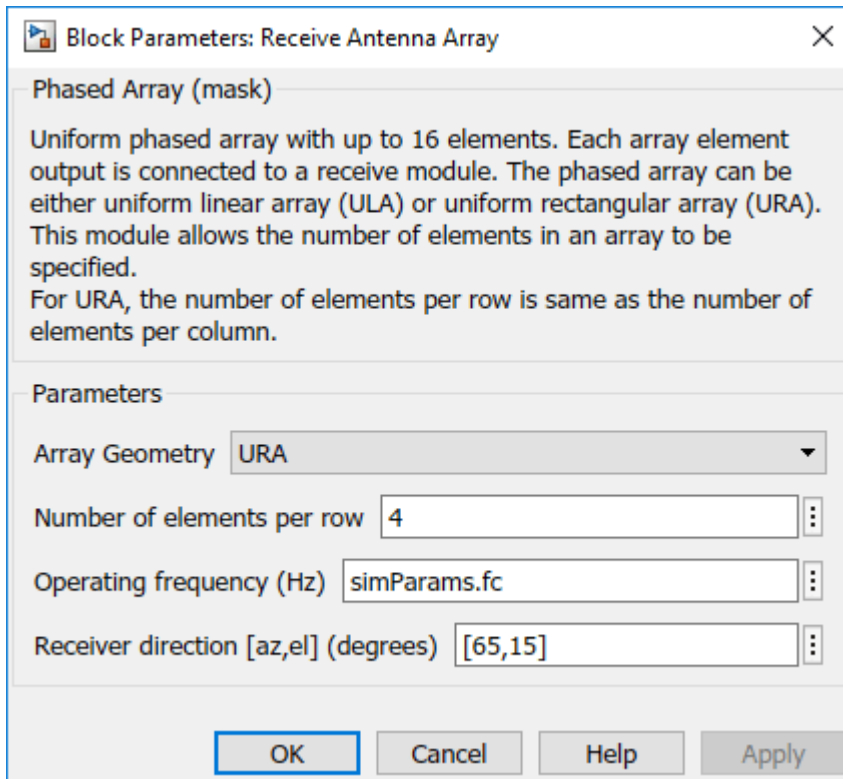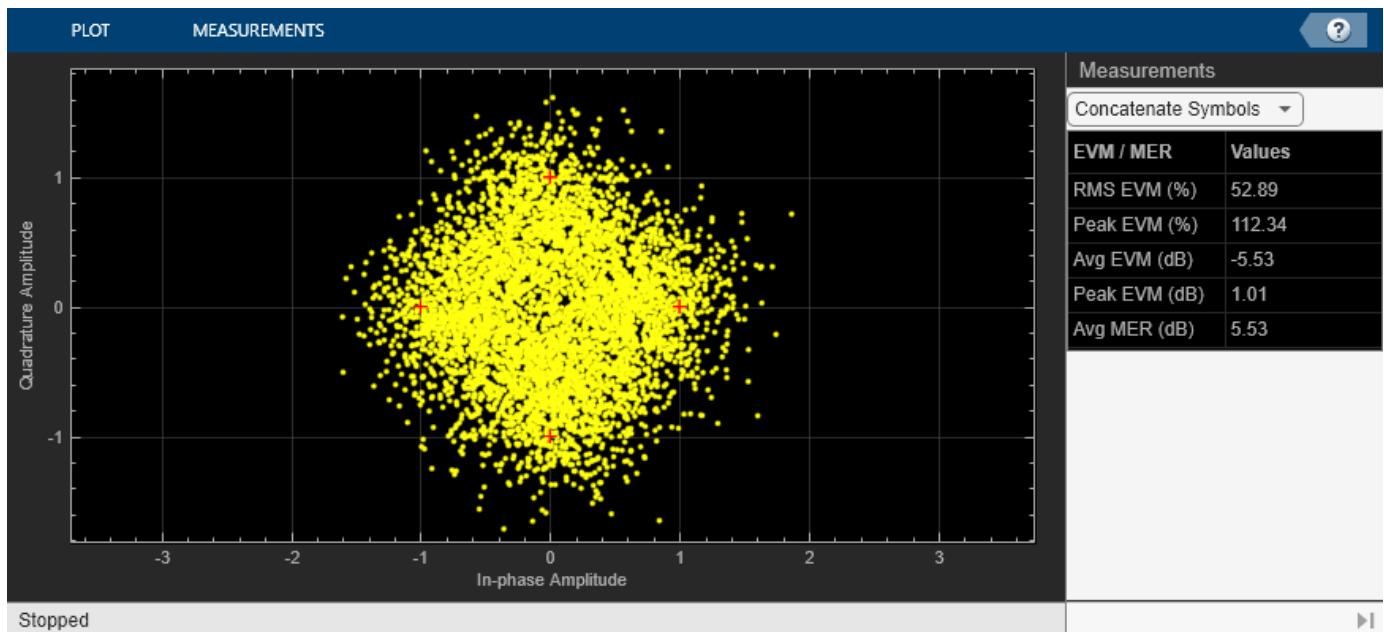
If you change the `Beamforming direction` value in the RF receiver such that the main beam points towards the transmitter, the EVM improves and packets are successfully decoded.
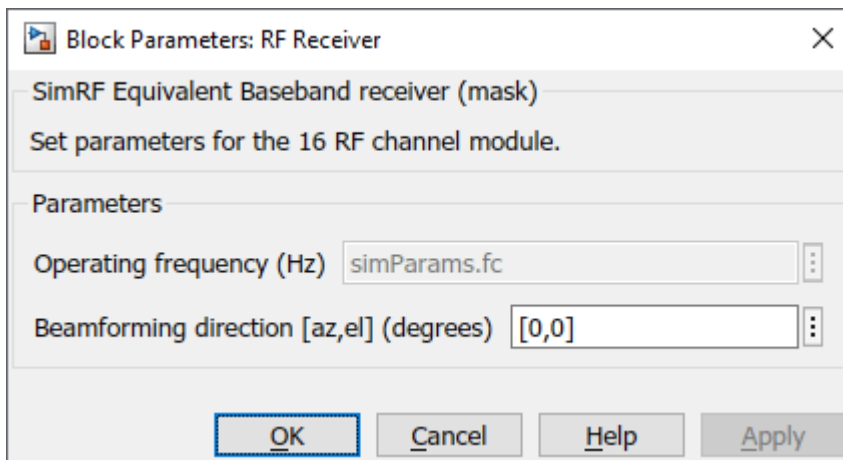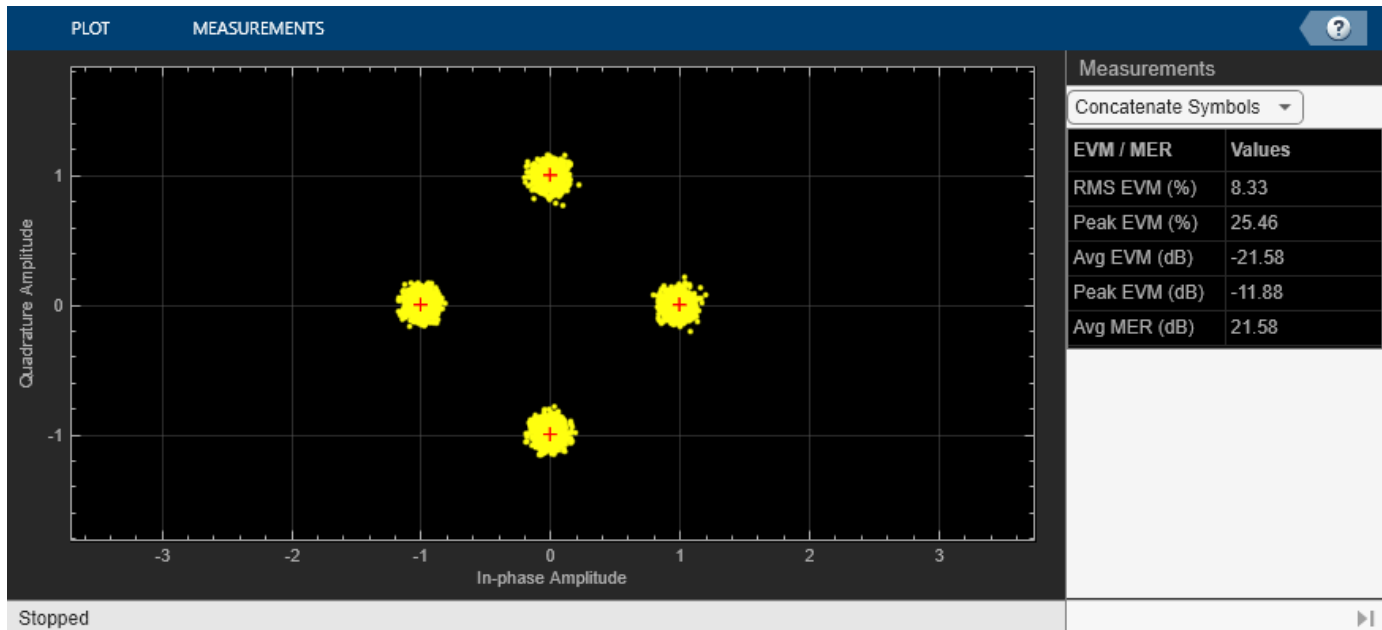
**Exploring the Example**

- Try changing the signal to noise ratio (SNR) value in the Model Parameters block. Increasing SNR leads to lower packet error rates and improved EVM of equalized symbols constellation. The SNR specified is the signal to noise ratio at the input to the ADC, if a single receive chain is used. The SNR accounts for free space path loss, thermal noise and the noise figure of RF components.

- You can change the array geometry and the number of elements in an array present in the receive antenna array block. Increasing the number of antenna elements improves the EVM. The diversity gain due to receiver antenna array can be observed in the equalized symbols constellation.

**Appendix**

This example uses the following helper functions:

- dmgCFOEstimate.m
- dmgPacketDetect.m
- dmgSingleCarrierFDE.m
- dmgSTFNoiseEstimate.m
- dmgTimingAndChannelEstimate.m
- dmgUniqueWordPhaseTracking.m

**Selected Bibliography**

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

# 802.11ad Waveform Generation with Beamforming

This example shows how to beamform an IEEE® 802.11ad™ DMG waveform with a phased array using WLAN Toolbox™ and Phased Array System Toolbox™.

**Introduction**

IEEE 802.11ad [ 1 on page 18-521 ] defines the directional multi-gigabit (DMG) transmission format operating at 60 GHz. To overcome the large path loss experienced at 60 GHz, the IEEE 802.11ad standard is designed to support directional beamforming. By using phased antenna arrays you can apply an antenna weight vector (AWV) to focus the antenna pattern in the desired direction. Each packet is transmitted on all array elements, but the AWV applies a phase shift to each element to steer the transmission. The quality of a communication link can be improved by appending optional training fields to DMG packets, and testing different AWVs at the transmitter or receiver. This process is called beam refinement.

A DMG packet consists of the following fields:

**1**  STF - The short training field, which is used for synchronization.

**2**  CE - Channel estimation field, which is used for channel estimation.

**3**  Header - The signaling field, which the receiver decodes to determine transmission parameters.

**4**  Data - The data field, which carries the user data payload.

**5**  AGC Subfields - Optional automatic gain control (AGC) subfields, used for beam refinement.

**6**  Training Subfields - Optional training subfields, used for beam refinement.

The STF and CE fields form the preamble. The preamble, header, and data fields of a DMG packet are transmitted with the same AWV. For transmitter beam refinement training, up to 64 training (TRN) subfields can be appended to the packet. Each TRN subfield is transmitted using a different AWV. This allows the performance of up to 64 different AWVs to be measured, and the AWV for the preamble, header, and data fields to be refined for subsequent transmissions. CE subfields are periodically transmitted, one for every four TRN subfields, among the TRN subfields. Each CE subfield is transmitted using the AWV as the preamble. To allow the receiver to reconfigure AGC before receiving the TRN subfields, the TRN subfields are preceded by AGC subfields. For each TRN subfield, an AGC subfield is transmitted using the same AWV applied to the individual TRN subfield. This allows a gain to be set at the receiver, suitable to measuring all TRN subfields. The diagram below shows the packet structure with four AGC and TRN subfields numbered and highlighted. Therefore, four AWVs are tested as part of beam refinement. The same AWVs are applied to AGC and TRN subfields with the same number.
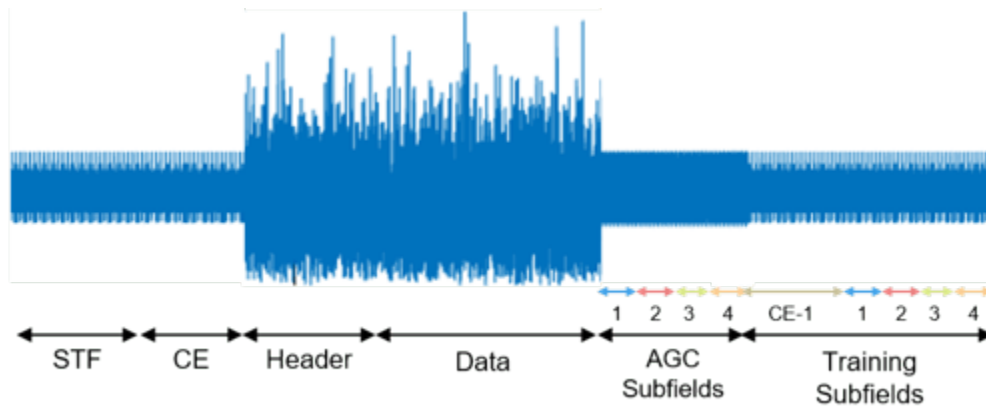
This example simulates transmitter training by applying different AWVs to each of the training subfields to steer the transmission in multiple directions. The strength of each training subfield is evaluated at a receiver by evaluating the far-field plane wave to determine which transmission AWV is optimal. This simulation does not include a channel or path loss.

This example requires "WLAN Toolbox" and "Phased Array System Toolbox".

### Waveform Specification

The waveform is configured for a DMG packet transmission with single-carrier modulation (SC) physical layer, a 100-byte physical layer service data unit (PSDU), and four transmitter training subfields. The four training subfields allow four AWVs to be tested for beam refinement. Using the function `wlanDMGConfig` (WLAN Toolbox), create a DMG configuration object. A DMG configuration object specifies transmission parameters.

```
dmg = wlanDMGConfig;
dmg.MCS = 1;                % Single-carrier modulation
dmg.TrainingLength = 4;    % Use 4 training subfields
dmg.PacketType = 'TRN-T'; % Transmitter training
dmg.PSDULength = 100;     % Bytes
```

### Beamforming Specification

The transmitter antenna pattern is configured as a 16-element uniform linear array with half-wavelength spacing. Using the objects `phased.ULA` and `phased.SteeringVector`, create the phased array and the AWVs. The location of the receiver for evaluating the transmission is specified as an offset from the boresight of the transmitter.

```
receiverAz = 6; % Degrees off the transmitter's boresight
```

A uniform linear phased array with 16 elements is created to steer the transmission.

```
N = 16;                            % Number of elements
c = physconst('LightSpeed'); % Propagation speed (m/s)
fc = 60.48e9;                     % Center frequency (Hz)
lambda = c/fc;                    % Wavelength (m)
d = lambda/2;                     % Antenna element spacing (m)
TxArray = phased.ULA('NumElements',N,'ElementSpacing',d);
```

The AWVs are created using a `phased.SteeringVector` object. Five steering angles are specified to create five AWVs, one for the preamble and data fields, and one for each of the four the training

subfields. The preamble and data fields are transmitted at boresight. The four training subfields are transmitted at angles around boresight.

```
% Create a directional steering vector object
SteeringVector = phased.SteeringVector('SensorArray',TxArray);

% The directional angle for the preamble and data is 0 degrees azimuth, no
% elevation, therefore at boresight. [Azimuth; Elevation]
preambleDataAngle = [0; 0];

% Each of the four training fields uses a different set of weights to steer
% to a slightly different direction. [Azimuth; Elevation]
trnAngle = [[-10; 0] [-5; 0] [5; 0] [10; 0]];

% Generate the weights for all of the angles
weights = SteeringVector(fc,[preambleDataAngle trnAngle]);

% Each row of the AWV is a weight to apply to a different antenna element
preambleDataAWV = conj(weights(:,1)); % AWV used for preamble, data and CE fields
trnAWV = conj(weights(:,2:end));       % AWV used for each TRN subfield
```

Using the plotArrayResponse helper function, the array response shows the direction of the receiver is most aligned with the direction of training subfield TRN-SF3.

```
plotArrayResponse(TxArray,receiverAz,fc,weights);
```
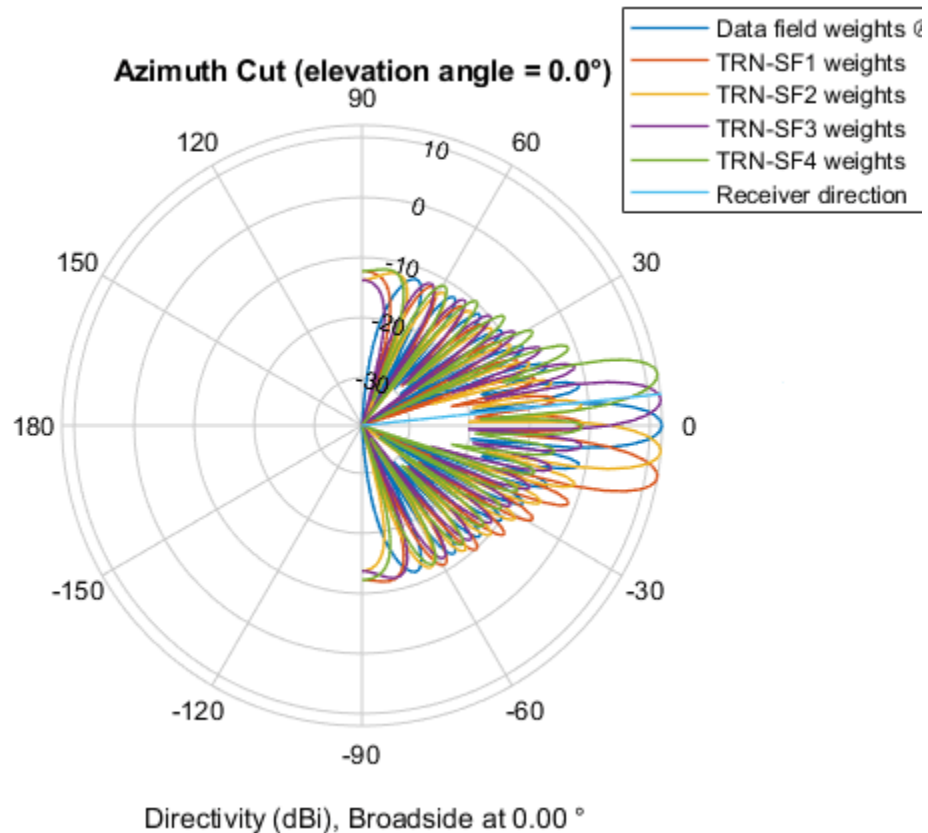
**Generate Baseband Waveform**

Use the configured DMG object and a PSDU filled with random data as inputs to the waveform generator, `wlanWaveformGenerator` (WLAN Toolbox). The waveform generator modulates PSDU bits according to a format configuration.

```
% Create a PSDU of random bits
s = rng(0); % Set random seed for repeatable results
psdu = randi([0 1],dmg.PSDULength*8,1);

% Generate packet
tx = wlanWaveformGenerator(psdu,dmg);
```

**Apply Weight Vectors to Each Field**

A `phased.Radiator` object is created to apply the AWVs to the waveform, combine the radiated signal from each element to form a plane wave, and determine the plane wave at the angle of interest, `receiverAz`. Each portion of the DMG waveform `tx` is passed through the Radiator with a specified set of AWVs, and the angle at which to evaluate the plane wave.

```
Radiator = phased.Radiator;
Radiator.Sensor = TxArray;          % Use the uniform linear array
Radiator.WeightsInputPort = true; % Provide AWV as argument
Radiator.OperatingFrequency = fc; % Frequency in Hertz
Radiator.CombineRadiatedSignals = true; % Create plane wave

% The plane wave is evaluated at a direction relative to the radiator
steerAngle = [receiverAz; 0]; % [Azimuth; Elevation]

% The beamformed waveform is evaluated as a plane wave at the receiver
planeWave = zeros(size(tx));

% Get indices for fields
ind = wlanFieldIndices(dmg);

% Get the plane wave while applying the AWV to the preamble, header, and data
idx = (1:ind.DMGData(2));
planeWave(idx) = Radiator(tx(idx),steerAngle,preambleDataAWV);

% Get the plane wave while applying the AWV to the AGC and TRN subfields
for i = 1:dmg.TrainingLength
    % AGC subfields
    agcsfIdx = ind.DMGAGCSubfields(i,1):ind.DMGAGCSubfields(i,2);
    planeWave(agcsfIdx) = Radiator(tx(agcsfIdx),steerAngle,trnAWV(:,i));
    % TRN subfields
    trnsfIdx = ind.DMGTRNSubfields(i,1):ind.DMGTRNSubfields(i,2);
    planeWave(trnsfIdx) = Radiator(tx(trnsfIdx),steerAngle,trnAWV(:,i));
end

% Get the plane wave while applying the AWV to the TRN-CE
for i = 1:dmg.TrainingLength/4
    trnceIdx = ind.DMGTRNCE(i,1):ind.DMGTRNCE(i,2);
    planeWave(trnceIdx) = Radiator(tx(trnceIdx),steerAngle,preambleDataAWV);
end
```

**Evaluate the Beamformed Waveform**

The helper function plotDMGWaveform plots the magnitude of the beamformed plane wave. When evaluating the magnitude of the beamformed plane wave we can see that the fields beamformed in the direction of the receiver are stronger than other fields.

plotDMGWaveform(planeWave,dmg,'Beamformed Plane Wave with Fields Highlighted');
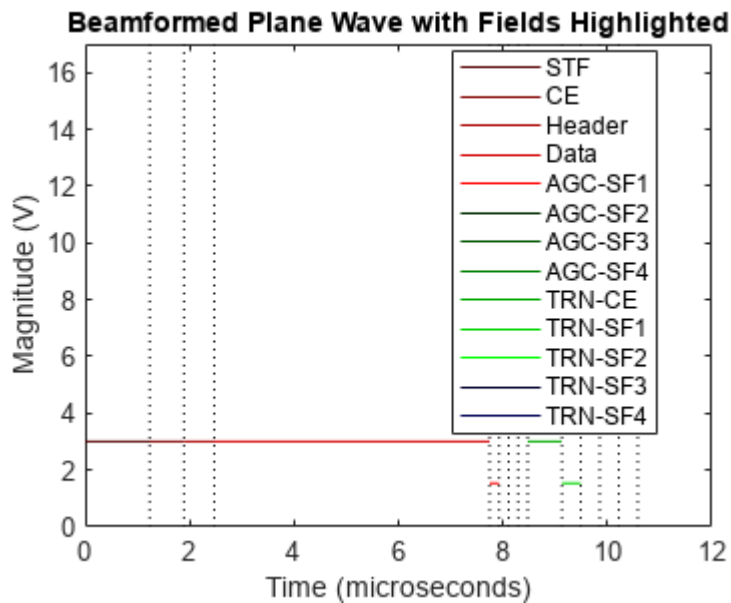


rng(s); % Restore random state

**Conclusion**

This example showed how to generate an IEEE 802.11ad DMG waveform and apply AWVs to different portions of the waveform. The example uses WLAN Toolbox to generate a standard-compliant waveform, and Phased Array System Toolbox to apply the AWVs and evaluate the magnitude of the resultant plane wave in the direction of a receiver.

**Selected Bibliography**

1   IEEE Std 802.11™-2020 IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

# Radar and Communications Waveform Classification Using Deep Learning

This example shows how to classify radar and communications waveforms using the Wigner-Ville distribution (WVD) and a deep convolutional neural network (CNN).

Modulation classification is an important function for an intelligent receiver. Modulation classification has numerous applications, such as cognitive radar and software-defined radio. Typically, to identify these waveforms and classify them by modulation type, it is necessary to define meaningful features and input them into a classifier. While effective, this procedure can require extensive effort and domain knowledge to yield an accurate classification. This example explores a framework to automatically extract time-frequency features from signals and perform signal classification using a deep learning network.

The first part of this example simulates a radar classification system that synthesizes three pulsed radar waveforms and classifies them. The radar waveforms are:

- Rectangular
- Linear frequency modulation (LFM)
- Barker Code

A radar classification system does not exist in isolation. Rather, it resides in an increasingly occupied frequency spectrum, competing with other transmitted sources such as communications systems, radio, and navigation systems. The second part of this example extends the network to include additional communication modulation types. In addition to the first set of radar waveforms, the extended network synthesizes and identifies these communication waveforms:

- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast frequency modulation (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

This example primarily focuses on radar waveforms, with the classification being extended to include a small set of amplitude and frequency modulation communications signals. See "Modulation Classification with Deep Learning" (Communications Toolbox) for a full workflow of modulation classification with a wide array of communication signals.

**Generate Radar Waveforms**

Generate 3000 signals with a sample rate of 100 MHz for each modulation type. Use `phased.RectangularWaveform` for rectangular pulses, `phased.LinearFMWaveform` for LFM, and `phased.PhaseCodedWaveform` for phase coded pulses with Barker code.

Each signal has unique parameters and is augmented with various impairments to make it more realistic. For each waveform, the pulse width and repetition frequency will be randomly generated. For LFM waveforms, the sweep bandwidth and direction are randomly generated. For Barker waveforms, the chip width and number are generated randomly. All signals are impaired with white Gaussian noise using the `awgn` function with a random signal-to-noise ratio in the range of [–6, 30] dB. A frequency offset with a random carrier frequency in the range of [Fs/6, Fs/5] is applied to

each signal using the `comm.PhaseFrequencyOffset` object. Lastly, each signal is passed through a multipath Rician fading channel, `comm.RicianChannel`.

The helper function `helperGenerateRadarWaveforms` creates and augments each modulation type.

```
rng default
[wav, modType] = helperGenerateRadarWaveforms();
```

Plot the Fourier transform for a few of the LFM waveforms to show the variances in the generated set.

```
idLFM = find(modType == "LFM",3);
nfft = 2^nextpow2(length(wav{1}));
f = (0:(nfft/2-1))/nfft*100e6;

figure
subplot(1,3,1)
Z = fft(wav{idLFM(1)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,2)
Z = fft(wav{idLFM(2)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,3)
Z = fft(wav{idLFM(3)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
```
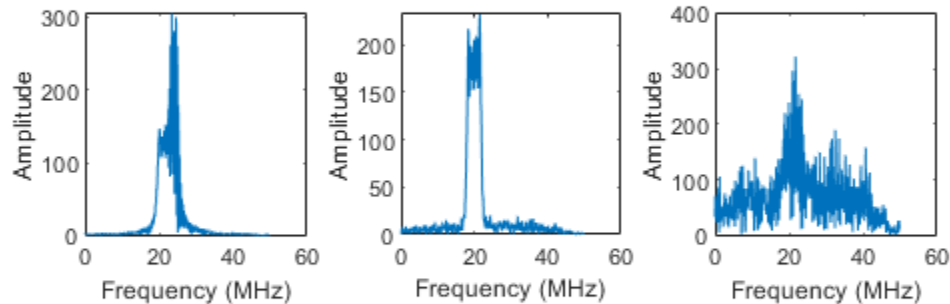
### Feature Extraction Using Wigner-Ville Distribution

To improve the classification performance of machine learning algorithms, a common approach is to input extracted features in place of the original signal data. The features provide a representation of the input data that makes it easier for a classification algorithm to discriminate across the classes. The Wigner-Ville distribution represents a time-frequency view of the original data that is useful for time varying signals. The high resolution and locality in both time and frequency provide good features for the identification of similar modulation types. Use the `wvd` function to compute the smoothed pseudo WVD for each of the modulation types.

```
figure
subplot(1,3,1)
wvd(wav{find(modType == "Rect",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Rect')
subplot(1,3,2)
wvd(wav{find(modType == "LFM",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('LFM')
subplot(1,3,3)
wvd(wav{find(modType == "Barker",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Barker')
```
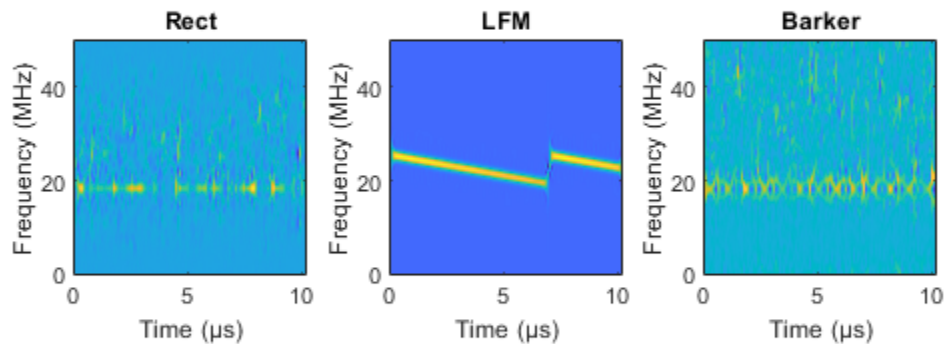
To store the smoothed-pseudo Wigner-Ville distribution of the signals, first create the directory `TFDDatabase` inside your temporary directory `tempdir`. Then create subdirectories in `TFDDatabase` for each modulation type. For each signal, compute the smoothed-pseudo Wigner-Ville distribution, and downsample the result to a 227-by-227 matrix. Save the matrix as a `.png` image file in the subdirectory corresponding to the modulation type of the signal. The helper function `helperGenerateTFDfiles` performs all these steps. This process will take several minutes due to the large database size and the complexity of the `wvd` algorithm. You can replace `tempdir` with another directory where you have write permission.

```
parentDir = tempdir;
dataDir = 'TFDDatabase';
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,100e6)
```

Create an image datastore object for the created folder to manage the image files used for training the deep learning network. This step avoids having to load all images into memory. Specify the label source to be folder names. This assigns the modulation type of each signal according to the folder name.

```
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker'});
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

The network is trained with 80% of the data and tested on with 10%. The remaining 10% is used for validation. Use the `splitEachLabel` function to divide the `imageDatastore` into training, validation, and testing sets.

```
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

**Set Up Deep Learning Network**

Before the deep learning network can be trained, define the network architecture. This example utilizes transfer learning SqueezeNet, a deep CNN created for image classification. Transfer learning is the process of retraining an existing neural network to classify new targets. This network accepts image input of size 227-by-227-by-3. Prior to input to the network, the custom read function `readTFDForSqueezeNet` transforms the two-dimensional time-frequency distribution to an RGB image of the correct size. SqueezeNet performs classification of 1000 categories in its default configuration.

Load SqueezeNet.

```
net = squeezenet;
```

Extract the layer graph from the network. Confirm that SqueezeNet is configured for images of size 227-by-227-by-3.

```
lgraphSqz = layerGraph(net);
lgraphSqz.Layers(1)

ans =
  ImageInputLayer with properties:

                    Name: 'data'
               InputSize: [227 227 3]

   Hyperparameters
         DataAugmentation: 'none'
            Normalization: 'zerocenter'
   NormalizationDimension: 'auto'
                     Mean: [1×1×3 single]
```

To tune SqueezeNet for this example, three of the last six layers need to be modified to classify the three radar modulation types of interest. Inspect the last six network layers.

```
lgraphSqz.Layers(end-5:end)

ans =
  6×1 Layer array with layers:

     1   'drop9'                            Dropout                    50% dropout
     2   'conv10'                           Convolution                1000 1×1×512 convolutio
     3   'relu_conv10'                      ReLU                       ReLU
     4   'pool10'                           2-D Global Average Pooling 2-D global average pool
     5   'prob'                             Softmax                    softmax
     6   'ClassificationLayer_predictions'  Classification Output      crossentropyex with 'te
```

Replace the drop9 layer, the last dropout layer in the network, with a dropout layer of probability 0.6.

```
tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);
```

The last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, conv10. Replace the layer with a new convolutional layer with the number of filters equal to the number of modulation types. Also increase the learning rate factors of the new layer.

```
numClasses = 3;
tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);
```

Replace the classification layer with a new one without class labels.

```
tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Inspect the last six layers of the network. Confirm the dropout, convolutional, and output layers have been changed.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
  6×1 Layer array with layers:

     1   'new_dropout'       Dropout                   60% dropout
     2   'new_conv'          Convolution               3 1×1 convolutions with stride [1  1] a
     3   'relu_conv10'       ReLU                      ReLU
     4   'pool10'            2-D Global Average Pooling  2-D global average pooling
     5   'prob'              Softmax                   softmax
     6   'new_classoutput'   Classification Output     crossentropyex
```

Choose options for the training process that ensures good network performance. Refer to the `trainingOptions` documentation for a description of each option.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',128, ...
    'MaxEpochs',5, ...
    'InitialLearnRate',1e-3, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ValidationData',imdsValidation);
```

**Train the Network**

Use the `trainNetwork` command to train the created CNN. Because the data set is large in size, the process can take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB® automatically uses the GPU for training. Otherwise, it uses the CPU. The training accuracy plots in the figure show the progress of the network's learning across all iterations. On the three radar modulation types, the network classifies almost 100% of the training signals correctly.

```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```
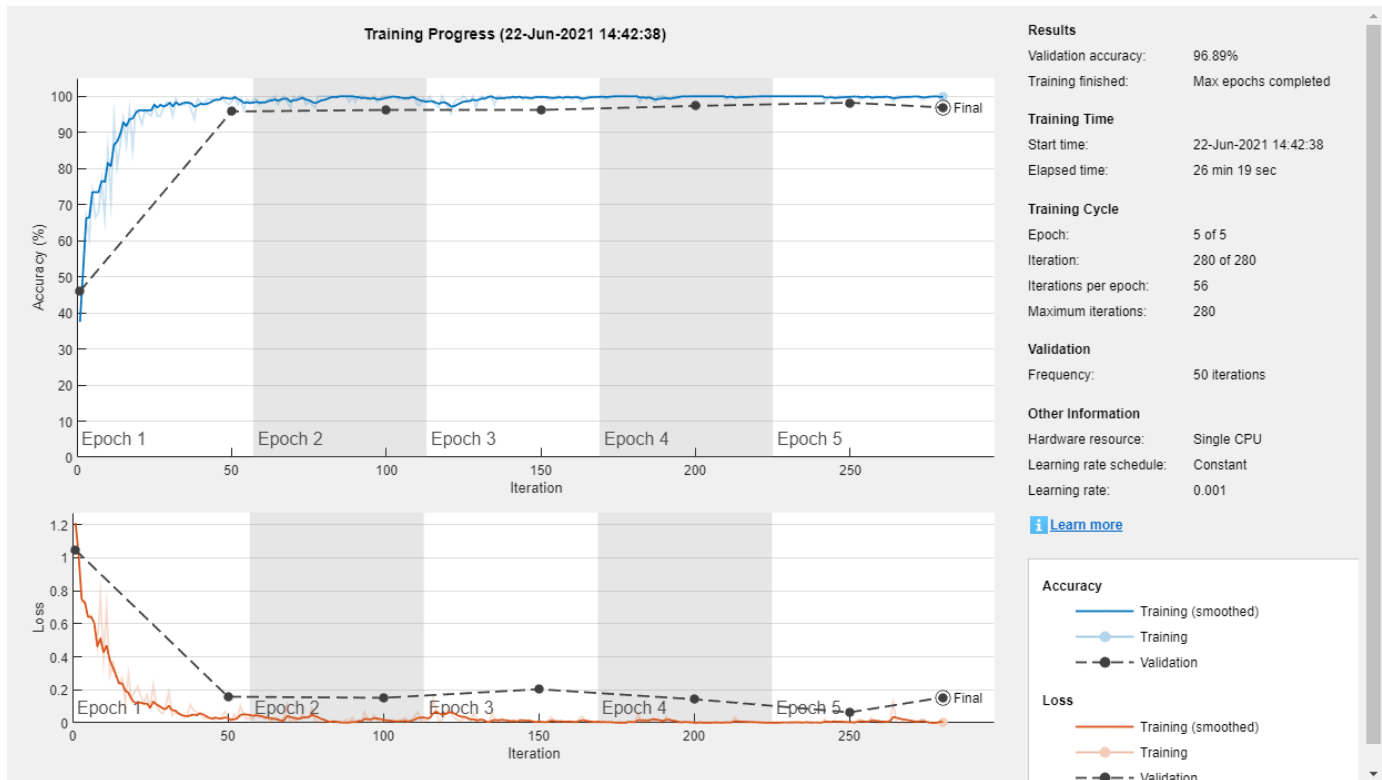
### Evaluate Performance on Radar Waveforms

Use the trained network to classify the testing data using the `classify` command. A confusion matrix is one method to visualize classification performance. Use the `confusionchart` command to calculate and visualize the classification accuracy. For the three modulation types input to the network, almost all of the phase coded, LFM, and rectangular waveforms are correctly identified by the network.

```
predicted = classify(trainedNet,imdsTest);
figure
confusionchart(imdsTest.Labels,predicted,'Normalization','column-normalized')
```
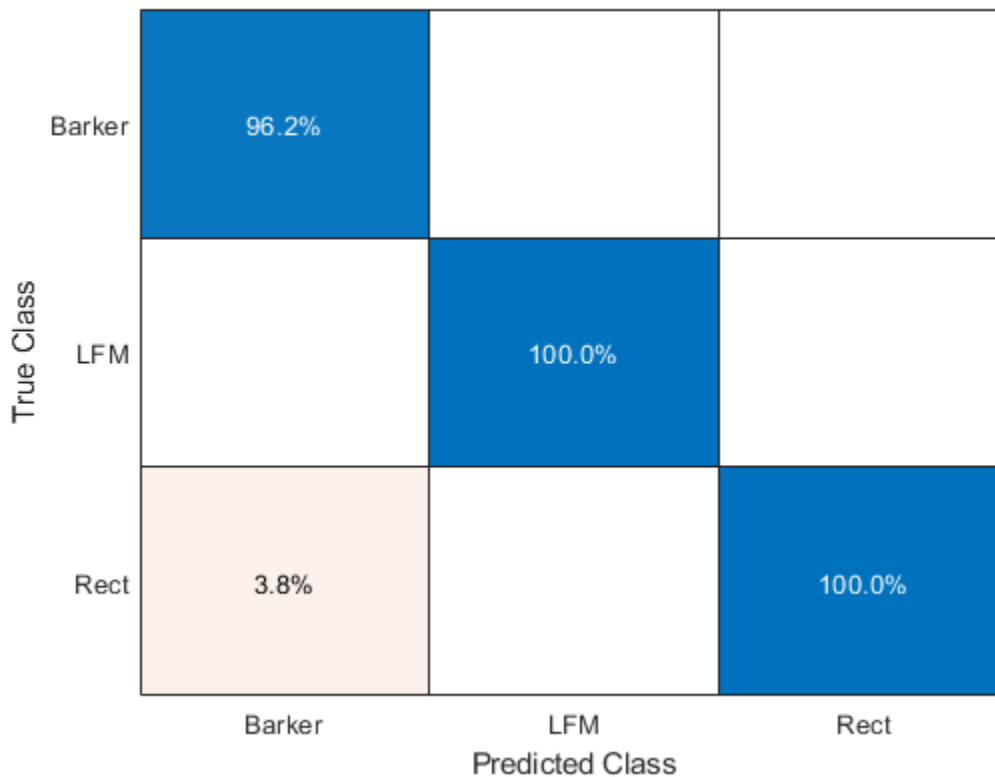
### Generate Communications Waveforms and Extract Features

The frequency spectrum of a radar classification system must compete with other transmitted sources. See how the created network extends to incorporate other simulated modulation types. Another MathWorks® example, "Modulation Classification with Deep Learning" (Communications Toolbox), performs modulation classification of several different modulation types using Communications Toolbox™. The helper function `helperGenerateCommsWaveforms` generates and augments a subset of the modulation types used in that example. Since the WVD loses phase information, a subset of only the amplitude and frequency modulation types is used.

See the example link for an in-depth description of the workflow necessary for digital and analog modulation classification and the techniques used to create these waveforms. For each modulation type, use `wvd` to extract time-frequency features and visualize.

```
[wav, modType] = helperGenerateCommsWaveforms();
```

```
figure
subplot(2,3,1)
wvd(wav{find(modType == "GFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('GFSK')
subplot(2,3,2)
wvd(wav{find(modType == "CPFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('CPFSK')
subplot(2,3,3)
wvd(wav{find(modType == "B-FM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('B-FM')
subplot(2,3,4)
```

```
wvd(wav{find(modType == "SSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('SSB-AM')
subplot(2,3,5)
wvd(wav{find(modType == "DSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('DSB-AM')
```
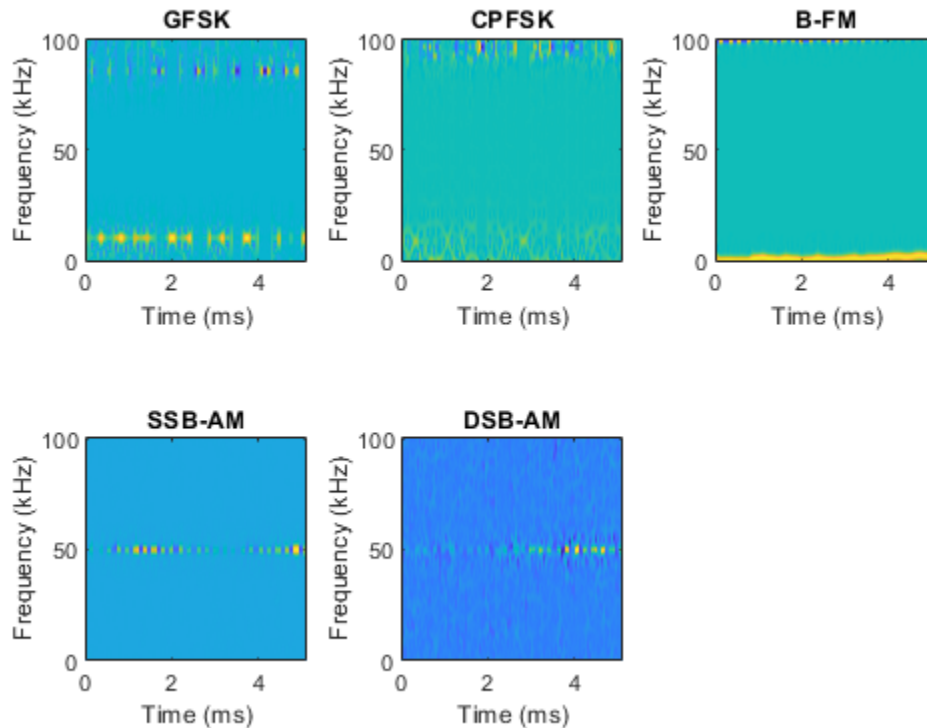


Use the helper function `helperGenerateTFDfiles` again to compute the smoothed pseudo WVD for each input signal. Create an image datastore object to manage the image files of all modulation types.

```
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,200e3)
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker','GFSK','CPFSK','B-FM','SSB-AM','DSB-
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

Again, divide the data into a training set, a validation set, and a testing set using the `splitEachLabel` function.

```
rng default
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

**Adjust Deep Learning Network Architecture**

Previously, the network architecture was set up to classify three modulation types. This must be updated to allow classification of all eight modulation types of both radar and communication signals. This is a similar process as before, with the exception that the `fullyConnectedLayer` now requires an output size of eight.

```
numClasses = 8;
net = squeezenet;
lgraphSqz = layerGraph(net);

tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);

tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);

tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Create a new set of training options.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',150, ...
    'MaxEpochs',10, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ValidationData',imdsValidation);
```

Use the `trainNetwork` command to train the created CNN. For all modulation types, the training converges with an accuracy of about 95% correct classification.

```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```
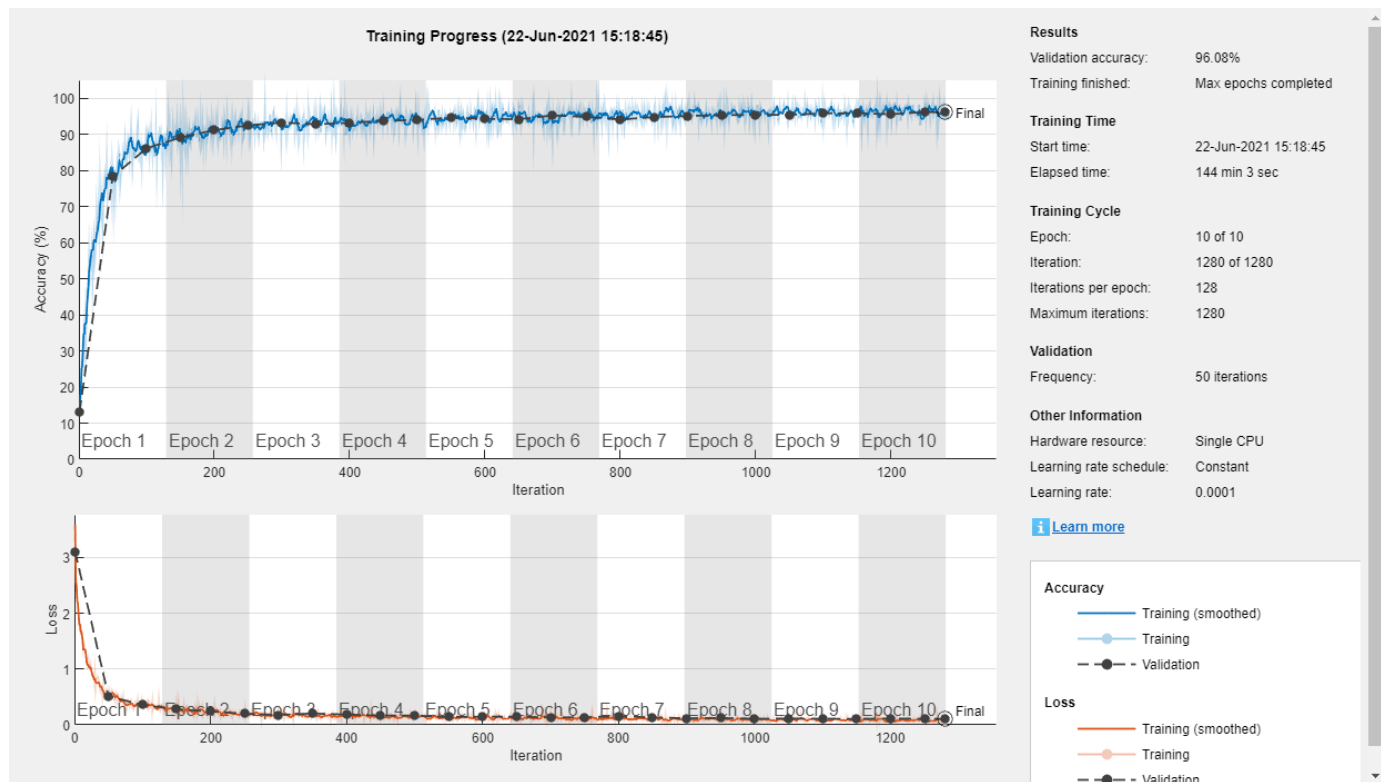
### Evaluate Performance on All Signals

Use the `classify` command to classify the signals held aside for testing. Again, visualize the performance using `confusionchart`.

```
predicted = classify(trainedNet,imdsTest);
figure;
confusionchart(imdsTest.Labels,predicted,'Normalization','column-normalized')
```

For the eight modulation types input to the network, about 98% of B-FM, CPFSK, GFSK, Barker, and LFM modulation types were correctly classified. On average, about 85% of AM signals were correctly identified. From the confusion matrix, a high percentage of SSB-AM signals were misclassified as DSB-AM and DSB-AM signals as SSB-AM.

Investigate a few of these misclassifications to gain insight into the network's learning process. Use the `readimage` function on the image datastore to extract from the test dataset a single image from each class. The displayed WVD visually looks very similar. Since DSB-AM and SSB-AM signals have a very similar signature, this explains in part the network's difficulty in correctly classifying these two types. Further signal processing could make the differences between these two modulation types clearer to the network and result in improved classification.

```
DSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'DSB-AM'),1));
DSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'SSB-AM'),1));
SSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'DSB-AM'),1));
SSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'SSB-AM'),1));

figure
subplot(2,2,1)
imagesc(DSB_DSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: DSB-AM'})
subplot(2,2,2)
imagesc(DSB_SSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: SSB-AM'})
subplot(2,2,3)
imagesc(SSB_DSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: DSB-AM'})
```

```
subplot(2,2,4)
imagesc(SSB_SSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: SSB-AM'})
```
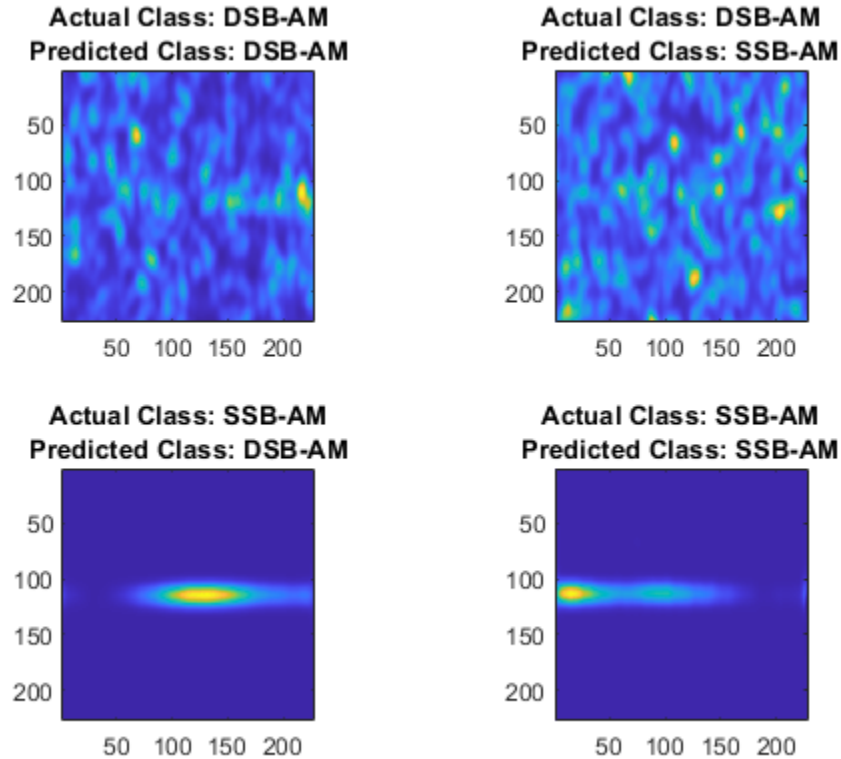


## Summary

This example showed how radar and communications modulation types can be classified by using time-frequency techniques and a deep learning network. Further efforts for additional improvement could be investigated by utilizing time-frequency analysis available in Wavelet Toolbox™ and additional Fourier analysis available in Signal Processing Toolbox™.

## References

[1] Brynolfsson, Johan, and Maria Sandsten. "Classification of one-dimensional non-stationary signals using the Wigner-Ville distribution in convolutional neural networks." *25th European Signal Processing Conference (EUSIPCO)*. IEEE, 2017.

[2] Liu, Xiaoyu, Diyu Yang, and Aly El Gamal. "Deep neural network architectures for modulation classification." *51st Asilomar Conference on Signals, Systems and Computers*. 2017.

[3] `Wang, Chao, Jian Wang, and Xudong Zhang. "Automatic radar waveform recognition based on time-frequency analysis and convolutional neural network." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017.

# Hybrid MIMO Beamforming with QSHB and HBPS Algorithms

This example presents a Simulink® model of a multiple input multiple output (MIMO) wireless communication system. The wireless system uses hybrid beamforming technique to improve system throughput.

**Introduction**

5G and other modern wireless communication systems extensively use MIMO beamforming technology for signal to noise ratio (SNR) enhancement and spatial multiplexing to improve the data throughput in scatterer rich environments. In a scatterer-rich environment, there may not exist line-of-sight (LOS) paths between the transmit and receive antennas. To gain the high throughput, MIMO beamforming implements precoding on the transmitter side and combining on the receiver side to increase SNR and separate spatial channels. A full digital beamforming structure requires each antenna to have a dedicated RF-to-baseband chain, which makes the overall hardware expensive and power consumption high. As a solution, hybrid MIMO beamforming is proposed [1], in which fewer RF-to-baseband chains are employed and partial of precoding and combining are implemented in the RF portion. With deliberate selection of the weights for precoding and combining, hybrid beamforming can achieve comparable performance as that of full beamforming.

In this example, we introduce a Simulink model with hybrid MIMO beamforming. This model shows two hybrid beamforming algorithms: Quantized Sparse Hybrid Beamforming (QSHB) [2] and Hybrid Beamforming with Peak Search (HBPS).

The following figure shows the structure of a hybrid beamforming system.
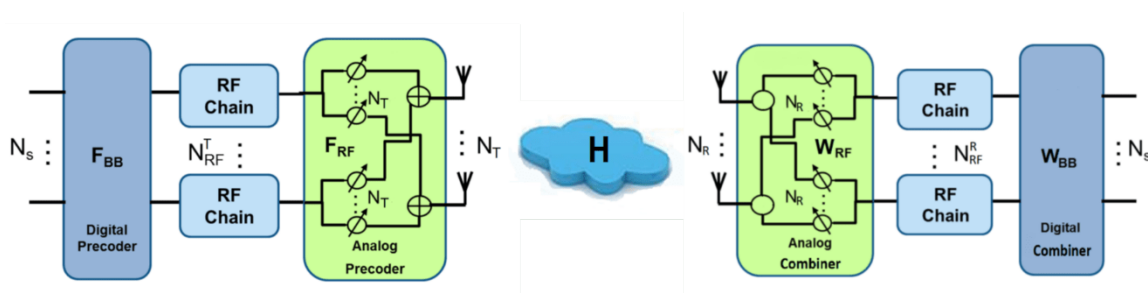


In the figure, $N_s$ is the number of signal streams; $N_T$ is the number of transmit antennas; $N_{RF}^T$ is the number of transmit RF chains; $N_R$ is the number of receive antennas; and $N_{RF}^R$ is the number of receive RF chains. In this example, two signal streams, 64 transmit antennas, 4 transmit RF chains, 16 receive antennas, and 4 receive RF chains.

The scattering channel is denoted by $H$. The hybrid beamforming weights are represented by the analog precoder $F_{RF}$, digital precoder $F_{BB}$, analog combiner $W_{RF}$, and digital combiner $W_{BB}$. For a more detailed introduction to hybrid beamforming, please refer to the MATLAB® "Introduction to Hybrid Beamforming" on page 18-371 example.

**Exploring the Model**

The Simulink model consists of four main components: MIMO Transmitter, MIMO Channel, MIMO Receiver, and Weights Calculation.
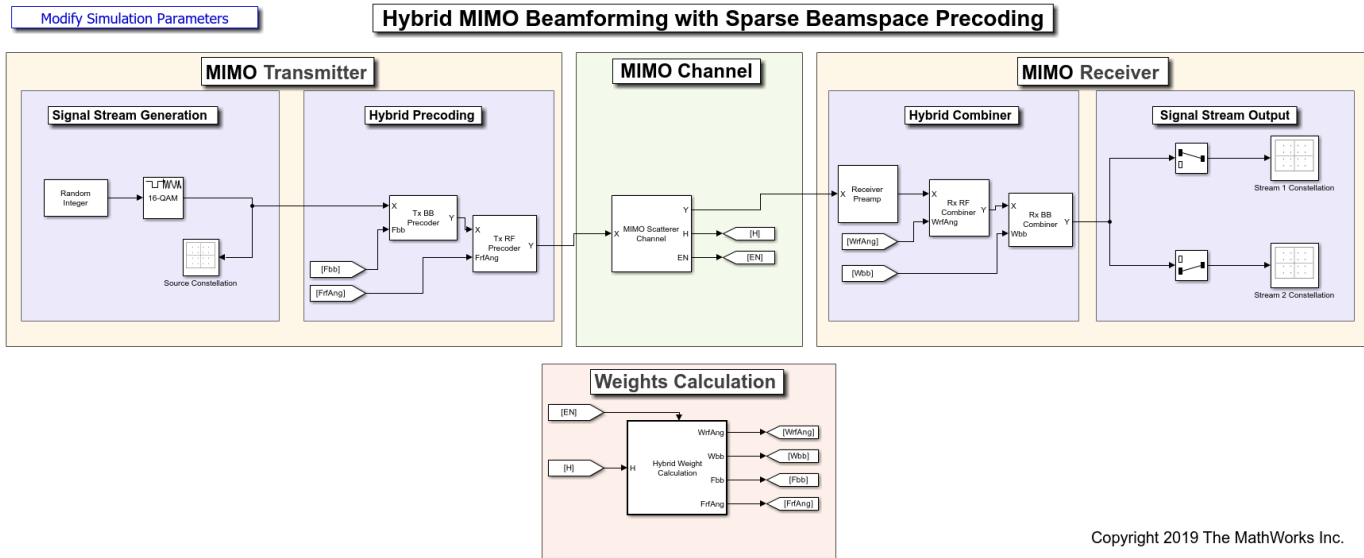
The MIMO transmitter generates the signal stream and then applies the precoding. The modulated signal is propagated through a scattering channel defined in the MIMO channel and then decoded and demodulated at the receiver side.

**MIMO Scattering Channel**

The MIMO scattering channel is represented by a channel matrix. In addition, this example uses an enabled subsystem to periodically change this matrix to simulate the fact that a MIMO channel may vary over time.
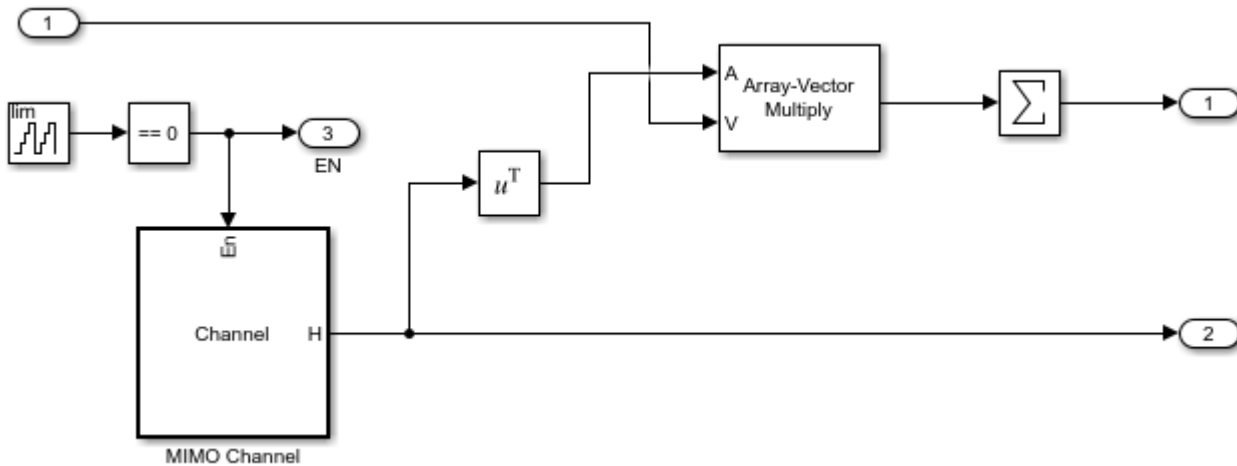


**Hybrid Beamforming Weights Computation**

In a hybrid beamforming system, both the precoding and the corresponding combining process are done partly at baseband and partly in the RF band. In general, the beamforming achieved in the RF band only involves phase shifts. Therefore, a critical part in such a system is to determine how to distribute the weights between the baseband and the RF band based on the channel. This is done in the Weight Calculation block where the precoding weights, `Fbb` and `FrfAng`, and combining weights,

`Wbb` and `WrfAng`, are computed based on the channel matrix, `H`. In this example, we assume the channel matrix is known and provide both QSHB and HBPS algorithms.



**Quantized Sparse Hybrid Beamforming (QSHB)**

Literature [2, 3] shows that given the channel matrix, H, of a MIMO scattering channel, the hybrid beamforming weights can be computed via an iterative algorithm [2]. Using an orthogonal matching pursuit algorithm, the resulting analog precoding/combining weights are just steering vectors corresponding to the dominant modes of the channel matrix. For the detailed description of the algorithm, please refer to the "Introduction to Hybrid Beamforming" on page 18-371 example.

**Quantized Sparse Hybrid Beamforming with Peak Search (HBPS)**

HBPS is a simplified version of QSHB. Instead of searching for the dominant mode of channel matrix iteratively, HBPS projects all the digital weights into a grid of directions and identifies the $N_{RF}^{T}$ and $N_{RF}^{R}$ peaks to form the corresponding analog beamforming weights. This works well especially for large arrays, like arrays used in massive MIMO systems, since for large arrays, the directions are more likely to be orthogonal.

Because the channel matrix can change over time, the weights computation also needs to be performed periodically to accommodate the channel variation.

**Results and Displays**

## QSHB

Following figures shows the recovered 16 QAM symbol streams at the receiver using QSHB algorithms. The resulting constellation shows that compared to the source constellation, the recovered symbols properly located in both streams. This means that using the hybrid beamforming technique, we can improve the system capacity by sending the two streams simultaneously. In addition, the constellation diagram shows that the variance of the first recovered stream is better than the second recovered stream as the points are less dispersed in the constellation of the first stream. This is because the first stream uses the most dominant mode of the MIMO channel so it has the best SNR.
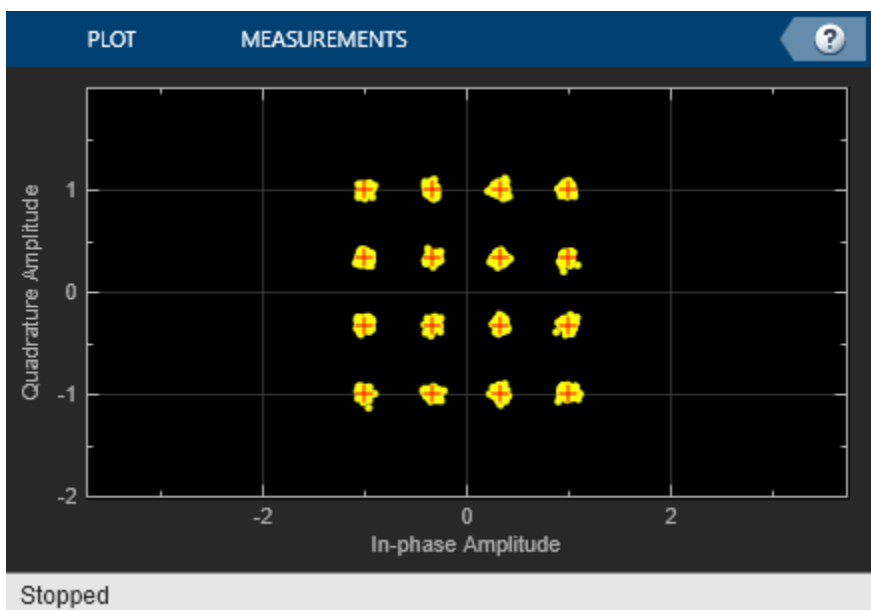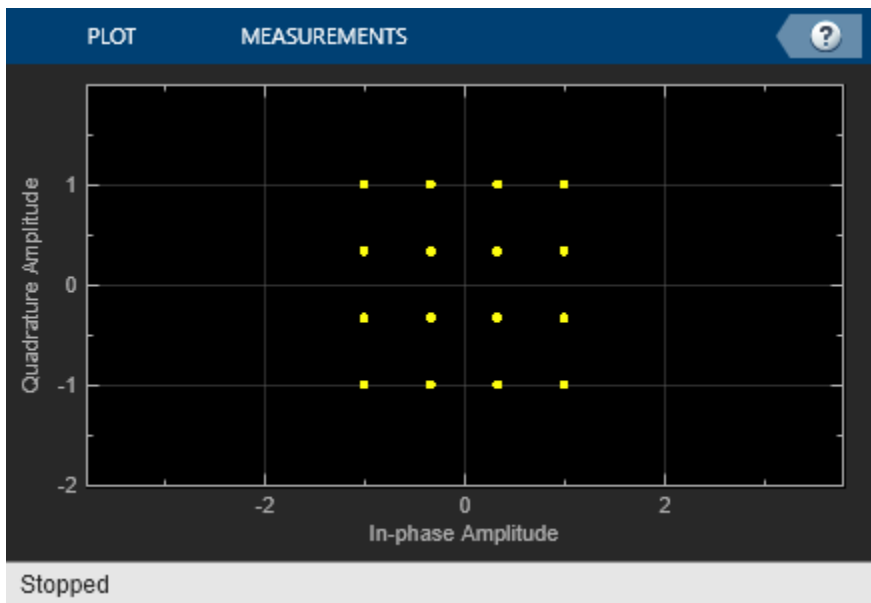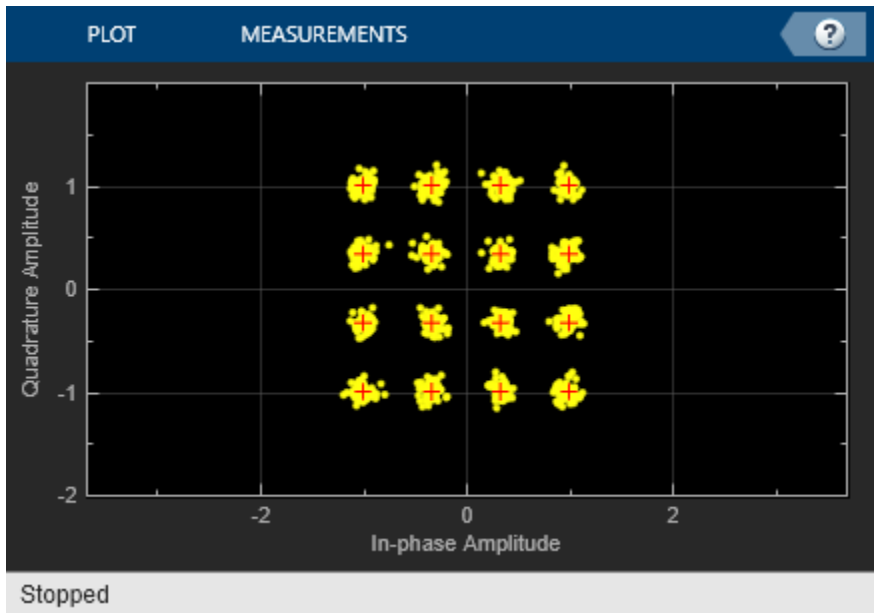
**HBPS**

The result of HBPS is shown in the following figures. The constellation diagram shows that it achieves similar performance compared to QSHB. This means that the HBPS is a good choice for the simulated 64x16 MIMO system.
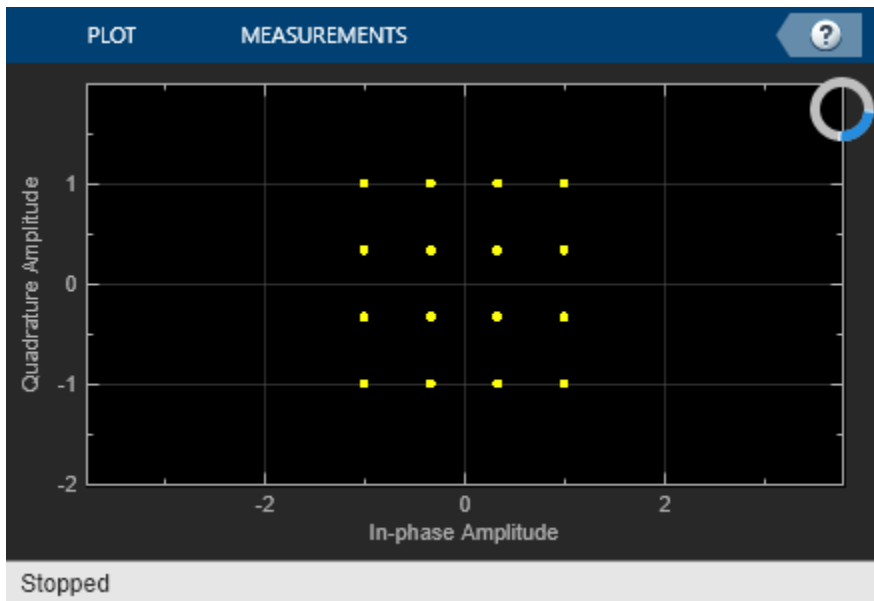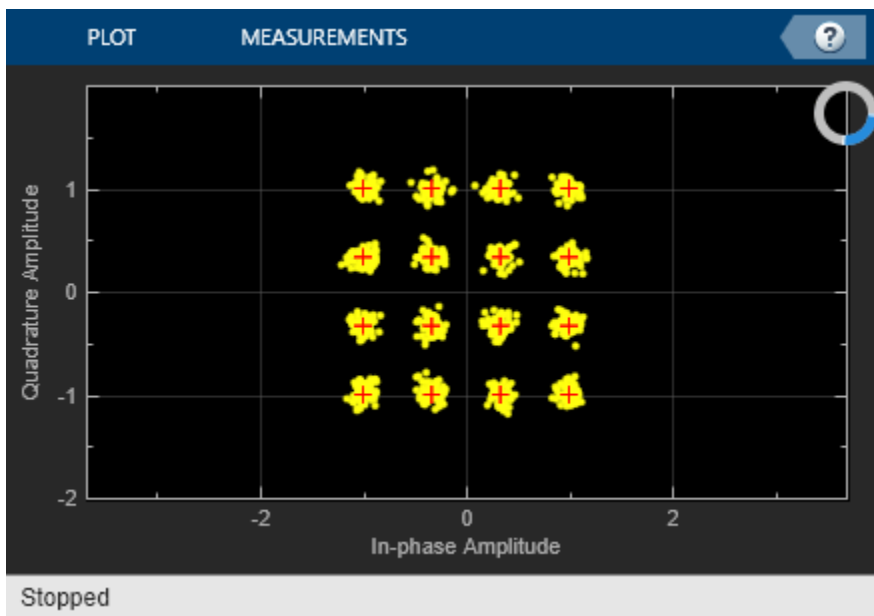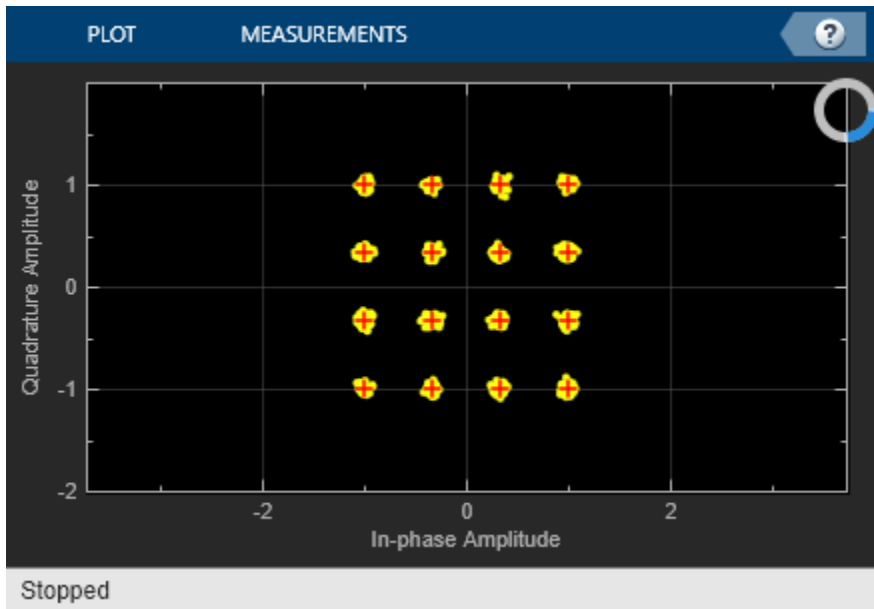
**Summary**

This example provides the Simulink model of two hybrid beamforming methods, QSHB and HBPS. The MIMO scattering channel is used to provide a realistic channel model for massive MIMO systems. The Simulink model is partitioned according to the functions in the signal flow, which gives guidance for hardware implementation. For a given H, the number of symbols can vary to simulate the variable coherent channel length. With this Simulink model, various system parameters and new hybrid beamforming algorithms can be studied. The system structure facilitates the hardware implementation.

**Reference**

[1] Andreas F. Molisch, et al. "Hybrid Beamforming for Massive MIMO: A Survey", IEEE Communications Magazine, Vol. 55, No. 9, September 2017, pp. 134-141

[2] Oma El Ayach, et al. "Spatially Sparse Precoding in Millimeter wave MIMO Systems, IEEE Transactions on Wireless Communications", Vol. 13, No. 3, March 2014

[3]. Emil Bjornson, Jakob Hoydis, Luca Sanguinetti, "Massive MIMO Networks: Spectral, Energy, and Hardware Efficiency", Foundations and Trends in Signal Processing: Vol. 11, No. 3-4, 2017

# Array Synthesis for Lidar Systems

This example shows how to design and optimize a phased array for lidar applications. The example describes the following workflow:

1  Import an antenna element pattern generated with Lumerical tools into MATLAB®
2  Design a linear phased array using the Lumerical antenna pattern for each element in the array
3  Determine the array element spacing and weighting for the array such that the azimuth pattern and elevation pattern (generated from the Lumerical tools) are closely matched between a desired steering range

A related example can be found on Lumerical's website at Lumerical Lidar Antenna Example. The Lumerical's example provides more details on how the antenna element is designed with the FDE solver; how the antenna element design is verified and extracted with 3D FDTD technique; and how the array designed using Phased Array System Toolbox™ is integrated in Lumerical's INTERCONNECT software.

This example requires Optimization Toolbox™.

**Introduction**

Lidar is used as a perception sensor in autonomous systems. Lidar sensors are capable of ranging millions of points per second due to high angular resolutions and fast steering speeds. Beam steering in lidar architectures can be accomplished with optical phased arrays. This example shows how to design the integrated optical phased array antenna which can be used for both transmitting and receiving functions.

**Modeling the Phased Array**

First, import the antenna pattern generated using Lumerical tools.

```
load('Lumerical_antenna_data.mat'); % Load Lumerical antenna pattern
```

The antenna occupies a large bandwidth. The data set generated with Lumerical tools contains antenna responses for 50 frequencies.

```
nfreq = numel(freqVector);              % 50 frequency vectors
ant = phased.CustomAntennaElement('FrequencyVector',freqVector,...
    'FrequencyResponse',zeros(1,nfreq),...
    'AzimuthAngles',az,'ElevationAngles',el,...
    'MagnitudePattern',pat_azel,'PhasePattern',zeros(size(pat_azel)));
```

The array to be designed is a 48-element linear array along azimuth direction. The goal is to

The array to be designed is a 48-element linear array with a hamming taper. The number of elements was chosen in this example to provide a narrow beam. The element spacing is set at 1.2 wavelength corresponding to the center frequency due to physical constrains in building lidar arrays. Since the spacing is larger than half wavelength, grating lobe may occur. However, in this application, the array only scans a limited range around the boresight. Therefore, grating lobes are not a concern.

```
c = 3e8;
lambda = c/freqVector(25);
N = 48;
```

```
antarray = phased.ULA(N,1.2*lambda,'Element',ant,'Taper',hamming(N));
stv = phased.SteeringVector('SensorArray',antarray,'PropagationSpeed',c);
```

Following figures show the resulting array patterns at boresight for 3 frequency values (frequency bands 1, 25, and 50) to see the resulting 3D beam pattern across the full range. Note how the main beam changes with frequency. In the plots below, the beams are steered at 0 degrees azimuth.

```
svang = [0;0];      % Steer at 0 degrees in azimuth

n = 1;
fc = freqVector(n);
pattern(antarray,fc,'Type','powerdb','Weights',stv(fc,svang));
title(sprintf('Array Pattern for Frequency Band %d',n));
snapnow;

n = 25;
fc = freqVector(n);
pattern(antarray,fc,'Type','powerdb','Weights',stv(fc,svang));
title(sprintf('Array Pattern for Frequency Band %d',n));
snapnow;

n = 50;
fc = freqVector(n);
pattern(antarray,fc,'Type','powerdb','Weights',stv(fc,svang));
title(sprintf('Array Pattern for Frequency Band %d',n));
snapnow;
```



Array Pattern for Frequency Band 1

Array Pattern for Frequency Band 25
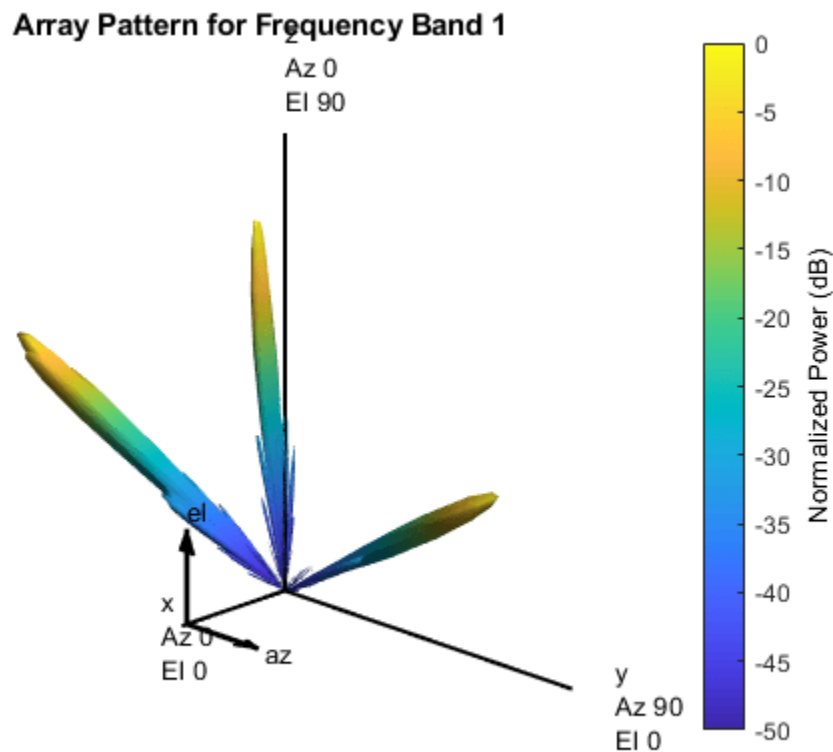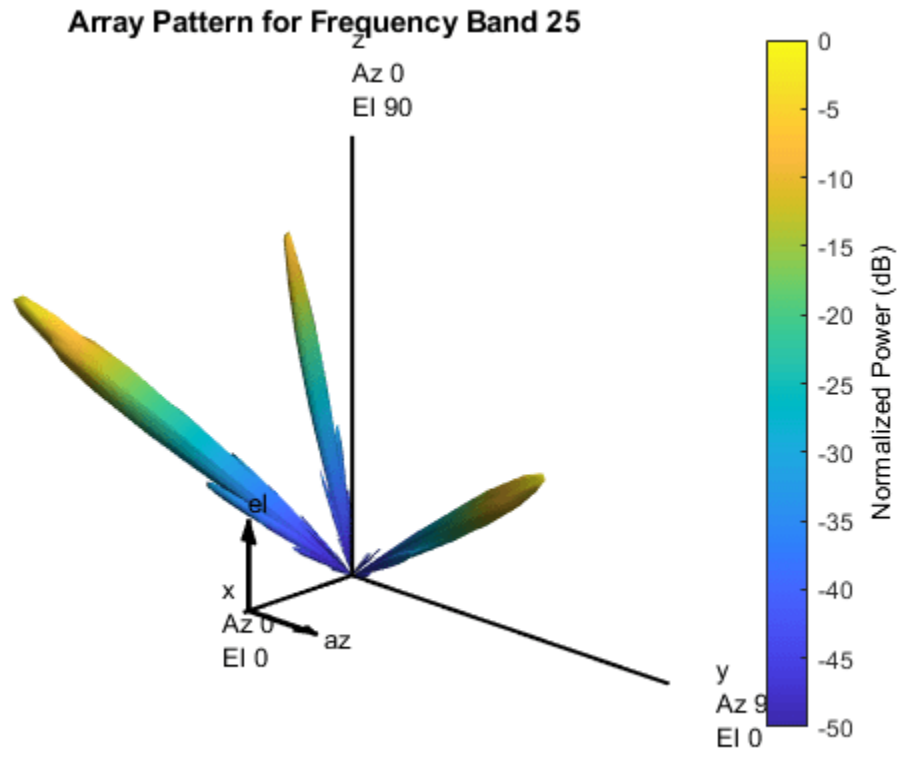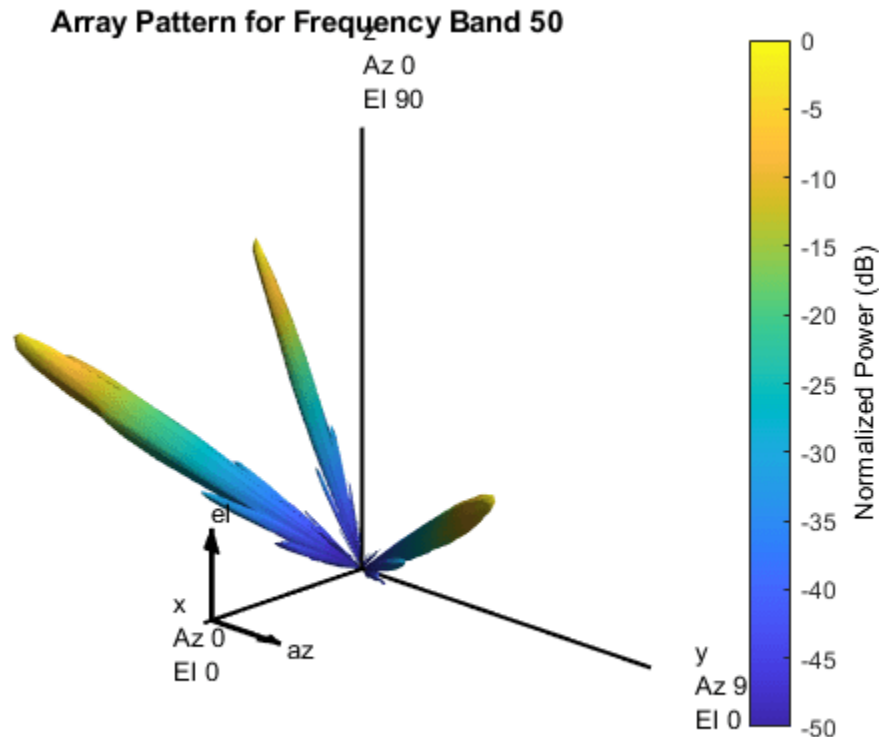
**Synthesizing Array Pattern**

In this example, the array steering in elevation is done using different carrier frequencies. However, the array steering in azimuth is done by weighting the elements in the linear array. Therefore, our goal, through optimization, is to find weights and element spacing such that the shape of the beam in the azimuth cut matches the shape in the elevation cut.

To get the best match across the frequency range, we start with the elevation cut of the middle frequency value (band 25) as the desired shape. Because the application only requires the scanning between $\pm 20$ degrees in azimuth, we will focus the pattern within the $\pm 40$ degrees region to ensure there is no grating lobes come into $\pm 20$ degrees region during scanning. Note that reflectors will be used on the physical array to ensure other transmissions are not sent out at angles outside $\pm 20$ degrees.

```
azimuth = -40:40;
n = 25;                % Use center frequency as the basis for optimization
fc = freqVector(n);
Beam_d = pattern(antarray,fc,azimuth,0,'Type','efield','Weights',stv(fc,svang),'Normalize',false
antpat = pattern(ant,fc,azimuth,0,'Type','efield','Normalize',false).';
```

The objective function is set to minimize the distance between the desired pattern and the one that is generated as a result of synthesis. For this optimization, we want to generate a common spacing value between elements and unique real weights for each element to facilitate the implementation phase in the Lumerical Interconnect tool. In addition, to ensure the lidar array can be realized, a starting point of 1.1 wavelength is set.

**18-545**

```
w_i_re = ones(N,1)./2;                  % initial real values
w_i_im = zeros(N,1);                    % initial imaginary values
lambda_i = 1.1;                         % initial wavelength

objfun = @(x)norm(abs((x(1:N))'*...
    steervec((-(N-1)/2:(N-1)/2)*x(end),azimuth).*antpat)-Beam_d);

x_ini = [w_i_re;lambda_i];

[x_o, fval, ef, output] = ...
    fmincon(objfun,x_ini,[],[],[],[],[(zeros(N,1));1],[ones(N,1);1.2],[],...
    optimoptions(@fmincon,'Display','final-detailed','MaxFunctionEvaluations',1e4));
```

```
Optimization completed: The relative first-order optimality measure, 6.222617e-07,
is less than options.OptimalityTolerance = 1.000000e-06, and the relative maximum constraint
violation, 0.000000e+00, is less than options.ConstraintTolerance = 1.000000e-06.
```

Next plot shows the comparison between the desired pattern and the synthesized pattern.

```
azplot = -40:40;
Beam_d_plot = pattern(antarray,fc,azplot,0,'Type','efield','Weights',stv(fc,svang),'Normalize',fa
antpat_plot = pattern(ant,fc,azplot,0,'Type','efield','Normalize',false).';
Beam_syn_plot = abs((x_o(1:N))'*steervec((-(N-1)/2:(N-1)/2)*x_o(end),azplot).*antpat_plot);
plot(azplot,mag2db(Beam_d_plot),'-',azplot,mag2db(Beam_syn_plot),'--');
legend('Desired','Synthesized')
title(sprintf('Frequency band %d with spacing of %5.2f wavelength',n,x_o(end)));
xlabel('Angle (deg)')
ylabel('Array Pattern')
```
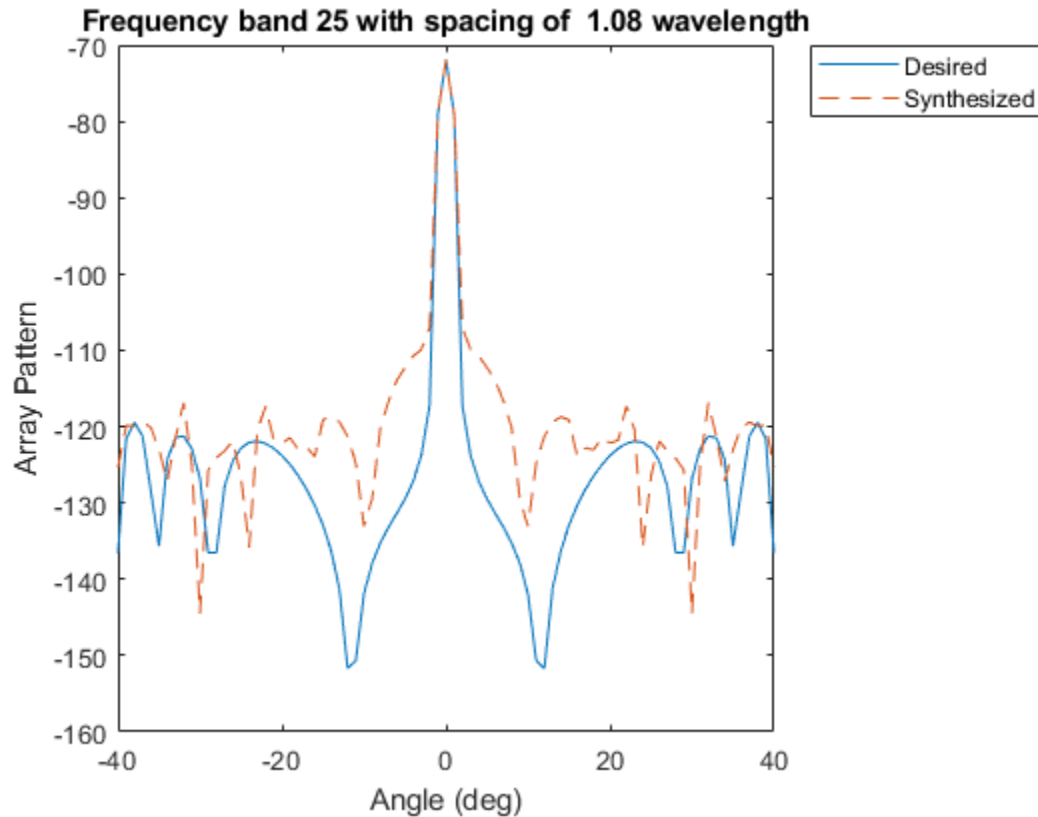
The figure shows a strong match between the desired pattern and the synthesized pattern between ±
20 degrees. Again, anything outside this angle range will be blocked by a reflector in the actual
system.

**Verifying Scanning Behavior**

To verify the resulting weights and element spacing, we steer the array to 20 degrees azimuth at both
frequency bands #1 and #50 and examine if the array performance satisfies the application needs.

```
n = 50;
fc = freqVector(n);
wmag = x_o(1:N);

svang = [20;0];
azplot = -40:40;

Beam_d_plot = pattern(antarray,fc,azplot,0,'Type','efield','Weights',stv(fc,svang),'Normalize',fa

antpat_plot = pattern(ant,fc,azplot,0,'Type','efield','Normalize',false).';
weights_o = wmag.*steervec((-(N-1)/2:(N-1)/2)*x_o(end),svang);
Beam_syn_plot = abs(weights_o'*steervec((-(N-1)/2:(N-1)/2)*x_o(end),azplot).*antpat_plot);

plot(azplot,mag2db(Beam_d_plot),'-',azplot,mag2db(Beam_syn_plot),'--');
legend('Desired','Synthesized')
title(sprintf('Frequency band %d with spacing of %5.2f wavelength',n,x_o(end)));
xlabel('Angle (deg)')
ylabel('Array Pattern')
```

```
snapnow;

n = 1;
fc = freqVector(n);

Beam_d_plot = pattern(antarray,fc,azplot,0,'Type','efield','Weights',stv(fc,svang),'Normalize',fa

antpat_plot = pattern(ant,fc,azplot,0,'Type','efield','Normalize',false).';
weights_o = wmag.*steervec((-(N-1)/2:(N-1)/2)*x_o(end),svang);
Beam_syn_plot = abs(weights_o'*steervec((-(N-1)/2:(N-1)/2)*x_o(end),azplot).*antpat_plot);

plot(azplot,mag2db(Beam_d_plot),'-',azplot,mag2db(Beam_syn_plot),'--');
legend('Desired','Synthesized')
title(sprintf('Frequency band %d with spacing of %5.2f wavelength',n,x_o(end)));
xlabel('Angle (deg)')
ylabel('Array Pattern')
snapnow;
```
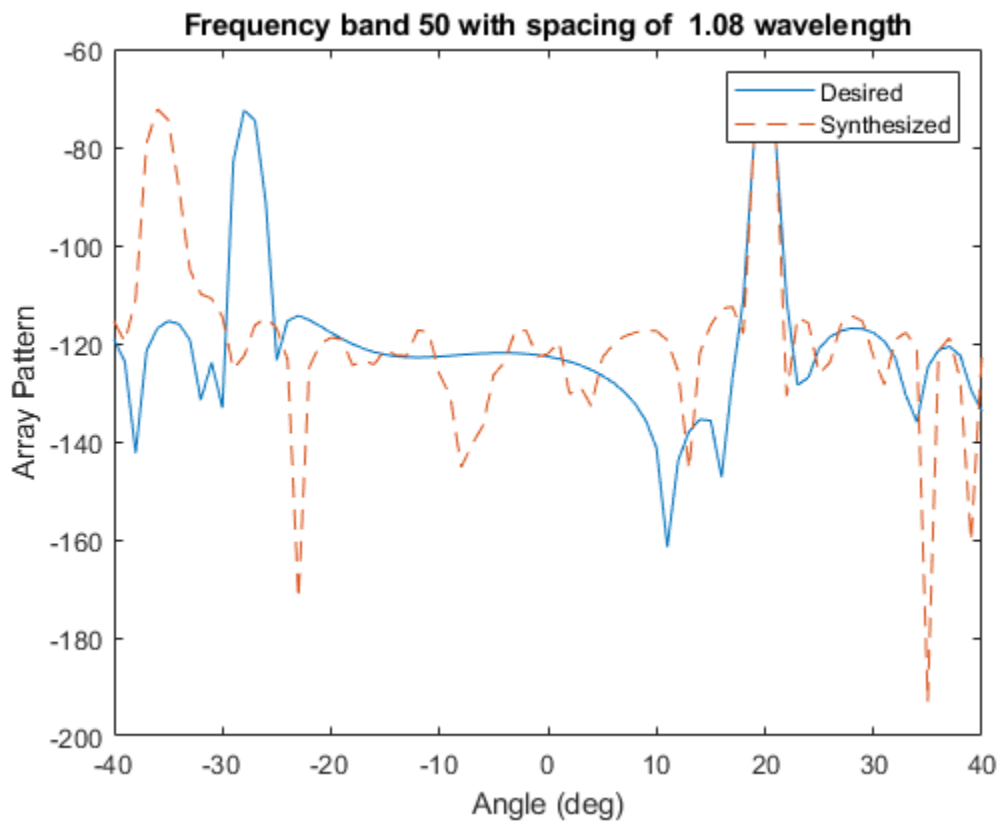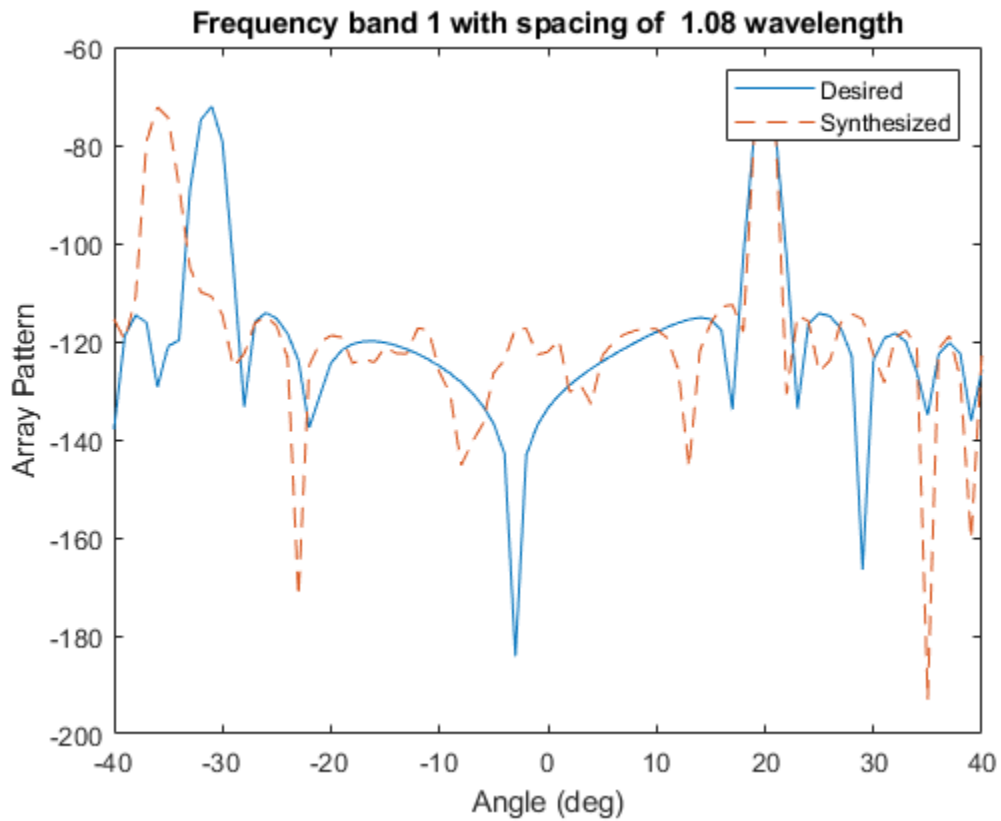
Finally, the following figure shows the resulting weights from the optimization.

```
% plot the results
plot(1:N,wmag);
xlabel('Antenna element number');
ylabel('Normalized weight');
title('Final antenna weights after optimization');
```

Final antenna weights after optimization
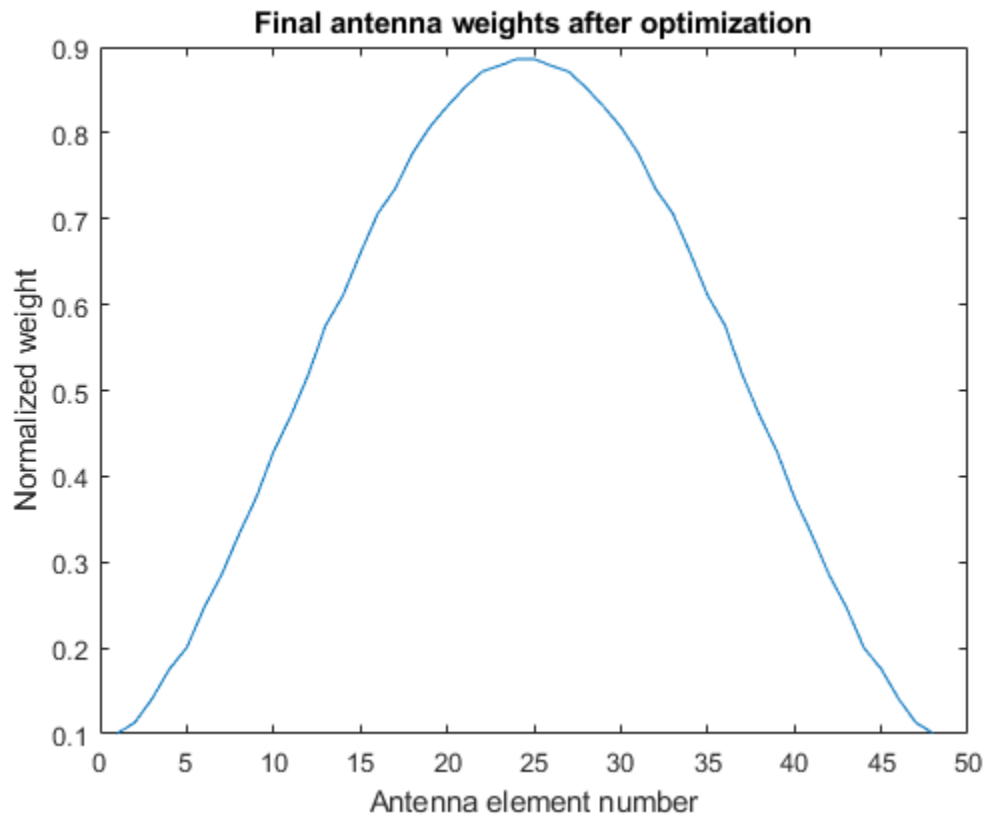
**Summary**

This example shows how array synthesis techniques can be applied to help design a phased array lidar to achieve a desired beam pattern.

# Estimate Range and Doppler Using Pulse Compression

This example shows the effects of *pulse compression,* where a transmitted pulse is modulated and correlated with the received signal. Radar and sonar systems use pulse compression to improve signal-to-noise ratio (SNR) and range resolution by shortening the duration of echoes. This example also demonstrates *Doppler processing,* where the radial velocity of a target is determined from the Doppler shift created by target motion.

### Determining Range and Speed of Targets

The following radar system propagates a pulse train of linear frequency modulated (FM) waveforms to a moving target and receives the echoes. By applying matched filtering and Doppler processing to the echoes, the radar system can effectively detect the range and speed of the target.

Specify the requirements for the radar system. This example uses a carrier frequency `fc` of 3 gHz and a sampling rate `fs` of 1 mHz.

```
fc = 3e9;
fs = 1e6;
c = physconst('LightSpeed');
```

Create the system objects to model a radar system. The system is monostatic. The transmitter is located at (0,0,0) and is stationary, while the target is located at (5000,5000,0) with a velocity of (25,25,0) with a radar cross-section (RCS) of 1 square meter.

```
antenna = phased.IsotropicAntennaElement('FrequencyRange',[1e8 10e9]);
transmitter = phased.Transmitter('Gain',20,'InUseOutputPort',true);
txloc = [0;0;0];
tgtloc = [5000;5000;0]; % Radial Dist ~= 7071 m
tgtvel = [25;25;0]; % Radial Speed ~= 35.4 m/s
target = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1,'OperatingFrequency',fc);
antennaplatform = phased.Platform('InitialPosition',txloc);
targetplatform = phased.Platform('InitialPosition',tgtloc,'Velocity',tgtvel);
radiator = phased.Radiator('PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
channel = phased.FreeSpace('PropagationSpeed',c,...
    'OperatingFrequency',fc,'TwoWayPropagation',false);
collector = phased.Collector('PropagationSpeed',c,...
    'OperatingFrequency',fc,'Sensor',antenna);
receiver = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SeedSource','Property','Seed',2e3);
```

Create a linear FM waveform with a pulse width of 10 µs, a pulse repetition frequency of 10 kHz, and a sweep bandwidth of 100 kHz. The matched filter coefficients are generated from this waveform.

```
waveform = phased.LinearFMWaveform('PulseWidth',10e-6,'PRF',10e3,'OutputFormat','Pulses','NumPuls
wav = waveform();
c = physconst('LightSpeed');
maxrange = c/(2*waveform.PRF);
SNR = npwgnthresh(1e-6,1,'noncoherent');
lambda = c/fc;
tau = waveform.PulseWidth;
Ts = 290;

filter = phased.MatchedFilter(...
```

```
    'Coefficients',getMatchedFilter(waveform),...
    'GainOutputPort',true);
```

To improve the Doppler resolution, the system emits 64 pulses, and the echoes are stored in `rxsig`. The data matrix stores the fast time samples (time within each pulse) along each column and the slow time samples (time between pulses) along each row.

```
numPulses = 64;
rxsig = zeros(length(wav),numPulses);

for n = 1:numPulses
    [tgtloc,tgtvel] = targetplatform(1/waveform.PRF);
    [tgtrng,tgtang] = rangeangle(tgtloc,txloc);

    [txsig, txstatus] = transmitter(wav);
    txsig = radiator(txsig,tgtang);
    txsig = channel(txsig,txloc,tgtloc,[0;0;0],tgtvel);
    txsig = target(txsig);
    txsig = channel(txsig,tgtloc,txloc,tgtvel,[0;0;0]);
    txsig = collector(txsig,tgtang);
    rxsig(:,n) = receiver(txsig,~txstatus);
end

prf = waveform.PRF;
fs = waveform.SampleRate;
response = phased.RangeDopplerResponse('DopplerFFTLengthSource','Property','DopplerFFTLength',204
filt = getMatchedFilter(waveform);
[resp,rng_grid,dop_grid] = response(rxsig,filt);

plotResponse(response,rxsig,filt,'Unit','db')
ylim([0 12000])
```

**Range-Speed Response Pattern**
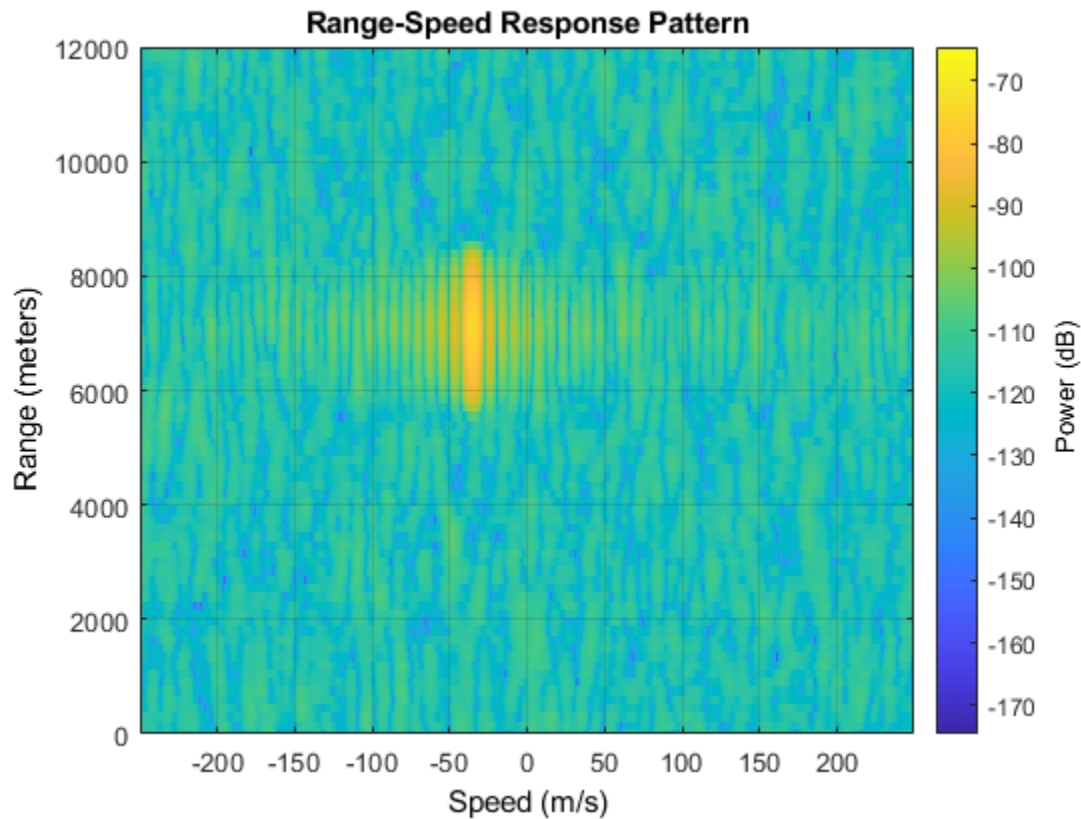


The response shows that the target is moving at about –40 m/s, and that the target is moving away from the transmitter since the speed is negative.

Calculate the range gates corresponding to the travel speed of the signal. The plot of the first slow time sample shows the largest peak around 7000 m, which corresponds with the range-speed response pattern plot.

```
fasttime = unigrid(0,1/fs,1/prf,'[)');
rangebins = (physconst('Lightspeed')*fasttime/2);

plot(rangebins,abs(rxsig(:,1)))
```
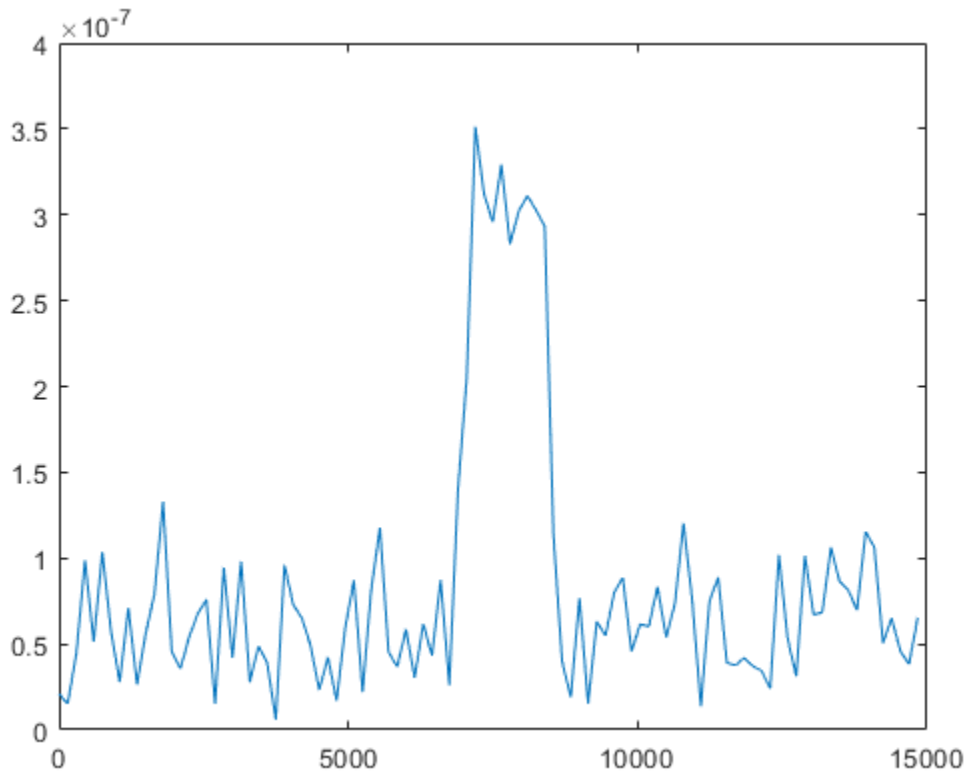
**Determining Range**

Create a threshold where the probability of false alarm is less than 1e-6. Use noncoherent integration of the 64 pulses assuming that the signal is in white Gaussian noise. Take the largest peak above the threshold and dipslay the estimated target range.

```
pfa = 1e-6;
NoiseBandwidth = 5e6/2;
npower = noisepow(NoiseBandwidth, receiver.NoiseFigure,receiver.ReferenceTemperature);
thresh = npwgnthresh(pfa,numPulses,'noncoherent');
thresh = npower*db2pow(thresh);
[pks,range_detect] = findpeaks(pulsint(rxsig,'noncoherent'),'MinPeakHeight',thresh,'SortStr','des
range_estimate = rangebins(range_detect(1));
fprintf("Range Estimate: %3.2f m", range_estimate);
```

Range Estimate: 7344.92 m

**Determining Speed**

Take the slow-time samples corresponding to the range bin containing the detected target and plot the power spectral density estimate of the slow-time samples using the `periodogram` function.

```
ts = rxsig(range_detect(1),:).';
periodogram(ts,[],256,prf,'centered')
```

**Power Spectral Density**
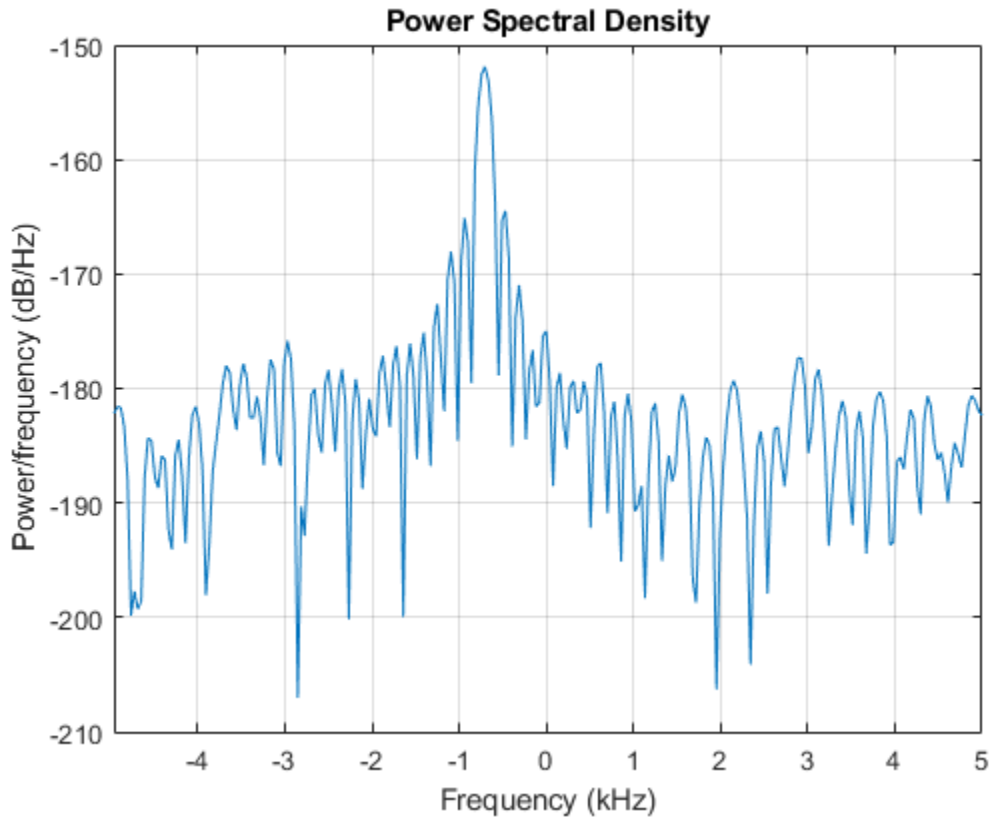


The peak frequency corresponds to the Doppler shift divided by 2, which can be converted into target speed. A positive speed means that the target is approaching the transmitter, while a negative speed indicates that the target is moving away from the transmitter.

```
[Pxx,F] = periodogram(ts,[],256,prf,'centered');
```

```
[Y,I] = max(Pxx);
lambda = physconst('Lightspeed')/fc;
tgtspeed = dop2speed(F(I)/2,lambda);
fprintf("Doppler Shift Estimate: %2.2f Hz",F(I)/2)
```

```
Doppler Shift Estimate: -351.56 Hz
```

```
fprintf("Speed Estimate: %2.2f m/s",tgtspeed)
```

```
Speed Estimate: -35.13 m/s
```

# Compare Ambiguity Functions for Different Wave Modulation Schemes

This example shows how to visualize and interpret different waveform processing schemes and their tradeoffs in the **Pulse Waveform Analyzer** app.

**Introduction**

Radar systems use *matched filters* in the receiver chain to improve signal-to-noise ratio (SNR). Matched filters are time-reversed and conjugated versions of the transmitted signal. The *ambiguity function* is the output of a matched filter for a given input waveform. The ambiguity function is used to see a waveform's resolution and ambiguities in both the Doppler and range domains. The ideal ambiguity function is a two-dimensional Dirac delta function, similar to a thumbtack shape, that has no ambiguities. However, this function is unachievable because it requires a waveform with infinite duration and *bandwidth*. Bandwidth is the difference between the upper and lower frequencies of a waveform. The ambiguity functions of different waveforms provide insight into their advantages and disadvantages.

This example uses the radar system requirements introduced and characterized in the "Waveform Analysis Using the Ambiguity Function" on page 18-155 example and approximates the speed of light to be 3e8 m/s. The desired system has a maximum unambiguous range of 15 km and a range resolution of 1.5 km.

**Rectangular Waveforms**

**Description**

The most basic waveform is the rectangular waveform, whose amplitude alternates between two values, similar to a square wave. The default waveform in the **Pulse Waveform Analyzer** is a rectangular waveform.

Create a waveform with these parameters:

- **Name:** Rect
- **Waveform:** Rectangular
- **Sample Rate (Hz):** 200 kHz
- **PRF (Hz):** 10 kHz
- **Pulse Width (s):** 10 µs

Use a sample rate that is at least double the highest frequency component of the waveform, which in this case is the bandwidth. In a rectangular waveform, the bandwidth is the reciprocal of the pulse width. Since the bandwidth is 100 kHz, the sample rate is 200 kHz.
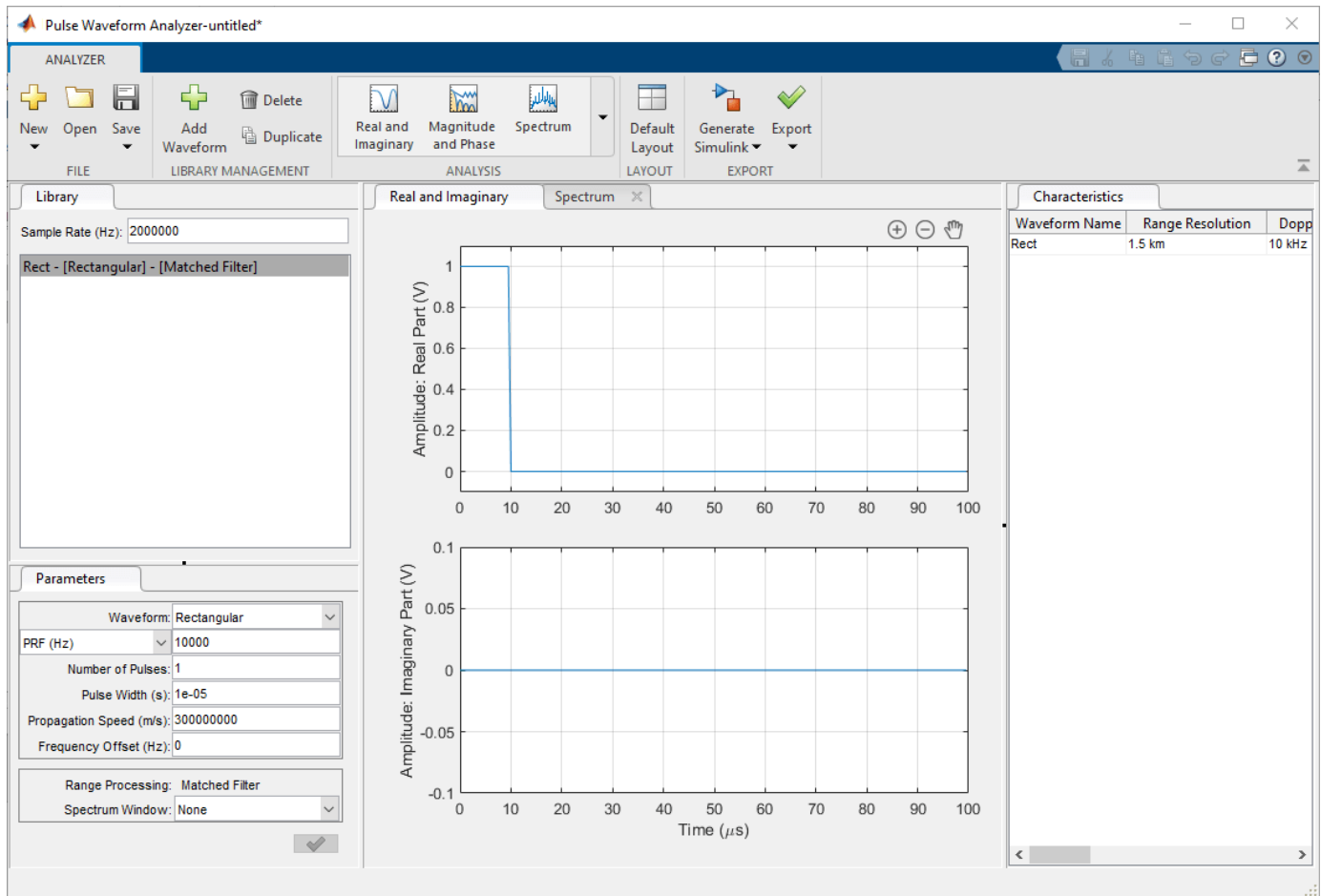
## Range and Resolution

The **Characteristics** tab shows the properties of a radar system that uses a given rectangular waveform. To observe the relationship between the pulse repetition frequency (PRF) and other characteristics, increase and decrease the PRF values.

- Duplicate the rectangular waveform twice, and change their PRFs to 5 kHz and 20 kHz.

The **Characteristics** tab shows that the PRF is inversely proportional to the maximum unambiguous range because the maximum unambiguous range is determined by the amount of time between pulses. The relationship between PRF and maximum unambiguous range applies for all pulsed waveforms, as the farther apart the pulses are spread, the farther the signal can propagate and return before the next pulse is emitted.

Beyond simply measuring range, radar systems measure velocity by using the Doppler effect. The greater the Doppler shift is compared to the original signal, the faster the target is moving. As such, Doppler shift and speed are directly proportional and are often used interchangeably. The **Characteristics** tab displays Doppler resolution and maximum Doppler shift, which correspond to the speed resolution and the maximum detectable speed, respectively. These waveforms also demonstrate the *Doppler Dilemma*, where a small PRF gives a larger maximum unambiguous range but poor maximum Doppler while a larger PRF gives better maximum Doppler but worse maximum unambiguous range.

- Change the PRF for each waveform back to 10 kHz and change the pulse width of the duplicate waves to 20 and 30 µs.

The **Characteristics** tab shows that smaller pulse widths give better range resolution and a smaller minimum range. The tradeoff for a smaller pulse width is that it requires a higher peak power for the return echo to be detected reliably.

The original waveform has a maximum range of 15 km and a range resolution of 1.5 km, but its Doppler resolution is 10 kHz. Assuming that the radar is operating at 1 GHz, the Doppler resolution dictates that the radar system cannot separate targets with a speed difference smaller than 30 km/s, which is far too large to be practical for many real-world radar systems. To further visualize the range and Doppler domains, view the different ambiguity plots.

- Click on the **Analysis** drop-down menu and, under the **Ambiguity Plots** section, add the Contour, Surface, Delay Cut, and Doppler Cut plots.
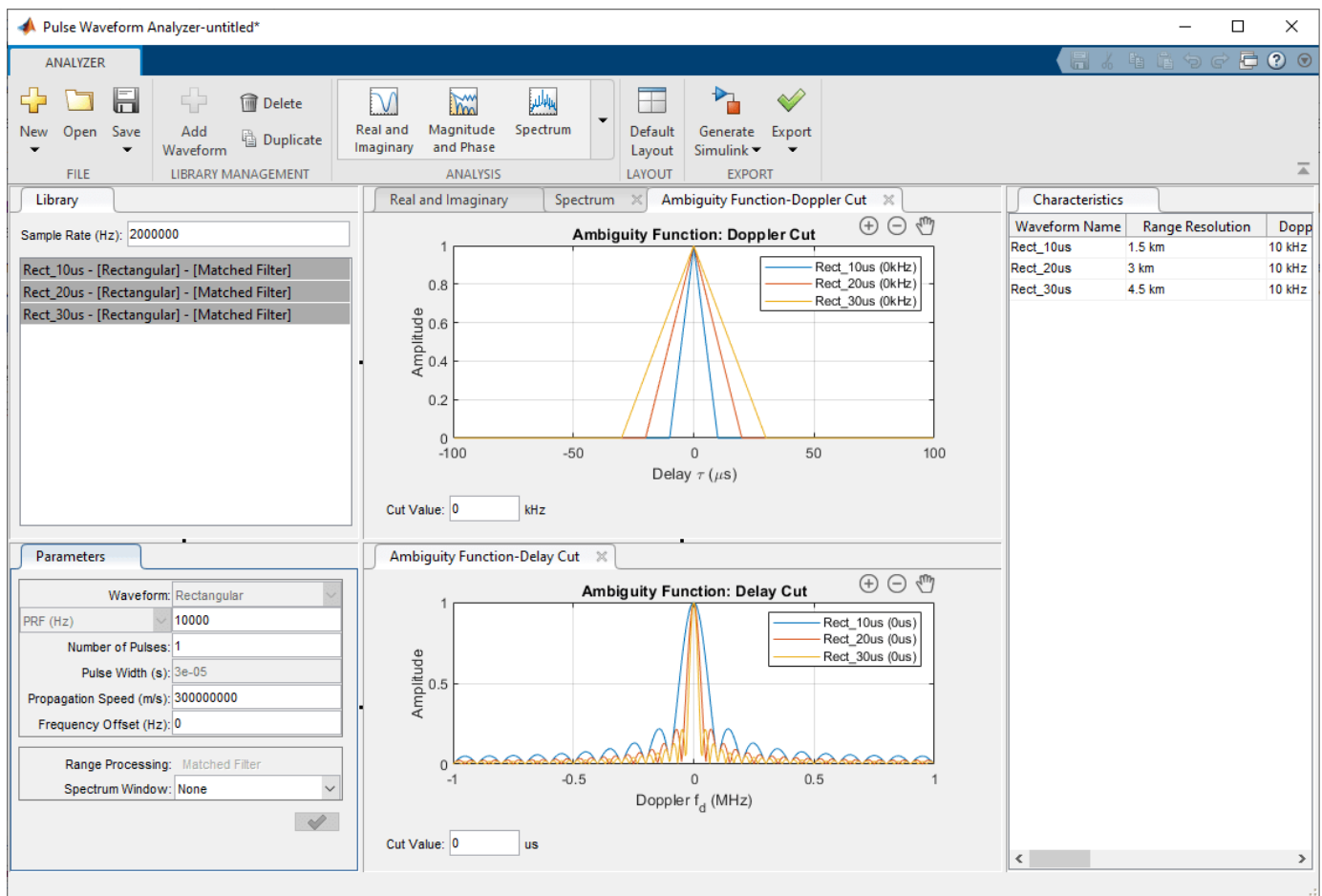


The Doppler Cut graph at 0 kHz shows the autocorrelation function (ACF) of the waveform, which corresponds to the matched filter response of a stationary target. The first null response for each waveform is the same as its respective pulse width. To obtain the range resolution, multiply the pulse width by the speed of light over 2 to account for the round trip. For the example waveforms, the range resolutions are 1.5 km, 3 km, and 4.5 km.

The zero-delay cut plot shows large gaps until the first null response for each of the waveforms. Looking at the 10 µs pulse width waveform, the first null is at 0.1 MHz, which translates to a Doppler shift of 100 kHz or 30 km/s. In other words, two targets must have speeds that differ by over 30 km/s to be separated via Doppler response, which is unrealistic in most radar situations.
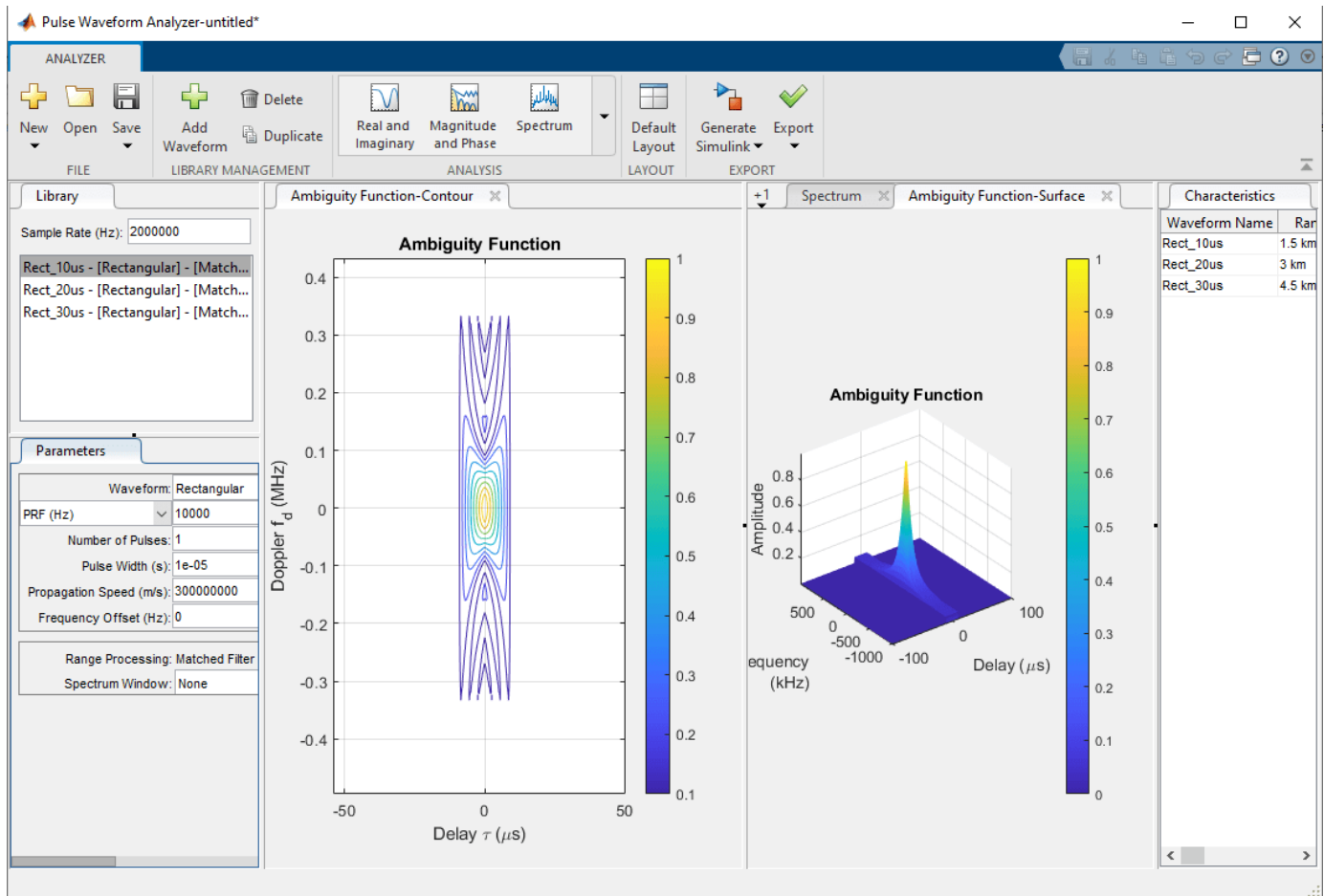


The Contour plot displays the nonzero response of the ambiguity function. Because the *duty cycle*, or the ratio of the pulse width to the pulse period, is 10%, the nonzero response only occupies about 10% of all delays. The surface ambiguity function shows the response in relation to both delay and Doppler, which is simply another way to visualize the 3-D contour plot.

The changes in the pulse width and the PRF show how to improve maximum ambiguous range and range resolution, but the Doppler resolution remains poor. The solution is to use a smaller pulse width with a larger PRF, but both changes greatly reduce the maximum power and thus the SNR, making it more difficult to detect objects. The product of the bandwidth and the pulse length is called the *time-bandwidth product*, and since the bandwidth and the pulse length of a rectangular wavelength are inversely proportional, the time-bandwidth product cannot exceed 1 for this waveform. Because of these tradeoffs, rectangular waveforms are rarely used in practical radar systems.

**Linear FM**

**Description**

Linear frequency modulated (FM) waveforms are phase-modulated waveforms whose frequency either increases or decreases linearly throughout the duration of the pulse. Linear FM waveforms are a popular choice for radar systems because, unlike in rectangular waveforms, the pulse width and the energy of the pulse are decoupled due to the changing frequency. This decoupling makes the time-bandwidth product exceed 1 and allows for improved target detection ability. The **Pulse Waveform Analyzer** app has the functionality to model these waveforms as well. For more information on linear FM waveforms, view "Linear Frequency Modulated Pulse Waveforms" on page 4-6

Create a waveform with these parameters:

- **Name:** LFM1
- **Waveform:** Linear FM
- **Sample Rate (Hz):** 200 kHz
- **PRF (Hz):** 10 kHz
- **Pulse Width (s):** 50 µs
- **Sweep Bandwidth (Hz):** 100 kHz

Click on the **Spectrum** tab to view the peak power.
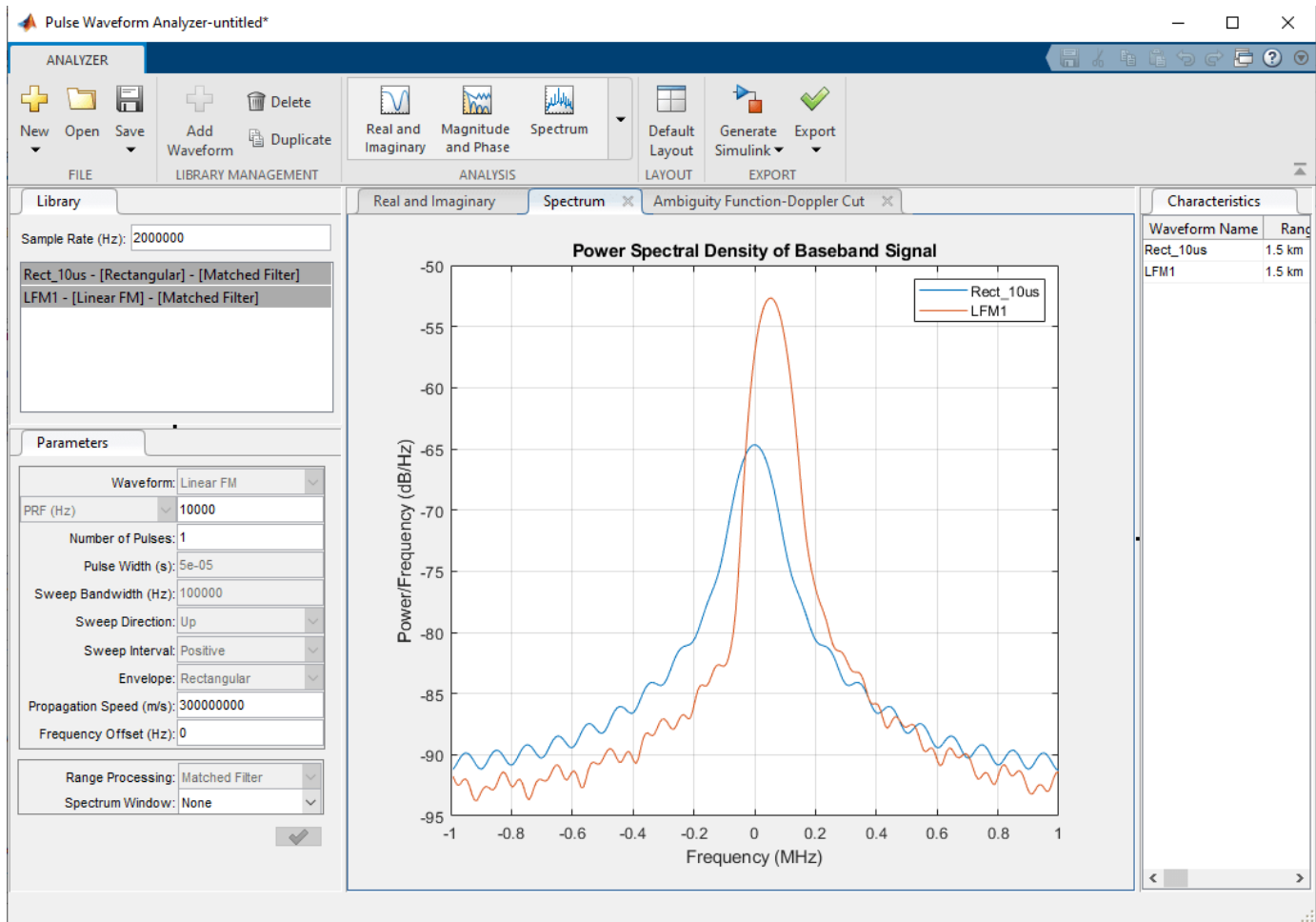
## Range and Resolution

Since the waveform is longer, the power is increased. However, the range resolution, the Doppler resolution, and the maximum unambiguous range are the same as for the rectangular waveform due to the effects of the frequency modulation. The **Characteristics** tab shows that the minimum range is greatly increased, meaning that the system cannot detect any objects that are closer than 7.5 km. To improve the Doppler resolution and decrease the minimum range, use numerous pulses.

- Decrease the pulse width to 10 μs and increase the number of pulses to 5.

By using a coherent pulse train and Doppler processing, the Doppler resolution improves in proportion to the number of pulses added. The **Characteristics** tab shows that range resolution, maximum Doppler, and maximum unambiguous range still remain the same, but the Doppler resolution is much improved. The tradeoff for adding more pulses is that sidelobes are now present, which can be seen in the **Matched Filter Response** tab. One way to reduce these sidelobes is to apply a window.

- Duplicate the linear FM waveform, and in the copied waveform, change the **Spectrum Window** from None to Hann and set the **Spectrum Range** from 0 to 200 kHz.
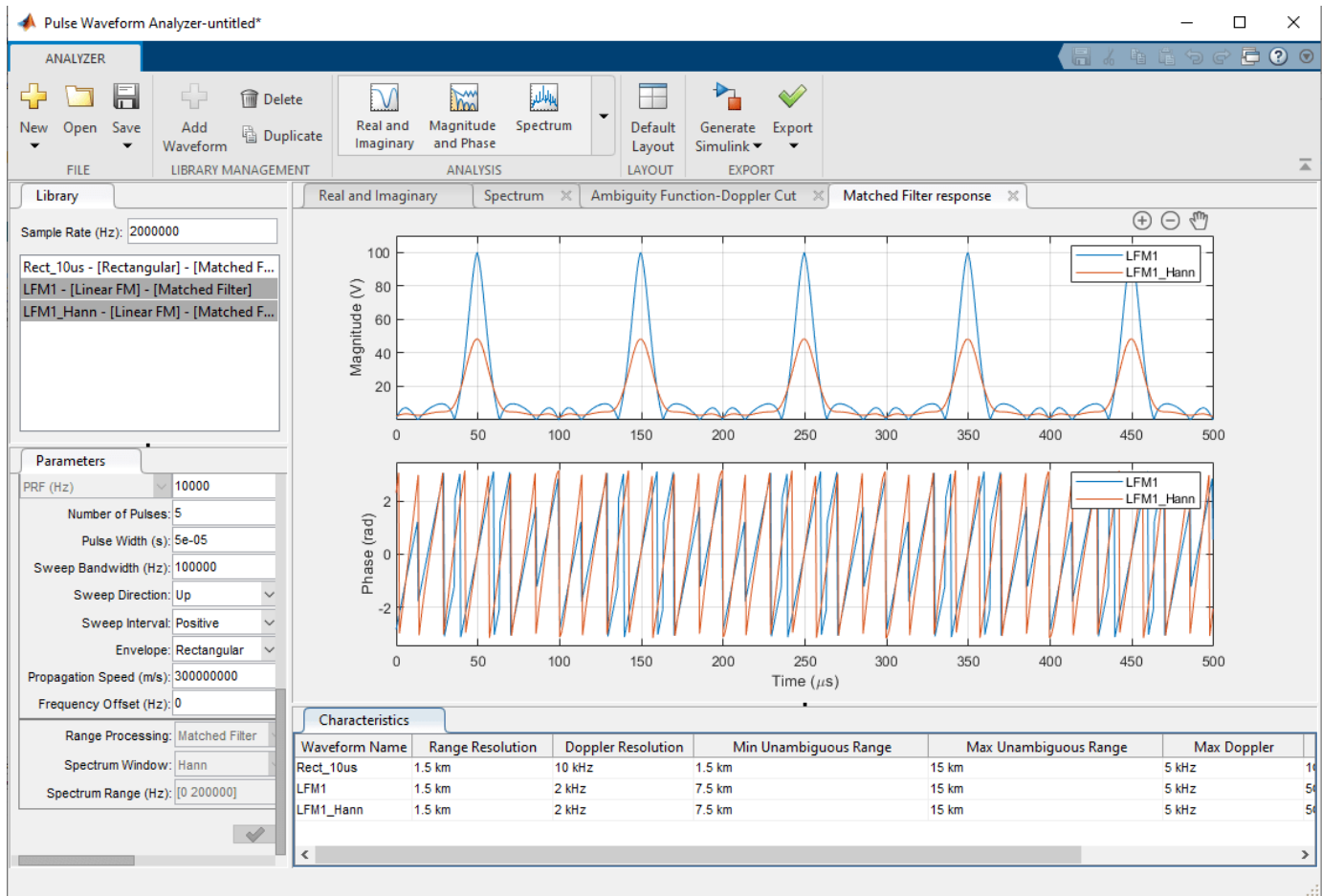
Press **Ctrl** and click on the two waveforms to compare them side by side. The window decreases the mainlobe's power from 100 V to less than 50 V, which is a loss of around 33 dB in power, and the width of the mainlobe is also wider compared to the waveform where you did not apply a window. However, the window does flatten the sidelobes, which makes detection using thresholding more reliable.

### Frequency Modulated Continuous Waveforms (FMCW)

### Description

Frequency Modulated Continuous Waveforms (FMCW) are similar to linear FM waveforms but are continuous rather than pulsed, which is effectively a linear FM waveform but with a duty cycle of 100%. FMCWs are often used in short-range automotive radar systems due to their sharp resolution for both Doppler and range.

Create a waveform with these parameters:

- **Name:** FMCW
- **Waveform:** FMCW
- **Sample Rate (Hz):** 200 kHz
- **Sweep Time (s):** 100 µs

- **Sweep Bandwidth (Hz):** 500 kHz
- **Number of Sweeps:** 1

**Range and Resolution**

Check the **Characteristics** tab and see that most of the characteristics are the same as for the rectangular waveform. One notable difference is that with continuous waveforms, the receiver always remains on, so the minimum range is always 0. To improve the range resolution, increase the sweep bandwidth.

- Increase the sweep bandwidth to 500 kHz.

The range resolution improves by a factor of 5 to 0.3 km. However, the Doppler resolution is still poor at 10 kHz. One way to improve this is by increasing the sweep time, which essentially lengthens the duration of the frequency modulation.

- Increase the sweep time from 0.1 ms to 1 ms.

The increased sweep time improves the Doppler resolution by a factor of 10, reducing it to 1 kHz. Although the Doppler resolution improves, the tradeoff for lengthening the signal is that the maximum Doppler decreases by the same factor, so if there are fast moving targets that exceed this Doppler limit, the radar system is unable to determine their speed without additional processing complexity.

- Revert the sweep time back to 0.1 ms and now change the number of sweeps to 10 from 1.

Increasing the number of sweeps follows the same principle of coherent pulse trains to improve the Doppler resolution from 10 kHz to 1 kHz. The Doppler Cut tab displays the ambiguity function, which has sizable sidelobes.
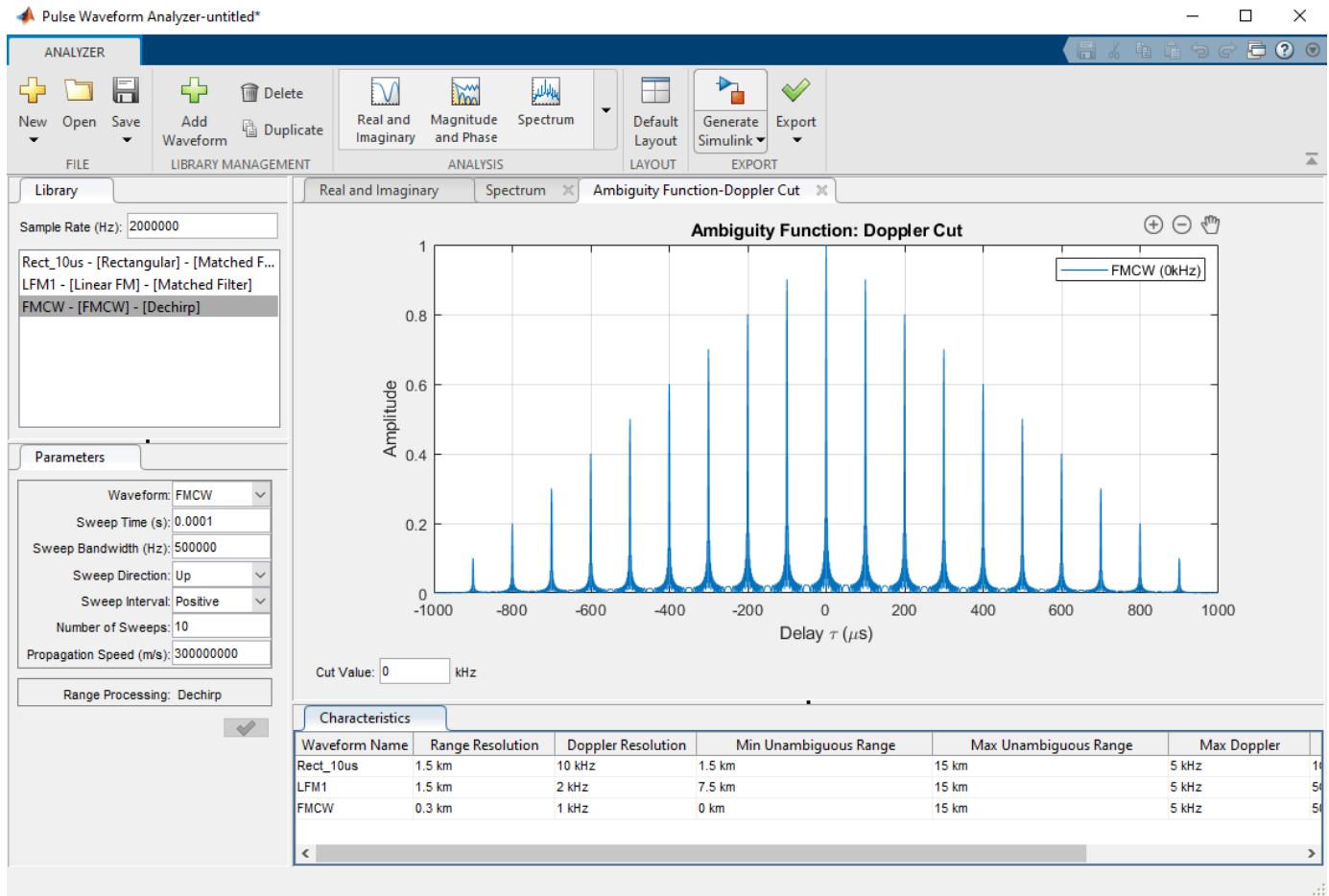
An ambiguity function that has numerous sidelobes is often referred to as a "bed of nails" ambiguity function. To get a better look, compare the 3-D ambiguity plots side by side.
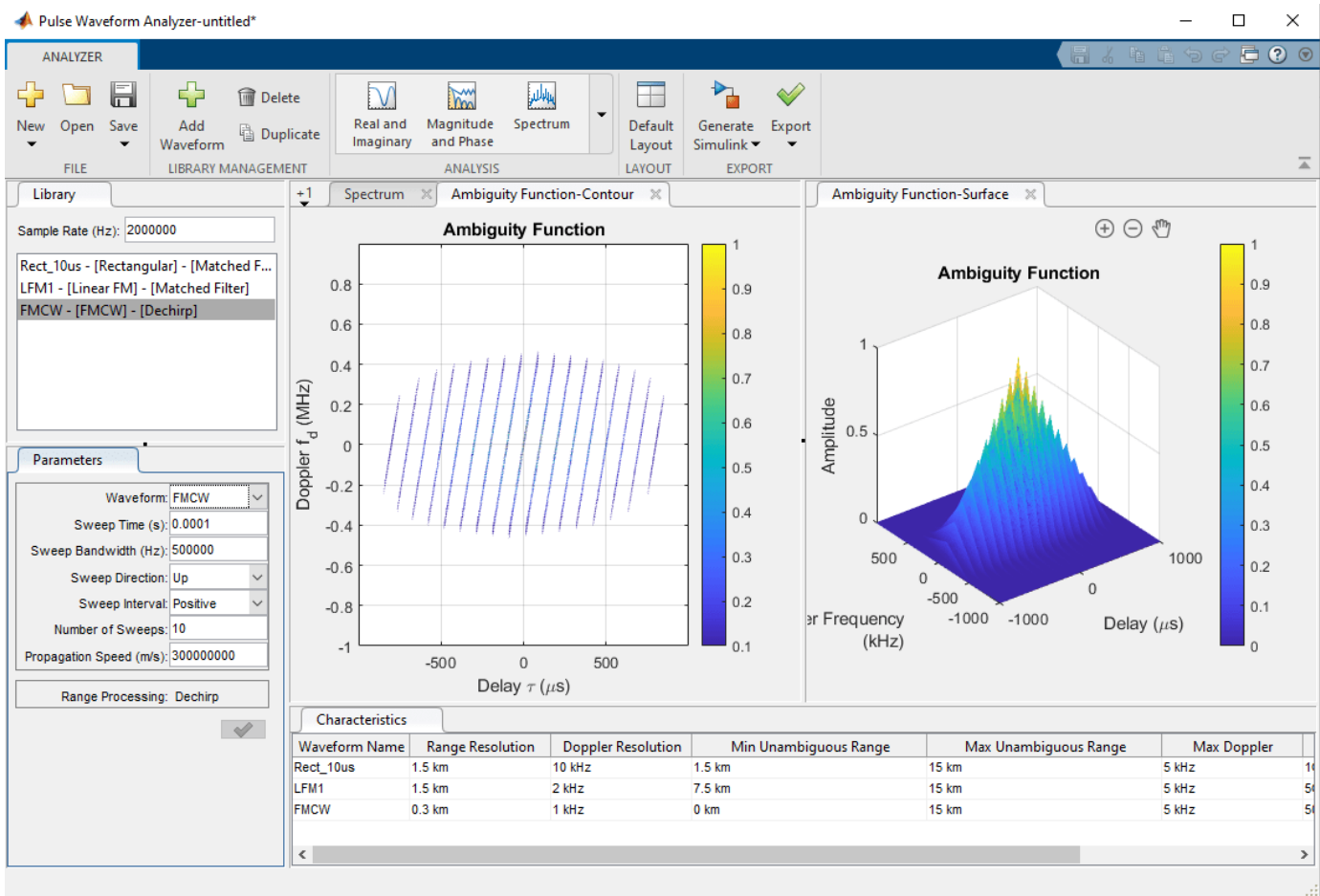
The ambiguity function is slightly skewed along the Doppler-Delay plane, which shows that slight changes in Doppler can cause errors in range measurements. This phenomenon is called range-Doppler coupling and occurs commonly in linear FM waveforms.

Another tradeoff with the FMCW waveform is that the maximum unambiguous range is a function of the sweep time, which can be difficult to increase past a certain point. Thus, many FMCW radars are limited to a short range, but because of their improved range and Doppler resolution compared to other waveforms, FMCW waveforms are usually used in systems that require high measurement accuracy.

## Summary

This example shows how to use the **Pulse Waveform Analyzer** app to compare different types of waveforms, including rectangular, linear FM, and FMCW waveforms. The ambiguity function of a waveform is the output of a matched filter with the waveform as input, and the ambiguity function serves as a valuable tool for determining the effectiveness of a waveform for a given radar system.

# NR Downlink Transmit-End Beam Refinement Using CSI-RS

This example demonstrates the downlink transmit-end beam refinement procedure using the channel state information reference signal (CSI-RS) from 5G Toolbox™. The example shows how to transmit multiple CSI-RS resources in different directions in a scattering environment and how to select the optimal transmit beam based on reference signal received power (RSRP) measurements.

**Introduction**

In NR 5G, frequency range 2 (FR2) operates at millimeter wave (mmWave) frequencies (24.25 GHz to 52.6 GHz). As the frequency increases, the transmitted signal is prone to high path loss and penetration loss, which affects the link budget. To improve the gain and directionality of the transmission and reception of the signals at higher frequencies, beamforming is essential. Beam management is a set of Layer 1 (physical layer) and Layer 2 (medium access control) procedures to establish and retain an optimal beam pair (transmit beam and a corresponding receive beam) for good connectivity. TR 38.802 Section 6.1.6.1 [1 on page 18-577] defines beam management as three procedures:

**Procedure 1 (P-1):** This procedure focuses on the initial acquisition based on synchronization signal blocks (SSB). During the initial acquisition, beam sweeping takes place at both transmit and receive ends to select the best beam pair based on the RSRP measurements. In general, the selected beams are wide and may not be an optimal beam pair for the data transmission and reception. For more details on this procedure, see "NR SSB Beam Sweeping" (5G Toolbox).

**Procedure 2 (P-2):** This procedure focuses on transmit-end beam refinement, where the beam sweeping happens at the transmit end by keeping the receive beam fixed. The procedure is based on non-zero-power CSI-RS (NZP-CSI-RS) for downlink transmit-end beam refinement and sounding reference signal (SRS) for uplink transmit-end beam refinement.

After the initial beam establishment, obtaining a unicast data transmission with high directivity and high gain requires a beam much finer than the SSB beam. Therefore, a set of reference signal resources are configured and transmitted in different directions by using finer beams within the angular range of the beam from the initial acquisition process. Then the user equipment (UE) or the access network node (gNB) measures all these beams by capturing the signals with a fixed receive beam. Finally, the best transmit beam is selected based on the RSRP measurements on all transmit beams.

**Procedure 3 (P-3):** This procedure focuses on receive-end beam adjustment, where the beam sweeping happens at the receive end given the current transmit beam. This process aims to find the best receive beam, which can be a neighbor beam or a refined beam. For this procedure, a set of reference signal resources (NZP-CSI-RS for downlink and SRS for uplink) are transmitted with the same transmit beam and the UE or gNB receives the signal using different beams from different directions covering an angular range. Finally, the best receive beam is selected based on the RSRP measurements on all receive beams.

This example focuses on downlink beam refinement at the transmitter. The example works for both frequency range 1 (FR1) and frequency range 2 (FR2) of NR 5G. This figure depicts the transmit-end beam refinement procedure, considering four NZP-CSI-RS resources transmitted in four different directions.
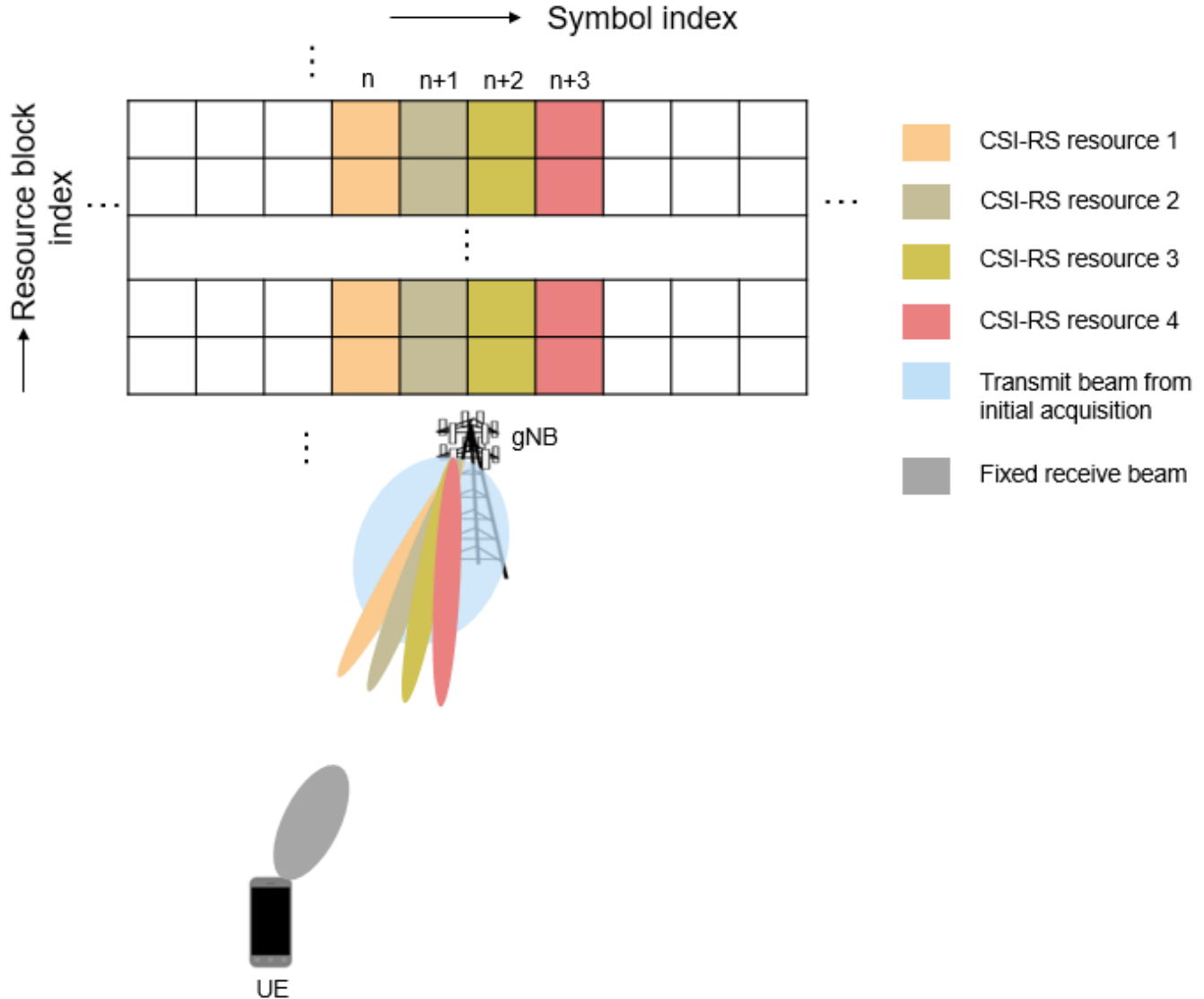
This figure shows the main processing steps of this example with the transmit-end beam refinement process-related steps in color.
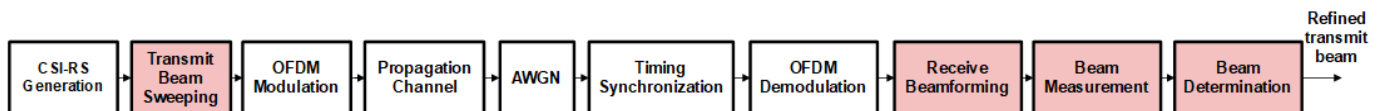


**Generate CSI-RS Resources**

**Configure Carrier**

Create a carrier configuration object representing a 50 MHz carrier with subcarrier spacing of 30 kHz.

```
carrier = nrCarrierConfig;
% Maximum transmission bandwidth configuration for 50 MHz carrier with 30 kHz subcarrier spacing
```

```
carrier.NSizeGrid = 133;
carrier.SubcarrierSpacing = 30;
carrier.NSlot = 0;
carrier.NFrame = 0

carrier =
  nrCarrierConfig with properties:

              NCellID: 1
    SubcarrierSpacing: 30
          CyclicPrefix: 'normal'
            NSizeGrid: 133
           NStartGrid: 0
                NSlot: 0
               NFrame: 0

  Read-only properties:
       SymbolsPerSlot: 14
     SlotsPerSubframe: 2
        SlotsPerFrame: 20
```

**Configure CSI-RS**

Create a CSI-RS configuration object representing an NZP-CSI-RS resource set with `numNZPRes` number of NZP-CSI-RS resources. For Layer 1 RSRP measurements, configure all the CSI-RS resources in a resource set with the same number of antenna ports (either single-port or dual-port), as specified in TS 38.215 Section 5.1.2 [2 on page 18-577] or TS 38.214 Section 5.1.6.1.2 [3 on page 18-577]. This example works for single-port CSI-RS.

```
numNZPRes = 12;
csirs = nrCSIRSConfig;
csirs.CSIRSType = repmat({'nzp'},1,numNZPRes);
csirs.CSIRSPeriod = 'on';
csirs.Density = repmat({'one'},1,numNZPRes);
csirs.RowNumber = repmat(2,1,numNZPRes);
csirs.SymbolLocations = {0,1,2,3,4,5,6,7,8,9,10,11};
csirs.SubcarrierLocations = repmat({0},1,numNZPRes);
csirs.NumRB = 25

csirs =
  nrCSIRSConfig with properties:

                CSIRSType: {'nzp'  'nzp'  'nzp'  'nzp'  'nzp'  'nzp'  'nzp'  'nzp'  'nzp'  'nzp'
              CSIRSPeriod: 'on'
                RowNumber: [2 2 2 2 2 2 2 2 2 2 2 2]
                  Density: {'one'  'one'  'one'  'one'  'one'  'one'  'one'  'one'  'one'  'one'
          SymbolLocations: {[0]  [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]  [10]  [11]}
      SubcarrierLocations: {[0]  [0]  [0]  [0]  [0]  [0]  [0]  [0]  [0]  [0]  [0]  [0]}
                    NumRB: 25
                 RBOffset: 0
                      NID: 0

  Read-only properties:
           NumCSIRSPorts: [1 1 1 1 1 1 1 1 1 1 1 1]
                  CDMType: {'noCDM'  'noCDM'  'noCDM'  'noCDM'  'noCDM'  'noCDM'  'noCDM'  'noCDM'
```

```
% Validate CSI-RS antenna ports
validateCSIRSPorts(csirs);

% Get the binary vector to represent the presence of each CSI-RS resource
% in a specified slot
csirsTransmitted = getActiveCSIRSRes(carrier,csirs);
```

Configure the power scaling of all NZP-CSI-RS resources in decibels (dB).

```
powerCSIRS = 0;
```

### Generate CSI-RS Symbols and Indices

Generate CSI-RS symbols and indices by using the `carrier` and `csirs` configuration objects. To distinguish each CSI-RS resource output separately, specify the `OutputResourceFormat,'cell'` name-value pair.

```
csirsSym = nrCSIRS(carrier,csirs,'OutputResourceFormat','cell')
```

```
csirsSym=1×12 cell array
    {25x1 double}    {25x1 double}    {25x1 double}    {25x1 double}    {25x1 double}    {25x1 d
```

```
csirsInd = nrCSIRSIndices(carrier,csirs,'OutputResourceFormat','cell')
```

```
csirsInd=1×12 cell array
    {25x1 uint32}    {25x1 uint32}    {25x1 uint32}    {25x1 uint32}    {25x1 uint32}    {25x1 u
```

### Configure Antenna Arrays and Scatterers

### Configure Transmit and Receive Antenna Arrays

Configure the carrier frequency and the signal propagation speed.

```
% Set the carrier frequency
fc = 3.5e9;
freqRange = validateFc(fc);

% Set the propagation speed
c = physconst('LightSpeed');
% Calculate wavelength
lambda = c/fc;
```

Configure the size of transmit and receive antenna arrays as a two-element vector, where the first element represents the number of rows and the second element represents the number of columns in the antenna array.

```
txArySize = [8 8];
rxArySize = [2 2];
```

Calculate the total number of transmit and receive antenna elements.

```
nTx = prod(txArySize);
nRx = prod(rxArySize);
```

Configure the positions of transmit and receive antenna arrays. Then calculate the free space path loss based on the spatial separation between transmit and receive antenna array positions.

```
% Configure antenna array positions
txArrayPos = [0;0;0];
rxArrayPos = [100;50;0];

% Calculate the free space path loss
toRxRange = rangeangle(txArrayPos,rxArrayPos);
spLoss = fspl(toRxRange,lambda);
```

Configure the uniform linear array (ULA) or uniform rectangular array (URA) based on the sizes of antennas arrays.

```
% Initialize the flags to choose between URA and ULA
isTxRectArray = false;
isRxRectArray = false;

% Enable isTxRectArray if both the number of rows and columns of transmit
% antenna array are greater than one
if ~any(txArySize == 1)
    isTxRectArray = true;
end
% Enable isRxRectArray if both the number of rows and columns of receive
% antenna array are greater than one
if ~any(rxArySize == 1)
    isRxRectArray = true;
end

% Configure the transmit and receive antenna elements
txAntenna = phased.IsotropicAntennaElement('BackBaffled',true);  % To avoid transmission beyond -
                                                                  % degrees from the broadside, ba
                                                                  % the back of the transmit anten
                                                                  % element by setting the BackBaf
                                                                  % property to true
rxAntenna = phased.IsotropicAntennaElement('BackBaffled',false); % To receive the signal from 360
                                                                  % set the BackBaffled property t

% Configure transmit antenna array
if isTxRectArray
    % Create a URA System object for signal transmission
    txArray = phased.URA('Element',txAntenna,'Size',txArySize,'ElementSpacing',lambda/2);
else
    % Create a ULA System object for signal transmission
    txArray = phased.ULA('Element',txAntenna,'NumElements',nTx,'ElementSpacing',lambda/2);
end

% Configure receive antenna array
if isRxRectArray
    % Create a URA System object for signal reception
    rxArray = phased.URA('Element',rxAntenna,'Size',rxArySize,'ElementSpacing',lambda/2);
else
    % Create a ULA System object for signal reception
    rxArray = phased.ULA('Element',rxAntenna,'NumElements',nRx,'ElementSpacing',lambda/2);
end
```

**Configure Scatterers**

```
fixedScatMode = true;
rng(42);
if fixedScatMode
```

```
    % Fixed single scatterer location
    numScat = 1;
    scatPos = [60;10;15];
else
    % Generate scatterers at random positions
    numScat = 10; %#ok<UNRCH>
    azRange = -180:180;
    randAzOrder = randperm(length(azRange));
    elRange = -90:90;
    randElOrder = randperm(length(elRange));
    azAngInSph = deg2rad(azRange(randAzOrder(1:numScat)));
    elAngInSph = deg2rad(elRange(randElOrder(1:numScat)));
    r = 20;

    % Transform spherical coordinates to Cartesian coordinates
    [x,y,z] = sph2cart(azAngInSph,elAngInSph,r);
    scatPos = [x;y;z] + (txArrayPos + rxArrayPos)/2;
end
```

**Transmit Beamforming and OFDM Modulation**

**Calculate the Steering Vectors**

Create the steering vector System object™ for transmit antenna array.

```
txArrayStv = phased.SteeringVector('SensorArray',txArray,'PropagationSpeed',c);
```

Calculate the angle of scatterer position with respect to the transmit antenna array.

```
[~,scatAng] = rangeangle(scatPos(:,1),txArrayPos); % Pointing towards the first scatterer positio
```

Configure the azimuth and elevation beamwidths of SSB transmit beam from the initial acquisition process (P-1).

```
azTxBeamWidth = 30; % In degrees
elTxBeamWidth = 30; % In degrees
```

Get the SSB transmit beam direction which is aligned (partially or fully) to the position of scatterer, by using the beamwidths in azimuth and elevation planes.

```
ssbTxAng = getInitialBeamDir(scatAng,azTxBeamWidth,elTxBeamWidth);
```

Calculate the beam directions (azimuth and elevation angle pairs) for all active CSI-RS resources within the angular range covered by the SSB transmit beam.

```
% Get the number of transmit beams based on the number of active CSI-RS resources in a slot
numBeams = sum(csirsTransmitted);

% Get the azimuthal sweep range based on the SSB transmit beam direction
% and its beamwidth in azimuth plane
azSweepRange = [ssbTxAng(1) - azTxBeamWidth/2 ssbTxAng(1) + azTxBeamWidth/2];

% Get the elevation sweep range based on the SSB transmit beam direction
% and its beamwidth in elevation plane
elSweepRange = [ssbTxAng(2) - elTxBeamWidth/2 ssbTxAng(2) + elTxBeamWidth/2];

% Get the azimuth and elevation angle pairs for all NZP-CSI-RS transmit beams
azBW = beamwidth(txArray,fc,'Cut','Azimuth');
```

```
elBW = beamwidth(txArray,fc,'Cut','Elevation');
csirsBeamAng = hGetBeamSweepAngles(numBeams,azSweepRange,elSweepRange,azBW,elBW);
```

Calculate the steering vectors for all active CSI-RS resources.

```
wT = zeros(nTx,numBeams);
for beamIdx = 1:numBeams
    tempW = txArrayStv(fc,csirsBeamAng(:,beamIdx));
    wT(:,beamIdx) = tempW;
end
```

**Apply Digital Beamforming**

Loop over all NZP-CSI-RS resources and apply the digital beamforming to all the active ones. Digital beamforming is considered to offer frequency selective beamforming within the same OFDM symbol.

```
% Number of CSI-RS antenna ports
ports = csirs.NumCSIRSPorts(1);
% Initialize the beamformed grid
bfGrid = nrResourceGrid(carrier,nTx);
% Get the active NZP-CSI-RS resource indices
activeRes = find(logical(csirsTransmitted));
for resIdx = 1:numNZPRes
    % Initialize the carrier resource grid for one slot and map NZP-CSI-RS symbols onto
    % the grid
    txSlotGrid = nrResourceGrid(carrier,ports);
    txSlotGrid(csirsInd{resIdx}) = db2mag(powerCSIRS)*csirsSym{resIdx};
    reshapedSymb = reshape(txSlotGrid,[],ports);

    % Get the transmit beam index
    beamIdx = find(activeRes == resIdx);

    % Apply the digital beamforming
    if ~isempty(beamIdx)
        bfSymb = reshapedSymb * wT(:,beamIdx)';
        bfGrid = bfGrid + reshape(bfSymb,size(bfGrid));
    end
end
```

**Perform OFDM Modulation**

Generate the time-domain waveform by performing the OFDM modulation.

```
% Perform OFDM modulation
[tbfWaveform,ofdmInfo] = nrOFDMModulate(carrier,bfGrid);

% Normalize the beamformed time-domain waveform over the number of transmit
% antennas
tbfWaveform = tbfWaveform/sqrt(nTx);
```

**Scattering MIMO Channel and AWGN**

**Configure the Channel**

Configure the scattering-based MIMO propagation channel by using the System object `phased.ScatteringMIMOChannel`. This channel model applies time delay, gain, Doppler shift, phase change, free space path loss, and optionally, other atmospheric attenuations to the input.

```
chan = phased.ScatteringMIMOChannel;
chan.PropagationSpeed = c;
chan.CarrierFrequency = fc;
chan.Polarization = 'none';
chan.SpecifyAtmosphere = false;
chan.SampleRate = ofdmInfo.SampleRate;
chan.SimulateDirectPath = false;
chan.ChannelResponseOutputPort = true;

% Configure transmit array parameters
chan.TransmitArray = txArray;
chan.TransmitArrayMotionSource = 'property';
chan.TransmitArrayPosition = txArrayPos;

% Configure receive array parameters
chan.ReceiveArray = rxArray;
chan.ReceiveArrayMotionSource = 'property';
chan.ReceiveArrayPosition = rxArrayPos;

% Configure scatterers
chan.ScattererSpecificationSource = 'property';
chan.ScattererPosition = scatPos;
chan.ScattererCoefficient = ones(1,numScat);

% Get the maximum channel delay by transmitting random signal
[~,~,tau] = chan(complex(randn(chan.SampleRate*1e-3,nTx), ...
    randn(chan.SampleRate*1e-3,nTx)));
maxChDelay = ceil(max(tau)*chan.SampleRate);
```

**Send the Waveform through the Channel**

Append zeros at the end of the transmitted waveform to flush the channel content and then pass the time-domain waveform through the scattering MIMO channel. These zeros take into account any delay introduced in the channel.

```
% Append zeros to the transmit waveform to account for channel delay
tbfWaveform = [tbfWaveform; zeros(maxChDelay,nTx)];
% Pass the waveform through the channel
fadWave = chan(tbfWaveform);
```

**Apply AWGN**

Configure and apply the receive gain to the faded waveform, to compensate for the path loss. Then apply AWGN to the resultant waveform. For an explanation of the SNR definition that this example uses, see "SNR Definition Used in Link Simulations" (5G Toolbox).

```
% Configure the receive gain
rxGain = 10.^((spLoss)/20); % Gain in linear scale
% Apply the gain
fadWaveG = fadWave*rxGain;

% Configure the SNR in dB
SNRdB = 20;
SNR = 10^(SNRdB/10); % SNR in linear scale
% Calculate the standard deviation for AWGN
N0 = 1/sqrt(2.0*nRx*double(ofdmInfo.Nfft)*SNR);

% Generate AWGN
```

```
noise = N0*complex(randn(size(fadWaveG)),randn(size(fadWaveG)));
% Apply AWGN to the waveform
rxWaveform = fadWaveG + noise;
```

**Timing Synchronization**

Perform the timing synchronization by cross correlating the received reference symbols with a local copy of NZP-CSI-RS symbols.

```
% Generate reference symbols and indices
refSym = nrCSIRS(carrier,csirs);
refInd = nrCSIRSIndices(carrier,csirs);

% Estimate timing offset
offset = nrTimingEstimate(carrier,rxWaveform,refInd,refSym);
if offset > maxChDelay
    offset = 0;
end

% Correct timing offset
syncTdWaveform = rxWaveform(1+offset:end,:);
```

**OFDM Demodulation and Receive Beamforming**

**OFDM Demodulation**

OFDM demodulate the synchronized time-domain waveform.

```
rxGrid = nrOFDMDemodulate(carrier,syncTdWaveform);
```

**Calculate Steering Vector**

Create the steering vector System object for the receive antenna array.

```
rxArrayStv = phased.SteeringVector('SensorArray',rxArray,'PropagationSpeed',c);
```

Calculate the angle of scatterer position with respect to the receive antenna array. Assuming this as receive beam direction from the initial acquisition process using SSB.

```
[~,scatRxAng] = rangeangle(scatPos(:,1),rxArrayPos); % Pointing towards the first scatterer posi
```

Configure the azimuth and elevation beamwidths of receive beam from the initial acquisition process (P-1).

```
azRxBeamWidth = 30; % In degrees
elRxBeamWidth = 30; % In degrees
```

Get the initial receive beam direction which is aligned (partially or fully) to the position of scatterer, by using the beamwidths in azimuth and elevation planes from P-1.

```
rxAng = getInitialBeamDir(scatRxAng,azRxBeamWidth,elRxBeamWidth);
```

Calculate the steering vector for the angle of reception.

```
wR = rxArrayStv(fc,rxAng);
```

**Apply Receive Beamforming**

To perform digital beamforming at the receiver side, apply the steering weights to `rxGrid`, with the assumption that there is no other signal present in `rxGrid` (single UE scenario). Combine the signals

from all receive antenna elements in case of FR2, as specified in TS 38.215 Section 5.1.2 [2 on page 18-577].

```
temp = rxGrid;
if strcmpi(freqRange,'FR1')
    % Beamforming without combining
    rbfGrid = reshape(reshape(temp,[],nRx).*wR',size(temp,1),size(temp,2),[]);
else % 'FR2'
    % Beamforming with combining
    rbfGrid = reshape(reshape(temp,[],nRx)*conj(wR),size(temp,1),size(temp,2),[]);
end
```

**Plot the Scattering MIMO Scenario**

Configure the MIMO scene parameters.

```
sceneParams.TxArray = txArray;
sceneParams.RxArray = rxArray;
sceneParams.TxArrayPos = txArrayPos;
sceneParams.RxArrayPos = rxArrayPos;
sceneParams.ScatterersPos = scatPos;
sceneParams.Lambda = lambda;
sceneParams.ArrayScaling = 100;    % To enlarge antenna arrays in the plot
sceneParams.MaxTxBeamLength = 45; % Maximum length of transmit beams in the plot
sceneParams.MaxRxBeamLength = 25; % Maximum length of receive beam in the plot
```

Plot the scattering MIMO scenario (including transmit and receive antenna arrays, scatterer positions and their paths, and all the transmit and receive antenna array beam patterns) by using the helper function `hPlotSpatialMIMOScene`. Beam patterns in this figure resemble the power patterns in linear scale.

```
hPlotSpatialMIMOScene(sceneParams,wT,wR);
axis tight;
view([74 29]);
```
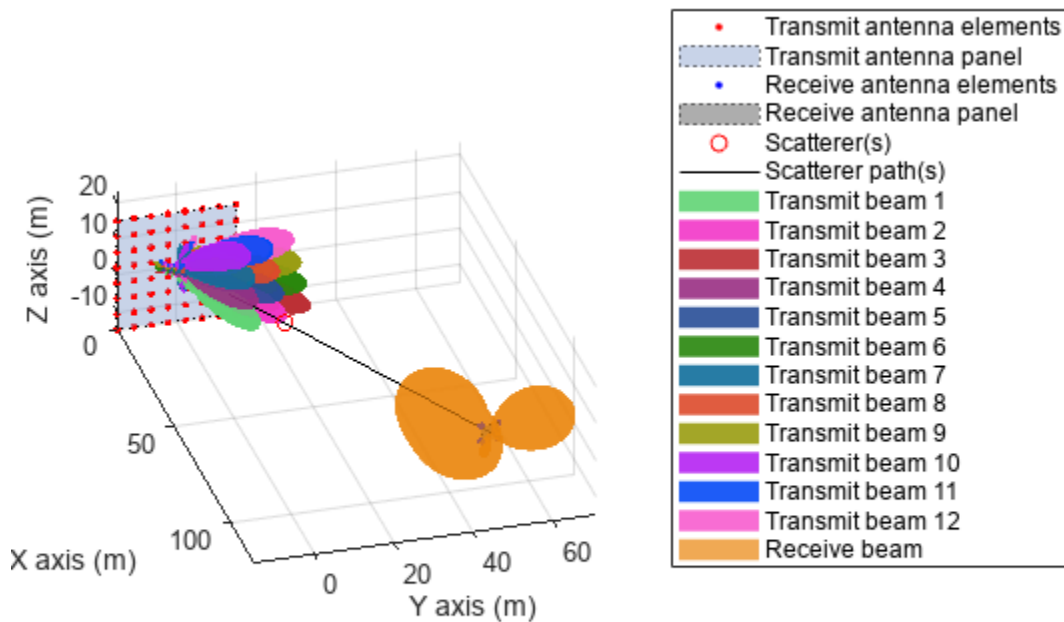
**Beam Determination**

After the OFDM demodulation, the UE measures the RSRP for all the CSI-RS resources transmitted in different beams, given the current receive beam. Perform these measurements by using the nrCSIRSMeasurements function.

```
% Perform RSRP measurements
meas = nrCSIRSMeasurements(carrier,csirs,rbfGrid);

% Display the measurement quantities for all CSI-RS resources in dBm
RSRPdBm = max(meas.RSRPPerAntenna,[],1);
disp(['RSRP measurements of all CSI-RS resources (in dBm):' 13 num2str(RSRPdBm)]);

RSRP measurements of all CSI-RS resources (in dBm):
42.3147       33.237     29.1999      45.1102      37.0922      31.4861      39.9414      33.505
```

Identify the maximum RSRP value from the measurements and find the best corresponding beam.

```
% Get the transmit beam index with maximum RSRP value
[~,maxRSRPIdx] = max(RSRPdBm(logical(csirsTransmitted)));

% Get the CSI-RS resource index with maximum RSRP value
[~,maxRSRPResIdx] = max(RSRPdBm);
```

Calculate the beamwidth which corresponds to the refined transmit beam.

```
% Get the steering weights corresponding to refined transmit beam
if numBeams == 0
```

```
        disp('Refinement has not happened, as NZP-CSI-RS is not transmitted')
else
        refBeamWts = wT(:,maxRSRPIdx);
        csirsAzBeamWidth = beamwidth(txArray,fc,'PropagationSpeed',c,'Weights',refBeamWts,'CutAngle'
        csirsElBeamWidth = beamwidth(txArray,fc,'PropagationSpeed',c,'Weights',refBeamWts,'Cut','Elev
        disp(['From initial beam acquisition:' 13 '     Beamwidth of initial SSB beam in azimuth plane
            num2str(azTxBeamWidth) ' degrees' 13 ...
            '     Beamwidth of initial SSB beam in elevation plane is: '...
            num2str(elTxBeamWidth) ' degrees' 13 13 ...
            'With transmit-end beam refinement:' 13 '     Refined transmit beam ('...
            num2str(maxRSRPIdx) ') corresponds to CSI-RS resource '...
            num2str(maxRSRPResIdx) ' is selected in the direction ['...
            num2str(csirsBeamAng(1,maxRSRPIdx)) ';' num2str(csirsBeamAng(2,maxRSRPIdx))...
            ']' 13 '     Beamwidth of refined transmit beam in azimuth plane is: '...
            num2str(csirsAzBeamWidth) ' degrees' 13 ...
            '     Beamwidth of refined transmit beam in elevation plane is: '...
            num2str(csirsElBeamWidth) ' degrees']);
end
```

```
From initial beam acquisition:
    Beamwidth of initial SSB beam in azimuth plane is: 30 degrees
    Beamwidth of initial SSB beam in elevation plane is: 30 degrees

With transmit-end beam refinement:
    Refined transmit beam (4) corresponds to CSI-RS resource 4 is selected in the direction [10;1
    Beamwidth of refined transmit beam in azimuth plane is: 13.46 degrees
    Beamwidth of refined transmit beam in elevation plane is: 13.24 degrees
```

**Summary and Further Exploration**

This example highlights the beam refinement procedure (P-2) using NZP-CSI-RS. The procedure identifies a transmit beam that is finer than the beam from the initial acquisition.

You can configure multiple CSI-RS resources, transmit and receive antenna array configurations, and multiple scatterers to see the variations in the selection of the refined beam. You can also configure the azimuth and elevation angle pairs for the signal transmission and reception.

**References**

**1**  3GPP TR 38.802. "Study on New Radio access technology physical layer aspects." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

**2**  3GPP TS 38.215. "NR; Physical layer measurements." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

**3**  3GPP TS 38.214. "NR; Physical layer procedures for data." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

**Local Functions**

```
function validateCSIRSPorts(csirs)
%   validateCSIRSPorts validates the CSI-RS antenna ports, given the
%   CSI-RS configuration object CSIRS.

    numPorts = csirs.NumCSIRSPorts;
    if any(numPorts > 1)
        error('nr5g:PortsGreaterThan1','CSI-RS resources must be configured for single-port for F
```

```matlab
        end
    end

    function csirsTransmitted = getActiveCSIRSRes(carrier,csirs)
    %   getActiveCSIRSRes returns a binary vector indicating the presence of
    %   all CSI-RS resources in a specified slot, given the carrier
    %   configuration object CARRIER and CSI-RS configuration object CSIRS.

        % Extract the following properties of carrier
        NSlotA       = carrier.NSlot;        % Absolute slot number
        NFrameA      = carrier.NFrame;       % Absolute frame number
        SlotsPerFrame = carrier.SlotsPerFrame;  % Number of slots per frame

        % Calculate the appropriate frame number (0...1023) based on the
        % absolute slot number
        NFrameR = mod(NFrameA + fix(NSlotA/SlotsPerFrame),1024);
        % Relative slot number (0...slotsPerFrame-1)
        NSlotR = mod(NSlotA,SlotsPerFrame);

        % Loop over the number of CSI-RS resources
        numCSIRSRes = numel(csirs.CSIRSType);
        csirsTransmitted = zeros(1,numCSIRSRes);
        csirs_struct = validateConfig(csirs);
        for resIdx = 1:numCSIRSRes
            % Extract the CSI-RS slot periodicity and offset
            if isnumeric(csirs_struct.CSIRSPeriod{resIdx})
                Tcsi_rs = csirs_struct.CSIRSPeriod{resIdx}(1);
                Toffset = csirs_struct.CSIRSPeriod{resIdx}(2);
            else
                if strcmpi(csirs_struct.CSIRSPeriod{resIdx},'on')
                    Tcsi_rs = 1;
                else
                    Tcsi_rs = 0;
                end
                Toffset = 0;
            end

            % Check for the presence of CSI-RS, based on slot periodicity and offset
            if (Tcsi_rs ~= 0) && (mod(SlotsPerFrame*NFrameR + NSlotR - Toffset, Tcsi_rs) == 0)
                csirsTransmitted(resIdx) = 1;
            end
        end
    end

    function freqRange = validateFc(fc)
    %   validateFc validates the carrier frequency FC and returns the frequency
    %   range as either 'FR1' or 'FR2'.

        if fc >= 410e6 && fc <= 7.125e9
            freqRange = 'FR1';
        elseif fc >= 24.25e9 && fc <= 52.6e9
            freqRange = 'FR2';
        else
            error('nr5g:invalidFreq',['Selected carrier frequency is outside '...
                'FR1 (410 MHz to 7.125 GHz) and FR2 (24.25 GHz to 52.6 GHz).']);
        end
    end
```

```matlab
function beamDir = getInitialBeamDir(scatAng,azBeamWidth,elBeamWidth)
%   getInitialBeamDir returns the initial beam direction BEAMDIR, given the
%   angle of scatterer position with respect to transmit or receive antenna
%   array SCATANG, beamwidth of transmit or receive beam in azimuth plane
%   AZBEAMWIDTH, and beamwidth of transmit or receive beam in elevation
%   plane ELBEAMWIDTH.

    % Azimuth angle boundaries of all transmit/receive beams
    azSSBSweep = -180:azBeamWidth:180;
    % Elevation angle boundaries of all transmit/receive beams
    elSSBSweep = -90:elBeamWidth:90;

    % Get the azimuth angle of transmit/receive beam
    azIdx1 = find(azSSBSweep <= scatAng(1),1,'last');
    azIdx2 = find(azSSBSweep >= scatAng(1),1,'first');
    azAng = (azSSBSweep(azIdx1) + azSSBSweep(azIdx2))/2;

    % Get the elevation angle of transmit/receive beam
    elIdx1 = find(elSSBSweep <= scatAng(2),1,'last');
    elIdx2 = find(elSSBSweep >= scatAng(2),1,'first');
    elAng = (elSSBSweep(elIdx1) + elSSBSweep(elIdx2))/2;

    % Form the azimuth and elevation angle pair (in the form of [az;el])
    % for transmit/receive beam
    beamDir = [azAng;elAng];
end
```

# CDL Channel Model Customization with Ray Tracing

This example shows how to customize the CDL channel model parameters by using the output of a ray tracing analysis. The example shows how to:

- Specify the location of a transmitter and a receiver in a 3D environment.
- Use ray tracing to calculate the geometric aspects of the channel: number of rays, angles, delays and attenuations.
- Configure the CDL channel model with the result of ray tracing analysis.
- Specify the channel antenna arrays using Phased Array System Toolbox™.
- Visualize the transmit and receive array radiation patterns based on singular value decomposition of a perfect channel estimate.

**Base Station and UE Configuration**

The example assumes that both the base station and the UE use rectangular arrays. The array orientations are specified as a pair of values representing azimuth and elevation. Both angles are in degrees.

```
fc = 6e9;                                 % carrier frequency (Hz)
bsPosition = [22.287495, 114.140706];     % lat, lon
bsAntSize = [8 8];                        % number of rows and columns in rectangular array (base sta
bsArrayOrientation = [-30 0].';           % azimuth (0 deg is East, 90 deg is North) and elevation (
uePosition = [22.287323,114.140859];      % lat, lon
ueAntSize = [2 2];                        % number of rows and columns in rectangular array (UE).
ueArrayOrientation = [180 45].';          % azimuth (0 deg is East, 90 deg is North) and elevation (
reflectionsOrder = 1;                     % number of reflections for ray tracing analysis (0 for LOS

% Bandwidth configuration, required to set the channel sampling rate and for perfect channel esti
SCS = 15; % subcarrier spacing
NRB = 52; % number of resource blocks, 10 MHz bandwidth
```

**Import and Visualize 3-D Environment with Buildings for Ray Tracing**

Launch Site Viewer with buildings in Hong Kong. For more information about the osm file, see [1] on page 18-588.

```
if exist('viewer','var') && isvalid(viewer) % viewer handle exists and viewer window is open
    viewer.clearMap();
else
    viewer = siteviewer("Basemap","openstreetmap","Buildings","hongkong.osm");
end
```
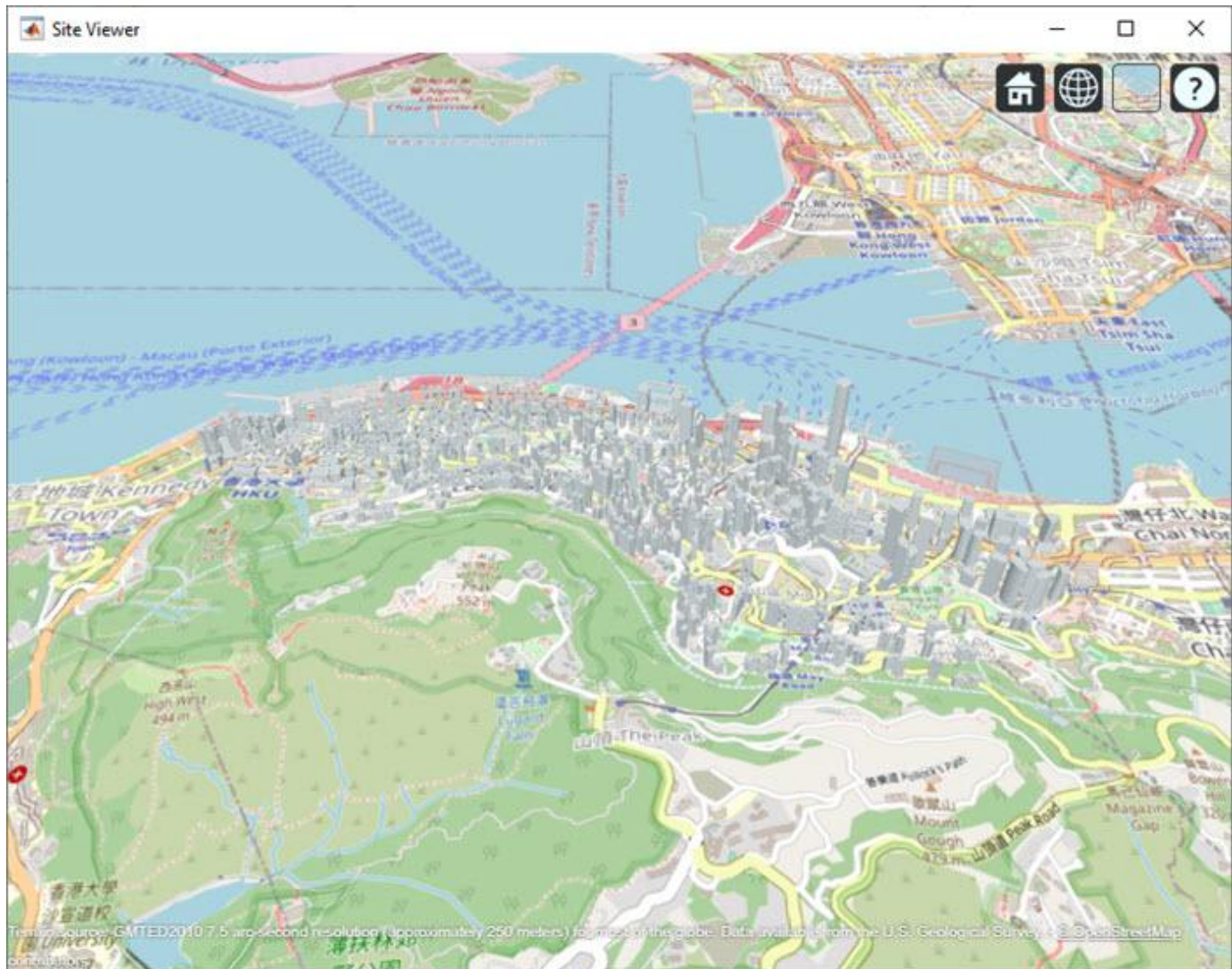
**Create Base Station and UE**

Locate the base station and the UE on the map.

```
bsSite = txsite("Name","Base station", ...
    "Latitude",bsPosition(1),"Longitude",bsPosition(2),...
    "AntennaAngle",bsArrayOrientation(1:2),...
    "AntennaHeight",4,...  % in m
    "TransmitterFrequency",fc);

ueSite = rxsite("Name","UE", ...
    "Latitude",uePosition(1),"Longitude",uePosition(2),...
    "AntennaHeight",1,... % in m
    "AntennaAngle",ueArrayOrientation(1:2));
```

Visualize the location of the base station and the UE. The base station is on top of a building.

```
bsSite.show();
ueSite.show();
```
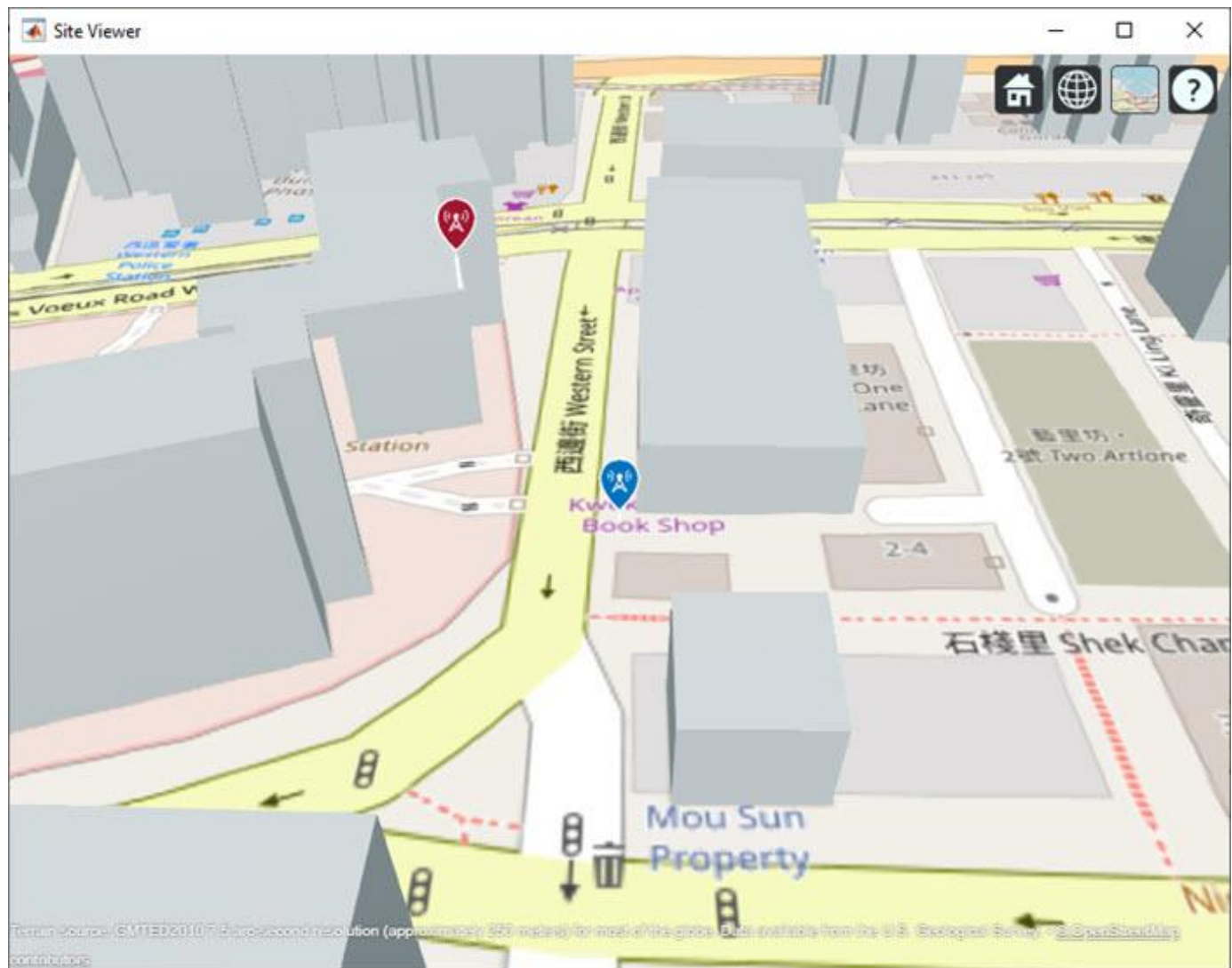
**Ray Tracing Analysis**

Perform ray tracing analysis using the shooting and bouncing rays (SBR) method. The SBR method includes effects from surface reflections and diffractions but does not include effects from refraction or scattering.

```
pm = propagationModel("raytracing","Method","sbr","MaxNumReflections",reflectionsOrder);
rays = raytrace(bsSite,ueSite,pm,"Type","pathloss");
```

Display the rays in Site Viewer.

```
plot(rays{1})
```
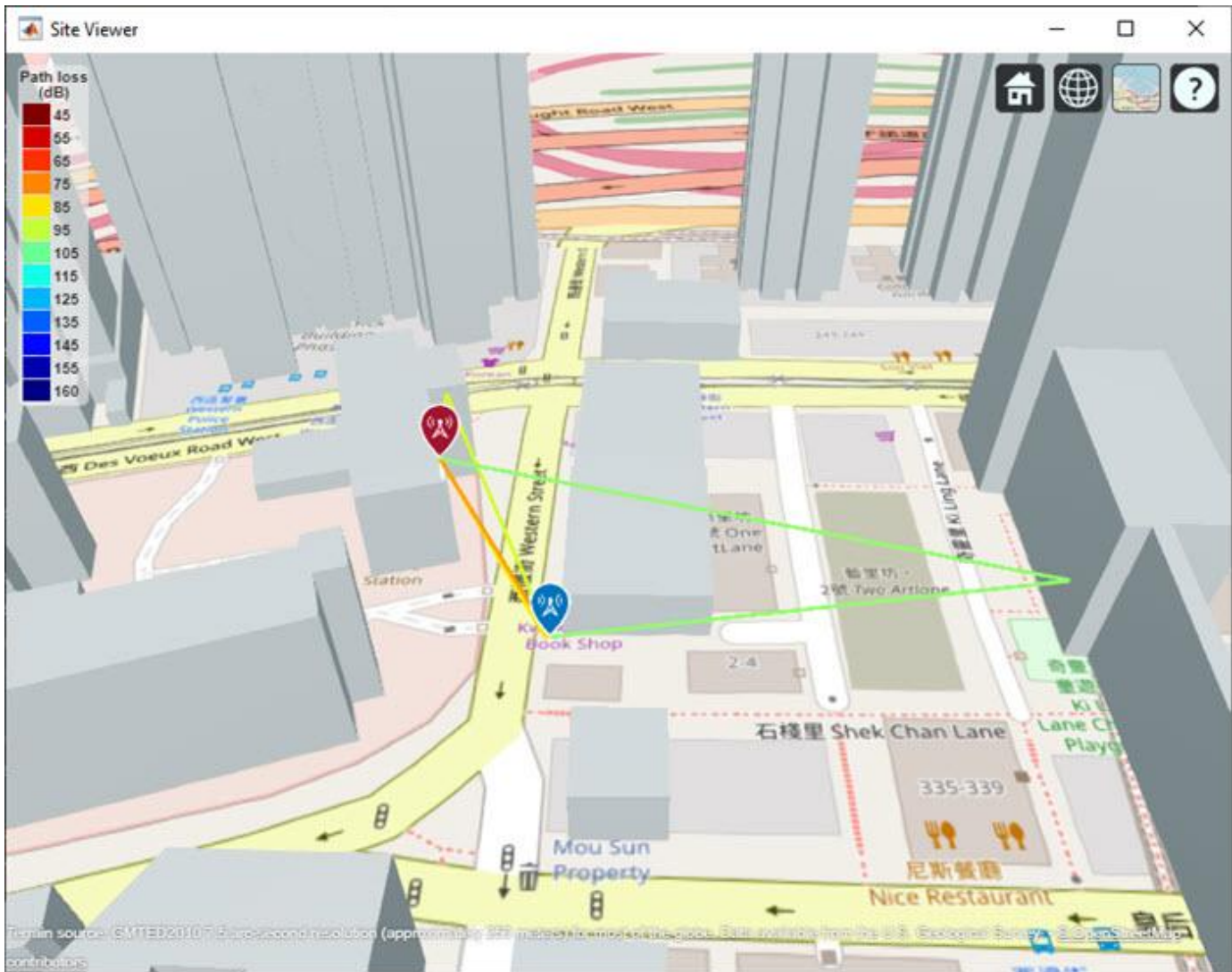
From the obtained rays, get the times of arrival, the average path gains, and the angles of departure and arrival. For simplicity, normalize the propagation delay so that the first path occurs at time 0 sec, corresponding to no delay. Use the path loss to obtain the average path gains.

```
pathToAs = [rays{1}.PropagationDelay]-min([rays{1}.PropagationDelay]);   % Time of arrival of each
avgPathGains  = -[rays{1}.PathLoss];                                     % Average path gains of e
pathAoDs = [rays{1}.AngleOfDeparture];                                   % AoD of each ray
pathAoAs = [rays{1}.AngleOfArrival];                                     % AoA of each ray
isLOS = any([rays{1}.LineOfSight]);                                      % Line of sight flag
```

**Set Up CDL Channel Model**

Configure the CDL channel model with the information generated by the ray tracing analysis. Set the DelayProfile property to 'Custom' to specify the path delays, the average path gains and the angles of arrival and departure (both in azimuth and zenith).

When configuring the channel model, take into account that:

- The ray tracer finds individual rays between the base station and the UE, while the CDL channel models clusters of rays whose properties are determined by the cluster average path gain (`AveragePathGains`), average angles of arrival and departure (`AnglesAoA`, `AnglesZoA`, `AnglesAoD`, and `AnglesZoD`), and the spread of the rays in the cluster (`AngleSpreads`). The information retrieved from the ray tracing analysis for individual rays configures the cluster average values of the CDL channel.

- The ray tracer performs a static analysis, while the CDL channel models UE movement. Therefore, the CDL channel introduces small-scale fading.

- The path gains obtained from ray tracing are considered as average path gains. Therefore, because of fading, instantaneous channel path gains will differ from the average values, but over long simulations their mean value will match the specified average path gains when using isotropic antennas.

- The CDL channel uses zenith angles, while the ray tracing analysis returns elevation angles, therefore you must convert between the two.

- If any of the calculated rays is a line of sight (LOS) ray (no reflection), set the `HasLOSCluster` CDL channel property to `true`. For LOS cases, the CDL model splits the first path into two components, one being LOS and the other having a Rayleigh fading characteristic. This results in a combined Rician fading characteristic. Therefore, in LOS cases, when you specify *N* rays, the CDL channel models *N*+1 paths internally.

```
channel = nrCDLChannel;
channel.DelayProfile = 'Custom';
channel.PathDelays = pathToAs;
channel.AveragePathGains = avgPathGains;
channel.AnglesAoD = pathAoDs(1,:);          % azimuth of departure
channel.AnglesZoD = 90-pathAoDs(2,:);       % channel uses zenith angle, rays use elevation
channel.AnglesAoA = pathAoAs(1,:);          % azimuth of arrival
channel.AnglesZoA = 90-pathAoAs(2,:);       % channel uses zenith angle, rays use elevation
channel.HasLOSCluster = isLOS;
channel.CarrierFrequency = fc;
channel.NormalizeChannelOutputs = false; % do not normalize by the number of receive antennas, th
channel.NormalizePathGains = false;      % set to false to retain the path gains
```

Specify the channel antenna arrays by using Phased Array System Toolbox array objects. The array orientation properties of the CDL channel model use azimuth and downtilt while the `ueArrayOrientation` and `bsArrayOrientation` objects use azimuth and elevation. Therefore, convert the elevation to downtilt by changing the sign.

```
c = physconst('LightSpeed');
lambda = c/fc;

% UE array (single panel)
ueArray = phased.NRRectangularPanelArray('Size',[ueAntSize(1:2) 1 1],'Spacing', [0.5*lambda*[1 1]
ueArray.ElementSet = {phased.IsotropicAntennaElement};    % isotropic antenna element
channel.ReceiveAntennaArray = ueArray;
channel.ReceiveArrayOrientation = [ueArrayOrientation(1); (-1)*ueArrayOrientation(2); 0];  % the

% Base station array (single panel)
bsArray = phased.NRRectangularPanelArray('Size',[bsAntSize(1:2) 1 1],'Spacing', [0.5*lambda*[1 1]
bsArray.ElementSet = {phased.NRAntennaElement('PolarizationAngle',-45) phased.NRAntennaElement('I
channel.TransmitAntennaArray = bsArray;
channel.TransmitArrayOrientation = [bsArrayOrientation(1); (-1)*bsArrayOrientation(2); 0];   % th
```

**Set Channel Sampling Rate**

The signal going through the channel determines the channel sampling rate. Consider a signal with subcarrier spacing of 15 kHz and 52 resource blocks (RBs), equivalent to a bandwidth of 10 MHz. To obtain the sampling rate, call the `nrOFDMInfo` function.

```
ofdmInfo = nrOFDMInfo(NRB,SCS);

channel.SampleRate = ofdmInfo.SampleRate;
```

**Channel Estimation**

For simplicity, this example assumes perfect channel estimation. Setting the `ChannelFiltering` property to `false` allows you to get the channel path gains without sending a signal through the channel.

```
channel.ChannelFiltering = false;
[pathGains,sampleTimes] = channel();
```

Plot the path gains returned by the channel. Compare the results with the specified average path gains obtained from the ray attenuation values.

- For LOS cases, because the first two paths correspond to the first ray, the first two paths must be added together.
- The CDL channel model is a statistical channel model and considers UE motion. Therefore, the returned path gains are the instantaneous gains. The path gains from the ray tracing analysis are interpreted as average path gains by the channel model.
- The instantaneous path gains returned by the channel model include the gain of the antenna element in the direction of each ray. The custom path gains obtained from the ray tracing analysis do not include the antenna element gain. Therefore, in the mean, the channel path gains match the average gains only for isotropic antenna elements.

```
pg=permute(pathGains,[2 1 3 4]); % first dimension is the number of paths
if isLOS
    % in LOS cases sum the first to paths, they correspond to the LOS ray
    pg = [sum(pg(1:2,:,:,:)); pg(3:end,:,:,:)];
end
pg = abs(pg).^2;
plot(pow2db(pg(:,1,1,1)),'o-.');hold on
plot(avgPathGains,'x-.');hold off
legend("Instantaneous (1^{st} tx - 1^{st} rx antenna)","Average (from ray tracing)")
xlabel("Path number"); ylabel("Gain (dB)")
title('Path gains')
```
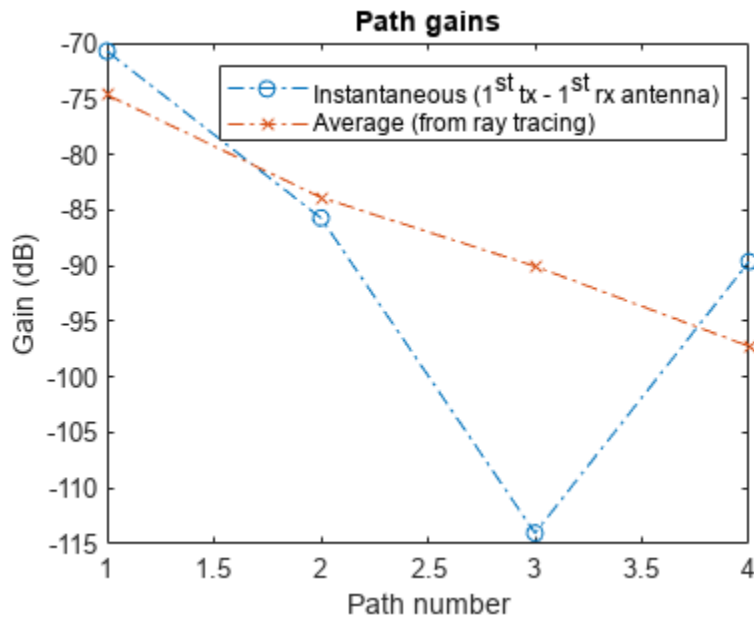
Obtain a perfect channel estimate for slot 0.

```
pathFilters = getPathFilters(channel);
nSlot = 0;
[offset,~] = nrPerfectTimingEstimate(pathGains,pathFilters);
hest = nrPerfectChannelEstimate(pathGains,pathFilters,NRB,SCS,nSlot,offset,sampleTimes);
```

Plot the channel response in time and frequency between the first transmit and the first receive antenna. This plot shows how the channel behaves in time and frequency. For low Doppler shifts, the channel doesn't change much during the observation period of one slot.

```
surf(pow2db(abs(hest(:,:,1,1)).^2));
shading('flat');
xlabel('OFDM Symbols');ylabel('Subcarriers');zlabel('Magnitude Squared (dB)');
title('Channel Magnitude Response (1^{st} tx - 1^{st} rx antenna)');
```
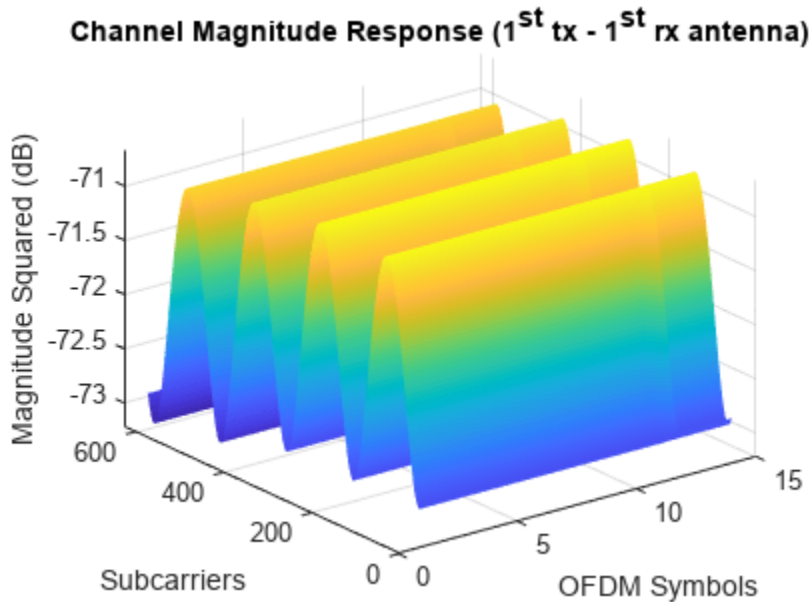
**Get Beamforming Weights**

Calculate the beamforming weights using singular value decomposition (SVD). Assume 1 layer. The `getBeamformingWeights` function averages the channel conditions over a number of resource blocks, starting at an offset from the edge of the band (first subcarrier), enabling subband beamforming.

```
nLayers = 1;
scOffset = 0;    % no offset
noRBs = 1;       % average channel conditions over 1 RB to calculate beamforming weights
[wbs,wue,~] = getBeamformingWeights(hest,nLayers,scOffset,noRBs);
```

**Plot Radiation Patterns**

Plot the radiation patterns obtained for the UE and the base station.

```
% Plot UE radiation pattern
ueSite.Antenna = clone(channel.ReceiveAntennaArray); % need a clone, otherwise setting the Taper
ueSite.Antenna.Taper = wue;
pattern(ueSite,fc,"Size",4);

% Plot BS radiation pattern
bsSite.Antenna = clone(channel.TransmitAntennaArray); % need a clone, otherwise setting the Taper
bsSite.Antenna.Taper = wbs;
pattern(bsSite,fc,"Size",5);
```
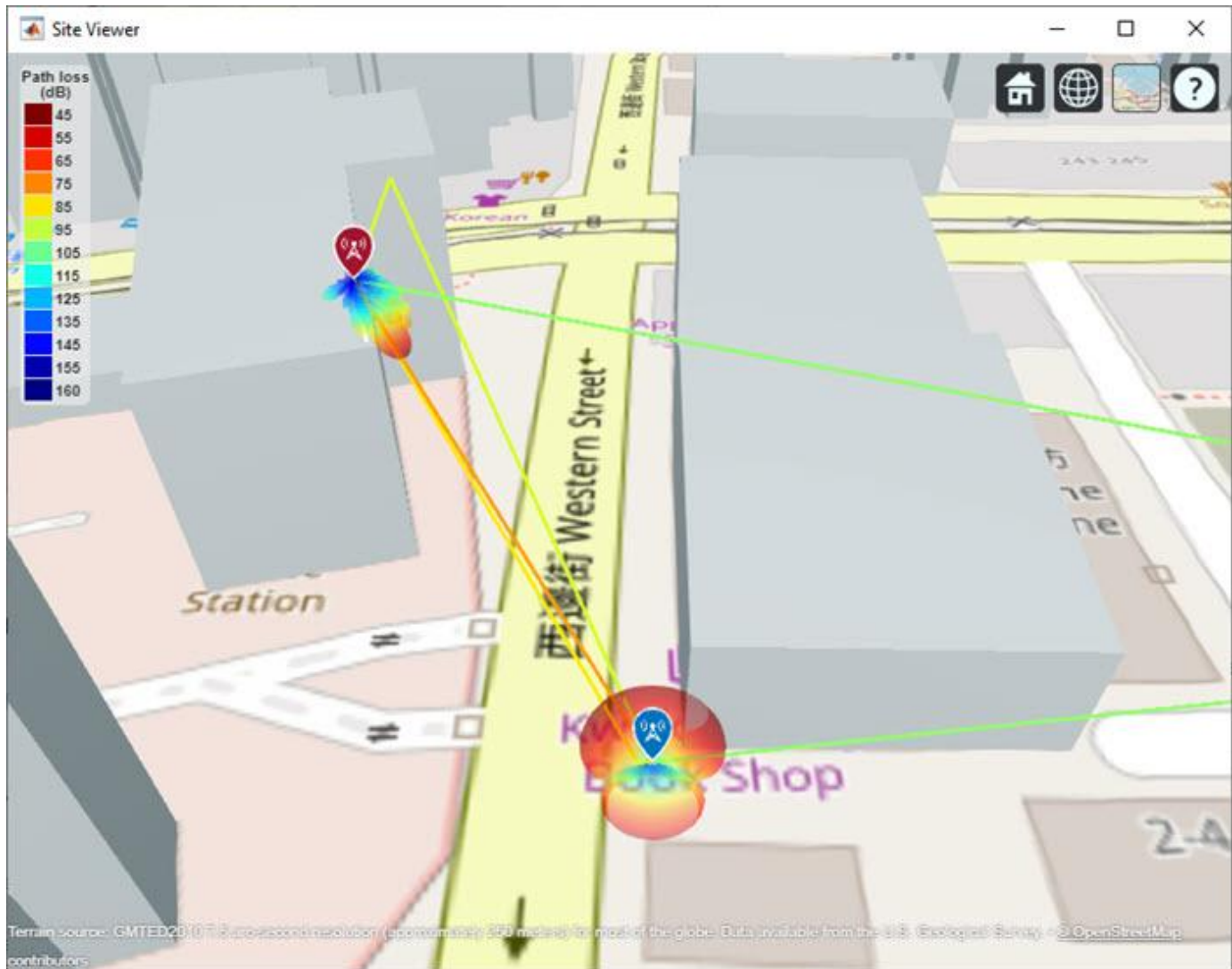
**References**

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

**Local Functions**

```
function [wtx,wrx,D] = getBeamformingWeights(hEst,nLayers,scOffset,noRBs)
% Get beamforming weights given a channel matrix hEst and the number of
% layers nLayers. One set of weights is provided for the whole bandwidth.
% The beamforming weights are calculated using singular value (SVD)
% decomposition.
%
% Only part of the channel estimate is used to get the weights, this is
% indicated by an offset SCOFFSET (offset from the first subcarrier) and a
% width in RBs (NORBS).

% Average channel estimate
```

```
[~,~,R,P] = size(hEst);
%H = permute(mean(reshape(hEst,[],R,P)),[2 3 1]);

scNo = scOffset+1;
hEst = hEst(scNo:scNo+(12*noRBs-1),:,:,:);
H = permute(mean(reshape(hEst,[],R,P)),[2 3 1]);

% SVD decomposition
[U,D,V] = svd(H);
wtx = V(:,1:nLayers).';
wrx = U(:,1:nLayers)';
end
```

# NR SSB Beam Sweeping

This example shows how to employ beam sweeping at both the transmitter (gNB) and receiver (UE) ends of a 5G NR system. Using synchronization signal blocks (SSB), this example illustrates some of the beam management procedures used during initial access. To accomplish beam sweeping, the example uses several components from Phased Array System Toolbox™.

**Introduction**

The support of millimeter wave (mmWave) frequencies requires directional links, which led to the specification of beam management procedures for initial access in NR. Beam management is a set of Layer 1 (physical) and Layer 2 (medium access control) procedures to acquire and maintain a set of beam pair links (a beam used at gNB paired with a beam used at UE). Beam management procedures are applied for both downlink and uplink transmission and reception [ 1 ], [ 2 ]. These procedures include:

- Beam sweeping
- Beam measurement
- Beam determination
- Beam reporting
- Beam recovery

This example focuses on initial access procedures for idle users when a connection is established between the user equipment (UE) and access network node (gNB). At the physical layer, using synchronization signal blocks (SSB) transmitted as a burst in the downlink direction (gNB to UE), the example highlights both transmit/receive point (TRP) beam sweeping and UE beam sweeping to establish a beam pair link. Among the multiple beam management procedures, TR 38.802 defines this dual-end sweep as procedure P-1 [ 1 ].

Once connected, the same beam pair link can be used for subsequent transmissions. If necessary, the beams are further refined using CSI-RS (for downlink) and SRS (for uplink). In case of beam failure, these pair links can be reestablished. For an example of beam pair refinement, see "NR Downlink Transmit-End Beam Refinement Using CSI-RS" (5G Toolbox).

This example generates an NR synchronization signal burst, beamforms each of the SSBs within the burst to sweep over both azimuth and elevation directions, transmits this beamformed signal over a spatial scattering channel, and processes this received signal over the multiple receive-end beams. The example measures the reference signal received power (RSRP) for each of the transmit-receive beam pairs (in a dual loop) and determines the beam pair link with the maximum RSRP. This beam pair link thus signifies the best beam-pair at transmit and receive ends for the simulated spatial scenario. This figure shows the main processing steps with the beam management ones highlighted in color.



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

**Simulation Parameters**

Define system parameters for the example. Modify these parameters to explore their impact on the system.

```
prm.NCellID = 1;                    % Cell ID
prm.FreqRange = 'FR1';              % Frequency range: 'FR1' or 'FR2'
prm.CenterFreq = 3.5e9;             % Hz
prm.SSBlockPattern = 'Case B';      % Case A/B/C/D/E
prm.SSBTransmitted = [ones(1,8) zeros(1,0)];   % 4/8 or 64 in length

prm.TxArraySize = [8 8];            % Transmit array size, [rows cols]
prm.TxAZlim = [-60 60];             % Transmit azimuthal sweep limits
prm.TxELlim = [-90 0];              % Transmit elevation sweep limits

prm.RxArraySize = [2 2];            % Receive array size, [rows cols]
prm.RxAZlim = [-180 180];           % Receive azimuthal sweep limits
prm.RxELlim = [0 90];               % Receive elevation sweep limits

prm.ElevationSweep = false;         % Enable/disable elevation sweep
prm.SNRdB = 30;                     % SNR, dB
prm.RSRPMode = 'SSSwDMRS';          % {'SSSwDMRS', 'SSSonly'}
```

The example uses these parameters:

- Cell ID for a single-cell scenario with a single BS and UE
- Frequency range, as a string to designate FR1 or FR2 operation
- Center frequency, in Hz, dependent on the frequency range
- Synchronization signal block pattern as one of Case A/B/C for FR1 and Case D/E for FR2. This also selects the subcarrier spacing.
- Transmitted SSBs in the pattern, as a binary vector of length 4 or 8 for FR1 and length 64 for FR2. The number of SSBs transmitted sets the number of beams at both transmit and receive ends.
- Transmit array size, as a two-element row vector specifying the number of antenna elements in the rows and columns of the transmit array, respectively. A uniform rectangular array (URA) is used when both values are greater than one.
- Transmit azimuthal sweep limits in degrees to specify the starting and ending azimuth angles for the sweep
- Transmit elevation sweep limits in degrees to specify the starting and ending elevation angles for the sweep
- Receive array size, as a two-element row vector specifying the number of antenna elements in the rows and columns of the receive array, respectively. A uniform rectangular array (URA) is used when both values are greater than one.
- Receive azimuthal sweep limits in degrees to specify the starting and ending azimuth angles for the sweep
- Receive elevation sweep limits in degrees to specify the starting and ending elevation angles for the sweep
- Enable or disable elevation sweep for both transmit and receive ends. Enable elevation sweep for FR2 and/or URAs
- Signal-to-noise ratio in dB
- Measurement mode for SSB to specify the use of only secondary synchronization signals ('SSSonly') or use of PBCH DM-RS along with secondary synchronization signals ('SSSwDMRS')

```
prm = validateParams(prm);
```

**Synchronization Signal Burst Configuration**

Set up the synchronization signal burst parameters by using the specified system parameters. For initial access, set the SSB periodicity to 20 ms.

```
txBurst = nrWavegenSSBurstConfig;
txBurst.BlockPattern = prm.SSBlockPattern;
txBurst.TransmittedBlocks = prm.SSBTransmitted;
txBurst.Period = 20;
txBurst.SubcarrierSpacingCommon = prm.SubcarrierSpacingCommon;

% Configure an nrDLCarrierConfig object to use the synchronization signal
% burst parameters and to disable other channels. This object will be used
% by nrWaveformGenerator to generate the SS burst waveform.

cfgDL = configureWaveformGenerator(prm,txBurst);
```

Refer to the tutorial "Synchronization Signal Blocks and Bursts" (5G Toolbox) for more details on synchronization signal blocks and bursts.

**Burst Generation**

Create the SS burst waveform [ 3 ] by calling the `nrWaveformGenerator` function. The generated waveform is not yet beamformed.

```
burstWaveform = nrWaveformGenerator(cfgDL);

% Display spectrogram of SS burst waveform
figure;
ofdmInfo = nrOFDMInfo(cfgDL.SCSCarriers{1}.NSizeGrid,prm.SCS);
nfft = ofdmInfo.Nfft;
spectrogram(burstWaveform,ones(nfft,1),0,nfft,'centered',ofdmInfo.SampleRate,'yaxis','MinThresho
title('Spectrogram of SS burst waveform')
```

Spectrogram of SS burst waveform
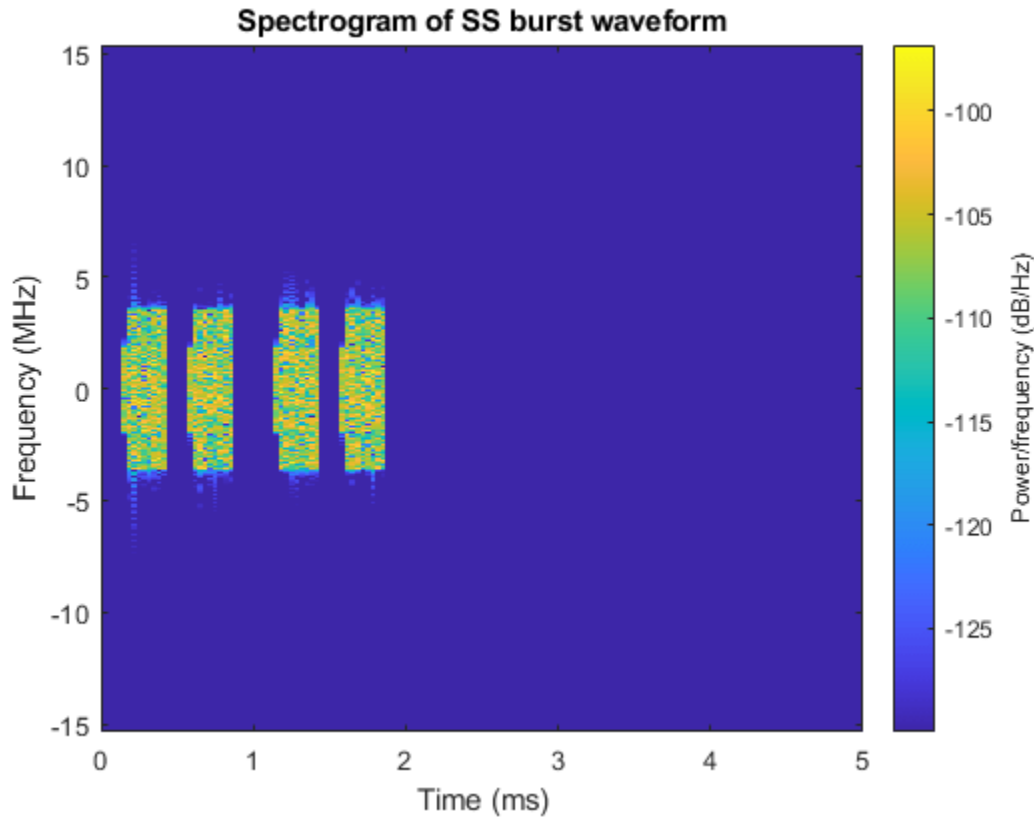
### Channel Configuration

Configure a spatial scattering MIMO channel `channel`. This channel model applies free space path loss and, optionally, other atmospheric attenuations to the input. Specify the locations for the BS and UE as [x,y,z] coordinates in a Cartesian system. Depending on the array sizes specified, employ either uniform linear arrays (ULA) or uniform rectangular arrays (URA). Use isotropic antenna elements for the arrays.

```matlab
c = physconst('LightSpeed');    % Propagation speed
lambda = c/prm.CenterFreq;      % Wavelength

prm.posTx = [0;0;0];            % Transmit array position, [x;y;z], meters
prm.posRx = [100;50;0];         % Receive array position, [x;y;z], meters

toRxRange = rangeangle(prm.posTx,prm.posRx);
spLoss = fspl(toRxRange,lambda);     % Free space path loss

% Transmit array
if prm.IsTxURA
    % Uniform rectangular array
    arrayTx = phased.URA(prm.TxArraySize,0.5*lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform linear array
    arrayTx = phased.ULA(prm.NumTx, ...
        'ElementSpacing',0.5*lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
```

```
        end

        % Receive array
        if prm.IsRxURA
            % Uniform rectangular array
            arrayRx = phased.URA(prm.RxArraySize,0.5*lambda, ...
                'Element',phased.IsotropicAntennaElement);
        else
            % Uniform linear array
            arrayRx = phased.ULA(prm.NumRx, ...
                'ElementSpacing',0.5*lambda, ...
                'Element',phased.IsotropicAntennaElement);
        end

        % Scatterer locations
        prm.FixedScatMode = true;
        if prm.FixedScatMode
            % Fixed single scatterer location
            prm.ScatPos = [50; 80; 0];
        else
            % Generate scatterers at random positions
            Nscat = 10;          % Number of scatterers
            azRange = -180:180;
            elRange = -90:90;
            randAzOrder = randperm(length(azRange));
            randElOrder = randperm(length(elRange));
            azAngInSph = azRange(randAzOrder(1:Nscat));
            elAngInSph = elRange(randElOrder(1:Nscat));
            r = 20;              % radius
            [x,y,z] = sph2cart(deg2rad(azAngInSph),deg2rad(elAngInSph),r);
            prm.ScatPos = [x;y;z] + (prm.posTx + prm.posRx)/2;
        end

        % Configure channel
        channel = phased.ScatteringMIMOChannel;
        channel.PropagationSpeed = c;
        channel.CarrierFrequency = prm.CenterFreq;
        channel.SampleRate = ofdmInfo.SampleRate;
        channel.SimulateDirectPath = false;
        channel.ChannelResponseOutputPort = true;
        channel.Polarization = 'None';
        channel.TransmitArray = arrayTx;
        channel.TransmitArrayPosition = prm.posTx;
        channel.ReceiveArray = arrayRx;
        channel.ReceiveArrayPosition = prm.posRx;
        channel.ScattererSpecificationSource = 'Property';
        channel.ScattererPosition = prm.ScatPos;
        channel.ScattererCoefficient = ones(1,size(prm.ScatPos,2));

        % Get maximum channel delay
        [~,~,tau] = channel(complex(randn(ofdmInfo.SampleRate*1e-3,prm.NumTx), ...
            randn(ofdmInfo.SampleRate*1e-3,prm.NumTx)));
        maxChDelay = ceil(max(tau)*ofdmInfo.SampleRate);
```

**Transmit-End Beam Sweeping**

To achieve TRP beam sweeping, beamform each of the SS blocks in the generated burst using analog beamforming. Based on the number of SS blocks in the burst and the sweep ranges specified,

determine both the azimuth and elevation directions for the different beams. Then beamform the individual blocks within the burst to each of these directions.

```
% Number of beams at both transmit and receive ends
numBeams = sum(txBurst.TransmittedBlocks);

% Transmit beam angles in azimuth and elevation, equi-spaced
azBW = beamwidth(arrayTx,prm.CenterFreq,'Cut','Azimuth');
elBW = beamwidth(arrayTx,prm.CenterFreq,'Cut','Elevation');
txBeamAng = hGetBeamSweepAngles(numBeams,prm.TxAZlim,prm.TxELlim, ...
    azBW,elBW,prm.ElevationSweep);

% For evaluating transmit-side steering weights
SteerVecTx = phased.SteeringVector('SensorArray',arrayTx, ...
    'PropagationSpeed',c);

% Get the set of OFDM symbols occupied by each SSB
numBlocks = length(txBurst.TransmittedBlocks);
burstStartSymbols = ssBurstStartSymbols(txBurst.BlockPattern,numBlocks);
burstStartSymbols = burstStartSymbols(txBurst.TransmittedBlocks==1);
burstOccupiedSymbols = burstStartSymbols.' + (1:4);

% Apply steering per OFDM symbol for each SSB
gridSymLengths = repmat(ofdmInfo.SymbolLengths,1,cfgDL.NumSubframes);
%   repeat burst over numTx to prepare for steering
strTxWaveform = repmat(burstWaveform,1,prm.NumTx)./sqrt(prm.NumTx);
for ssb = 1:numBeams

    % Extract SSB waveform from burst
    blockSymbols = burstOccupiedSymbols(ssb,:);
    startSSBInd = sum(gridSymLengths(1:blockSymbols(1)-1))+1;
    endSSBInd = sum(gridSymLengths(1:blockSymbols(4)));
    ssbWaveform = strTxWaveform(startSSBInd:endSSBInd,1);

    % Generate weights for steered direction
    wT = SteerVecTx(prm.CenterFreq,txBeamAng(:,ssb));

    % Apply weights per transmit element to SSB
    strTxWaveform(startSSBInd:endSSBInd,:) = ssbWaveform.*(wT');

end
```

The beamformed burst waveform is then transmitted over the spatially-aware scattering channel.

**Receive-End Beam Sweeping and Measurement**

For receive-end beam sweeping, the transmitted beamformed burst waveform is received successively over each receive beam. For N transmit beams and M receive beams in procedure P-1, each of the N beams is transmitted M times from gNB so that each transmit beam is received over the M receive beams.

The example assumes both N and M to be equal to the number of SSBs in the burst. For simplicity, the example generates only one burst, but to mimic the burst reception over the air M times, the receiver processes this single burst M times.

This figure shows a beam-based diagram for the sweeps at both gNB and UE for N = M = 4, in the azimuthal plane. The diagram shows the time taken for the dual sweep, where each interval at gNB

corresponds to an SSB and each interval at the UE corresponds to the SS burst. For the depicted scenario, beams S3 and U2 are highlighted as the selected beam-pair link notionally. The example implements the dual-sweep over a time duration of N*M time instants.
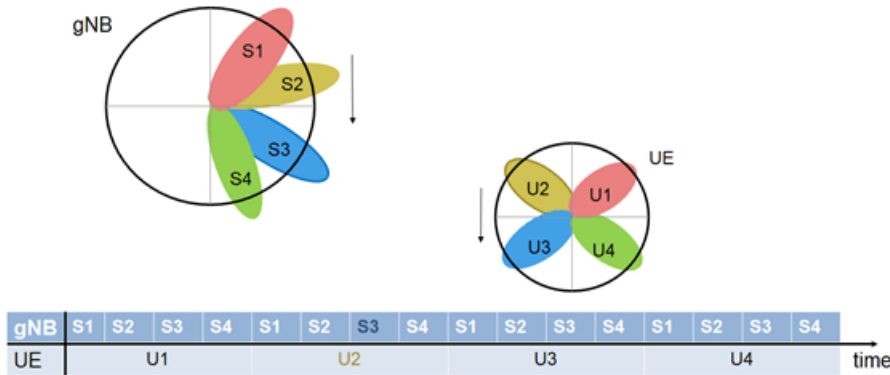


The receive processing of the transmitted burst includes

- Application of the spatially-aware fading channel
- Receive gain to compensate for the induced path loss and AWGN
- Receive-end beamforming
- Timing correction
- OFDM demodulation
- Extracting the known SSB grid
- Measuring the RSRP based on the specified measurement mode

The processing repeats these steps for each of the receive beams, then selects the best beam-pair based on the complete set of measurements made.

To highlight beam sweeping, the example assumes known SSB information at the receiver. For more details on recovery processing see "NR Cell Search and MIB and SIB1 Recovery" (5G Toolbox).

For the idle mode SS-RSRP measurement, use either only the secondary synchronization signals (SSS) or the physical broadcast channel (PBCH) demodulation reference signals (DM-RS) in addition to the SSS (Section 5.1.1. of [ 4 ]). Specify this by the RSRPMode parameter of the example. For FR2, the RSRP measurement is based on the combined signal from antenna elements, while the measurement is per antenna element for FR1.

```
% Receive beam angles in azimuth and elevation, equi-spaced
azBW = beamwidth(arrayRx,prm.CenterFreq,'Cut','Azimuth');
elBW = beamwidth(arrayRx,prm.CenterFreq,'Cut','Elevation');
rxBeamAng = hGetBeamSweepAngles(numBeams,prm.RxAZlim,prm.RxELlim, ...
    azBW,elBW,prm.ElevationSweep);

% For evaluating receive-side steering weights
SteerVecRx = phased.SteeringVector('SensorArray',arrayRx, ...
    'PropagationSpeed',c);

% AWGN level
SNR = 10^(prm.SNRdB/20);                         % Convert to linear gain
N0 = 1/(sqrt(2.0*prm.NumRx*double(ofdmInfo.Nfft))*SNR); % Noise Std. Dev.
```

```matlab
% Receive gain in linear terms, to compensate for the path loss
rxGain = 10^(spLoss/20);

% Generate a reference grid for timing correction
%   assumes an SSB in first slot
carrier = nrCarrierConfig('NCellID',prm.NCellID);
carrier.NSizeGrid = cfgDL.SCSCarriers{1}.NSizeGrid;
carrier.SubcarrierSpacing = prm.SCS;
pssRef = nrPSS(carrier.NCellID);
pssInd = nrPSSIndices;
ibar_SSB = 0;
pbchdmrsRef = nrPBCHDMRS(carrier.NCellID,ibar_SSB);
pbchDMRSInd = nrPBCHDMRSIndices(carrier.NCellID);
pssGrid = zeros([240 4]);
pssGrid(pssInd) = pssRef;
pssGrid(pbchDMRSInd) = pbchdmrsRef;
refGrid = zeros([12*carrier.NSizeGrid ofdmInfo.SymbolsPerSlot]);
burstOccupiedSubcarriers = carrier.NSizeGrid*6 + (-119:120).';
refGrid(burstOccupiedSubcarriers, ...
    burstOccupiedSymbols(1,:)) = pssGrid;

% Loop over all receive beams
rsrp = zeros(numBeams,numBeams);
for rIdx = 1:numBeams

    % Fading channel, with path loss
    txWave = [strTxWaveform; zeros(maxChDelay,size(strTxWaveform,2))];
    fadWave = channel(txWave);

    % Receive gain, to compensate for the path loss
    fadWaveG = fadWave*rxGain;

    % Add WGN
    noise = N0*complex(randn(size(fadWaveG)),randn(size(fadWaveG)));
    rxWaveform = fadWaveG + noise;

    % Generate weights for steered direction
    wR = SteerVecRx(prm.CenterFreq,rxBeamAng(:,rIdx));

    % Apply weights per receive element
    if strcmp(prm.FreqRange, 'FR1')
        strRxWaveform = rxWaveform.*(wR');
    else  % for FR2, combine signal from antenna elements
        strRxWaveform = rxWaveform*conj(wR);
    end

    % Correct timing
    offset = nrTimingEstimate(carrier, ...
        strRxWaveform(1:ofdmInfo.SampleRate*1e-3,:),refGrid*wR(1)');
    if offset > maxChDelay
        offset = 0;
    end
    strRxWaveformS = strRxWaveform(1+offset:end,:);

    % OFDM Demodulate
    rxGrid = nrOFDMDemodulate(carrier,strRxWaveformS);

    % Loop over all SSBs in rxGrid (transmit end)
```

```
    for tIdx = 1:numBeams
        % Get each SSB grid
        rxSSBGrid = rxGrid(burstOccupiedSubcarriers, ...
            burstOccupiedSymbols(tIdx,:),:);

        if strcmpi(prm.RSRPMode,'SSSwDMRS')
            meas = nrSSBMeasurements(rxSSBGrid,carrier.NCellID,mod(tIdx-1,8));
        else
            meas = nrSSBMeasurements(rxSSBGrid,carrier.NCellID);
        end
        rsrp(rIdx,tIdx) = max(meas.RSRPPerAntenna);
    end
end
```

**Beam Determination**

After the dual-end sweep and measurements are complete, determine the best beam-pair link based on the RSRP measurement.

```
[m,i] = max(rsrp,[],'all','linear');    % First occurrence is output
% i is column-down first (for receive), then across columns (for transmit)
[rxBeamID,txBeamID] = ind2sub([numBeams numBeams],i(1));

% Display the selected beam pair
disp(['Selected Beam pair with RSRP: ' num2str(rsrp(rxBeamID, ...
    txBeamID)) ' dBm', 13 '  Transmit #' num2str(txBeamID) ...
    ' (Azimuth: ' num2str(txBeamAng(1,txBeamID)) ', Elevation: ' ...
    num2str(txBeamAng(2,txBeamID)) ')' 13 '  Receive #' num2str(rxBeamID) ...
    ' (Azimuth: ' num2str(rxBeamAng(1,rxBeamID)) ', Elevation: ' ...
    num2str(rxBeamAng(2,rxBeamID)) ')' ]);

% Display final beam pair patterns
h = figure('Position',figposition([32 55 32 40]),'MenuBar','none');
h.Name = 'Selected Transmit Array Response Pattern';
wT = SteerVecTx(prm.CenterFreq,txBeamAng(:,txBeamID));
pattern(arrayTx,prm.CenterFreq,'PropagationSpeed',c,'Weights',wT);

h = figure('Position',figposition([32 55 32 40]),'MenuBar','none');
h.Name = 'Selected Receive Array Response Pattern';
wR = SteerVecRx(prm.CenterFreq,rxBeamAng(:,rxBeamID));
pattern(arrayRx,prm.CenterFreq,'PropagationSpeed',c,'Weights',wR);

% Plot MIMO scenario with tx, rx, scatterers, and determined beams. Beam
% patterns in this figure resemble the power patterns in linear scale.
prmScene = struct();
prmScene.TxArray = arrayTx;
prmScene.RxArray = arrayRx;
prmScene.TxArrayPos = prm.posTx;
prmScene.RxArrayPos = prm.posRx;
prmScene.ScatterersPos = prm.ScatPos;
prmScene.Lambda = lambda;
prmScene.ArrayScaling = 1;      % To enlarge antenna arrays in the plot
prmScene.MaxTxBeamLength = 45; % Maximum length of transmit beams in the plot
prmScene.MaxRxBeamLength = 25; % Maximum length of receive beam in the plot
hPlotSpatialMIMOScene(prmScene,wT,wR);
if ~prm.ElevationSweep
    view(2);
end
```

Selected Beam pair with RSRP: 45.0787 dBm  Transmit #8 (Azimuth: 60, Elevation: 0)  Receive #6 (/

3D Directivity Pattern
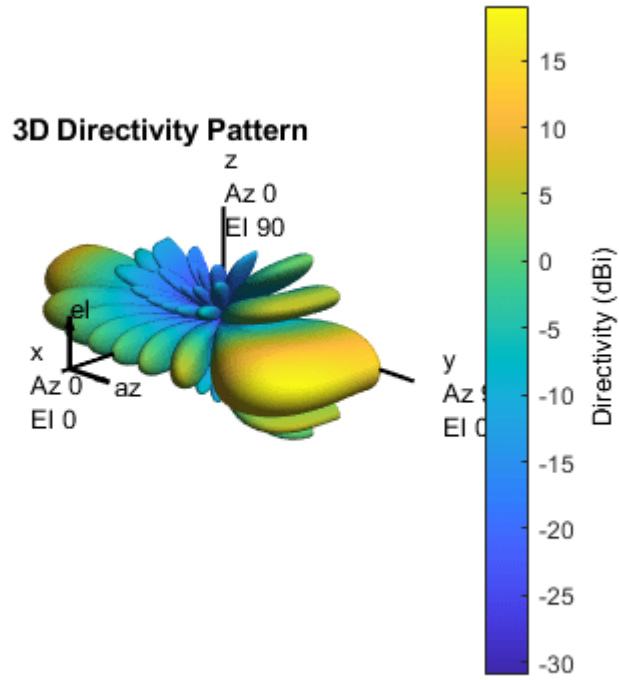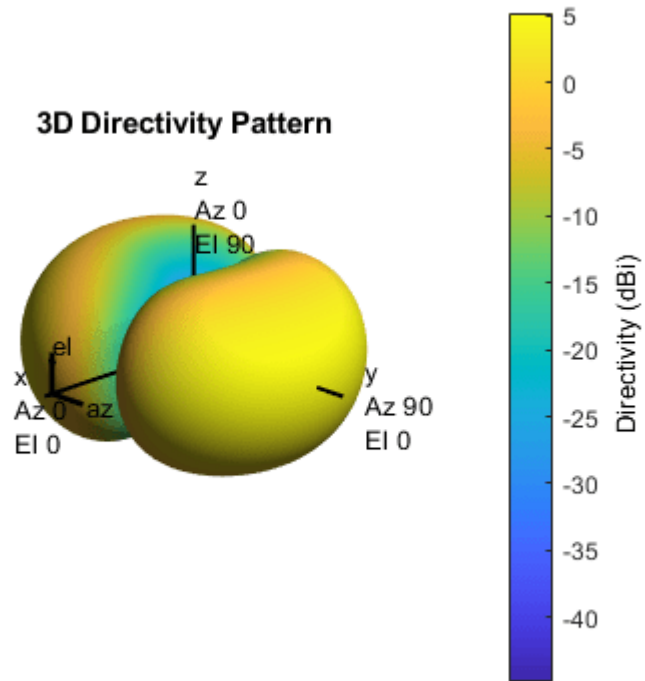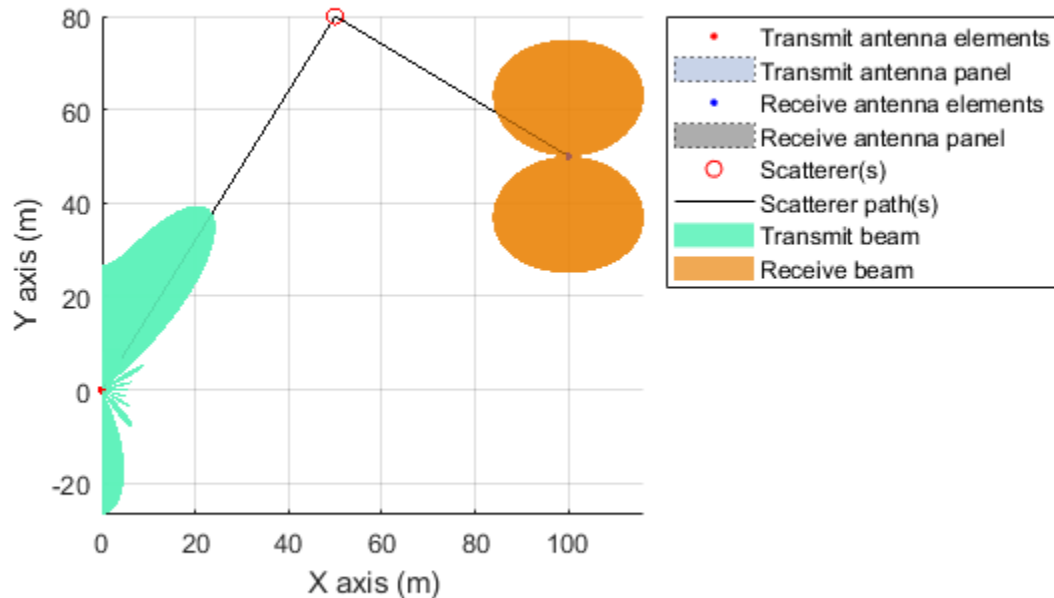
These plots highlight the transmit directivity pattern, receive directivity pattern, and the spatial scene, respectively. The results are dependent on the individual beam directions used for the sweeps. The spatial scene offers a combined view of the transmit and receive arrays and the respective determined beams, along with the scatterers.

**Summary and Further Exploration**

This example highlights the P-1 beam management procedure by using synchronization signal blocks for transmit-end and receive-end beam sweeping. By measuring the reference signal received power for SSBs, you can identify the best beam pair link for a selected spatial environment.

The example allows variation on frequency range, SSB block pattern, number of SSBs, transmit and receive array sizes, transmit and receive sweep ranges, and the measuring mode. To see the impact of parameters on the beam selection, experiment with different values. The receive processing is simplified to highlight the beamforming aspects for the example.

For an example of the P-2 procedures of transmit-end beam sweeping using CSI-RS signals for the downlink, see "NR Downlink Transmit-End Beam Refinement Using CSI-RS" (5G Toolbox). You can use these procedures for beam refinement and adjustment in the connected mode, once the initial beam pair links are established [ 5 ], [ 6 ].

**References**

1    3GPP TR 38.802. "Study on New Radio access technology physical layer aspects." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

**2**   Giordani, M., M. Polese, A. Roy, D. Castor, and M. Zorzi. "A tutorial on beam management for 3GPP NR at mmWave frequencies." *IEEE Comm. Surveys & Tutorials*, vol. 21, No. 1, Q1 2019.

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

**4**   3GPP TS 38.215. "NR; Physical layer measurements." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

**5**   Giordani, M., M. Polese, A. Roy, D. Castor, and M. Zorzi. "Standalone and non-standalone beam management for 3GPP NR at mmWaves." *IEEE Comm. Mag.*, April 2019, pp. 123-129.

**6**   Onggosanusi, E., S. Md. Rahman, et al. "Modular and high-resolution channel state information and beam management for 5G NR." *IEEE Comm. Mag.*, March 2018, pp. 48-55.

**Local Functions**

```matlab
function prm = validateParams(prm)
% Validate user specified parameters and return updated parameters
%
% Only cross-dependent checks are made for parameter consistency.

    if strcmpi(prm.FreqRange,'FR1')
        if prm.CenterFreq > 7.125e9 || prm.CenterFreq < 410e6
            error(['Specified center frequency is outside the FR1 ', ...
                    'frequency range (410 MHz - 7.125 GHz).']);
        end
        if strcmpi(prm.SSBlockPattern,'Case D') ||  ...
           strcmpi(prm.SSBlockPattern,'Case E')
            error(['Invalid SSBlockPattern for selected FR1 frequency ' ...
                    'range. SSBlockPattern must be one of ''Case A'' or ' ...
                    '''Case B'' or ''Case C'' for FR1.']);
        end
        if ~((length(prm.SSBTransmitted)==4) || ...
             (length(prm.SSBTransmitted)==8))
            error(['SSBTransmitted must be a vector of length 4 or 8', ...
                    'for FR1 frequency range.']);
        end
        if (prm.CenterFreq <= 3e9) && (length(prm.SSBTransmitted)~=4)
            error(['SSBTransmitted must be a vector of length 4 for ' ...
                    'center frequency less than or equal to 3GHz.']);
        end
        if (prm.CenterFreq > 3e9) && (length(prm.SSBTransmitted)~=8)
            error(['SSBTransmitted must be a vector of length 8 for ', ...
                    'center frequency greater than 3GHz and less than ', ...
                    'or equal to 7.125GHz.']);
        end
    else % 'FR2'
        if prm.CenterFreq > 52.6e9 || prm.CenterFreq < 24.25e9
            error(['Specified center frequency is outside the FR2 ', ...
                    'frequency range (24.25 GHz - 52.6 GHz).']);
        end
        if ~(strcmpi(prm.SSBlockPattern,'Case D') || ...
                strcmpi(prm.SSBlockPattern,'Case E'))
            error(['Invalid SSBlockPattern for selected FR2 frequency ' ...
                    'range. SSBlockPattern must be either ''Case D'' or ' ...
                    '''Case E'' for FR2.']);
        end
        if length(prm.SSBTransmitted)~=64
            error(['SSBTransmitted must be a vector of length 64 for ', ...
```

```matlab
                    'FR2 frequency range.']);
        end
    end

    prm.NumTx = prod(prm.TxArraySize);
    prm.NumRx = prod(prm.RxArraySize);
    if prm.NumTx==1 || prm.NumRx==1
        error(['Number of transmit or receive antenna elements must be', ...
                ' greater than 1.']);
    end
    prm.IsTxURA = (prm.TxArraySize(1)>1) && (prm.TxArraySize(2)>1);
    prm.IsRxURA = (prm.RxArraySize(1)>1) && (prm.RxArraySize(2)>1);

    if ~( strcmpi(prm.RSRPMode,'SSSonly') || ...
            strcmpi(prm.RSRPMode,'SSSwDMRS') )
        error(['Invalid RSRP measuring mode. Specify either ', ...
                '''SSSonly'' or ''SSSwDMRS'' as the mode.']);
    end

    % Select SCS based on SSBlockPattern
    switch lower(prm.SSBlockPattern)
        case 'case a'
            scs = 15;
            cbw = 10;
            scsCommon = 15;
        case {'case b', 'case c'}
            scs = 30;
            cbw = 25;
            scsCommon = 30;
        case 'case d'
            scs = 120;
            cbw = 100;
            scsCommon = 120;
        case 'case e'
            scs = 240;
            cbw = 200;
            scsCommon = 120;
    end
    prm.SCS = scs;
    prm.ChannelBandwidth = cbw;
    prm.SubcarrierSpacingCommon = scsCommon;

end

function ssbStartSymbols = ssBurstStartSymbols(ssbBlockPattern,Lmax)
% Starting OFDM symbols of SS burst.

    % 'alln' gives the overall set of SS block indices 'n' described in
    % TS 38.213 Section 4.1, from which a subset is used for each Case A-E
    alln = [0; 1; 2; 3; 5; 6; 7; 8; 10; 11; 12; 13; 15; 16; 17; 18];

    cases = {'Case A' 'Case B' 'Case C' 'Case D' 'Case E'};
    m = [14 28 14 28 56];
    i = {[2 8] [4 8 16 20] [2 8] [4 8 16 20] [8 12 16 20 32 36 40 44]};
    nn = [2 1 2 16 8];

    caseIdx = find(strcmpi(ssbBlockPattern,cases));
    if (any(caseIdx==[1 2 3]))
```

**18-603**

```matlab
        if (Lmax==4)
            nn = nn(caseIdx);
        elseif (Lmax==8)
            nn = nn(caseIdx) * 2;
        end
    else
        nn = nn(caseIdx);
    end

    n = alln(1:nn);
    ssbStartSymbols = (i{caseIdx} + m(caseIdx)*n).';
    ssbStartSymbols = ssbStartSymbols(:).';

end

function cfgDL = configureWaveformGenerator(prm,txBurst)
% Configure an nrDLCarrierConfig object to be used by nrWaveformGenerator
% to generate the SS burst waveform.

    cfgDL = nrDLCarrierConfig;
    cfgDL.SCSCarriers{1}.SubcarrierSpacing = prm.SCS;
    if (prm.SCS==240)
        cfgDL.SCSCarriers = [cfgDL.SCSCarriers cfgDL.SCSCarriers];
        cfgDL.SCSCarriers{2}.SubcarrierSpacing = prm.SubcarrierSpacingCommon;
        cfgDL.BandwidthParts{1}.SubcarrierSpacing = prm.SubcarrierSpacingCommon;
    else
        cfgDL.BandwidthParts{1}.SubcarrierSpacing = prm.SCS;
    end
    cfgDL.PDSCH{1}.Enable = false;
    cfgDL.PDCCH{1}.Enable = false;
    cfgDL.ChannelBandwidth = prm.ChannelBandwidth;
    cfgDL.FrequencyRange = prm.FreqRange;
    cfgDL.NCellID = prm.NCellID;
    cfgDL.NumSubframes = 5;
    cfgDL.WindowingPercent = 0;
    cfgDL.SSBurst = txBurst;

end
```

# Introduction to Differential Beamforming

This example shows the basic concept of differential beamforming and how to use that technology to form a linear differential microphone array.

**Additive vs. Differential Microphone Arrays**

Microphone arrays have been deployed in many audio applications. Depending on the layout, microphone arrays can be divided into two main categories: additive microphone arrays and differential microphone arrays. The additive microphone arrays are similar to the arrays used in other applications, like wireless communications. For an additive microphone array, the goal is to coherently combine the signals from each channel so we can form a narrow beam towards the source and improve the signal to noise ratio. Several such beamforming algorithms are covered in the "Acoustic Beamforming Using a Microphone Array" on page 18-185 example.

Although additive microphone arrays are very useful, they suffer from several limitations. For illustration purpose, let us define a 2-element array with a 5 cm spacing and look at its pattern between 1-4 kHz. Note that although the audio frequency band spans from 20 Hz to 20 kHz, the 1-4 kHz bands are particularly important for speech intelligibility. The 5 cm spacing in the array is approximately half wavelength at 4 kHz.

```
c = 343;
f = (1:4)*1e3;
fmax = f(end);
lambda = c/fmax;

d = 0.05;
N = 2;
micarraya = phased.ULA(N,d);

pattern(micarraya,f,-180:180,0,'PropagationSpeed',c)
```

Azimuth Cut (elevation angle = 0.0°)



Directivity (dBi), Broadside at 0.00 °
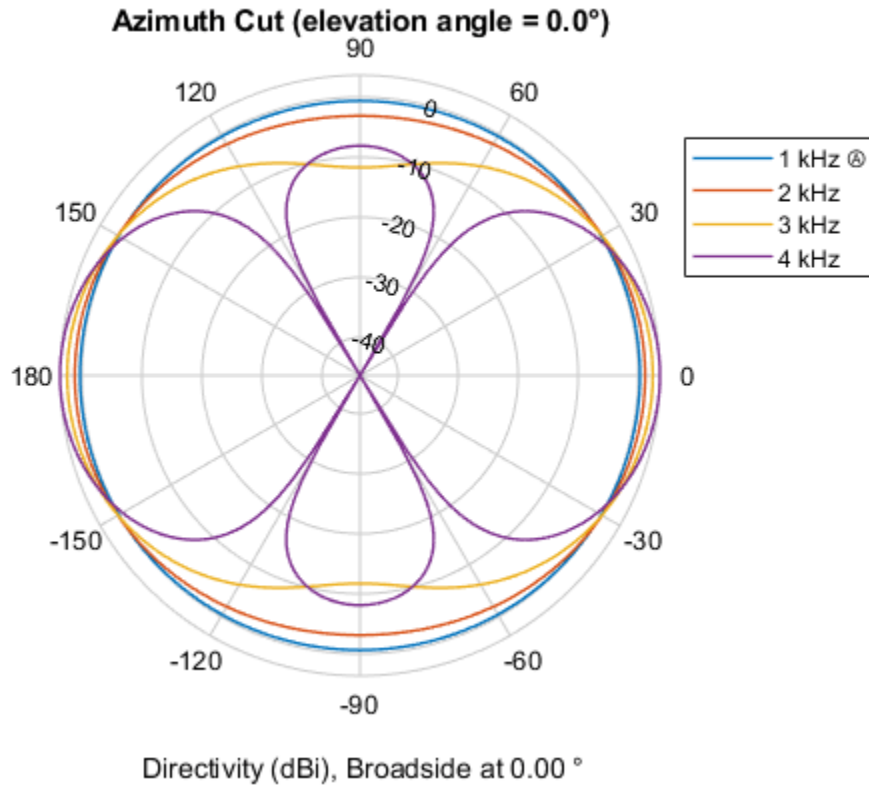
The first thing we can observe from the plots is probably how much the pattern changed across this frequency range. Although the array exhibits a clear directional response at 4 kHz, at 1 kHz, its pattern is essentially omnidirectional. As a result, at lower frequencies, the array cannot achieve much spatial filtering. This, in turn, reduces the directivity of the array. The directivity along the broadside across the frequency range is captured below.

```
Da = directivity(micarraya,f,[0;0],'PropagationSpeed',c);
clf;
plot(f/1e3,Da,'-o');
xlabel('Frequency (kHz)');
ylabel('Directivity (dBi)');
title('Additive Microphone Array Directivity');
```

**Additive Microphone Array Directivity**
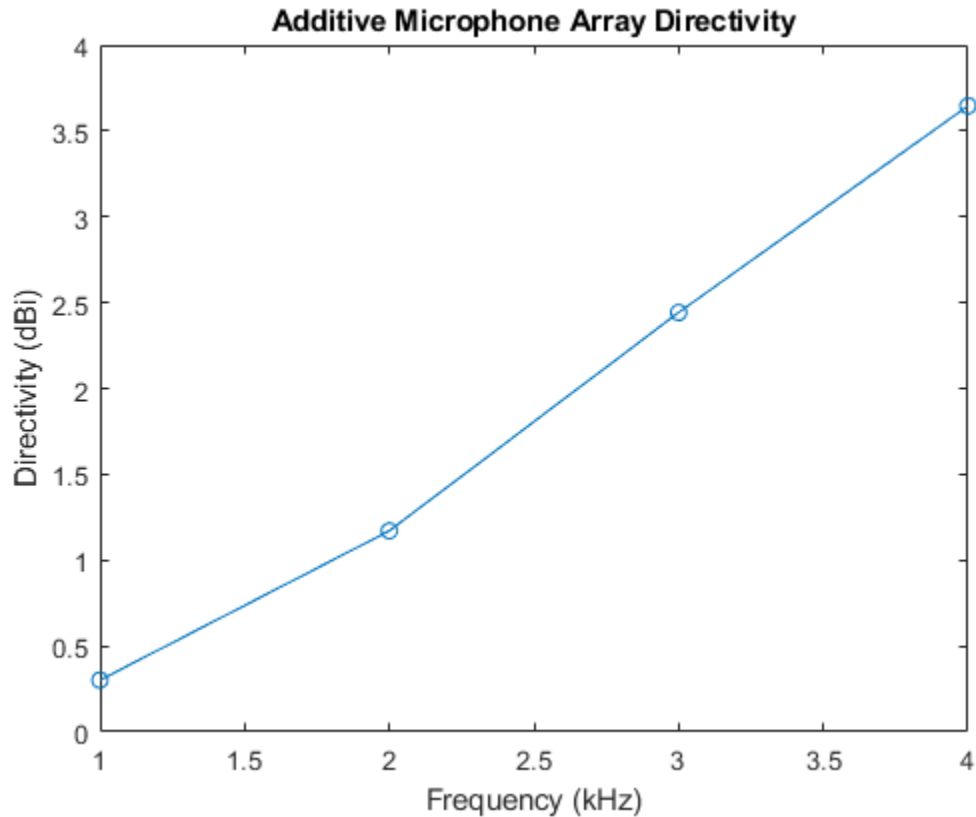


Note that the signal of interest for a microphone array, such as speech and music, is always wideband. Therefore, the pattern difference across the frequency range also causes distortions on the beamformed signal.

Many array geometries have been explored to get around these limitations. A nested array is one such example. In a nested array, small arrays are embedded in a large array. You can then activate different elements across different frequency bands to produce similar patterns across the band. Consider the following 4-element array

```
Nn = 4;
nestedpos = [zeros(1,Nn);[0 1 2 4]*d;zeros(1,Nn)];
micarrayn = phased.ConformalArray('ElementPosition',nestedpos);
viewArray(micarrayn);
set(gcf,'Color','w');
```

Array Geometry
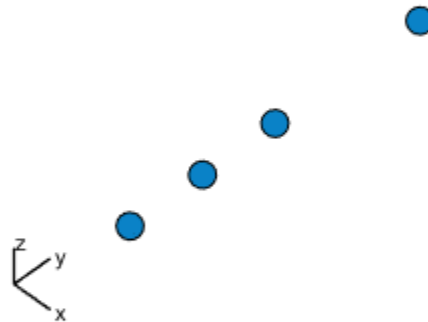


Array Span:
X axis = 0.0 m
Y axis = 0.2 m
Z axis = 0.0 m

If you use the first and the second elements for bands around 4 kHz; the first and the third elements for bands between 2 and 3 kHz; and the first and the fourth elements for bands around 1 kHz, the resulting beam patterns look like

```
wn = [1 1 1 1;0 0 0 1;0 1 1 0;1 0 0 0];
pattern(micarrayn,f,-180:180,0,'PropagationSpeed',c,'Weights',wn);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
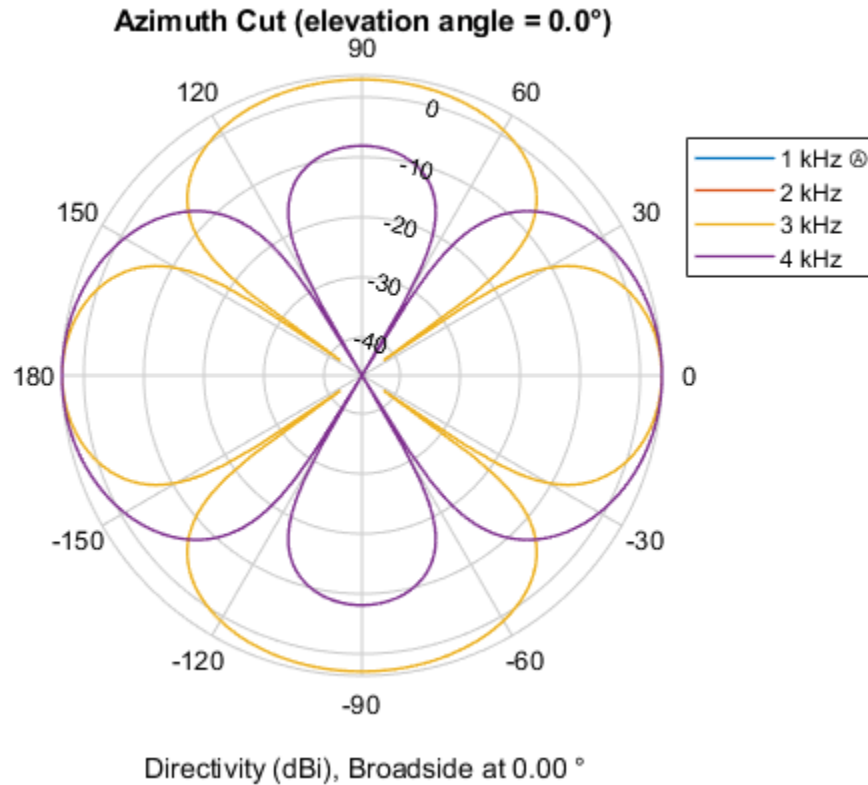
In the plot the patterns from 1, 2, and 4 kHz are essentially identical to each other, but the 3kHz pattern is still different. In addition, the number of elements are doubled in this design. In reality, to get a nested array that can have a frequency invariant pattern across a large band, you will need a large number of microphone elements, so this approach is not very practical.

Differential microphone array (DMA) technology is another possible solution. Because DMAs can form frequency invariant patterns, they become valuable tools for audio applications.

**First Order Linear DMA**

Unlike an additive microphone array, a DMA is more sensitive to the spatial derivative of the acoustic field around the array, thus the name "differential". Since it is impossible to compute the true derivative between microphone elements, the difference between the measurements at each microphone element is used to approximate the derivative. Because a pair of closely located elements can measure derivative more accurately, the element spacing in a DMA is in general much smaller than the wavelength. The following code constructs a 2-element linear array with a spacing of 1/10 wavelength.

```
dd = lambda/10;
micarrayd = phased.ULA(N,dd);
viewArray(micarrayd)
set(gcf,'Color','w');
```

Array Geometry

Aperture Size:
Y axis = 17.15 mm
Element Spacing:
$\Delta$ y = 8.575 mm
Array Axis: Y axis
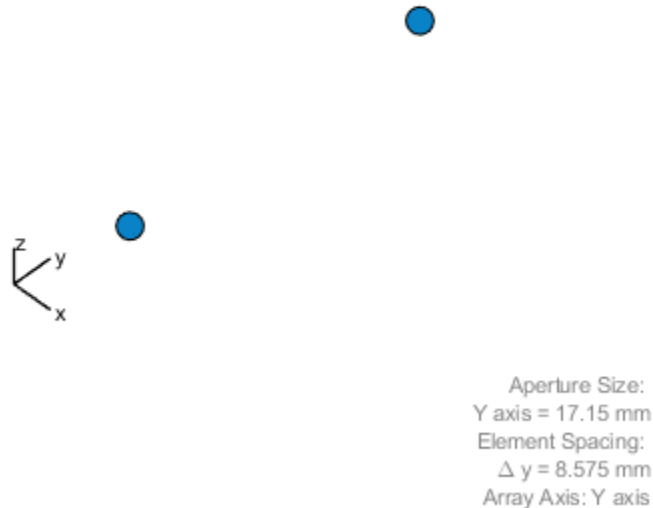
One benefit comes with a close spacing is that the entire array aperture is fairly small. As a result, such linear DMAs are popular choices for hearing aids.

Because differential beamforming measures the field derivatives, its mainlobe points toward the endfire direction. The endfire direction is along the axis of the linear array. This is understandable because for an additive array, the mainlobe is at the broadside, which is the direction perpendicular to the array axis, and the derivative at that direction is 0. Therefore, the design of a DMA is often about null placement. For a two-element linear DMA, when the mainlobe is at endfire, you can only control one null location. This is referred to as a first order DMA. However, even though there is only one null location, varying the location can result in several interesting patterns.

First, put the null at the broadside of the array. This means that we need to compute a weight vector, when combined with the steering vectors for endfire and broadside directions, to generate unit response and null response, respectively. Such a weight vector can be derived using a least square approach.

```
ang_d = 90;  % endfire
ang_n = 0;   % broadside
dd_f = dd./(c./f);
w = complex(zeros(N,numel(f)));
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA');
end

clf;
pattern(micarrayd,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

**Azimuth Cut (elevation angle = 0.0°)**



Directivity (dBi), Broadside at 0.00 °
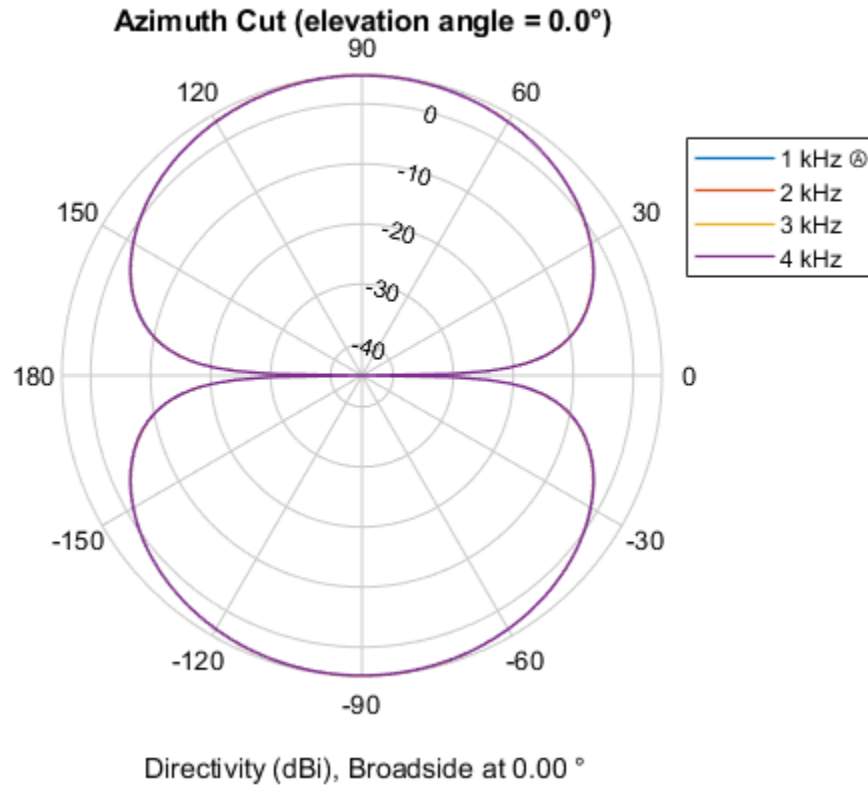
The resulting pattern has the shape of a dipole. Note that the patterns from all frequencies are overlap so they are invariant across the entire frequency band. The directivities are now a constant over the frequency, as shown in the following figure.

```
Dd = directivity(micarrayd,f,ang_d,'PropagationSpeed',c,'Weights',w);
clf;
plot(f/1e3,Dd,'-o');
ylim([0 5]);
xlabel('Frequency (kHz)');
ylabel('Directivity (dBi)');
title('2-element Linear DMA Dipole Directivity');
```
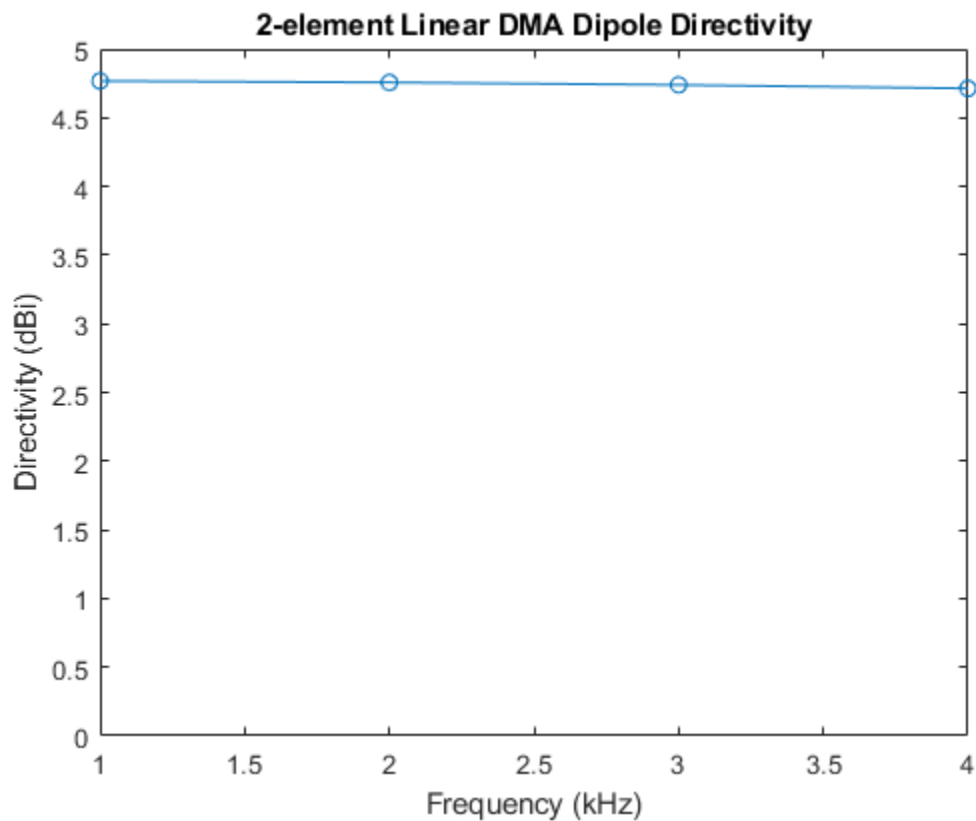
Another important charactersitic associated with a microphone array is its signal to noise ratio (SNR) gain over white noise.

```
agd = phased.ArrayGain('SensorArray',micarrayd,'PropagationSpeed',c,'WeightsInputPort',true);
wngd = agd(f,ang_d,w);
clf;
plot(f/1e3,wngd,'-*');
xlabel('Frequency (kHz)');
ylabel('White Noise Gain (dB)');
title('2-element Linear DMA Dipole SNR Gain');
```

**2-element Linear DMA Dipole SNR Gain**
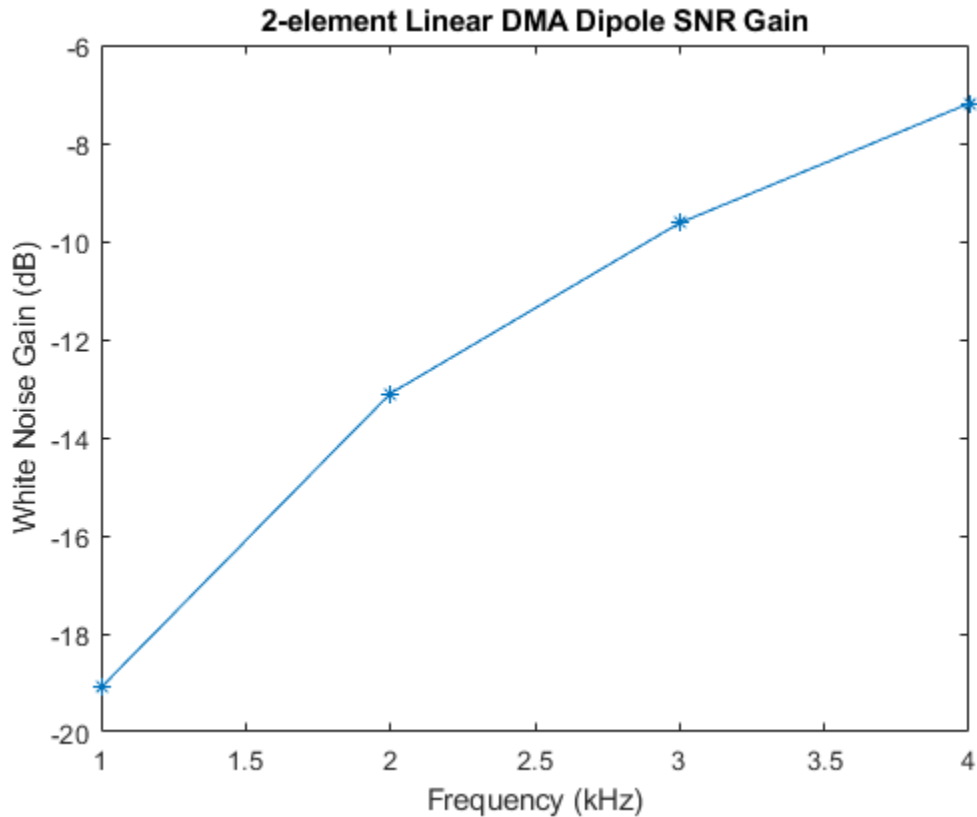


As can be seen from the plot, the SNR gain of this DMA is below 0 dB across the entire frequency range, this means that the noise gets amplified significantly, especially in low frequency regions. Compared to additive arrays, this is probably the biggest issue associated with DMAs. Therefore, engineers may need to make a tradeoff between a more focused beam, which helps dealing with reverberation in a crowded environment, and a higher received signal SNR.

What if the null is located at the other end of the endfire direction?

```
ang_n = -90;    % endfire
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA');
end

pattern(micarrayd,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
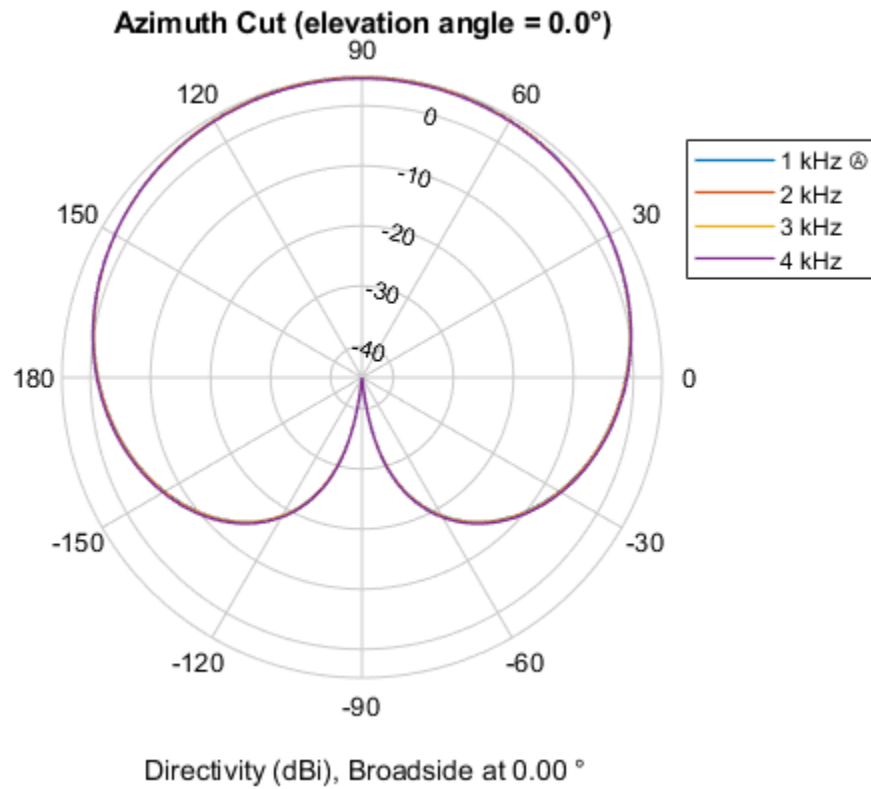
Now the resulting pattern is a cardioid. You can also put the null at -45 degrees to get a supercardioid shape.

```
ang_n = -45;    % endfire
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA');
end

pattern(micarrayd,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
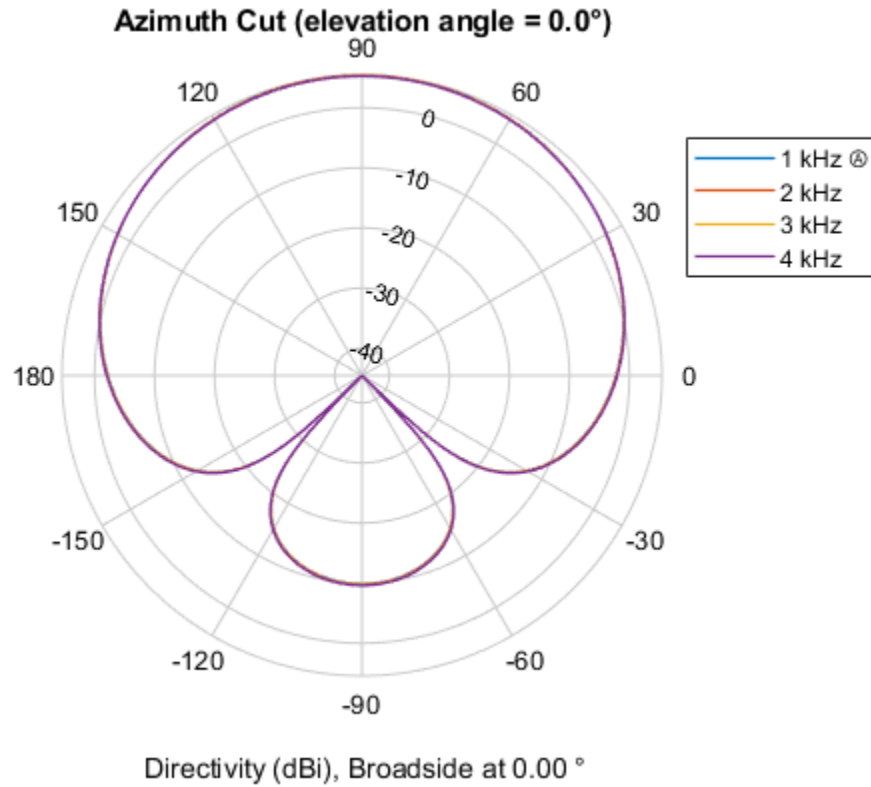
```
wngd = agd(f,ang_d,w);
clf;
plot(f/1e3,wngd,'-*');
xlabel('Frequency (kHz)');
ylabel('White Noise Gain (dB)');
title('2-element Linear DMA Supercardioid SNR Gain');
```
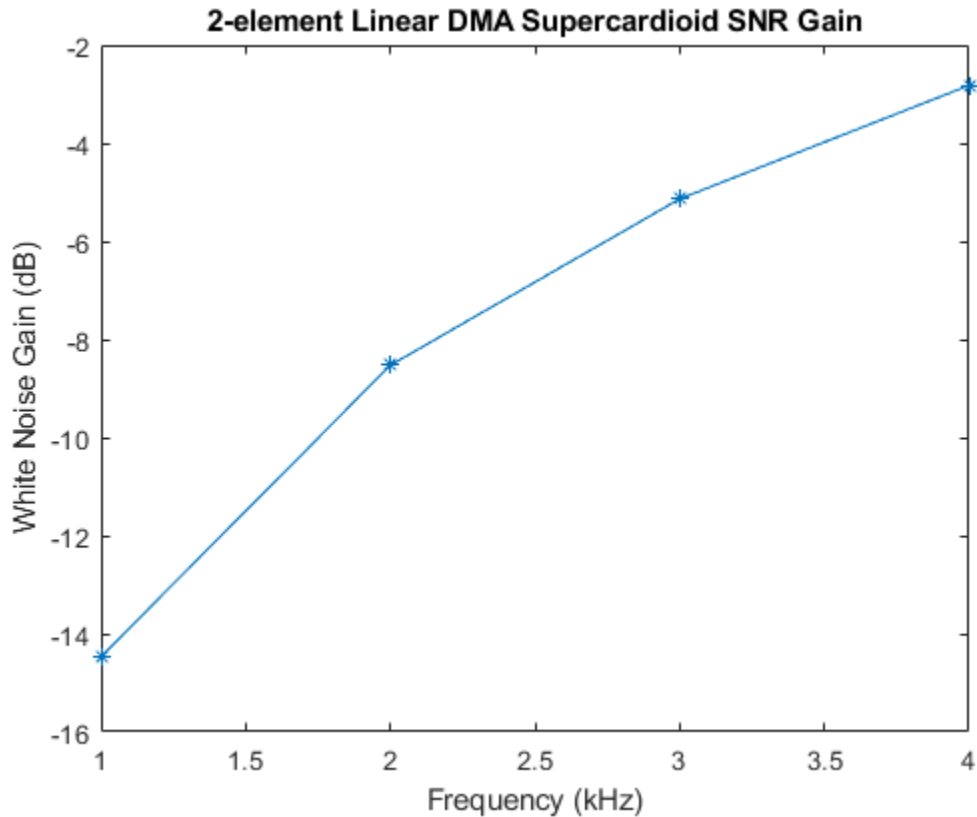
Note that supercardioid shape provides better SNR gain than the dipole shape.

**Higher Order Linear DMA**

You probably have noticed that for a first order linear DMAs, you can assign 1 null direction. In general, an N-element linear array can form an (N-1)-th order DMA, with the capability of allocating N-1 nulls. For example, for a four-microphone, third order DMA, you can put 3 nulls for the supercardioid beam pattern.

```
N = 4;
micarrayd3 = phased.ULA(N,dd);
ang_d = 90;              % endfire
ang_n = [0 -30 -90];     % nulls
w = complex(zeros(N,numel(f)));
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA');
end

pattern(micarrayd3,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
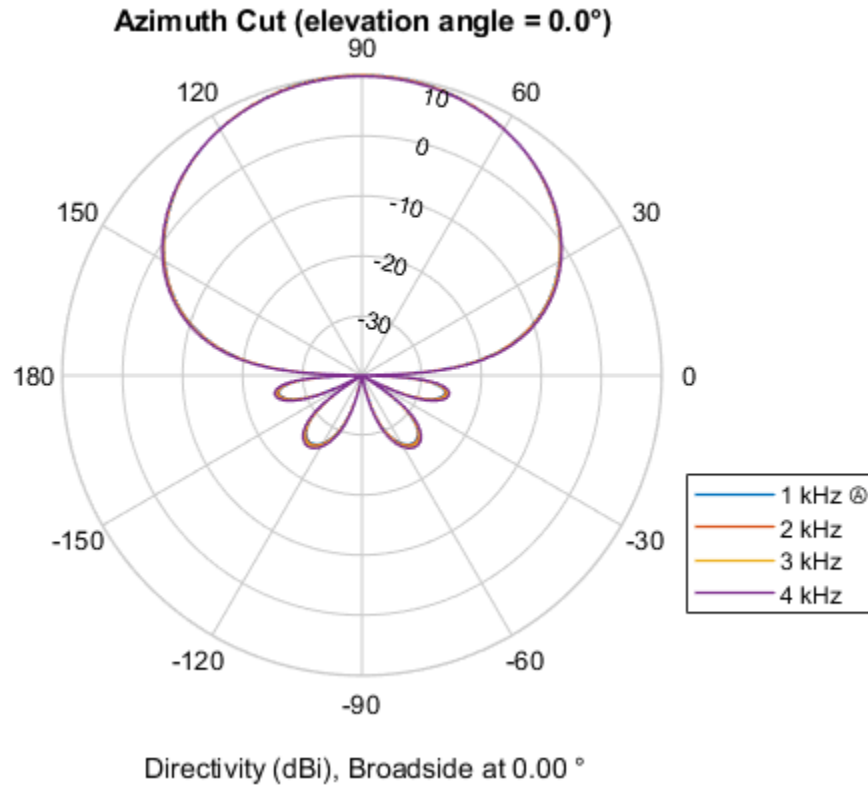
The SNR gain of higher oder DMAs still bears the same trend as a first order DMA.

```
agd = phased.ArrayGain('SensorArray',micarrayd3,'PropagationSpeed',c,'WeightsInputPort',true);
wngd = agd(f,ang_d,w);
clf;
plot(f/1e3,wngd,'-*');
xlabel('Frequency (kHz)');
ylabel('White Noise Gain (dB)');
title('4-element Linear DMA Supercardioid SNR Gain');
```
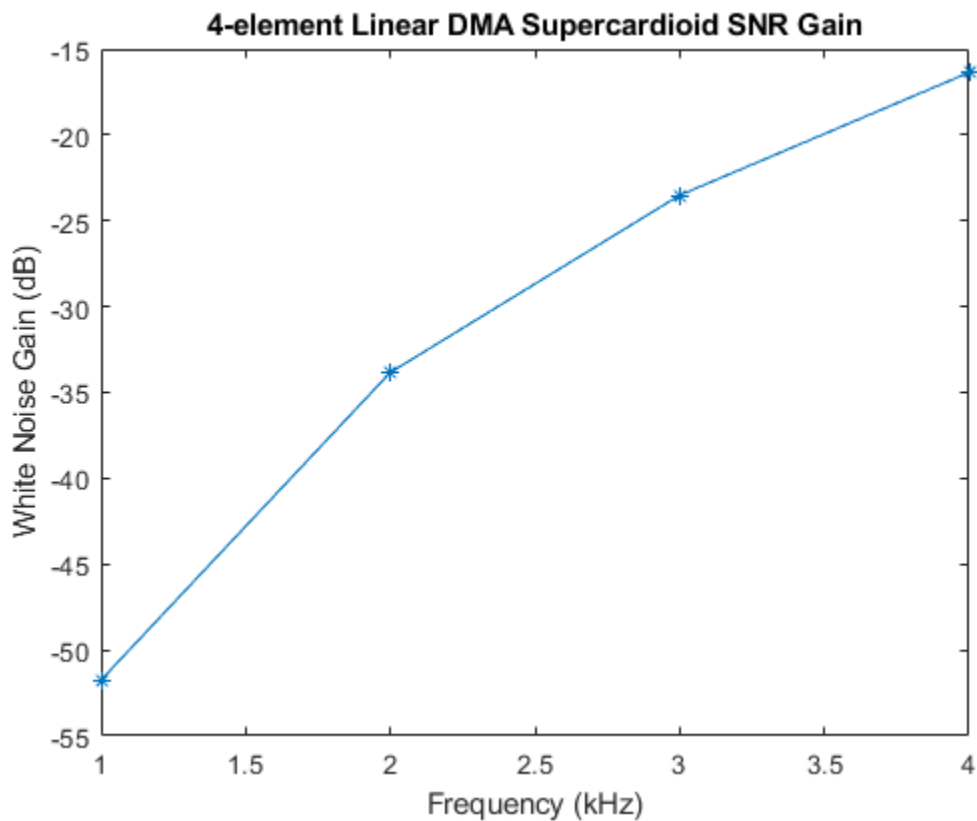
It is possible to assign nulls with multiplicity too. Here is an example where you set the null at -90 degrees to a multiplicity of 2.

```
ang_n = [0 -90 -90];    % nulls
w = complex(zeros(N,numel(f)));
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA');
end

pattern(micarrayd3,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
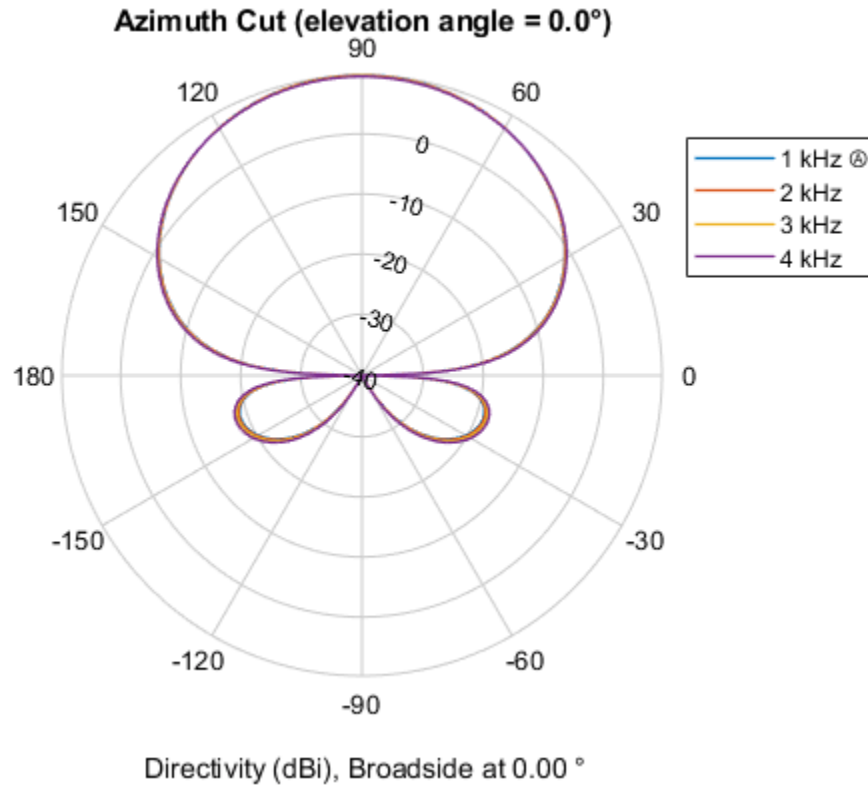
Although the endfire mainlobe of a default DMA works well for hearing aid applications, it is not ideal for other applications such as a sound bar. In those applications, you probably would expect the main beam to be more or less steered toward the broadside of the array. However, although a linear DMA can be steered, it has several unique characteristics compared to an additive array:

1  The steering sacrifices degree of freedoms from the array. For an L-th order linear DMA, you can at most specify L-1 nulls, the remaining null is picked by algorithm based on the specified null locations. Therefore, a first order linear DMA is not steerable.

2  Unlike an additive array, the beam shape of a linear DMA is not preserved when steered.

The following snippet shows how to steer a linear DMA. In this case, the main beam is steered towards the broadside. In addition, a null is placed at 70 degrees off the broadside. The remaining null is derived by the steering weights algorithm.

```
ang_d = 0;    % broadside
ang_n = 70;   % nulls
w = complex(zeros(N,numel(f)));
for m = 1:numel(f)
    w(:,m) = diffbfweights(N,dd_f(m),ang_n,'ArrayGeometry','ULA','SteerAngle',ang_d);
end

pattern(micarrayd3,f,-180:180,0,'PropagationSpeed',c,'Weights',w);
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
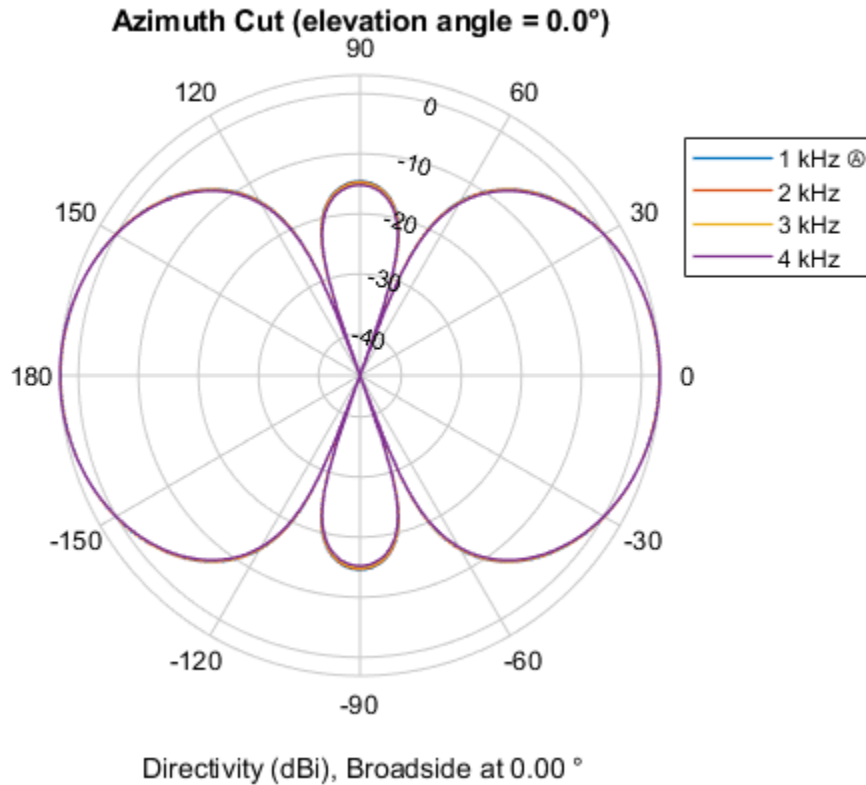
Note that the mainlobe is now toward the broadside direction. Due to the symmetry of the linear array, the resulting pattern also has an equivalent backlobe. In some designs, this can be mitigated by adopting a microphone element with small backlobes.

**Summary**

This example introduced the basic concept of differential microphone arrays. The example showed how to compute the differential beamforming weights to form many classic DMA beam shapes and compared the performance of a linear DMA with a linear additive microphone array. The array beam pattern clearly illustrated that the DMA can provide a frequency invariant pattern. The example ends with a discussion on how to steer a higher order linear DMA.

**References**

[1] Jingdong Chen, Jacob Benesty, and Chao Pan, On the design and Implementation of Linear Differential Microphone Arrays, The Journal of the Acoustical Society of America, Vol. 136, No. 6, 2014

[2] Jacob Benesty, Jingdong Chen, and Chao Pan, Fundamentals of Differential Beamforming, Springer, 2016.

[3] Jilu Jin et al., Steering Study of Linear Differential Microphone Arrays, IEEE/ACM Transactions on Audio Speech, and Language Processing, Vol. 29, 2021

# Examine the Response of a Focused Phased Array

### Introduction

This example introduces the concept of a focused beam, and shows how to use `phased.FocusedSteeringVector` to generate the required element weights for a phased array. It also shows how to use `phased.SphericalWavefrontArrayResponse` to compute the array response of an array at a given angle and range. First you will look at the response when a single beam is formed and examine characteristics of the focal region, then emulate a collection strategy commonly used in ultrasound imaging to see how a focused beam appears in an image.

### Focused Beamforming

### Spherical Wavefront Delay-and-Sum

Beamforming under the typical far-field assumption enables modeling the signal wavefront as a plane which is propagating along its normal direction. When the signal is incident on an array, the wavefront intersects individual elements with a relative time delay that is proportional to the distance of the element along the wave's propagation direction. Under this model, a delay (or phase shift, in the narrowband case) can be applied to each element such that the outputs across elements (whether on transmit or receive) have constructive phase when coherently summed.

Without the far-field assumption, the wavefront emanating from a point source is modeled as a spherical surface centered at that soure. This wavefront intersects the elements of the phased array with relative time delay dictated by the hyperbolic range across colinear elements. While this model would simply add unneccessary computation cost to the generation of a far-field pattern, the spherical wavefront model is necessary to understand near-field beamforming and focusing.

The function `helperPlotULAWavefronts` is provided to demonstrate the difference in element delays and wavefront shape between a steered beam with and without focusing. Element delays are relative to the array center.

```
figure
helperPlotULAWavefronts(14,1e6,1540,20,inf)
title('Steered Wavefront')
```
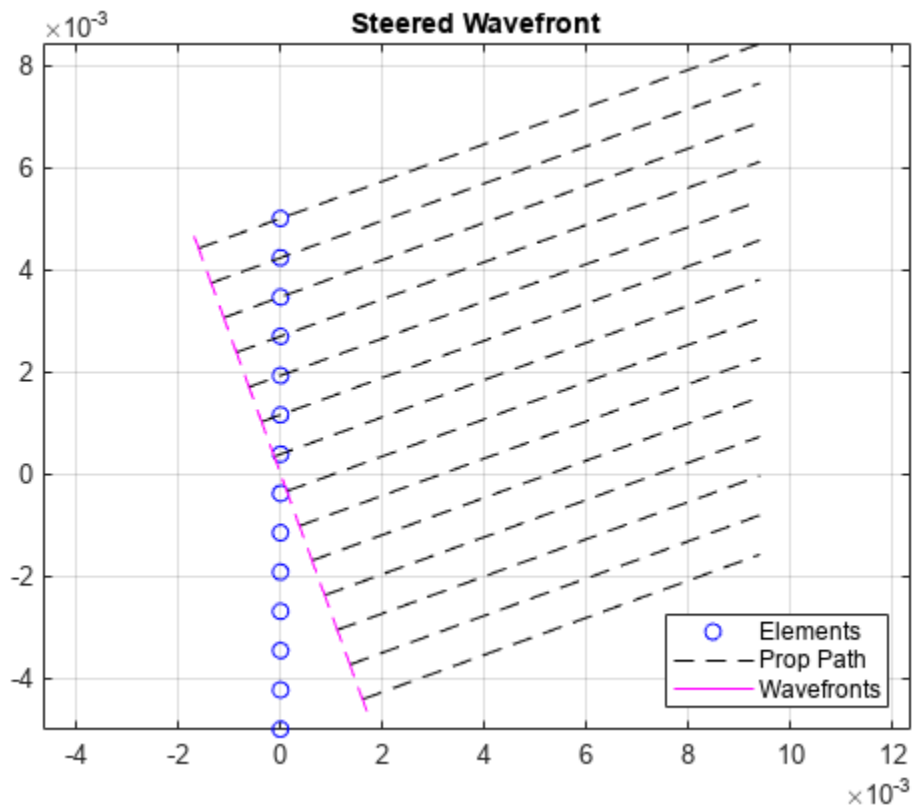
```
figure
helperPlotULAWavefronts(14,1e6,1540,20,0.01)
title('Steered and Focused Wavefront')
```

**Steered and Focused Wavefront**
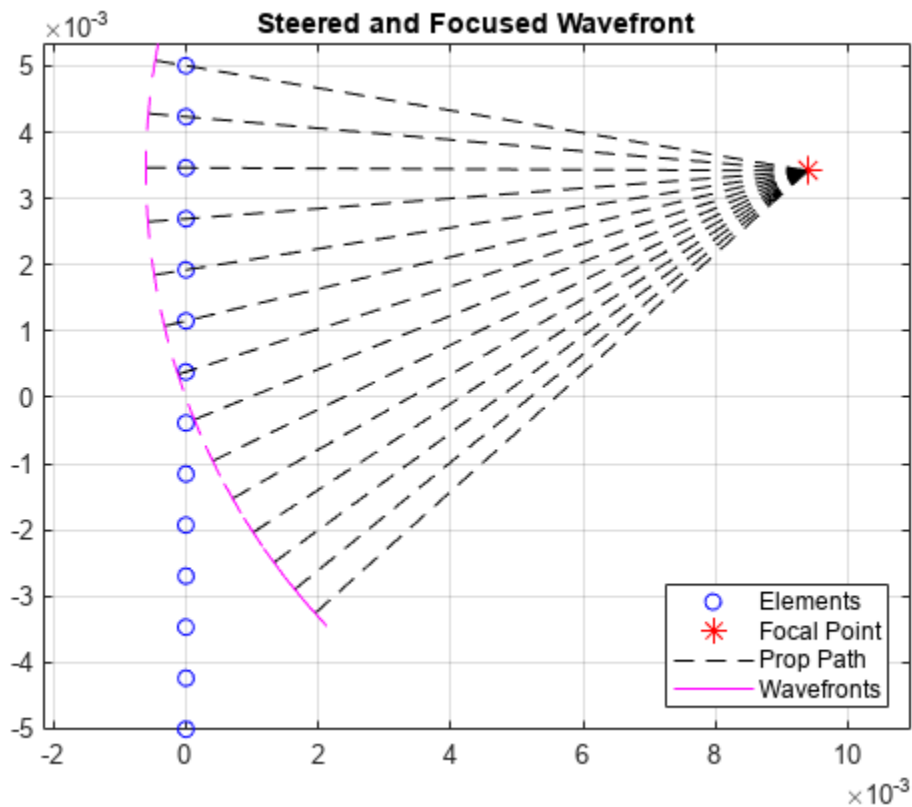
## The Focal Region and Near/Far Boundary

In the far field, a steered array has a well-defined pattern in angle space, which is not dependent on range. In the near field, a steered (but unfocused) array has no discernabe lobe structure at all. Thanks to the nonlinear phase relationship between elements, a response that has equal magnitude at all ranges in a given direction is not possible. Instead, by focusing a beam one can obtain a small region, bounded in both angle and range, in which the response resembles a far-field response, known as the *focal region*. The angular position of the focal region may be controlled as easily as with a far-field beam. The position of the focal region in range is determined by the *focal range*, and the *depth of field* (DoF), which is the range extent of the region.

The figures below shows the progression of beam shape with range for focused uniform and rectangular arrays. The function `helperPlotResponseSlices` is provided to demonstrate how to generate this type of figure.
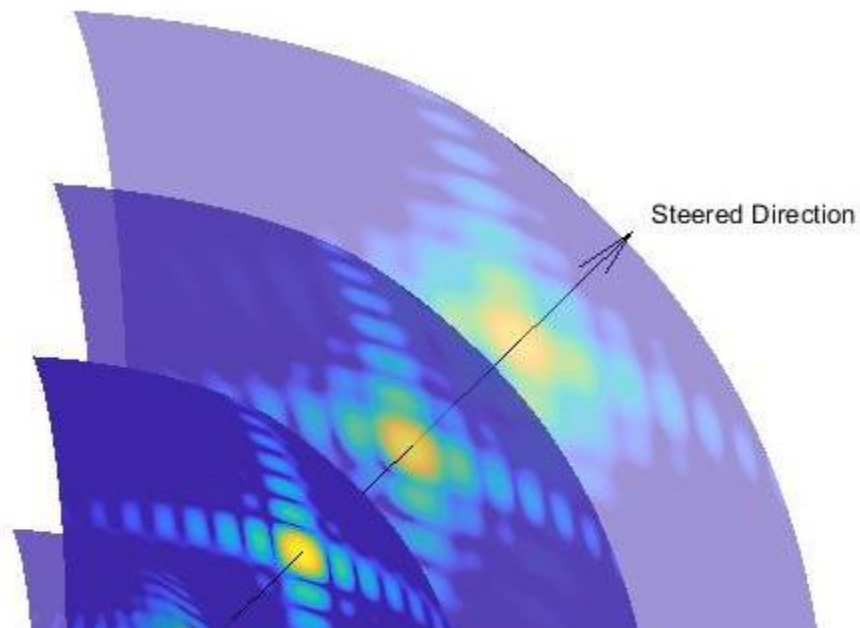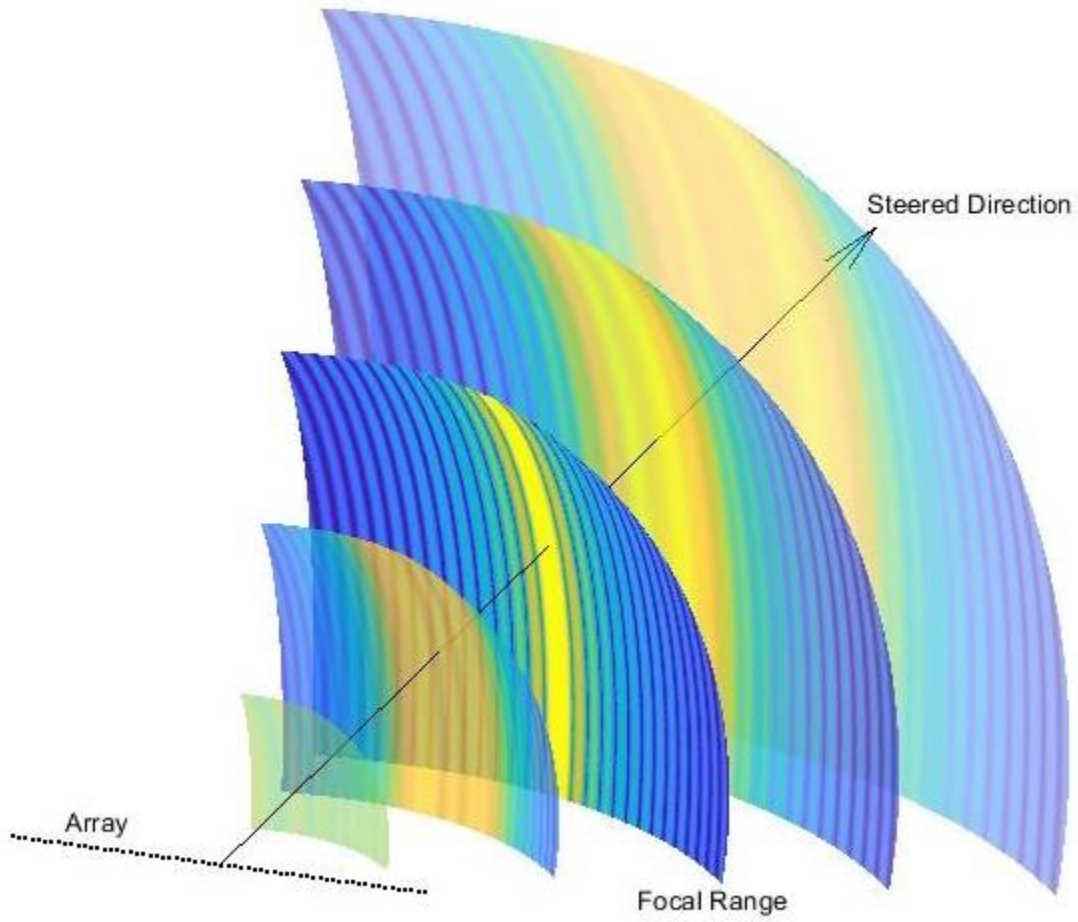
At the focal range, our beam in angle space closely resembles that of a far-field pattern.

The *near/far boundary*, similar in concept to other near-field/far-field boundaries, is the boundary past which focusing is not possible. Conversely, a steered but unfocused beam is not possible on the near side of the boundary. The figure below demonstrates this fact. Notice that the steered beam only begins to take shape on the far side, and the focused beam is only focused on the near side.
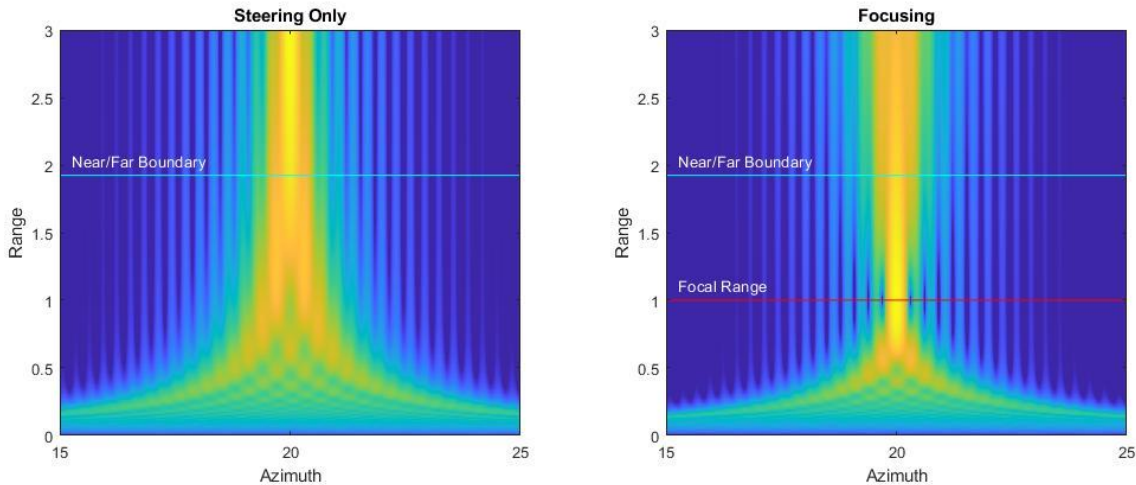


The range where this boundary can be found is well-documented in the medical imaging literature as $L_{\mathrm{array}}^2/4\lambda$, where $L_{\mathrm{array}}$ is the length of the array. For a simple uniform linear array with $N$ critically-spaced elements, this can be expressed as $N^2\lambda/16$.

**Generating the Response for a Single Beam**

This sections shows how to generate and plot a single beam. Start by setting up the phased array and other System objects. Common medical imaging ultrasound systems operate between 2 and 20 MHz, and the commonly-accepted average propagation speed of sound in soft tissue is 1540 m/s.

```
rng('default')
freq = 4e6;
c = 1540;
lambda = c/freq;
```

Ultrasound transducers come in a wide variety of topologies suited for specific activities. This example simply uses a uniform linear array with critical element spacing.

```
numElems = 256;
elemSpacing = lambda/2;
array = phased.ULA(numElems,elemSpacing);
```

The System objects `phased.FocusedSteeringVector` and `phased.SphericalWavefrontArrayResponse` are used in tandem to generate steered and focused element weights, and to compute the response over some domain. Pass the array and propagation speed specification from above to the constructors. For the array response, also turn on the weights input port.

```
SV = phased.FocusedSteeringVector('SensorArray',array,'PropagationSpeed',c);
AR = phased.SphericalWavefrontArrayResponse('SensorArray',array,'PropagationSpeed',c,'WeightsInp
```

Now you have the minimal setup required to form and inspect a focused beam. Use a 10 degree steer in azimuth and a focal range of 40 mm.

```
azSteer = 10;
focalRange = 0.04;
```

Use a domain that starts in front of the array and extends out to twice the focal range, and covers the length of the array in the lateral direction. The array elements lie along the Y axis, and the array normal direction is +X.

```
arrayLength = numElems*elemSpacing;
x = linspace(1e-3,2*focalRange,200);
y = linspace(-arrayLength/2,arrayLength/2,200);
```

Convert to spherical coordinates for input to the steering vector and response computation.

```
[az,el,rng] = cart2sph(x,y',0);
ang = rad2deg([az(:) el(:)]');
rng = rng(:)';
```

Now compute the response of the focused array over the specified domain.

```
weights = SV(freq,[azSteer;0],focalRange);
beam = AR(freq,ang,rng,weights);
```

Reshape, normalize, and use log-scale.

```
beam = reshape(beam,numel(y),numel(x));
beam = beam/max(abs(beam(:)));
beam = mag2db(abs(beam));
```

Use the provided function `helperPlotResponse` to plot the result. The positions of the array elements are indicated by red markers.

```
figure
helperPlotResponse(beam,x,y,array)
title('Single Steered and Focused Beam')
```

**Single Steered and Focused Beam**
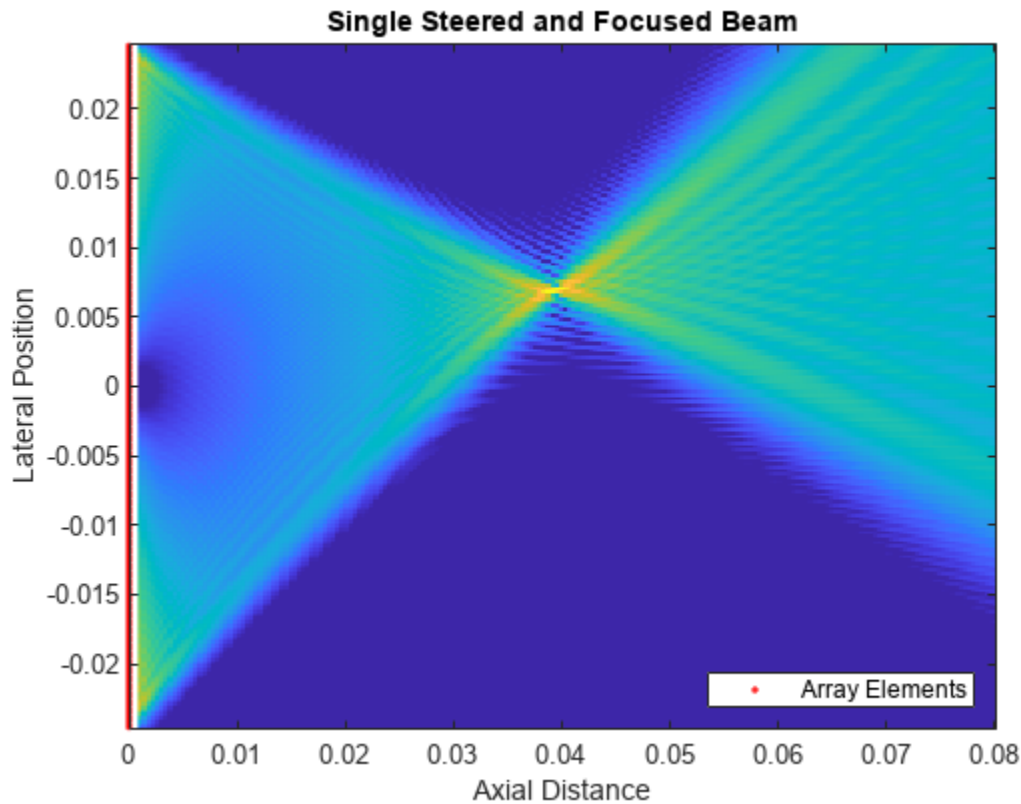
The focal region is clearly visible at the specified range and angle. As with the far-field response (pattern), the magnitude of the spherical wavefront response takes into account the varying phase across elements but, unlike the far-field response, the spherical wavefront response also includes the effect of varying free-space propagation loss across elements. This is essentially a slight amplitude modulation across elements, the effect of which is visible in the beam's sidelobes. A flat gain equal to the focal range is applied to each element, so that the overall amplitude weighting on an element is the ratio of the distance between the response point and the array center to the distance from the response point to the individual element: $R_{\text{resp}}/R_{\text{elem}}$. For example, the contribution to the sum beam from an element located at the origin would have unit magnitude.

**Focal Region**

For a focal region to be bounded in range, the focal range must be small enough that the entire region (with extent determined by the DoF) must be closer than the near/far boundary. DoF can be expressed as a function of a related quantity, commonly seen in optics, known as the *F-number*, which is the ratio of the focal range to the length of the array: $F = R_{\text{foc}}/L_{\text{array}}$. From this quantity, a good estimate of the DoF is $d_F = 7.1\lambda\, F^2$. For a fixed array and frequency, the DoF increases with the square of the focal range.

This section takes a look at how the DoF changes with focal range. Generate the response over an interval of focal ranges, and examine the changing DoF. The function `helperPlotBeamMarkers` will display an indication of the DoF for each focal range. The focal region is roughly elliptical, and is not centered on the focal range. Another rule of thumb can be used to find the offset from the focal range to the center of the region: $d_F/4$.

```
nearField = arrayLength^2/(4*lambda); % Near/far boundary range

x = linspace(1e-3,nearField*1.2,100);
y = linspace(-0.01,0.01,100);

coeff = 0.1:0.1:0.9; % Proportion of near/far boundary
focalRanges = coeff*nearField;

fnum = focalRanges/arrayLength;
dof = 7.1*lambda*fnum.^2;
focalRegionCenter = focalRanges + dof/4;

figure
for ind = 1:numel(focalRanges)
    beam = helperMakeSingleBeam( SV,AR,freq,0,focalRanges(ind),x,y );
    helperPlotResponse(beam,x,y)
    caxis([-6 0]) % only view the greatest 6 dB of the beam
    helperPlotBeamMarkers(focalRanges(ind),focalRegionCenter(ind),nearField,dof(ind),0.001)
    title(sprintf('Focal Range = %d%% of Near/Far Boundary',round(coeff(ind)*100)))
    drawnow
end
```
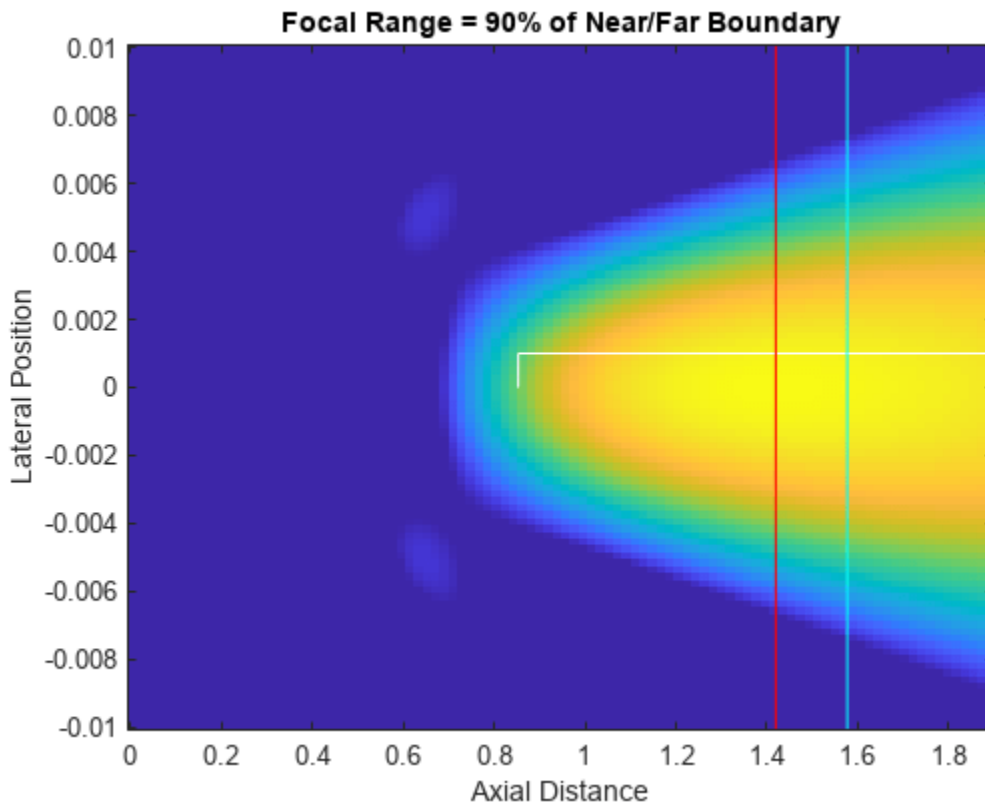


By the time the focal range has increased to 60% of the near/far boundary, the eccentricity of the focal region has become quite large. By 80%, the focal region has intersected the near/far boundary and has essentially become an unfocused beam.

Beamwidth in the focal region, as with a far-field response, may still be roughly computed with the usual $\lambda / L_{\mathrm{array}}$, the ratio of wavelength to array length. Thus the width (in the lateral direction) of the focal region can be approximated with $R_{\mathrm{foc}} \lambda / L_{\mathrm{array}}$.

**Single-Line Acquisition with Linear Subarray Shifting**

**A-scans and Image Formation**

Unlike radar systems, where direction of arrival can be estimated efficiently through beamforming of far-field signals, the near-field beamforming and latency requirements of ultrasound systems often necessitate a simpler strategy to locate the source of returned energy. A common class of collection strategy involves multiple *A-scans*, reflectivity profiles along a given line segment. The placement of these lines within the formed image is determined simply by the known position of the beam.

To form a rectangular image, as is done in this example, *lineary subarray shifting* can be used. With this method a subset of the array elements (a subarray) is used for each pulse, with no steering, to get a range profile originating at the center of the subarray and extending in the axial direction. Successive lines are formed by shifting the subarray selection to a different set of elements.

The choice of focal range can be considered separately from beam pointing. Some systems may keep a fixed focal range, or use dynamic focusing on receive along with apodization or windowing to generate a broad image that covers much of the near-field region. In this example focal range is kept fixed while the lateral position of the subarray is varied.

The helper class `helperSubarray` is provided to emulate subarray selection and simplify the simulation loop. This class keeps track of which elements belong to the current subarray and handles the necessary transforms between the global and subarray frames.

**Image Formation**

In order to demonstrate the effects of focused beamforming, this example uses a simple image formation strategy that constructs each range profile by quantizing and accumulating the response at each pulse, then inserting that profile into the completed image based on the location of the subarray. This simulates an ideal delta pulse in free space, which allows for a comparison of lateral resolution inside and outside the focal region. This method disregards the effects of waveform choice and multipath reflections, and treats scatterers as perfect point isotropic reflectors.

**Simulation**

The same system parameters will be used as in the previous section. Each subarray will consist of 64 elements. The subarray starts at the end of the array and is shifted by one element on each pulse.

```
numSubElems = 64;
subarray = helperSubarray(array,numSubElems);
```

If subarray is shifted by one element on each pulse and all contiguous subarrays are covered, the total number of pulses will be

```
numPulses = numElems - numSubElems + 1
```

```
numPulses = 193
```

Following the analysis in the previous section, check the focal region parameters for this subarray. Get the subarray length, near/far boundary, DoF, and lateral width of the focal region.

```
subarrayLength = numSubElems*elemSpacing;
nearFieldSub = subarrayLength^2/(4*lambda); % Near/far boundary for the subarray
```

```
fnumSub = focalRange/subarrayLength;
dofSub = 7.1*lambda*fnumSub.^2
```

```
dofSub = 0.0288
```

```
widthSub = lambda/subarrayLength*focalRange
```

```
widthSub = 0.0013
```

The subarray beam has a DoF of about 28.8 mm, and the lateral width of the focal region is about 1.3 mm. Check that the focal region is well within the near/far boundary

```
boundedFocalRegion = focalRange + dofSub < nearFieldSub
```

```
boundedFocalRegion = logical
   1
```

To demonstrate the primary usefulness of a focused beam, decreased beamwidth in the focal region, use multiple parallel lines of scatterers along the axial direction (depth lines). Specify the desired max depth of scatterers, and use the provided helper function, `helperGetResponsePoints`, to get the scatterer positions. Let all scatterers have reflectivity with unit amplitude. The scatterer positions are perturbed to avoid artifacting due to symmetry.

For the lateral spacing of the lines, use 4 times the calculated focal region width. Because the subarray is simply shifted by one element at a time, which is a shorter distance than our beam width in the focal region, return from each line of scatterers shows up in more than one row of the image, making them appear to have greater width.

```
maxDepth = nearFieldSub;
lineSpacing = 4*widthSub;
[sx,sy] = helperGetResponsePoints(maxDepth,arrayLength,lambda,lineSpacing);
```

To visualize the scene, plot the scatterer positions along with the array elements.

```
figure
plot(subarray)
hold on
plot(sx,sy,'.')
hold off
legend('Array Elements','Initial Subarray','Scatterers')
xlabel('Axial Distance')
ylabel('Lateral Position')
```
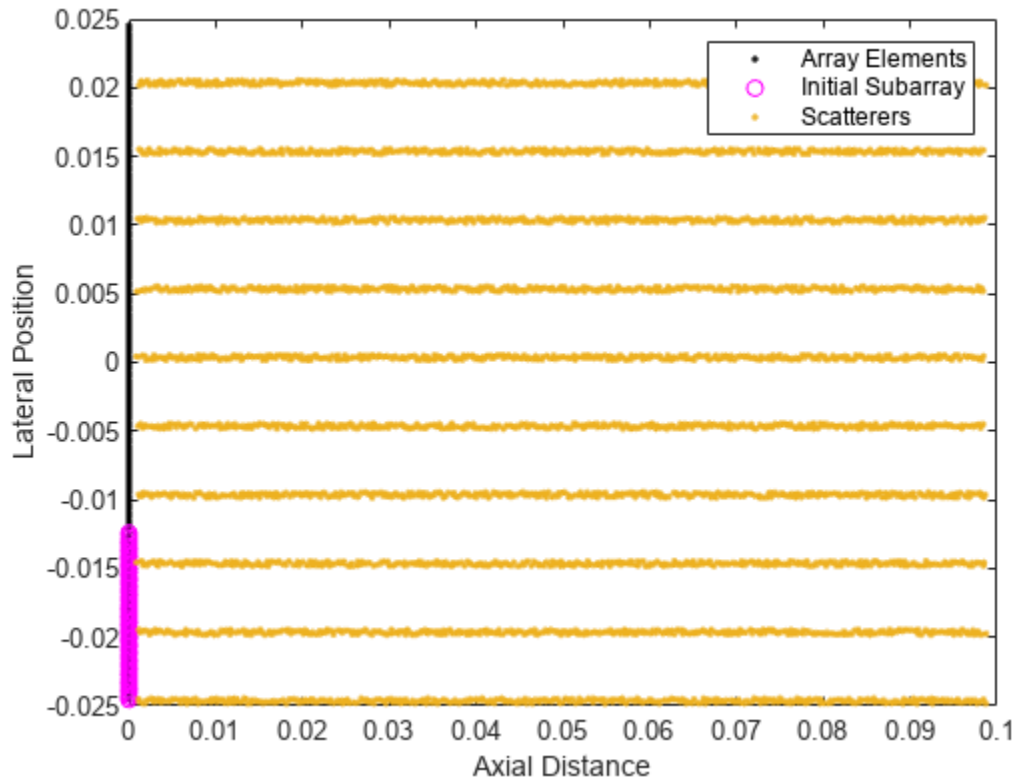
To form a range profile and capture the effects of interference from sidelobe return, range sampling parameters must be defined. Modern ultrasound systems use a relatively high sampling frequency, on the same order as the center frequency of the transmitted waveform. Use a range bin size of 1 mm, corresponding to a sampling rate of about 1.5 MHz. The provided helper function `helperFormRangeProfile` is used to form the range profile from the array response data.

```
rangeBinSize = 1e-3;
Fs = c/rangeBinSize;
rangeBins = 0:rangeBinSize:maxDepth;
numRangeSamples = numel(rangeBins);
```

Get the scatterer positions in spherical coordinates (angles in degrees) for input to `SphericalWavefrontArrayResponse`.

```
[respAng,~,respRng] = cart2sph(sx,sy,0);
respAng = rad2deg(respAng(:)');
respRng = respRng(:)';
```

Each loop of the simulation involves computing weights for the current subarray, computing the response, forming a range profile, and forming the image line by line. We'll also apply range-dependent gain for uniform brightness (see `helperFormRangeProfile`).

```
im = zeros(numPulses,numRangeSamples);
center = zeros(3,numPulses);

for pulse = 1:numPulses
```

```
        center(:,pulse) = subarray.center; % Center position of our current subarray
        [focAzGlobal,~,focRngGlobal] = subarray.localToGlobalSph( 0,0,focalRange ); % Angle and range

        weights = SV(freq,[focAzGlobal;0],focRngGlobal);
        weights(~subarray.selection) = 0; % Zero-out weights for unused elements

        resp = AR(freq,respAng,respRng,weights);
        resp = resp./respRng(:); % Undo normalization to use actual propagation loss

        im(pulse,:) = helperFormRangeProfile(resp,sx,sy,center(:,pulse),rangeBins); % Add line to ima

        if pulse < numPulses
            subarray.shift(1) % Shift subarray if there are pulses remaining
        end

    end
```

Normalize and plot the image, along with an overlay of the scatterer positions.

```
im = im/max(abs(im(:)));
figure
subplot(1,2,1)
helperPlotResponse(mag2db(abs(im)),rangeBins,center(2,:))
title('Linear Subarray Shift Image')
subplot(1,2,2)
helperPlotResponse(mag2db(abs(im)),rangeBins,center(2,:))
hold on
plot(sx,sy,'.r','markersize',1)
hold off
title('Scatterer Overlay')
set(gcf,'Position',get(gcf,'Position')+[0 0 560 0]);
```
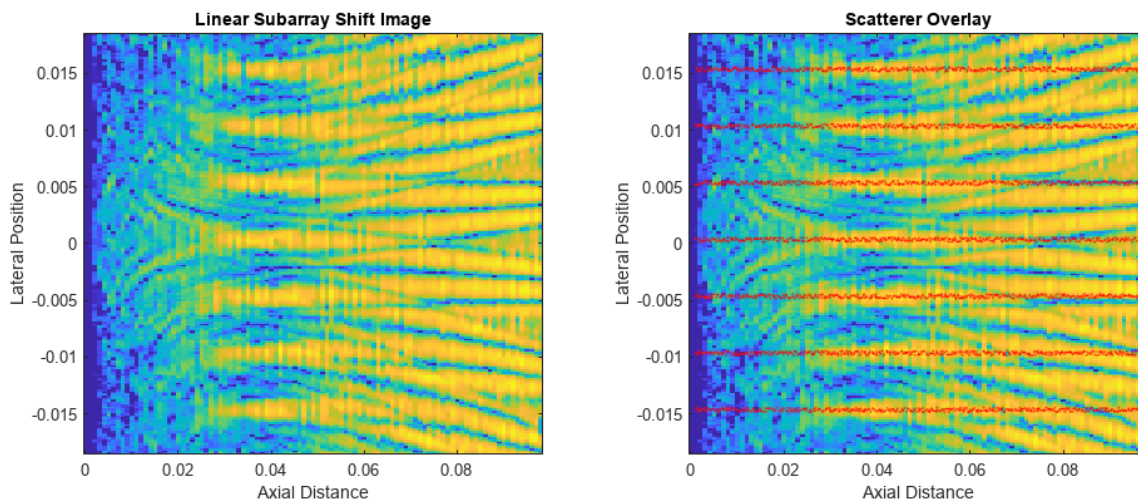


The depth lines are resolvable about the focal range, over a range interval roughly equal to the computed DoF for the subarray beam. Away from the focal point, the parallel lines of scatterers quickly become indistinguishable thanks to the wide angular spread of the beam outside the focal region.

Note that, though all the scatterers were used to compute energy return, not all lines are visible due to the clipping between the full array size and the extent of subarray center positions. A wider total field of view would be obtainable with a shorter subarray, at the cost of reduced lateral resolution.

**Conclusion**

This example introduced two System objects for computing focused weights and for computing the non-far-field response of an array with spherical wavefronts. It showed how to examine some basic characteristics of a focused beam, and how to generate a basic image to visualize the effect of a focused beam on lateral resolution with a linear subarray shift collection method.

**References**

[1] Demi, Libertario. "Practical Guide to Ultrasound Beam Forming: Beam Pattern and Image Reconstruction Analysis." *Applied Sciences* 8, no. 9 (September 3, 2018): 1544. https://doi.org/10.3390/app8091544

[2] Ramm, O. T. Von, and S. W. Smith. "Beam Steering with Linear Arrays." *IEEE Transactions on Biomedical Engineering* BME-30, no. 8 (August 1983): 438–52.

**Helper Functions**

**helperMakeSingleBeam**

```
function [beam,x,y] = helperMakeSingleBeam( SV,AR,freq,azSteer,focalRange,x,y )
% Get the element weighting for a single beam and compute the response

weights = SV(freq,[azSteer;0],focalRange);

% Domain of response
if nargin < 6
    x = linspace(1e-3,2*focalRange,200);
end
if nargin < 7
    pos = SV.SensorArray.getElementPosition;
    halfy = max(pos(2,:))*1.2;
    y = linspace(-halfy,halfy,200);
end
[az,el,rng] = cart2sph(x,y',0);
ang = rad2deg([az(:) el(:)]');
rng = rng(:)';

% Generate response
beam = AR(freq,ang,rng,weights);
beam = reshape(beam,numel(y),numel(x));

% Normalize and use log scale
beam = beam/max(abs(beam(:)));
beam = mag2db(abs(beam));

end
```

**helperPlotResponse**

```
function helperPlotResponse(R,x,y,array)
% Plot the response, R, on the domain defined by x and y.

imagesc(x,y,R)
```

```
set(gca,'ydir','normal')
caxis([-32 0])
xlabel('Axial Distance')
ylabel('Lateral Position')

if nargin > 3
    pos = getElementPosition(array);
    hold on
    h = plot(pos(1,:),pos(2,:),'.r');
    hold off
    xl = xlim;
    xlim([min(pos(1,:)) xl(2)])
    legend(h,'Array Elements','Location','southeast','AutoUpdate','off')
end

end
```

**helperPlotBeamMarkers**

```
function helperPlotBeamMarkers(focalRange,center,nearField,dof,offset)
% Put informative markers on a beam plot

line([center-dof/2 center+dof/2],[offset offset],'color','white')
line([center-dof/2 center-dof/2],[0 offset],'color','white')
line([center+dof/2 center+dof/2],[0 offset],'color','white')

line([focalRange focalRange],ylim,'color','red')
line([nearField nearField],ylim,'color','cyan')

end
```

**helperGetResponsePoints**

```
function [sx,sy] = helperGetResponsePoints( maxDepth,arrayLength,lambda,dy )
% Make parallel lines of scatterers along X

sx = linspace(0.001,maxDepth,400);
sy = -arrayLength/2:dy:arrayLength/2;

[sx,sy] = meshgrid(sx,sy);
sx = sx(:);
sy = sy(:);

sx = sx + (rand(size(sx))-1/2)*lambda;
sy = sy + (rand(size(sy))-1/2)*lambda;

end
```

**helperFormRangeProfile**

```
function rangeProf = helperFormRangeProfile(resp,sx,sy,center,rangeBins)
% This helper function quantizes a response in range, coherently
% accumulates the return in each bin, and applies amplitude weighting per
% range bin

rangeBinSize = rangeBins(2) - rangeBins(1);
numRangeSamples = numel(rangeBins);

% Range of scatterers relative to subarray center
```

```
scatRngRel = sqrt((center(1)-sx).^2 + (center(2)-sy).^2);

% Quantize scatterer ranges into fast-time sampling vector
scatRidx = 1 + floor(scatRngRel/rangeBinSize);

% Only keep samples below max depth
I = scatRidx <= numRangeSamples;
scatRidx = scatRidx(I);
resp = resp(I);

% Accumulate return into fast-time sampling grid
rangeProf = accumarray(scatRidx,resp,[numRangeSamples 1]);
rangeProf = rangeProf';

% Apply range-dependent gain
rangeProf = rangeProf.*rangeBins;

end
```

**helperPlotULAWavefronts**

```
function helperPlotULAWavefronts( numElems,f,c,az,r )
% Plot the wavefronts for the given ULA with ArrayAxis 'y', for the given
% azimuth angle and focal range.
%
% For the far-field wavefront, use inf for focal range

lambda = c/f;
array = phased.ULA(numElems,lambda/2);
pos = getElementPosition(array);
arrayLength = max(pos(2,:)) - min(pos(2,:));

% get relative path lengths
if isinf(r)
    L = phased.internal.elemdelay(pos,c,[az;0])*c;
else
    L = phased.internal.sphericalelemdelay(pos,c,[az;0],r)*c;
end

% plot element positions
plot(pos(1,:),pos(2,:),'oblue');
hold on;

% if near field, plot source
if ~isinf(r)
    [src(1,1),src(2,1),src(3,1)] = sph2cart(az*pi/180,0,r);
    plot(src(1),src(2),'*r','markersize',10);
end

% wavefront marker width
s = lambda/6;

% far field prop path
if isinf(r)
    [los(1,1),los(2,1),los(3,1)] = sph2cart(az*pi/180,0,1);
end

for ind = 1:size(pos,2) % for each element
```

```
        p = pos(:,ind);

        if isinf(r)
            src = p + los*arrayLength;
        end

        path = src - p;
        path = path/norm(path);

        wp = p + path*L(ind); % position of wavefront

        % prop path
        line([wp(1) src(1)],[wp(2) src(2)],'color','black','linestyle','--');

        % wavefront marker
        u = cross([0;0;1],path)*s;
        line([wp(1)-u(1) wp(1)+u(1)],[wp(2)-u(2) wp(2)+u(2)],'color','magenta');

    end

    hold off;
    grid on;
    axis equal;

    if isinf(r)
        legend('Elements','Prop Path','Wavefronts','Location','SouthEast');
    else
        legend('Elements','Focal Point','Prop Path','Wavefronts','Location','SouthEast');
    end

end
```

**helperPlotResponseSlices**

```
function helperPlotResponseSlices
% Demonstrates how to visualize range slices of a spherical wavefront response

f = 2e6;
c = 1540;
lambda = freq2wavelen(f,c);

array = phased.URA([32 32],lambda/2);
elemPos = array.getElementPosition;

focalRange = 0.03;
sampleRanges = .01:.01:.05;

% domain of each slice
azSteer = -20;
elSteer = 20;
az = azSteer + (-30:.1:30);
el = elSteer + (-30:.1:30);
[az,el] = meshgrid(az,el);
ang = [az(:) el(:)]';
[x,y,z] = sph2cart(az*pi/180,el*pi/180,1);

SV = phased.FocusedSteeringVector('SensorArray',array,'PropagationSpeed',c);
```

```matlab
AR = phased.SphericalWavefrontArrayResponse('SensorArray',array,'PropagationSpeed',c,'WeightsInpu

w = SV(f,[azSteer;elSteer],focalRange);

for ind = 1:numel(sampleRanges)
    resp = AR(f,ang,sampleRanges(ind),w);
    resp = resp / array.getNumElements;
    resp = reshape(resp,size(x));
    alpha = 1 - (abs(sampleRanges(ind) - focalRange)/(max(sampleRanges)-min(sampleRanges))); % t

    surf(x*sampleRanges(ind),y*sampleRanges(ind),z*sampleRanges(ind),mag2db(abs(resp)),'FaceAlpha
    hold on
    shading flat
end

% plot element positions and boresight vector
caxis([-32 0])
plot3(elemPos(1,:),elemPos(2,:),elemPos(3,:),'.black','markersize',4);
[b(1),b(2),b(3)] = sph2cart(azSteer*pi/180,elSteer*pi/180,max(sampleRanges)*1.2);
quiver3(0,0,0,b(1),b(2),b(3),'black','autoscale','off')
axis equal
axis off
hold off
set(gca,'view',[-70 22])

end
```

# Simulate and Visualize Range and Doppler Intensity Data in an End-to-end Radar System

### Introduction

This example demonstrates how to model an end-to-end radar system and use the scopes to visualize range-intensity and Doppler-intensity data over time in Simulink®. The range-time intensity and Doppler-time intensity scope help to acquire the target in the environment, by showing how the target's range and Doppler change over time.

### Exploring the example

The following example shows an end-to-end simulation of a monostatic radar system. The system is divided into following sections: the transmitter, the platform subsystem, the propagation channel, the targets, the receiver subsystem and the visualization. The flow of signal is as shown starting from the transmitter, through the channel and target and reflected to receiver. The received data is then processed in range and Doppler domain independently to obtain the range-intensity and Doppler-intensity data for visualization.



Monostatic Radar with multiple targets
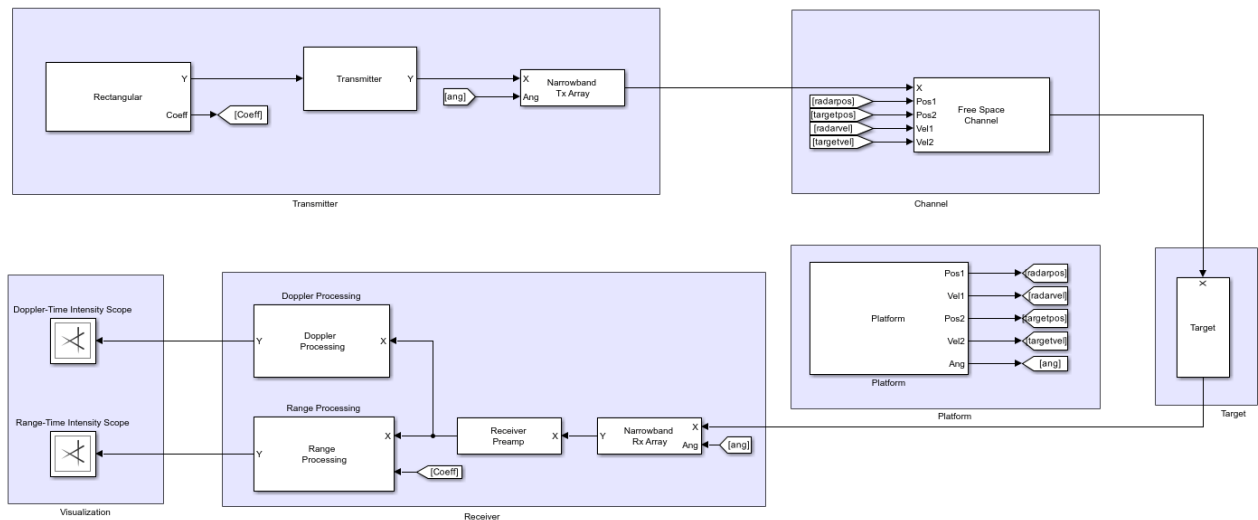
Copyright 2021 The Mathworks Inc.

### Transmitter

**`Rectangular`** - Creates rectangular pulse as the transmitter waveform of 3 MHz bandwidth, corresponding to a 50-meter range resolution.

**`Radar Transmitter`** - Amplifies the pulse. The operating frequency of the transmitter is 1 GHz.

**`Narrowband Tx Array`** - Simulates the signals radiated from the antenna. The first input (X) to this block is a column vector which contains the transmitted pulses. The pulses are assumed to be narrowband with a carrier frequency equal to the operating frequency specified in the block's dialog panel. The second input (Ang) specifies the radiating direction of the pulses. The antenna is configured in `Sensor Array` tab of the block's dialog panel.

### Platform

The platform subsystem models the motion of radar as well as the target. The positions and velocities of the radar and targets are used by the Channel block to model propagation and by the Range Angle block to compute the signal's incident angles at the target location.

`Radar Platform` - Used to model the radar's motion. The radar is mounted on a stationary platform located at the origin.

`Target Platform` - Used to model the target's motion. In this example, there are three targets. The first one positioned at approximately 2000 meters from the radar and is moving away from the radar at 1500 m/s, second target positioned at 3845 meters from the radar and is moving towards the radar at 1500 m/s and third one is a stationary target at 3532 meters from the radar.

`Range Angle` - Computes the propagated signal's incident angles in azimuth and elevation at the target's location.
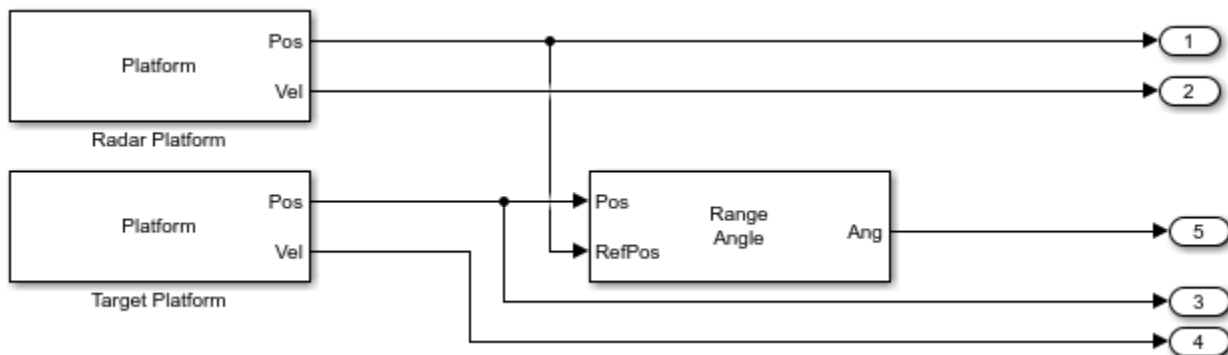


### Channel and Targets

`Channel` - This block performs two-way propagation, propagating signal from transmitter to target and propagating the reflected signal from target to receiver. The signal output has three columns one for two-way propagation path of each target.

`Targets` - Reflects the signal according to the radar cross section (RCS) specified in the mean radar cross section parameter of the Target block.

### Receiver

`Narrowband Rx Array` - Simulates the signals received at the antenna. The first input to this block is a matrix which contains the received pulses from each of the target. The pulses are assumed to be narrowband with a carrier frequency equal to the operating frequency specified in the block's dialog panel. The second input (Ang) specifies the incident direction of the pulses. The antenna is configured in `Sensor Array` tab of the block's dialog panel.

`Receiver Preamp` - Receives the target echoes and adds noise.

`Range Processing Subsystem` - Performs matched filtering to improve SNR, followed by time-varying gain to compensate for range loss and finally integrates 10 pulses noncoherently.

**Doppler Processing Subsystem** - The received signal is buffered for 10-pulse burst, followed by FFT along the Doppler dimension to estimate the Doppler information of the received pulse.



**Visualization**

**Range-Time Intensity Scope** - This scope is used to visualize the range-intensity data over time. The scope is configured prior to the simulation start to visualize the target's range variation over time. The range processed data is input to the block.

**Doppler-Time Intensity Scope** - This scope is used to visualize the Doppler-intensity data over time. The scope is configured prior to the simulation start to visualize the target's radial speed over time. The Doppler processed data is input to the block.
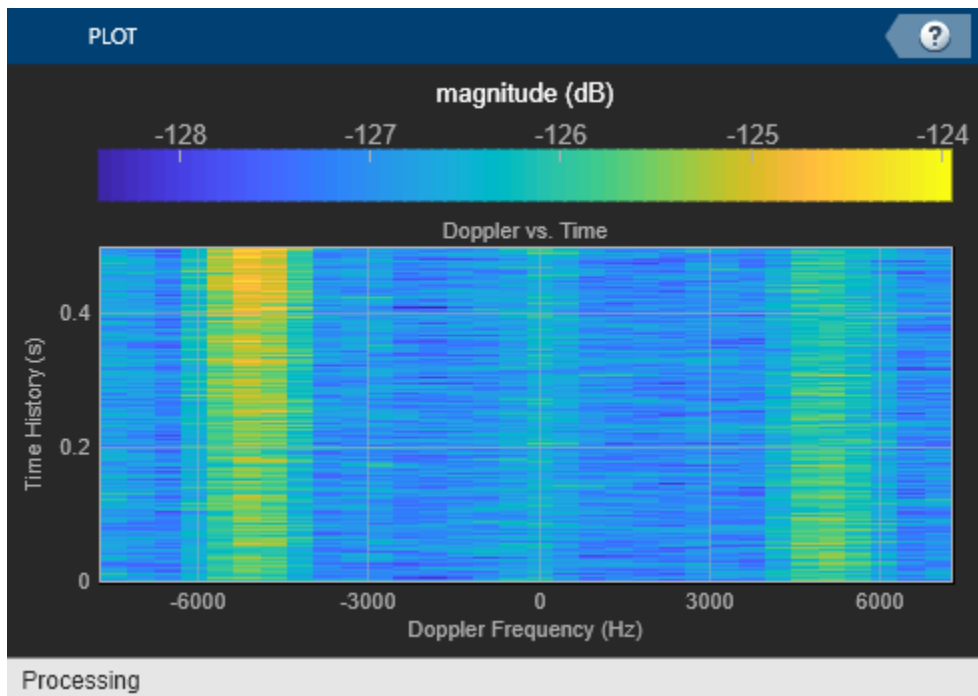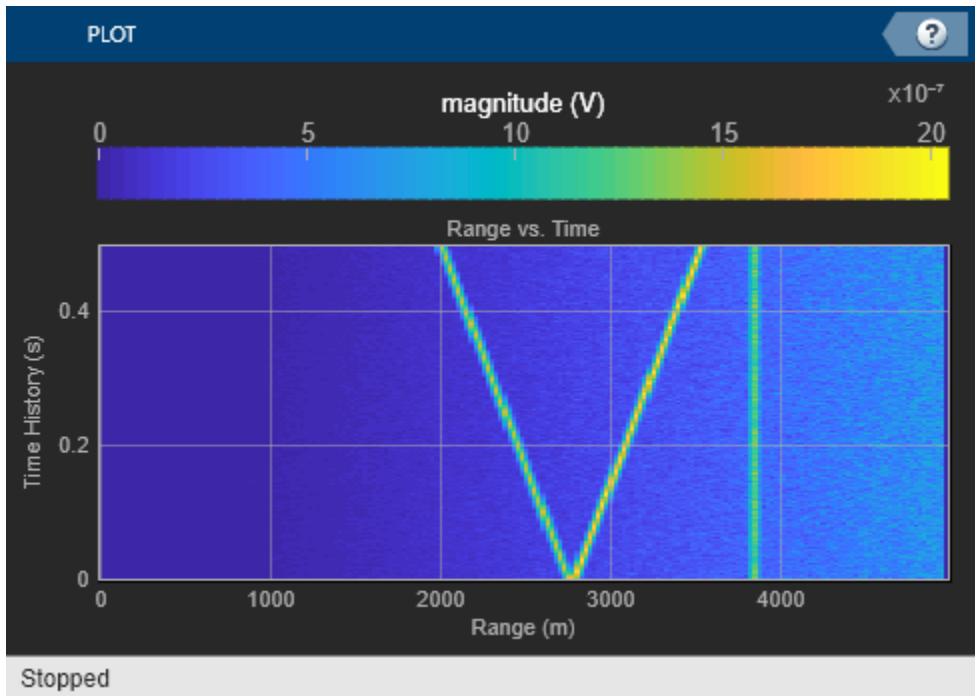
**Exploring the model**

Several dialog parameters of the model are calculated by the helper function helperslexRadarSystemVisualParam. To open the function from the model, click on Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

**Results and Displays**

The range-time intensity scope and Doppler-time intensity scope shows range and Doppler detections of three targets over time. From the Range-Time Intensity scope and Doppler-Time Intensity scope it can be seen that there are three targets, out of which two are moving along the x-axis and one is stationary.
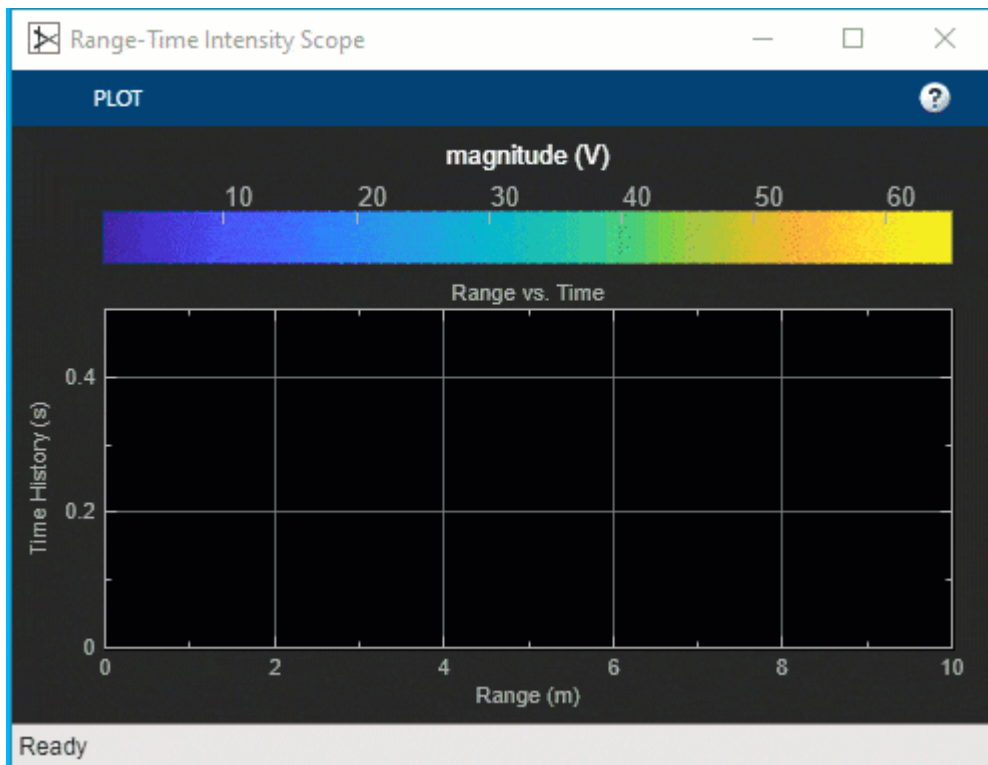
The Range-Time Intensity scope shows the variation in the range intensity data of each of the targets over time. Two targets are moving along the x-axis starting at 2000 km and 3500 km respectively and one target is stationary at 4000 km.
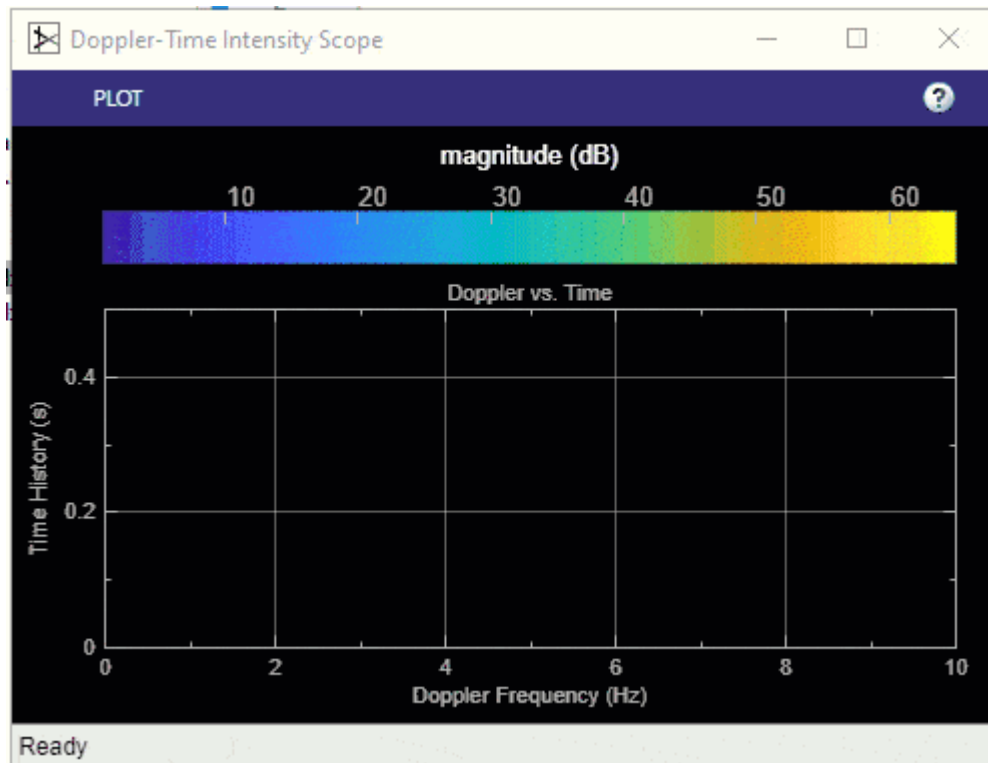
The Doppler-Time Intensity scope shows that there are 2 moving targets and one stationary target which agrees with the target velocity configuration.

**Summary**

This example demonstrates how an end-to-end radar system can be modeled within Simulink and how to visualize the range and Doppler intensity data over time using the Range-Time Intensity scope and Doppler-Time Intensity scope respectively.

# Interference from Satellite Constellation on Communications Link

This example shows how to analyze interference on a downlink from a constellation of satellites in a medium-Earth orbit (MEO) to a ground station located in the Pacific Ocean. The interfering constellation consists of 40 satellites in a low-Earth orbit (LEO). This example determines the times at which the downlink is closed, the carrier to noise plus interference ratio, and the link margin.

This example requires Satellite Communications Toolbox. If you have also have Antenna Toolbox™, you can use this example to learn how to import antennas from Antenna Toolbox into satellite scenario. If you also have Phased Array System Toolbox™, you can use this example to learn how to import antennas from Phased Array System Toolbox into satellite scenario and use beamforming to improve the carrier-to-noise-and-interference ratio (CNIR).

### Create Satellite Scenario

Create a satellite scenario. Define the start time and stop time of the scenario. Set the sample time to 60 seconds.

```
startTime = datetime(2021,3,17,22,52,0);          % 17 March 2021 10:52 PM UTC
stopTime = startTime + minutes(10);               % 17 March 2021 11:02 PM UTC
sampleTime = 60;                                  % In s
sc = satelliteScenario(startTime,stopTime,sampleTime);
```

### Add Medium-Earth Orbit Satellite

Add a satellite in an MEO by specifying its Keplerian orbital elements. This satellite is the satellite from which data is downlinked.

```
semiMajorAxis = 12000000;                 % In m
eccentricity = 0;
inclination = 8;                          % In degrees
raan = 0;                                 % Right ascension of ascending node, in degrees
argOfPeriapsis = 0;                       % In degrees
trueAnomaly = 343.9391;                   % In degrees
meoSat = satellite(sc, semiMajorAxis, ...
    eccentricity, ...
    inclination, ...
    raan, ...
    argOfPeriapsis, ...
    trueAnomaly, ...
    Name = "MEO Satellite", ...
    OrbitPropagator = "two-body-keplerian");
```

### Add Interfering Satellite Constellation

Add the interfering satellite constellation from a two-line-element (TLE) file. These satellites are placed in LEO.

```
interferingSat = satellite(sc,"leoSatelliteConstellation.tle");
```

### Add Transmitter to MEO Satellite

Add a transmitter to the MEO satellite. This transmitter is used for the downlink. Define the antenna specifications and set the operating carrier frequency to 3 GHz.

```
txMEOFreq = 3e9;                    % In Hz
txMEOSat = transmitter(meoSat, ...
    Frequency = txMEOFreq, ...      % In Hz
    Power = 11);                    % In dBW
gaussianAntenna(txMEOSat, ...
    DishDiameter = 1);              % In m
```

**Add Transmitter to LEO Satellites**

Add a transmitter to each satellite in the LEO constellation, and then define the antenna specifications. These transmitters are the ones that interfere with the downlink from the MEO satellite. Set the operating carrier frequency of the interfering satellites to 2.99 GHz. The example assigns each interfering satellite a random power in the range from 10 to 20 dBW.

```
interferenceFreq = 2.99e9;                          % In Hz
rng("default");
txInterferingSat = transmitter(interferingSat, ...
    Frequency = interferenceFreq, ...               % In Hz
    Power = 10+10*rand(1,numel(interferingSat)));   % In dBW
gaussianAntenna(txInterferingSat, ...
    DishDiameter = 0.2);                            % In m
```

**Add Ground Station**

Add a ground station to the satellite scenario by specifying its latitude and longitude.

```
gs = groundStation(sc, ...
    0, ...                      % Latitude in degrees
    180, ...                    % Longitude in degrees
    Name = "Ground station");
```

**Specify Ground Station Antenna Type**

For this example, you can choose from one of the following antennas:

- Gaussian Antenna
- Parabolic Reflector from Antenna Toolbox
- Uniform Rectangular Array from Phased Array System Toolbox

```
% Select the desired ground station antenna.
```

groundStationAntennaType = `Gaussian Antenna ▼` ;

**Add Receiver to Ground Station**

Add a receiver to the ground station. If you selected Gaussian Antenna or Parabolic Reflector, attach the receiver to a gimbal, which is in turn attached to the ground station. Configure the gimbal to track the MEO satellite, so that the antenna also tracks the MEO satellite. If you selected Uniform Rectangular Array, attach the receiver directly to the ground station. Specify the mounting location and mounting angles of the gimbal and receiver, and the antenna specifications appropriately.

```
switch groundStationAntennaType
    case {"Gaussian Antenna","Parabolic Reflector"}
        % When Gaussian Antenna or Parabolic Reflector is selected, attach
        % a gimbal to the ground station.
        gim = gimbal(gs, ...
            MountingLocation = [0;0;-5], ... % In m
```

```
                MountingAngles = [0;180;0]);     % In degrees

        % Set the gimbal to track the MEO satellite.
        pointAt(gim,meoSat);

        if groundStationAntennaType == "Gaussian Antenna"
            % When Gaussian Antenna is selected

            % Create the receiver object and add it to the gimbal
            rxGs = receiver(gim, ...
                MountingLocation = [0;0;1]); % In m

            % Provide the Gaussian Antenna specifications
            gaussianAntenna(rxGs, ...
                DishDiameter = 0.8);          % In m
        else
            % When Parabolic Reflector is selected

            % Size the antenna based on the frequency
            % Requires Antenna Toolbox(TM)
            ant = design(reflectorParabolic,txMEOFreq);

            % Create the receiver object and add it to the gimbal
            rxGs = receiver(gim, ...
                Antenna = ant, ...
                MountingLocation = [0;0;1]); % In m
        end
    case "Uniform Rectangular Array"
        % When Uniform Rectangular Array is selected

        % Determine the wavelength of the downlink signal
        c = physconst('LightSpeed');
        lambda = c/txMEOFreq;

        % Define array size
        nrow = 8;
        ncol = 8;

        % Define element spacing
        drow = lambda/2;
        dcol = lambda/2;

        % Create a back-baffled 6-by-6 antenna array
        % Requires Phased Array System Toolbox(TM)
        ant = phased.URA(Size = [nrow ncol], ...
            ElementSpacing = [drow dcol]);
        ant.Element.BackBaffled = true;

        % Create the receiver object and add it to the ground station
        rxGs = receiver(gs, ...
            Antenna = ant, ...
            MountingAngles = [0;90;0]); % In degrees
end
```

**Create Access Analysis Between Interfering Satellite Constellation and Ground Station**

Add an access analysis between each satellite in the interfering constellation and the ground station. This analysis enables the visualization of interference in the satellite scenario viewer that will be

launched later. Any time a satellite in the constellation is visible to the ground station, there is some level of interference from that visible satellite.

```
ac = access(interferingSat,gs);
ac.LineColor = [1 1 0];                % Yellow
```

**Set Tracking Targets for Satellites**

Set the satellites to track the ground station. This ensures that the transmitter antennas on board each satellite track the ground station. Setting the interfering satellite transmitters to track the ground station results in the worst-case interference on the downlink.

```
pointAt([meoSat interferingSat],gs);
```

**Calculate Weights of Uniform Rectangular Array**

If you selected Uniform Rectangular Array as the ground station antenna, compute the weights that are required to point the main lobe towards the MEO satellite, and the nulls towards the interfering satellites, thereby cancelling the interference. Assign the computed weights using `pointAt`.

```
if groundStationAntennaType == "Uniform Rectangular Array"
    % Find the LEO satellites that are in the line of sight of the ground
    % station. These satellites are the potential interferers.
    currentInterferingSat = interferingSat(accessStatus(ac,sc.StartTime) == true);

    % Calculate the direction of the MEO satellite with respect to the
    % array. This is the lookout direction.
    [azd,eld] = aer(rxGs,meoSat,sc.StartTime,CoordinateFrame='body');

    % Calculate the directions of the potentially interfering satellites
    % with respect to the array. These are the null directions.
    [azn,eln] = aer(rxGs,currentInterferingSat,sc.StartTime,CoordinateFrame='body');

    % Calculate the steering vectors for the lookout direction.
    % Requires Phased Array System Toolbox.
    wd = steervec(getElementPosition(ant)/lambda,[wrapTo180(azd);-eld]);

    % Calculate the steering vector for null directions.
    % Requires Phased Array System Toolbox.
    wn = steervec(getElementPosition(ant)/lambda,[wrapTo180(azn)';-eln']);

    % Compute the response of the desired steering at null directions.
    rn = (wn'*wn)\(wn'*wd);

    % Sidelobe canceler - remove the response at null directions.
    w = wd-wn*rn;

    % Assign the weights to the phased array.
    pointAt(rxGs,Weights=w);
end
```

**Create Desired Downlink**

Create a downlink from the transmitter on board the MEO satellite to the receiver on board the ground station. This link is the downlink which encounters interference from the LEO constellation.

```
downlink = link(txMEOSat,rxGs);
```

**Create Interfering Links**

Create a link between the transmitter on board each satellite in the LEO constellation and the receiver on board the ground station. These links are the interferer links with the desired downlink.

```
lnkInterference = link(txInterferingSat,rxGs);
```

**Launch Satellite Scenario Viewer**

Launch the Satellite Scenario Viewer with `ShowDetails` set to false. When the `ShowDetails` property is set to `false`, only the satellites, the ground station, accesses, and links will be shown. The labels and orbits will be hidden. Mouse over the satellites and the ground stations to show their labels. Click on the MEO satellite so that its orbit projected up to the scenario `StopTime` and its label are visible without mousing over. Click on the ground station so that its label is visible without mousing over. The presence of the green line between the transmitter on board the MEO satellite and the receiver on board the ground station signifies that the downlink can be closed successfully assuming no interference from the satellite constellation exists. The presence of yellow lines between a given satellite in the constellation and the ground station signifies that they have access to one another, and as a result, interference from that satellite exists.

```
v = satelliteScenarioViewer(sc,ShowDetails=false);
```
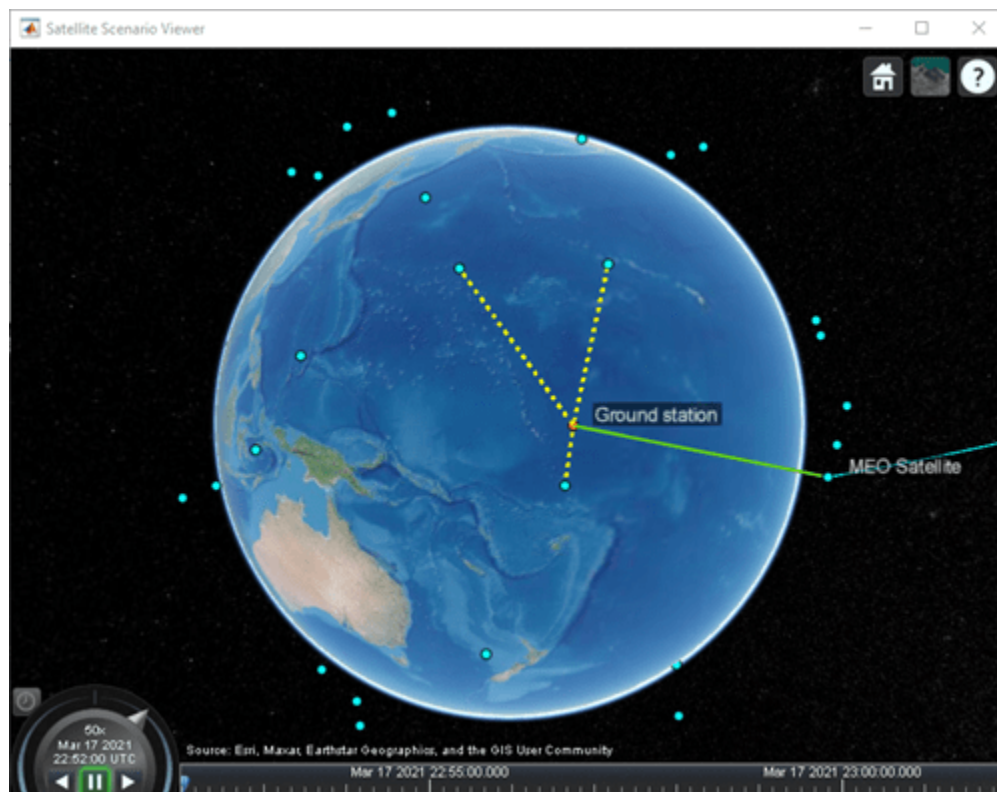


**Visualize Radiation Pattern of Antennas Involved in Downlink**

Visualize the radiation pattern of the transmitter antenna on board the MEO satellite and the receiver on board the ground station.

```
pattern(txMEOSat, ...
    Size = 1000000);         % In m
```

```
pattern(rxGs,txMEOFreq, ...
    Size = 1000000);          % In m
```

**Radiation Pattern of MEO Satellite Antenna**
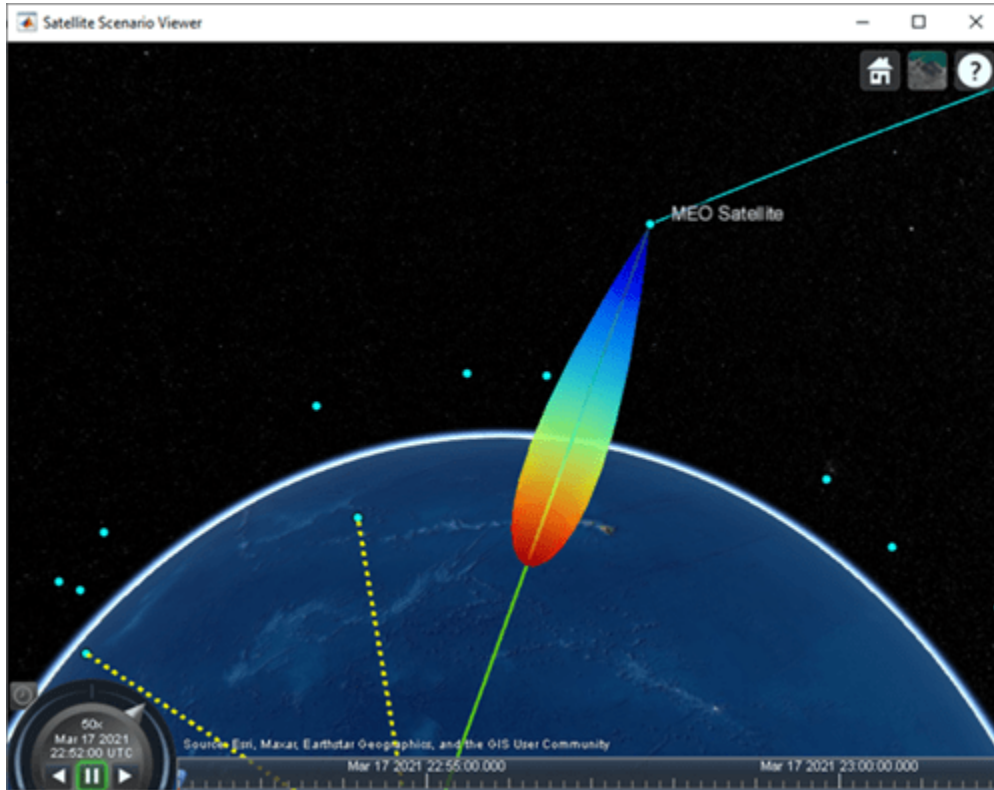


**Set Camera to View Ground Station Antenna Radiation Pattern**

```
% Set camera position and orientation to view the ground station antenna
% radiation pattern.
campos(v,-8,172,2500000);
camheading(v,40);
campitch(v,-60);
```
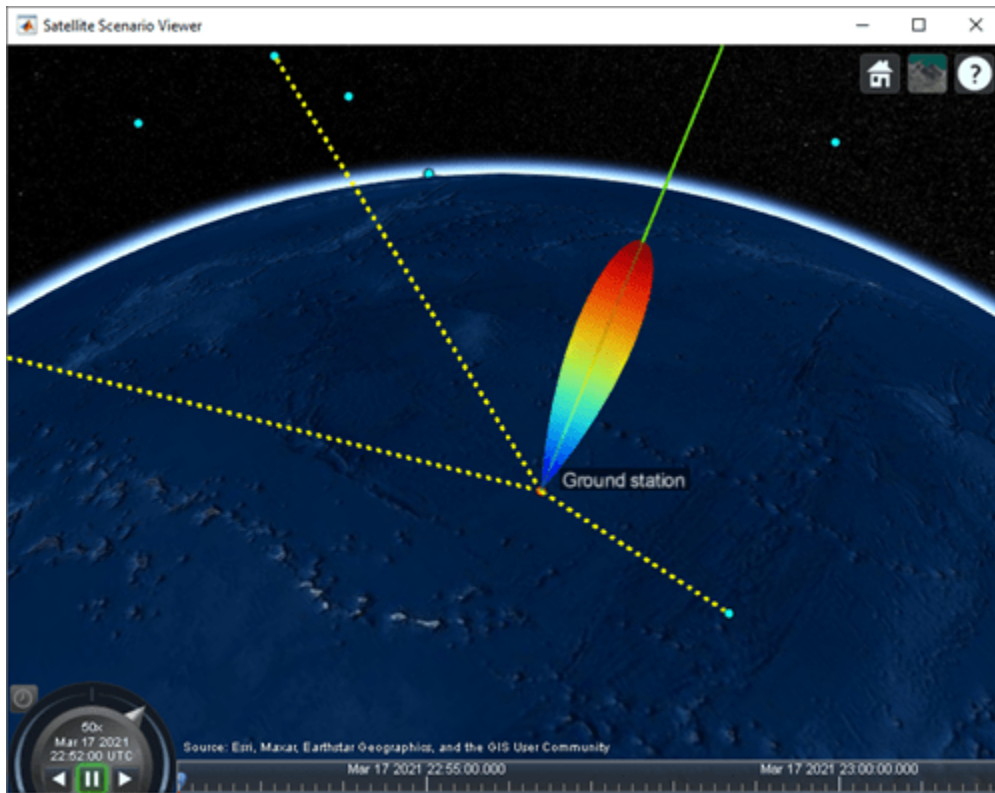
**Gaussian Antenna**
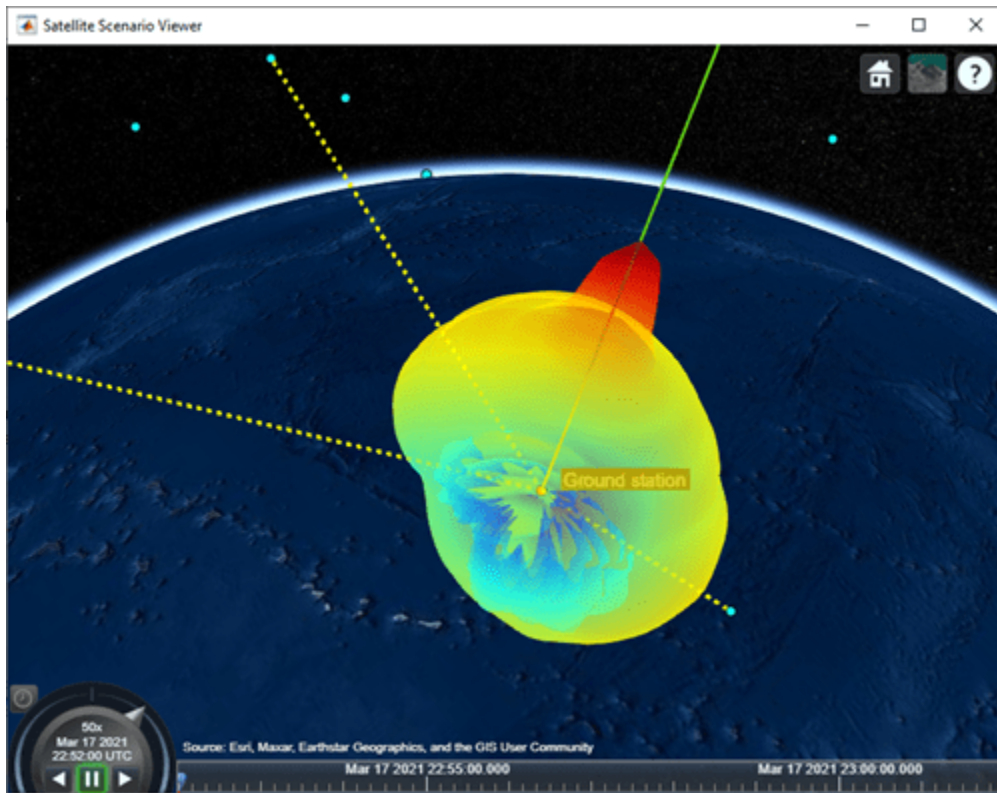
**Parabolic Reflector**
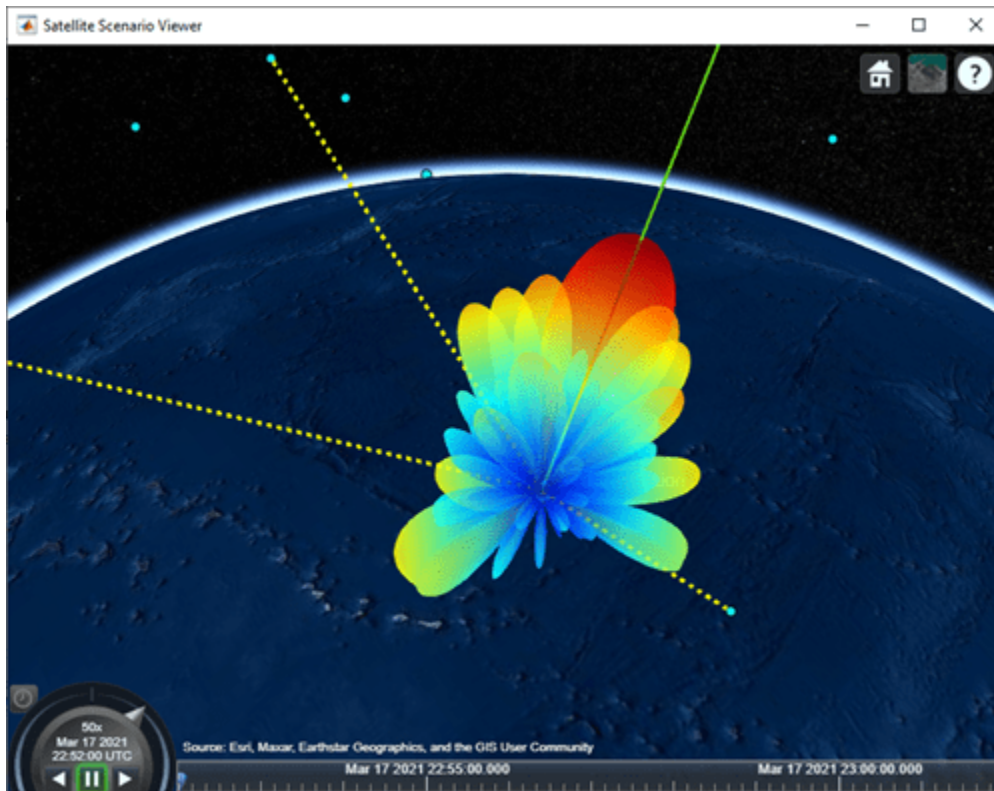
**Uniform Rectangular Array**

**Simulate Scenario and Visualize**

**With Gaussian Antenna or Parabolic Reflector**

If you selected Gaussian Antenna or Parabolic Reflector (requires Antenna Toolbox), use `play` to visualize the scenario from `StartTime` to `StopTime`. This will automatically simulate the scenario before playing back the visualization. Note how the antenna pointing changes as the gimbal tracks the MEO satellite.

```
if groundStationAntennaType == "Gaussian Antenna" || groundStationAntennaType == "Parabolic Refle
    play(sc);
    campos(v,-8,172,2500000);
    camheading(v,40);
    campitch(v,-60);
end
```
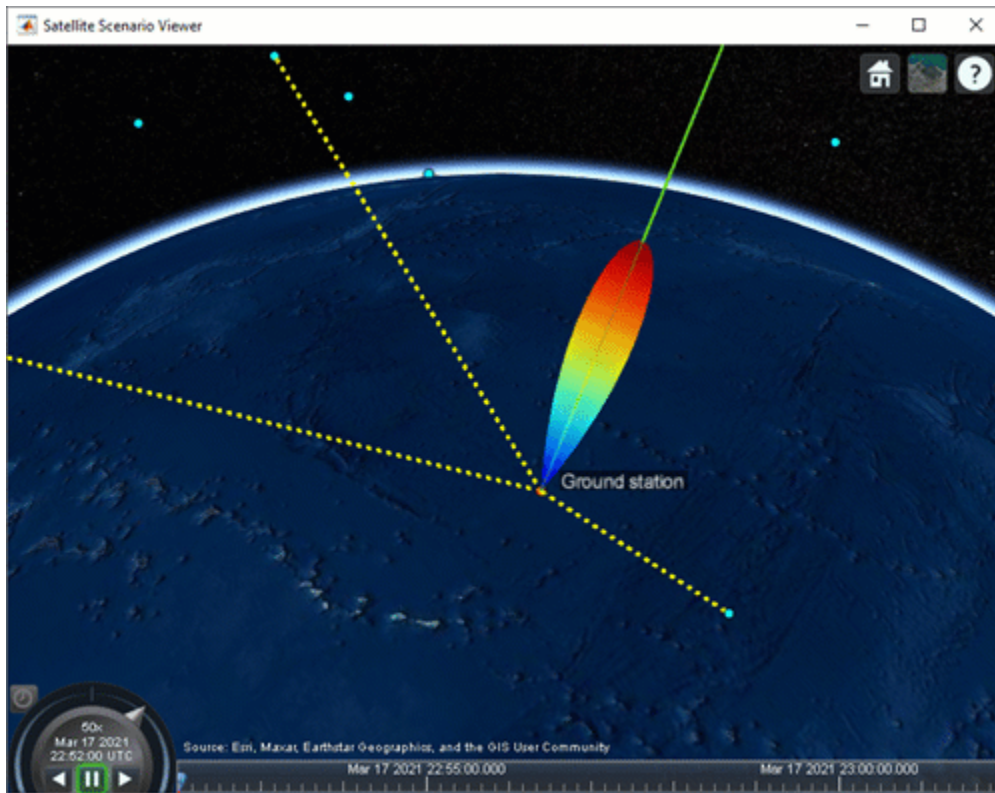
**With Gaussian Antenna**
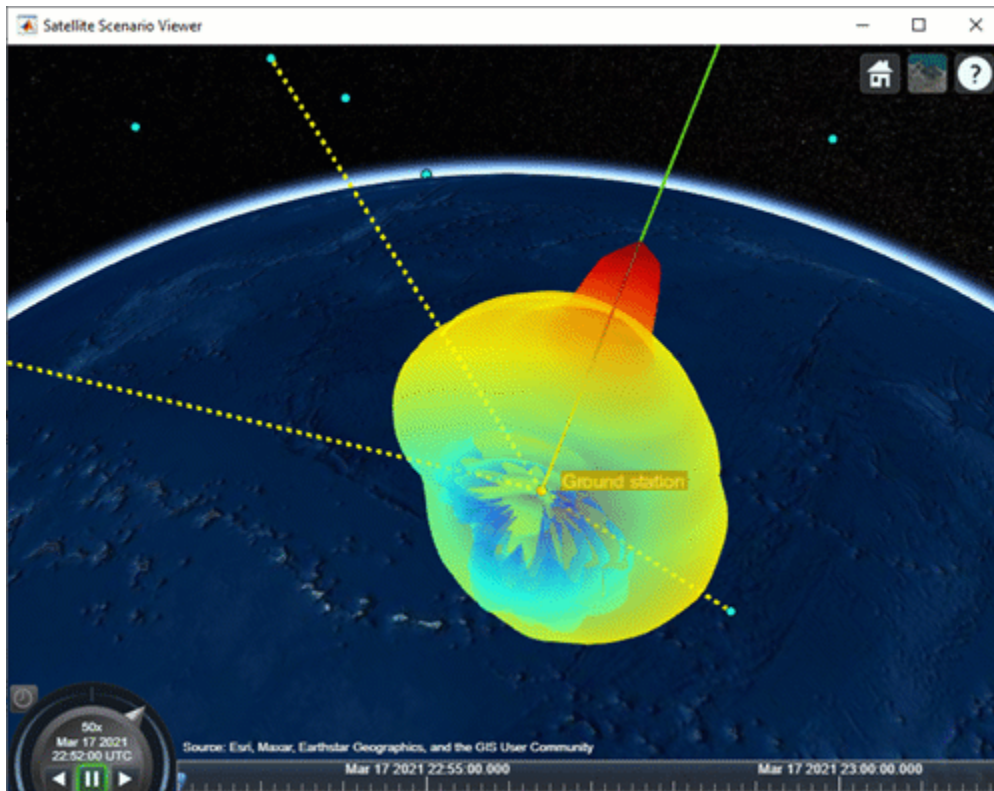
**With Parabolic Reflector**

**With Uniform Rectangular Array**

If you selected Uniform Rectangular Array (requires Phased Array System Toolbox), you must manually step through the simulation so that you can recompute the weights at each time step based on the new position of the MEO satellite and the interfering LEO satellites. To manually step through the simulation, first set `AutoSimulate` to false. Following this, you can call `advance` to move the simulation by one time step. The first call to `advance` will compute the simulation states at `StartTime`. Subsequent calls will advance the time step by one `SampleTime` and compute the states accordingly.

```
if groundStationAntennaType == "Uniform Rectangular Array"
    % Set AutoSimulate to false.
    sc.AutoSimulate = false;

    % Manually step through the simulation.
    while advance(sc)
        % Determine the access status history for each LEO satellite
        % corresponding to the current SimulationTime.
        acStatusHistory = accessStatus(ac);
        acStatus = acStatusHistory(:,end);

        % Determine the LEO satellites that are visible to the ground
        % station. These are the satellites that will potentially
        % interfere with the ground station at the current simulation
        % time.
        currentInterferingSat = interferingSat(acStatus == true);
```

```matlab
        % Determine the direction of the MEO satellite in the body frame of
        % the Uniform Rectangular Array. This is the lookout direction of
        % the array.
        [azdHistory,eldHistory] = aer(rxGs,meoSat,CoordinateFrame='body');
        azd = azdHistory(:,end);
        eld = eldHistory(:,end);

        % Determine the direction of these interfering satellites in
        % the body frame of the Uniform Rectangular Array. These are
        % the directions in which the array must point a null.
        [aznHistory,elnHistory] = aer(rxGs,currentInterferingSat,CoordinateFrame='body');
        azn = aznHistory(:,end);
        eln = elnHistory(:,end);

        % Calculate the steering vectors for lookout direction.
        % Requires Phased Array System Toolbox.
        wd = steervec(getElementPosition(ant)/lambda,[wrapTo180(azd);-eld]);

        % Calculate the steering vector for null directions.
        % Requires Phased Array System Toolbox.
        wn = steervec(getElementPosition(ant)/lambda,[wrapTo180(azn)';-eln']);

        % Compute the response of desired steering at null direction.
        rn = (wn'*wn)\(wn'*wd);

        % Sidelobe canceler - remove the response at null direction.
        w = wd-wn*rn;

        % Assign the weights to the phased array.
        pointAt(rxGs,Weights=w);
    end
end
```
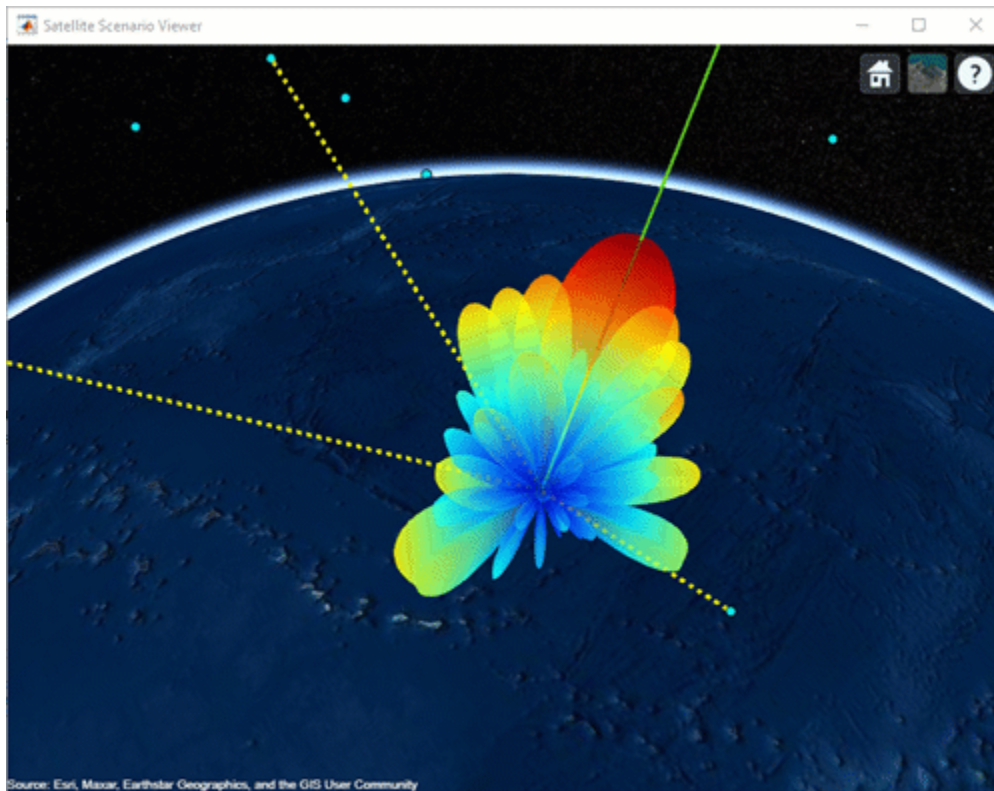
**Plot Downlink Closure Status Neglecting Interference**

Determine the closure status of the desired downlink from the MEO satellite. The `linkStatus` function neglects interference from other transmitters. Any time the downlink is closed, the status is true. Otherwise, the status is false. The status is indicated by 1 and 0, respectively in the plot.

```
[downlinkStatus,t] = linkStatus(downlink);
plot(t,downlinkStatus,"-g",LineWidth=2);
xlabel("Time");
ylabel("Downlink Closure Status");
title("Link Status as a Function of Time");
grid on;
```

**Link Status as a Function of Time**
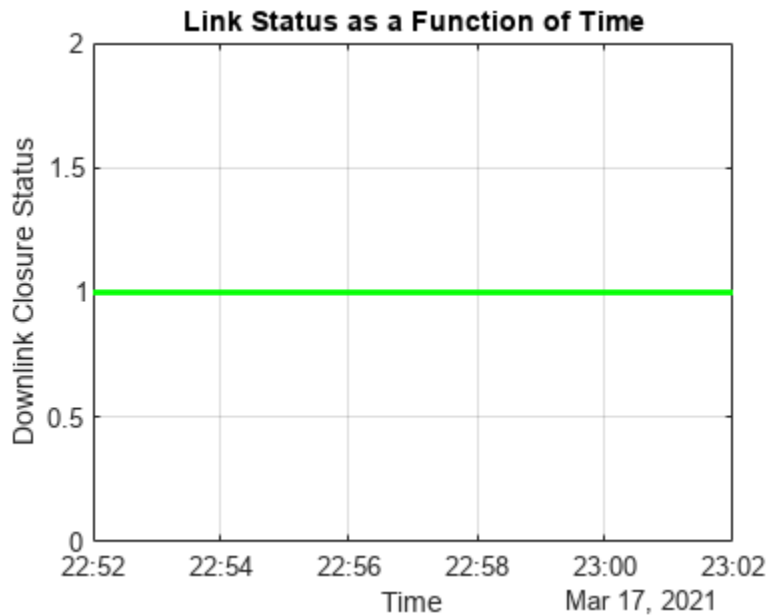
**Calculate Downlink Closure Status with Interference**

Calculate the downlink closure status with interference by first calculating the MEO downlink and interference signal power levels at the ground station receiver input using `sigstrength`. The locations of received power measurements and losses are illustrated in the ground station receiver diagram below.
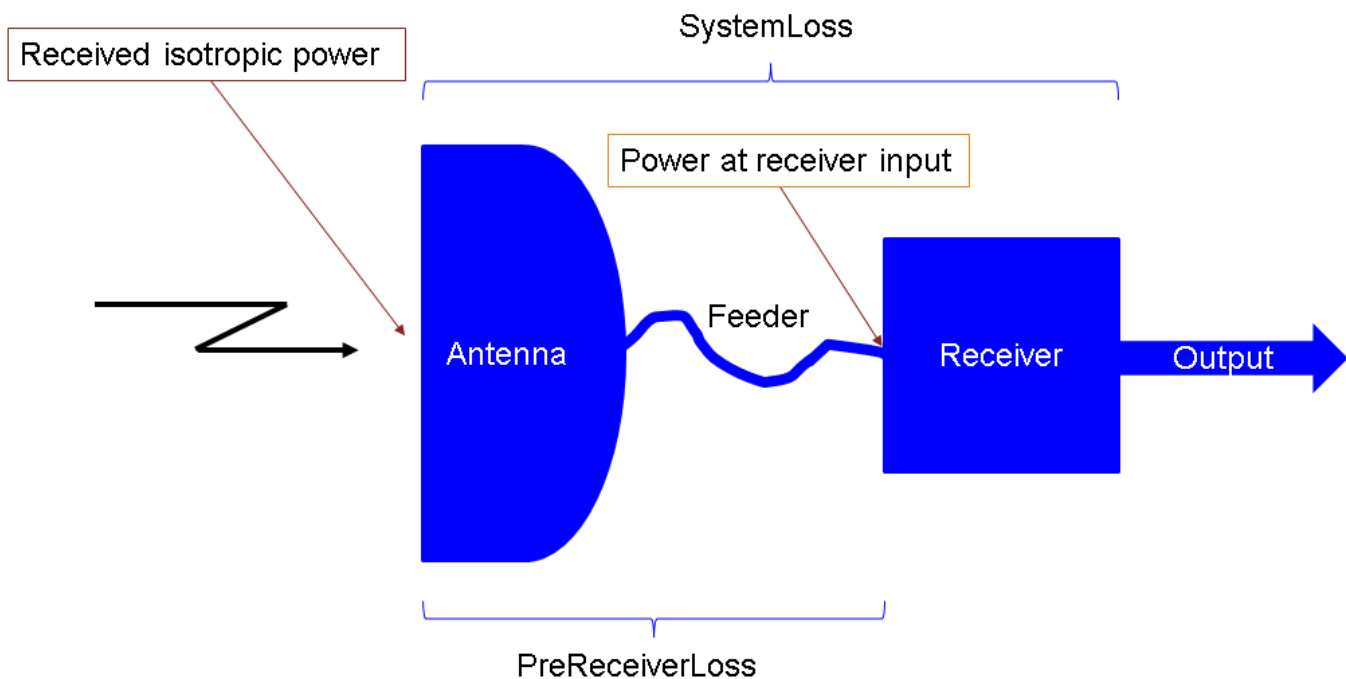


```
% Calculate the power at receiver input corresponding to the downlink from
% the MEO satellite.
```

```
[~,downlinkPowerRxInput] = sigstrength(downlink); % In dBW

% Calculate the interference power at receiver input corresponding to each
% LEO satellite.
[~,interferencePowerRxInput] = sigstrength(lnkInterference); % In dBW
```

Calculate total interfering signal power at the receiver input. Get this quantity by summing the individual power levels from the interfering LEO satellites in Watts.

```
interferencePowerRxInputW = 10.^(interferencePowerRxInput/10); % W
interferencePowerRxInputSumW = sum(interferencePowerRxInputW); % W
```

Calculate the amount of total interfering signal power that contributes to interference in the signal bandwidth by following these steps.

1) Calculate the overlapping portion of the signal bandwidth with the bandwidth of the interferers. This example considers the transmission power of interfering satellites and the MEO satellite as constant across the whole bandwidth of respective MEO satellite and interfering satellites.

2) Calculate the amount of interference power that acts as interference to signal bandwidth.

This diagram shows the power spectral density (PSD) plot, which shows the actual interference power and modeled interference power when the transmission bandwidth and interfering bandwidth overlap. The actual interference power is the area occupied by the interference power density in the overlapped bandwidth region. This actual interference power is then spread across the entire transmission bandwidth and assumed to be noise-like.
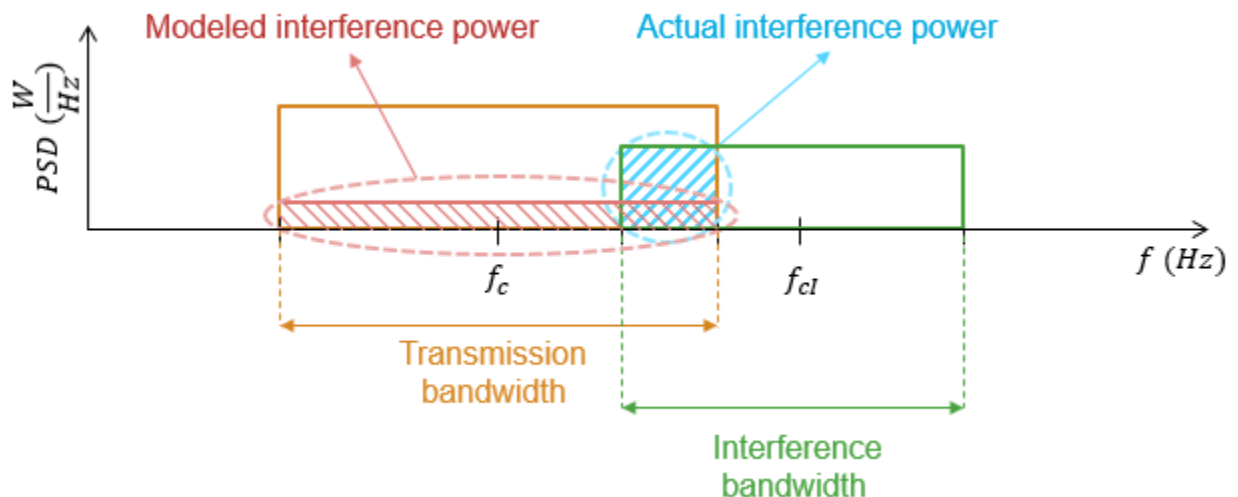


This example assumes that the transmission (or signal) bandwidth of the MEO satellite is 30 MHz and that the bandwidth of the interfering signal is 20 MHz.

```
txBandwidth = 30e6;                                             % In Hz
interferenceBandWidth = 20e6;                                   % In Hz

% Get the overlap portion of the interfering bandwidth and the bandwidth of
% interest. The assumption is to the have same interference power across
% the whole bandwidth.
overlapFactor = getOverlapFactor(txMEOFreq,txBandwidth, ...
```

```
        interferenceFreq,interferenceBandWidth);

% Get the interference power that contributes as interference to the signal
% of interest from the total interference power
interferencePowerRxInputActual = interferencePowerRxInputSumW*overlapFactor; % In W
```

Interference is modeled by treating the contribution of interfering signal power in the overlapped bandwidth as noise. Accordingly, add this quantity to the thermal noise at the ground station receiver input. Note that the interference and noise power levels must be added in Watts.

```
% Calculate the thermal noise at the ground station receiver input.
T = HelperGetNoiseTemperature(txMEOFreq,rxGs); % In K
kb = physconst("Boltzmann");
thermalNoise = kb*T*txBandwidth;                % In W

% Calculate the noise plus interference power at the receiver input.
noisePlusInterferencePowerW = thermalNoise + interferencePowerRxInputActual; % In W
noisePlusInterferencePower = 10*log10(noisePlusInterferencePowerW);          % In dBW
```

Calculate the carrier to noise plus interference power spectral density ratio at the demodulator input as follows:

$$C/(N_0 + I_0) = P_{RxInput} - (N + I) + 10\log_{10} TxBandwidth - LOSS_{Rx},$$

where:

- $C/(N_0 + I_0)$ is the carrier to noise plus interference power density ratio at the demodulator input (in dB).
- $P_{RxInput}$ is the received downlink power from the MEO satellite measured at the ground station receiver input (in dBW).
- $(N + I)$ is the sum of receiver system thermal noise and the contribution of interfering signal power in the overlapped bandwidth measured at the receiver input (in dBW).
- $TxBandwidth$ is the downlink transmission bandwidth from MEO satellite (in Hz).
- $LOSS_{Rx}$ is the loss that occurs between the receiver input and the demodulator input (in dB).

```
% Calculate loss that occurs between the receiver input and the demodulator
% input.
rxGsLoss = rxGs.SystemLoss - rxGs.PreReceiverLoss;

% Calculate C/(N0+I0) at the demodulator input.
CNoPlusInterference = downlinkPowerRxInput - ...
    noisePlusInterferencePower + 10*log10(txBandwidth) - rxGsLoss;
```

Calculate the energy per bit to noise plus interference power spectral density ratio at the demodulator input as follows:

$$E_b/(N_0 + I_0) = C/(N_0 + I_0) - 10\log_{10} BITRATE - 60,$$

where:

- $E_b/(N_0 + I_0)$ is the energy per bit to noise plus interference power spectral density ratio at the demodulator input (in dB).
- $BITRATE$ is the bit rate of the downlink from the MEO satellite (in Mbps).

```
bitRate = txMEOSat.BitRate;
ebNoPlusInterference = CNoPlusInterference - 10*log10(bitRate) - 60;
```

Calculate the link margin as follows:

$$MARGIN = E_b/(N_0 + I_0) - (E_b/N_0)_{Required}$$

where:

- *MARGIN* is the link margin (in dB).
- $(E_b/N_0)_{Required}$ is the minimum received energy per bit to noise power spectral density ratio at the demodulator input that is required to close the link (in dB).

```
marginWithInterference = ebNoPlusInterference - rxGs.RequiredEbNo;
```

Calculate the downlink closure status with interference. The status is true whenever the link margin is greater than or equal to 0 dB.

```
downlinkStatusWithInterference = marginWithInterference >= 0;
```

**Calculate Energy per Bit to Noise Power Spectral Density Ratio**

Calculate the energy per bit to noise power spectral density ratio (Eb/N0) of the downlink and interfering links at the demodulator input for analysis later.

```
ebnoDownlink = ebno(downlink);          % In dB
ebnoInterference = ebno(lnkInterference); % In dB
```

**Plot Downlink Closure Status with Interference**

Plot the new downlink closure status that accounts for interference. Compare the new link status with the previous case when interference was neglected.

```
plot(t,downlinkStatusWithInterference,"-r",t,downlinkStatus,"--g",LineWidth=2);
legend("Interference accounted","Interference neglected");
xlabel("Time");
ylabel("Downlink Closure Status");
title("Link Status as a Function of Time");
ylim([0 1.2]);
grid on
```
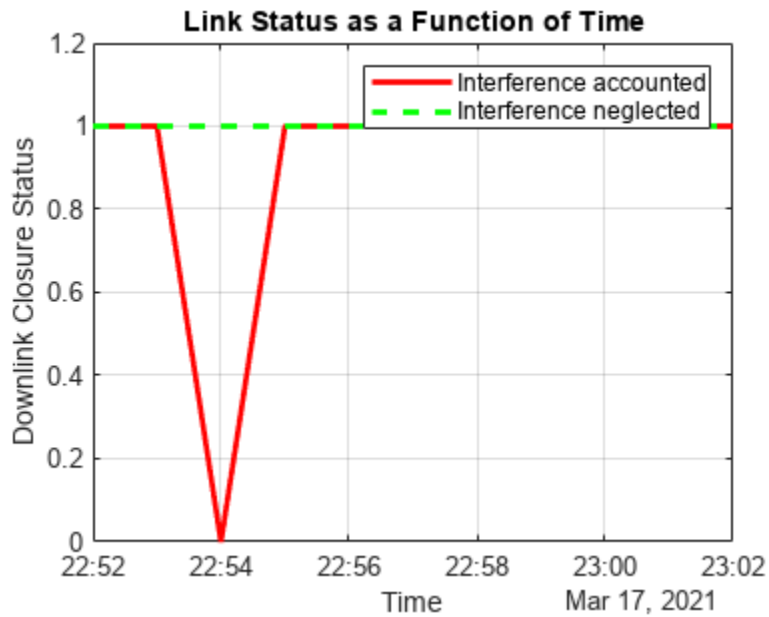
**When Gaussian Antenna or Parabolic Reflector are Chosen**
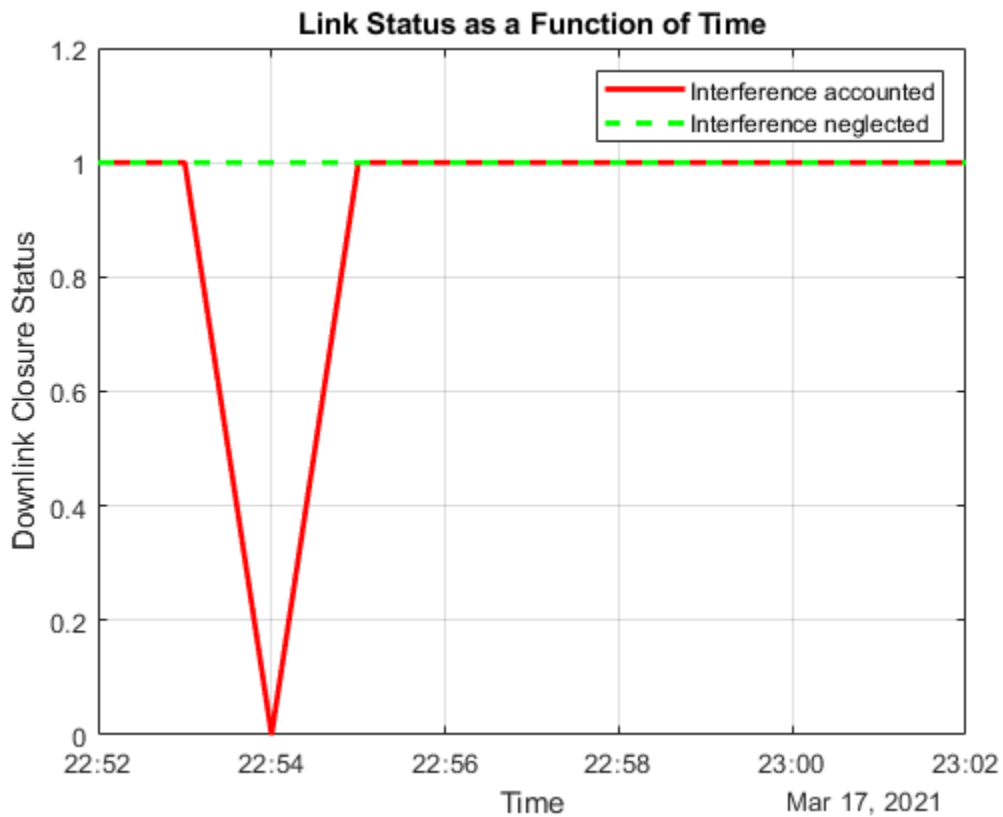


The plot shows that at 10:54 PM, the downlink cannot be closed because of excessive interference. This occurs because `Satellite 10` of the LEO constellation flies overhead, and its transmission is

picked up by its main lobe. This can also be visually confirmed by setting the current time of the viewer to 10:54 PM and clicking on the satellite near the main lobe of the antenna. Note that you require Antenna Toolbox to select Parabolic Reflector.

```
if groundStationAntennaType == "Gaussian Antenna" || groundStationAntennaType == "Parabolic Refle
    v.CurrentTime = datetime(2021,3,17,22,54,0);
end
```
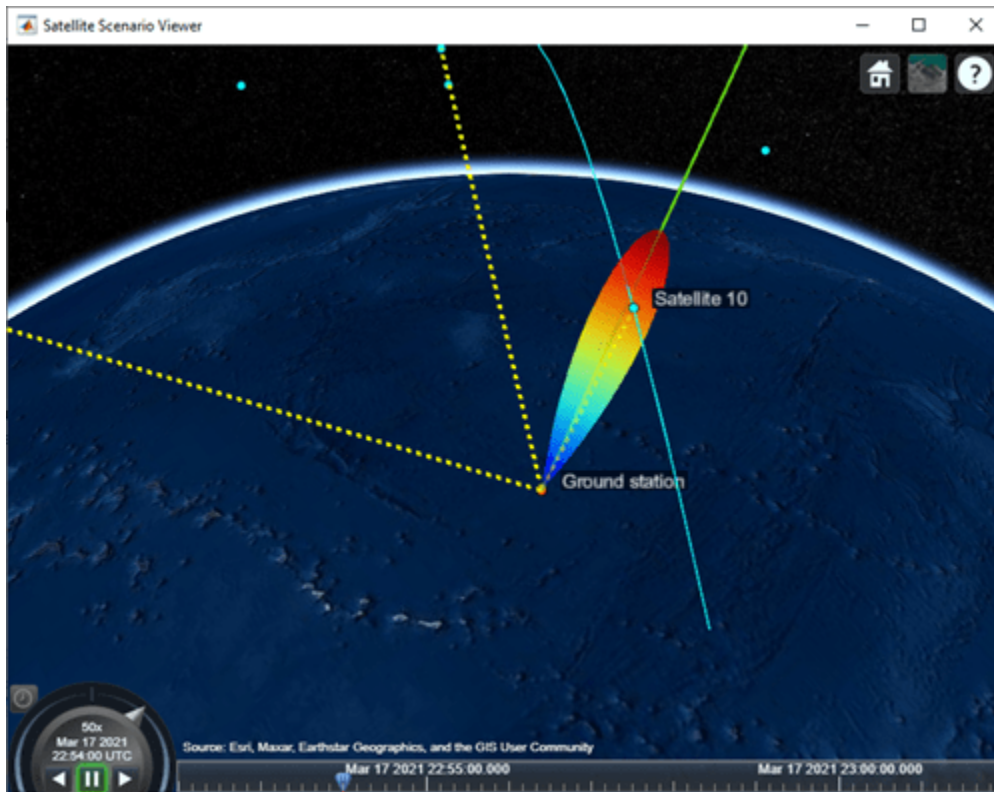


**Gaussian Antenna**
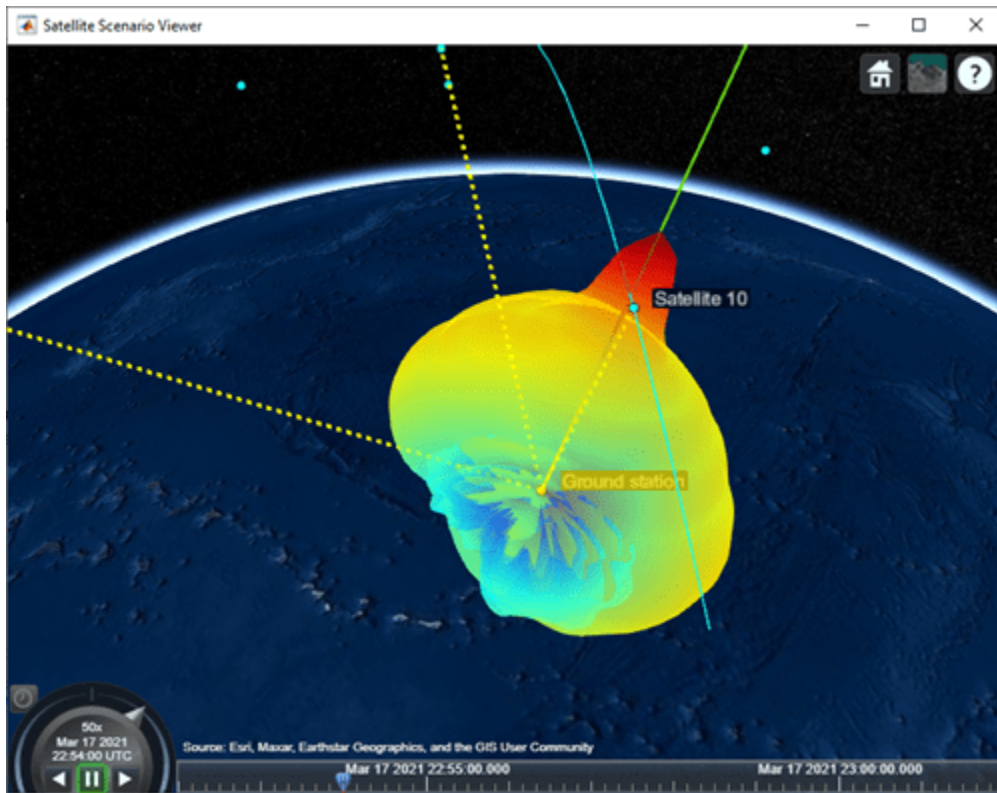
**Parabolic Reflector**

**When Uniform Rectangular Array with Interference Cancellation Is Chosen**
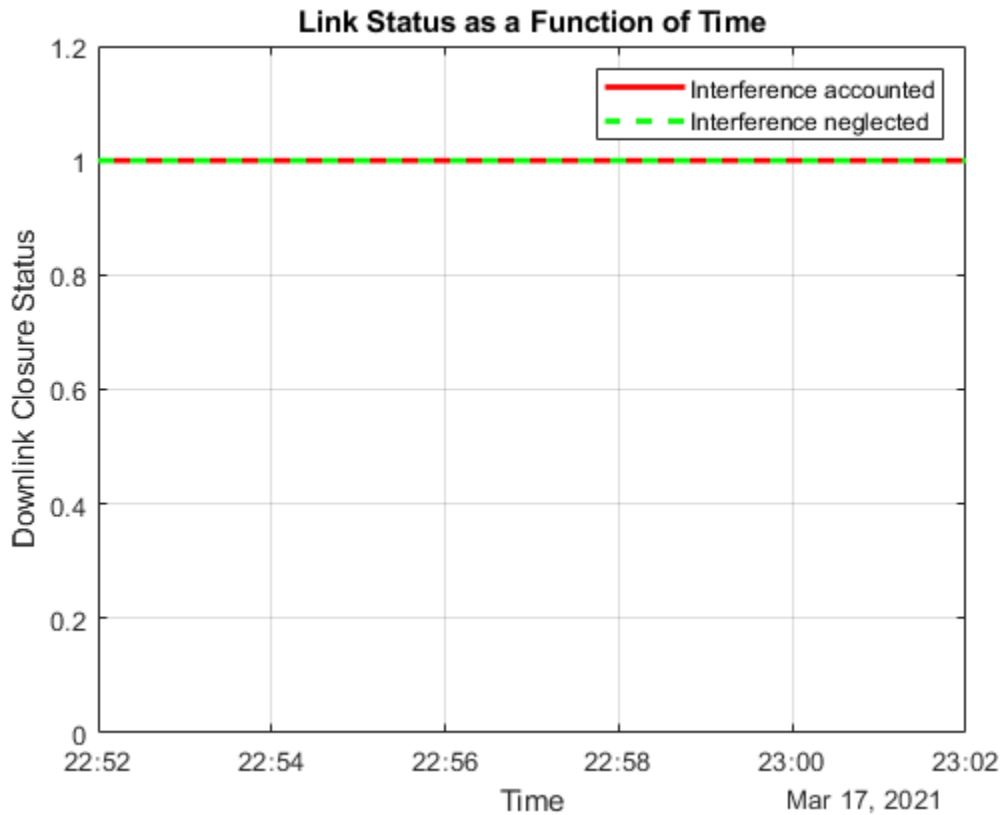


If you selected Uniform Rectangular Array (requires Phased Array System Toolbox), the plot shows that the downlink can be closed for the duration of the scenario because the array is pointing nulls towards the direction of the interfering LEO satellites. This can also be visually confirmed by setting the current time of the viewer to 10:54 PM and 10:55 PM. To be able to manually set the viewer `CurrentTime`, you must change `AutoSimulate` to true. Note that this will clear the simulation data. Also, you will be required to re-compute the weights for these times and assign them to the array using `pointAt`. The satellite that is overflying the ground station is `Satellite 10`. Click on it to see its name and orbit. Drag the mouse while holding down on the left mouse button or scroll button to bring the camera to the desired position and orientation. Rotate the scroll wheel to control camera zoom. Additionally, make the radiation pattern opaque to clearly visualize the position of `Satellite 10` with respect to the lobes. You can see that at both times, `Satellite 10` is in between lobes. This is because the array is pointing a null towards the satellite, thereby cancelling interference from it.

```
if groundStationAntennaType == "Uniform Rectangular Array"
    % Set AutoSimulate to true.
    sc.AutoSimulate = true;

    % Set viewer CurrentTime to 10:54 PM.
    time = datetime(2021,3,17,22,54,0);
    v.CurrentTime = time;

    % Calculate the weights and assign them to the array.
    currentInterferingSat = interferingSat(accessStatus(ac,time) == true);
    [azd,eld] = aer(rxGs,meoSat,time,CoordinateFrame='body');
```

```
[azn,eln] = aer(rxGs,currentInterferingSat,time,CoordinateFrame='body');

% Requires Phased Array System Toolbox.
wd = steervec(getElementPosition(ant)/lambda,[wrapTo180(azd);-eld]);
wn = steervec(getElementPosition(ant)/lambda,[wrapTo180(azn)';-eln']);

rn = (wn'*wn)\(wn'*wd);
w = wd-wn*rn;
pointAt(rxGs,Weights=w);

% Make the radiation pattern opaque.
pattern(rxGs,txMEOFreq, ...
    Size = 1000000, ...
    Transparency = 1);
end
```
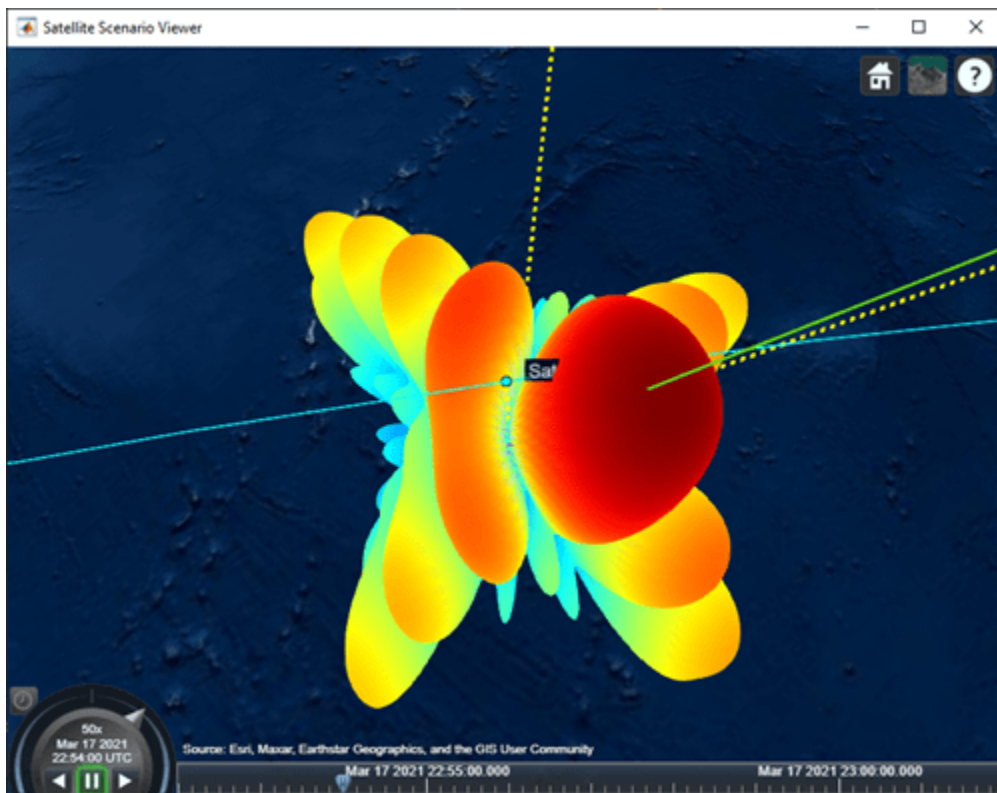


**At 10:54 PM UTC**

You can run the above code with time set to 10:55 PM and observe the nulls pointing towards the new positions of the interfering satellites.
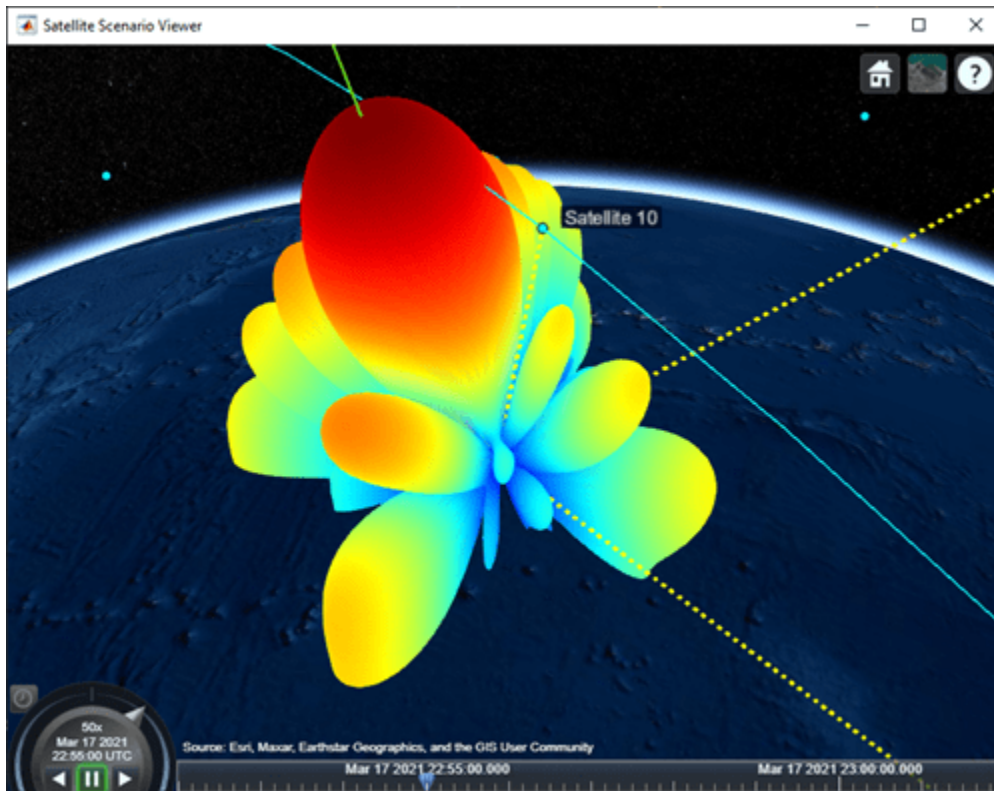
**At 10:55 PM UTC**

**Calculate and Plot Carrier to Noise Ratio and Carrier to Noise Plus Interference Ratio**

Calculate the carrier to noise ratio (CNR) at the demodulator input as follows:

$C/N_0 = E_b/N_0 + 10\log_{10}BITRATE + 60,$

$C/N = C/N_0 - 10\log_{10}TxBandwidth,$

where:

- $C/N$ is the carrier to noise ratio at the demodulator input (in dB).
- $E_b/N_0$ is the carrier to noise power spectral density ratio at the demodulator input (in dB).

```
% Calculate the carrier to noise power spectral density ratio.
CNoDownlink = ebnoDownlink + 10*log10(bitRate) + 60;
```

```
% Calculate the carrier to noise ratio.
cByN = CNoDownlink - 10*log10(txBandwidth);
```

Calculate the carrier to noise plus interference ratio (CNIR) at the demodulator input as follows:

$$C/(N + I) = C/(N_0 + I_0) - 10\log_{10}TxBandwidth$$

```
cByNPlusI = CNoPlusInterference - 10*log10(txBandwidth);
```

Plot CNR and CNIR

```
plot(t,cByNPlusI,"-r",t,cByN,"--g",LineWidth=2);
legend("CNIR", "CNR",Location="south");
xlabel("Time");
ylabel("CNR or CNIR (dB)");
title("CNR and CNIR vs. Time for " + groundStationAntennaType);
grid on
```

CNR and CNIR vs. Time for Gaussian Antenna

CNR and CNIR vs. Time for Parabolic Reflector

**CNR and CNIR vs. Time for Uniform Rectangular Array**
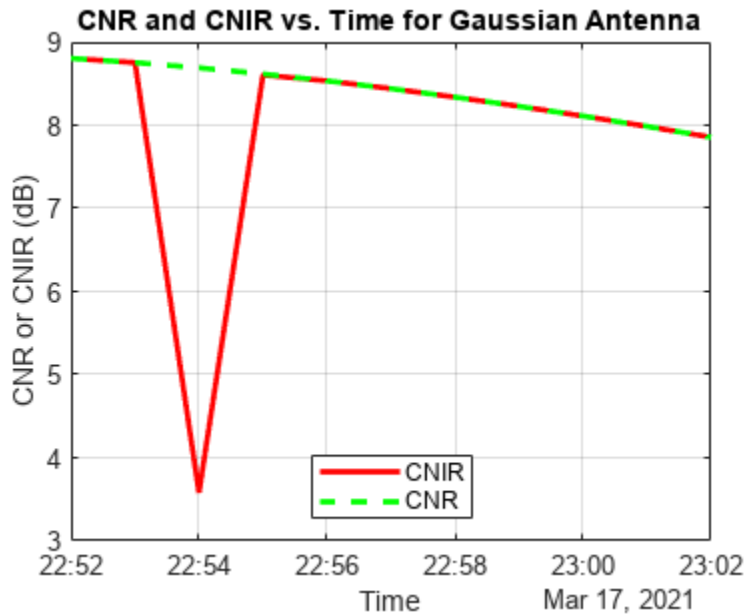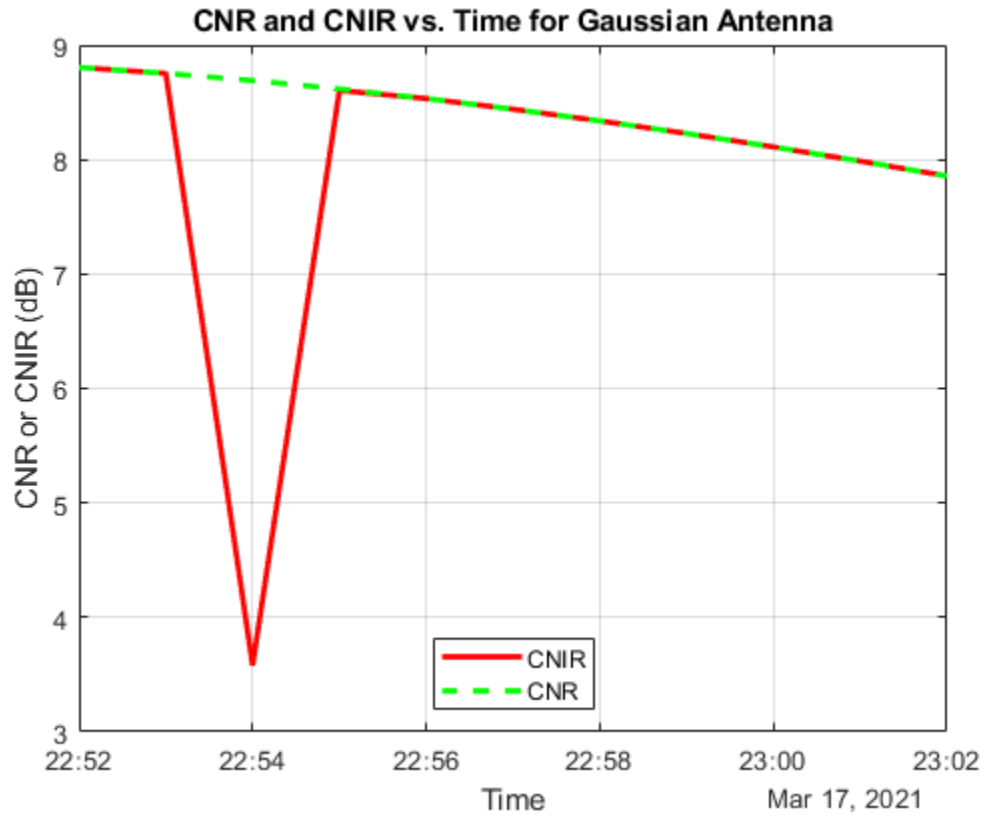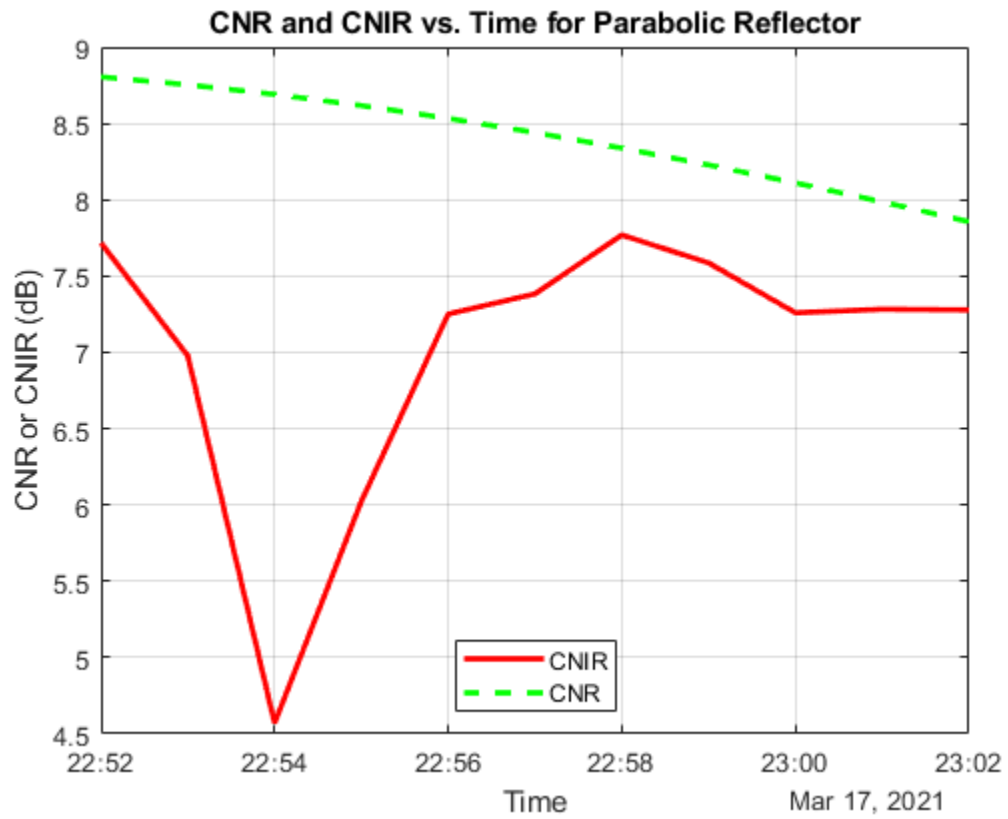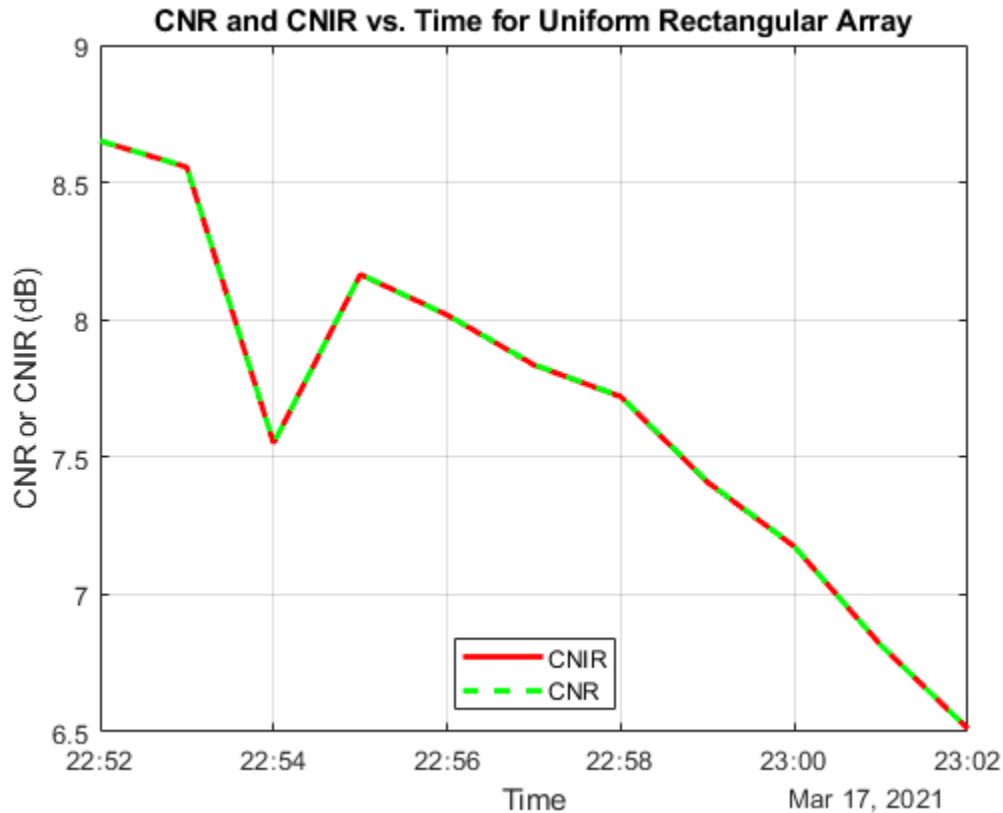


When Uniform Rectangular Array (requires Phased Array System Toolbox) with MEO satellite tracking and interference cancellation is chosen, both CNR and CNIR overlap because there is no interference from the LEO satellites. This is because the array is pointing nulls towards the LEO satellites. This can be confirmed by noting that the maximum power at ground station receiver input from the interfering satellites is about -167.3 dBW, which is very low. For all other antennas used in this example, the maximum power at ground station receiver input is much higher (-125.9 dBW for Gaussian Antenna, and -127.3 dBW for Parabolic Reflector).

```
maxInterferencePowerRxInput = max(interferencePowerRxInput,[],'all');
disp("The maximum power at ground station receiver input from the interfering LEO satellites ove
```

The maximum power at ground station receiver input from the interfering LEO satellites over the e

**Compare Link Margins with and without Interference**

Calculate the link margin without interference as follows:

$$MARGIN = E_b/N_0 - (E_b/N_0)_{Required}$$

```
marginWithoutInterference = ebnoDownlink - rxGs.RequiredEbNo;
```

Plot the link margins with and without interference.

```
plot(t,marginWithInterference,"-r",t,marginWithoutInterference,"--g",LineWidth=2);
legend("With interference","Without interference",Location="south");
xlabel("Time");
ylabel("Margin (dB)");
```

```
title("Link Margin vs. Time for " + groundStationAntennaType);
grid on
```

Link Margin vs. Time for Parabolic Reflector
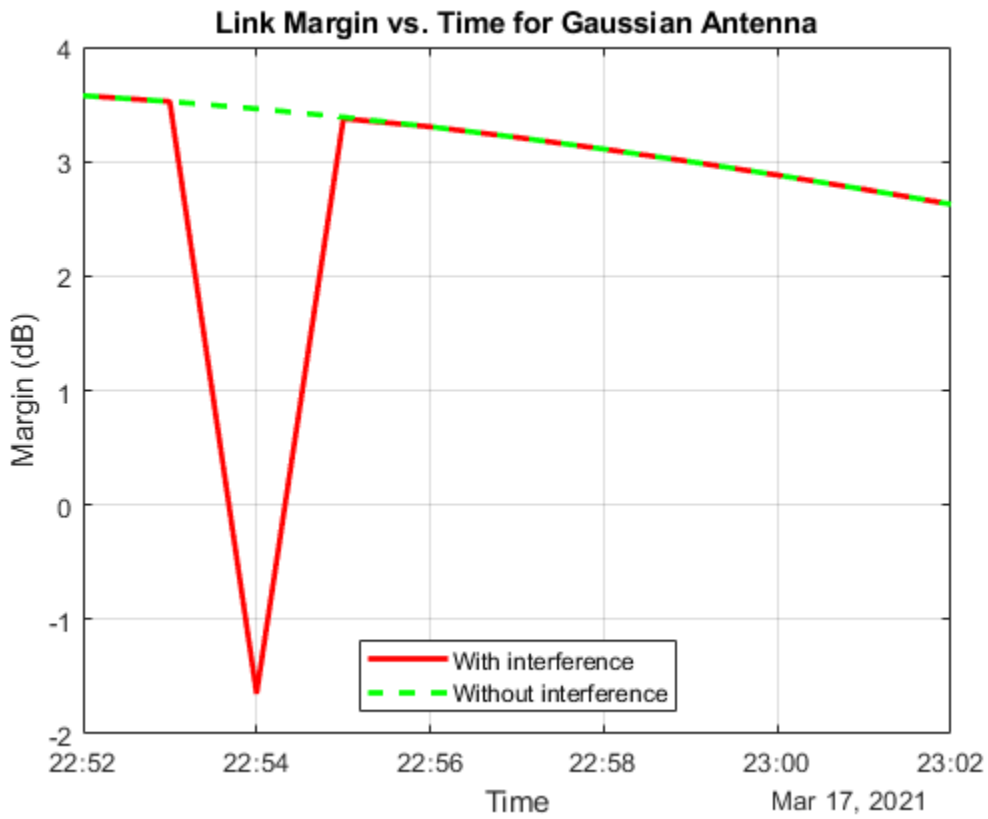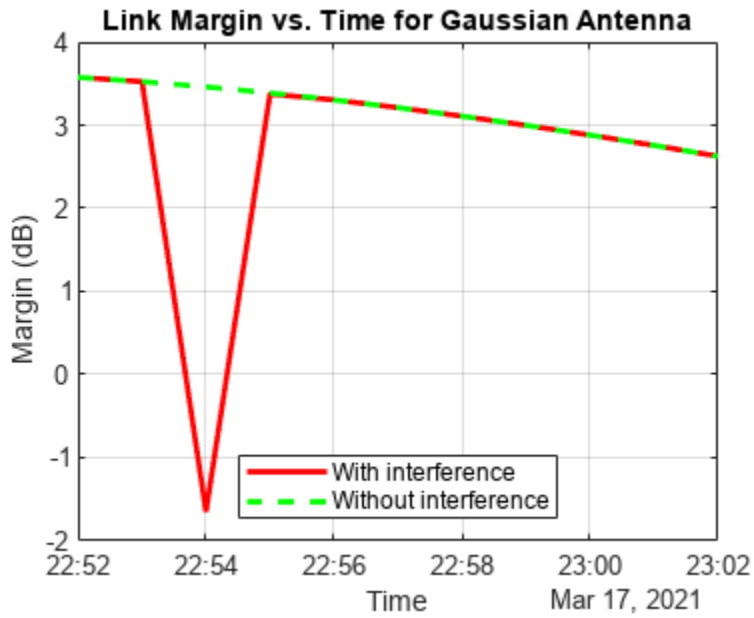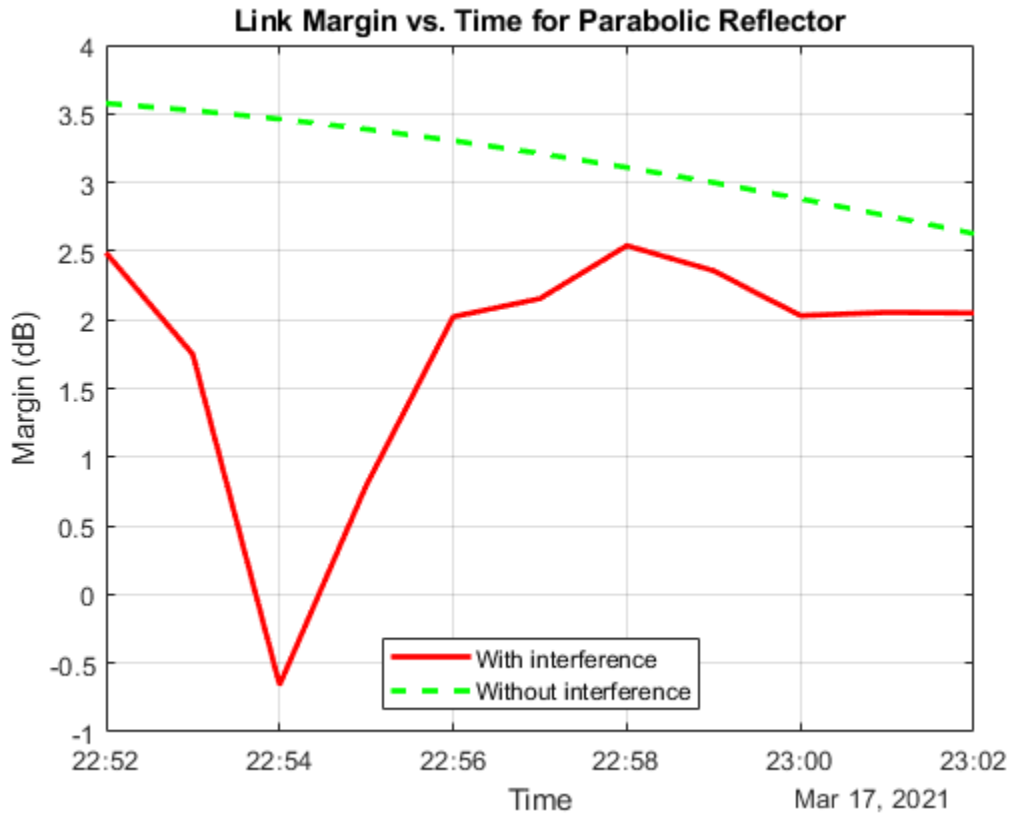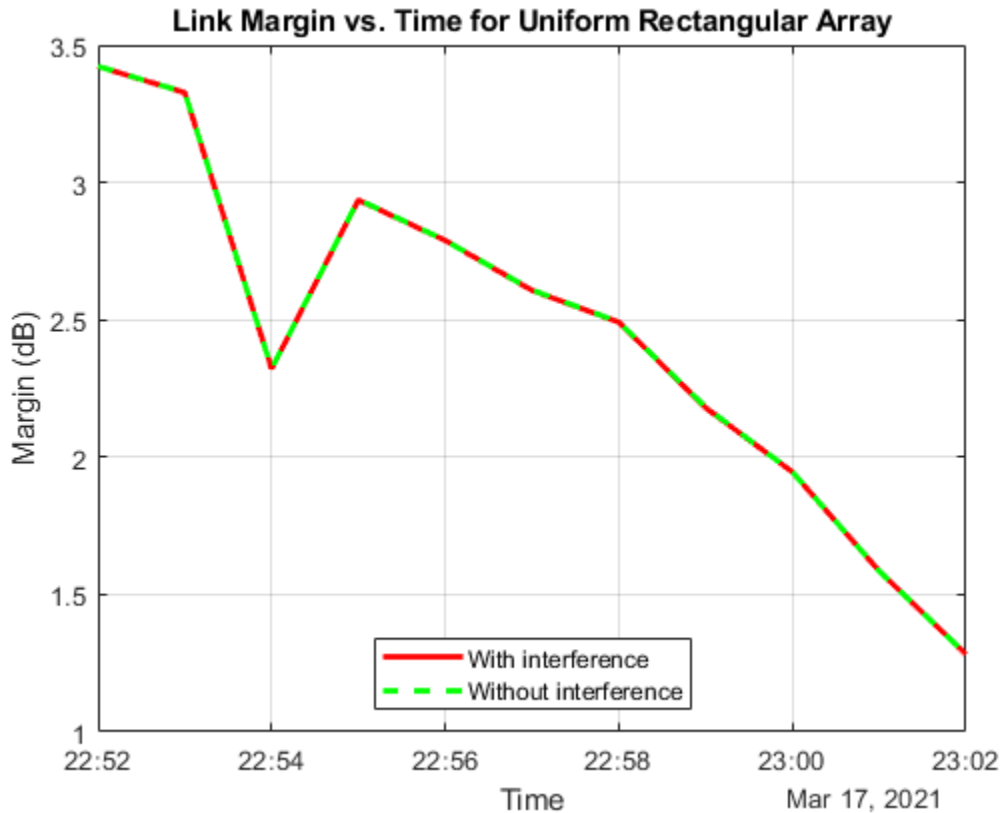
Any time the link margin is greater than or equal to 0 dB, the downlink is closed. With Gaussian Antenna, Parabolic Reflector (requires Antenna Toolbox), and Uniform Rectangular Array (requires Phased Array System Toolbox) without interference cancellation, there exist times when the link margin dips below 0 dB because of interference. At these times, the downlink is broken.

**Further Exploration**

This example demonstrates how to analyze interference on a satellite communication link. The link closure times are a function of these parameters:

- The orbit of the satellites
- The position of the ground station
- The specifications of the transmitters and the receiver
- The specifications of the transmitter and receiver antennas
- Weights if using a Uniform Rectangular Array
- The signal and interference bandwidth

Modify these parameters to observe their influence on the level of interference on the link. You can also choose the different antennas from Antenna toolbox and Phased Array System Toolbox for transmitters and receivers and observe the link performance. When using phased arrays and if you are only interested in making the main lobe track a single target and not deal with pointing nulls, you can use `pointAt` to automatically track other satellites, ground stations, and geographic locations without having to manually simulate by setting `AutoSimulate` to false. The limitation of calling `play`

when using dynamically steered phased arrays is that you cannot visualize the variation of their radiation pattern over the course of the simulation.

**Helper Functions**

The example uses the helper function HelperGetNoiseTemperature to obtain the noise temperature of the receiver antenna.

The example also uses this local function to compute the amount of overlap between the transmission bandwidth and the interfering bandwidth.

```matlab
function overlapFactor = getOverlapFactor(txFreq,txBW,interferenceFreq,interferenceBW)
% getOverlapFactor provides the amount of interference bandwidth overlapped
% with transmission bandwidth

    txFreq_Limits = [txFreq-(txBW/2) txFreq+(txBW/2)];
    interferenceFreq_Limits = [interferenceFreq-(interferenceBW/2) ...
        interferenceFreq+(interferenceBW/2)];
    if (interferenceFreq_Limits(2) < txFreq_Limits(1)) || ...
            (interferenceFreq_Limits(1) > txFreq_Limits(2))
        % If no overlap exists between transmission bandwidth and
        % interfering bandwidth, then overlap factor is 0
        overlapFactor = 0;
    elseif (interferenceFreq_Limits(2) <= txFreq_Limits(2)) && ...
            (interferenceFreq_Limits(1) >= txFreq_Limits(1))
        % If interfering bandwidth lies completely within transmission
        % bandwidth, then overlap factor is 1
        overlapFactor = 1;
    elseif (interferenceFreq_Limits(2) > txFreq_Limits(2)) && ...
            (interferenceFreq_Limits(1) < txFreq_Limits(1))
        % If transmission bandwidth lies completely within interfering
        % bandwidth, then overlap factor is the ratio of transmission
        % bandwidth with that of interference bandwidth
        overlapFactor = txBW/interferenceBW;
    elseif (interferenceFreq_Limits(2) <= txFreq_Limits(2)) && ...
            (interferenceFreq_Limits(1) <= txFreq_Limits(1))
        % If the start edge of transmission bandwidth lies within
        % interfering bandwidth, then overlap factor is the ratio of
        % difference from last edge of interfering bandwidth and first edge
        % of signal bandwidth, with that of interference bandwidth
        overlapFactor = (interferenceFreq_Limits(2)-txFreq_Limits(1))/interferenceBW;
    else
        % If the last edge of transmission bandwidth lies within
        % interfering bandwidth, then overlap factor is the ratio of difference
        % from last edge of signal bandwidth and first edge of interfering
        % bandwidth, with that of interference bandwidth
        overlapFactor = (-interferenceFreq_Limits(1)+txFreq_Limits(2))/interferenceBW;
    end

end
```

# Array Pattern Synthesis Part III: Deep Learning

This example shows how to design and train a convolutional neural network (CNN) to compute the element weights that produce a desired pattern.

**Introduction**

Pattern synthesis is an important topic in array processing. Array weights help shape the beam pattern of a sensor array to match a desired pattern. Traditionally, pattern synthesis algorithms are often borrowed from filter design techniques due to the similarity between the spatial signal processing and frequency domain signal processing. Many such algorithms are covered in our Array Pattern Synthesis Part I example. Unfortunately, those algorithms are often not very flexible to accommodate different kinds of constraints. Therefore, as a more general solution, various optimization techniques are used to produce desired patterns. Some frequently used optimization solvers are introduced in the Array Pattern Synthesis Part II example. Although the optimization-based algorithms are very flexible, for large arrays, it takes a while to converge into the optimal solution. This pitfall makes it difficult to form a given beam pattern in real time.

Deep learning techniques have seen many successes in computer vision and natural language processing. Although a deep learning network needs to be trained off line; once trained, the resulting network can be used to achieve real time performance. Hence, a deep learning network may be able to provide a solution for real time pattern synthesis, as suggested in [1].

**Array Definition**

Consider a circular planar array with a radius of 3 meters. The elements are located on a rectangular grid with an element spacing of 0.5 meters.

```
r = 3;
delta = 0.5;
lambda = 1;
[pos,N] = getCircularPlanarArrayPositions(r,delta,lambda);
```

The array aperture looks like below. The array is in the y-z plane with broadside aligned with x axis.

```
ant = phased.ConformalArray('ElementPosition',pos);
viewArray(ant,'title','Circular Planar Array Aperture')
```

Circular Planar Array Aperture



Array Span:
X axis = 0.0 m
Y axis = 5.5 m
Z axis = 5.5 m
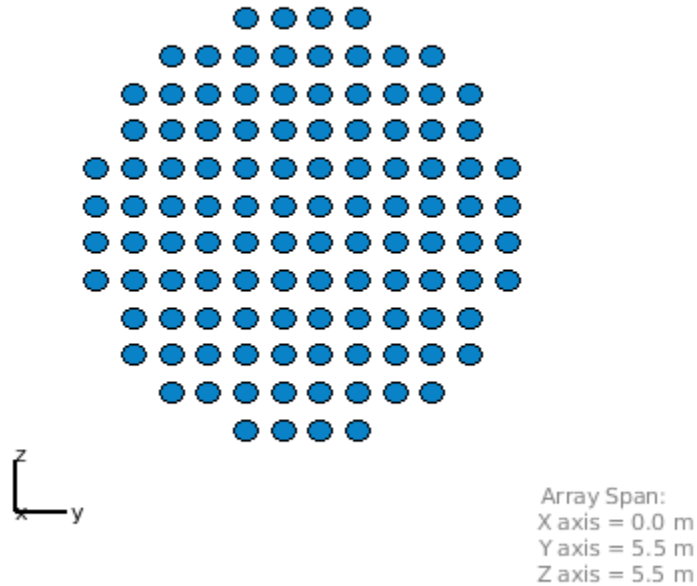
**Pattern Synthesis with Optimization**

As explained in the Array Pattern Synthesis Part II example, optimization techniques can be used to derive the pattern synthesis weights. Assume we would like to have a pattern whose main lobe is along azimuth and elevation 0 degrees. The pattern should also satisfy the following constraints:

- Maximize the directivity
- Suppress interferences 30 dB below mainlobe
- Keep sidelobe levels within -20 and 20 degrees azimuth or elevation be 17 dB below mainlobe

```
% define main lobe direction
ang_d = [0;0];

% define interference directions and corresponding desired response
ang_i = [12 10;13 10];
r_i = [-30 -30];

% define the region where sidelobe levels below desired level
ang_c_az_grid = [-20:0.5:-10 10:0.5:20]+ang_d(1);
ang_c_el_grid = [-20:0.5:-10 10:0.5:20]+ang_d(2);
[ang_c_az,ang_c_el] = meshgrid(ang_c_az_grid,ang_c_el_grid);
ang_c = [ang_c_az(:).';ang_c_el(:).'];
ang_c = setdiff(ang_c.',ang_i.','rows').';
r_c_th = -17;
r_c = r_c_th*ones(1,size(ang_c,2));
```

With all constraints defined, we can use optimization solver to derived the array weights that can provide us the desired pattern. For details on the optimization solver, please refer to the Array Pattern Synthesis Part II example.

```matlab
% Compute weights using optimization solvers
tic;
[w_op,Rn] = helperPatternSynthesis(pos,ang_d,[],[],[ang_i,ang_c],[r_i,r_c]);
```

Optimal solution found.

```matlab
t = toc;
```

The resulting pattern is displayed below

```matlab
az_plot = -90:0.25:90;
el_plot = -90:0.25:90;
w = w_op(1:N)+1i*w_op(N+1:2*N);
pat_opt = computePattern(pos,az_plot,el_plot,w);
plotPattern(az_plot,el_plot,pat_opt,'Optimization');
```
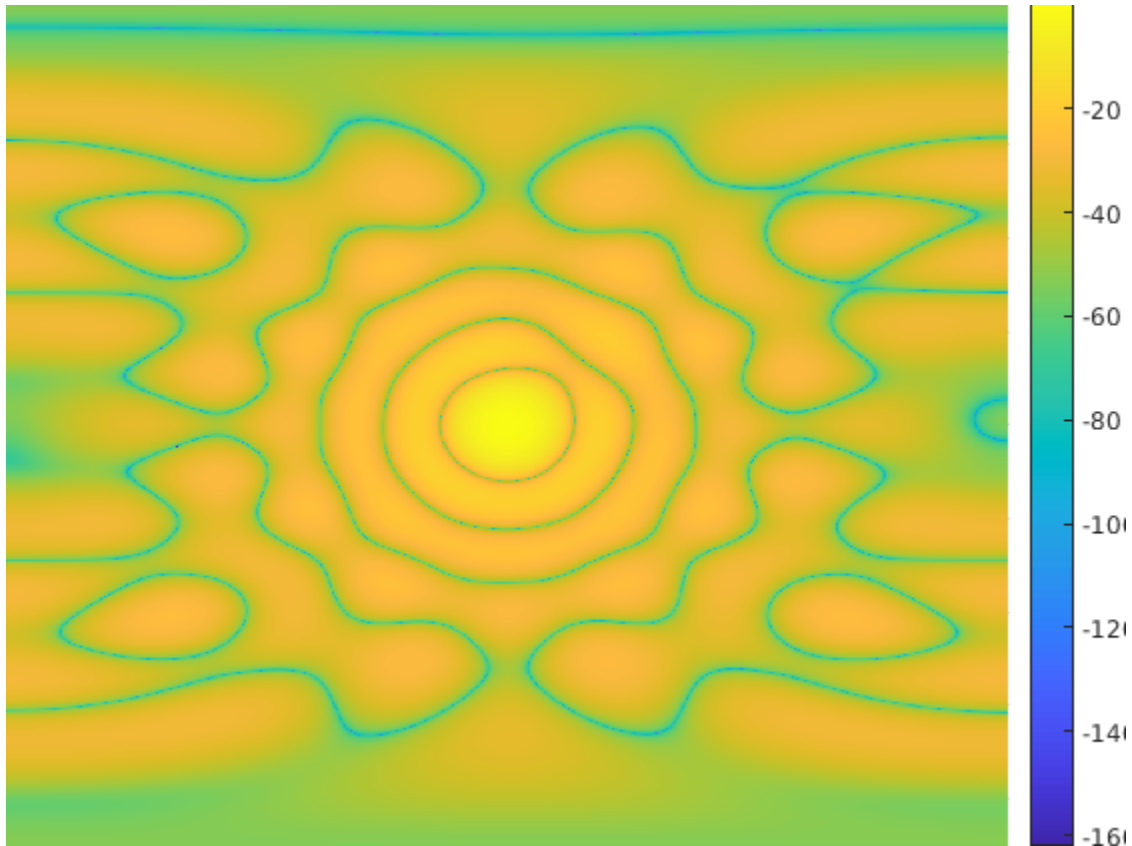


We can measure the quality of this pattern by measuring whether the resulting pattern matches the requirements.

```matlab
[d,intdb,sllpct] = measurePattern(pat_opt,ang_d,ang_i,ang_c,az_plot,el_plot,r_c_th,Rn,w);
d_max = pow2db(4*pi*N^2/abs(sum(Rn,"all")));
Metrics = ["Directivity (dBi)";"Interference Suppression (dB)";"Sidelobe Over Requirement Percent
Measured = {d;intdb;sllpct};
```

```
Requirement = {d_max;r_i;0};
metrictab = table(Metrics,Measured,Requirement)
```

```
metrictab=3×3 table
              Metrics                              Measured             Requirement
    _____       _____    _____

    "Directivity (dBi)"                      {[            22.2814]}    {[22.1824]}
    "Interference Suppression (dB)"          {[-29.3295 -30.6141]}     {[-30 -30]}
    "Sidelobe Over Requirement Percentage"   {[                 0]}    {[      0]}
```

We can see that the resulting metrics are very good. However, it does take some time to calculate the weights.

```
fprintf('Weights computation time: %f seconds.\n',t);
```

```
Weights computation time: 14.607490 seconds.
```

**Deep Learning Network**

To perform pattern synthesis using deep learning, we create a convolution neural network (CNN) as outlined in [1]. A beam pattern is defined over a range of azimuth and elevation angles. Therefore, the pattern can be represented as an image and our input layer take the image as the input. The output is the weights that produces such a pattern.

```
layers = [
imageInputLayer([721 721 1],'Normalization','zerocenter')

convolution2dLayer([1,64],1)
batchNormalizationLayer
reluLayer

convolution2dLayer([64,128],1)
batchNormalizationLayer
reluLayer

convolution2dLayer([128,128],1)
batchNormalizationLayer
reluLayer

convolution2dLayer([128,128],1)
batchNormalizationLayer
reluLayer

convolution2dLayer([128,128],1)
batchNormalizationLayer
reluLayer

fullyConnectedLayer(2000)
batchNormalizationLayer
reluLayer

fullyConnectedLayer(2000)
batchNormalizationLayer
reluLayer
```

```
fullyConnectedLayer(224)
regressionLayer()
];

analyzeNetwork(layers);
```

**Analysis for trainNetwork usage**

**Name:** layers

**Analysis date:** 15-Dec-2021 11:52:51

| | | 87.6M | 24 | 0 ⚠ | 0 🛑 |
| | | total learnables | layers | warnings | errors |

**ANALYSIS RESULT**

| | Name | Type | Activations | Learnable Prope... |
|---|---|---|---|---|
| 1 | imageinput<br>721×721×1 images with 'zerocenter' nor... | Image Input | 721(S) × 721(S) × 1(C) × 1(B) | - |
| 2 | conv_1<br>1 1×64 convolutions with stride [1 1] and... | Convolution | 721(S) × 658(S) × 1(C) × 1(B) | Weights 1 × 64<br>Bias 1 × 1 |
| 3 | batchnorm_1<br>Batch normalization | Batch Normalization | 721(S) × 658(S) × 1(C) × 1(B) | Offset 1 × 1<br>Scale 1 × 1 |
| 4 | relu_1<br>ReLU | ReLU | 721(S) × 658(S) × 1(C) × 1(B) | - |
| 5 | conv_2<br>1 64×128 convolutions with stride [1 1] a... | Convolution | 658(S) × 531(S) × 1(C) × 1(B) | Weights 64 × 128<br>Bias 1 × 1 |
| 6 | batchnorm_2<br>Batch normalization | Batch Normalization | 658(S) × 531(S) × 1(C) × 1(B) | Offset 1 × 1<br>Scale 1 × 1 |
| 7 | relu_2<br>ReLU | ReLU | 658(S) × 531(S) × 1(C) × 1(B) | - |
| 8 | conv_3<br>1 128×128 convolutions with stride [1 1] ... | Convolution | 531(S) × 404(S) × 1(C) × 1(B) | Weigh... 128 × 1...<br>Bias 1 × 1 |
| 9 | batchnorm_3<br>Batch normalization | Batch Normalization | 531(S) × 404(S) × 1(C) × 1(B) | Offset 1 × 1<br>Scale 1 × 1 |
| 10 | relu_3<br>ReLU | ReLU | 531(S) × 404(S) × 1(C) × 1(B) | - |
| 11 | conv_4<br>1 128×128 convolutions with stride [1 1] ... | Convolution | 404(S) × 277(S) × 1(C) × 1(B) | Weigh... 128 × 1...<br>Bias 1 × 1 |
| 12 | batchnorm_4<br>Batch normalization | Batch Normalization | 404(S) × 277(S) × 1(C) × 1(B) | Offset 1 × 1<br>Scale 1 × 1 |
| 13 | relu_4<br>ReLU | ReLU | 404(S) × 277(S) × 1(C) × 1(B) | - |
| 14 | conv_5<br>1 128×128 convolutions with stride [1 1] ... | Convolution | 277(S) × 150(S) × 1(C) × 1(B) | Weigh... 128 × 1...<br>Bias 1 × 1 |
| 15 | batchnorm_5<br>Batch normalization | Batch Normalization | 277(S) × 150(S) × 1(C) × 1(B) | Offset 1 × 1<br>Scale 1 × 1 |
| 16 | relu_5<br>ReLU | ReLU | 277(S) × 150(S) × 1(C) × 1(B) | - |
| 17 | fc_1<br>2000 fully connected layer | Fully Connected | 1(S) × 1(S) × 2000(C) × 1(B) | Weig... 2000 × 4...<br>Bias 2000 × 1 |
| 18 | batchnorm_6<br>Batch normalization | Batch Normalization | 1(S) × 1(S) × 2000(C) × 1(B) | Offs... 1 × 1 × 2...<br>Scale 1 × 1 × 2... |
| 19 | relu_6<br>ReLU | ReLU | 1(S) × 1(S) × 2000(C) × 1(B) | - |
| 20 | fc_2<br>2000 fully connected layer | Fully Connected | 1(S) × 1(S) × 2000(C) × 1(B) | Weigh... 2000 × 2...<br>Bias 2000 × 1 |
| 21 | batchnorm_7<br>Batch normalization | Batch Normalization | 1(S) × 1(S) × 2000(C) × 1(B) | Offs... 1 × 1 × 2...<br>Scale 1 × 1 × 2... |
| 22 | relu_7<br>ReLU | ReLU | 1(S) × 1(S) × 2000(C) × 1(B) | - |
| 23 | fc_2 | Fully Connected | 1(S) × 1(S) × 224(C) × 1(B) | Weigh... 224 × 20... |

Left panel node list: imageinput, conv_1, batchnorm_1, relu_1, conv_2, batchnorm_2, relu_2, conv_3, batchnorm_3, relu_3, conv_4, batchnorm_4, relu_4, conv_5, batchnorm_5, relu_5, fc_1, batchnorm_6, relu_6, fc_2, batchnorm_7, relu_7, fc_3, regressionoutput

**Training and Testing Data Synthesis**

To train and test the network, we can generate patterns with random mainlobe and interference placements. We will derive the optimal weights via optimization solver and then use the derived pattern as the input to compute the weights through network and hope that the resulting weights will match the optimal weights.

```
dataURL = 'https://ssd.mathworks.com/supportfiles/phased/data/ArraySynthesisDLData.zip';
saveFolder = tempdir;
dataDir = 'datasetSmallE';
zipFile = fullfile(saveFolder,'ArraySynthesisDLData.zip');
if ~exist(zipFile,'file')
    websave(zipFile,dataURL);
```

```
    % Unzip the data
    unzip(zipFile,tempdir)
end

signalDs= signalDatastore(fullfile(saveFolder,dataDir),"ReadFcn",@readTrainAndValFile);
signalDs =shuffle(signalDs);
```

Split the dataset into training and validation set

```
[trainInd,valInd,testInd] = dividerand(1:numel(signalDs.Files),0.8,0.1,0.1);
trainDs = subset(signalDs,trainInd);
validDs = subset(signalDs,valInd);
testDs = subset(signalDs,testInd);
testDs.ReadFcn = @readTestFile;
% Number of files used for training, validation and testing
fprintf('Number of training, validating, and testing data are %d, %d, and %d, respectively.',...
    numel(trainDs.Files),numel(validDs.Files),numel(testDs.Files));
```

```
Number of training, validating, and testing data are 778, 97, and 97, respectively.
```

**Train Network**

Now we can train the network

```
trainNetworkNow = false;
if trainNetworkNow
    options = trainingOptions('adam', ...
        'MaxEpochs',30,...
        'GradientDecayFactor', 0.9,...
        'SquaredGradientDecayFactor',0.999,...
        'InitialLearnRate',1e-3, ...
        'Verbose',false, ...
        'Plots','training-progress',...
        'LearnRateSchedule','none',...
        'ValidationData',validDs,...
        'ValidationFrequency',3); %#ok<UNRCH>
    net = trainNetwork(trainDs,layers,options);
else
    load(fullfile(saveFolder,'helperPatternSynthesisDL.mat'),'net');
end
```

**Test Trained Network**

Use the pattern derived in the early section, let us see if the trained network can provide a sastifying pattern

```
tic;
w_dl = double(predict(net, cast(pat_opt,'int8')))';
t_dl = toc;
w = w_dl(1:N)+1i*w_dl(N+1:2*N);
w = w./norm(arrayfactor(pos,ang_d,w));
pat_dl = computePattern(pos,az_plot,el_plot,w);
plotPattern(az_plot,el_plot,pat_dl,'Deep Learning');
```
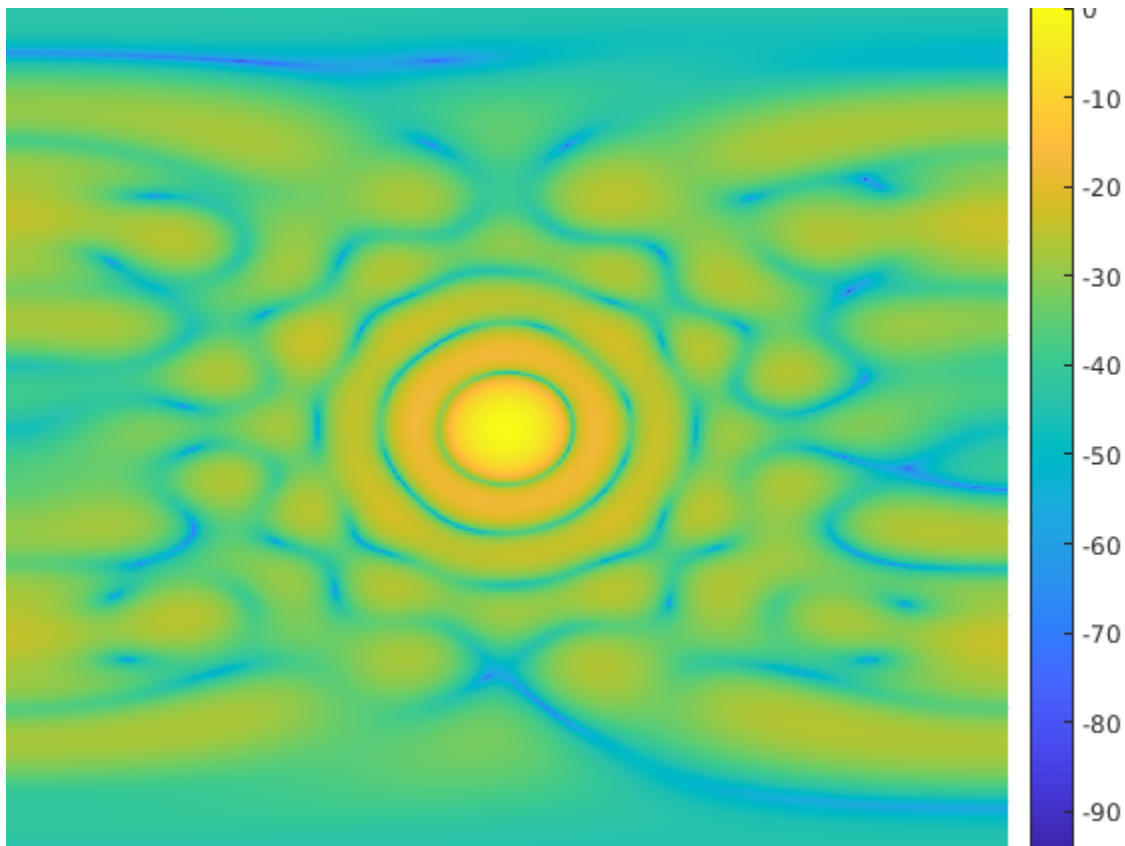
```
[d,intdb,sllpct] = measurePattern(pat_dl,ang_d,ang_i,ang_c,az_plot,el_plot,r_c_th,Rn,w);
d_max = pow2db(4*pi*N^2/abs(sum(Rn,"all")));
Metrics = ["Directivity (dBi)";"Interference Suppression (dB)";"Sidelobe Over Requirement Percen
Measured = {d;intdb;sllpct};
Requirement = {d_max;r_i;0};
metrictab = table(Metrics,Measured,Requirement)
```

metrictab=*3×3 table*

| Metrics | Measured | Requirement |
|---|---|---|
| "Directivity (dBi)" | {[ 22.1074]} | {[22.1824]} |
| "Interference Suppression (dB)" | {[-20.5114 -20.8268]} | {[-30 -30]} |
| "Sidelobe Over Requirement Percentage" | {[ 0]} | {[ 0]} |

```
fprintf('Weights computation time: %f seconds.\n',t_dl);
```

Weights computation time: 5.339215 seconds.

We can see that the predicted pattern gives good result in terms of directivity and sidelobe suppression. It is not as good as achieving the interference suppression. We may be able to improve this with more training data.

**Conclusion**

This example shows how to create and train a CNN to perform pattern synthesis for a given array. Although the deep learning network can generate pattern synthesis weights much faster, it has its

own drawbacks. For example, the network requires a large amount of data to train. In addition, the network is specific to a specific array geometry. Therefore, if the array formation changes, the network needs to be retrained.

### References

[1] Bianco, Simone, Maurizio Feo, Paolo Napoletano, Giovanni Petraglia, Alberto Raimondi, and Pietro Vinetti. "AESA Adaptive Beamforming Using Deep Learning." *Proceedings of 2020 IEEE Radar Conference,* 2020.

### Support Files

### Define Array

```
function [pos,N] = getCircularPlanarArrayPositions(radius,delta,lambda)
    n = round(radius/delta*2);
    htemp = phased.URA(n, delta, ...
        'Lattice', 'Rectangular');
    pos = getElementPosition(htemp)/lambda;
    elemToRemove = sum(pos.^2)>radius^2;
    pos(:,elemToRemove) = [];
    N = size(pos,2);
end
```

### Compute Pattern

```
function pat = computePattern(pos,az_plot,el_plot,w)
    pat = zeros(numel(el_plot),numel(az_plot));
    for m = 1:numel(el_plot)
        pat(m,:) = mag2db(abs(arrayfactor(pos,[az_plot;el_plot(m)*ones(size(az_plot))],w)));
    end
end
```

### Plot Pattern

```
function plotPattern(az_plot,el_plot,pat,algstr)
    surf(az_plot,el_plot,pat,'EdgeColor','none');
    colorbar;
    view(0,90);
    xlabel('Azimuth (deg)');
    ylabel('Elevation (deg)');
    title(sprintf('Beam Pattern via %s',algstr));
    axis tight
end
```

### Measure Pattern Metrics

```
function [d,intdb,sllpct] = measurePattern(pat,ang_d,ang_i,ang_c,az_plot,el_plot,sllbar,Rn,w)
az_step = mean(diff(az_plot));
el_step = mean(diff(el_plot));
d = pat(sub2ind(size(pat),(ang_d(1)-az_plot(1))/az_step+1,(ang_d(2)-el_plot(1))/el_step+1))+...
    pow2db(4*pi)-pow2db(abs(w'*Rn*w));
intdb = pat(sub2ind(size(pat),(ang_i(1,:)-az_plot(1))/az_step+1,(ang_i(2,:)-el_plot(1))/el_step+1
slldb = pat(sub2ind(size(pat),(ang_c(1,:)-az_plot(1))/az_step+1,(ang_c(2,:)-el_plot(1))/el_step+1
sllpct = sum(slldb>sllbar)/numel(slldb);
end
```

**Read Data Files**

```
function dataOut = readTrainAndValFile(filename)

    data = struct2cell(load(filename));
    data = data{1};

    dataOut{1} = data{1};
    dataOut{2} = normalize(data{2})';
end


function dataOut = readTestFile(filename)

    data = struct2cell(load(filename));
    data = data{1};
    dataOut = data;
end
```

# Spectrum Sensing with Deep Learning for Radar and Wireless Communications

This example shows how to perform spectrum sensing using a semantic segmentation neural network trained with synthesized radar and wireless communication signals. The trained neural network can identify radar and wireless communication signals that appear in the same received spectrum. In addition, the network can identify the occupied bandwidth of the received signals.

**Introduction**

Due to the increasing demands for higher speeds and greater coverage, modern wireless communication systems are moving to higher frequency bands and larger signal bandwidths. These higher frequency bands are at ranges traditionally used by radar systems. As a result, the spectrum of radar systems and wireless communication systems may overlap, which drives the need for spectrum sharing. This requires that future radar and wireless communication systems include spectrum sensing to detect occupied space to avoid conflicts. This example shows how to model a spectrum sensing system. We first synthesize a data set that consists of radar and wireless communications signals. This data is used to train and test a deep learning network. The trained network is then used to identify the signals and the corresponding occupied bandwidth of these signals.

**Setup the Scene**

Consider an airport surveillance radar located at the scenario origin. The radar operates at 2.8 GHz and uses a reflector antenna with a gain of 32.8 dB. The transmit power is set at 25 kW.

```
% setup waveform
fc=2.8e9;                    % center frequency [Hz]
fs=61.44e6;                  % sampling frequency [Hz]
prf=fs/ceil(fs/1050);        % pulse repetition rate [Hz]
pulseWidth=1e-06;            % pulsewdith [s]
wav = phased.RectangularWaveform('SampleRate',fs,'PulseWidth',pulseWidth,'PRF',prf,'NumPulses',3

% setup antenna
rad=2.5;           % radius [m]
flen=2.5;          % focal length [m]
antele=design(reflectorParabolic('Exciter',horn),fc);
antele.Exciter.Tilt=90;
antele.Exciter.TiltAxis=[0 1 0];
antele.Tilt=90;
antele.TiltAxis=[0 1 0];
antele.Radius=rad;
antele.FocalLength=flen;
ant=phased.ConformalArray('Element',antele);

% setup transmitter and receiver
power_ASR=25000;    % transmit power [W]
gain_ASR=32.8;      % transmit gain [dB]
radartx = phased.Transmitter('PeakPower',power_ASR,'Gain',gain_ASR);
```

Since 5G and LTE signals are present in the vicinity of airport radars, we define a set of signals for these wireless standards using 5G Toolbox and LTE Toolbox. For more details on how to generate these types of signals, please refer to the example "Spectrum Sensing with Deep Learning to Identify 5G and LTE Signals" .

For the wireless signals, the gain and the power of the transmitter may change from frame to frame. The receivers are assumed to be randomly placed in a 2 km x 2km region and equipped with isotropic antennas.

```
rxpos_horiz_minmax=[-1000 1000];
rxpos_vert_minmax=[0 2000];
```

The region also contains many scatterers which makes the propagation channels more challenging to operate in. In this example, we assume the channel contains 30 scatterers, randomly distributed in the region. You can use phased.ScatteringMIMOChannel to simulate the scatterers.

```
% define radar position
radartxpos=[0 0 15]';
radartxaxes=rotz(0);
radarchan=phased.ScatteringMIMOChannel('TransmitArray',ant,...
    'ReceiveArray',phased.ConformalArray('Element',phased.IsotropicAntennaElement),...
    'CarrierFrequency',fc,...
    'SpecifyAtmosphere',true,...
    'SampleRate',fs,...
    'SimulateDirectPath',false,...
    'MaximumDelaySource','Property',...
    'MaximumDelay',1,...
    'TransmitArrayMotionSource','Property',...
    'TransmitArrayPosition',radartxpos,...
    'TransmitArrayOrientationAxes',radartxaxes,...
    'ReceiveArrayMotionSource','Input port',...
    'ScattererSpecificationSource','Input port');

% define wireless transmitter position
commtxpos=[200 0 450]';
commtxaxes=rotz(0);
commchan=phased.ScatteringMIMOChannel('TransmitArray',phased.ConformalArray('Taper',10),...
    'ReceiveArray',phased.ConformalArray('Element',phased.IsotropicAntennaElement),...
    'CarrierFrequency',fc,...
    'SpecifyAtmosphere',true,...
    'SampleRate',fs,...
    'SimulateDirectPath',false,...
    'MaximumDelaySource','Property',...
    'MaximumDelay',1,...
    'TransmitArrayMotionSource','Property',...
    'TransmitArrayPosition',commtxpos,...
    'TransmitArrayOrientationAxes',commtxaxes,...
    'ReceiveArrayMotionSource','Input port',...
    'ScattererSpecificationSource','Input port');
```

**Generate Training Data**

To generate the training data, define the data folder and class names for noise only data, LTE data, 5G NR data, and radar data. Because we are looking for the waveform type and the occupied bandwidth, the helper function below generates a spectrogram.

```
% Number of data generated
numTrainingData=500;
imageSize=[256 256];

% Define data directory
classNames = ["Noise" "LTE" "NR" "Radar"];
pixelLabelID = [0 1 2 3];
```

```
useDownloadedDataSet = true;
saveFolder = tempdir;
trainingFolder=fullfile(saveFolder,'RadarCommTrainData');
if ~useDownloadedDataSet
    % Generate data
    helperGenerateRadarCommData(fs,wav,radartx,radarchan,commchan,rxpos_horiz_minmax,rxpos_vert_
else
    dataURL = 'https://ssd.mathworks.com/supportfiles/phased/data/RadarCommSpectrumSensingData.zi
    zipFile = fullfile(saveFolder,'RadarCommSpectrumSensingData.zip');
    if ~exist(zipFile,'file')
        websave(zipFile,dataURL);
        % Unzip the data
        unzip(zipFile,saveFolder)
    end

end
```

**Load Training Data**

Use the `imageDatastore` function to load training images with the spectrogram of radar and wireless communication signals. The `imageDatastore` function enables you to efficiently load a large collection of images from disk. Spectrogram images are stored in .png files.

```
% Data
trainingFolder = fullfile(saveFolder,'RadarCommTrainData');
imds = imageDatastore(trainingFolder,'IncludeSubfolders',false,'FileExtensions','.png');
```

To label the occupied bandwidth for each signal, use the `pixelLabelDatastore` (Computer Vision Toolbox) function to load spectrogram pixel label image data. Each pixel in the spectrogram images is labeled as one of "Radar", "NR", "LTE" or "Noise" categories. A pixel label datastore encapsulates the pixel label data and the label ID to a class name mapping. Pixel labels are stored in .hdf files.

```
% Label
pxdsTruth = pixelLabelDatastore(trainingFolder,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
```

**Analyze Dataset Statistics**

To see the distribution of class labels in the training dataset, use the `countEachLabel` (Computer Vision Toolbox) function to count the number of pixels by class label, and plot the pixel counts by class.

```
tbl = countEachLabel(pxdsTruth);
frequency = tbl.PixelCount/sum(tbl.PixelCount);
figure
bar(1:numel(classNames),frequency)
```

Warning: MATLAB has disabled some advanced graphics rendering features by switching to software (

```
grid on
xticks(1:numel(classNames))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```
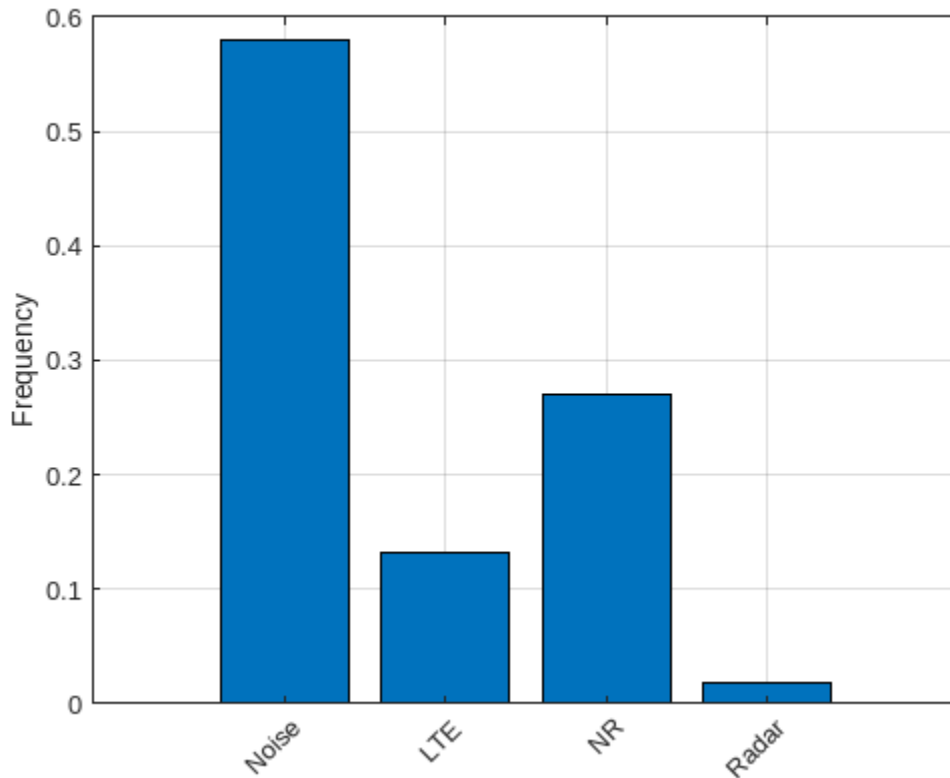
**Prepare Training, Validation, and Test Sets**

The deep neural network uses 80% of the single signal images from the dataset for training and, 20% of the images for validation. The `helperSpecSensePartitionData` function randomly splits the image and pixel label data into training and validation sets.

```
[imdsTrain,pxdsTrain,imdsVal,pxdsVal] = helperRadarCommPartitionData(imds,pxdsTruth,[80 20]);
cdsTrain = pixelLabelImageDatastore(imdsTrain,pxdsTrain,'OutputSize',imageSize);
cdsVal = pixelLabelImageDatastore(imdsVal,pxdsVal,'OutputSize',imageSize);
```

**Train Deep Neural Network**

Use the `deeplabv3plusLayers` (Computer Vision Toolbox) function to create a semantic segmentation neural network. Choose `resnet50` (Deep Learning Toolbox) as the base network and specify the input image size (number of pixels used to represent time and frequency axes) and the number of classes. If the Deep Learning Toolbox™ Model *for ResNet-50 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet50` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
baseNetwork = 'resnet50';
lgraph = deeplabv3plusLayers(imageSize,numel(classNames),baseNetwork);
```

18-687

**Balance Classes Using Class Weighting**

To improve training when classes in the training set are not balanced, you can use class weighting to balance the classes. Use the pixel label counts computed earlier with the `countEachLabel` function and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;
```

Specify the class weights using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights
lgraph = replaceLayer(lgraph,"classification",pxLayer);
```

**Select Training Options**

Configure training using the `trainingOptions` (Deep Learning Toolbox) function to specify the stochastic gradient descent with momentum (SGDM) optimization algorithm and the hyper-parameters used for SGDM. To get the best performance from the network, you can use the Experiment Manager (Deep Learning Toolbox) to optimize training options.

```
opts = trainingOptions("sgdm",...
  MiniBatchSize = 40,...
  MaxEpochs = 40, ...
  LearnRateSchedule = "piecewise",...
  InitialLearnRate = 0.02,...
  LearnRateDropPeriod = 10,...
  LearnRateDropFactor = 0.1,...
  ValidationData = cdsVal,...
  ValidationPatience = 5,...
  Shuffle="every-epoch",...
  Plots = 'training-progress',...
  ExecutionEnvironment = 'auto');
```

Train the network using the combined training data store, `cdsTrain`. The combined training data store contains single signal frames and true pixel labels. One example of training progress is shown in the following figure.
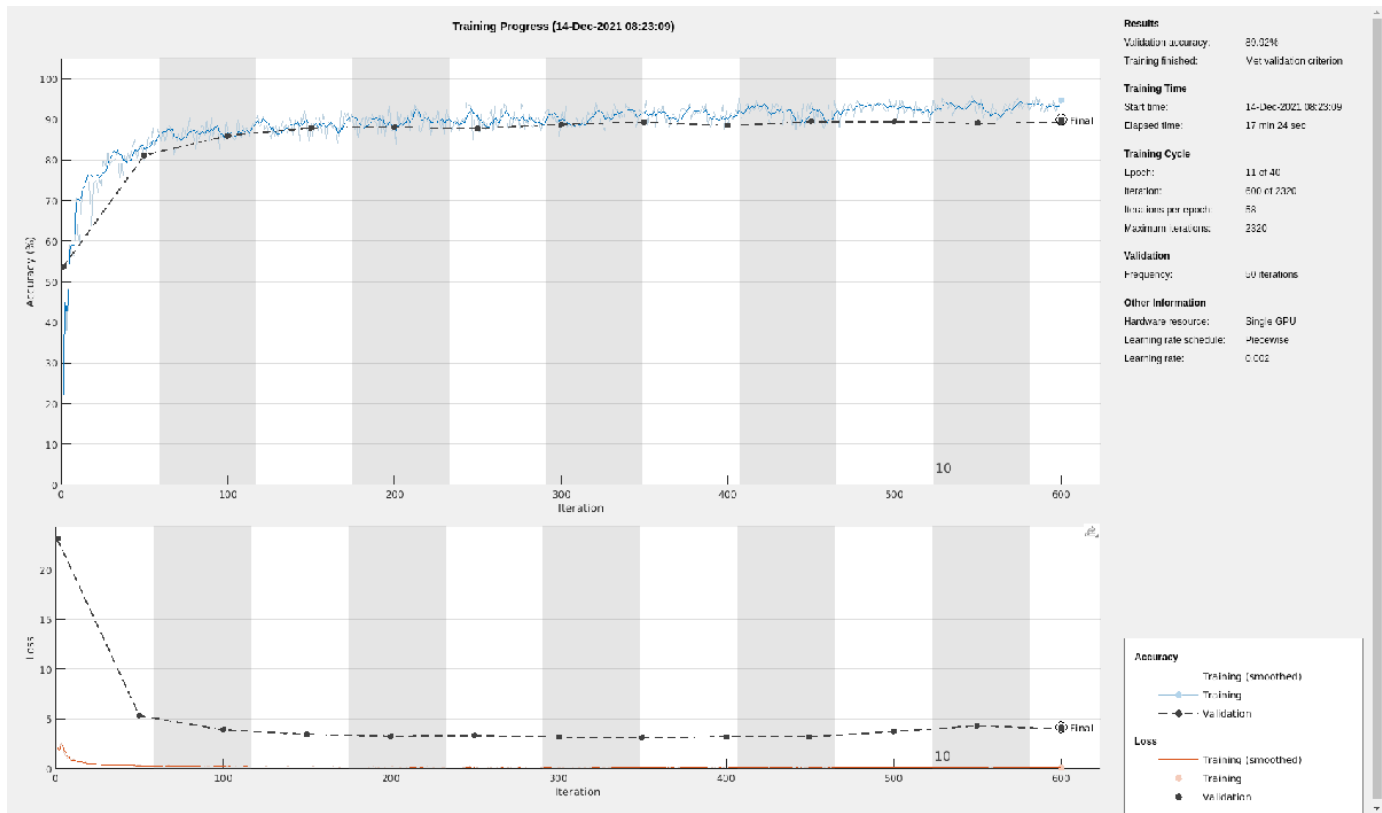
If you want to train the network, set `trainNow` to `true`. Otherwise, the example will load a pre-trained network.

```
trainNow = false;
if trainNow
  [net,trainInfo] = trainNetwork(cdsTrain,lgraph,opts);   %#ok<UNRCH>
else
  load(fullfile(saveFolder,'helperRadarCommTrainedNet.mat'),'net');
end
```

**Test with Synthetic Signals**

Test the network signal identification performance using signals that contain 5G NR, LTE, and radar signals. Use the `semanticseg` (Computer Vision Toolbox) function to get the pixel estimates of the spectrogram images in the test data set. Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute various metrics to evaluate the quality of the semantic segmentation results.

```
% Generate testing data
numTestData=100;
testFolder = fullfile(saveFolder,'RadarCommTestData');
if ~useDownloadedDataSet
    helperGenerateRadarCommData(fs,wav,radartx,radarchan,commchan,rxpos_horiz_minmax,rxpos_vert_r
end

% Prep and test the testing data
imds = imageDatastore(testFolder,'IncludeSubfolders',false,'FileExtensions','.png');
pxdsResults = semanticseg(imds,net,"WriteLocation",saveFolder,'ExecutionEnvironment','auto');
```

```
Running semantic segmentation network
-------------------------------------
* Processed 588 images.
```

```
% Prep the ground truth testing labels
pxdsTruth = pixelLabelDatastore(testFolder,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTruth);
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 588 images.
* Finalizing... Done.
* Data set metrics:
```

| GlobalAccuracy | MeanAccuracy | MeanIoU | WeightedIoU | MeanBFScore |
| --- | --- | --- | --- | --- |
| 0.9035 | 0.91356 | 0.73372 | 0.83048 | NaN |

Plot the normalized confusion matrix for all test frames.

```
cm = confusionchart(metrics.ConfusionMatrix.Variables, ...
  classNames, Normalization='row-normalized');
cm.Title = 'Normalized Confusion Matrix';
```

**Normalized Confusion Matrix**

| True Class \ Predicted Class | LTE | NR | Noise | Radar |
| --- | --- | --- | --- | --- |
| LTE | 89.1% | 7.9% | 1.6% | 1.5% |
| NR | 4.6% | 84.7% | 8.9% | 1.8% |
| Noise | 1.5% | 3.5% | 93.2% | 1.8% |
| Radar | 0.5% | 0.3% | 0.8% | 98.4% |
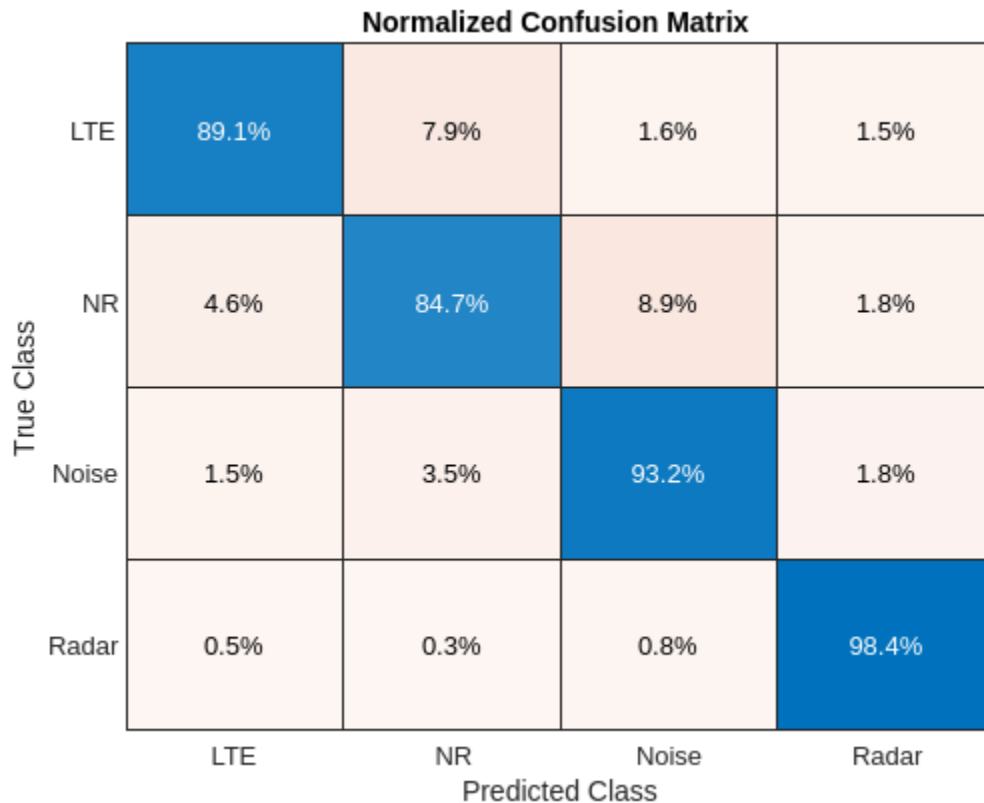
Plot the histogram of the per-image intersection over union (IoU). For each class, IoU is the ratio of correctly classified pixels to the total number of ground truth and predicted pixels in that class.

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
grid on
xlabel('IoU')
ylabel('Number of Frames')
title('Frame Mean IoU')
```
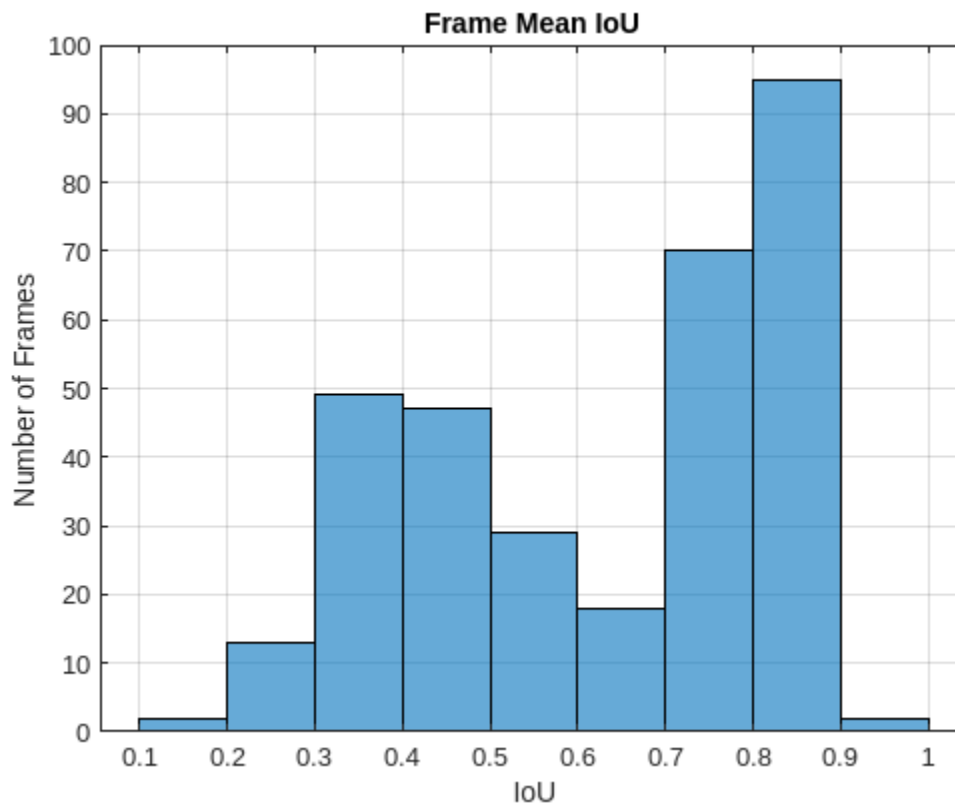


### Identify Radar, 5G NR, and LTE Signals in Spectrogram

Visualize the received spectrum, true labels, and predicted labels for the image with index 142. We can see that the network correctly picked out the radar signal from wireless communicatino signals.

```
imgIdx = 142;
rcvdSpectrogram = readimage(imds,imgIdx);
trueLabels = readimage(pxdsTruth,imgIdx);
predictedLabels = readimage(pxdsResults,imgIdx);
figure
helperSpecSenseDisplayResults(rcvdSpectrogram,trueLabels,predictedLabels, ...
  classNames,fs,0,1/prf)
```
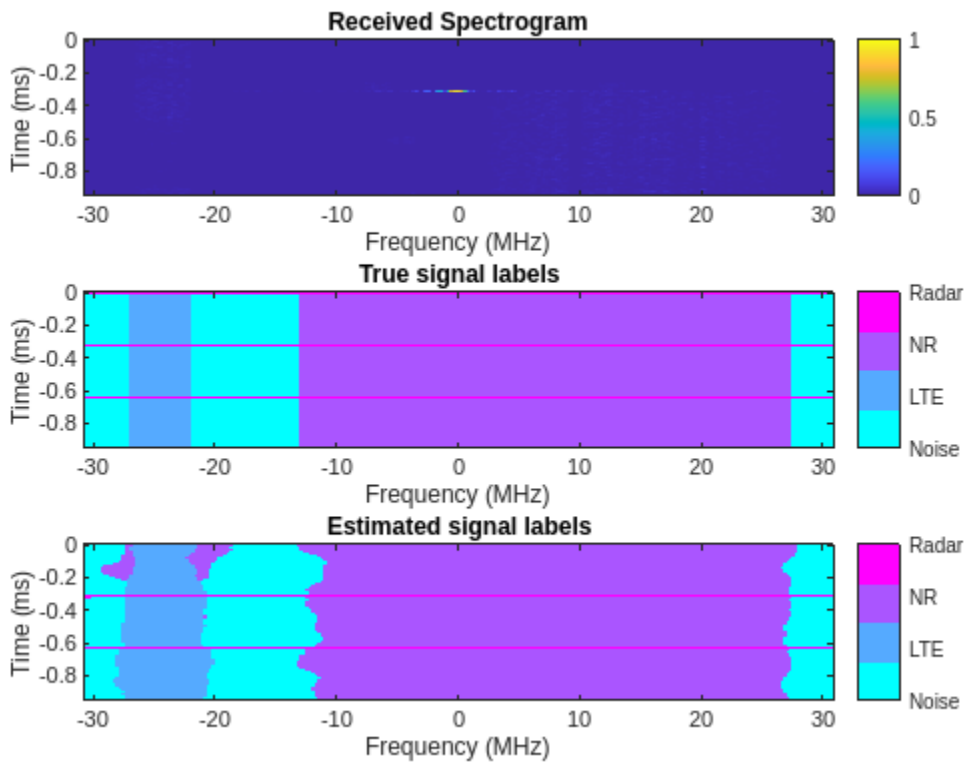
## Conclusions

The trained network can distinguish radar and wireless communication signals. The network may not be able to identify every captured signal correctly. In such cases, enhance the training data either by generating more representative synthetic signals or capturing over-the-air signals and including these in the training set.

# Evaluating Interference from 5G New Radio (NR) Signals at an Airport Surveillance Radar

As the frequency range of 5G NR systems increase beyond the bands used in LTE, spectrum management becomes more complex. The demand for expanded 5G coverage is driven by the benefits of higher data rates and lower latencies. The implementation of new 5G base stations in turn drives the need to understand how these signals affect installed systems that operate around the same frequency bands. One such system is an air traffic control radar.

In this example, you will learn how to model an air traffic control radar which operates in the vicinity of a 5G base station. We will look at how 5G signals affect the received radar signals. We use a scenario with an aircraft approaching the airport from the same direction as the base station. The radar return from the aircraft is processed with and without the presence of the 5G waveform.

Many different mitigation techniques can be applied to reduce interference, but in this example, we implement a beamformer at the base station that reduces transmitted energy in the direction of the radar.

**Scenario**

We first set up a scenario with the basestation location based on guidance in the 3GPP TR 38.901 specification. The base station is mounted on a rooftop on a building at a distance of 2.5 km from the airport.

The airport radar model is based on an Airport Surveillance Radar (ASR-11).

An aircraft is approaching the radar from the same direction as the base station.

Scatterers are added to the scenario to account for multipath reflections due to buildings in the path between the aircraft and the radar.

```
% Airport Radar position
asrPos=[0; 0; 15];
asrAxes = eye(3);
asrVel = [0; 0; 0];

% 5G Base Station position
bsPos = [2.5e+03; 0; 25];
bsAxes = rotz(180);

% Approaching aircraft starting position and velocity
targetPos = [50E3;0;450];
targetVel = [0;0;0];

% Channel parameters

% Generate 15 scatterers at random positions
rng(2021);                          % Initialize random number generator
numScat = 15;
azRange = -180:+180;
randAzOrder = randperm(length(azRange));
elRange = 0;
rRange = 0.5e3 : 2.5 : 1e3;
randROrder = randperm(length(rRange));
```

```
azAngInSph = deg2rad(azRange(randAzOrder(1:numScat)));
elAngInSph = zeros(size(azAngInSph));
r = (rRange(randROrder(1:numScat)));

% Transform spherical coordinates to Cartesian coordinates
[x,y,z] = sph2cart(azAngInSph,elAngInSph,r);
scatPos = [x;y;z] + bsPos;

scatCoef = ones(numScat,1)*1e-1;
```

You can plot the location of the 5G tower, the radar, and the scatterers.

```
figure; hold on; grid on;
plot3(asrPos(1),asrPos(2),asrPos(3),...
    'color','b','marker','o','MarkerFaceColor','b', ...
    'DisplayName','Airport Radar')
plot3(scatPos(1,:),scatPos(2,:),scatPos(3,:), ...
    'LineStyle','none','color','k','marker','square','MarkerFaceColor','k', ...
    'DisplayName','Scatterers')
plot3(bsPos(1),bsPos(2),bsPos(3), ...
    'color','r','marker','hexagram','MarkerFaceColor','r', ...
    'DisplayName','Base Station')
xlabel('X (m)'); ylabel('Y (m)'); legend('location','best')
```
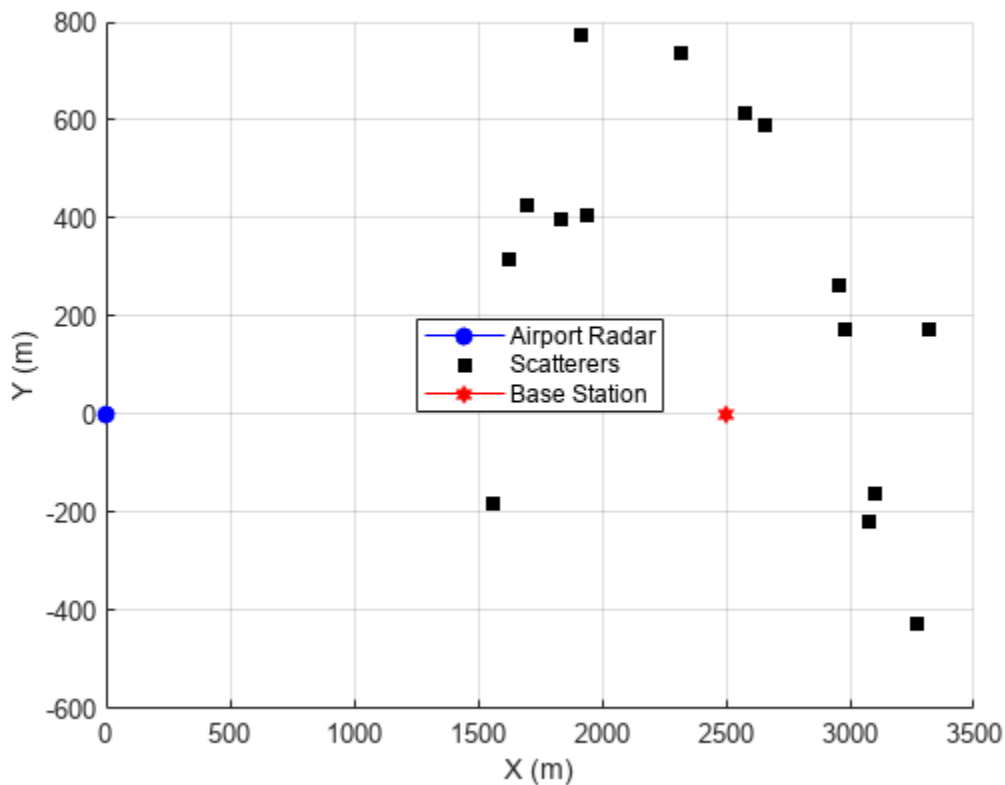


**Radar Configuration**

You configure a radar model to match the parameters of an Airport Surveillance Radar (ASR-11).

```
fc    = 2.8e9;                      % Radar carrier frequency (Hz)
c     = physconst('Lightspeed');    % Propagation speed (m/s)
lambda = c/fc;                      % Radar wavelength (m)
```

The radar transmits a linear FM waveform.

```
asrSamplingFrequency = 1e6; %  (Hz)
PRF = 1000;                   % Pulse Repition Frequency (Hz)
dutyCycle = 0.1;


asrWaveform = phased.LinearFMWaveform( ...
    'SampleRate', asrSamplingFrequency, ...
    'DurationSpecification', 'Duty cycle', ...
    'DutyCycle', dutyCycle, ...
    'PRF', PRF);
```

The ASR-11 uses a parabolic reflector with a horn exciter. You can use an antenna element from Antenna Toolbox to generate the design.

```
rad=2.5;        % radius [m]
flen=2.5;       % focal length
% Create antenna
% (create)
asrAntennaElement=design(reflectorParabolic('Exciter',horn),fc);
% (orient)
asrAntennaElement.Exciter.Tilt=90;
asrAntennaElement.Exciter.TiltAxis=[0 1 0];
asrAntennaElement.Tilt=90;
asrAntennaElement.TiltAxis=[0 1 0];
% (params)
asrAntennaElement.Radius=rad;
asrAntennaElement.FocalLength=flen;

% (final create)
asrAntenna = phased.ConformalArray('Element',asrAntennaElement);


asrRadiator = phased.Radiator('OperatingFrequency',fc,...
    'Sensor', asrAntenna);
```

Define the radar transmitter

```
asrPower = 25000; % [W]
asrGain = 32.8; % [dB]
asrTransmitter = phased.Transmitter('PeakPower',asrPower, ...
    'Gain',asrGain);
```

Define the radar receiver antenna array and he receiver pre-amplifier.

```
asrCollector = phased.Collector('OperatingFrequency', fc, ...
    'Sensor', asrAntenna);

asrReceiver = phased.ReceiverPreamp( 'SampleRate', asrSamplingFrequency, ...
    'Gain', 20, ...
```

**18-695**

```
    'NoiseMethod','Noise temperature',...
    'NoiseFigure',6.31);
```

Define the approaching aircraft with radar cross section of 5m^2.

```
radarTarget = phased.RadarTarget(...
    'MeanRCS',5, ...
    'OperatingFrequency',fc);
```

Define the radar signal propagation environment as free space.

```
radarEnv = phased.FreeSpace('TwoWayPropagation',true, ...
    'SampleRate', asrSamplingFrequency, ...
    'OperatingFrequency', fc, ...
    'PropagationSpeed', c);
```

**Base Station Configuration**

You can generate a range of 5G waveforms using 5G Toolbox. Waveform data for this example is loaded from the referenced .mat file.

```
load helperRadarNRInterferenceData.mat
bsSamplingFrequency = waveStruct.Fs;
bsWaveform = waveStruct.waveform;
```

You define a base station antenna array based on the specification in 3GPP TR 28.901:

```
bsArray = phased.NRRectangularPanelArray('Size', [3, 2, 2, 2], ...
    'Spacing',[0.5*lambda, 0.5*lambda, 3*lambda, 3*lambda]);

bsSteeringVec = phased.SteeringVector('SensorArray',bsArray, ...
    'PropagationSpeed',c);

% Gain per antenna element
bsTxPower = 9; % watts
bsTxGain = -8; % dB
bsNumArrayElements = prod(bsArray.Size);
bsTransmitter = phased.Transmitter(...
    'PeakPower',bsTxPower/bsNumArrayElements, ...
    'Gain',bsTxGain);

[bs2asrRange, bs2asrAngle] = rangeangle(asrPos,bsPos,rotz(180));

bsArrayWeights = bsSteeringVec(fc,bs2asrAngle(1)+1);
bsArray.Taper = bsArrayWeights;

% Channel for base station signal
bsChannel = phased.ScatteringMIMOChannel('TransmitArray',bsArray,...
    'ReceiveArray',asrAntenna,...
    'CarrierFrequency',fc,...
    'SpecifyAtmosphere',true,...
    'SampleRate',bsSamplingFrequency,...
    'SimulateDirectPath',true,...
    'MaximumDelaySource','Property',...
    'MaximumDelay',1e-4,...
    'TransmitArrayMotionSource','Property',...
    'TransmitArrayPosition',bsPos,...
```

```
    'TransmitArrayOrientationAxes',bsAxes,...
    'ReceiveArrayMotionSource', 'Property', ...
    'ReceiveArrayPosition', asrPos, ...
    'ReceiveArrayOrientationAxes', asrAxes, ...
    'ScattererSpecificationSource','Property',...
    'ScattererPosition',scatPos,...
    'ScattererCoefficient',scatCoef');
```

### Received Signals Generation

You simulate the radar return and the 5G signal propagation to generate the received signal at the radar receiver.

### Simulating the Radar Return

The simulation steps to transmit and receive the radar signals follow. Note that the antenna dish of ASR-11 normally rotates and this can be modeled as well but for this example, we consider the simplified case where the ASR-11 is pointing toward the base station. In reality, the scatterers defined in the MIMO channel would cause multipath effects and generate radar reflections. However, these reflections are cancelled by signal processing techniques such as moving target indicator (MTI) processing. To simplify the simulation without loss of generality, we model the radar signal propagation in free space.

```
asrSignal = asrWaveform();
asrAmplifiedSignal = asrTransmitter(asrSignal);

[asr2TragetRange, asr2TargetAngle] = ...
    rangeangle(targetPos,asrPos);

asrRadiatedSignal = asrRadiator(asrAmplifiedSignal, ...
    asr2TargetAngle);

asrPropagatedSignal = radarEnv(...
    asrRadiatedSignal, ...
    asrPos,...
    targetPos,...
    asrVel, ...
    targetVel);


asrTargetReflection = radarTarget(asrPropagatedSignal);

asrCollectedSignal = asrCollector(asrTargetReflection, ...
    asr2TargetAngle);
```

### Simulating the 5G Signal Propagation

The 5G signal is propagated form the base station to the radar system.

```
bsAmplifiedSignal = bsTransmitter(bsWaveform);
bsCollectedSignal = bsChannel(repmat(bsAmplifiedSignal,1,2*bsNumArrayElements));
```

### Sampling Base Station and Radar Signals

You resample the received 5G signal to match the sampling frequency of the radar.

```
asrNsamples = length(asrSignal);
asrTvec = (0:asrNsamples-1)/asrSamplingFrequency;
```

```
bsNsamples = length(bsWaveform);
bsTvec = (0:bsNsamples-1)/bsSamplingFrequency;
bsCollectedSignal_Resampled = interp1( ...
    bsTvec,bsCollectedSignal,asrTvec).';

ReceivedSignal = asrReceiver(asrCollectedSignal + bsCollectedSignal_Resampled );
```

**Signal Processing**

You apply matched filtering, time variying gain, and CFAR detection to detect the target in the received signal.

```
% Matched filter
coeff = getMatchedFilter(asrWaveform);
mf = phased.MatchedFilter('Coefficients',coeff,'GainOutputPort',true);
matchingdelay = size(coeff,1)-1;

% Time varying gain
rgates = asrTvec*3e8/2;
tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(rgates,lambda),...
    'ReferenceLoss', ...
    2*fspl(asrNsamples/asrSamplingFrequency*c/2,lambda));

% CFAR Detector
cfar = phased.CFARDetector('NumTrainingCells',50,'NumGuardCells',2);
cfar.ThresholdFactor = 'Custom';
cfar.CustomThresholdFactor = 6;
cfar.ThresholdOutputPort = true;


[sigp,Gmf] = mf(ReceivedSignal);
sigp = buffer(sigp(matchingdelay+1:end),size(sigp,1));


sigp = tvg(sigp);
[detected, th] = cfar(abs(sigp).^2,1:length(sigp));

figure; hold on;
plot(rgates*1e-3,abs(sigp).^2);
plot(rgates*1e-3,th);
plot(rgates(detected)*1e-3,abs(sigp(detected)).^2,'o','markersize',10)
legend('Signal','Threshold');
xlabel('range (km)');
```
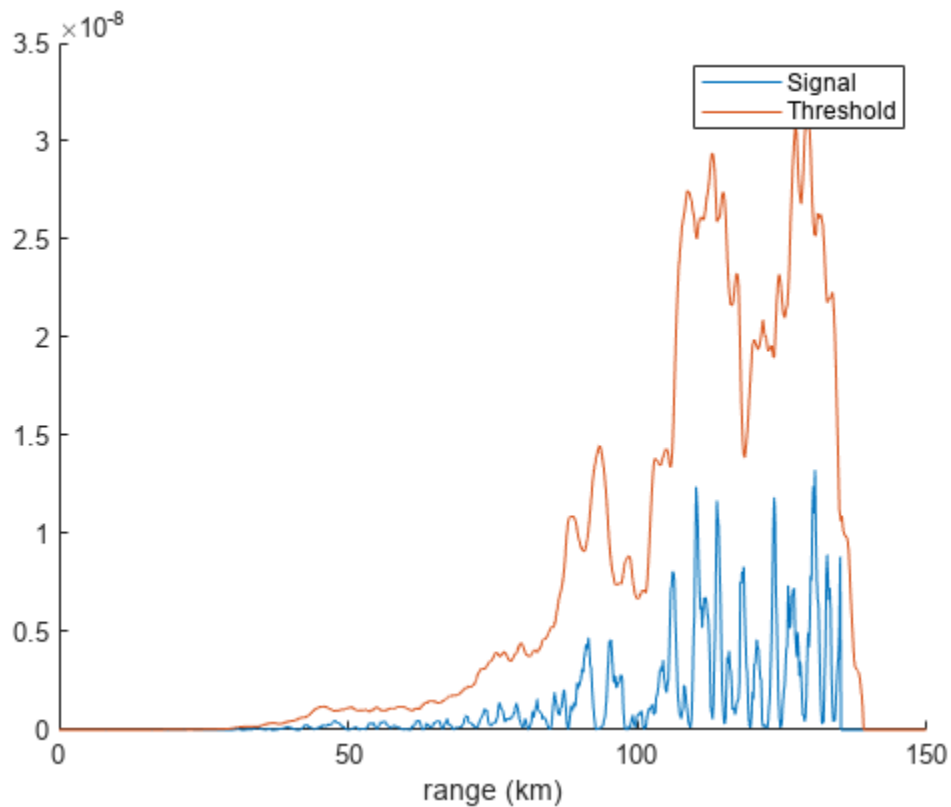
**5G Base Station Beamforming**

To eliminate the interference, a beamformer is added to the base station. Since the location of the airport radar is fixed, a null can be placed at this direction.

```
release(bsChannel)
% Calculate the steering vector for null directions
wn = bsSteeringVec(fc,bs2asrAngle(1));

% Calculate the steering vectors for lookout directions
bsLookAngle = 1; % Note: This angle should be different than the null angle.
wd = bsSteeringVec(fc,bsLookAngle);

% Compute the response of desired steering at null direction
rn = wn'*wd/(wn'*wn);

% Sidelobe canceler - remove the response at null direction
bsArrayWeights = wd-wn*rn;

bsChannel.TransmitArray.Taper = bsArrayWeights;
figure;
pattern(bsChannel.TransmitArray,fc,-180:180,0)
```

Azimuth Cut (elevation angle = 0.0°)



Directivity (dBi), Broadside at 0.00 °
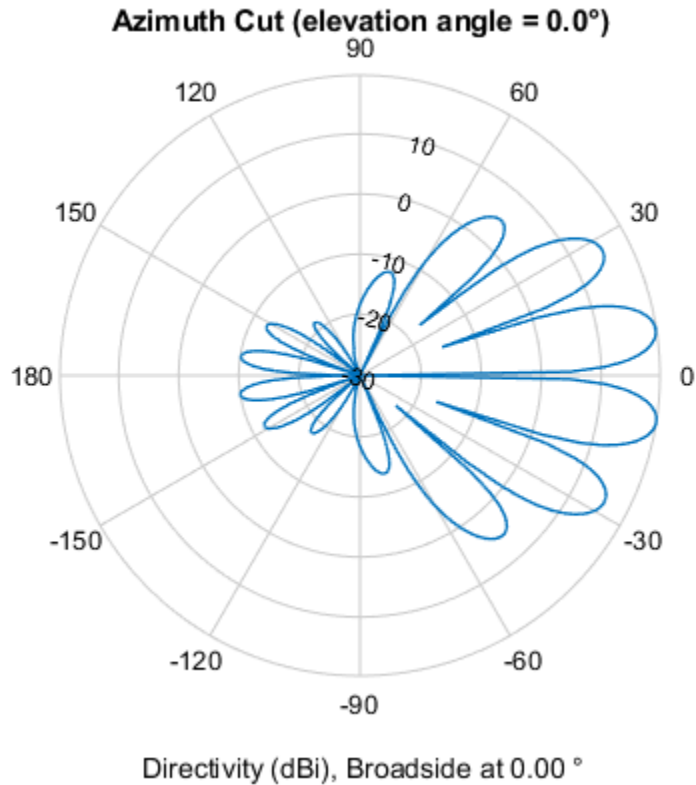
**Simulating the 5G Signal Propagation after Beamforming**

```
bsCollectedSignal_BF = bsChannel(repmat(bsAmplifiedSignal,1,2*bsNumArrayElements));
bsCollectedSignal_BF_Resampled = interp1( ...
    bsTvec,bsCollectedSignal_BF,asrTvec).';

ReceivedSignal_BF = asrReceiver(asrCollectedSignal + bsCollectedSignal_BF_Resampled);


[sigp_BF,Gmf_BF] = mf(ReceivedSignal_BF);
sigp_BF = buffer(sigp_BF(matchingdelay+1:end),size(sigp_BF,1));

sigp_BF = tvg(sigp_BF);
[detected_BF, th_BF] = cfar(abs(sigp_BF).^2,1:length(sigp_BF));
```

With beamforming in place, the signal level from the base station as seen by the radar is reduced such that it doesn't interfere with the radar return.

```
figure; hold on;
plot(rgates*1e-3,abs(sigp_BF.^2));
plot(rgates*1e-3,th_BF);
plot(rgates(detected_BF)*1e-3,abs(sigp_BF(detected_BF)).^2,'o','markersize',10)
legend('Signal','Threshold','Detections');
xlabel('range (km)');
```
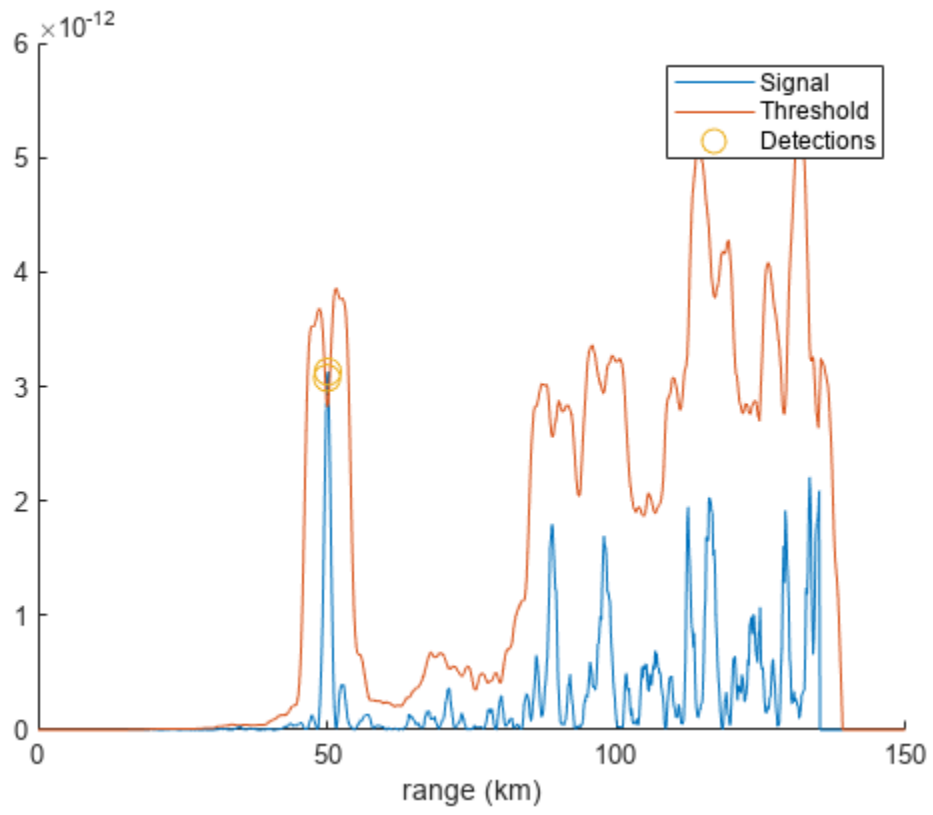
# Neural Network for Beam Selection

This example shows how to use a neural network to reduce the overhead in the beam selection task. In the example, you use only the location of the receiver rather than knowledge of the communication channels. Instead of an exhaustive beam search over all the beam pairs, you can reduce beam sweeping overhead by searching among the selected $K$ beam pairs. Considering a system with a total of 16 beam pairs, simulation results in this example show the designed machine learning algorithm can achieve an accuracy of 90% by performing an exhaustive search over only half of the beam pairs.

**Introduction**

To enable millimeter wave (mmWave) communications, beam management techniques must be used due to the high pathloss and blockage experienced at high frequencies. Beam management is a set of Layer 1 (physical layer) and Layer 2 (medium access control) procedures to establish and retain an optimal beam pair (transmit beam and a corresponding receive beam) for good connectivity [1 on page 18-721]. For simulations of 5G New Radio (NR) beam management procedures, see the "NR SSB Beam Sweeping" (5G Toolbox) and "NR Downlink Transmit-End Beam Refinement Using CSI-RS" (5G Toolbox) examples.

This example considers beam selection procedures when a connection is established between the user equipment (UE) and access network node (gNB). In 5G NR, the beam selection procedure for initial access consists of beam sweeping, which requires exhaustive searches over all the beams on the transmitter and the receiver sides, and then selection of the beam pair offering the strongest reference signal received power (RSRP). Since mmWave communications require many antenna elements, implying many beams, an exhaustive search over all beams becomes computationally expensive and increases the initial access time.

To avoid repeatedly performing an exhaustive search and to reduce the communication overhead, machine learning has been applied to the beam selection problem. Typically, the beam selection problem is posed as a classification task, where the target output is the best beam pair index. The extrinsic information, including lidar, GPS signals, and roadside camera images, is used as input to the machine learning algorithms [2 on page 18-721]-[6 on page 18-721]. Specifically, given this out-of-band information, a trained machine learning model recommends a set of $K$ good beam pairs. Instead of an exhaustive search over all the beam pairs, the simulation reduces beam sweeping overhead by searching only among the selected $K$ beam pairs.

This example uses a neural network to perform beam selection using only the GPS coordinates of the receiver. Fixing the locations of the transmitter and the scatterers, the example generates a set of training samples: Each sample consists of a receiver location (GPS data) and the true optimal beam pair index (found by performing exhaustive search over all the beam pairs at transmit and receive ends). The example designs and trains a neural network that uses the location of the receiver as the input and the true optimal beam pair index as the correct label. During the testing phase, the neural network first outputs $K$ good beam pairs. An exhaustive search over these $K$ beam pairs is followed, and the beam pair with the highest average RSRP is selected as the final predicted beam pair by the neural network.

The example measures the effectiveness of the proposed method using two metrics: average RSRP and top-$K$ accuracy [2 on page 18-721]-[6 on page 18-721]. This figure shows the main processing steps.
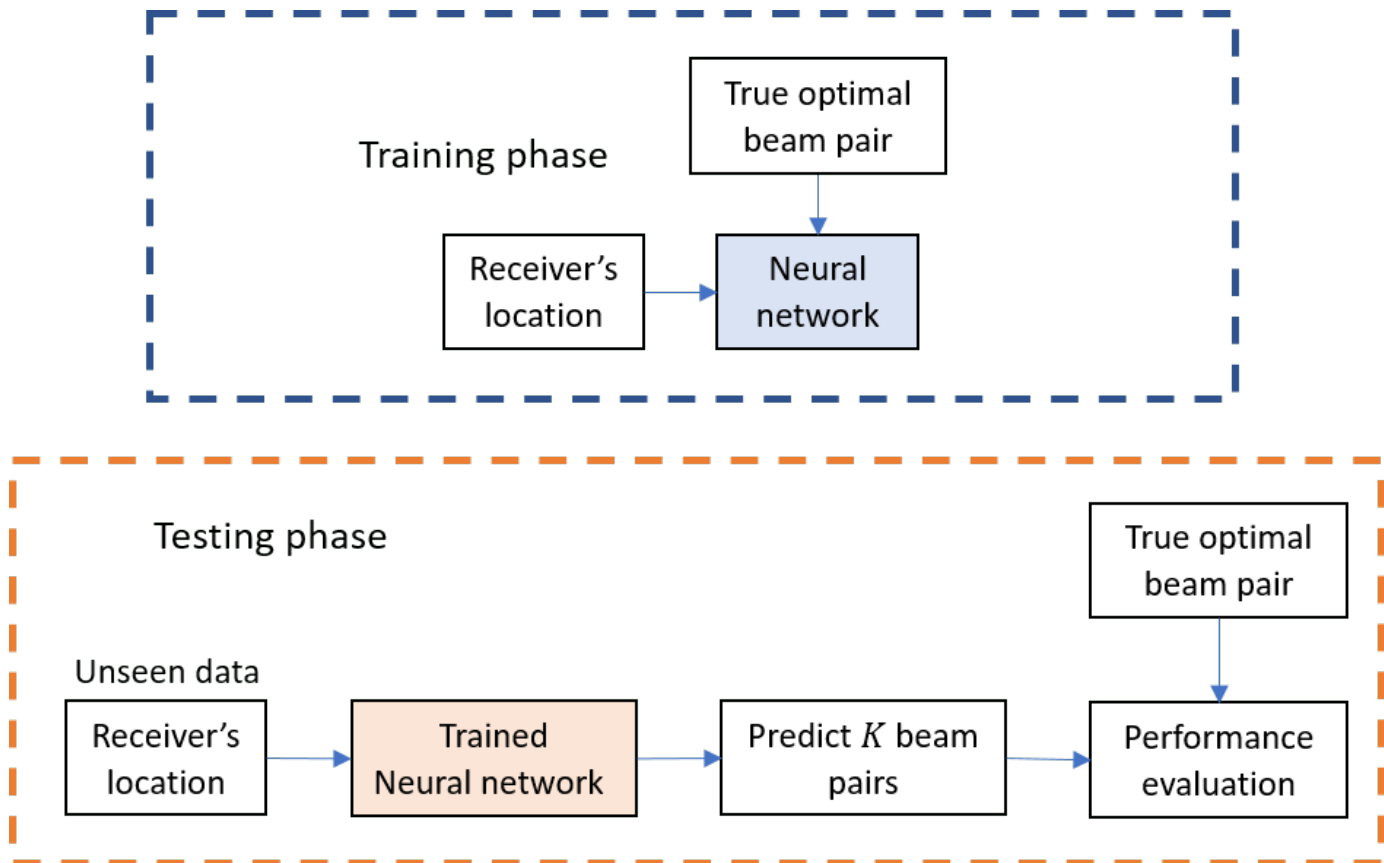
```
rng(211);                              % Set RNG state for repeatability
```

**Generate Training Data**

In the prerecorded data, receivers are randomly distributed on the perimeter of a 6-meter square and configured with 16 beam pairs (four beams on each end, analog beamformed with 1 RF chain). After setting up a MIMO scattering channel, the example considers 200 different receiver locations in the training set and 100 different receiver locations in the test sets. The prerecorded data uses 2-D location coordinates. Specifically, the third GPS coordinate of each sample is always zero. As in the `NR SSB Beam Sweeping` example, for each location, SSB-based beam sweeping is performed for an exhaustive search over all 16 beam pairs. Since AWGN is added during the exhaustive search, for each location, the example runs four different trials and determines the true optimal beam pair by picking the beam pair with the highest average RSRP.

To generate new training and test sets, you can adjust the `useSavedData` and `SaveData` logicals. Be aware that regenerating data takes a significant amount of time.

```
useSavedData = true;
saveData = false;

if useSavedData
    load nnBS_prm.mat;            % Load beam selection system parameters
    load nnBS_TrainingData.mat;   % Load prerecorded training samples
    %   (input: receiver's location; output: optimal beam pair indices)
    load nnBS_TestData.mat;       % Load prerecorded test samples
else
```

**Configure Frequency and Beam Sweeping Angles**

```
prm.NCellID = 1;                    % Cell ID
prm.FreqRange = 'FR1';              % Frequency range: 'FR1' or 'FR2'

prm.CenterFreq = 2.5e9;             % Hz
prm.SSBlockPattern = 'Case B';      % Case A/B/C/D/E
prm.SSBTransmitted = [ones(1,4) zeros(1,0)]; % 4/8 or 64 in length

prm.TxArraySize = [8 8];            % Transmit array size, [rows cols]
prm.TxAZlim = [-163 177];           % Transmit azimuthal sweep limits
prm.TxELlim = [-90 0];              % Transmit elevation sweep limits

prm.RxArraySize = [2 2];            % Receive array size, [rows cols]
prm.RxAZlim = [-177 157];           % Receive azimuthal sweep limits
prm.RxELlim = [0 90];               % Receive elevation sweep limits

prm.ElevationSweep = false;         % Enable/disable elevation sweep
prm.SNRdB = 30;                     % SNR, dB
prm.RSRPMode = 'SSSwDMRS';          % {'SSSwDMRS', 'SSSonly'}

prm = validateParams(prm);
```

**Synchronization Signal Burst Configuration**

```
txBurst = nrWavegenSSBurstConfig;
txBurst.BlockPattern = prm.SSBlockPattern;
txBurst.TransmittedBlocks = prm.SSBTransmitted;
txBurst.Period = 20;
txBurst.SubcarrierSpacingCommon = prm.SubcarrierSpacingCommon;
```

**Scatterer Configuration**

```
c = physconst('LightSpeed');    % Propagation speed
prm.lambda = c/prm.CenterFreq; % Wavelength

prm.rightCoorMax = 10;     % Maximum x-coordinate
prm.topCoorMax = 10;       % Maximum y-coordinate
prm.posTx = [3.5;4.2;0];   % Transmit array position, [x;y;z], meters

% Scatterer locations
% Generate scatterers at random positions
Nscat = 10;         % Number of scatterers
azRange = prm.TxAZlim(1):prm.TxAZlim(2);
elRange = -90:90;

% More evenly spaced scatterers
randAzOrder = round(linspace(1, length(azRange), Nscat));
azAngInSph = azRange(randAzOrder(1:Nscat));

% Consider a 2-D area, i.e., the elevation angle is zero
elAngInSph = zeros(size(azAngInSph));
r = 2;              % radius
[x,y,z] = sph2cart(deg2rad(azAngInSph),deg2rad(elAngInSph),r);
prm.ScatPos = [x;y;z] + [prm.rightCoorMax/2;prm.topCoorMax/2;0];
```

**Antenna Array Configuration**

```matlab
% Transmit array
if prm.IsTxURA
    % Uniform rectangular array
    arrayTx = phased.URA(prm.TxArraySize,0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
else
    % Uniform linear array
    arrayTx = phased.ULA(prm.NumTx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement('BackBaffled',true));
end

% Receive array
if prm.IsRxURA
    % Uniform rectangular array
    arrayRx = phased.URA(prm.RxArraySize,0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement);
else
    % Uniform linear array
    arrayRx = phased.ULA(prm.NumRx, ...
        'ElementSpacing',0.5*prm.lambda, ...
        'Element',phased.IsotropicAntennaElement);
end
```

**Determine Tx/Rx Positions**

```matlab
% Receiver locations
% Training data: X points around a rectangle: each side has X/4 random points
% X: X/4 for around square, X/10 for validation => lcm(4,10) = 20 smallest
NDiffLocTrain = 200;
pointsEachSideTrain = NDiffLocTrain/4;
prm.NDiffLocTrain = NDiffLocTrain;

locationX = 2*ones(pointsEachSideTrain, 1);
locationY = 2 + (8-2)*rand(pointsEachSideTrain, 1);

locationX = [locationX; 2 + (8-2)*rand(pointsEachSideTrain, 1)];
locationY = [locationY; 8*ones(pointsEachSideTrain, 1)];

locationX = [locationX; 8*ones(pointsEachSideTrain, 1)];
locationY = [locationY; 2 + (8-2)*rand(pointsEachSideTrain, 1)];

locationX = [locationX; 2 + (8-2)*rand(pointsEachSideTrain, 1)];
locationY = [locationY; 2*ones(pointsEachSideTrain, 1)];

locationZ = zeros(size(locationX));
locationMat = [locationX locationY locationZ];

% Fixing receiver's location, run repeated simulations to consider
% different realizations of AWGN
prm.NRepeatSameLoc = 4;

locationMatTrain = repelem(locationMat,prm.NRepeatSameLoc, 1);

% Test data: Y points around a rectangle: each side has Y/4 random points
% Different data than test, but a smaller number
```

```
        NDiffLocTest = 100;
        pointsEachSideTest = NDiffLocTest/4;
        prm.NDiffLocTest = NDiffLocTest;

        locationX = 2*ones(pointsEachSideTest, 1);
        locationY = 2 + (8-2)*rand(pointsEachSideTest, 1);

        locationX = [locationX; 2 + (8-2)*rand(pointsEachSideTest, 1)];
        locationY = [locationY; 8*ones(pointsEachSideTest, 1)];

        locationX = [locationX; 8*ones(pointsEachSideTest, 1)];
        locationY = [locationY; 2 + (8-2)*rand(pointsEachSideTest, 1)];

        locationX = [locationX; 2 + (8-2)*rand(pointsEachSideTest, 1)];
        locationY = [locationY; 2*ones(pointsEachSideTest, 1)];

        locationZ = zeros(size(locationX));
        locationMat = [locationX locationY locationZ];

        locationMatTest = repelem(locationMat,prm.NRepeatSameLoc,1);

        [optBeamPairIdxMatTrain,rsrpMatTrain] = hGenDataMIMOScatterChan('training',locationMatTrain,
        [optBeamPairIdxMatTest,rsrpMatTest] = hGenDataMIMOScatterChan('test',locationMatTest,prm,txBu

        % Save generated data
        if saveData
            save('nnBS_prm.mat','prm');
            save('nnBS_TrainingData.mat','optBeamPairIdxMatTrain','rsrpMatTrain','locationMatTrain')
            save('nnBS_TestData.mat','optBeamPairIdxMatTest','rsrpMatTest','locationMatTest');
        end
end
```

**Plot Transmitter and Scatterer Locations**

```
figure
scatter(prm.posTx(1),prm.posTx(2),100,'r^','filled');
hold on;
scatter(prm.ScatPos(1,:),prm.ScatPos(2,:),100,[0.9290 0.6940 0.1250],'s','filled');
xlim([0 10])
ylim([0 10])
title('Transmitter and Scatterers Positions')
legend('Transmitter','Scatterers')
xlabel('x (m)')
ylabel('y (m)')
```

**Transmitter and Scatterers Positions**
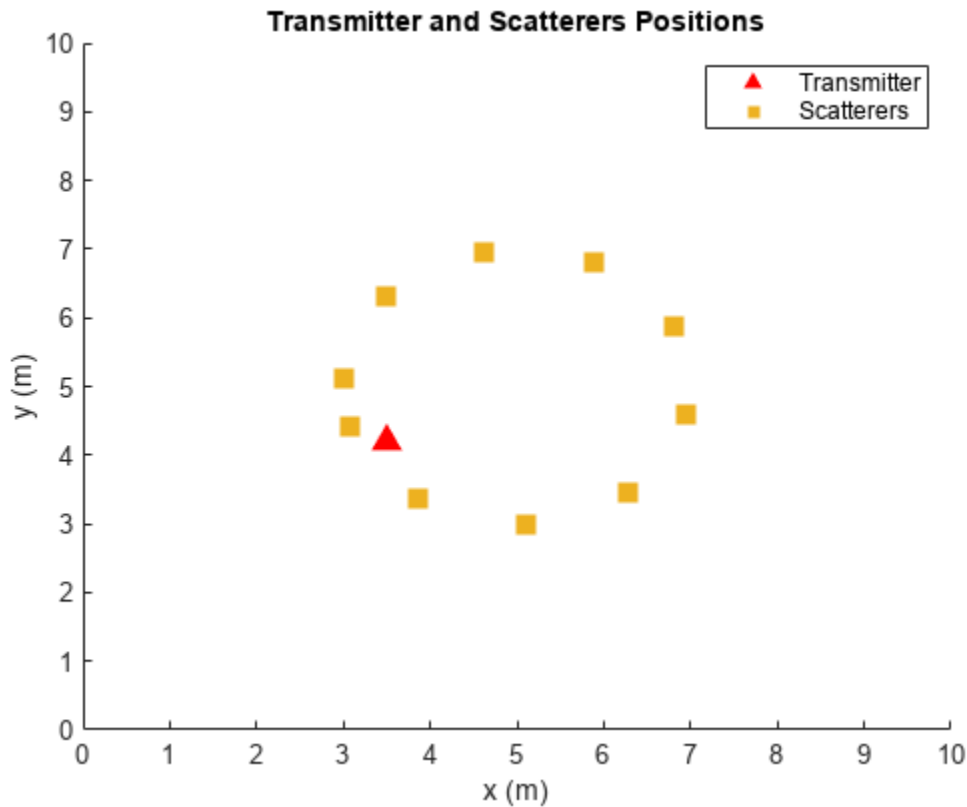
### Data Processing and Visualization

Next, label the beam pair with the highest average RSRP as the true optimal beam pair. Convert one-hot encoding labels to categorical data to use for classification. Finally, augment the categorical data so that it has 16 classes total to match the possible number of beam pairs (although classes may have unequal number of elements). The augmentation is to ensure that the output of the neural network has the desired dimension 16.

### Process Training Data

```
% Choose the best beam pair by picking the one with the highest average RSRP
% (taking average over NRepeatSameLoc different trials at each location)
avgOptBeamPairIdxCellTrain = cell(size(optBeamPairIdxMatTrain, 1)/prm.NRepeatSameLoc, 1);
avgOptBeamPairIdxScalarTrain = zeros(size(optBeamPairIdxMatTrain, 1)/prm.NRepeatSameLoc, 1);
for locIdx = 1:size(optBeamPairIdxMatTrain, 1)/prm.NRepeatSameLoc
    avgRsrp = squeeze(rsrpMatTrain(:,:,locIdx));
    [~, targetBeamIdx] = max(avgRsrp(:));
    avgOptBeamPairIdxScalarTrain(locIdx) = targetBeamIdx;
    avgOptBeamPairIdxCellTrain{locIdx} = num2str(targetBeamIdx);
end

% Even though there are a total of 16 beam pairs, due to the fixed topology
% (transmitter/scatterers/receiver locations), it is possible
% that some beam pairs are never selected as an optimal beam pair
%
% Therefore, we augment the categories so 16 classes total are in the data
% (although some classes may have zero elements)
```

```
allBeamPairIdxCell = cellstr(string((1:prm.numBeams^2)'));
avgOptBeamPairIdxCellTrain = categorical(avgOptBeamPairIdxCellTrain, allBeamPairIdxCell);
NBeamPairInTrainData = numel(categories(avgOptBeamPairIdxCellTrain)); % Should be 16
```

**Process Testing Data**

```
% Decide the best beam pair by picking the one with the highest avg. RSRP
avgOptBeamPairIdxCellTest = cell(size(optBeamPairIdxMatTest, 1)/prm.NRepeatSameLoc, 1);
avgOptBeamPairIdxScalarTest = zeros(size(optBeamPairIdxMatTest, 1)/prm.NRepeatSameLoc, 1);
for locIdx = 1:size(optBeamPairIdxMatTest, 1)/prm.NRepeatSameLoc
    avgRsrp = squeeze(rsrpMatTest(:,:,locIdx));
    [~, targetBeamIdx] = max(avgRsrp(:));
    avgOptBeamPairIdxScalarTest(locIdx) = targetBeamIdx;
    avgOptBeamPairIdxCellTest{locIdx} = num2str(targetBeamIdx);
end
% Augment the categories such that the data has 16 classes total
avgOptBeamPairIdxCellTest = categorical(avgOptBeamPairIdxCellTest, allBeamPairIdxCell);
NBeamPairInTestData = numel(categories(avgOptBeamPairIdxCellTest)); % Should be 16
```

**Create Input/Output Data for Neural Network**

```
trainDataLen = size(locationMatTrain, 1)/prm.NRepeatSameLoc;
trainOut = avgOptBeamPairIdxCellTrain;
sampledLocMatTrain = locationMatTrain(1:prm.NRepeatSameLoc:end, :);
trainInput = sampledLocMatTrain(1:trainDataLen, :);

% Take 10% data out of test data as validation data
valTestDataLen = size(locationMatTest, 1)/prm.NRepeatSameLoc;
valDataLen = round(0.1*size(locationMatTest, 1))/prm.NRepeatSameLoc;
testDataLen = valTestDataLen-valDataLen;

% Randomly shuffle the test data such that the distribution of the
% extracted validation data is closer to test data
rng(111)
shuffledIdx = randperm(prm.NDiffLocTest);
avgOptBeamPairIdxCellTest = avgOptBeamPairIdxCellTest(shuffledIdx);
avgOptBeamPairIdxScalarTest = avgOptBeamPairIdxScalarTest(shuffledIdx);
rsrpMatTest = rsrpMatTest(:,:,shuffledIdx);

valOut = avgOptBeamPairIdxCellTest(1:valDataLen, :);
testOutCat = avgOptBeamPairIdxCellTest(1+valDataLen:end, :);

sampledLocMatTest = locationMatTest(1:prm.NRepeatSameLoc:end, :);
sampledLocMatTest = sampledLocMatTest(shuffledIdx, :);

valInput = sampledLocMatTest(1:valDataLen, :);
testInput = sampledLocMatTest(valDataLen+1:end, :);
```

**Plot Optimal Beam Pair Distribution for Training Data**

Plot the location and the optimal beam pair for each training sample (200 in total). Each color represents one beam pair index. In other words, the data points with the same color belong to the same class. Increase the training data set to possibly include each beam pair value, though the actual distribution of the beam pairs would depend on the scatterer and transmitter locations.

```
figure
rng(111)    % for colors in plot
color = rand(NBeamPairInTrainData, 3);
```

```
uniqueOptBeamPairIdx = unique(avgOptBeamPairIdxScalarTrain);
for n = 1:length(uniqueOptBeamPairIdx)
    beamPairIdx = find(avgOptBeamPairIdxScalarTrain == uniqueOptBeamPairIdx(n));
    locX = sampledLocMatTrain(beamPairIdx, 1);
    locY = sampledLocMatTrain(beamPairIdx, 2);
    scatter(locX, locY, [], color(n, :));
    hold on;
end
scatter(prm.posTx(1),prm.posTx(2),100,'r^','filled');
scatter(prm.ScatPos(1,:),prm.ScatPos(2,:),100,[0.9290 0.6940 0.1250],'s','filled');
hold off
xlabel('x (m)')
ylabel('y (m)')
xlim([0 10])
ylim([0 10])
title('Optimal Beam Pair Indices (Training Data)')
```
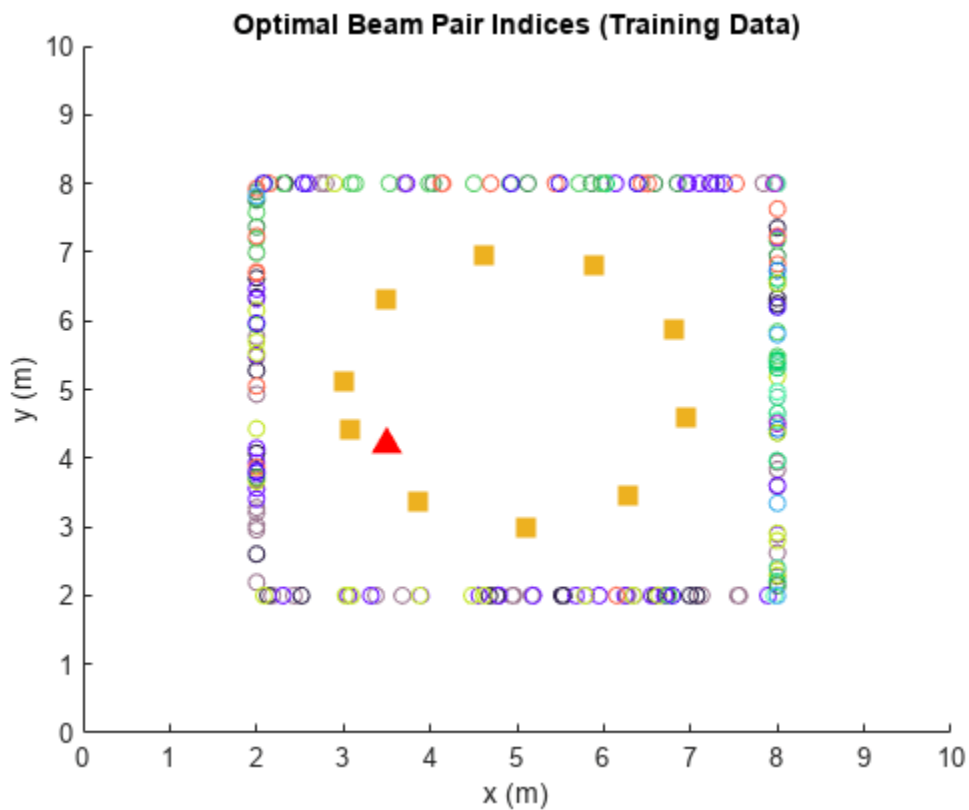


```
figure
histogram(trainOut)
title('Histogram of Optimal Beam Pair Indices (Training Data)')
xlabel('Beam Pair Index')
ylabel('Number of Occurrences')
```
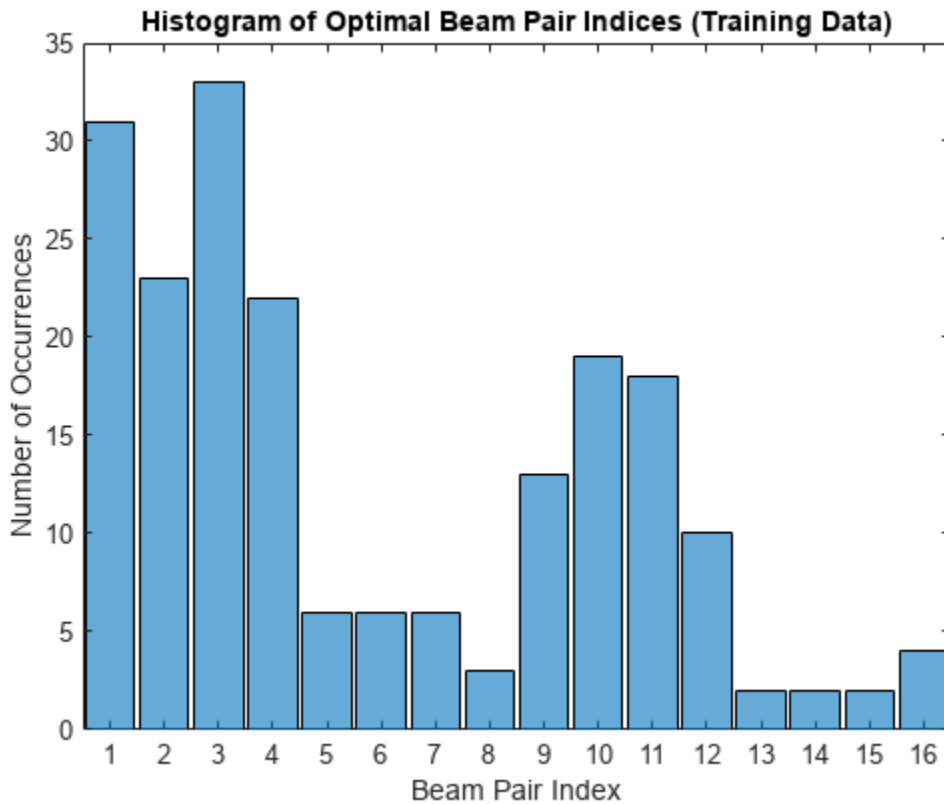
**Plot Optimal Beam Pair Distribution for Validation Data**

```
figure
rng(111)     % for colors in plot
color = rand(NBeamPairInTestData, 3);
uniqueOptBeamPairIdx = unique(avgOptBeamPairIdxScalarTest(1:valDataLen));
for n = 1:length(uniqueOptBeamPairIdx)
    beamPairIdx = find(avgOptBeamPairIdxScalarTest(1:valDataLen) == uniqueOptBeamPairIdx(n));
    locX = sampledLocMatTest(beamPairIdx, 1);
    locY = sampledLocMatTest(beamPairIdx, 2);
    scatter(locX, locY, [], color(n, :));
    hold on;
end
scatter(prm.posTx(1),prm.posTx(2),100,'r^','filled');
scatter(prm.ScatPos(1,:),prm.ScatPos(2,:),100,[0.9290 0.6940 0.1250],'s','filled');
hold off
xlabel('x (m)')
ylabel('y (m)')
xlim([0 10])
ylim([0 10])
title('Optimal Beam Pair Indices (Validation Data)')
```

**Optimal Beam Pair Indices (Validation Data)**
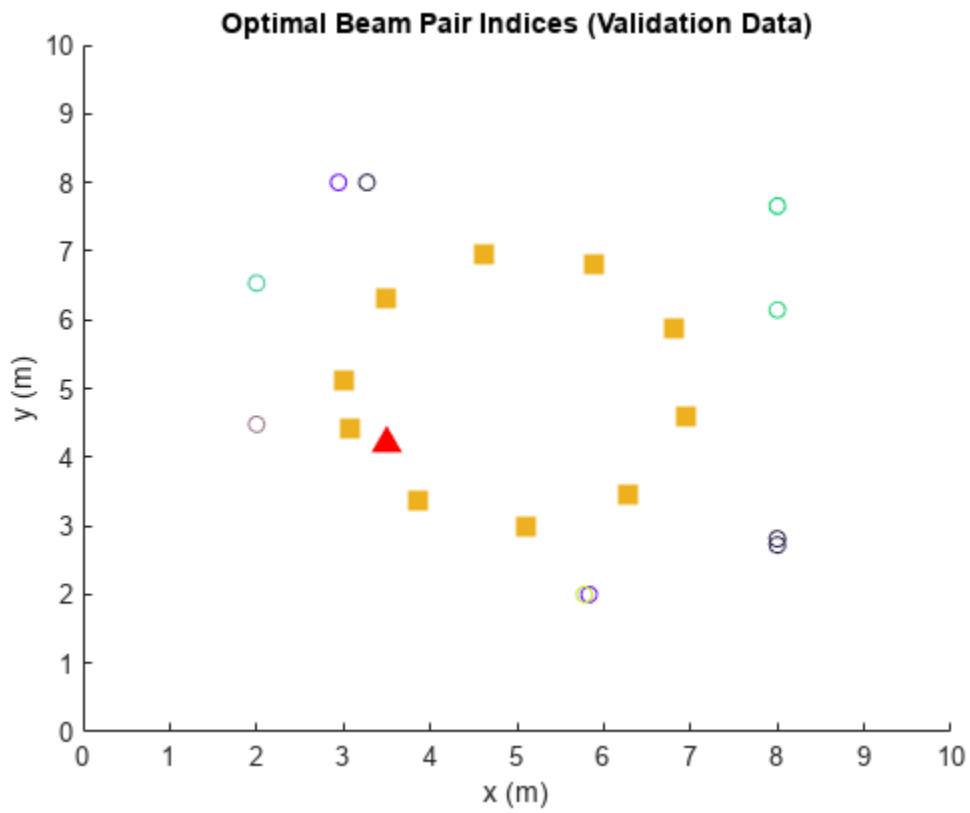
```
figure
histogram(valOut)
title('Histogram of Optimal Beam Pair Indices (Validation Data)')
xlabel('Beam Pair Index')
ylabel('Number of Occurrences')
```
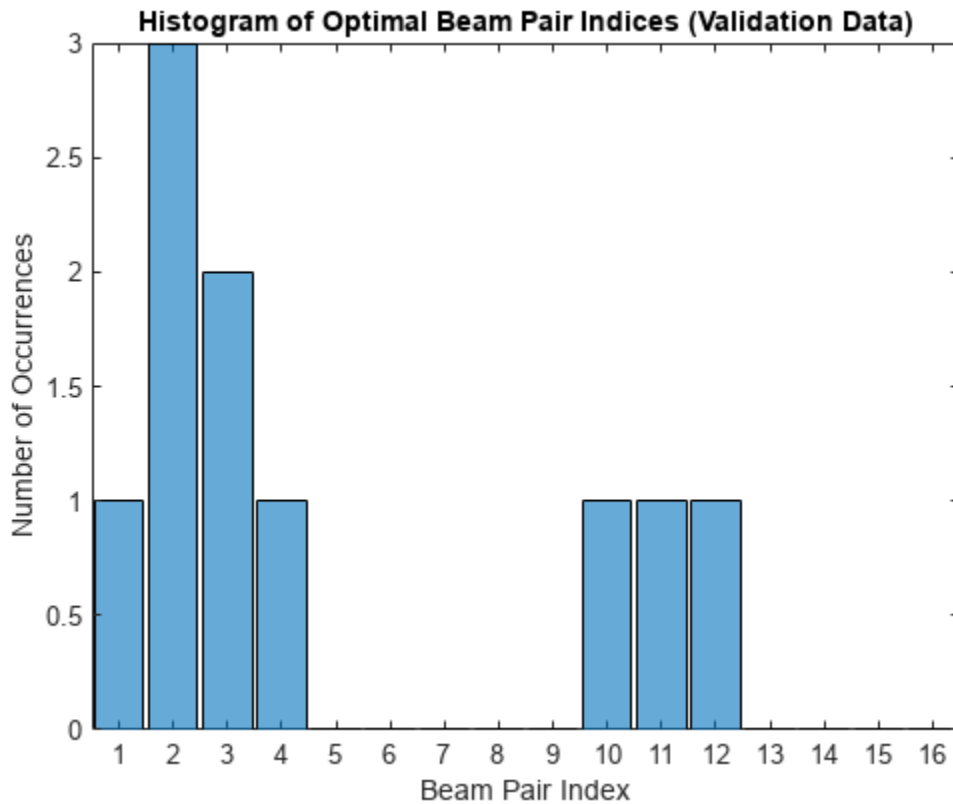
**Plot Optimal Beam Pair Distribution for Test Data**

```
figure
rng(111)     % for colors in plots
color = rand(NBeamPairInTestData, 3);
uniqueOptBeamPairIdx = unique(avgOptBeamPairIdxScalarTest(1+valDataLen:end));
for n = 1:length(uniqueOptBeamPairIdx)
    beamPairIdx = find(avgOptBeamPairIdxScalarTest(1+valDataLen:end) == uniqueOptBeamPairIdx(n))
    locX = sampledLocMatTest(beamPairIdx, 1);
    locY = sampledLocMatTest(beamPairIdx, 2);
    scatter(locX, locY, [], color(n, :));
    hold on;
end
scatter(prm.posTx(1),prm.posTx(2),100,'r^','filled');
scatter(prm.ScatPos(1,:),prm.ScatPos(2,:),100,[0.9290 0.6940 0.1250],'s','filled');
hold off
xlabel('x (m)')
ylabel('y (m)')
xlim([0 10])
ylim([0 10])
title('Optimal Beam Pair Indices (Test Data)')
```

Optimal Beam Pair Indices (Test Data)
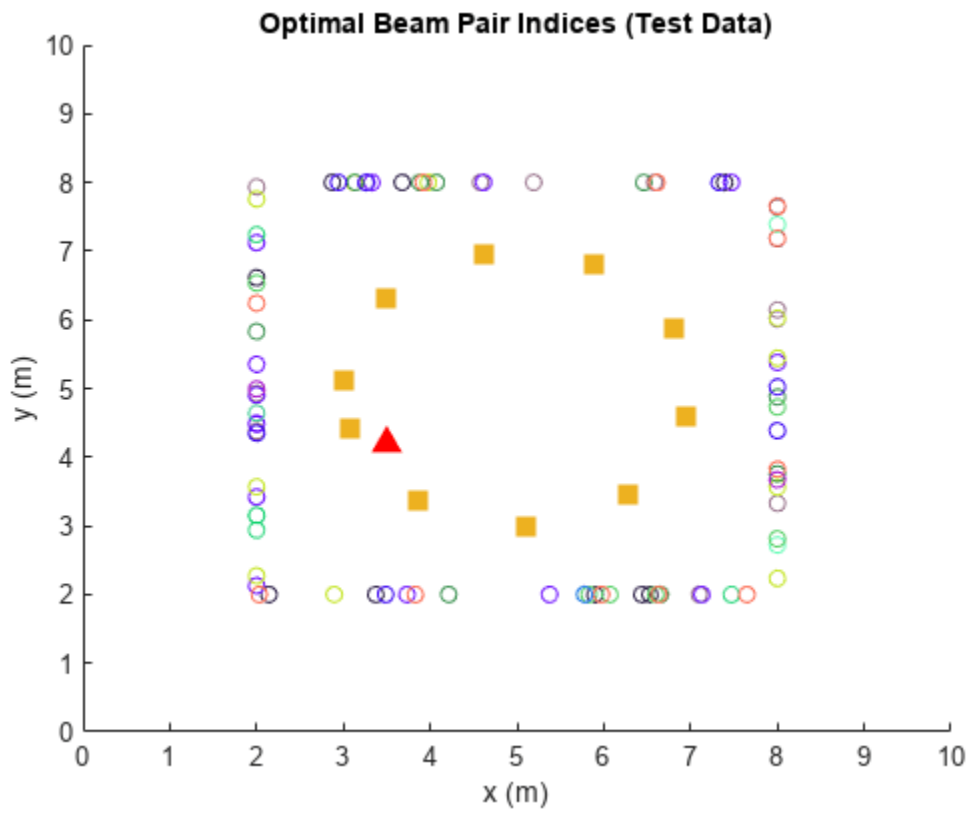
```
figure
histogram(testOutCat)
title('Histogram of Optimal Beam Pair Indices (Test Data)')
xlabel('Beam Pair Index')
ylabel('Number of Occurrences')
```
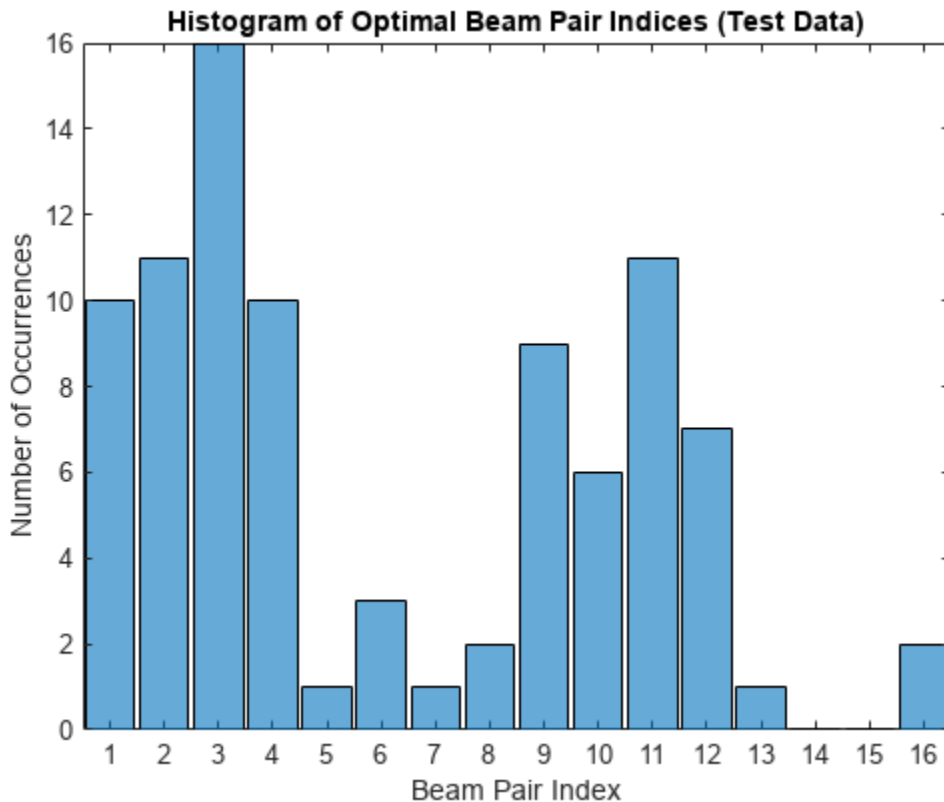
**Design and Train Neural Network**

Train a neural network with four hidden layers. The design is motivated by [3 on page 18-721] (four hidden layers) and [5 on page 18-721] (two hidden layers with 128 neurons in each layer) in which the receiver locations are also considered as the input to the neural network. To enable training, adjust the `doTraining` logical.

This example also provides an option to weight the classes. Classes that occur more frequently have smaller weights and classes that occur less frequently have larger weights. To use class weighting, adjust the `useDiffClassWeights` logical.

Modify the network to experiment with different designs. If you modify one of the provided data sets, you must retrain the network with the modified data sets. Retraining the network can take a significant amount of time. Adjust the `saveNet` logical to use the trained network in subsequent runs.

```
doTraining = false;
useDiffClassWeights = false;
saveNet = false;

if doTraining
    if useDiffClassWeights
        catCount = countcats(trainOut);
        catFreq = catCount/length(trainOut);
        nnzIdx = (catFreq ~= 0);
        medianCount = median(catFreq(nnzIdx));
        classWeights = 10*ones(size(catFreq));
        classWeights(nnzIdx) = medianCount./catFreq(nnzIdx);
```

```matlab
            filename = 'nnBS_trainedNetwWeighting.mat';
        else
            classWeights = ones(1,NBeamPairInTestData);
            filename = 'nnBS_trainedNet.mat';
        end

        % Neural network design
        layers = [ ...
            featureInputLayer(3,'Name','input','Normalization','rescale-zero-one')

            fullyConnectedLayer(96,'Name','linear1')
            leakyReluLayer(0.01,'Name','leakyRelu1')

            fullyConnectedLayer(96,'Name','linear2')
            leakyReluLayer(0.01,'Name','leakyRelu2')

            fullyConnectedLayer(96,'Name','linear3')
            leakyReluLayer(0.01,'Name','leakyRelu3')

            fullyConnectedLayer(96,'Name','linear4')
            leakyReluLayer(0.01,'Name','leakyRelu4')

            fullyConnectedLayer(NBeamPairInTrainData,'Name','linear5')
            softmaxLayer('Name','softmax')
            classificationLayer('ClassWeights',classWeights,'Classes',allBeamPairIdxCell,'Name','outp

        maxEpochs = 1000;
        miniBatchSize = 256;

        options = trainingOptions('adam', ...
            'MaxEpochs',maxEpochs, ...
            'MiniBatchSize',miniBatchSize, ...
            'InitialLearnRate',1e-4, ...
            'ValidationData',{valInput,valOut}, ...
            'ValidationFrequency',500, ...
            'OutputNetwork', 'best-validation-loss', ...
            'Shuffle','every-epoch', ...
            'Plots','training-progress', ...
            'ExecutionEnvironment','cpu', ...
            'Verbose',0);

        % Train the network
        net = trainNetwork(trainInput,trainOut,layers,options);

        if saveNet
            save(filename,'net');
        end
    else
        if useDiffClassWeights
            load 'nnBS_trainedNetwWeighting.mat';
        else
            load 'nnBS_trainedNet.mat';
        end
    end
end
```

**Compare Different Approaches: Top-*K* Accuracy**

This section tests the trained network with unseen test data considering the top-K accuracy metric. The top-K accuracy metric has been widely used in the neural network-based beam selection task [2 on page 18-721]-[6 on page 18-721].

Given a receiver location, the neural network first outputs *K* recommended beam pairs. Then it performs an exhaustive sequential search on these *K* beam pairs and selects the one with the highest average RSRP as the final prediction. If the true optimal beam pair is the final selected beam pair, then a successful prediction occurs. Equivalently, a success occurs when the true optimal beam pair is one of the *K* recommended beam pairs by the neural network.

Three benchmarks are compared. Each scheme produces the *K* recommended beam pairs.

1  KNN - For a test sample, this method first collects *K* closest training samples based on GPS coordinates. The method then recommends all the beam pairs associated with these *K* training samples. Since each training sample has a corresponding optimal beam pair, the number of beam pairs recommended is at most *K*(some beam pairs might be the same).

2  Statistical Info [5 on page 18-721] - This method first ranks all the beam pairs according to their relative frequency in the training set, and then always selects the first *K* beam pairs.

3  Random [5 on page 18-721] - For a test sample, this method randomly chooses *K* beam pairs.

The plot shows that for *K* = 8, the accuracy is already more than 90%, which highlights the effectiveness of using the trained neural network for the beam selection task. When *K* = 16, every scheme (except KNN) is relaxed to the exhaustive search over all the 16 beam pairs, and hence achieves an accuracy of 100%. However, when *K* = 16, KNN considers 16 closest training samples, and the number of *distinct* beam pairs from these samples is often less than 16. Hence, KNN does not achieve an accuracy of 100%.

```
rng(111)     % for repeatability of the "Random" policy
testOut = avgOptBeamPairIdxScalarTest(1+valDataLen:end, :);
statisticCount = countcats(testOutCat);
predTestOutput = predict(net,testInput,'ExecutionEnvironment','cpu');

K = prm.numBeams^2;
accNeural = zeros(1,K);
accKNN = zeros(1,K);
accStatistic = zeros(1,K);
accRandom = zeros(1,K);
for k = 1:K
    predCorrectNeural = zeros(testDataLen,1);
    predCorrectKNN = zeros(testDataLen,1);
    predCorrectStats = zeros(testDataLen,1);
    predCorrectRandom = zeros(testDataLen,1);
    knnIdx = knnsearch(trainInput,testInput,'K',k);

    for n = 1:testDataLen
        trueOptBeamIdx = testOut(n);

        % Neural Network
        [~, topKPredOptBeamIdx] = maxk(predTestOutput(n, :),k);
        if sum(topKPredOptBeamIdx == trueOptBeamIdx) > 0
            % if true, then the true correct index belongs to one of the K predicted indices
            predCorrectNeural(n,1) = 1;
        end
```

```matlab
        % KNN
        neighborsIdxInTrainData = knnIdx(n,:);
        topKPredOptBeamIdx= avgOptBeamPairIdxScalarTrain(neighborsIdxInTrainData);
        if sum(topKPredOptBeamIdx == trueOptBeamIdx) > 0
            % if true, then the true correct index belongs to one of the K predicted indices
            predCorrectKNN(n,1) = 1;
        end

        % Statistical Info
        [~, topKPredOptBeamIdx] = maxk(statisticCount,k);
        if sum(topKPredOptBeamIdx == trueOptBeamIdx) > 0
            % if true, then the true correct index belongs to one of the K predicted indices
            predCorrectStats(n,1) = 1;
        end

        % Random
        topKPredOptBeamIdx = randperm(prm.numBeams*prm.numBeams,k);
        if sum(topKPredOptBeamIdx == trueOptBeamIdx) > 0
            % if true, then the true correct index belongs to one of the K predicted indices
            predCorrectRandom(n,1) = 1;
        end

    end

    accNeural(k)    = sum(predCorrectNeural)/testDataLen*100;
    accKNN(k)       = sum(predCorrectKNN)/testDataLen*100;
    accStatistic(k) = sum(predCorrectStats)/testDataLen*100;
    accRandom(k)    = sum(predCorrectRandom)/testDataLen*100;

end

figure
lineWidth = 1.5;
colorNeural = [0 0.4470 0.7410];
colorKNN = [0.8500 0.3250 0.0980];
colorStats = [0.4940 0.1840 0.5560];
colorRandom = [0.4660 0.6740 0.1880];
plot(1:K,accNeural,'--*','LineWidth',lineWidth,'Color',colorNeural)
hold on
plot(1:K,accKNN,'--o','LineWidth',lineWidth,'Color',colorKNN)
plot(1:K,accStatistic,'--s','LineWidth',lineWidth,'Color',colorStats)
plot(1:K,accRandom,'--d','LineWidth',lineWidth,'Color',colorRandom)
hold off
grid on
xticks(1:K)
xlabel('$K$','interpreter','latex')
ylabel('Top-$K$ Accuracy','interpreter','latex')
title('Performance Comparison of Different Beam Pair Selection Schemes')
legend('Neural Network','KNN','Statistical Info','Random','Location','best')
```

Performance Comparison of Different Beam Pair Selection Schemes
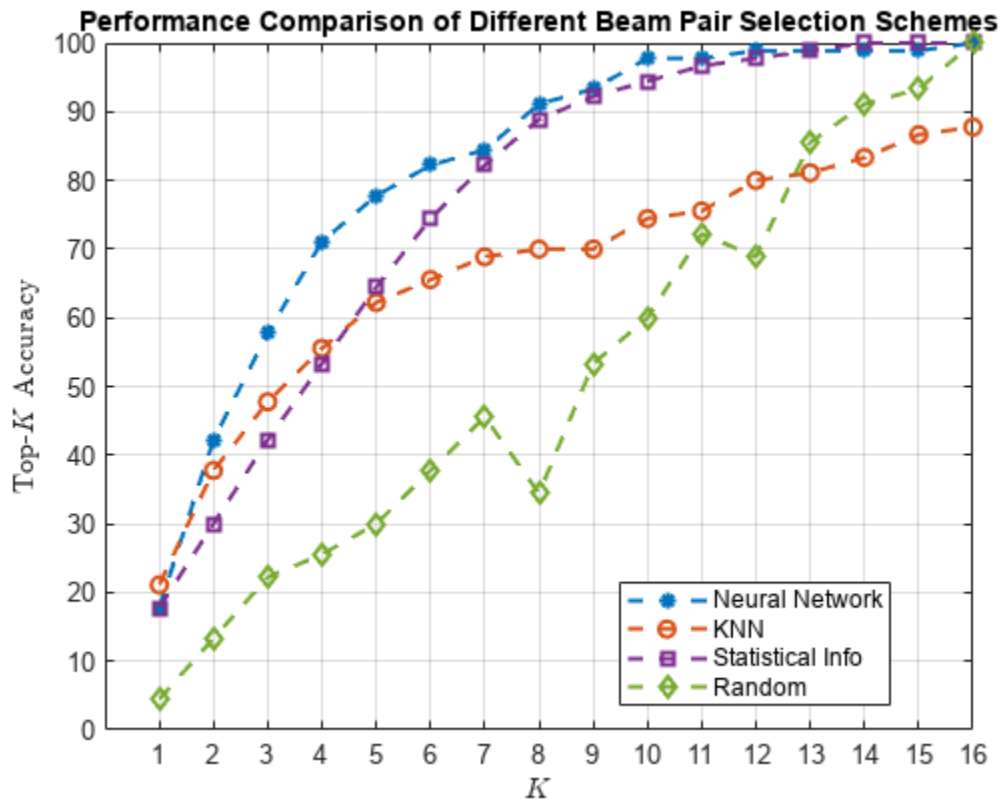


**Compare Different Approaches: Average RSRP**

Using unseen test data, compute the average RSRP achieved by the neural network and the three benchmarks. The plot shows that using the trained neural network results in an average RSRP close to the optimal exhaustive search.

```
rng(111)     % for repeatability of the "Random" policy
K = prm.numBeams^2;
rsrpOptimal = zeros(1,K);
rsrpNeural = zeros(1,K);
rsrpKNN = zeros(1,K);
rsrpStatistic = zeros(1,K);
rsrpRandom = zeros(1,K);
for k = 1:K
    rsrpSumOpt = 0;
    rsrpSumNeural = 0;
    rsrpSumKNN = 0;
    rsrpSumStatistic = 0;
    rsrpSumRandom = 0;

    knnIdx = knnsearch(trainInput,testInput,'K',k);

    for n = 1:testDataLen
        % Exhaustive Search
        trueOptBeamIdx = testOut(n);
        rsrp = rsrpMatTest(:,:,valDataLen+n);
        rsrpSumOpt = rsrpSumOpt + rsrp(trueOptBeamIdx);
```

```matlab
        % Neural Network
        [~, topKPredOptCatIdx] = maxk(predTestOutput(n, :),k);
        rsrpSumNeural = rsrpSumNeural + max(rsrp(topKPredOptCatIdx));

        % KNN
        neighborsIdxInTrainData = knnIdx(n,:);
        topKPredOptBeamIdxKNN = avgOptBeamPairIdxScalarTrain(neighborsIdxInTrainData);
        rsrpSumKNN = rsrpSumKNN + max(rsrp(topKPredOptBeamIdxKNN));

        % Statistical Info
        [~, topKPredOptCatIdxStat] = maxk(statisticCount,k);
        rsrpSumStatistic = rsrpSumStatistic + max(rsrp(topKPredOptCatIdxStat));

        % Random
        topKPredOptBeamIdxRand = randperm(prm.numBeams*prm.numBeams,k);
        rsrpSumRandom = rsrpSumRandom + max(rsrp(topKPredOptBeamIdxRand));
    end
    rsrpOptimal(k)   = rsrpSumOpt/testDataLen/prm.NRepeatSameLoc;
    rsrpNeural(k)    = rsrpSumNeural/testDataLen/prm.NRepeatSameLoc;
    rsrpKNN(k)       = rsrpSumKNN/testDataLen/prm.NRepeatSameLoc;
    rsrpStatistic(k) = rsrpSumStatistic/testDataLen/prm.NRepeatSameLoc;
    rsrpRandom(k)    = rsrpSumRandom/testDataLen/prm.NRepeatSameLoc;
end

figure
lineWidth = 1.5;
plot(1:K,rsrpOptimal,'--h','LineWidth',lineWidth,'Color',[0.6350 0.0780 0.1840]);
hold on
plot(1:K,rsrpNeural,'--*','LineWidth',lineWidth,'Color',colorNeural)
plot(1:K,rsrpKNN,'--o','LineWidth',lineWidth,'Color',colorKNN)
plot(1:K,rsrpStatistic,'--s','LineWidth',lineWidth,'Color',colorStats)
plot(1:K,rsrpRandom,'--d','LineWidth',lineWidth, 'Color',colorRandom)
hold off
grid on
xticks(1:K)
xlabel('$K$','interpreter','latex')
ylabel('Average RSRP')
title('Performance Comparison of Different Beam Pair Selection Schemes')
legend('Exhaustive Search','Neural Network','KNN','Statistical Info','Random','Location','best')
```
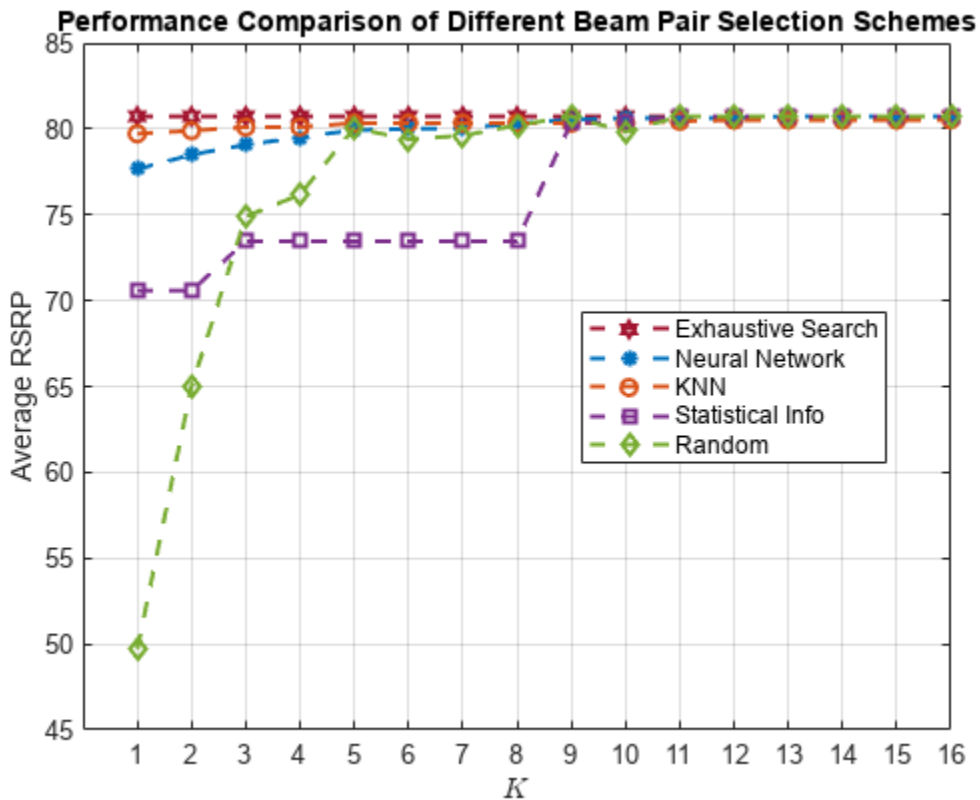
Compare the end values for the optimal, neural network, and KNN approaches.

```
[rsrpOptimal(end-3:end); rsrpNeural(end-3:end); rsrpKNN(end-3:end);]
```

ans = *3×4*

```
    80.7363    80.7363    80.7363    80.7363
    80.7363    80.7363    80.7363    80.7363
    80.5067    80.5068    80.5069    80.5212
```

The performance gap between KNN and the optimal methods indicates that the KNN might not perform well even when a larger set of beam pairs is considered, say, 256.

**Plot Confusion Matrix**

We observe that the classes with fewer elements are negatively impacted with the trained network. Using different weights for different classes could avoid this. Explore the same with the `useDiffClassWeights` logical and specify custom weights per class.

```
predLabels = classify(net,testInput,'ExecutionEnvironment','cpu');
figure;
cm = confusionchart(testOutCat,predLabels);
title('Confusion Matrix')
```

**Confusion Matrix**

## Conclusion and Further Exploration

This example describes the application of a neural network to the beam selection task for a 5G NR system. You can design and train a neural network that outputs a set of $K$ good beam pairs. Beam sweeping overhead can be reduced by an exhaustive search only on those selected $K$ beam pairs.

The example allows you to specify the scatterers in a MIMO channel. To see the impact of the channel on the beam selection, experiment with different scenarios. The example also provides presaved datasets that can be used to experiment with different network structures and training hyperparameters.

From simulation results, for the prerecorded MIMO scattering channel for 16 beam pairs, the proposed algorithm can achieve a top-$K$ accuracy of 90% when $K = 8$. This indicates with the neural network it is sufficient to perform an exhaustive search over only half of all the beam pairs, reducing the beam sweeping overhead by 50%. Experiment with varying other system parameters to see the efficacy of the network by regenerating data, then retraining and retesting the network.

## References

**1**   3GPP TR 38.802, "Study on New Radio access technology physical layer aspects." 3rd Generation Partnership Project; Technical Specification Group Radio Access Network.

**2**   Klautau, A., González-Prelcic, N., and Heath, R. W., "LIDAR data for deep learning-based mmWave beam-selection," IEEE Wireless Communications Letters, vol. 8, no. 3, pp. 909–912, Jun. 2019.

**3** Heng, Y., and Andrews, J. G., "Machine Learning-Assisted Beam Alignment for mmWave Systems," 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1-6, doi: 10.1109/GLOBECOM38437.2019.9013296.

**4** Klautau, A., Batista, P., González-Prelcic, N., Wang, Y., and Heath, R. W., "5G MIMO Data for Machine Learning: Application to Beam-Selection Using Deep Learning," 2018 Information Theory and Applications Workshop (ITA), 2018, pp. 1-9, doi: 10.1109/ITA.2018.8503086.

**5** Matteo, Z., <https://github.com/ITU-AI-ML-in-5G-Challenge/PS-012-ML5G-PHY-Beam-Selection_BEAMSOUP> (This is the team achieving the highest test score in the ITU Artificial Intelligence/Machine Learning in 5G Challenge in 2020).

**6** Sim, M. S., Lim, Y., Park, S. H., Dai, L., and Chae, C., "Deep Learning-Based mmWave Beam Selection for 5G NR/6G With Sub-6 GHz Channel Information: Algorithms and Prototype Validation," IEEE Access, vol. 8, pp. 51634-51646, 2020.

**Local Function**

```matlab
function prm = validateParams(prm)
% Validate user specified parameters and return updated parameters
%
% Only cross-dependent checks are made for parameter consistency.

    if strcmpi(prm.FreqRange,'FR1')
        if prm.CenterFreq > 7.125e9 || prm.CenterFreq < 410e6
            error(['Specified center frequency is outside the FR1 ', ...
                    'frequency range (410 MHz - 7.125 GHz).']);
        end
        if strcmpi(prm.SSBlockPattern,'Case D') ||  ...
            strcmpi(prm.SSBlockPattern,'Case E')
             error(['Invalid SSBlockPattern for selected FR1 frequency ' ...
                   'range. SSBlockPattern must be one of ''Case A'' or ' ...
                   '''Case B'' or ''Case C'' for FR1.']);
        end
        if ~((length(prm.SSBTransmitted)==4) || ...
             (length(prm.SSBTransmitted)==8))
             error(['SSBTransmitted must be a vector of length 4 or 8', ...
                   'for FR1 frequency range.']);
        end
        if (prm.CenterFreq <= 3e9) && (length(prm.SSBTransmitted)~=4)
            error(['SSBTransmitted must be a vector of length 4 for ' ...
                   'center frequency less than or equal to 3GHz.']);
        end
        if (prm.CenterFreq > 3e9) && (length(prm.SSBTransmitted)~=8)
            error(['SSBTransmitted must be a vector of length 8 for ', ...
                   'center frequency greater than 3GHz and less than ', ...
                   'or equal to 7.125GHz.']);
        end
    else % 'FR2'
        if prm.CenterFreq > 52.6e9 || prm.CenterFreq < 24.25e9
            error(['Specified center frequency is outside the FR2 ', ...
                   'frequency range (24.25 GHz - 52.6 GHz).']);
        end
        if ~(strcmpi(prm.SSBlockPattern,'Case D') || ...
                strcmpi(prm.SSBlockPattern,'Case E'))
            error(['Invalid SSBlockPattern for selected FR2 frequency ' ...
                   'range. SSBlockPattern must be either ''Case D'' or ' ...
                   '''Case E'' for FR2.']);
        end
```

```matlab
        if length(prm.SSBTransmitted)~=64
            error(['SSBTransmitted must be a vector of length 64 for ', ...
                    'FR2 frequency range.']);
        end
    end

    % Number of beams at transmit/receive ends
    prm.numBeams = sum(prm.SSBTransmitted);

    prm.NumTx = prod(prm.TxArraySize);
    prm.NumRx = prod(prm.RxArraySize);
    if prm.NumTx==1 || prm.NumRx==1
        error(['Number of transmit or receive antenna elements must be', ...
                ' greater than 1.']);
    end
    prm.IsTxURA = (prm.TxArraySize(1)>1) && (prm.TxArraySize(2)>1);
    prm.IsRxURA = (prm.RxArraySize(1)>1) && (prm.RxArraySize(2)>1);

    if ~( strcmpi(prm.RSRPMode,'SSSonly') || ...
          strcmpi(prm.RSRPMode,'SSSwDMRS') )
        error(['Invalid RSRP measuring mode. Specify either ', ...
                '''SSSonly'' or ''SSSwDMRS'' as the mode.']);
    end

    % Select SCS based on SSBlockPattern
    switch lower(prm.SSBlockPattern)
        case 'case a'
            scs = 15;
            cbw = 10;
            scsCommon = 15;
        case {'case b', 'case c'}
            scs = 30;
            cbw = 25;
            scsCommon = 30;
        case 'case d'
            scs = 120;
            cbw = 100;
            scsCommon = 120;
        case 'case e'
            scs = 240;
            cbw = 200;
            scsCommon = 120;
    end
    prm.SCS = scs;
    prm.ChannelBandwidth = cbw;
    prm.SubcarrierSpacingCommon = scsCommon;
end
```

# Aircraft-to-Satellite Communication for ADS-B Out

This example examines the Automatic Dependent Surveillance–Broadcast (ADS-B) Out connectivity of a flight from JFK International Airport in New York to L.F. Wade International Airport in Bermuda. The ADS-B Out system can connect to a variety of targets including airports and the Iridium NEXT satellite constellation. As a result, even while the aircraft does not have a direct line of sight with any ground station due to the Earth curvature over open water, the satellite ADS-B connection allows the aircraft to still provide continuous attitude, GPS position, and status to all other aircraft and airports interested in the aircraft state. The ADS-B system responds to the strongest ADS-B update initiator. The omnidirectional antennas on the top and bottom of the aircraft broadcast out to initiate the ADS-B update.

This example consists of three main parts:

1    Create the scenario containing the aircraft, airports, and Iridium satellite network.
2    Perform access analysis to determine when the aircraft has line-of-sight access with the airports or satellites within the Iridium network.
3    Perform link closure analysis of the ADS-B Out system with the airports and the satellites.

This example uses:

- Sensor Fusion and Tracking Toolbox™ `geoTrajectory` (Sensor Fusion and Tracking Toolbox) function to generate the aircraft trajectory.
- (Optionally) You can also use the `geoTrajectory` (Radar Toolbox) function in Radar Toolbox.
- Satellite Communications Toolbox `satelliteScenario` (Satellite Communications Toolbox) object to analyze and visualize the results for both the satellite, aircraft, and ground station.
- (Optionally) You can also create `satelliteScenario` (Aerospace Toolbox) object using the Aerospace Toolbox.
- Satellite Communications Toolbox `link` (Satellite Communications Toolbox) and `linkIntervals` (Satellite Communications Toolbox) functions to determine communications links.
- (Optionally) Phased Array System Toolbox `phased.CosineAntennaElement` functions to approximate the 48-beam antenna pattern of the Iridium satellite.

For more information on the Iridium NEXT satellite constellation and network, see https://www.iridium.com/network/.

**Construct the Scenario**

Create a satellite scenario. Specify a start date and duration for the scenario. Choose the duration to equal to the flight time from New York to Bermuda, approximately six hours.

```
startTime = datetime(2020,5,5,0,0,0);
stopTime = startTime + hours(6);
sampleTime = 60;                                        %seconds
sc = satelliteScenario(startTime,stopTime,sampleTime);
viewer = satelliteScenarioViewer(sc);
```

**Airports**

Create the airports properties in the scenario.

```
airportName = ["JFK International (New York)";...
    "L.F. Wade International Airport (Bermuda)"];
airportLat = [40.6413;32.3634];
airportLon = [-73.7781;-64.7053];
```

Add the airports to the scenario using the `groundStation` (Satellite Communications Toolbox) function.

```
airports = groundStation(sc,airportLat,airportLon,Name=airportName);
```

**Aircraft**

Create a `geoTrajectory` (Sensor Fusion and Tracking Toolbox) starting and ending latitude/longitude/altitude (LLA) objects based on the airport locations and flying altitude of 10.6 km, which is typical of large commercial aircraft. The `geoTrajectory` function generates trajectories based on waypoints in geodetic coordinates. Set the flight time based on the scenario duration. Sample the trajectory every 30 seconds.

```
startLLA = [airportLat(1) airportLon(1) 10600];
endLLA = [airportLat(2) airportLon(2) 10600];
timeOfTravel = [0 seconds(sc.StopTime-sc.StartTime)];
sampleRate   = 1/30;

trajectory = geoTrajectory([startLLA;endLLA],timeOfTravel,...
    SampleRate=sampleRate,...
    AutoPitch=true,AutoBank=true);
```

Output the LLA waypoints of the trajectory and orientation of the aircraft at the sample rate of the scenario.

```
[positionLLA,orientation] = trajectory();

[aircraftPosition,aircraftOrientation] = lookupPose(trajectory,...
    trajectory.TimeOfArrival(1):(1/trajectory.SampleRate):trajectory.TimeOfArrival(end));
```

Generate a `timetable` for the aircraft position and orientation. Use the `retime` function to interpolate the position and orientation into the same time step as the scenario, every 60 seconds.

```
aircraftPositionTT = timetable(aircraftPosition,...
    StartTime=sc.StartTime,...
    TimeStep=seconds(1/trajectory.SampleRate),...
    VariableNames="Lat-Lon-Alt");
aircraftOrientationTT = timetable(compact(aircraftOrientation),...
    StartTime=sc.StartTime,...
    TimeStep=seconds(1/trajectory.SampleRate),...
    VariableNames="Orientation");

aircraftPositionTT = retime(aircraftPositionTT,'regular','linear',TimeStep=seconds(sc.SampleTime
aircraftOrientationTT = retime(aircraftOrientationTT,'regular','nearest',TimeStep=seconds(sc.Samp
```

Note: You sample the aircraftOrientationTT to the nearest neighbor to avoid interpolation errors of the quaternion values.

Visualize the aircraft trajectory using `geoplot`.

```
lat = aircraftPositionTT.("Lat-Lon-Alt")(:,1);
lon = aircraftPositionTT.("Lat-Lon-Alt")(:,2);
geoplot(lat, lon, "b-")
```

```
geolimits([31,42],[-76,-63]);
geobasemap topographic;
```
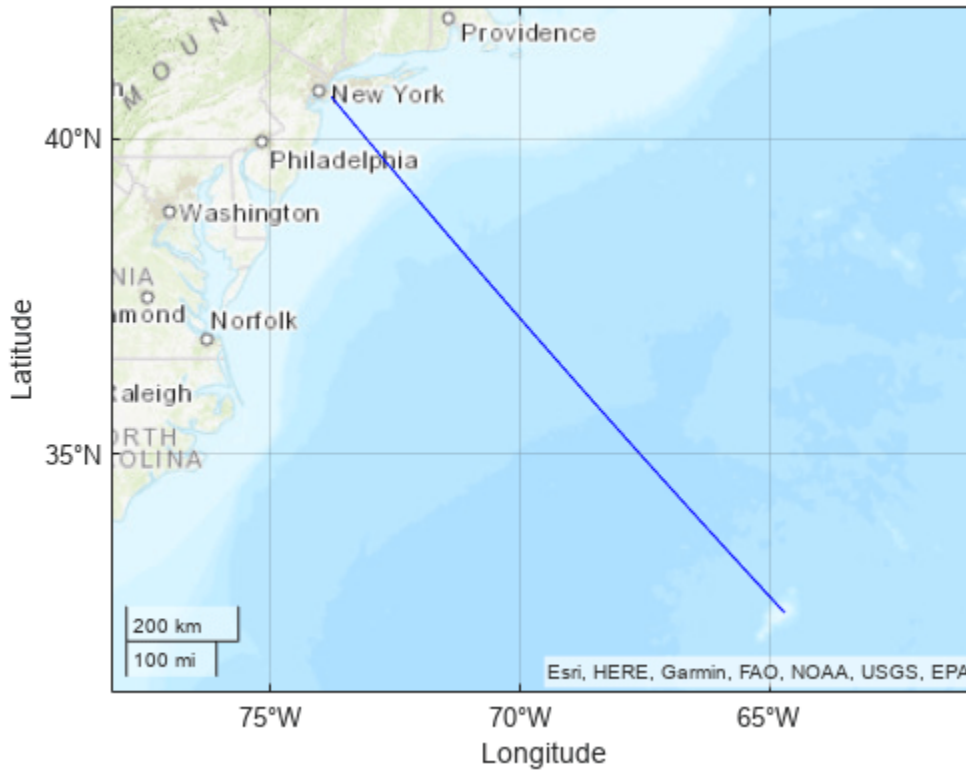


Add the aircraft to the scenario using the `satellite` (Satellite Communications Toolbox) function. The satellite function approximates the aircraft as a satellite with the ephemeris specified by `aircraftPositionTT` and `aircraftOrientationTT`.

```
aircraft = satellite(sc,aircraftPositionTT, CoordinateFrame="geographic", Name="Aircraft");
pointAt(aircraft,aircraftOrientationTT,CoordinateFrame="ned");
hide(aircraft.Orbit);
show(aircraft.GroundTrack);
```

**Satellites**

The Iridium NEXT satellite network, launched between 2018 and 2019 [1], contains 66 active LEO satellites with:

- Six orbital planes with an approximate inclination of 86.6 degrees and difference of RAAN of approximately 30 degrees between planes [1].
- 11 satellites per orbital plane with an approximate difference in true anomaly of 32.7 degrees between satellites [1].

Add the active Iridium NEXT satellites to the scenario. Create the orbital elements for the satellites in the Iridium network and create the satellites using the `satellite` (Satellite Communications Toolbox) function.

```matlab
numSatellitesPerOrbitalPlane = 11;
numOrbits = 6;

RAAN = [];
trueanomaly = [];

for i = 1:numOrbits
    for j = 1:numSatellitesPerOrbitalPlane

        RAAN(end+1) = 180*(i-1)/numOrbits; %#ok<SAGROW>
        if mod(i,2)
            trueanomaly(end+1) = 360*(j-1)/numSatellitesPerOrbitalPlane; %#ok<SAGROW>
        else
            % Satellites offset in alternating orbits
            trueanomaly(end+1) = 360*(j-1 + 0.5)/numSatellitesPerOrbitalPlane; %#ok<SAGROW>
        end
    end
end

semimajoraxis = repmat((6371 + 780)*1e3,size(RAAN)); % kms
inclination = repmat(86.4,size(RAAN)); % degrees
eccentricity = zeros(size(RAAN)); % degrees
argofperiapsis = zeros(size(RAAN)); % degrees

iridiumSatellites = satellite(sc,semimajoraxis,eccentricity,inclination,RAAN,argofperiapsis,truea
```

If you have a license for Aerospace Toolbox, you can also create the Iridium constellation using the `walkerStar` (Aerospace Toolbox) function.

Since many satellites share common orbital planes, display the orbital path of only the first satellite in each plane.

```matlab
hide(iridiumSatellites.Orbit,viewer);
show(iridiumSatellites(1:numSatellitesPerOrbitalPlane:end).Orbit,viewer);
```

Add conical sensors to the Iridium satellites to act as a camera to establish visual access between the satellites and the aircraft.

```matlab
iridiumConicalSensors = conicalSensor(iridiumSatellites,"MaxViewAngle",125);
```

**Aircraft Access Analysis**

Determine access analysis between the aircraft, airports, and satellites using the scenario. Calculate the access interval between the aircraft and airports using the `access` (Satellite Communications Toolbox) and `accessIntervals` (Satellite Communications Toolbox) functions based on when the aircraft has line-of-sight access to the airport. JFK and L.F. Wade airports can see the aircraft during the first and last two hours of the flight, respectively. Otherwise the aircraft is blocked by the curvature of the Earth.

```matlab
acAirport = access(aircraft,sc.GroundStations);
airportAccessIntvls = accessIntervals(acAirport)
```

airportAccessIntvls=*2×8 table*

| Source | Target | IntervalNumber | StartTime |
| --- | --- | --- | --- |
| "Aircraft" | "JFK International (New York)" | 1 | 05-May-2020 00 |

```
    "Aircraft"    "L.F. Wade International Airport (Bermuda)"         1         05-May-2020 04
```

Calculate a similar access analysis between the aircraft and the satellites based on the visibility of the aircraft within the conical sensors' field of view. This serves as a first-round approximation of the Iridium multibeam antenna shown in the next section.

```matlab
acSatellite = access(aircraft,iridiumConicalSensors);
satelliteAccessIntvls = accessIntervals(acSatellite);

[sSatellite,time] = accessStatus(acSatellite);
satVisPlotData = double(sSatellite);
satVisPlotData(satVisPlotData == false) = NaN; % Hide invisible satellites.
satVisPlotData = satVisPlotData + (0:numel(iridiumSatellites)-1)'; % Add space to satellites to l

plot(time,satVisPlotData," .",Color="blue")
yticks(1:5:66)
yticklabels(iridiumSatellites.Name(1:5:66))
title("Satellite Visibility")
grid on
xlabel("Time")
axis tight
```
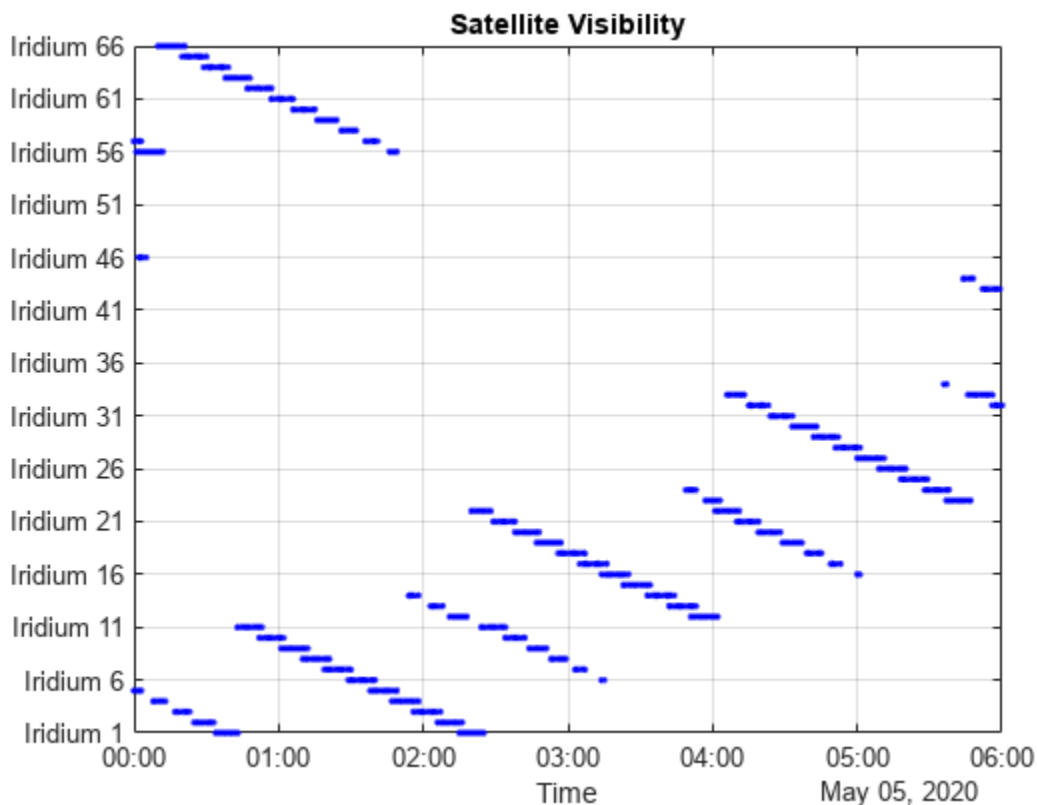


The access intervals computation shows that the aircraft has continuous satellite communication access with the Iridium network when the aircraft is over the open ocean. Combined with the airport access, the aircraft has continuous line-of-sight access throughout the flight.

### Aircraft Link Analysis

After establishing that the aircraft maintains a direct connection with either airports and the Iridium network throughout the entire flight, generate a link budget for the respective ground and satellite-based ADS-B Out systems. The ADS-B Out system uses the existing Mode A/C and S transponder antennas located on the top and bottom of the aircraft, respectively. ADS-B uses the Mode S of the transponder and operates the ADS-B Out on a frequency of 1090 MHz with a minimum effective isotropic radiated power (EIRP) of 125 Watts.

```
fADSB = 1090e6; % 1090 MHz
```

For this example, model the pair of Mode-S transponders as a single isotropic transmitter at the center of the aircraft.

```
aircraftADSBAntenna = arrayConfig("Size",[1 1]); % Create an isotropic antenna element

aircraftADSBTransmitter = transmitter(aircraft, ...
    Antenna = aircraftADSBAntenna, ...
    Frequency = fADSB,...
    Power=10*log10(125),...      % ADS-B Out minimum as EIRP of 125 Watts
    MountingAngles = [0;0;0],... % In degrees
    Name="ADS-B Aircraft Transmitter");
pattern(aircraftADSBTransmitter,Size=50000);
```

### Aircraft to Airport ADS-B Out Link

Add isotropic antennas and receivers to the airports.

```
% Airport Antenna
airportADSBAntenna = arrayConfig("Size",[1 1]); % Create an isotropic antenna element
airportADSBReceiver = receiver(...
    airports, ...
    Antenna=airportADSBAntenna, ...
    Name=airports.Name + " Receiver");
pattern(airportADSBReceiver,fADSB,Size=50000);
```

### Aircraft to Satellite ADS-B Out Link

Using only Satellite Communication Toolbox, you can model the antenna on the Iridium satellite as an isotropic antenna. If you have a license for Phased Array Toolbox, you can model an approximation of the custom 48-beam antenna used by the Iridium NEXT satellites [2]. Use this dropdown to select the antenna to model.

```
antennaType = Isotropic           ▼ ;
```

### Simulate Iridium Satellites Link with Isotropic Antennas

Create and add an isotropic antenna receiver to the Iridium satellites.

```
if antennaType == "Isotropic"

    % Create and add an isotropic antenna receiver to the Iridium satellites.
    satelliteADSBAntenna = arrayConfig("Size",[1 1]); % Add code comment
    satelliteADSBReceiver = receiver(iridiumSatellites, ...
        Antenna=satelliteADSBAntenna, ...
        MountingAngles=[0,0,0], ...
        Name=iridiumSatellites.Name + " Receiver");
```

```
        pattern(satelliteADSBReceiver,fADSB,Size=50000);

        % Play the scenario.
        play(sc);
    end
```
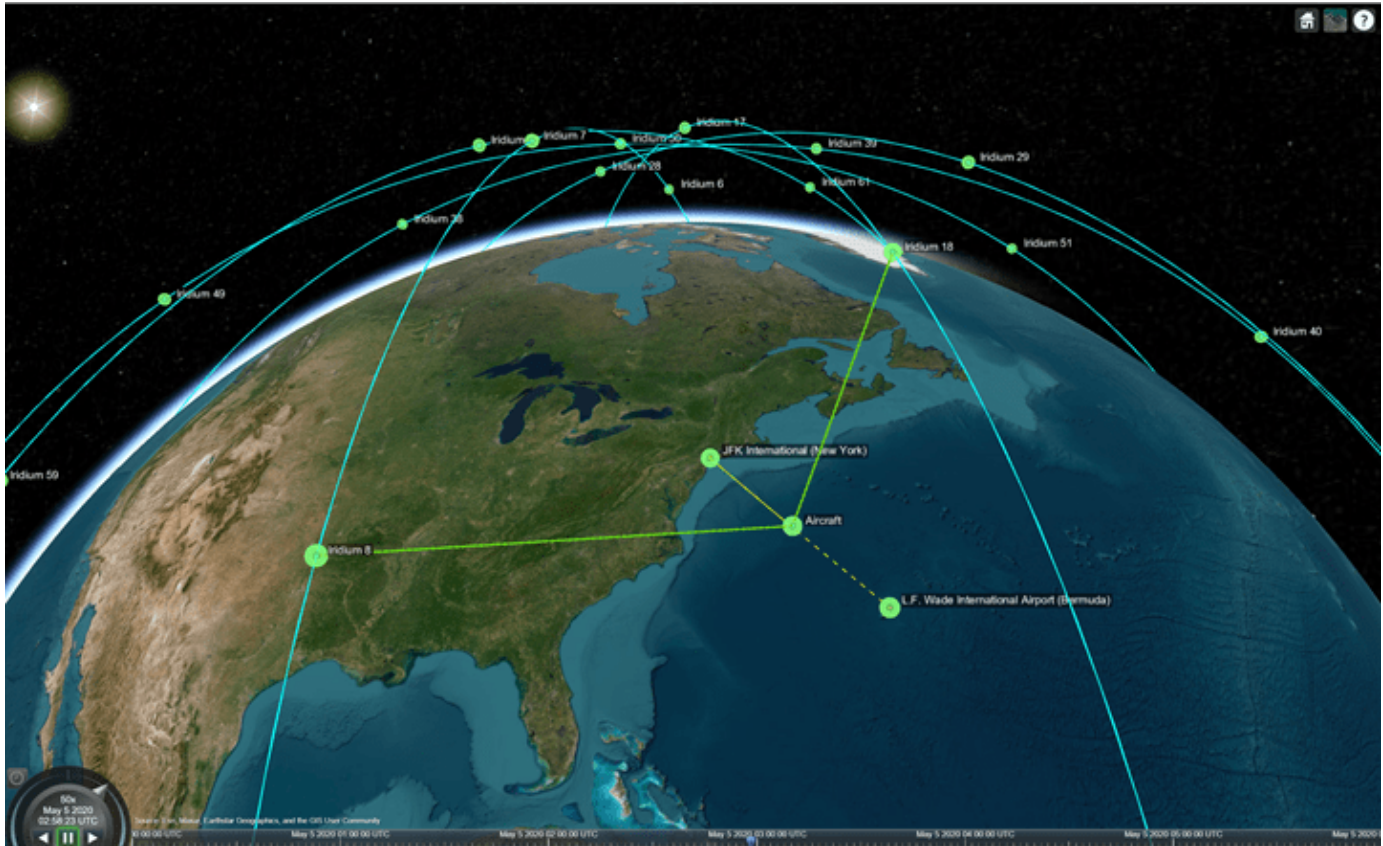


**Simulate Iridium Satellites Link with Custom 48-Beam Antenna**

Create and add a custom antenna element receiver to the Iridium satellites using the
**HelperCustom48BeamAntenna** convenience function.

Note: The mounting angle matches the default frames of Phased Array System Toolbox and the
Satellite Communications Toolbox.

```
if antennaType == "Custom 48-Beam"

    % Use Custom Antenna Element on 48-spot beam
    satelliteADSBAntenna = HelperCustom48BeamAntenna(fADSB);
    satelliteADSBReceiver = receiver(iridiumSatellites, ...
        Antenna=satelliteADSBAntenna, ...
        MountingAngles=[0,-90,0], ...
        Name=iridiumSatellites.Name + " Receiver");
    pattern(satelliteADSBReceiver,fADSB,Size=300000);

    % Play the scenario.
    play(sc);
end
```
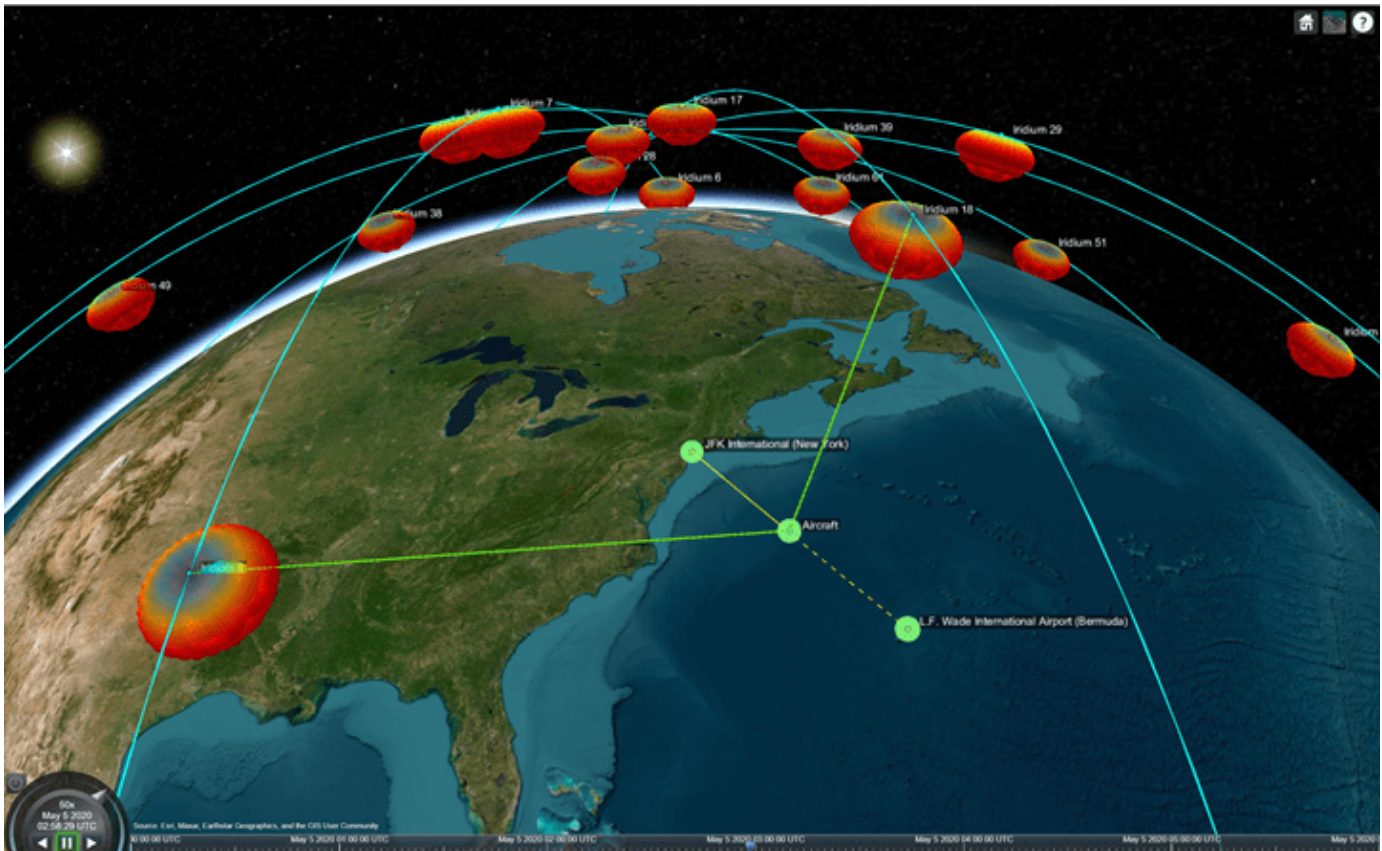
**ADS-B Out Link Closure Times**

Add ADS-B Out link analysis from the aircraft to airports and satellites.

```
lnkADSB = link(aircraftADSBTransmitter, [airportADSBReceiver, satelliteADSBReceiver]);
```

Plot the maximum margin for the link to any of the available receivers. The margin is the received Eb/No minus the required Eb/No for that receiver.

```
[eL,time] = ebno(lnkADSB);
marginADSB = eL - repmat([airportADSBReceiver.RequiredEbNo,satelliteADSBReceiver.RequiredEbNo]',
figure;
plot(time,max(marginADSB),"b")
xlabel("Time");
ylabel("Margin (dB)");
title("ADS-B Out Link Margin vs. Time");
```
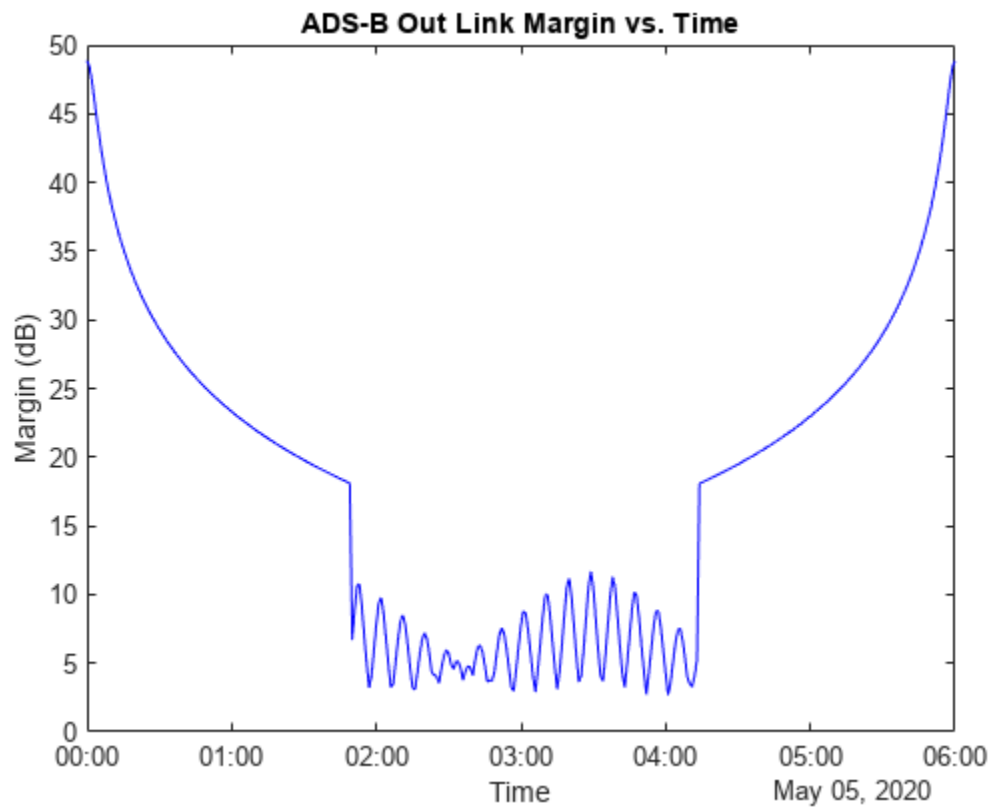
**References**

[1] *Attachment EngineeringStatement SAT-MOD-20131227-00148*. https://fcc.report/IBFS/SAT-MOD-20131227-00148/1031348. Accessed 17 Jan. 2023.

[2] *Attachment Exhibit A SAT-MOD-20131227-00148*. https://fcc.report/IBFS/SAT-MOD-20131227-00148/1031240. Accessed 17 Jan. 2023.

# Waveform Design for a Dual-Function MIMO RadCom System

This example shows how to design a set of waveforms for a dual-function multiple-input-multiple-output (MIMO) radar-communication (RadCom) system. It then uses the designed waveforms to simulate a RadCom system that can detect targets in range and azimuth, as well as transmit data to multiple downlink users.

**Joint Radar-Communication**

Radar systems require large bandwidth to achieve high resolution while communication systems require large bandwidth to achieve high data rates. With a growing number of connected devices, the frequency spectrum becomes increasingly congested. To make the frequency spectrum usage more efficient the radar and the communication communities have recently proposed a host of approaches for radar-communication coexistence. These approaches range from non-cooperative spectrum sharing that rely on spectrum sensing and dynamic spectrum access to joint RadCom systems optimized for both functions [1].
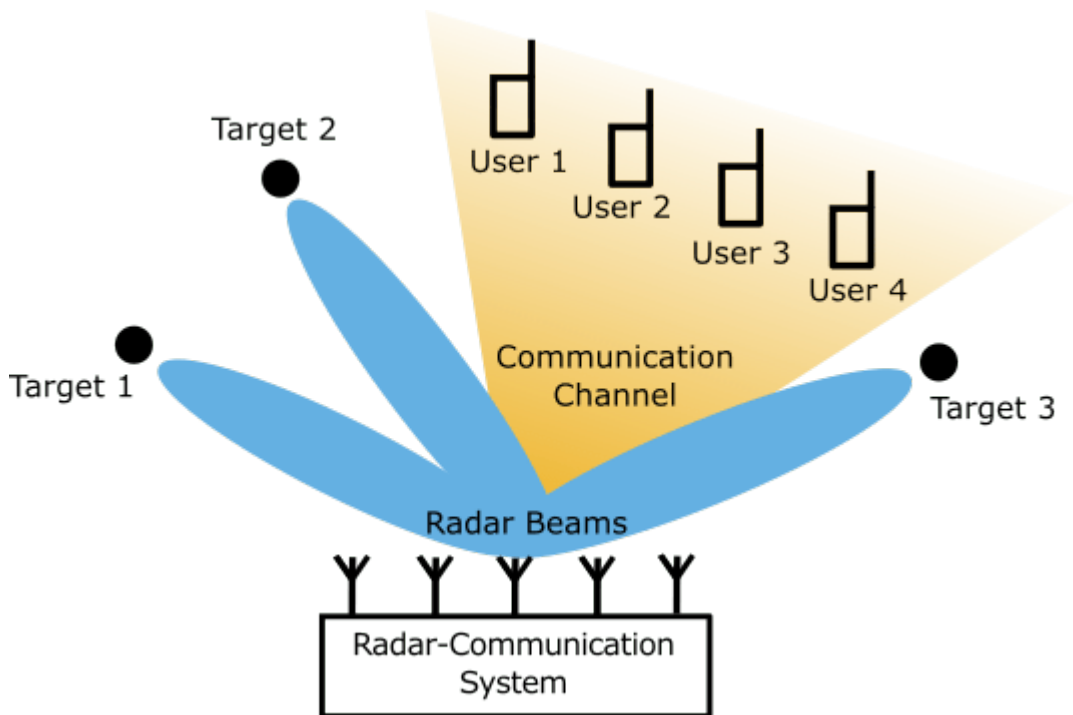


This example shows how to design a set of dual-function waveforms for a joint RadCom system. This RadCom system uses a phased array to facilitate MIMO communication. It also takes advantage of waveform diversity to achieve good radar performance. Assume the carrier frequency is 6GHz, the peak transmit power is 0.5 MW, and the same uniform linear array (ULA) is used for transmitting and receiving. Create a ULA with $N = 16$ isotropic antenna elements spaced half of a wavelength apart.

```
rng('default');                    % Set random number generator for reproducibility

fc = 6e9;                          % Carrier frequency (Hz)
Pt = 0.5e6;                        % Peak transmit power (W)
lambda = freq2wavelen(fc);         % Wavelength (m)
```

```
N = 16;                                  % Number of array elements
d = 0.5*lambda;                          % Array element spacing (m)

% Use isotropic antenna elements
element = phased.IsotropicAntennaElement('BackBaffled', true);
array = phased.ULA('Element', element, 'NumElements', N, 'ElementSpacing', d, 'ArrayAxis', 'y');
```

Since the objectives of the radar and the communication components of the RadCom system are fundamentally different, each component imposes different requirements on the transmitted waveforms.

**Objective of the Radar Component**

The goal of the Rad component is to form multiple transmit beams in the directions of targets of interest (for example, the tracker subsystem can provide predicted target positions).

Create a desired transmit beam pattern with 10 degree beams pointing at each target.

```
% Three targets of interest
tgtAz = [-60 10 45];                 % Azimuths of the targets of interest
tgtRng = [5.31e3, 6.23e3, 5.7e3];    % Ranges of the targets of interest

ang = linspace(-90, 90, 200);        % Grid of azimuth angles
beamwidth = 10;                      % Desired beamwidth

% Desired beam pattern
idx = false(size(ang));
for i = 1:numel(tgtAz)
    idx = idx | ang >= tgtAz(i)-beamwidth/2 & ang <= tgtAz(i)+beamwidth/2;
end

Bdes = zeros(size(ang));
Bdes(idx) = 1;

figure
plot(ang, Bdes, 'LineWidth', 2)
xlabel('Azimuth (deg)')
title('Desired Beam Pattern')
grid on
ylim([0 1.1])
```

**Desired Beam Pattern**
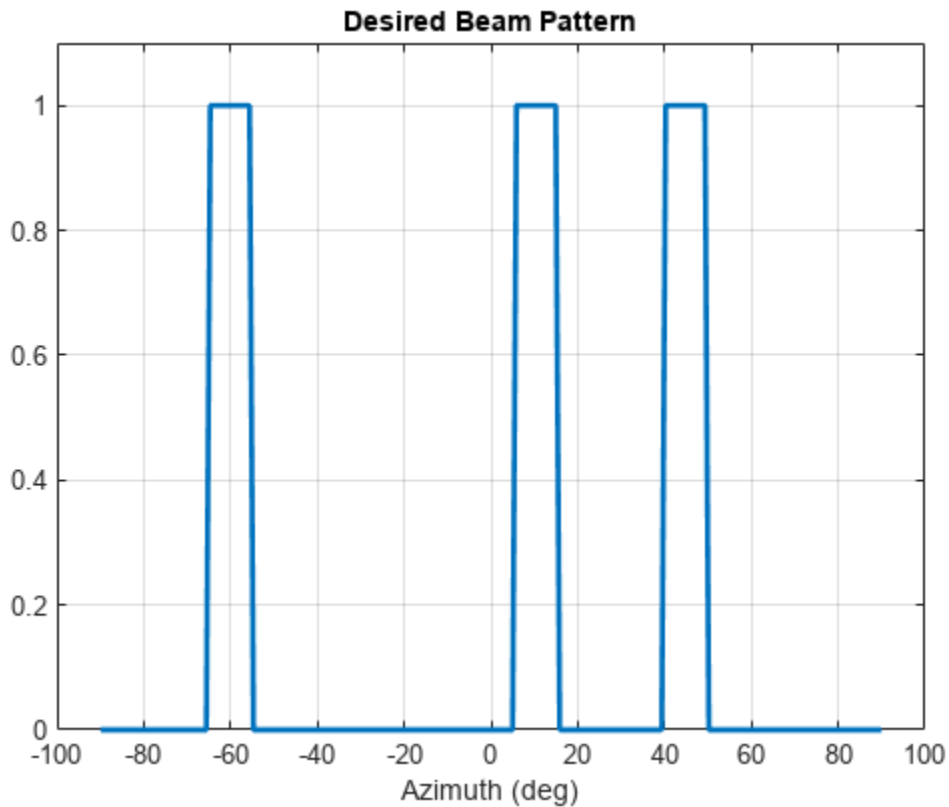


## Objective of the Communication Component

The goal of the Com component is to transmit communication symbols to downlink users. Let there be K=4 single-antenna users and a communication message consisting of M=30 symbols. Create a K-by-M matrix S of QPSK symbols that represents the desired communication message.

```
K = 4;                              % Number of communication users
M = 30;                             % Number of communication symbols

Q = 4;
data = randi([0 Q-1], K, M);        % Binary data
S = pskmod(data, Q, pi/Q);          % QPSK symbols
```

Create a K-by-N matrix H that models a scattering channel between N RadCom antenna elements and K downlink users. This matrix is typically assumed to be known to the RadCom system prior to the waveform design.

```
% User locations are random
txpos = [rand(1, K)*1.5e3; rand(1, K)*2.4e3 - 1.2e3; zeros(1, K)];

% Create a scattering channel matrix assuming 100 independent scatterers
numscat = 100;
rxpos = array.getElementPosition();
H = scatteringchanmtx(rxpos, txpos, numscat).'

H = 4×16 complex
```

```
    6.9220 - 1.1247i    7.1017 - 0.9594i    7.1987 - 0.8061i    7.2116 - 0.6666i    7.1400 - 0.5421i
  -10.6335 - 8.1281i   -9.9214 - 8.4327i   -9.1425 - 8.7545i   -8.3116 - 9.0850i   -7.4454 - 9.4151i
    4.5973 + 3.1755i    4.6600 + 3.4353i    4.7256 + 3.6782i    4.8029 + 3.9025i    4.9008 + 4.1061i
   -3.2160 - 5.8685i   -3.3480 - 5.4857i   -3.5441 - 5.0363i   -3.8054 - 4.5215i   -4.1313 - 3.9437i
```

Next, the example will show how to create a set of radar waveforms that satisfy the radar objective and form the desired transmit beam pattern. It will then show how to embed the desired communication message into the synthesized waveforms. Finally, the example will model a transmission from the RadCom system. It will plot the resulting range-angle response and compute the symbol error rate to verify that the system is capable of performing both functions.

**MIMO Radar Waveform Synthesis**

The Rad component of the RadCom system is a MIMO radar. Unlike a traditional phased array radar that transmits a scaled version of the same waveform from each antenna element, a MIMO radar utilizes waveform diversity and transmits $N$ different waveforms. These waveforms combine to form the transmit beam pattern. By adjusting the transmitted waveforms, a MIMO radar can dynamically adjust its transmit beam pattern.

Let an $N$-by-$M$ matrix $\mathbf{X}$ represent the transmitted waveforms such that the rows of $\mathbf{X}$ correspond to the $N$ antenna elements and the columns of $\mathbf{X}$ to $M$ subpulses. The resulting transmit beam pattern is [2]

$$B(\theta) = \frac{1}{4\Pi}\mathbf{a}^H(\theta) \cdot \mathbf{R} \cdot \mathbf{a}(\theta)$$

where $\mathbf{a}(\theta)$ is the array steering vector, $\theta$ is the azimuth angle, and $\mathbf{R} = \frac{1}{M}\mathbf{X}\mathbf{X}^H$ is the waveform covariance matrix. This relationship shows that design of a transmit beam pattern for a MIMO radar is equivalent to designing the waveform covariance matrix. Therefore, the waveform design process can be broken into two steps. First, you find a covariance matrix that produces the desired beam pattern, and then synthesize the waveforms based on the found covariance matrix. Designing the covariance matrix first is much easier compared to selecting the waveforms directly, since typically $M \gg N$, and thus $\mathbf{R}$ has much fewer unknowns than $\mathbf{X}$.

**From Beam Pattern to Covariance Matrix**

Given a desired beam pattern $B_{des}(\theta)$, a waveform covariance matrix $\mathbf{R}$ can be found by minimizing the mean squared error between the desired and the actual beam pattern [2]

$$\min_{\mathbf{R}} \int_\theta \left| B_{des}(\theta) - \frac{1}{4\Pi}\mathbf{a}^H(\theta) \cdot \mathbf{R} \cdot \mathbf{a}(\theta) \right|^2 cos(\theta)d\theta$$

s.t. $diag(\mathbf{R}) = \frac{P_t}{N}\mathbf{1}$

$\quad\mathbf{R} \in \mathbb{S}_+^N$

where $P_t$ is the total transmit power. The first constraint in this optimization problem requires that the diagonal elements of $\mathbf{R}$ are equal to $\frac{P_t}{N}$. This guarantees that each antenna element transmits equal power, and the total transmitted power is equal to $P_t$. The second constraint restricts $\mathbf{R}$ to the set of positive semidefinite matrices, which is a necessary requirement for a valid covariance matrix.

Use the `helperMMSECovariance` helper function to solve this optimization problem for the desired beam pattern in `Bdes`. The solution is the optimal covariance matrix `Rmmse` that results in a beam pattern closest to the desired beam pattern in the minimum mean squared error (MMSE) sense.

```matlab
% Normalize the antenna element positions by the wavelength
normalizedPos = rxpos/lambda;

% Solve the optimization problem to find the covariance matrix
Rmmse = helperMMSECovariance(normalizedPos, Bdes, ang);

% helperMMSECovariance returns a covariance matrix with 1s along the
% diagonal such that the total transmit power is equal to N. Renormalize
% the covariance matrix to make the total transmit power equal to Pt.
Rmmse = Rmmse*Pt/N;

% Matrix of steering vectors corresponding to the angles in the grid ang
A = steervec(normalizedPos, [ang; zeros(size(ang))]);

% Compute the resulting beam pattern given the found covariance matrix
Bmmse = abs(diag(A'*Rmmse*A))/(4*pi);
```

Compare the beam pattern obtained through the mean squared optimization with the desired beam pattern.

```matlab
figure
hold on
plot(ang, pow2db(Bdes + eps), 'LineWidth', 2)
plot(ang, pow2db(Bmmse/max(Bmmse)), 'LineWidth', 2)

grid on
xlabel('Azimuth (deg)')
ylabel('(dB)')
legend('Desired', 'MMSE Covariance', 'Location', 'southoutside', 'Orientation', 'horizontal')
ylim([-30 1])
title('Transmit Beam Pattern')
```
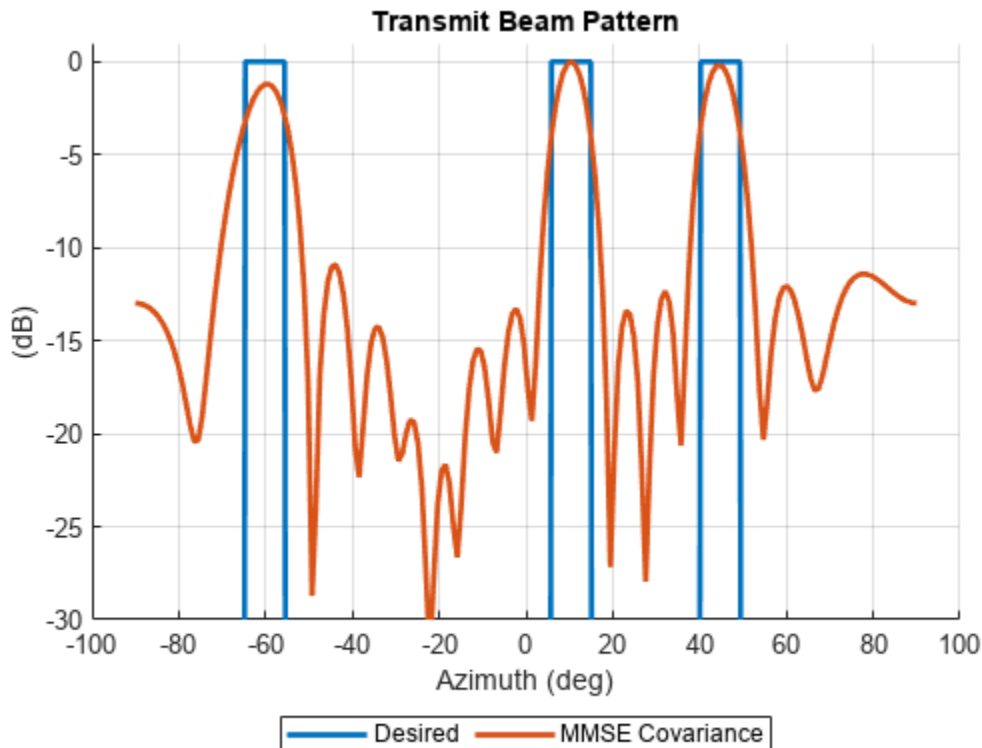
The beam pattern obtained from the covariance matrix in `Rmmse` has three main lobes that match the main lobes of the desired beam pattern and correspond to the directions of the targets of interest. The resulting sidelobe level depends on the available aperture size.

**From Covariance Matrix to Waveform**

Once the covariance matrix is known, the problem of synthesizing radar waveforms that produce the desired beam pattern reduces to the problem of finding a set of waveforms with a given covariance **R**. But to be useful in practice these waveforms also must meet a number of constraints. For power-efficient transmission both radar and communication systems typically require the waveforms to have a constant modulus. Since, it might be numerically difficult to find a set of constant modulus waveforms with a given covariance matrix, the waveforms can be constrained in terms of the peak-to-average-power ratio (PAR) instead. The PAR for the $n$th transmitted waveform is

$$PAR_n = \frac{1}{P_n} max_m |X(n, m)|^2$$

where $P_n = \frac{1}{M} \sum_{m=1}^{M} |X(n, m)|^2$ is the average power, $n = 1, 2, \dots. N$, and $m = 1, 2, \dots, M$. Notice that the diagonal elements of the waveform covariance matrix $R(n, n)$ are equal to the average powers of the waveforms $P_n$. If each antenna element transmits equal power, and the total transmit power is $P_t$, then $P_n = \frac{P_t}{N}$. For a set of $N$ waveforms, the following low PAR constraint can be set

$$
\begin{cases}
\dfrac{1}{M}\displaystyle\sum_{m=1}^{M}|X(n,m)|^2 = \dfrac{P_t}{N} & n = 1, 2, ..., N \\[2ex]
|X(n,m)|^2 \leq \eta\dfrac{P_t}{N} & \forall n, m \text{ and } \eta \in [1, M]
\end{cases}
$$

This constraint requires that the average power of each waveform in the set is equal to the total power divided by the number of antenna elements and that the power for each subpulse does not exceed the average power by more than a factor of $\eta$. If $\eta$ is set to 1, the low PAR constraint becomes the constant modulus constraint.

In [3] the authors propose a cyclic algorithm (CA) for synthesizing waveforms given a desired covariance matrix such that the PAR constraint is satisfied. The helper function `helperCAWaveformSynthesis` implements a version of this algorithm described in [4] that also minimizes the autocorrelation sidelobes for each waveform. Set the parameter $\eta$ to 1.1 and use `helperCAWaveformSynthesis` to find a set of waveforms that have a covariance matrix equal to the optimal covariance matrix in `Rmmse`. Set the number of subpulses in the generated waveform to be equal to the number of symbols in the desired communication symbol matrix `S`.

```matlab
eta = 1.1;                              % Parameter that controls low PAR constraint

% Find a set of waveform with the covariance equal to Rmmse using the
% cyclic algorithm. The length of each waveform is M.
Xca = helperCAWaveformSynthesis(Rmmse, M, eta);

% Covariance matrix of the computed waveforms
Rca = Xca*Xca'/M;

% The resulting beam pattern
Bca = abs(diag(A'*Rca*A))/(4*pi);
```

Compare the beam pattern formed by the synthesized waveforms `Xca` with the beam pattern produced by `Rmmse` and the desired beam pattern `Bdes`.

```matlab
figure
hold on
plot(ang, pow2db(Bdes + eps), 'LineWidth', 2)
plot(ang, pow2db(Bmmse/max(Bmmse)), 'LineWidth', 2)
plot(ang, pow2db(Bca/max(Bca)), 'LineWidth', 2)

grid on
xlabel('Azimuth (deg)')
ylabel('(dB)')
legend('Desired', 'MMSE Covariance', 'Radar Waveforms', 'Location', 'southoutside', 'Orientation
ylim([-30 1])
title('Transmit Beam Pattern')
```
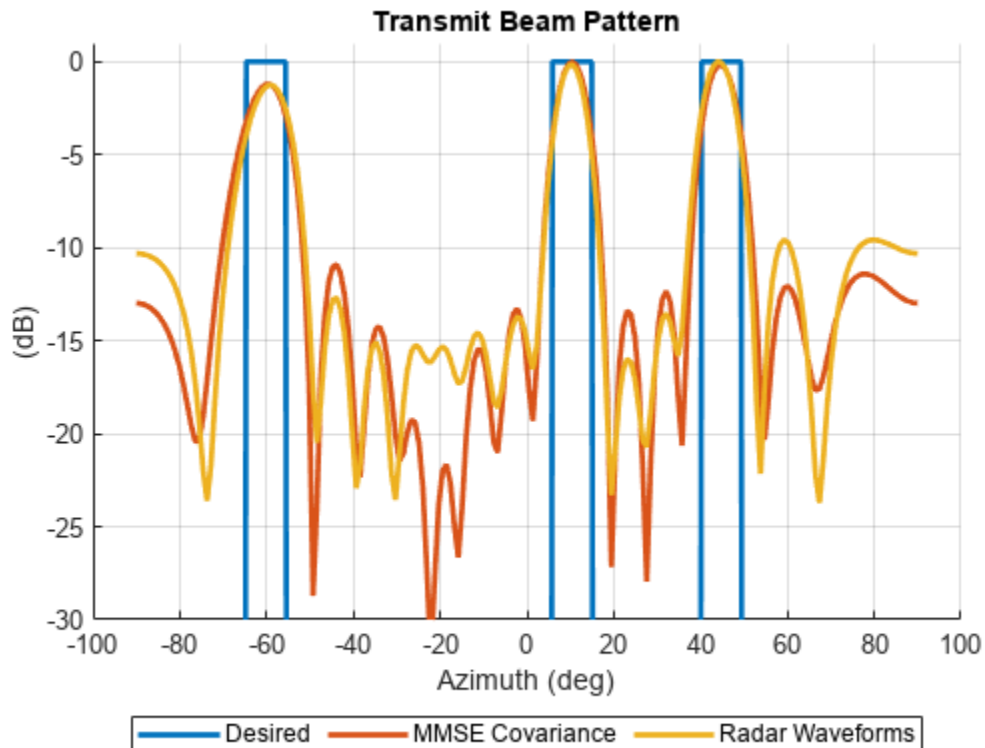
The waveforms synthesized by CA form a beam pattern close to the beam pattern produced by `Rmmse`. Some distortion is present, but the main lobes are mostly undistorted. Since the diagonal elements of `Rmmse` were constraint to be equal to $\frac{P_t}{N}$, the synthesized waveforms `Xca` must have average power equal to 0.5e6/16=31250 watt. Verify that the defined low PAR constraint is satisfied for all synthesized waveforms.

```
% Average power
Pn = diag(Rca);

% Peak-to-average power ratio
PARn = max(abs(Xca).^2, [], 2)./Pn;

array2table([Pn.'; PARn.'], 'VariableNames', compose('%d', 1:N), 'RowNames', {'Average Power', '|
```

*ans=2×16 table*

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Average Power | 31250 | 31250 | 31250 | 31250 | 31250 | 31250 | 31250 |
| Peak-to-Average Power Ratio | 1.1 | 1.1 | 1.1 | 1.1 | 1.1 | 1.1 | 1.1 |

To verify that the generated waveforms have low range sidelobes (an important requirement for a good radar waveform), plot the autocorrelation function for one of the waveforms in `Xca`.

```
% Plot the autocorrelation function for the nth waveform in Xca
n = 1;
[acmag_ca, delay] = ambgfun(Xca(n, :), 1, 1/M, "Cut", "Doppler");

figure;
ax = gca;
colors = ax.ColorOrder;
plot(delay, mag2db(acmag_ca), 'Color', colors(3, :), 'Linewidth', 2)
xlabel('Lag (subpulses)')
ylabel('(dB)')
title(sprintf('Autocorrelation Function for Waveform #%d', n))
grid on;
ylim([-45 0]);
```
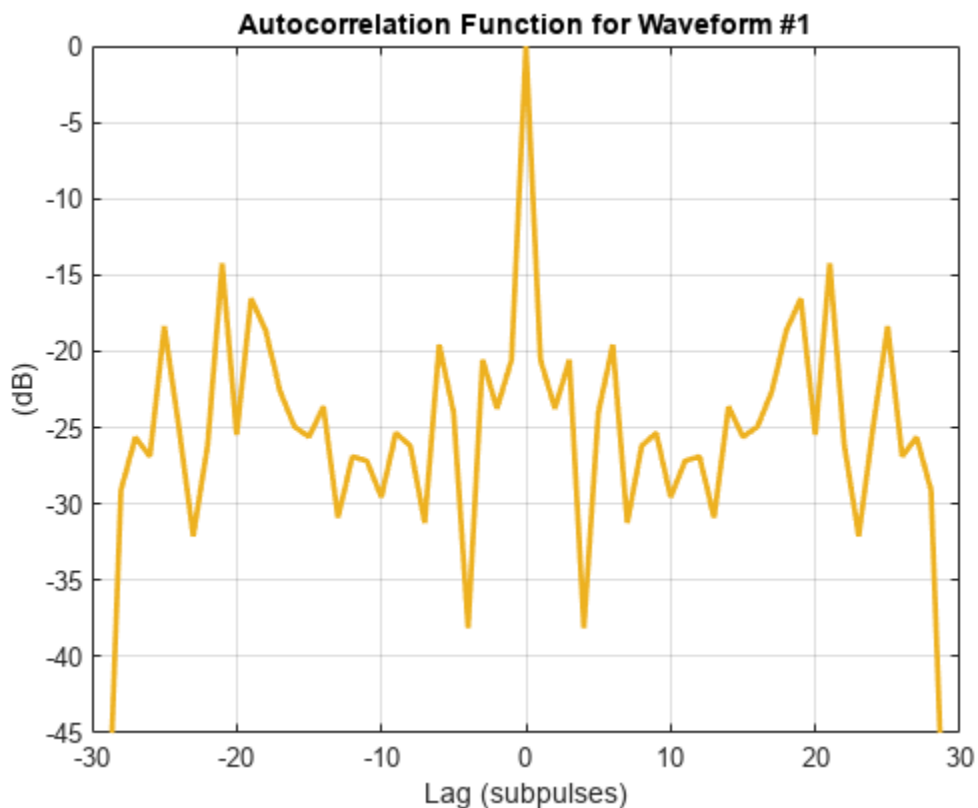


The range sidelobes for the generated waveforms are on the order of or lower than -13 dB, which is the sidelobe level of a rectangular LFM pulse.

**Radar-Communication Waveform Synthesis**

The RadCom system uses the same waveforms for joint radar and communication. If **H** is a known channel matrix and **X** is a matrix containing transmitted waveforms, the signal received by the downlink users is a product **HX**. One way to design a set of dual-function waveforms is to embed the communication symbols into the radar waveforms such that the received signal **HX** is as close as possible to the set of the desired communication symbols **S**. In other words, such that the difference **HX − S** is minimized. This difference is known as the multi-user interference (MUI).

A set of suitable waveforms is a solution to the following trade-off objective function with two terms: the first term is the MUI, while the second term represents the mismatch between the desired and the resulting beam pattern

$$\min_{\mathbf{X}} \rho \|\mathbf{HX} - \mathbf{S}\|_F^2 + (1 - \rho)\|\mathbf{X} - \mathbf{X}_0\|_F^2$$

s.t. $\dfrac{1}{L} diag(\mathbf{XX}^H) = \dfrac{P_t}{N}\mathbf{1}$

here $\mathbf{X}_0$ is the initial set of radar waveforms that form the desired beam pattern and $\rho$ is a parameter controlling the trade-off between the radar and the communication function. Again, the constraint in this optimization problem guarantees that all antenna elements transmit equal power, and the total power is equal to $P_t$.

If $\rho$ is close to zero, the obtained waveforms will be as close as possible to the initial radar waveforms guaranteeing a good radar performance. However, the MUI in this case can be high resulting in a poor communication performance. When $\rho$ is close to one, the MUI minimization is prioritized resulting in a good communication performance, but possibly poor radar performance. In this case the resulting waveforms might diverge significantly from the initial radar waveforms and therefore the resulting beam pattern might be significantly different from the desired beam pattern.

Authors in [5] use the Riemannian Conjugate Gradient (RCG) method to solve the trade-off optimization and find a set of dual-function waveforms. Use the `helperRadComWaveform` helper function that implements the RCG method to embed the 4-by-30 set of QPSK symbols `S` into the set of synthesized radar waveforms `Xca` using the generated 4-by-16 channel matrix `H`.

```
% Radar-communication trade-off parameter
rho = 0.4;

% Use RCG to embed the communication symbols in S into the waveforms in Xca
% assuming known channel matrix H.
Xradcom = helperRadComWaveform(H, S, Xca, Pt, rho);

% Compute the corresponding waveform covariance matrix and the transmit beam pattern
Rradcom = Xradcom*Xradcom'/M;
Bradcom = abs(diag(A'*Rradcom*A))/(4*pi);
```

Compare the beam pattern formed by the obtained waveforms `Xradcom` against the desired beam pattern `Pdes`, the beam pattern obtained using the optimal covariance matrix `Rmmse`, and the beam pattern formed by the radar waveforms `Xca` generated by CA.

```
figure;
hold on;
plot(ang, pow2db(Bdes + eps), 'LineWidth', 2);
plot(ang, pow2db(Bmmse/max(Bmmse)), 'LineWidth', 2);
plot(ang, pow2db(Bca/max(Bca)), 'LineWidth', 2);
plot(ang, pow2db(Bradcom/max(Bradcom)), 'LineWidth', 2);

grid on;
xlabel('Azimuth (deg)');
ylabel('(dB)');
legend('Desired', 'MMSE Covariance', 'Radar Waveforms', 'RadCom Waveforms',...
    'Location', 'southoutside', 'Orientation', 'horizontal', 'NumColumns', 2)
ylim([-30 1])
title('Transmit Beam Pattern')
```

**Transmit Beam Pattern**
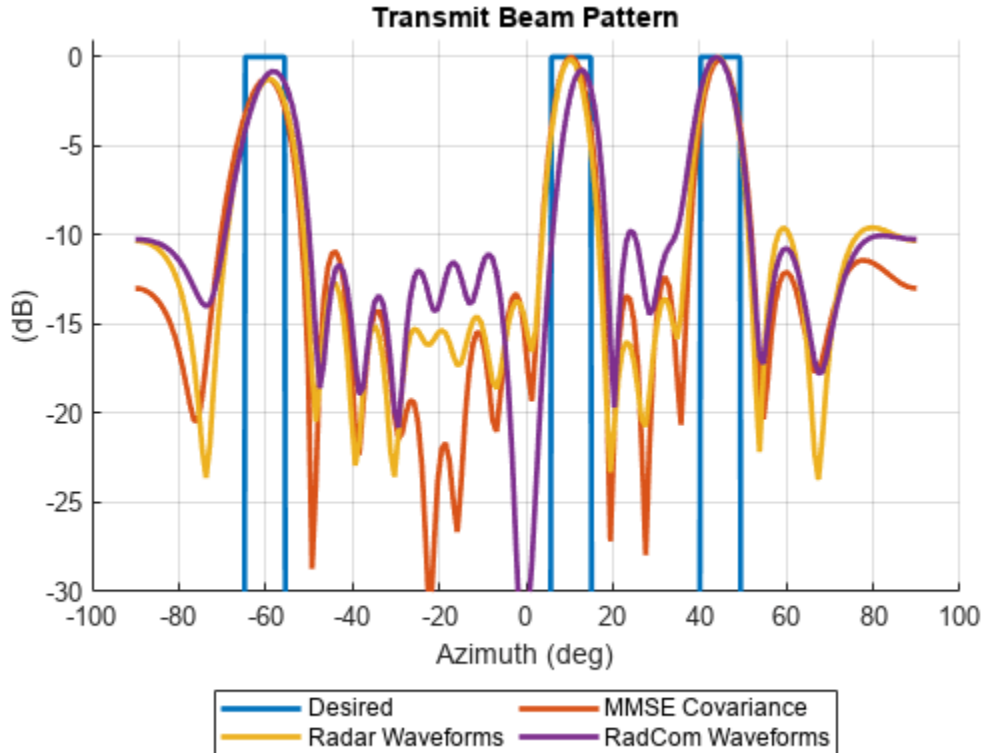


Embedding communication data into the radar waveform causes some transmit beam pattern distortion, however the main lobes are not affected in a significant way. To show the effect of MUI minimization compare the autocorrelation functions of a dual-function waveform in Xradcom and its radar counterpart from Xca.

```
[acmag_rc, delay] = ambgfun(Xradcom(n, :), 1, 1/M, "Cut", "Doppler");

figure
ax = gca;
colors = ax.ColorOrder;

hold on;
plot(delay, mag2db(acmag_ca), 'Color', colors(3, :), 'LineWidth', 2)
plot(delay, mag2db(acmag_rc), 'Color', colors(4, :), 'LineWidth', 2)
xlabel('Lag (subpulses)')
ylabel('(dB)')
title(sprintf('Autocorrelation Function for Waveform #%d', n))
grid on;
ylim([-30 0]);

legend('Radar Waveform', 'RadCom Waveform');
```

**Autocorrelation Function for Waveform #1**
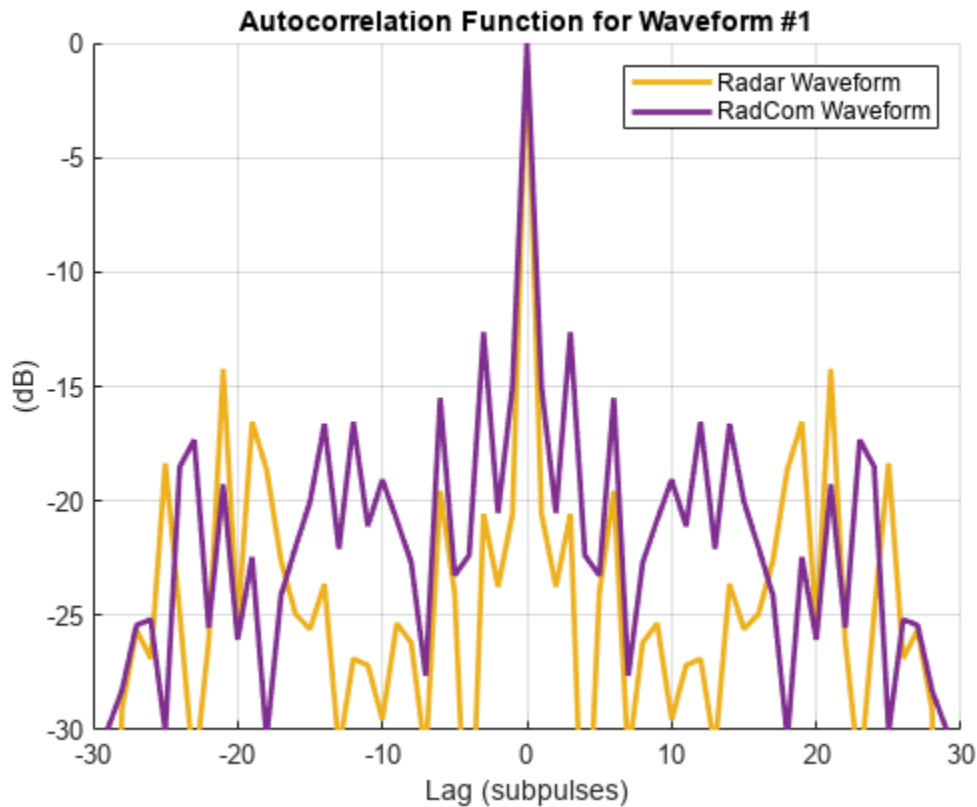
It is evident that after embedding the communication information into the radar waveform the close range sidelobes have increased. The overall peak-to-sidelobe level has not decreased significantly and is still close to 15 dB. The behavior of the sidelobes depends on the value of the parameter $\rho$. In general, increasing $\rho$ results in increased range sidelobes, since more priority is given to the communication function and less to the radar.

**Joint Radar-Communication Simulation**

You will use the designed dual-function waveforms to compute a range-angle response and a symbol error rate. Assume that the RadCom system is located at the origin. Let the duration of each subpulse be $\tau=0.25\mu$s. This means that the bandwidth of the system is $B = 1/\tau=4$ MHz. Let the pulse repetition frequency be equal to 15 KHz.

```
tau = 5e-7;                           % Subpulse duration
B = 1/tau;                            % Bandwidth
prf = 15e3;                           % PRF
fs = B;                               % Sample rate

% The RadCom system is located at the origin and is not moving
radcompos = [0; 0; 0];
radcomvel = [0; 0; 0];
```

From each antenna element of the transmit array transmit a pulse that consists of the corresponding dual-function waveform.

```
t = 0:1/fs:(1/prf);
```

```
waveform = zeros(numel(t), N);
waveform(1:M, :) = Xradcom' / sqrt(Pt/N);

transmitter = phased.Transmitter('Gain', 0, 'PeakPower', Pt/N, 'InUseOutputPort', true);
txsig = zeros(size(waveform));
for n = 1:N
    txsig(:, n) = transmitter(waveform(:, n));
end
```

Radiate the transmitted pulses and propagate them towards the targets through a free space channel.

```
% Positions of the targets of interest in Cartesian coordinates
[x, y, z] = sph2cart(deg2rad(tgtAz), zeros(size(tgtAz)), tgtRng);
tgtpos = [x; y; z];

% Assume the targets are static
tgtvel = zeros(3, numel(tgtAz));

% Calculate the target angles as seen from the transmit array
[tgtRng, tgtang] = rangeangle(tgtpos, radcompos);

radiator = phased.Radiator('Sensor', array, 'OperatingFrequency', fc, 'CombineRadiatedSignals', 
radtxsig = radiator(txsig, tgtang);

channel = phased.FreeSpace('SampleRate', fs, 'TwoWayPropagation', true, 'OperatingFrequency',fc)
radtxsig = channel(radtxsig, radcompos, tgtpos, radcomvel, tgtvel);
```

Reflect the transmitted pulses off the targets and receive the reflected echoes by the receive array.

```
% Target radar cross sections
tgtRCS = [2.7 3.1 4.3];

% Reflect pulse off targets
target = phased.RadarTarget('Model', 'Nonfluctuating', 'MeanRCS', tgtRCS, 'OperatingFrequency', 
tgtsig = target(radtxsig);

% Receive target returns at the receive array
collector = phased.Collector('Sensor', array, 'OperatingFrequency', fc);
rxsig = collector(tgtsig, tgtang);

receiver = phased.ReceiverPreamp('Gain', 0, 'NoiseFigure', 2.7, 'SampleRate', fs);
rxsig = receiver(rxsig);
```

Process the received pulses by computing the range-angle response.

```
% Compute range-angle response
rngangresp = phased.RangeAngleResponse(...
    'SensorArray',array,'OperatingFrequency',fc,...
    'SampleRate',fs,'PropagationSpeed',collector.PropagationSpeed);

resp = zeros(numel(t), rngangresp.NumAngleSamples, N);

% Apply matched filter N times. One time for each waveform. Then integrate
% the results.
for i = 1:N
    [resp(:, :, i), rng_grid, ang_grid] = rngangresp(rxsig, flipud(Xradcom(i, :).'));
end
```

**18-745**

```
% Plot the range angle response
figure
resp_int = sum(abs(resp).^2, 3);
resp_max = max(resp_int, [], 'all');
imagesc(ang_grid, rng_grid, pow2db(resp_int/resp_max))
clim([-20 1])
axis xy
xlabel('Azimuth (deg)')
ylabel('Range (m)')
title('Range-Angle Response')
cbar = colorbar;
cbar.Label.String = '(dB)';
```
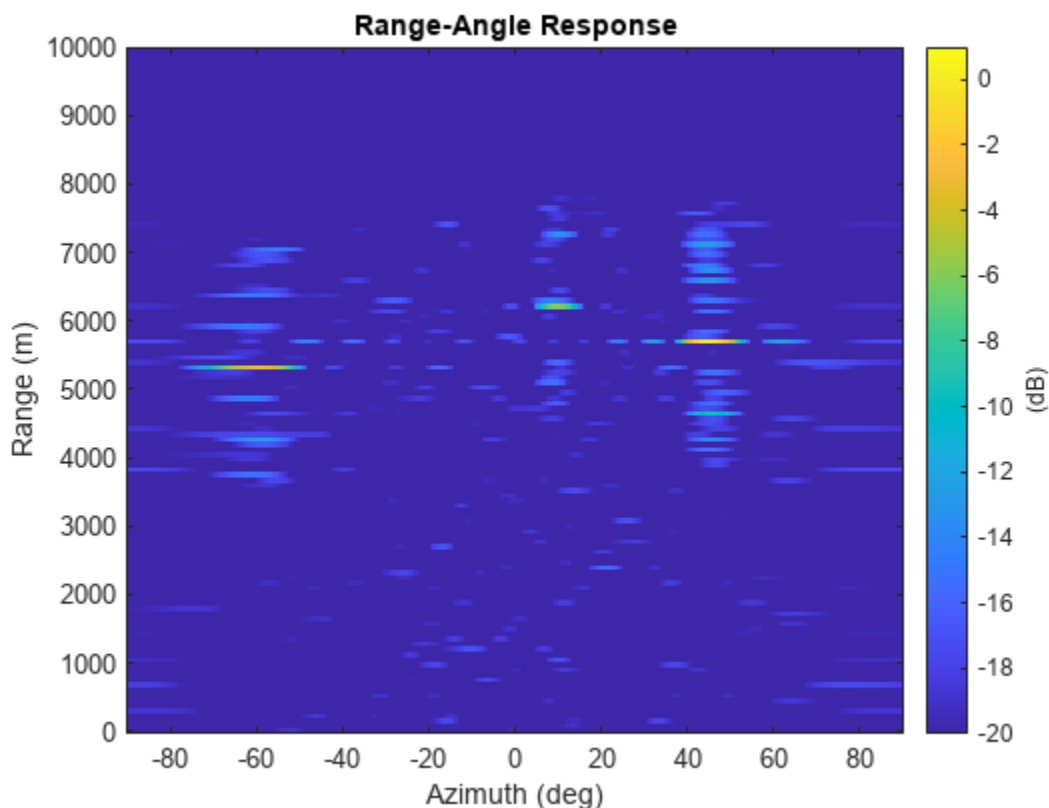


All three targets of interest are visible on the range-angle response. Now, transmit the dual-function waveforms through the MIMO scattering channel H and compute the resulting error rate.

```
rd = pskdemod(H*Xradcom, Q, pi/Q);
[numErr, errRatio] = symerr(data, rd)
```

```
numErr = 6
```

```
errRatio = 0.0500
```

The error rate is close to 4.2%. Note that the resulting performance heavily depends on the trade-off parameter $\rho$. To show this for different values of $\rho$ compute the symbol error rate, the peak-to-sidelobe level ratio of the range response (averaged over N waveforms), and the beam pattern mismatch between Xca and Xradcom.

```matlab
% Vary the radar-communication trade-off parameter from 0 to 1
rho = 0.0:0.1:1;

er = zeros(size(rho));
psl = zeros(size(rho));
bpmm = zeros(size(rho));

for i = 1:numel(rho)
    Xrc = helperRadComWaveform(H, S, Xca, Pt, rho(i));

    % Transmit through the communication channel and compute the error
    % rate
    rd = pskdemod(H*Xrc, Q, pi/Q);
    [~, er(i)] = symerr(data, rd);

    % Compute the peak to sidelobe level ratio
    psl(i) = helperAveragePSL(Xrc);

    % Compute the beam pattern
    Rrc = Xrc*Xrc'/M;
    Brc = abs(diag(A'*Rrc*A))/(4*pi);

    % Mismatch between the beam pattern produced by the radar waveforms Xca
    % and the RadCom waveforms Xrc
    bpmm(i) = sqrt(trapz(deg2rad(ang), (Bca - Brc).^2.*cosd(ang).'/(2*pi)));
end

figure;
tiledlayout(3, 1);

ax = nexttile;
plot(rho, er, 'LineWidth', 2);
grid on;
title('Symbol Error Rate');

nexttile;
plot(rho, psl, 'LineWidth', 2);
ylabel('(dB)');
grid on;
title('Average Peak to Sidelobe Level Ratio');

nexttile;
plot(rho, bpmm, 'LineWidth', 2);
grid on;
ylabel('(W)');
title('Beam Pattern Mismatch');

xlabel('\rho');
```
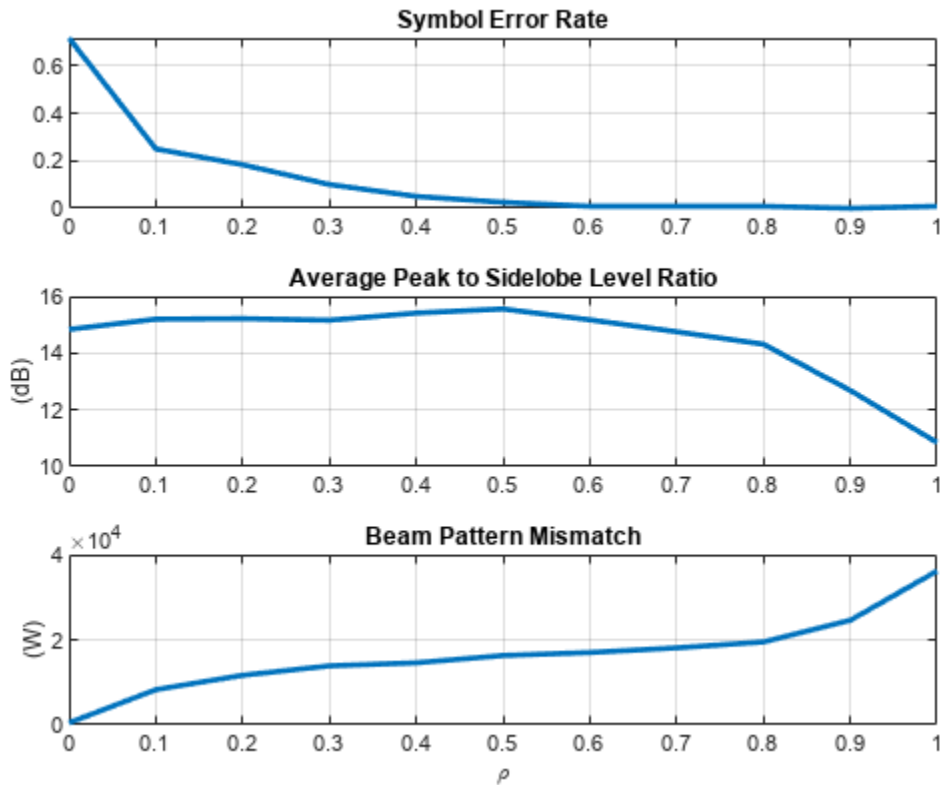
Increasing $\rho$ decreases the symbol error rate but results in the higher sidelobes and the deterioration of the transmit beam pattern.

**Conclusions**

This example shows how to design a set of dual-function waveforms for a joint MIMO RadCom system. The resulting waveforms must form a desired beam pattern to steer transmit beams towards multiple radar targets of interest and transmit a desired symbol matrix to the downlink users over a known MIMO channel. First, the example shows how to synthesize a set of radar waveforms that produce the desired transmit beam pattern. Then, the communication symbols are embedded into the synthesized radar waveforms such that the multi-user interference and the distortions in the transmit beam pattern are minimized. The example then simulates a transmission of the designed waveforms and computes the resulting range-angle response and symbol error rate.

**References**

**1** Martone, Anthony, and Moeness Amin. "A view on radar and communication systems coexistence and dual functionality in the era of spectrum sensing." *Digital Signal Processing* 119 (2021): 103135.

**2** Fuhrmann, Daniel R., and Geoffrey San Antonio. "Transmit beamforming for MIMO radar systems using signal cross-correlation." *IEEE Transactions on Aerospace and Electronic Systems* 44, no. 1 (2008): 171-186

**3** Stoica, Petre, Jian Li, and Xumin Zhu. "Waveform synthesis for diversity-based transmit beampattern design." *IEEE Transactions on Signal Processing* 56, no. 6 (2008): 2593-2598.

**4**    He, Hao, Jian Li, and Petre Stoica. *Waveform design for active sensing systems: a computational approach*. Cambridge University Press, 2012.

**5**    Liu, Fan, Longfei Zhou, Christos Masouros, Ang Li, Wu Luo, and Athina Petropulu. "Toward dual-functional radar-communication systems: Optimal waveform design." *IEEE Transactions on Signal Processing* 66, no. 16 (2018): 4264-4279.

**Supporting Functions**

```
type 'helperMMSECovariance.m'
```

```
function R = helperMMSECovariance(elPos, Pdesired, ang)
% This function computes a waveform covariance matrix that generates a
% desired transmit beam pattern. The computation is based on the squared
% error optimization described in
%
% Fuhrmann, Daniel R., and Geoffrey San Antonio. "Transmit beamforming for
% MIMO radar systems using signal cross-correlation." IEEE Transactions on
% Aerospace and Electronic Systems 44, no. 1 (2008): 171-186.
%
% elPos is a 3-by-N matrix of array element positions normalized by the
% wavelength. Pdesired is the desired beam pattern evaluated at the angles
% specified in ang.

N = size(elPos, 2);

% Initial covariance is random. x_ is a vector of all elements that are
% in the upper triangular part of the matrix above the main diagonal.
x_ = initialCovariance(N);

% Normalized the desired beam pattern such that the total transmit power is
% equal to the number of array elements N.
Pdesired = N * Pdesired / (2*pi*trapz(deg2rad(ang), Pdesired.*cosd(ang)));
Pdesired = Pdesired * 4 * pi;

% Matrix of steering vectors corresponding to angles in ang
A = steervec(elPos, [ang; zeros(size(ang))]);

% Parameters of the barrier method
mu = 4;

% The barrier term is weighted by 1/t. At each iteration t is multiplied
% by mu to decrease the contribution of the barrier function.
t = 0.02;
epsilon = 1e-1;
stopCriteriaMet = false;

J_ = squaredErrorObjective(x_, t, A, Pdesired, ang);
while ~stopCriteriaMet
    % Run Newton optimization using x_ as a starting point
    x = newtonOptimization(x_, t, A, Pdesired, ang);

    J = squaredErrorObjective(x, t, A, Pdesired, ang);

    if abs(J) < abs(J_)
        stopCriteriaMet = abs(J - J_) < epsilon;
        x_ = x;
        J_ = J;
```

```
        else
            % Increased t by too much, step back a little.
            t = t / mu;
            mu = max(mu * 0.8, 1.01);
        end

        t = t * mu;
    end

    R = constrainedCovariance(x, N);
end

function x = newtonOptimization(x_, t, A, Pdesired, ang)
    epsilon = 1e-3;

    % Parameters for Armijo rule
    s = 2;
    beta = 0.5;
    sigma = 0.1;

    stopCriteriaMet = false;
    J_ = squaredErrorObjective(x_, t, A, Pdesired, ang);

    while ~stopCriteriaMet
        [g, H] = gradientAndHessian(x_, t, A, Pdesired, ang);

        % Descent direction
        d = -(H\g);

        % Compute step size and the new value x using the Armijo rule
        m = 0;
        gamma = g'*d;
        while true
            mu = (beta^m)*s;
            x = x_ + mu*d;

            J = squaredErrorObjective(x, t, A, Pdesired, ang);

            if abs(J_) - abs(J) >= (-sigma*mu*gamma)
                x_ = x;
                stopCriteriaMet = abs(J - J_) < epsilon;
                J_ = J;
                break;
            end
            m = m + 1;
        end
    end
end

function [G, H] = gradientAndHessian(x, t, A, Pdesired, ang)
    N = size(A, 1);
    M = N*(N-1);

    F = constrainedCovariance(x, N);

    numAng = numel(ang);

    FinvFi = zeros(N, N, M);
```

```
    Alpha = zeros(M, numAng);

    idxs = find(triu(ones(N, N), 1));
    [r, c] = ind2sub([N N], idxs);

    for i = 1:M/2
        [Fi_re, Fi_im] = basisMatrix(N, r(i), c(i));

        % Matrix inverses used in Eq. (26) and (27)
        FinvFi(:, :, i) = F\Fi_re;
        FinvFi(:, :, i + M/2) = F\Fi_im;

        % Eq. (29)
        Alpha(i, :) = 2*real(conj(A(r(i), :)) .* A(c(i), :));
        Alpha(i + M/2, :) = -2*imag(conj(A(r(i), :)) .* A(c(i), :));
    end

    G = zeros(M, 1);
    H = zeros(M, M);

    D = (real(diag(A'*F*A).') - Pdesired) .* cosd(ang);

    ang_rad = deg2rad(ang);

    for i = 1:M
        % Eq. (33a)
        G(i) = -trace(squeeze(FinvFi(:, :, i))) * (1/t) + 2*trapz(ang_rad, Alpha(i, :) .* D);

        for j = i:M
            % Eq. (33b)
            H(i, j) = trace(squeeze(FinvFi(:, :, i))*squeeze(FinvFi(:, :, j))) * (1/t) ...
                + 2*trapz(ang_rad, Alpha(i, :).*Alpha(j, :).*cosd(ang));
        end
    end

    H = H + triu(H, 1)';
end

function [Fi_re, Fi_im] = basisMatrix(N, i, j)
    Fi_re = zeros(N, N);
    Fi_re(i, j) = 1;
    Fi_re(j, i) = 1;

    Fi_im = zeros(N, N);
    Fi_im(i, j) = 1i;
    Fi_im(j, i) = -1i;
end

function J = squaredErrorObjective(x, t, A, Pdesired, ang)
    % Squared error between the desired beam pattern in Pdesired and the
    % beam pattern formed by a covariance matrix defined by the vector x
    % containing the above diagonal elements
    N = size(A, 1);

    % Beam patter defined by x
    F = constrainedCovariance(x, N);
    P_ = real(diag(A'*F*A).');
```

```
    % Squared error weighted by angle
    E = abs(Pdesired - P_).^2 .* cosd(ang);

    % Total error over all angles
    J = trapz(deg2rad(ang), E);

    % Barrier function
    d = eig(F);
    if all(d >= 0)
        phi = -log(prod(d));
    else
        phi = Inf;
    end

    J = J + (1/t)*phi;
end

function F = constrainedCovariance(x, N)
    % Reconstruct the covariance matrix from a vector x of above diagonal
    % values. The diagonal elements are all equal to 1.
    Re = zeros(N, N);
    Im = zeros(N, N);
    M = numel(x);

    idxs = triu(ones(N, N), 1) == 1;
    Re(idxs) = x(1:M/2);
    Im(idxs) = x(M/2+1:end);

    F = eye(N, N);
    F = F + Re + Re.' + 1i*Im - 1i*Im.';
end

function x = initialCovariance(N)
    % Create a random covariance matrix
    X = randn(N, 10*N) + 1i*randn(N, 10*N);
    L = sum(conj(X) .* X, 2);

    X = X./sqrt(L);
    R = X*X';

    M = N*(N-1);
    x = zeros(M, 1);

    % Select the elements that are above the main diagonal
    idxs = triu(ones(N, N), 1) == 1;
    x(1:M/2) = real(R(idxs));
    x(M/2+1:end) = imag(R(idxs));
end
```

type 'helperCAWaveformSynthesis.m'

```
function X = helperCAWaveformSynthesis(R, M, rho)
% This function generates a set of waveforms that form the desired
% covariance matrix R. M is the number of waveform samples and rho is a
% parameter controlling the resulting peak-to-average power ratio (PAR).
% This algorithm is described in the chapter 14 of
%
% He, Hao, Jian Li, and Petre Stoica. Waveform design for active sensing
```

```
% systems: a computational approach. Cambridge University Press, 2012.
%
% and in
%
% Stoica, Petre, Jian Li, and Xumin Zhu. "Waveform synthesis for
% diversity-based transmit beampattern design." IEEE Transactions on Signal
% Processing 56, no. 6 (2008): 2593-2598.

N = size(R, 1);
epsilon = 0.1;

% P can control the autocorrelation level for each waveform. If P = M the
% optimization is trying to minimize sidelobes for across all M-1 lags.
P = M;

% Arbitrary semi-unitary matrix
U = (randn(P + M - 1, N*P) + 1i*randn(P + M - 1, N*P))/sqrt(2);

% Expended covariance is used to minimize the autocorrelation sidelobes. See
% eq. (14.10) and (14.11)
Rexp = kron(R, eye(P));

% Square root of the expended covariance
[L, D] = ldl(Rexp);
Rexp_sr = sqrt(D)*L';

% Cyclic algorithm
numIter = 0;
maxNumIter = 1000;
while true

    % Z has waveforms that form R but do not satisfy the PAR constraint
    % Eq. (14.6)
    Z = sqrt(M) * U * Rexp_sr;

    X = zeros(M, N);
    Xexp = zeros(P + M - 1, N*P);

    % Go through each waveform
    for n = 1:N
        % Retrieve the nth waveform from the expended matrix Z
        zn = getzn(Z, M, P, n);
        gamma = R(n, n);

        % Enforces PAR constraint using nearest vector algorithm
        xn = nearestVector(zn, gamma, rho);
        X(:, n) = xn;

        % Compute new expended waveform matrix
        for p = 1:P
            Xexp(p:p+M-1, (n-1)*P + p) = xn;
        end
    end

    % Recompute U. Eq. (5) and (11) in the paper
    [Ubar, ~, Utilde] = svd(sqrt(M)*Rexp_sr*Xexp', 'econ');
    U_ = Utilde * Ubar';
```

```
        numIter = numIter + 1;

        if norm(U-U_) < epsilon || numIter > maxNumIter
            break;
        else
            U = U_;
        end
    end

    X = X';
end

function zn = getzn(Z, M, P, n)
    zn = zeros(M, P);
    for i = 1:P
        zn(:, i) = Z(i:i+M-1, (n-1)*P + i);
    end

    zn = mean(zn, 2);
end

% Nearest vector algorithm, pages 70-71
function z = nearestVector(z, gamma, rho)
    M = numel(z);
    S = z'*z;
    z = sqrt(M*gamma/S)*z;

    beta = gamma * rho;

    if all(abs(z).^2 <= beta)
        return;
    else
        ind = true(M, 1);
        for i = 1:M
            [~, j] = max(abs(z).^2);

            z(j) = sqrt(beta)*exp(1i*angle(z(j)));

            ind(j) = false;
            S = z(ind)'*z(ind);
            z(ind) = sqrt((M-i*rho)*gamma/S)*z(ind);

            if all(abs(z(ind)).^2 <= (beta+eps))
                return;
            end
        end
    end

end

type 'helperRadComWaveform.m'

function X = helperRadComWaveform(H, S, X0, Pt, rho)
% This function implements Riemannian Conjugate Gradient (RCG) algorithm
% described in
%
% Liu, Fan, Longfei Zhou, Christos Masouros, Ang Li, Wu Luo, and Athina
% Petropulu. "Toward dual-functional radar-communication systems: Optimal
```

```
% waveform design." IEEE Transactions on Signal Processing 66, no. 16
% (2018): 4264-4279.
%
% H is a K-by-N channel matrix where K is the number of down-link users
% and N is the number of antennas at the RadCom base station. S is a K-by-L
% symbol matrix, where L is the number of symbols. X0 is a N-by-L matrix of
% waveforms that generate the desired radar beam pattern. Pt is total
% transmit power, and rho is a trade-off parameter to balance the radar and
% communication performance.

delta = 1e-6;
maxNumIterations = 500;

[N, M] = size(X0);
S = S * sqrt(Pt/size(S, 1));

A = [sqrt(rho)*H; sqrt(1-rho)*eye(N)];
B = [sqrt(rho)*S; sqrt(1-rho)*X0];

X = (randn(N, M) + 1i*randn(N, M))/sqrt(2);
X = Rx(X, 0, Pt);

% Euclidean gradient
dF = 2 * A' * (A * X - B);

% Projection of the gradient onto the tangent space at X
gradF = Px(X, dF, Pt);

% Descent direction
G = -gradF;

% Armijo rule parameters
s = 1;
beta = 0.5;
sigma = 0.01;

k = 1;
while true
    fx = norm(A*X-B, 'fro').^2;

    % Iterate until the maximum number of iterations is reached or the norm
    % of the gradient is less than delta
    if k > maxNumIterations || norm(gradF, 'fro') < delta
        break
    end

    % If the inner product of the gradient and the descent direction is
    % positive reset the descent direction and make it equal to the
    % negative gradient
    gamma = real(trace(gradF*G'));
    if gamma > 0
        G = -gradF;
        gamma = real(trace(gradF*G'));
    end

    % Step size and new X computation using Armijo rule
    m = 0;
    mu = 1;
```

```
    while (-sigma*mu*gamma) > 0
        mu = (beta^m)*s;
        X_ = Rx(X, mu*G, Pt);

        fx_ = norm(A*X_ - B, 'fro').^2;

        if (fx(1) - fx_(1)) >= (-sigma*mu*gamma)
            X = X_;
            break;
        end
        m = m + 1;
    end

    % Previous projected gradient
    gradF_ = gradF;

    % Previous descent direction
    G_ = G;

    % Euclidean gradient at X
    dF = 2 * A' * (A * X - B);

    % Projection of the gradient onto the tangent space at X
    gradF = Px(X, dF, Pt);

    % Polak-Ribiere combination coefficient
    tau = real(trace(gradF'* (gradF -  Px(X, gradF_, Pt)))) / real(trace(gradF_'*gradF_));

    % New descent direction is a combination of the current negative
    % gradient and the previous gradient translated into the current
    % tangent space
    G = -gradF + tau *  Px(X, G_, Pt);

    k = k + 1;
end

end

% Tangent space projector
function PxZ = Px(X, Z, Pt)
    [N, M] = size(X);
    d = diag(Z*X');
    PxZ = Z - diag(d)*X*(N/(M*Pt));
end

% Retractor mapping
function RxZ = Rx(X, Z, Pt)
    [N, M] = size(X);
    Y = X + Z;
    d = diag(Y*Y').^(-1/2);
    RxZ = sqrt(M * Pt / N)*diag(d) * Y;
end

type 'helperAveragePSL.m'

function avpsl = helperAveragePSL(X)
    [n, m] = size(X);
    psl = zeros(n, 1);
```

```
for i = 1:n
    acmag = ambgfun(X(i, :), 1, 1/m, "Cut", "Doppler");
    psl(i) = acmag(m)/max([acmag(1:m-1) acmag(m+1:end)]);
end

avpsl = mag2db(mean(psl));
end
```

# Spectrum Sharing using Spectrum Sensing and Waveform Notching

This example illustrates the implementation and analysis of a technique for reducing mutual interference between non-cooperative radar and communication systems by inserting notches into the transmitted radar waveform. The frequency spectrum is analyzed by the radar system to determine regions occupied by a communications waveform, and a notched radar waveform that minimizes the power contained in those occupied regions is generated. This type of spectrum sharing approach enables coexistence of radar and communications systems operating in the same region of the frequency spectrum.

### Introduction

Coexistence between radar and communications systems continues to be an active area of research as the frequency spectrums that these systems operate in become more crowded, particularly with the proliferation of 5G communications networks. Non-cooperative coexistence is an approach in which a radar and communication system are operating in the same frequency spectrum independently of one another and without any data exchange between systems.

The coexistence strategy explored in this example involves a radar and communications system operating in the same frequency band (a 100 MHz band around 3.5 GHz). The spectrum sharing behavior is realized by a cognitive radar system that adapts its waveform to avoid spectral overlap with the communications signal. The communications system operates freely and does not vary it's transmission scheme based on the presence of the radar. Meanwhile, the radar detects the spectral location of the communications signal and inserts frequency nulls (referred to as "notches") into it's transmit waveform, thereby reducing the mutual interference between the systems.

In this example, we focus on the signal level implementation of a spectrum sensing and waveform generation technique that make up the basis for this proposed coexistence strategy. Specifically, we implement the following two step process for generating a radar waveform with notches in the portion of the frequency spectrum occupied by the communications signal [1]:

1  Determine the optimal frequency region for radar waveform transmission in the presence of an Orthogonal Frequency-Division Multiplexing (OFDM) communications waveform based on a Fast Spectrum Sensing (FSS) algorithm [5].

2  Create and analyze a Pseudo-Random Optimized Frequency Modulation (PRO-FM) pulsed waveform with notches (areas of reduced spectrum power) in the region occupied by the communications waveform [2].

Once the process for spectrum sensing and waveform generation is examined, a simulation is run to determine the performance benefit of using a notched radar waveform in the presence of a communications signal.

### Spectrum Sensing

In the coexistence scheme described in this example, a radar system minimizes the impact that it's transmit waveform has on a communication system while simultaneously improving radar performance by notching the radar transmit waveform. In order for such a coexistence scheme to be effective, the radar system must continuously and quickly (on the order of a pulse repetition interval) determine an unoccupied region of the spectrum for transmission.

A popular FSS algorithm is implemented in this example [5]. This approach groups the spectrum into regions of high and low power interference plus noise, and then selects the transmit region by

maximizing an objective function that balances the trade-off between the total bandwidth of the radar waveform with the anticipated Signal to Interference plus Noise Ratio (SINR) based on measurements of the spectrum that contains a communication waveform.

**Generate Communications Signal**

The communications signal occupying the frequency spectrum is made up of two non-overlapping OFDM transmissions. Each of these OFDM transmissions contains four carriers separated by 1 MHz (occupying a total of 4 MHz of bandwidth).

```matlab
fc = 3.5e9;                           % Communications center frequency
separationFrequency = 1e6;            % 1 MHz carrier separation frequency
numCarriers = 4;                      % 4 carriers with 1 MHz separation
ofdmLocations = [fc-25e6;fc+25e6];    % One OFDM transmission at center frequency - 25 MHz, one
sampleRate = 200e6;                   % 200 MHz sample rate
numSamples = 400;                     % Collect 400 communications samples

% Helper used to generate communications signal
commsSignal = helperGenerateCommunicationsSignal(sampleRate,numCarriers,numSamples,separationFreq
```

Once the communications signal has been generated, calculate and plot it's power spectrum.

```matlab
% Get the communications signal power spectrum
pComms = pspectrum(commsSignal,sampleRate);
pCommsDb = pow2db(pComms);

% Get the frequency samples of communications spectrum
fComms = helperSpectrumFrequencies(sampleRate,numel(pComms));

% Plot the power spectrum of the communications signal
plot(fComms/10^6,pCommsDb,'LineWidth',1);
xlabel('Frequency (MHz)'); ylabel('Power Spectrum (dB)');
title('Power Spectrum of Communications Signal');
```
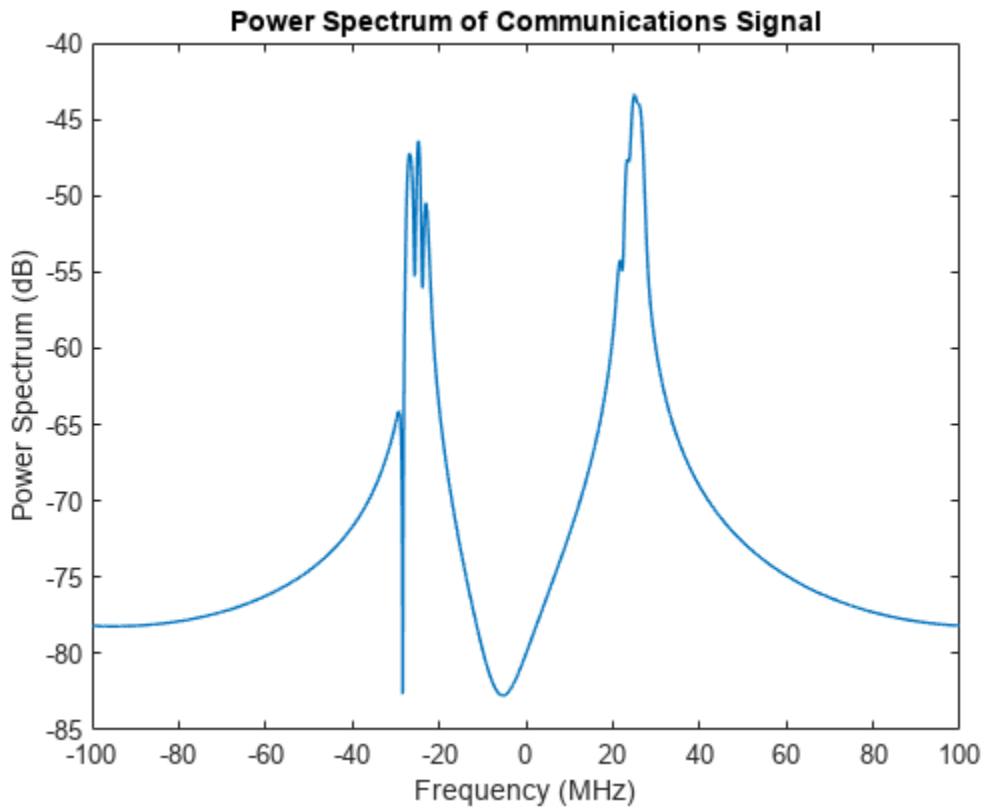
The power spectrum of the communications signal shows two occupied frequency bands. The ideal radar waveform in the coexistence scheme described in this example would contain very little power in the regions of the frequency spectrum where the communications signals are transmitted.

**Separate Spectrum Into High and Low Power Meso-Bands**

The first step in the FSS algorithm is to divide the spectrum into high and low power interference plus noise regions. These regions are referred to as "meso-bands". No noise is added in this example, so in this case the spectrum power is solely due to the presence of the communications signal. The algorithm for splitting the spectrum into high and low power meso-bands is:

**1**    Create an alternating set of high and low power interference plus noise regions called meso-bands by comparing each sample to a user specified power threshold.

**2**    For each low power region, check if it is narrower than a user specified minimum meso-band bandwidth. If a low power region has a width lower than the user specified value, merge it with the adjacent high power bands.

First, set up the power threshold and minimum bandwidth based on the values used in [1].

```
pCommsDbMax = max(pCommsDb);      % Get the maximum communications signal power level
pThresholdDb = pCommsDbMax-15;    % User specified power threshold is set to be 15 dB below the m
minBandwidth = 4e6;               % Minimum meso-band bandwidth is set to be 4 MHz
```

A helper function is used to separate the spectrum into high and low power meso-bands using the algorithm described above.

```
mesoBands = helperMesoBands(pComms,pThresholdDb,minBandwidth,sampleRate);
```

A plot of meso-bands shows that the frequency spectrum is separated into only 5 high and low power bands. This reduces the computational complexity of determining the optimal transmission region. Grouping the frequency spectrum into high and low power meso-bands using the user specified power threshold and minimum bandwidth heuristics is what makes this spectrum sensing algorithm computationally efficient.

```
a = helperPlotMesoBands(pComms,mesoBands,pThresholdDb,fComms,"Merged Meso-Bands",[-50 50]);
```
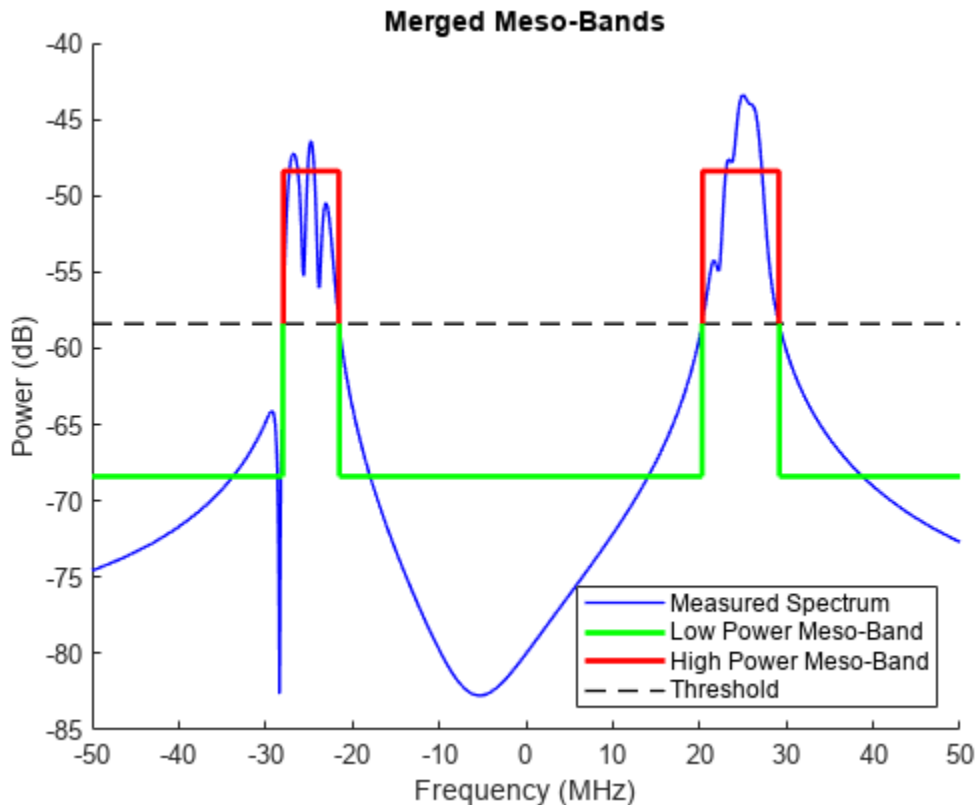


**Select Meso-Band Combination that Maximizes Objective Function**

The goal of this spectrum sensing algorithm is to transmit in the frequency range that maximizes the following objective function [5]:

$$\text{Objective} = \alpha * \overline{\text{SINR}} + (1 - \alpha) * \overline{\text{Bandwidth}}$$

where $\overline{\text{SINR}}$ and $\overline{\text{Bandwidth}}$ are the normalized SINR and bandwidth, and $\alpha$ is a user specified weighting term that balances the tradeoff between increased bandwidth and increased SINR. The following equation is used to calculate SINR for a combination of meso-bands:

$$\text{SINR} = \frac{\tau * P_s}{P_{\text{IN}}}$$

where $\tau$ is the pulse width as a fraction of the pulse repetition interval, $P_s$ is the received signal power in a given meso-band based on the desired spectrum, and $P_{IN}$ is the received interference plus noise power in one pulse width in a given meso-band.

This objective function is calculated for every meso-band combination, and the combination that maximizes the objective function is selected as the radar waveform transmission region. In this case, there are only 5 meso-bands, so there are only $2^5$-1=31 combinations of bands to calculate the objective value for.

For the purposes of this example, assume that the received radar signal power is 20 dB below the peak of the received communications signal power, the radar waveform pulse width is 5% of the pulse repetition interval (PRI), and the alpha value is set to 0.5.

```
peakSinrDb = -20;      % SINR of peak communications signal power vs. peak radar signal power
tau = 0.05;            % Pulse width as a fraction of pulse repitition interval
alpha = 0.5;           % Weighting factor between SINR and bandwidth
```

In order to determine the optimal transmission region, the desired waveform spectrum power is assumed to have a Gaussian shape. This spectral shape is desirable for radar waveforms because it results in low range sidelobes. This power spectrum is scaled to maintain the specified SINR.

```
% The desired waveform spectrum power has a Gaussian shape.
pWaveform = gausswin(numSamples);
pWaveformDb = pow2db(pWaveform);

% Scale the spectrum power for desired SINR
pWaveformScaledDb = pWaveformDb + (pCommsDbMax-max(pWaveformDb)) + peakSinrDb;
pWaveformScaled = db2pow(pWaveformScaledDb);
```

Plot the Gaussian radar waveform spectrum power along with the communications signal spectrum power used to calculate the objective function for all meso-band combinations.

```
% Plot the communications signal spectrum power along with the desired radar waveform
% spectrum power
helperPlotCommsWithRadarSpectrum(pComms,pWaveformScaled,'Full Radar Waveform',sampleRate,'Radar W
```
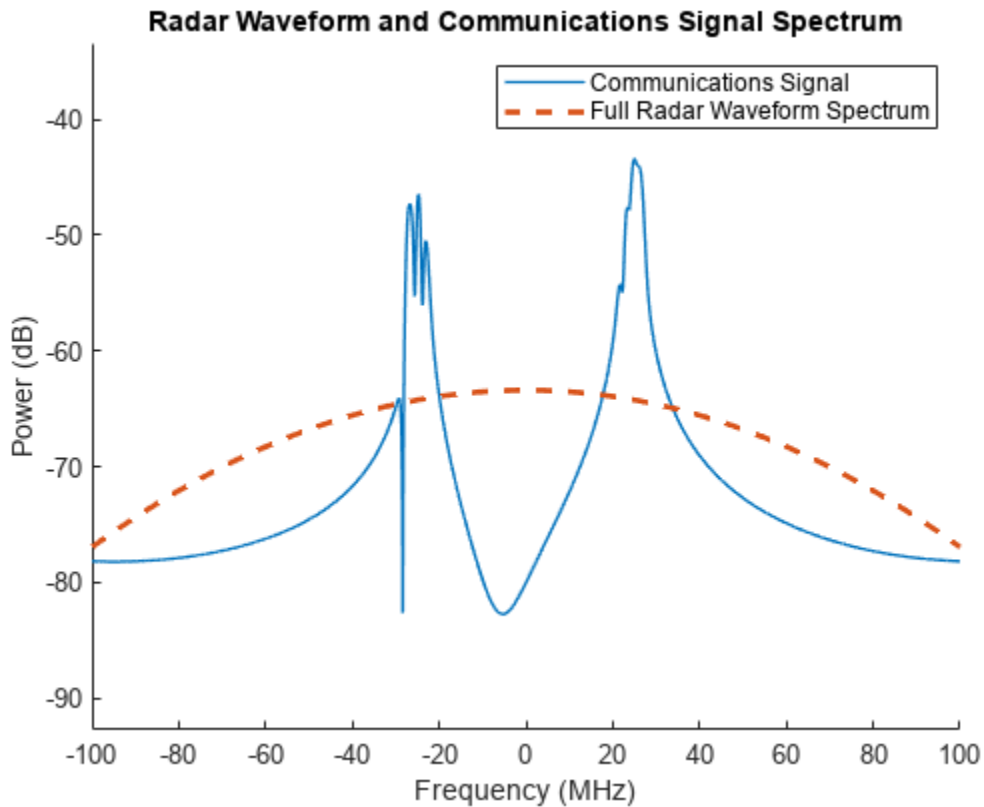
This plot shows that by using the full spectrum for radar transmission, there would be mutual interference between the radar waveform and the communications waveform, resulting in performance degradation for both systems.

Next, select the transmission region that maximizes the objective function. The notch frequencies are the frequencies excluded from the transmission region.

```
% Calculate the desired location of frequency notches in the waveform
notchFrequencies = helperTxNotchFrequencies(mesoBands,pWaveformScaled,pComms,tau,alpha,sampleRate
```

Once the location of the frequency notches are determined, a desired waveform spectrum can be generated by inserting notches into the full waveform spectrum. Plot the desired notched radar waveform spectrum power along with the communications signal spectrum power.

```
% Generate notched spectrum
desiredRadarSpectrum = helperNotchSpectrum(pWaveformScaled,sampleRate,notchFrequencies);

% Plot communications signal spectrum along with the radar waveform spectrum desired
% for mutual interference mitigation
helperPlotCommsWithRadarSpectrum(pComms,desiredRadarSpectrum,'Desired Radar Waveform',sampleRate
```
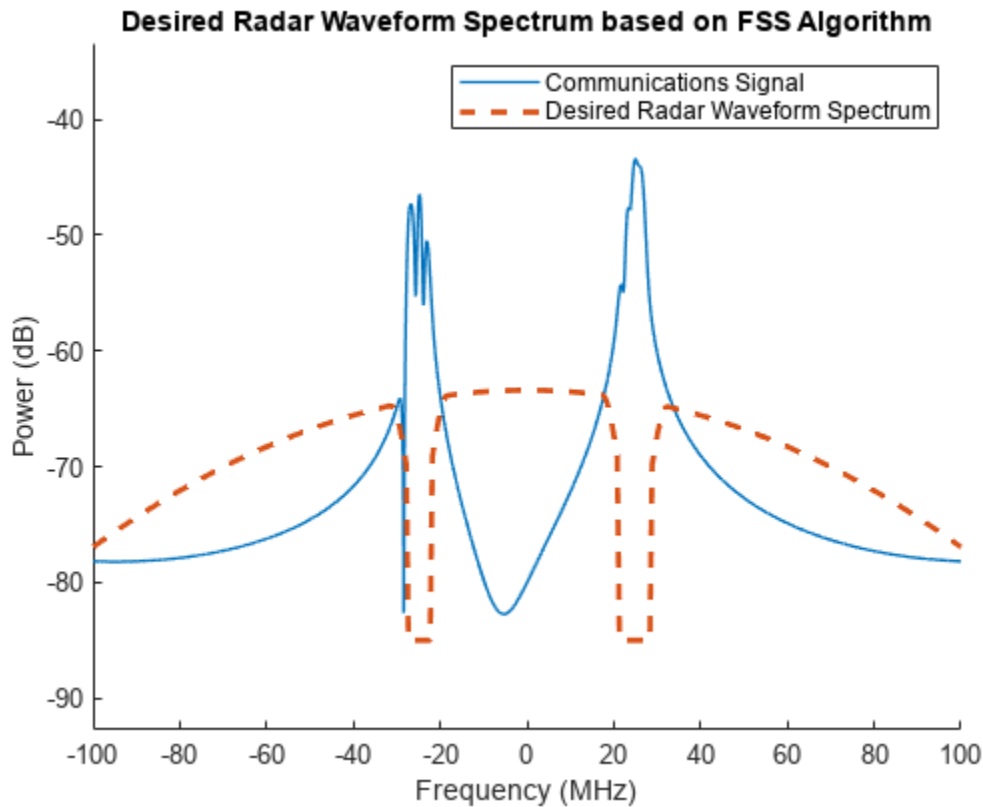
This plot illustrates that using this FSS algorithm, a desired radar waveform spectrum is generated that has notches in the region of highest communications signal power. A radar waveform with this spectrum would substantially reduce mutual interference between the radar and communications systems.

**Notched Waveform Design and Analysis**

After the desired spectrum is determined based on the FSS algorithm, a waveform must be generated that conforms to this spectrum. In this example, we use a PRO-FM waveform [1]. This is a noise waveform that contains desirable range sidelobe characteristics and is amenable to spectral notching due to the nature of its construction. These characteristics make this a versatile waveform that is a good candidate for spectrum sharing. To generate this waveform, a seed waveform is optimized iteratively to contain a desired frequency spectrum shape and uniform amplitude [2].

**Generate Initial PCFM Waveform**

In the process of generating a PRO-FM waveform, you start with a seed waveform made up of data symbols with random phase. The frequency spectrum of this waveform is dictated by the symbol rate. This seed waveform is a Polyphase-Coded FM (PCFM) waveform that has random frequency values within the desired frequency band throughout the duration of the pulse [3]. This type of waveform has been shown to be amenable to spectral shaping via the PRO-FM process [2].

The transmit waveform has a design bandwidth of 100 MHz, a pulse width of 2 us, a pulse repetition interval of 80 us, and a coherent processing interval (CPI) with 200 pulses.

```
rng("default")          % Set random number generator for reproducibility
bandwidth = 100e6;      % Design bandwidth is 100 MHz
```

```
pulseWidth = 2e-6;          % Pulse width is 2 us
pri = 80e-6;                % PRI is 80 us
numPulses = 200;            % 200 pulses in a CPI

% Sample at 2x our design bandwidth
sampleRate = 2*bandwidth;
```

The following steps describe the process of generating the PCFM waveform with random frequencies throughout the pulse duration.

**1**    Randomly generate N+1 symbol phases $\theta$, where $\theta \in [-\pi \ \pi]$

**2**    Calculate N phases differences $\alpha_n$, where $\alpha_n = \theta_n - \theta_{n-1}$ for n = 1,...,N. If $|\alpha_n| > \pi$, $\alpha_n = \alpha_n - 2\pi * \mathrm{sgn}(\alpha_n)$. In this equation, sgn is the sign function.

**3**    Create an impulse train of phase differences using the equation $\sum_{n=1}^{N} \alpha_n * \delta(t - (n-1)T_p)$, where $T_p$ is the time separation between impulses and $\delta(t)$ is a unit impulse at time t. This impulse train carries the random frequency information of the desired signal.

**4**    Filter the impulse train to create smooth frequency transitions between samples. In this case, we use a raised-cosine filter.

**5**    Integrate the filtered impulse train F(t) from 0 to t to recover the desired signal phase values. Assume $\theta_0$ as the initial condition: $P(t) = \int_0^t F(t)d\tau + \theta_0$

**6**    Recover the complex signal from the phase values: $s(t) = e^{\mathrm{i}P(t)}$, where s(t) is the final PCFM waveform.

The following code generates the seed waveform using the algorithm described above, and plots the waveform characteristics.

```
% Generate the PCFM seed waveform
pcfmWaveform = helperSeedWaveform(bandwidth,pulseWidth,sampleRate,numPulses);

% Plot the waveform spectrogram for a single pulse
stft(pcfmWaveform(:,1),sampleRate); title('PCFM Seed Waveform Spectrogram');
```
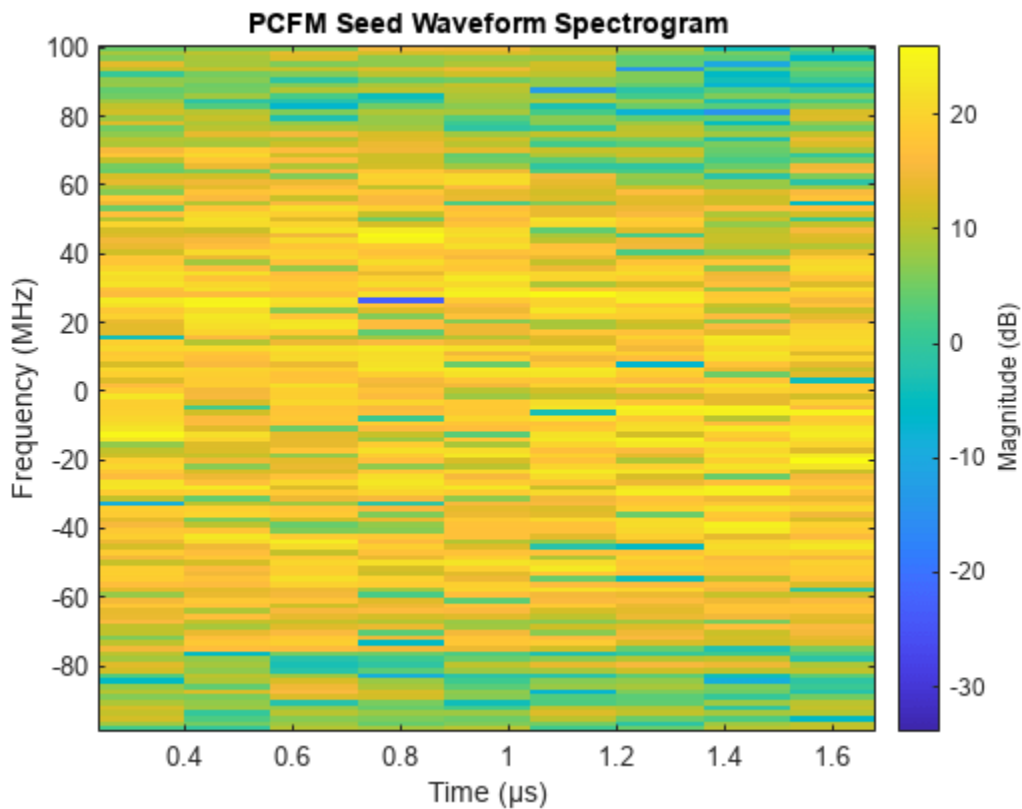
The spectrogram of the PCFM signal shows that the frequency is randomly distributed as a function of time.

```
% Plot the seed waveform power spectrum averaged across pulses
pSeed = pspectrum(pcfmWaveform,sampleRate);
pSeedMean = mean(pSeed,2);
fSeed = helperSpectrumFrequencies(sampleRate,numel(pSeedMean));
plot(fSeed/10^6,pow2db(pSeedMean)); xlabel('Frequency (MHz)'); ylabel('Power Spectrum (dB)'); ti
```
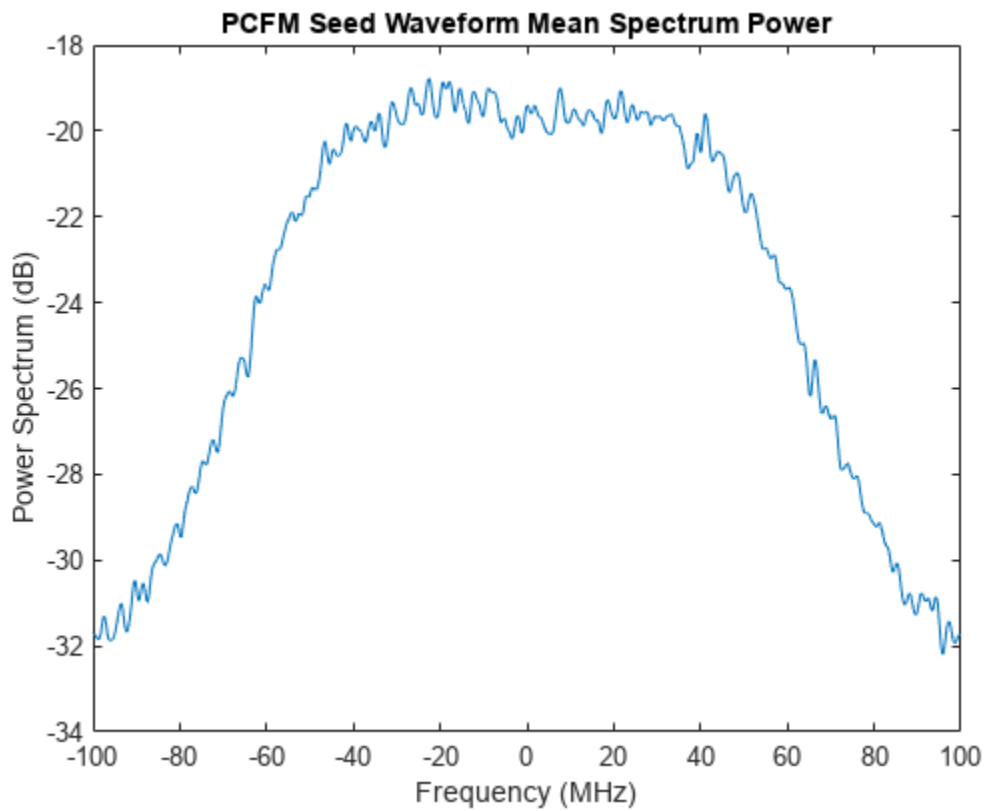
The power spectrum clearly illustrates that more power is concentrated towards the center of the spectrum, although the spectral shape is not Gaussian.

```
% Plot the ambiguity function for a single pulse
ambgfun(pcfmWaveform(:,1),sampleRate,1/pulseWidth); title('PCFM Seed Waveform Ambiguity Function
```
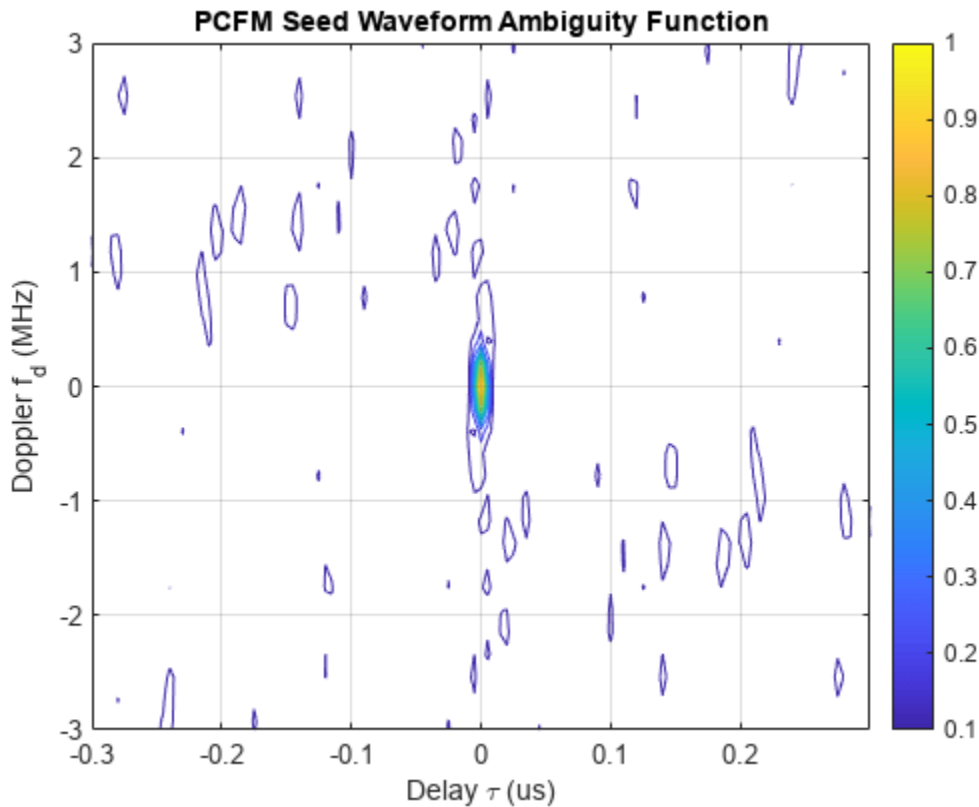
Some Range-Doppler sidelobes are visible in the ambiguity function and appear to be distributed randomly, which is not unexpected due to the nature of the waveform's construction.

**Generate PRO-FM Notched Waveform**

The generation of the PRO-FM waveform is an iterative process where a desired frequency spectrum mask is repeatedly applied to the seed waveform signal in the frequency domain and then converted back to the time domain while maintaining constant amplitude. The following steps describe the generation of the PRO-FM waveform:

**1**    Convert the previous constant amplitude time domain signal ($s_k$) to the next time domain signal ($r_{k+1}$). First convert to the frequency domain using the Fast Fourier Transform (FFT), extract phase ($\angle$), apply desired spectrum ($g$), and perform the Inverse Fast Fourier Transform (IFFT):

$$r_{k+1} = \text{ifft}\left(g * e^{i\angle\text{fft}(s_k)}\right)$$

**2**    Enforce constant amplitude of time domain signal: $s_{k+1} = e^{i\angle r_{k+1}}$

**3**    Repeat steps (1) and (2) for K iterations

K is set to 500 for this example [1]. In practice, the number of iterations would be empirically determined or the iterative process could continue until the waveform characteristics meet some stopping criteria. The desired spectrum ($g$) can take on any shape depending on the requirements of the radar system. In this case, the notched Gaussian spectrum determined using the FSS algorithm from the previous section is used to generate the PRO-FM waveform.

```
iterations = 500;                              % Go through 500 waveform iterations
notchedSpectrum = desiredRadarSpectrum;        % The desired spectrum is the notched spectrum from
```

The algorithm described above is used to generate the notched waveform PRO-FM waveform, and the waveform characteristics are plotted.

```
% Generate a PRO-FM waveform with a notched frequency spectrum
proFmNotched = helperProFm(pcfmWaveform,notchedSpectrum,iterations);

% Plot the waveform spectrogram for a single pulse
stft(proFmNotched(:,1),sampleRate); title('PRO-FM Notched Waveform Spectrogram');
```
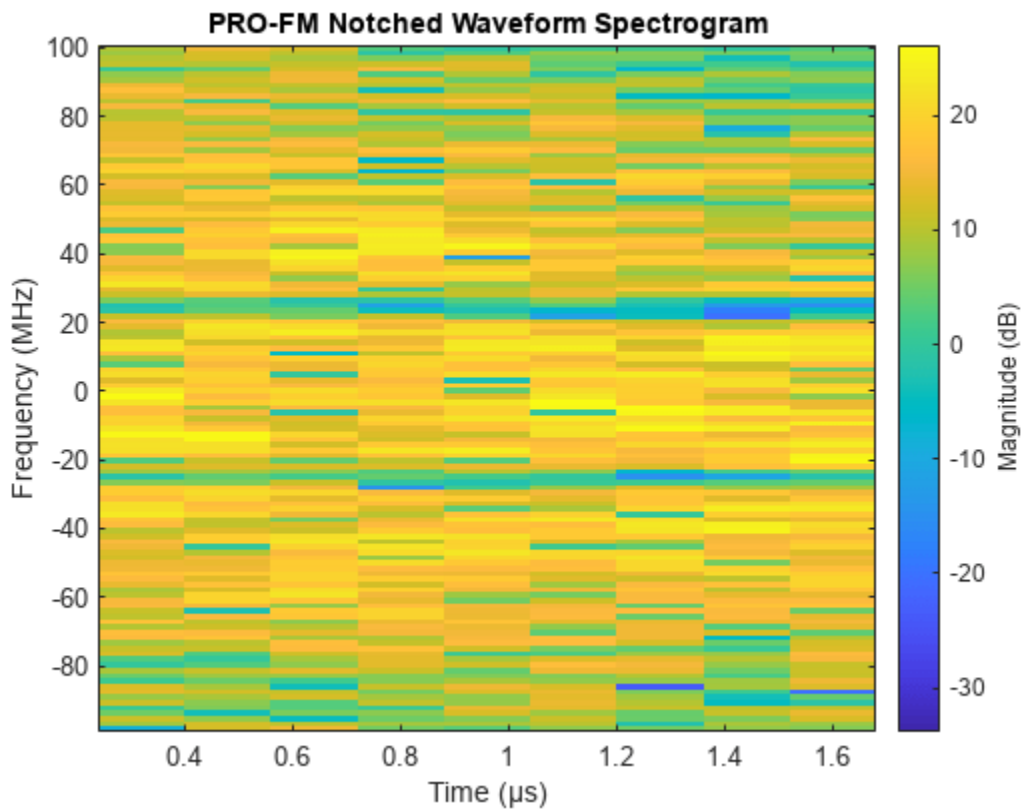


The PRO-FM spectrogram still shows a randomly varying frequency throughout the duration of the pulse, although the power is more concentrated towards the center of the distribution and there are regions of low power where the frequency spectrum notches are.

```
% Plot the PRO-FM waveform power spectrum averaged across pulses
pProFm = pspectrum(proFmNotched,sampleRate);
pProFmMean = mean(pProFm,2);
fProFm = helperSpectrumFrequencies(sampleRate,numel(pProFmMean));
plot(fProFm/10^6,pow2db(pProFmMean)); xlabel('Frequency (MHz)'); ylabel('Power Spectrum (dB)'); t
```
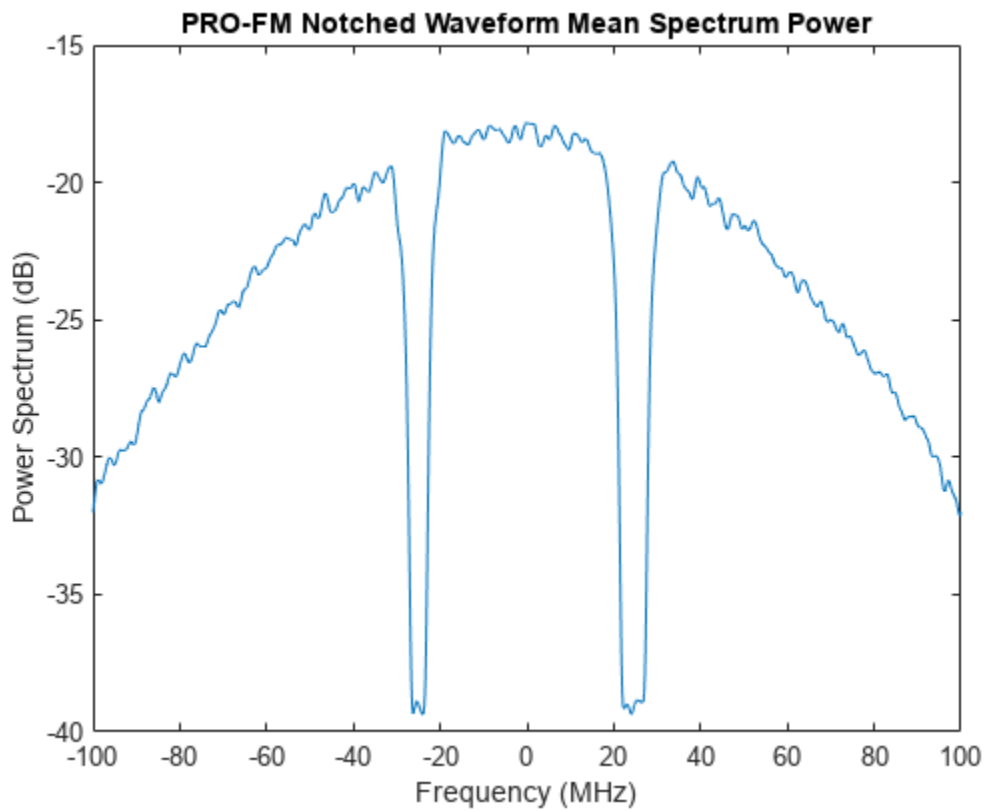
The power spectrum clearly illustrates the Gaussian shape of the spectrum and the presence of notches at the locations determined by FSS.

```
% Plot the ambiguity function for a single pulse
ambgfun(proFmNotched(:,1),sampleRate,1/pulseWidth); title('PRO-FM Notched Waveform Ambiguity Func
```
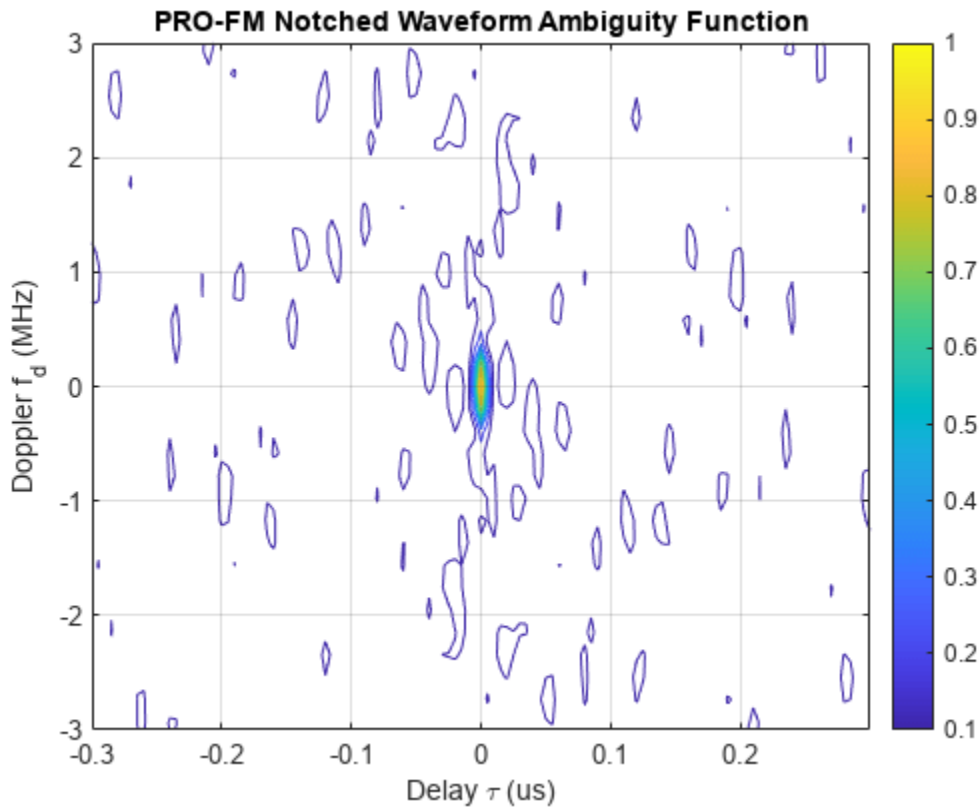
The ambiguity function shows the presence of significant Range-Doppler sidelobes. The issue of sidelobes in the waveform ambiguity function is largely mitigated by altering the transmit waveform in a CPI on a pulse to pulse basis. Using this variable waveform approach, the location of these Range-Doppler sidelobes change throughout a CPI and their impact on performance is significantly reduced [1].

**Impact of Notches on Radar Performance**

The previous two sections in this example showed how to determine the waveform transmission region using an FSS algorithm and how to subsequently generate a PRO-FM notched waveform using the desired transmission spectrum obtained from the FSS algorithm.

This section will analyze the impact that waveform notching has on radar performance by simulating the transmission of the notched vs. unnotched waveforms in the presence of the communications signal.

The communications signal is assumed to occupy a fixed region for the duration of a radar CPI for simplicity. This example could be adjusted so that the location of the OFDM transmission changes throughout the CPI, but for this example only the simplest case is examined. The communications signal generated in the prior section is reused here.

**Generate Unnotched and Notched Waveforms**

In order to analyze the impact of waveform notching on radar performance, we generate unnotched and notched PRO-FM waveforms. A new waveform is generated for each pulse in the CPI. The notched waveform has notches placed at the frequencies determined using the FSS algorithm.

```
% Generate notched and unnotched waveforms for a 200 pulse CPI.
unnotchedWaveformCpi = helperGenerateUnnotchedWaveform(bandwidth,pulseWidth,sampleRate,numPulses
notchedWaveformCpi = helperGenerateNotchedWaveform(bandwidth,pulseWidth,sampleRate,numPulses,notc
```

Plot the unnotched and notched waveform spectrums alongside the communications signal spectrum
to visualize the spectrum of the waveforms that are being used for transmission in this section.

```
% Plot communications signal as well as notched and unnotched waveform
helperPlotCommsWithRadar(commsSignal,unnotchedWaveformCpi,notchedWaveformCpi,peakSinrDb,sampleRat
```
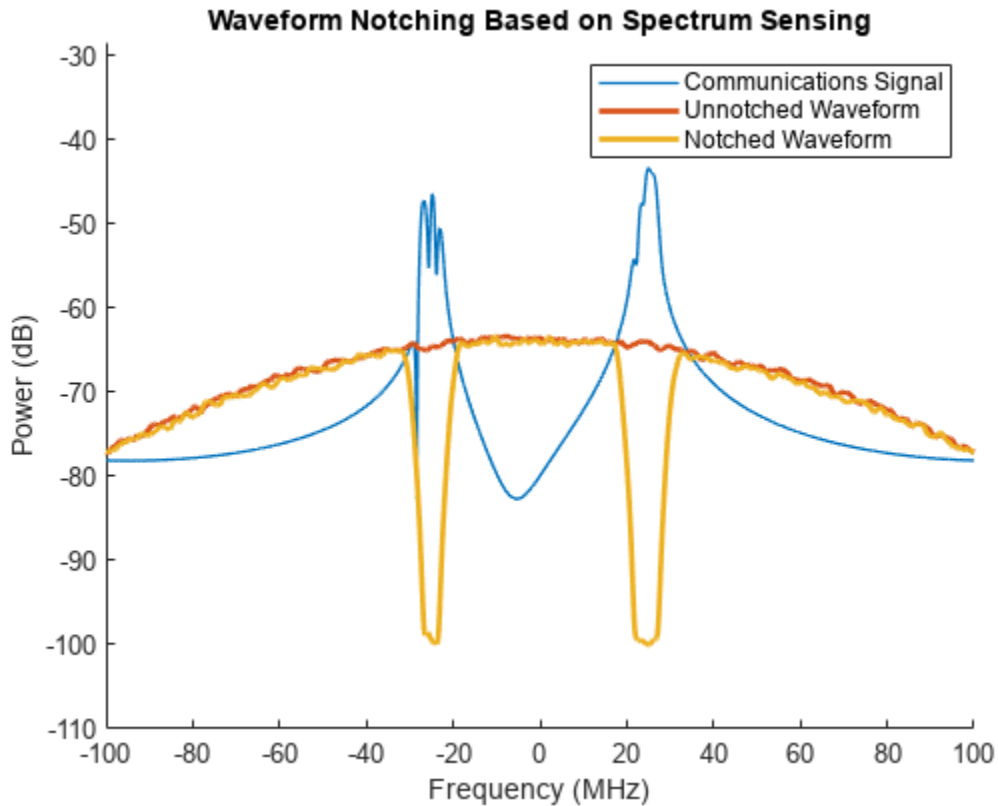


**Simulate Transmission of Waveforms and Analyze Performance**

In this section, we run a simulation to analyze the performance of the unnotched and notched
waveforms in the absence and presence of the communications signal.

Prior to running this simulation, we create a set of transmit radar waveforms combined with
communications waveforms. An alternative approach would be to add the communications signal to
the entire receive pulse instead of just the transmit waveform. In this case, the communications
signal is added only to the transmit waveform for simplicity to easily maintain the desired SINR on
receive. Prior to combining the radar waveform with the communications waveform, this function
scales the signals so that the SINR is set to the desired level.

```
% Add communications signal to waveforms with desired SINR
notchedWaveformCpiWithComms = helperCombineSignals(notchedWaveformCpi,commsSignal,peakSinrDb);
unnotchedWaveformCpiWithComms = helperCombineSignals(unnotchedWaveformCpi,commsSignal,peakSinrDb)
```

Once the radar waveforms have been generated and combined with the interfering communications waveform, set up target locations and velocities. Use 6 targets total with some moving towards the platform and some moving away from the platform to simulate a roadway.

```
% Set up target ranges and velocities
targetPositions = [[500;0;0],[500;0;0],[600;0;0],...
                   [600;0;0],[700;0;0],[700;0;0]];

targetVelocities = [[5;0;0],[-5;0;0],[10;0;0],...
                    [-10;0;0],[15;0;0],[-15;0;0]];

% Velocity and range limits for the Range-Doppler Plot
rLim = [450 750];
vLim = [-30 30];
```

After generating the waveforms and setting up the target properties, run the simulation of a single CPI and plot the Range-Doppler response of the unnotched and notched waveforms in the absence and presence of interference. For more information on simulating the transmission of custom waveforms, see "Waveform Design to Improve Range Performance of an Existing System" on page 18-178.

The first two plots show the Range-Doppler map for the notched and unnotched waveforms in the absence of the communications signal.

```
% Run simulation and plot Range-Doppler response for unnotched waveform
% without communications signal
[rdUnnotchedNoComms,rdRange,rdVelocity] = helperRunSimulation(unnotchedWaveformCpi,unnotchedWave
```



Range-Doppler Map:
Unnotched Radar Waveform, No Comms Signal
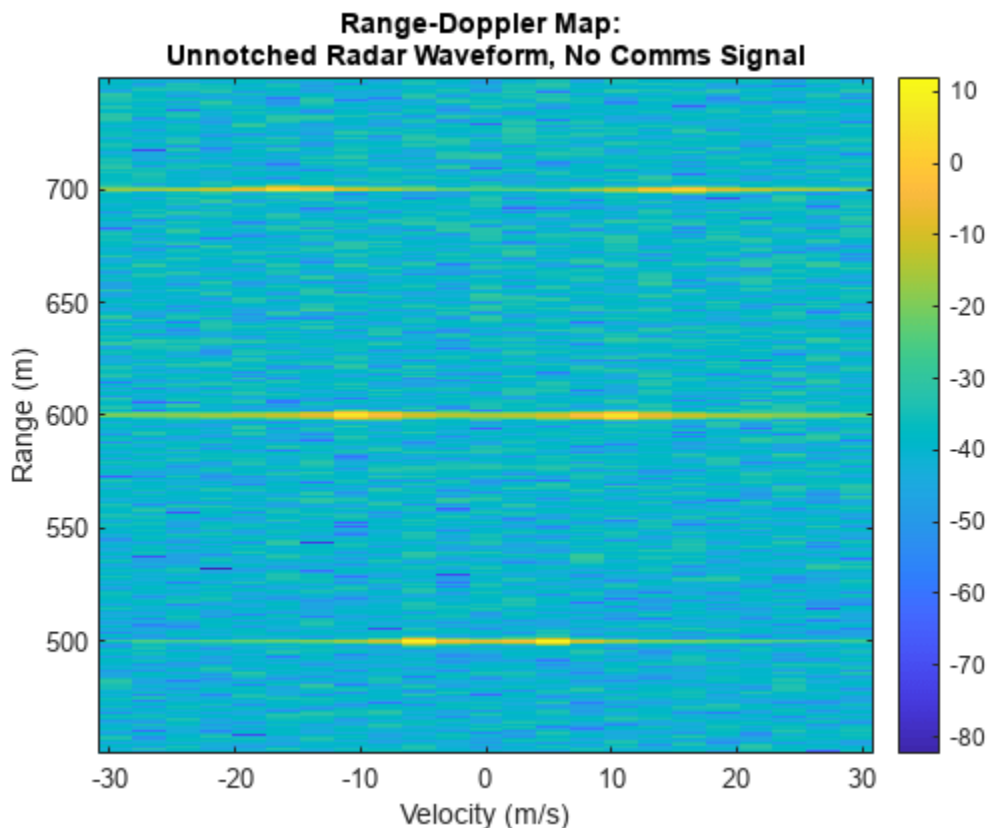
```
% Run simulation and plot Range-Doppler response for notched waveform
% without communications signal
rdNotchedNoComms = helperRunSimulation(notchedWaveformCpi,notchedWaveformCpi,fc,sampleRate,pri,ta
```
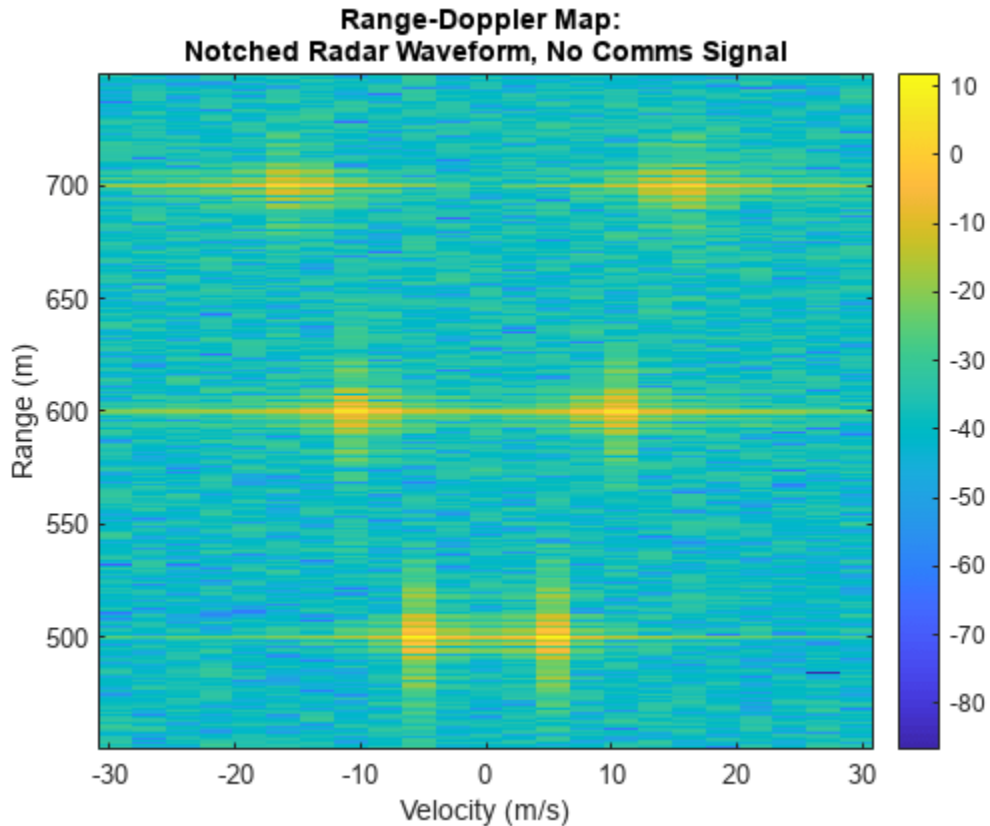


Higher range sidelobes are evident in the notched waveform when compared with the unnotched waveform. Without the presence of the communications signal, the performance of the unnotched waveform qualitatively appears to be better than the notched waveform, which is expected.

The plots below show performance of the notched and unnotched waveforms in the presence of the communications signal. This is where the performance benefits of the notched waveform are evident.

```
% Run simulation and plot Range-Doppler response for unnotched waveform
% with communications signal
rdUnnotchedWithComms = helperRunSimulation(unnotchedWaveformCpiWithComms,unnotchedWaveformCpi,fc
```
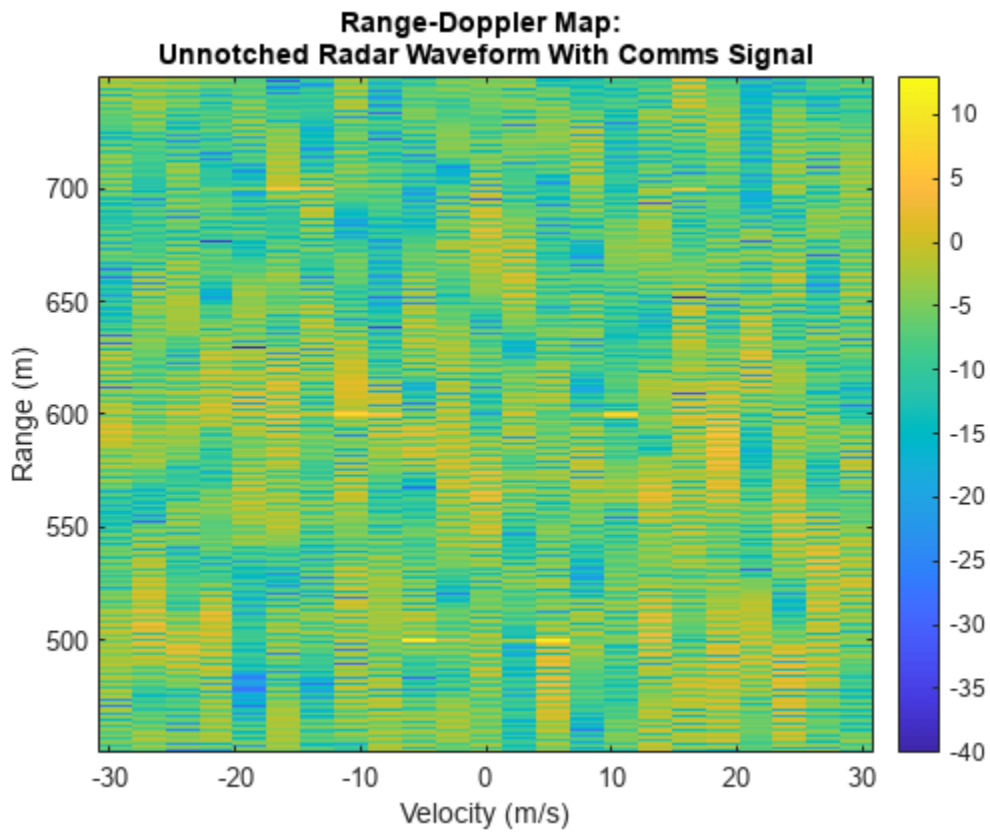
**Range-Doppler Map:
Unnotched Radar Waveform With Comms Signal**



```
% Run simulation and plot Range-Doppler response for unnotched waveform
% with communications signal
rdNotchedWithComms = helperRunSimulation(notchedWaveformCpiWithComms,notchedWaveformCpi,fc,sample
```

Range-Doppler Map:
Notched Radar Waveform With Comms Signal
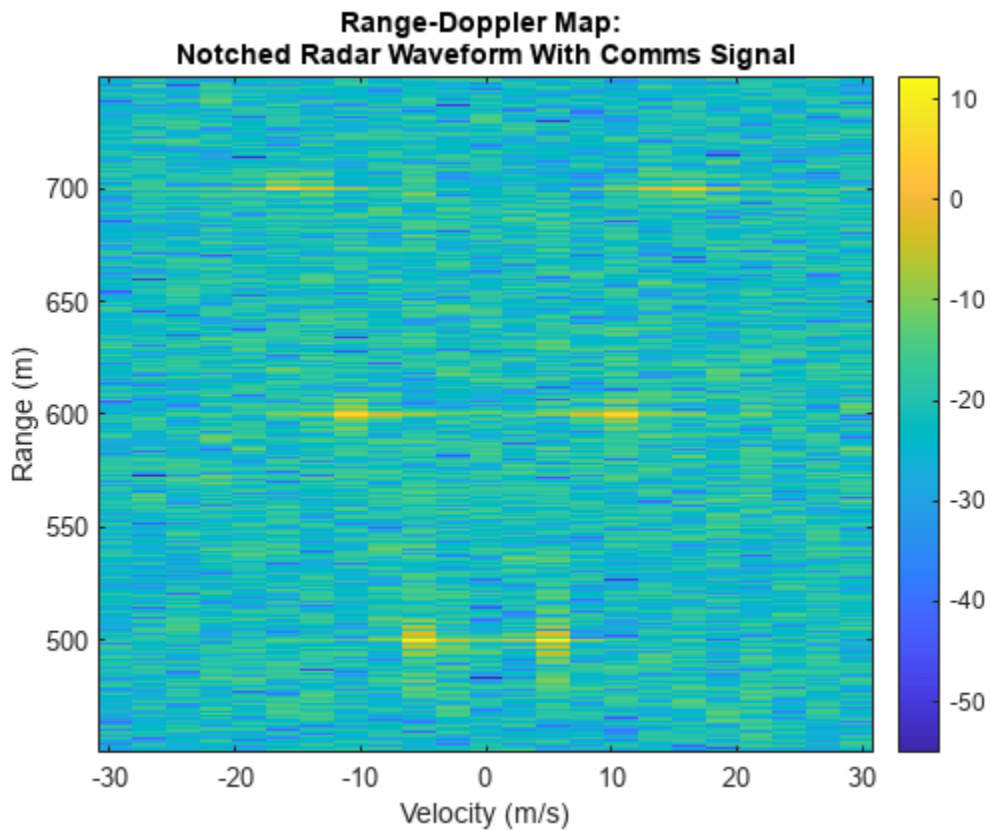
When using an unnotched waveform in the presence of the communications signal, the targets become difficult to distinguish on the Range-Doppler map. Contrast this with the notched waveform Range-Doppler map, where the targets are still clearly visible.

In order to quantify the performance difference between each test case, the average SINR of all the targets is compared. For this example, the noise level is calculated in an unoccupied region from 620 to 680 meters in range and -10 to 10 m/s in Doppler velocity.

```
% Use 620 to 680 m, -10 to 10 m/s Doppler, as the region to calculate noise
% power (excluding target locations)
noiseRange = [620 680];
noiseVelocity = [-10 10];
```

```
% Display a table of target SINR values as well as differences between each
% test case and the base case of unnotched waveform no communications signal.
T = helperGetSinrTable(rdUnnotchedNoComms,rdNotchedNoComms,rdUnnotchedWithComms,rdNotchedWithComm
disp(T);
```

| | SINR at Target Locations (dB) | Delta with Unnotched, No Con |
|---|---|---|
| Unnotched, No Comms Signal | 45.797 | 0 |
| Notched, No Comms Signal | 43.724 | -2.0728 |
| Unnotched, With Comms Signal | 15.623 | -30.173 |
| Notched, With Comms Signal | 28.518 | -17.278 |

This quantitative analysis confirms what is observed qualitatively in the Range-Doppler plots - without any communications signal, the unnotched radar waveform has a marginally higher SINR than the notched radar waveform. However, when the communications signal is present, the SINR of the notched radar waveform is significantly higher than that of the unnotched radar waveform.

Using this type of notching, radar systems can coexist with communication systems while continuing to detect targets effectively.

**Summary**

In this example, you learned how to implement a fast spectrum sensing algorithm to determine the desirable frequency transmission region, as well a waveform notching algorithm to limit transmit frequency to the desirable region. The effectiveness of this approach was then analyzed by simulating the transmission, propagation, and reception of the custom waveform in the presence of a communications signal. This type of workflow may become more common as the need for coexistence schemes between radar and communications systems become more pressing in the future.

**References**

[1] Ravenscroft et al. *Experimental demonstration and analysis of cognitive spectrum sensing and notching for radar.* IET Radar, Sonar & Navigation. 2018.

[2] Jakabosky et al. *Waveform Design and Receive Processing for Nonrecurrent Nonlinear FMCW Radar.* IEEE. 2015.

[3] Blunt et al. *Polyphase-Coded FM (PCFM) Radar Waveforms, Part 1: Implementation.* IEEE. 2014.

[4] Higgins et al. *Mitigating Interference via Spatial and Spectral Nulls.* IEEE International Conference on Radar Systems. 2012.

[5] Martone et al. *Spectrum Allocation for Non-Cooperative Radar Coexistence.* IEEE Transactions on Aerospace and Electronic Systems. 2018.

**Helper Functions**

The following group of helper functions implement the spectrum sensing algorithms discussed in [5].

```
function mesoBands = helperMesoBands(interferenceSpectrumPower,powerThresholdDb,minBandwidth,samp
    % Get the initial meso bands in high and low power
    initialMesoBands = helperInitialMesoBands(interferenceSpectrumPower,powerThresholdDb);

    % Merge high and low power meso bands
    mesoBands = helperMergeMesoBands(initialMesoBands,minBandwidth,sampleRate);
end

function initialMesoBands = helperInitialMesoBands(spectrumPower,powerThresholdDb)
% Generate initial meso-bands from a power spectrum for a given
% threshold

% Get the spectrum power in dB
spectrumPowerDb = pow2db(spectrumPower);

% Hold meso-band start/end index, and whether it is high power
initialMesoBands = [];

% Initialize loop parameters
```

```matlab
spectrumSamples = length(spectrumPowerDb);
iStart = 1;
mesoBandIdx = 1;

for iSample = 1:spectrumSamples
    % Check if the current band start and stop powers are high
    startHighPower = spectrumPowerDb(iStart) > powerThresholdDb;
    stopHighPower = spectrumPowerDb(iSample) > powerThresholdDb;

    % If the start and stop values are in the same power band, advance the
    % stop index continue, otherwise this is the end of a meso-band.
    if startHighPower ~= stopHighPower
        initialMesoBands = helperSetMesoBands(initialMesoBands,mesoBandIdx,iStart,iSample-1,start
        mesoBandIdx = mesoBandIdx + 1;
        iStart = iSample;
    end
end

% Set the last meso-band
initialMesoBands = helperSetMesoBands(initialMesoBands,mesoBandIdx,iStart,spectrumSamples,spectru

end

function mergedMesoBands = helperMergeMesoBands(initialMesoBands,minBandwidth,sampleRate)
% Merge low power meso-bands for a given minimum bandwidth

% Get the number of samples in the spectrum
interferenceSamples = initialMesoBands(end).iStop-initialMesoBands(1).iStart+1;

% Calculate the minimum number of samples based on minimum bandwidth
Lmin = floor(interferenceSamples/sampleRate*minBandwidth);

% Set the initial index based on whether 1st meso-band is high power
mergedMesoBands = initialMesoBands;
if mergedMesoBands(1).HighPower
    currentIdx = 2;
else
    currentIdx = 1;
end

[~,numBands] = size(mergedMesoBands);

% Merge low power bands if they are too narrow
while currentIdx <= numBands
    currentBand = mergedMesoBands(currentIdx);
    currentBandSampleNum = currentBand.iStop - currentBand.iStart + 1;
    if currentBandSampleNum < Lmin
        [mergedMesoBands,currentIdx,numBands] = mergeSingleMesoBand(mergedMesoBands,currentIdx);
    else
        % Increment by 2 to ensure we only look at low power bands
        currentIdx = currentIdx + 2;
    end
end

% Merge the meso-band at the given index with neighbors
function [newMesoBands,currentIdx,newNumBands] = mergeSingleMesoBand(mesoBands,currentIdx)
    [~,initialNumBands] = size(mesoBands);
```

```matlab
        % Determine the range of bands to combine. This is different if we
        % are on the first or the last meso-band.
        if currentIdx == 1
            newNumBands = initialNumBands-1;
            currentIdx = 2;
            combineBandRange = 1:2;
        elseif currentIdx == initialNumBands
            newNumBands = initialNumBands-1;
            combineBandRange = finalNumBands-1:finalNumBands;
        else
            newNumBands = initialNumBands-2;
            combineBandRange = currentIdx-1:currentIdx+1;
        end

        % Get the start and end index of the meso-bands that are being
        % combined.
        startIdx = mesoBands(combineBandRange(1)).iStart;
        endIdx = mesoBands(combineBandRange(end)).iStop;

        % Create the new meso-band, which is always high power
        newBand = helperCreateMesoBand(startIdx,endIdx,true);

        % Get the indexs of the prior and next bands after combining
        priorBandIdx = combineBandRange(1)-1;
        nextBandIdx = combineBandRange(end)+1;

        % Set the new meso-bands after combining the specified band
        newMesoBands = [];
        if priorBandIdx > 0
            newMesoBands = mesoBands(1:priorBandIdx);
        end

        newMesoBands = [newMesoBands,newBand];

        if nextBandIdx <= initialNumBands
            newMesoBands = [newMesoBands,mesoBands(nextBandIdx:end)];
        end
    end
end

function mesoBands = helperSetMesoBands(mesoBands,mesoBandIdx,iStart,iStop,highPower)
% Create meso-band struct row, add it to the meso-bands struct

if isempty(mesoBands)
    mesoBands = helperCreateMesoBand(iStart,iStop,highPower);
else
    mesoBands(mesoBandIdx) = helperCreateMesoBand(iStart,iStop,highPower);
end
end

function newBand = helperCreateMesoBand(startIdx,endIdx,highPower)
% Create a new meso-band

newBand = struct('iStart',startIdx,'iStop',endIdx,'HighPower',highPower);
end

function a = helperPlotMesoBands(spectrumPower,mesoBands,threshold,spectrumFrequencies,plotTitle
% Plot meso-bands along with the frequency spectrum
```

```matlab
% Offset the high and low power bands by 10 dB for display purposes
offset = 10;

% Get the spectrum power in dB
spectrumPowerDb = pow2db(spectrumPower);

% Get the start and stop idxs
threshIdxs = [mesoBands(1).iStart mesoBands(end).iStop];

% Get spectrum frequencies in MHz
spectrumFrequencies = spectrumFrequencies/1e6;

% Plot the spectrum and meso-bands
a = axes(figure);
hold(a,"on");
lThresh = plot(a,spectrumFrequencies(threshIdxs),[threshold threshold],'DisplayName','Threshold'
lSpec = plot(a,spectrumFrequencies,spectrumPowerDb,'DisplayName','Measured Spectrum','LineWidth'
hold(a,"off");
lLow = plotMesoBandLevel(mesoBands(~[mesoBands.HighPower]),threshold,threshold-offset,spectrumFre
lHigh = plotMesoBandLevel(mesoBands([mesoBands.HighPower]),threshold,threshold+offset,spectrumFre
ylabel(a,'Power (dB)')
xlabel(a, 'Frequency (MHz)')
legend(a,[lSpec,lLow,lHigh,lThresh],"Location","southeast");
title(a,plotTitle);
xlim(a,plotXlim);

% Plot high or low power meso-bands
function l = plotMesoBandLevel(mesoBands,threshold,plotPower,freq,description,color,a)
    [~,numBands] = size(mesoBands);

    hold(a,"on");
    for iBand = 1:numBands
        start = mesoBands(iBand).iStart;
        stop = mesoBands(iBand).iStop;
        range = start:stop;
        x = [start,range,stop];
        y = [threshold,repmat(plotPower,[1,length(range)]),threshold];
        l = plot(a,freq(x),y,'DisplayName',description,'Color',color,'LineWidth',2);
    end
    hold(a,"off");
end

end

function notchFrequencies = helperTxNotchFrequencies(mesoBands,signalSpectrumPower,interferenceSp
% Determine notch frequency locations by maximizing the objective function
% for all combinations of messo-bands
% Determine the frequencies to notch

% Calculate the tx band objective values
txBandObjValues = getTxBandObjectiveValues(mesoBands,signalSpectrumPower,interferenceSpectrumPowe

% Calculate normalized objective values for each tx band
txBandObjValuesNorm = normalizeTxBandObjectiveValues(txBandObjValues);

% Get the transmit region with the max objective function
txMesoBands = getMaxObjectiveBands(txBandObjValuesNorm,alpha);
```

```matlab
% Get the notch frequency regions
spectrumFrequencies = helperSpectrumFrequencies(sampleRate,numel(interferenceSpectrumPower));
notchFrequencies = getNotchFrequencyFromTxRegion(mesoBands,txMesoBands,spectrumFrequencies);

function txBandObjectives = getTxBandObjectiveValues(mesoBands,signalSpectrumPower,interferenceSp
    % Get all possible band combinations
    [~,numBands] = size(mesoBands);
    possibleTxBands = getAllPossibleTxBands(numBands);

    % Calculate objectives for each tx band combination
    txBandObjectives = calculateTxBandObjectiveValues(possibleTxBands,mesoBands,signalSpectrumPow

    function possibleBands = getAllPossibleTxBands(numBands)
        % Get every permutation of meso-band combinations
        possibleBands = struct('BandCombo',[],'SINR',[],'Bandwidth',[]);
        bandComboIdx = 1;

        availableBands = 1:numBands;

        for iBandCombos = 1:numBands
            combos = nchoosek(availableBands,iBandCombos);
            [numCombos,~] = size(combos);
            for iCombo = 1:numCombos
                possibleBands(bandComboIdx,1).BandCombo = combos(iCombo,:);
                bandComboIdx = bandComboIdx+1;
            end
        end
    end

    function txBandCalculations = calculateTxBandObjectiveValues(txBandCalculations,mesoBands,sig
        % Calculate the objective functions for each possible meso-band combination

        [numPossibleBands,~] = size(txBandCalculations);

        % Get the frequency values for interference and signal spectrums
        fInterference = helperSpectrumFrequencies(sampleRate,numel(interferenceSpectrumPower));
        fStepInterference = fInterference(2)-fInterference(1);
        fSignal = helperSpectrumFrequencies(sampleRate,numel(signalSpectrumPower));
        fStepSignal = fSignal(2)-fSignal(1);

        % Set the sinr and bandwidth for all possible bands
        for iBandCombo = 1:numPossibleBands
            % Get the indexes of the interference and signal in this
            % meso-band
            bands = txBandCalculations(iBandCombo).BandCombo;
            [intBandIdxs,sigBandIdxs] = getBandIdxs(mesoBands,bands,fInterference,fSignal);

            % Get the power contained in the signal. Multiplication by tau in order to
            % capture effect of pulse width.
            mesoBandSignalPower = sum(signalSpectrumPower(sigBandIdxs))*tau*fStepSignal;

            % Get the power contained in the interference
            mesoBandInterferencePower = sum(interferenceSpectrumPower(intBandIdxs))*fStepInterfer

            % Calculate SINR and bandwidth for this mesoband
            sinr = mesoBandSignalPower / mesoBandInterferencePower;
            bandwidth = length(intBandIdxs)*fStepInterference;
```

```matlab
                    % Store the SINR and bandwidth
                    txBandCalculations(iBandCombo).SINR = sinr;
                    txBandCalculations(iBandCombo).Bandwidth = bandwidth;
                end

                function [intBandIdxs,sigBandIdxs] = getBandIdxs(mesoBands,bands,fInterference,fSignal)
                % Get band indexes from meso-bands.
                    intBandIdxs = [];
                    sigBandIdxs = [];
                    for iBand = 1:length(bands)
                        currentIntIdxs = mesoBands(bands(iBand)).iStart:mesoBands(bands(iBand)).iStop;
                        currentSigIdxs = getIdxFromFreq(fInterference(currentIntIdxs(1)),fInterference(cu
                        intBandIdxs = [intBandIdxs,currentIntIdxs];
                        sigBandIdxs = [sigBandIdxs,currentSigIdxs];
                    end

                    function idxs = getIdxFromFreq(startFreq,stopFreq,freqSamples)
                    % Get the frequency sample indexes from start and stop frequencies
                        startIdxs = find(startFreq <= freqSamples);
                        stopIdxs = find(stopFreq >= freqSamples);
                        idxs = (startIdxs(1):stopIdxs(end));
                    end
                end
            end
        end

    function txBandsNormValues = normalizeTxBandObjectiveValues(txBandsObjValues)
        % Normalize tx band objective values
        normColumns = {'SINR','Bandwidth'};
        txBandsNormValues = normalizeColumns(txBandsObjValues,normColumns);

        function txBandsNorm = normalizeColumns(txBands,normColumnNames)
            txBandsNorm = txBands;
            [rows,~] = size(txBands);

            for iColumn = 1:length(normColumnNames)
                name = normColumnNames{iColumn};
                normName = ['norm',name];
                maxValue = max([txBands.(name)]);
                for iRow = 1:rows
                    txBandsNorm(iRow).(normName) = txBands(iRow).(name)/maxValue;
                end
            end
        end
    end

    function txBands = getMaxObjectiveBands(normObjective,alpha)
        % Get the idxs that maximize the objective function
        maxObjective = -Inf;
        maxObjectiveRow = 0;
        [rows,~] = size(normObjective);

        for iRow = 1:rows
            objective = alpha*normObjective(iRow).normSINR + (1-alpha)*normObjective(iRow).normBandw:
            if objective > maxObjective
                maxObjectiveRow = iRow;
                maxObjective = objective;
```

```
            end
        end

    txBands = normObjective(maxObjectiveRow).BandCombo;
end

function notchFrequency = getNotchFrequencyFromTxRegion(mesoBands,txMesoBands,spectrumFrequencies
    % Get the frequency ranges of the notches based on the selected tx
    % meso-bands
    notchMesoBands = mesoBands;
    notchMesoBands(txMesoBands) = [];

    [~,numNotches] = size(notchMesoBands);
    notchFrequency = zeros(numNotches,2);
    for iNotch = 1:numNotches
        bandStart = notchMesoBands(iNotch).iStart;
        bandStop = notchMesoBands(iNotch).iStop;
        notchFrequency(iNotch,:) = [spectrumFrequencies(bandStart) spectrumFrequencies(bandStop)]
    end
end

end
```

The following group of helper functions implement the waveform generation algorithms.

```
function waveform = helperGenerateUnnotchedWaveform(bandwidth,pulseWidth,sampleRate,numPulses)
% Generate an unnotched waveform using the approach described in [1].

% Call helperGenerateNotchedWaveform with empty notch frequencies
notchFrequencies = [];
waveform = helperGenerateNotchedWaveform(bandwidth,pulseWidth,sampleRate,numPulses,notchFrequenci

end

function waveform = helperGenerateNotchedWaveform(bandwidth,pulseWidth,sampleRate,numPulses,notch
% Generate a notched waveform using the approach described in [1].

% Generate the seed waveform
pcfmWaveform = helperSeedWaveform(bandwidth,pulseWidth,sampleRate,numPulses);

% Calculate the number of samples in the pulse, and get the frequencies in
% the spectrum
numSamples = pulseWidth*sampleRate;

% Generate the unnotched gaussian spectrum.
unnotchedSpectrum = gausswin(numSamples);

% Notch the spectrum
notchedSpectrum = helperNotchSpectrum(unnotchedSpectrum,sampleRate,notchFrequencies);

% Generate the notched waveform
iterations = 500;
waveform = helperProFm(pcfmWaveform,notchedSpectrum,iterations);

end

function pcfmWaveform = helperSeedWaveform(bandwidth,pulseWidth,sampleRate,numPulses)
% Generate a PCFM waveform with randomized frequencies following the
```

```matlab
    % process described in [3].

    % Get the number of samples in a symbol and the number of symbols in a
    % pulse
    samplesPerPulse = pulseWidth*sampleRate;
    samplesPerSymbol = sampleRate/bandwidth;
    symbolsPerPulse = samplesPerPulse/samplesPerSymbol;
    samplesPerPulse = samplesPerSymbol*symbolsPerPulse;

    % Perform randomized PCFM steps
    randomPhases = 2*pi*(rand(symbolsPerPulse+1,numPulses)-0.5);

    % Calculate the phase difference
    phaseDifference = pcfmCalculatePhaseDifference(randomPhases);

    % Generate impulse train
    impulseTrain = pcfmGenerateImpulseTrain(phaseDifference,numPulses,samplesPerSymbol,samplesPerPuls

    % Filter impulses
    filteredImpulseTrain = pcfmFilterImpulseTrain(impulseTrain,samplesPerSymbol);

    % Integrate to get phase
    pcfmPhases = randomPhases(1,:) + cumtrapz(filteredImpulseTrain,1);

    % Generate samples based on phase
    pcfmWaveform = exp(1i*pcfmPhases);

    % Get the phase difference of the random phases
    function phaseDifference = pcfmCalculatePhaseDifference(randomPhases)
        phaseDifference = diff(randomPhases,[],1);
        greaterThanPi = abs(phaseDifference) > pi;
        phaseDifference(greaterThanPi) = phaseDifference(greaterThanPi) - 2*pi*sign(phaseDifference(g
    end

    % Get the impulse train representing frequency
    function impulseTrain = pcfmGenerateImpulseTrain(phaseDifference,numPulses,samplesPerSymbol,total
        impulseTrain = zeros(totalSamples,numPulses);
        impulseTrain(1:samplesPerSymbol:totalSamples,:) = phaseDifference;
    end

    % Get the filtered impulse train representing frequency
    function filteredImpulseTrain = pcfmFilterImpulseTrain(impulseTrain,samplesPerSymbol)
        raisedCosFilter = rcosdesign(0.25,6,samplesPerSymbol)';

        [rows,cols] = size(impulseTrain);
        filteredImpulseTrain = zeros(rows,cols);
        for iCol = 1:cols
            filteredImpulseTrain(:,iCol) = conv(impulseTrain(:,iCol),raisedCosFilter,'same');
        end
    end

end

function proFmWaveform = helperProFm(seedWaveform,desiredSpectrumPower,iterations)
% Perform the PRO-FM technique following the process described in [2].

i = 1;
proFmWaveform = seedWaveform;
```

```matlab
        desiredSpectrumMagnitude = desiredSpectrumPower.^(1/2);

while i < iterations

    % Get the fft of the current signal
    proFmWaveformSpectrum = fftshift(fft(proFmWaveform));

    % Get the phase of the current signal spectrum
    proFmWaveformSpectrumPhase = angle(proFmWaveformSpectrum);

    % Phase shift the desired spectrum magnitude so that it's phase matches
    % the current signal phase
    filteredFrequencySignal = desiredSpectrumMagnitude.*exp(1i*proFmWaveformSpectrumPhase);

    % Get the new signal in the time domain
    filteredTimeSignal = ifft(fftshift(filteredFrequencySignal));

    % Normalize the new signal to be constant amplitude
    proFmWaveform = exp(1i*angle(filteredTimeSignal));

    i = i + 1;
end

end

function notchedSpectrum = helperNotchSpectrum(initialSpectrum,sampleRate,notchFrequencies)
% Insert notches into the given spectrum. Use Tukey Tapers following the
% process described in [3].

% Get the spectrum in dB.
logSpectrum = pow2db(initialSpectrum);

% Get the spectrum frequency samples
spectrumFrequencies = helperSpectrumFrequencies(sampleRate,numel(initialSpectrum));

% Set the notch floor to -85 dB
minValue = -85;

% Get the number of samples in the spectrum
numSamples = length(initialSpectrum);

% taper is B/32, number if samples is 2B, so taper is samples/64
taperLength = ceil(numSamples/64);

% Get the indexes of the notch frequencies in the spectrum
notchIdxs = getNotchStartEndIdx(notchFrequencies,spectrumFrequencies);
[notchNum,~] = size(notchIdxs);

% For each notch, update the spectrum
for iNotch = 1:notchNum
    idxs = notchIdxs(iNotch,:);
    logSpectrum(idxs(1):idxs(end)) = minValue;

    % set tukey taper on leading and trailing edges of notch
    fallingTaperStart = idxs(1)-taperLength;
    fallingTaperStartValue = logSpectrum(fallingTaperStart);
    fallingTaper = createTukeyTaper(taperLength,fallingTaperStartValue,minValue);
    logSpectrum(fallingTaperStart:idxs(1)) = fallingTaper;
```

```matlab
        risingTaperEnd = idxs(end)+taperLength;
        risingTaperEndValue = logSpectrum(risingTaperEnd);
        risingTaper = createTukeyTaper(taperLength,minValue,risingTaperEndValue);
        logSpectrum(idxs(end):risingTaperEnd) = risingTaper;
    end

    % Return spectrum power as amplitude
    notchedSpectrum = db2pow(logSpectrum);

    function taper = createTukeyTaper(numSamples,startValue,endValue)
    % Insert a taper where spectral notches occur.
        height = abs(startValue-endValue)/4;
        lowestValue = max([startValue,endValue])-height;
        if startValue > endValue
            flipTaper = false;
        else
            flipTaper = true;
        end

        % create taper
        taper = cos(0.5*pi/numSamples*(0:numSamples));

        % stretch taper to height
        taper = taper*height;

        % move taper down to end value
        taper = taper+lowestValue;

        if flipTaper
            taper = flip(taper);
        end
    end

    % Get the start and end index of the notch frequencies
    function notchIdxs = getNotchStartEndIdx(notchFreq,spectrumFrequencies)
        [numNotches,~] = size(notchFreq);
        notchIdxs = zeros(numNotches,2);

        for iNotchFreq = 1:numNotches
            notchStart = notchFreq(iNotchFreq,1);
            notchEnd = notchFreq(iNotchFreq,2);

            aboveNotchStart = find(spectrumFrequencies >= notchStart);
            belowNotchEnd = find(spectrumFrequencies <= notchEnd);
            notchIdxs(iNotchFreq,:) = [aboveNotchStart(1) belowNotchEnd(end)];
        end
    end

end
```

*Copyright 2023 The MathWorks, Inc.*

# Audio Beamforming on a Multicore Processor

This example shows how to implement an audio beamforming application on a multicore processor. To guarantee that the beamforming algorithms execute in real-time, individual beamforming tasks run on different processor cores.

**Algorithm Description**

The acoustic beamforming in this example uses a uniform linear array (ULA) of microphones. The application receives three simulated audio signals from different directions on a twenty-element uniformly spaced linear microphone array. After the addition of thermal noise at the receiver, beamforming is applied to each of three different source angles to separate the sources. The image illustrates the application composition.
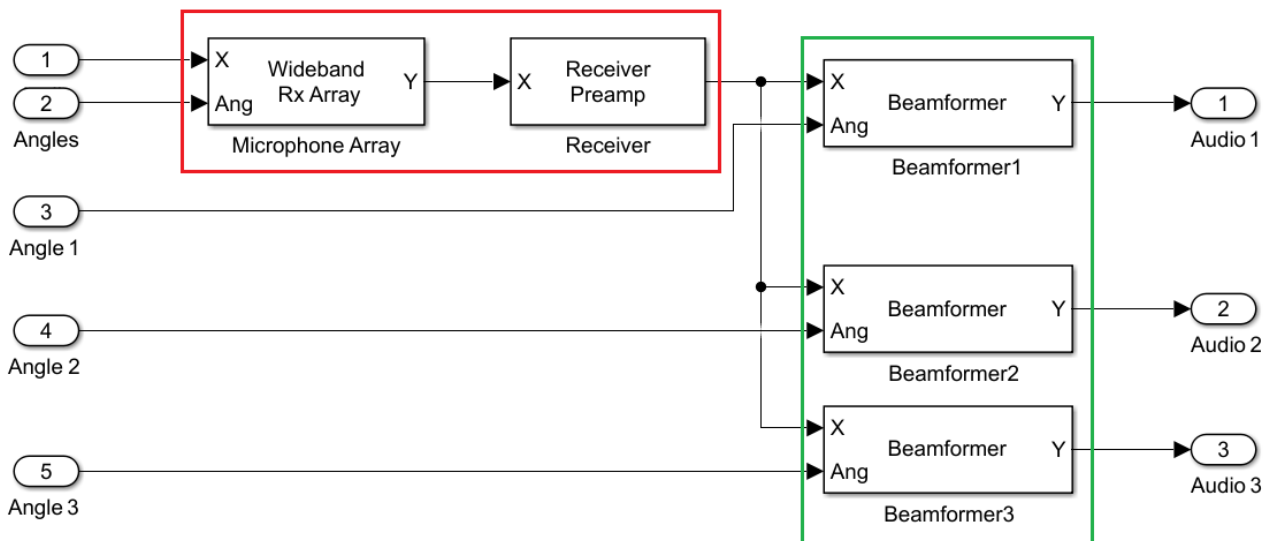


In this example, the audio sources are simulated by reading the audio samples from binary files. The microphone array and the receiver are also simulated and marked by the red box. The beamformers are the part of the algorithm and they are marked by the green box.

**Algorithm Partitioning into Tasks**

In this example, the signal generation, the microphone array, and the receiver are simulated, and they are marked as Task1. The model performs three independent beamforming algorithms for each of the expected audio sources. They are marked as Task2, Task3, and Task4.

```
soc_beamforming_proc;
```
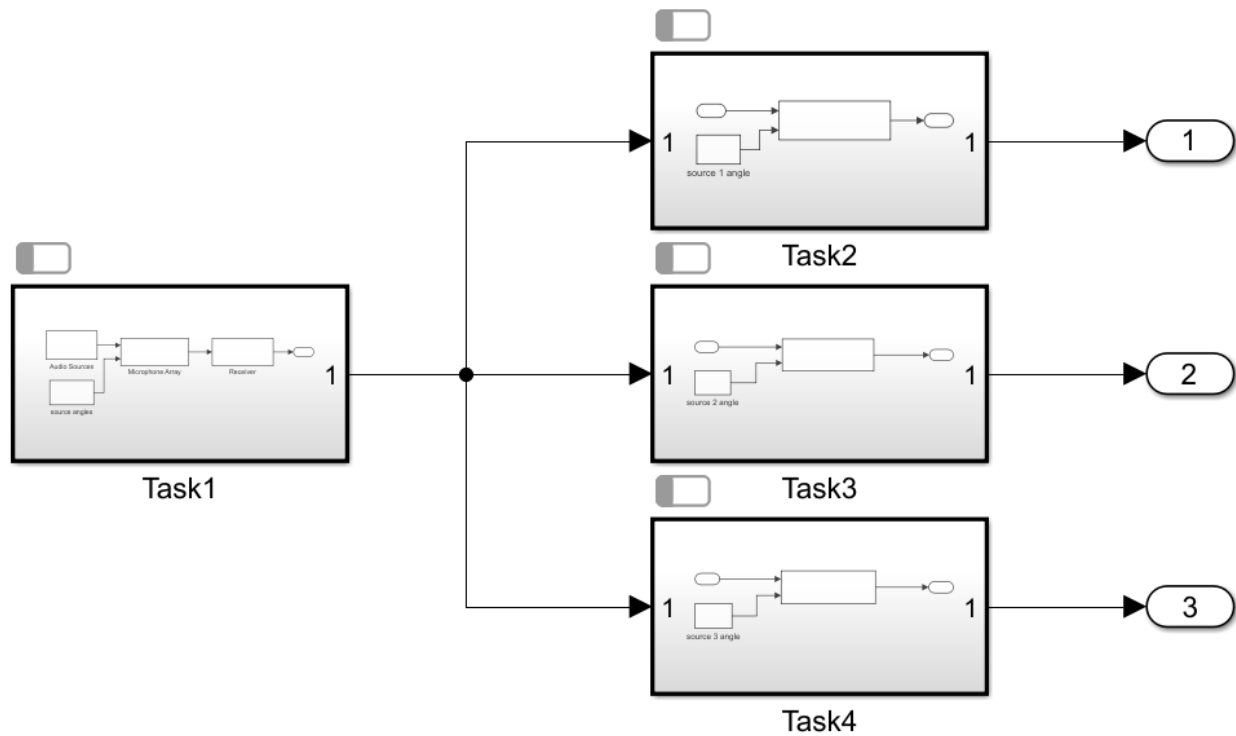
Each task block in this model is created as an explicit periodic task partition, with the period of each partition set to FRAMERATE=FRAMESIZE/SAMPLINGFREQUENCY.

**Top Model Description**

The task partitions are scheduled for real time execution using the Task Manger block in the top model. The Task Manager blocks sets tasks for each task partition described in the model in the previous step. The tasks are set for the same period as the corresponding partition.

```
soc_beamforming_top;
```
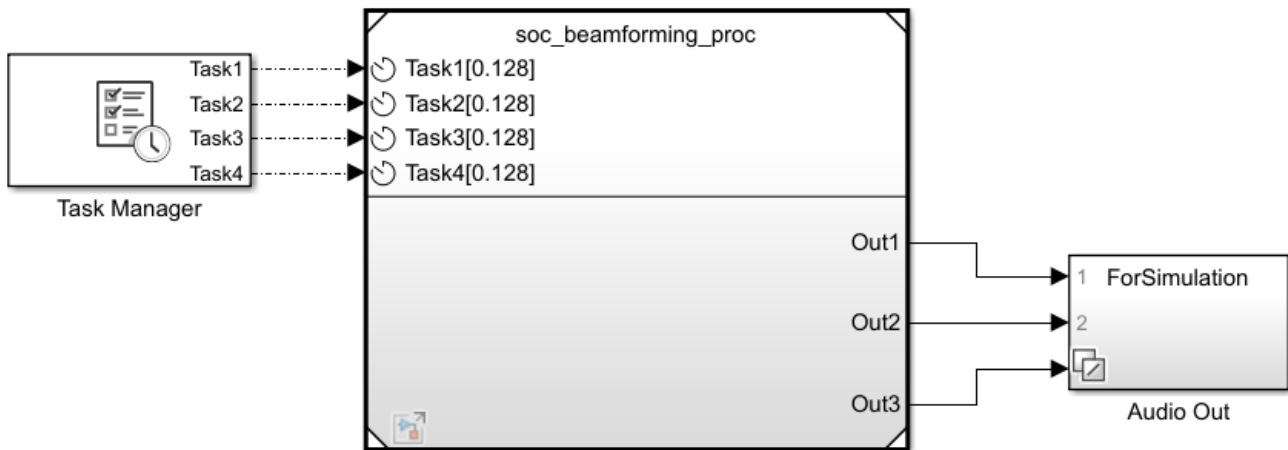
In addition, the top model contains the Audio Out subsystem that has the Audio Device Writer block that plays the audio stream to the computer's audio device in simulation. The Multiport Switch block in the same subsystem allows selecting which of the three audio streams are directed to the audio device.

**Simulate the Model**

**Prepare for Simulation**

Three audio source files with different audio recordings are used in this example: *soc_dftvoice.wav*, *soc_cleanvoice.wav*, and *soc_laughter.wav*. The files are provided in the Waveform Audio File format but need to be converted to a binary format so that they can be used both for simulation and for deployed application. Run the provided MATLAB function to perform the conversion.

```
soc_beamforming_create_bin;
```

Three binary files are created as a result: *src1.bin*, *src2.bin*, and *src3.bin*.

**Run Simulation**

To run the simulation, click **Run** in the **SIMULATION** section of the Simulink toolstrip. Use the Multiport Switch block in the Audio Out subsystem to selecting which audio source to listen to. Change the selected source during the simulation and confirm that the audio stream changes. Observe that there is some interference from other sources, which is expected. The magnitude of the interference depends on the distance among the audio sources.

As it runs, the application writes its outputs to three binary files: *output1.bin, output2.bin*, and *output3.bin.* You can play each of these files to an audio device using the following model by entering the file name in the Binary File Reader block.

```
soc_beamforming_test_output;
```

You can also analyze the signal spectrum using the Spectrum analyzer block in addition to the Audio Device Writer block. Or you can load the data into MATLAB and perform even more complex analysis, such as correlation.

**Deploy the Model to Hardware**

**Prepare for Deployment**

Before running the model on hardware, make sure that you install the SoC Blockset Support Package for Xilinx Devices. The example models are set to Xilinx Zynq UltraScale+ RFSoC ZCU111 Evaluation Kit, but you can use any board supported by the support package if it has at least four processor cores. In that case, change the board for all models. Also, run the **Setup Hardware** app for your board from the **HARDWARE BOARD** section of the Simulink toolstrip to set up the correct board connection information.

To run the model on hardware, you must set the top model to use the Fixed-step solver. Click the **Model Settings** in the **MODELING** section of the Simulink toolstrip and set the **Type** to **Fixed-step** in the **Solver selection** of the **Solver** page.

To run the model on hardware you also must copy the audio source files you generated to your hardware board. Use the following steps:

**Create Connection your Board**

Create a connection object to yor board.

```
hw = socHardwareBoard('Xilinx Zynq UltraScale+ RFSoC ZCU111 Evaluation
Kit','hostname','<ipaddress>','username','root','password','root');
```

If you use a different board, replace the board name with your board name. Also replace <ipaddress> string with your board IP address, which is the same one that you set up earlier via the **Setup Hardware** app.

**Copy Audio Source Files to the Board**

Use the connection object *hw* you created in the previous step to copy the files *src1.bin*, *src2.bin*, and *src3.bin* to your board using the following command in MATLAB for each file.

```
putFile(hw,'<filename>');
```

The files will be copied into your home folder on the board. The application will also run in this folder.

**Deploy to a Single Core**

First, run the entire application on a single core. All tasks in the Task Manger block are set to core 0.

| Task | Core |
|------|------|
| Task1 | 0 |
| Task2 | 0 |
| Task3 | 0 |
| Task4 | 0 |

For example, Task4 is set to core 0.

Run the model on hardware by clicking the **Monitor & Tune** in the **RUN ON HARDWARE** section of the Simulink toolstrip. The model runs in external mode and the execution time statistics are collected for each task.

**Analyze the Execution Performance**

Click **Execution Report** in the **REVIEW RESULTS** section of the Simulink toolstrip to get the summary report.

**SoC Blockset Task Execution Report**

Model:   soc_beamforming_top

Date:    23-Dec-2022 21:30:13

Task execution statistics

| Name | Finished | Min | Mean | Max | SD | Min Turn | Mean Turn | Max Turn |
|------|----------|-----|------|-----|-----|----------|-----------|----------|
| Task1 | 257 | 0.0098 | 0.0108 | 0.2263 | 0.0135 | 0.0098 | 0.0108 | 0.2263 |
| Task2 | 256 | 0.0191 | 0.0195 | 0.0198 | 5.7961e-05 | 0.0191 | 0.0195 | 0.0297 |
| Task3 | 256 | 0.0195 | 0.0199 | 0.0201 | 5.3456e-05 | 0.0195 | 0.0199 | 0.0201 |
| Task4 | 255 | 0.0213 | 0.0217 | 0.0224 | 7.0391e-05 | 0.0213 | 0.0220 | 0.1107 |

Core usage statistics

| Core | Task1 | Task2 | Task3 | Task4 | Idle |
|------|-------|-------|-------|-------|------|
| Core: 0 | 8.4636 | 15.2484 | 15.5671 | 16.9422 | 43.7787 |
| Core: 1 | 0 | 0 | 0 | 0 | 100.0000 |
| Core: 2 | 0 | 0 | 0 | 0 | 100.0000 |
| Core: 3 | 0 | 0 | 0 | 0 | 100.0000 |

You can visualize the task execution and the core usage by clicking the **Data Inspector** in the **REVIEW RESULTS** section of the Simulink toolstrip. As expected, all tasks run on core 0, one after another.
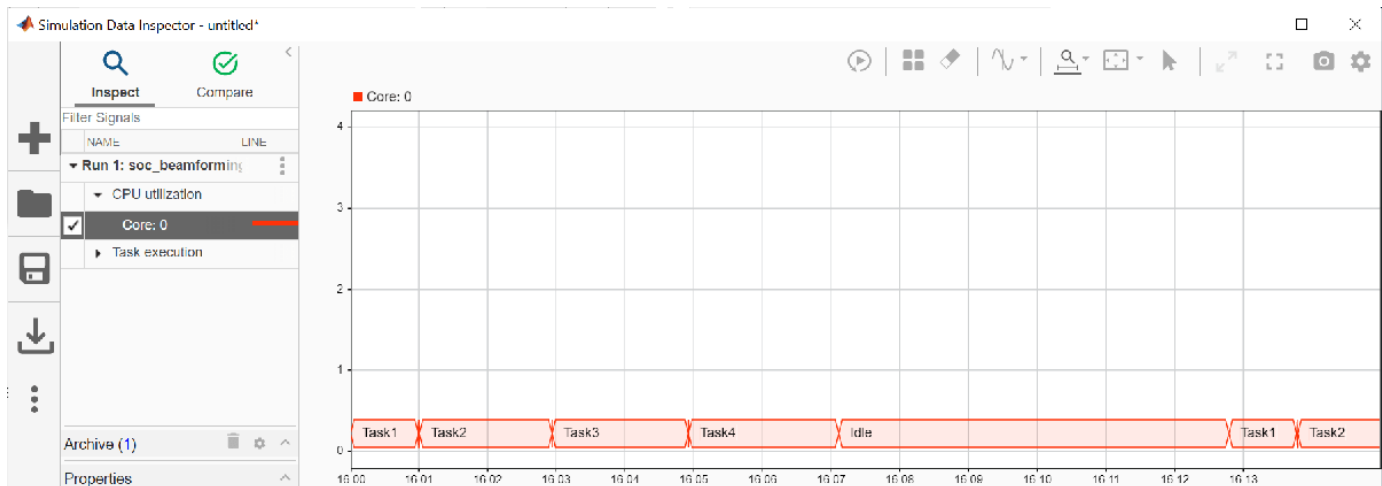


It is evident that running all tasks on one core does not leave a lot of headroom to handle unexpected events and processes in the operating system. Thus, it is advisable to spread the computational burden to other cores.

**Verify Application Output**

As it runs on hardware, the application writes its outputs to the binary files: *output1.bin, output2.bin,* and *output3.bin.* To verify that the application behavior is correct, copy the files from your board to MATLAB for further analysis. Use the same connection object *hw* you created earlier to copy each file by using the following command in MATLAB.

```
getFile(hw,'<filename>');
```

Play each of these files to an audio device using the following model by entering the file name in the Binary File Reader block.

```
soc_beamforming_test_output;
```

**Deploy to Multiple Cores**

Next, run the application on multiple cores. Each task in the Task Manger block is set to a different core.

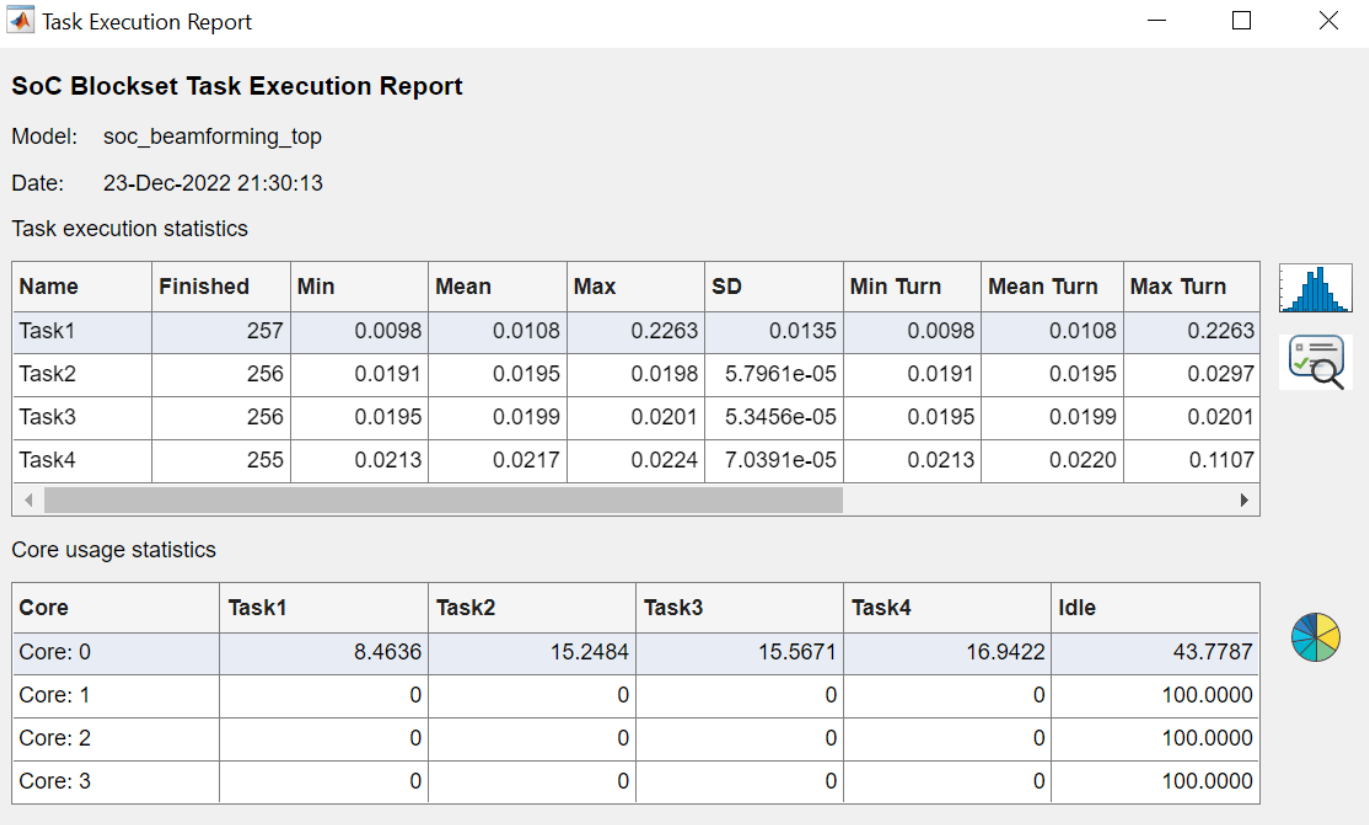| Task | Core |
|------|------|
| Task1 | 0 |
| Task2 | 1 |
| Task3 | 2 |
| Task4 | 3 |

For example, Task4 is set to core 3.
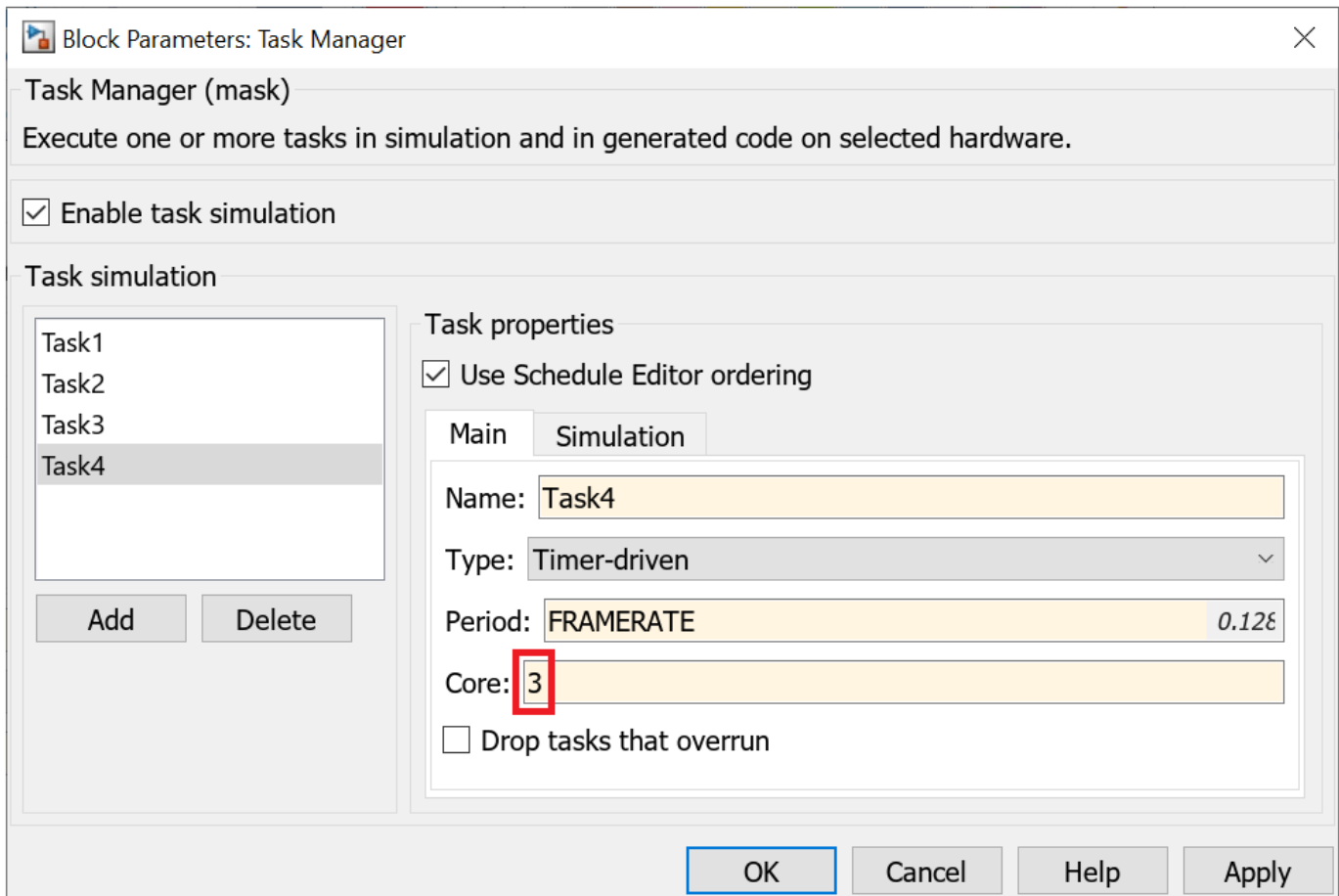
Run the model on hardware by clicking the **Monitor & Tune** in the **RUN ON HARDWARE** section of the Simulink toolstrip. The model runs in external mode and the execution time statistics are collected for each task.

**Analyze the Execution Performance**

Click **Execution Report** in the **REVIEW RESULTS** section of the Simulink toolstrip to get the summary report.
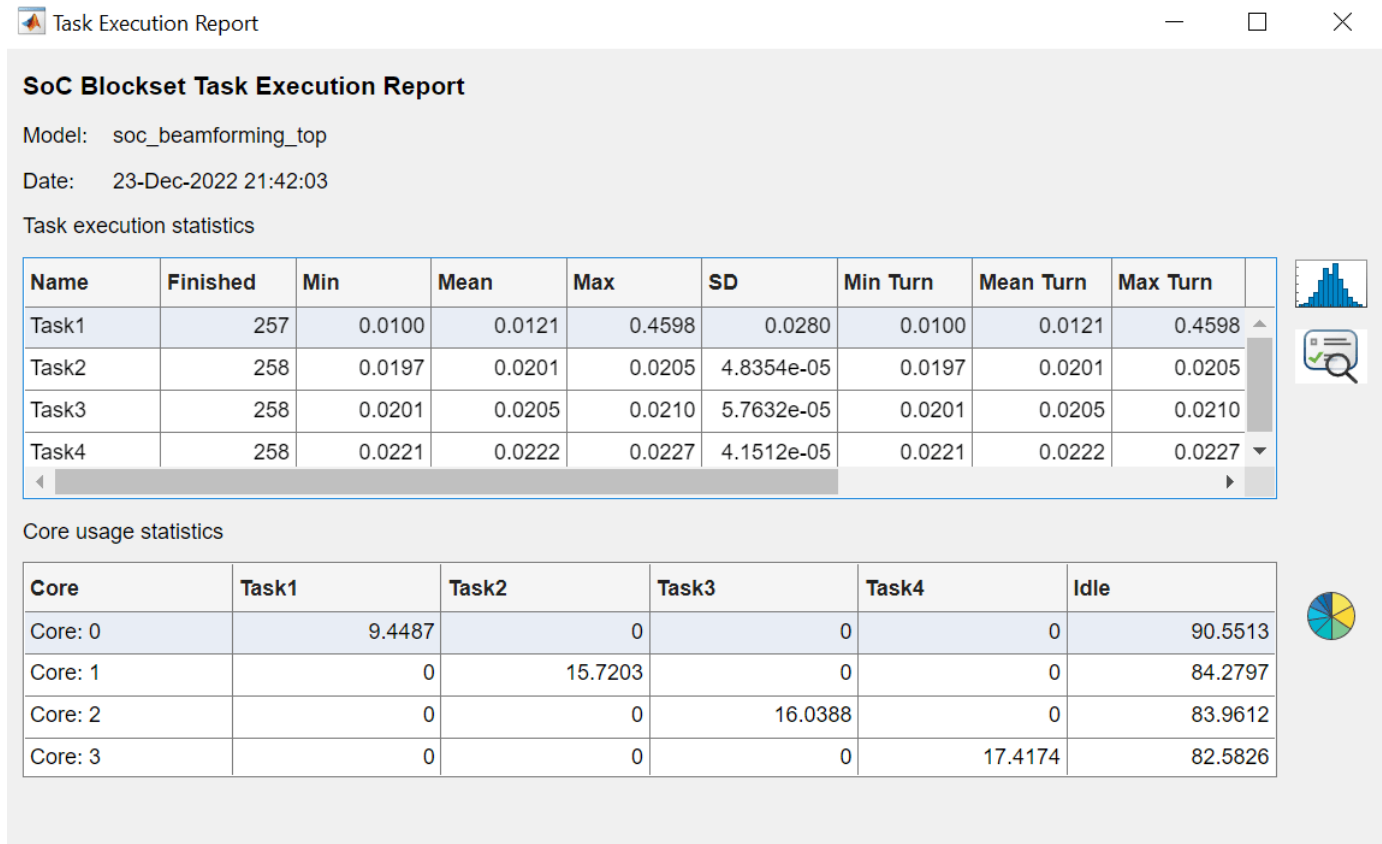
You can visualize the task execution and the core usage by selecting the **Data Inspector** in the **REVIEW RESULTS** section of the Simulink toolstrip. All tasks run on different cores concurrently.
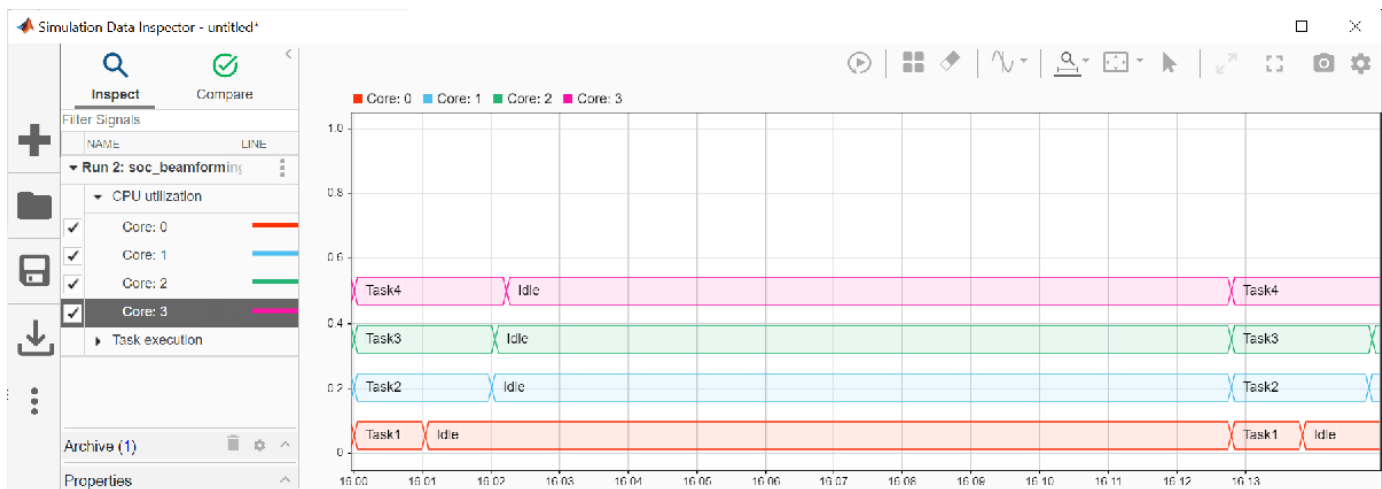


You can notice that running tasks on different cores increases the headroom quite substantially and the application is then a lot more robust and able to handle unexpected events and processes in the operating system.

**Verify Application Output**

As it runs on hardware, the application writes its outputs to the binary files: *output1.bin, output2.bin,* and *output3.bin.* To verify that the application behavior is correct, copy the files from your board to MATLAB for further analysis. Use the same connection object *hw* you created earlier to copy each file by using the following command in MATLAB.

```
getFile(hw,'<filename>');
```

Play each of these files to an audio device using the following model by entering the file name in the Binary File Reader block.

```
soc_beamforming_test_output;
```

**Conclusions**

This example showed how to accelerate an algorithm by running it on a multicore processor. By going from one to four cores, the algorithm was accelerated by the factor of ~3.5.

**Further Exploration**

Change the distance between the audio sources by changing the source angle inputs to the microphone array block and the beamformer blocks. Observe the change in the magnitude of the interference.

# See Also

# Related Examples

- "Acoustic Beamforming Using a Microphone Array" on page 18-185
- "Wideband Beamforming" on page 5-15
- "Adaptive Beamforming" on page 5-11
- "Introduction to Hybrid Beamforming" on page 18-371

# Direction of Arrival Estimation Using Sparse Arrays

Direction of arrival (DOA) estimation is an important application for a sensor array. However, an N-element uniform linear array (ULA) can only resolve N-1 uncorrelated signal sources, which is a significant limitation for many applications. As a result, sparse arrays have attracted a lot of interests because you can achieve a larger array aperture with a given number of sensors. This example constructs several popular sparse array architectures and shows how they can be used to perform DOA estimation.

For the sake of simplicity, the example focuses on linear arrays. Assume there are 5 signal sources.

```
theta_az=[-45,-20,0,20,40]; % azimuth
Nsig = length(theta_az); % the number of target
```

**Minimum Redundancy Array**

One of the earliest sparse array designs is minimum redundancy array (MRA). An MRA is designed to provide maximum resolution for a given number of elements in the array. The idea behind an MRA is to use true elements to measure the phase difference with unique element spacing. Then using these phase shifts, we can recover the phase shifts incurred from a full size uniform linear array.
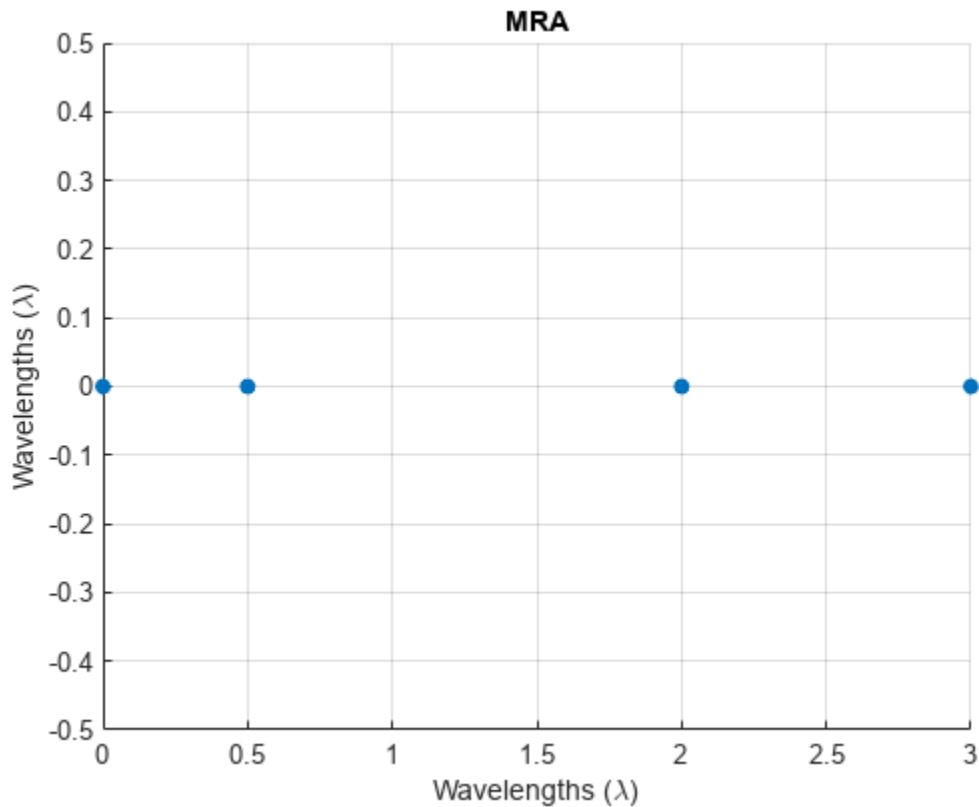
For example, let us consider a 4-element linear array, which is less than the number of signals in our scene. For a 4-element MRA, the element position is given by

```
d = 0.5;  % element spacing in wavelength
pos_mra = [0 1 4 6]*d;
```

This MRA geometry is shown in the following figure.

```
helperDisplayArray(pos_mra,'MRA');
```

It is well known that when a signal arrives at a uniform linear array, it introduces constant phase difference between adjacent sensor elements. Therefore, the correlation between two sensor elements in the array is spacing invariant, i.e., the correlation between the two sensor elements only depends on the spacing between the two sensor elements. First, let us explore what these redundancies look like for the chosen MRA.

```
pos_mra_idx = round(pos_mra/d);
% compute index differences between all element pairs
pos_mra_idx_diff = pos_mra_idx-pos_mra_idx.';
% find unique differences
pos_mra_idx_unique = sort(unique(pos_mra_idx_diff(:).'))
```

pos_mra_idx_unique = *1×13*

    -6    -5    -4    -3    -2    -1    0    1    2    3    4    5    6

This means that, even though we have only 4 elements in this MRA, we could take advantage of the redundancy and reconstruct a full array that is 13 elements long.

```
pos_mra_idx_reconstruct = min(pos_mra_idx_unique):max(pos_mra_idx_unique);
pos_mra_reconstruct = pos_mra_idx_reconstruct*d;

helperDisplayArray(pos_mra_reconstruct,'Full Aray Reconstructed from MRA');
```
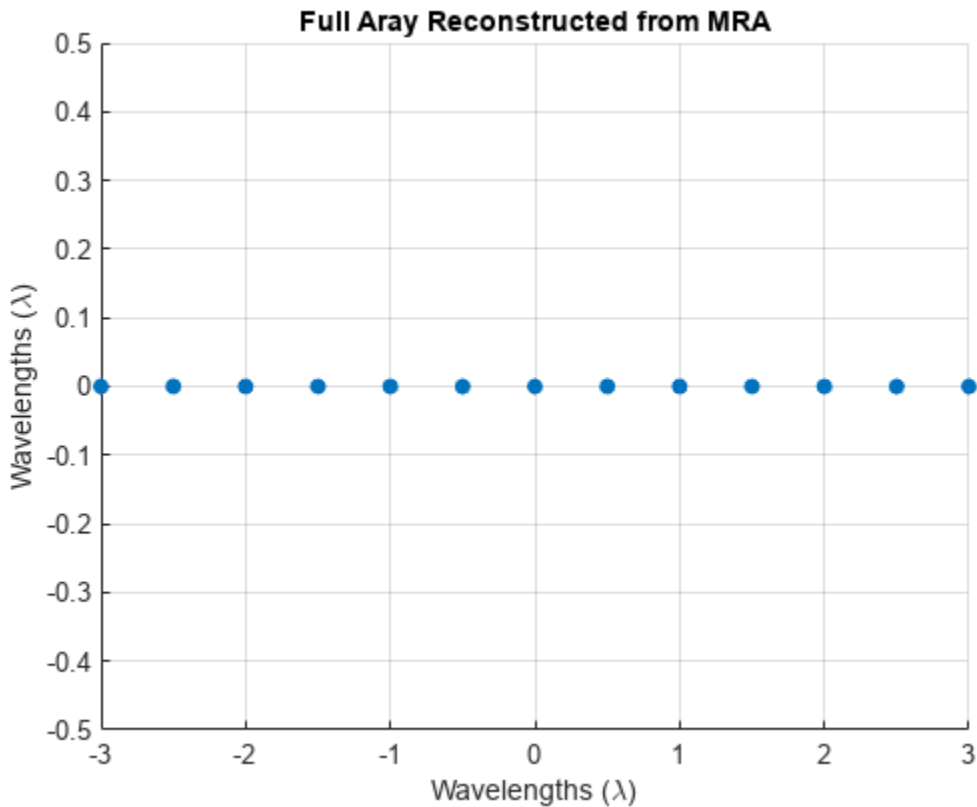
To reconstruct the measurements of the full array, we need to first generate multiple snapshots of measurements of the MRA. In this example, we simulate a 50-snapshot signal from the sources with a 10dB SNR.

```
N_snap = 50;
rng(2022);
x_mra = sensorsig(pos_mra,N_snap,theta_az,db2pow(-10));
```

We now construct the covariance matrix of the MRA.

```
Rxx_mra = x_mra'*x_mra/N_snap;
```

Then using the entries in this covariance, we can construct the full array's measurements.

```
% Find entries in covariance matrix that correspond to phase shifts of
% elements in the full array
[~,mra_cov_idx] = ismember(pos_mra_idx_reconstruct(:),pos_mra_idx_diff);
x_full_mra = Rxx_mra(mra_cov_idx);
```

This is one snapshot of the reconstructed full array measurements. We can then use beam scan method to estimate the DOAs. Note that the full array not only improves the angular resolution of the original array, but also increases the degrees of freedom to 13.

```
ang = -90:90;
sv = steervec(pos_mra_reconstruct,ang);
P_full_mra = sv'*x_full_mra;
```

```
helperDisplayAngularSpectrum(ang,P_full_mra,theta_az,'Beamscan Spectrum for MRA');
```
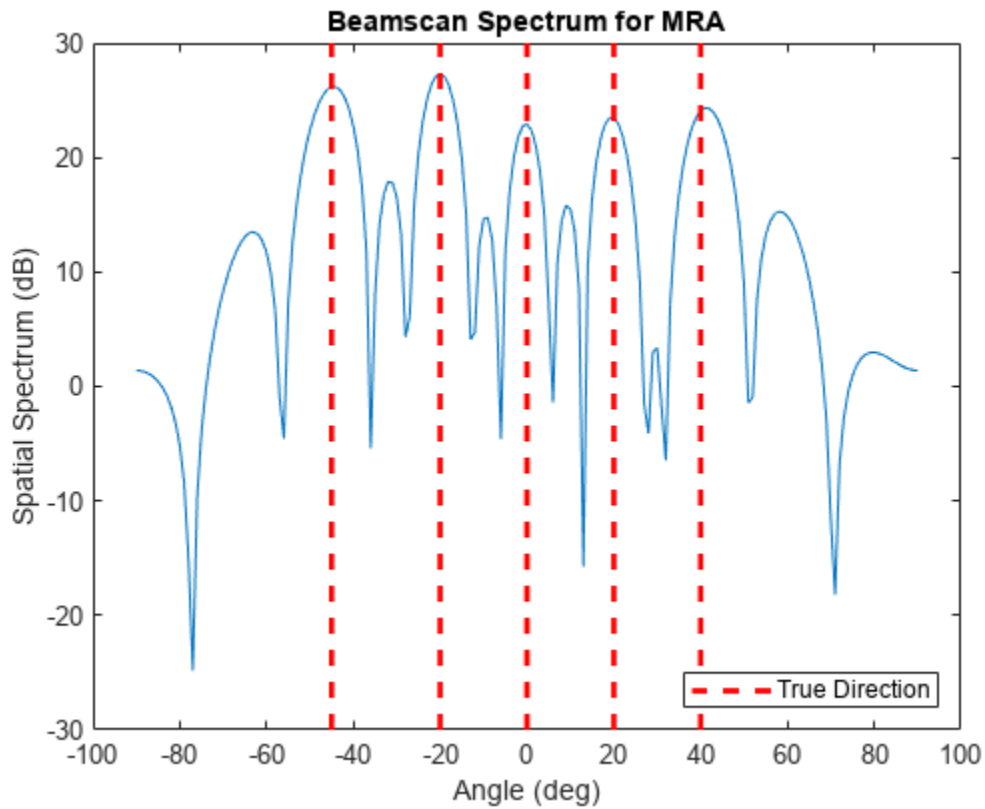
**Nested Array**

Unfortunately, identifying the architecture for an MRA is not trivial, so such configuration is often not readily available. Recently, nested arrays and coprime arrays have become popular choices for sparse arrays.

A nested array embeds a ULA with small element spacing into another ULA with larger element spacing. Here is an example

```
Mn = 3;
Nn = 3;
pos_Mn = (0:Mn-1)*d;
pos_Nn = Mn*d:(Mn+1)*d:Nn*(Mn+1)*d;
pos_nested = unique([pos_Mn,pos_Nn]);

helperDisplayArray(pos_nested,'Nested Array');
```
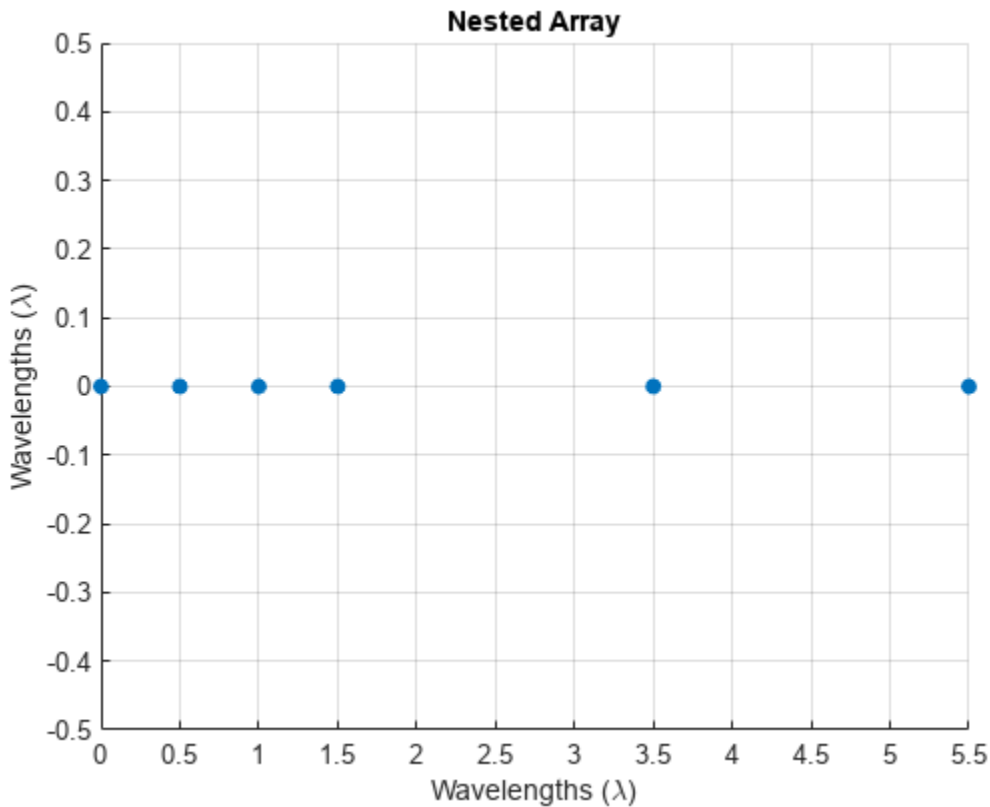
As you can see, this nested array is formed with a small ULA with half wavelength spacing and then a sparser ULA with 2-wavelength spacing. There are 6 elements in total. For such a nested array, the redundancies can be computed as

```
pos_nested_idx = round(pos_nested/d);
pos_nested_idx_diff = pos_nested_idx-pos_nested_idx.';
pos_nested_idx_unique = sort(unique(pos_nested_idx_diff(:).'))
```

pos_nested_idx_unique = *1×23*

   -11   -10   -9   -8   -7   -6   -5   -4   -3   -2   -1   0   1   2   3   4

Note that we now can construct a 23-element full array. In fact, the degrees of freedom of the full array can be predicted as

```
N_nested = Mn+Nn;
dof_nested = (N_nested.^2-2)/2+N_nested
```

dof_nested = 23

Not only a nested array easier to construct than an MRA, but its performance can be easily predicted. For example, there is no closed form solution to construct a 6-element MRA that can be used to reconstruct a full array without any missing elements. The reconstruction process for the nested array is similar to the MRA and is illustrated below.

```
pos_nested_idx_reconstruct = min(pos_nested_idx_unique):max(pos_nested_idx_unique);
pos_nested_reconstruct = pos_nested_idx_reconstruct*d;
```

```
x_nested = sensorsig(pos_nested,N_snap,theta_az,db2pow(-10));
Rxx_nested = x_nested'*x_nested/N_snap;
[~,nested_cov_idx] = ismember(pos_nested_idx_reconstruct(:),pos_nested_idx_diff);
x_full_nested = Rxx_nested(nested_cov_idx);

sv = steervec(pos_nested_reconstruct,ang);
P_full_nested = sv'*x_full_nested;

helperDisplayAngularSpectrum(ang,P_full_nested,theta_az,'Beamscan Spectrum for Nested Array');
```
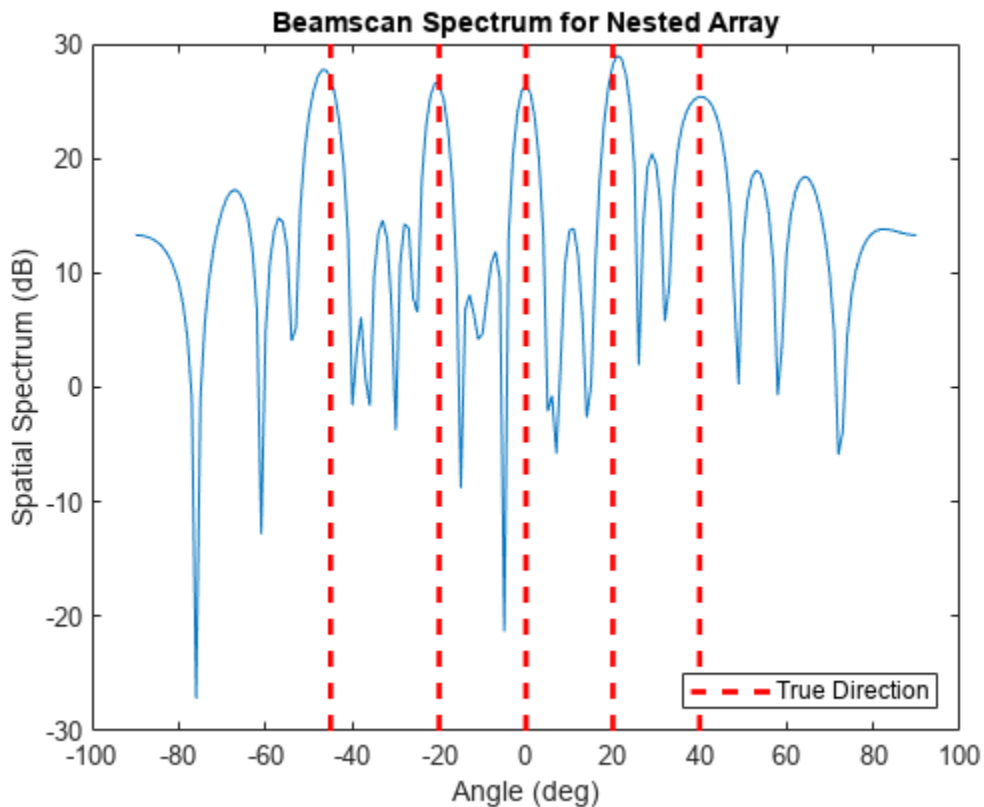


**Coprime Arrays**

Coprime array is another popular sparse array architecture. The idea of coprime array is to form the array with two subarrays whose element spacings are coprime to each other. The issue with a nested array is that it contains a dense array, making it more prone to mutual coupling effects between array elements. A coprime array is also easy to construct and on average it places elements further apart, thus alleviating the mutual coupling effects.

There are many proposed coprime arrays. This example uses the original coprime array architecture, often termed as the prototype coprime array. A linear prototype coprime array consists of two uniform linear arrays, one with Mc elements and the other Nc elements. Note that Mc and Nc must be coprime. These two arrays share the starting element. In addition, the element spacing of the Mc-element array is Nc*d and the element spacing of the Nc-element array is Mc*d. Assume Mc is 2 and Nc is 3, the sparse array architecture is

```
Mc = 4;
Nc = 3;
pos_cp_1 = (0:Mc-1)*Nc*d;
pos_cp_2 = (0:Nc-1)*Mc*d;
pos_cp = unique([pos_cp_1 pos_cp_2]);

helperDisplayArray(pos_cp,'Coprime Array');
```
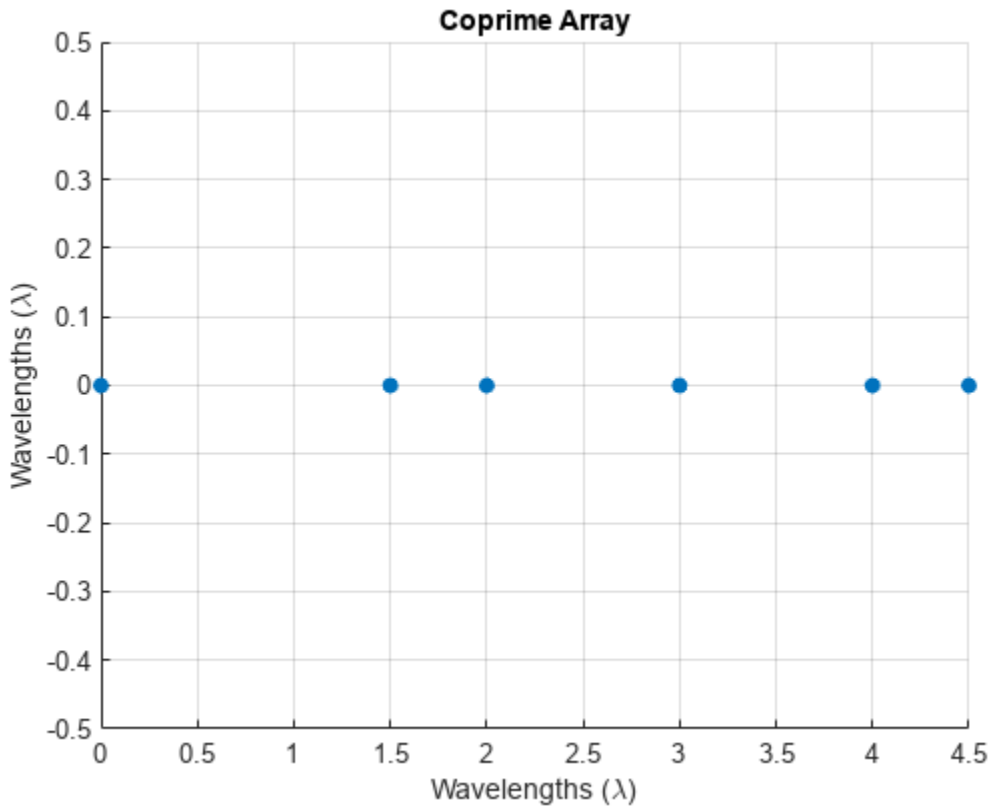


The degrees of freedom of the reconstructed array from a prototype coprime array can be computed as

```
N_cp = Mc+Nc;
dof_cp = 2*N_cp-1
```

```
dof_cp = 13
```

Let us verify if the degrees of freedom of the reconstructed array matches this.

```
pos_cp_idx = round(pos_cp/d);
pos_cp_idx_diff = pos_cp_idx-pos_cp_idx.';
pos_cp_idx_unique = sort(unique(pos_cp_idx_diff(:).'))
```

```
pos_cp_idx_unique = 1×17
```

| -9 | -8 | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 8 |
|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|

Notice that even though the reconstructed element lags seem to cover from -9 to 9, there is a gap between lag 6 and 8. Therefore, if we want to reconstruct a fully populated ULA, we can only go from

lag -6 to 6, matching the predicted degrees of freedom, 13. The reconstruction process is again similar to other sparse array architectures.

```
pos_cp_idx_reconstruct = -(dof_cp-1)/2:(dof_cp-1)/2;
pos_cp_reconstruct = pos_cp_idx_reconstruct*d;

x_cp = sensorsig(pos_cp,N_snap,theta_az,db2pow(-10));
Rxx_cp = x_cp'*x_cp/N_snap;
[~,cp_cov_idx] = ismember(pos_cp_idx_reconstruct(:),pos_cp_idx_diff);
x_full_cp = Rxx_cp(cp_cov_idx);

sv = steervec(pos_cp_reconstruct,ang);
P_full_cp = sv'*x_full_cp;

helperDisplayAngularSpectrum(ang,P_full_cp,theta_az,'Beamscan Spectrum for Copime Array');
```
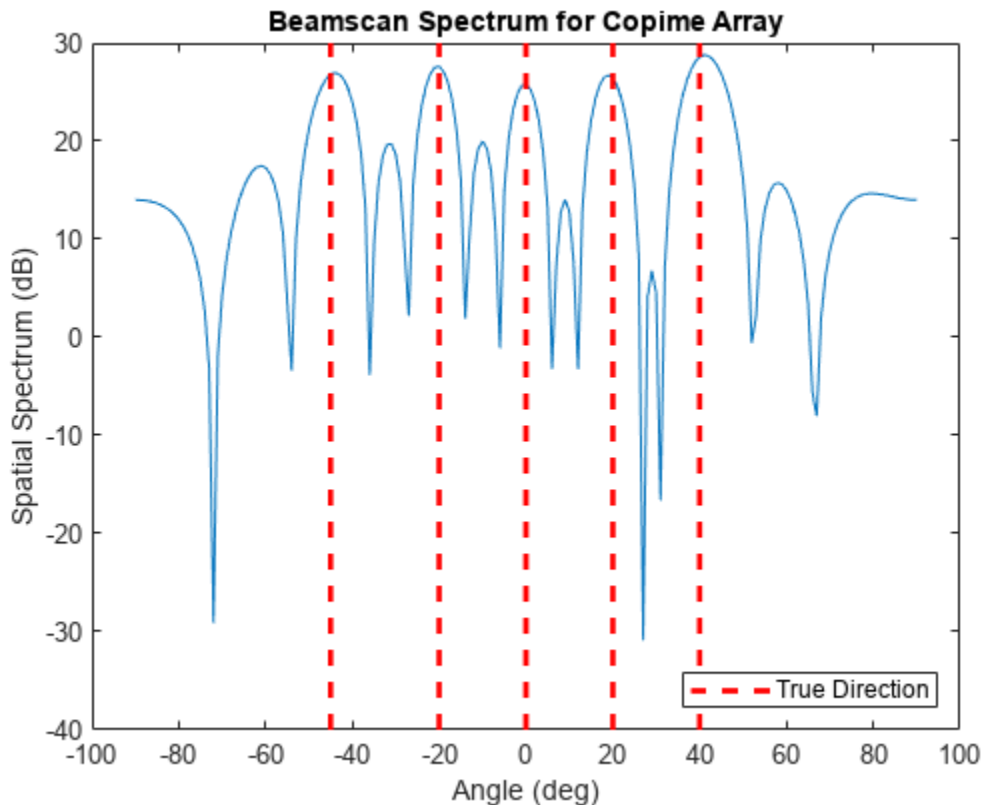


### DOA Estimation Using MUSIC and ESPRIT

Once we obtain the full array, we can just apply any known DOA estimation algorithm to the reconstructed signal measurements. However, it is worth noting that we used many snapshots of a sparse array to derive just one snapshot of the full array. Algorithms like MUSIC and ESPRIT rely on a good estimate of the covariance matrix of the full array, yet we only have one snapshot available. To overcome this issue, we can apply spatial smoothing to the resulting sample covariance matrix to restore enough rank for the DOA estimation, like what we do to handle coherent signal for those algorithms. This certainly reduces the degrees of freedom of the reconstructed array.

Next, use the fully reconstructed array from the coprime array and apply MUSIC algorithm to do the DOA estimation. The illustrated workflow is applicable to both MRA and the nested array.

```
R_full = x_full_cp*x_full_cp';
% spatial smoothing
ssfactor = (length(x_full_cp)+1)/2;
R_ss = spsmooth(R_full,ssfactor);
[doas_music,spec_music,ang_music] = musicdoa(R_ss,Nsig);

helperDisplayAngularSpectrum(ang_music,spec_music,theta_az,'MUSIC Spectrum for Coprime Array');
```
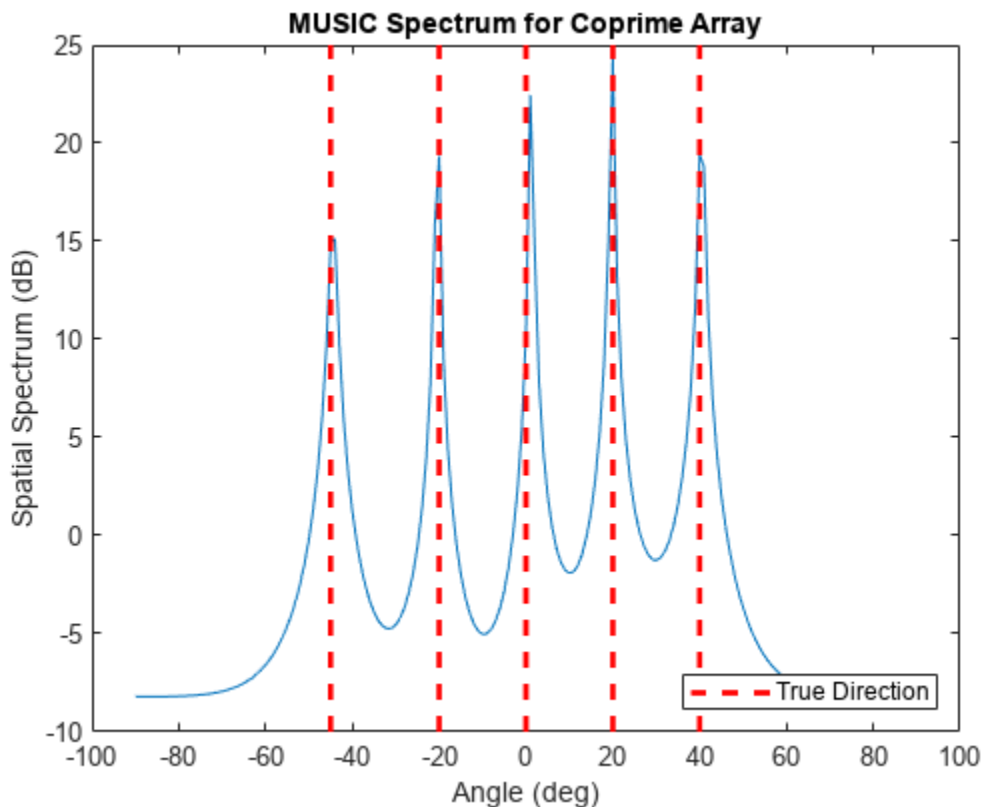


```
fprintf("The estimated directions are: %s",mat2str(sort(doas_music),2))
```

The estimated directions are: [-44 -20 1 20 40]

Applying ESPRIT algorithm follows the same recipe.

```
doa_esprit = espritdoa(R_ss,Nsig);
fprintf("The estimated directions are: %s",mat2str(sort(doa_esprit),2))
```

The estimated directions are: [-45 -20 1.3 20 40]

**DOA Estimation Using Compressed Sensing**

In general, in a DOA estimation scenario, the number of incoming directions is far less than the number of elements. Therefore, the problem is sparse. In addition, trying to identify these directions from a single snapshot also makes the problem underdetermined. Compressive sensing is a family of recently developed techniques that deal with such problems. In this section, we will use orthogonal

matching pursuit (OMP) algorithm, which is a basic greedy algorithm in the compressive sensing algorithm family, to perform DOA estimation.

The OMP algorithm relies on a data dictionary. In our case, it consists of steering vectors from all directions. Again, we will use the coprime array architecture to illustrate the workflow.

```
ang_cs = -90:0.1:90;
sv_dict = steervec(pos_cp_reconstruct,ang_cs);
```

Then, with a specified maximum sparsity, the OMP algorithm will find a set of steering vectors that best match the given signal. The angles corresponding to those steering vectors are then our DOA estimates.

```
spec_cs = 1e-3*ones(size(ang_cs)); %-60 dB floor
[p_sp,~,idx_sp] = ompdecomp(x_full_cp,sv_dict,'MaxSparsity',Nsig);
spec_cs(idx_sp) = p_sp;

helperDisplayAngularSpectrum(ang_cs,spec_cs,theta_az,'OMP Spectrum for Comprime Array');
```
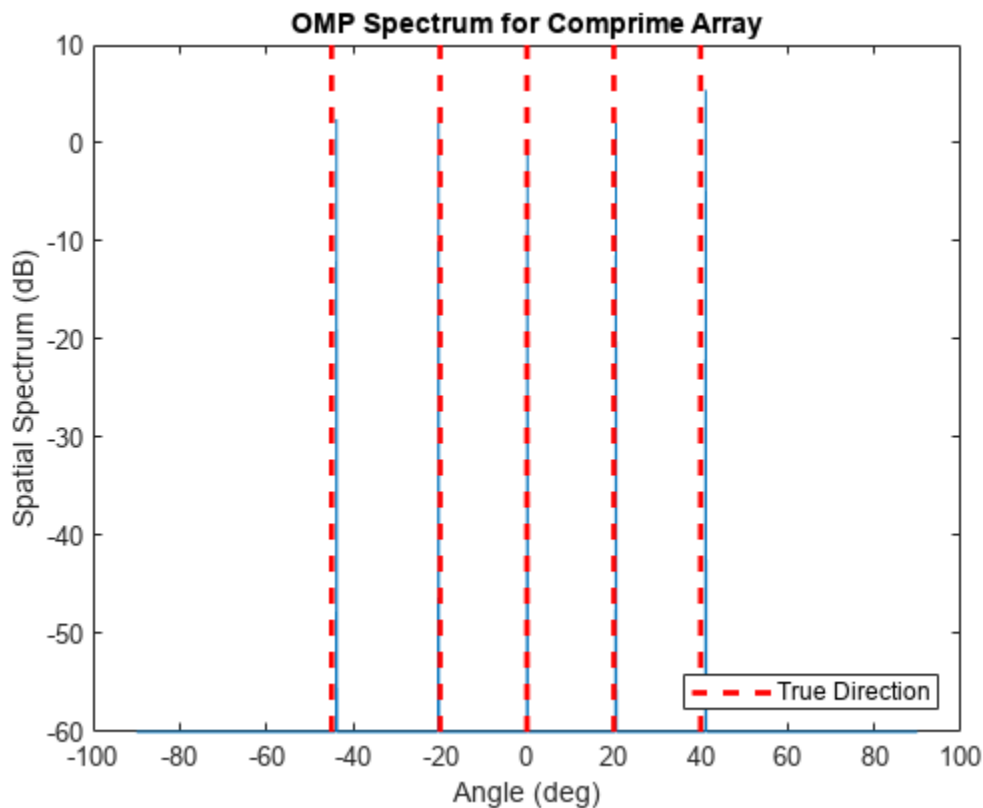


```
doa_cs = ang_cs(idx_sp);
fprintf("The estimated directions are: %s",mat2str(sort(doa_cs),2))
```

The estimated directions are: [-44 -20 0.1 20 41]

**Summary**

In this example, we illustrated several popular sparse array architectures, including MRA, nested arrays, and coprime arrays. We also showed how to reconstruct a full array out of these sparse arrays

and then perform either conventional beamscan DOA estimation algorithm or high-resolution DOA estimation algorithms like MUSIC and ESPRIT. We also included a brief discussion on estimating DOAs with compressive sensing using a single snapshot of the array measurements. Sparse arrays allow us to use a smaller number of elements to achieve resolutions, as well as degrees of freedom, of a much larger array. However, a sparse array needs to use many snapshots to reconstruct a single snapshot of the fully reconstructed array. Therefore, this approach may not work very well in rapidly changing environment.

### References

[1] Qin, Si, Zhang, Yimin D., and Amin, Moeness G. "Generalized coprime array configurations for direction-of-arrival estimation." *IEEE Transactions on Signal Processing* 63.6 (2015): 1377-1390.

[2] Vaidyanathan, P. P. and Pal, Piya. "Sparse sensing with coprime arrays." *2010 Conference record of the forty fourth Asilomar conference on signals, systems and computers*. IEEE, 2010.

[3] Raza, Ahsan, Liu, Wei, and Shen, Qing. "Thinned Coprime Arrays for DOA Estimation", European Signal Processing Conference, 2017

[4] Pal, Piya and Vaidyanathan, P. P. " Nested Arrays: A Novel Approach to Array Processing with Enhanced Degrees of Freedom" *IEEE Transactions on Signal Processing,* Vol. 58, No. 8, 2010

[5] Moffet, Alan, "*Minimum-Redundancy Linear Arrays*" IEEE Transactions on Antennas and Propagation, Vol. AP-16, No. 2, 1968

### Helper Functions

helperDisplayArray function

```
function helperDisplayArray(pos,titlestr)
    scatter(pos,zeros(size(pos)),'o','filled'); grid on
    xlabel('Wavelengths (\lambda)'); ylabel('Wavelengths (\lambda)');
    xlim([min(pos) max(pos)]);
    ylim([-0.5 0.5]);
    title(titlestr);
end
```

helperDisplayAngularSpectrum function

```
function helperDisplayAngularSpectrum(ang,spec,trueang,titlestr)
    plot(ang,mag2db(abs(spec)));
    hold on;
    yb = ylim;
    plot(ones(2,1)*trueang,yb(:)*ones(size(trueang)),'r--','LineWidth',2);
    xlabel('Angle (deg)');
    ylabel('Spatial Spectrum (dB)');
    title(titlestr);
    legend('','True Direction','Location','SouthEast');
    hold off;
end
```

# Distributed Beamforming Using 1-Bit Feedback

This example shows how to synchronize the frequency and phase of distributed nodes to enable beamforming. The nodes are synchronized using Gaussian Minimum Shift Keying (GMSK) modulated feedback signals from the base station that is receiving the beamformed continuous wave (CW) signals. The phases of the nodes are synchronized using a 1-bit feedback. The root mean square (RMS) amplitude of the beamformed signals is evaluated.

**Introduction**

Small distributed wireless systems have recently attracted researchers due to their ability to transmit and/or receive similar or higher power compared to bulky platform-centric systems. Distributed wireless systems can be formed by multiple user equipment (UE), CubeSats/satellites, and/or stationary antennas. Once the electric states of distributed wireless systems are synchronized, beamforming on either transmit or receive is possible leading to high power improvements. The main parameters that need to be synchronized are the frequency, phase, and time. Node synchronization can be achieved using a multitude of architectures; in particular, one can synchronize the beamforming nodes using only 1-bit feedback from the receiver, which can be referred to as base station.

In this example, coherent signals are transmitted towards a base station from four nodes. The synchronization is performed over multiple iterations, where the beamformed signals are continuously evaluated to generate the desired feedback to the nodes. The effects of phase noise and frequency random walks of the oscillators are taken into account. The frequency of every oscillator is considered stable for the duration of a given iteration. The base station and all the beamforming nodes are assumed to be time-aligned.

Feedback
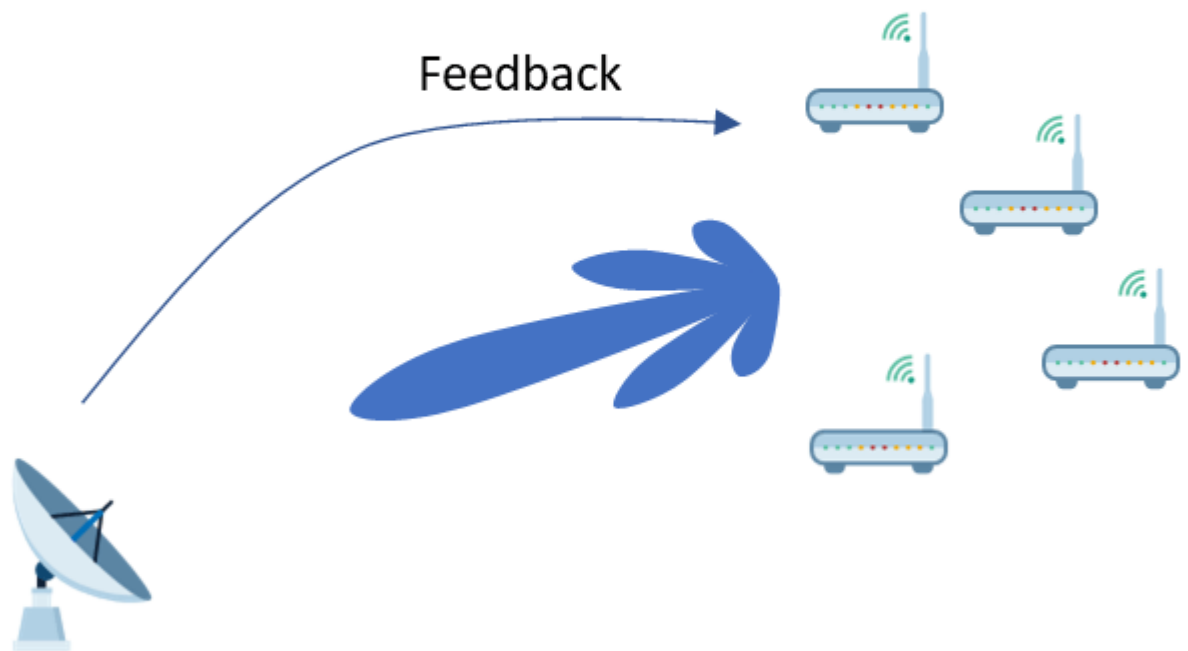
### Initialization

First, the parameters that are used in this example are initialized. The transmit carrier frequency of the base station and the receive carrier frequency of the beamforming nodes are both set to 964 MHz. These two carrier frequencies have to be equal so that the beamforming nodes can receive the synchronization signals from the base station. On the other hand, the transmit carrier frequency of the beamforming nodes and the receive carrier frequency of the base station are both set to 892 MHz. Two different bands were selected so that we can simulate the case where both synchronization and beamforming are performed simultaneously. The packet size of the GMSK synchronization signals is set to 150 bits. The baseband frequency of the beamformed CW signals is set to 1 kHz; and the sampling rate for the transmitted and received signals is set to 20 kHz.

```
rng('default');
nodeNbr = 4;              % The number of beamforming nodes
fcBsTx  = 964e6;          % Base station transmit (Tx) carrier frequency (Hz)
```

```
fcBnTx   = 892e6;         % Beamforming nodes Tx carrier frequency (Hz)
fcBsRx   = fcBnTx;        % Base station receive (Rx) carrier frequency (Hz)
fcBnRx   = fcBsTx;        % Beamforming nodes RX carrier frequency (Hz)
fbB      = 1e3;           % Baseband frequency of the beamformed signals (Hz)
fs       = 20e3;          % Sample rate (Hz)
packetSize      = 175; % GMSK packet size
samplesPerSymbol = 2;    % Samples per symbol for the GMSK signals
cwSamples       = 300; % Sample size of the CW signals
```

The phase correction for every node was performed by adding a random value on every iteration from a set of options using a uniform distribution. If the randomly added phases for all the nodes improved the RMS amplitude of the beamformed signals, the additional phases are retained, otherwise the added phases are reverted. The base station transmits the feedback about the change of the RMS amplitude to the beamforming nodes using a 1-bit feedback.

```
iterNbr        = 150;                    % The total number of iterations
rmsSigBBsRx    = zeros(iterNbr,1);       % RMS amplitude of beamformed signals
deltaRMSBs     = 0;                      % RMS amplitude change of the beamformed signals
bitFeedbackRX = ones(1,nodeNbr);        % Received bit feedback
deltaPhaseOptions = deg2rad([-15,15]); % Options for phase adjustments (rad)
```

To model the frequency drift at receive for the beamforming nodes relatively to the frequency of the base station transmitter, which is caused by the imperfections of real voltage-controlled oscillators (VCOs), every beamforming node is initialized with a random frequency offset. This frequency offset will be later bounded between +/- 2 kHz.

```
fOffsetBnRx       = zeros(iterNbr,nodeNbr);
fOffsetBnRx(1,:) = 404*randn(1,nodeNbr); % Frequency offset at the beamforming node Rx (Hz)
fOffsetBnRxMax   = 1e3;  % Maximum allowed frequency offset at the beamforming node Rx (Hz)
fOffsetBnRxMin   = -1e3; % Minimum allowed frequency offset at beamforming node Rx (Hz)
```

The frequency offset on receive is going to be estimated. Based on the estimated frequency offset, a certain frequency correction will be applied at receive and transmit. Similar to the frequency offset, every node has a phase offset relative to the phase of the base station. The phase offset changes based on the frequency offset between the nodes. Even when the nodes are frequency synchronized, a small frequency offset will exist. Thus, there will always be a random phase shift at the beginning of every iteration.

```
pfoGMSK = comm.PhaseFrequencyOffset(SampleRate=fs,...
    FrequencyOffsetSource="Input port");
pfoCW   = clone(pfoGMSK);
phaseOffsetBnTx = -pi + (2*pi)*rand(1,nodeNbr); % Initial phase offset (rad)

[fOffsetEstBnRx,fCorrBnRx,fCorrBnTx,fcBnTxAct,phaseCurrentCorrBnTx,...
    phaseTotalCorrBnTx,phaseBnTxAct,phaseOffsetActBnRx,phaseOffsetActBnTx,...
    phaseOffsetEstBnTx,phaseOffsetEstBnRx] = helperVarInitialization(iterNbr,nodeNbr);
```

Phase noise objects for the VCOs are initialized to be the same at transmit and receive regardless of whether the signal is upconverted or downconverted. The phase noise is only modeled for the beamforming nodes since the phase noise of the base station should not affect the RMS amplitude of the beamformed signals.

```
phNzLevel    = [-100,-140,-155];    % Phase Noise (dBc/Hz)
phNzFreqOff  = [1e2,1e3,9.5e3];     % Phase Offset (Hz)
pnoiseUpBS   = comm.PhaseNoise(Level=phNzLevel, ...
    FrequencyOffset=phNzFreqOff, SampleRate=fs);
pnoiseDownBS = comm.PhaseNoise(Level=phNzLevel, ...
```

```
    FrequencyOffset=phNzFreqOff, SampleRate=fs);
pnoiseUpBN   = clone(pnoiseUpBS);
pnoiseDownBN = clone(pnoiseDownBS);
```

The transmit and receive objects of the GMSK and Sinusoidal wave objects are initialized.

```
gmskModulatorData = comm.GMSKModulator('BitInput',true,'PulseLength',1,...
    'SamplesPerSymbol',samplesPerSymbol,'InitialPhaseOffset',0);
gmskDemodulator   = comm.GMSKDemodulator('BitOutput',false,'PulseLength',1,...
    'SamplesPerSymbol',samplesPerSymbol,'InitialPhaseOffset',0);
sine = dsp.SineWave('Frequency',fbB,'SampleRate',fs,...
    'PhaseOffset',0,'ComplexOutput',true,'SamplesPerFrame',cwSamples);
```

In addition, the header portion of the GMSK signal is initialized.

```
bitHeader            = repmat([1;1;1;1;1;0;0;1;1;0;1;0;1],[2,1]); % Packet header bits
gmskModulatorHeader = clone(gmskModulatorData);
headerSignal         = gmskModulatorHeader(bitHeader); % Signal portion of the header
```

The peak transmit power of the base station is set to 1 W with a gain of 40 dB, while the peak transmit power of the beamforming nodes is set to 0.1 W with a gain of 10 dB. The receivers were also modeled to have different gain and loss factors.

```
transmitterBs = phased.Transmitter('PeakPower',1,'Gain',40);
transmitterBn = phased.Transmitter('PeakPower',0.1,'Gain',10);
receiverBs    = phased.ReceiverPreamp('NoiseFigure',4, ...
    'SampleRate',fs,'NoiseComplexity','Real','LossFactor',10,'Gain',40);
receiverBn    = phased.ReceiverPreamp('NoiseFigure',7, ...
    'SampleRate',fs,'NoiseComplexity','Real','LossFactor',3,'Gain',10);
```

The location and velocity of all the nodes are selected.

```
posBs      = zeros(3,1);          % Position of the base station (m)
posBn      = repmat([0;0;0],[1,nodeNbr]); % Position of the beamforming nodes (m)
posBn(1,:) = 775 + 50.*rand(1,nodeNbr);
posBn(2,:) = -25 + 50.*rand(1,nodeNbr);
velBs      = zeros(3,1);          % Velocity of the base station (m/s)
velBn      = zeros(3,nodeNbr);   % Velocity of the beamforming nodes (m/s)

[~,angAzElBs] = rangeangle(posBn,posBs);
[~,angAzElBn] = rangeangle(posBs,posBn);

figure
scatter(posBs(1),posBs(2),100,'filled')
hold on; grid on
scatter(posBn(1,:),posBn(2,:),25,'filled')
xlim([-100,900]); ylim([-300,300])
legend('Base Station','Beamforming Nodes')
xlabel ('x Coordinates (m)'); ylabel ('y Coordinates (m)')
```
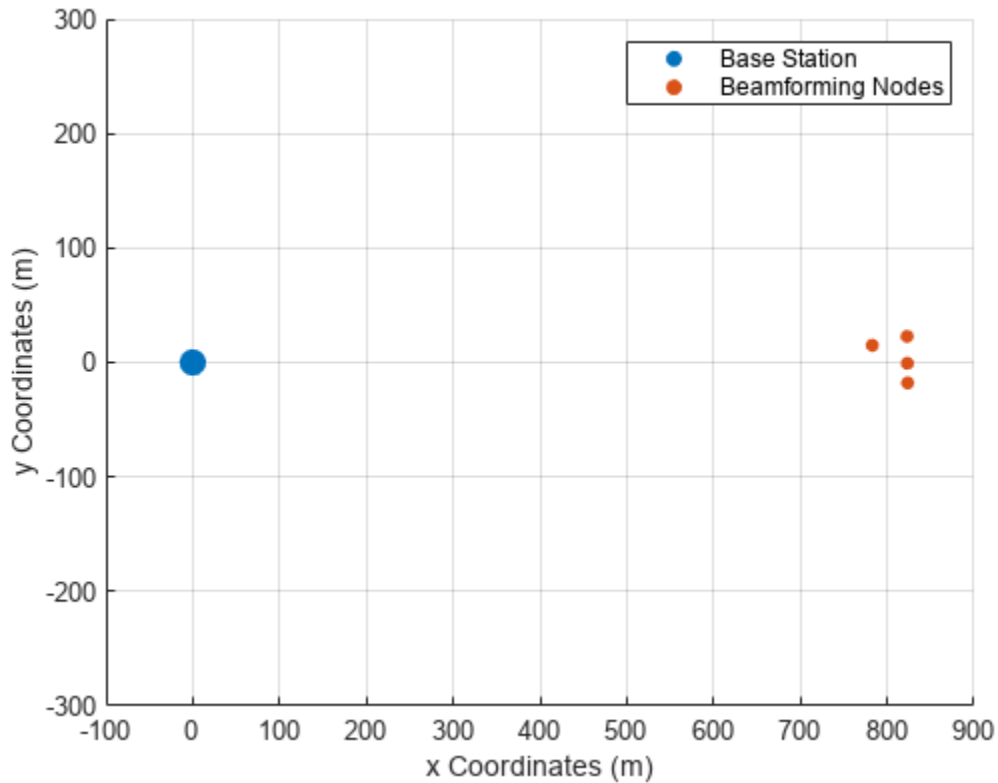
The antennas of the base station were selected as uniform rectangular arrays, while the antennas of the beamforming nodes were selected as isotropic antenna elements.

```
antennaBn    = phased.IsotropicAntennaElement;
antennaBs    = phased.URA('Size',[3 3]);

radiatorBs   = phased.Radiator('Sensor',antennaBs,...
    'OperatingFrequency',fcBsTx,'CombineRadiatedSignals',true);
radiatorBn   = phased.Radiator('Sensor',antennaBn,...
    'OperatingFrequency',fcBnTx);

collectorBn  = phased.WidebandCollector('Sensor',antennaBn,...
    'SampleRate',fs,'CarrierFrequency',fcBnRx);
collectorBs  = phased.WidebandCollector('Sensor',antennaBs,...
    'SampleRate',fs,'CarrierFrequency',fcBnRx);

beamformerBs = phased.PhaseShiftBeamformer('SensorArray',antennaBs,...
    'OperatingFrequency',fcBsRx,'Direction',[0;0]);

henvBsTx     = phased.FreeSpace('SampleRate',fs,...
    'OperatingFrequency',fcBsTx);
henvBnTx     = phased.FreeSpace('SampleRate',fs,...
    'OperatingFrequency',fcBnTx);
```

Beamforming performance is evaluated over 150 iterations.

```
for iter = 1:iterNbr
```

**Synchronization of the Beamforming Nodes**

Beamforming is enabled by first synchronizing the frequencies and phases of the 4 beamforming nodes. The nodes are assumed to be time-aligned. The beamforming nodes synchronize themselves based on the feedback signal transmitted from the base station. The feedback is represented by a GMSK signal with a known header, 1-bit feedback, and a payload with random bits. The beamforming nodes synchronize their frequencies to the center frequency of the received feedback signal. The 1-bit feedback is used to decide whether the last update to the phases increased the beamformed power or decreased it. If there was an improvement in the received power at the base station, the last phase updates are kept; otherwise they are reversed. A block diagram summarizing the main steps for the process of synchronization and beamforming is shown below.
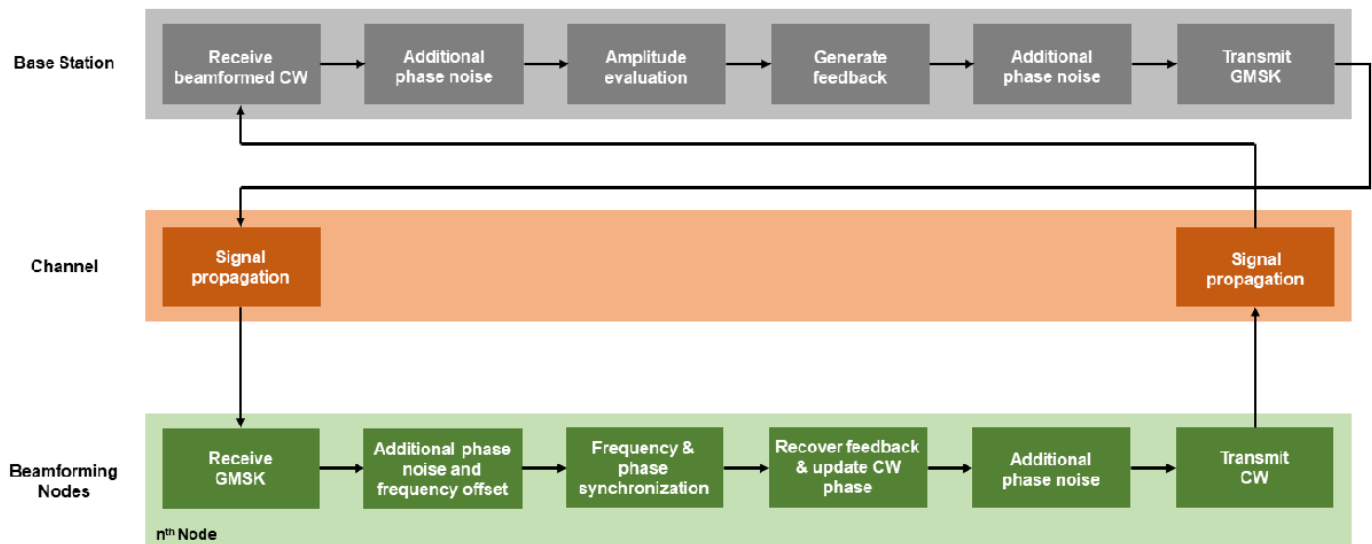


The signal packet of the synchronization signals transmitted from the base station contains header bits, a feedback bit, and a payload that contains random bits.

```
bitFeedback = deltaRMSBs >= 0;
bitRemainingPayload = randi([0 1],packetSize - length([bitHeader;bitFeedback]),1);
dataBsTx     = [bitHeader;bitFeedback;bitRemainingPayload];
```

The GMSK signal is generated for the desired packet and additional phase noise from the VCO of the base station transmitter is added. Afterwards, the signal is transmitted.

```
gmskBsTx = gmskModulatorData(dataBsTx);
gmskBsTx = pnoiseUpBS(gmskBsTx);
gmskBsTx = transmitterBs(gmskBsTx);
gmskBsTx = radiatorBs(gmskBsTx,angAzElBs);
```

The frequency offset of every beamforming node is updated for every iteration to model the random frequency walk/drift of a VCO.

```
fOffsetBnRx(iter,:) = fOffsetBnRx(max(1,iter - 1),:) + 2*randn(1,nodeNbr);
fOffsetBnRx(iter,:) = min(fOffsetBnRxMax,max(fOffsetBnRxMin,fOffsetBnRx(iter,:)));
```

Once the transmitted synchronization signals are received by the beamforming nodes, they are processed to extract the frequency, phase, and feedback bit. As a first step, the correction to the

receiver frequency of the beamforming node is computed from the correction applied to the transmit frequency. Afterwards, the actual received frequency for every node is identified.

```
fCorrBnRx = fCorrBnTx(max(1,iter - 1),:).*(fcBnRx/fcBnTx);
fcBnRxAct = fcBnRx + fOffsetBnRx(iter,:) + fCorrBnRx;
```

Additional phase offset is added on receive to make it non-coherent on receive.

```
phaseOffsetActBnRx = 2*pi*rand(1,nodeNbr);
```

Afterwards, every beamforming node should process the synchronization signals and extract the desired frequency and phase offsets that are needed and then extract the feedback bit to decide whether to keep the last applied phase correction.

```
sigBBnRx = complex(zeros(length(gmskBsTx),nodeNbr));
for node = 1:nodeNbr
```

Receive the signals and add the channel effects.

```
    gmskBsTx(:,node) = henvBsTx(gmskBsTx(:,node),posBs,posBn(:,node),velBs,velBn(:,node));
    gmskBsTx(:,node) = collectorBn(gmskBsTx(:,node),angAzElBn(:,node));
    gmskBsTx(:,node) = receiverBn(gmskBsTx(:,node));
```

Frequency and phase offsets are added to the received signals to model the mismatch with the characteristics of the base station. As has been described in [1], the complex envelope of a GMSK signal can be defined as

$$s(t) = \exp\left(j\frac{\pi}{2}\sum_i a_i q(t - iT)\right)$$

where $a_i$ is the symbol stream, $q(t)$ is the Gaussian phase pulse, and $T$ is the symbol period. In the presence of VCO frequency mismatch, the received signal over an AWGN channel can be modeled as

$$r(kT_s) = \exp(j(wkT_s + \phi)) \cdot \exp\left(j\frac{\pi}{2}\sum_i a_i q(t - ikT_s)\right) + v(kT_s)$$

where $w$ and $\phi$ are the VCO frequency and phase offsets respectively, $T_s$ is the sample period, $k$ is the sample number, and $v(kT_s)$ represents the noise term. If the receiver of the beamforming node has a higher carrier frequency than the transmitter of the base station, the demodulated baseband signal will have a lower frequency than what is anticipated.

```
    sigBBnRx(:,node) = pfoGMSK(gmskBsTx(:,node),fcBsTx - fcBnRxAct(node));
    sigBBnRx(:,node) = exp(1j*phaseOffsetActBnRx(node)).*sigBBnRx(:,node);
```

Additional phase noise from the VCO of the receiver is taken into account.

```
    sigBBnRx(:,node) = pnoiseDownBN(sigBBnRx(:,node));
```

Estimating the frequency offset of $r(kT_s)$ is possible when the following non-linear transformation is applied:

$$z[k] = (-1)^k r^2(kPT_s)$$

where $P$ is the over-sampling rate such that $T = PT_S$. As shown in [2] and [3], $z[k]$ can be approximated as a discrete time sine wave with its frequency equal to twice the frequency offset in $r(kPT_S)$:

$$z[k] \approx A\exp(j(2wkPT_S + 2\phi)) + w[k]$$

where $w[k]$ represents the noise term. The frequency of $z[k]$ is estimated from its discrete Fourier transform.

```
fOffsetEstBnRx(node) = helperFrequencyEstimation(sigBBnRx(:,node), fs, samplesPerSymbol);
```
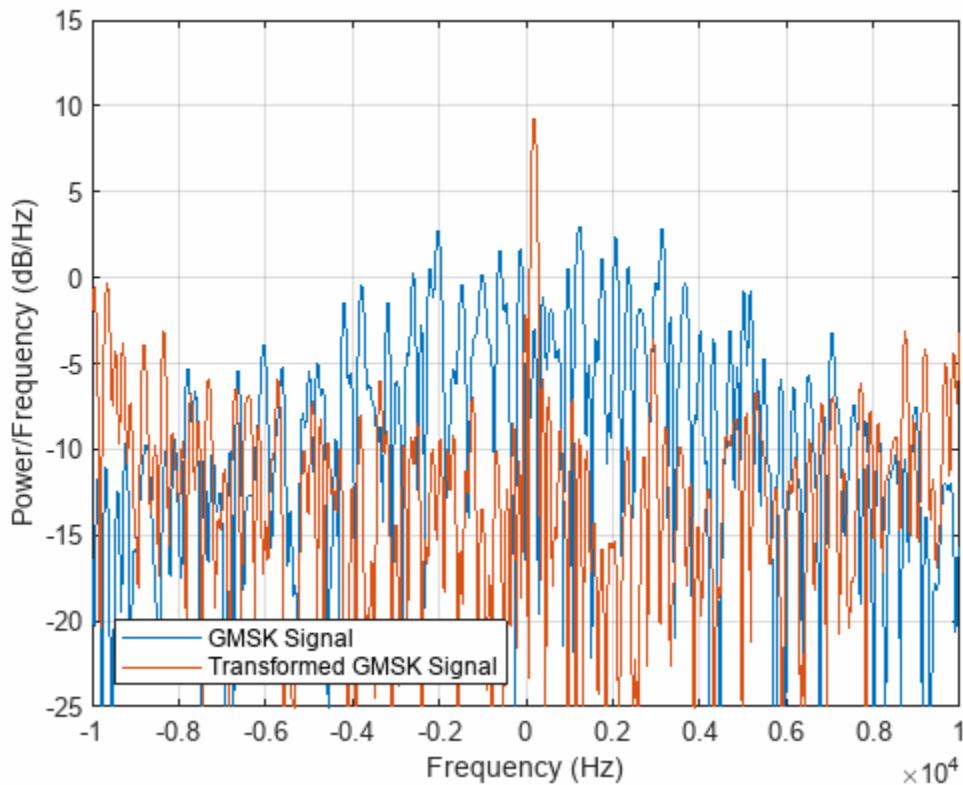


The Phase offset is estimated by first performing the following correlation [1]:

$$\gamma[k] = \sum_{n=1}^{N} r'[k-n]h[n]$$

where $r'[n]$ represents the frequency-compensated signal, and $h[n]$ represents the known header signal. The phase offset is then computed from $\gamma[k]$ when $|\gamma[k]|$ is maximum; at this point, $r'[n]$ and $h[n]$ are aligned.

```
k       = 0:length(sigBBnRx(:,node)) - 1;
fCWave = exp(-1j*2*pi*fOffsetEstBnRx(node).*(k'./fs)).*sigBBnRx(:,node);

phaseOffsetEstBnRx(node) = helperPhaseEstimation(fCWave, headerSignal, samplesPerSymbol);
```

The received bits are then demodulated from the received feedback signal.

```
    release(gmskDemodulator);
    gmskDemodulator.InitialPhaseOffset = -phaseOffsetEstBnRx(node);
    receivedData = gmskDemodulator(fCWave);
```

It is then determined if the beamformed signals had an increase or decrease in RMS amplitude based on the last phase updates. The received bits are correlated with the known packet header to determine the location of the feedback bit.

```
    bitHeaderTransformation = bitHeader;
    bitHeaderTransformation(bitHeaderTransformation == 0) = -1;
    [~,corrWithBiHeaderLoc] = max(xcorr(receivedData,bitHeaderTransformation));
    headerFirstBitLoc       = corrWithBiHeaderLoc - length(receivedData) + 1;
    feedbackBitLoc          = min(packetSize, headerFirstBitLoc + length(bitHeader));

    bitFeedbackRX(node)     = receivedData(feedbackBitLoc);
end
```

The last added phase shift is kept or reverted depending on whether it improved the total received power at the base station or not.

```
phaseTotalCorrBnTx(iter,:) = phaseTotalCorrBnTx(max(1,iter - 1),:) - ...
    phaseCurrentCorrBnTx.*(bitFeedbackRX == -1);
```

**Transmission of the Coherent CW Signals from the Beamforming Nodes to the Base Station**

Once the beamforming nodes are synchronized, they transmit CW signals to the base station. The initial phase of the transmitted signals is randomly increased or decreased by 15 degrees. If the phase updates resulted in an increased received power at the receiver of the base station, the updates are maintained, otherwise they are discarded.

```
cwBnTx = complex(zeros(cwSamples,nodeNbr));
```

Random phase updates are added for every beamforming node based on the phase update options in `DeltaPhaseOptions`.

```
phaseCurrentCorrBnTx = deltaPhaseOptions(randi(numel(deltaPhaseOptions),1,nodeNbr));
phaseTotalCorrBnTx(iter,:) = phaseTotalCorrBnTx(iter,:) + phaseCurrentCorrBnTx;
phaseBnTxAct(iter,:) =  mod(phaseOffsetBnTx + phaseTotalCorrBnTx(iter,:),2*pi) - pi;
```

The carrier frequency correction, carrier frequency offset, and actual transmitted carrier frequency of the beamforming nodes are calculated. In addition, the estimated and actual phase offset of the transmitters are calculated.

```
fCorrBnTx(iter,:) = fCorrBnTx(max(1,iter - 1),:) + fOffsetEstBnRx.*(fcBnTx/fcBnRx);
fOffsetBnTx = fOffsetBnRx(iter,:).*(fcBnTx/fcBnRx);
fcBnTxAct(iter,:) = fcBnTx + fOffsetBnTx + fCorrBnTx(iter,:);

phaseOffsetEstBnTx(iter,:) = phaseOffsetEstBnRx.*(fcBnTx/fcBnRx);
phaseOffsetActBnTx(iter,:) = phaseOffsetActBnRx.*(fcBnTx/fcBnRx);

for node = 1:nodeNbr
```

The CW signals are generated for every beamforming node.

```
    release(sine);
    sine.PhaseOffset = phaseBnTxAct(iter,node);
    modSignalBnTx    = sine();
```

Compensating for phase and frequency offsets that were observed from the synchronization signals. There might be an additional phase biases to PhaseOffsetActBnTx, but any additional phase shift will be equal for all the nodes.

```
    cwBnTx(:,node) = exp(1j*(-phaseOffsetActBnTx(iter,node) ...
        + phaseOffsetEstBnTx(iter,node))).*modSignalBnTx;
    cwBnTx(:,node) = pfoCW(cwBnTx(:,node),fcBnTxAct(iter,node) - fcBsRx);
```

The generated CW signals are transmitted and the phase noise from the VCO is added.

```
    cwBnTx(:,node) = pnoiseUpBN(cwBnTx(:,node));
    cwBnTx(:,node) = transmitterBn(cwBnTx(:,node));
    release(radiatorBn);
    radiatorBn.OperatingFrequency = fcBnTxAct(iter,node) + fbB;
    cwBnTx(:,node) = radiatorBn(cwBnTx(:,node),angAzElBn(:,node));
```

Adding channel effects to the received signals.

```
    release(henvBnTx);
    henvBnTx.OperatingFrequency = fcBnTxAct(iter,node) + fbB;
    cwBnTx(:,node) = henvBnTx(cwBnTx(:,node),posBn(:,node),posBs, ...
        velBn(:,node),velBs);
end
```

The beamformed signal at the base station is computed.

```
cwBsRx = collectorBs(cwBnTx,angAzElBs);
cwBsRx = beamformerBs(cwBsRx);
cwBsRx = receiverBs(cwBsRx);
```

Additional phase noise from the Rx VCO of the base station.

```
sigBBsRx = pnoiseDownBS(cwBsRx);
```

Normalized RMS amplitude of the beamformed signal at the base station.

```
rmsSigBBsRx(iter) = rms(sigBBsRx)/...
    (nodeNbr*rms(receiverBs(reshape(cwBnTx,[],1))));
```

The maximum RMS amplitude for the previous 4 captured signals at the base station is computed to be used as a reference when evaluating the RMS amplitude of the current received signals.

```
maxPrevRMSSigBBsRx = max(rmsSigBBsRx(max(1,iter - 4):max(1,iter - 1)));
```

The difference between the current RMS amplitude and the maximum RMS amplitude of the previous 4 received signals is computed.

```
deltaRMSBs = rmsSigBBsRx(iter) - maxPrevRMSSigBBsRx;
end
```
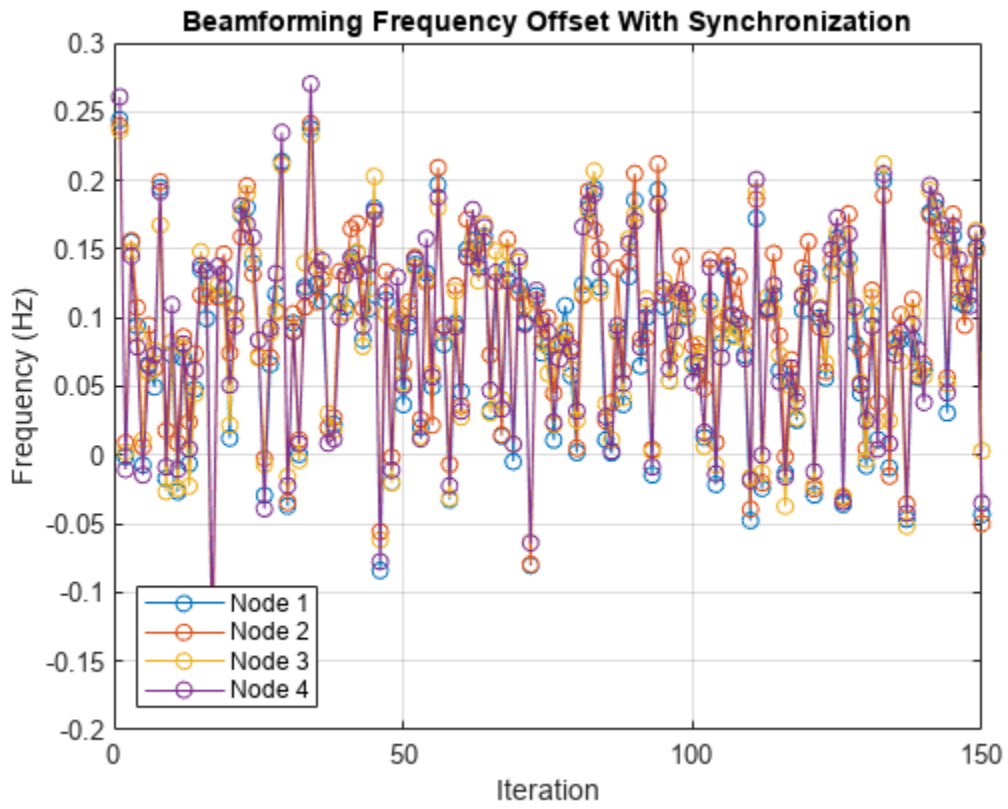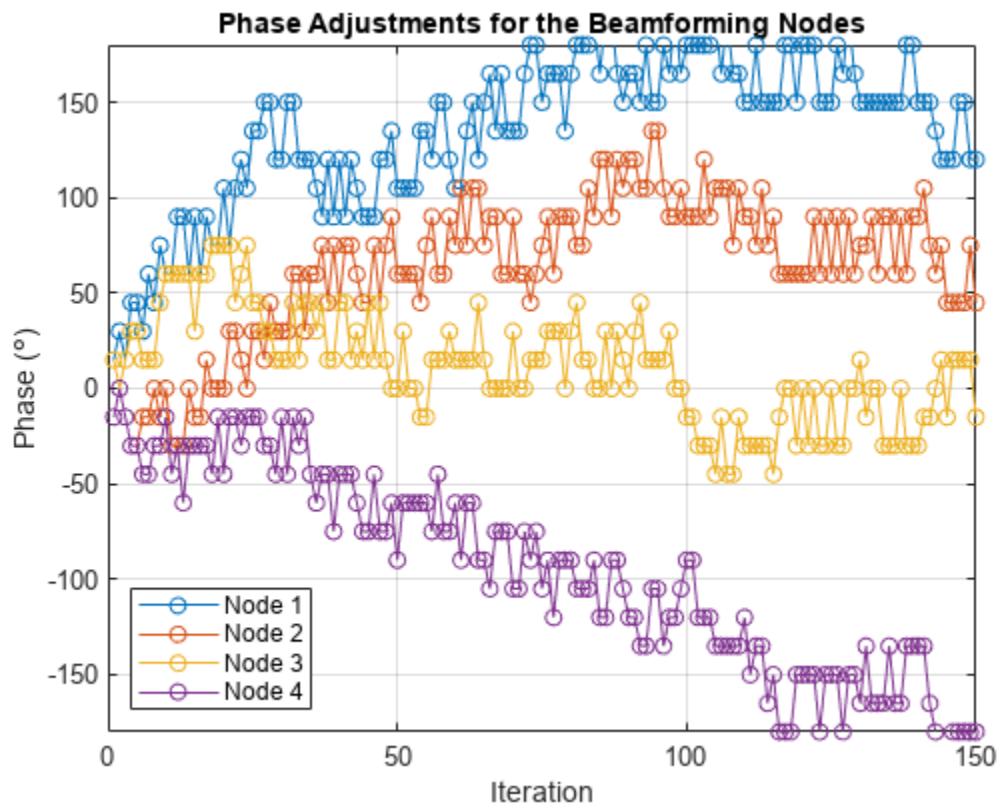
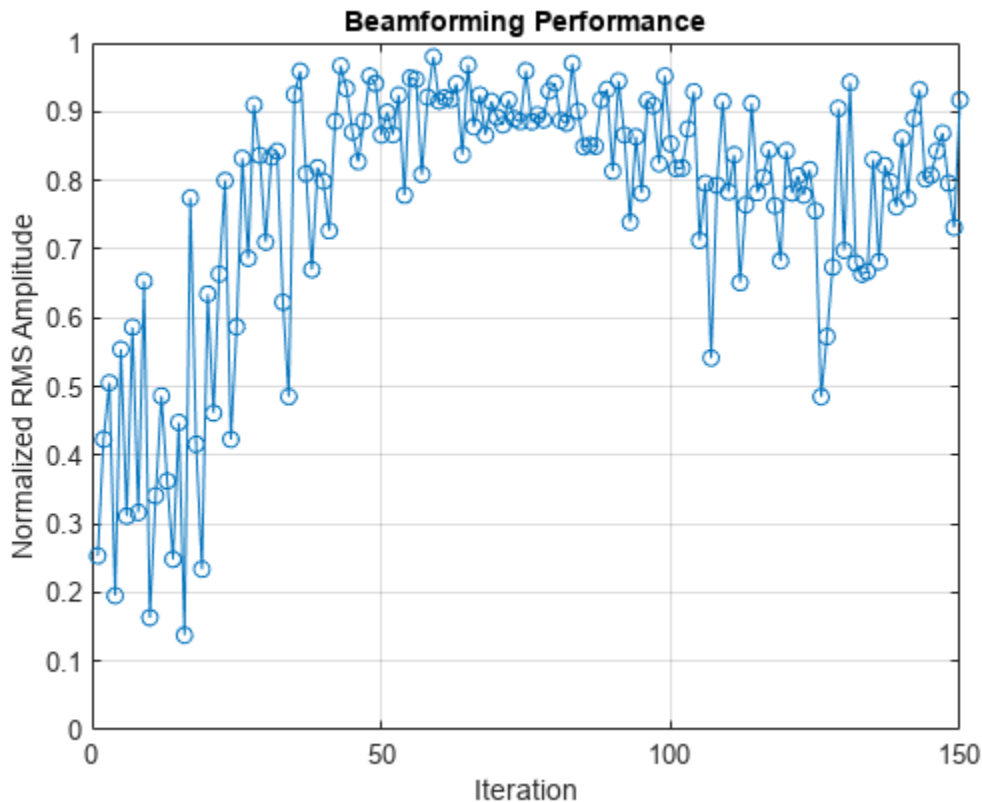**Evaluation of the Beamforming Results**

A helper function is used to plot the node characteristics and the results.

```
helperPlotResults(iter,rmsSigBBsRx,phaseTotalCorrBnTx,fcBnTx,fcBnTxAct,fOffsetBnRx);
```

Beamforming Frequency Offset Without Synchronization

Phase Adjustments for the Beamforming Nodes
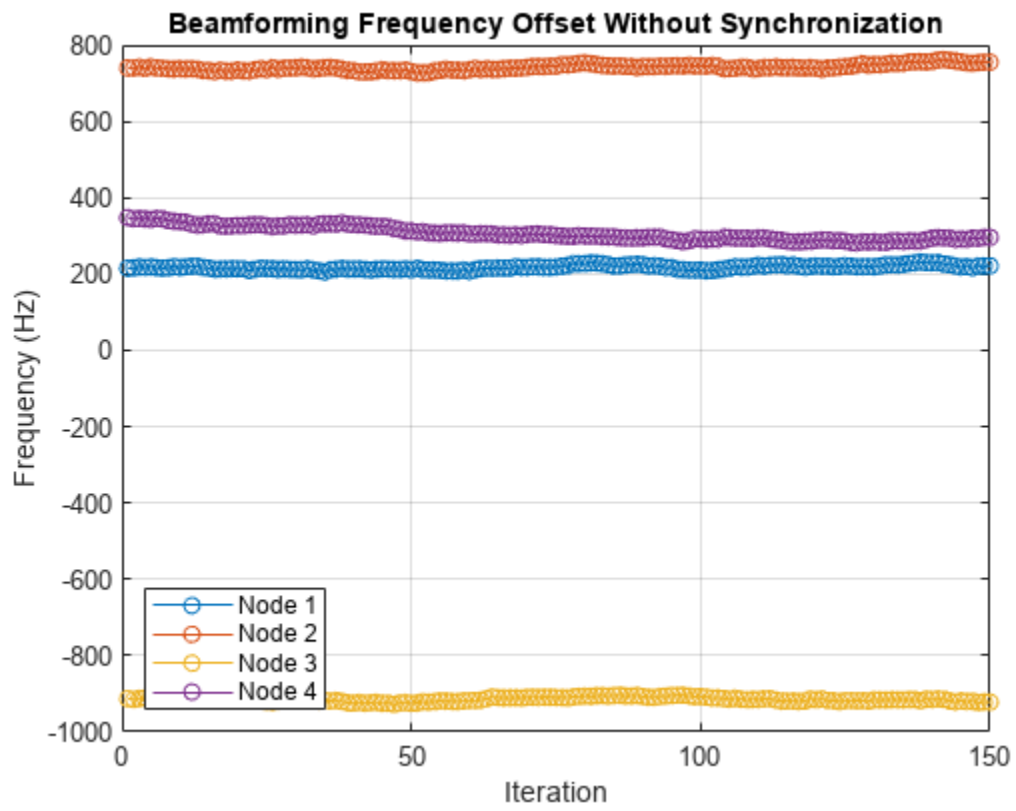
To show the importance of the coherent summation of the signals on receive, the transmitted CW signals were evaluated for 4 and 16 distributed nodes where the synchronization of frequency and phase was not achieved. When the signals are summed coherently, which was the case so far, the received amplitude increases as a multiple of $N$, where $N$ is the number of nodes. This increase in amplitude is observed once the nodes are fully synchronized, which sometimes can be performed in few iterations while in other times it can take more than a hundred iterations. On the other hand, the amplitude increases by a factor of $\sqrt{(N)}$ for the case of incoherent summation. The results of incoherent summation are shown below. The RMS amplitude of the received signals was normalized in reference to the case where all the signals are added coherently.

```
rmsSigBBsRxNonCoherent4  = helperIncoherentTransmission(4);
rmsSigBBsRxNonCoherent16 = helperIncoherentTransmission(16);

figure; plot([rmsSigBBsRxNonCoherent4,rmsSigBBsRxNonCoherent16],'-o')
grid on; ylim([0 1])
title('Summation of Incoherent Signals')
xlabel('Iteration'); ylabel('Normalized RMS Amplitude')
legend('4 Nodes','16 Nodes','Location','northeast')
```

## Summation of Incoherent Signals
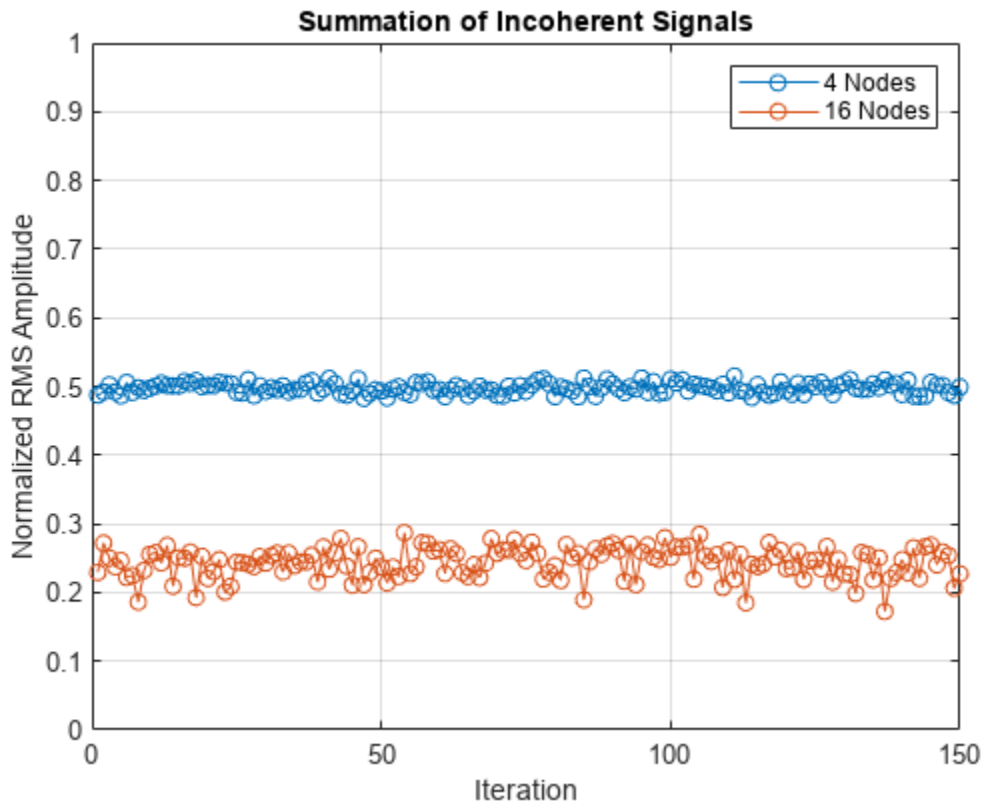


As can be seen, when 4 nodes are considered, the received amplitude at the base station is on average half of the maximum achievable amplitude. In the case of 16 nodes, the RMS amplitude for noncoherent summation is on average 25% of the achievable maximum RMS amplitude if the nodes are fully synchronized. The importance of coherent beamforming is clear once the beamforming nodes are fully synchronized, especially if a high number of nodes is used.

**Summary**

This example shows how to synchronize distributed nodes to enable beamforming. Frequency and phase synchronization were of interest, while the beamforming nodes were assumed to be time aligned. The example shows how to estimate the center frequency and phase of GMSK signals. In addition, a 1-bit feedback was extracted from the synchronization signals and was used to determine the appropriate adjustment to the phase of the transmitted coherent signals from every node. The beamforming results were compared to the case where the transmitted signals were summed incoherently.

**Reference**

[1] F. Quitin, M. M. U. Rahman, R. Mudumbai, and U. Madhow, "A scalable architecture for distributed transmit beamforming with commodity radios: Design and proof of concept," IEEE Trans. Wireless Commun., vol. 12, no. 3, pp. 1418–1428, Mar. 2013.

[2] M. Morelli and U. Mengali, "Feedforward carrier frequency estimation with MSK-type signals," IEEE Commun. Lett., vol. 2, no. 8, pp. 235– 237, Aug. 1998.

[3] H. Peng, J. Li, and L. Ge, "Non-data-aided carrier frequency offset estimation of GMSK signals in burst mode transmission," in Proc. 2003 IEEE International Conf. Acoustics, Speech, Signal Process., vol. 4, pp. 576–579.

[4] S. Prager, M. S. Haynes, and M. Moghaddam, "Wireless subnanosecond RF synchronization for distributed ultrawideband software-defined radar networks," IEEE Trans. Microw. Theory Techn., vol. 68, no. 11, pp. 4787–4804, Nov. 2020.

**Supporting Functions**

**helperVarInitialization Function**

```
function [fOffsetEstBnRx,fCorrBnRx,fCorrBnTx,fcBnTxAct,phaseCurrentCorrBnTx,...
    phaseTotalCorrBnTx,phaseBnTxAct,phaseOffsetActBnRx,phaseOffsetActBnTx,...
    phaseOffsetEstBnTx,phaseOffsetEstBnRx] = helperVarInitialization(iterNbr, nodeNbr)

% Helper function that initializes part of the variables.

fOffsetEstBnRx = zeros(1,nodeNbr); % Estimated Frequency offset by each node (Hz)
fCorrBnRx  = zeros(1,nodeNbr); % Frequency added at the beamforming node RX to correct for the o
fCorrBnTx  = zeros(iterNbr,nodeNbr); % Frequency added at beamforming node Tx to correct for the
fcBnTxAct  = zeros(iterNbr,nodeNbr); % Actual Tx carrier frequency for the beamforming nodes (Hz
phaseCurrentCorrBnTx = zeros(1,nodeNbr); % Current phase increment added at the beamforming node
phaseTotalCorrBnTx = zeros(iterNbr,nodeNbr); % Total phase added at the beamforming node Tx to co
phaseBnTxAct       = zeros(iterNbr,nodeNbr); % phaseOffsetBnTx + phaseTotalCorrBnTx (rad)
phaseOffsetActBnRx = zeros(1,nodeNbr); % Actual phase offset of incoming synchronization signals
phaseOffsetActBnTx = zeros(iterNbr,nodeNbr); % Actual phase offset of transmitted CW signals fro
phaseOffsetEstBnTx = zeros(iterNbr,nodeNbr); % Estimated phase offset of transmitted CW signals
phaseOffsetEstBnRx = zeros(1,nodeNbr); % Estimated phase offset of incoming synchronization signa
end
```

**helperFrequencyEstimation Function**

```
function fOffsetEst = helperFrequencyEstimation(wave, fs, samplesPerSymbol)

% Helper function that estimates the frequency of the GMSK signals.

% Waveform transformation to extract the frequency shift
% fs should be high enough to support high samplesPerSymbol values in z
z = (-1).^(0:(length(wave)/samplesPerSymbol) -1).*(wave(1:samplesPerSymbol:end).').^2;

zPadded  = [z(:);zeros(length(z)*7,1)];
freqBbRX = fft(zPadded);
freqShiftedBbRX    = fftshift(freqBbRX);
freqShiftedAbsBbRX = abs(freqShiftedBbRX);
l  = length(zPadded);
dF = fs/l;
f  = (-fs/2:dF:fs/2 - dF);

% Locating the peak of the spectrum
[peak,locs] = findpeaks(freqShiftedAbsBbRX);
[~,maxPeakIndex] = max(peak);
maxPeakLoc = locs(maxPeakIndex);

% Ratio of the original received waveform length over the length of the padded waveform for FFT 
oWLvsPWL    = length(wave)/length(freqShiftedAbsBbRX);
% Refinement of the frequency estimate
[freqEs, ~] = helperNLLSCurveFitting(f, freqShiftedAbsBbRX, maxPeakLoc, oWLvsPWL);
```

```
% Frequency of the downconverted GMSK signal
fOffsetEst  = freqEs/(2*samplesPerSymbol);
end
```

**helperPhaseEstimation Function**

```
function refinedTheta = helperPhaseEstimation(fCWave, headerSignal, samplesPerSymbol)

% Helper function that estimates the initial phase of the received signals.

% Correlation of frequency-compensated waveform with the header
gamma        = xcorr(fCWave,headerSignal);
% Peak of gamma
gammaAbs     = abs(gamma);
gammaSamples = 1:length(gamma);
[peak,locs]  = findpeaks(gammaAbs);
[~,maxPeakIndex] = max(peak);
maxPeakLoc   = locs(maxPeakIndex);
[~, newPeak] = helperNLLSCurveFitting(gammaSamples, gammaAbs, maxPeakLoc, 1/samplesPerSymbol);

% Angle estimation from gamma
theta          = angle(gamma);
thetaUnwrapped = unwrap(theta);
% Angle estimate refinement using linear interpolation
[refinedThetaUnwrapped] = interp1(1:length(thetaUnwrapped),thetaUnwrapped,newPeak,'linear');
refinedTheta   = mod(refinedThetaUnwrapped,2*pi);
end
```

**helperNLLSCurveFitting Function**

```
function [refinedPeakSamplePoint, refinedMaxPeakLoc, meanDeltaY] = helperNLLSCurveFitting(sampleI

% Helper function that estimates accurately the peak from three discrete points. Inspired by [4]
% oWLVsPWL represents the original waveform length over the padded waveform length.

sampleValues = sampleValues(:);

NLLSidx = 15;
l0 = zeros(1,NLLSidx);
l1 = zeros(1,NLLSidx);
l2 = zeros(1,NLLSidx);

x = (-1:1).';

l0(1) = sampleValues(maxPeakLoc);
l1(1) = 0.1;
l2(1) = oWLVsPWL;

y = sampleValues(maxPeakLoc + x);

for m = 1:NLLSidx - 1
    arg = l2(m).*(x - l1(m));

    func1 = sinc(arg);
    func2 = l0(m).*(sinc(arg) - cos(pi.*arg))./(x - l1(m));
    func3 = l0(m).*(cos(pi.*arg) - sinc(arg))./l2(m);
    j = [func1 func2 func3];
```

```
    func   = l0(m).*sinc(arg);

    deltaY = y - func;

    deltaL = (j'*j)\j'*deltaY;

    l0(m + 1) = l0(m)+deltaL(1);
    l1(m + 1) = min(0.6, max(-0.6,l1(m) + deltaL(2)));
    l2(m + 1) = l2(m) + deltaL(3);
end

meanDeltaY = mean(deltaY);

deltaSamplePoints     = samplePoints(2) - samplePoints(1);
refinedMaxPeakLoc     = maxPeakLoc + l1(NLLSidx);
refinedPeakSamplePoint = samplePoints(maxPeakLoc) + deltaSamplePoints*l1(NLLSidx);
end
```

**helperPlotResults Function**

```
function helperPlotResults(iter, rmsSigBBsRx, phaseTotalCorrBnTx, fcBnTx, fcBnTxAct, fOffsetBnRx

% Helper function that plots the synchronization and distributed beamforming results.

iterPlt = max(iter,2);

nodeNbr = size(phaseTotalCorrBnTx,2);
legendText = cell(1,nodeNbr);
for i = 1:nodeNbr
legendText{i} = sprintf('Node %s',int2str(i));
end

figure
plot(fOffsetBnRx(1:iterPlt,:),'-o'); grid on
title('Beamforming Frequency Offset Without Synchronization')
xlabel('Iteration');ylabel('Frequency (Hz)')
legend(legendText,'Location','southwest')

figure
plot(fcBnTxAct(1:iterPlt,:) - fcBnTx,'-o'); grid on;
title('Beamforming Frequency Offset With Synchronization')
xlabel('Iteration'); ylabel('Frequency (Hz)')
legend(legendText,'Location','southwest')

figure
plot(rad2deg(phaseTotalCorrBnTx(1:iterPlt,:)),'-o'); grid on
ylim([-180 180])
title('Phase Adjustments for the Beamforming Nodes')
xlabel('Iteration'); ylabel('Phase (\circ)')
legend(legendText,'Location','southwest')

figure
plot(rmsSigBBsRx(1:iterPlt),'-o'); grid on
ylim([0 1])
title('Beamforming Performance')
xlabel('Iteration'); ylabel('Normalized RMS Amplitude')
end
```