Phased Array System Toolbox™

User's Guide
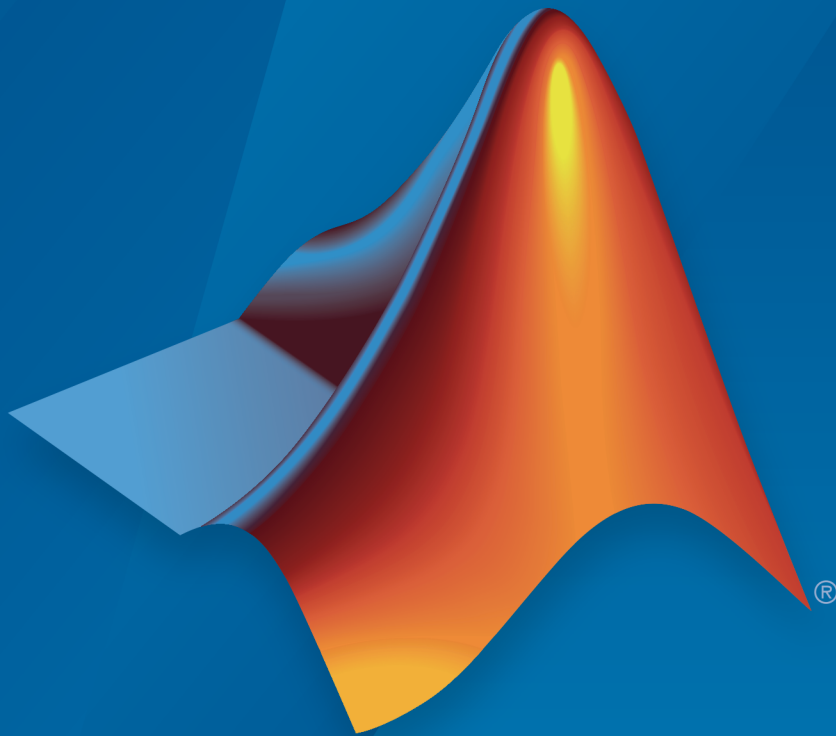
MATLAB&SIMULINK®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Revision History

April 2011	Online only	Revised for Version 1.0 (R2011a)
September 2011	Online only	Revised for Version 1.1 (R2011b)
March 2012	Online only	Revised for Version 1.2 (R2012a)
September 2012	Online only	Revised for Version 1.3 (R2012b)
March 2013	Online only	Revised for Version 2.0 (R2013a)
September 2013	Online only	Revised for Version 2.1 (R2013b)
March 2014	Online only	Revised for Version 2.2 (R2014a)
October 2014	Online only	Revised for Version 2.3 (R2014b)
March 2015	Online only	Revised for Version 3.0 (R2015a)
September 2015	Online only	Revised for Version 3.1 (R2015b)
March 2016	Online only	Revised for Version 3.2 (R2016a)
September 2016	Online only	Revised for Version 3.3 (R2016b)

Contents

Phased Arrays

Antenna and Microphone Elements

1

Array Geometries and Analysis

2

Signal Radiation and Collection

3

Waveforms, Transmitter, and Receiver

4

Beamforming

5

Direction-of-Arrival Estimation

6

Space-Time Adaptive Processing (STAP)

7

Detection

8

Environment and Target Models

9

Coordinate Systems and Motion Modeling

10

Using Polarization

11

Antenna and Array Definitions

12

Code Generation

13

Define New System Objects

14

Phased Arrays

1

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:

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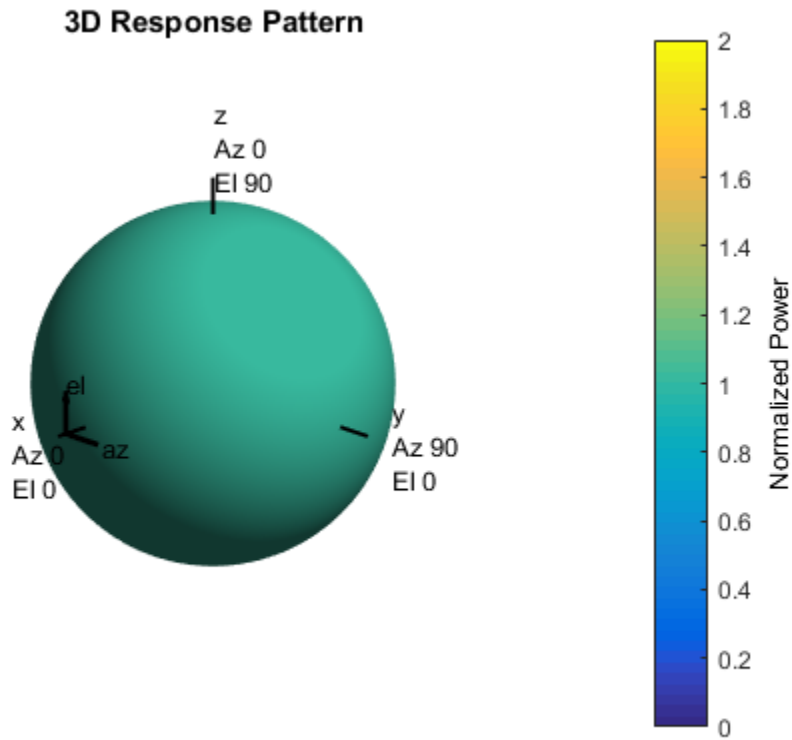
Using the antenna `pattern` method, plot the antenna response at zero degrees elevaton for all azimuth angles at 1 GHz.

```
pattern(antenna,1e9,[-180:180],0,'CoordinateSystem','rectangular',...
    'Type','powerdb')
```
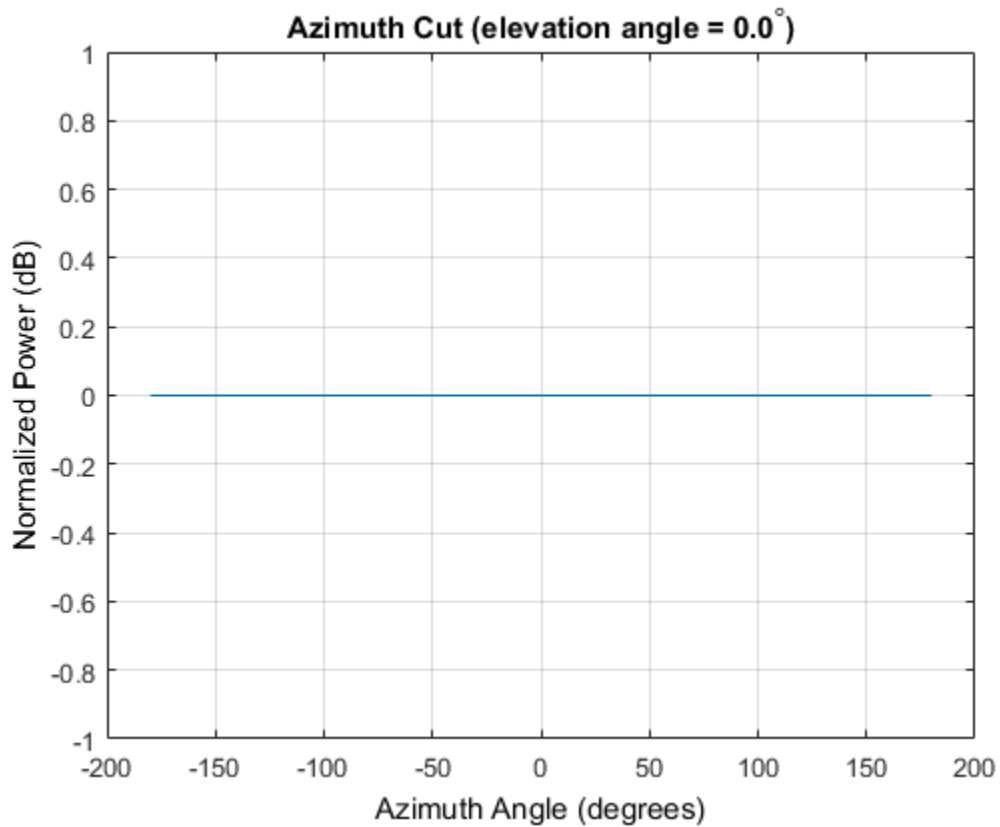
Setting the `BackBaffled` property to `true` restricts the antenna response to azimuth angles in the interval [-90,90] degrees. In this case, plot the antenna response in three dimensions.

```
antenna.BackBaffled = true;
pattern(antenna,fc,[-180:180],[-90:90],'CoordinateSystem','polar',...
    'Type','power')
```
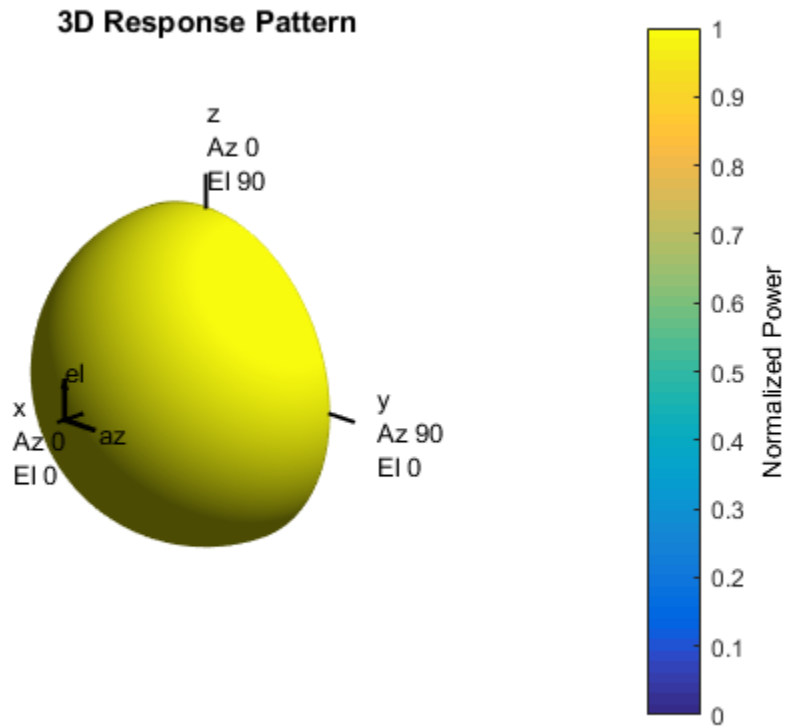
Response of Backbaffled Isotropic Antenna Element

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 =

     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 =

     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

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 step method returns the *field response* (or *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 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')
```
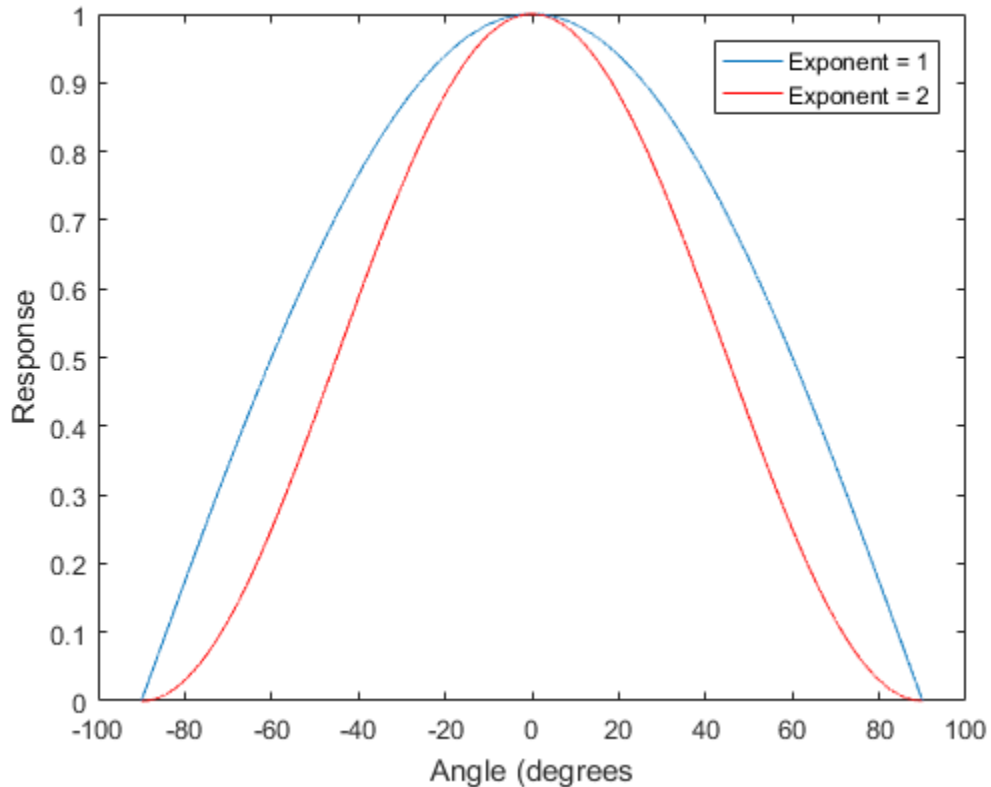
Plot 3-D Response of Cosine Antenna Element

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')
```
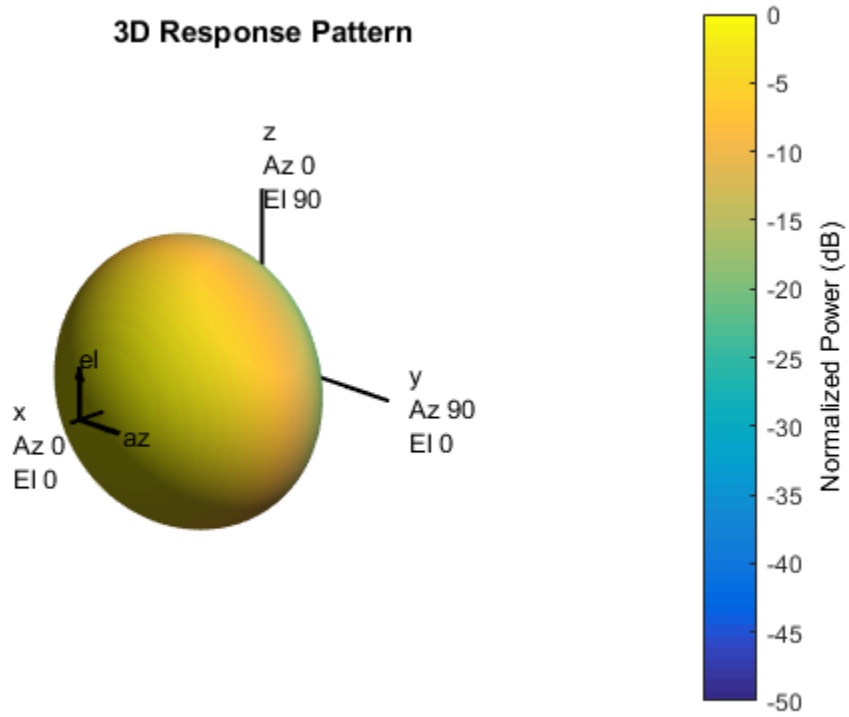
Custom Antenna Element

In this section...
"Support for Custom Antenna Elements" on page 1-11
"Antenna with Custom Radiation Pattern" on page 1-11

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 φ/θ 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

This example shows how to construct a custom antenna element object. The radiation pattern is independent of azimuth angle and has a cosine dependence on elevation angle.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
az = -180:90:180;
el = -90:45:90;
elresp = cosd(el);
antenna = phased.CustomAntennaElement('AzimuthAngles',az,...
    'ElevationAngles',el,...
    'RadiationPattern',repmat(elresp',1,numel(az)));
```

Show the radiation pattern.

```
disp(antenna.RadiationPattern)

        0        0        0        0        0
   0.7071   0.7071   0.7071   0.7071   0.7071
   1.0000   1.0000   1.0000   1.0000   1.0000
   0.7071   0.7071   0.7071   0.7071   0.7071
        0        0        0        0        0
```

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

```
ang = [-30 0; -45 0];
resp = antenna(500e6,ang);
disp(resp)

   1.0848
   1.1220
```

The following code illustrates how nearest-neighbor interpolation method 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.RadiationPattern),...
    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 output of the `step` method.

```
disp(antresp)
```

```
    1.0848
    1.1220
```

Omnidirectional Microphone

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

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

3D Response Pattern
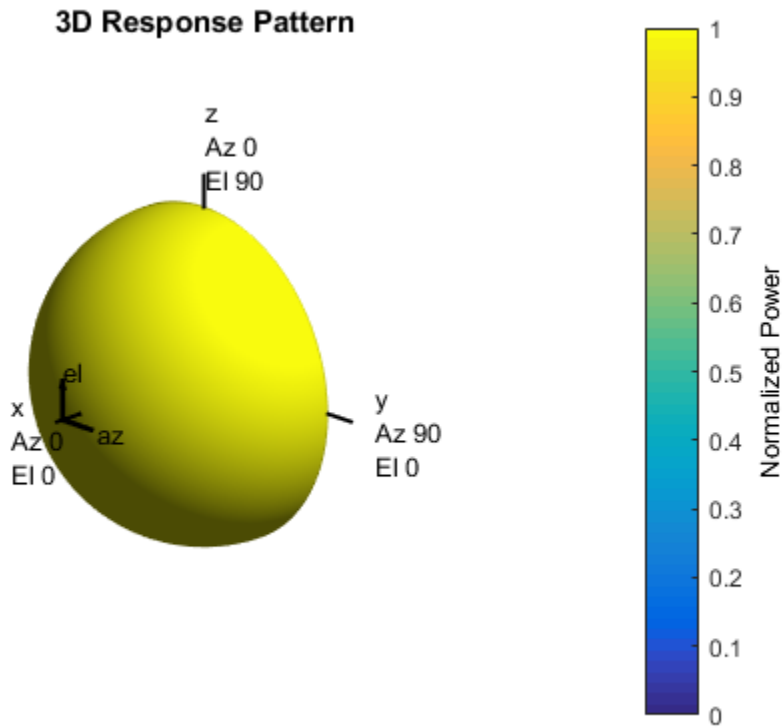
In many applications, you sometimes need to examine the microphone directionality, or polar pattern. To obtain an azimuth cut, set the elevation argument of the `pattern` method to a single angle such as zero.

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

Azimuth Cut (elevation angle = 0.0°)

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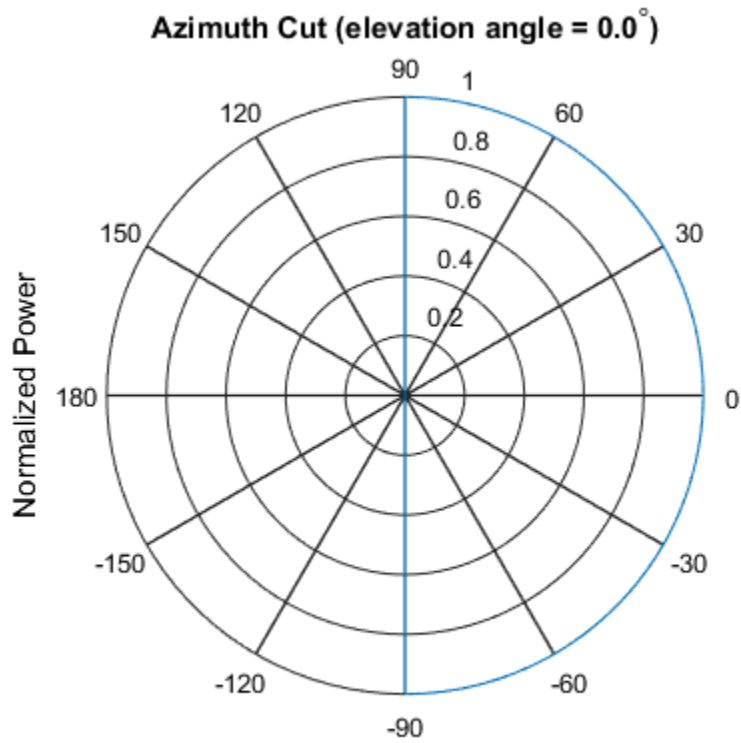
To obtain an elevation cut, set the azimuth argument of the `pattern` method to a single angle such as zero.

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

Elevation Cut (azimuth angle = 0.0°)

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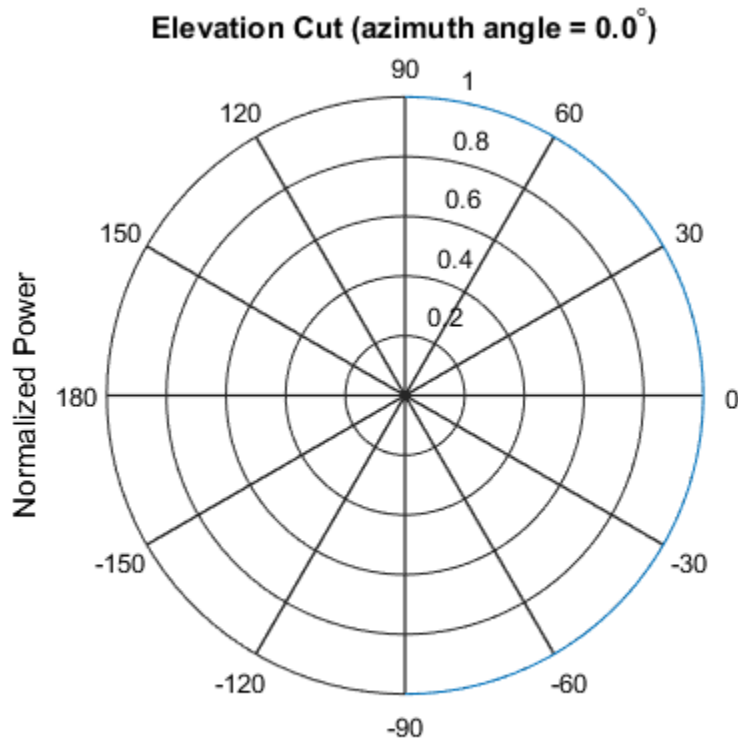
Obtain the microphone magnitude response at the specified azimuth angles and frequencies. By default, when the `ang` argument is a single row, the elevation angles are 0 degrees. Note the response is unity at all azimuth angles and frequencies, as expected.

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


response =

    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

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')
```
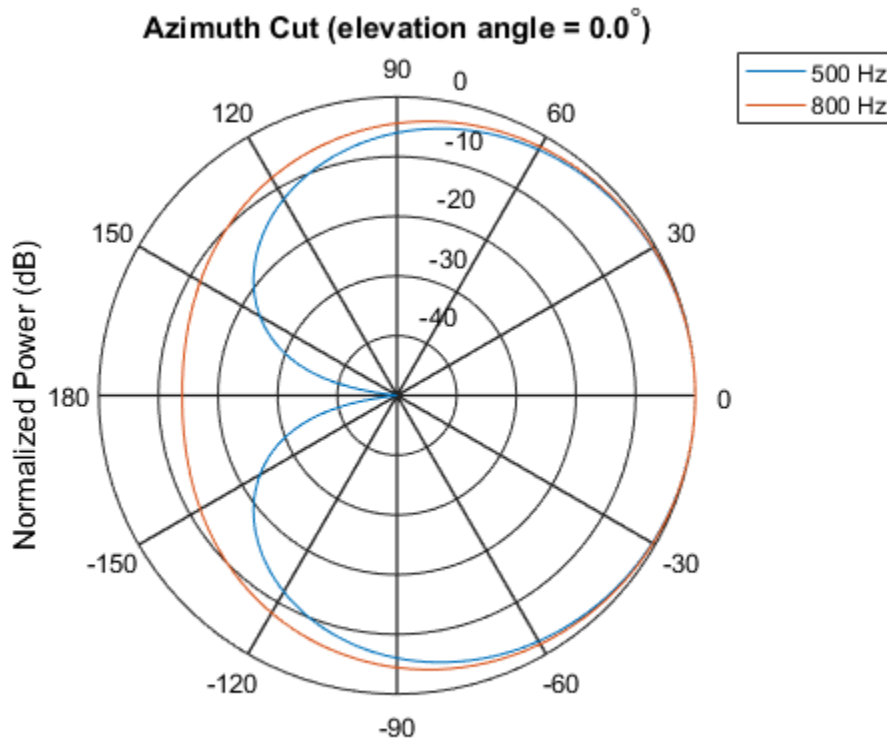
Normalized Power (dB), Broadside at 0.00 degrees

Related Examples

- "Microphone ULA Array" on page 2-11

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

$$E_r = 0$$
$$E_H = 0$$
$$E_V = -\frac{iZ_0IL}{2\lambda}\cos\mathrm{el}\,\frac{e^{-ikr}}{r}$$

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°.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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

**Vertical Polarization
vs Elevation Angle**

**Horizontal Polarization
vs Elevation Angle**

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

$$E_r = 0$$

$$E_H = -\frac{iZ_0IL}{2\lambda}\cos az \frac{e^{-ikr}}{r}$$

$$E_V = \frac{iZ_0IL}{2\lambda}(\sin el \sin az + i\cos el)\frac{e^{-ikr}}{r}$$

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.

LHCP and RHCP Polarization Components

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

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Create the `phased.CrossedDipoleAntennaElement` System 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° elevation.

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

LHCP and RHCP vs Azithmuth Angle

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]*
pattern(crosseddipoleantenna,fc,[-180:180],0,...
    'Type','powerdb')
```

Azimuth Cut (elevation angle = 0.0°)
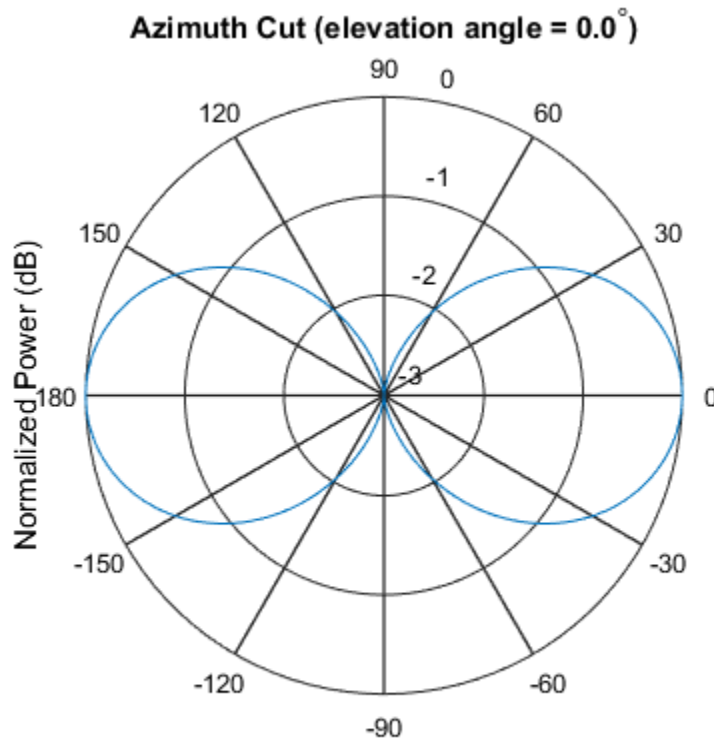
Normalized Power (dB), Broadside at 0.00 degrees

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')
```

Azimuth Cut (elevation angle = 0.0°)
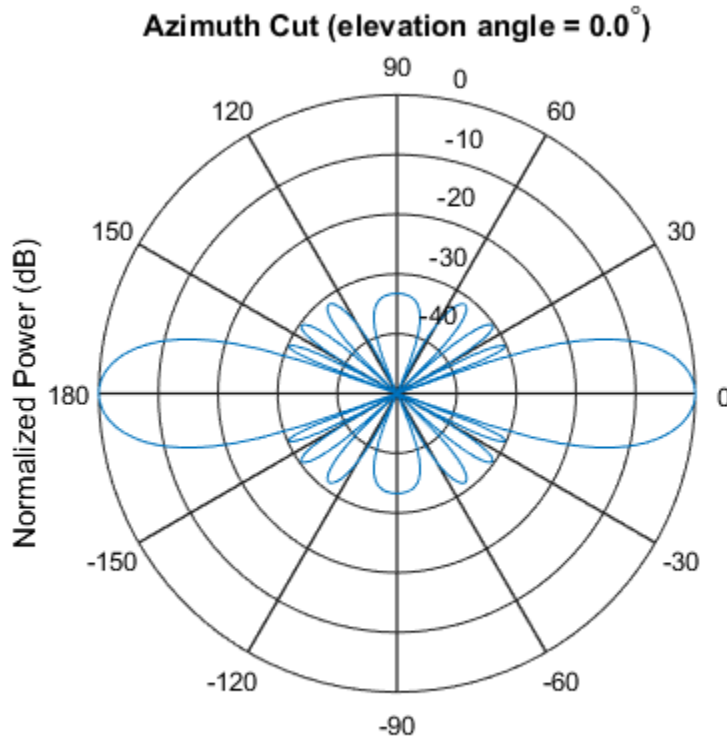
Normalized Power (dB), Broadside at 0.00 degrees

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)
```
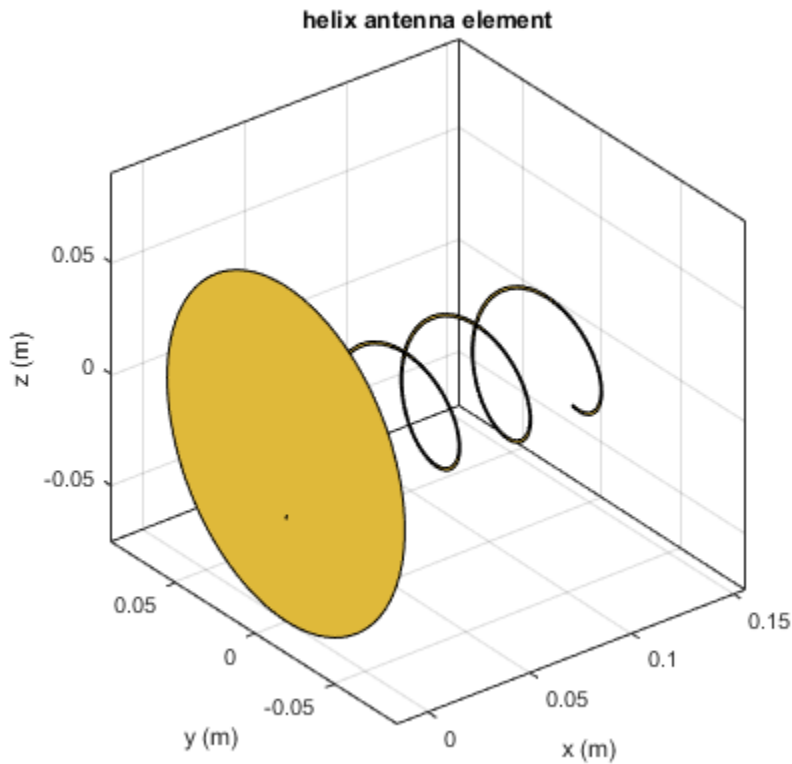

Then, draw the azimuth antenna pattern at 0 degrees elevation at the operating frequency of 2 GHz. Use the pattern function from Antenna Toolbox.

```
pattern(helixantenna,fc,[-180:180],0,...
```

```
'Type','powerdb')
```

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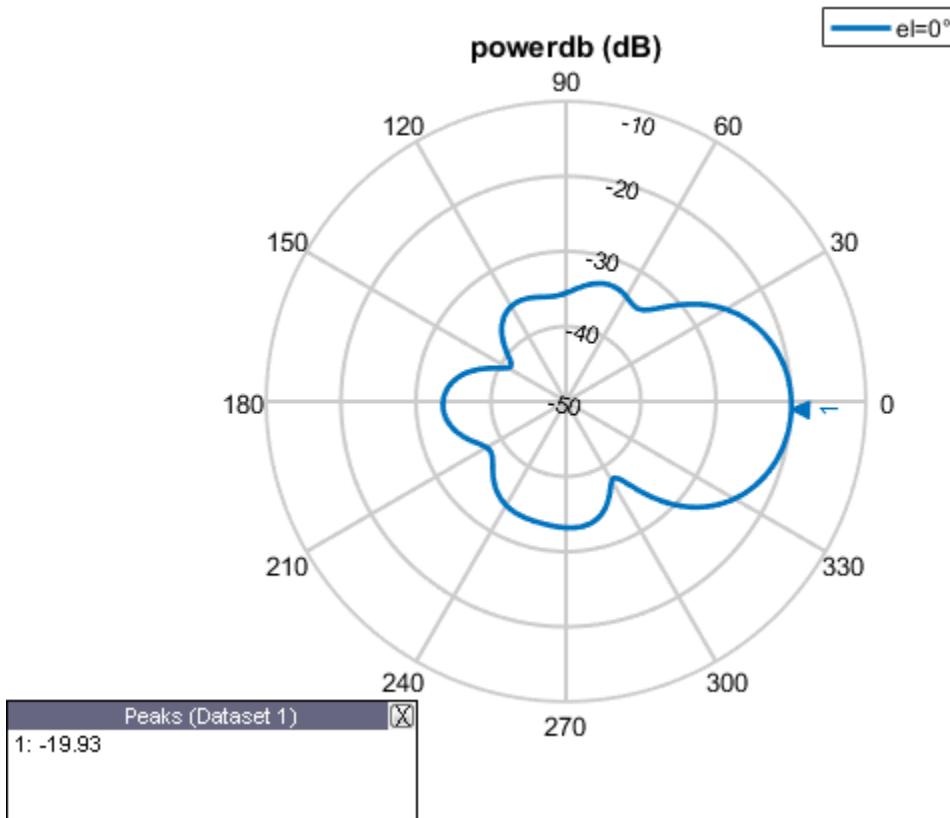
Next, construct an 11-element tapered uniform linear array of helix antennas with elements spaced at 0.4 wavelengths. Taper the array with a Taylor window. You can use the same `phased.ULA` System object from Phased Array System Toolbox to create this array.

```
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°)
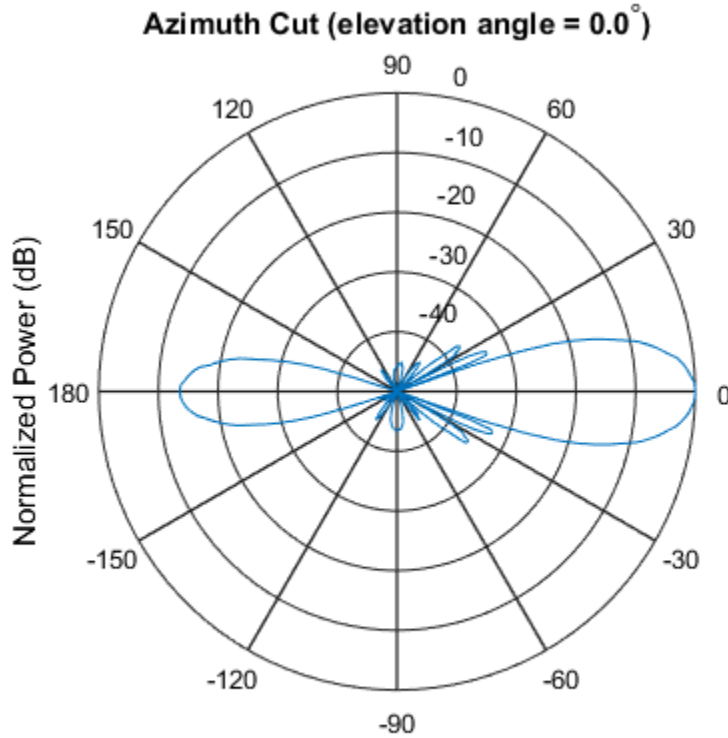
Normalized Power (dB), Broadside at 0.00 degrees

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.

2

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

Aperture Size:
Y axis = 2 m
Element Spacing:
Δ y = 500 mm
Array Axis: Y axis
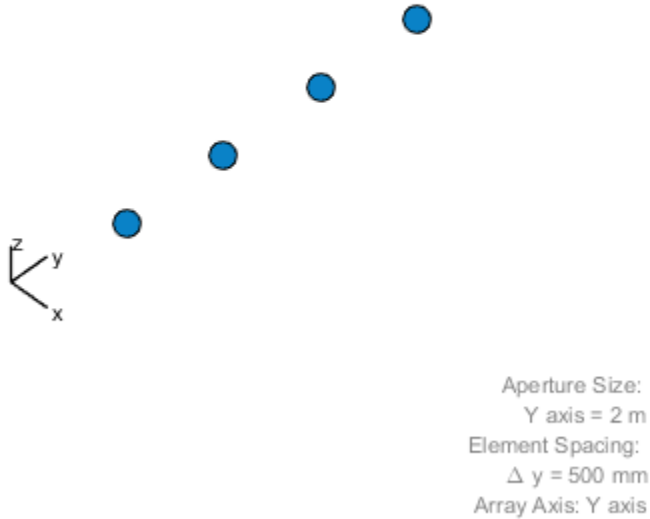
You can return the coordinates of the array sensor elements in the form `[x;y;z]` by using the `getElementPosition` method. See "Rectangular Coordinates" for toolbox conventions.

```
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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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 =

   1.0e-08 *

   -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" 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 =

     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 =

    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 τ_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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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


sv =

  -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 =

  -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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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)')
```
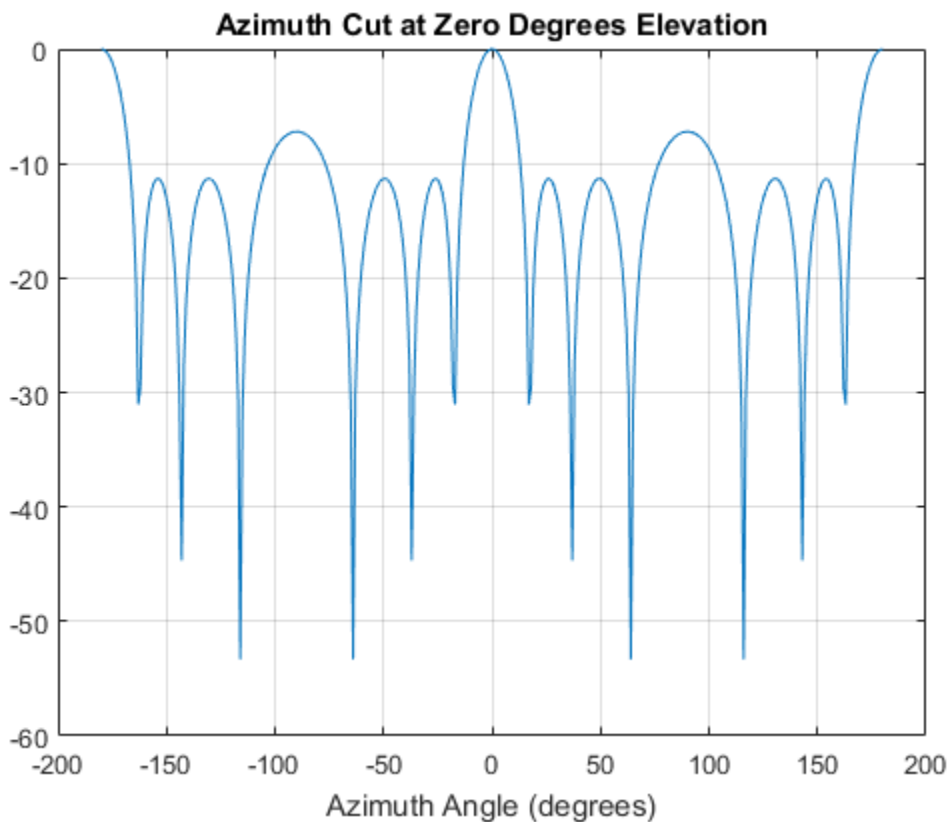

Visualize the array response using the `pattern` method. Create a 3-D plot of the response in *UV* space; other plotting options are available.

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

```
'PropagationSpeed',physconst('Lightspeed'))
```


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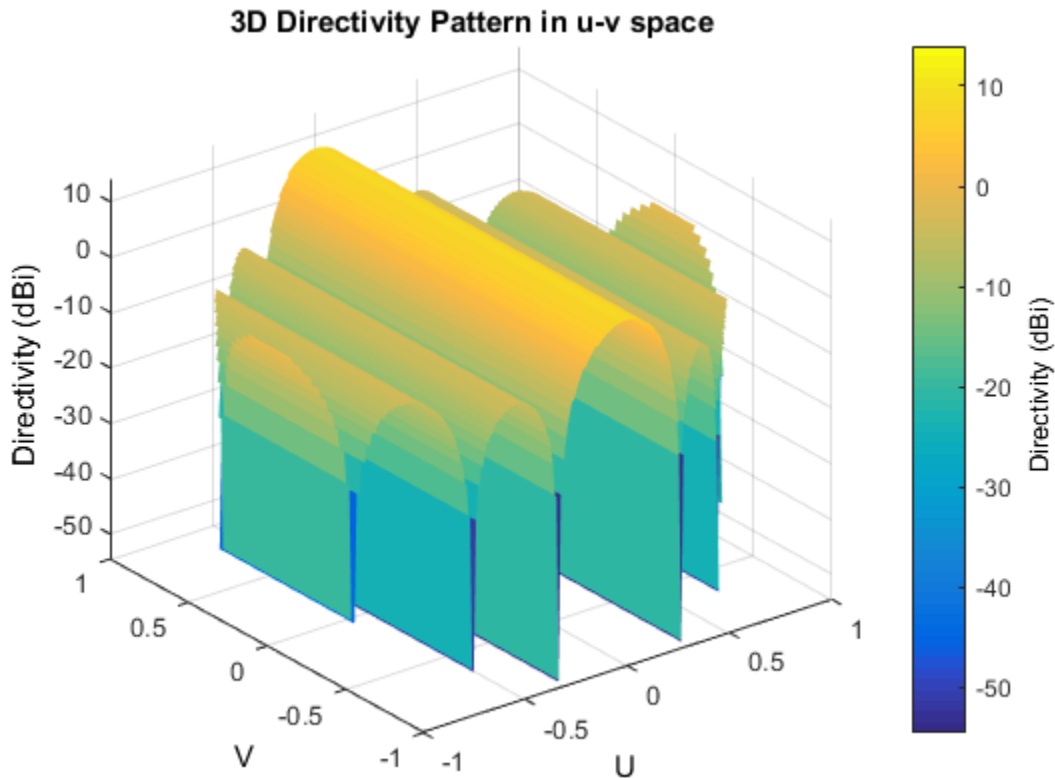
Reception of Plane Wave Across Array

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)*.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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.';
```

Related Examples

*   "Microphone ULA Array" on page 2-11

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);
```

Azimuth Cut (elevation angle = 0.0°)
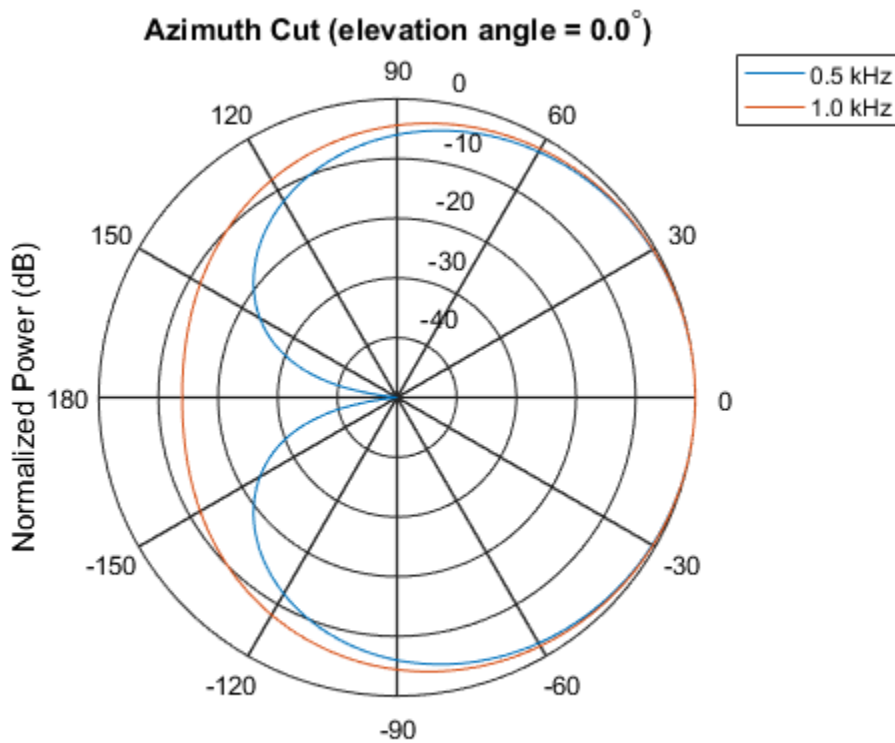
Normalized Power (dB), Broadside at 0.00 degrees

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)

0.5 kHz
1.0 kHz
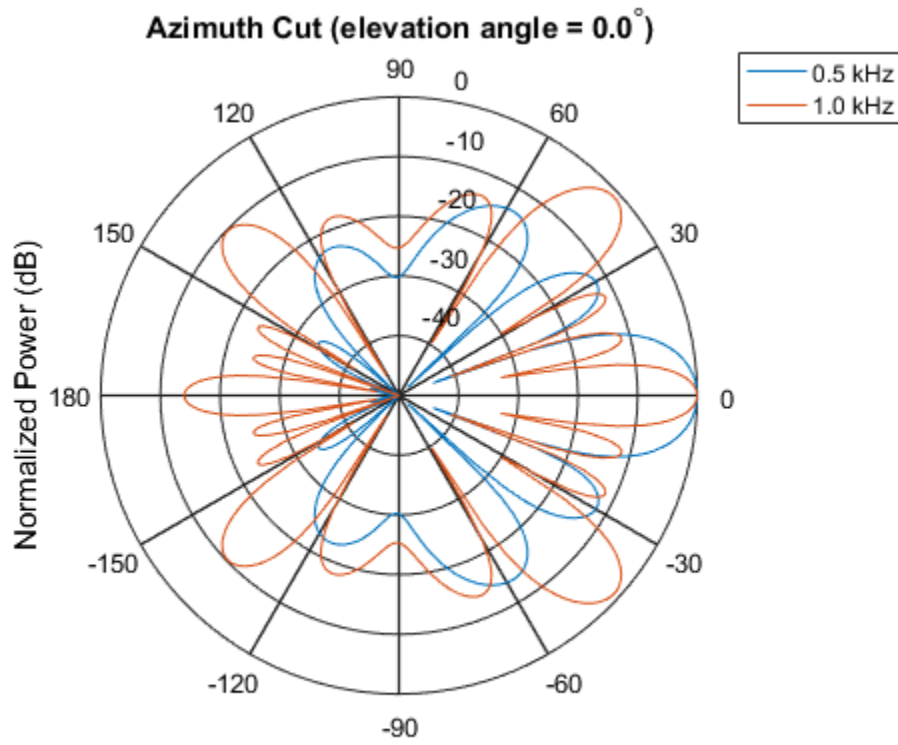
Normalized Power (dB), Broadside at 0.00 degrees

Uniform Rectangular Array

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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)
pos = getElementPosition(array);
```

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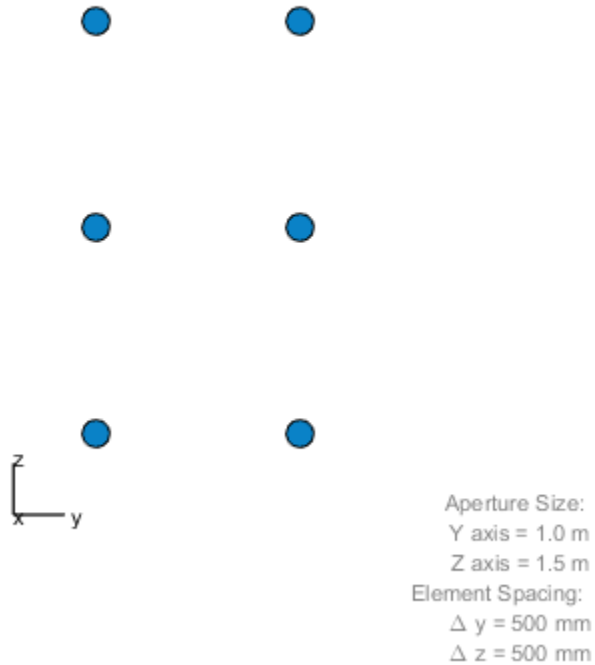
The *x*-coordinate is zero for all elements of the array.

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')
```
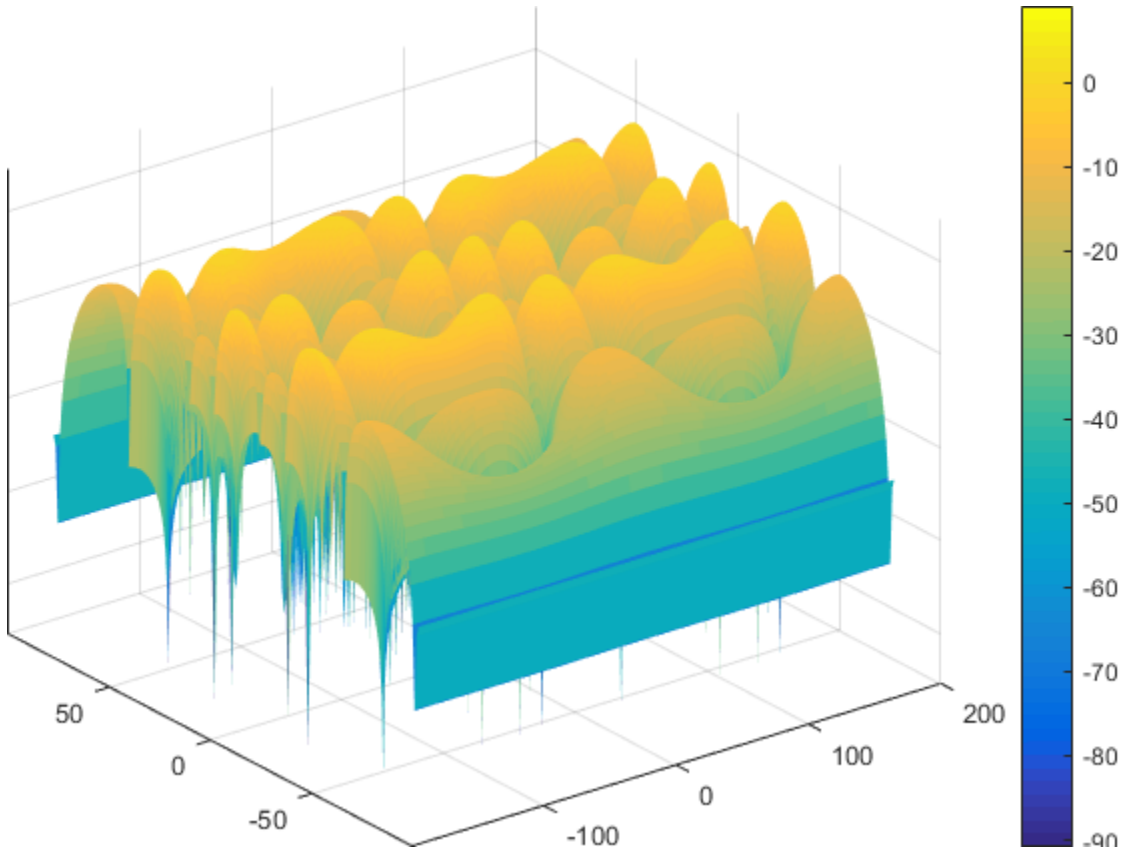
Conformal Array

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: [1×1 phased.IsotropicAntennaElement]
    ElementPosition: [3×1 double]
```

```
    ElementNormal: [2×1 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 object™. 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 0.5λ where λ 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 ϕ_n is the azimuth angle of the nth 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
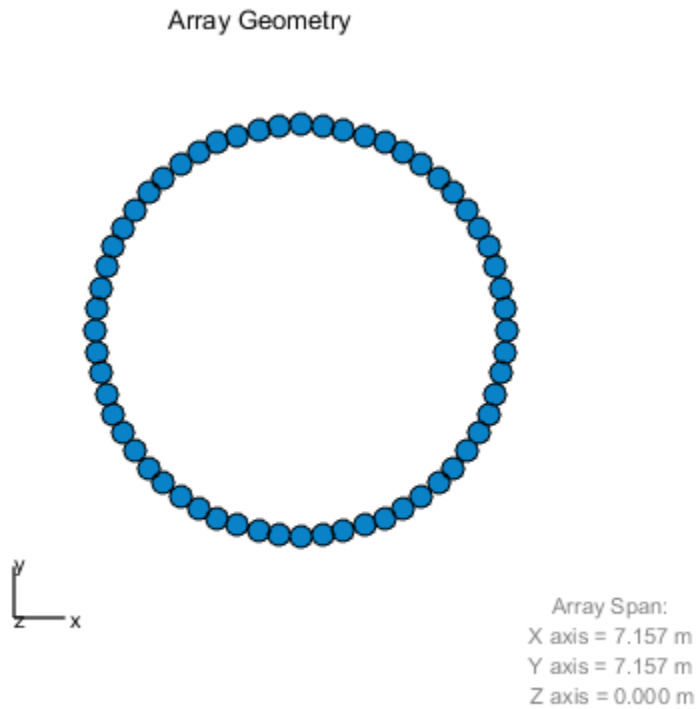
Plot the array response pattern at 1 GHz.

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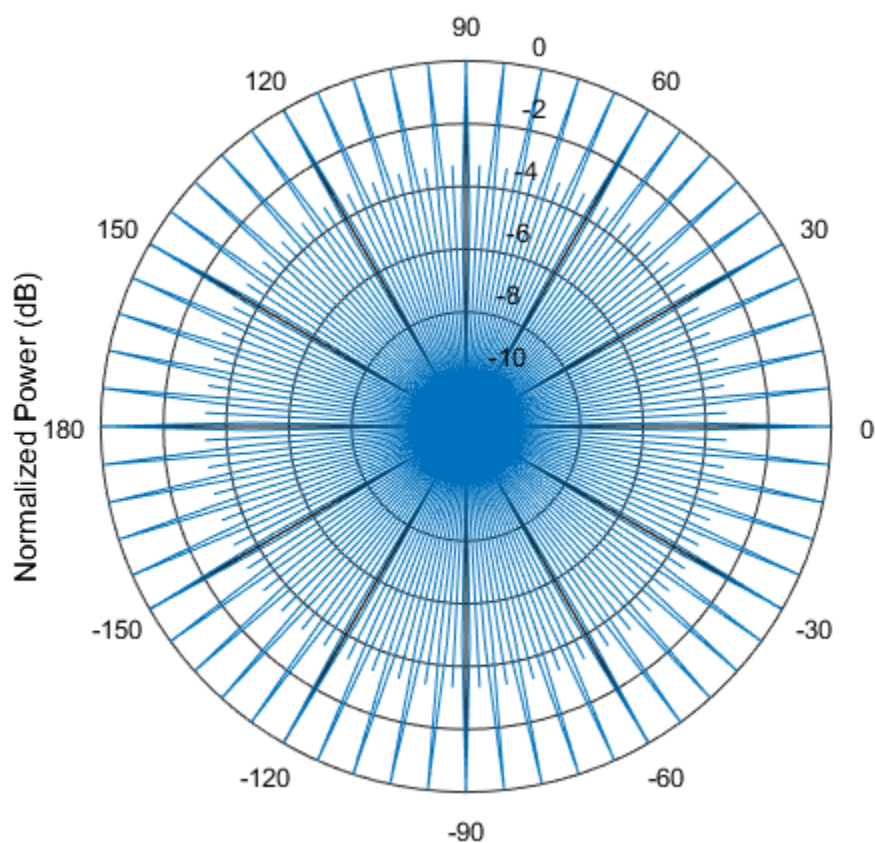
Custom Antenna Array

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,...
    'RadiationPattern',repmat(elresp',1,numel(az)));
pattern(antenna,3e8,0,el,'CoordinateSystem','polar','Type','powerdb',...
    'Normalize',true);
```


Elevation Cut (azimuth angle = 0.0°)
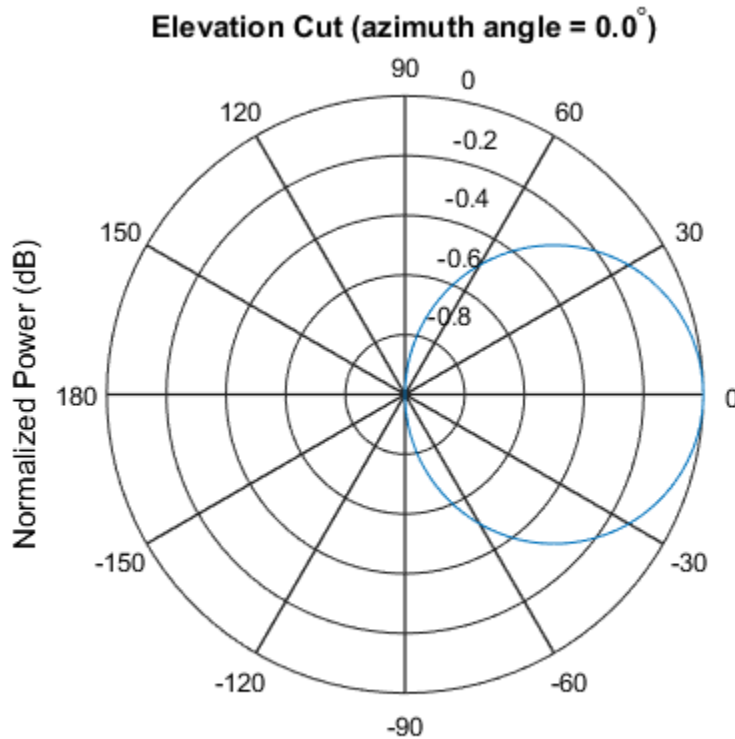
Normalized Power (dB), Broadside at 0.00 degrees

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
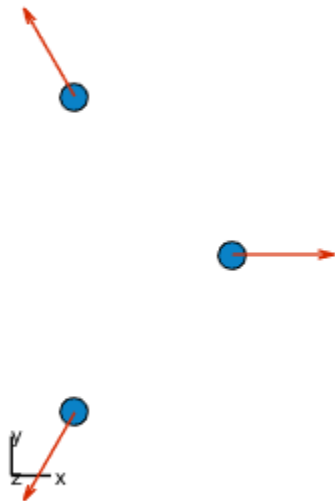
Array Span:
X axis = 1 m
Y axis = 2 m
Z axis = 0 m

```
pattern(array,fc,az,el,'CoordinateSystem','polar','Type','powerdb',...
```

'Normalize',true,'PropagationSpeed',physconst('LightSpeed'))
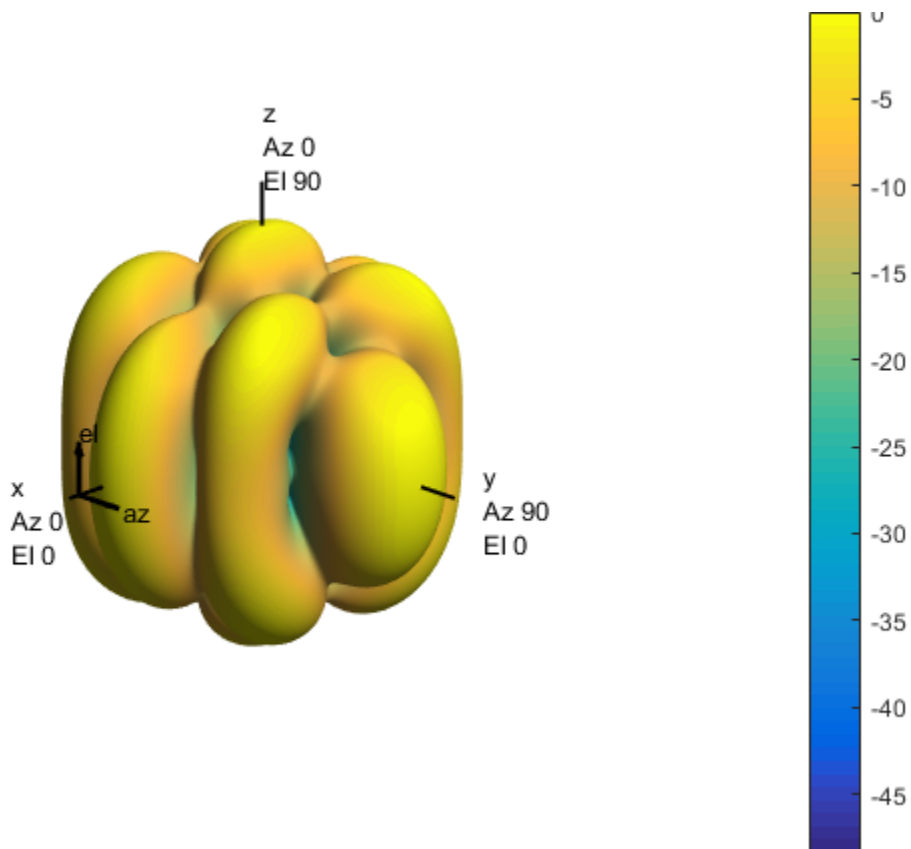
Subarrays Within Arrays

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:
Δy = 500 mm
Δz = 500 mm
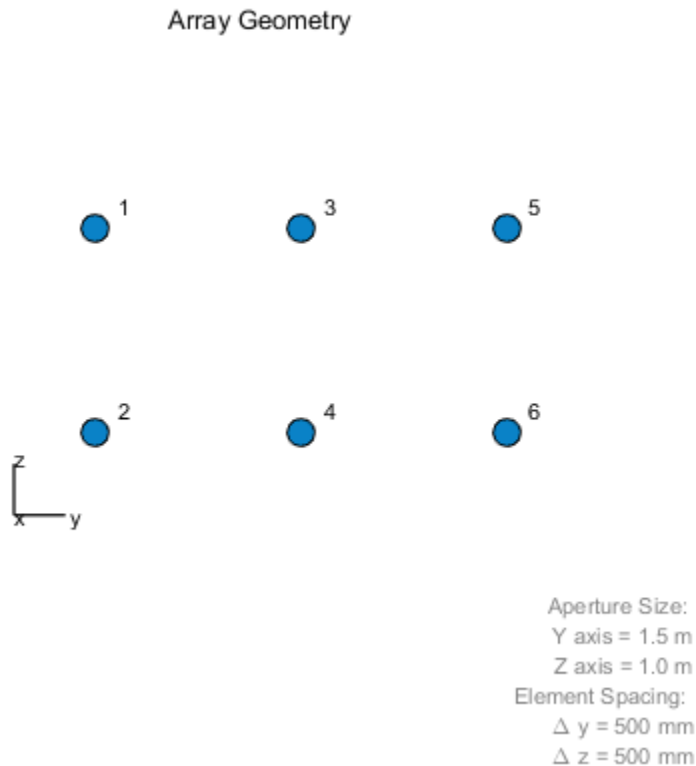
Create and view an array consisting of three 2-element linear subarrays each parallel to the *z*-axis. Use the indices from the plot to form the matrix for the `SubarraySelection` property. The `getSubarrayPosition` method returns the phase centers of the three subarrays.

2-27

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

```
subarray1pos =

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


Array Geometry

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)
subarraypos2 = getSubarrayPosition(partitionedarray2)


subarraypos2 =

         0         0
         0         0
   -0.2500    0.2500
```

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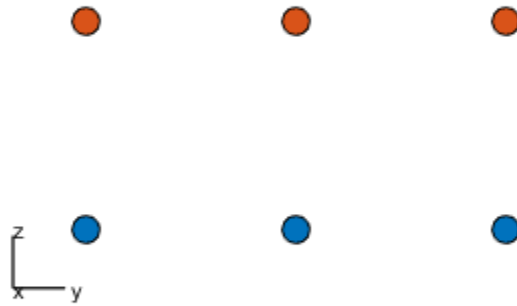
Array Span:
X axis =   0 mm
Y axis = 1000 mm
Z axis = 500 mm

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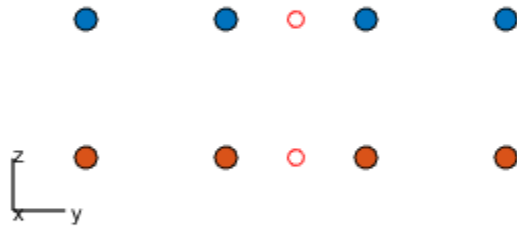
Array Span:
X axis =   0 mm
Y axis = 1500 mm
Z axis = 500 mm

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))];
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
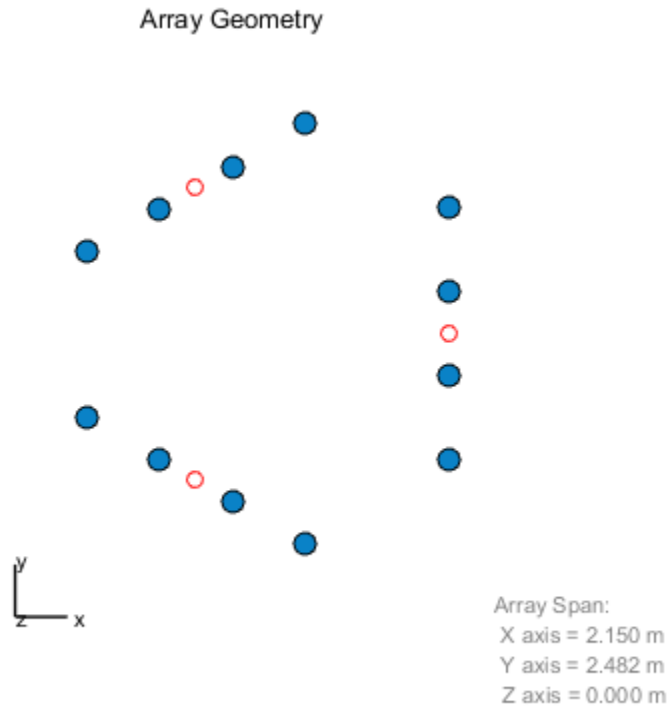
Related Examples

- Subarrays in Phased Array Antennas

Plot Array Directivity Using Sensor Array Analyzer App

The `sensorArrayAnalyzer` is a Matlab™ App that lets you examine important properties of a phased array such as 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.

```
sensorArrayAnalyzer;
```
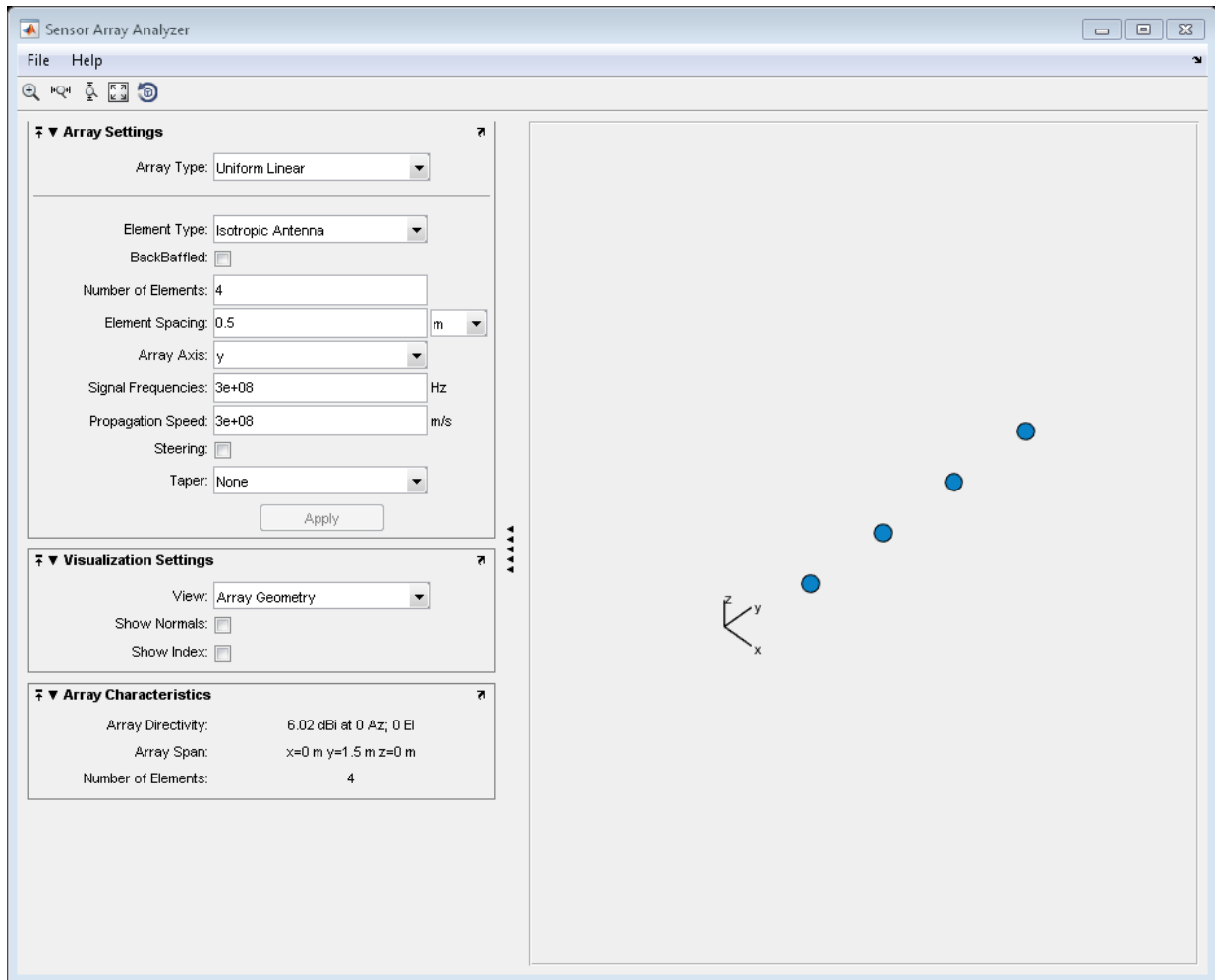
Create 3D Directivity Plot of 4-by-4 URA

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 Type** to `Uniform Rectangular`

2 Set the **Element Type** to `Cosine Antenna`

3 Set the **Size** of the arry to `[4 4]`

4 Set the **Element Spacing** to `[0.4 0.4]` wavelengths

5 Make sure to select the spacing units to wavelength

6 Select the **Steering** check box

7 Set the **Steer Angle** to `[-30,0]` to show a steered array

8 Choose the type of **View** as `3D Array Directivity`

9 Choose the view **Option** as `Polar` to get a polar diagram

10 Select the **Show Geometry** check box to show the array shape as well

Then, you will see a plot of array directivity similar to this.

```
filenm = fullfile(matlabroot,'examples','phased','SensorArrayAnalyzerAppExample_O2.png
im = imread(filenm);
figure('Position',[315 160 906 690])
image(im)
axis off
set(gca,'Position',[0.078 0.077 0.845 0.896])
```
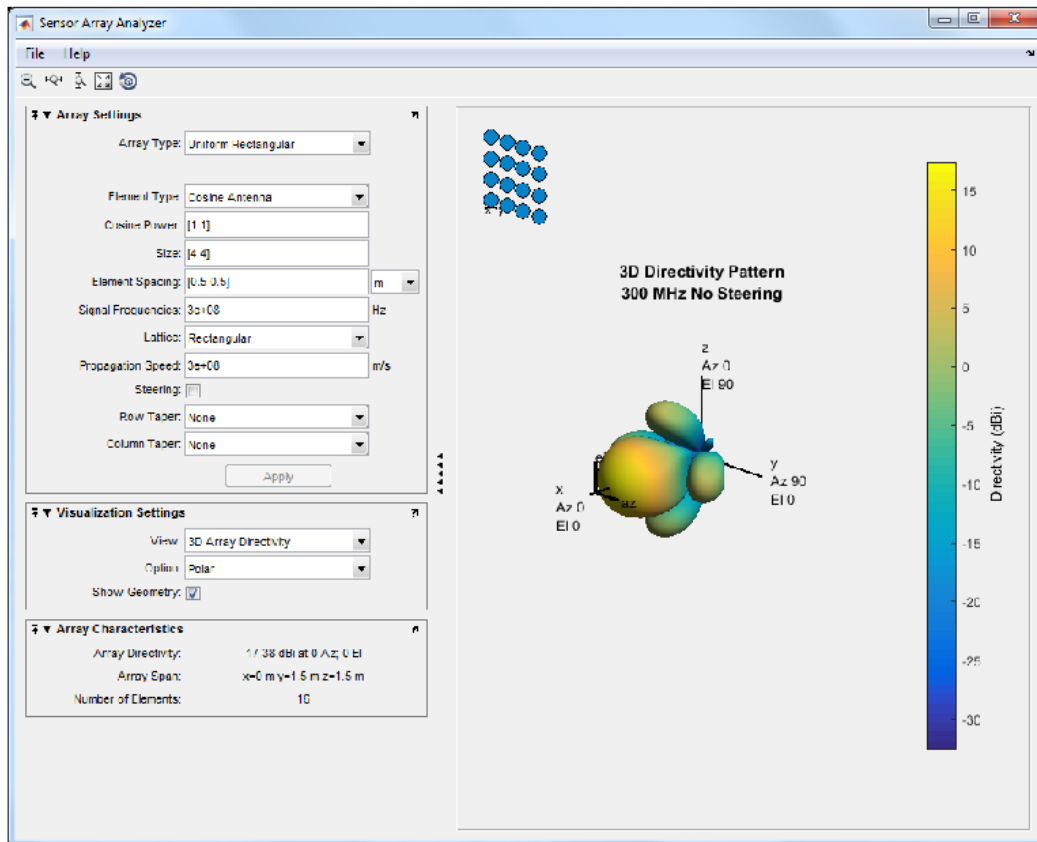
Signal Radiation and Collection

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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

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 4-39 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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')


collector =

  phased.Collector with properties:

                 Sensor: [1×1 phased.ULA]
       PropagationSpeed: 299792458
      OperatingFrequency: 300000000
              Wavefront: 'Plane'
      EnablePolarization: false
        WeightsInputPort: false
```

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

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


y =

   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 =
```

```
  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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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

```
resp =

    0.7071
```

Elevation Cut (azimuth angle = 0.0°)
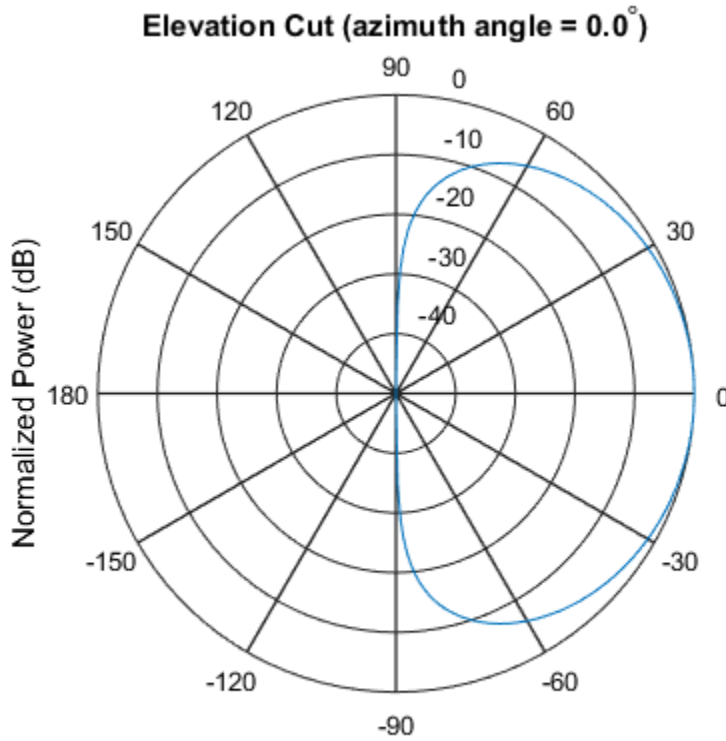
Normalized Power (dB), Broadside at 0.00 degrees

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 =

   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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

```
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

In this section...

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 ω_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 τ 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)
```
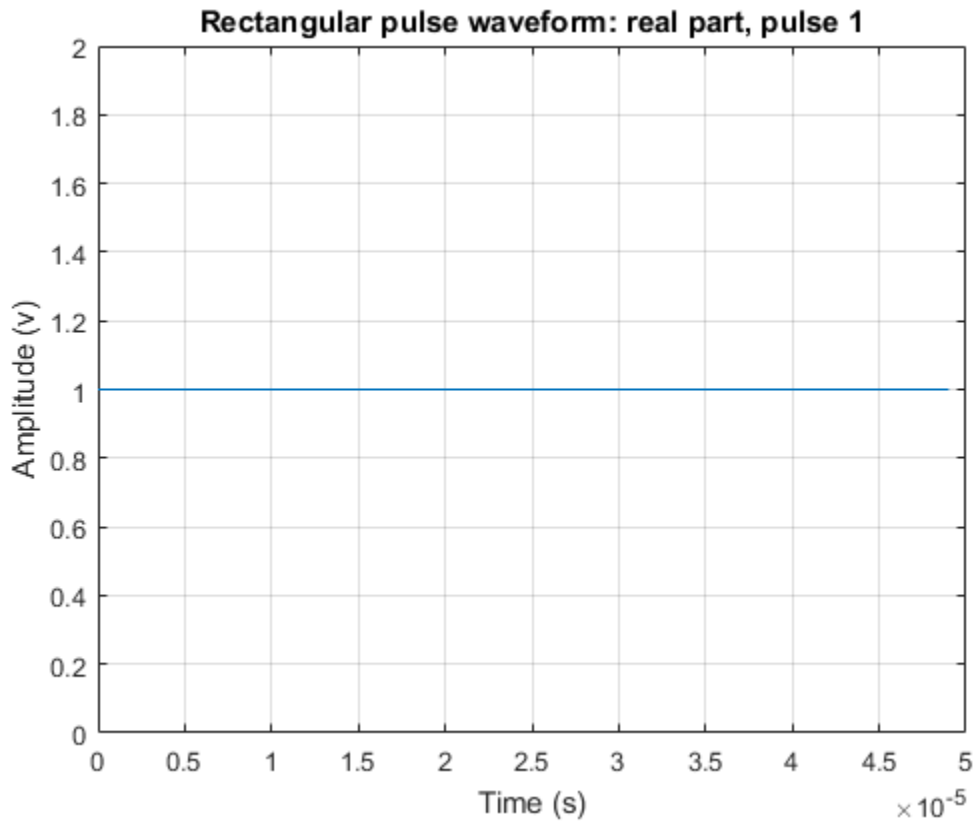

Find the bandwidth of the rectangular pulse.

```
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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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;
```

```matlab
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)
```
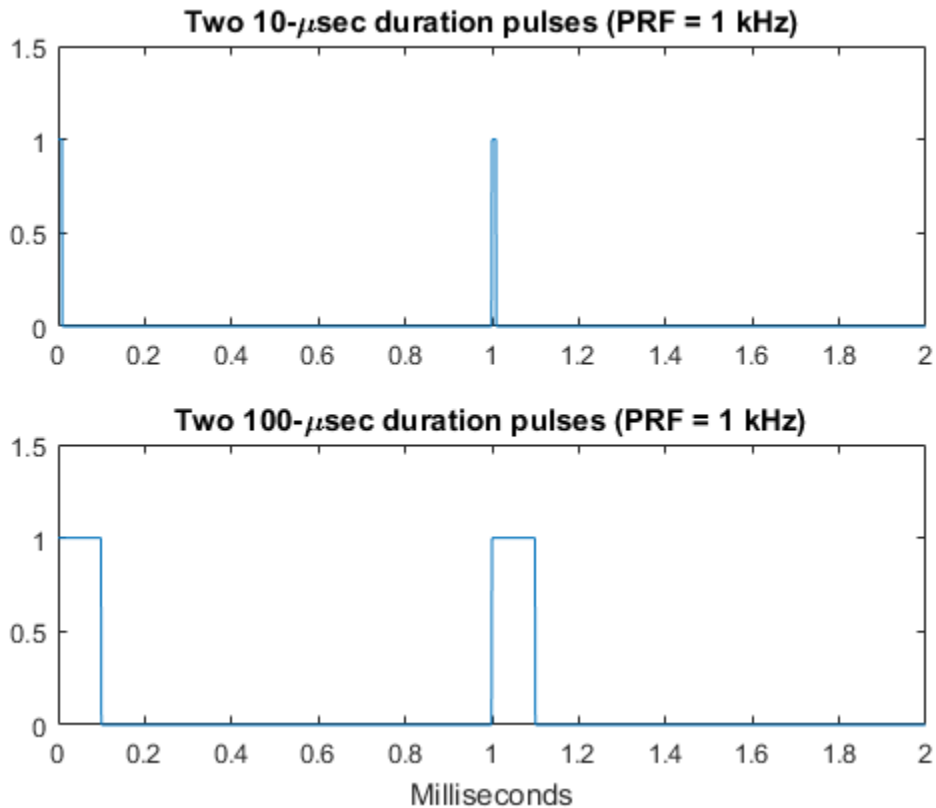
Linear Frequency Modulated Pulse Waveforms

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 ß is the bandwidth and τ 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 β/τ.

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 τ is the pulse duration.

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

  - The Gaussian envelope is:

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

4-7

- 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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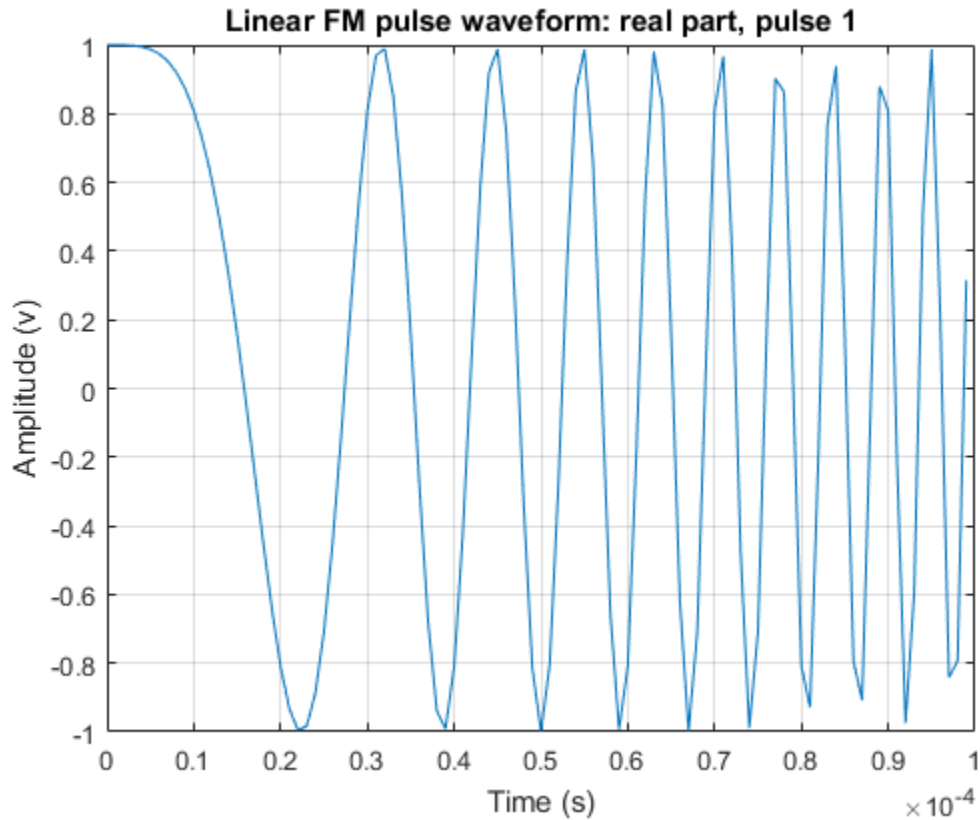
Use the `step` method to obtain one full repetition interval of the signal. Plot the real and imaginary parts.

```
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
```
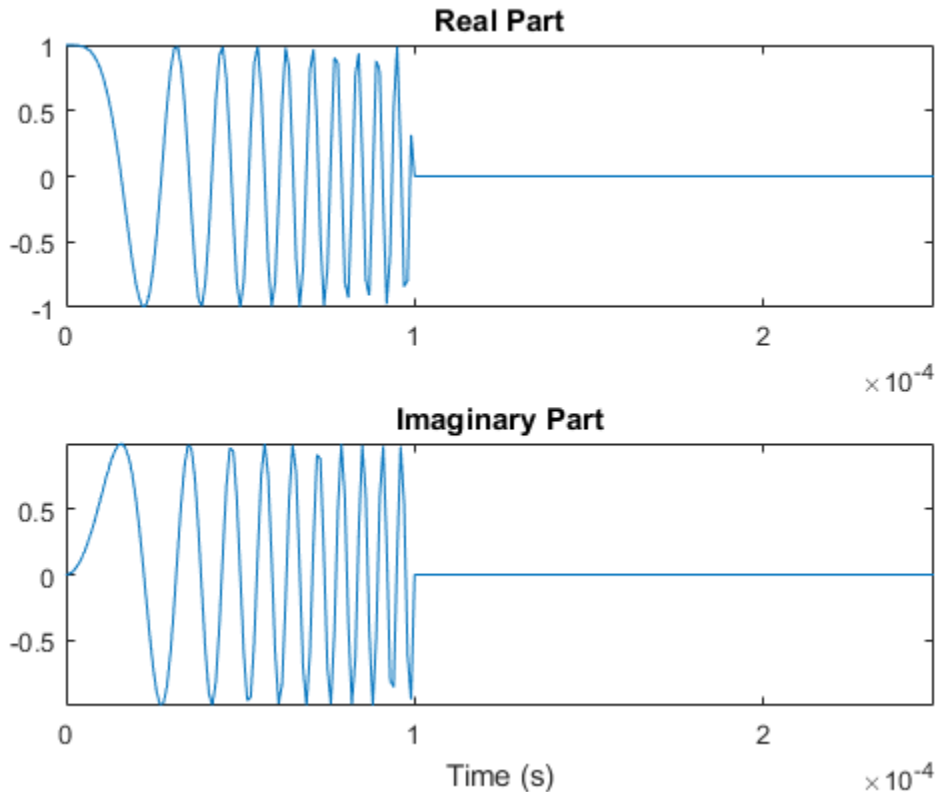

Ambiguity Function of Linear FM Waveform

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

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Define and set up the linear FM waveform.

```
waveform = phased.LinearFMWaveform('PulseWidth',100e-6,...
```

```matlab
    'SweepBandwidth',2e5,'PRF',1e3);
```

Generate samples of the waveform.

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

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

```matlab
[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')
```

Linear FM Pulse Waveform Ambiguity Function
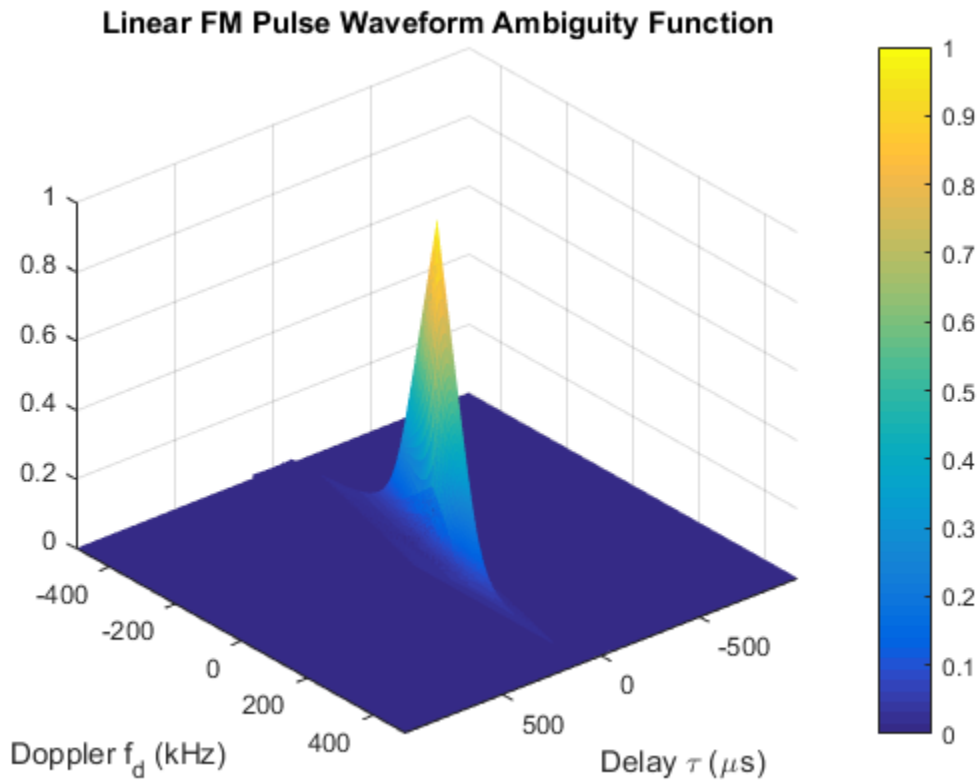
The surface has a narrow ridge that is slightly tilted. The tilt indicates better resolution in the zero delay cut.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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))
```
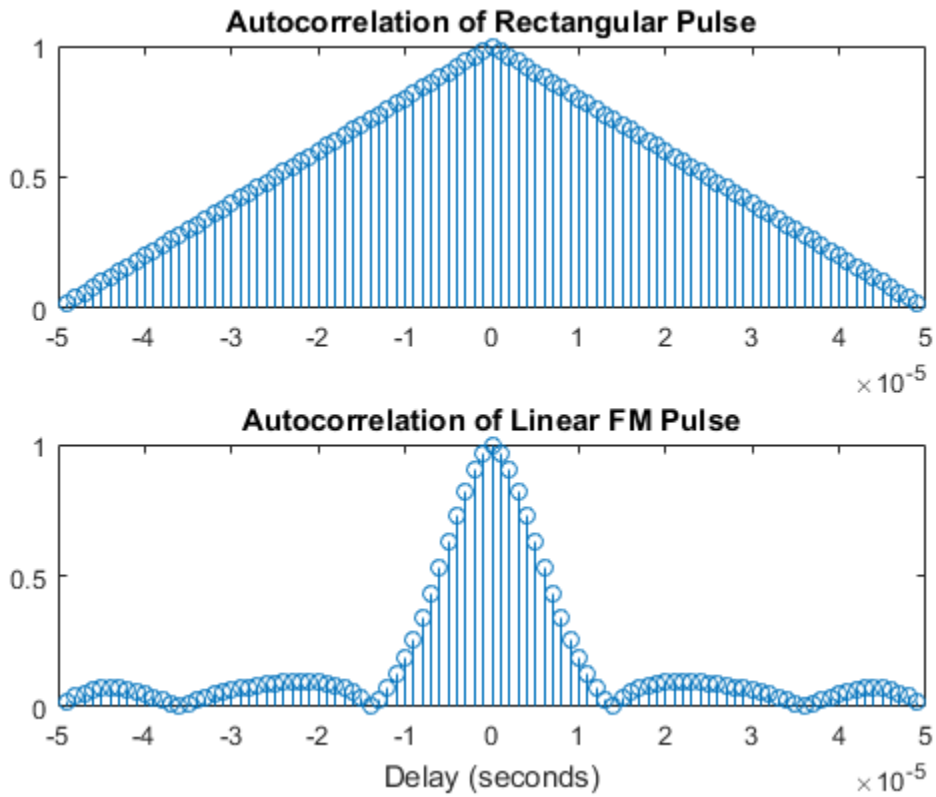
Related Examples

- "Waveform Analysis Using the Ambiguity Function"

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. Set the pulse width (duration) to 50 μs, the pulse repetition frequency (PRF) to 10 kHz, and the frequency step to 20 kHz. The sampling rate is 1 MHz. By default the `OutputFormat` property is set to `'Pulses'` and the number of pulses in the output is set to one.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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

Use the `bandwidth` function to show that the bandwidth of the stepped FM pulse waveform is the product of the frequency step 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 remainder of the PRI consists of zeros.

The frequency of the initial pulse is zero Hz (DC). With the `NumPulses` property set to one, 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 obkect 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
```
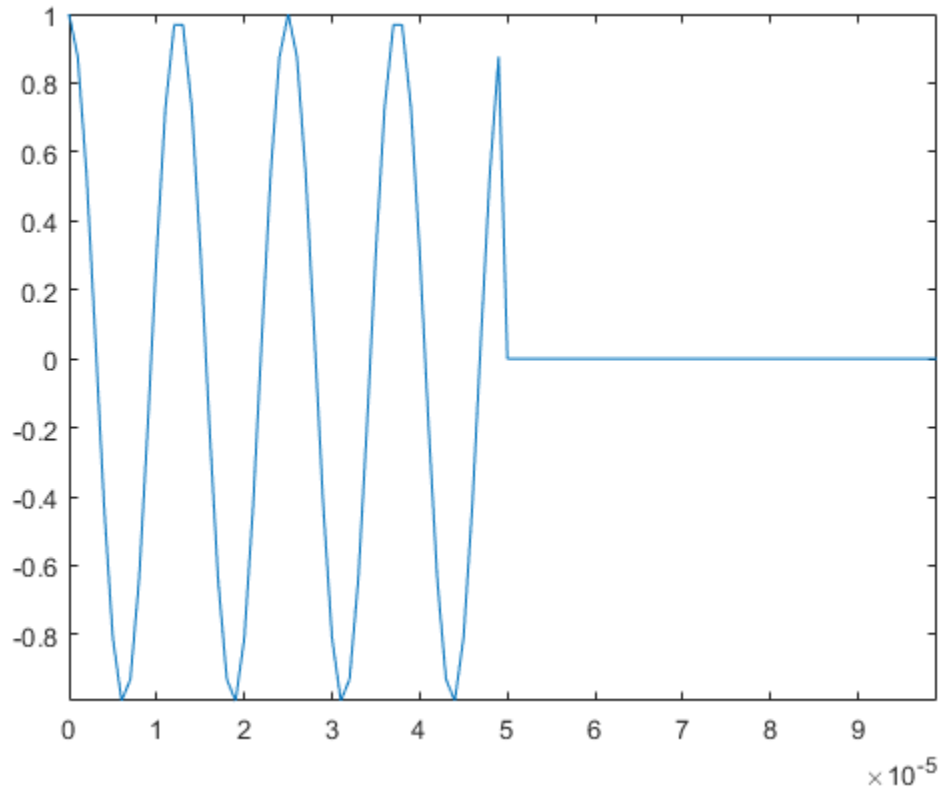
This plot shows how the pulse returns to its DC value.

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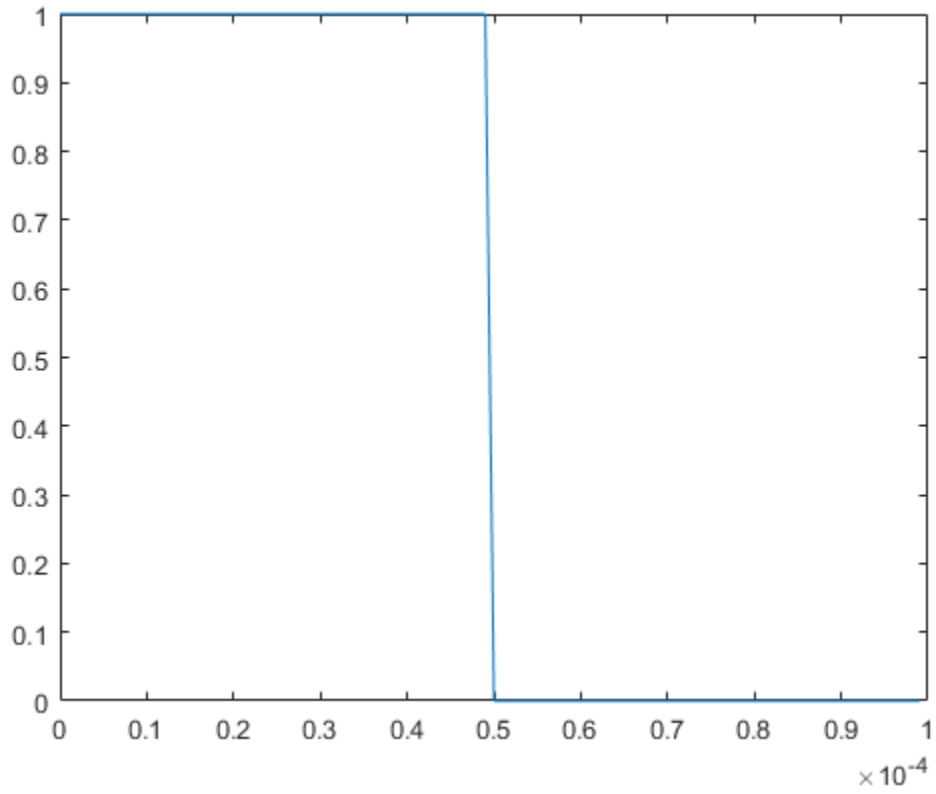
FMCW Waveforms

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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)')
```
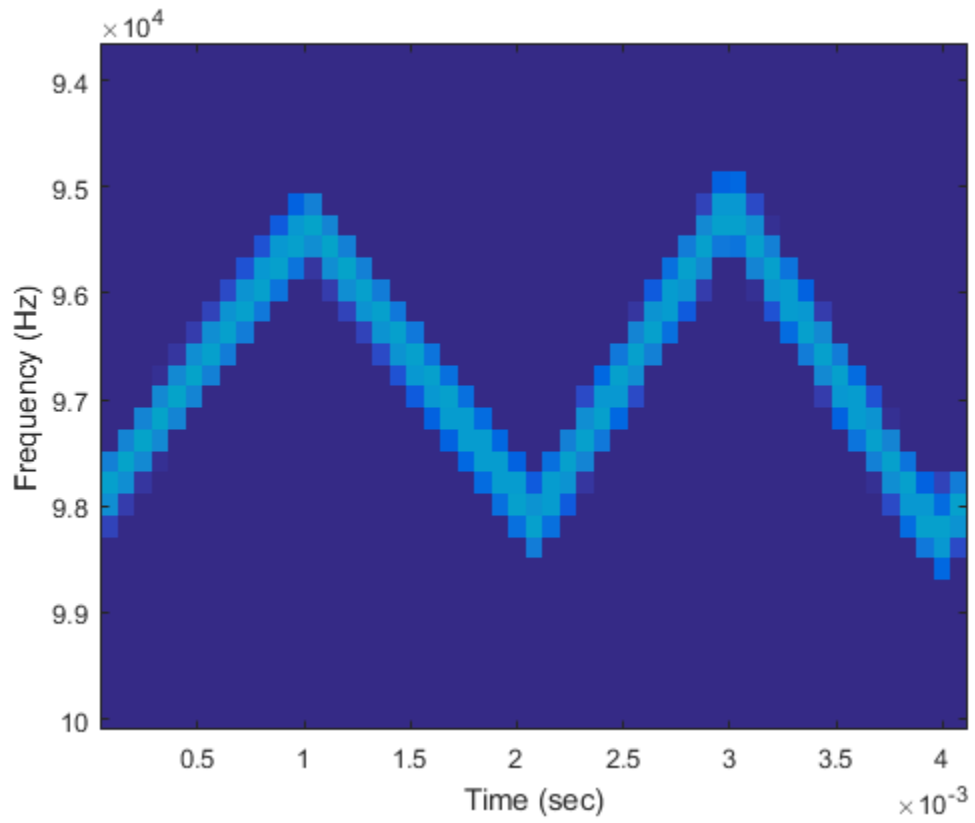
Phase-Coded Waveforms

In this section...
"When to Use Phase-Coded Waveforms" on page 4-22
"How to Create Phase-Coded Waveforms" on page 4-22

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 "End-to-End Radar System" example, you can use a phase-coded waveform instead of a rectangular waveform. To do so, replace the `phased.RectangularWaveform` System object™ with `phased.PhaseCodedWaveform`.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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 "End-to-End Radar System" 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);

Pd = 0.9;
Pfa = 1e-6;
numpulses = 10;
SNR = albersheim(Pd,Pfa,10);
maxrange = 1.5e4;
lambda = physconst('LightSpeed')/4e9;
Pt = radareqpow(lambda,maxrange,SNR,tau,'RCS',0.5,'Gain',20);
transmitter = phased.Transmitter('PeakPower',50e3,'Gain',20,...
    'LossFactor',0,'InUseOutputPort',true,...
```

```matlab
    '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'),...
    '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
```
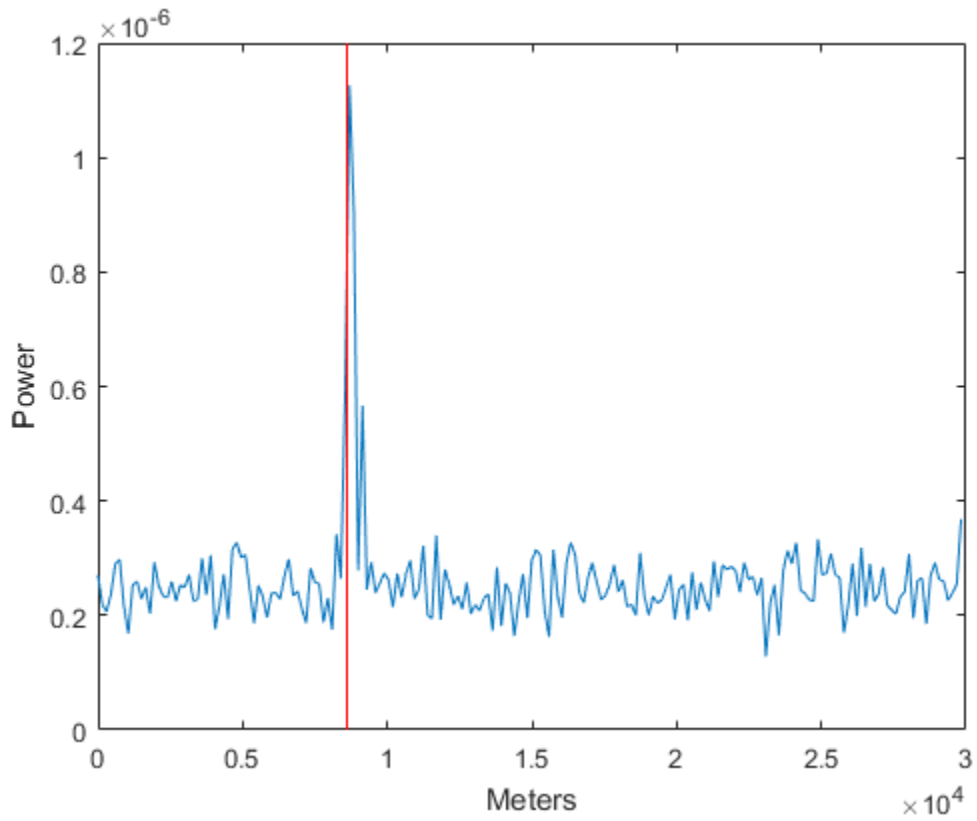
Waveforms with Staggered PRFs

In this section...
"When to Use Staggered PRFs" on page 4-27
"Linear FM Waveform with Staggered PRF" on page 4-27

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

```
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')
```
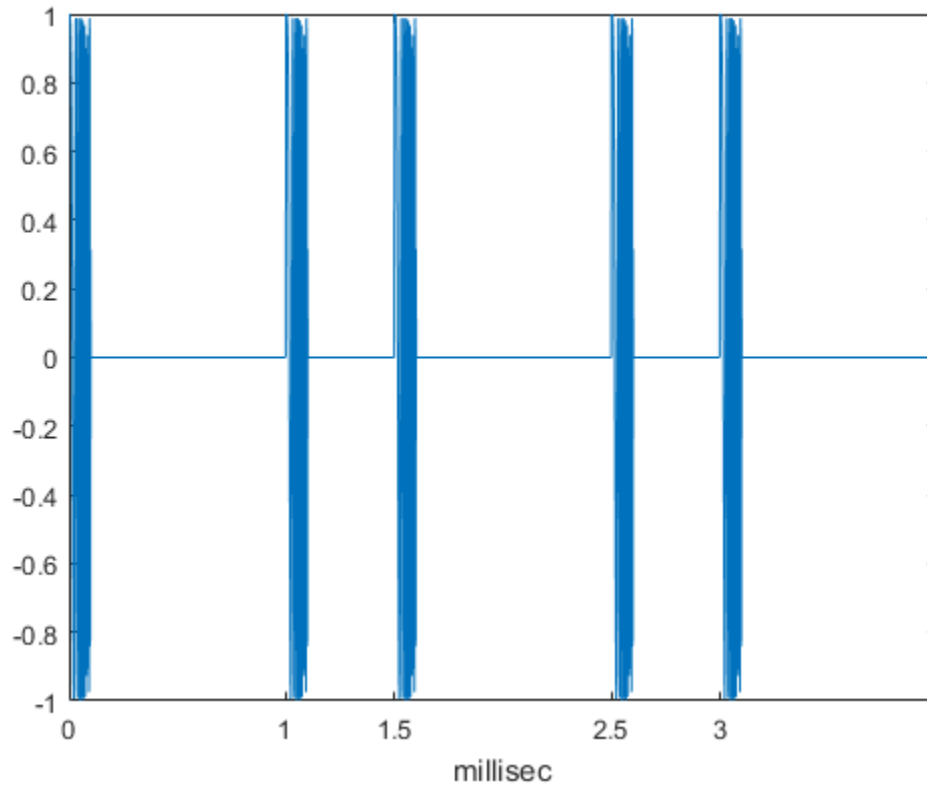
Plot Spectrogram Using Radar Waveform Analyzer App

The `radarWaveformAnalyzer` is a Matlab™ App that lets you explore important properties of a signal such as its waveform, spectrum, and ambiguity function.

Open radarWaveformAnalyzer App

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

```
radarWaveformAnalyzer
```
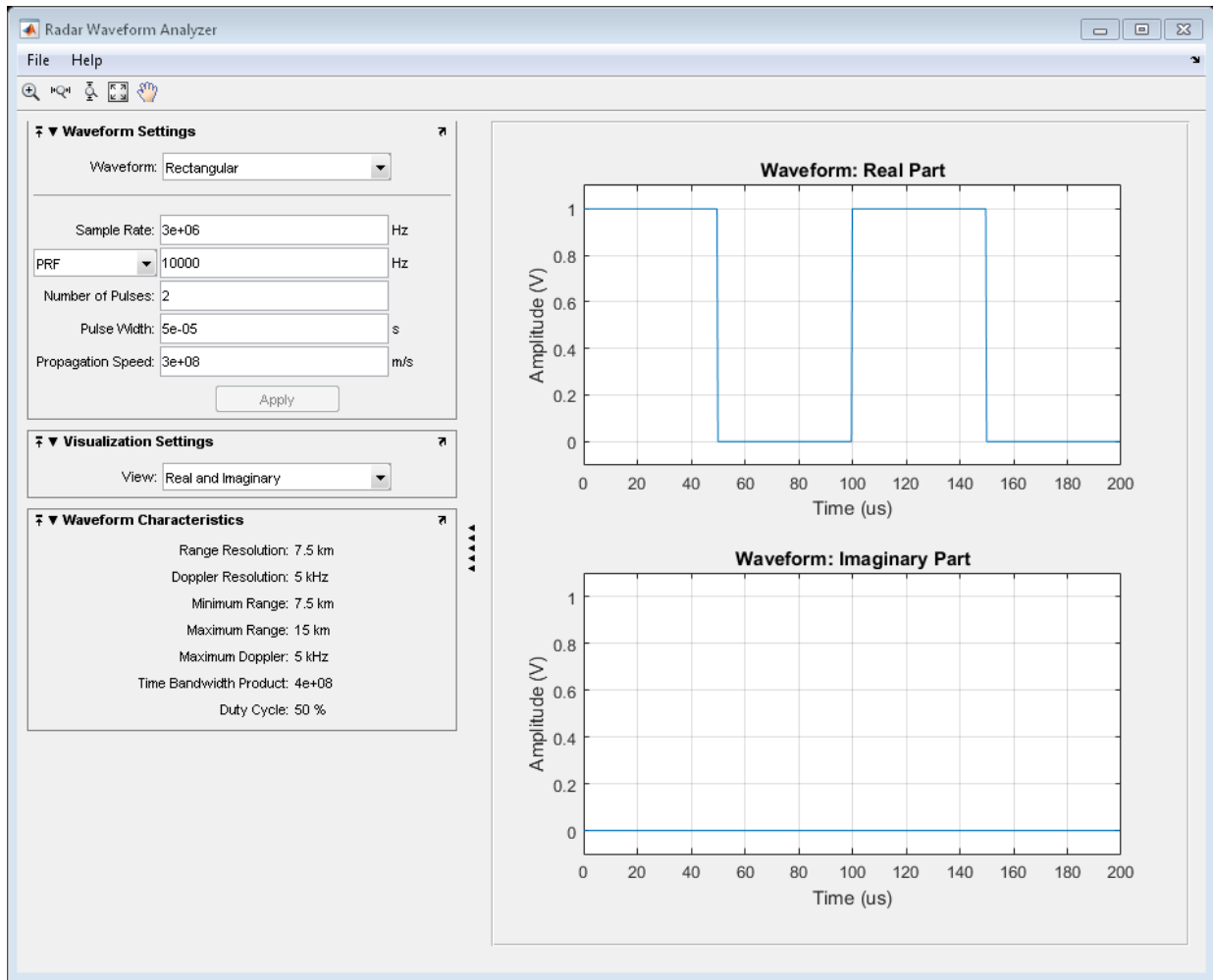
Show the spectrogram of baseband FMCW signal

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 Set the **View** to `Spectrogram`

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

```
filenm = fullfile(matlabroot,'examples','phased','radarWaveformAnalyzerAppExample_02.pr
im = imread(filenm);
figure('Position',[315 160 906 690])
image(im)
axis off
set(gca,'Position',[0.078 0.077 0.845 0.896])
```
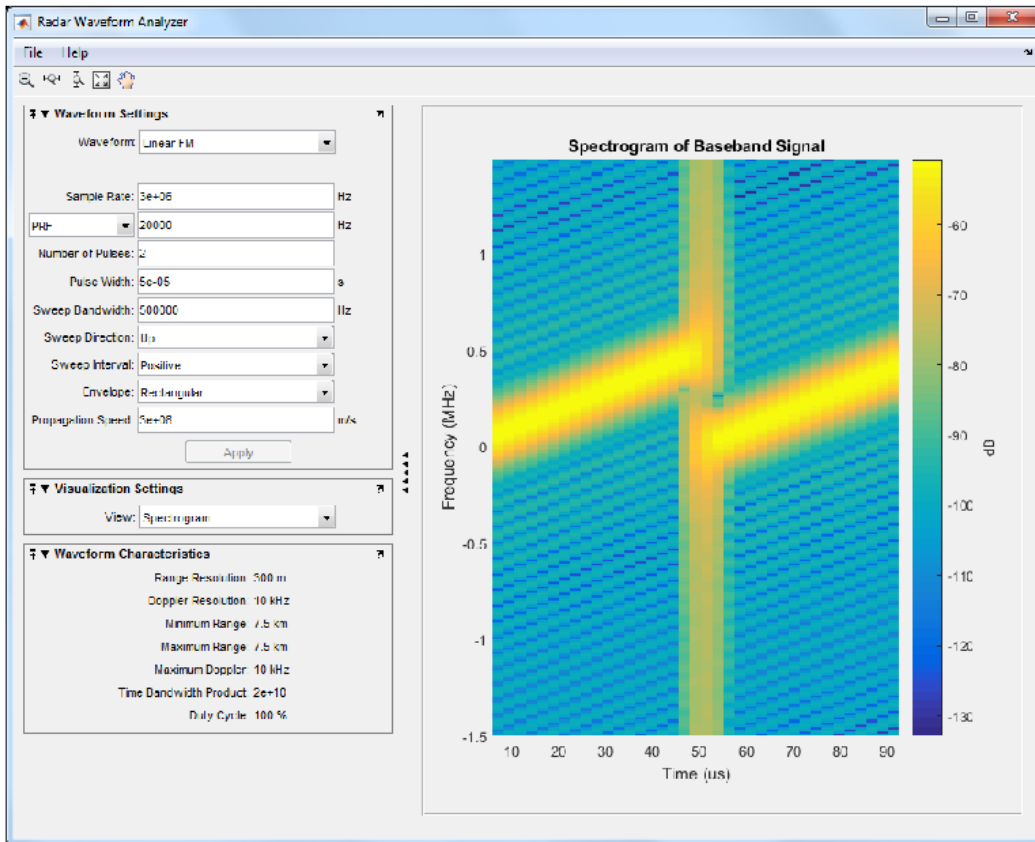
Transmitter

In this section...
"Transmitter Object" on page 4-32
"Phase Noise" on page 4-35

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. You can use `phased.Transmitter` together with `radareqpow`, `radareqrng`, and `radareqsnr`, to relate the received echo power to your transmitter specifications.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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.

```matlab
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.

```matlab
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')
```
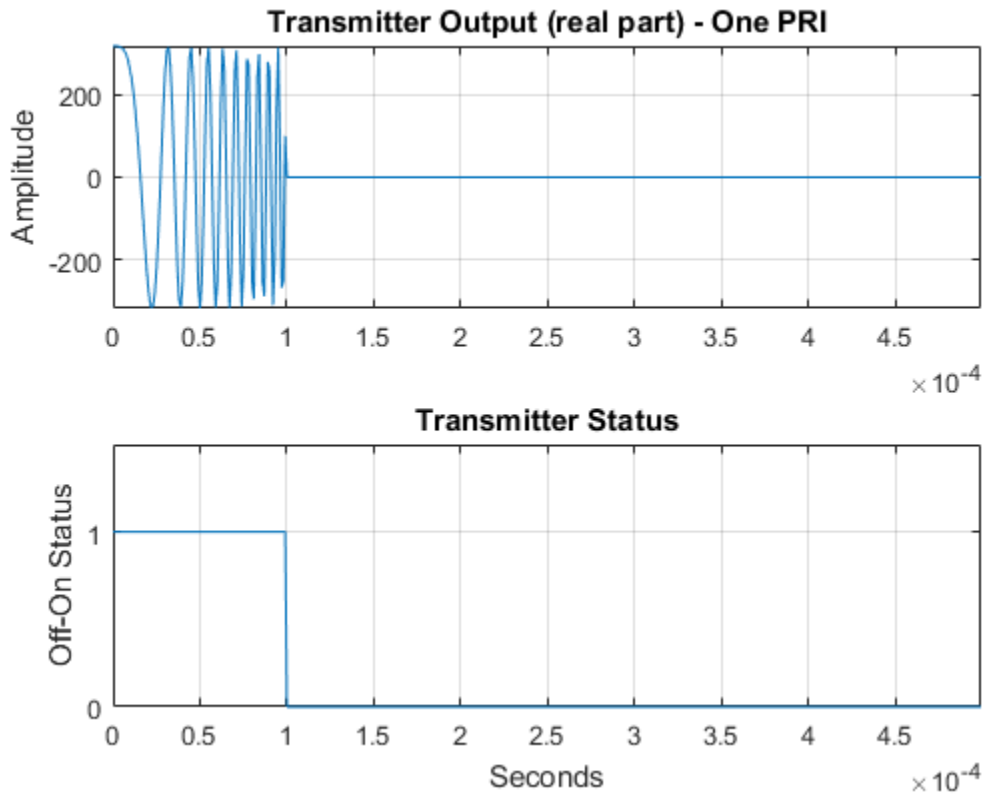
Phase Noise

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™.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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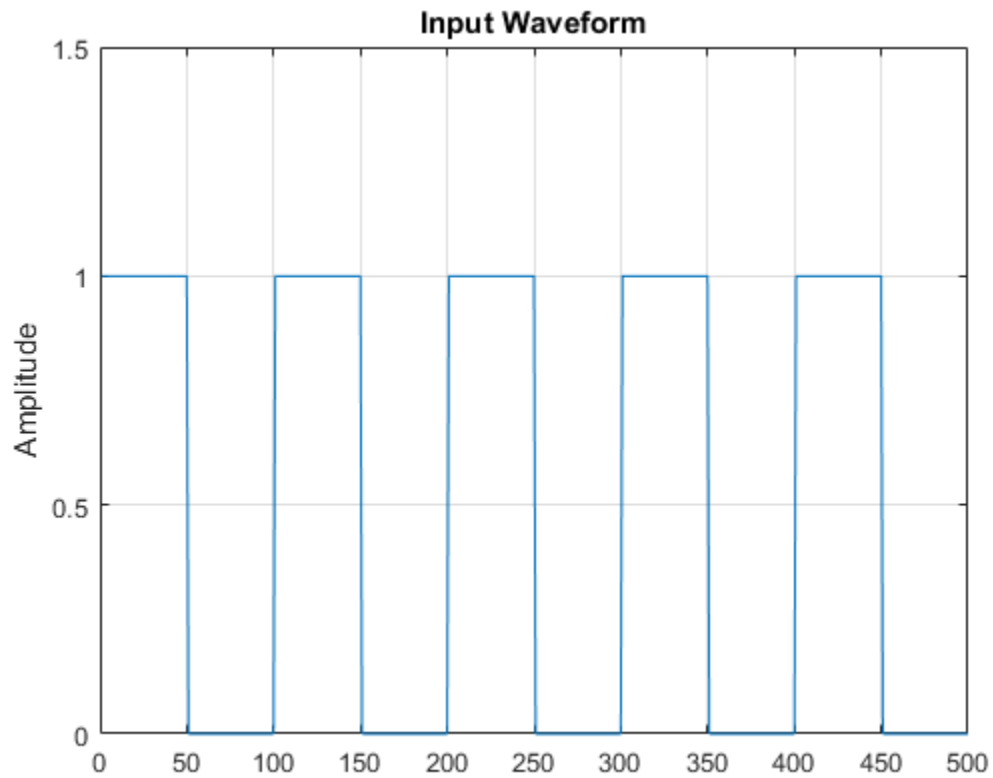
```
subplot(2,1,1)
plot(rad2deg(atan2(imag(txtoutput),real(txtoutput))))
title('Phase of the Output')
ylabel('Degrees')
axis([0 length(wf) -180 180])
grid on
subplot(2,1,2)
plot(phdeg)
title('Phase Noise'); ylabel('Degrees')
axis([0 length(wf) -180 180])
grid on
```

Phase of the Output

Phase Noise
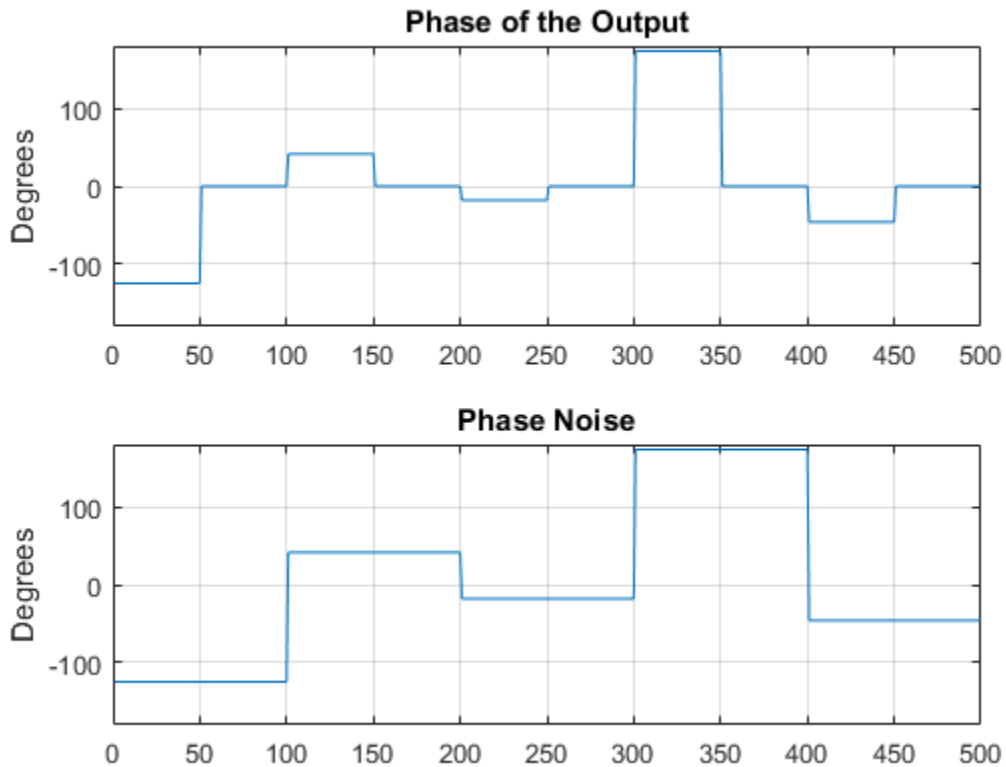
The first figure shows the waveform. The phase of each pulse at the input to the transmitter is zero. In the second figure, the top plot shows the phase of the transmitter output waveform. The bottom plot shows the phase added to each sample. Focus on the first 100 samples. The pulse waveform is equal 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 4-32.

- `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 (σ^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

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

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, σ^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

$$\sigma^2 = k_B BTF$$

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
waveform = phased.RectangularWaveform('NumPulses',5);
transmitter = phased.Transmitter('CoherentOnTransmit',false,...
    'PhaseNoiseOutputPort',true,'Gain',0,'PeakPower',1,...
    'SeedSource','Property','Seed',1000,'InUseOutputPort',true);
```

```matlab
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')
```
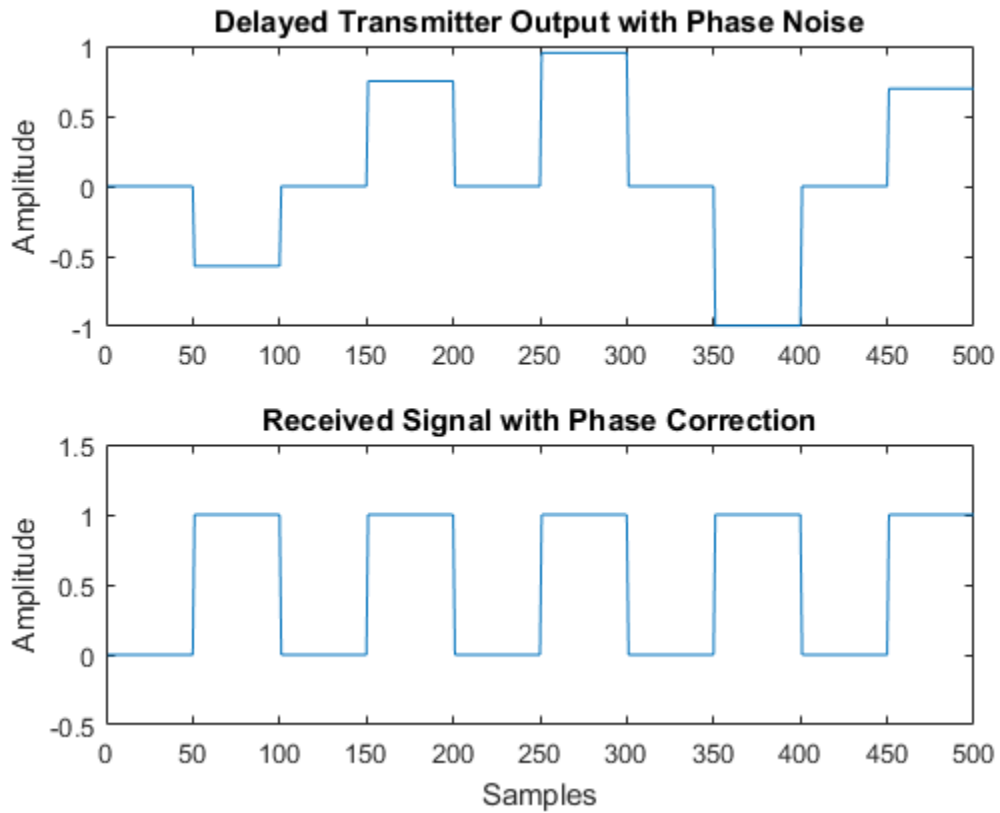
Radar Equation

In this section...

Radar Equation Theory

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. In this equation, the signal model is assumed to be deterministic. The equation for the power at the input to the receiver is:

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R_t^2 R_r^2 L}$$

where the terms in the equation are:

- P_r — Received power in watts.
- P_t — Peak transmit power in watts.
- G_t — Transmitter gain.
- G_r — Receiver gain.
- λ — Radar operating frequency wavelength in meters.
- σ — Target's nonfluctuating radar cross section in square meters.
- L — General loss factor to account for both system and propagation loss.
- R_t — Range from the transmitter to the target.
- R_r — Range from the receiver to the target. If the radar is monostatic, the transmitter and receiver ranges are identical.

The equation for the power at the input to the receiver represents the signal term in the signal-to-noise (SNR) ratio. To model the noise term, assume the thermal noise in the receiver has a white noise power spectral density (PSD) given by:

$$P(f) = kT$$

where k is the Boltzmann constant and T is the effective noise temperature. The receiver acts as a filter to shape the white noise PSD. Assume that the magnitude squared receiver frequency response approximates a rectangular filter with bandwidth equal to the reciprocal of the pulse duration, $1/\tau$. The total noise power at the output of the receiver is:

$$N = \frac{kTF_n}{\tau}$$

where F_n is the receiver *noise figure*.

The product of the effective noise temperature and the receiver noise factor is referred to as the *system temperature* and is denoted by T_s, so that $T_s = TF_n$.

Using the equation for the received signal power and the output noise power, the receiver output SNR is:

$$\frac{P_r}{N} = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R_t^2 R_r^2 L}$$

Solving for the required peak transmit power:

$$P_t = \frac{P_r (4\pi)^3 k T_s R_t^2 R_r^2 L}{N \tau G_t G_r \lambda^2 \sigma}$$

The preceding equations are implemented in the Phased Array System Toolbox by the functions: `radareqpow`, `radareqrng`, and `radareqsnr`. These functions and the equations on which they are based are valuable tools in radar system design and analysis.

Link Budget Calculation Using the Radar Equation

This example shows how to compute the required peak transmit power using the radar equation. You implement a noncoherent detector with a monostatic radar operating at 5 GHz. Based on the noncoherent integration of 10 1μs pulses, you want to achieve a

detection probability of 0.9 with a maximum false-alarm probability of 10^{-6} for a target with a nonfluctuating radar cross section (RCS) of $1m^2$ at 30 km. The transmitter gain is 30 dB. Determine the required SNR at the receiver and use the radar equation to calculate the required peak transmit power.

Use Albersheim's equation to determine the required SNR for the specified detection and false-alarm probabilities

```
Pd = 0.9;
Pfa = 1e-6;
NumPulses = 10;
SNR = albersheim(Pd,Pfa,10)


SNR =

    4.9904
```

The required SNR is approximately 5 dB. Use the function `radareqpow` to determine the required peak transmit power in watts.

```
tgtrng = 30e3;
fc = 5e9;
c = physconst('Lightspeed');
lambda = c/fc;
RCS = 1;
pulsedur = 1e-6;
G = 30;
Pt = radareqpow(lambda,tgtrng,SNR,pulsedur,'rcs',RCS,'gain',G)


Pt =

   5.6485e+03
```

The required peak power is approximately 5.6 kW.

Maximum Detectable Range for a Monostatic Radar

Assume that the minimum detectable SNR at the receiver of a monostatic radar operating at 1 GHz is 13 dB. Use the radar equation to determine the maximum

detectable range for a target with a nonfluctuating RCS of $0.5m^2$ if the radar has a peak transmit power of 1 MW. Assume the transmitter gain is 40 dB and the radar transmits a pulse that is 0.5µs in duration.

```
tau = 0.5e-6;
G = 40;
RCS = 0.5;
Pt = 1e6;
lambda = 3e8/1e9;
SNR = 13;
maxrng = radareqrng(lambda,SNR,Pt,tau,'rcs',RCS,'gain',G)


maxrng =

   3.4516e+05
```

The maximum detectable range is approximately 345 km.

Output SNR at Receiver in Bistatic Radar

Estimate the output SNR for a target with an RCS of $1m^2$. The radar is bistatic. The target is located 50 km from the transmitter and 75 km from the receiver. The radar operating frequency is 10 GHz. The transmitter has a peak transmit power of 1 MW with a gain of 40 dB. The pulse width is 1 µs. The receiver gain is 20 dB.

```
fc = 10e9;
lambda = physconst('LightSpeed')/10e9;
tau = 1e-6;
Pt = 1e6;
TxRvRng =[50e3 75e3];
Gain = [40 20];
snr = radareqsnr(lambda,TxRvRng,Pt,tau,'Gain',Gain)


snr =

   9.0547
```

The estimated SNR is approximately 9 dB.

Display Vertical Coverage Diagram

Display the vertical coverage diagram of an antenna transmitting at 100 MHz and placed 20 meters above the ground. Set the free-space range to 100 km. Use default plotting parameters.

```
freq = 100e6;
ant_height = 20;
rng_fs = 100;
[vcp, vcpangles] = radarvcd(freq,rng_fs,ant_height);
blakechart(vcp, vcpangles);
```
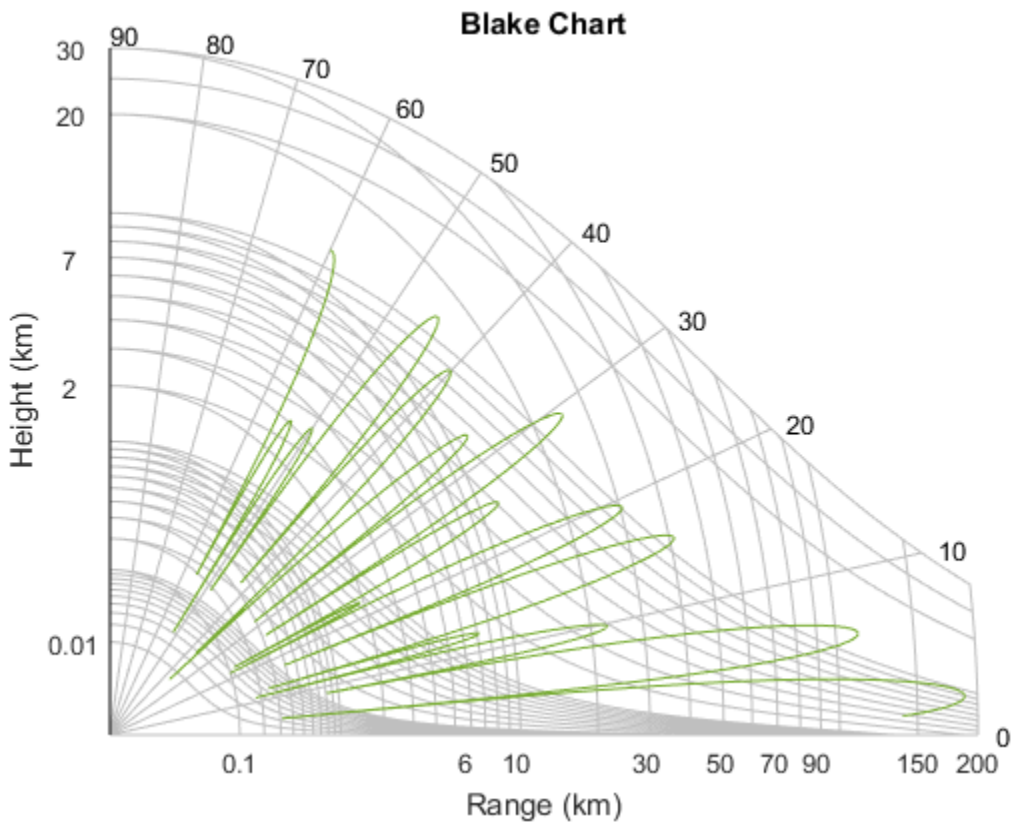
Compute Peak Power Using Radar Equation Calculator App

The `radarEquationCalculator` is a Matlab™ App that lets you determine key radar characteristics such as detection range, required peak transmit power, and SNR. The App works for monostatic and bistatic radars.

Open radarEquationCalculator App

When you type `radarEquationCalculator` from the command line or select the app from the **App Toolstrip**, an interactive window opens. The default window shows a calculation of target range from SNR, power, and other parameters. You can then select various options to compute different radar parameters.

```
radarEquationCalculator
```

Compute Required Peak Transmit Power of Monostatic Radar

As an example, use the app to compute the required peak transmit power for a monostatic radar to detect a large target at 100 km. The radar operates at 10 GHz with

a 40 dB antenna gain. Set the probability of detection to 0.9 and the probability of false alarm to 0.0001.

1. From the **Calculation Type** drop-down list, choose `Peak Transmit Power`
2. Set the **Wavelength** to `3` cm
3. Specify the **Pulse Width** as `2` microseconds
4. Assume total **System Losses** of `5` dB
5. Assuming the target is a large airplane, set **Target Radar Cross Section** value to `100` m2
6. Choose **Configuration** as `Monostatic`
7. Set the **Gain** to be `40` dB
8. Open the **SNR** box
9. Specify the **Probability of Detections** as `0.9`
10. Specify the **Probability of False Alarm** as `0.0001`

Close the App window. Normally, you close the App using the close button.

```
hg = findall(0,'Name','Radar Equation Calculator');
close(hg)
```

You can see from this previously prepared screen shot that the required peak transmit power is .2095 W.

```
filenm = fullfile(matlabroot,'examples','phased','radarEquationExample_03.png');
im = imread(filenm);
figure('Position',[344 206 849 644])
image(im)
axis off
set(gca,'Position',[0.083 0.083 0.834 0.888])
```

5

Beamforming

Conventional Beamforming

In this section...
"Uses for Beamformers" on page 5-2
"Support for Conventional Beamforming" on page 5-2
"Narrowband Phase Shift Beamformer For a ULA" on page 5-2

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 5-13.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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')
```
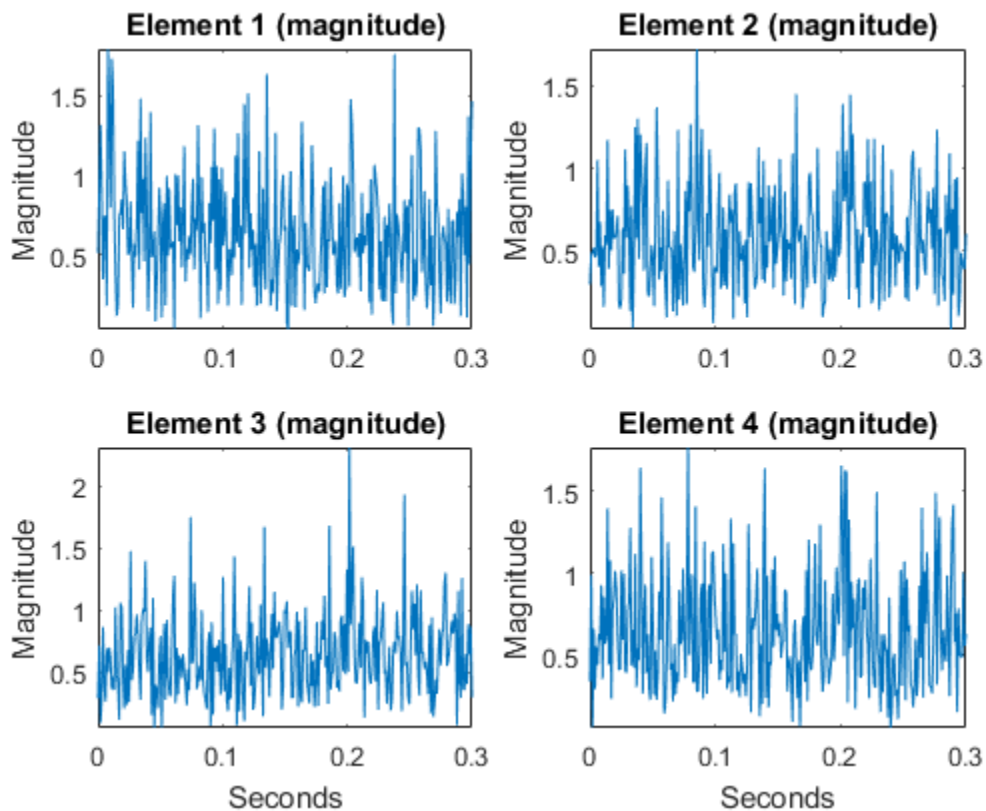

Construct a phase-shift beamformer. Set the `WeightsOutputPort` property to `true` to output the spatial filter weights that point the beamformer to the angle of arrival.

```
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')
```
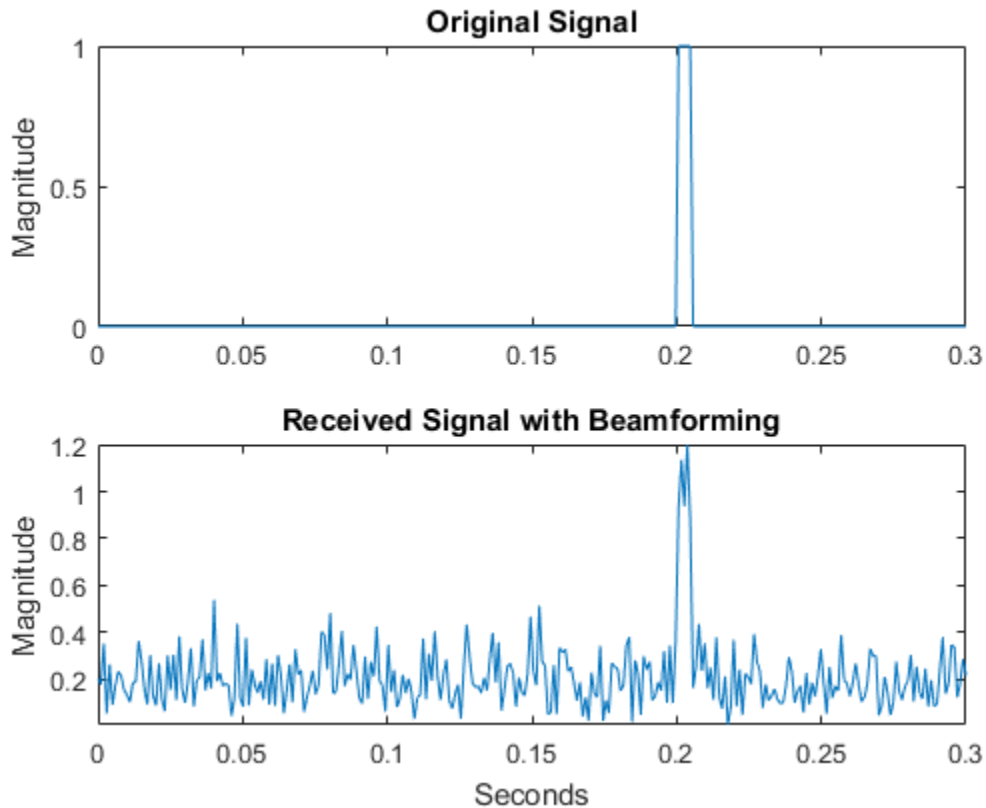
To examine the effect of beamforming weights on the array response, plot the array normalized power response with and without beamforming weights.

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

```
title('Array Response with Beamforming Weights')
```
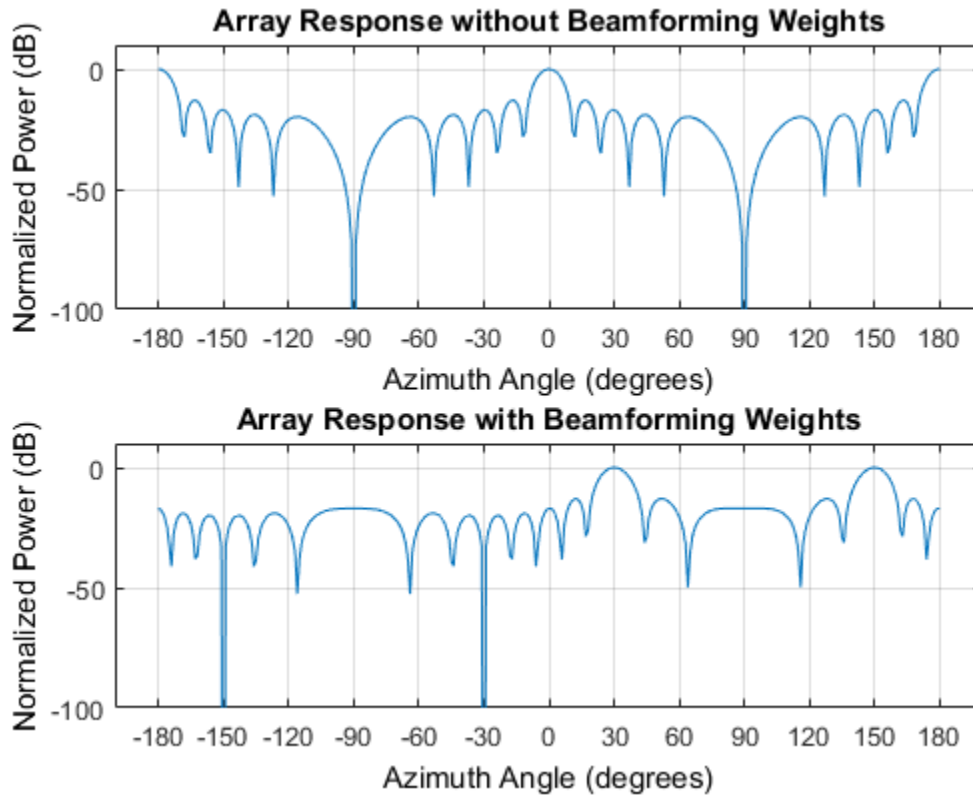

Related Examples

*   "Conventional and Adaptive Beamformers"

Adaptive Beamforming

In this section...
"Benefits of Adaptive Beamforming" on page 5-8
"Support for Adaptive Beamforming" on page 5-8
"LCMV Beamformer" on page 5-8

Benefits of Adaptive Beamforming

"Narrowband Phase Shift Beamformer For a ULA" on page 5-2 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

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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 an interference source using the `phased.BarrageJammer` System object. 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.

```
jammer = phased.BarrageJammer('ERP',10,'SamplesPerFrame',300);
jamsig = jammer();
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
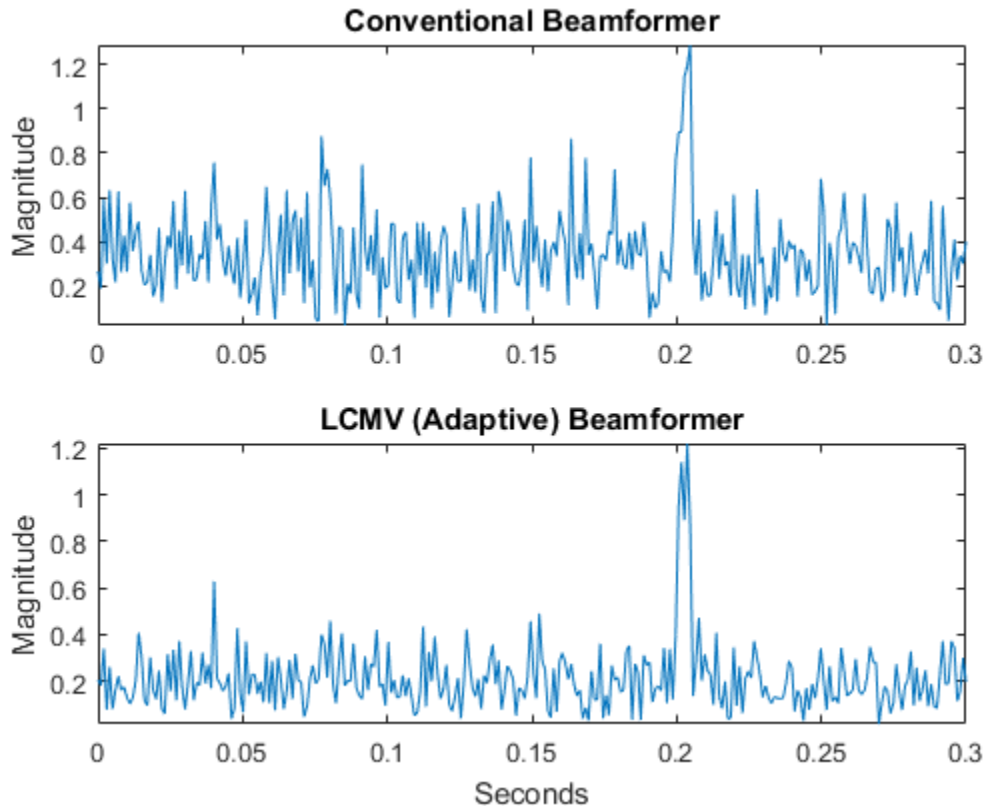
The adaptive beamformer significantly improves the SNR of the rectangular pulse at 0.2 s.

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');
```
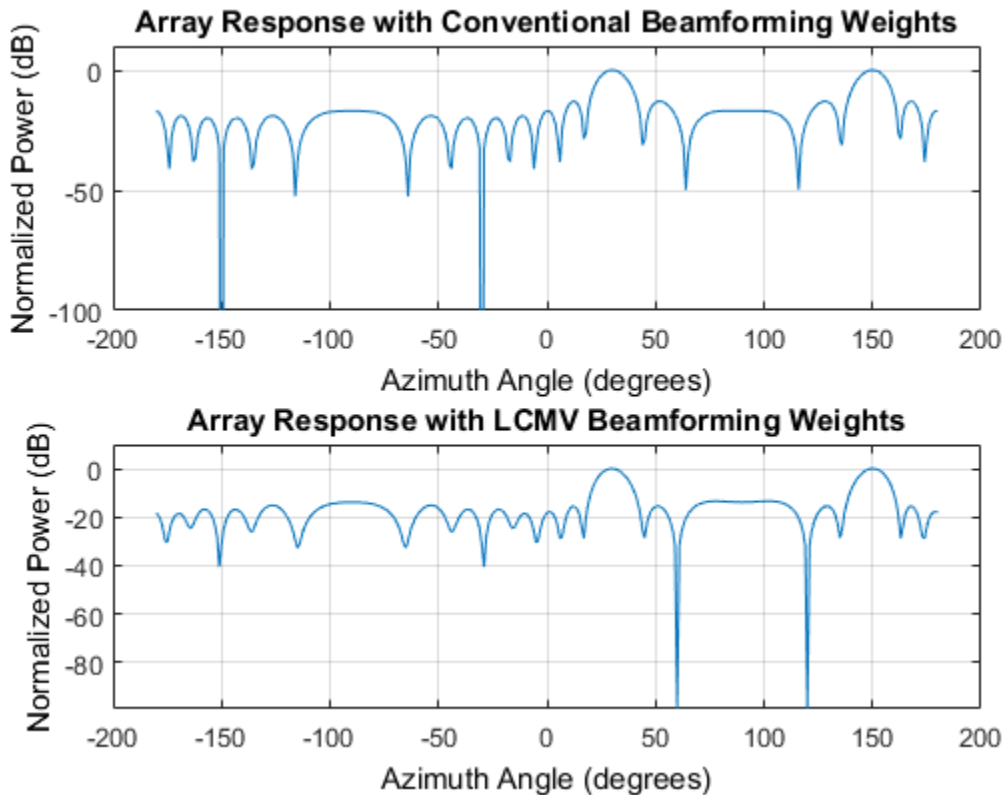

The adaptive beamform places a null at the arrival angle of the interference signal, 120°.

See Also

phased.FrostBeamformer | phased.LCMVBeamformer | phased.MVDRBeamformer

Related Examples

- "Conventional and Adaptive Beamformers"

Wideband Beamforming

In this section...
"Support for Wideband Beamforming" on page 5-13
"Time-Delay Beamforming of Microphone ULA Array" on page 5-13
"Visualization of Wideband Beamformer Performance" on page 5-15

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" 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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 45° 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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')
```

Azimuth Cut (elevation angle = 0.0°)

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)
```


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','re
    '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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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 45° 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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','re
    '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.

Direction-of-Arrival Estimation

Beamscan Direction-of-Arrival Estimation

This example shows how to use the nonparametric beamscan technique to estimate the direction of arrival (DOA) of 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Construct a ULA consisting of ten elements. Assume the carrier frequency of the incoming narrowband sources is 1 GHz.

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

Assume that there is a wavefield incident on the ULA consisting of two linear FM pulses. 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(sULA,[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. Output the DOA estimates, and plot the spatial spectrum.

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

```
plotSpectrum(estimator)
```
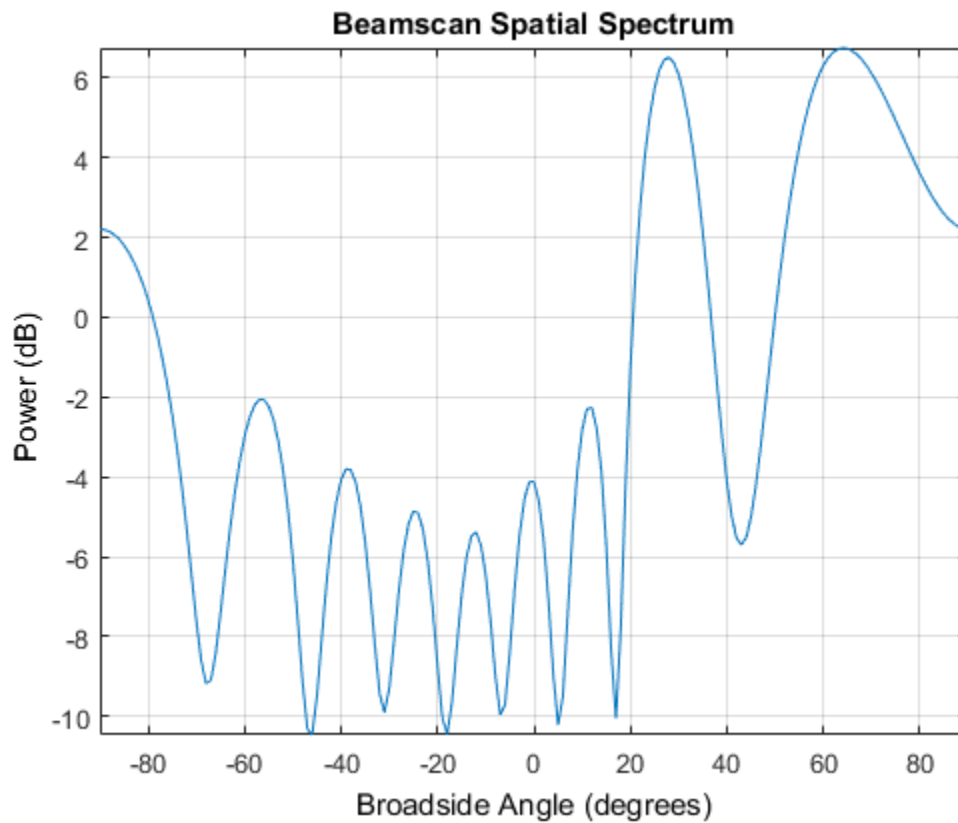

Related Examples

- "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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

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]);
```
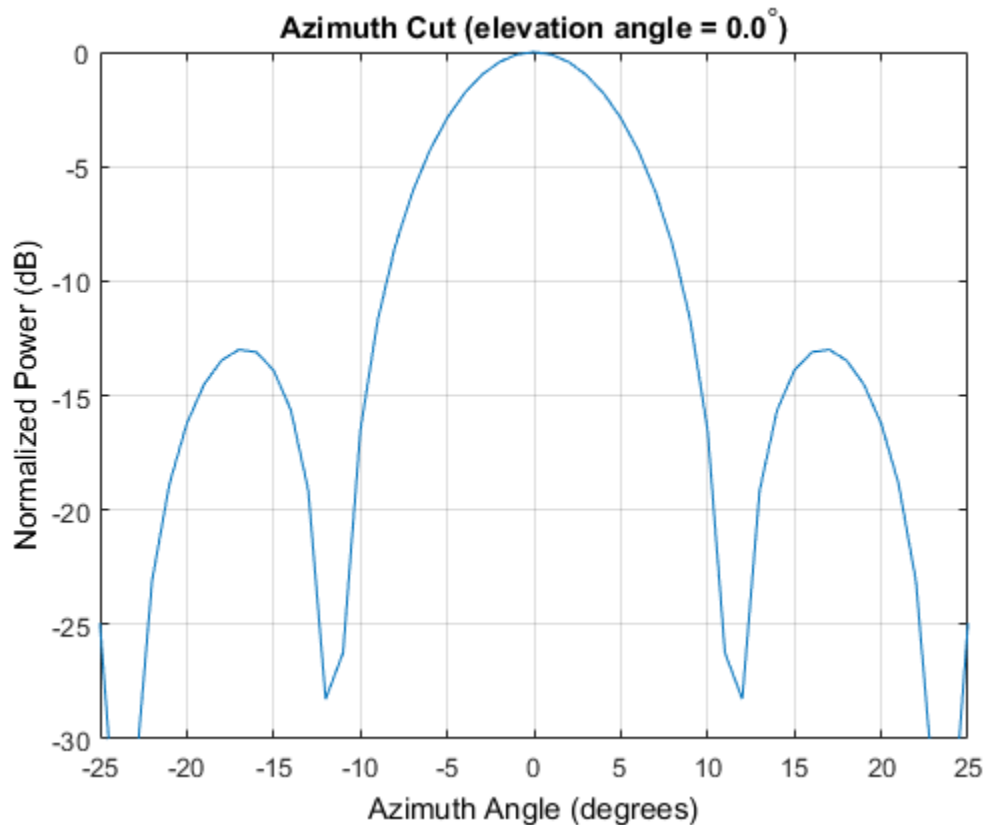
Receive two signal sources with DOAs separated by 10°

```
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)
```
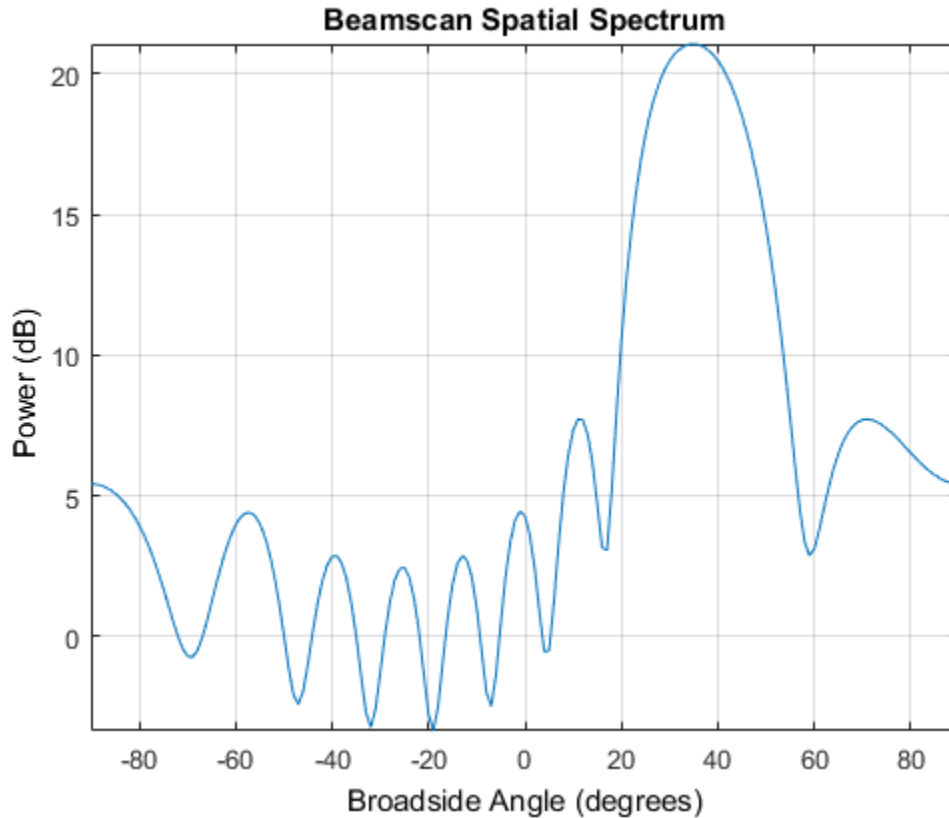

Use the super-resolution DOA estimator to estimate the two directions. This estimator offers better resolution than the nonparametric beamscan estimator.

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

```
doa_est =

    40.0091    30.0048
```

This estimator correctly identifies the two distinct directions of arrival.

See Also

phased.RootMUSICEstimator

Related Examples

- "Beamscan Direction-of-Arrival Estimation" on page 6-2
- High Resolution Direction of Arrival Estimation

MUSIC Super-Resolution DOA Estimation

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.

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

- $x(t)$ 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, θ_d. θ_d is the direction of arrival angle for the dth 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.*

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

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.

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

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 σ_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 σ_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 these cases, you can use root-finding methods to solve for the roots, z_i. These roots do not necessarily lie on the unit circle. 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 root.

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

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

for signals arriving from the broadside angles φ_1 and φ_2. The quantity k is the signal wave number. $a(\varphi)$ represents an arrival vector at the angle φ.

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

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

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.
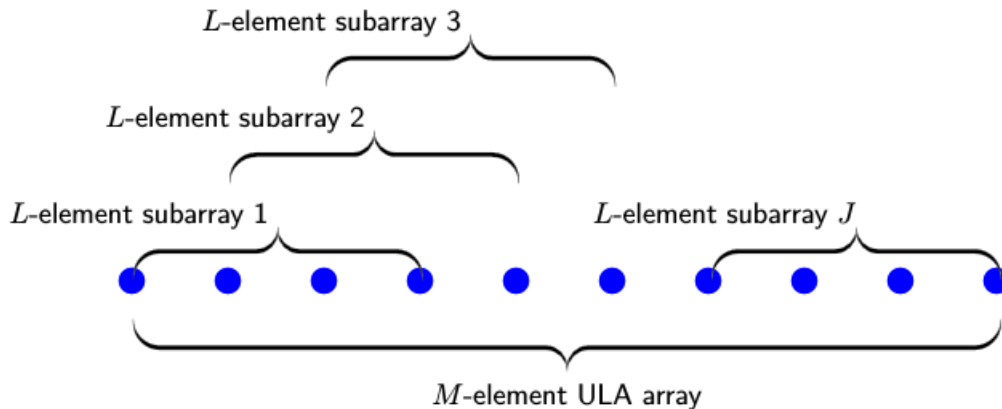
For the pth subarray, the source signal arrival matrix is

$$A_p = \left[e^{ik(p-1)d\cos\varphi_1} a(\varphi_1) \mid e^{ik(p-1)d\cos\varphi_2} a(\varphi_2) \right]$$

$$= \left[a(\varphi_1) \mid a(\varphi_2) \right] \begin{bmatrix} e^{ik(p-1)d\cos\varphi_1} & 0 \\ 0 & e^{ik(p-1)d\cos\varphi_2} \end{bmatrix} = A_1 P^{p-1}$$

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

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.

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

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

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

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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(n));
```

```
    [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)')
```
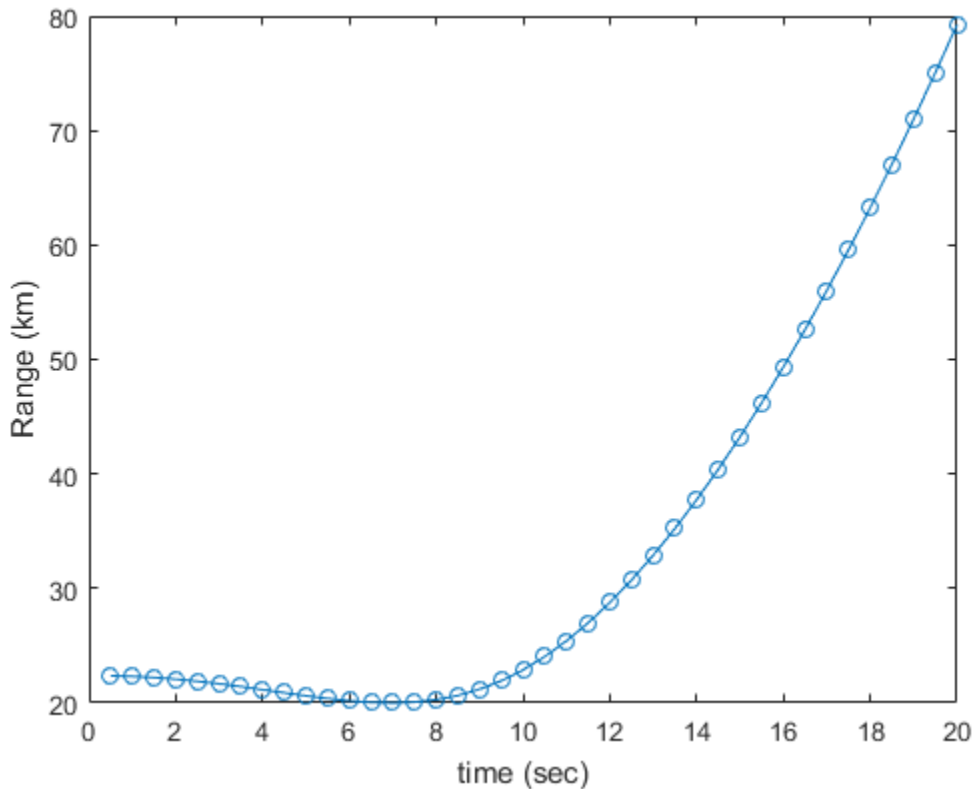
Plot the estimated broadside angle as a function of time.

```
plot(t,broadang_actual,'-o')
xlabel('time (sec)')
ylabel('Broadside angle (deg)')
```
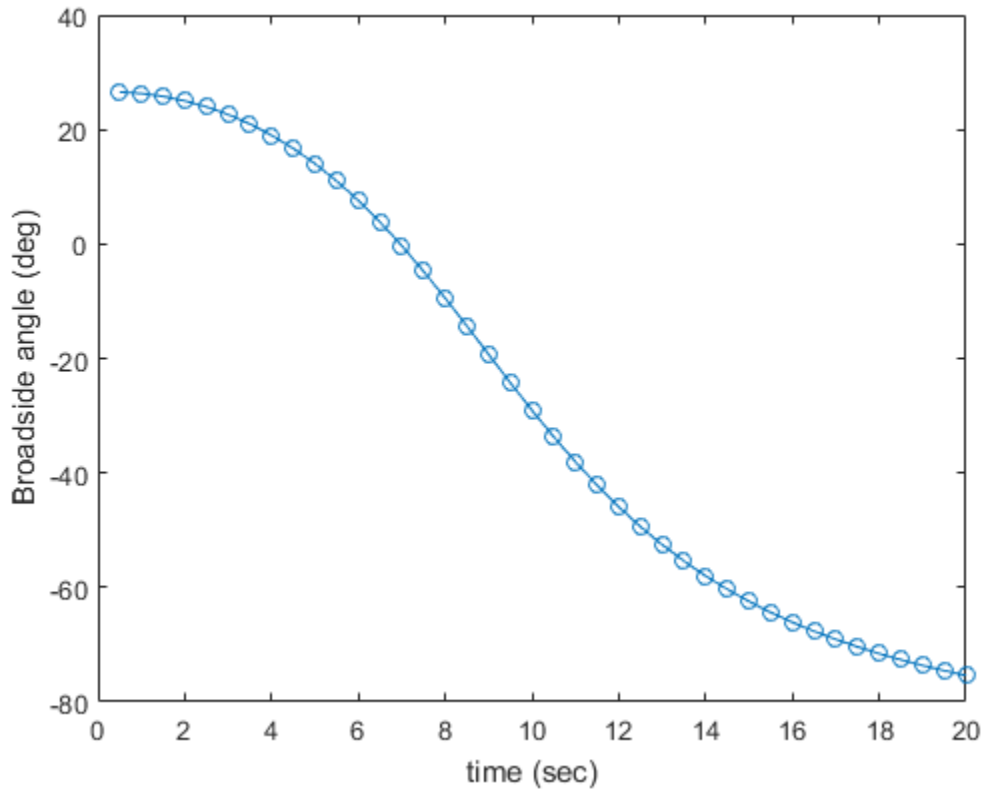

A monopulse tracker cannot solve for the direction angle if the angular separation between samples is too large. The maximum allowable angular separation is approximately one-half the null-to-null beamwidth of the array. For an 8-element, half-wavelength-spaced ULA, the half-beamwidth is approximately 14.3 degrees at broadside. In this simulation, the largest angular difference between samples is

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

```
disp(maxangdiff)
```

```
4.9546
```

Therefore, 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)')
```
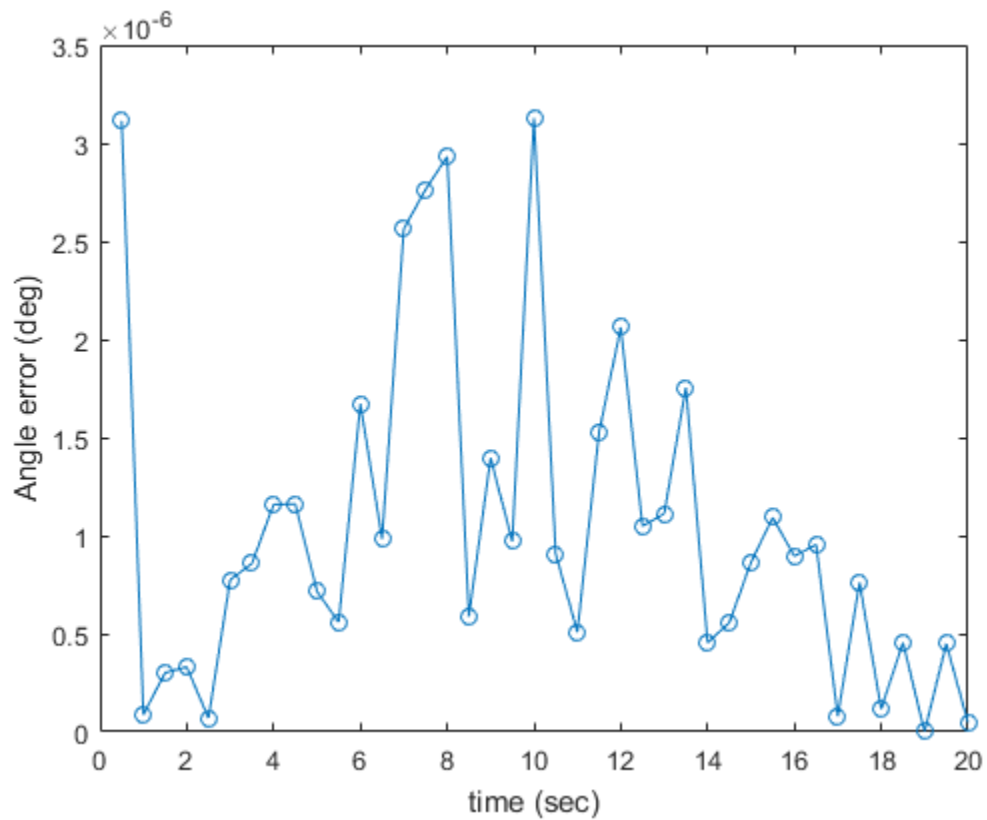
See Also

Functions
musicdoa

System Objects
phased.MUSICEstimator | phased.MUSICEstimator2D

Space-Time Adaptive Processing (STAP)

Angle-Doppler Response

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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

```
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('LightSpeed')/4e9;
```

```matlab
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.

```matlab
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');
```
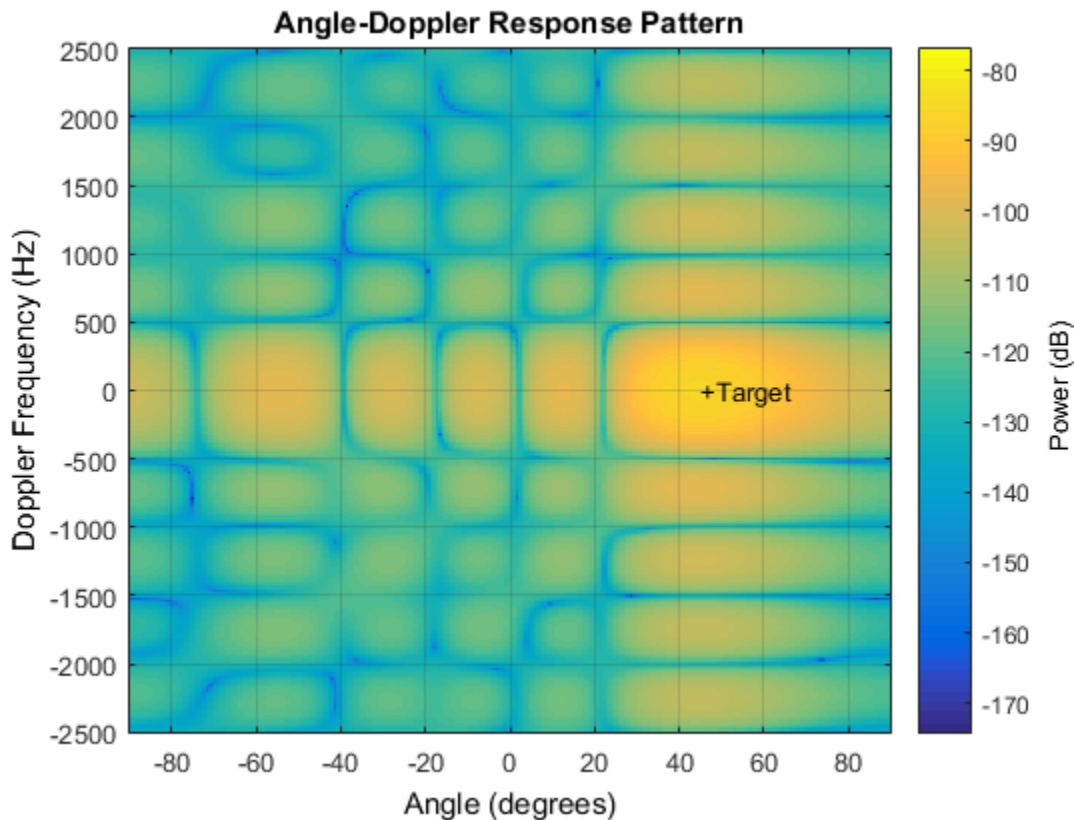
As expected, the angle-Doppler response shows the greatest response at zero Doppler and 45° azimuth.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

The scenario in this example is identical to that of "Angle-Doppler Response of Stationary Array to Stationary Target" 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".

```matlab
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.

```matlab
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');
```

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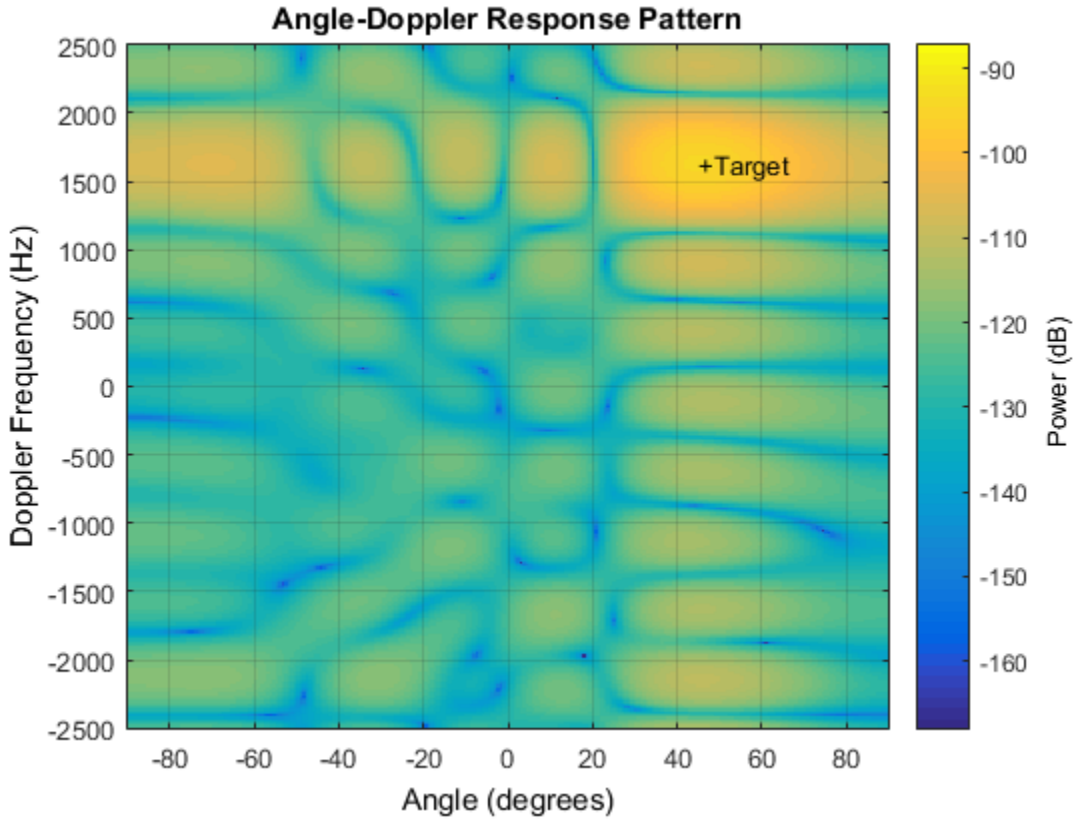
The angle-Doppler response shows the greatest response at 45° azimuth at the expected Doppler shift.

Displaced Phase Center Antenna Pulse Canceller

In this section...
"When to Use the DPCA Pulse Canceller" on page 7-9
"DPCA Pulse Canceller to Reject Clutter" on page 7-9

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

  $$vT = d / 2$$

  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 interva
- 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)*.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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]);
clutter = phased.ConstantGammaClutter('Sensor',array,...
    'PropagationSpeed',radiator.PropagationSpeed,...
    'OperatingFrequency',radiator.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',radiator.OperatingFrequency),...
    'EarthModel','Flat',...
```

```
    'BroadsideDepressionAngle',0,...
    'MaximumRange',radiator.PropagationSpeed/(2*PRF),...
    'PlatformHeight',transmitplatform.InitialPosition(3),...
    'PlatformSpeed',norm(transmitplatform.Velocity),...
    'PlatformDirection',[90;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);
```

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.

```
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;
clutter.SeedSource = 'Property';
clutter.Seed = 5;

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
    csig(:,:,n) = clutter(txsig1(abs(txsig1)>0)); % collect clutter
    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');
```
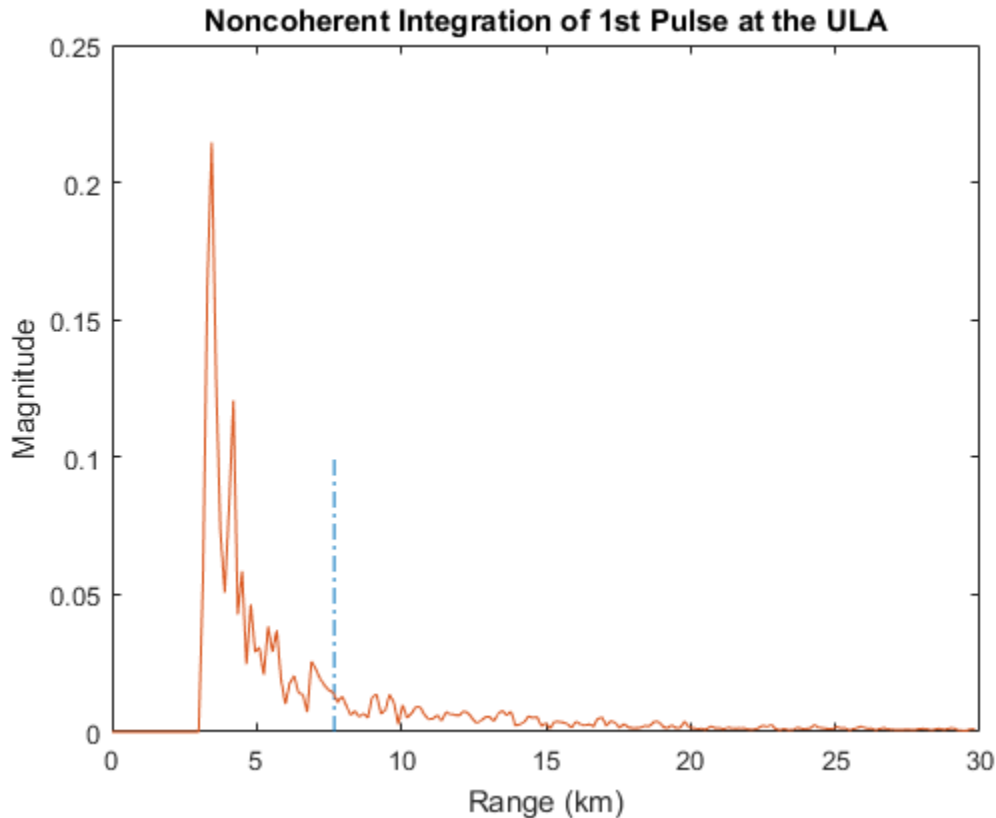
The large-magnitude clutter returns obscure the presence of the target. Apply the DPCA pulse canceller to reject the clutter.

```
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')
```
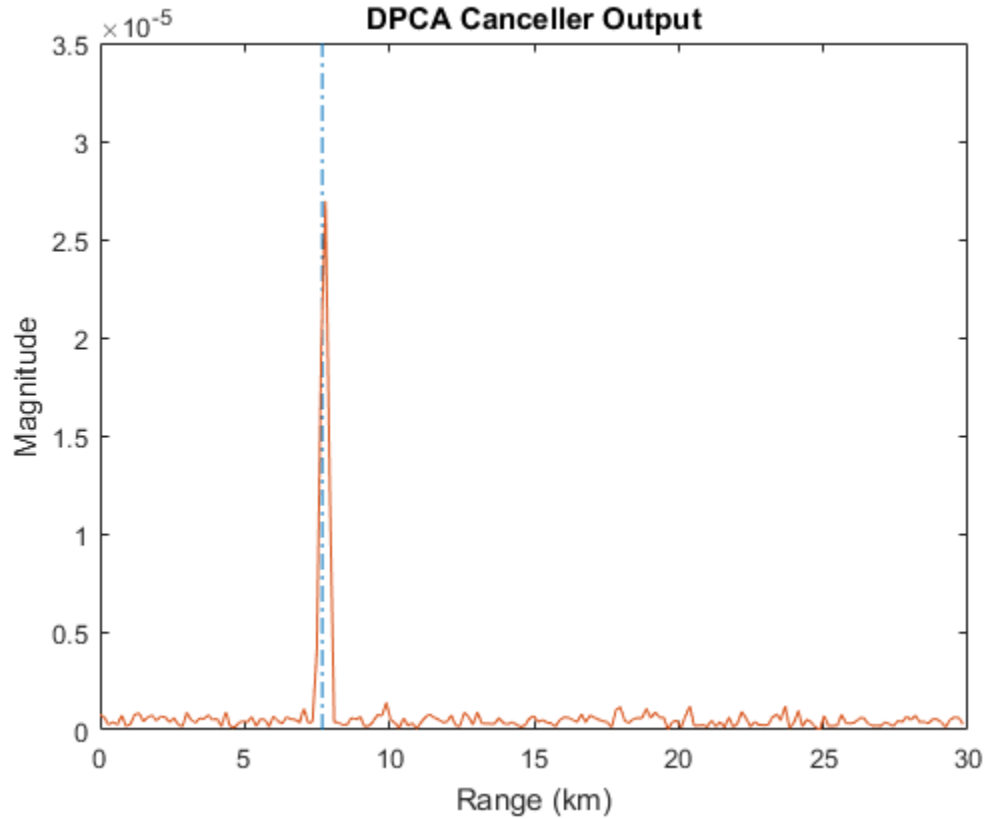

The DPCA pulse canceller has significantly rejected the clutter. As a result, the target is visible at the expected range gate.

Adaptive Displaced Phase Center Antenna Pulse Canceller

In this section...
"When to Use the Adaptive DPCA Pulse Canceller" on page 7-15
"Adaptive DPCA Pulse Canceller" on page 7-15

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

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" except that a stationary broadband barrage jammer is added at *(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 interva
- 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).

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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]);
clutter = phased.ConstantGammaClutter('Sensor',array,...
    'PropagationSpeed',radiator.PropagationSpeed,...
    'OperatingFrequency',radiator.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',radiator.OperatingFrequency),...
    'EarthModel','Flat',...
    'BroadsideDepressionAngle',0,...
```

```
    'MaximumRange',radiator.PropagationSpeed/(2*PRF),...
    'PlatformHeight',transmitterplatform.InitialPosition(3),...
    'PlatformSpeed',norm(transmitterplatform.Velocity),...
    'PlatformDirection',[90;0]);
target = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
jammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200);
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;
clutter.SeedSource = 'Property';
clutter.Seed = 40543;
jammer.SeedSource = 'Property';
jammer.Seed = 96703;
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
        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) = 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')
```
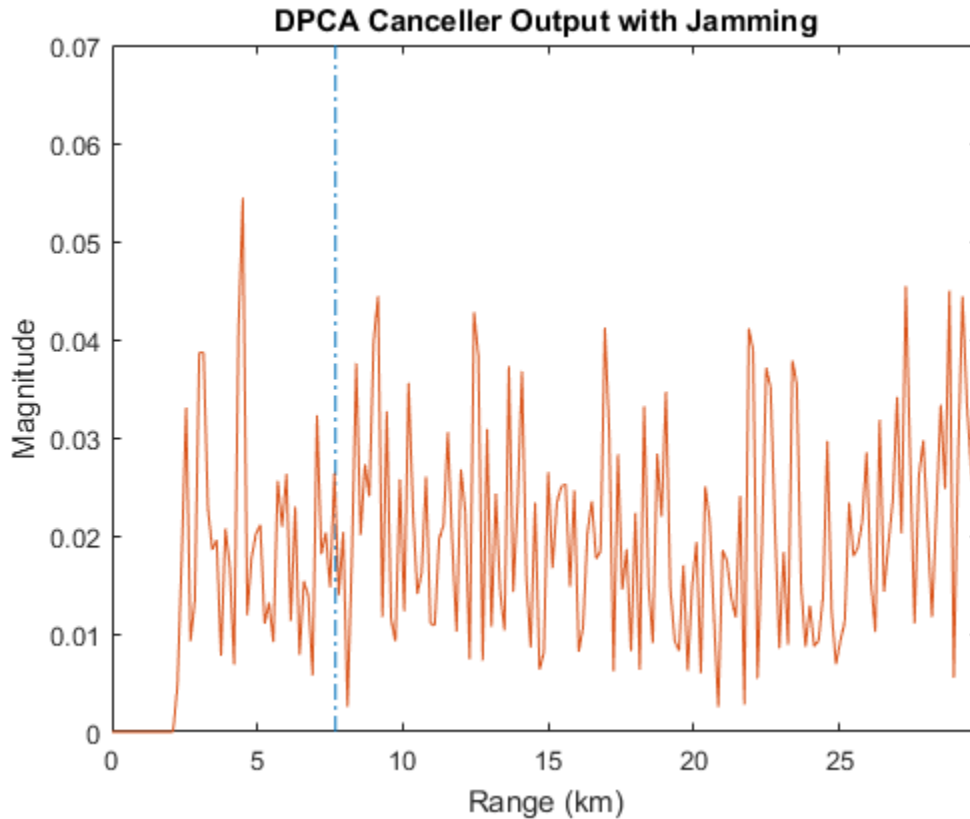

Apply the adaptive DPCA pulse canceller. Use 100 training cells and 4 guard cells, two on each side of the target range gate.

```
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')
```
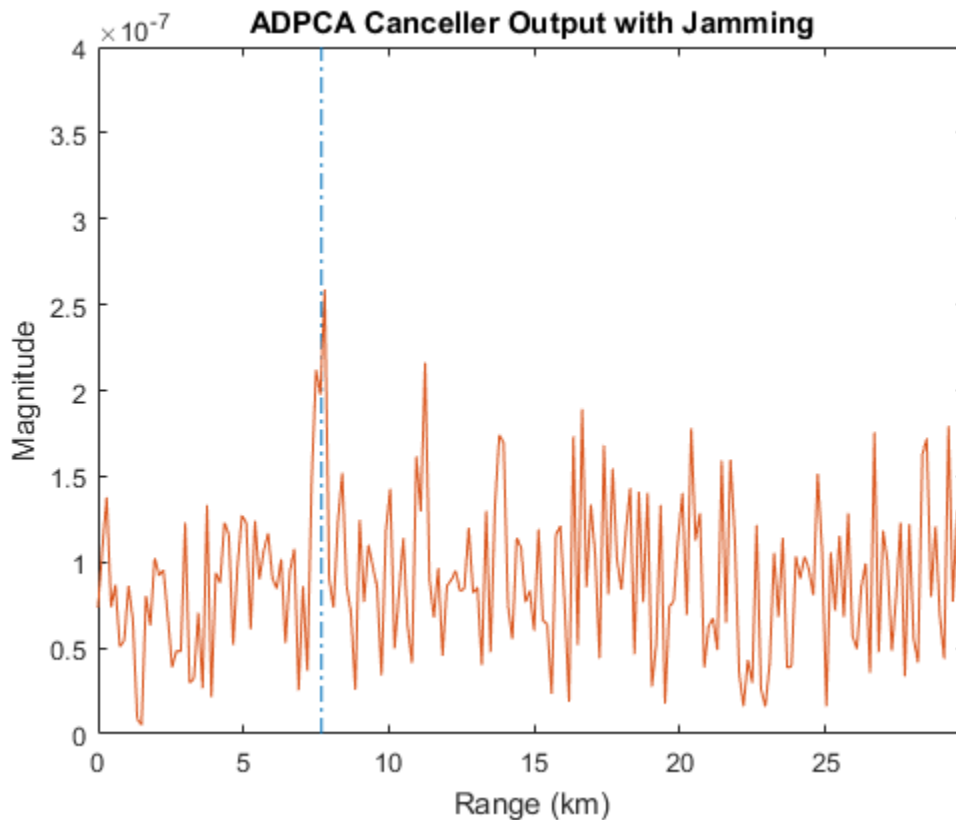
Examine the angle-Doppler response. Notice the presence of the clutter ridge in the angle-Doppler plane and the null at the jammer's broadside angle for all Doppler frequencies.

```
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')
```
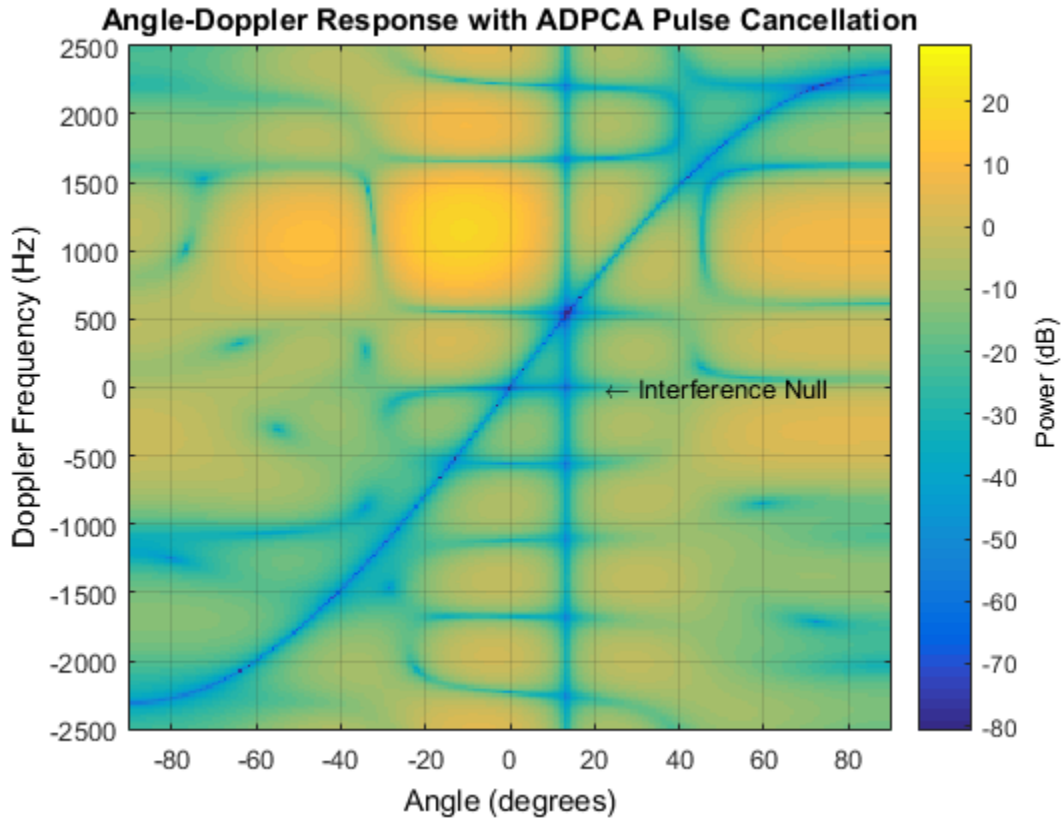
Sample Matrix Inversion Beamformer

In this section...
"When to Use the SMI Beamformer" on page 7-22
"Sample Matrix Inversion Beamformer" on page 7-22

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". 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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]);
clutter = phased.ConstantGammaClutter('Sensor',array,...
    'PropagationSpeed',radiator.PropagationSpeed,...
    'OperatingFrequency',radiator.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',radiator.OperatingFrequency),...
    'EarthModel','Flat',...
    'BroadsideDepressionAngle',0,...
    'MaximumRange',radiator.PropagationSpeed/(2*PRF),...
    'PlatformHeight',transmitterplatform.InitialPosition(3),...
    'PlatformSpeed',norm(transmitterplatform.Velocity),...
    'PlatformDirection',[90;0]);
target = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
targetplatform = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
jammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200);
```

```matlab
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')
```
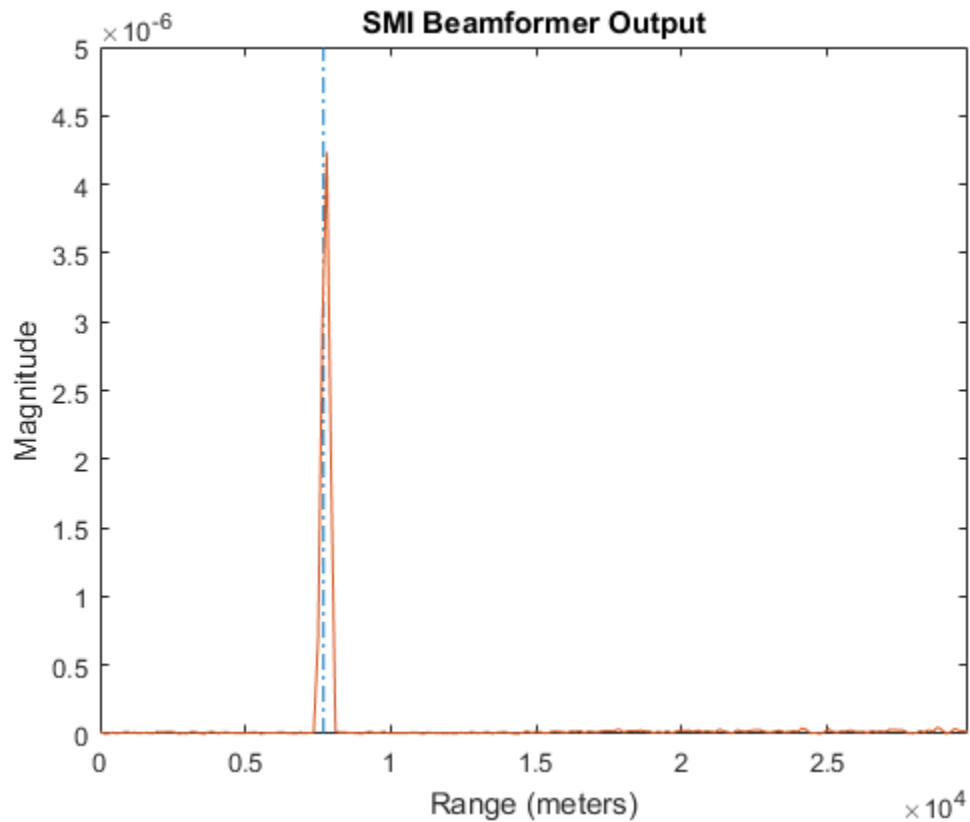
Plot the angle-Doppler response with the beamforming weights.

```
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
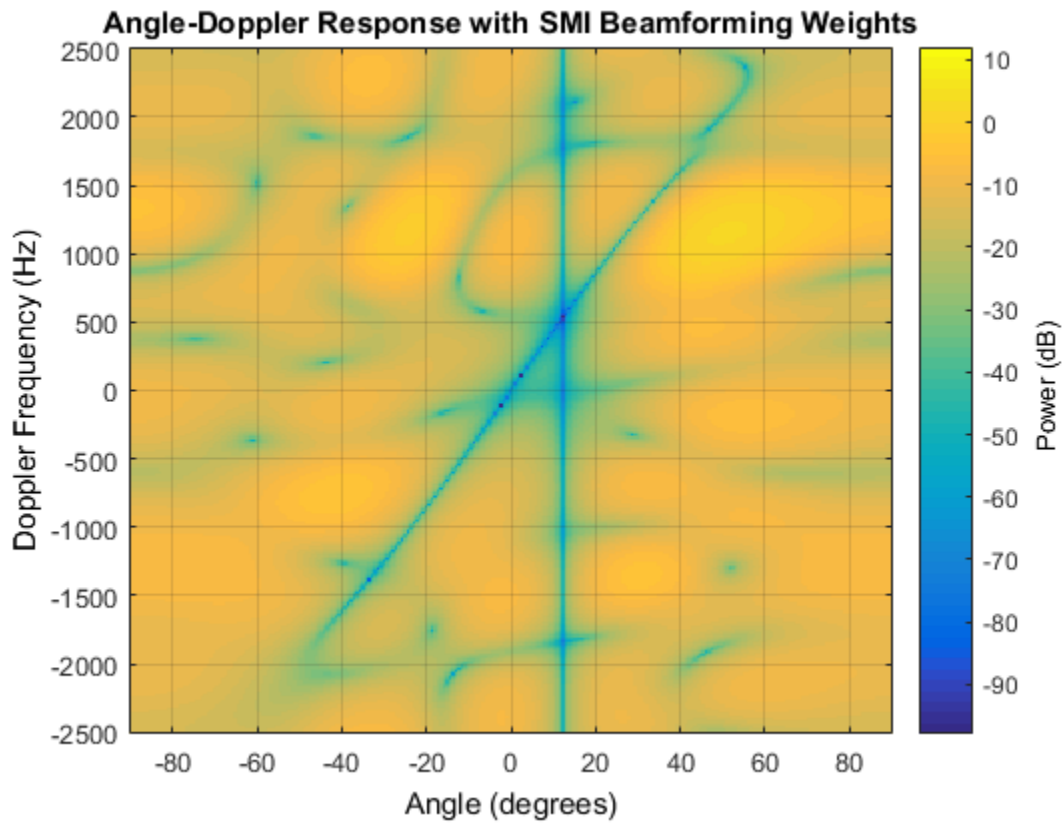
Detection

Neyman-Pearson Hypothesis Testing

In this section...

Purpose of Hypothesis Testing

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')
```
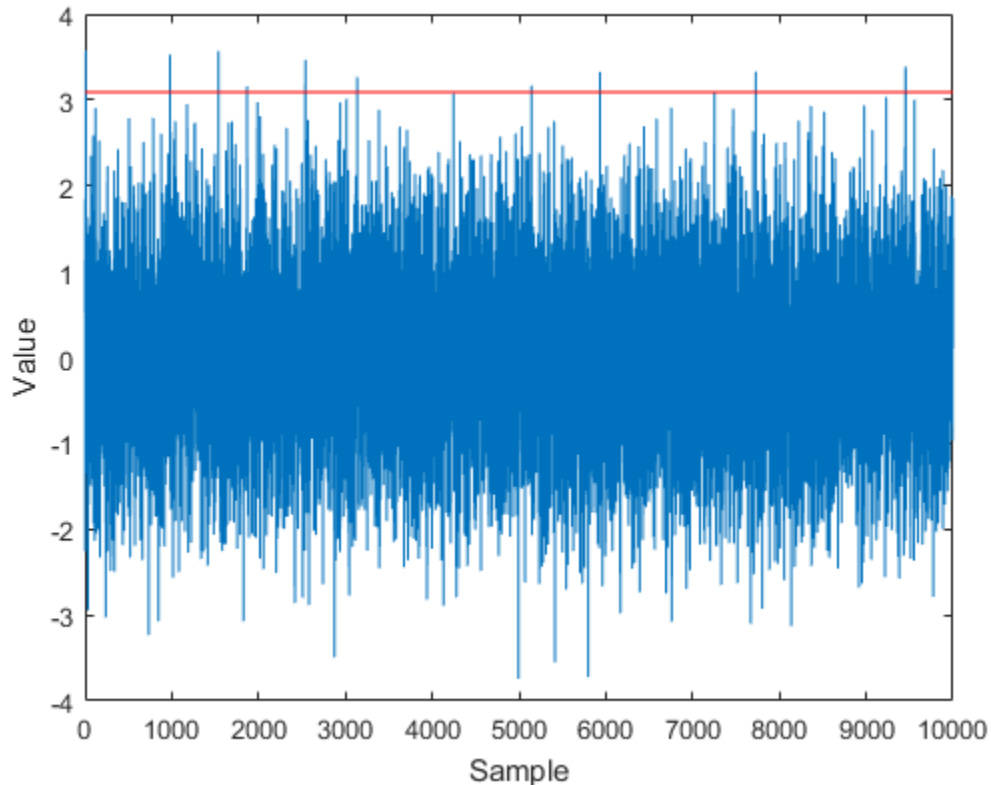
You can see that few sample values exceed the threshold. This result is expected because of the small false-alarm probability.

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')
```
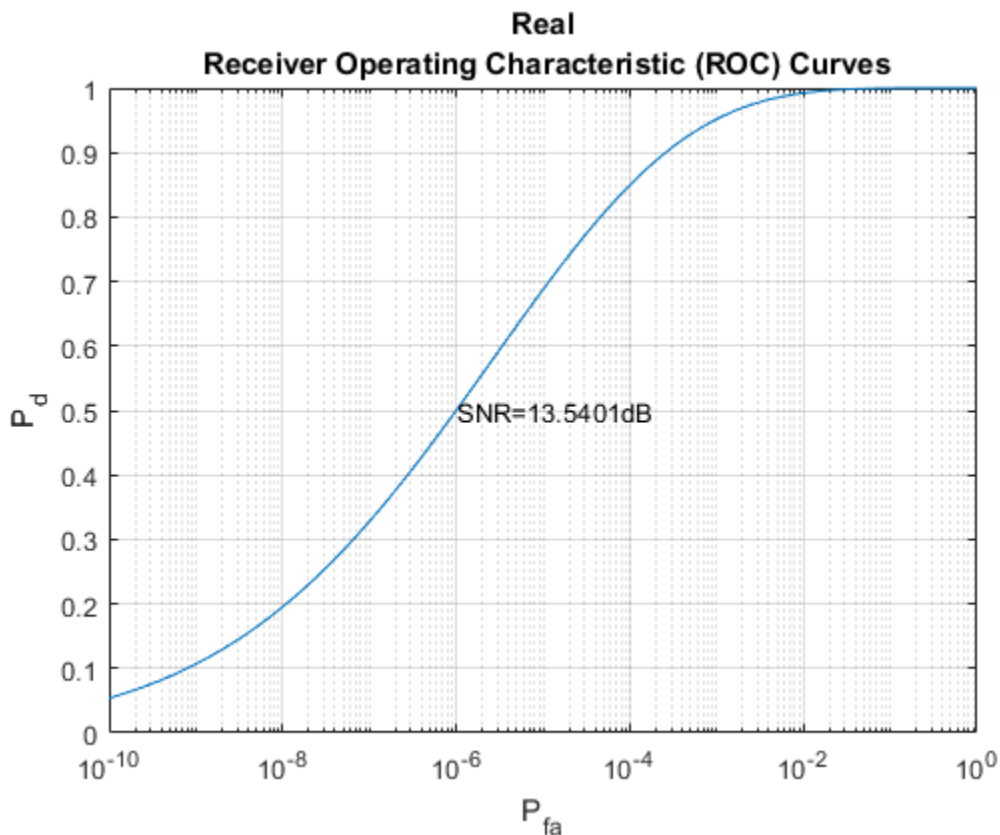
The ROC curve lets you easily read off the probability of detection for a given false-alarm rate.

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);
```

```matlab
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
```
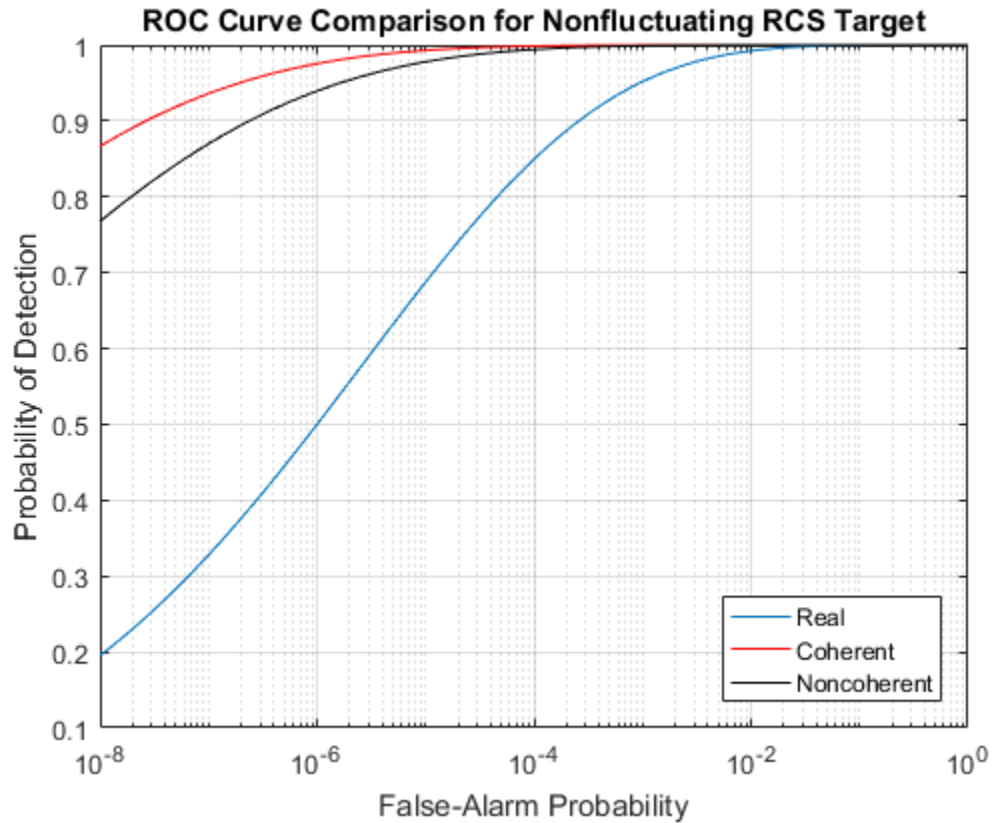

The ROC curves clearly demonstrate the superior probability of detection performance for coherent and noncoherent detectors over the real-valued case.

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')
```
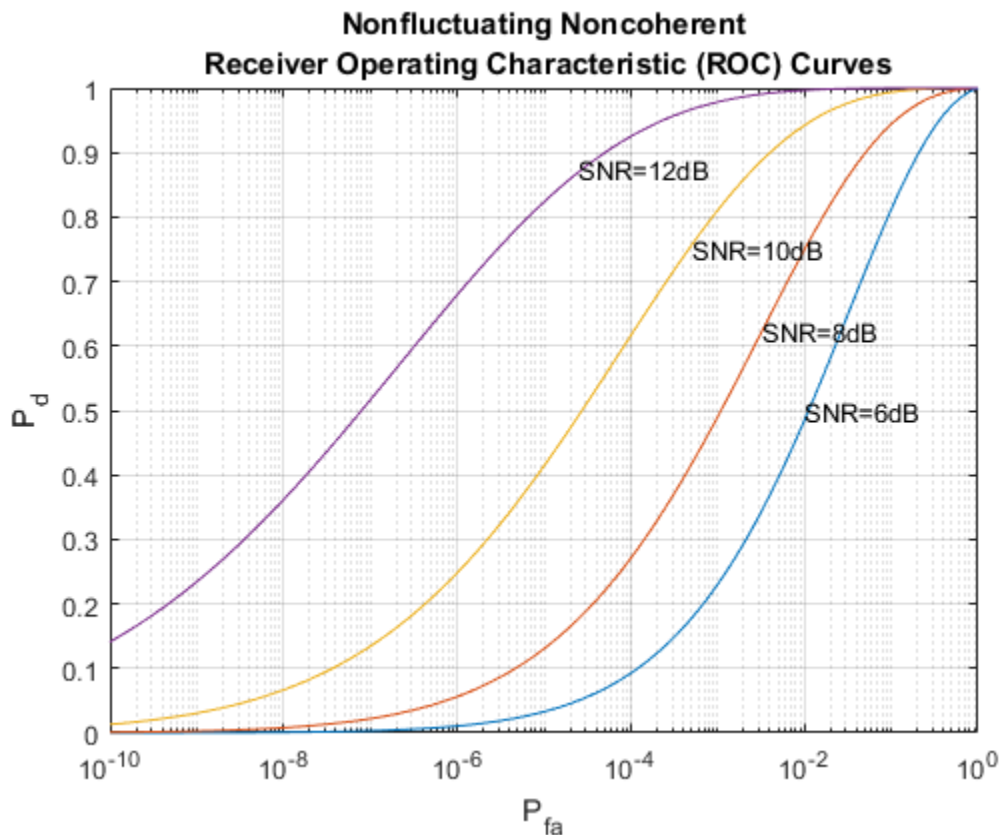

The graph shows that, as the SNR increases, the supports of the probability distributions under the null and alternative hypotheses become more disjoint. Therefore, for a given false-alarm probability, the probability of detection increases.

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')
```
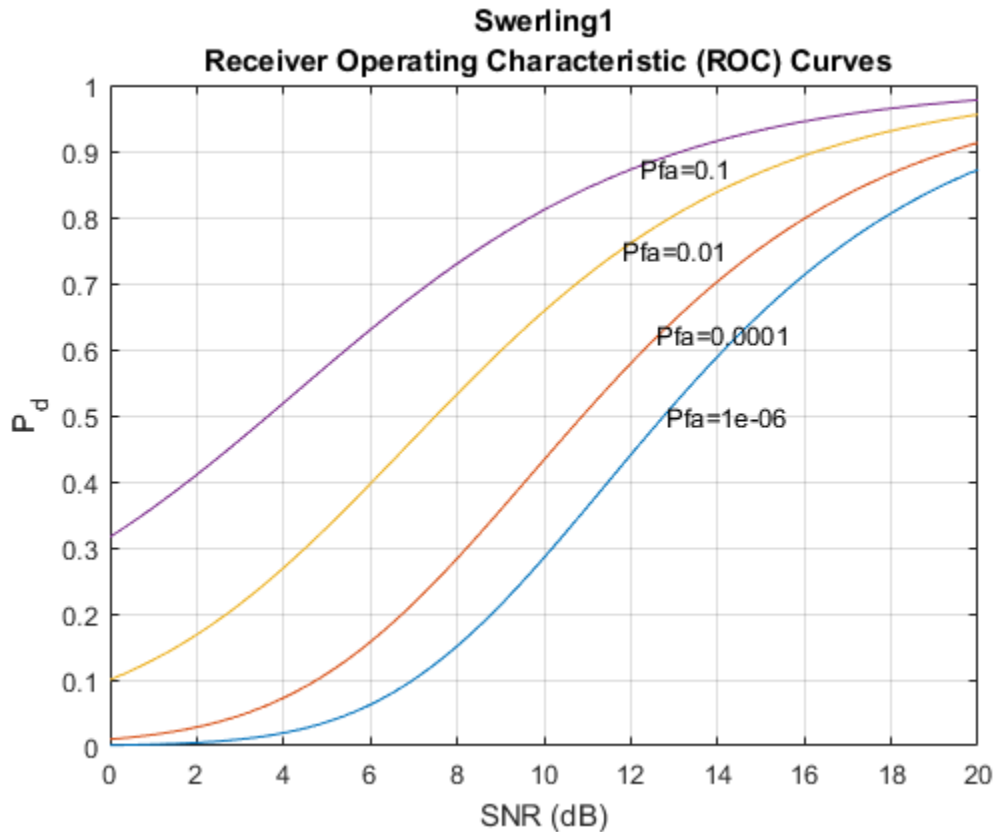

% Use `rocpfa` to examine the effect of SNR on the probability of detection for a detector using noncoherent integration with a false-alarm probability of *1e-4*. Assume the target has a nonfluctuating RCS and that you are integrating over 5 pulses.

```
[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)');
```

Nonfluctuating Noncoherent Detector (5 Pulses)
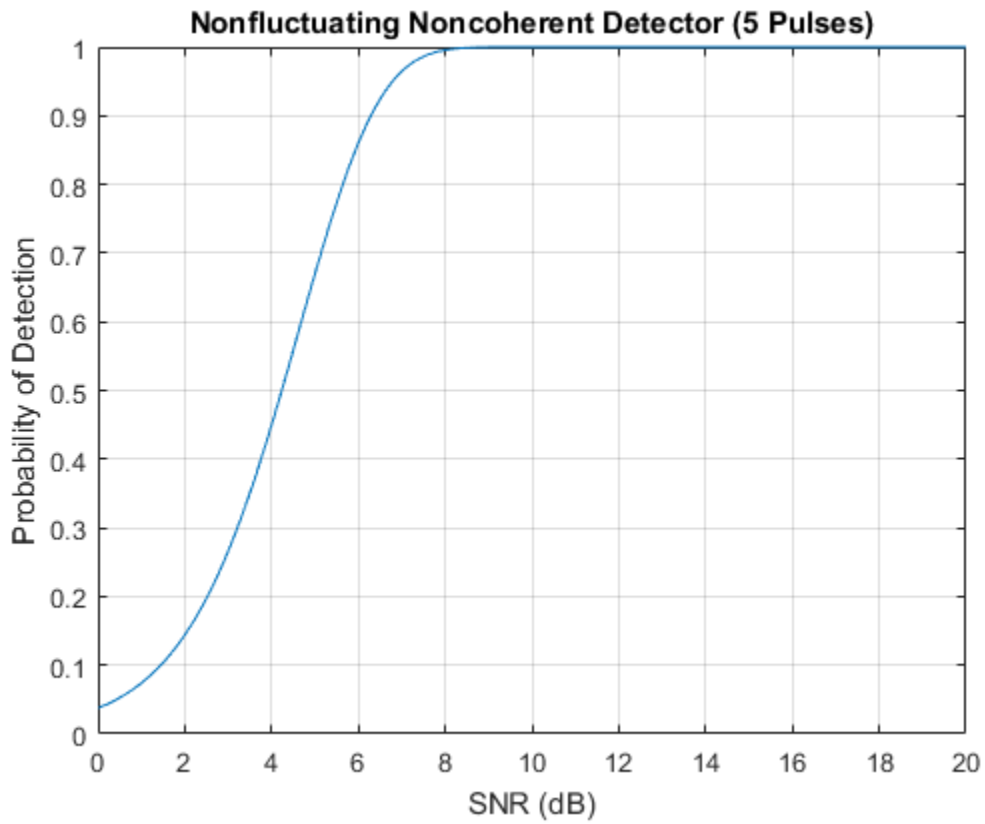
Related Examples

- 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 acheive that 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 to achieve this SNR, sufficient power must be transmitted to the target. Use the `radareqpow` function to estimate the peak transmit 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.

```
txrx_gain = 20;
peak_power = radareqpow(lambda,target_range,...
    snr_min,pulse_duration,'RCS',tgt_rcs,...
    'Gain',txrx_gain,'Ts',290.0);
```

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

Specified 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 to 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
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');
```


Target-Absent Vs Target-Present Histograms
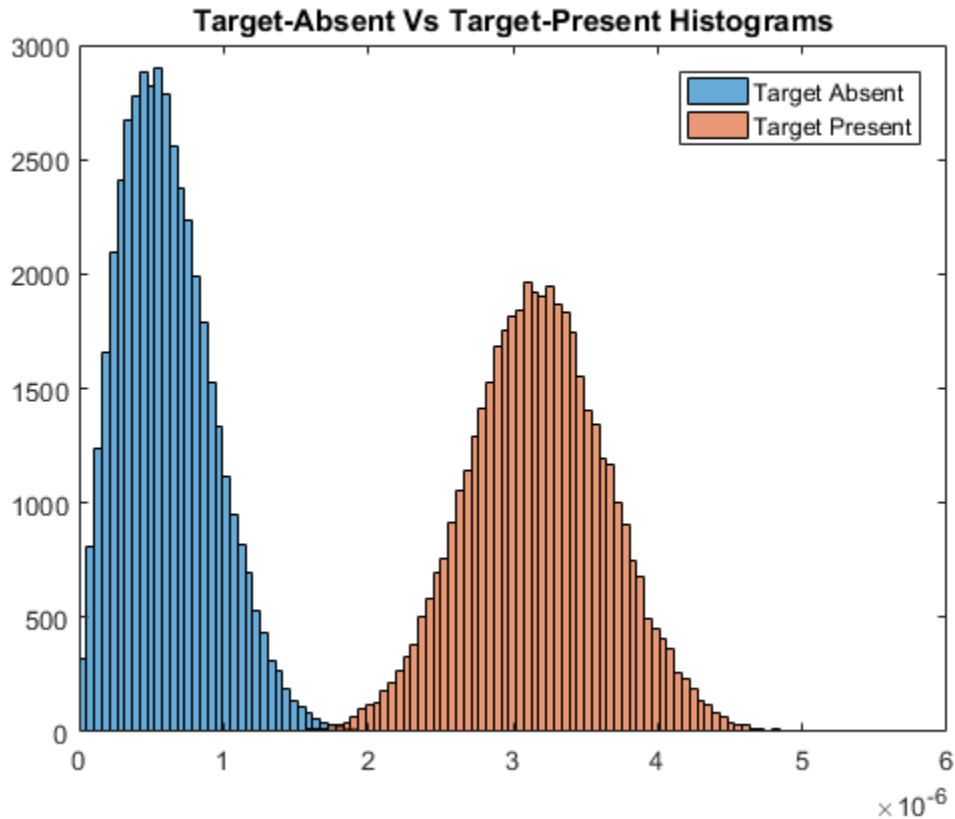
Compare Simulated and Theoretical *Pd* and *Pfa*

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');
```


Simulated and Theoretical ROC Curves
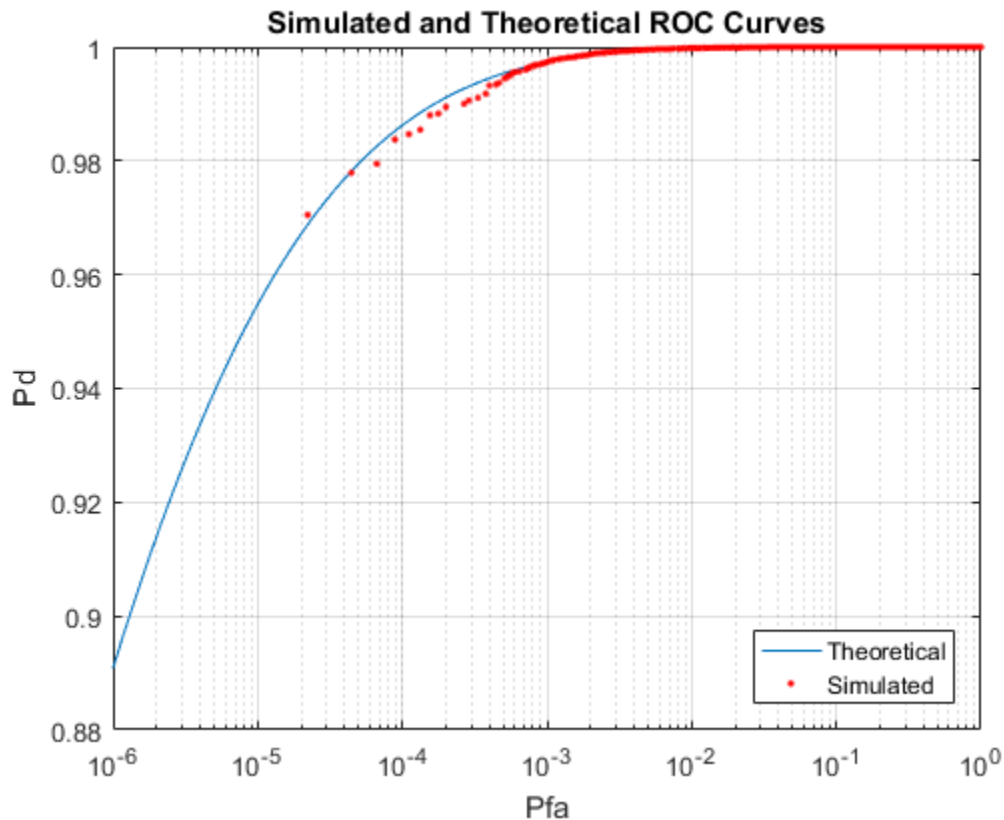
Improve Simulation Using One Million Pulses

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.

Simulated and Theoretical ROC Curves
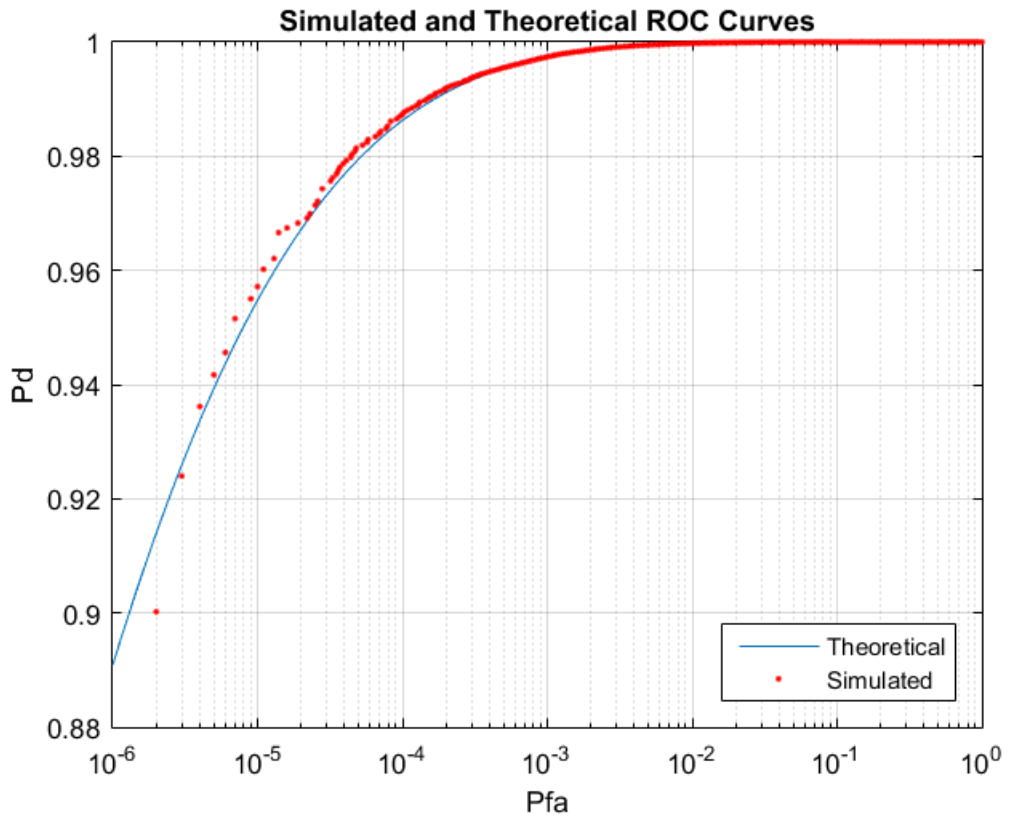
Matched Filtering

In this section...
"Reasons for Using Matched Filtering" on page 8-23
"Support for Matched Filtering" on page 8-23
"Matched Filtering of Linear FM Waveform" on page 8-23
"Matched Filtering to Improve SNR for Target Detection" on page 8-26

Reasons for Using Matched Filtering

You can see from the results in "Receiver Operating Characteristics" on page 8-7 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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')
```
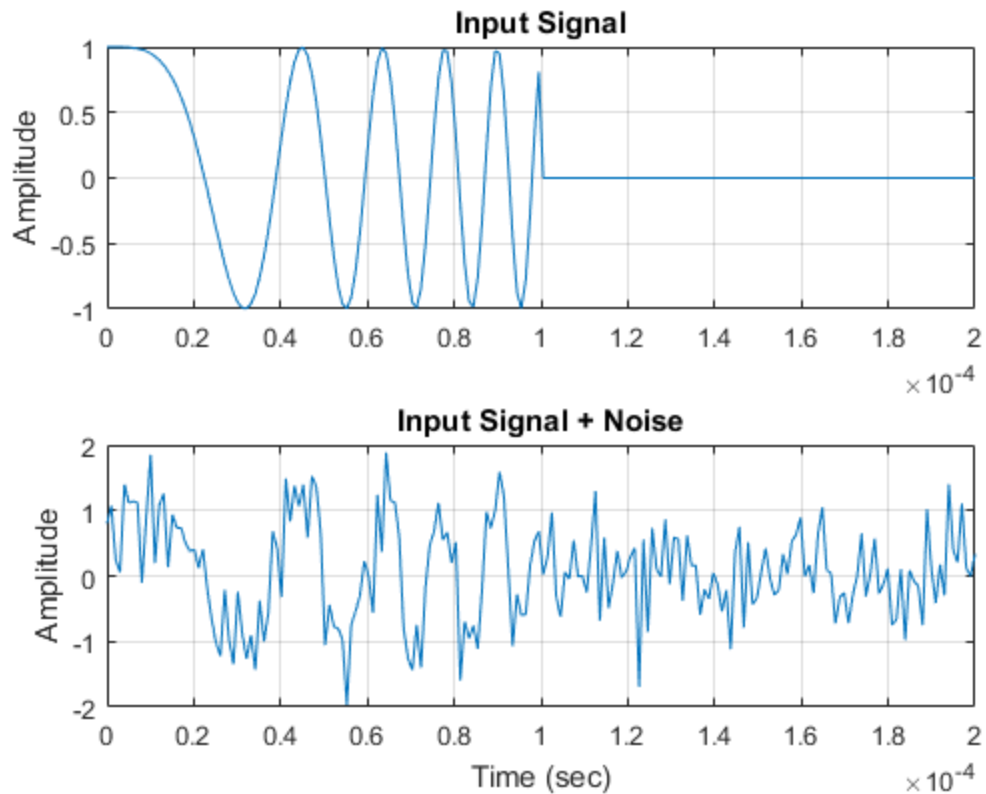

Plot the magnitudes of the two matched filter outputs.

```
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
```
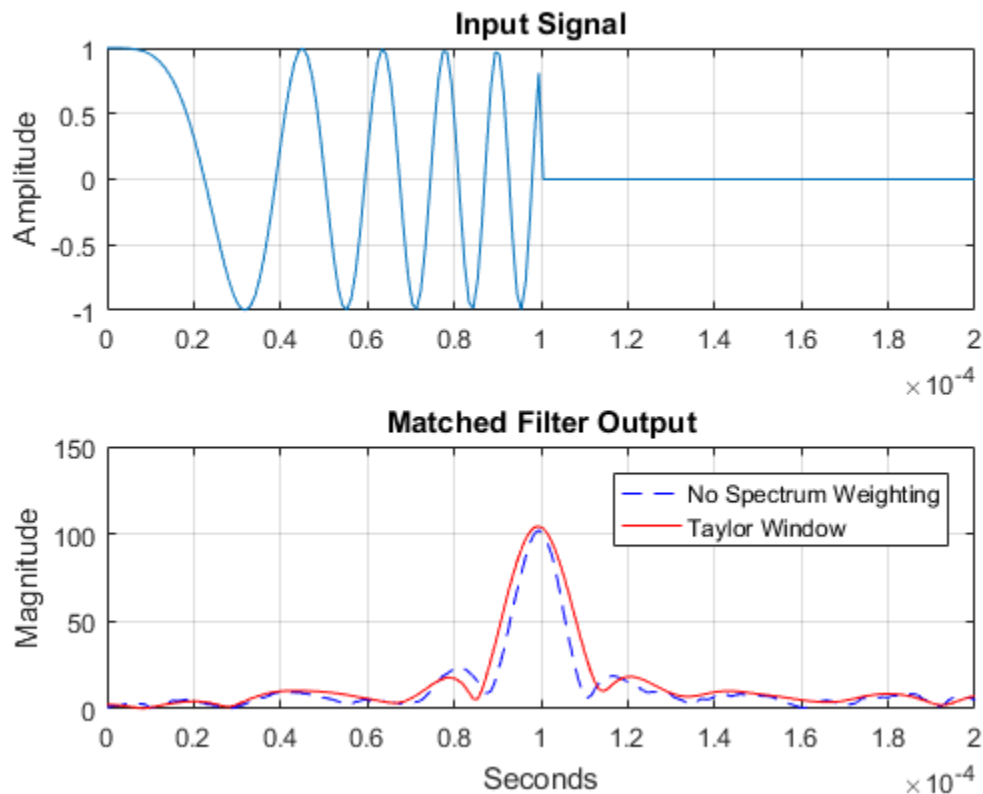

Matched Filtering to Improve SNR for Target Detection

This example shows how to improve the SNR by performing matched filtering.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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 `radareqpow` function 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');
Pt = radareqpow(c/fc,maxrange,SNR,...
    waveform.PulseWidth,'RCS',target.MeanRCS,'Gain',transmitter.Gain);
```

Set the peak transmit power to the output value from `radareqpow`.

```
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], [O 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], [O max(abs(mf_puls))],'r')
hold off
```
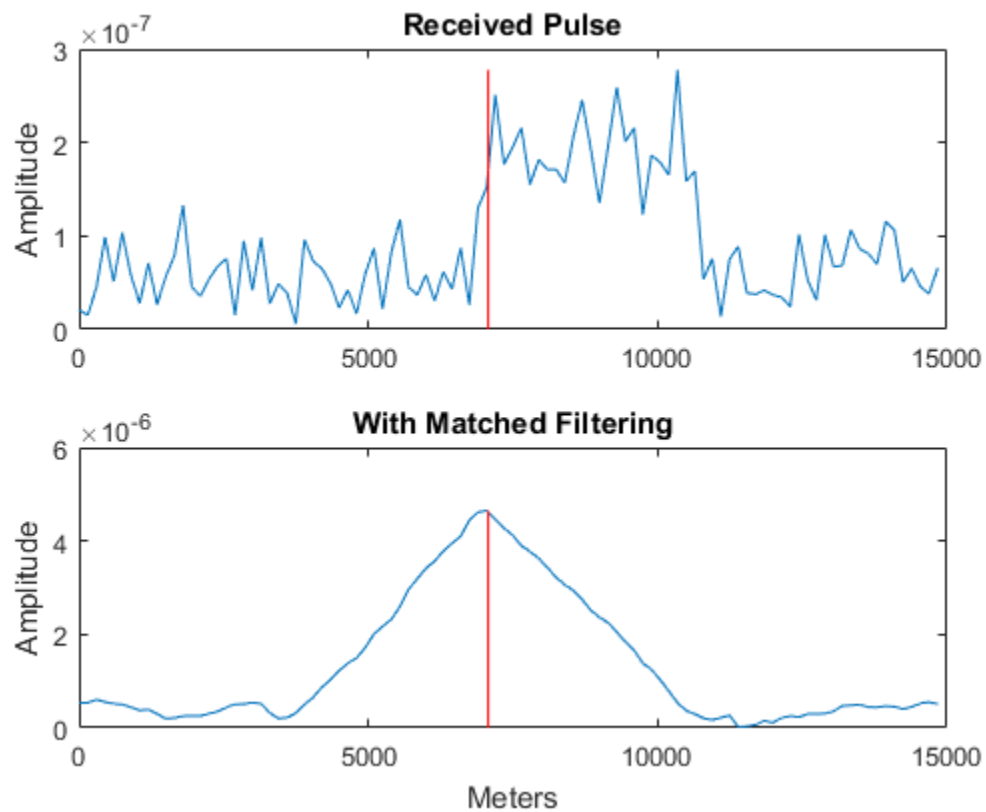
Stretch Processing

In this section...
"Reasons for Using Stretch Processing" on page 8-31
"Support for Stretch Processing" on page 8-31
"Stretch Processing Procedure" on page 8-31

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 | `findpeaks` | `periodogram` | `stretchfreq2rng`

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 | `beat2range` | `dechirp` | `findpeaks` | `periodogram` | `pwelch` | `range2beat` | `range2bw` | `range2time` | `rdcoupling` | `rootmusic` | `time2range`

Related Examples

- Automotive Adaptive Cruise Control Using FMCW Technology

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 8-36 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 8-37.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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 `radareqpow` function 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');
Pt = radareqpow(c/fc,maxrange,SNR,...
    waveform.PulseWidth,'RCS',target.MeanRCS,'Gain',transmitter.Gain);
```

Set the peak transmit power to the output value from `radareqpow`.

```
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))
```
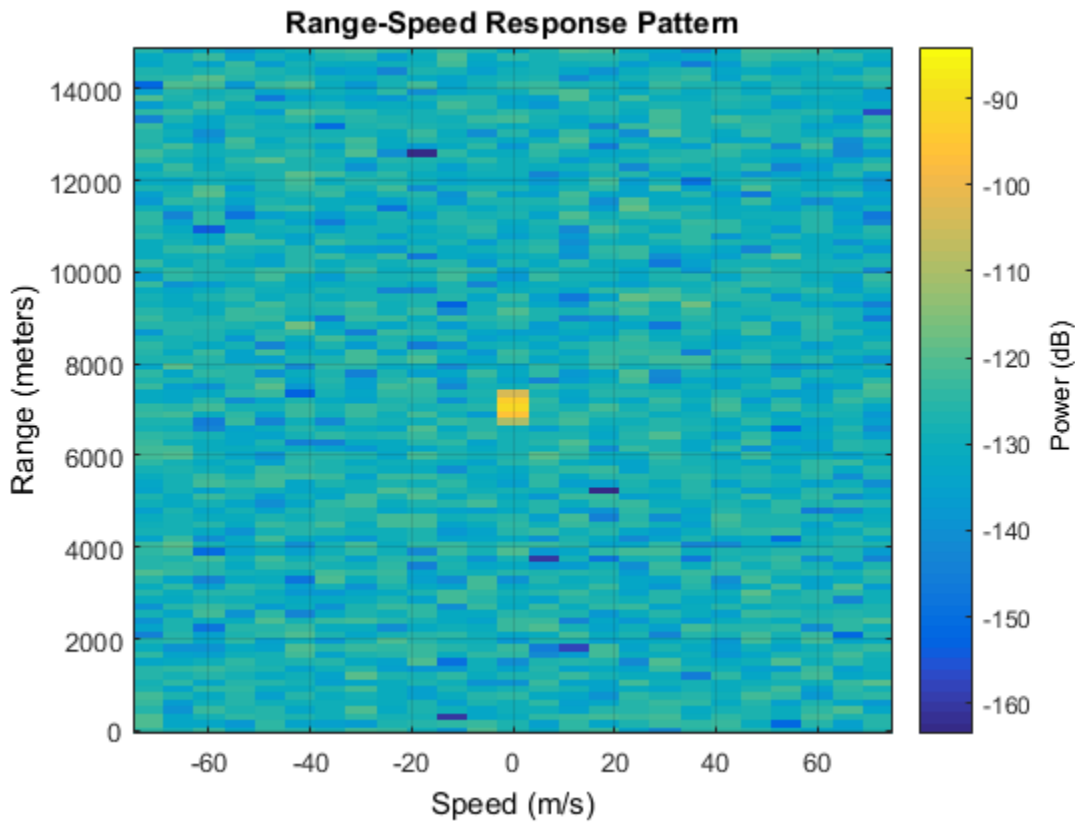

The plot shows the stationary target at a range of approximately 7000 m.

See Also
phased.RangeDopplerResponse

Related Examples
· Automotive Adaptive Cruise Control Using FMCW Technology

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)')
```

Increase in P_{FA} due to Noise Variance
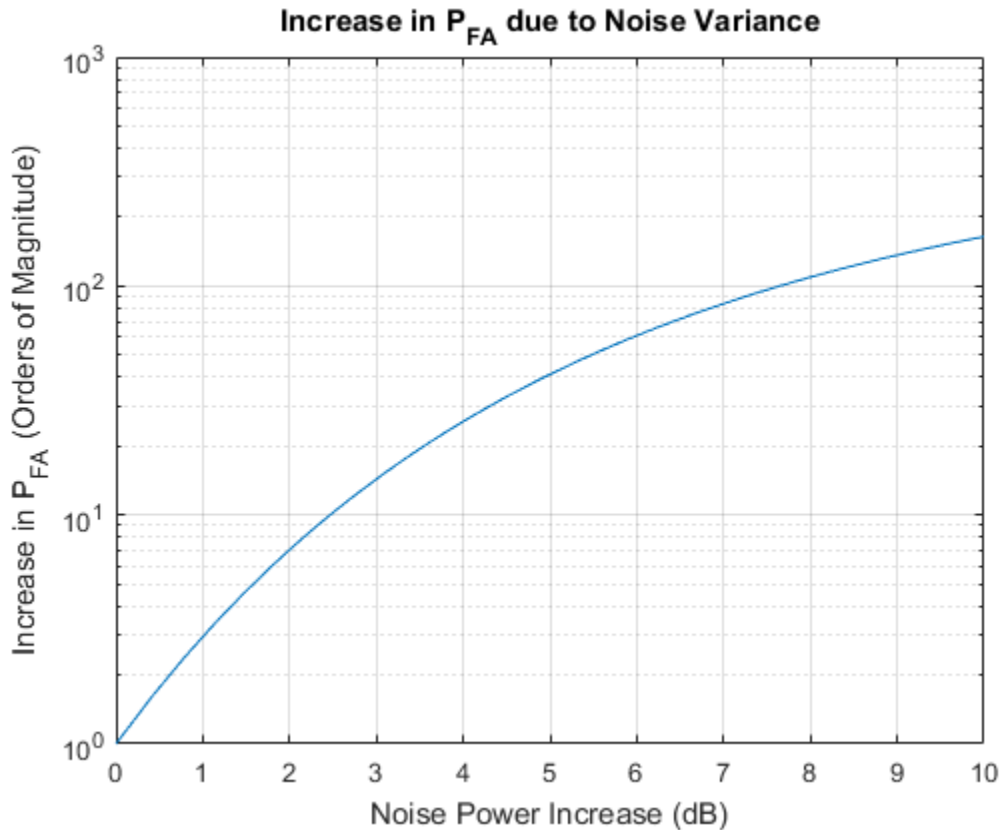
Cell-Averaging CFAR Detector

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.
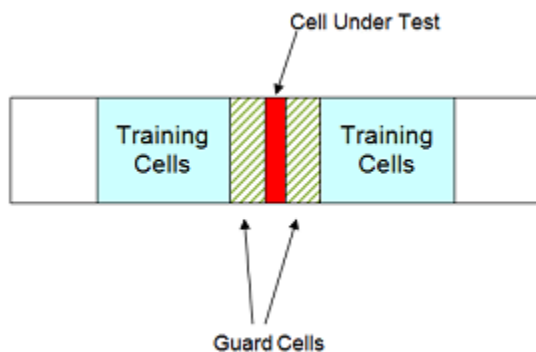
The optimum estimator for the noise variance depends on distributional assumptions and the type of detector. Assume the following:

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 σ^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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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_2 = sum(Z_detect(end,:))/Ntrials
Pfa = max(sum(Z_detect(2:end-1,:),2)/Ntrials)
```

```
Pd_1 =

     0


Pd_2 =

     1


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_2 = sum(Z_detect(end,:))/Ntrials
Pfa = max(sum(Z_detect(2:end-1,:),2)/Ntrials)
```

```
Pd_1 =

    0.5820


Pd_2 =

     1


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 8-51, 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.
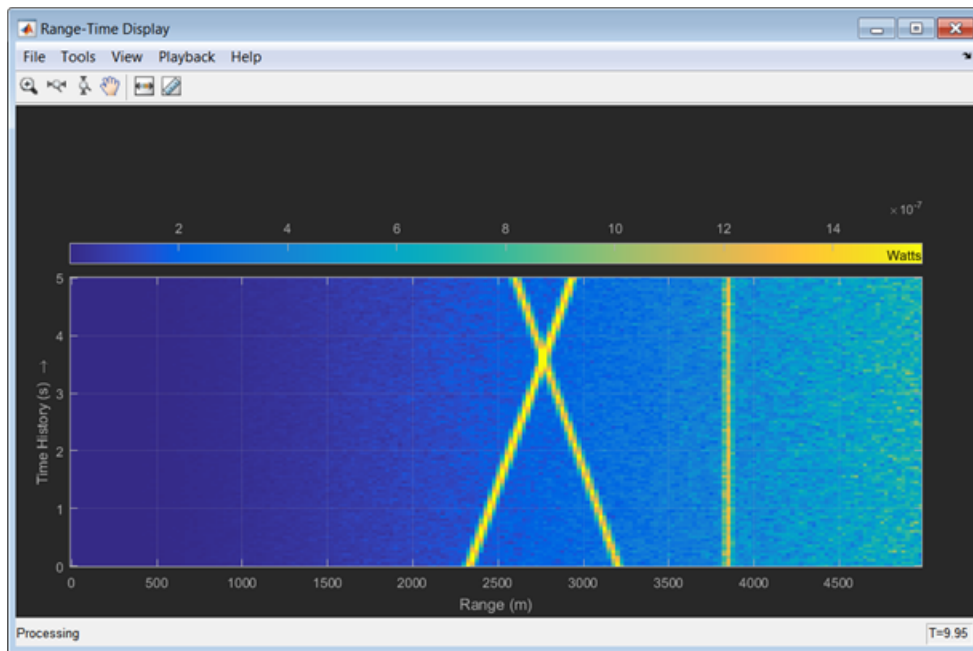
To examine the data, click the Cursor Measurement button in the Toolbar. You see two cursors, each of which is represented by pairs of cross-hairs. To distinguish cursors, one pair consists of solid lines and the second pair consists of dashed lines and are tagged with a **1** or a **2**.
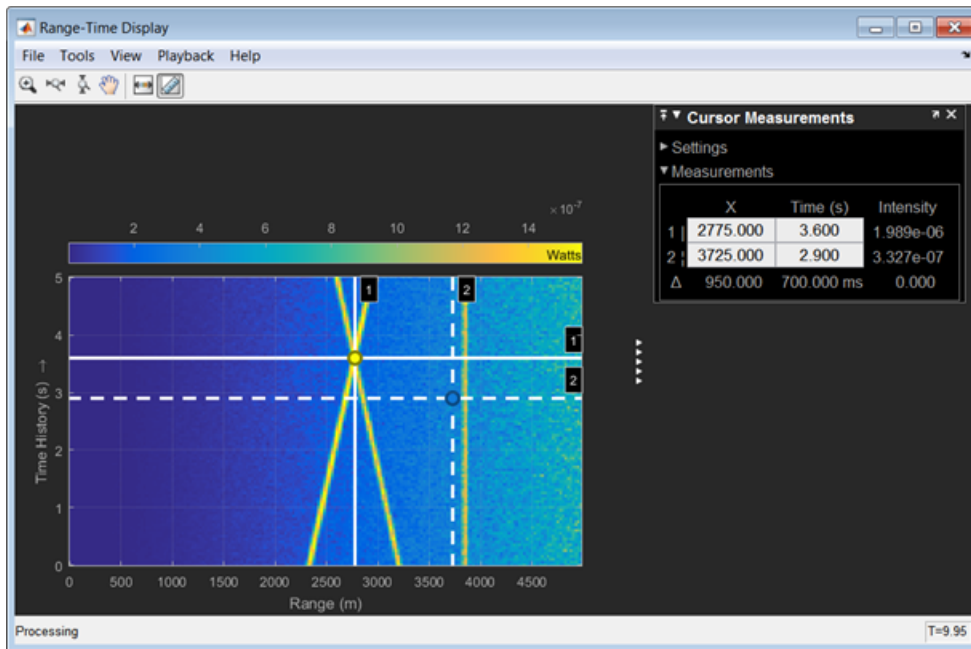
Cursor 1 has solid cross-hairs and overlays the intersection of two signal lines. Cursor 2 has dashed cross-hairs and overlays a signal-free region. The **Cursor Measurements** pane shows the coordinates of the cursors in time and range (labelled **X**) and the intensities at these positions. *Cursor 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 complete radar system simulation. The radar scenario contains a stationary single-element monostatic radar and three moving targets.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

Set Radar Operating Parameters

Set the probability of detection, probability of false alarm, maximum range, range
resolution, operating frequency, transmitter gain, and target radar cross-section.

```
pd = 0.9;
pfa = 1e-6;
max_range = 5000;
range_res = 50;
fc = 10e9;
tx_gain = 20;
tgt_rcs = 1;
```

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 bandwith 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);
```

Compute Transmit Power

Use the Albersheim equation to compute the SNR required to meet the desired
probability of detection and probability of false alarm. Then, use the radar equation to
compute the power needed to achieve the required SNR.

```
snr_min = albersheim(pd, pfa, num_pulse_int);
peak_power = radareqpow(lambda,max_range,snr_min,pulse_length,...
    'RCS',tgt_rcs,'Gain',tx_gain);
```

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 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','dB');
rtiscope = phased.IntensityScope('Name','Range-Time Display',...
```

```
'XLabel','Range (m)', ...
'XResolution',c/(2*fs), ...
'TimeResolution',0.05,'TimeSpan',5,'IntensityUnits','dB');
```

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 Doppler shifts of 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
```
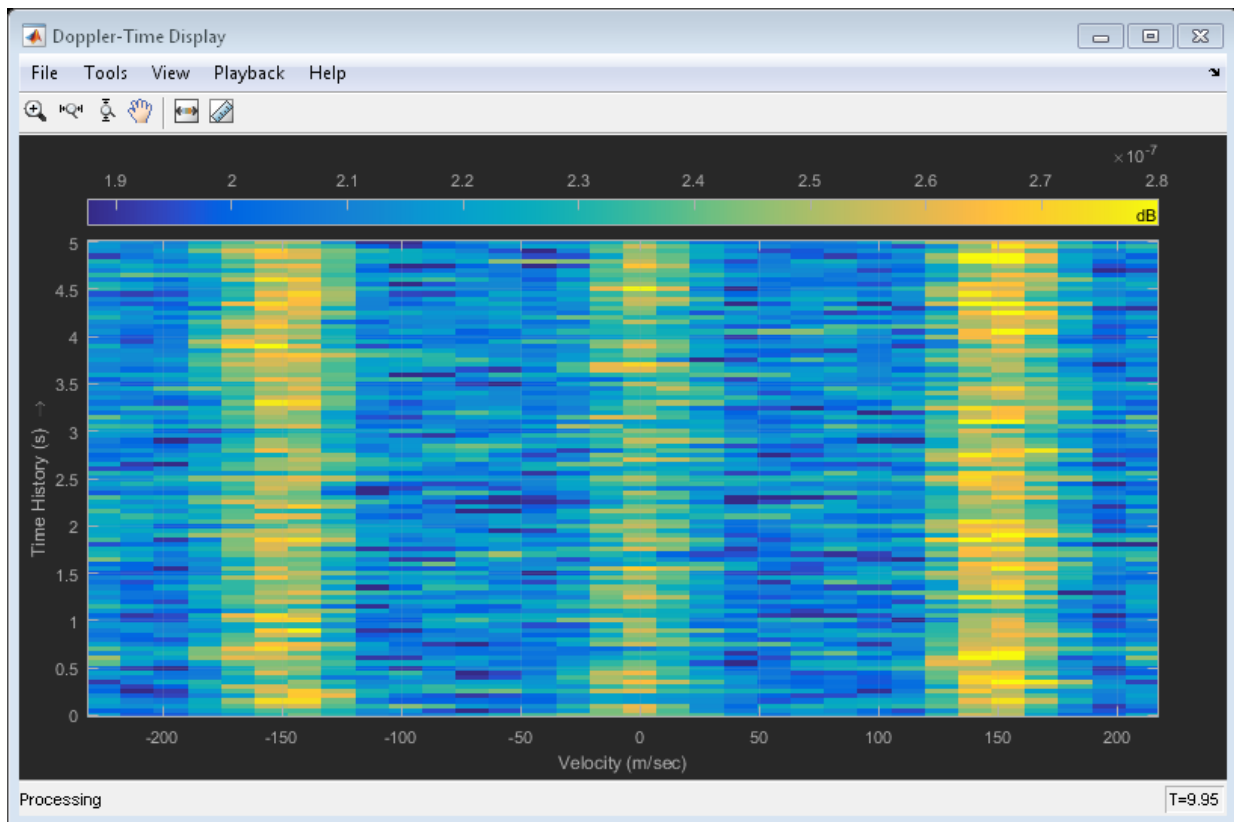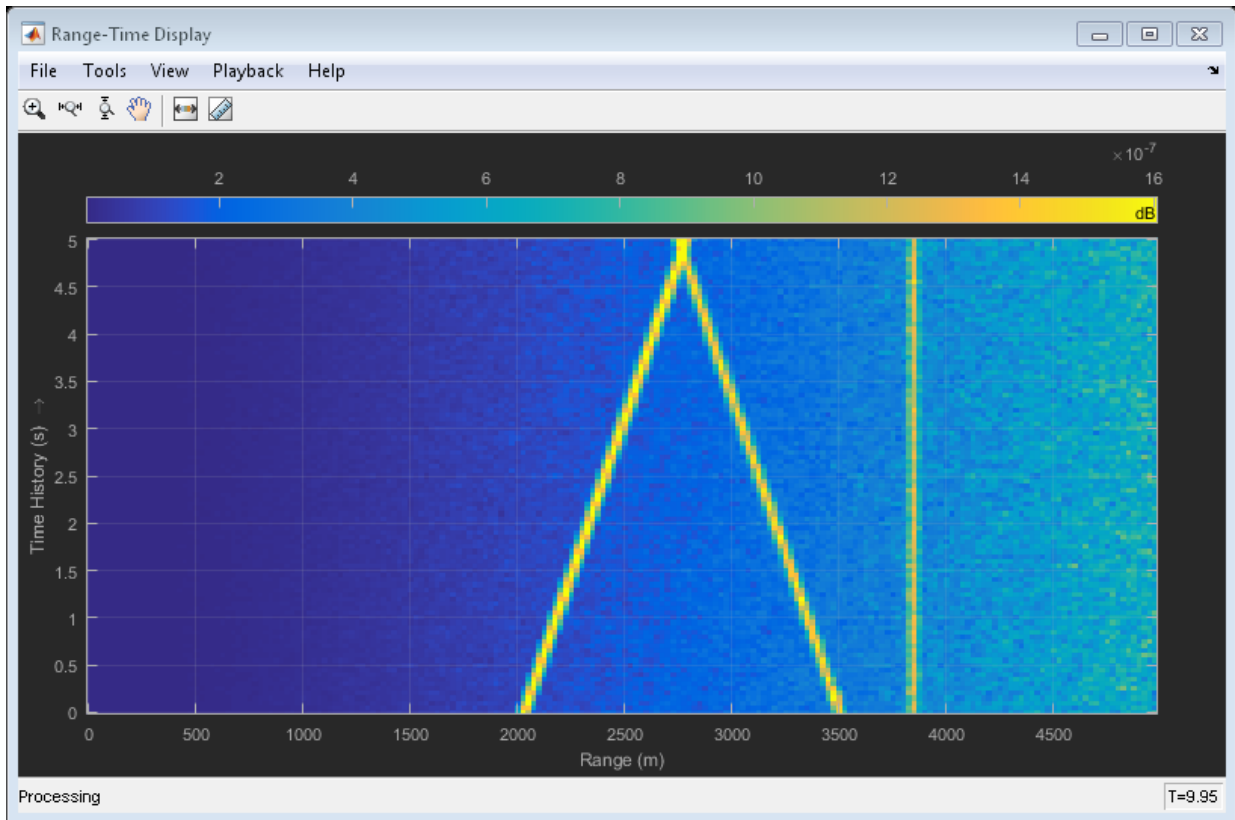
All of the targets lie on the x-axis. Two targets are moving along the x-axis and one is stationary. Because the radar is at the origin, you can read the target speed directly from the Doppler-Time Display window. The values agree with the specified velocities of -150, 150, and 0 m/sec.

9

Environment and Target Models

Free Space Path Loss

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 `step` on 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]);
```

Multiply the range by two for two-way propagation.

```
tgtrng = 2*tgtrng;
```

Determine the wavelength for 1 GHz.

```
lambda = physconst('LightSpeed')/1e9;
```

Solve for the loss using `fspl`.

```
L = fspl(tgtrng,lambda)
```

```
L =

  104.7524
```

The free space path loss in decibels is approximately 105 dB.

Alternatively, you can compute the loss directly from

```
Loss = pow2db((4*pi*tgtrng/lambda)^2)
```

```
Loss =

  104.7524
```

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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')
```
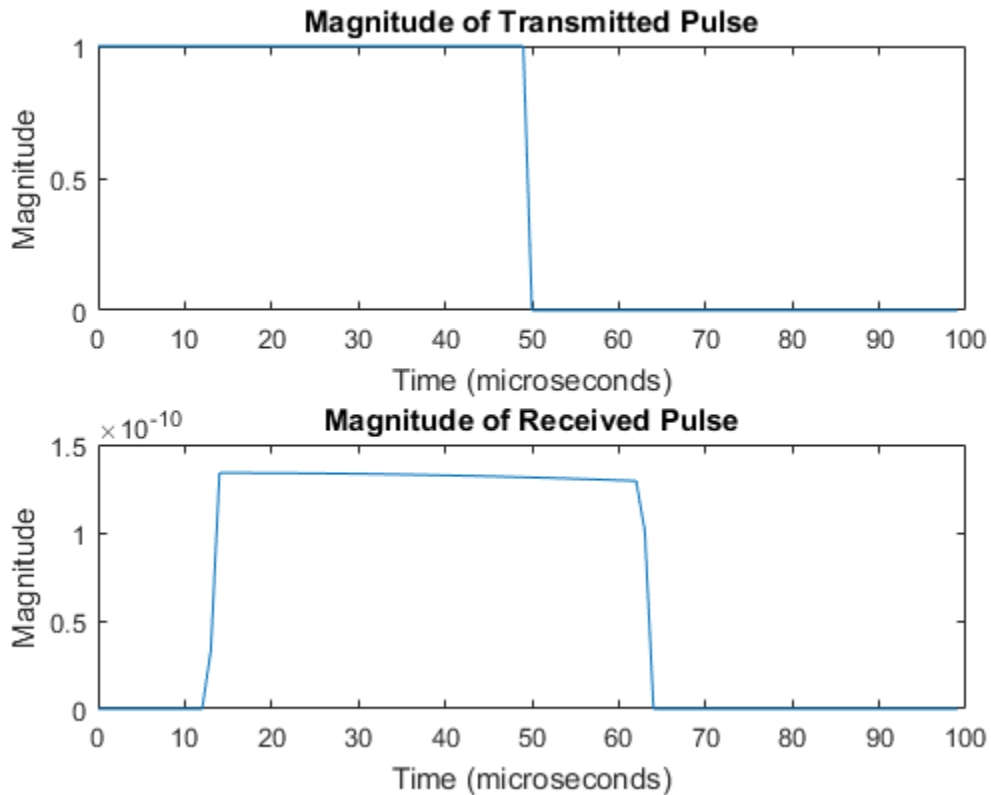
The delay in the received pulse is approximately 14 µs, the expected value for a distance of 4.123 km.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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
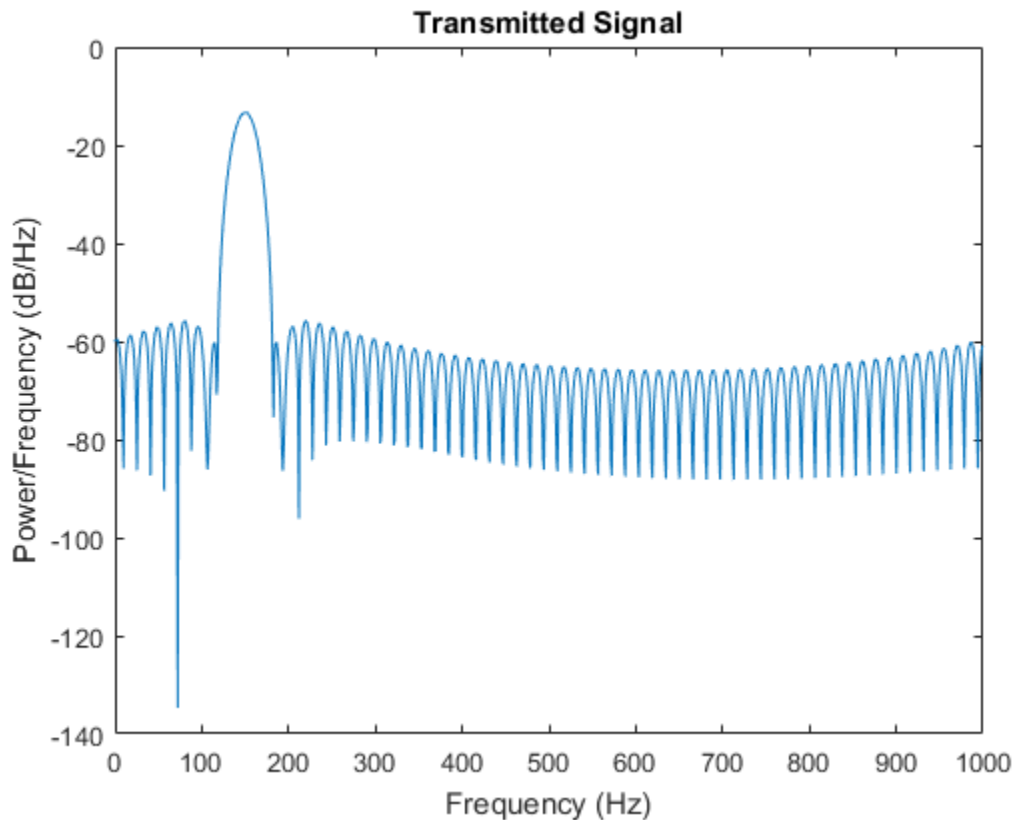
Plot the spectrum of the propagated signal. The peak at 250 Hz reflects the frequency of the signal plus the Doppler shift of 100 Hz.

```
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')
```
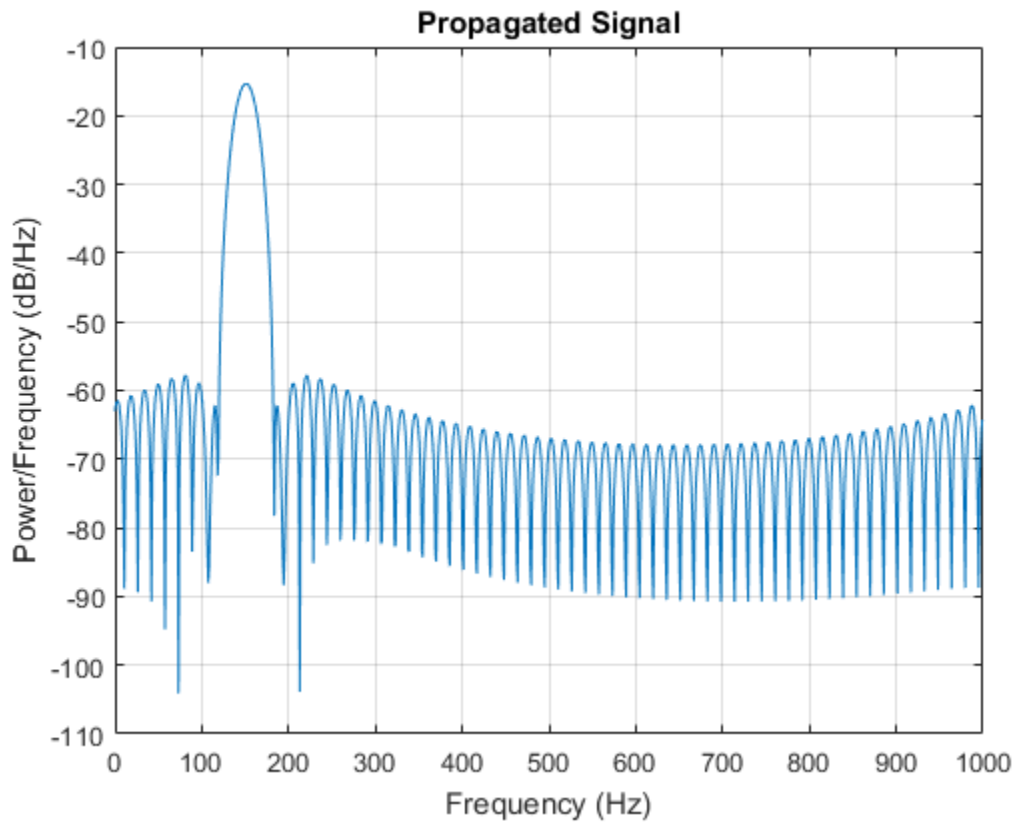
The doppler shift is too small to see. Overlay the two spectra in the region of 150 Hz.

```
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')
```
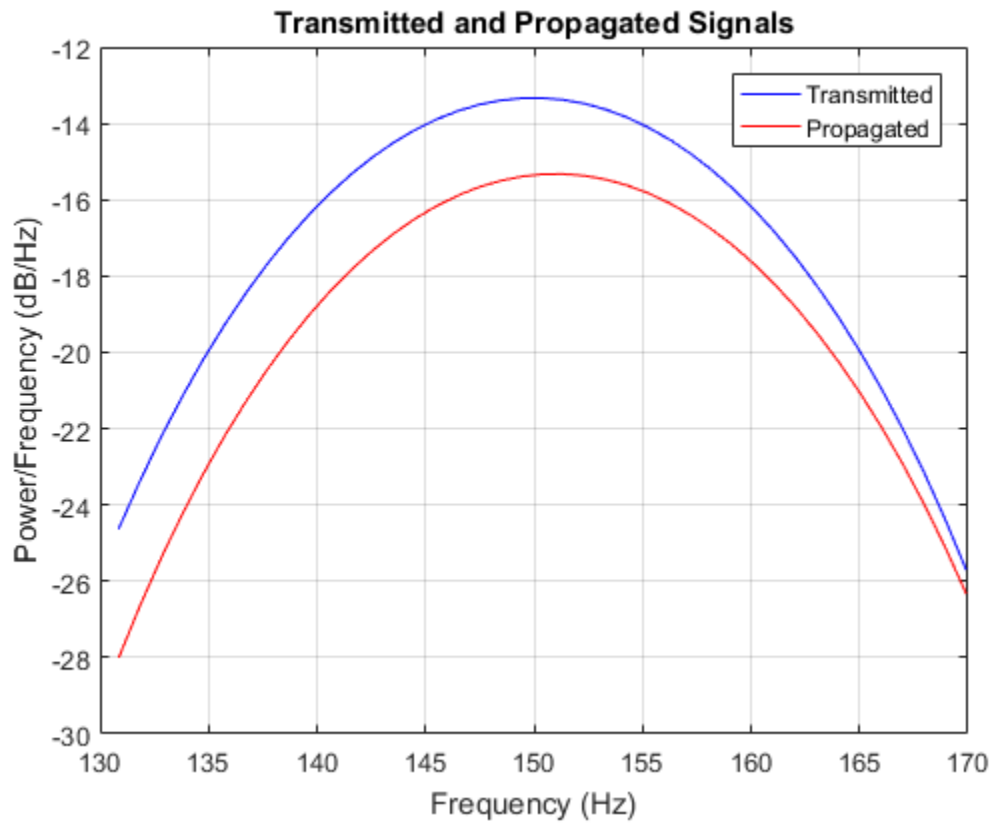
The peak shift appears to be approximately 1 Hz.

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 phased.TwoRayChannel and phased.WidebandTwoRayChannel System objects model propagation time delay, phase shift, Doppler shift, and loss effects for both paths. For the reflected path, loss effects include reflection loss at the boundary.

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, θ'_{los} and θ'_{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, θ_{los} and θ_{rp}. In the global coordinates, the angle of reflection at the boundary is the same as the angles θ_{rp} and θ'_{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.
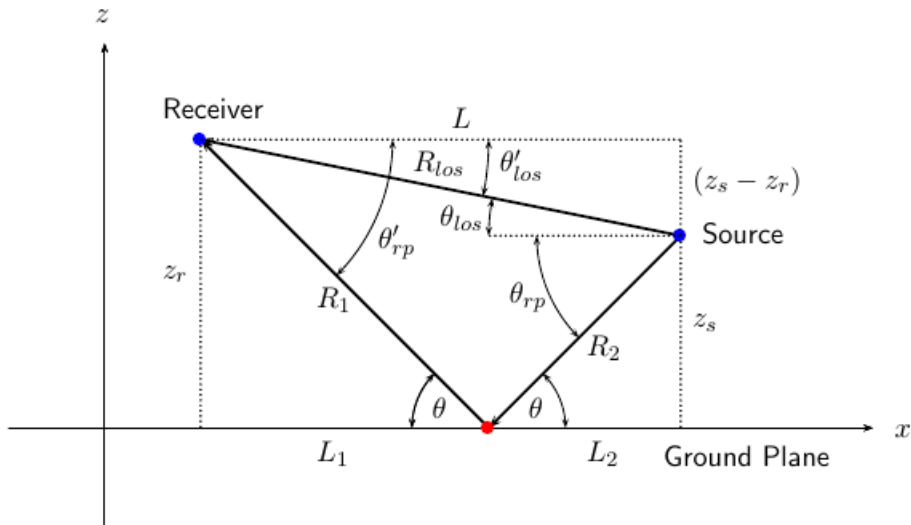
You can easily derive exact formulas for path lengths and angles in terms of the ground range and objects heights in the global coordinate system.

$$\vec{R} = \vec{x}_s - \vec{x}_r$$

$$R_{los} = \left| \vec{R} \right| = \sqrt{\left(z_r - z_s \right)^2 + L^2}$$

$$R_1 = \frac{z_r}{z_r + z_z} \sqrt{\left(z_r + z_s \right)^2 + L^2}$$

$$R_2 = \frac{z_s}{z_s + z_r} \sqrt{\left(z_r + z_s \right)^2 + L^2}$$

$$R_{rp} = R_1 + R_2 = \sqrt{\left(z_r + z_s \right)^2 + L^2}$$

$$\tan \theta_{los} = \frac{\left(z_s - z_r \right)}{L}$$

$$\tan \theta_{rp} = -\frac{\left(z_s + z_r \right)}{L}$$

$$\theta'_{los} = -\theta_{los}$$

$$\theta'_{rp} = \theta_{rp}$$

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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');
```
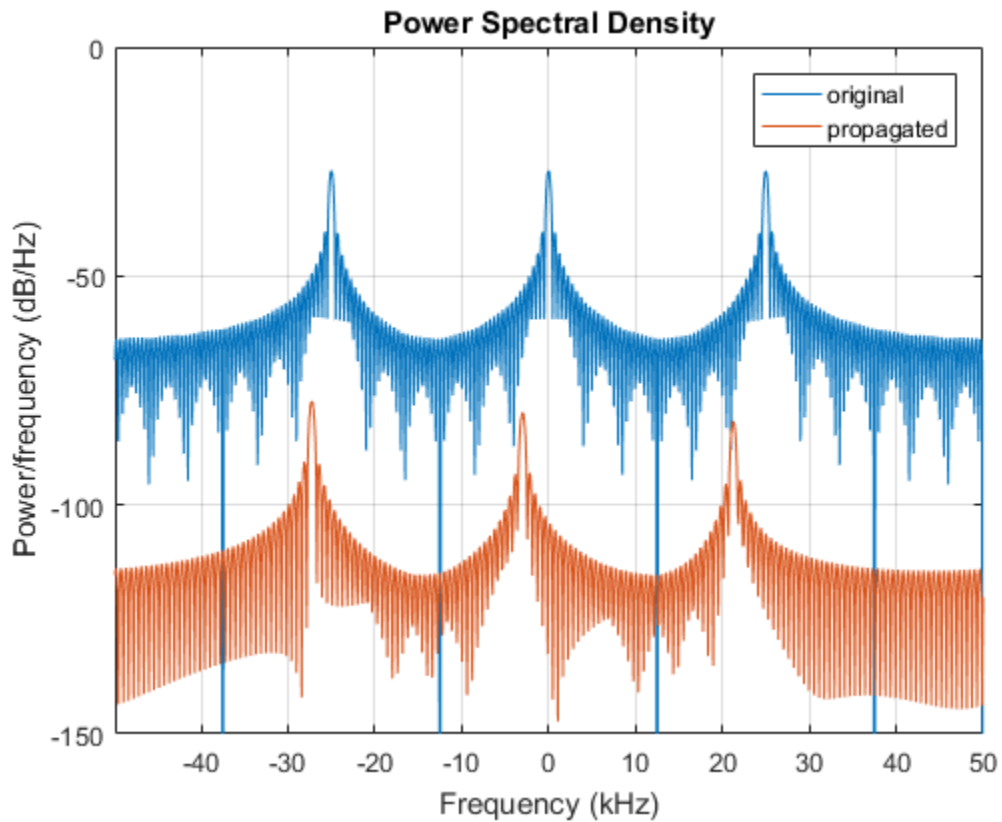
For this wideband signal, you can see that the magnitude of the Doppler shift increases with frequency. In contrast, for narrowband signals, the Doppler shift is assumed constant over the band.

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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, replace myObject(x) with step(myObject,x).

```
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 =

   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. Use the step method and set the `UPDATERCS` input argument 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°.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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 of a rotating monostatic radar and a target having 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

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 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 properly employing the `step` method of 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 `step` method at the beginning of the scan, set the `updatercs` argument to `true`. Set `updatercs` to `false` for the remaining calls to `step` during the scan. This 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'`.

```
rng default
tgtmodel = 'Swerling1';
```

Set up radar model System object™ components

Set up the radiating antenna. Assume the operating frequency of the antenna is 1GHz.

```
fc = 1e9;
sAnt = phased.IsotropicAntennaElement('BackBaffled',true);
SRad = phased.Radiator('OperatingFrequency',fc, ...
    'Sensor',sAnt);
```

Specify the location of the stationary antenna.

```
sRadar = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
sTarget = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the `step` method.

```
sWav = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifer.

```
sTransmit = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
sFS = 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.

```
sTgt = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
sColl = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',sAnt);
```

Define a matched filter to process the incoming signal.

```
waveform = step(sWav);
sMF = phased.MatchedFilter(...
    'Coefficients',getMatchedFilter(sWav));
```

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 radar rarget
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;
            TXwaveform = step(sTransmit,waveform);
```

Find the radar and target positions

```
            [xradar,vradar] = step(sRadar,t);
            [xtgt,vtgt] = step(sTarget,t);
```

Radiate waveform to target

```
            [~,ang] = rangeangle(xtgt,xradar);
            WFrad = step(SRad,TXwaveform,ang);
```

Propagate waveform to and from the target

```
            WFprop = step(sFS,WFrad,sRadar.InitialPosition,...
                sTarget.InitialPosition,[0;0;0],[0;0;0]);
```

Reflect waveform from target. Set the updatercs flag.

```
            WFreflect = step(sTgt,WFprop,updatercs);
```

Collect the received waveform

```
            WFcol = step(sColl,WFreflect,ang);
```

Apply matched filter to incoming signal

```
            y = step(sMF,WFcol);
            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 properly employing the `step` method of 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 the every call to the `step` method, set the `updatercs` argument to `true`. This means that the radar cross-section is updated at every pulse.

Set the target model to `'Swerling1'`.

```
rng default
tgtmodel = 'Swerling1';
```

Set up radar model System object™ components

Set up the radiating antenna. Assume the operating frequency of the antenna is 1GHz.

```
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 `step` method.

```
waveform = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifer.

```
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 rarget

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;
        TXwaveform = transmitter(wav);
```

Find the radar and target positions

```
        [xradar,vradar] = radarplatform(t);
        [xtgt,vtgt] = targetplatform(t);
```

Radiate waveform to target

```
        [~,ang] = rangeangle(xtgt,xradar);
        WFrad = radiator(TXwaveform,ang);
```

Propagate waveform to and from the target

```
        WFprop = channel(WFrad,radarplatform.InitialPosition,...
            targetplatform.InitialPosition,[0;0;0],[0;0;0]);
```

Reflect waveform from target. Set the updatercs flag.

```
        WFreflect = target(WFprop,updatercs);
```

Collect the received waveform

```
        WFcol = collector(WFreflect,ang);
```

Apply matched filter to incoming signal

```
        y = filter(WFcol);
        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')
```
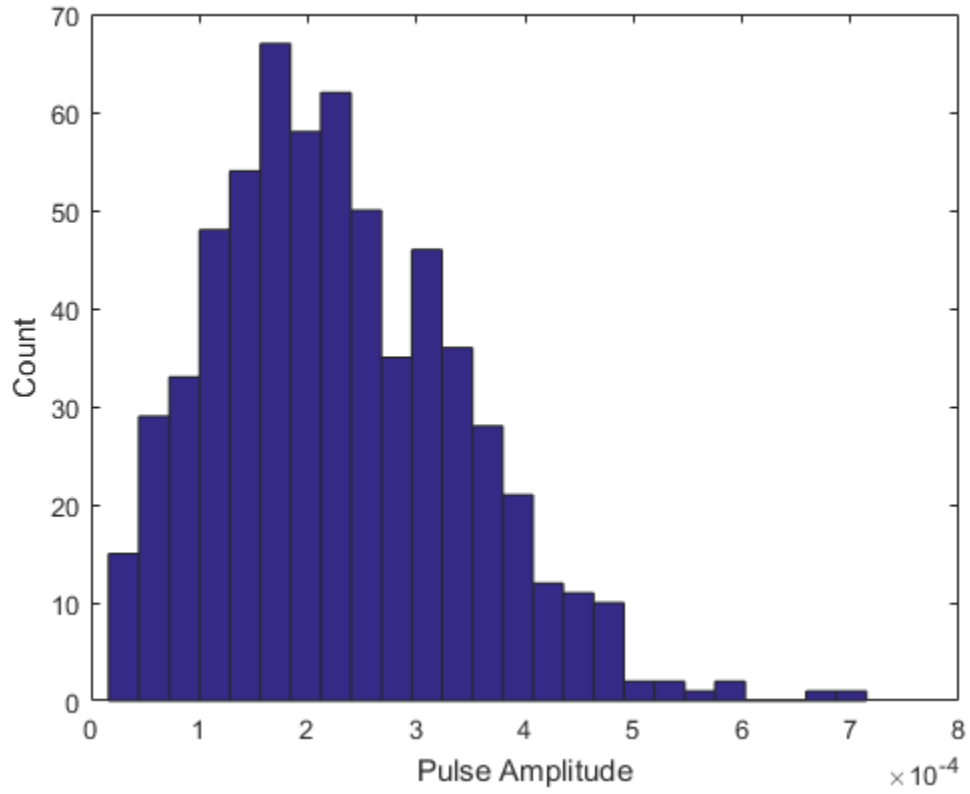
Swerling 3 Target Models

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 properly employing the `step` method of 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 `step` method at the beginning of the scan, set the `updatercs` argument to `true`. Set `updatercs` to `false` for the remaining calls to `step` during the scan. This 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'` .

```
rng default
tgtmodel = 'Swerling3';
```

Set up radar model System object™ components

Set up the radiating antenna. Assume the operating frequency of the antenna is 1GHz.

```
fc = 1e9;
sAnt = phased.IsotropicAntennaElement('BackBaffled',true);
SRad = phased.Radiator('OperatingFrequency',fc, ...
    'Sensor',sAnt);
```

Specify the location of the stationary antenna.

```
sRadar = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
sTarget = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the `step` method.

```
sWav = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifer.

```
sTransmit = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
sFS = 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.

```
sTgt = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
sColl = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',sAnt);
```

Define a matched filter to process the incoming signal.

```
waveform = step(sWav);
sMF = phased.MatchedFilter(...
    'Coefficients',getMatchedFilter(sWav));
```

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 rarget
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;
            TXwaveform = step(sTransmit,waveform);
```

Find the radar and target positions

```
            [xradar,vradar] = step(sRadar,t);
            [xtgt,vtgt] = step(sTarget,t);
```

Radiate waveform to target

```
            [~,ang] = rangeangle(xtgt,xradar);
            WFrad = step(SRad,TXwaveform,ang);
```

Propagate waveform to and from the target

```
            WFprop = step(sFS,WFrad,sRadar.InitialPosition,...
                sTarget.InitialPosition,[0;0;0],[0;0;0]);
```

Reflect waveform from target. Set the updatercs flag.

```
            WFreflect = step(sTgt,WFprop,updatercs);
```

Collect the received waveform

```
            WFcol = step(sColl,WFreflect,ang);
```

Apply matched filter to incoming signal
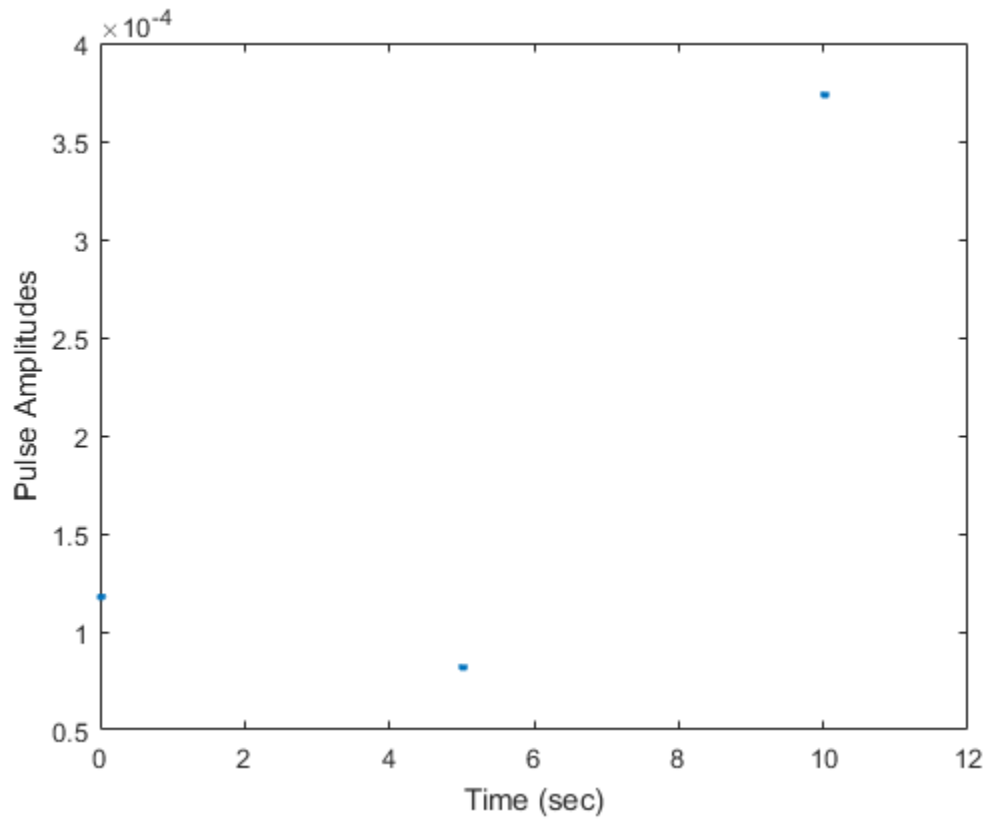
```
            y = step(sMF,WFcol);
            z(m,k) = max(abs(y));
            tp(m,k) = t;

    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 properly employing the step method of 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 the every call to the step method, set the updatercs argument to true. This means that the radar cross section is updated for every pulse in the scan.

Set the target model to 'Swerling4' .

```
rng default
tgtmodel = 'Swerling4';
```

Set up radar model System object™ components

Set up the radiating antenna. Assume the operating frequency of the antenna is 1GHz.

```
fc = 1e9;
sAnt = phased.IsotropicAntennaElement('BackBaffled',true);
SRad = phased.Radiator('OperatingFrequency',fc, ...
    'Sensor',sAnt);
```

Specify the location of the stationary antenna.

```
sRadar = phased.Platform('InitialPosition',[0;0;0]);
```

Specify the location of a stationary target.

```
sTarget = phased.Platform('InitialPosition',[2000; 0; 0]);
```

The transmitted signal is a linear FM waveform. Transmit one pulse per call to the step method.

```
sWav = phased.LinearFMWaveform('PulseWidth',50e-6,...
    'OutputFormat','Pulses','NumPulses',1);
```

Set up the transmitting amplifer.

```
sTransmit = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Set up the propagation environment to be free space.

```
sFS = 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.

```
sTgt = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',fc,...
    'Model',tgtmodel);
```

Set up the radar collector.

```
sColl = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',sAnt);
```

Define a matched filter to process the incoming signal.

```
waveform = step(sWav);
sMF = phased.MatchedFilter(...
    'Coefficients',getMatchedFilter(sWav));
```

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 rarget

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;
```

```matlab
for k = 1:Np

    t = t + pri;
    TXwaveform = step(sTransmit,waveform);
```

Find the radar and target positions

```matlab
    [xradar,vradar] = step(sRadar,t);
    [xtgt,vtgt] = step(sTarget,t);
```

Radiate waveform to target

```matlab
    [~,ang] = rangeangle(xtgt,xradar);
    WFrad = step(SRad,TXwaveform,ang);
```

Propagate waveform to and from the target

```matlab
    WFprop = step(sFS,WFrad,sRadar.InitialPosition,...
        sTarget.InitialPosition,[0;0;0],[0;0;0]);
```

Reflect waveform from target. Set the `updatercs` flag.

```matlab
    WFreflect = step(sTgt,WFprop,updatercs);
```

Collect the received waveform

```matlab
    WFcol = step(sColl,WFreflect,ang);
```

Apply matched filter to incoming signal

```matlab
    y = step(sMF,WFcol);
    z(m,k) = max(abs(y));
    tp(m,k) = t;

    end
end
```
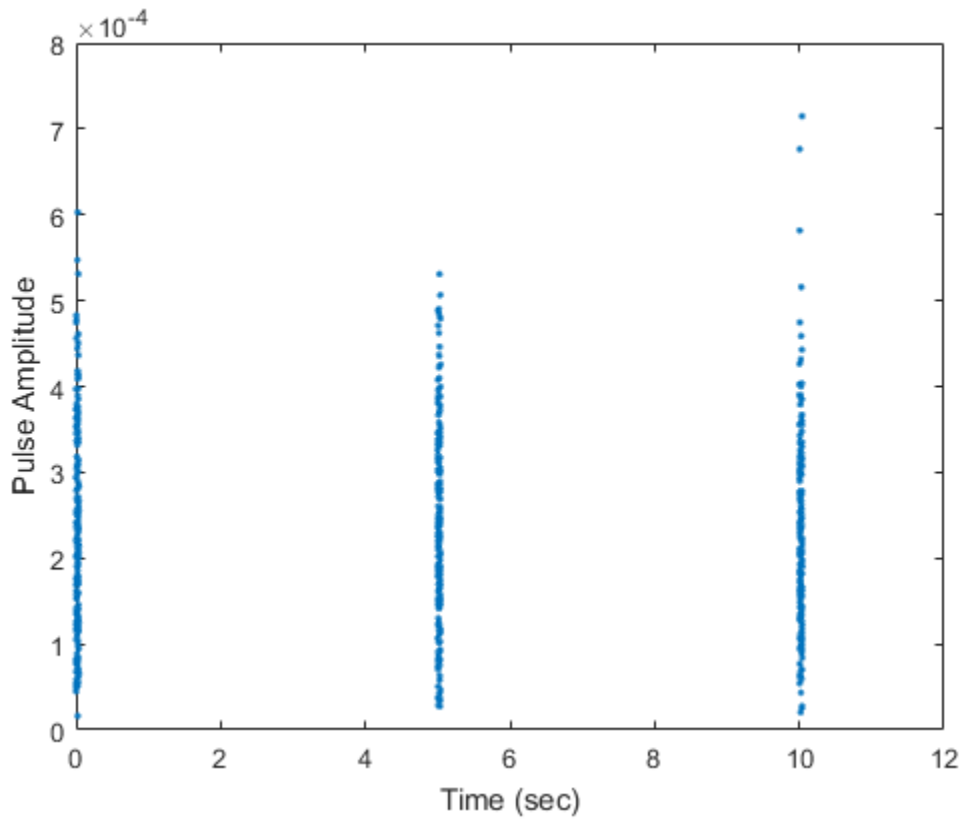
Plot the pulse amplitudes

Plot the amplitudes of the pulses for the scan as a function of time.

```matlab
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')
```
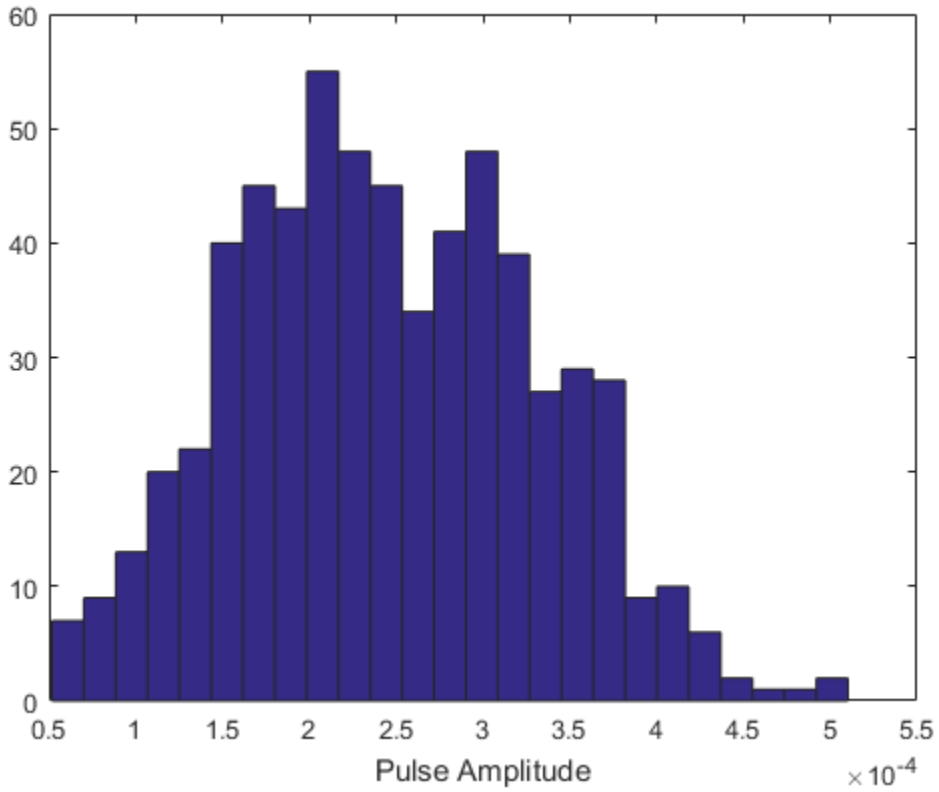
Clutter Modeling

In this section...

Surface Clutter Overview

Surface clutter refers to reflections of a radar signal from land, sea, or the land-sea interface. When trying to detect or track targets moving on or above the surface, you must be able to distinguish between clutter and the targets of interest. For example, a ground moving target indicator (GMTI) radar application should detect targets on the ground while accounting for radar reflections from trees or houses.

If you are simulating a radar system, you might want to incorporate surface clutter into the simulation to ensure the system can overcome the effects of surface clutter. If you are analyzing the statistical performance of a radar system, you might want to incorporate clutter return distributions into the analysis.

Approaches for Clutter Simulation or Analysis

Phased Array System Toolbox software offers these tools to help you incorporate surface clutter into your simulation or analysis:

- phased.ConstantGammaClutter, a System object that simulates clutter returns using the constant gamma model
- Utility functions to help you implement your own clutter models:

  - `billingsleyicm`
  - `depressionang`
  - `effearthradius`
  - `grazingang`
  - `horizonrange`
  - `surfclutterrcs`

• `surfacegamma`

Considerations for Setting Up a Constant Gamma Clutter Simulation

When you use `phased.ConstantGammaClutter`, you must configure the object for the situation you are simulating, and confirm that the assumptions the software makes are valid for your system.

Physical Configuration Properties

The `ConstantGammaClutter` object has properties that correspond to physical aspects of the situation you are modeling. These properties include:

- Propagation speed, sample rate, and pulse repetition frequency of the signal
- Operating frequency of the system
- Altitude, speed, and direction of the radar platform
- Depression angle of the broadside of the radar antenna array

Clutter-Related Properties

The object has properties that correspond to the clutter characteristics, location, and modeling fidelity. These properties include:

- Gamma parameter that depends on the terrain type and system's operating frequency.
- Azimuth coverage and maximum range for the clutter simulation.
- Azimuth span of each clutter patch. The software internally divides the clutter ring into a series of adjacent, nonoverlapping clutter patches.
- Clutter coherence time. This value indicates how frequently the software changes the set of random numbers in the clutter simulation.

  In the simulation, you can use identical random numbers over a time interval or uncorrelated random numbers. Simulation behavior slightly differs from reality, where a moving platform produces clutter returns that are correlated with each other over small time intervals.

Working with Samples or Pulses

The `ConstantGammaClutter` object has properties that let you obtain results in a convenient format. Using the `OutputFormat` property, you can choose to have the `step` method produce a signal that represents:

- A fixed number of pulses. You indicate the number of pulses using the `NumPulses` property of the object.
- A fixed number of samples. You indicate the number of samples using the `NumSamples` property of the object. Typically, you use the number of samples in one pulse. In staggered PRF applications, you might find this option more convenient because the `step` output always has the same matrix size.

Assumptions

The clutter simulation that `ConstantGammaClutter` provides is based on these assumptions:

- The radar system is monostatic.
- The propagation is in free space.
- The terrain is homogeneous.
- The clutter patch is stationary during the coherence time. *Coherence time* indicates how frequently the software changes the set of random numbers in the clutter simulation.
- The signal is narrowband. Thus, the spatial response can be approximated by a phase shift. Similarly, the Doppler shift can be approximated by a phase shift.
- The radar system maintains a constant height during simulation.
- The radar system maintains a constant speed during simulation.

Related Examples

- Ground Clutter Mitigation with Moving Target Indication (MTI) Radar
- Introduction to Space-Time Adaptive Processing
- "DPCA Pulse Canceller to Reject Clutter" on page 7-9
- "Adaptive DPCA Pulse Canceller" on page 7-15
- "Sample Matrix Inversion Beamformer" on page 7-22

Barrage Jammer

In this section...
"Support for Modeling Barrage Jammer" on page 9-46
"Model Barrage Jammer Output" on page 9-46
"Model Effect of Barrage Jammer on Target Echo" on page 9-49

Support for Modeling Barrage Jammer

The phased.BarrageJammer object models a broadband jammer. The output of `phased.BarrageJammer` is a complex white Gaussian noise sequence. The modifiable properties of the barrage jammer are:

- `ERP` — Effective radiated power in watts
- `SamplesPerFrameSource` — Source of number of samples per frame
- `SamplesPerFrame` — Number of samples per frame
- `SeedSource` — Source of seed for random number generator
- `Seed` — Seed for random number generator

The real and imaginary parts of the complex white Gaussian noise sequence each have variance equal to 1/2 the effective radiated power in watts. Denote the effective radiated power in watts by P. The barrage jammer output is:

$$w[n] = \sqrt{\frac{P}{2}}x[n] + j\sqrt{\frac{P}{2}}y[n]$$

In this equation, `x[n]` and `y[n]` are uncorrelated sequences of zero-mean Gaussian random variables with unit variance.

Model Barrage Jammer Output

This example examines the statistical properties of the barrage jammer output and how they relate to the effective radiated power *(ERP)*. Create a barrage jammer using

an effective radiated power of 5000 watts. Generate output at 500 samples per frame. Then call the `step` function once to generate a single frame of complex data. Using the `histogram` function, show the distribution of barrage jammer output values. The `BarrageJammer` System object uses a random number generator. In this example, the random number generator seed is fixed for illustrative purposes and can be removed.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
rng default
jammer = phased.BarrageJammer('ERP',5000,...
    'SamplesPerFrame',500);
y = jammer();
subplot(2,1,1)
histogram(real(y))
title('Histogram of Real Part')
subplot(2,1,2)
histogram(imag(y))
title('Histogram of Imaginary Part')
xlabel('Watts')
```

Histogram of Real Part

Histogram of Imaginary Part

Watts
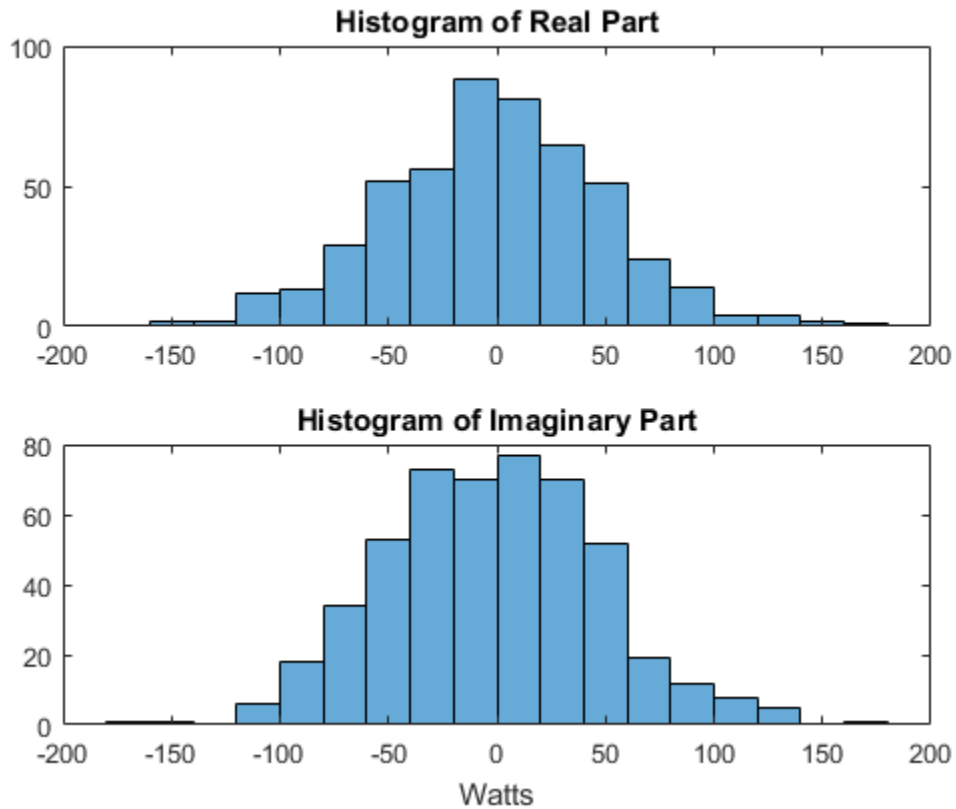
The mean values of the real and imaginary parts are

```
mean(real(y))
mean(imag(y))
```

```
ans =

    -1.0961
```

```
ans =

    -2.1671
```

which are effectively zero. The standard deviations of the real and imaginary parts are

```
std(real(y))
std(imag(y))


ans =

   50.1950


ans =

   49.7448
```

which agree with the predicted value of $\sqrt{ERP/2}$.

Model Effect of Barrage Jammer on Target Echo

This example demonstrates how to simulate the effect of a barrage jammer on a target echo. First, create the required objects. You need an array, a transmitter, a radiator, a target, a jammer, a collector, and a receiver. Additionally, you need to define two propagation paths: one from the array to the target and back, and the other path from the jammer to the array.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
antenna = phased.ULA(4);
Fs = 1e6;
fc = 1e9;
rng('default')
waveform = phased.RectangularWaveform('PulseWidth',100e-6,...
    'PRF',1e3,'NumPulses',5,'SampleRate',Fs);
transmitter = phased.Transmitter('PeakPower',1e4,'Gain',20,...
    'InUseOutputPort',true);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc);
jammer = phased.BarrageJammer('ERP',1000,...
    'SamplesPerFrame',waveform.NumPulses*waveform.SampleRate/waveform.PRF);
target = phased.RadarTarget('Model','Nonfluctuating',...
```

```
    'MeanRCS',1,'OperatingFrequency',fc);
targetchannel = phased.FreeSpace('TwoWayPropagation',true,...
    'SampleRate',Fs,'OperatingFrequency', fc);
jammerchannel = phased.FreeSpace('TwoWayPropagation',false,...
    'SampleRate',Fs,'OperatingFrequency', fc);
collector = phased.Collector('Sensor',antenna,...
    'OperatingFrequency',fc);
amplifier = phased.ReceiverPreamp('EnableInputPort',true);
```

Assume that the array, target, and jammer are stationary. The array is located at the global origin, *(0,0,0)*. The target is located at *(1000,500,0)*, and the jammer is located at *(2000,2000,100)*. Determine the directions from the array to the target and jammer.

```
targetloc = [1000 ; 500; 0];
jammerloc = [2000; 2000; 100];
[~,tgtang] = rangeangle(targetloc);
[~,jamang] = rangeangle(jammerloc);
```

Finally, transmit the rectangular pulse waveform to the target, reflect it off the target, and collect the echo at the array. Simultaneously, the jammer transmits a jamming signal toward the array. The jamming signal and echo are mixed at the receiver. Generate waveform

```
wav = waveform();
% Transmit waveform
[wav,txstatus] = transmitter(wav);
% Radiate pulse toward the target
wav = radiator(wav,tgtang);
% Propagate pulse toward the target
wav = targetchannel(wav,[0;0;0],targetloc,[0;0;0],[0;0;0]);
% Reflect it off the target
wav = target(wav);
% Collect the echo
wav = collector(wav,tgtang);
```

Generate the jamming signal

```
jamsig = jammer();
% Propagate the jamming signal to the array
jamsig = jammerchannel(jamsig,jammerloc,[0;0;0],[0;0;0],[0;0;0]);
% Collect the jamming signal
jamsig = collector(jamsig,jamang);

% Receive target echo alone and target echo + jamming signal
pulsewave = amplifier(wav,~txstatus);
```

```
pulsewave_jamsig = amplifier(wav + jamsig,~txstatus);
```

Plot the result, and compare it with received waveform with and without jamming.

```
subplot(2,1,1)
t = unigrid(0,1/Fs,size(pulsewave,1)*1/Fs,'[)');
plot(t*1000,abs(pulsewave(:,1)))
title('Magnitudes of Pulse Waveform Without Jamming--Element 1')
ylabel('Magnitude')
subplot(2,1,2)
plot(t*1000,abs(pulsewave_jamsig(:,1)))
title('Magnitudes of Pulse Waveform with Jamming--Element 1')
xlabel('millisec')
ylabel('Magnitude')
```

10
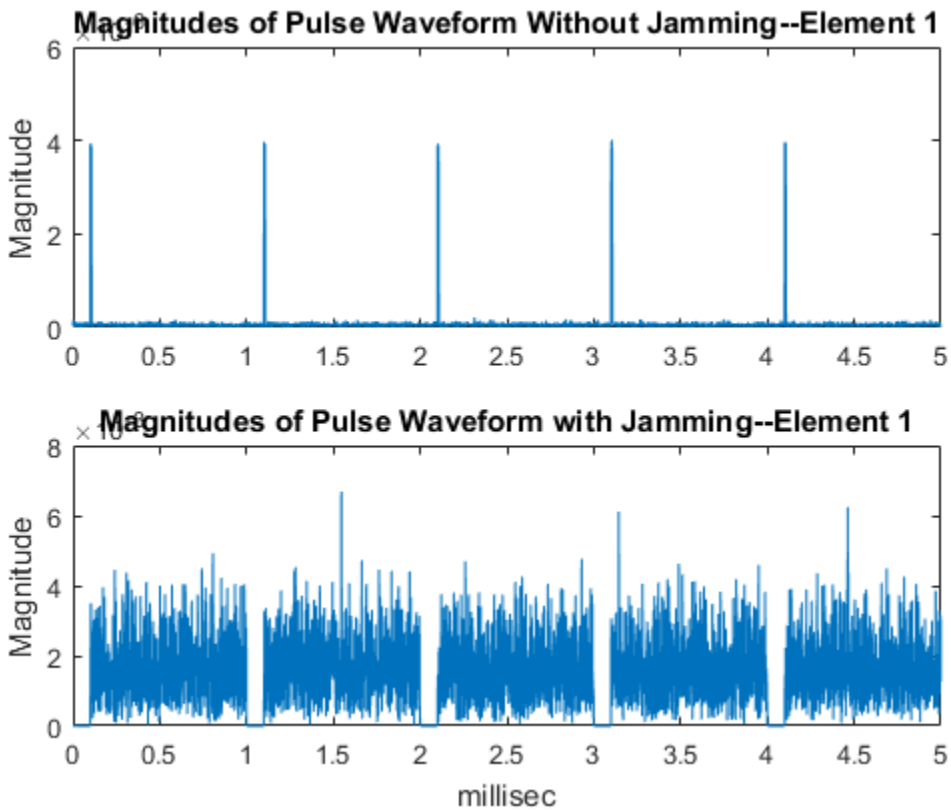
Coordinate Systems and Motion Modeling

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.
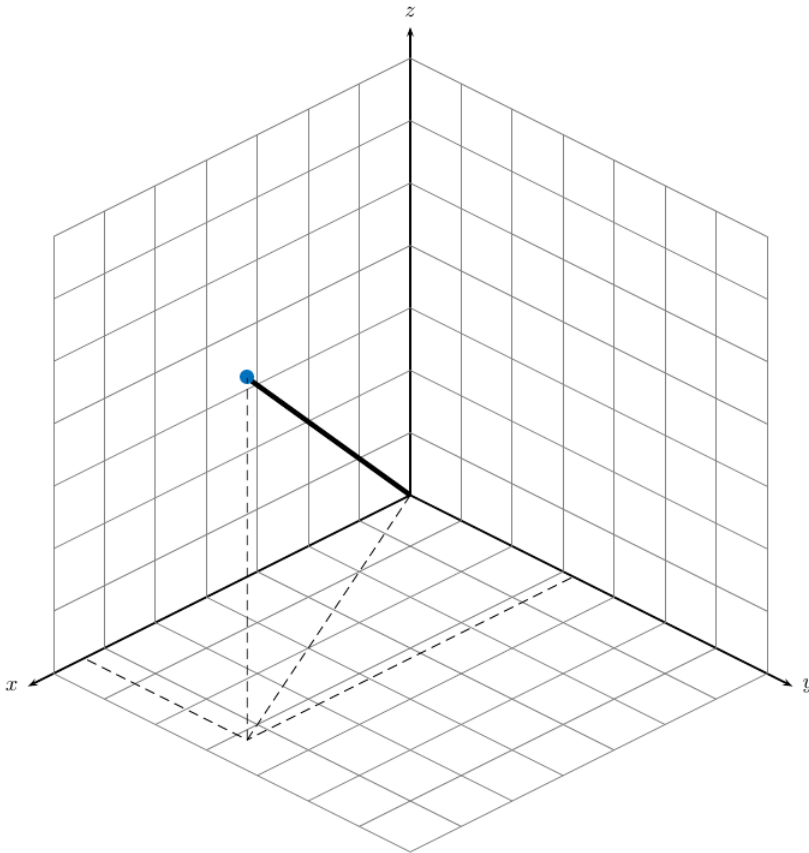
Rectangular coordinates specify a position in space in a given coordinate system as an ordered 3-tuple of real numbers, (x,y,z), with respect to the origin $(0,0,0)$. Considerations for choosing the origin are discussed in "Global and Local Coordinate Systems" on page 10-21.

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.
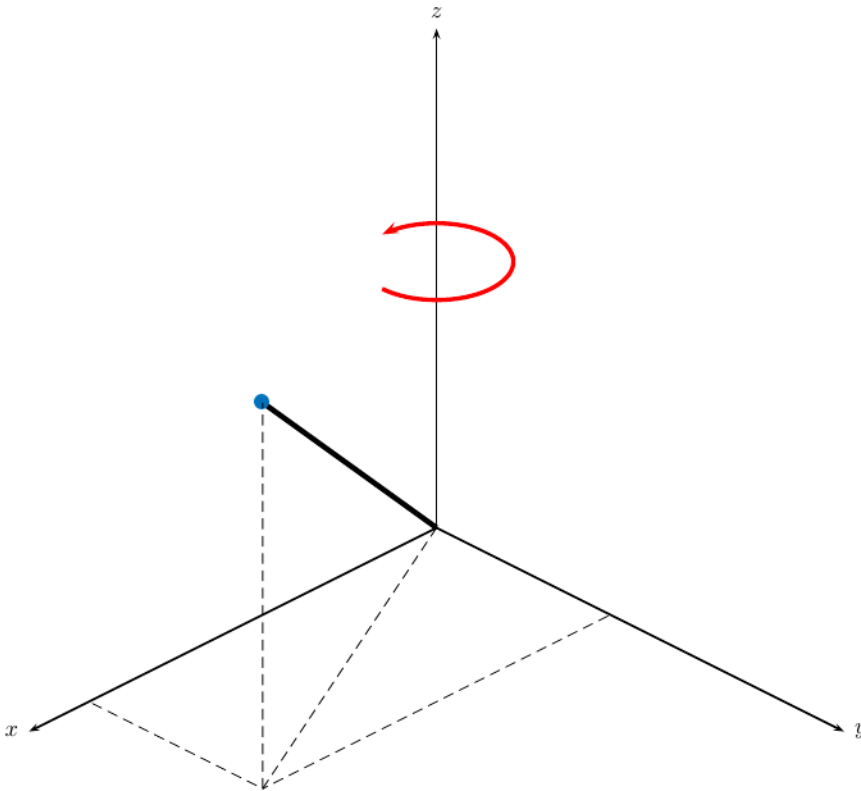
If you take your right hand and point it along the positive *x*-axis with your palm facing the positive *y*-axis and extend your thumb, your thumb indicates the positive direction of the *z*-axis.

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, one can rotated a vector 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:
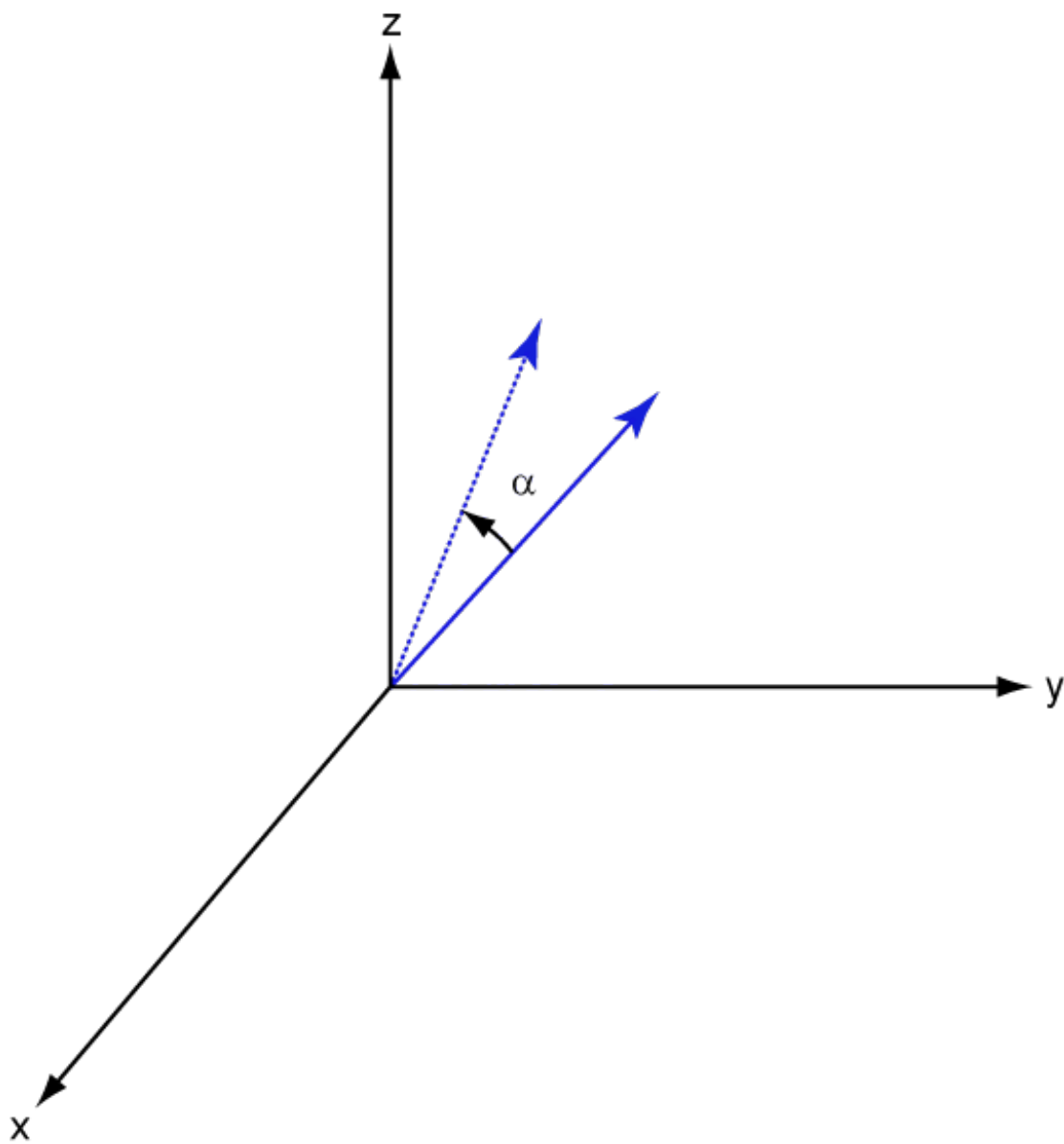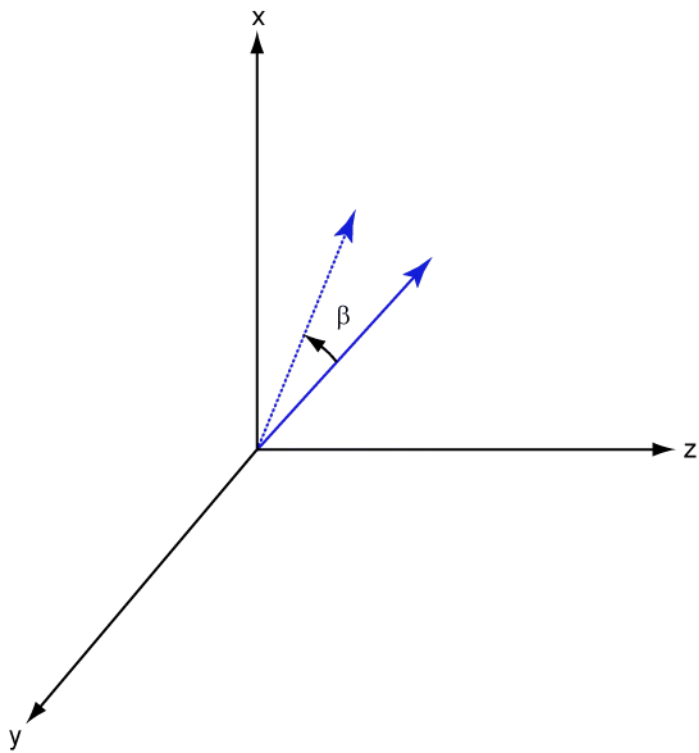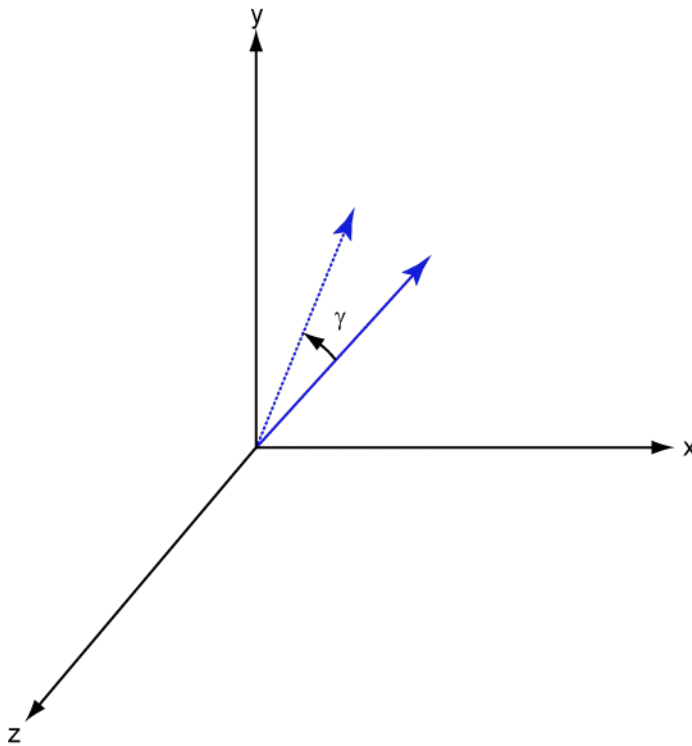
For any rotation, there is an inverse rotation satisfying $A^{-1}A = 1$. For example, the inverse of the x-axis rotation matrix is obtained by changing the sign of the angle:

$$R_x^{-1}(\alpha) = R_x(-\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} = R_x^{'}(\alpha)$$

This example illustrates a basic property: the inverse rotation matrix equals 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, $\mathbf{i}, \mathbf{j}, \mathbf{k}$, and rotate them all using the rotation matrix *A*. This produces a new set of basis

vectors $\mathbf{i}^{'}, \mathbf{j}^{'}, \mathbf{k}^{'}$ related to the original by:

$$
\begin{aligned}
\mathbf{i}' &= A\mathbf{i} \\
\mathbf{j}' &= A\mathbf{j} \\
\mathbf{k}' &= A\mathbf{k}
\end{aligned}
$$

The new basis vectors can be written as linear combinations of the old ones and involve the transpose:

$$
\begin{bmatrix} \mathbf{i}' \\ \mathbf{j}' \\ \mathbf{k}' \end{bmatrix} = A' \begin{bmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \end{bmatrix}
$$

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 some algebraic manipulation, one can derive the transformation of components for a fixed vector when the basis (or coordinate system) rotates

$$
\begin{bmatrix} v_x' \\ v_y' \\ v_z' \end{bmatrix} = A^{-1} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = A' \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}
$$

Thus the change in components of a vector when the coordinate system rotates involves the transpose of the rotation matrix. The next figure illustrates how a vector stays fixed as the coordinate system rotates around the x-axis. The figure after shows how this can be interpreted as a rotation *of the vector* in the opposite direction.
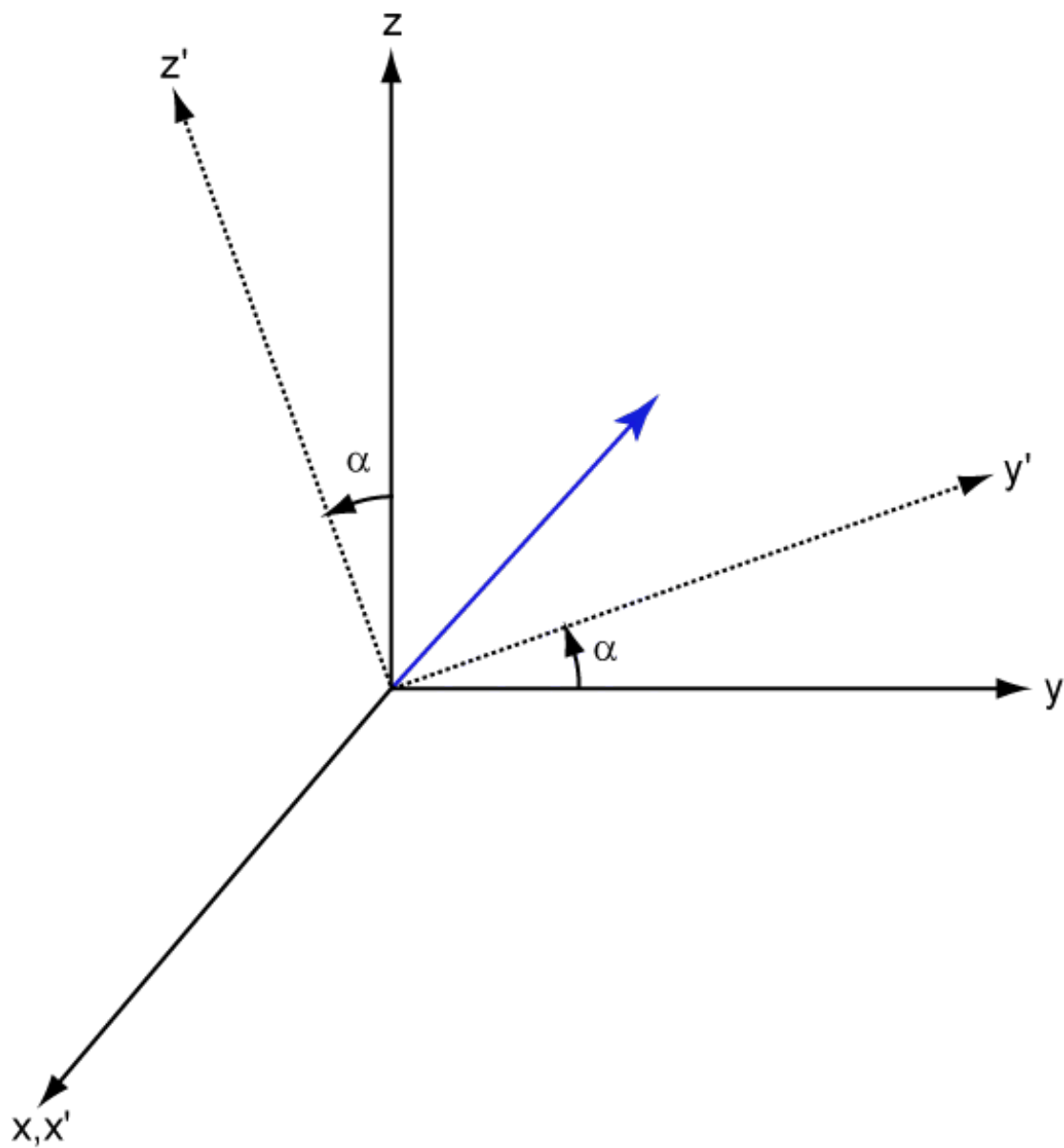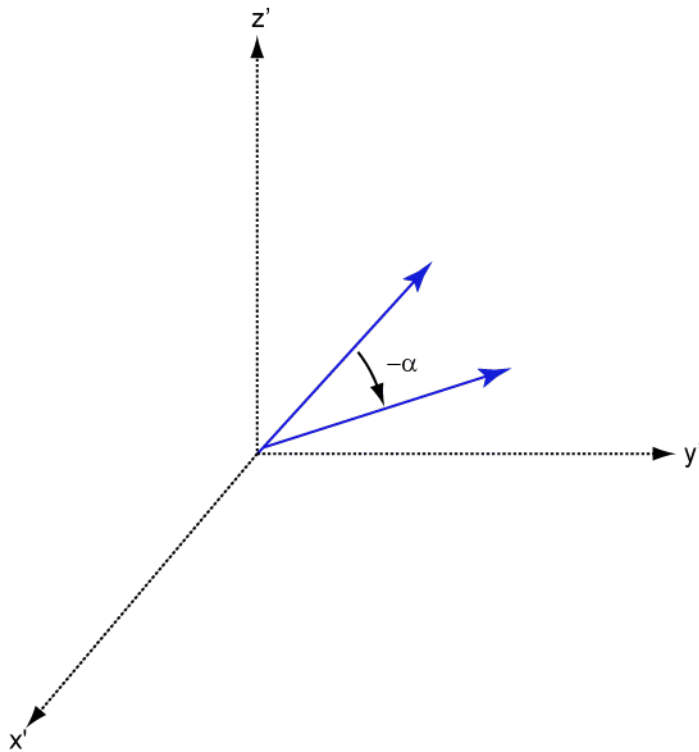
More About

- "Global and Local Coordinate Systems" on page 10-21

Spherical Coordinates

In this section...

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 10-14 and "U and V Coordinates" on page 10-15.

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* is the angle from the positive *x*-axis toward the positive *y*-axis, to the vector's orthogonal projection onto the *xy* plane. The azimuth angle is between –180 and

180 degrees. The *elevation angle* is the angle from the vector's orthogonal projection onto the *xy* plane toward the positive *z*-axis, to the vector. The elevation angle is between –90 and 90 degrees. These definitions assume the boresight direction is the positive *x*-axis.

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 that appears as a green solid line. The coordinate system is relative to the center of a uniform linear array, whose elements appear as blue circles.
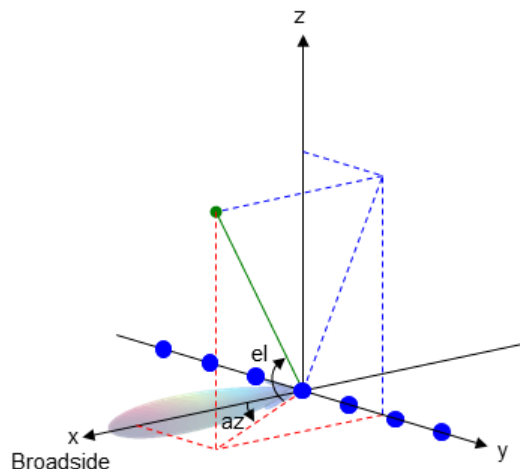
Phi and Theta Angles

As an alternative to azimuth and elevation angles, you can use angles denoted by φ and θ to express the location of a point on the unit sphere. To convert the φ/θ representation to and from the corresponding azimuth/elevation representation, use coordinate conversion functions, `phitheta2azel` and `azel2phitheta`.

The φ angle is the angle from the positive *y*-axis toward the positive *z*-axis, to the vector's orthogonal projection onto the *yz* plane. The φ angle is between 0 and 360 degrees. The θ angle is the angle from the *x*-axis toward the *yz* plane, to the vector itself. The θ angle is between 0 and 180 degrees.

The figure illustrates φ and θ for a vector that appears as a green solid line. The coordinate system is relative to the center of a uniform linear array, whose elements appear as blue circles.
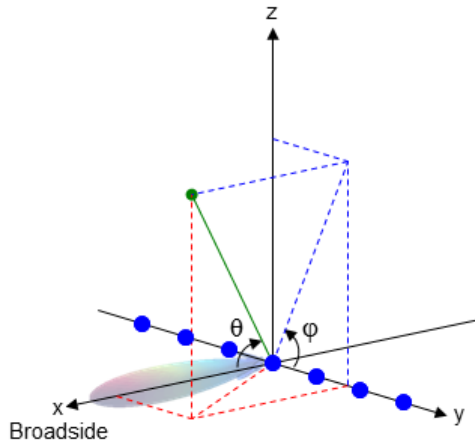
The coordinate transformations between φ/θ and *az/el* are described by the following equations

$$\sin(\text{el}) = \sin\phi\sin\theta$$
$$\tan(\text{az}) = \cos\phi\tan\theta$$

$$\cos\theta = \cos(\text{el})\cos(\text{az})$$
$$\tan\phi = \tan(\text{el}) / \sin(\text{az})$$

U and V Coordinates

In radar applications, it is often useful to parameterize the hemisphere x ≥ 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.

In terms of azimuth and elevation, the u and v coordinates are

$$u = \cos el \sin az$$
$$v = \sin el$$

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

$$\tan\phi = u\,/\,v$$
$$\sin\theta = \sqrt{u^2 + v^2}$$

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 Angle

The special case of the uniform linear arrays (ULA) uses the concept of the *broadside angle*. The broadside angle is the angle measured from array normal direction projected onto the plane determined by the signal incident direction and the array axis to the signal incident direction. Broadside angles assume values in the interval [–90,90] degrees. The following figure illustrates the definition of the broadside angle.
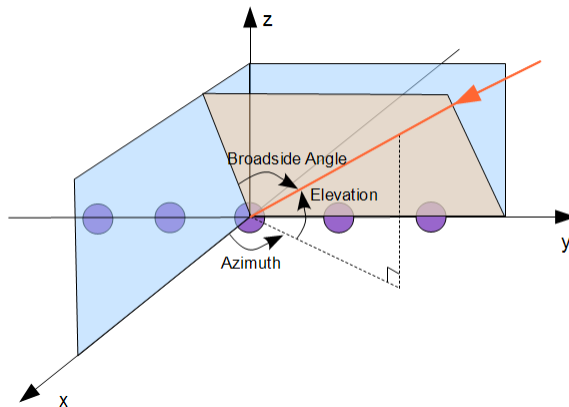
The shaded gray area in the figure is the plane determined by the signal incident direction and the array axis. The broadside angle is positive when measured toward the

positive direction of the array axis. A number of algorithms for ULA's use the broadside angle instead of the azimuth and elevation angles. The algorithms do so because the broadside angle more accurately describes the ability to discern direction of arrival with this geometry.

Phased Array System Toolbox software provides functions `az2broadside` and `broadside2az` for converting between azimuth and broadside angles. The following equation determines the broadside angle, ß, from the azimuth and elevation angles, az and el:

$$\beta = \sin^{-1}(\sin(az)\cos(el))$$

Expressing the broadside angle in terms of the azimuth and elevation angles reveals a number of important characteristics, including:

- For an elevation angle of zero degrees, the broadside angle is equal to the azimuth angle.
- Elevation angles equally above and below the xy plane result in identical broadside angles.

The following figure depicts a ULA with elements spaced d meters apart. The ULA is illuminated by a plane wave emitted from a point source in the far field. For convenience, the elevation angle is zero degrees. The plane determined by the signal incident direction and the array axis is the xy plane. The broadside angle reduces to the azimuth angle.
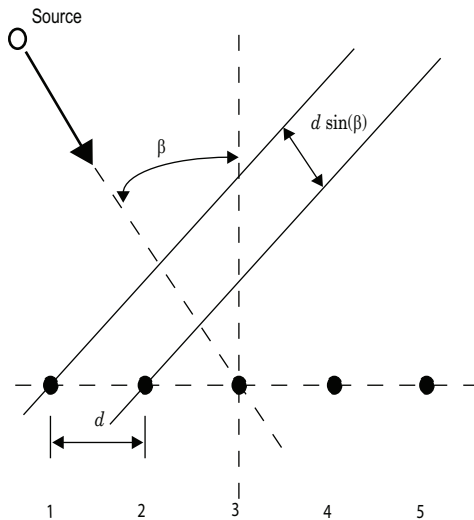
Because of the angle of arrival, the array elements are not simultaneously illuminated by the plane wave. The additional distance the incident wave travels between array elements is d sin(β) where *d* is the distance between array elements. Therefore, the constant time delay between array elements is:

$$\tau = \frac{d \sin(\beta)}{c},$$

where *c* is the speed of the wave.

For broadside angles of ±90 degrees, the plane wave is incident on the array along the array axis and the time delay between sensors reduces to ±*d/c*. For a broadside angle of 0 degrees, the plane wave illuminates all elements of the ULA simultaneously and the time delay between elements is zero.

The following examples show the use of the utility functions `az2broadside` and `broadside2az`:

A target is located at an azimuth angle of 45 degrees and elevation angle of 60 degrees relative to a ULA. Determine the corresponding broadside angle:

```
bsang = az2broadside(45,60)
% approximately 21 degrees
```

Calculate the azimuth corresponding to a broadside angle of 45 degrees and an elevation of 20 degrees:

```
az = broadside2az(45,20)
% approximately 49 degrees
```

Global and Local Coordinate Systems

In this section...
"Global Coordinate System" on page 10-21
"Local Coordinate Systems" on page 10-21
"Converting Between Global and Local Coordinate Systems" on page 10-40

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 and 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, WidebandFreeSpace, phased.LOSChannel, or phased.WidebandLOSChannel. If you model two-ray multipath propagation using the phased.TwoRayChannel System object, 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 fusilage 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.
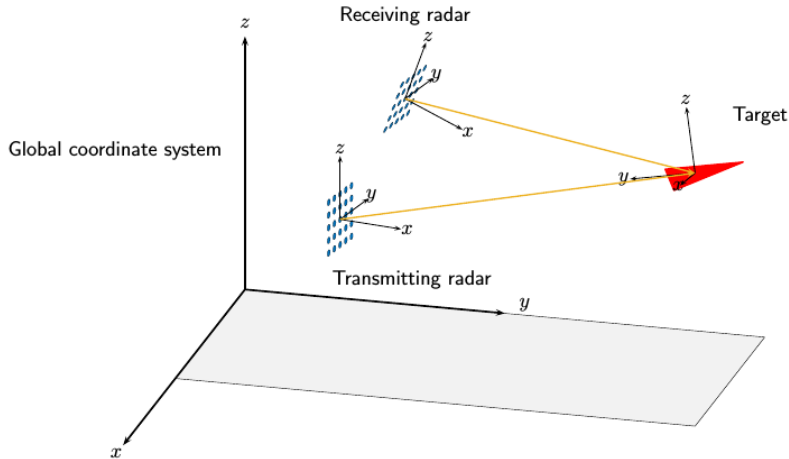
The next few sections review the local coordinate systems used by arrays.

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 thephased.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 are 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 position 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
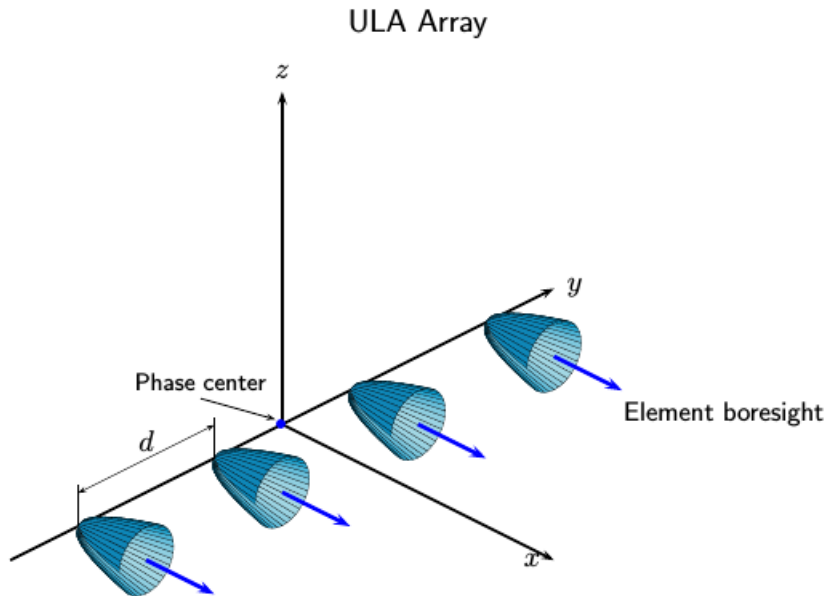
ULA Element Boresight Direction

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)
viewArray(sULA)
```

```
ElementLocs =

         0          0          0          0
   -0.7500    -0.2500     0.2500     0.7500
         0          0          0          0
```

Array Geometry

Aperture Size:
Y axis = 2 m
Element Spacing:
Δ y = 500 mm
Array Axis: Y axis
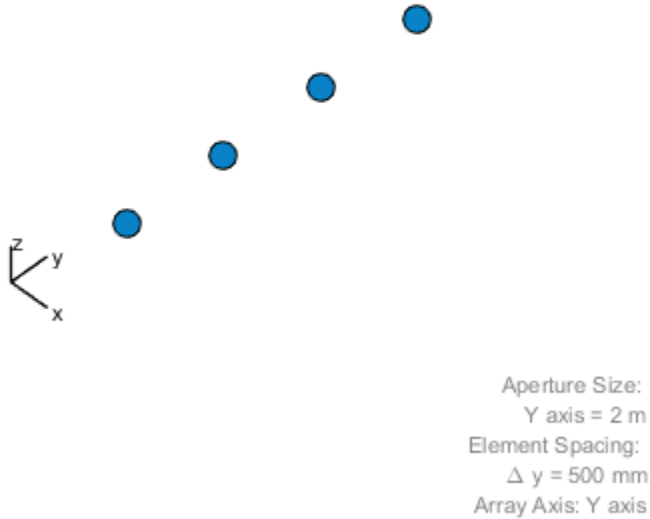
The origin of the array-centric local coordinate system is set to the phase center of the array. The phase center is the average value of the array element positions.

```
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)
viewArray(sULA1)


ElementLocs =

        0          0        0         0          0
  -0.6000    -0.3000        0    0.3000     0.6000
        0          0        0         0          0
```

Array Geometry
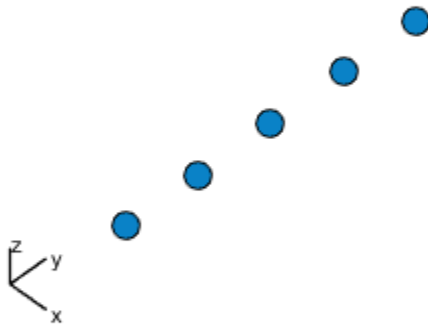
Aperture Size:
Y axis = 1.5 m
Element Spacing:
△ y = 300 mm
Array Axis: Y axis

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.

URA Array
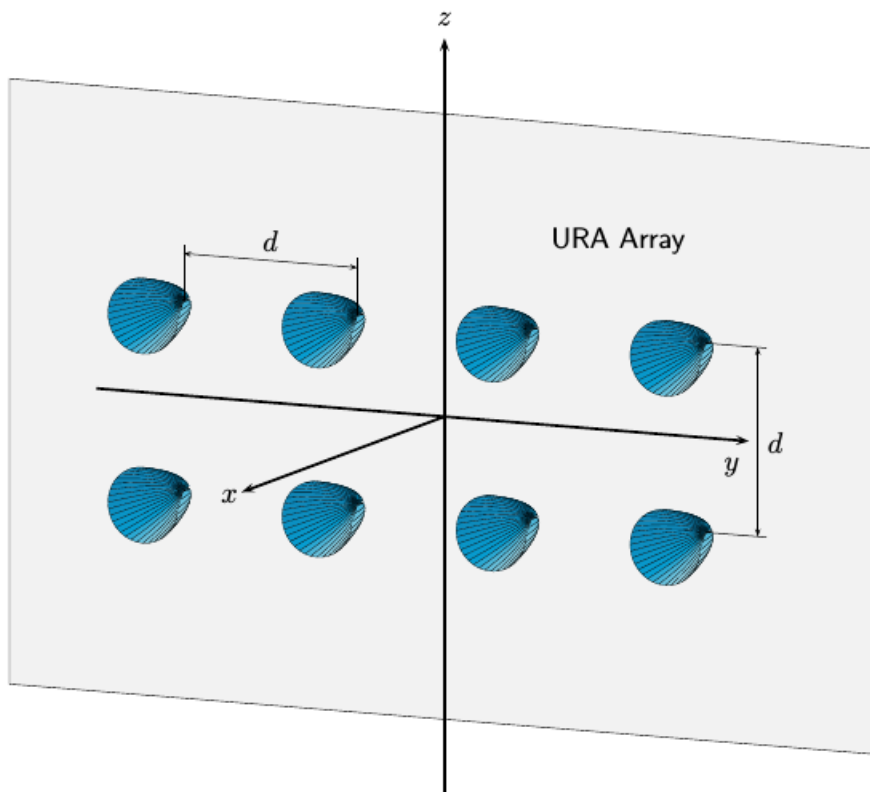
Element Boresight Direction

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)
viewArray(sURA)


ElementLocs =

  Columns 1 through 7

        0         0         0         0         0         0         0
  -0.7500   -0.7500   -0.2500   -0.2500    0.2500    0.2500    0.7500
   0.2500   -0.2500    0.2500   -0.2500    0.2500   -0.2500    0.2500

  Column 8

        0
   0.7500
  -0.2500
```

Array Geometry
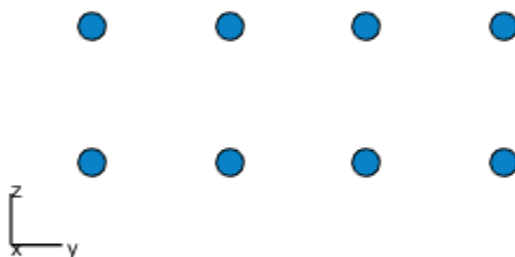
Aperture Size:
Y axis = 2 m
Z axis = 1 m
Element Spacing:
Δ y = 500 mm
Δ z = 500 mm

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)
viewArray(sURA1)


ElementLocs =

  Columns 1 through 7

        0        0        0        0        0        0        0
  -0.3000  -0.3000  -0.3000  -0.3000  -0.3000        0        0
   0.6000   0.3000        0  -0.3000  -0.6000   0.6000   0.3000

  Columns 8 through 14

        0        0        0        0        0        0        0
        0        0        0   0.3000   0.3000   0.3000   0.3000
        0  -0.3000  -0.6000   0.6000   0.3000        0  -0.3000

  Column 15

        0
   0.3000
  -0.6000
```
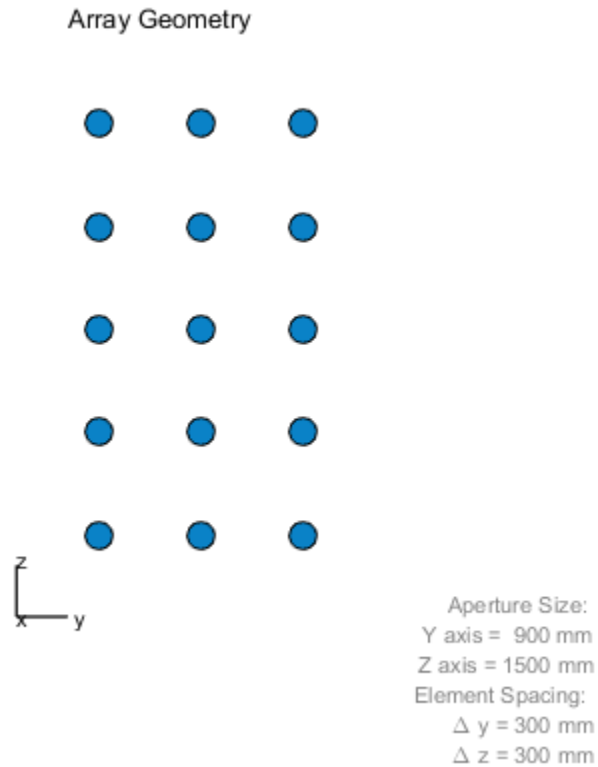
Array Geometry

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 =

        1000
```

```
ang =

         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.

UCA Array

Phase center
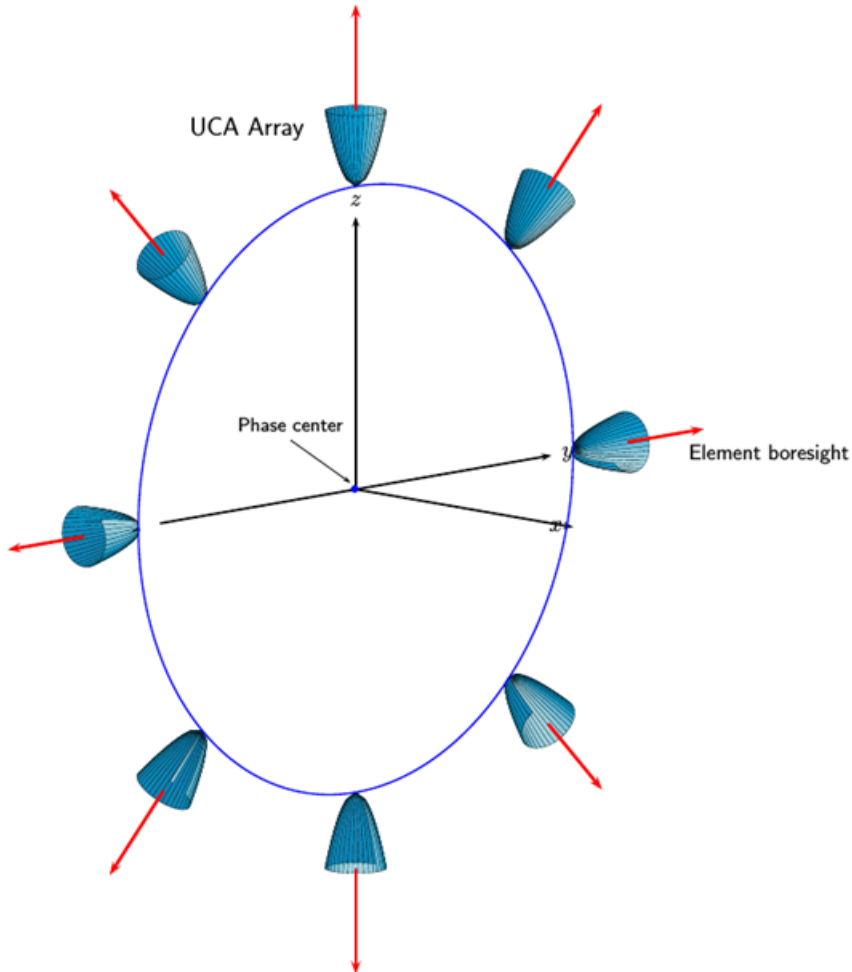
Element boresight

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.
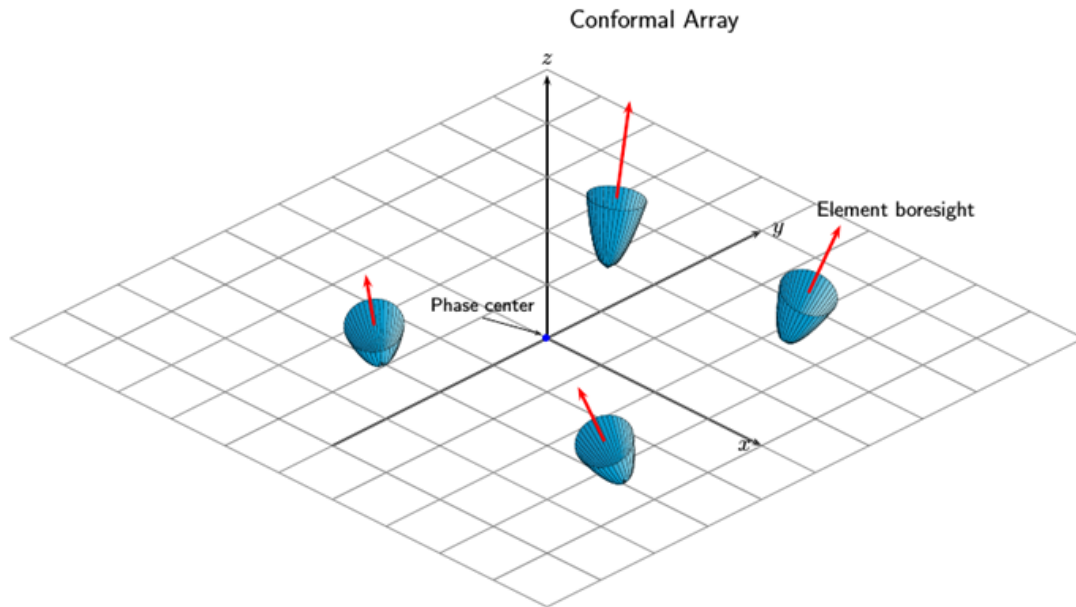
Conformal Array

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)
normvec = getElementNormal(sConformArray)
viewArray(sConformArray,'ShowIndex','All','ShowNormal',true)


pos =

    0.1666   -0.0833         0    0.1332
   -0.0666   -0.1666    0.0833    0.2498
   -0.0500    0.0500    0.0666         0


normvec =

  -140   -140     90     90
    80     80     80     80
```

Array Geometry

Array Span:
X axis = 249.827 mm
Y axis = 416.378 mm
Z axis = 116.586 mm
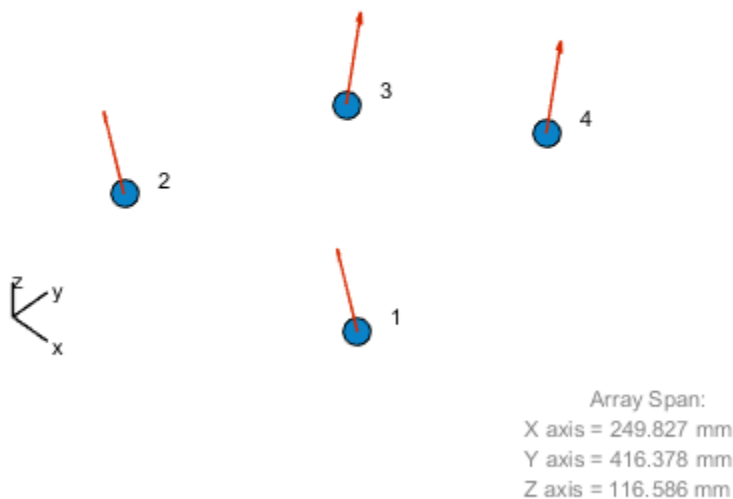
Converting Between Global and Local Coordinate Systems

In many array processing applications, it is necessary to convert between global and local coordinates. Two utility functions, `global2localcoord` and `local2globalcoord`, enable you to do this conversion.

Convert Local Spherical Coordinates to Global Rectangular Coordinates

Assume a stationary target 1000 meters from a URA at an azimuth angle of 30 degrees and elevation angle of 45 degrees. The phase center of the URA is located at the rectangular coordinates [1000 500 100] in the global coordinate system. The local coordinate axes of the URA are parallel to the global coordinate axes. Determine the position of the target in rectangular coordinates in the global coordinate system.

In this example, the target's location is specified in local spherical coordinates. The target is 1000 meters from the array, which means that R=1000. The azimuth angle of 30 degrees and elevation angle of 45 degrees give the direction of the target from the array. The spherical coordinates of the target in the local coordinate system are (30,45,1000). To convert to global rectangular coordinates, you must know the position of the array in global coordinates. The phase center of the array is located at [1000 500 100]. To convert from local spherical coordinates to global rectangular coordinates, use the `'sr'` option.

```
gCoord = local2globalcoord([30; 45; 1000],'sr',...
    [1000; 500; 100]);
```

Convert Global Rectangular Coordinates to Local Spherical Coordinates

Assume a stationary target with global rectangular coordinates *(5000,3000 ,50)*. The phase center of a URA has global rectangular coordinates *(1000,500,10)*. The local coordinate axes of the URA are *(0,1,0)*, *(1,0,0)*, and *(0,0,–1)*. Determine the position of the target in local spherical coordinates.

```
lCoord = global2localcoord([5000; 3000; 50],'rs',...
    [1000; 500; 10],[0 1 0;1 0 0;0 0 -1]);
```

The output `lCoord` takes the form (*az*,*el*,*R*). The target in local coordinates has an azimuth of approximately 58 degrees, an elevation of 0.5 degrees, and a range of 4717.16 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, reflected 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 tranmitted 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.

```
sAnt = phased.IsotropicAntennaElement('BackBaffled',false);
stxURA = phased.URA('Element',sAnt','Size',[5,5],'ElementSpacing',0.4*lam*[1,1]);
txradarAx = rotz(1)*roty(2)*rotz(90);
sTxRadar = phased.Platform('InitialPosition',[50;50;50],...
    'Velocity',[0;0;0],'InitialOrientationAxes',txradarAx,...
    'OrientationAxesOutputPort',true);
sRad = phased.Radiator('Sensor',stxURA,'PropagationSpeed',c,...
    'WeightsInputPort',true,'OperatingFrequency',fc);
sSV = phased.SteeringVector('SensorArray',stxURA,'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 `step` method. Set up a simple inline function, `rcsval`, to compute ficticious but reasonable values for RCS at different ray angles.

```
tgtAx = rotz(10)*roty(15)*rotx(20);
sTarget = phased.Platform('InitialPosition',[100; 10000; 100],...
    'MotionModel','Acceleration','InitialVelocity',[-20;0;0],'Acceleration',[.015;.015;
    'InitialOrientationAxes',tgtAx,'OrientationAxesOutputPort',true);
sTgt = phased.RadarTarget('MeanRCS',1,'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);
sRxRadar = phased.Platform('InitialPosition',[50;50;200],...
    'Velocity',[0;0;0],'InitialOrientationAxes',rxradarAx,...
    'OrientationAxesOutputPort',true);
srxURA = phased.URA('Element',sAnt','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.

Receiving radar coordinate system

Transmitting radar coordinate system

Target coordinate system

Global coordinate system
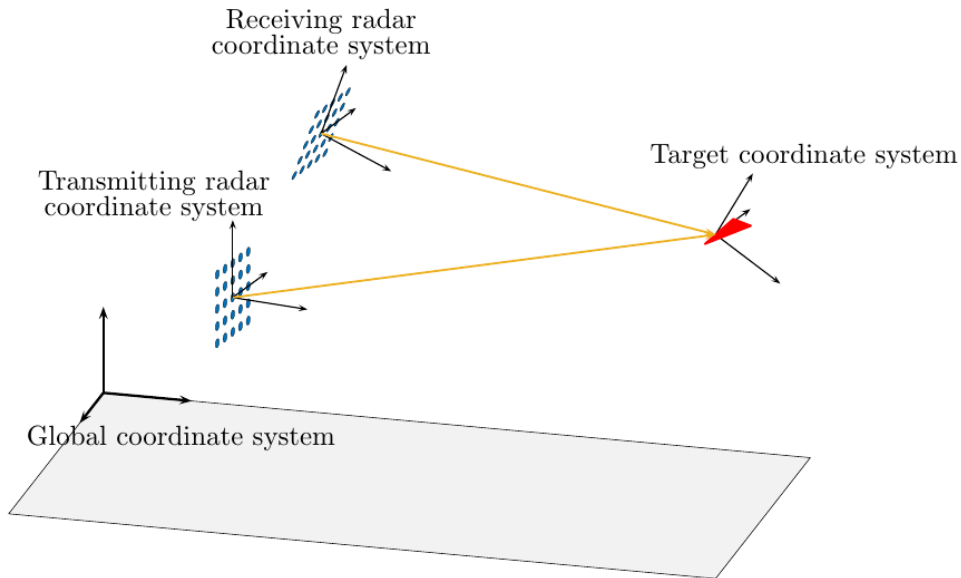
Specify Transmitted Waveform and Transmitter Amplification

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;
sWav = phased.LinearFMWaveform('PulseWidth',tau,...
    'OutputFormat','Pulses','NumPulses',1,'PRF',prf,'SampleRate',fs);
sTransmit = phased.Transmitter('PeakPower',1000.0,'Gain',40);
```

Create a matched filter from the transmitted waveform.

```
sMF = phased.MatchedFilter('Coefficients',getMatchedFilter(sWav));
```

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.

```
sFS1 = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false);
sFS2 = 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];
srxBF = phased.PhaseShiftBeamformer('SensorArray',srxURA,...
    'DirectionSource','Property','Direction',rxangsteer,...
    'PropagationSpeed',c,'OperatingFrequency',fc);
```

Simulation loop

Each iteration of the processing lool 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 find its peak value.

Transmit 100 pulses of the waveform. Transmit one pulse every 100 milliseconds.

```
rng default
t = 0;
Npulse = 100;
dt = 0.1;
```

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.

```
for k = 1:Npulse

    t = t + dt;
```

Generate the transmitted waveform.

```
    waveform = step(sWav);
```

First, 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 `step` 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] = step(sTxRadar,t);
    [rxradarPos,rxradarVel] = step(sRxRadar,t);
    [tgtPos,tgtVel,tgtAx1] = step(sTarget,t);
```

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 = step(sTransmit,waveform);
```

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];
sv1 = step(sSV,fc,txangsteer);
wavrad = step(sRad,txwaveform,tgtang_tlcs,conj(sv1));
```

Propagate the signal from the transmitting radar to the target. Propagation coordinates are in the global coordinate system.

```
wavprop1 = step(sFS1,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 = step(sTgt,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 = step(sFS2,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(srxURA,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 = step(srxBF,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 = step(sMF,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.

```
figure
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
```


Target Track in Transmitter Local Coordinates
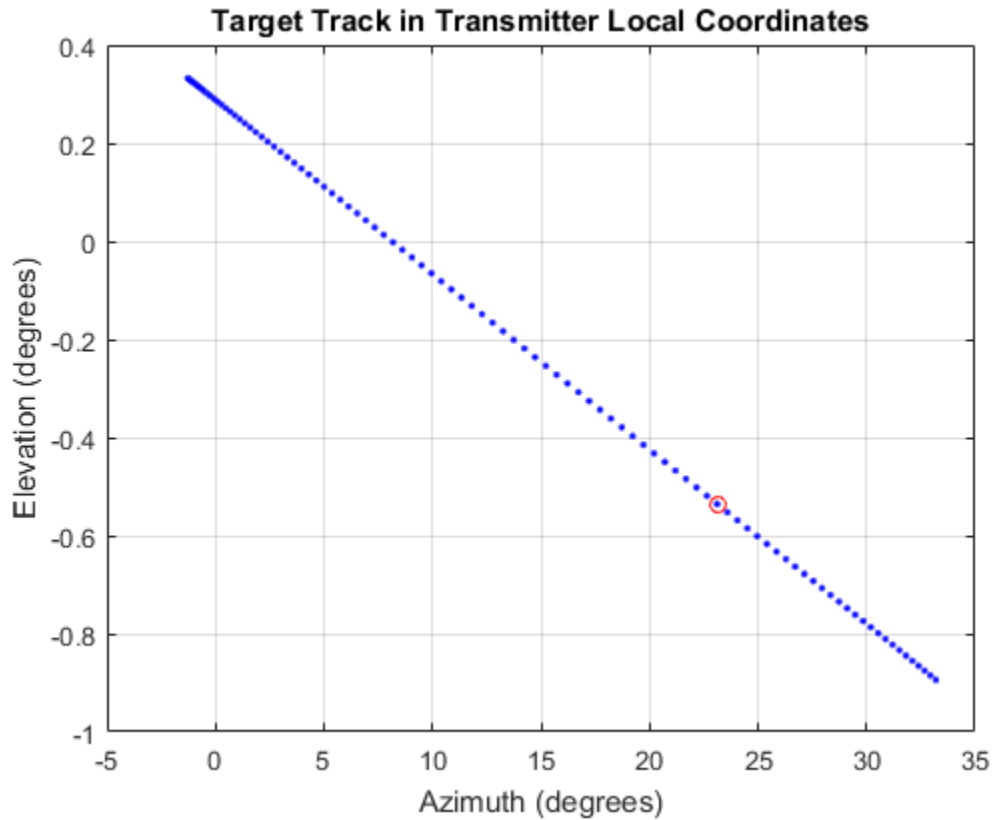
Plot the target track in azimuth and elevation with respect to the receiver local coordinates. The red circle denotes the direction toward which the beamformer points.

```
figure
plot(azes2,elevs2,'.b')
grid
xlabel('Azimuth (degrees)')
ylabel('Elevation (degrees)')
title('Target Track in Receiver Local Coordinates')
```

```
hold on
plot(rxangsteer(1),rxangsteer(2),'or')
hold off
```


Target Track in Receiver Local Coordinates
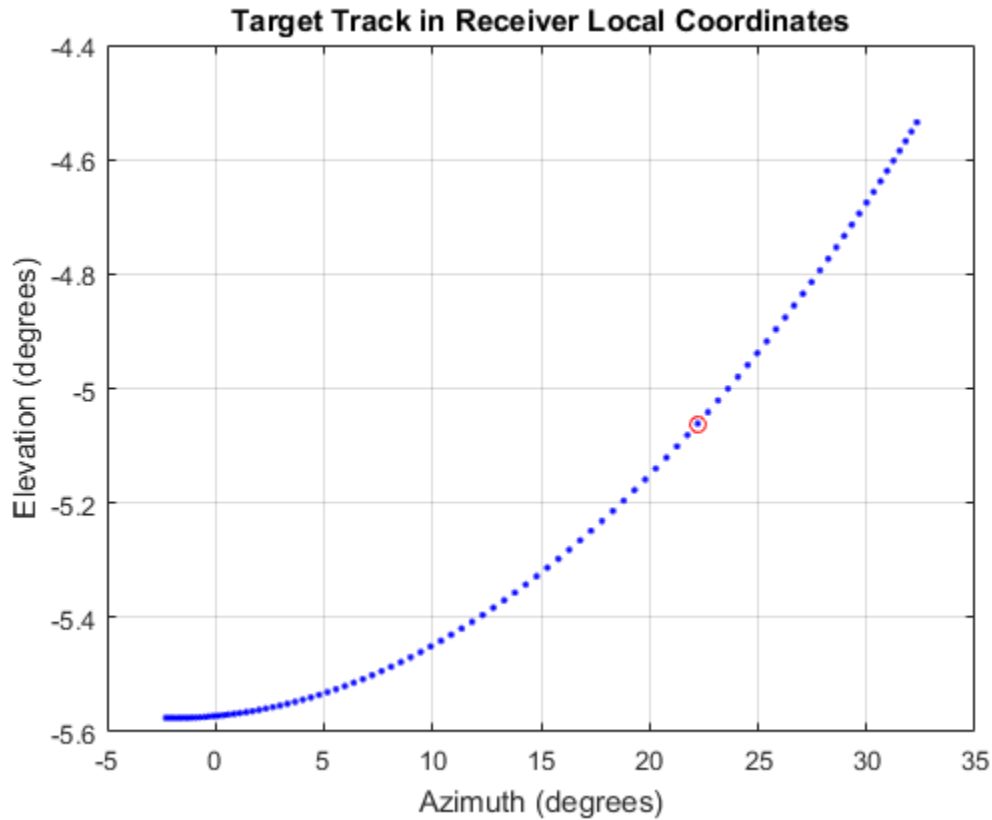
Plot the returned signal amplitude vs azimuth in the receiver local coordinates. The value of the amplitude depends on several factors.

```
figure;
plot(azes2,rxsig,'.')
grid
xlabel('Azimuth (degrees)')
ylabel('Amplitude')
title('Amplitude vs Azimuth in Receiver Local Coordinates')
```
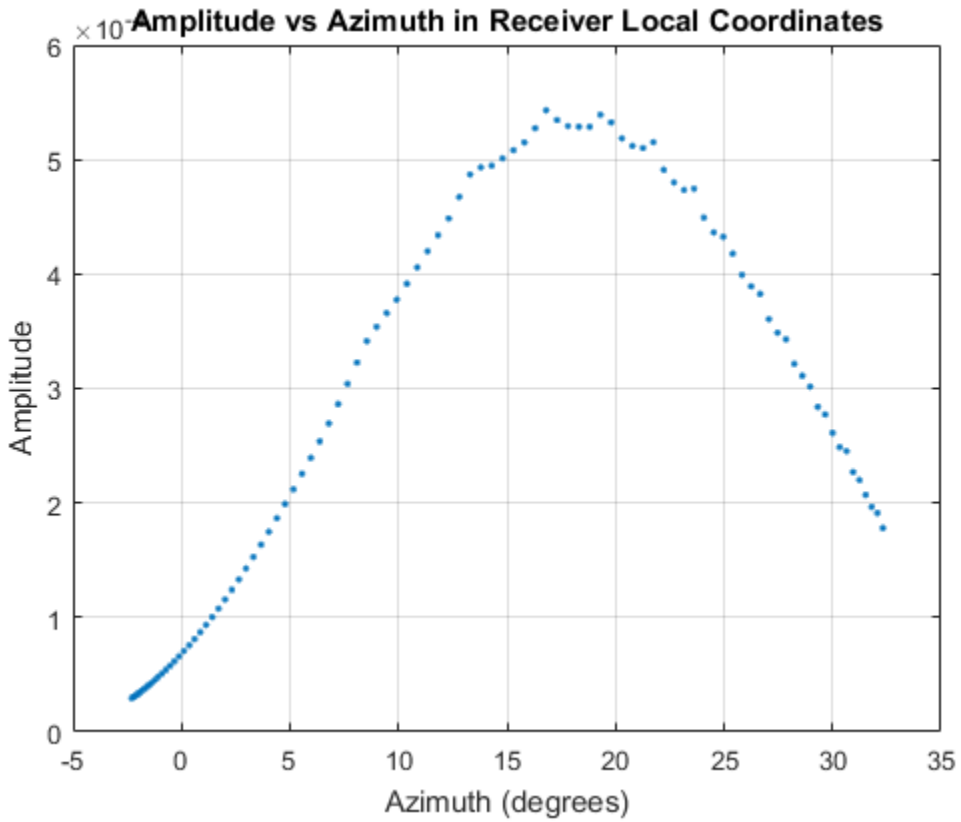
Motion Modeling in Phased Array Systems

In this section...
"Support for Motion Modeling" on page 10-52
"Platform Motion with Constant Velocity" on page 10-53
"Platform Motion with Nonconstant Velocity" on page 10-54
"Track Range and Angle Changes Between Platforms" on page 10-55

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 10-21. 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.
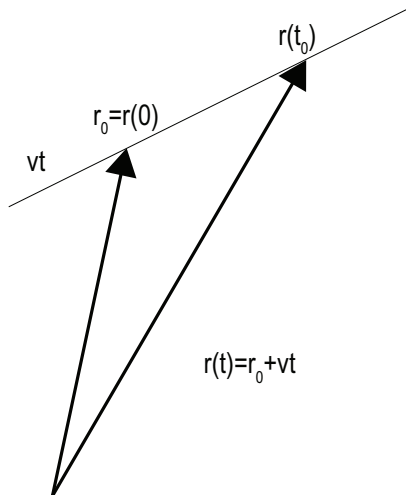
When the platform represents a sensor element or array, it is important to know the orientation of the element or array *local coordinate axes*. For example, the orientation of the local coordinate axes is necessary to extract angle information from incident waveforms. See "Global and Local Coordinate Systems" on page 10-21 for a description of global and local coordinate systems in the software. Finally, for platforms with nonconstant 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's initial position and velocity. Assume that the initial position of the platform is 100 meters (m) 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).

```
hplat = 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` object, the property is assigned its default value of `[1 0 0;0 1 0;0 0 1]`.

Use the step method to simulate the translational motion of the platform.

```
InitialPos = hplat.InitialPosition;
for k = 1:Nsteps
    pos = step(hplat,Tstep);
end
FinalPos = pos+hplat.Velocity*Tstep;
DistTravel = norm(FinalPos-InitialPos);
```

The `step` method returns the current position of the platform and then updates the platform position based on the time step and velocity. Equivalently, the first time you invoke the `step` method, 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. Invoking the step method 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 Nonconstant 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's speed, you can often approximate the velocity as constant. However, there are situations where you must update the platform's velocity over time. You can do so with `phased.Platform` because the `Velocity` property is *tunable*. See "What You Cannot Change While Your System Is Running" for details.

In this example, assume you model 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;
hplat = phased.Platform('InitialPosition',[100;100;0]);
for k = 1:Nsteps/2
    [pos,vel] = step(hplat,Tstep);
end
hplat.Velocity = [7.07; 7.07; 0];
for k=Nsteps/2+1:Nsteps
    [pos,vel] = step(hplat,Tstep);
end
```

Track Range and Angle Changes Between Platforms

This example uses the phased.Platform object to model the changes 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.

The example uses `phased.Platform` to model the motion of the target and radar. The `global2localcoord` function translates the target's rectangular coordinates in the global coordinate system to spherical coordinates in the local coordinate system of the radar.

```
PRF = 1e3;
Tstep = 1/PRF;
hradar = phased.Platform('InitialPosition',[1000;1000;0]);
htgt = phased.Platform('InitialPosition',[5000;8000;0],...
    'Velocity',[-30;-45;0]);
% Calculate initial target range and angle
[InitRng, InitAng] = rangeangle(htgt.InitialPosition,...
    hradar.InitialPosition);
% Calculate relative radial speed
v = radialspeed(htgt.InitialPosition,htgt.Velocity,...
    hradar.InitialPosition);
% Simulate target motion
Npulses = 10;                          % Number of pulses
for num = 1:Npulses
    tgtpos = step(htgt,Tstep);
end
tgtpos = tgtpos+htgt.Velocity*Tstep;
% Calculate final target range and angle
[FinalRng,FinalAng] = rangeangle(tgtpos,...
```

```
    hradar.InitialPosition);
DeltaRng = FinalRng-InitRng;
```

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.

Related Examples

*   "Introduction to Space-Time Adaptive Processing"

Model Motion of Circling Airplane

Start with an airplane moving at 150 kmh in a circle of radius 10 km and descending at the same time at a rate of 20 m/sec. Compute the motion of the airplane from its instantaneous acceleration as an argument to the `step` method. Set the initial orientation of the platform to the identity, coinciding with the global coordinate system.

Set up the scenario

Specify the initial position and velocity of the airplane. The airplane has a ground range of 10 km and an altitude of 20 km.

```
range = 10000;
alt = 20000;
initPos = [cosd(60)*range;sind(60)*range;alt];
originPos = [1000,1000,0]';
originVel = [0,0,0]';
vs = 150.0;
phi = atan2d(initPos(2)-originPos(2),initPos(1)-originPos(1));
phi1 = phi + 90;
vx = vs*cosd(phi1);
vy = vs*sind(phi1);
initVel = [vx,vy,-20]';
platform = phased.Platform('MotionModel','Acceleration',...
    'AccelerationSource','Input port','InitialPosition',initPos,...
    'InitialVelocity',initVel,'OrientationAxesOutputPort',true,...
    'InitialOrientationAxes',eye(3));
relPos = initPos - originPos;
relVel = initVel - originVel;
rel2Pos = [relPos(1),relPos(2),0]';
rel2Vel = [relVel(1),relVel(2),0]';
r = sqrt(rel2Pos'*rel2Pos);
accelmag = vs^2/r;
unitvec = rel2Pos/r;
accel = -accelmag*unitvec;
T = 0.5;
N = 1000;
```

Compute the trajectory

Specify the acceleration of an object moving in a circle in the *x-y* plane. The acceleration is v^2/r towards the orign

```
posmat = zeros(3,N);
```

```
r1 = zeros(N);
v = zeros(N);
for n = 1:N
    [pos,vel,oax] = platform(T,accel);
    posmat(:,n) = pos;
    vel2 = vel(1)^2 + vel(2)^2;
    v(n) = sqrt(vel2);
    relPos = pos - originPos;
    rel2Pos = [relPos(1),relPos(2),0]';
    r = sqrt(rel2Pos'*rel2Pos);
    r1(n) = r;
    accelmag = vel2/r;
    accelmag = vs^2/r;
    unitvec = rel2Pos/r;
    accel = -accelmag*unitvec;
end
```

Display the final orientation of the local coordinate system.

```
disp(oax)
```

```
   -0.3658   -0.9307   -0.0001
    0.9307   -0.3658   -0.0010
    0.0009   -0.0005    1.0000
```

Plot the trajectory and the origin position

```
posmat = posmat/1000;
figure(1)
plot3(posmat(1,:),posmat(2,:),posmat(3,:),'b.')
hold on
plot3(originPos(1)/1000,originPos(2)/1000,originPos(3)/1000,'ro')
xlabel('X (km)')
ylabel('Y (km)')
zlabel('Z (km)')
grid
hold off
```
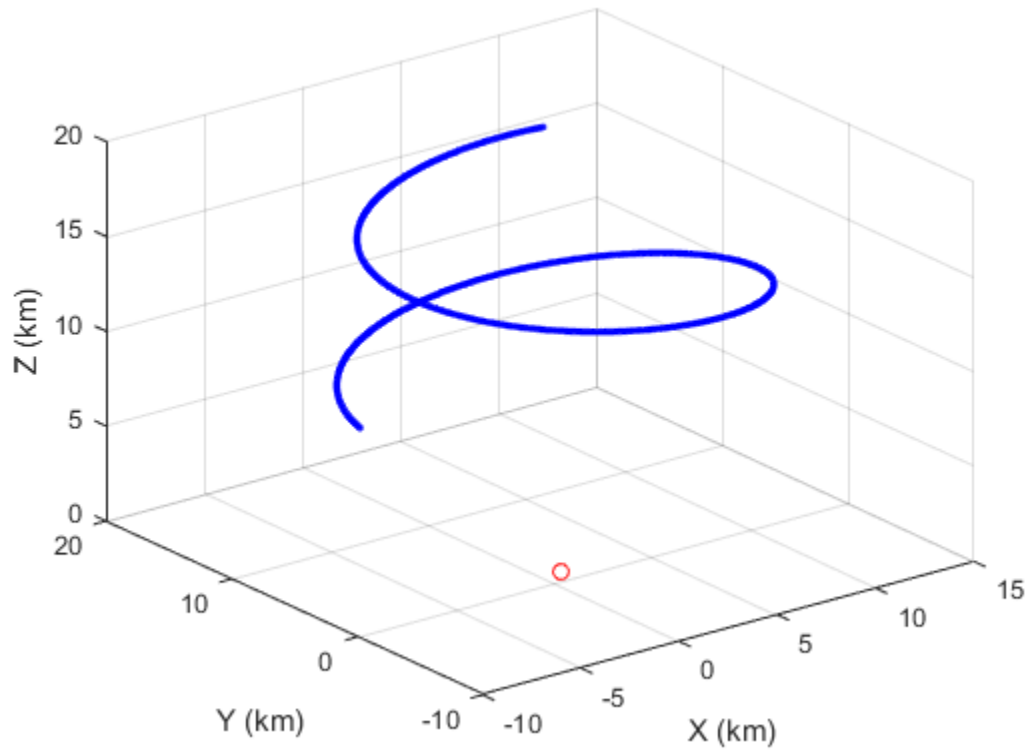
Visualize Multiplatform Scenario

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];
sAirplane1 = 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]';
sGroundRadar = 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]';
sGroundVehicle = 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]';
sAirplane2 = phased.Platform('MotionModel','Velocity',...
    'InitialPosition',airplane2Pos,'Velocity',airplane2Vel,...
    'InitialOrientationAxes',eye(3));
```

Set up the scenario viewer. Specity 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];
sSV = phased.ScenarioViewer('BeamRange',8.0e3,'BeamWidth',[2;30],'UpdateRate',updateRat
    'PlatformNames',{'Ground Radar','Turning Airplane','Vehicle','Airplane 2'},'ShowPos
    '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] = step(sGroundRadar,updateRate);
    [airplane1Pos,airplane1Vel,airplane1Axes] = ...
        step(sAirplane1,updateRate,airplane1Accel);
    [vehiclePos,vehicleVel] = step(sGroundVehicle,updateRate);
    [airplane2Pos,airplane2Vel] = step(sAirplane2,updateRate);
    step(sSV,groundRadarPos,groundRadarVel,[airplane1Pos,vehiclePos,airplane2Pos],...
        [airplane1Vel,vehicleVel,airplane2Vel]);
    BeamSteering = sSV.BeamSteering(1);
    BeamSteering = mod(BeamSteering + 4,360.0);
    if BeamSteering > 180.0
        BeamSteering = BeamSteering - 360.0;
    end
    sSV.BeamSteering(1) = BeamSteering;
    pause(0.2);
end
```
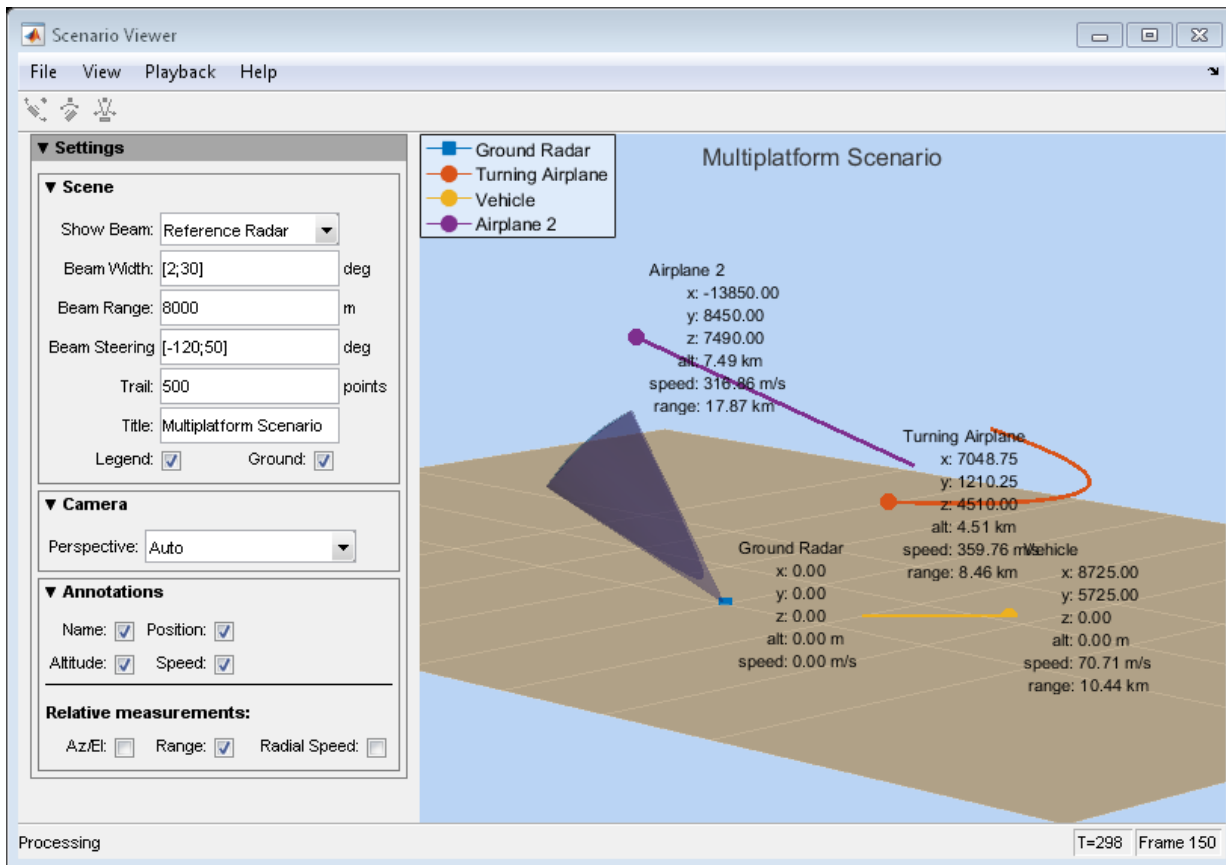
Doppler Shift and Pulse-Doppler Processing

In this section...
"Support for Pulse-Doppler Processing" on page 10-63
"Convert Speed to Doppler Shift" on page 10-63
"Convert Doppler Shift to Speed" on page 10-64
"Pulse-Doppler Processing of Slow-Time Data" on page 10-64
"Range and Speed using Pulse-Doppler Processing" on page 10-65

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 illustrate 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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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);
```

10-65

```
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 targ
    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')
```

Periodogram Spectrum Estimate
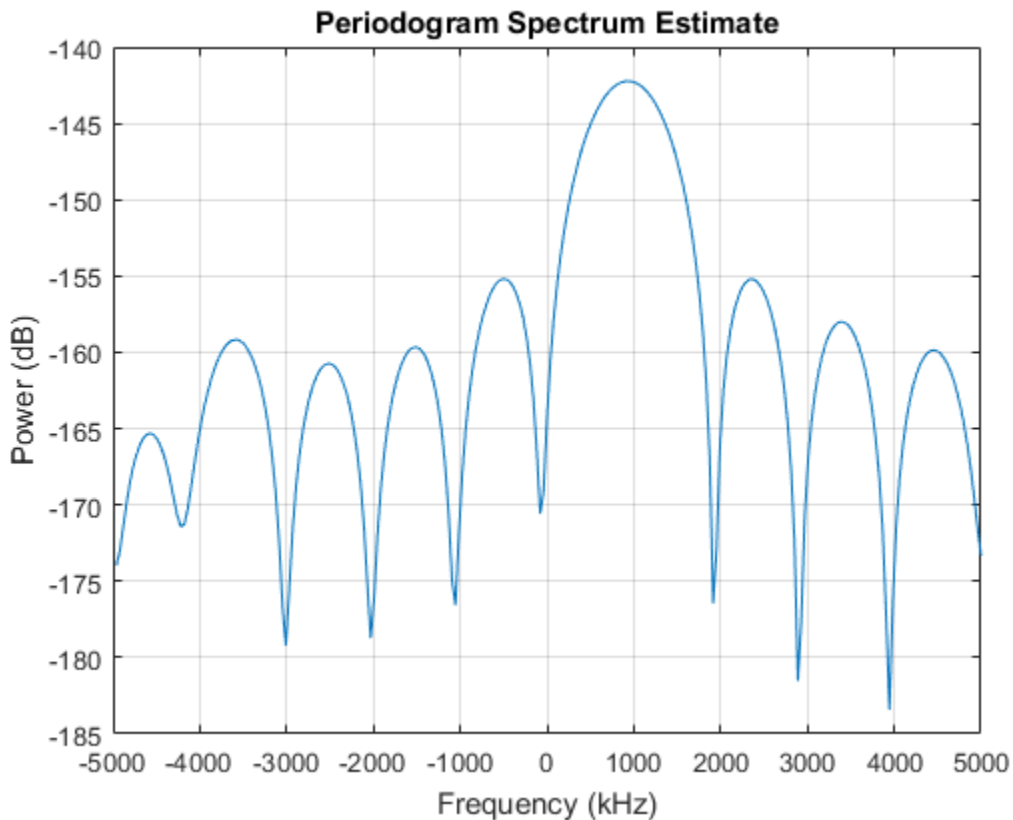

```
[Y,I] = max(Pxx);
lambda = physconst('LightSpeed')/1e9;
tgtspeed = dop2speed(F(I)/2,lambda);
fprintf('Estimated range of the target is %4.2f meters.\n',...
    range_estimate)
fprintf('Estimated target speed is %3.1f m/sec.\n',tgtspeed)
if F(I)>0
    fprintf('The target is approaching the radar.\n')
else
    fprintf('The target is moving away from the radar.\n')
end

Estimated range of the target is 1439.00 meters.
Estimated target speed is 140.5 m/sec.
```

```
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.

Related Examples

- "Doppler Estimation"
- "Scan Radar Using a Uniform Rectangular Array"

Using Polarization

Polarized Fields

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.

$$\mathbf{E} = E_x \hat{\mathbf{e}}_x + E_y \hat{\mathbf{e}}_y$$

The unit vectors together with the unit vector in direction of propagation

$$\{\breve{\mathbf{e}}_x, \breve{\mathbf{e}}_y, \mathbf{s}\}.$$

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. In radar systems, it is common to use the subscripts, H and V, denoting the horizontal and vertical components, instead of x and y. 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

$$\mathbf{E} = E_{x0} \cos\left(\omega t - \mathbf{k} \cdot \mathbf{x} + \phi_x\right) \hat{\mathbf{e}}_x + E_{y0} \cos\left(\omega t - \mathbf{k} \cdot \mathbf{x} + \phi_y\right) \hat{\mathbf{e}}_y = E_x \hat{\mathbf{e}}_x + E_y \hat{\mathbf{e}}_y$$

The quantities E_{x0} and E_{y0} are the real-valued, non-negative, amplitudes of the components of the electric field and ϕ_x and ϕ_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

$$\mathbf{E} = E_{x0}\cos(\omega t + \phi_x)\hat{\mathbf{e}}_x + E_{y0}\cos(\omega t + \phi_y)\hat{\mathbf{e}}_y = E_x\hat{\mathbf{e}}_x + E_y\hat{\mathbf{e}}_y$$

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 $\pi/2$ in the phase. You can see this dependency by examining the motion of the tip of the vector field. Assume the common phase angle, $\phi = 0$. 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 $+\pi/2$ 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.
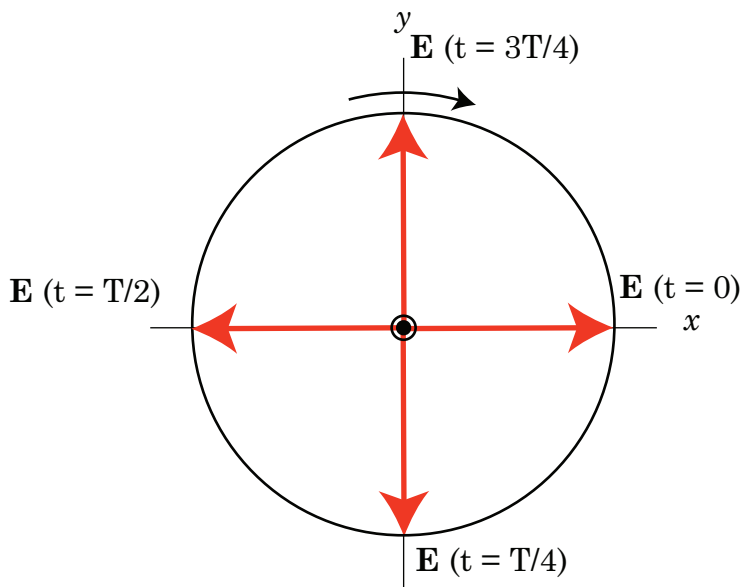
Left hand circular polarization. The direction of the electric vector at 0, 1/4, 1/2, and 3/4 periods, T. The z-axis points out of the page.

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 $-\pi/2$ 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.
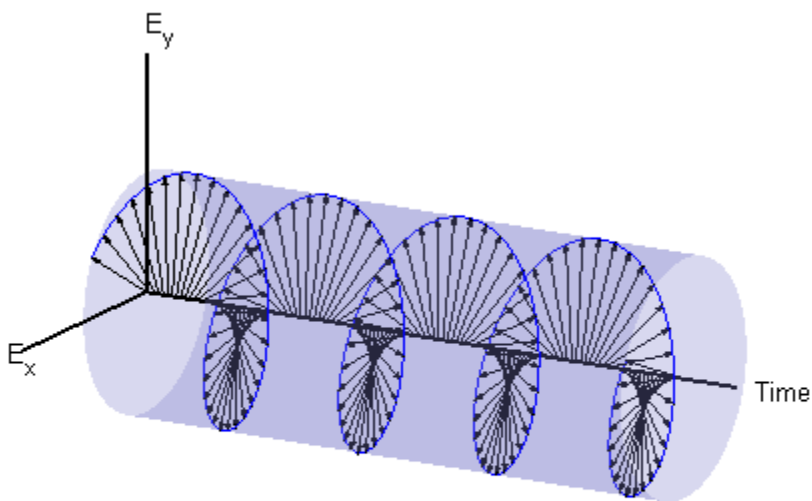
Left-Handed Circular Polarization

The figure below summarizes the appearance of linear and circularly polarized fields as they move towards you along the *s*-direction.
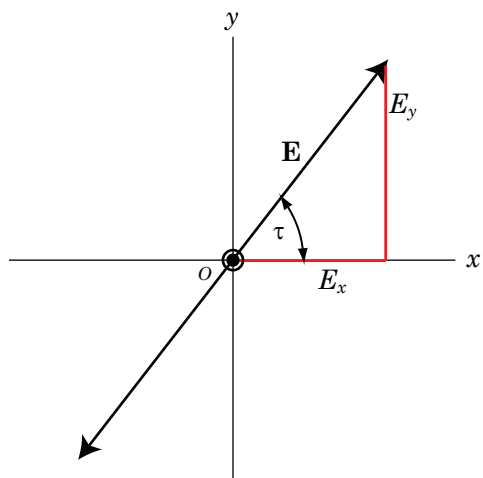
Linear polarization with positive slope
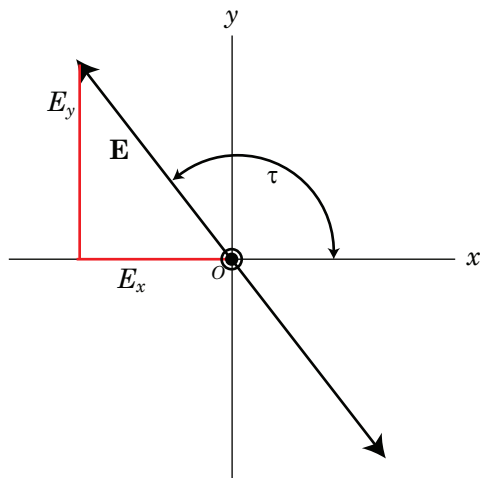
Linear polarization with negative slope
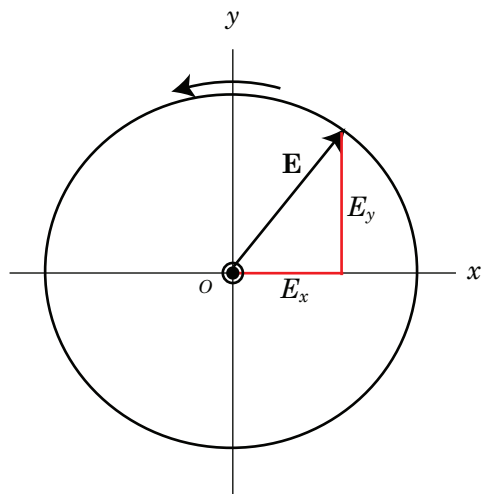
Right hand circular polarization
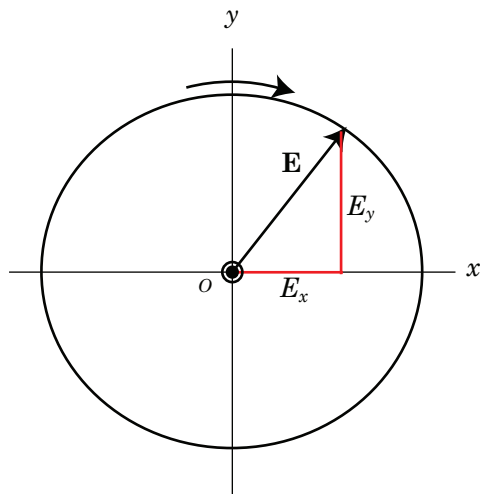
Left hand circular polarization

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, τ. The following figure illustrates the three parameters of a tilted ellipse. You can derive them from the two electric field amplitudes and phase difference.
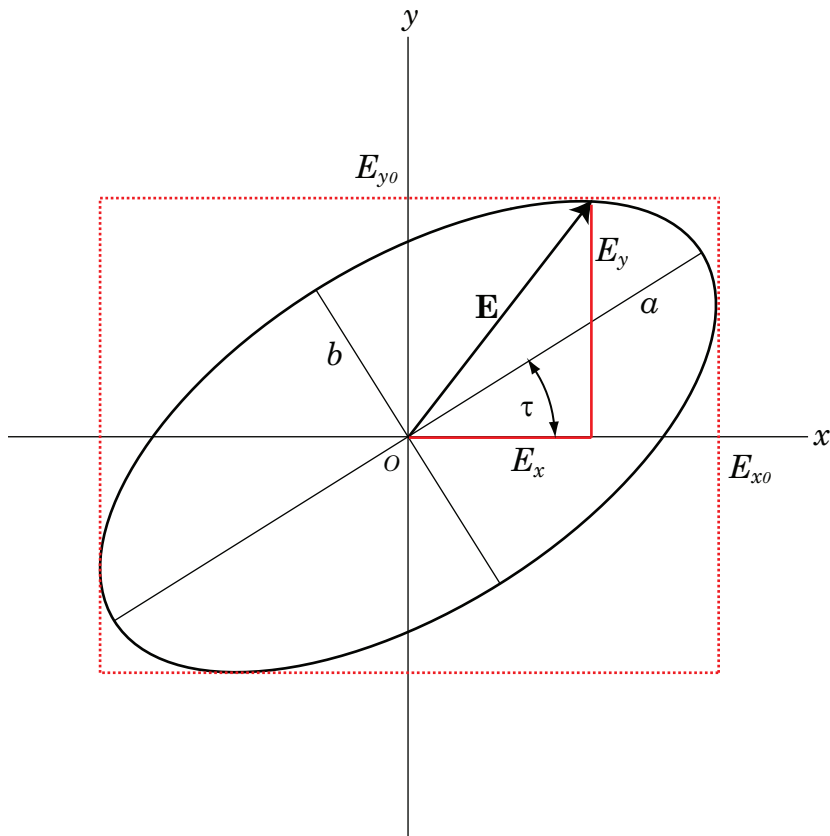
Polarization Ellipse

Polarization can best be understood in terms of complex signals. The complex representation of a polarized wave has the form

$$\mathbf{E} = E_{x0}e^{i\phi_x}e^{i\omega t}\,\hat{\mathbf{e}}_x + E_{y0}e^{i\phi_y}e^{i\omega t}\,\hat{\mathbf{e}}_y = \left(E_{x0}e^{i\phi_x}\,\hat{\mathbf{e}}_x + E_{y0}e^{i\phi_y}\,\hat{\mathbf{e}}_y\right)e^{i\omega t}$$

Define the complex *polarization ratio* as the ratio of the complex amplitudes

$$\rho = \frac{E_{y0}}{E_{x0}}e^{i\left(\phi_y - \phi_x\right)} = \frac{E_{y0}}{E_{x0}}e^{i\phi}$$

where $\phi = \phi_y - \phi_x$.

It is useful to introduce the *polarization vector*. For the complex polarized electric field above, the polarization vector, $\mathbf{P}$, is obtained by normalizing the electric field

$$\mathbf{P} = \frac{E_{x0}}{E_m}\,\hat{\mathbf{e}}_x + \frac{E_{y0}}{E_m}\,e^{i\left(\phi_y - \phi_x\right)}\hat{\mathbf{e}}_y = \frac{E_{x0}}{E_m}\,\hat{\mathbf{e}}_x + \frac{E_{y0}}{E_m}\,e^{i\phi}\,\hat{\mathbf{e}}_y$$

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*, τ. 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*, ε.

In the Phased Array System Toolbox software, you can use the `polratio` function to convert the complex amplitudes `fv=[Ey;Ex]` to the polarization ratio.

```
p = polratio(fv)
```

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, τ, needs 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

$$\tan 2\tau = \frac{2E_{x0}E_{y0}}{E_{x0}^2 - E_{y0}^2}\cos\phi$$

where $\varphi = \varphi_y - \varphi_x$. 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 *x*- and *y*-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 $-\pi/4 \leq \tau \leq \pi/4$.

If you first introduced the *auxilliary angle*, *a*, by

$$\tan \alpha = \frac{E_{y0}}{E_{x0}}$$

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=[Ey;Ex]` 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_x, E_y.	$\varphi_y = \varphi_x$	Any non-negative real number
Linear negative slope	Any non-negative real values for E_x, E_y	$\varphi_y = \varphi_x + \pi$	Any negative real number
Right-Handed Circular	$E_x = E_y$	$\varphi_y = \varphi_x - \pi/2$	$-i$
Left-Handed Circular	$E_x = E_y$	$\varphi_y = \varphi_x + \pi/2$	i
Right-Handed Elliptical	Any non-negative real values for E_x, E_y	$sin(\varphi_y - \varphi_x) < 0$	$sin(arg\ \rho) < 0$
Left-Handed Elliptical	Any non-negative real values for E_x, E_y	$sin(\varphi_y - \varphi_x) > 0$	$sin(arg\ \rho) > 0$

Linear and Circular Polarization Bases

As shown earlier, you can express a polarized electric field as a linear combination of basis vectors along the x and y 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:

$$\mathbf{E} = \text{Re}[E_0(\mathbf{e}_x \mp i\mathbf{e}_y)e^{i(\omega t + \phi)}]$$

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

$$\mathbf{e}_r = \frac{1}{\sqrt{2}}(\mathbf{e}_x - i\mathbf{e}_y)$$

$$\mathbf{e}_l = \frac{1}{\sqrt{2}}(\mathbf{e}_x + i\mathbf{e}_y)$$

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

$$\mathbf{e}_x = \frac{1}{\sqrt{2}}(\mathbf{e}_r + \mathbf{e}_l)$$

$$\mathbf{e}_y = \frac{1}{\sqrt{2}i}(\mathbf{e}_r - \mathbf{e}_l)$$

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

$$
\begin{aligned}
S_0 \quad &= E_{x0}^2 + E_{y0}^2 \\
S_1 \quad &= E_{x0}^2 - E_{y0}^2 \\
S_2 \quad &= 2 E_{x0} E_{y0} \cos\phi \\
S_3 \quad &= 2 E_{x0} E_{y0} \sin\phi
\end{aligned}
$$

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, τ and ε

$$
\begin{aligned}
S_1 \quad &= S_0 \cos 2\tau \cos 2\varepsilon \\
S_2 \quad &= S_0 \sin 2\tau \cos 2\varepsilon \\
S_3 \quad &= S_0 \sin 2\varepsilon
\end{aligned}
$$

and inversely by

$$
\begin{aligned}
\tan 2\tau \quad &= \frac{S_2}{S_1} \\[2mm]
\sin 2\varepsilon \quad &= \frac{S_3}{S_0}
\end{aligned}
$$

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, ε. 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, τ.
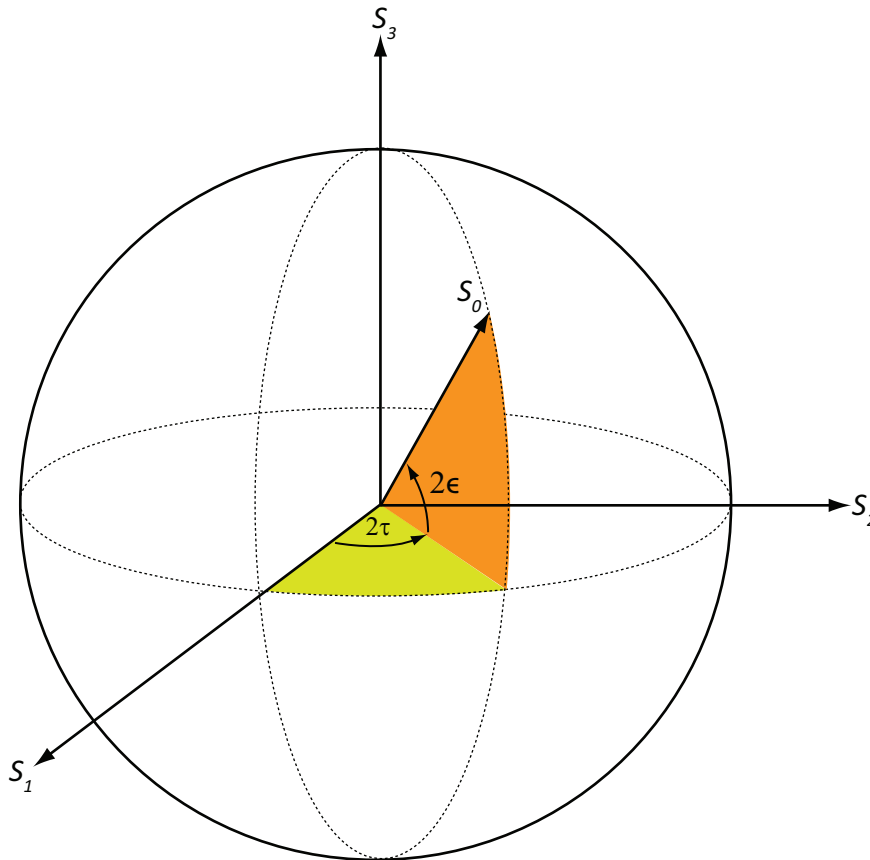
As an example, solve for the Stokes parameters of a RHCP field, `fv=[1,-i]`, using the `stokes` function.

```
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 10-21 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, $\{\hat{\mathbf{e}}_H, \hat{\mathbf{e}}_V, \hat{\mathbf{r}}\}$. 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

$\hat{\mathbf{r}}$. 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

$$\mathbf{E} = \mathbf{F}(\phi, \theta) \frac{e^{ikr}}{r} = \left(F_H(\phi, \theta) \mathbf{e}_H + F_V(\phi, \theta) \mathbf{e}_V \right) \frac{e^{ikr}}{r}$$

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.

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

$$E_r = 0$$
$$E_H = 0$$
$$E_V = -\frac{iZ_0 IL}{2\lambda}\cos \text{el}\,\frac{e^{-ikr}}{r}$$

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°.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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)
polar(el*pi/180.0,abs(resp.V.'),'b')
hold on
polar((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)
polar(el*pi/180.0,abs(resp.H.'),'b')
hold on
polar((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-handed and left-handed circular polarization components of fields generated by a crossed-dipole antenna at 1.5 GHz. You can see how the circular polarization chnages from pure RHCP at 0° azimuth angle to pure LHCP at 180° azimuth angle, both at 0° elevation angle.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Create the `phased.CrossedDipoleAntennaElement` System 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° elevation.

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

LHCP and RHCP vs Azithmuth Angle

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 (τ, ε). 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
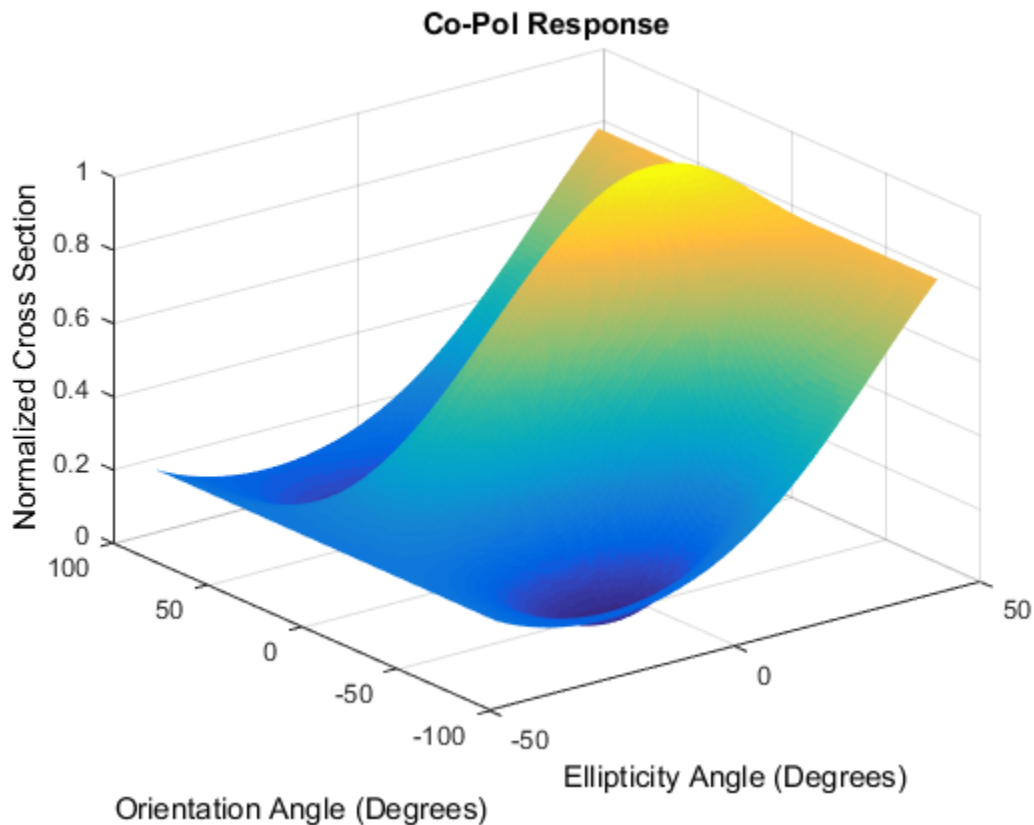
Plot the cross-polarizations signatures for all incident polarizations.

```
polsignature(rscmat,'x',el,tilt)
```

Cross-Pol Response
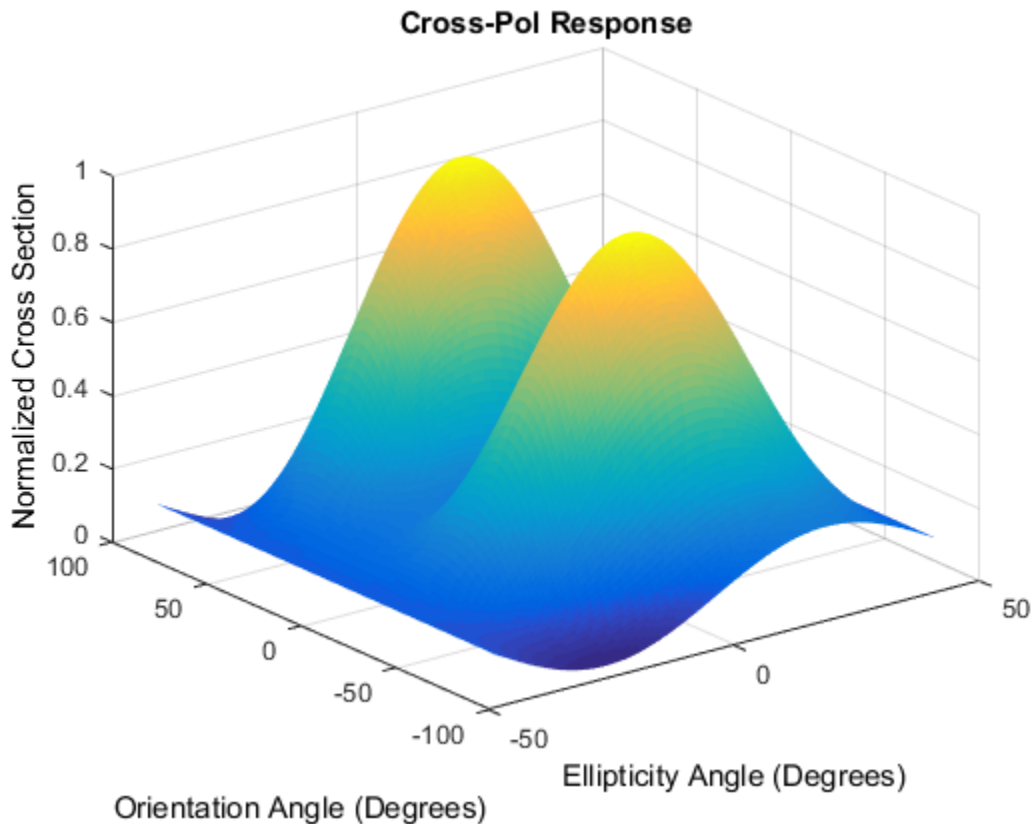
Polarization Loss Due to Field and Receiver Mismatch

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

$$\mathbf{E} = E_{iH}\hat{\mathbf{e}}_H + E_{iV}\hat{\mathbf{e}}_V = E_m\mathbf{P}_i$$

You can represent the receiver's polarization vector, (P_H, P_V), in the receiver's local spherical basis by:

$$\mathbf{P} = P_H\hat{\mathbf{e}}'_H + P_V\hat{\mathbf{e}}'_V$$

The next figure shows the construction of the transmitter and receiver spherical basis vectors.
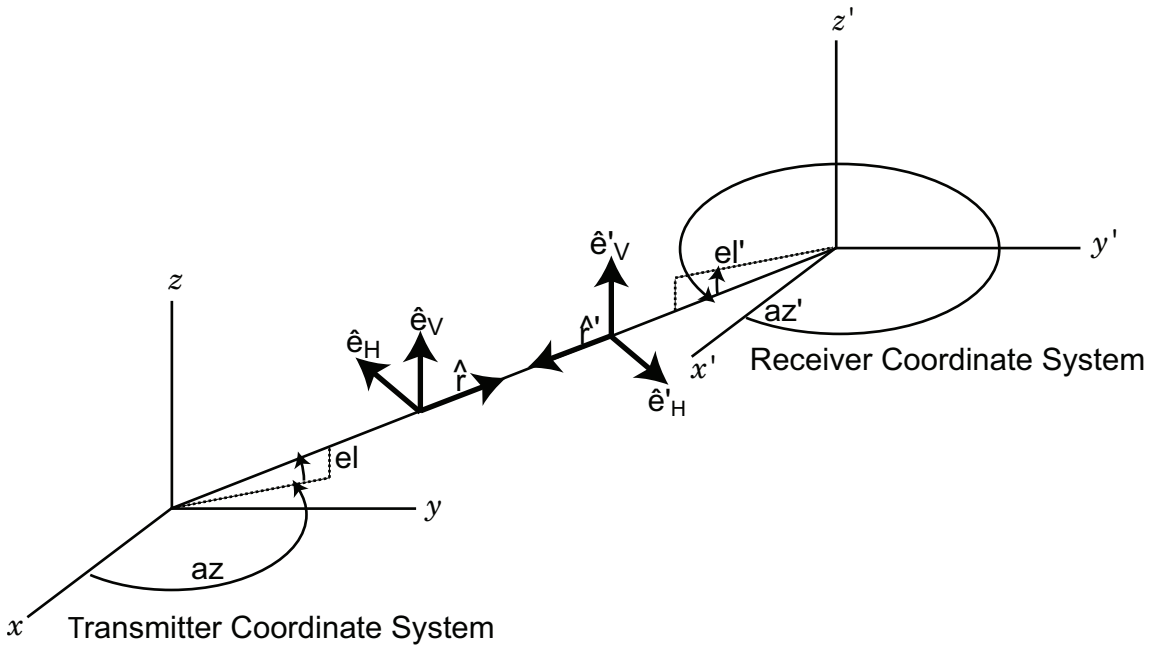
The polarization loss is defined by:

$$\rho = \frac{|\mathbf{E}_i \cdot \mathbf{P}|^2}{|\mathbf{E}_i|^2|\mathbf{P}|^2}$$

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 is 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	∞
LHCP	$\mathbf{e}_l = \frac{1}{\sqrt{2}} (\mathbf{e}_x + i\mathbf{e}_y)$	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.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

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 you select,|range_res|. 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 requirment.

```
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 alont 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');
```

3D Response Pattern
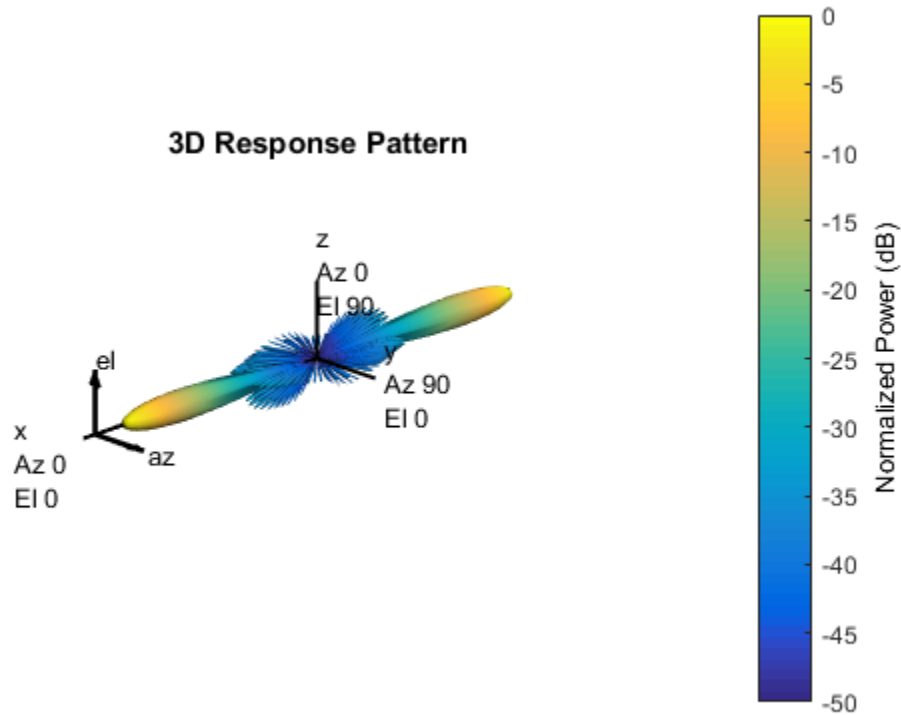
Radar Platform Motion

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 `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, `phased.Radiator`, System object and two collector, `phased.Collector, System objects. 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,...
    'EnablePolarization',true,'WeightsInputPort',true);
collector1 = phased.Collector('Sensor',array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'Wavefront','Plane','EnablePolarization',true,...
    'WeightsInputPort',false);
collector2 = phased.Collector('Sensor',array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'Wavefront','Plane','EnablePolarization',true,...
    '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 <function `albersheim` function. Then, compute the peak power from the radar equation using the `radareqpow` function. Among the inputs to this function are the overall signal gain, which is the sum of the transmitting element gain, `TransmitterGain` and the array gain, `AG`. Another input is 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.

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 for the radar equation even though the analysis calls for the full scattering matrix.

11-35

- 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;
peak_power = radareqpow(lambda,rangegate,snr_min,waveform.PulseWidth,...
    'RCS',tgt_rcs,'Gain',TransmitterGain + AG);
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/sec.

```
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
% 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
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
```
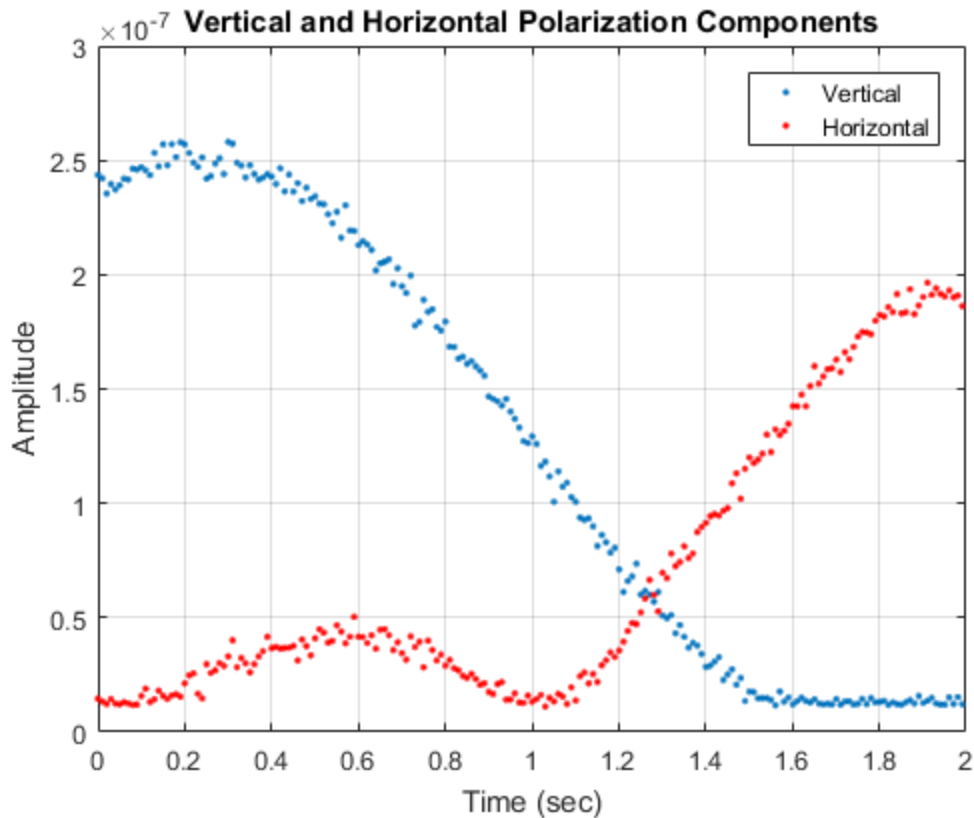
Antenna and Array Definitions

Element and Array Radiation and Response Patterns

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. The electromagnetic fields created by an antenna, or the acoustic field created by a transducer (called a speaker in speech acoustics or hydrophone in ocean acoustics), depend on the distance from the sources and the direction specified by angular coordinates. 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

$r \gg \lambda^2/L$

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. We shall often add the term "field" or "power" to be more specific: contrast element field pattern versus element power pattern. The radiation power pattern describes the field's radiant intensity as a function of direction. Power units are watts/steradian.

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 may 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

$$Af(\theta,\phi)\frac{e^{-ikr}}{r}$$

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's `step` method are represented simply as $A\ f(\theta,\varphi)$. You can display the nominal element field pattern by invoking the element's `pattern` method, choosing `'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, its normalized field pattern is given by $|E_H(\theta,\varphi)|/E_{H,max}|$.

```
pattern(elem,'Polarization','H','Normalize', true,'Unit''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 (θ,φ) 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 given by

$$U_{rad}(\theta,\phi) = \frac{r^2}{2Z_0}\left(|E_H|^2 + |E_V|^2\right)$$

where Z_0 is the characteristic impedance of free space. The radiant intensity of an acoustic field is given by

$$U_{rad}(\theta,\phi) = \frac{r^2}{2Z} \mid p \mid^2$$

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 field magnitudes are all collected in the nominal field amplitudes defined above. Then the radiant intensity can generally be written

$$U_{rad}(\theta,\phi) = \mid Af(\theta,\phi) \mid^2$$

The radiant intensity pattern is the quantity returned by the elements `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

$$U_{norm}(\theta,\phi) = \frac{U_{rad}(\theta,\phi)}{U_{rad,max}} = \mid f(\theta,\phi) \mid^2$$

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 radiant intensity transmitted by an isotropic radiator with 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 given by

$$D(\theta,\phi) = \frac{3}{2}\cos^2\theta$$

Often, the largest value of $D(\theta,\varphi)$ is specified as an antenna operating parameter. The direction in which $D(\theta,\varphi)$ 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 that for 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 actually radiated into space. Then, $P_{rad} = \eta P_{total}$. The element gain is defined by

$$G(\theta,\phi) = 4\pi \frac{U_{rad}(\theta,\phi)}{P_{total}} = 4\pi\eta \frac{U_{rad}(\theta,\phi)}{P_{rad}} = \eta D(\theta,\phi)$$

and represents the power radiated away from the element compared to the total power supplied to the element.

Using the element's `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 may be constructed to have a very narrow angular distribution, i.e. 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 an plane incident wave. By reciprocity, the response pattern is identical to the radiation pattern.

For transmitting arrays, the voltage driving the elements may be phase-adjusted to allow the maximum radiant intensity to be transmitted in a particular direction. For receiving arrays, the arriving signals may 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 may 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 mth 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

$$\mathbf{s}(\theta,\phi) = \{\ldots,e^{ik\mathbf{u}\cdot\mathbf{x}_m},\ldots\}$$

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

$$|\, Af(\theta,\phi)\mathbf{w}^H\mathbf{s}\,|^2 = |\, Af(\theta,\phi)F_{array}(\theta,\phi)\,|^2 = |\, AB(\theta,\phi)\,|^2$$

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 $\boldsymbol{u_0}$ is the desired pointing direction, then the weights which maximize the power and response in this direction have the general form

$$\mathbf{w} = |\, w_m\,|\, e^{-ik\mathbf{u}_0 \cdot \mathbf{x}_m}$$

For these weights, the array factor becomes

$$F_{array}(\theta,\phi) = \sum_{m=0}^{M-1} |w_m|\, e^{-ik(\mathbf{u}-\mathbf{u}_0)\cdot\mathbf{x}_m}$$

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's total radiated power divided by *4π*. In terms of the arrays weights and steering vectors, the directivity can be written as

$$D(\theta,\phi) = 4\pi \frac{|Af(\theta,\phi)\mathbf{w}^H\mathbf{s}|^2}{P_{total}}$$

where P_{total} is the total radiated power from the array. In a discrete implementation, the total radiated power can be computed by summing intensity values over a uniform grid of angles that covers the full sphere surrounding the array

$$P_{total} = \frac{2\pi^2}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |Af(\theta_m,\phi_n)\mathbf{w}^H\mathbf{s}(\theta_m,\phi_n)|^2 \cos\theta_m$$

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(θ,φ)*, the directivity can also be written in terms of the beampattern

$$D(\theta,\phi) = 4\pi \frac{|B(\theta,\phi)|^2}{\int |B(\theta,\phi)|^2 \cos\theta d\theta d\phi}$$

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

$$G = \frac{\text{SNR}_{array}}{\text{SNR}_{element}} = \frac{|\mathbf{w}^H \mathbf{s}|^2}{\mathbf{w}^H \mathbf{w}}$$

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);
```

Grating Lobe Diagram in U-V Space
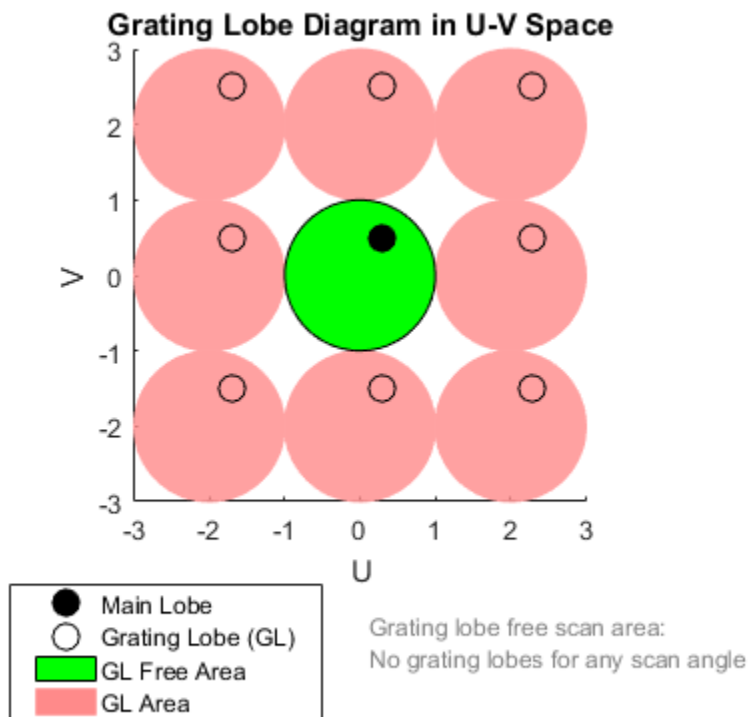
Plot the grating lobes. The main lobe of the array is indicated by a filled black circle. The grating lobes in visible and nonvisible regions are indicated by unfilled black circles. The visible region is the region in u-v coordinates for which $u^2 + v^2 \leq 1$. The visible region is shown as a unit 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.

13

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 "Getting Started with 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's 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 "C/C++ Code Generation" 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". 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 "Functions and System Objects Supported for C/C++ Code Generation" on page 13-19.

Limitations Specific to Phased Array System Toolbox

Code Generation has the following limitations when used with the Phased Array System Toolbox software:

- Heterogeneous array, phased.HeterogeneousConformalArray, phased.HeterogeneousConformalArray, and phased.HeterogeneousConformalArray, are not supported.

- 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`.

```matlab
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 = step(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

```
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
sURA = phased.URA('Element',sSD,...
    'Size',[3,3],...
    'ElementSpacing',[0.75,0.75]);
```

is valid for `codegen` but

```
sSD = phased.ShortDipoleAntennaElement(...
    'FrequencyRange',[100e6,600e6],'AxisDirection','Y');
sURA = phased.URA('Size',[3,3],...
    'ElementSpacing',[0.75,0.75]);
sURA.Element = sSD;
```

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.

Function or System Object	Limitation
ambgfun	Supported only when output arguments are specified.
blakechart	This function is not supported.
pambgfun	Supported only when output arguments are specified.
polsignature	Supported only when output arguments are specified.
rocpfa	• Supported only when output arguments are specified. • The NonfluctuatingNoncoherent signal type is not supported.
rocsnr	• Supported only when output arguments are specified. • 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.HeterogeneousConformalArray	This System object is not supported.
phased.HeterogeneousULA	This System object is not supported.
phased.HeterogeneousURA	This System object is not supported.
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".

- 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 = step(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 = step(sURA,fc,ang);
```

and then compile

```
codegen codegen_const2
```

- You can assign a nontunable System object property value only once before a `step` method is executed. This requirement differs from MATLAB usage where you can initialize these properties multiple times before the `step` method is executed.

  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]';
sIso = phased.IsotropicAntennaElement('FrequencyRange',[100e6,300e6]);
sULA = phased.ULA('Element',sIso,'NumElements',10,'ElementSpacing',lambda/2);
pos = getElementPosition(sULA)/lambda;
sig = sensorsig(pos,nsamp,ang,sigma^2);
sDOA = phased.RootMUSICEstimator('SensorArray',sULA,...
    'OperatingFrequency',fc,...
    'NumSignalsSource','Property','NumSignals',2);
doas = step(sDOA,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]';
sIso = phased.IsotropicAntennaElement('FrequencyRange',[100e6,300e6]);
sULA = phased.ULA('Element',sIso,'NumElements',10,'ElementSpacing',lambda/2);
pos = getElementPosition(sULA)/lambda;
sig = sensorsig(pos,nsamp,ang,sigma^2);
sDOA = phased.RootMUSICEstimator('SensorArray',sULA,...
    'OperatingFrequency',fc,...
    'NumSignalsSource','Property','NumSignals',2);
doas = step(sDOA,sig);
az = broadside2az(sort(doas),[20,60]);
```

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:

```matlab
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 = step(sAR,fc,[az;el]);
end
```

To create the code, run codegento create the mex-file plot_ULA_response_mex, and execute the mex-file at the command line:

```
codegen plot_ULA_response
plot_ULA_response_mex;
```

which yields the plot
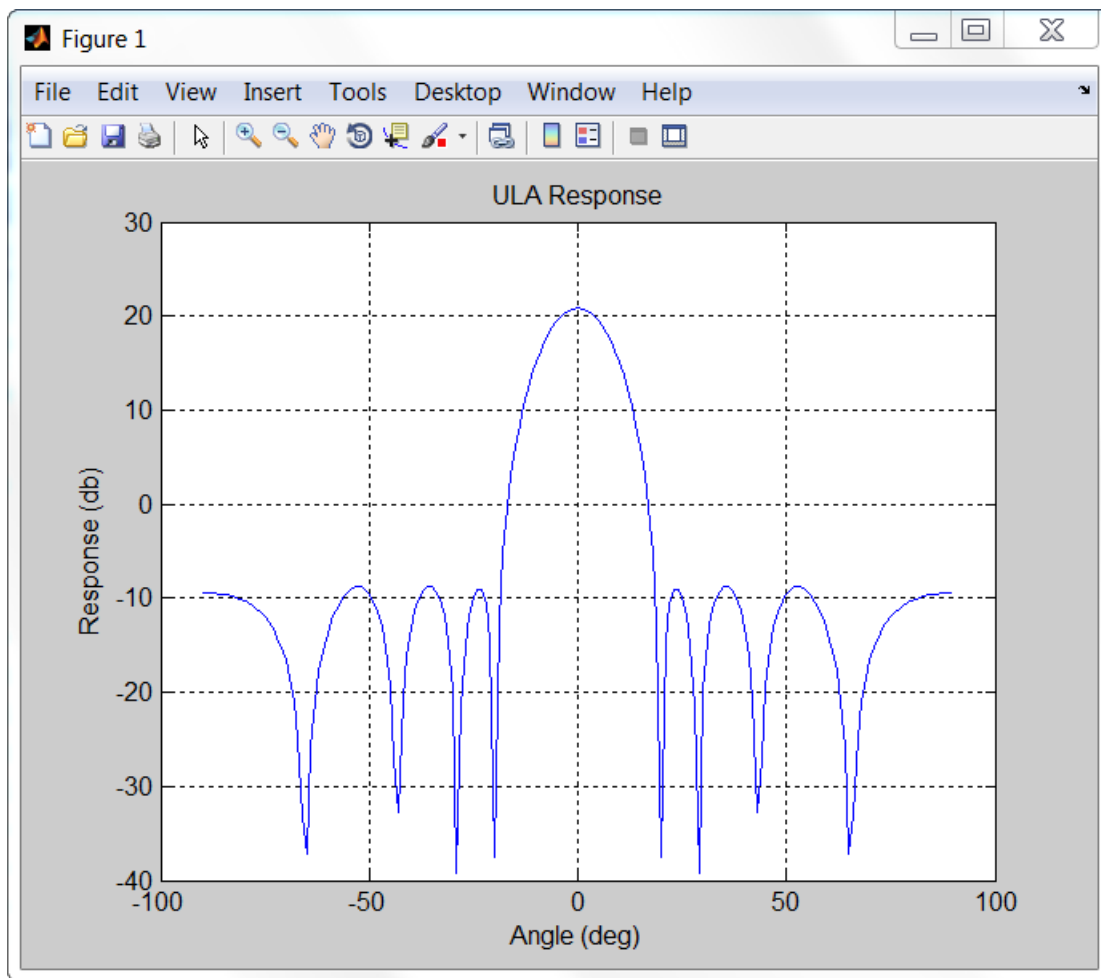
Functions and System Objects Supported for C/C++ Code Generation

Name	Remarks and Limitations
Antenna and Microphone Elements	
`aperture2gain`	Does not support variable-size inputs.
`azel2phithetapat`	Does not support variable-size inputs.
`azel2uvpat`	Does not support variable-size inputs.
`circpol2pol`	Does not support variable-size inputs.
`gain2aperture`	Does not support variable-size inputs.
phased.CosineAntennaElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.CrossedDipoleAntennaElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.CustomAntennaElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.CustomMicrophoneElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.IsotropicAntennaElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported.

Name	Remarks and Limitations
	• See "System Objects in MATLAB Code Generation".
phased.OmnidirectionalMicrophoneElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported.
	• See "System Objects in MATLAB Code Generation".
phased.ShortDipoleAntennaElement	• `pattern`, `patternAzimuth`, `patternElevation`, and `plotResponse` methods are not supported.
	• See "System Objects in MATLAB Code Generation".
`phitheta2azelpat`	Does not support variable-size inputs.
`phitheta2uvpat`	Does not support variable-size inputs.
`pol2circpol`	Does not support variable-size inputs.
`polellip`	Does not support variable-size inputs.
`polloss`	Does not support variable-size inputs.
`polratio`	Does not support variable-size inputs.
`polsignature`	• Does not support variable-size inputs. • Supported only when output arguments are specified.
`stokes`	• Does not support variable-size inputs. • Supported only when output arguments are specified.
`uv2azelpat`	Does not support variable-size inputs.
`uv2phithetapat`	Does not support variable-size inputs.
Array Geometries and Analysis	
`az2broadside`	Does not support variable-size inputs.
`broadside2az`	Does not support variable-size inputs.
`pilotcalib`	Does not support variable-size inputs.

Name	Remarks and Limitations
phased.ArrayGain	• Does not support arrays containing polarized antenna elements, that is, the `phased.ShortDipoleAntennaElement` or `phased.CrossedDipoleAntennaElement` antennas. • See "System Objects in MATLAB Code Generation".
phased.ArrayResponse	See "System Objects in MATLAB Code Generation".
phased.ConformalArray	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.ElementDelay	See "System Objects in MATLAB Code Generation".
phased.PartitionedArray	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.ReplicatedSubarray	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.SteeringVector	See "System Objects in MATLAB Code Generation".
phased.UCA	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".

Name	Remarks and Limitations
phased.ULA	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".
phased.URA	• `pattern`, `patternAzimuth`, `patternElevation`, `plotResponse`, and `viewArray` methods are not supported. • See "System Objects in MATLAB Code Generation".
`taylortaperc`	Does not support variable-size inputs.
Signal Radiation and Collection	
phased.Collector	See "System Objects in MATLAB Code Generation".
phased.Radiator	See "System Objects in MATLAB Code Generation".
phased.WidebandCollector	• Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation" on page 13-12. • See "System Objects in MATLAB Code Generation".
phased.WidebandRadiator	See "System Objects in MATLAB Code Generation".
`sensorsig`	Does not support variable-size inputs.
Transmitters and Receivers	
`delayseq`	Does not support variable-size inputs.
`noisepow`	Does not support variable-size inputs.
phased.ReceiverPreamp	See "System Objects in MATLAB Code Generation".
phased.Transmitter	See "System Objects in MATLAB Code Generation".
`systemp`	Does not support variable-size inputs.

Name	Remarks and Limitations
Waveform Design and Analysis	
ambgfun	• Does not support variable-size inputs. • Supported only when output arguments are specified.
pambgfun	• Does not support variable-size inputs. • Supported only when output arguments are specified.
phased.FMCWWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation". .
phased.LinearFMWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation".
phased.MFSKWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation".
phased.PhaseCodedWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation".
phased.RectangularWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation".
phased.SteppedFMWaveform	• `plot` method is not supported. • See "System Objects in MATLAB Code Generation".
range2bw	Does not support variable-size inputs.
range2time	Does not support variable-size inputs.
time2range	Does not support variable-size inputs.
unigrid	Does not support variable-size inputs.

Name	Remarks and Limitations
Beamforming	
cbfweights	Does not support variable-size inputs.
lcmvweights	Does not support variable-size inputs.
mvdrweights	Does not support variable-size inputs.
phased.FrostBeamformer	• Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation" on page 13-12. • See "System Objects in MATLAB Code Generation".
phased.GSCBeamformer	See "System Objects in MATLAB Code Generation".
phased.LCMVBeamformer	See "System Objects in MATLAB Code Generation".
phased.MVDRBeamformer	See "System Objects in MATLAB Code Generation".
phased.PhaseShiftBeamformer	See "System Objects in MATLAB Code Generation".
phased.SteeringVector	See "System Objects in MATLAB Code Generation".
phased.SubbandMVDRBeamformer	See "System Objects in MATLAB Code Generation".
phased.SubbandPhaseShiftBeamformer	See "System Objects in MATLAB Code Generation".
phased.TimeDelayBeamformer	• Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation" on page 13-12. • See "System Objects in MATLAB Code Generation".

Name	Remarks and Limitations
phased.TimeDelayLCMVBeamformer	• Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation" on page 13-12. • See "System Objects in MATLAB Code Generation".
sensorcov	Does not support variable-size inputs.
steervec	Does not support variable-size inputs.
Direction of Arrival (DOA) Estimation	
aictest	Does not support variable-size inputs.
espritdoa	Does not support variable-size inputs.
gccphat	Does not support variable-size inputs.
mdltest	Does not support variable-size inputs.
musicdoa	Does not support variable-size inputs.
phased.BeamscanEstimator	See "System Objects in MATLAB Code Generation".
phased.BeamscanEstimator2D	See "System Objects in MATLAB Code Generation".
phased.BeamspaceESPRITEstimator	See "System Objects in MATLAB Code Generation".
phased.ESPRITEstimator	See "System Objects in MATLAB Code Generation".
phased.GCCEstimator	See "System Objects in MATLAB Code Generation".
phased.MUSICEstimator	See "System Objects in MATLAB Code Generation".
phased.MUSICEstimator2D	See "System Objects in MATLAB Code Generation".
phased.MVDREstimator	See "System Objects in MATLAB Code Generation".
phased.MVDREstimator2D	See "System Objects in MATLAB Code Generation".

Name	Remarks and Limitations
phased.RootMUSICEstimator	See "System Objects in MATLAB Code Generation".
phased.RootWSFEstimator	See "System Objects in MATLAB Code Generation".
phased.SumDifferenceMonopulseTracker	See "System Objects in MATLAB Code Generation".
phased.SumDifferenceMonopulseTracker2D	See "System Objects in MATLAB Code Generation".
rootmusicdoa	Does not support variable-size inputs.
spsmooth	Does not support variable-size inputs.
Space-Time Adaptive Processing (STAP)	
dopsteeringvec	Does not support variable-size inputs.
phased.ADPCACanceller	See "System Objects in MATLAB Code Generation".
phased.AngleDopplerResponse	See "System Objects in MATLAB Code Generation".
phased.DPCACanceller	See "System Objects in MATLAB Code Generation".
phased.STAPSMIBeamformer	See "System Objects in MATLAB Code Generation".
val2ind	Does not support variable-size inputs.
Detection, Range, and Doppler Estimation	
albersheim	Does not support variable-size inputs.
beat2range	Does not support variable-size inputs.
dechirp	Does not support variable-size inputs.
npwgnthresh	Does not support variable-size inputs.
phased.CFARDetector	See "System Objects in MATLAB Code Generation".
phased.CFARDetector2D	See "System Objects in MATLAB Code Generation".

Name	Remarks and Limitations
phased.MatchedFilter	• The `CustomSpectrumWindow` property is not supported. • See "System Objects in MATLAB Code Generation".
phased.RangeDopplerResponse	• The `CustomRangeWindow` and `CustomDopplerWindow` properties are not supported. • The `plotResponse` method is not supported. • See "System Objects in MATLAB Code Generation".
phased.StretchProcessor	See "System Objects in MATLAB Code Generation".
phased.TimeVaryingGain	See "System Objects in MATLAB Code Generation".
pulsint	Does not support variable-size inputs.
radareqpow	Does not support variable-size inputs.
radareqrng	Does not support variable-size inputs.
radareqsnr	Does not support variable-size inputs.
radarvcd	• Does not support variable-size inputs. • Supported only when output arguments are specified.
range2beat	Does not support variable-size inputs.
rdcoupling	Does not support variable-size inputs.
rocpfa	• Does not support variable-size inputs. • Supported only when output arguments are specified. • The `NonfluctuatingNoncoherent` signal type is not supported.

Name	Remarks and Limitations
rocsnr	• Does not support variable-size inputs. • Supported only when output arguments are specified. • The NonfluctuatingNoncoherent signal type is not supported.
shnidman	Does not support variable-size inputs.
stretchfreq2rng	Does not support variable-size inputs.
Targets, Interference, and Signal Propagation	
billingsleyicm	Does not support variable-size inputs.
depressionang	Does not support variable-size inputs.
effearthradius	Does not support variable-size inputs.
fspl	Does not support variable-size inputs.
fogpl	Does not support variable-size inputs.
gaspl	Does not support variable-size inputs.
grazingang	Does not support variable-size inputs.
horizonrange	Does not support variable-size inputs.
phased.BackscatterRadarTarget	See "System Objects in MATLAB Code Generation"
phased.BarrageJammer	See "System Objects in MATLAB Code Generation".
phased.ConstantGammaClutter	See "System Objects in MATLAB Code Generation".
phased.FreeSpace	• Requires dynamic memory allocation. See "Limitations for System Objects that Require Dynamic Memory Allocation" on page 13-12. • See "System Objects in MATLAB Code Generation".
phased.LOSChannel	See "System Objects in MATLAB Code Generation"

Name	Remarks and Limitations
phased.RadarTarget	See "System Objects in MATLAB Code Generation".
phased.TwoRayChannel	See "System Objects in MATLAB Code Generation".
phased.WidebandFreeSpace	See "System Objects in MATLAB Code Generation".
phased.WidebandBackscatterRadarTarget	See "System Objects in MATLAB Code Generation".
phased.WidebandLOSChannel	See "System Objects in MATLAB Code Generation"
phased.WidebandTwoRayChannel	See "System Objects in MATLAB Code Generation"
physconst	Does not support variable-size inputs.
surfacegamma	Does not support variable-size inputs.
surfclutterrcs	Does not support variable-size inputs.
rainpl	Does not support variable-size inputs.
Motion Modeling and Coordinate Systems	
azel2phitheta	Does not support variable-size inputs.
azel2uv	Does not support variable-size inputs.
azelaxes	Does not support variable-size inputs.
cart2sphvec	Does not support variable-size inputs.
dop2speed	Does not support variable-size inputs.
global2localcoord	Does not support variable-size inputs.
local2globalcoord	Does not support variable-size inputs.
phased.Platform	See "System Objects in MATLAB Code Generation".
phitheta2azel	Does not support variable-size inputs.
phitheta2uv	Does not support variable-size inputs.
radialspeed	Does not support variable-size inputs.
rangeangle	Does not support variable-size inputs.

Name	Remarks and Limitations
`rotx`	Does not support variable-size inputs.
`roty`	Does not support variable-size inputs
`rotz`	Does not support variable-size inputs.
`speed2dop`	Does not support variable-size inputs.
`sph2cartvec`	Does not support variable-size inputs.
`uv2azel`	Does not support variable-size inputs.
`uv2phitheta`	Does not support variable-size inputs.

14

Define New System Objects

Define Basic System Objects

This example shows how to create a basic System object that increments a number by one. The class definition file used in the example contains the minimum elements required to define a System object.

Create System Object

You can create and edit a MAT-file or use the MATLAB Editor to create your System object. This example describes how to use the **New** menu in the MATLAB Editor.

In MATLAB, on the Editor tab, select **New** > **System Object** > **Basic**. A simple System object template opens.

Subclass your object from `matlab.System`. Replace `Untitled` with `AddOne` in the first line of your file.

```
classdef AddOne < matlab.System
```

Save the file and name it `AddOne.m`.

Define Algorithm

The `stepImpl` method contains the algorithm to execute when you run your object. Define this method so that it contains the actions you want the System object to perform.

1 In the basic System object you created, inspect the `stepImpl` method template.

```
methods (Access = protected)
    function y = stepImpl(obj,u)
        % Implement algorithm. Calculate y as a function of input u and
        % discrete states.
        y = u;
    end
end
```

The `stepImpl` method access is always set to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. The default value, inserted by MATLAB Editor, is `obj`. You can use any name for your System object handle.

By default, the number of inputs and outputs are both 1. Inputs and outputs can be added using **Inputs/Outputs**. If you use variable number of inputs or outputs, insert the appropriate `getNumInputsImpl` or `getNumOutputsImpl` method.

Alternatively, if you create your System object by editing a MAT-file, you can add the `stepImpl` method using **Insert Method** > **Implement algorithm**.

2 Change the computation in the y function to add 1 to the value of u.

```
methods (Access = protected)

    function y = stepImpl(~,u)
      y = u + 1;
    end
```

Note: Instead of passing in the object handle, you can use the tilde (~) to indicate that the object handle is not used in the function. Using the tilde instead of an object handle prevents warnings about unused variables.

3 Remove the additional, unused methods that are included by default in the basic template. Alternatively, you can modify these methods to add more System object actions and properties. You can also make no changes, and the System object still operates as intended.

The class definition file now has all the code necessary for this System object.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value one greater than the input value

  % All methods occur inside a methods declaration.
  % The stepImpl method has protected access
  methods (Access = protected)

    function y = stepImpl(~,u)
      y = u + 1;
    end
  end
end
```

See Also
matlab.System | getNumInputsImpl | getNumOutputsImpl | stepImpl

Related Examples

- "Change Number of Inputs or Outputs" on page 14-6

More About

- "System Design and Simulation in MATLAB"

Change Number of Inputs or Outputs

This example shows how to specify two inputs and two outputs to a System object .

If you specify the inputs and outputs to the `stepImpl` method, you do not need to specify the `getNumInputsImpl` and `getNumOutputsImpl` methods. If you have a variable number of inputs or outputs (using `varargin` or `varargout`), include the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively, in your class definition file.

Note: You should only use `getNumInputsImpl` or `getNumOutputsImpl` methods to change the number of System object inputs or outputs. Do not use any other handle objects within a System object to change the number of inputs or outputs.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to specify two inputs and two outputs. You do not need to implement associated `getNumInputsImpl` or `getNumOutputsImpl` methods.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(~,x1,x2)
      y1 = x1 + 1
      y2 = x2 + 1;
   end
end
```

Update the Algorithm and Associated Methods

Update the `stepImpl` method to use `varargin` and `varargout`. In this case, you must implement the associated `getNumInputsImpl` and `getNumOutputsImpl` methods to specify two or three inputs and outputs.

```
methods (Access = protected)
   function varargout = stepImpl(obj,varargin)
      varargout{1} = varargin{1}+1;
      varargout{2} = varargin{2}+1;
      if (obj.numInputsOutputs == 3)
         varargout{3} = varargin{3}+1;
      end
```

```
        end

        function validatePropertiesImpl(obj)
            if ~((obj.numInputsOutputs == 2) ||...
                (obj.numInputsOutputs == 3))
                error('Only 2 or 3 input and outputs allowed.');
            end
        end

        function numIn = getNumInputsImpl(obj)
            numIn = 3;
            if (obj.numInputsOutputs == 2)
                numIn = 2;
            end
        end

        function numOut = getNumOutputsImpl(obj)
            numOut = 3;
            if (obj.numInputsOutputs == 2)
                numOut = 2;
            end
        end
    end
end
```

Use this syntax to run the algorithm with two inputs and two outputs.

```
addit = AddOne;
x1 = 3;
x2 = 7;
[y1,y2] = addit(x1,x2);
```

To change the number of inputs or outputs, you must release the object before rerunning it.

```
release(addit)
x1 = 3;
x2 = 7;
x3 = 10
[y1,y2,y3] = addit(x1,x2,x3);
```

Complete Class Definition File with Multiple Inputs and Outputs

```
 classdef AddOne < matlab.System
% ADDONE Compute output values one greater than the input values
```

```matlab
% This property is nontunable and cannot be changed
% after the setup method has been called or when
% the object is running.
properties (Nontunable)
  numInputsOutputs = 3;   % Default value
end

% All methods occur inside a methods declaration.
% The stepImpl method has protected access
methods (Access = protected)
  function varargout = stepImpl(obj,varargin)
    if (obj.numInputsOutputs == 2)
      varargout{1} = varargin{1}+1;
      varargout{2} = varargin{2}+1;
    else
      varargout{1} = varargin{1}+1;
      varargout{2} = varargin{2}+1;
      varargout{3} = varargin{3}+1;
    end
  end

  function validatePropertiesImpl(obj)
    if ~((obj.numInputsOutputs == 2) ||...
        (obj.numInputsOutputs == 3))
      error('Only 2 or 3 input and outputs allowed.');
    end
  end

  function numIn = getNumInputsImpl(obj)
    numIn = 3;
    if (obj.numInputsOutputs == 2)
      numIn = 2;
    end
  end

  function numOut = getNumOutputsImpl(obj)
    numOut = 3;
    if (obj.numInputsOutputs == 2)
      numOut = 2;
    end
  end
end
```

```
end
```

See Also
getNumInputsImpl | getNumOutputsImpl

Related Examples
- "Validate Property and Input Values" on page 14-10
- "Define Basic System Objects" on page 14-3

More About
- "System Object Input Arguments and ~ in Code Examples" on page 14-60

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

Validate Properties

This example shows how to validate the value of a single property using set.*PropertyName* syntax. In this case, the *PropertyName* is Increment.

```
methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
          error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
  end
```

This example shows how to validate the value of two interdependent properties using the validatePropertiesImpl method. In this case, the UseIncrement property value must be true and the WrapValue property value must be less than the Increment property value.

```
methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
          error('Wrap value must be less than increment value');
        end
    end
end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access = protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
```

```
        end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
% ADDONE Compute an output value by incrementing the input value

  % All properties occur inside a properties declaration.
  % These properties have public access (the default)
  properties (Logical)
    UseIncrement = true
  end

  properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
  end

  methods
    % Validate the properties of the object
    function set.Increment(obj,val)
        if val >= 10
          error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
  end

  methods (Access = protected)
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue > obj.Increment
          error('Wrap value must be less than increment value');
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
          error('Input must be numeric');
        end
    end

    function out = stepImpl(obj,in)
      if obj.UseIncrement
        out = in + obj.Increment;
```

```
        else
          out = in + 1;
        end
      end
  end
end
```

Note: See "Change Input Complexity or Dimensions" for more information.

See Also
validateInputsImpl | validatePropertiesImpl

Related Examples
- "Define Basic System Objects" on page 14-3

More About
- "Methods Timing" on page 14-57
- "Property Set Methods"
- "System Object Input Arguments and ~ in Code Examples" on page 14-60

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you run the object.

Define Public Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable character vector, `default.bin`. Users cannot change *nontunable* properties after the `setup` method has been called. Refer to the Methods Timing section for more information.

```
properties (Nontunable)
  Filename = 'default.bin'
end
```

Define Private Properties to Initialize

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as *hidden* to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access = private)
  pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once the first time you run the object. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
  function setupImpl(obj)
    obj.pFileID = fopen(obj.Filename,'wb');
    if obj.pFileID < 0
        error('Opening the file failed');
    end
```

14-13

```
   end
end
```

Although not part of setup, you should close files when your code is done using them. You
use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```matlab
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or the object
    % is running.
    properties (Nontunable)
        Filename = 'default.bin' % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods (Access = protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
```

```
end
```

See Also

releaseImpl | setupImpl | stepImpl

Related Examples

- "Release System Object Resources" on page 14-31
- "Define Property Attributes" on page 14-20

More About

- "Methods Timing" on page 14-57

Set Property Values at Construction Time

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (`MyFile` in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
    function obj = MyFile(varargin)
        setProperties(obj,nargin,varargin{:});
    end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
% MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup method has been called or while the
    % object is running.
    properties (Nontunable)
        Filename ='default.bin' % the name of the file to create
        Access = 'wb' % The file access character vector (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access = private)
        pFileID; % The identifier of the file to open
    end

    methods
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
          setProperties(obj,nargin,varargin{:});
```

```
            end
        end

    methods (Access = protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object™ writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID,data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
    end
end
```

See Also
nargin | setProperties

Related Examples
- "Define Property Attributes" on page 14-20
- "Release System Object Resources" on page 14-31

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

`pCount` is an internal counter property of the System object `obj`. The user calls the `reset` method on the locked object, which calls the `resetImpl` method. In this example , `pCount` resets to 0.

Note: When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
% Counter System object™ that increments a counter

   properties (Access = private)
      pCount
   end

   methods (Access = protected)
      % Increment the counter and return
      % its value as an output
      function c = stepImpl(obj)
         obj.pCount = obj.pCount + 1;
         c = obj.pCount;
      end

      % Reset the counter to zero.
      function resetImpl(obj)
         obj.pCount = 0;
      end
   end
```

```
end
```

See "Methods Timing" on page 14-57 for more information.

See Also

resetImpl

More About

· "Methods Timing" on page 14-57

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports as nontunable.

System object users cannot change nontunable properties after the `setup` method has been called or while the object is running. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
    InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter `1` or `0` or any value that can be converted to a logical. The value, however, displays as `true` or `false`. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property. The following restrictions apply to a property with the `Logical` attribute,

- Cannot also be `Dependent` or `PositiveInteger`
- Default value must be `true` or `false`. You cannot use `1` or `0` as a default value.

```
properties (Logical)
    Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `MaxValue` is constrained to accept only real, positive integers. You cannot use sparse values. The following restriction applies to a property with the `PositiveInteger` attribute,

- Cannot also be `Dependent` or `Logical`

```
properties (PositiveInteger)
   MaxValue
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute . Properties with this attribute display their state values when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or fi value, but not a scaled double fi value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, `Constant`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess = Public` by default
- Value set only using the `setupImpl` method or when the System object is locked during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
   Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
% Counter Increment a counter to a maximum value

  % These properties are nontunable. They cannot be changed
  % after the setup method has been called or while the
  % object is running.
```

```matlab
    properties (Nontunable)
        % The inital value of the counter
        InitialValue = 0
    end
    properties (Nontunable, PositiveInteger)
        % The maximum value of the counter
        MaxValue = 3
    end

    properties (Logical)
        % Whether to increment the counter
        Increment = true
    end

    properties (DiscreteState)
        % Count state variable
        Count
    end

    methods (Access = protected)
        % Increment the counter and return its value
        % as an output

        function c = stepImpl(obj)
            if obj.Increment && (obj.Count < obj.MaxValue)
                obj.Count = obj.Count + 1;
            else
                disp(['Max count, ' num2str(obj.MaxValue) ',reached'])
            end
            c = obj.Count;
        end

        % Setup the Count state variable
        function setupImpl(obj)
            obj.Count = 0;
        end

        % Reset the counter to one.
        function resetImpl(obj)
            obj.Count = obj.InitialValue;
        end
    end
```

```
end
```

More About

- "Class Attributes"
- "Property Attributes"
- "What You Cannot Change While Your System Is Running"
- "Methods Timing" on page 14-57

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the isInactivePropertyImpl method to hide a property from displaying. If the isInactiveProperty method returns true to the property you pass in, then that property does not display.

```matlab
methods (Access = protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

Complete Class Definition File with Hidden Inactive Property

```matlab
classdef Counter < matlab.System
  % Counter Increment a counter

  % These properties are nontunable. They cannot be changed
  % after the setup method has been called or when the
  % object is running.
  properties (Nontunable)
    % Allow the user to set the initial value
    UseRandomInitialValue = true
    InitialValue = 0
  end

  % The private count variable, which is tunable by default
  properties (Access = private)
    pCount
  end

  methods (Access = protected)
    % Increment the counter and return its value
    % as an output
    function c = stepImpl(obj)
```

```
      obj.pCount = obj.pCount + 1;
      c = obj.pCount;
    end

    % Reset the counter to either a random value or the initial
    % value.
    function resetImpl(obj)
      if obj.UseRandomInitialValue
        obj.pCount = rand();
      else
        obj.pCount = obj.InitialValue;
      end
    end

    % This method controls visibility of the object's properties
    function flag = isInactivePropertyImpl(obj,propertyName)
      if strcmp(propertyName,'InitialValue')
        flag = obj.UseRandomInitialValue;
      else
        flag = false;
      end
    end
  end
end
```

See Also

isInactivePropertyImpl

Limit Property Values to Finite List

This example shows how to limit a property to accept only a finite set of character vector values.

Specify a Set of Valid Character Vector Values

String sets use two related properties. You first specify the user-visible property name and default character vector value. Then, you specify the associated hidden property by appending "Set" to the property name. You must use a capital "S" in "Set."

In the "Set" property, you specify the valid character vector values as a cell array of the `matlab.system.StringSet` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
   Color = 'blue'
end

properties (Hidden,Transient)
   ColorSet = matlab.system.StringSet({'red','blue','green'});
end
```

Complete Class Definition File with StringSet

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ illustrates the use of StringSets

   properties
     Color = 'blue'
   end

   properties (Hidden,Transient)
     % Let them choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]),randn([2,1]), ...
```

```
                    'Color',obj.Color(1));
        end
        function releaseImpl(obj)
            cla(Whiteboard.getWhiteboard());
            hold on
        end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold on
          end
          a = gca;
      end
    end
end
```

String Set System Object Example

```
%%
% Each time you run the object, it draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk  = Whiteboard;

% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color = 'blue';

% Take a few steps
for i=1:3
  hGreenInk();
  hBlueInk();
end

%% Clear the whiteboard
hBlueInk.release();
```

```
%% Display System object used in this example
type('Whiteboard.m');
```

See Also

matlab.system.StringSet

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods  (Access = protected)
   function processTunedPropertiesImpl(obj)
      propChange = isChangedProperty(obj,'NumNotes')||...
         isChangedProperty(obj,'MiddleC')
      if propChange
         obj.pLookupTable = obj.MiddleC *...
            (1+log(1:obj.NumNotes)/log(12));
      end
   endend
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
  % TuningFork Illustrate the processing of tuned parameters
  %

  properties
    MiddleC = 440
    NumNotes = 12
  end

  properties (Access = private)
    pLookupTable
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.MiddleC = 440;
      obj.pLookupTable = obj.MiddleC * ...
```

```
            (1+log(1:obj.NumNotes)/log(12));
        end

        function hz = stepImpl(obj,noteShift)
          % A noteShift value of 1 corresponds to obj.MiddleC
          hz = obj.pLookupTable(noteShift);
        end

        function processTunedPropertiesImpl(obj)
          propChange = isChangedProperty(obj,'NumNotes')||...
              isChangedProperty(obj,'MiddleC')
          if propChange
            obj.pLookupTable = obj.MiddleC *...
              (1+log(1:obj.NumNotes)/log(12));
          end
        end
    end
end
```

See Also

processTunedPropertiesImpl

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
   function releaseImpl(obj)
      cla(Whiteboard.getWhiteboard());
      hold on
   end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
% Whiteboard Draw lines on a figure window
%
% This System object™ shows the use of StringSets
%
   properties
     Color = 'blue'
   end

   properties (Hidden)
     % Let user choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods (Access = protected)
     function stepImpl(obj)
        h = Whiteboard.getWhiteboard();
        plot(h, ...
          randn([2,1]), randn([2,1]), ...
          'Color',obj.Color(1));
     end

     function releaseImpl(obj)
        cla(Whiteboard.getWhiteboard());
        hold on
```

```
        end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold on
          end
          a = gca;
      end
    end
end
```

See Also
releaseImpl

Related Examples
·   "Initialize Properties and Setup One-Time Calculations" on page 14-13

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a bandpass filter System object from separate highpass and lowpass filter System objects.

Store System Objects in Properties

To define a System object from other System objects, store those other objects in your class definition file as properties. In this example, the highpass and lowpass filters are the separate System objects defined in their own class-definition files.

```
properties (Access = private)
   % Properties that hold filter System objects
   pLowpass
   pHighpass
end
```

Complete Class Definition File of Bandpass Filter Composite System Object

```
classdef BandpassFIRFilter < matlab.System
% Implements a bandpass filter using a cascade of eighth-order lowpass
% and eighth-order highpass FIR filters.

    properties (Access = private)
        % Properties that hold filter System objects
        pLowpass
        pHighpass
    end

    methods (Access = protected)
        function setupImpl(obj)
            % Setup composite object from constituent objects
            obj.pLowpass = LowpassFIRFilter;
            obj.pHighpass = HighpassFIRFilter;
        end

        function yHigh = stepImpl(obj,u)
            yLow = obj.pLowpass(u);
            yHigh = obj.pHighpass(yLow);
        end

        function resetImpl(obj)
```

```
                    reset(obj.pLowpass);
                    reset(obj.pHighpass);
                end
        end
    end
end
```

Class Definition File for Lowpass FIR Component of Bandpass Filter

```matlab
classdef LowpassFIRFilter < matlab.System
% Implements eighth-order lowpass FIR filter with 0.6pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006,-0.0133,-0.05,0.26,0.6,0.26,-0.05,-0.0133,0.006];
    end

    properties (DiscreteState)
        State
    end

    methods (Access = protected)
        function setupImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
        function y = stepImpl(obj,u)
            [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
        end
        function resetImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
    end
end
```

Class Definition File for Highpass FIR Component of Bandpass Filter

```matlab
classdef HighpassFIRFilter < matlab.System
% Implements eighth-order highpass FIR filter with 0.4pi cutoff

    properties (Nontunable)
        % Filter coefficients
        Numerator = [0.006,0.0133,-0.05,-0.26,0.6,-0.26,-0.05,0.0133,0.006];
    end

    properties (DiscreteState)
        State
```

```
        end

    methods (Access = protected)
        function setupImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end

        function y = stepImpl(obj,u)
            [y,obj.State] = filter(obj.Numerator,1,u,obj.State);
        end

        function resetImpl(obj)
            obj.State = zeros(length(obj.Numerator)-1,1);
        end
    end
end
```

See Also

```
nargin
```

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
```

2 Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
   function bDone = isDoneImpl(obj)
      bDone = obj.NumSteps==2
   end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
  % RunTwice System object that runs exactly two times
  %
  properties (Access = private)
    NumSteps
  end

  methods (Access = protected)
    function resetImpl(obj)
      obj.NumSteps = 0;
    end

    function y = stepImpl(obj)
      if ~obj.isDone()
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
      else
        y = 0;
      end
    end

    function bDone = isDoneImpl(obj)
```

```
            bDone = obj.NumSteps==2;
        end
    end
end
```

See Also

matlab.system.mixin.FiniteSource

More About

- "What Are Mixin Classes?" on page 14-61
- "Subclass Multiple Classes"
- "System Object Input Arguments and ~ in Code Examples" on page 14-60

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods (Access = protected)
  function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    s.child = matlab.System.saveObject(obj.child);
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;
    if isLocked(obj)
      s.state = obj.state;
    end
  end
end
```

Complete Class Definition Files with Save and Load

The `Counter` class definition file sets up an object with a count property. This counter is used in the `MySaveLoader` class definition file to count the number of child objects.

```
classdef Counter < matlab.System
  properties(DiscreteState)
    Count
  end
  methods (Access=protected)
    function setupImpl(obj, ~)
      obj.Count = 0;
    end
    function y = stepImpl(obj, u)
      if u > 0
        obj.Count = obj.Count + 1;
      end
      y = obj.Count;
    end
```

```
      end
    end

    classdef MySaveLoader < matlab.System

      properties (Access = private)
        child
        pdependentprop = 1
      end

      properties (Access = protected)
        protectedprop = rand;
      end

      properties (DiscreteState = true)
        state
      end

      properties (Dependent)
        dependentprop
      end

      methods
        function obj = MySaveLoader(varargin)
          obj@matlab.System();
          setProperties(obj,nargin,varargin{:});
        end

        function set.dependentprop(obj, value)
          obj.pdependentprop = min(value, 5);
        end

        function value = get.dependentprop(obj)
          value = obj.pdependentprop;
        end
      end

      methods (Access = protected)
        function setupImpl(obj)
          obj.state = 42;
          obj.child = Counter;
        end
        function out = stepImpl(obj,in)
          obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
          out = obj.child(obj.state);
```

```
        end
      end

    % Serialization
    methods (Access = protected)
      function s = saveObjectImpl(obj)
        % Call the base class method
        s = saveObjectImpl@matlab.System(obj);

        % Save the child System objects
        s.child = matlab.System.saveObject(obj.child);

        % Save the protected & private properties
        s.protectedprop = obj.protectedprop;
        s.pdependentprop = obj.pdependentprop;

        % Save the state only if object locked
        if isLocked(obj)
          s.state = obj.state;
        end
      end

      function loadObjectImpl(obj,s,wasLocked)
        % Load child System objects
        obj.child = matlab.System.loadObject(s.child);

        % Load protected and private properties
        obj.protectedprop = s.protectedprop;
        obj.pdependentprop = s.pdependentprop;

        % Load the state only if object locked
        if wasLocked
          obj.state = s.state;
        end

        % Call base class method to load public properties
        loadObjectImpl@matlab.System(obj,s,wasLocked);
      end
    end
end
```

See Also

loadObjectImpl | saveObjectImpl

Related Examples

- "Load System Object" on page 14-42

Load System Object

This example shows how to load and save a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to load the child System object, load protected and private properties, load the state if the object is locked, and use `loadObjectImpl` from the base class to load public properties.

```
methods (Access = protected)
  function loadObjectImpl(obj,s,wasLocked)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if wasLocked
      obj.state = s.state;
    end

    loadObjectImpl@matlab.System(obj,s,wasLocked);
  end
end
```

Complete Class Definition Files with Save and Load

The Counter class definition file sets up an object with a count property. This counter is used in the MySaveLoader class definition file to count the number of child objects.

```
classdef Counter < matlab.System
  properties(DiscreteState)
    Count
  end
  methods (Access=protected)
    function setupImpl(obj, ~)
      obj.Count = 0;
    end
    function y = stepImpl(obj, u)
      if u > 0
        obj.Count = obj.Count + 1;
      end
      y = obj.Count;
```

```
      end
    end
end

classdef MySaveLoader < matlab.System

  properties (Access = private)
    child
    pdependentprop = 1
  end

  properties (Access = protected)
    protectedprop = rand;
  end

  properties (DiscreteState = true)
    state
  end

  properties (Dependent)
    dependentprop
  end

  methods
    function obj = MySaveLoader(varargin)
      obj@matlab.System();
      setProperties(obj,nargin,varargin{:});
    end

    function set.dependentprop(obj, value)
      obj.pdependentprop = min(value, 5);
    end

    function value = get.dependentprop(obj)
      value = obj.pdependentprop;
    end
  end

  methods (Access = protected)
    function setupImpl(obj)
      obj.state = 42;
      obj.child = Counter;
    end
    function out = stepImpl(obj,in)
      obj.state = in + obj.state + obj.protectedprop + obj.pdependentprop;
```

```matlab
      out = obj.child(obj.state);
    end
  end

  % Serialization
  methods (Access = protected)
    function s = saveObjectImpl(obj)
      % Call the base class method
      s = saveObjectImpl@matlab.System(obj);

      % Save the child System objects
      s.child = matlab.System.saveObject(obj.child);

      % Save the protected & private properties
      s.protectedprop = obj.protectedprop;
      s.pdependentprop = obj.pdependentprop;

      % Save the state only if object locked
      if isLocked(obj)
        s.state = obj.state;
      end
    end

    function loadObjectImpl(obj,s,wasLocked)
      % Load child System objects
      obj.child = matlab.System.loadObject(s.child);

      % Load protected and private properties
      obj.protectedprop = s.protectedprop;
      obj.pdependentprop = s.pdependentprop;

      % Load the state only if object locked
      if wasLocked
        obj.state = s.state;
      end

      % Call base class method to load public properties
      loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
  end
end
```

```
end
```

See Also
loadObjectImpl | saveObjectImpl

Related Examples

·   "Save System Object" on page 14-38

Define System Object Information

This example shows how to define information to display for a System object.

Define System Object Info

You can define your own `info` method to display specific information for your System object. The default `infoImpl` method returns an empty struct. This `infoImpl` method returns detailed information when the `info` method is called using `info(x,'details')` or only count information if it is called using `info(x)`.

```
methods (Access = protected)
   function s = infoImpl(obj,varargin)
      if nargin>1 && strcmp('details',varargin(1))
         s = struct('Name','Counter',...
              'Properties', struct('CurrentCount', ...
              obj.Count,'Threshold',obj.Threshold));
      else
         s = struct('Count',obj.Count);
      end
   end
end
```

Complete Class Definition File with InfoImpl

```
classdef Counter < matlab.System
   % Counter Count values above a threshold

   properties
      Threshold = 1
   end

   properties (DiscreteState)
      Count
   end

   methods (Access = protected)
      function setupImpl(obj)
         obj.Count = O;
      end

      function resetImpl(obj)
         obj.Count = O;
      end
```

```matlab
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end

        function s = infoImpl(obj,varargin)
            if nargin>1 && strcmp('details',varargin(1))
                s = struct('Name','Counter',...
                'Properties', struct('CurrentCount', ...
                obj.Count,'Threshold',obj.Threshold));
            else
                s = struct('Count',obj.Count);
            end
        end
    end
end
```

See Also

infoImpl

Add Data Types Tab to MATLAB System Block

This example shows how to add a Data Types tab to the MATLAB System block dialog box. This tab includes fixed-point data type settings.

Display Data Types Tab

This example shows how to use `matlab.system.showFiSettingsImpl` to display the Data Types tab in the `MATLAB System` block dialog.

```
methods (Static, Access = protected)
    function showTab = showFiSettingsImpl
        showTab = true;
    end
end
```

Complete Class Definition File with Data Types Tab

Use `showFiSettingsImpl` to display the Data Types tab for a System object that adds an offset to a fixed-point input.

```
classdef FiTabAddOffset < matlab.System
% FiTabAddOffset Add an offset to input

    properties
        Offset = 1;
    end

    methods
        function obj = FiTabAddOffset(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access = protected)
        function y = stepImpl(~,u)
            y = u + obj.Offset;
        end
    end

    methods(Static, Access=protected)
        function header = getHeaderImpl
            header = matlab.system.display.Header('Title',...
                'Add Offset','Text','Add an offset to the input');
```

```
        end

        function isVisible = showFiSettingsImpl
            isVisible = true;
        end
    end
end
```
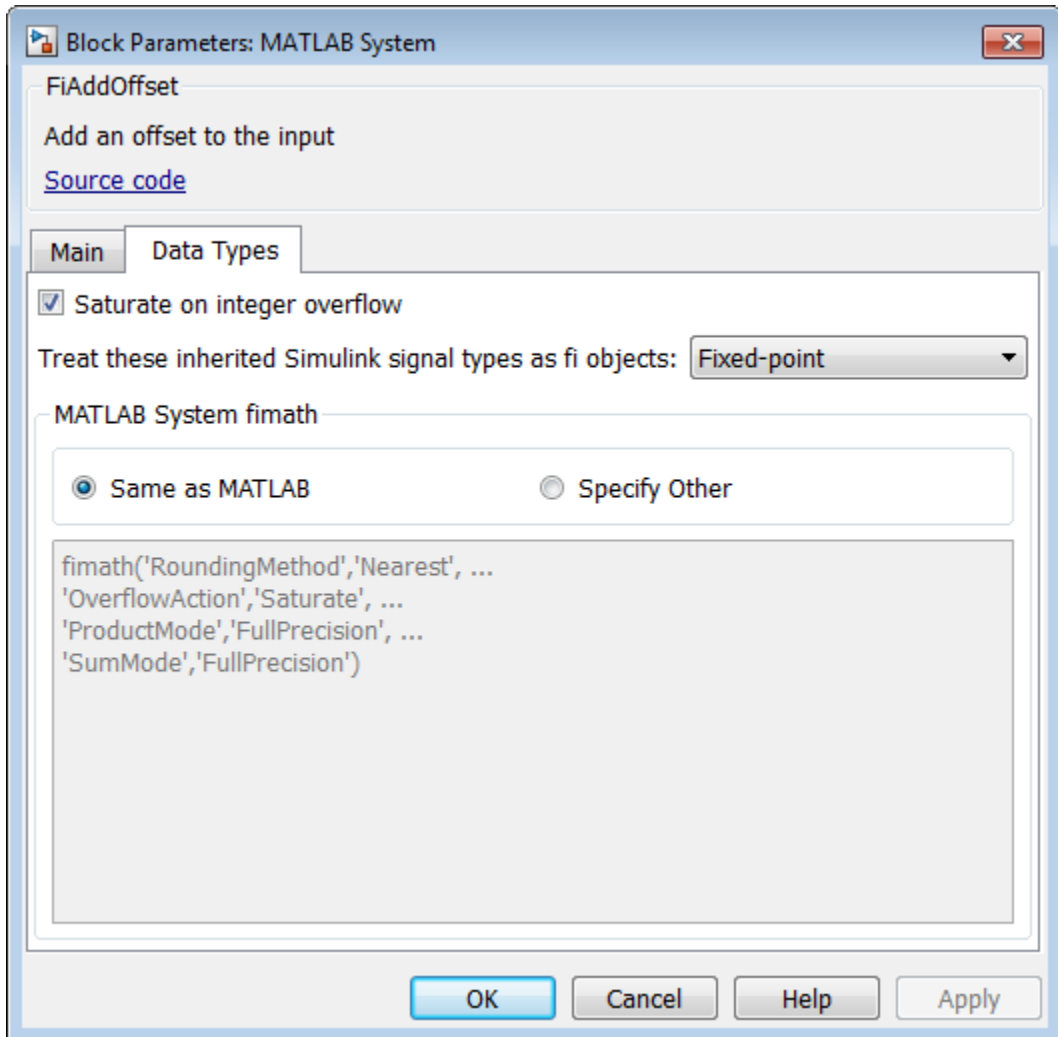
Add Button to MATLAB System Block

This example shows how to add a button to the MATLAB System block dialog box. This button launches a figure that plots a ramp function.

Define Action for Dialog Button

This example shows how to use `matlab.system.display.Action` to define the MATLAB function or code associated with a button in the MATLAB System block dialog. The example also shows how to set button options and use an actionData object input to store a figure handle. This part of the code example uses the same figure when the button is clicked multiple times, rather than opening a new figure for each button click.

```matlab
methods(Static,Access = protected)
  function group = getPropertyGroupsImpl
    group = matlab.system.display.Section(mfilename('class'));
    group.Actions = matlab.system.display.Action(@(actionData,obj)...
        visualize(obj,actionData),'Label','Visualize');
  end
end

methods
  function obj = ActionDemo(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj,actionData)
    f = actionData.UserData;
    if isempty(f) || ~ishandle(f)
      f = figure;
      actionData.UserData = f;
    else
      figure(f); % Make figure current
    end

    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

Complete Class Definition File for Dialog Button

Define a property group and a second tab in the class definition file.

```matlab
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
      group = matlab.system.display.Section(mfilename('class'));
      group.Actions = matlab.system.display.Action(@(actionData,obj)...
        visualize(obj,actionData),'Label','Visualize');
    end
  end

  methods
    function obj = ActionDemo(varargin)
      setProperties(obj,nargin,varargin{:});
    end

    function visualize(obj,actionData)
      f = actionData.UserData;
      if isempty(f) || ~ishandle(f)
        f = figure;
        actionData.UserData = f;
      else
        figure(f); % Make figure current
      end

      d = 1:obj.RampLimit;
      plot(d);
    end
  end
end
```

end

Function Block Parameters: MATLAB System

PlotRamp

Display a button to launch a plot figure.

Source code

Parameters

RampLimit: 10

Visualize

Simulate using: Code generation

OK   Cancel   Help   Apply

More About

- "System Object Input Arguments and ~ in Code Examples" on page 14-60

Specify Locked Input Size

This example shows how to specify whether the size of a System object input is locked. The size of a locked input cannot change until the System object is unlocked. Run the object to lock it. Use `release` to unlock the object.

Use the `isInputSizeLockedImpl` method to specify that the input size is locked.

```
methods (Access = protected)
  function flag = isInputSizeLockedImpl(~,~)
    flag = true;
  end
end
```

View the method in the complete class definition file.

```
classdef Counter < matlab.System
  %Counter Count values above a threshold

  properties
    Threshold = 1
  end

  properties (DiscreteState)
    Count
  end

  methods
    function obj = Counter(varargin)
      setProperties(obj,nargin,varargin{:});
    end
  end

  methods (Access=protected)
    function resetImpl(obj)
      obj.Count = 0;
    end

    function y = stepImpl(obj, u1)
      if (any(u1 >= obj.Threshold))
        obj.Count = obj.Count + 1;
      end
      y = obj.Count;
    end
```

```
      function flag = isInputSizeLockedImpl(~,~)
        flag = true;
      end
    end
end
```

See Also

isInputSizeLockedImpl

Set Model Reference Discrete Sample Time Inheritance

This example shows how to disallow model reference discrete sample time inheritance for a System object. The System object defined in this example has one input, so by default, it allows sample time inheritance. To override the default and disallow inheritance, the class definition file for this example includes the `allowModelReferenceDiscreteSampleTimeInheritanceImpl` method, with its output set to `false`.

```
methods (Access = protected)
   function flag = ...
       allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
     flag = false;
   end
end
```

View the method in the complete class definition file.

```
classdef MyCounter < matlab.System

   % MyCounter Count values

    properties
       Threshold = 1;
    end

    properties (DiscreteState)
       Count
    end

    methods (Static, Access = protected)
      function header = getHeaderImpl
        header = matlab.system.display.Header('MyCounter',...
          'Title','My Enhanced Counter',...
          'Text', 'This counter is an enhanced version.');
      end
    end

    methods (Access = protected)
      function flag = ...
          allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
          flag = false
      end
      function setupImpl(obj,u)
```

```matlab
            obj.Count = 0;
        end
        function y = stepImpl(obj,u)
            if (u > obj.Threshold)
                obj.Count = obj.Count + 1;
            end
            y = obj.Count;
        end
        function resetImpl(obj)
            obj.Count = 0;
        end
    end
end
```

Methods Timing

The call sequence diagrams show the order in which actions are performed when you run the specified method. The background color or each action indicates the method type.

- White background — Sealed method
- Green background — User-implemented method
- White and green background — Sealed method that calls a user-implemented method

Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.
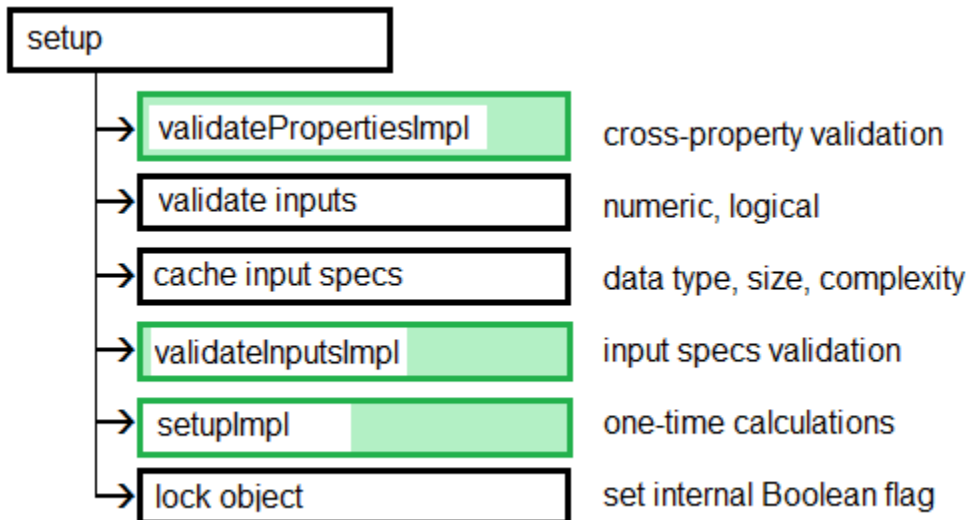
Running the Object (Step Method) Call Sequence

This hierarchy shows the actions performed when you call the step method.
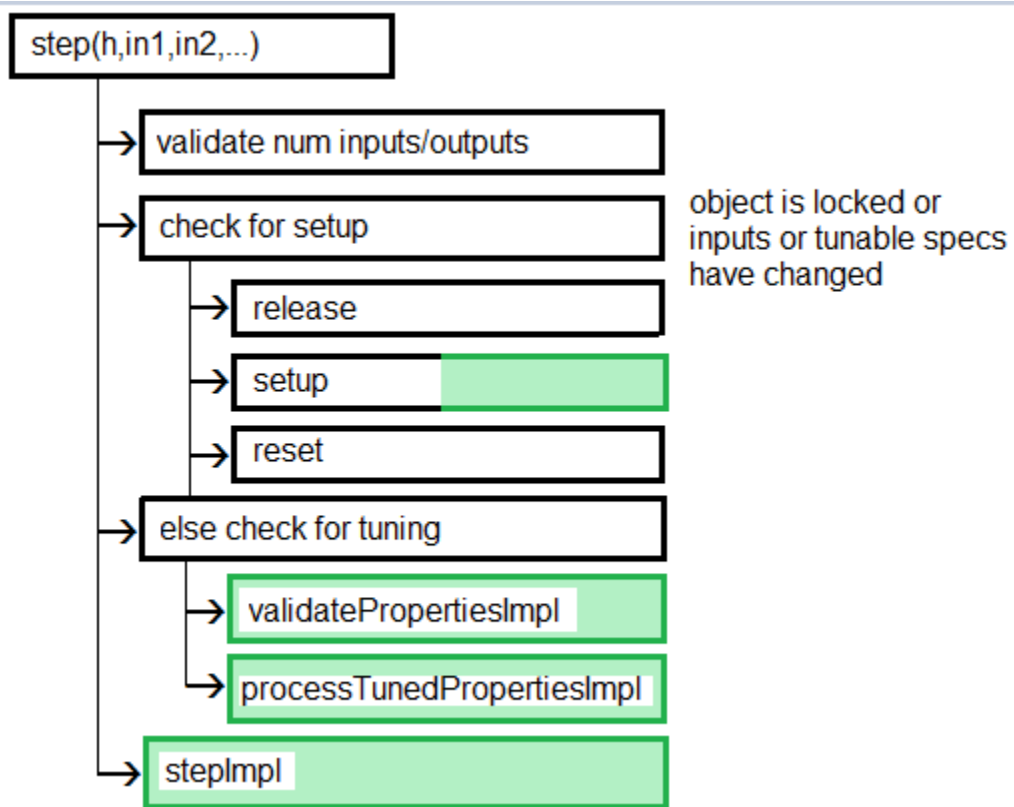
Reset Method Call Sequence

This hierarchy shows the actions performed when you call the reset method.
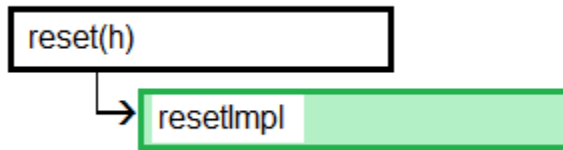
Release Method Call Sequence

This hierarchy shows the actions performed when you call the release method.
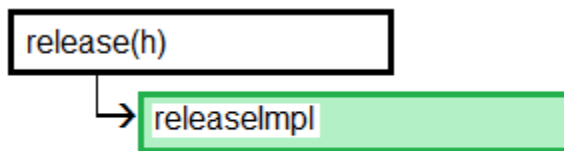
See Also

releaseImpl | resetImpl | setupImpl | stepImpl

Related Examples

- "Release System Object Resources" on page 14-31
- "Reset Algorithm State" on page 14-18
- "Set Property Values at Construction Time" on page 14-16
- "Define Basic System Objects" on page 14-3

More About

- "What Are System Object Methods?"
- "Running a System Object"
- "Common Methods"

System Object Input Arguments and ~ in Code Examples

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In many examples, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables.

What Are Mixin Classes?

Mixin classes are partial classes that you can combine in various combinations to form desired behaviors using multiple inheritance. System objects are composed of a base class, `matlab.System` and may include one or more mixin classes. You specify the base class and mixin classes on the first line of your class definition file.

The following mixin classes are available for use with System objects.

- `matlab.system.mixin.CustomIcon` — Defines a block icon for System objects in the MATLAB System block
- `matlab.system.mixin.FiniteSource` — Adds the `isDone` method to System objects that are sources
- `matlab.system.mixin.Nondirect` — Allows the System object, when used in the MATLAB System block, to support nondirect feedthrough by making the runtime callback functions, `output` and `update` available
- `matlab.system.mixin.Propagates` — Enables System objects to operate in the MATLAB System block using the interpreted execution

Best Practices for Defining System Objects

A System object is a specialized kind of MATLAB object that is optimized for iterative processing. Use System objects when you need to run an object multiple times or process data in a loop. When defining your own System object, use the following suggestions to help your code run efficiently.

- Define all one-time calculations in the `setupImpl` method and cache the results in a private property. Use the `stepImpl` method for repeated calculations.

- If properties are accessed more than once in the `stepImpl` method, cache those properties as local variables inside the method. A typical example of multiple property access is a loop. Iterative calculations using cached local variables run faster than calculations that must access the properties of an object. When the calculations for the method complete, you can save the local cached results back to the properties of that System object. Copy frequently used tunable properties into private properties. This best practice also applies to the `updateImpl` and `outputImpl` methods.

  In this example, `k` is accessed multiple times in each loop iteration, but is saved to the object property only once.

  ```
  function y = stepImpl(obj,x)
    k = obj.MyProp;
    for p=1:100
      y = k * x;
      k = k + 0.1;
    end
    obj.MyProp = k;
  end
  ```

- Property default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

- Do not use character vector comparisons or character vector-based switch statements in the `stepImpl` method. Instead, create a method handle in `setupImpl`. This handle points to a method in the same class definition file. Use that handle in a loop in `stepImpl`.

  This example shows how to use method handles and cached local variables in a loop to implement an efficient object. In `setupImpl`, choose `myMethod1` or `myMethod2` based on a character vector comparison and assign the method handle to the `pMethodHandle` property. Because there is a loop in `stepImpl`, assign the

pMethodHandle property to a local method handle, myFun, and then use myFun inside the loop.

```
classdef MyClass < matlab.System
  function setupImpl(obj)
    if strcmp(obj.Method, 'Method1')
      obj.pMethodHandle = @myMethod1;
    else
      obj.pMethodHandle = @myMethod2;
    end
  end
  function y = stepImpl(obj,x)
    myFun = obj.pMethodHandle;
      for p=1:1000
        y = myFun(obj,x)
      end
    end
  end
  function y = myMethod1(x)
    y = x+1;
  end
  function y = myMethod2(x)
    y = x-1;
  end
end
```

- If the number of System object inputs does not change, do not implement the getNumInputsImpl method. Also do not implement the getNumInputsImpl method when you explicitly list the inputs in the stepImpl method instead of using varargin. The same caveats apply to the getNumOutputsImpl and varargout outputs.

- For the getNumInputsImpl and getNumOutputsImpl methods, if you set the return argument from an object property, that object property must have the Nontunable attribute.

- If the variables in a method do not need to retain their values between calls use local scope for those variables in that method.

- For properties that do not change, define them in as Nontunable properties. Tunable properties have slower access times than Nontunable properties

- Use the protected or private attribute instead of the public attribute for a property, whenever possible. Some public properties have slower access times than protected and private properties.

- Avoid using acustomized stepImpl, get, or set methods, whenever possible.

- Avoid using character vector comparisons within a customized `stepImpl`, `get`, or `set` methods, whenever possible. Use `setupImpl` for character vector comparisons instead.
- Specify Boolean values using `true` or `false` instead of `1` or `0`, respectively.

Insert System Object Code Using MATLAB Editor

In this section...

Define System Objects with Code Insertion

You can define System objects from the MATLAB Editor using code insertion options. When you select these options, the MATLAB Editor adds predefined properties, methods, states, inputs, or outputs to your System object. Use these tools to create and modify System objects faster, and to increase accuracy by reducing typing errors.

To access the System object editing options, create a new System object, or open an existing one.
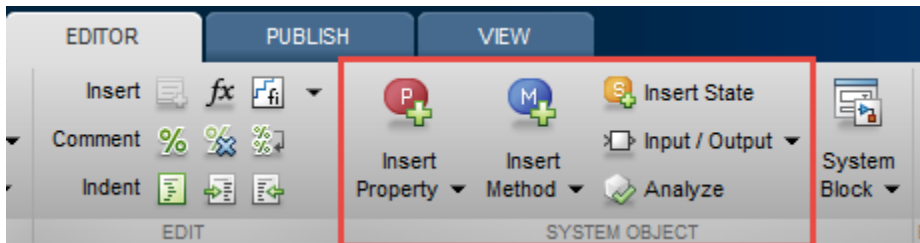
To add predefined code to your System object, select the code from the appropriate menu. For example, when you click **Insert Property** > **Numeric**, the MATLAB Editor adds the following code:

```
properties(Nontunable)
    Property
end
```

The MATLAB Editor inserts the new property with the default name `Property`, which you can rename. If you have an existing properties group with the `Nontunable`

14-65

attribute, the MATLAB Editor inserts the new property into that group. If you do not have a property group, the MATLAB Editor creates one with the correct attribute.
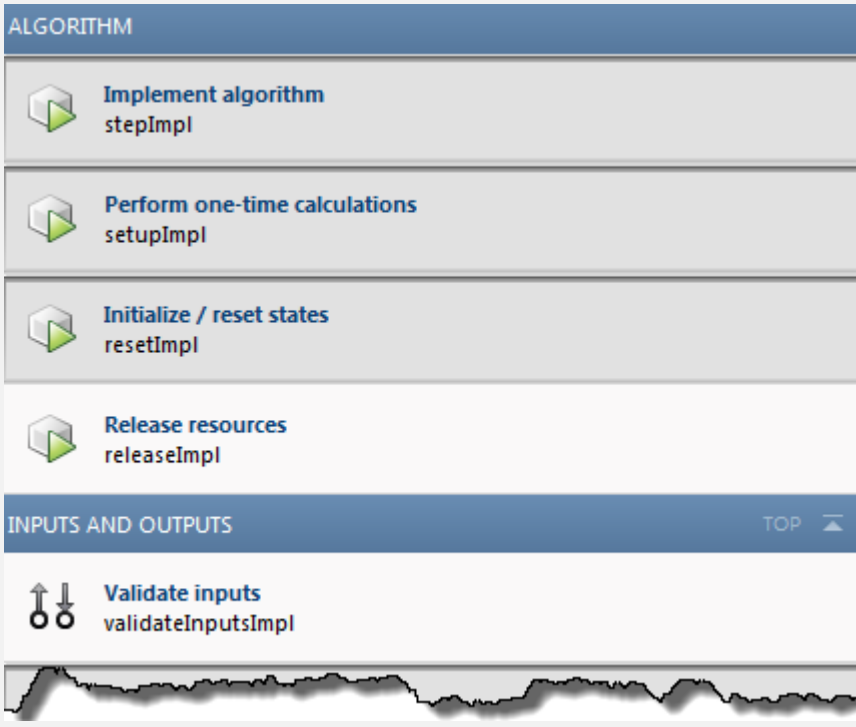
Insert Options

Properties	Properties of the System object: Numeric, Logical, String Set, Positive Integer, Tunable Numeric, Private, Protected, and Custom. When you select String Set or Custom Properties, a separate dialog box opens to guide you in creating these properties.

Methods	Methods commonly used in System object definitions. The MATLAB Editor creates only the method structure. You specify the actions of that method.
	
	 The **Insert Method** menu organizes methods by categories, such as **Algorithm**, **Inputs and Outputs**, and **Properties and States**. When you select a method from the menu, the MATLAB Editor inserts the method template in your System object code. In this example, selecting **Insert Method** > **Release resources** inserts the following code:
	
	`function releaseImpl(obj)`
	`    % Release resources, such as file handles`
	`end`
	
	If an method from the **Insert Method** menu is present in the System object code, that method is shown shaded on the **Insert Method** menu:

States	Properties containing the `DiscreteState` attribute.
Inputs / Outputs	Inputs, outputs, and related methods, such as **Validate inputs** and **Lock input size**.

When you select an input or output, the MATLAB Editor inserts the specified code in the `stepImpl` method. In this example, selecting **Insert** > **Input** causes the MATLAB Editor to insert the required input variable `u2`. The MATLAB Editor determines the variable name, but you can change it after it is inserted.

```
function y = stepImpl(obj,u,u2)
    % Implement algorithm. Calculate y as a function of
    % input u and discrete states.
    y = u;
end
```

Create Fahrenheit Temperature String Set

1 Open a new or existing System object.

2 In the MATLAB Editor, select **Insert Property** > **String Set**.

3 In the **String Set** dialog box, under **Name**, replace `Color` with `TemperatureUnit`.

4 Remove the existing `Color` property values with the - (minus) button.

5 Add a property value with the + (plus) button. Enter `Fahrenheit`.

6 Add another property value with +. Enter `Celsius`.

7 Add another property value with +. Enter `Kelvin`.

8 Select `Fahrenheit` as the default value by clicking **Default**.

The dialog box now looks as shown:
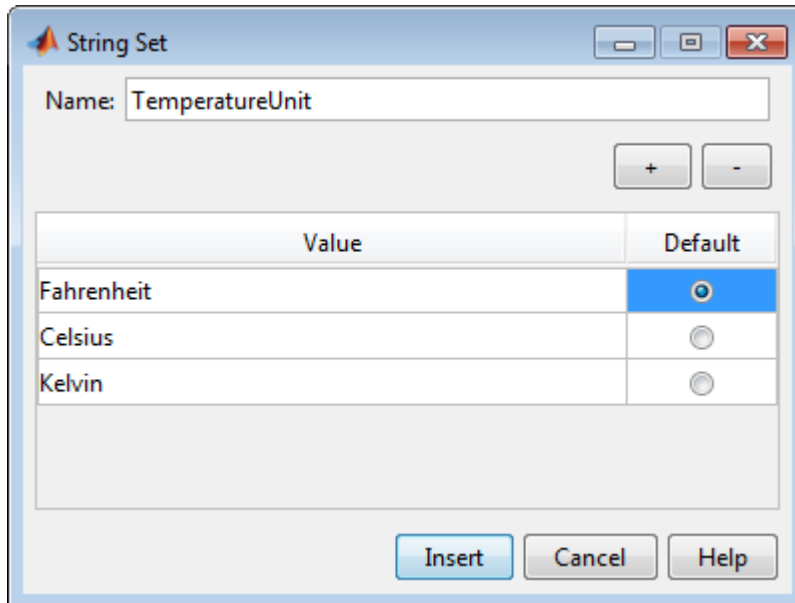
9 To create this string set and associated properties, with the default value selected, click **Insert**.

Examine the System object definition. The MATLAB Editor has added the following code:

```
properties (Nontunable)
```

```
    TemperatureUnit = 'Fahrenheit';
  end


  properties(Constant, Hidden)
    TemperatureUnitSet = matlab.system.StringSet({'Fahrenheit','Celsius','Kelvin'});
  end
```

For more information on the `StringSet` class, see matlab.System.StringSet.

Create Custom Property for Freezing Point

1 Open a new or existing System object.

2 In the MATLAB Editor, select **Insert Property** > **Custom Property**.

3 In the Custom Property dialog box, under **System Object Attributes**, select
   Nontunable. Under **MATLAB Property Attributes**, select **Constant**. Leave
   GetAccess as `public`. **SetAccess** is grayed out because properties of type constant
   can not be set using System object methods.

   The dialog box now looks as shown:
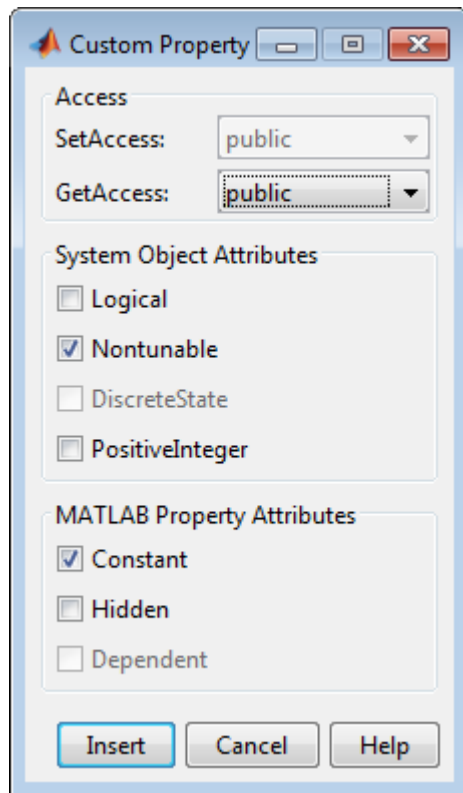
4   To insert the property into the System object code, click **Insert**.

```
properties(Nontunable, Constant)
    Property
end
```

5   Replace `Property` with your property.

```
properties(Nontunable, Constant)
    FreezingPointFahrenheit = 32;
end
```

Define Input Size As Locked

1   Open a new or existing System object.

2 In the MATLAB Editor, select **Insert Method** > **Lock input size**.

The MATLAB Editor inserts this code into the System object:

```
function flag = isInputSizeLockedImpl(obj,index)
    % Return true if input size is not allowed to change while
    % system is running
    flag = true;
end
```

Related Examples

- "Analyze System Object Code" on page 14-72

Analyze System Object Code

In this section...
"View and Navigate System object Code" on page 14-72
"Example: Go to StepImpl Method Using Analyzer" on page 14-72

View and Navigate System object Code

View and navigate System object code using the Analyzer.

The Analyzer displays all elements in your System object code.

- Navigate to a specific input, output, property, state, or method by clicking the name of that element.
- Expand or collapse element sections with the arrow buttons.
- Identify access levels for properties and custom methods with the + (public), # (protected), and – (private) symbols.

Example: Go to StepImpl Method Using Analyzer

1 Open an existing System object.
2 Select **Analyze**.
3 Click stepImpl.
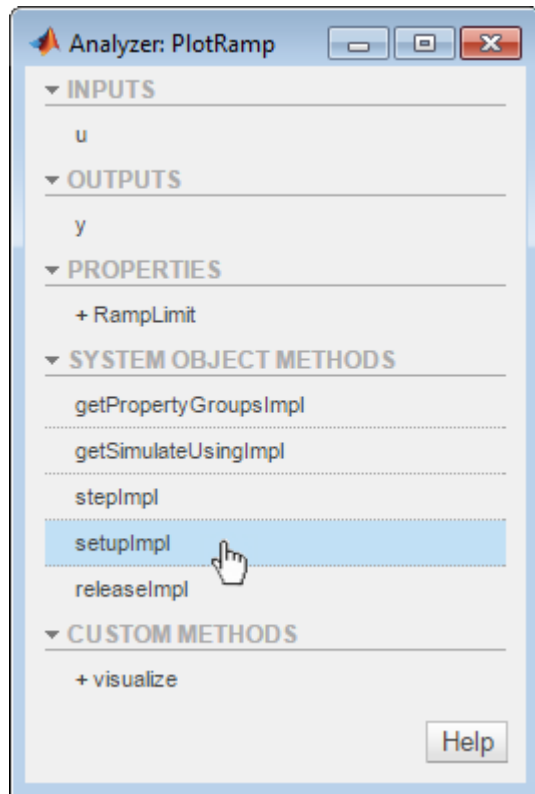
The cursor in the MATLAB Editor window jumps to the `stepImpl` method.

```
        d = 1:obj.RampLimit;
        plot(d);
    end
end

methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(~,obj)...
            visualize(obj),'Label','Visualize');
    end

    function simMode = getSimulateUsingImpl
        % Return only allowed simulation mode in System block dialog
        simMode = 'Interpreted execution';
    end
end

methods(Access = protected)
    function y = stepImpl(~,u)
        % Implement algorithm. Calculate y as a function of input u and
        % discrete states.
        y = u;
    end
```

Related Examples

- "Insert System Object Code Using MATLAB Editor" on page 14-65

Define System Object for Use in Simulink

In this section...

Develop System Object for Use in System Block

You can develop a System object for use in a System block and interactively preview the block dialog box. This feature requires Simulink.

With the **System Block** editing options, the MATLAB Editor inserts predefined code into the System object. This coding technique helps you create and modify your System object faster and increases accuracy by reducing typing errors.

Using these options, you can also:

- View and interact with the block dialog design as you define the System object.
- Add dialog customization methods. If the block dialog box is open when you make changes, the block dialog design preview updates the display on saving the file.
- Add icon methods. However, these elements display only on the MATLAB System Block in Simulink, not in the Preview Dialog Box.
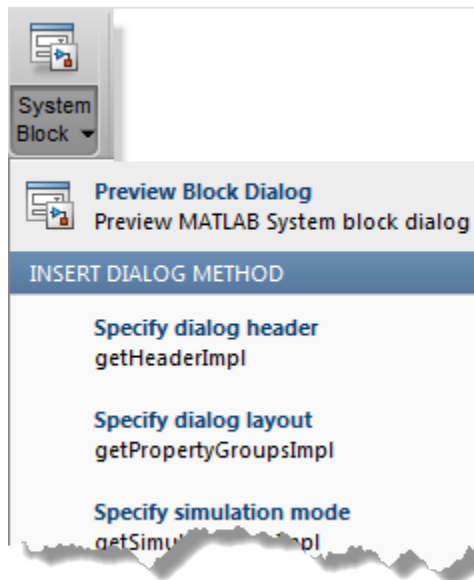
Define Block Dialog Box for Plot Ramp

1 Create a System object and name it `PlotRamp`. This name becomes the block dialog box title. Save the System object.

2 Add a comment that contains the block description.

```
% Display a button to launch a plot figure.
```

This comment becomes the block parameters dialog box description, under the block title.

3 Select **System Block** > **Preview Block Dialog**. The block dialog box displays as you develop the System object.
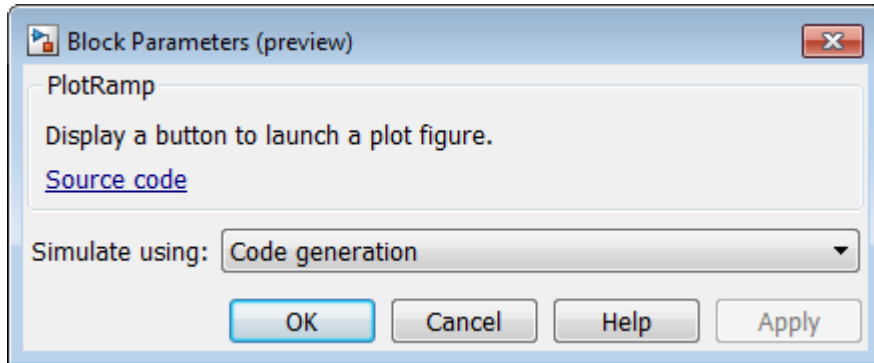
4 Add a ramp limit by selecting **Insert Property** > **Numeric**. Then change the property name and set the value to 10.

```
properties (Nontunable)
  RampLimit = 10;
end
```

5 Using the **System Block** menu, insert the getPropertyGrouplsImpl method.

```
methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
    end
end
```

6 Add code to create the group for the visualize action..

```
methods(Access = protected, Static)
    function group = getPropertyGroupsImpl
        % Define property section(s) for System block dialog
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(~,obj)...
           visualize(obj),'Label','Visualize');
    end
end
```

7 Add a function that adds code to display the **Visualize** button on the dialog box.

```
methods
  function visualize(obj)
    figure;
```

```
        d = 1:obj.RampLimit;
        plot(d);
    end
  end
```

8 As you add elements to the System block definition, save your file. Observe the effects of your code additions to the System block definition.
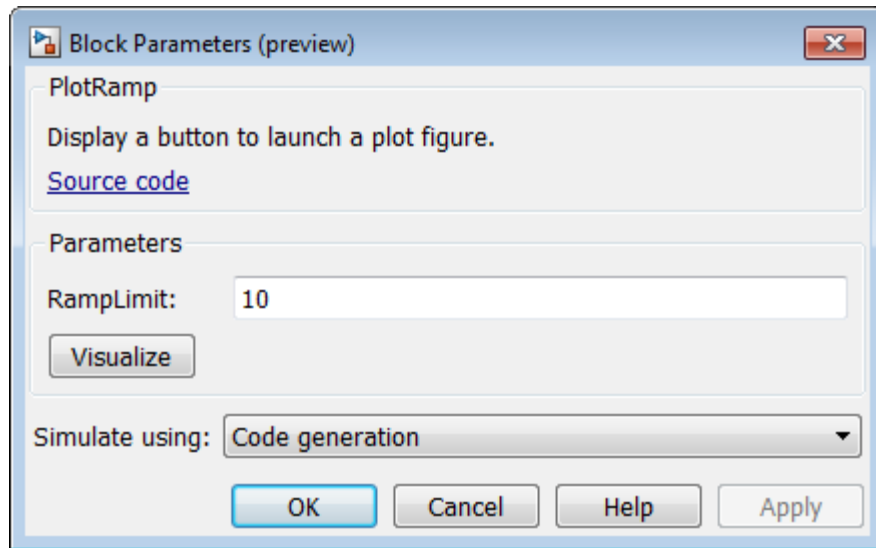
The **System Block** menu also displays checks next to the methods you have implemented, which can help you track your development.
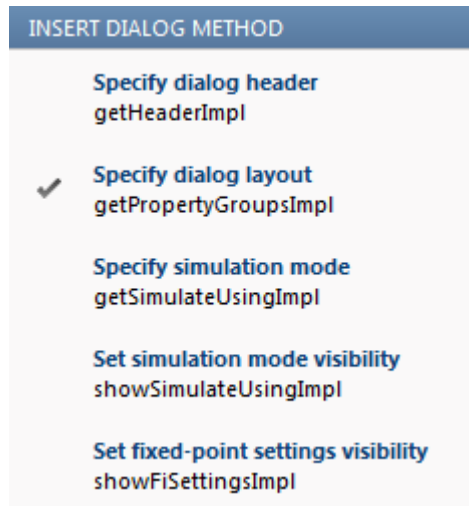
The class definition file now has all the code necessary for the PlotRamp System object.

```matlab
classdef PlotRamp < matlab.System
  % Display a button to launch a plot figure.

  properties (Nontunable)
    RampLimit = 10;
  end

  methods(Static, Access=protected)
    function group = getPropertyGroupsImpl
      group = matlab.system.display.Section(mfilename('class'));
      group.Actions = matlab.system.display.Action(@(~,obj)...
            visualize(obj),'Label','Visualize');
    end
  end

  methods
    function visualize(obj)
      figure;
      d = 1:obj.RampLimit;
      plot(d);
    end
  end
end
```

After you complete your System block definition, save it, and then load it into a MATLAB System block in Simulink.

Related Examples

- "Insert System Object Code Using MATLAB Editor" on page 14-65

Use Enumerations in System Objects

Enumerated data is data that is restricted to a finite set of values. To use enumerated data in a System object in MATLAB or Simulink, you refer to them in your System objectclass definition and define your enumerated class in a separate class definition file.

For a System object that will be used in MATLAB only, see "Enumerations".

For a System object that will be used in a MATLAB System block in Simulink, see "Enumerated Data"

Enumerations can derive from any integer type smaller than or equal to an `int32`. For example,

```
classdef Bearing < uint8
   enumeration
      North (0)
      East  (90)
      South (180)
      West  (270)
   end
end
```

Enumerations can also derive from `Simulink.IntEnumType`. You use this type of enumeration to add attributes, such as custom headers, to the input or output of the MATLAB System block. See "Use Enumerated Data in Simulink Models".

Use Global Variables in System Objects

Global variables are variables that you can access in other MATLAB functions or Simulink blocks.

System Object Global Variables in MATLAB

For System objects that are used only in MATLAB, you define global variables in System object class definition files in the same way that you define global variables in other MATLAB code (see "Global Variables").

System Object Global Variables in Simulink

For System objects that are used in the MATLAB System block in Simulink, you also define global variables as you do in MATLAB. However, to use global variables in Simulink, you need to declare global variables in the , , or method if you have declared them in methods called by `stepImpl`, `updateImpl`, or `outputImpl`, respectively.

You set up and use global variables for the MATLAB System block in the same way as you do for the MATLAB Function block (see "Data Stores" and "Share Data Globally"). Like the MATLAB Function block, you must also use variable name matching with a Data Store Memory block to use global variables in Simulink.

For example, this class definition file defines a System object that increments the first row of a matrix by 1 at each time step. You must include if the class file is P-coded.

```
classdef GlobalSysObjMatrix < matlab.System
   methods (Access = protected)
      function y = stepImpl(obj)
         global B;
         B(1,:) = B(1,:)+1;
         y = B;
      end

      % Include getGlobalNamesImpl only if the class file is P-coded.
      function globalNames = getGlobalNamesImpl(~)
         globalNames = {'B'};
      end
   end
end
```

This model includes the `GlobalSysObjMatrix` object in a MATLAB System block and the associated Data Store Memory block.

Block Parameters: Data Store Memory

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

| Main | Signal Attributes | Diagnostics | Logging |

Data store name: B   Rename All...

Corresponding Data Store Read/Write blocks: refresh
    globalMatrix/MATLAB System

OK   Cancel   Help   Apply

Block Parameters: Data Store Memory

DataStoreMemory

Define a memory region for use by the Data Store Read and Data Store Write blocks. All Read and Write blocks that are in the current (sub)system level or below and have the same data store name will be able to read from or write to this block.

| Main | Signal Attributes | Diagnostics | Logging |

Initial value: [74 75 5 1; 22 23 24 32; 33 34 55 22]

Minimum: []   Maximum: []

Data type: int8   >>

☐ Lock output data type setting against changes by the fixed-point tools

Dimensions (-1 to infer from Initial value): -1

☑ Interpret vector parameters as 1-D

Signal type: real

☐ Data store name must resolve to Simulink signal object

Code Generation

Signal object class: Simulink.Signal

Storage class: SimulinkGlobal

Alias:

Alignment: -1

OK   Cancel   Help   Apply