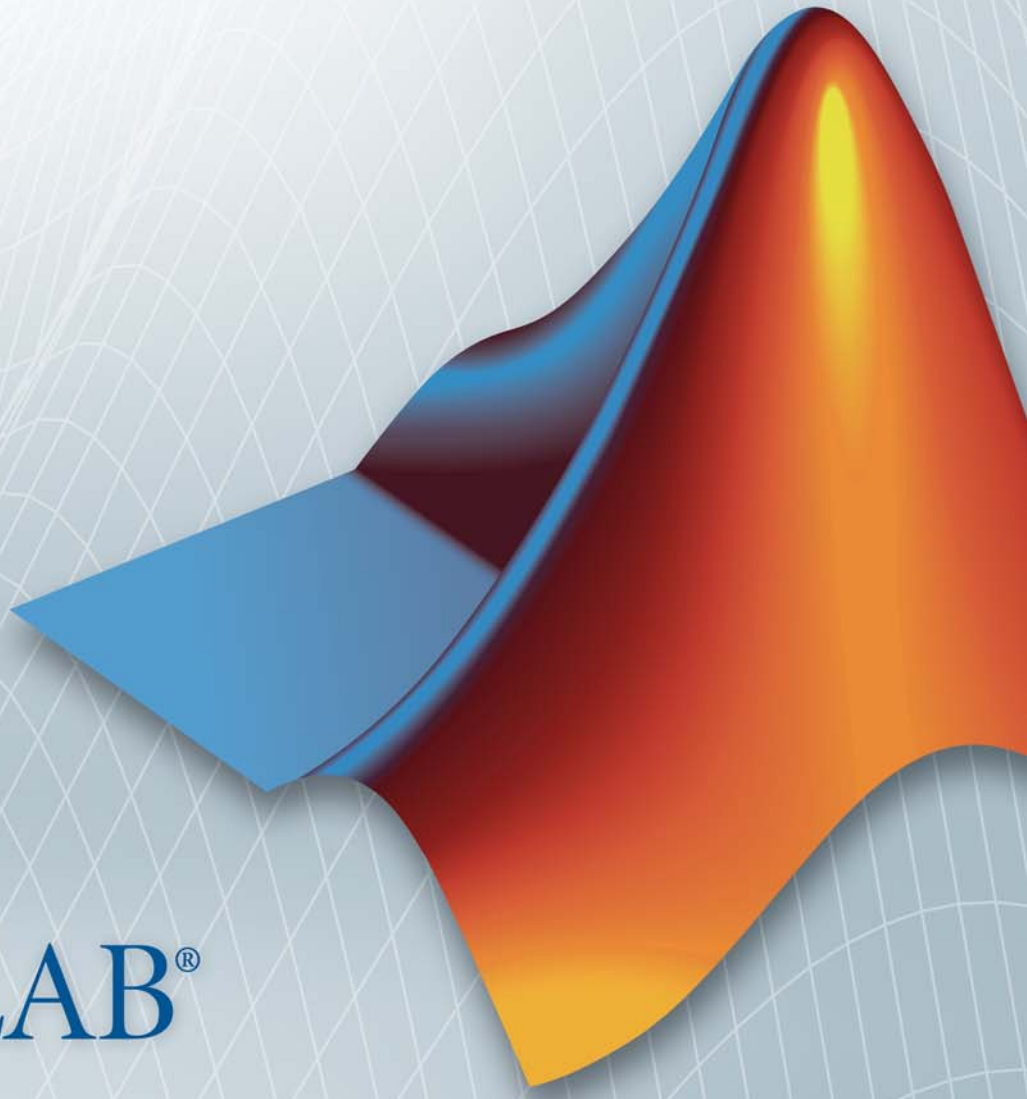
Phased Array System Toolbox™ 1

User's Guide



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Phased Array System Toolbox™ User's Guide

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

April 2011 Online only Revised for Version 1.0 (R2011a)

Contents

1 Phased Arrays

2 Waveforms, Transmitter, and Receiver

3 Beamforming

4 Direction of Arrival (DOA) Estimation

5 Space-Time Adaptive Processing (STAP)

6 Detection

7 Environment and Target Models

8 Coordinate Systems and Motion Modeling

1

Phased Arrays

Antenna and Microphone Elements

In this section...

Isotropic Antenna Element

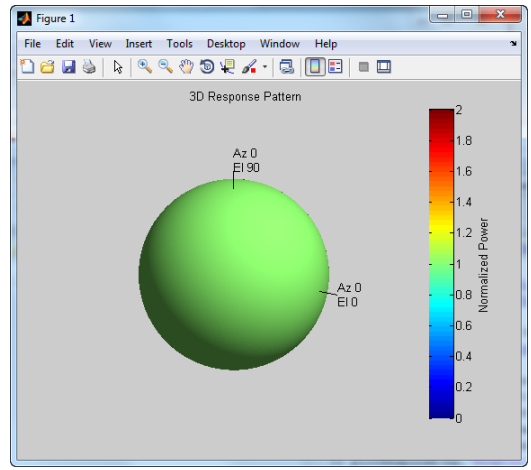
An isotropic antenna element radiates equal power in all non-baffled directions. To construct an isotropic antenna, use `phased.IsotropicAntennaElement`.

The isotropic antenna element object has two modifiable properties:

- `FrequencyRange` — The operating frequency range of the antenna.
- `BackBaffled` — A logical property indicating whether the response of the antenna is baffled at azimuth angles outside the interval [-90,90].

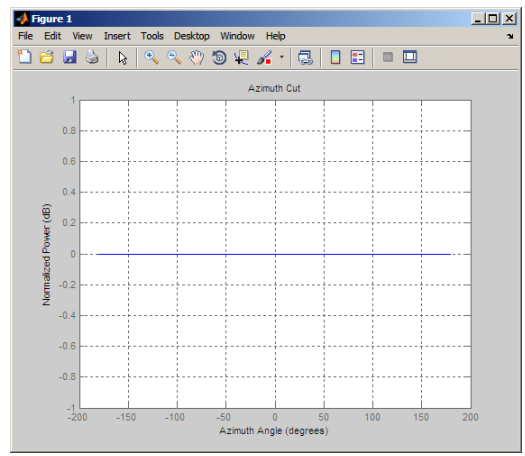
Construct an isotropic antenna element with a uniform frequency response over azimuth angles from [-180,180] degrees and elevation angles from [-90,90] degrees. See "Rectangular and Spherical Coordinates" on page 8-2 for how the toolbox defines azimuth and elevation angles. The antenna operates between 300 megahertz (MHz) and 1 gigahertz (GHz). Plot the antenna response at 1 GHz.

```
ha = phased.IsotropicAntennaElement('FrequencyRange',[3e8 1e9],...
    'BackBaffled',false)
plotResponse(ha,1e9,'RespCut','3D','Format','Polar','Unit','pow');
```



`plotResponse` is a method of `phased.IsotropicAntennaElement`. By default, `plotResponse` plots the response of the antenna element in decibels (dB) at zero degrees elevation.

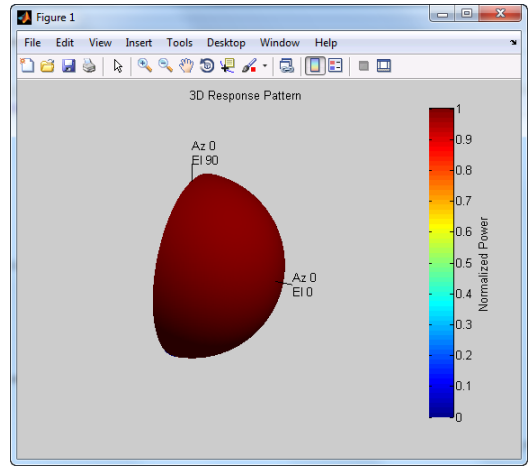
```
plotResponse(ha,1e9);
```



Setting the `BackBaffled` property to `true` limits the response to azimuth angles in the interval [-90,90].

```
ha = phased.IsotropicAntennaElement('FrequencyRange',[3e8 1e9],...
```

```
    'BackBaffled',true);
plotResponse(ha,1e9,'RespCut','3D','Format','Polar','Unit','pow');
```



You can find your isotropic antenna element's voltage response at specific frequencies and angles using the antenna element's `step` method.

Design Backbaffled Isotropic Antenna Element and Obtain Element Response

Construct an isotropic antenna element to operate in the IEEE® X band between 8 and 12 GHz. Backbaffle the response of the antenna. Obtain your antenna element's response at 4 GHz intervals between 6 and 14 GHZ and at azimuth angles between [-100,100] in 50 degree increments.

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

The antenna response in `anresp` is a matrix with its 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 two and last two columns of `anresp` is zero because those columns contain the antenna response at 6 and 14 GHz respectively. These frequencies are not included in the antenna's operating frequency range.

Similarly, the first and last rows of `anresp` contain all zeros because the `BackBaffled` property is set to `true`. The first and last row contain the antenna's response at azimuth angles outside of [-90,90].

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

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

Note that `anresp(1,2)` and `anresp(5,2)` represent the antenna voltage response at the aziumth-elevation pairs (-100,-90) and (100,90). These responses are equal to 1 even though the `BackBaffled` property is `true` because the elevation angles are equal to +/- 90 degrees. This results in the elevation cut degenerating into a point.

Cosine Antenna Element

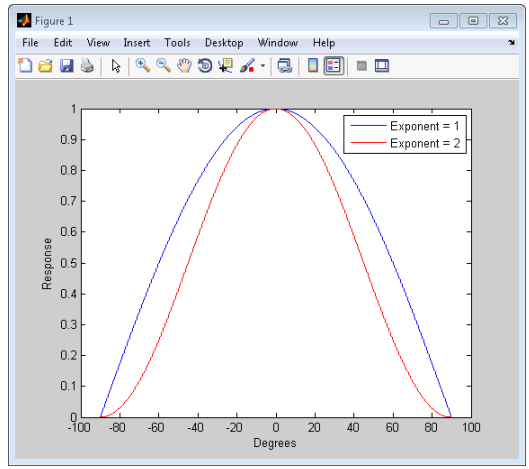
The `phased.CosineAntennaElement` object models an antenna element with a cosine response raised to a specified power in azimuth and elevation. The power response pattern of the cosine antenna is:

$$P(\theta,\phi) = \cos^m(\theta)\cos^n(\phi) \quad m,n \geq 1$$

where θ and φ are the azimuth and elevation angles respectively. The cosine response pattern achieves a maximum value of 1 at 0 degrees azimuth and elevation. The response pattern has zeros at -90 and 90 degrees in azimuth and elevation. The cosine antenna element response is identically zero for azimuth and elevation angles outside of [-90,90] degrees. Raising the response pattern to powers greater than one concentrates the response in aziumth or elevation.

To illustrate this effect, the following example returns the cosine response with powers equal to 1 and 2 for a single angle between -90 and 90 degrees.

```
theta = -90:.01:90;
Cos1 = cosd(theta);
Cos2 = Cos1.^2;
plot(theta,Cos1); hold on;
plot(theta,Cos2,'r');
legend('Exponent = 1','Exponent = 2','location','northeast');
xlabel('Degrees'); ylabel('Response');
```



The modifiable properties of the `phased.CosineAntennaElement` object are:

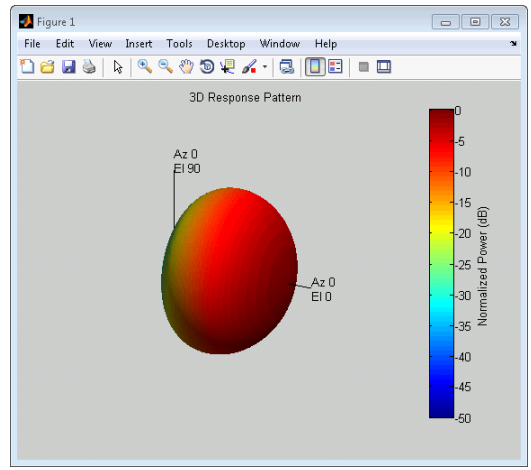
- `FrequencyRange` — The operating frequency range of the antenna
- `CosinePower` — Exponent of cosine pattern

Cosine Antenna Element Operating from 1 to 10 GHz

Construct an antenna with a cosine squared response in both azimuth and elevation and plot the antenna response. The operating frequency range of the antenna is from 1 to 10 GHz.

```
hcos = phased.CosineAntennaElement('FrequencyRange',[1e9 1e10],...
    'CosinePower',[2 2])
```

```
plotResponse(hcos,5e9,'RespCut','3D','Format','Polar');
```



Custom Antenna Element

The `phased.CustomAntennaElement` object enables you to model a custom antenna element.

The modifiable properties of the custom antenna element object are:

- `AzimuthAngles`— Azimuth angles where the custom response is evaluated. The azimuth angles must lie between [-180,180] degrees and you must specify at least three azimuth angles.
- `ElevationAngles`— Elevation angles where the custom response is evaluated. The elevation angles must lie between [-90,90] degrees and you must specify at least three elevation angles.
- `FrequencyVector`— Operating frequency vector for the antenna element
- `FrequencyResponse`— Frequency response of the element in dB at the frequencies in `FrequencyVector`.
- `RadiationPattern`— The magnitude radiation pattern in dB. This is the spatial response of the antenna. `RadiationPattern` is a Q-by-P matrix, where Q is the number of elements in the `ElevationAngles` property and P is the number of elements in the `AzimuthAngles` property.

For your custom antenna element, the antenna response (the output of `step`) depends on the values of the `FrequencyResponse` and `RadiationPattern` properties.

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.

To illustrate this process, construct a simple custom antenna element object. The radiation pattern is constant over each azimuth angle and has a cosine pattern for the elevation angles.

```
Az = -180:90:180;
El = -90:45:90;
Elresp = cosd(El);
ha = phased.CustomAntennaElement('AzimuthAngles',Az,...
    'ElevationAngles',El,'RadiationPattern',repmat(Elresp',1,numel(Az)));
ha.RadiationPattern
```

Use the `step` method to calculate the antenna response at the azimuth-elevation pairs `[-30 0; -45 0]`; for a frequency of 500 MHz.

```
ANG = [-30 0; -45 0];
resp = step(ha,5e8,ANG)
```

The following illustrates the nearest-neighbor interpolation method used to find the antenna voltage response.

```
G = interp2(degtorad(ha.AzimuthAngles),...
                degtorad(ha.ElevationAngles),...
                db2mag(ha.RadiationPattern),...
                degtorad(ANG(1,:))', degtorad(ANG(2,:))','nearest',0);
H = interp1(ha.FrequencyVector,db2mag(ha.FrequencyResponse),...
                5e8,'nearest',0);
antresp = H.*G
```

Compare the value of `antresp` to the output of the `step` method.

Omnidirectional Microphone

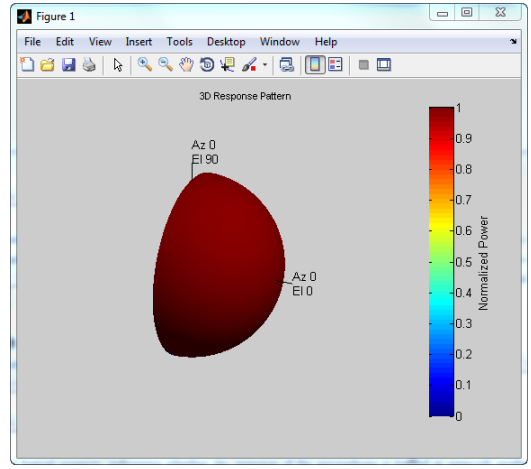
An omnidirectional microphone has a response which is equal to one in all non-baffled directions. The modifiable properties of the `phased.OmnidirectionalMicrophoneElement` object are:

- `FrequencyRange`— The operating frequency range of the microphone.
- `BackBaffled`— A logical property indicating whether the response of the microphone is baffled at azimuth angles outside the interval [-90,90].

Backbaffled Omnidirectional Microphone with Frequency Response from 20 Hz to 20 kHz

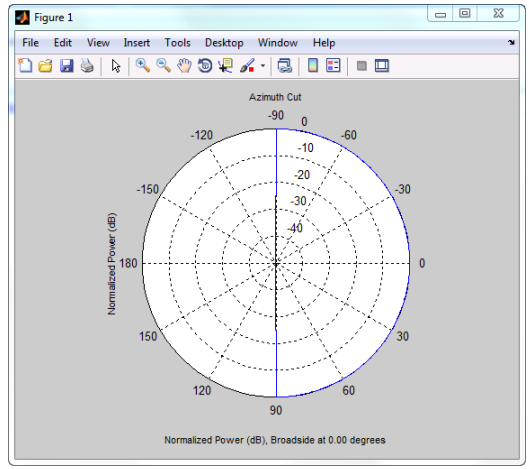
Construct an omnidirectional microphone element using the human audible frequency range of 20 to 20,000 Hz. Baffle the microphone response for azimuth angles outside of +/- 90 degrees. Plot the microphone's power response at 1000 Hz in polar form.

```
hmic = phased.OmnidirectionalMicrophoneElement('BackBaffled',true,...
    'FrequencyRange',[20 20e3]);
plotResponse(hmic,1e3,'RespCut','3D','Format','Polar','Unit','pow');
```



In many applications, it is common to examine the microphone's directionality, or polar pattern. You can view this by specifying `'RespCut'` as one of the 2D options: `'Az'` (default), or `'El'` and specifying the `'Format'` as `"Polar"`.

```
% Using the default azimuth cut
plotResponse(hmic,1e3,'Format','Polar');
```



Use `step` to obtain the microphone's magnitude response at the specified azimuth angles and frequencies. The elevation angles are 0 degrees. Note the response is one at all azimuth angles and frequencies as expected.

```
freq = 100:250:1e3;
ang = -90:30:90;
micresp = step(hmic,freq,ang)
```

Custom Microphone Element

You can model a microphone with your custom response using `phased.CustomMicrophoneElement`. The modifiable properties of the `phased.CustomMicrophoneElement` object:

- `FrequencyVector`— The frequencies where you specify your response.
- `FrequencyResponse`— The frequency response in decibels corresponding to the frequencies specified in the `FrequencyVector`.
- `PolarPatternFrequencies`— The frequencies at which the microphone's polar pattern is measured. The polar pattern frequencies must lie within the frequency range specified in `FrequencyVector`.

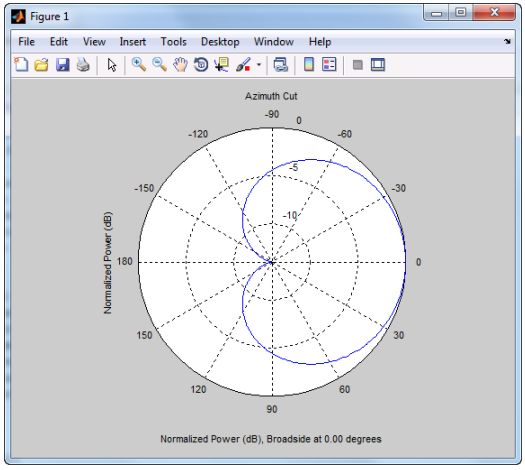
- `PolarPatternAngles`— The angles at which the microphone's polar pattern are measured.
- `PolarPattern`— An M-by-N matrix containing the microphone's magnitude response in decibels. The row dimension, M, is the number of frequencies in `PolarPatternFrequencies`. The column dimension, N, is the number of angles in `PolarPatternAngles`.

Microphone with Cardioid Response Pattern

Construct a microphone element with a cardioid response pattern. Use the default `FrequencyVector` of `[20 20e3]`. Specify the polar pattern frequencies as `[500 1000]`.

Plot the microphone's polar pattern.

```
hmic = phased.CustomMicrophoneElement('PolarPatternFrequencies',[500 1000
hmic.PolarPattern= mag2db([...
            0.5+0.5*cosd(hmic.PolarPatternAngles);...
            0.6+0.4*cosd(hmic.PolarPatternAngles)]);
plotResponse(hmic,1e3,'Format','Polar');
```



Calculate the microphone's response at 30 degrees and 60 degrees azimuth with corresponding elevation angles of 0 degrees.

```
micresp = step(hmic,1e3,[30 60])
```

Array Geometries and Analysis

In this section...

Uniform Linear Array

The uniform linear array (ULA) arranges sensor elements along a line in space with uniform spacing. You can design a ULA with `phased.ULA`. The modifiable properties of `phased.ULA` are:

- `Element`— The sensor elements of the array
- `ElementSpacing`—The spacing between array elements in meters
- `NumElements`— The number of elements in the ULA

Create a ULA with two isotropic antenna elements separated by 0.5 meters:

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5)
```

You can return the coordinates of the array sensor elements in the form `[x;y;z]` by using the `getElementPosition` method. See "Rectangular and Spherical Coordinates" on page 8-2 for toolbox conventions.

```
sensorpos = getElementPosition(hula)
```

`sensorpos` is a 3-by-2 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.

The default element for a ULA is the `phased.IsotropicAntennaElement` object. You can specify an alternative element with the `Element` property. For more information on constructing antenna and microphone objects, see "Antenna and Microphone Elements" on page 1-2.

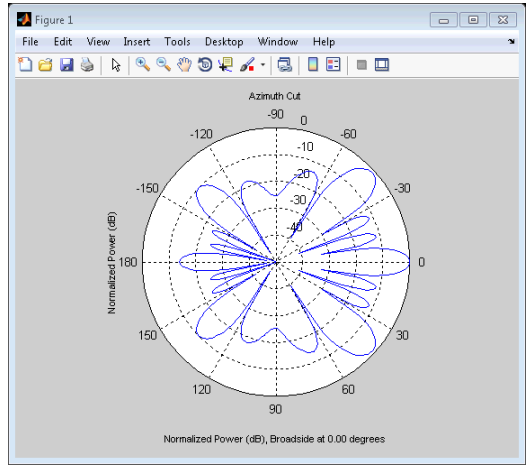
Four-element ULA with Cardioid Microphone Elements

Construct a four-element ULA with custom cardioid microphone elements.

```
% Specify the microphone element
hmic = phased.CustomMicrophoneElement;
hmic.PolarPatternFrequencies = [500 1000];
hmic.PolarPattern = mag2db([...
          0.5+0.5*cosd(hmic.PolarPatternAngles);...
          0.6+0.4*cosd(hmic.PolarPatternAngles)]);
% Assign the custom microphone element as the Element property
ha = phased.ULA('NumElements',4,'ElementSpacing',0.5,'Element',hmic);
```

To view the response of an array, you can use the `plotResponse` method.

```
plotResponse(ha,1e3,340,'Format','Polar')
```



By default, `plotResponse` shows the array's normalized power response in decibels (dB) at zero degrees elevation for azimuth angles from [-180,180]. You can view azimuth cuts of the array response from various elevation angles and elevation cuts from various azimuth angles by specifying name-value pairs in `plotResponse`.

To obtain the responses of your array elements, you can use the array's `step` method.

```
% Construct antenna for the array elements
hant = phased.IsotropicAntennaElement('FrequencyRange',[3e8 1e9])
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5,...
    'Element',hant)
% Obtain element responses at 1 GHz
freq = 1e9;
% for azimuth angles from -180:180
azangles = -180:180;
% elementresponses
elementresponses = step(hula,1e9,azangles);
```

`elementresponses` is a 2-by-361 matrix where each column contains the element responses for the 361 azimuth angles. Because the elements of the ULA are isotropic antennas, `elementresponses` is a matrix of ones.

You can use the `phased.ElementDelay` and `phased.SteeringVector` objects to obtain important information about your array.

To determine the signal delay in seconds between array elements, use `phased.ElementDelay`. The incident waveform is assumed to satisfy the far-field assumption.

The following example computes the delay between elements of a 4–element ULA for a signal incident on the array from –90 degrees azimuth and zero degrees elevation. The delays are computed with respect to the phase center of the array. By default, `phased.ElementDelay` assumes that the incident waveform is an electromagnetic wave propagating at the speed of light.

```
% Construct 4-element ULA using value-only syntax
hula = phased.ULA(4)
hdelay = phased.ElementDelay('SensorArray',hula);
tau = step(hdelay,[-90;0]);
```

`tau` is a 4-by-1 vector of delays with respect to the phase center of the array, which is the origin of the local coordinate system `[0;0;0]`. See “Global and Local Coordinate Systems” on page 8-14 for a description of global and local coordinate systems. Negative delays indicate that the signal is incident on an element before it reaches the phase center of the array. Because the waveform arrives from an azimuth angle of –90 degrees, the signal impinges

on the first and second elements of the ULA before it reaches the phase center resulting in negative delays.

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

```
tau = step(hdelay,[0;0]);
```

If the incident signal is an acoustic pressure waveform propagating at the speed of sound, you can calculate the element delays by specifying the `PropagationSpeed` property.

```
hdelay = phased.ElementDelay('SensorArray',hula,...
    'PropagationSpeed',340)
tau = step(hdelay,[90;0]);
```

In the preceding code, the propagation speed is set to 340 meters per second, which is the approximate speed of sound at sea level.

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:

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

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

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

```
hula = phased.ULA(4)
hsv = phased.SteeringVector('SensorArray',hula)
sv = step(hsv,1e9,[45; 10])
```

You can obtain the steering vector with the following equivalent code.

```
hdelay = phased.ElementDelay('SensorArray',hula);
tau = step(hdelay,[45;10]);
exp(-1j*2*pi*1e9*tau)
```

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

Construct a two-element ULA with elements spaced at 0.5 meters. Obtain the array's magnitude response (absolute value of the complex-value array response) for aziumth angles `-180:180` and plot the normalized magnitude response in dB.

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
har = phased.ArrayResponse('SensorArray',hula);
resp = abs(step(har,1e9,azangles));
plot(azangles,mag2db((resp/max(resp)))); grid on;
title('Azimuth cut at zero degrees elevation');
xlabel('azimuth angle (in degrees)');
```

Compare the figure produced by the preceding code to the figure obtained using the `plotResponse` method.

```
plotResponse(hula,1e9,physconst('lightspeed'))
```

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

`collectPlaneWave` 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. `collectPlaneWave` does not account for the response of individual elements in the array.

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]`.

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

The above code is equivalent to the following.

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

Uniform Rectangular Array

You can implement a uniform rectangular array (URA) with `phased.URA`. The modifiable properties of `phased.URA` are:

- `Element` — The sensor elements of the array.
- `ElementSpacing` — The spacing between the array elements in meters. You can specify the element spacing separately for the row and column dimensions of your URA.
- `Size` — The size of the URA as a row vector. The array elements are distributed in the YZ plane with the array look direction along the positive X axis.

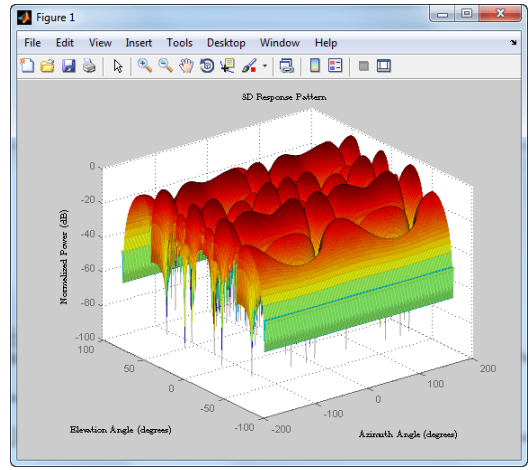
Create a six-element URA with two elements along the X axis and three elements along the Z axis. Use the default spacing of 0.5 meters along both the row and column dimensions of the array. Return the positions of the array elements.

```
hura = phased.URA([2 3])
getElementPosition(hura)
```

Note that the X coordinate is zero for all elements in the array.

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

```
% Plot the response in 3D
plotResponse(hura,1e9,physconst('LightSpeed'),'RespCut','3D')
```



Calcuate the element delays for signals arriving from +/– 45 degrees azimuth and zero degrees elevation.

```
hed = phased.ElementDelay('SensorArray',hura);
ang = [45 -45];
tau = step(hed,ang)
```

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

The next example simulates the reception of two sine waves arriving from far field sources. One of the signals is a 100–Hz sine wave arriving from 20 degrees azimuth and 10 degrees elevation. The other signal is a 300–Hz sine wave arriving from -30 degrees azimuth and 5 degrees elevation. Both signals have a one GHz carrier frequency.

```
t = linspace(0,1,1000);
x = cos(2*pi*100*t)';
y = cos(2*pi*300*t)';
angx = [20; 10];
```

```
angy = [-30;5];
recsig = collectPlaneWave(hura,[x y],[angx angy],1e9);
```

Each column of `recsig` represents the received signal at the corresponding element of the URA, `hura`.

Conformal Array

The `phased.ConformalArray` object provides you with significant flexibility in constructing your phased array. For example, you can use `phased.ConformalArray` to design a planar array with a non-rectangular geometry, such as a circular array. You can also use `phased.ConformalArray` to design non-planar arrays.

The modifiable properties of `phased.ConformalArray` are:

- `Element` — Element of the array
- `ElementPosition` — Element positions
- `ElementNormal` — Element normal directions. Specify the direction normal to the array element in `[azimuth; elevation]` form.

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

```
hcon = phased.ConformalArray
```

Construct a three-element conformal array with elements at `[1;0;0]`, `[0; 1; 0]`, and `[0;-1;0]`. Specify the element normal azimuth angles as 0, 45, and 135 degrees respectively. All normal elevation angles are 0 degrees.

```
ha = phased.ConformalArray;
ha.ElementPosition = [1 0 0; 0 1 -1; 0 0 0];
ha.ElementNormal = [0 45 135; 0 0 0]
```

Uniform Circular Array

Construct a uniform circular array consisting of 60 elements. Assume an operating frequency of 400 MHz. Specify the arc length between the elements

to be 0.5λ where λ is the wavelength of the operating frequency. The element normal directions are equal to `[ang; 0]` where `ang` is the azimuth angle of the array element.

```
% Angle spacing in degrees
theta = 360/60;
% Angle spacing in radians
thetarad = degtorad(theta);
% Arc length 0.5*wavelength of operating frequency
arclength = 0.5*(physconst('LightSpeed')/4e8);
radius = arclength/thetarad;
% Number of elements
N = 60;
% Element angles in degrees
ang = (0:N-1)*theta;
% Azimuth angles must be between [-180,180]
ang(32:end)=ang(32:end)-360;
hcirc = phased.ConformalArray;
hcirc.ElementPosition = [radius*cosd(ang); radius*sind(ang); zeros(1,N)];
hcirc.ElementNormal = [ang; zeros(1,N)];
% Plot the response
plotResponse(hcirc,1e9,physconst('lightspeed'),'Format','Polar')
```

Signal Radiation and Collection

In this section...

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. For information on modeling single antenna or microphone elements, see "Antenna and Microphone Elements" on page 1-2.

To radiate a signal from a sensor array, use `phased.Radiator`. The modifiable properties of `phased.Radiator` are:

- `CombineRadiatedSignals` — A logical property which determines whether the output of all sensor elements is combined.
- `OperatingFrequency` — The operating frequency of the array in hertz.
- `PropagationSpeed` — Propagation speed of the wave in meters per second.
- `Sensor` — Handle of the sensor (single element) or sensor array.
- `WeightsInputPort` — A logical property indicating whether to apply weights to signals radiated by different elements in the array. If you set this property to `true`, input the actual weights when you call the `step` method.

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.

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
hrad = phased.Radiator('Sensor',hula,'OperatingFrequency',3e8,...
    'PropagationSpeed',physconst('lightspeed'),...
```

```
    'CombineRadiatedSignals',true)
% create signal to radiate
x = [1 -1 1 -1]';
% model far field radiation at an angle of [45;0]
y = step(hrad,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 (see `phased.SteeringVector`).

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

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
hsv = phased.SteeringVector('SensorArray',hula,...
    'IncludeElementResponse',true);
sv = step(hsv,3e8,[45;0]);
y1 = x*sum(sv);
% compare y1 to y
```

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 2-16 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 modifiable properties of the narrowband collector, `phased.Collector`, object are:

- `OperatingFrequency` — The operating frequency of the array in hertz.
- `PropagationSpeed` — Propagation speed of the wave in meters per second.
- `Sensor` — Handle of the sensor (single element) or sensor array.
- `Wavefront` — Specifies the type of incoming wave as `'Plane'` or `'Unspecified'`. When you set the `Wavefront`property to `'Plane'`, the input signals are multiple plane waves impinging on the entire array. Each plane wave is received by all collecting elements. If you set the `Wavefront` property to `'Unspecified'`, the input signal are individual waves impinging on individual sensors.
- `WeightsInputPort` — A logical property indicating whether to apply weights to signals collected by different elements in the array. If you set this property to `true`, input the actual weights when you call the `step` method.

Narrowband Collector for Uniform Linear Array

Construct a narrowband collector that models a plane wave impinging on a two-element uniform linear array with an element spacing of 0.5 meters (default ULA). The operating frequency of the array is 300 MHz.

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),...
    'OperatingFrequency',3e8,'Wavefront','Plane')
% create signal to create
x =[1 -1 1 -1]';
% simulate reception from an angle of [45;0]
y = step(hcol,x,[45;0]);
```

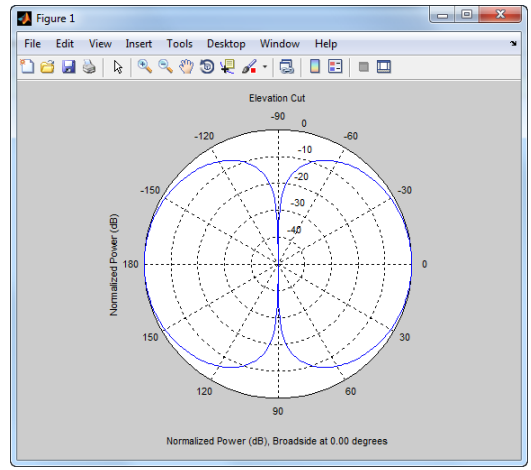
In the preceding case, the collector object multiplies the input signal, `x`, by the corresponding element of the steering vector for the two-element ULA. The following code produces the response in an equivalent manner.

```
% default ULA
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
% Construct steering vector
hsv = phased.SteeringVector('SensorArray',hula);
sv = step(hsv,3e8,[45;0]);
x =[1 -1 1 -1]';
y1 = x*sv.';
% compare y1 to y
```

Narrowband Collector for a Single Antenna Element

The `Sensor` property of `phased.Collector` can consist of a single antenna element. In this example, create a custom antenna element using `phased.CustomAntennaElement`. The antenna element has a cosine response over elevation angles from [-90,90] degrees. Plot the polar pattern response of the antenna at 1 GHz using an elevation cut at zero degrees azimuth. Determine the antenna voltage response at 0 degrees azimuth and 45 degrees elevation.

```
ha = phased.CustomAntennaElement;
ha.AzimuthAngles = -180:180;
ha.ElevationAngles = -90:90;
ha.RadiationPattern = mag2db(repmat(cosd(ha.ElevationAngles)',...
       1,numel(ha.AzimuthAngles)));
plotResponse(ha,1e9,'Format','polar','RespCut','El');
resp = step(ha,1e9,[0; 45])
```



The antenna voltage response at zero degrees azimuth and 45 degrees elevation is `cosd(45)` as expected.

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

```
hc = phased.Collector('Sensor',ha,'OperatingFrequency',1e9)
x =[1 -1 1 -1]';
y = step(hc,x,[0; 45]);
% equivalent to y1 = x*cosd(45);
```

Wideband Signal Collection

Use `phased.WidebandCollector` to model the collection of a wideband input by a sensor element or an array. The wideband collector decomposes the input into subbands and computes the steering vector for each subband. The modifiable properties of the `phased.WidebandCollector` object are:

- `CarrierFrequency` — Carrier frequency in hertz.
- `ModulatedInput` — A logical property indicating whether the signal is demodulated to the baseband.
- `PropagationSpeed` — Propagation speed of the waveform.

- `SampleRate` — Sampling rate
- `Sensor` — Handle to a sensor element or sensor array.
- `Wavefront` — Specifies the type of incoming wave as `'Plane'` or `'Unspecified'`. When you set the `Wavefront`property to `'Plane'`, the input signals are multiple plane waves impinging on the entire array. Each plane wave is received by all collecting elements. If you set the `Wavefront` property to `'Unspecified'`, the input signal are individual waves impinging on individual sensors.
- `WeightsInputPort` — A logical property indicating whether to apply weights to signals radiated by different elements in the array. If you set this property to `true`, input the actual weights when you call the `step` method.

Simulate the reception of a wideband acoustic signal by a single omnidirectional microphone element.

```
x = randn(10,1);
hmic = phased.OmnidirectionalMicrophoneElement('FrequencyRange',...
       [20 20e3],'BackBaffled',true)
hwb = phased.WidebandCollector('Sensor',hmic,'PropagationSpeed',340,...
     'SampleRate',50e3,'ModulatedInput',false)
y = step(hwb,x,[30;10]);
```

2

Waveforms, Transmitter, and Receiver

Waveforms

In this section...

Rectangular Pulse Waveforms

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.

To create a rectangular pulse waveform use `phased.RectangularWaveform` .

Enter the following to construct a rectangular pulse waveform with a duration of 50 microseconds, a sample rate of 1 megahertz (Hz), and a pulse repetition frequency (PRF) of 10 kHz.

```
hrect = phased.RectangularWaveform('SampleRate',1e6,'PulseWidth',5e-5,..
    'PRF',1e4);
```

The rectangular pulse waveform has six modifiable properties:

- `SampleRate` — Sampling rate in Hz
- `PulseWidth` — Pulse duration in seconds

- `PRF` — Pulse repetition frequency in Hz
- `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'`

A rectangular pulse has a bandwidth in Hz that is approximately the reciprocal of its duration. You can determine the bandwidth of the rectangular pulse with:

```
bandwidth(hrect)
% compare to
1/hrect.PulseWidth
```

You can specify the pulse width and sample rate either at construction using Name-Value pairs, or by using the `ObjectHandle.Property` syntax after construction.

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

```
hrect = 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 microseconds.

```
hrect1 = clone(hrect);
hrect.PulseWidth = 10e-6;
```

`hrect1` and `hrect` now specify different rectangular pulses because you changed the pulse width of `hrect`.

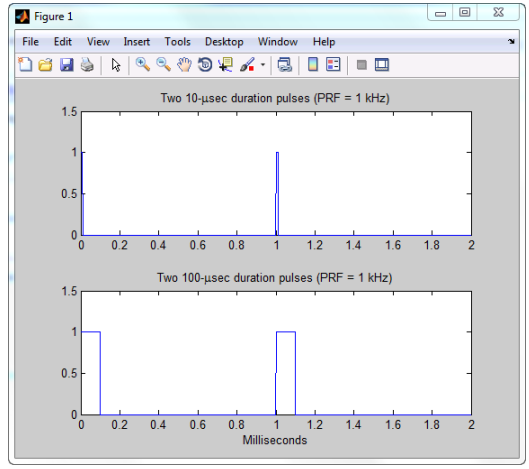
Use the `step` method to return two pulses of your rectangular pulse waveforms.

```
y = step(hrect);
```

```
y1 = step(hrect1);
```

Plot the waveforms.

```
totaldur = 2*1/hrect.PRF;
totnumsamp = totaldur*hrect.SampleRate;
t = unigrid(0,1/hrect.SampleRate,totaldur,'[)');
subplot(2,1,1)
plot(t.*1000,y); axis([0 totaldur*1e3 0 1.5]);
title('Two 10-\musec duration pulses (PRF = 1 kHz)');
subplot(2,1,2);
plot(t.*1000,y1); axis([0 totaldur*1e3 0 1.5]);
xlabel('Milliseconds');
title('Two 100-\musec duration pulses (PRF = 1 kHz)');
```



You can plot a single rectangular pulse by calling `plot` directly on the object handle.

Because `plot` is a method of `phased.RectangularWaveform`, the method knows how to produce an annotated graph of your pulse waveform. To produce an equivalently informative graph of the output of `step`, you must write additional MATLAB® code.

Compare:

```
plot(hrect)
figure;
plot(real(y(1:10)))
```

Linear Frequency Modulated Pulse Waveforms

Increasing the duration of a transmitted pulse increases its energy and improves target detection capability. On the other hand, 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 allow you to decouple the duration of the pulse from its energy by effectively creating different durations for the transmitted pulse and processed echo. The use of a linear frequency modulated pulse waveform is a popular choice for pulse compression.

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 Θ(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 ß/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, ßτ, greater than 1.

To create a linear FM pulse waveform use `phased.LinearFMWaveform` .

The linear FM pulse waveform object has eight modifiable properties:

- `SampleRate` — Sample rate in Hz
- `PulseWidth` — Duration of a single pulse in seconds
- `PRF` — Pulse repetition frequency in Hz
- `SweepBandwidth` — Sweep bandwidth in Hz
- `SweepDirection` — Sweep direction as `'Up'` (default) or `'Down'` corresponding to increasing and decreasing instantaneous frequency.
- `Envelope` — Amplitude modulation of the pulse waveform. `Envelope` can be either `'Rectangular'` (default) or `'Gaussian'`

  The rectangular envelope is:

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

  where τ is the pulse duration.

  The Gaussian envelope is:

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

- `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'`

Linear FM Pulse Waveforms

Enter the following to construct a linear FM pulse with a sample rate of 1 MHz, a pulse duration of 50 microseconds with an increasing instantaneous frequency, and a sweep bandwidth of 100 kHz. The amplitude modulation is rectangular.

```
hfm = phased.LinearFMWaveform('SampleRate',1e6,'PulseWidth',5e-5,...
    'PRF',1e4,'SweepBandwidth',1e5,'SweepDirection','Up',...
    'Envelope','Rectangular','OutputFormat','Pulses','NumPulses',1)
```

Design a linear FM pulse waveform with a duration of 100 microseconds, a bandwidth of 200 kHz, and a PRF of 1 kHz. Use the default values for the other properties. Compute the time-bandwidth product and plot the real part of the pulse waveform.

```
hfm = phased.LinearFMWaveform('PulseWidth',100e-6,...
    'SweepBandwidth',2e5,'PRF',1e3)
% time-bandwidth product
hfm.PulseWidth*hfm.SweepBandwidth
% plot the real part of the pulse
plot(hfm)
```

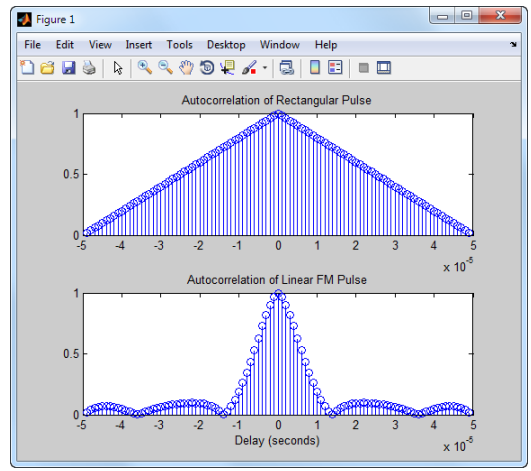
Use the `step` method to obtain your pulse waveform and plot the real and imaginary parts of one pulse repetition interval.

```
y = step(hfm);
t = unigrid(0,1/hfm.SampleRate,1/hfm.PRF,'[)');
subplot(2,1,1)
plot(t,real(y))
axis tight;
title('Real Part');
subplot(2,1,2);
plot(t,imag(y)); xlabel('Seconds');
title('Imaginary Part');
axis tight;
```

Use `ambgfun` to compute the ambiguity function magnitudes for a rectangular and linear FM pulse waveform of the same duration and PRF. Use the zero Doppler cut (magnitudes of the autocorrelation sequences) to illustrate pulse compression in the linear FM pulse waveform.

```
hrect = phased.RectangularWaveform('PRF',2e4)
hfm = phased.LinearFMWaveform('PRF',2e4)
xrect = step(hrect);
xfm = step(hfm);
[ambrect,delay] = ambgfun(xrect,hrect.SampleRate,hrect.PRF,...
    'Cut','Doppler');
```

```
[ambfm,delay] = ambgfun(xfm,hfm.SampleRate,hfm.PRF,...
    'Cut','Doppler');
% Plot the results
subplot(211);
stem(delay,ambrect)
title('Autocorrelation of Rectangular Pulse');
subplot(212);
stem(delay,ambfm)
xlabel('Delay (seconds)');
title('Autocorrelation of Linear FM Pulse');
```



See Waveform Analysis Using the Ambiguity Function for a detailed demo.

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 allowing 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 eight 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'`

Enter the following to construct a stepped FM pulse waveform with a pulse duration (width) of 50 microseconds, a PRF of 10 kHz, and five steps of 20 kHz. The sampling rate is 1 MHz. By default the `OutputFormat` property is equal to `'Pulses'` and the number of pulses in the output is equal to one. The example uses the `bandwidth` method to demonstrate that the bandwidth of the stepped FM pulse waveform is the product of the frequency step and the number of steps `Obj.FrequencyStep*Obj.Numsteps`.

```
hs = phased.SteppedFMWaveform('SampleRate',1e6,'PulseWidth',5e-5,...
    'PRF',1e4,'FrequencyStep',2e4,'NumSteps',5);
bandwidth(hs)
% equal to hs.NumSteps*hs.FrequencyStep
```

Because the `OutputFormat` property is set to `"Pulses'` and the `NumPulses` property is set to 1, calling the step method returns one pulse repetition interval (PRI). The pulse duration within that interval is equal to the `PulseWidth` property. The remainder of the PRI consists of zeros.

The initial pulse has a frequency of zero. It is a DC pulse. With the `NumPulses` property set to 1, each time you use `step`, the frequency of the narrowband pulse increments by the value of the `FrequencyStep` property. If you call `step` more times than the value of the `NumSteps` property, the process repeats, starting over with the DC pulse.

Use `step` to return and plot successively higher frequency pulses. Pause the loop to visualize the increment in frequency with each successive call to `step`. Make a sixth call to `step` to demonstrate that the process starts over with the DC (rectangular) pulse.

```
t = unigrid(0,1/hs.SampleRate,1/hs.PRF,'[)');
for i = 1:hs.NumSteps
    plot(t,real(step(hs)));
    pause(0.5);
    axis tight;
end
% calling step again starts over with a DC pulse
y = step(hs);
```

Waveforms with Staggered PRFs

Utilizing a nonconstant PRF has a number of important applications in radar. This is referred to as *PRF staggering*, or *PRI staggering*.

Uses of staggered PRFs include:

- The 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

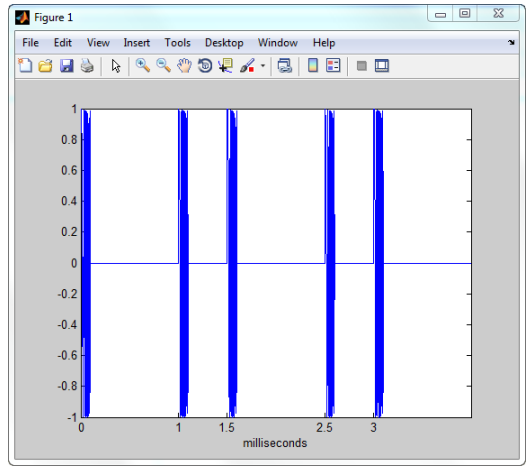
To implement a staggered PRF, the `PRF` property of the waveform objects accepts a vector input.

Linear FM Waveform with Staggered PRF

Model a linear FM pulse waveform with two PRFs, 1 and 2 kHz. Use a linear FM pulse with a sweep bandwidth of 200 kHz and a duration of 100 microseconds. The sample rate is 1 MHz. Output 5 pulses.

```
prfs = [1e3 2e3];
hfm = phased.LinearFMWaveform('PRF',prfs,'SweepBandwidth',200e3,...
    'PulseWidth',100e-6,'NumPulses',5);
wf = step(hfm);
T = length(wf)*(1/hfm.SampleRate);
t = unigrid(0,1/hfm.SampleRate,T,'[)');
plot(t.*1000,real(wf))
```

```
set(gca,'xtick',[0 1 1.5 2.5 3]);
xlabel('milliseconds');
```



Transmitter and Receiver

In this section...

Transmitter

Transmitter Object

The `phased.Transmitter` object enables you to 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 modifiable properties of the transmitter object are:

- `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.

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's status.

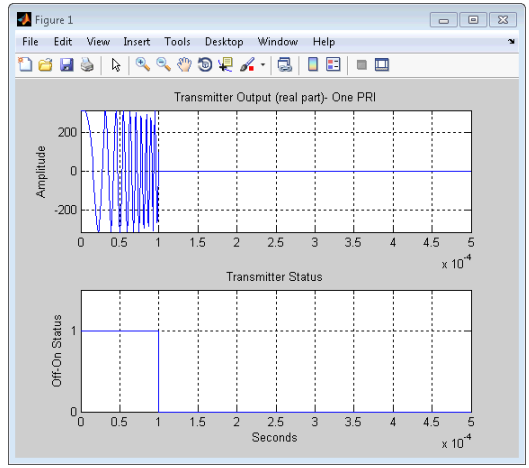
```
htx = phased.Transmitter('PeakPower',1e3,'Gain',20,...
    'LossFactor',0,'InUseOutputPort',true)
```

Construct a pulse waveform for transmission. In this example, use a 100–microsecond linear FM pulse with a bandwidth of 200 kHz. Use the default sweep direction and sample rate. Set the PRF to 2 kHz.

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

Obtain the pulse waveform using the `step` method of the waveform object. Transmit the waveform using the `step` method of the transmitter object, `hpuls`. The output is one pulse repetition interval because the `NumPulses` property of the waveform object is equal to 1. The pulse waveform values are scaled based on the peak transmit power and the ratio of the transmitter gain to loss factor. The scaling factor is `sqrt(htx.PeakPower*db2pow(htx.Gain-htx.LossFactor))`.

```
wf = step(hpuls);
[txoutput,txstatus] = step(htx,wf);
t = unigrid(0,1/hpuls.SampleRate,1/hpuls.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');
```



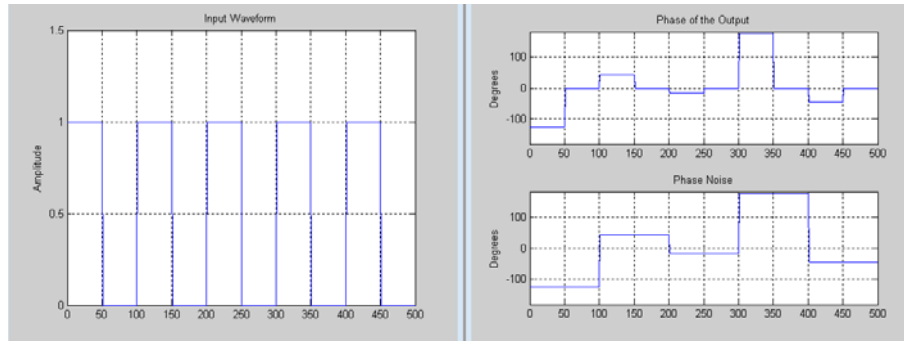
Phase Noise

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 with `step`.

To illustrate this process, the following example uses a rectangular pulse waveform with 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 by the `step` method.

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

```
hrect = phased.RectangularWaveform('NumPulses',5);
htx = phased.Transmitter('CoherentOnTransmit',false,...
    'PhaseNoiseOutputPort',true,'Gain',0,'PeakPower',1,...
    'SeedSource','Property','Seed',1000);
wf = step(hrect);
[txtoutput,phnoise] = step(htx,wf);
phdeg = radtodeg(phnoise);
phdeg(phdeg>180)= phdeg(phdeg>180)-360;
plot(wf); title('Input Waveform');
axis([0 length(wf) 0 1.5]); ylabel('Amplitude');
grid on;
figure;
subplot(2,1,1)
plot(radtodeg(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;
```



The figure on the left shows the waveform. The phase of each pulse at the input to the transmitter is zero. The bottom right figure shows the phase added to each sample. The top right show the phase of the transmitter output waveform. 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 degrees 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 degrees for samples 1–50 and 0 for 51–100. Examining the transmitter output and phase noise for samples where the input waveform is nonzero, you see that the phase output of `step` and the phase of the transmitter output agree.

Receiver Preamp

The `phased.ReceiverPreamp` object enables you to 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.

The `phased.ReceiverPreamp` 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 2-12.
- `Gain` — Gain in dB
- `LossFactor` — Loss factor in dB.
- `NoiseBandwidth` — Bandwidth of the noise spectrum in Hz
- `NoiseFigure` — Receiver noise figure in dB
- `ReferenceTemperature` — Reference temperature of the receiver in kelvin
- `EnableInputPort` — Add input to specify when the receiver is active
- `PhaseNoiseInputPort` — Add input to specify phase noise for coherent on receive receiver
- `SeedSource` — Enables you to specify the seed of the random number generator

The noise figure is a unitless measure, which indicates how much a receiver deviates from an ideal receiver in terms of internal noise. An ideal receiver only produces the expected thermal noise power for a given noise bandwidth

and temperature. A noise figure of 1 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 dB this means that the noise figure must be nonnegative. The minimum possible value is 0 dB.

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 and the noise figure converted to a power measurement with `db2pow`. See `systemp` for details.

`phased.ReceiverPreamp` computes the noise power as product of the Boltzmann constant, the effective system noise temperature, and the noise bandwidth.

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

The signal is scaled by the ratio of the receiver gain to the loss factor expressed as a power ratio. If you express the gain and loss factor as powers by G and L respectively and the noise power as σ^2, the output is equal to :

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

where *x[n]* is the complex-valued input signal and *w[n]* is a zero-mean complex white Gaussian noise sequence.

Model Receiver Effects on Sinusoidal Input

Specify a `phased.ReceiverPreamp` object with a gain of 20 dB, a noise bandwidth of 1 MHz, a noise figure of 0 dB, and a reference temperature of 290 kelvin.

```
hr = phased.ReceiverPreamp('Gain',20,'NoiseBandwidth',1e6,...
     'NoiseFigure',0,'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 all modeling is done in the baseband, use a complex exponential as the input to the `step` method.

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

The output of the `step` method is complex-valued as expected. To demonstrate how this output is produced, the noise power is equal to:

```
noisepow = physconst('Boltzmann')*systemp(0,hr.ReferenceTemperature)*.
    hr.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 G:

$$10\log_{10}(G^2) = 20$$

The scaling factor is 10. You can scale the input signal by a factor of ten and add complex white Gaussian noise with the appropriate variance to produce an output equivalent to the preceding call to `step`. The following code assumes that the noise bandwidth equals the sample rate of the receiver preamp.

```
s = RandStream('mt19937ar','Seed',1e3);
y1 = 10*x+sqrt(noisepow/2)*(randn(s,size(x))+1j*randn(s,size(x)));
```

Compare `y` to `y1`.

Model Coherent on Receive Behavior

To model 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. You can input a record of when the transmitter is active by

setting the `EnableInputPort` to `true` and providing this record to the `step` method.

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. You can input a record of the transmitter phase errors to `step` when you set the `PhaseNoiseInputPort` property to `true`.

To illustrate this, construct a rectangular pulse waveform with five pulses. The PRF is 10 kHz and the pulse width is 50 microseconds. The 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 watt.

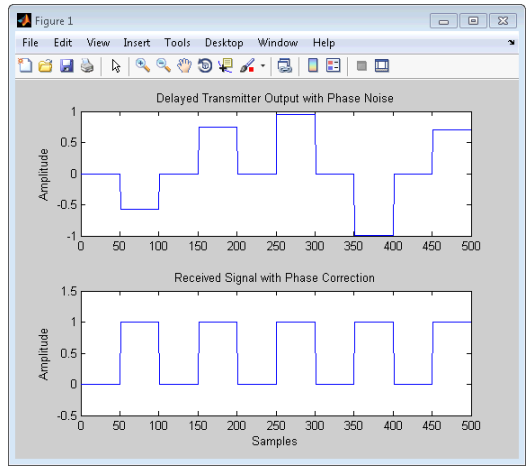
Use the `PhaseNoiseOutputPort` and `InUseOutputPort` properties on the transmitter to record the phase noise and the status of the transmitter.

Enable the `EnableInputPort` and `PhaseNoiseInputPort` properties on 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 `delayseq` to simulate the waveform arriving at the receiver preamp when the transmitter is inactive and the receiver is active.

```
hrect = phased.RectangularWaveform('NumPulses',5);
htx = phased.Transmitter('CoherentOnTransmit',false,...
  'PhaseNoiseOutputPort',true,'Gain',0,'PeakPower',1,...
  'SeedSource','Property','Seed',1000,'InUseOutputPort',true);
wf = step(hrect);
[txtoutput,txstatus,phnoise] = step(htx,wf);
txtoutput = delayseq(txtoutput,hrect.PulseWidth,hrect.SampleRate);
hrc = phased.ReceiverPreamp('Gain',0,'PhaseNoiseInputPort',true,...
    'EnableInputPort',true);
y = step(hrc,txtoutput,~txstatus,phnoise);
subplot(2,1,1)
plot(real(txtoutput));
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

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. The model is deterministic and assumes isotropic radiators. 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 in decibels
- G_r — Receiver gain in decibels. If the radar is monostatic, the transmitter and receiver gains are identical.
- λ — Radar operating frequency wavelength in meters
- σ — Target's nonfluctuating radar cross section in square meters
- L — General loss factor in decibels that accounts 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 factor*.

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.

Determine the Required Transmitter Peak Power Using the Radar Equation

You implement a noncoherent detector with a monostatic radar operating at 5 GHz. Based on the noncoherent integration of ten one-microsecond 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 1 square meter at 30 kilometers. 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)
```

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

```
tgtrng = 30e3;  % target range in meters
lambda = 3e8/5e9; % wavelength of the operating frequency
RCS = 1;   % target RCS
pulsedur = 1e-6;  %pulse duration
G = 30;  % transmitter and receiver gain (monostatic radar)
Pt = radareqpow(lambda,tgtrng,SNR,pulsedur,'rcs',RCS,'gain',G)
```

The required peak power is approximately 5.6 kilowatts.

Determine Maximum Detectable Range for a Monostatic Radar

Assume that the minimum detectable SNR at the receiver of a monostatic radar operating at 1GHz is 13 dB. Use the radar equation to determine the maximum detectable range for a target with a nonfluctuating RCS of 0.5 square meters if the radar has a peak transmit power of 1 megawatt. Assume the transmitter gain is 40 dB and the radar transmits a pulse that is 0.5 microseconds in duration.

```
tau = 0.5e-6;  % pulse duration
G = 40; % transmitter and receiver gain (monostatic radar)
RCS = 0.5;  % target RCS
Pt = 1e6;  %peak transmit power in watts
lambda = 3e8/1e9;
SNR = 13;  % required SNR in dB
maxrng = radareqrng(lambda,SNR,Pt,tau,'rcs',RCS,'gain',G)
```

The maximum detectable range is approximately 345 km.

Estimate Output SNR at the Receiver in a Bistatic Radar

Estimate the output SNR for a target with an RCS of 1 square meter. The radar is bistatic. The target is located 50 kilometers from the transmitter and 75 kilometers from the receiver. The radar operating frequency is 10 gigahertz. The transmitter has a peak transmit power of 1 megawatt with a gain of 40 decibels. The pulse width is 1 microsecond. The receiver gain is 20 decibels.

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

The estimated SNR is approximately 9 dB.

3

Beamforming

Conventional Beamforming

Assume N sensors, which are linear and time-invariant. Let $x_k(t)$ denote the space-time wavefield in continuous time at some reference point. For convenience, you can use the array center as the reference point. You can model the output signal, $y_k(t)$, at the k-th sensor as:

$$y_k(t) = h_k(t)x_k(t-\tau_k) + \varepsilon_k(t)$$

where $h_k(t)$ is the impulse response of the k-th sensor, $x_k(t\text{-}\tau_k)$ is the delayed space-time signal at the k-th sensor, and $\varepsilon_k(t)$ is the noise at the k-th sensor. The noise term includes contributions both external and internal to the sensor. The delay term, τ_k, is the model parameter that characterizes the source location with respect to the array.

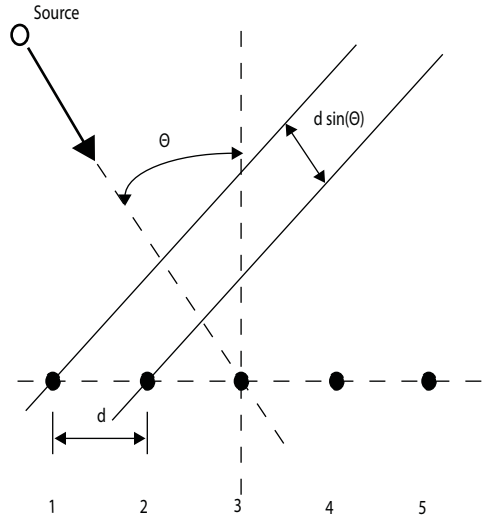
For a narrowband signal modulated around a carrier frequency, ω_c, the time delay at the k-th sensor is equivalent to a phase shift:

$$e^{-j\omega_c\tau_k}$$

Utilizing the plane wave assumption and knowledge of the array geometry, you can sometimes derive an explicit expression for the delay term. Consider the case of a uniform linear array (ULA) with a plane wave incident on the array from a broadside angle of θ. Taking sensor 1 as the reference point, the delay at the k-th sensor is:

$$\tau_k = \frac{(k-1)d\sin(\theta)}{c}$$

where c is the propagation speed of the wave and d is the distance between array elements. The following figure illustrates this scenario.



Assuming the sensors' frequency responses at the carrier frequency are identical, you can write the array transfer vector for the ULA as:

$$a(\theta) = \begin{pmatrix} 1 \\ e^{-jd\sin(\theta)/c} \\ e^{-j2d\sin(\theta)/c} \\ \vdots \\ e^{-j(N-1)d\sin(\theta)/c} \end{pmatrix}$$

By choosing an appropriate set of spatial filter weights, you can *steer* the array to accentuate or attenuate specific angles of arrival.

The filter weights introduce phase shifts (delays) in the sensor output so that when the individual sensor outputs are summed, the array output emphasizes a waveform incident from a particular angle of arrival.

You can implement a narrowband phase shift beamformer with `phased.PhaseShiftBeamformer`.

The phased.PhaseShiftBeamformer object has six modifiable properties.

- SensorArray — Handle to a sensor array
- PropagationSpeed — Signal propagation speed
- OperatingFrequency — System operating frequency
- DirectionSource — Source of the beamforming direction. You can set this property to either 'Property' (default) or 'InputPort'. When the DirectionSource is equal to 'Property', the beamforming direction is the value of the Direction property. When the DirectionSource is 'InputPort', you supply the beamforming direction as an input to the step method.
- Direction — Beamforming direction. This property only applies when the DirectionSource property is 'Property'.
- WeightsOutputPort — Logical property indicating whether or not to output the beamforming weights

Narrowband (phase shift) Beamformer with a ULA

Construct a ULA with 10 elements. Assume the carrier frequency is 1 GHz and set the array element spacing to be 1/2 the carrier frequency wavelength.

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

The ULA sensors are isotropic antenna elements (see phased.IsotropicAntennaElement). Set the frequency range of the antenna elements to position the carrier frequency in the middle of the operating range.

```
hula.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;
```

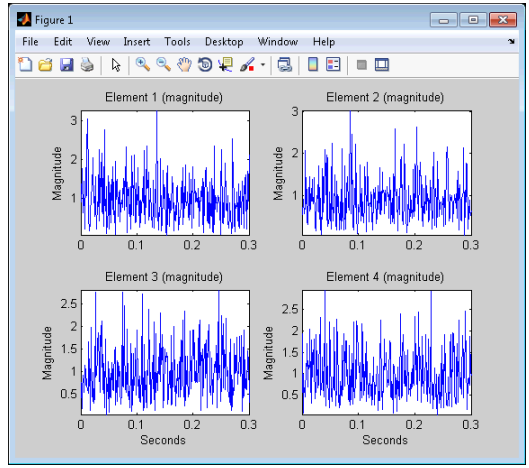
Assume the rectangular pulse is incident on the ULA from an angle of 30 degrees azimuth and 0 degrees elevation. Use the `collectPlaneWave` method of the ULA object to simulate reception of the pulse waveform from the specified angle.

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

`x` is a matrix with ten columns with each column representing the received signal at one of the array elements.

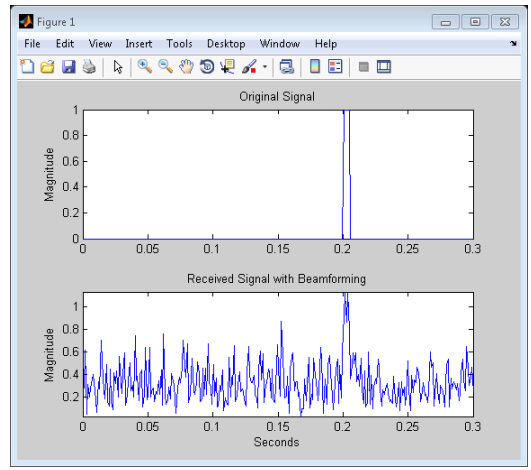
Corrupt the columns of `x` with complex-valued Gaussian noise. Reset the default random number stream for reproducible results. Plot the magnitudes of the received pulses at the first four elements of the ULA.

```
rng default
npower = 0.5;
x = x + sqrt(npower/2)*randn(size(x))+1j*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 your phase-shift beamformer. Set the `WeightsOutputPort` property to `true` to output the spatial filter weights. Apply the `step` method for the phase shift beamformer. The `step` method computes and applies the correct weights for the specified angle. The phase-shifted outputs from the ten array elements are summed. Plot the magnitude of the output waveform along with the original waveform for comparison.

```
hbf = phased.PhaseShiftBeamformer('SensorArray',hula,...
    'OperatingFrequency',1e9,'Direction',angle_of_arrival,...
    'WeightsOutputPort',true);
[y,w] = step(hbf,x);
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 the beamforming weights on the array response, plot the array normalized power response with and without the beamforming weights.

```
azang = -180:30:180;
subplot(211)
plotResponse(hula,fc,physconst('LightSpeed'));
set(gca,'xtick',azang);
title('Array Response without Beamforming Weights');
subplot(212)
plotResponse(hula,fc,physconst('LightSpeed'),'weights',w);
set(gca,'xtick',azang);
title('Array Response with Beamforming Weights');
```

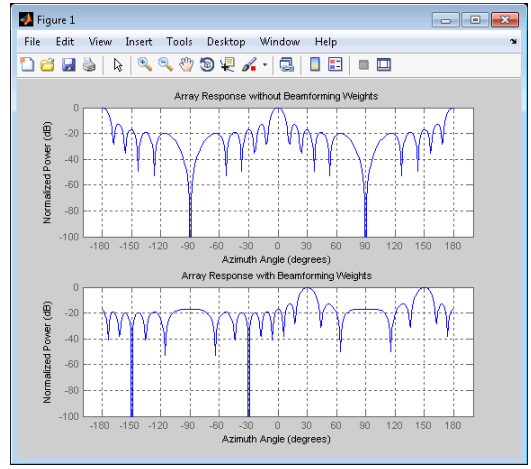


Figure 1
File Edit View Insert Tools Desktop Window Help
Array Response without Beamforming Weights
Normalized Power (dB)
0 -20 -40 -60 -80 -100
-180 -150 -120 -90 -60 -30 0 30 60 90 120 150 180
Azimuth Angle (degrees)
Array Response with Beamforming Weights
Normalized Power (dB)
0 -20 -40 -60 -80 -100
-180 -150 -120 -90 -60 -30 0 30 60 90 120 150 180
Azimuth Angle (degrees)

Adaptive Beamforming

The weights in the phase shift beamformer illustrated in Narrowband (phase shift) Beamformer with a ULA on page 3-4 are 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, but do not account for any interference scenarios. As a result, these *conventional* beamformers are susceptible to interference signals. This is especially true if the interference signals occur at sidelobes of the array response.

To account for interference signals, the Phased Array System Toolbox provides a number of adaptive, or statistically optimum beamformers.

The weights in an adaptive beamformer are chosen 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 at angles corresponding to the interference signals.

The following example demonstrates this for a linearly constrained minimum variance (LCMV) beamformer.

LCMV Beamformer

This example uses code from the Narrowband (phase shift) Beamformer with a ULA on page 3-4 example. Execute the code from that example before you run this example.

Use `phased.BarrageJammer` as the interference source. Specify the barrage jammer to have an effective radiated power of 10 watts. The interference signal from the barrage jammer is incident on the ULA at an angle of 120 degrees azimuth and 0 degrees elevation.

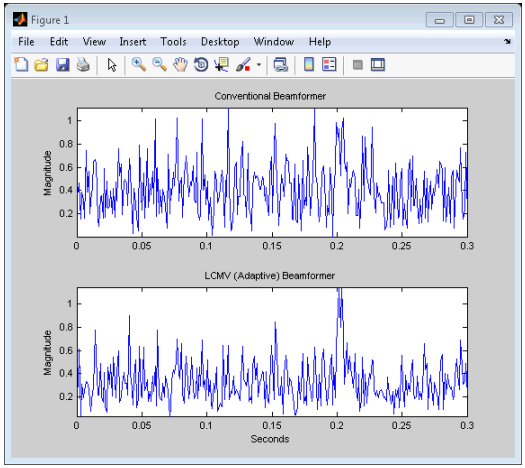
```
hjammer = phased.BarrageJammer('ERP',10,'SamplesPerFrame',300);
jamsig = step(hjammer);
jammer_angle = [120;0];
jamsig = collectPlaneWave(hula,jamsig,jammer_angle,fc);
```

Add some low-level complex white Gaussian noise to simulate noise contributions not directly associated with the jamming signal. Seed the random number generator for reproducible results.

```
noisePwr = 0.00001;   % noise power, 50dB SNR
rs = RandStream.create('mt19937ar','Seed',2008);
noise = sqrt(noisePwr/2)*(randn(rs,size(jamsig))+1j*randn(rs,size(jamsig)
jamsig = jamsig+noise;
rxsig = x+jamsig;
[yout,w] = step(hbf,rxsig);
```

Implement the LCMV beamformer. Use the target-free data,`jamsig`, as training data. Output the beamformer weights.

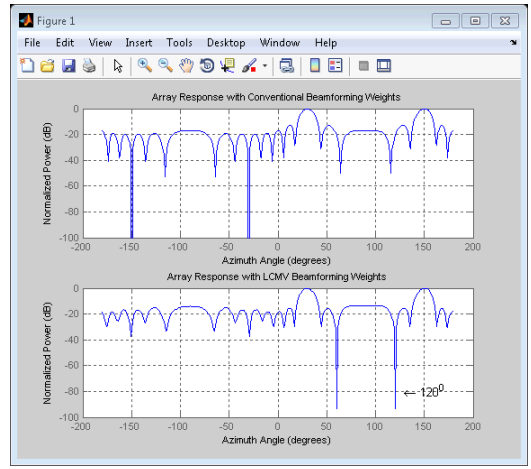
```
hstv = phased.SteeringVector('SensorArray',hula,...
       'PropagationSpeed',physconst('LightSpeed'));
hLCMV = phased.LCMVBeamformer('DesiredResponse',1,...
        'TrainingInputPort',true,'WeightsOutputPort',true);
hLCMV.Constraint = step(hstv,fc,angle_of_arrival);
hLCMV.DesiredResponse = 1;
[yLCMV,wLCMV]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



The adaptive beamformer significantly improves the SNR of the rectangular pulse at 0.2 seconds.

Plot the array normalized power response for the conventional and LCMV beamformers.

```
subplot(211)
plotResponse(hula,fc,physconst('LightSpeed'),'weights',w);
title('Array Response with Conventional Beamforming Weights');
subplot(212)
plotResponse(hula,fc,physconst('LightSpeed'),'weights',wLCMV);
title('Array Response with LCMV Beamforming Weights');
```



The LCMV beamforming weights place a null in the array response at the arrival angle of the interference signal.

The Phased Array System Toolbox provides additional narrowband adaptive beamformers. See `phased.FrostBeamformer` and `phased.MVDRBeamformer`. For a comprehensive list of supported algorithms, see “Beamformers”.

See Conventional and Adaptive Beamformers for a demo.

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.

The Phased Array System Toolbox provides a number of conventional and adaptive wideband beamformers. These include `phased.FrostBeamformer`, `phased.TimeDelayBeamformer`, and `phased.TimeDelayLCMVBeamformer`.

Wideband Conventional Time Delay Beamforming

Assume an acoustic (pressure wave) chirp signal. The chirp signal has a bandwidth of 1 kHz and propagates at a speed of 340 meters/second at sea level.

```
c = 340; % speed of sound at sea level
t = linspace(0,1,5e4)';
sig = chirp(t,0,1,1e3);
```

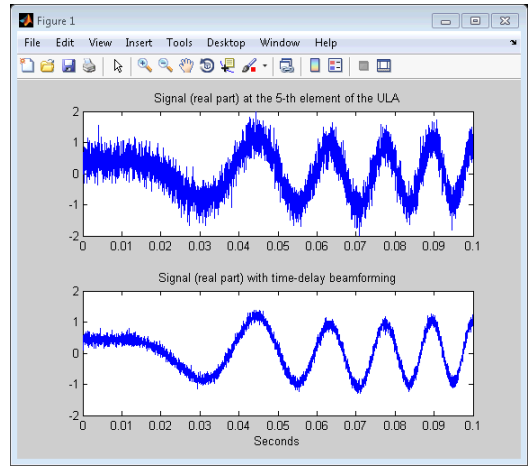
Collect the acoustic chirp with a ten-element ULA with omnidirectional microphone elements spaced less than 1/2 the wavelength of the 50 kHz sampling frequency. The chirp is incident on the ULA with an angle of 45 degrees azimuth and 0 degrees elevation.

```
hmic = phased.OmnidirectionalMicrophoneElement('FrequencyRange',[20 20e3]
hula = phased.ULA('Element',hmic,'NumElements',10,'ElementSpacing',0.01);
hcol = phased.WidebandCollector('Sensor',hula,'SampleRate',5e4,...
    'PropagationSpeed',c,'ModulatedInput',false);
sigang = [60; 0];
rsig = step(hcol,sig,sigang);
rsig = rsig+0.3*randn(size(rsig));
```

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

```
htbf = phased.TimeDelayBeamformer('SensorArray',hula,...
    'SampleRate',5e4,'PropagationSpeed',c,'Direction',sigang);
y = step(htbf,rsig);
subplot(2,1,1);
```

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



See Acoustic Beamforming Using a Microphone Array for a demo of using wideband beamforming to extract speech signals in noise.

4

Direction of Arrival (DOA) Estimation

Beamscan DOA Estimation

In the classic formulation of the direction of arrival (DOA) estimation problem, an array of N sensors receives a wavefield, which is the superposition of signals originating from $M < N$ narrowband sources. Denote the directions or angles of arrival of the M sources as $[\Theta_1, \Theta_2, ..., \Theta_M]$.

The Phased Array System Toolbox provides a variety of DOA estimation algorithms. See "Direction of Arrival (DOA)" for a comprehensive list of supported algorithms.

The following example illustrates one supported DOA algorithm, the nonparametric beamscan technique. The beamscan algorithm estimates the M DOAs by scanning the array beam over a region of interest. The output power is computed for each beam scan angle and the maxima are identified as the DOA estimates.

Beamscan DOA

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

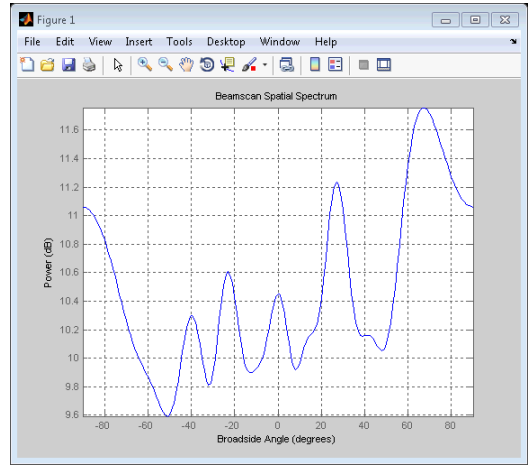
Assume that there is a wavefield incident on the ULA consisting of two linear FM pulses. The DOAs of the two sources are 30 degrees azimuth and 60 degrees azimuth. Both sources have elevation angles of zero degrees.

```
hwav = phased.LinearFMWaveform('SweepBandwidth',1e5,...
       'PulseWidth',5e-6,'OutputFormat','Pulses','NumPulses',1);
sig1 = step(hwav);
sig2 = sig1;
ang1 = [30; 0];
ang2 = [60;0];
arraysig = collectPlaneWave(hula,[sig1 sig2],[ang1 ang2],fc);
rng default
npower = 0.01;
```

```
noise = sqrt(npower/2)*randn(size(arraysig))+1j*randn(size(arraysig));
rxsig = arraysig+noise;
```

Implement a beamscan DOA estimator. Output the DOA estimates and plot the spatial spectrum.

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



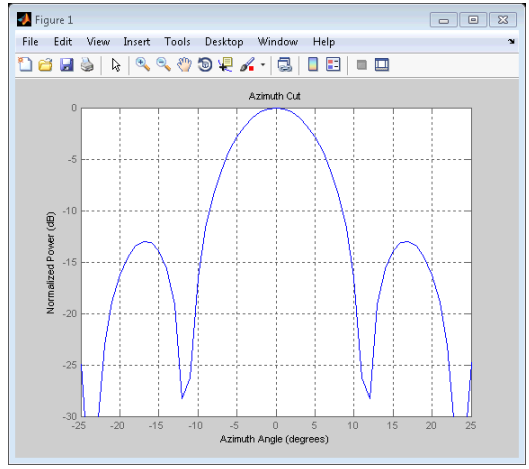
See Direction of Arrival Estimation with Beamscan and MVDR for a demo.

Superresolution DOA Estimation

The beamscan DOA estimator illustrated in Beamscan DOA on page 4-2 is not able to resolve two separate signal sources when the angles of arrival both fall within the main lobe of the array response.

To illustrate this, examine the array response of the ULA used in the preceding example. Plot the response and zoom in on the main lobe for visualization.

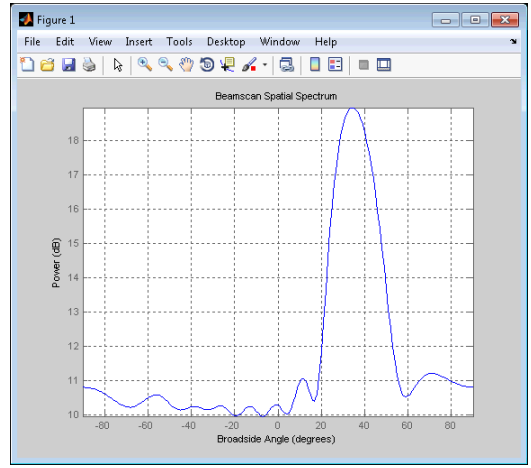
```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
hula = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
hula.Element.FrequencyRange = [8e8 1.2e9];
plotResponse(hula,fc,physconst('LightSpeed'));
axis([-25 25 -30 0]);
```



Assume that you have two signal sources with DOAs separated by ten degrees. Both DOAs fall inside the main lobe of the array response and therefore the beamscan DOA estimator can not resolve them as separate sources.

```
t = linspace(0,1,1000);
x1 = cos(2*pi*100*t)'; x2 = cos(2*pi*300*t)';
ang1 = [30; 0];
ang2 = [40;0];
```

```
arraysig = collectPlaneWave(hula,[x1 x2],[ang1 ang2],fc);
rng default
npower = 0.01;
noise = sqrt(npower/2)*randn(size(arraysig))+1j*randn(size(arraysig));
rxsig = arraysig+noise;

% Estimate DOAs using the beamscan estimator

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



The beamscan estimator does not resolve the two sources based on their DOAs.

The Phased Array System Toolbox provides superresolution DOA estimators that improve the resolution of the nonparametric beamscan estimator. The next example illustrates one such DOA estimator, using `phased.RootMUSICEstimator`.

Root MUSIC DOA Estimation

In this example, use the root MUSIC DOA estimator to resolve the two signal sources that fall within the main lobe of the array response. The entire code example is presented for convenience.

```
% Construct the array
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
hula = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
hula.Element.FrequencyRange = [8e8 1.2e9];
% Create the signals. Two sources with DOAs of 30 and 40
% degrees azimuth.
t = linspace(0,1,1000);
x1 = cos(2*pi*100*t)'; x2 = cos(2*pi*300*t)';
ang1 = [30; 0];
ang2 = [40;0];
arraysig = collectPlaneWave(hula,[x1 x2],[ang1 ang2],fc);
rng default
npower = 0.01;
noise = sqrt(npower/2)*randn(size(arraysig))+1j*randn(size(arraysig));
rxsig = arraysig+noise;

% Use a root MUSIC DOA estimator
hroot = phased.RootMUSICEstimator('SensorArray',hula,...
        'OperatingFrequency',fc,'NumSignalsSource','Property',...
        'NumSignals',2,'ForwardBackwardAveraging',true);
doa_est = step(hroot,rxsig)
% doa_est = [39.7984   30.1805 ]
```

See High Resolution Direction of Arrival Estimation for a demo.

5

Space-Time Adaptive Processing (STAP)

Angle-Doppler Response

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 a FFT-based Doppler filter to compute the angle-Doppler response.

Angle-Doppler Response of a Stationary Target at a Stationary Array

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]` and has a nonfluctuating radar cross section (RCS) of 1 square meter. Both the array and target are stationary.

The array operates at 4 GHz with elements spaced at 1/2 the operating wavelength. The array transmits a rectangular pulse 2 microseconds in duration with a pulse repetition frequency (PRF) of 5 kHz.

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

```
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('lightspeed')/4e9;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-006, 'PRF',5e3,...
    'SampleRate',1e6,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
htxplat = phased.Platform('InitialPosition',[0;0;0],'Velocity',[0;0;0]);
htgt = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[0;0;0]);
```

```
hspace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
hrx = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
    'SampleRate',1e6,'Gain',40);
htx = phased.Transmitter('PeakPower',1e4,'InUseOutputPort',true,'Gain',40
```

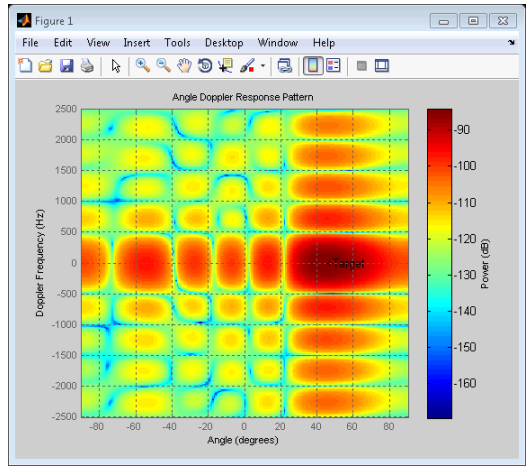
Propagate ten rectangular pulses to and from the target and collect the responses at the array.

```
PRF = 5e3;
NumPulses = 10;
wav = step(hwav);
tgtloc = htgtplat.InitialPosition;
txloc = htxplat.InitialPosition;
M = hwav.SampleRate*1/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);

for n = 1:NumPulses
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = step(htx,wav);  % transmit pulse
    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig);  % reflect pulse off stationary target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate pulse to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang);   % collect pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);    % receive pulse
end
```

Determine and plot the angle-Doppler response. Place the string `+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*hwav.SampleRate));
snapshot = shiftdim(rxsig(tgtcell,:,:));  % Remove singleton dim
hadresp = phased.AngleDopplerResponse('SensorArray',hula,...
              'OperatingFrequency',4e9, ...
```

```
                'PropagationSpeed',physconst('lightspeed'),...
                'PRF',PRF, 'ElevationAngle',tgtelang);
plotResponse(hadresp,snapshot);
text(tgtazang,tgtdoppler,'+Target');
```



As expected, the angle-Doppler response shows the greatest response at zero Doppler and 45 degrees azimuth.

If the transmitter platform is in motion, stationary targets exhibit nonzero Doppler shifts due to the motion of the platform. This is also true of stationary clutter. This complicates the detection of slow-moving targets because their Doppler shifts are obscured by the motion-induced Doppler shift and spread of the clutter returns.

Angle-Doppler Response of a Stationary Target Return at a Moving Array

This example illustrates the non-zero Doppler shift exhibited by a stationary target in the presence of array motion. The scenario is identical to Angle-Doppler Response of a Stationary Target at a Stationary Array on page 5-2 except that the ULA is moving at a constant velocity. For convenience, the MATLAB code to set up the objects is repeated. Note that `InitialPosition` and `Velocity` properties of the `htxplat` 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 "Displaced Phase Center Antenna (DPCA) Pulse Canceller" on page 5-8.

```
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('lightspeed')/4e9;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-006, 'PRF',5e3,...
    'SampleRate',1e6,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
vy = (hula.ElementSpacing*hwav.PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
htgt = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[0;0;0]);
hspace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
hrx = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
    'SampleRate',1e6,'Gain',40);
htx = phased.Transmitter('PeakPower',1e4,'InUseOutputPort',true,'Gain',40
```

Transmit ten rectangular pulses toward the target as the ULA is moving. Collect the received echoes.

```
PRF = 5e3;
NumPulses = 10;
wav = step(hwav);
tgtloc = htgtplat.InitialPosition;
M = hwav.SampleRate*1/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/hwav.SampleRate,1/PRF,'[)');
rangebins = (physconst('lightspeed')*fasttime)/2;

for n = 1:NumPulses
    [txloc,~] = step(htxplat,1/PRF); %move transmitter
```

```
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = step(htx,wav);  % transmit pulse
    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig);  % reflect pulse off stationary target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate pulse to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang);   % collect pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);    % receive pulse
end
```

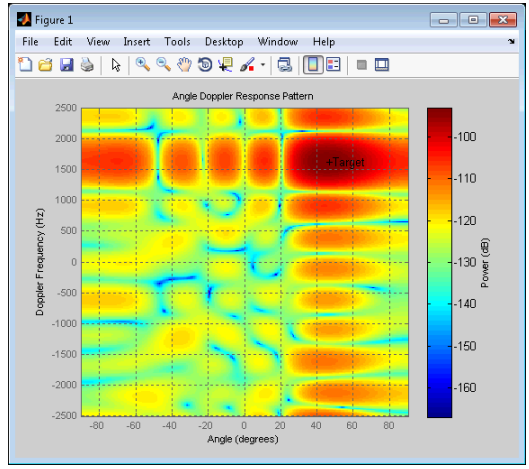
Calculate the target angles and range with respect to the ULA. Calculate the Doppler shift induced by the motion of the phased array.

```
sp = radialspeed(tgtloc, htgtplat.Velocity, ...
                 txloc, htxplat.Velocity);
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 degrees and is identical to the stationary ULA example.

Plot the angle-Doppler response.

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



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

Displaced Phase Center Antenna (DPCA) Pulse Canceller

One way to mitigate the effects of clutter returns in a *moving target indication* (MTI) radar is to implement a pulse canceller on the slow-time data. Assume a stationary radar platform. For a stationary radar, moving targets exhibit Doppler shifts while stationary sources of clutter do not. Let x[n] n=0,1,2,...N-1 denote the slow-time samples and y[n] = x[n]-x[n-1] define a simple two-pulse canceller. The two-pulse canceller is a high-pass Doppler filter. To see this, note that the impulse response is δ[n]-δ[n-1]. Viewed as an FIR filter, the filter coefficients are:

$$h[n] = \begin{cases} 1 & n = 0 \\ -1 & n = 1 \end{cases}$$

and the frequency response of the two-pulse canceller is:

$$H(e^{j\omega}) = 2je^{-j\omega/2}\sin(\omega/2)$$

The filter has a single zero on the unit circle at z=1+j0.

A popular pulse-canceller places an additional zero at z=1+j0 on the unit circle, resulting in the three-pulse canceller with filter taps `[1 -2 1]`. This three-pulse canceller is illustrated in the demo Ground Clutter Mitigation with Moving Target Indication (MTI) Radar demo.

If the radar platform is moving, you observe Doppler shifts and spread of stationary clutter returns. The Doppler shift of the clutter is proportional to the platform velocity and the angle of arrival. The Doppler shift and spread of the relatively large amplitude clutter returns can mask the detection of slow-moving targets. In this case, a simple Doppler high-pass filter does not mitigate the effect of clutter. The suppression of clutter returns received by a moving radar is the main application of space-time adaptive (STAP) processing.

The *displaced phase center antenna technique* (DPCA) pulse canceller is the oldest space-time processing technique for clutter rejection. Because the DPCA pulse canceller does not make explicit use of any received data, it is not technically a space-time *adaptive* technique.

You can implement a non-adaptive DPCA pulse canceller with `phased.DPCACanceller`. The DPCA pulse canceller implemented in the Phased Array System Toolbox 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. Mathematically, the DPCA condition is specified as

$$vT = d / 2$$

where v is the speed of the platform, T is the pulse repetition interval, and d is the inter-element spacing of the array. If the preceding condition is satisfied, the Doppler shift from stationary clutter at a given range gate encountered by the leading subarray (aperture) is identical to the Doppler shift encountered by the trailing subarray (aperture). Implementing a two-pulse canceller on the signals received by the leading and trailing subarrays acts as a high-pass Doppler filter. A moving target produces different Doppler shifts at the leading and trailing apertures and is not cancelled by the DPCA pulse canceller.

Example: DPCA Pulse Canceller

This examples 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 1/2 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 is moving along the array axis with a speed equal to 1/2 the product of the element spacing and the PRF. This satisfies the necessary condition to implement the DPCA pulse canceller. The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of `[15;15;0]`. The following MATLAB code constructs the required System objects to simulate the signal received by the ULA.

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

```
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
vy = (hula.ElementSpacing*hwav.PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
htgt = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hspace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
hrx = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
    'SampleRate',1e6,'Gain',40);
htx = phased.Transmitter('PeakPower',1e4,'InUseOutputPort',true,'Gain',40
```

Next load the clutter signal.

```
load('cluttersignaldemo.mat');
```

The clutter echoes are computed using the constant gamma model with a gamma value of -15dB. Literature shows that such a gamma value can be used to model terrain covered by woods. For each range, the clutter return can be thought of as a combination of the returns from many small clutter patches on that range ring. To simplify the computation, we also assumed that each patch had an azimuthal span of 30 degrees.

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

```
PRF = 5e3;
NumPulses = 10;
wav = step(hwav);
M = hwav.SampleRate*1/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/hwav.SampleRate,1/PRF,'[)');
rangebins = (physconst('lightspeed')*fasttime)/2;

for n = 1:NumPulses
    [txloc,~] = step(htxplat,1/PRF); %move transmitter
    [tgtloc,~] = step(htgtplat,1/PRF); %move target
```

```
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = step(htx,wav);  % transmit pulse
    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig);  % reflect pulse off stationary target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate pulse to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang);   % collect pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);    % receive pulse
end
```

Add the clutter interference to the received signal.

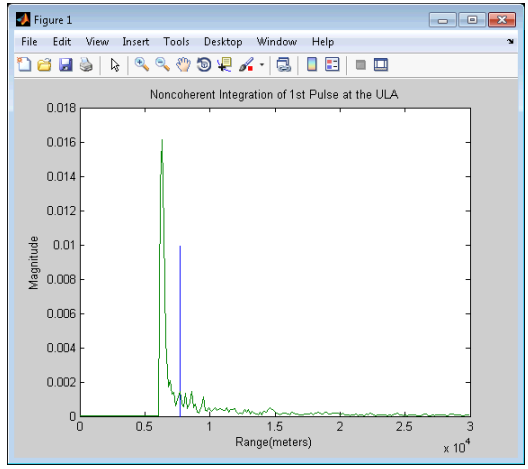
```
% csig is the clutter signal 200x6x10
ReceivedSig = rxsig+csig;
```

Determine the target's range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc, htgtplat.Velocity, ...
                txloc, htxplat.Velocity);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,physconst('lightspeed')/(2*hwav.SampleRate));
```

Use noncoherent pulse integration to visualize the signal received by the ULA for the first of the ten pulses. Mark the target's range gate with a vertical blue line.

```
firstpulse = pulsint(ReceivedSig(:,:,1),'noncoherent');
plot([tgtrng tgtrng],[0 0.01],rangebins,firstpulse);
title('Noncoherent Integration of 1st Pulse at the ULA');
xlabel('Range(meters)'); ylabel('Magnitude');
```

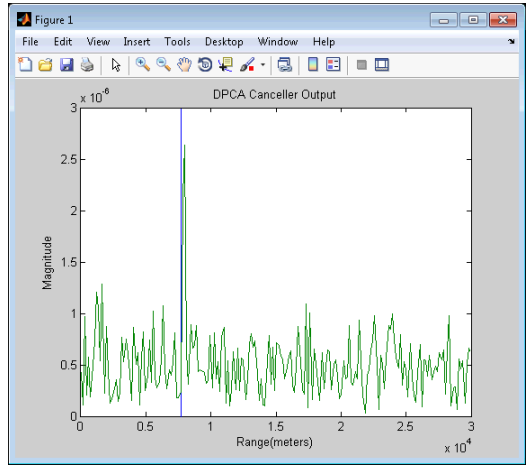


The large-magnitude clutter returns obscure the presence of the target. Apply the DPCA pulse canceller to reject the clutter.

```
hstap = phased.DPCACanceller('SensorArray',hula,'PRF',PRF,...
    'PropagationSpeed',physconst('lightspeed'), 'OperatingFrequency',4e9,
    'Direction',[0;0], 'Doppler',tgtdoppler, ...
    'WeightsOutputPort', true);
[y,w] = step(hstap,ReceivedSig,tgtcell);
```

Plot the result of applying the DPCA pulse canceller. Mark the target range gate with a vertical blue line.

```
plot([tgtrng,tgtrng],[0 3e-6],rangebins,abs(y));
title('DPCA Canceller Output');
xlabel('Range(meters)'), ylabel('Magnitude');
```



The DPCA pulse canceller has significantly rejected the clutter making the target visible at the expected range gate.

Adaptive Displaced Phase Center Antenna (ADPCA) Pulse Canceller

The displaced phase center antenna clutter-rejection technique introduced in "Displaced Phase Center Antenna (DPCA) Pulse Canceller" on page 5-8 imposes strict constraints on the platform motion. In practice, these constraints are difficult to satisfy both because of perturbations in the aircraft's flight path, such as *crabbing*, and errors introduced by the array components.

Further, the DPCA pulse canceller does make any use of the received data. This makes the DPCA pulse canceller susceptible to interference effects, such as jamming.

To overcome the difficulties with the DPCA pulse canceller, you can implement an adaptive DPCA (ADPCA) pulse canceller. The ADPCA pulse canceller uses the data received from two consecutive pulses to estimate the space-time interference covariance matrix.

Example: 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 Example: DPCA Pulse Canceller on page 5-9 except that a stationary broadband barrage jammer is added at `[3.5e3; 1e3; 0]`. The jammer has an effective radiated power of 1,000 watts. 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 1/2 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 is moving along the array axis with a speed equal to 1/2 the product of the element spacing and the PRF. This satisfies the necessary condition to implement the DPCA pulse canceller. The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of `[15;15;0]`. The following MATLAB code constructs the required System objects to simulate the scenario.

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

```
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-006, 'PRF',5e3,...
    'SampleRate',1e6,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
vy = (hula.ElementSpacing*hwav.PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
htgt = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hjammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200,...
    'SeedSource','Property','Seed',1000);
hjammerplat = phased.Platform('InitialPosition',[3.5e3; 1e3; 0],...
    'Velocity',[0;0;0]);
hspace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
hrx = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
    'SampleRate',1e6,'Gain',40);
htx = 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. Propagate the jamming signal from the jammer's location to the airborne ULA.

```
PRF = 5e3;
NumPulses = 10;
wav = step(hwav);
M = hwav.SampleRate*1/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/hwav.SampleRate,1/PRF,'[)');
rangebins = (physconst('lightspeed')*fasttime)/2;
jamloc = hjammerplat.InitialPosition;

for n = 1:NumPulses
    [txloc,~] = step(htxplat,1/PRF); %move transmitter
    [tgtloc,~] = step(htgtplat,1/PRF); %move target
```

```
        [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
        [txsig,txstatus] = step(htx,wav);  % transmit pulse
        txsig = step(hrad,txsig,tgtang); % radiate pulse
        txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
        txsig = step(htgt,txsig);  % reflect pulse off stationary target
        txsig = step(hspace,txsig,tgtloc,txloc); % propagate pulse to array
        rxsig(:,:,n) = step(hcol,txsig,tgtang);   % collect pulse
        rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);    % receive pulse

        jamsig = step(hjammer);
        % Get angle from jammer to transmitter
        [~,jamang] = rangeangle(jamloc,txloc);
        % Propagate signal from the jammer to transmitter
        jamsig = step(hspace,jamsig,jamloc,txloc);  % Propagate jammer
        jsig(:,:,n) = step(hcol,jamsig,jamang);
    end
```

Load the clutter signal. You can skip this step if you have the clutter signal, `csig`, in the MATLAB workspace from the DPCA example.

```
load('cluttersignaldemo.mat');
```

The clutter echoes are computed using the constant gamma model with a gamma value of -15dB. Literature shows that such a gamma value can be used to model terrain covered by woods. For each range, the clutter return can be thought of as a combination of the returns from many small clutter patches on that range ring. To simplify the computation, we also assumed that each patch had an azimuthal span of 30 degrees.

Add the clutter signal and jamming signal to the target echoes.

```
% csig is the clutter signal 200x6x10
% jsig is the interference signal
ReceivedSig = rxsig+csig+jsig;
```

First process the array responses using the non-adaptive DPCA pulse canceller.

Determine the target's range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc, htgtplat.Velocity, ...
```

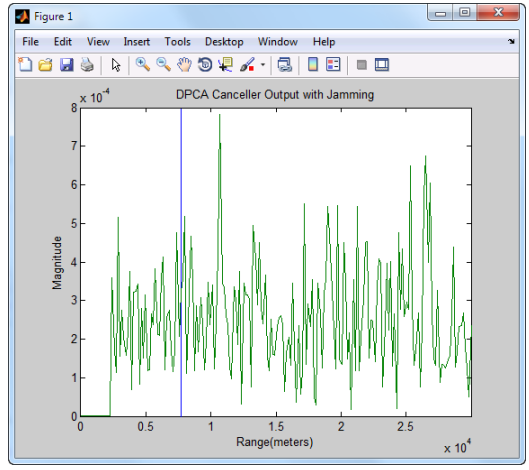
```
                txloc, htxplat.Velocity);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,physconst('lightspeed')/(2*hwav.SampleRate));
```

Construct the DPCA object and apply it to the received signals.

```
hstap = phased.DPCACanceller('SensorArray',hula,'PRF',PRF,...
    'PropagationSpeed',physconst('lightspeed'), 'OperatingFrequency',4e9,
    'Direction',[0;0], 'Doppler',tgtdoppler, ...
    'WeightsOutputPort', true);
[y,w] = step(hstap,ReceivedSig,tgtcell);
```

Plot the DPCA result with the target range marked by a vertical blue line. Note that the presence of the interference signal has obscured the target.

```
plot([tgtrng,tgtrng],[0 8e-4],rangebins,abs(y));
axis tight;
xlabel('Range(meters)'), ylabel('Magnitude');
title('DPCA Canceller Output with Jamming')
```

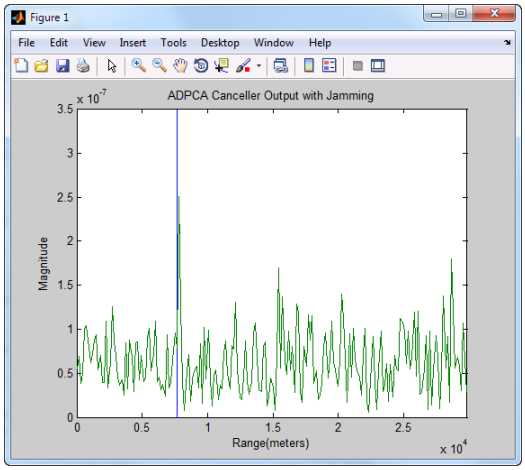


Apply the adaptive DPCA pulse canceller. Use 100 training cells and 4 guard cells, two on each side of the target range gate.

```
hstap = phased.ADPCACanceller('SensorArray',hula,'PRF',PRF,...
    'PropagationSpeed',physconst('lightspeed'), 'OperatingFrequency',4e9,
    'Direction',[0;0], 'Doppler',tgtdoppler, ...
    'WeightsOutputPort', true,'NumGuardCells',4,'NumTrainingCells',100);
[y_adpca,w_adpca] = step(hstap,ReceivedSig,tgtcell);
```

Plot the result with the target range marked by a blue vertical line. Note that the adaptive DPCA pulse cancellers enables you to detect the target in the presence of the jamming signal.

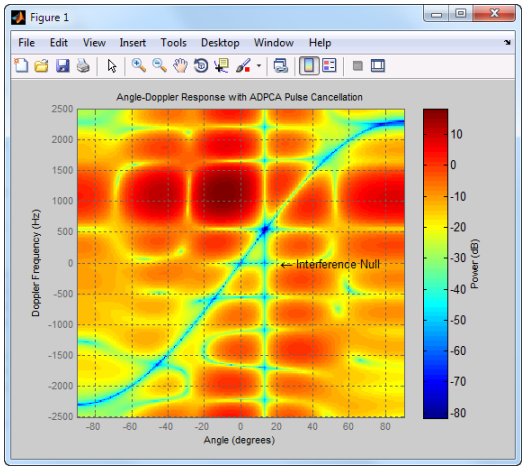
```
plot([tgtrng,tgtrng],[0 max(abs(y_adpca))+1e-7],rangebins,abs(y_adpca));
axis tight;
title('ADPCA Canceller Output with Jamming');
xlabel('Range(meters)'), ylabel('Magnitude');
```



Examine the angle-Doppler response. Note the presence of the clutter ridge in the angle-Doppler plane as well as the null at the jammer's broadside angle for all Doppler frequencies.

```
hadresp = phased.AngleDopplerResponse('SensorArray',hula,...
              'OperatingFrequency',4e9, ...
              'PropagationSpeed',physconst('lightspeed'),...
```

```
                'PRF',PRF, 'ElevationAngle',tgtelang);
plotResponse(hadresp,w_adpca);
title('Angle-Doppler Response with ADPCA Pulse Cancellation');
text(az2broadside(jamang(1),jamang(2))+10, 0,'\leftarrow Interference Nul
```



Sample Matrix Inversion (SMI) Beamformer

The optimum beamformer weights are

$$w = kR^{-1}v$$

where k is a scalar, R is the space-time covariance matrix, and v is the space-time steering vector.

Because the space-time covariance matrix is unknown, you must estimate it from the data. The sample matrix inversion (SMI) algorithm estimates the covariance matrix by designating a number of range gates to be *training cells*. Because the training cells are used to estimate the interference covariance, the training cells should not contain target returns. To prevent target returns from contaminating the estimate of the interference covariance, a specified number of guard cells are inserted before and after the designated target cell.

The general algorithm for estimating the space-time covariance matrix is:

- Assume you have a M-by-N-by-K matrix where M represents the number of slow-time samples, N the number of array sensors, and K the number of training cells (range gates for training). Assume that the number of training cells is an even integer and that you can designate K/2 training cells before and after the target range gate excluding the guard cells. Reshape the M-by-N-by-K matrix into a MN-by-K matrix. Let X denote the MN-by-K matrix.
- Estimate the space-time covariance matrix as

  $$\frac{1}{K}XX^H$$

- Invert the space-time covariance matrix estimate.
- Obtain the beamforming weights by multiplying the sample space-time covariance matrix inverse by the space-time steering vector.

Example: Sample matrix inversion (SMI) Beamformer

This scenario is identical to the one presented in Example: Adaptive DPCA Pulse Canceller on page 5-14. 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.

Assume you have an airborne radar platform modeled by a six-element ULA operating at 4 GHz. The array elements are spaced at 1/2 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 is moving along the array axis with a speed equal to 1/2 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 1,000 watts. The following MATLAB code constructs the required System objects to simulate the scenario.

```
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = physconst('lightspeed')/4e9;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-006, 'PRF',5e3,...
    'SampleRate',1e6,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',4e9);
vy = (hula.ElementSpacing*hwav.PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
htgt = phased.RadarTarget('MeanRCS',1,'Model','nonfluctuating');
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hjammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200,...
    'SeedSource','Property','Seed',1000);
hjammerplat = phased.Platform('InitialPosition',[3.5e3; 1e3; 0],...
    'Velocity',[0;0;0]);
hspace = phased.FreeSpace('OperatingFrequency',4e9,...
    'TwoWayPropagation',false,'SampleRate',1e6);
hrx = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
    'SampleRate',1e6,'Gain',40);
htx = phased.Transmitter('PeakPower',1e4,'InUseOutputPort',true,'Gain',40
```

Load the clutter signal. You can skip this step if you have the clutter signal, `csig`, in the MATLAB workspace from another example.

```
load('cluttersignaldemo.mat');
```

The clutter echoes are computed using the constant gamma model with a gamma value of -15dB. Literature shows that such a gamma value can be used to model terrain covered by woods. For each range, the clutter return can be thought of as a combination of the returns from many small clutter patches on that range ring. To simplify the computation, we also assumed that each patch had an azimuthal span of 30 degrees.

Propagate the ten rectangular pulses to and from the target and collect the responses at the array. Propagate the jamming signal from the jammer's location to the airborne ULA.

```
PRF = 5e3;
NumPulses = 10;
wav = step(hwav);
M = hwav.SampleRate*1/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/hwav.SampleRate,1/PRF,'[)');
rangebins = (physconst('lightspeed')*fasttime)/2;
jamloc = hjammerplat.InitialPosition;

for n = 1:NumPulses
    [txloc,~] = step(htxplat,1/PRF); %move transmitter
    [tgtloc,~] = step(htgtplat,1/PRF); %move target
    [~,tgtang] = rangeangle(tgtloc,txloc); % get angle to target
    [txsig,txstatus] = step(htx,wav);  % transmit pulse
    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig);  % reflect pulse off stationary target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate pulse to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang);   % collect pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);    % receive pulse

    jamsig = step(hjammer);
    % Get angle from jammer to transmitter
    [~,jamang] = rangeangle(jamloc,txloc);
```

```
    % Propagate signal from the jammer to transmitter
    jamsig = step(hspace,jamsig,jamloc,txloc);  % Propagate jammer
    jsig(:,:,n) = step(hcol,jamsig,jamang);
end
```

Add the clutter signal and jamming signal to the target echoes.

```
% csig is the clutter signal 200x6x10
% jsig is the interference signal
ReceivedSig = rxsig+csig+jsig;
```

Determine the target's range, range gate, and two-way Doppler shift.

```
sp = radialspeed(tgtloc, htgtplat.Velocity, ...
                txloc, htxplat.Velocity);
tgtdoppler = 2*speed2dop(sp,lambda);
tgtLocation = global2localcoord(tgtloc,'rs',txloc);
tgtazang = tgtLocation(1);
tgtelang = tgtLocation(2);
tgtrng = tgtLocation(3);
tgtcell = val2ind(tgtrng,physconst('lightspeed')/(2*hwav.SampleRate));
```

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.

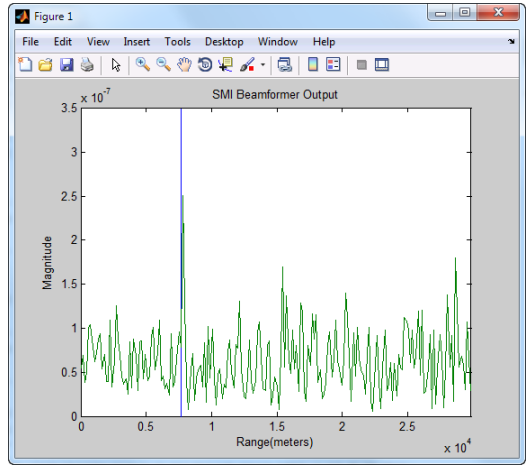
```
tgtang = [tgtazang; tgtelang];
hstap = phased.STAPSMIBeamformer('SensorArray', hula, 'PRF', PRF, ...
    'PropagationSpeed', physconst('lightspeed'), 'OperatingFrequency', 4e
    'Direction', tgtang, 'Doppler', tgtdoppler, ...
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100);
 [y,weights] = step(hstap,ReceivedSig,tgtcell);
```

Plot the resulting array output after beamforming. Plot the angle-Doppler response with the beamforming weights.

```
plot([tgtrng,tgtrng],[0 max(abs(y_adpca))+1e-7],rangebins,abs(y_adpca));
axis tight;
title('SMI Beamformer Output');
xlabel('Range(meters)'), ylabel('Magnitude');
```

```
% Construct an angle-Doppler response object and apply the
% beamforming weights
hresp = phased.AngleDopplerResponse('SensorArray',hula,...
            'OperatingFrequency',4e9,'PRF',PRF,...
            'PropagationSpeed',physconst('lightspeed'));
 plotResponse(hresp,weights);
 title('Angle-Doppler Response with SMI Beamforming Weights');
```

The beamformer output is shown in the following figure.



The angle-Doppler response with the beamforming weights is shown in the following figure.

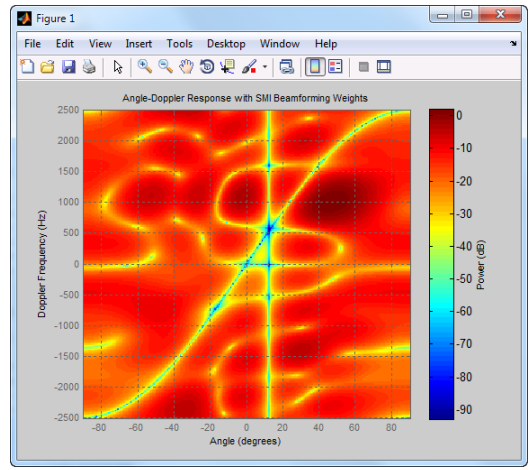


Figure 1
File Edit View Insert Tools Desktop Window Help
Angle-Doppler Response with SMI Beamforming Weights
Doppler Frequency (Hz)
2500
2000
1500
1000
500
0
-500
-1000
-1500
-2000
-2500
-80 -60 -40 -20 0 20 40 60 80
Angle (degrees)
Power (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
-90

6

Detection

Hypothesis Testing

In this section...

Neyman-Pearson Hypothesis Testing

A common scenario in phased array applications is the need to make a decision between two competing hypotheses about the state of nature. The state of nature is not directly observable and must be inferred from observations (data) received by the array. Because the data is affected by many factors in an unpredictable way, the decision is necessarily statistical. The goal is to formulate a decision rule that chooses between the two hypotheses based on some optimality criterion. There are three major optimality criterion used in formulating hypothesis tests: the Bayes risk criterion, the minimax criterion, and the Neyman-Pearson (NP) criterion. In phased array applications, such as radar and sonar, the Neyman-Pearson criterion is the most common.

In the Neyman-Pearson framework, the states of nature are characterized by probability distributions. The hypothesis test consists of deciding which probability distribution generated the observed data. In signal detection, the most common formulation is that under the null hypothesis the observed data consist of noise only. Under the alternative hypothesis, the data consist of some deterministic signal plus noise. For example:

$$H_0: \quad x[n] = \ [n]$$
$$H_1: \quad x[n] = s[n] + \ [n]$$

where ε[n] is a sequence of random variables drawn from some distribution and s[n] is a known deterministic signal.

There are two important conditional probabilities in this scenario: 1.) the probability that you decide the data consist of signal+noise when only noise is present. This is called a *false alarm*, or *Type I error*, and 2.) the probability you decide the data consist of noise only when there is actually a signal present. This is a called a *miss*, or *Type II error*. The complement of the

probability of a miss is the probability of detection, the probability you decide there is a signal present when, in fact, a signal is present.

The Neyman-Pearson criterion chooses a decision rule that maximizes the probability of detection subject to the constraint that the false-alarm probability is at most some specified number. Because the probability distributions under the null and alternative hypotheses do not have disjoint support, the smaller the maximum probability of false alarm, the smaller the probability of detection.

Likelihood Ratio Tests

Under the NP criterion, 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. Denote the probability that you observe a given data vector y under the alternative as $p(y \mid H_1)$. Similarly, denote the probability you observe a given data vector y under the null hypothesis as $p(y \mid H_0)$. The NP detector forms the ratio of these probabilities and decides between the two hypotheses based on a threshold λ. If the likelihood ratio exceeds λ choose H_1, if not choose H_0. The LRT is given by:

$$L(y) = \frac{p(y \mid H_1)}{p(y \mid H_0)} \underset{H_0}{\overset{H_1}{\gtrless}} \lambda$$

where λ is determined by the following:

1. Let $f(y \mid H_0)$ denote the probability density under the null hypothesis.
2. Set λ equal to that value such that the integral of $f(y \mid H_0)$ from $L(y)>\lambda$ to ∞ is equal to the desired false alarm probability.

You can write this as:

$$P_{FA} = \int_{\{y:\, L(y)>\lambda\}}^{\infty} f(y \mid H_0)\, dy$$

You can use `npwgnthresh` to determine the threshold for the detection of deterministic signals in white Gaussian noise based on the Neyman-Pearson criterion.

Assume that you collect N samples of a real-valued signal, which under the null hypothesis is a sequence of uncorrelated zero-mean Gaussian random variables with some variance, σ_2. Under the alternative hypothesis, the sequence is an unknown mean value plus white noise. You can summarize this hypothesis test as:

$$H_0: \quad x[n] = w[n] \quad n = 0,1,2,\ldots N-1$$
$$H_a: \quad x[n] = A + w[n] \quad n = 0,1,2,\ldots N-1$$

where the w[n] are distributed as N(0,σ_2) and A is a constant.

The preceding situation arises when processing samples, which may consist of a reflected rectangular pulse waveform plus noise, or noise only.

Threshold for Real-Valued Signal in White Gaussian Noise

To determine the required SNR in dB for the NP detector when the maximum tolerable the false-alarm probability is 10^{-3}, enter:

```
T = npwgnthresh(1e-3,1,'real')
```

The preceding is equivalent to:

```
10*log10(2*erfcinv(2*1e-3)^2)
```

If you know the variance, you can determine the actual threshold corresponding to the desired false-alarm probability. You can also determine the probability of detection.

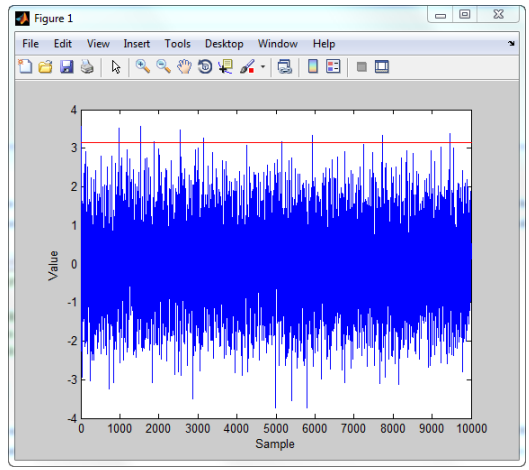
Assume the variance is 1. You can determine the threshold with the following code and verify empirically that the threshold results in the desired false-alarm probability under the null hypothesis.

```
threshold = sqrt(db2pow(T));
% The false-alarm probability is:
0.5*erfc(threshold/sqrt(2))
```

The following simulation empirically verifies that false-alarm probability performs as expected. Generate 10,000 samples of N(0,1) random variables. Set the maximum false-alarm probability to `1e-3`. Reset the random number

generator to produce repeatable results. Determine how many samples exceed the corresponding threshold.

```
rng default
variance = 1;
N = 1e4;
Pfa = 1e-3;
x = sqrt(variance)*randn(N,1);
T = npwgnthresh(Pfa,1,'real');
threshold = sqrt(variance*db2pow(T));
falsealarm = sum(x>threshold)/length(x)
plot(x)
line(1:length(x),threshold,'color',[1 0 0]);
xlabel('Sample'); ylabel('Value');
```



The red line is the plot is the threshold. You can see that very few sample values exceed the threshold as expected with the small false-alarm probability.

In this scenario, the probability of detection is:

```
Pd = 0.5*erfc(erfcinv(2*1e-6)-(1/sqrt(2)*sqrt(db2pow(T))))
```

In the next example assume that you employ pulse integration with real-valued pulses. Each sample is the sum of two samples, one from each of the pulses. Under the null hypothesis, each sample is a $N(0,\sigma^2)$ random

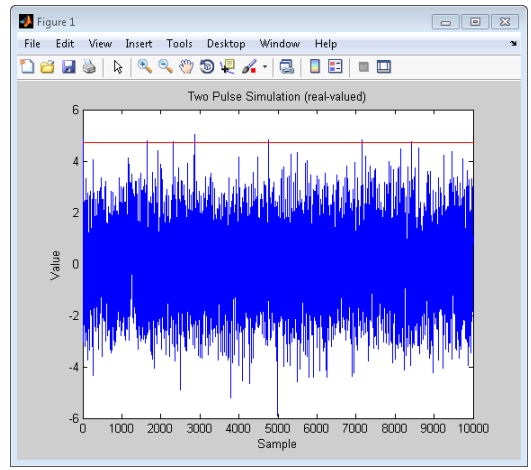
variable. The sum of these independent and identically-distributed Gaussian $N(0,\sigma^2)$ random variables is distributed as $N(0,2\sigma^2)$.

If you use the sum of the two real-valued white noise Gaussian samples as a sufficient statistic, the SNR threshold for a false-alarm probability of `1e-3` is:

```
T = npwgnthresh(1e-3,2,'real')
% equivalent to
% 10*log10(4*erfcinv(2*1e-3)^2)
```

Repeat the previous simulation for the two-pulse case.

```
rng default
variance = 1;
N = 1e4;
Pfa = 1e-3;
puls1 = sqrt(variance)*randn(N,1);
puls2 = sqrt(variance)*randn(N,1);
intpuls = sum([puls1 puls2],2);
T = npwgnthresh(Pfa,2,'real');
threshold = sqrt(variance*db2pow(T));
falsealarm = sum(intpuls>threshold)/length(intpuls);
figure;
plot(intpuls)
line(1:length(intpuls),threshold,'color',[1 0 0]);
xlabel('Sample'); ylabel('Value');
title('Two Pulse Simulation (real-valued)');
```



Threshold for Complex-Valued Signals in Complex White Gaussian Noise

Most phased array receivers utilize in-phase and quadrature channels. In this case, the data are complex-valued. Assume that you have a data vector, which consists of N samples of complex-valued white Gaussian noise under the null hypothesis, or a complex-valued constant plus complex white Gaussian noise under the alternative. You can summarize the hypothesis test as:

$$H_0: \quad x[n] = w[n] \quad n = 0,1,2,\ldots N-1$$
$$H_1: \quad x[n] = Ae^{j\phi} + w[n] \quad n = 0,1,2,\ldots N-1$$

The constant under the alternative hypothesis is complex-valued and as a result has both magnitude and phase information When phase information is available, the detector is referred to as *coherent*.

Assume you want the false-alarm probability to be at most `1e-3` in a coherent detection scheme with one sample.

```
T = npwgnthresh(1e-3,1,'coherent')
```

The sufficient statistic in the complex-valued case is the real part of received sample. You can test that this threshold empirically results in the correct false-alarm rate with the following code:

```
rng default
variance = 1;
N = 1e4;
Pfa = 1e-3;
x = sqrt(variance/2)*(randn(N,1)+1j*randn(N,1));
threshold = sqrt(variance*db2pow(T));
falsealarm = sum(real(x)>threshold)/length(x)
```

You can also implement a *noncoherent* detector, which uses only the magnitudes (moduli) of the complex-valued baseband samples.

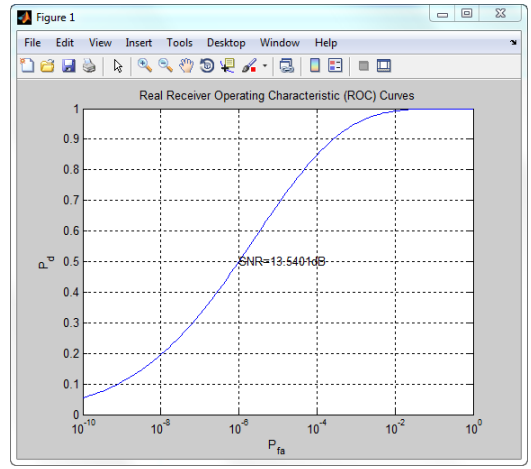
Receiver Operating Characteristic (ROC) Curves

ROC curves present graphical summaries of a detector's performance. The Phased Array System Toolbox has two utility function, which enable you to generate ROC curves: `rocpfa` and `rocsnr`.

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 demonstrate how 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')
```

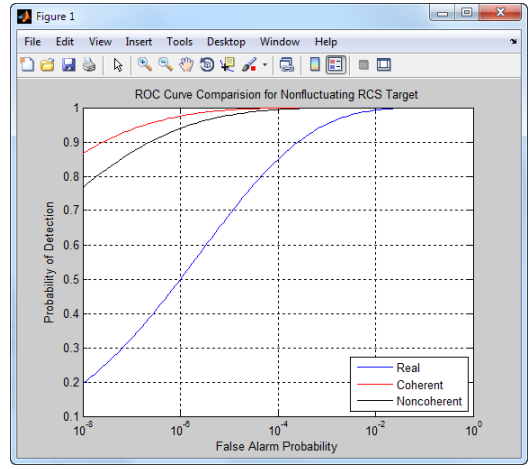


The ROC curve enables you to easily read off the probability of detection for a given false alarm rate.

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);
```

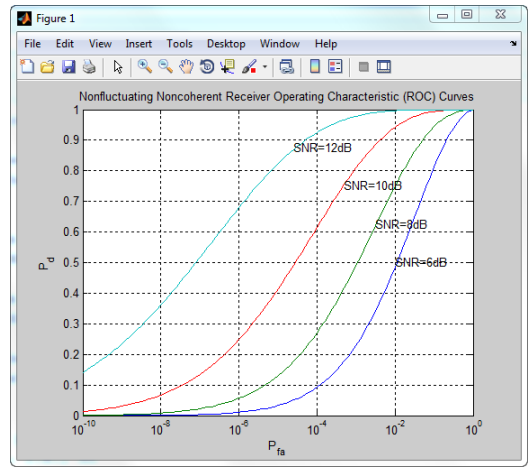
```
[Pd_coh,Pfa_coh] = rocsnr(SNR,'SignalType','NonfluctuatingCoherent',...
    'MinPfa',1e-8);
[Pd_noncoh,Pfa_noncoh] = rocsnr(SNR,'SignalType',...
    'NonfluctuatingNoncoherent','MinPfa',1e-8);
figure;
semilogx(Pfa_real,Pd_real); hold on; grid on;
semilogx(Pfa_coh,Pd_coh,'r');
semilogx(Pfa_noncoh,Pd_noncoh,'k');
xlabel('False Alarm Probability'); ylabel('Probability of Detection');
legend('Real','Coherent','Noncoherent','location','southeast');
title('ROC Curve Comparision for Nonfluctuating RCS Target');
```



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

`rocsnr` accepts an SNR vector input enabling you to quickly examine a number of ROC curves.

```
SNRs = (6:2:12);
figure;
rocsnr(SNRs,'SignalType','NonfluctuatingNoncoherent');
```

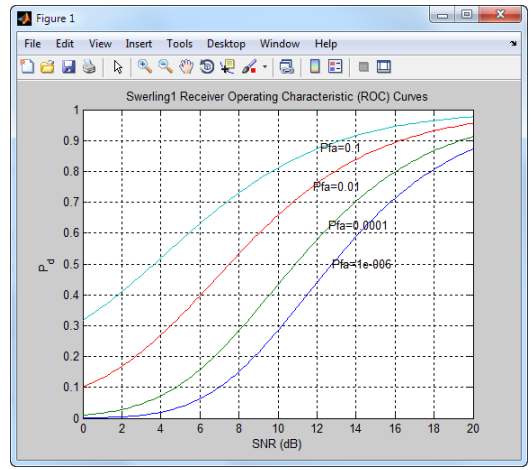


The graph demonstrates that as the SNR increases, the supports of the probability distributions under the null and alternative hypotheses become more disjoint. Therefore, for a given false-alarm probability, the probability of detection increases.

You can examine the probability of detection as a function of SNR for a fixed false-alarm probability with `rocpfa`.

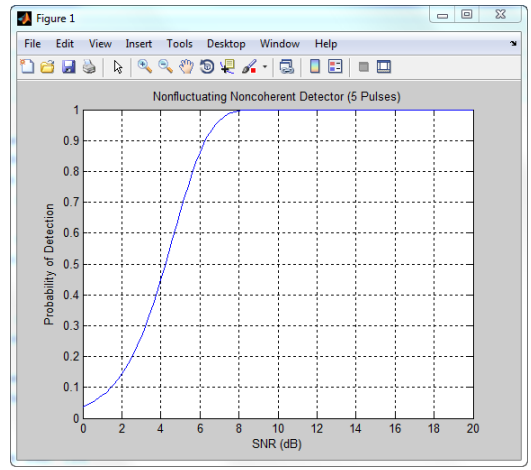
To obtain ROC curves for a Swerling I target model at false-alarm probabilities of `[1e-6 1e-4 1e-2 1e-1]`, enter:

```
Pfa = [1e-6 1e-4 1e-2 1e-1];
figure;
rocpfa(Pfa,'SignalType','Swerling1');
```



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

```
[Pd,SNR] = rocpfa(1e-4,'SignalType','NonfluctuatingNoncoherent',...
    'NumPulses',5);
figure;
plot(SNR,Pd); xlabel('SNR (dB)');
ylabel('Probability of Detection'); grid on;
title('Nonfluctuating Noncoherent Detector (5 Pulses)');
```



See Detector Performance Analysis using ROC Curves for a demo.

Matched Filtering

You can see from the results in "Receiver Operating Characteristic (ROC) Curves" on page 6-9 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. Use `phased.MatchedFilter` to implement a matched filter.

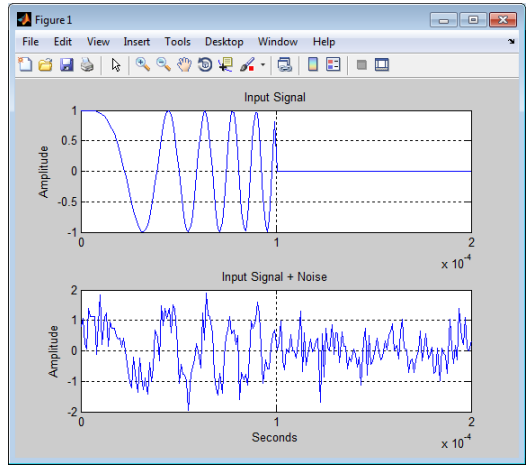
Characteristics of the matched filter that you can customize when you use `phased.MatchedFilter` include the matched filter coefficients and window for spectrum weighting. If you apply spectrum weighting, you can specify the coverage region and coefficient sample rate; Taylor, Chebyshev, and Kaiser windows have additional properties you can specify.

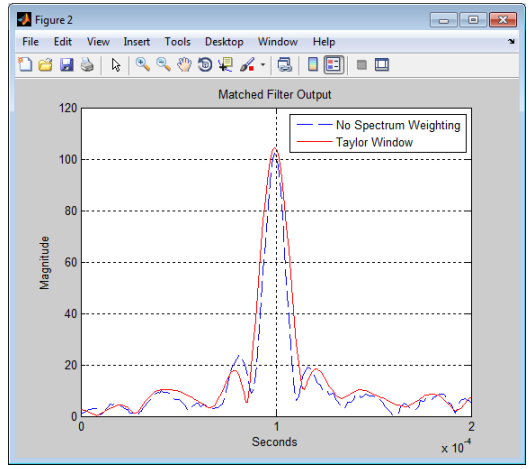
Matched Filtering of Linear FM Waveform

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 the matched filter. Spectrum weighting is often used with linear FM waveform to reduce the sidelobes in the time domain. This example compares the results using a matched filter with and without spectrum weighting.

```
% Specify the waveform.
hwav = phased.LinearFMWaveform('PulseWidth',1e-4,'PRF',5e3,...
    'SampleRate',1e6,'OutputFormat','Pulses','NumPulses',1,...
    'SweepBandwidth',1e5);
w = getMatchedFilter(hwav);

% Create a matched filter with no spectrum weighting, and a
% matched filter that uses a Taylor window for spectrum weighting.
hmf = phased.MatchedFilter('Coefficients',w);
hmf_taylor = phased.MatchedFilter('Coefficients',w,...
    'SpectrumWindow','Taylor');

% Create the signal and add noise.
sig = step(hwav);
rng(17)
```

```
x = sig+0.5*(randn(length(sig),1)+1j*randn(length(sig),1));

% Filter the noisy signal separately with each of the two filters.
y = step(hmf,x);
y_taylor = step(hmf_taylor,x);

% Plot the real parts of the waveform and noisy signal.
t = linspace(0,numel(sig)/hwav.SampleRate,hwav.SampleRate/hwav.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('Seconds'); ylabel('Amplitude');

% Plot the magnitudes of the two matched filter outputs.
figure;
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

The following examples demonstrates the SNR improvement obtained after matched filtering.

Place an isotropic antenna element at the global origin`[0;0;0]`. Place a target with a nonfluctuating RCS of one 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.

```
hsensor = phased.IsotropicAntennaElement('FrequencyRange',[5e9 15e9]);
htx = phased.Transmitter('Gain',20,'InUseOutputPort',true);
htgt = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1,...
    'OperatingFrequency',10e9);
htgtloc = phased.Platform('InitialPosition',[5000;5000;10]);
htxloc = phased.Platform('InitialPosition',[0;0;0]);
[tgtrng,tgtang] = rangeangle(htgtloc.InitialPosition,...
    htxloc.InitialPosition);
```

Create a rectangular pulse waveform 25 microseconds 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 `radareqpow` to determine the peak power required to detect a target with 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.

```
hwav = phased.RectangularWaveform('PulseWidth',25e-6,...
    'OutputFormat','Pulses','PRF',1e4,'NumPulses',1);
maxrange = physconst('LightSpeed')/(2*hwav.PRF);
SNR = npwgnthresh(1e-6,1,'noncoherent');
Pt = radareqpow(physconst('LightSpeed')/10e9,maxrange,SNR,...
    hwav.PulseWidth,'rcs',htgt.MeanRCS,'gain',htx.Gain);
```

Set the peak transmit power.

```
htx.PeakPower = Pt;
```

Create radiator and collector objects to operate at 10 GHz. Create a free space path for the propagation of the pulse to and from the target. Create an ideal receiver and a matched filter for the rectangular waveform.

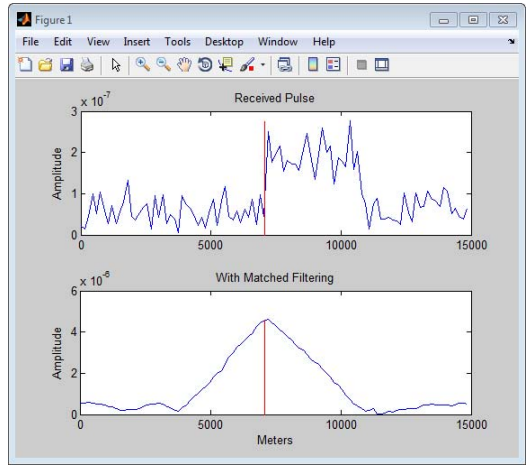
```
hrad = phased.Radiator('PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',10e9,'Sensor',hsensor);
hspace = phased.FreeSpace('PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',10e9,'TwoWayPropagation',false);
hcol = phased.Collector('PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',10e9,'Sensor',hsensor);
```

```
hrec = phased.ReceiverPreamp('NoiseFigure',0,'EnableInputPort',true,...
     'SeedSource','Property','Seed',2e3);
hmf = phased.MatchedFilter('Coefficients',getMatchedFilter(hwav),...
     'GainOutputPort',true);
```

Once you have created all the objects that define your model, you are ready to propagate the pulse to and from the target. After you collect the echo at the receiver, implement the matched filter to improve the SNR.

```
% Generate waveform
wf = step(hwav);
% Transmit waveform
[wf,txstatus] = step(htx,wf);
% Radiate pulse toward the target
wf = step(hrad,wf,tgtang);
% Propagate pulse toward the target
wf = step(hspace,wf,htxloc.InitialPosition,htgtloc.InitialPosition);
% Reflect it off the target
wf = step(htgt,wf);
% Propagate the pulse back to transmitter
wf = step(hspace,wf,htgtloc.InitialPosition,htxloc.InitialPosition);
% Collect the echo
wf = step(hcol,wf,tgtang);

% Receive target echo
rx_puls = step(hrec, wf,~txstatus);
[mf_puls,mfgain] = step(hmf,rx_puls);
% Get group delay of matched filter
Gd = length(hmf.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 = physconst('lightspeed').*t;
rangegates = rangegates/2;
plot(rangegates,abs(rx_puls)); title('Received Pulse');
ylabel('Amplitude'); hold on;
plot([tgtrng, tgtrng], [0 max(abs(rx_puls))],'r');
subplot(2,1,2)
```

```
plot(rangegates,abs(mf_puls)); title('With Matched Filtering');
xlabel('Meters'); ylabel('Amplitude'); hold on;
plot([tgtrng, tgtrng], [0 max(abs(mf_puls))],'r');
```



Constant False-Alarm Rate (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, calculating the false-alarm probability requires that you have an estimate of the noise variance. If the noise variance changes, you have to adjust the threshold in order to maintain a constant false-alarm rate. This requires an adaptive procedure. *constant false-alarm rate detectors* implement adaptive procedures that enable you to update the threshold level of your test in response to changes in the power of the interference.

To motivate the need for an adaptive procedure, assume a simple binary hypothesis test where you must decide between

$$H_0: \quad x[0] = w[0] \quad w[0] \sim N(0,1)$$
$$H_1: \quad x[0] = 4 + w[0]$$

for a single sample. Set the false-alarm rate to 0.001 and determine the threshold.

```
T = npwgnthresh(1e-3,1,'real');
threshold = sqrt(db2pow(T))
```

The threshold is 3.0902. You can check that this yields the desired false-alarm rate probability and compute the probability of detection.

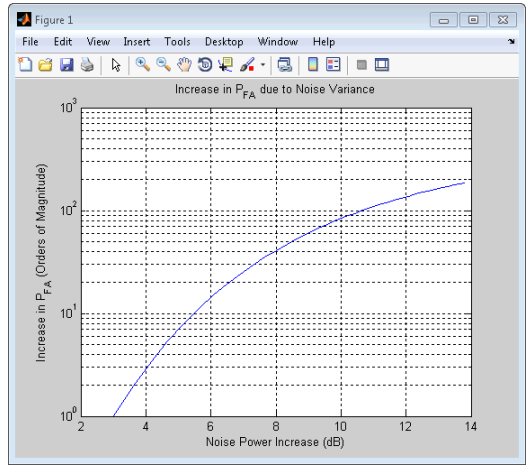
```
% check false-alarm probability
Pfa = 0.5*erfc(threshold/sqrt(2))
% compute probability of detection
Pd = 0.5*erfc((threshold-4)/sqrt(2))
```

Next assume that the noise power increases by 6.02 dB. In other words, the noise variance doubles. 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)
```

The following figure demonstrates the effect of increasing the noise variance on the false-alarm probability for a fixed threshold.

```
noisevar = 1:0.1:10;
Noisepower = 10*log10(noisevar);
Pfa = 0.5*erfc(threshold./sqrt(2*noisevar));
semilogy(Noisepower,Pfa./1e-3);
grid on; title('Increase in P_{FA} due to Noise Variance');
ylabel('Increase in P_{FA} (Orders of Magnitude)');
xlabel('Noise Power Increase (dB)');
```

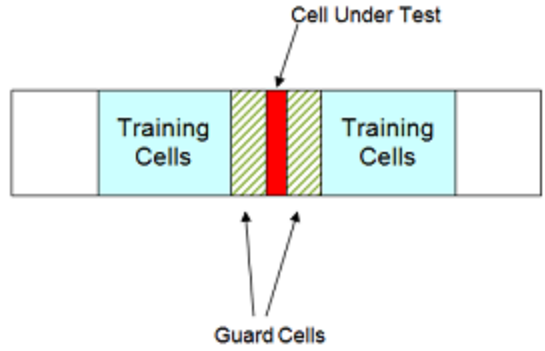


Cell-Averaging CFAR Detector

The cell-averaging CFAR detector estimates the noise variance for the range cell of interest, or *cell under test*, by analyzing data from neighboring range cells designated as *training cells*. The characteristics of the noise in the training cells are assumed to be identical to those 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. In other words, the data in the training cells are assumed to consist of noise only. In order 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.

To ensure that the noise characteristics of the training cells are indicative of the noise found in the CUT, it is important that the training cells do not represent range cells too distant in range from the CUT.

This situation is illustrated in the following figure.



The optimum estimator for the noise variance depends on distributional assumptions and the type of detector. Assume that you use a square-law detector. Further, assume that you have a Gaussian complex-valued random variable (RV) with independent real and imaginary parts. Assume the real and imaginary parts each have mean zero and variance equal to $\sigma^2/2$. 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`. The modifiable properties of `phased.CFARDetector` are:

- `NumGuardCells` — The number of guard cells. This is the total number of guard cells on either side of the cell under test.
- `NumTrainingCells` — The number of training cells on either of the cell under test.

- `ThresholdFactor` — Methods of obtaining the threshold factor
- `ProbabilityofFalseAlarm` — The desired false-alarm probability. This property applies only when you set the `ThresholdFactor` to `'Auto'`.
- `CustomThresholdFactor` — The custom threshold factor. This property applies when you set the `ThresholdFactor` property to `'Custom'`.
- `ThresholdOutputPort` — Output the detection threshold

Create a CFAR detector object with two guard cells, 20 training cells, and a false-alarm probability of 0.001.

```
hdetector = phased.CFARDetector('NumGuardCells',2,'NumTrainingCells',20,
    'ProbabilityFalseAlarm',1e-3);
```

By default the value of the `ThresholdFactor` property is `'Auto'`. This assumes a square-law detector with no pulse integration.

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;
```

Create an example to test the ability of the CFAR detector to adapt to the statistics of the input data. In this example, input noise-only trials to the CFAR detector. The trials are complex-valued white Gaussian noise with unknown variance. Use the default square-law detector and determine how close the empirical false alarm rate is to the desired false-alarm probability in 10,000 Monte Carlo trials.

Seed the random number generator for reproducible results.

```
seedval = RandStream('mt19937ar','Seed',1000);
```

Set the noise variance to 0.25. This corresponds to an approximate -6 dB SNR. Generate a 23-by-10000 matrix of complex-valued white Gaussian RVs 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(seedval,Ncells,Ntrials)+...
    1j*randn(seedval,Ncells,Ntrials));
```

Because you are implementing a square-law detector, take the squared magnitudes of the elements in the data matrix.

```
Z = abs(inputdata).^2;
```

Provide the output of the square-law operator and the index of the cell under test to CFAR detector's `step` method.

```
Z_detect = step(hdetector,Z,CUTidx);
```

The output is a logical vector `Z_detect` 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
```

The empirical false-alarm rate is 0.0013, very close to the desired false-alarm rate of 0.001.

7

Environment and Target Models

Free Space Path Loss

Propagation environments have significant effects on the amplitude, phase, and shape of propagating space-time wavefields. The `phased.FreeSpace` object enables you to model the range-dependent time delay, phase shift, and gain effects for a narrowband signal propagating in free space.

The free space path loss is:

$$L = \frac{(4\pi R)^2}{\lambda^2}$$

where R is the one-way distance between the source and the array in meters and λ is the signal wavelength.

You can use `fspl` to determine the free space path loss in dB for a given distance and wavelength.

Determine Free Space Path Loss in dB

Assume a transmitter is located at `[1000; 250; 10]` in the global coordinate system. Assume a target located at `[3000; 750; 20]`. The transmitter operates at 1 GHz. Determine the free space path loss in dB for a narrowband signal propagating to and from the target.

```
[tgtrng,~] = rangeangle([3000; 750; 20],[1000; 250; 10]);
% Multiply range by two for two-way propagation
tgtrng = 2*tgtrng;
% Determine the wavelength for 1 GHz
lambda = physconst('LightSpeed')/1e9;
L = fspl(tgtrng,lambda)
```

The free space path loss in dB is approximately 105 dB. This is equivalent to

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

which is a direct implementation of the equation for free space path loss.

In addition to modeling the path loss in dB, the `phased.FreeSpace` object accounts for the range-dependent delay in the signal. The time-dependent

delay is the ratio of the distance to the propagation speed of the waveform. The `phased.FreeSpace` object has four modifiable properties:

- `PropagationSpeed` — The propagation speed of the wave. The default is the speed of light.
- `OperatingFrequency` — The system operating frequency.
- `TwoWayPropagation` — Perform two-way propagation. This is a logical property. Set this property value equal to `true` to model two-way propagation. The default value of this property is `false`.
- `SampleRate` — The sampling rate in Hz.

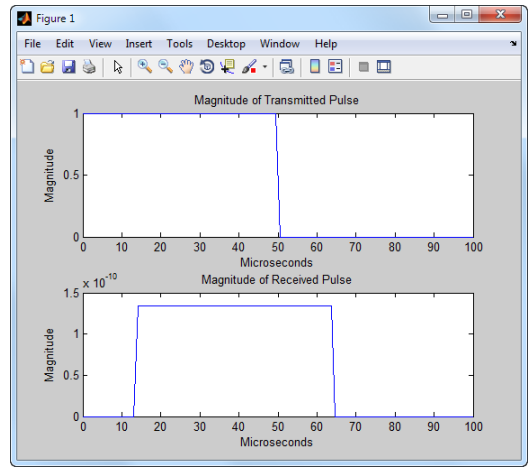
Propagate a Linear FM Pulse Waveform to and from a Target

Construct a linear FM pulse waveform 50 microseconds in duration with a bandwidth of 100 kHz. Model the range-dependent time delay and amplitude loss incurred by the two-way propagation of the pulse from the transmitter located at `[1000; 250; 10]` to a target located at `[3000; 750; 20]`.

```
hwav = phased.LinearFMWaveform('SweepBandwidth',1e5,...
         'PulseWidth',5e-5,'OutputFormat','Pulses',...
         'NumPulses',1,'SampleRate',1e6,'PRF',1e4);
wf = step(hwav);
hpath = phased.FreeSpace('SampleRate',1e6,'TwoWayPropagation',true,...
        'OperatingFrequency',1e9);
y = step(hpath,wf,[1000; 250; 10],[3000; 750; 20]);
```

Plot the magnitude of the transmitted and received pulse to show the amplitude loss and time delay. Scale the time axis in microseconds.

```
t = unigrid(0,1/hwav.SampleRate,1/hwav.PRF,'[)');
subplot(2,1,1)
plot(t.*1e6,abs(wf)); title('Magnitude of Transmitted Pulse');
xlabel('Microseconds'); ylabel('Magnitude');
subplot(2,1,2);
plot(t.*1e6,abs(y)); title('Magnitude of Received Pulse');
xlabel('Microseconds'); ylabel('Magnitude');
```



The delay in the received pulse is approximately 14 microseconds, which is exactly what you expect for a distance of 4.123 km at the speed of light.

Modeling One-way and Two-way Propagation

The `TwoWayPropagation` property of the `phased.FreeSpace` object enables you to use the `step` method for one or two-way propagation. The following example demonstrates this for a single linear FM pulse propagated to and from a target. The sensor is a single isotropic radiating antenna operating at 1 GHz located at `[1000; 250; 10]`. The target is located at `[3000; 750; 20]` and has a nonfluctuating RCS of 1 square meter .

The following code constructs the required objects and calculates the range and angle from the antenna to the target.

```
hwav = phased.LinearFMWaveform('SweepBandwidth',1e5,...
                'PulseWidth',5e-5,'OutputFormat','Pulses',...
            'NumPulses',1,'SampleRate',1e6);
hant = phased.IsotropicAntennaElement('FrequencyRange',[500e6 1.5e9]);
htx = phased.Transmitter('PeakPower',1e3,'Gain',20);
hrad = phased.Radiator('Sensor',hant,'OperatingFrequency',1e9);
hpath = phased.FreeSpace('SampleRate',1e6,'TwoWayPropagation',true,...
   'OperatingFrequency',1e9);
htgt = phased.RadarTarget('MeanRCS',1,'Model','Nonfluctuating');
```

```
hcol = phased.Collector('Sensor',hant,'OperatingFrequency',1e9);
sensorpos = [3000; 750; 20];
tgtpos = [1000; 250; 10];
[tgtrng,tgtang] = rangeangle(sensorpos,tgtpos);
```

Because the `TwoWayPropagation` property is set to `true`, you call the `step` method for the `phased.FreeSpace` object only once. In the following code, the `step` is called after the pulse is radiated from the antenna and before it is reflected from the target.

```
pulse = step(hwav);                                 % Generate pulse
pulse = step(htx,pulse);                            % Transmit pulse
pulse = step(hrad,pulse,tgtang);                       % Radiate pulse
% Propagate pulse to and from target
pulse = step(hpath,pulse,sensorpos,tgtpos);
pulse = step(htgt,pulse);                           % Reflect pulse
sig = step(hcol,pulse,tgtang);                         % Collect pulse
```

If you prefer to break up the two-way propagation into two separate calls to the `step` method, you can do that by setting the `TwoWayPropagation` property to `false`.

```
hpath = phased.FreeSpace('SampleRate',1e9,'TwoWayPropagation',false,...
    'OperatingFrequency',1e6);

pulse = step(hwav);                                 % Generate pulse
pulse = step(htx,pulse);                            % Transmit pulse
pulse = step(hrad,pulse,tgtang);                       % Radiate pulse
% Propagate pulse from the antenna to the target
pulse = step(hpath,pulse,sensorpos,tgtpos);
pulse = step(htgt,pulse);                           % Reflect pulse
% Propagate pulse from the target to the antenna
pulse = step(hpath,pulse,tgtpos,sensorpos);
sig = step(hcol,pulse,tgtang);                         % Collect pulse
```

Radar Target

The `phased.RadarTarget` object models a reflected signal from a target with nonfluctuating or fluctuating radar cross section (RCS). The modifiable properties of `phased.RadarTarget` are:

- `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

Create a radar target with a nonfluctuating RCS of 1 square meter, an operating frequency of 300 MHz, and a wave propagation speed equal to the speed of light.

```
hr = phased.RadarTarget('Model','nonfluctuating','MeanRCS',1,...
    'PropagationSpeed',physconst('lightspeed'),'OperatingFrequency',3e8);
```

The waveform incident on the target is scaled by the factor:

$$G = \sqrt{\frac{4\pi\sigma}{\lambda^2}}$$

where σ is the target mean RCS and λ is the wavelength of the operating frequency. Each element of the signal incident on the target is scaled by the preceding factor.

Create a target with a nonfluctuating RCS of 1 square meter. Set the operating frequency to 1 GHz. Set the signal incident on the target to be a vector of ones to demonstrate the gain factor.

```
hr = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',1e9);
x = ones(10,1);
y = step(hr,x);
```

The output vector `y` is vector equal to `11.8245*ones(10,1)`. The amplitude scaling factor is equal to:

```
lambda = hr.PropagationSpeed/hr.OperatingFrequency;
G = sqrt(4*pi*1/lambda^2)
```

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, which occur rapidly on pulse-to-pulse, or more slowly, on scan-to-scan time scales. Consider a target consisting of 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). Consider a target consisting of a dominant reflector along with several small reflectors. Subject to the how rapidly the aspect changes, or whether the radar uses frequency agility, these reflectors exhibit small magnitude fluctuations on pulse-to-pulse or scan-to-scan time scales. To account for significant fluctuations in the RCS, it is necessary to use statistical models. The four *Swerling* models are widely used to cover the fluctuating-RCS cases described here. The Swerling models are summarized in the following table:

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.

Swerling Case Number	Description
III	Scan-to-scan decorrelation. Chi-square PDF with 4 degrees of freedom. A number of scatterers with one dominant.
IV	Pulse-to-pulse decorrelation. Chi-square PDF with 4 degrees of freedom. A number of scatterers with one 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 off a Nonfluctuating Target

The following 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 approximately 1414 meters from the antenna at an angle of 45 degrees azimuth and 0 degrees elevation.

```
% Create objects and assign property values
% Isotropic antenna element
hant = phased.IsotropicAntennaElement('BackBaffled',true);
% Location of the antenna
harraypos = phased.Platform('InitialPosition',[0;0;0]);
% Location of the radar target
hrfpos = phased.Platform('InitialPosition',[1000; 1000; 0]);
% Linear FM waveform
hwav = phased.LinearFMWaveform('PulseWidth',100e-6);
% Transmitter
htx = phased.Transmitter('PeakPower',1e3,'Gain',40);
% Waveform radiator
hrad = phased.Radiator('OperatingFrequency',1e9, ...
    'Sensor',hant);
```

```
% Propagation environment to and from the RadarTarget
hspace = phased.FreeSpace('OperatingFrequency',1e9,...
    'TwoWayPropagation',true);
% Radar target
hr = phased.RadarTarget('MeanRCS',1,'OperatingFrequency',1e9);
% Collector
hc = phased.Collector('OperatingFrequency',1e9,...
    'Sensor',hant);

% Implement system
wf = step(hwav);  % generate waveform
txwf = step(htx,wf);  % transmit waveform
wfrad = step(hrad,txwf,[0 0]'); % radiate waveform
% propagate waveform to and from the RadarTarget
wfprop = step(hspace,wfrad,harraypos.InitialPosition,...
    hrfpos.InitialPosition);
wfreflect = step(hr,wfprop); % reflect waveform
wfcol = step(hc,wfreflect,[45 0]'); % collect waveform
```

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:

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

where *x[n]* and *y[n]* are mutually uncorrelated sequences of Gaussian random variables with zero mean and unit variance.

Model Real and Imaginary Parts of Barrage Jammer Output

Create a barrage jammer with the default effective radiated power of 5000 watts. Generate 500 samples per frame.

```
hjam = phased.BarrageJammer('ERP',5e3,'SamplesPerFrame',500)
y = step(hjam);
subplot(2,1,1)
hist(real(y)); title('Histogram of the Real Part');
subplot(2,1,2)
hist(imag(y)); title('Histogram of the Imaginary Part');
xlabel('watts');
```

Model Effect of Barrage Jammer on Target Echo

This examples 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.

```
hula = phased.ULA(4);
Fs = 1e6;
fc = 1e9;
hwav = phased.RectangularWaveform('PulseWidth',100e-6,...
    'PRF',1e3,'NumPulses',5,'SampleRate',Fs);
htx = phased.Transmitter('PeakPower',1e4,'Gain',20,...
    'InUseOutputPort',true);
hrad = phased.Radiator('Sensor',hula,'OperatingFrequency',fc);
hjammer = phased.BarrageJammer('ERP',1000,...
    'SamplesPerFrame',hwav.NumPulses*hwav.SampleRate/hwav.PRF);
htarget = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1, ...
    'OperatingFrequency', fc);
htargetpath = phased.FreeSpace('TwoWayPropagation',true,'SampleRate',Fs,
    'OperatingFrequency', fc);
hjammerpath = phased.FreeSpace('TwoWayPropagation',false,'SampleRate',Fs,
    'OperatingFrequency', fc);
hcollector = phased.Collector('Sensor',hula,'OperatingFrequency',fc);
hrc = 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
wf = step(hwav);
```

```
% Transmit waveform
[wf,txstatus] = step(htx,wf);
% Radiate pulse toward the target
wf = step(hrad,wf,tgtang);
% Propagate pulse toward the target
wf = step(htargetpath,wf,[0;0;0],targetloc);
% Reflect it off the target
wf = step(htarget,wf);
% Collect the echo
wf = step(hcollector,wf,tgtang);

% Generate the jamming signal
jamsig = step(hjammer);
% Propagate the jamming signal to the array
jamsig = step(hjammerpath,jamsig,jammerloc,[0;0;0]);
% Collect the jamming signal
jamsig = step(hcollector,jamsig,jamang);

% Receive target echo alone and target echo + jamming signal
pulsewave = step(hrc, wf,~txstatus);
pulsewave_jamsig = step(hrc,wf+jamsig,~txstatus);
```

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

```
subplot(2,1,1);
t = unigrid(0,1/Fs,size(pulsewave,1)*1/Fs,'[)');
plot(t,abs(pulsewave(:,1)));
title('Magnitudes of pulse waveform without jamming--Element 1');
ylabel('Magnitude');
subplot(2,1,2);
plot(t,abs(pulsewave_jamsig(:,1)));
title('Magnitudes of pulse waveform with jamming--Element 1');
xlabel('Seconds'); ylabel('Magnitude');
```

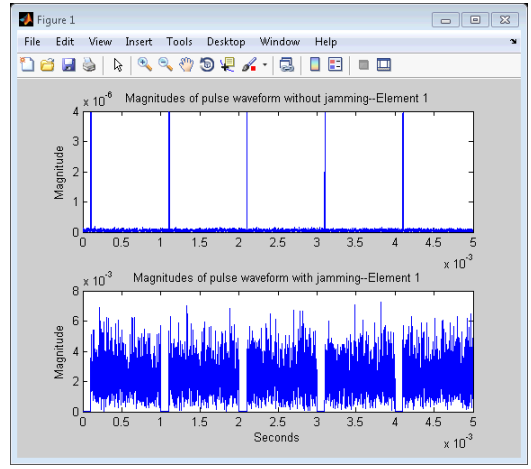


Figure 1
File Edit View Insert Tools Desktop Window Help
Magnitudes of pulse waveform without jamming--Element 1
x 10^-6
Magnitude
x 10^-3
Magnitudes of pulse waveform with jamming--Element 1
x 10^-3
Magnitude
Seconds
x 10^-3

8

Coordinate Systems and Motion Modeling

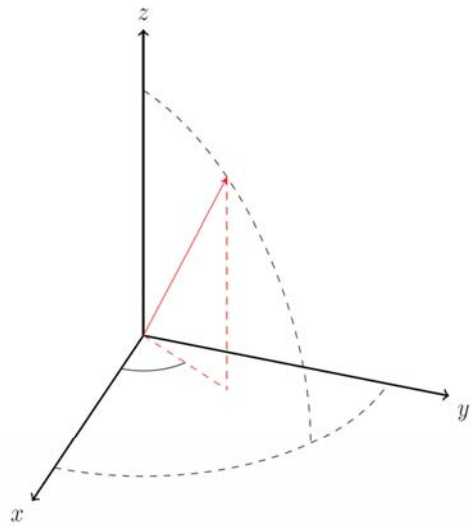
Rectangular and Spherical Coordinates

In this section...

Specifying a point in space with respect to some origin may be done in rectangular, spherical, or cylindrical coordinates. Of these, rectangular and spherical coordinates are the most useful in array processing. This section reviews the rectangular and spherical coordinate systems, emphasizing the conventions of their use in MATLAB and the Phased Array System Toolbox.

Rectangular 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 8-14.

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]}
```

In the Phased Array System Toolbox, vectors defining coordinate axes and points are both specified as column vectors.

Note In the Phased Array System Toolbox, 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 is specifying the axes of a coordinate system or direction, it is a vector. If the column vector is specifying coordinates, it is a point.

Any three linearly independent vectors define a basis for three-dimensional space. However, the Phased Array System Toolbox assumes that the basis vectors you use are orthogonal.

The standard distance measure in space is the l^2, or Euclidean norm. The Euclidean norm of a vector [*x y z*] is defined by:

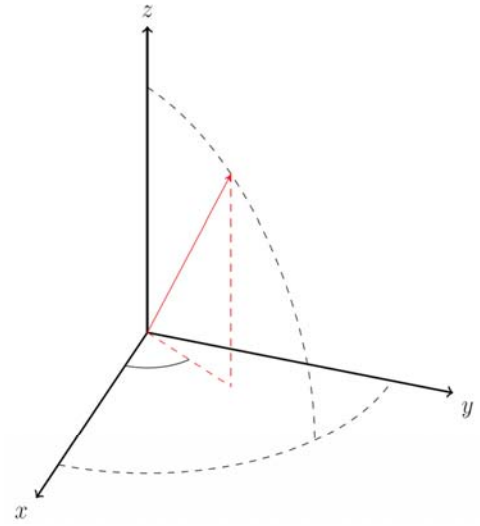
$$\sqrt{x^2 + y^2 + z^2}$$

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:

$$\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

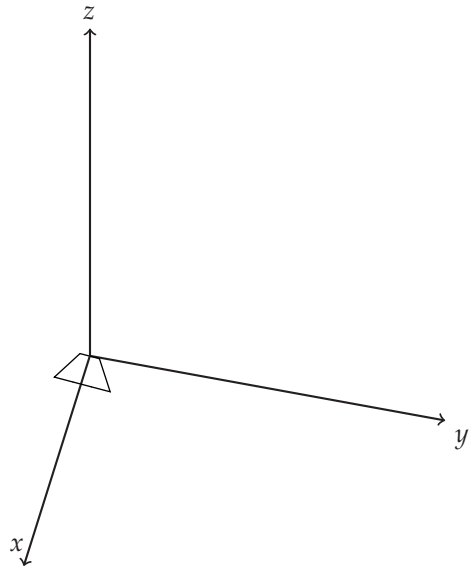
Orienting the coordinate axes

Given an orthonormal set of basis vectors representing the coordinate axes, there are a number of 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.

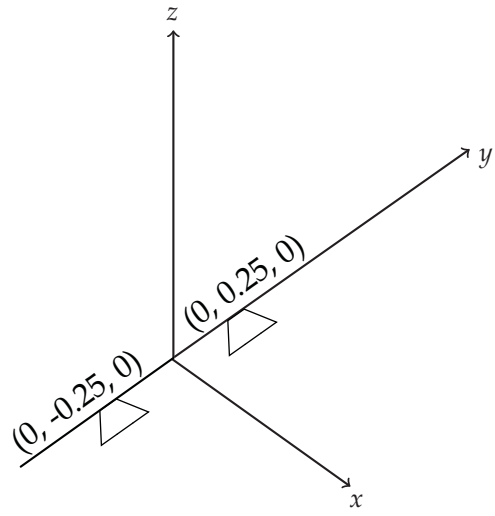
The direction that an antenna is facing when it transmits and receives a signal is referred to as the *boresight*, or *look* direction. In specifying coordinate axes, the convention in the Phased Array System Toolbox designates the positive x axis as boresight direction. The following figure shows a right-handed coordinate system with the positive x axis oriented along the sensor look direction.



The figure shows the single array element located at the origin. For a number of array geometries, it is convenient to define the origin as the phase center of the array. For example, create a default uniform linear array (ULA) and query the element coordinates:

```
H = phased.ULA('NumElements',2,'ElementSpacing',0.5)
getElementPosition(H)
```

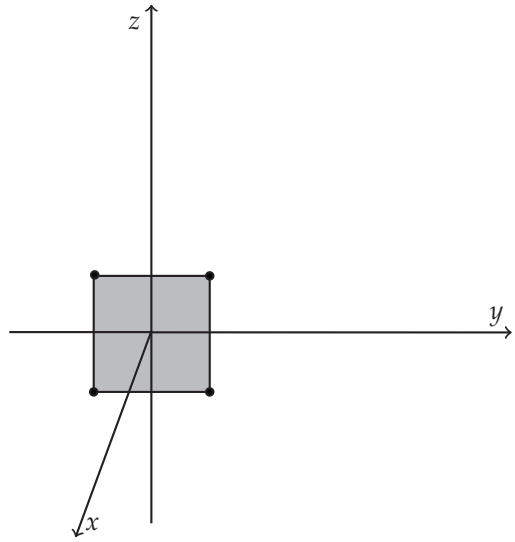
The following figure illustrates the default ULA with array elements located at (0, -0.25, 0) and (0, 0.25, 0).



The next example creates a default uniform rectangular array:

```
H = phased.URA('Size',[2 2],'ElementSpacing',[0.5 0.5])
ElementLocs = getElementPosition(H)
```

By default, the uniform rectangular array has 4 elements. Two elements are located along both the *y* and *z* axes.



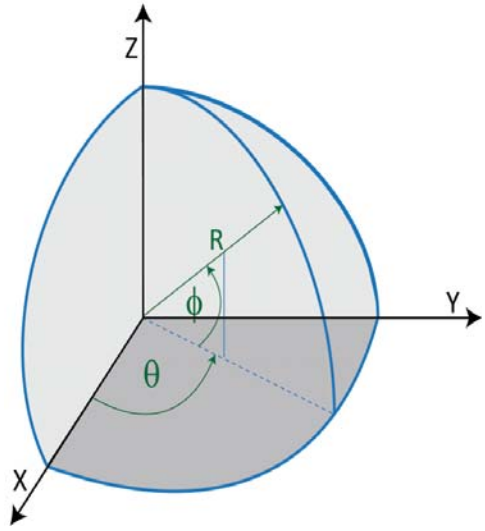
In the two preceding array geometries, the ULA and URA, the look directions of all the array elements are equal. In the case of the ULA, the *y* axis is aligned with the array element apertures. This is referred to as the *array axis*. The array elements have a common look direction, which the toolbox designates as the *x* axis. Note that the *x* axis is the direction orthogonal, or *normal* to the array axis. In the case of the URA, the array element apertures lie in the *yz* plane and also exhibit a common look direction. The *x* axis is normal to the plane containing the array elements. This illustrates the toolbox convention that the *x* axis is the direction normal to the array. In the case of conformal arrays, the elements do not share a common look direction and the direction normal to each element is a property of the individual array elements.

The previous examples are revisited in "Local Coordinate System for a Uniform Linear Array" on page 8-16 and "Local Coordinate System of a Uniform Rectangular Array" on page 8-17 as examples of local coordinate systems.

Spherical Coordinates

Spherical coordinates describe a vector or point in space with two angles and a distance. The two angles are the azimuth angle, θ, and the elevation angle, φ. All angles are specified in degrees. The Phased Array System Toolbox follows

the conventions used in MATLAB regarding the specification and ordering of spherical coordinates. The order of spherical coordinates in the Phased Array System Toolbox is: (θ,φ,*R*). The distance, *R*, is the usual Euclidean norm. The following figure illustrates the definition of the azimuth and elevation angles for an arbitrary vector (solid red line).



The azimuth angle is the angle between the positive *x* axis and the orthogonal projection of the vector onto the *xy* plane. Angles from the positive *x* axis toward the positive *y* axis are positive angles and angles from the positive *x* axis toward the negative *y* axis are negative angles. The azimuth angle assumes values in the interval [-180,180] degrees.

The elevation angle is the angle measured from the orthogonal projection of the vector in the *xy* plane toward the vector. Angles from the *xy* plane toward the positive *z* axis are positive. Angles from the *xy* plane toward the negative *z* axis are negative. The elevation angle assumes a value in the interval [-90,90] degrees.

Note The elevation angle is sometimes defined in the literature as the angle a vector makes with the positive z axis. This is not the definition used in MATLAB and in the Phased Array System Toolbox. The MATLAB convention measures the elevation angle from the xy plane. For more information see `cart2sph` and `sph2cart`.

The following equations define the relationships between rectangular and spherical coordinates following the conventions used in the Phased Array System Toolbox.

To convert from rectangular to spherical coordinates:

$$R = \sqrt{x^2 + y^2 + z^2}$$
$$\theta = \tan^{-1}(y / x)$$
$$\phi = \tan^{-1}(z / \sqrt{x^2 + y^2})$$

To convert from spherical to rectangular coordinates:

$$x = R\cos(\phi)\cos(\theta)$$
$$y = R\cos(\phi)\sin(\theta)$$
$$z = R\sin(\phi)$$

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.

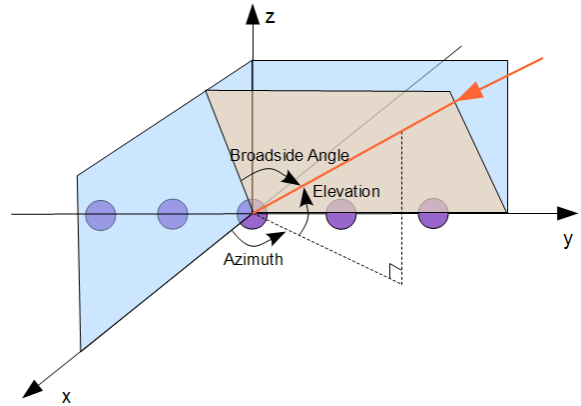
The following example illustrates the conversion between rectangular and spherical coordinates. Both the definitions and the MATLAB functions `cart2sph` and `sph2cart` are used. Because `sph2cart` and `cart2sph` input and output angles specified in radians, use `degtorad` and `radtodeg` to convert angles from degrees to radians.

An object is located 1000 meters from a sensor at an azimuth and elevation angle of 45 degrees. Determine the rectangular coordinates assuming the sensor is located at the origin.

```
[X,Y,Z] = sph2cart(degtorad(45),degtorad(45),1000);
% Using the defining relationships
x=1000*cosd(45)*cosd(45);
y=1000*cosd(45)*sind(45);
z=1000*sind(45);
% compare [X,Y,Z] and [x,y,z]
% convert back to spherical using cart2sph
[THETA,PHI,R]=cart2sph(x,y,z);
% angles in radians, convert to degrees
THETA=radtodeg(THETA);
PHI = radtodeg(PHI);
% Using the definitions
theta=radtodeg(atan2(y,x));
phi = radtodeg(atan2(z,sqrt(x^2+y^2)));
r=sqrt(x^2+y^2+z^2);
% compare [THETA,PHI,R] and [theta,phi,r]
```

Broadside Angle

For the special case of uniform linear arrays (ULA), it is useful to introduce 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 ULAs use the broadside angle instead of the azimuth and elevation angles, because the broadside angle more accurately describes the ability to discern direction of arrival with this geometry. The Phased Array System Toolbox provides two utility 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, θ and φ:

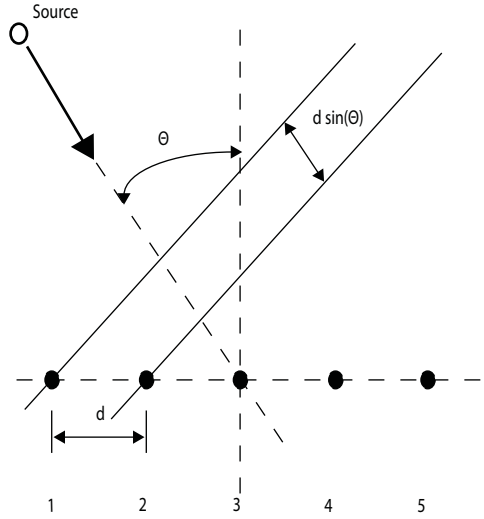
$$\beta = \sin^{-1}(\cos(\phi)\sin(\theta))$$

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 and the broadside angle reduces to the azimuth angle.



Due to 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(θ) where *d* is the distance between array elements. Therefore, the constant time delay between array elements is:

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

where *c* is the speed of the wave.

For broadside angles of plus or minus 90 degrees, the plane wave is incident on the array along the array axis and the time delay between sensors reduces to plus or minus 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 demonstrate 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

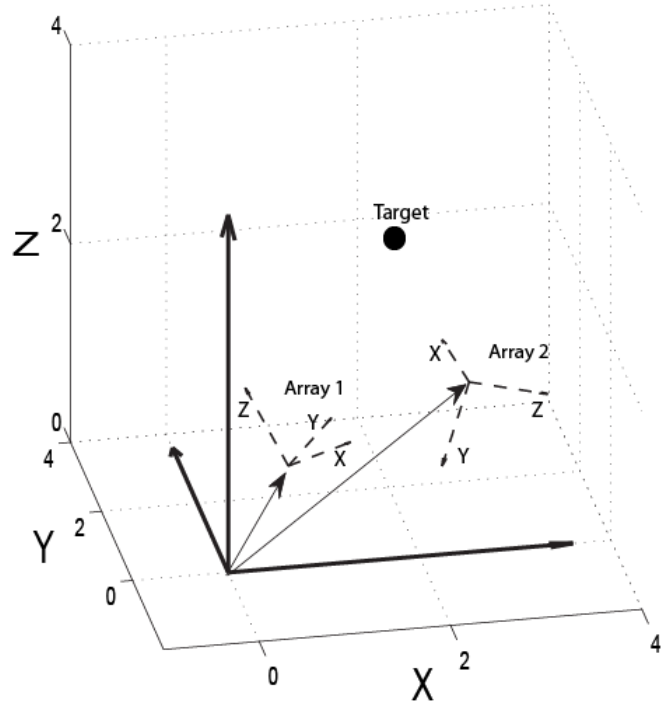
In this section...

In describing rectangular and spherical coordinates, the location of the origin, or zero vector, was assumed. In practice, you must define the coordinate system based on application-specific considerations. There are two general coordinate systems supported in the Phased Array System Toolbox: *global* and *local* coordinate systems. The following sections describe the "Global Coordinate System" on page 8-14, the "Local Coordinate System" on page 8-16, and "Converting between Global and Local Coordinate Systems" on page 8-19.

Global Coordinate System

As the word *global* indicates, the global coordinate system describes the entire environment that you want to model. Within this global coordinate system, you can have several phased array systems, both stationary and mobile. You can also have a number of stationary and mobile targets. Additionally, there are usually stationary and mobile environmental features that produce spurious signals you wish to ignore as well as stationary and mobile sources that are actively attempting to interfere with your phased arrays (jammers). In order to extract useful information from this environment, you often need to analyze data from multiple phased arrays over time. Each phased array senses the environment from its own *local* perspective. To put the information from each phased array into a global perspective, you must know the location of each array in the global coordinate system and the orientation of the array's coordinate axes.

In the following figure, the solid dark axes denote the coordinate axes of a global coordinate system. There are two phased arrays, Array 1, and Array 2. Each of the phased arrays defines its own coordinate system within the global system denoted by the dashed lines. A target is indicated by the black circle.



The two phased arrays detect the target and estimate target characteristics such as range and velocity. In order to translate information about the target derived from the two spatially-separated phased arrays, you must know the positions of the phased arrays and the orientation of their *local* coordinate axes with respect to the global coordinate system.

Note In specifying a global coordinate system, you can designate any point as the origin. The coordinate axes must be orthogonal.

Local Coordinate System

Local coordinate systems are defined by phased arrays located within the global coordinate system. The coordinate axes of a local coordinate system must be orthogonal, but they do not need to be parallel to the global coordinate axes. The local origin may be located anywhere in the global coordinate system and need not be stationary. For example, a vehicle-mounted phased array has its own local coordinate system, which moves within the global coordinate system.

It is common to specify target locations with respect to a local coordinate system in terms of *range* and *direction of arrival*. A target's range corresponds to R, the Euclidean distance in spherical coordinates, and direction of arrival corresponds to θ and φ, the azimuth and elevation angles. The Phased Array System Toolbox follows the MATLAB convention and lists spherical coordinates in the order: (θ,φ,R).

The position of all array elements in the Phased Array System Toolbox toolbox are in local coordinates. The following examples illustrate local coordinate systems for uniform linear, uniform rectangular, and conformal arrays.

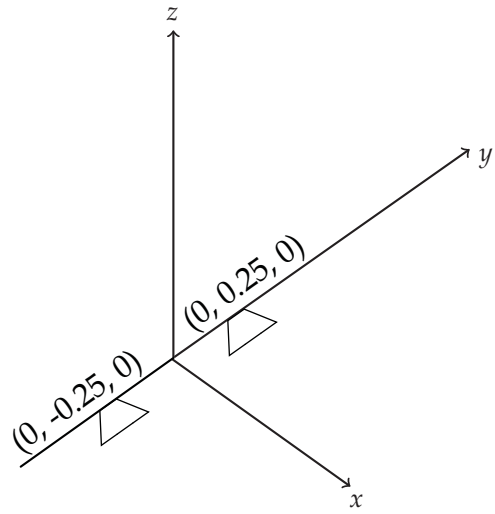
Local Coordinate System for a Uniform Linear Array

For a uniform linear array (ULA), the origin of the local coordinate system is the phase center of the array. The positive x axis is the direction normal to the array, and the elements of the array are located along the y axis. The y axis is referred to as the *array axis*. Define the axis normal to the array as the span of the vector [1 0 0] and the array axis as the span of the vector [0 1 0]. The z axis is the span of the vector [0 0 1], which is the cross product of the two vectors: [1 0 0] and [0 1 0].

Construct a default uniform linear array:

```
H = phased.ULA('NumElements',2,'ElementSpacing',0.5)
getElementPosition(H)
```

The following figure illustrates the default ULA in a local right-handed coordinate system:



The elements are located 0.25 meters from the phase center of the array and the distance between the two elements is 0.5 meters.

Construct a ULA with 8 elements spaced 0.25 meters apart:

```
H = phased.ULA('NumElements',8,'ElementSpacing',0.25)
% Invoke the getElementPosition method
% to see the local coordinates of the elements
getElementPosition(H)
```

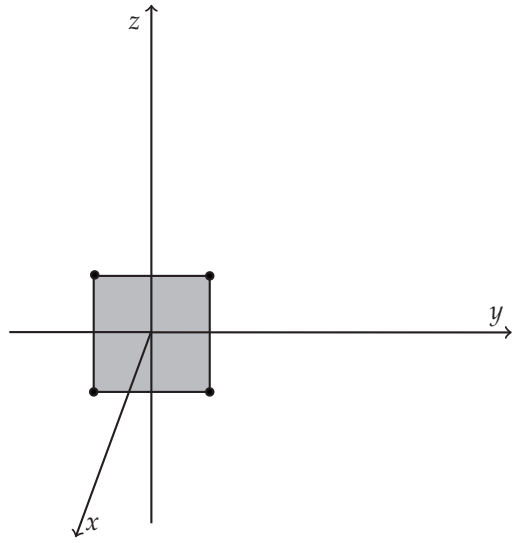
Local Coordinate System of a Uniform Rectangular Array

In a uniform rectangular array (URA), the origin of the local coordinate system is the phase center of the array. The *x* axis is the direction normal to the array, and the array elements are evenly spaced in the *yz* plane.

Construct a default URA:

```
H = phased.URA('Size',[2 2],'ElementSpacing',[0.5 0.5])
getElementPosition(H)
```

The following figure illustrates the default URA:



Construct a uniform rectangular array with two elements along the *y* axis and three elements along the *z* axis.

```
Ha = phased.URA([2 3])
getElementPosition(Ha)
```

Local Coordinate System of a Conformal Array

In a conformal array, the phase center of the array may be defined at an arbitrary point. In principle, the orientation of each element in a conformal array may be different. Therefore, it is convenient to define the array by giving the element locations with respect to the local coordinate system origin along with the azimuth and elevation angles defining the boresight directions.

Construct a default conformal array:

```
H = phased.ConformalArray
% query element position and element normal
H.ElementPosition
H.ElementNormal
```

The default conformal array consists of a single element located at [0 0 0], the origin of the local coordinate system. The boresight direction of the single

element is specified by the azimuth and elevation angles (in degrees) in the `ElementNormal` property, [0 0].

Construct a conformal array with three elements located at [1 0 0], [0 1 0], and [0 -1 0] with respect to the origin. Define the normal direction to the first element as 0 degrees azimuth and elevation. Define the normal direction to the second and third elements as 45 degrees azimuth and elevation.

```
 H = phased.ConformalArray('ElementPosition',[1 0 0; 0 1 0; 0 -1 0]',...
'ElementNormal',[0 45 45; 0 45 45])
```

Converting between Global and Local Coordinate Systems

In many array processing applications, it is necessary to convert between global and local coordinates. The Phased Array System Toolbox has two utility functions, `global2localcoord` and `local2globalcoord`, which enable you to do this conversion.

Convert Local Spherical Coordinates to Global Rectangular Coordinates

Assume a stationary target 1000 meters from a URA at a 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` is in the form (θ,φ,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 meters.

Motion Modeling in Phased Array Systems

A critical component in phased array system applications is the ability to model motion in space. This 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*.

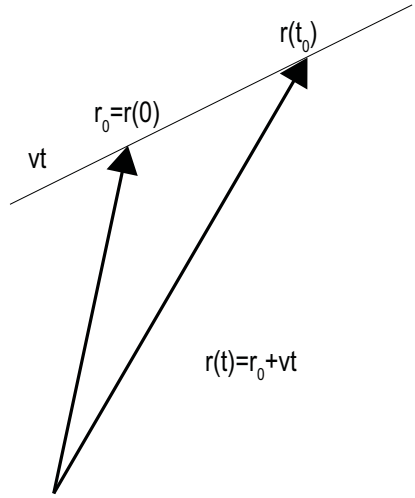
Extended bodies can undergo both *translational* and *rotational* motion in space. The Phased Array System Toolbox software currently supports modeling of translational motion.

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 8-14. 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 8-14 for a description of global and local coordinate systems in the Phased Array System Toolbox. Finally, for platforms with non-constant 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 one millisecond. Set the time step equal to pulse repetition interval.

```
PRF = 1e3;
Tstep = 1/PRF;
Nsteps = 10;
```

Next, construct a platform object specifying its 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 `OrientationAxes` property. You can view the value of this property by entering `hplat.OrientationAxes` at the MATLAB command prompt. Because the `OrientationAxes` 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 its position. Accordingly, you expect the final position to differ from the initial position by 0.30 meters. Confirm this by examining the value of `DistTravel`.

Platform Motion with Non-constant 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 this with `phased.Platform` because the `Velocity` property is *tunable*. See "Changing System Object Properties" 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 examples 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. `global2localcoord` 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 see that this is the case.

See the Introduction to Space-time Adaptive Processing demo for a detailed example of using `phased.Platform` to model the motion of a radar, target, and jammer.

Doppler Shift and 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 narowband 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.

Converting speed to Doppler shift

Assume a target approaching a stationary receiver with a radial speed of 23 meters per second. The target is reflecting a narrowband electromagnetic wave with a frequency of 1 GHz. Estimate the one-way Doppler shift.

```
freq = 1e9;
lambda = physconst('lightspeed')/freq;
DopplerShift = speed2dop(23,lambda)
```

The one-way Doppler shift is approximately 76.72 Hz. The fact that the target is approaching the receiver results in a positive Doppler shift.

You can use `dop2speed` to determine the radial speed of a target relative to a receiver based on the observed Doppler shift.

Converting Doppler shift to speed

Assume you observe a Doppler shift of 400 Hz for a waveform with a frequency of 9 GHz. Determine the radial velocity of the target.

```
freq = 9e9;
lambda = physconst('lightspeed')/freq;
speed = dop2speed(400,lambda)
```

The target speed is approximately 13.32 m/sec.

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 target (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.

To demonstrate pulse Doppler processing with the Phased Array System Toolbox software, 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 non-fluctuating radar cross section (RCS) of 1 square meter located initially at `[1000; 1000; 0]` and moving with a constant velocity of `[-100; -100; 0]`. The antenna operates at a frequency of 1 GHz and illuminates the target with 10 rectangular pulses at a PRF of 10 kHz.

Define the System objects needed for this example and set their properties. Seed the random number generator for the `phased.ReceiverPreamp` object to produce repeatable results.

```
hwav = phased.RectangularWaveform('SampleRate',5e6,'PulseWidth',6e-7,...
    'OutputFormat','Pulses','NumPulses',1,'PRF',1e4);
htgt = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1,...
    'OperatingFrequency',1e9);
htgtloc = phased.Platform('InitialPosition',[1000; 1000; 0],...
    'Velocity',[-100; -100; 0]);
hant = phased.IsotropicAntennaElement('FrequencyRange',[5e8 5e9]);
htrans = phased.Transmitter('PeakPower',5e3,'Gain',20,...
'InUseOutputPort',true);
htransloc = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0]);
hrad = phased.Radiator('OperatingFrequency',1e9,'Sensor',hant);
hcol = phased.Collector('OperatingFrequency',1e9,'Sensor',hant);
hspace = phased.FreeSpace('SampleRate',hwav.SampleRate,...
    'OperatingFrequency',1e9,'TwoWayPropagation',false);
hrx = phased.ReceiverPreamp('Gain',0,'LossFactor',0,'SampleRate',5e6,...
    'NoiseBandwidth',5e6/2,'NoiseFigure',5,'EnableInputPort',true,...
    'SeedSource','Property','Seed',1e3);
```

The following 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's position with the specified constant velocity.

```
NumPulses = 10;
sig = step(hwav);                       % get waveform
transpos = htransloc.InitialPosition;   % get transmitter position
rxsig = zeros(length(sig),NumPulses);
% transmit and receive ten pulses
for n = 1:NumPulses
    % update target position
    [tgtpos,tgtang] = step(htgtloc,1/hwav.PRF);
```

```
    [tgtrng,tgtang] = rangeangle(tgtpos,transpos);
    tpos(n) = tgtrng;
    [txsig,txstatus] = step(htrans,sig);        % transmit waveform
    txsig = step(hrad,txsig,tgtang);       % radiate waveform toward targ
    txsig = step(hspace,txsig,transpos,tgtpos); % propagate waveform to t
    txsig = step(htgt,txsig);                   % reflect the signal
    % propagate waveform from the target to the transmiter
    txsig = step(hspace,txsig,tgtpos,transpos);
    txsig = step(hcol,txsig,tgtang); % collect signal
    rxsig(:,n) = step(hrx,txsig,~txstatus); % receive the signal
  end
```

`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 = hwav.PRF;
fs = hwav.SampleRate;
fasttime = unigrid(0,1/fs,1/prf,'[)');
rangebins = (physconst('lightspeed')*fasttime)/2;
```

The next step is to detect range bins which contain targets. In this simple scenario, no matched filtering or time-varying gain compensation is utilized. See the Doppler Estimation demo for an example using matched filtering and range-dependent gain compensation to improve the SNR.

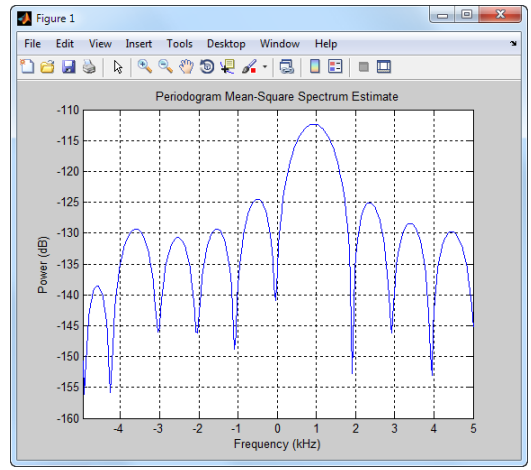
In this example, set the false-alarm probability to `1e-9`. Use noncoherent integration of the ten rectangular pulses and determine the corresponding threshold for detection in white Gaussian noise. Because this scenario contains only one target, take the largest peak above the threshold. Display the estimated target range.

```
probfa = 1e-9;
npower = noisepow(hrx.NoiseBandwidth,...
    hrx.NoiseFigure,hrx.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));
fprintf('Estimated range of the target is %4.2f meters.\n', range_estimat
```

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 `spectrum.periodogram` and find the peak frequency. Convert the peak Doppler frequency to a speed using `dop2speed`. 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 target.

```
ts = rxsig(range_detect(1),:).';
hper = spectrum.periodogram;
% zero pad the data to interpolate the spectral estimate
dopspec = msspectrum(hper,ts,'Fs',prf,'NFFT',256,'CenterDC',true);
plot(dopspec);
[Y,I] = max(dopspec.Data);
lambda = physconst('lightspeed')/1e9;
tgtspeed = dop2speed(dopspec.Frequencies(I)/2,lambda);
fprintf('Estimated target speed is %3.1f m/sec.\n',tgtspeed);
if dopspec.Frequencies(I)>0
    fprintf('The target is approaching the radar.\n');
else
    fprintf('The target is moving away from the radar.\n');
end
```



The code produces:

```
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.

See Doppler Estimation for a demo of pulse-Doppler processing with a single antenna and Scan Radar Using a Uniform Rectangular Array for an example of pulse-Doppler processing with a planar array.