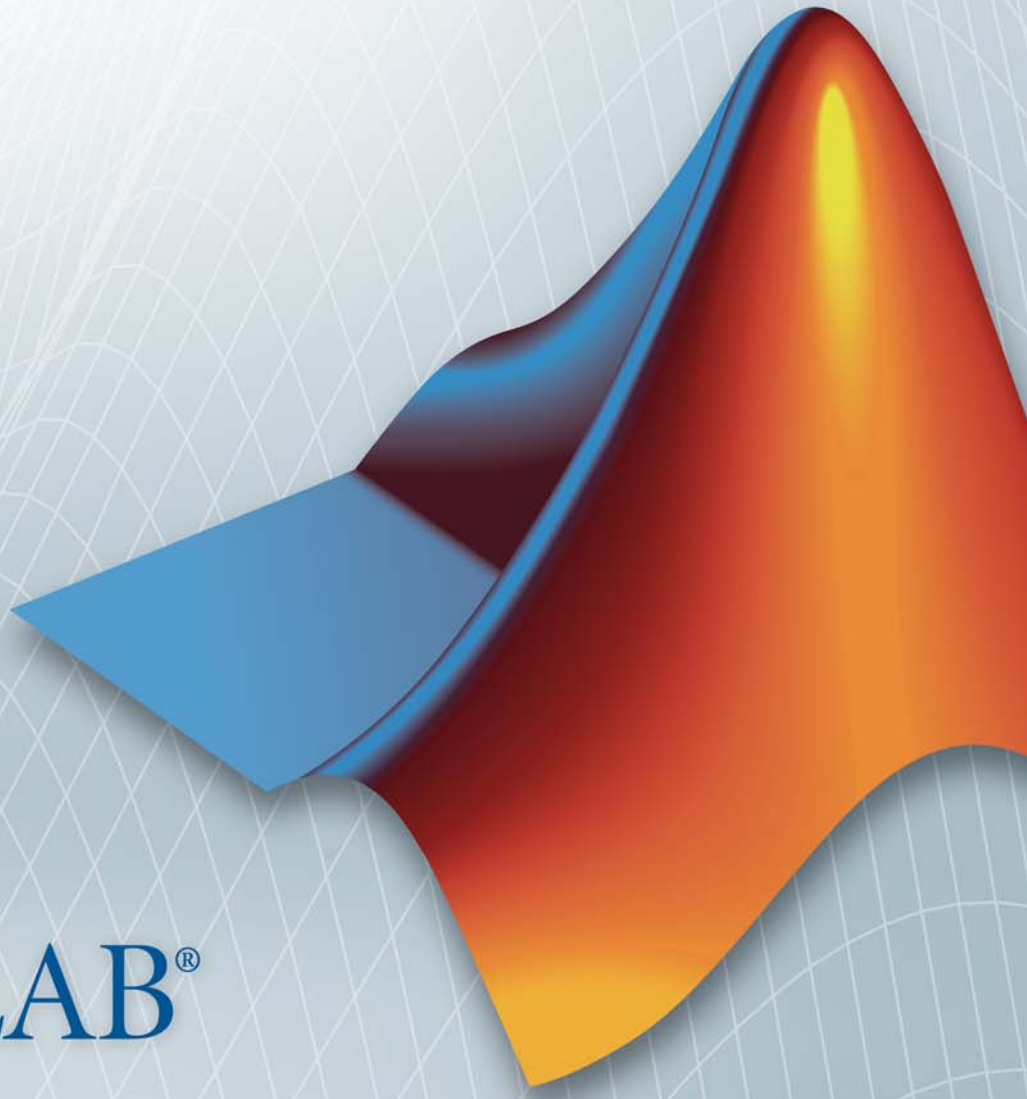
Phased Array System Toolbox™

User's Guide

R2012a



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)
September 2011	Online only	Revised for Version 1.1 (R2011b)
March 2012	Online only	Revised for Version 1.2 (R2012a)

Contents

Phased Arrays

1 Antenna and Microphone Elements

2 Array Geometries and Analysis

3 Signal Radiation and Collection

4 Waveforms, Transmitter, and Receiver

5 Beamforming

6 Direction-of-Arrival (DOA) Estimation

7 Space-Time Adaptive Processing (STAP)

8 Detection

9 Environment and Target Models

10 Coordinate Systems and Motion Modeling

11 Define New System Objects

Phased Arrays

1

Antenna and Microphone Elements

Isotropic Antenna Element

In this section...

Support for Isotropic Antenna Elements

An isotropic antenna element radiates equal power in all nonbaffled directions. To construct an isotropic antenna, use `phased.IsotropicAntennaElement`. When you use this object, you must specify these aspects of the antenna:

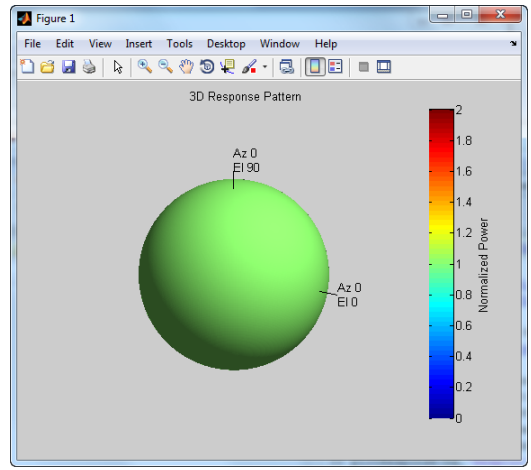
- Operating frequency range of the antenna
- Whether the response of the antenna is baffled at azimuth angles outside the interval [–90,90]

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

Backbaffled Isotropic Antenna

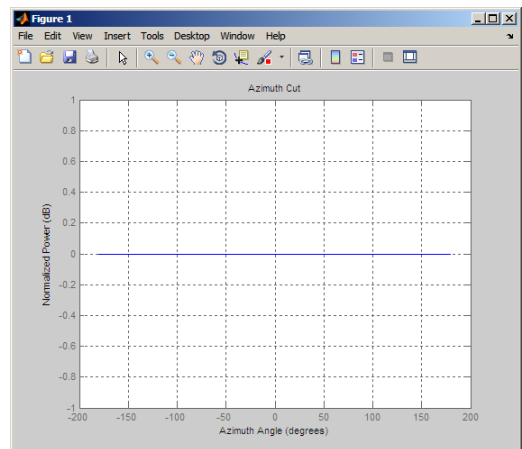
This example shows how to construct a backbaffled isotropic antenna element with a uniform frequency response over azimuth angles from [–180,180] degrees and elevation angles from [–90,90] degrees. The antenna operates between 300 megahertz (MHz) and 1 gigahertz (GHz). Plot the antenna response at 1 GHz.

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



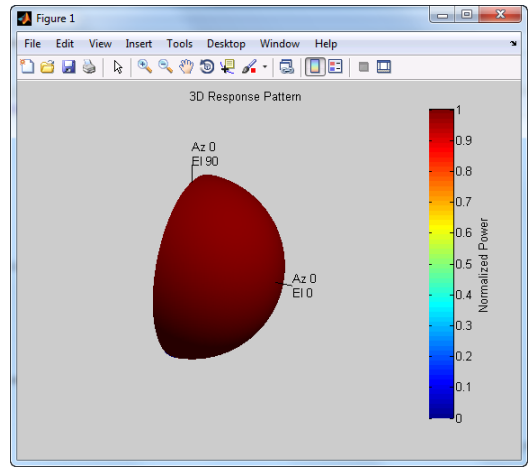
`plotResponse` is a method of `phased.IsotropicAntennaElement`. By default, `plotResponse` plots the response of the antenna element in decibels (dB) at zero degrees elevation.

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



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

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



Element Response of Backbaffled Isotropic Antenna Element

This example shows how to design a backbaffled isotropic antenna element and obtain the response of that element.

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 and 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 whose row dimension equals the number of azimuth angles in `respazangles` and whose column dimension equals 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). Because the elevation angles are equal to +/– 90 degrees, these responses are equal to one even though the `BackBaffled` property is set to `true`. Thus, the resulting elevation cut degenerates into a point.

Cosine Antenna Element

In this section...

Support for Cosine Antenna Elements

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

The *cosine response*, or *cosine pattern*, is given by:

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

In this expression:

- az is the azimuth angle.
- el is the elevation angle.
- The exponents m and n are real numbers greater than or equal to 1.

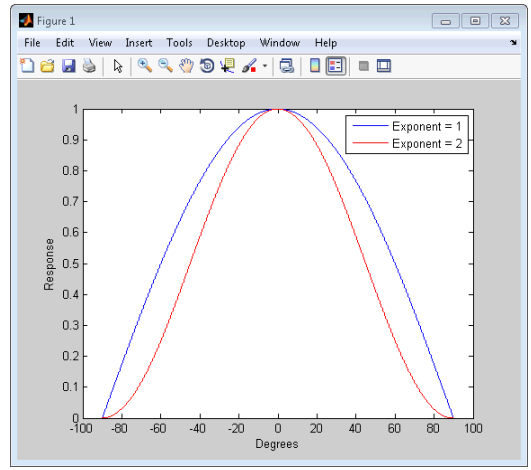
The response is defined for azimuth and elevation angles between –90 and 90 degrees, inclusive. There is no response at the back of a cosine antenna. The cosine response pattern achieves a maximum value of 1 at 0 degrees azimuth and elevation. Raising the response pattern to powers greater than one concentrates the response in azimuth or elevation.

When you use the cosine antenna element, you specify the exponent of the cosine pattern and the operating frequency range of the antenna.

Concentration of Cosine Response

This example shows how to visualize the effect of concentrating the response. The example computes and plots 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');
```



Cosine Antenna Element Operating from 1 to 10 GHz

This example shows how to 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');
```

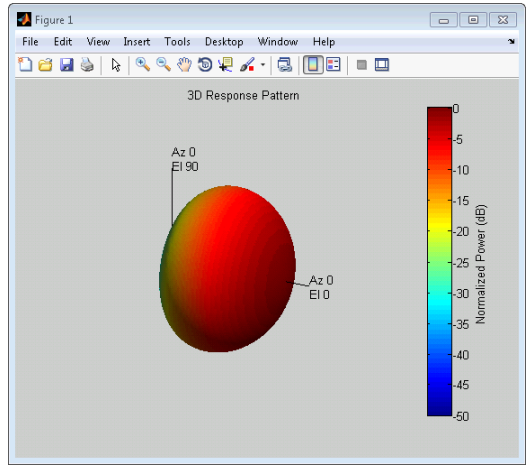


Figure 1
File Edit View Insert Tools Desktop Window Help
3D Response Pattern
Az 0
El 90
Az 0
El 0
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-50
Normalized Power (dB)

Custom Antenna Element

In this section...

Support for Custom Antenna Elements

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

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

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

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

Antenna with Custom Radiation Pattern

This example shows how to construct a custom antenna element object. The radiation pattern is 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

In this section...

Support for Omnidirectional Microphones

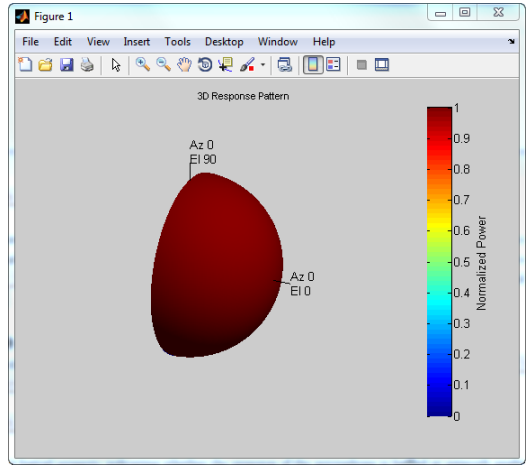
An omnidirectional microphone has a response which is equal to one in all nonbaffled directions. The `phased.OmnidirectionalMicrophoneElement` object enables you to model an omnidirectional microphone. When you use this object, you must specify these aspects of the microphone:

- Operating frequency range of the microphone
- Whether the response of the microphone is baffled at azimuth angles outside the interval [–90,90]

Backbaffled Omnidirectional Microphone

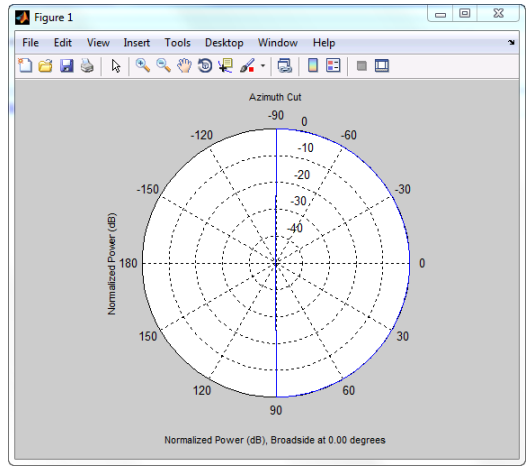
Construct an omnidirectional microphone element using the human audible frequency range of 20 to 20,000 Hz. Baffle the microphone response for azimuth angles outside of +/– 90 degrees. Plot the microphone's power response at 1000 Hz in polar form.

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



In many applications, you sometimes need to examine the microphone's directionality, or polar pattern. To do so, set the `RespCut` argument of `plotResponse` to one of the 2-D options and set the `Format` argument to `'Polar'`. The 2-D options for the cut of the response are `'Az'` (default), and `'El'`.

```
% 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

In this section...

Support for Custom Microphone Elements

You can model a microphone with your custom response using `phased.CustomMicrophoneElement`. When you use `phased.CustomMicrophoneElement`, you must specify these aspects of the microphone:

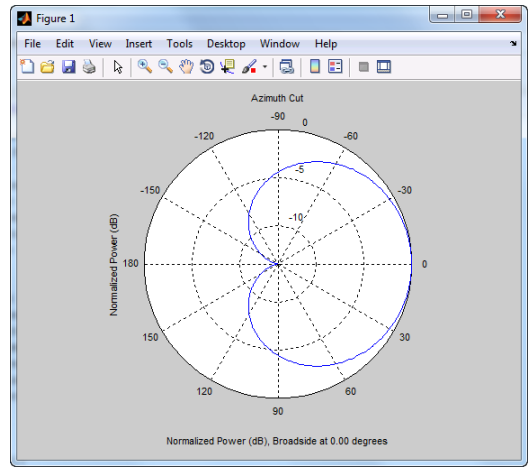
- Frequencies where you specify your response
- Frequency response corresponding to the specified frequencies
- Frequencies and angles at which the microphone's polar pattern is measured.
- Magnitude response of the microphone.

Microphone with Custom Response Pattern

This example shows how to 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])
```

2

Array Geometries and Analysis

Uniform Linear Array

In this section...

Support for Uniform Linear Arrays

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

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

Positions of Elements in Array

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 Coordinates" on page 10-2 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.

Identical Elements in Array

The default element for a ULA is the `phased.IsotropicAntennaElement` object. You can specify an alternative element by changing the `Element` property.

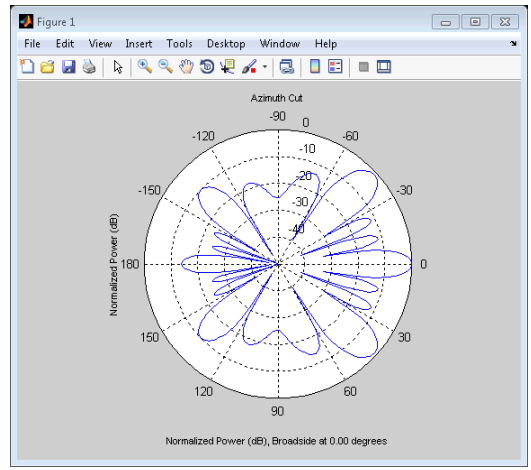
Uniform Linear Microphone Array

This example shows how to 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, 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`.

Response of Array Elements

To obtain the responses of your array elements, 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.

Signal Delay Between Array Elements

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 10-15 for a description of global and local coordinate systems. Negative delays indicate that the signal arrives at an element before reaching the phase center of the array. Because the waveform arrives from an azimuth angle of –90 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 m/s, which is the approximate speed of sound at sea level.

Steering Vector

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

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

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

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.

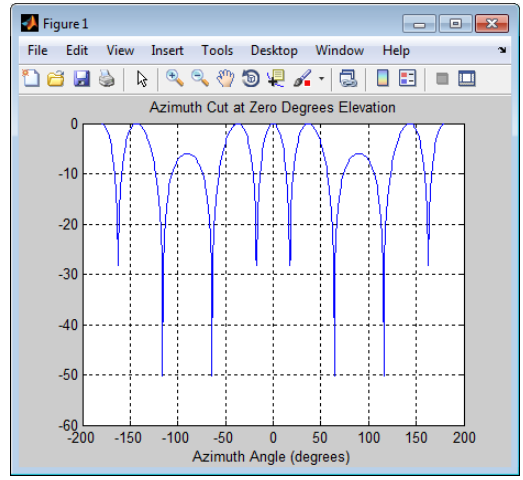
```
hdelay = phased.ElementDelay('SensorArray',hula);
tau = step(hdelay,[45;10]);
exp(-1j*2*pi*1e9*tau)
```

Array Response

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

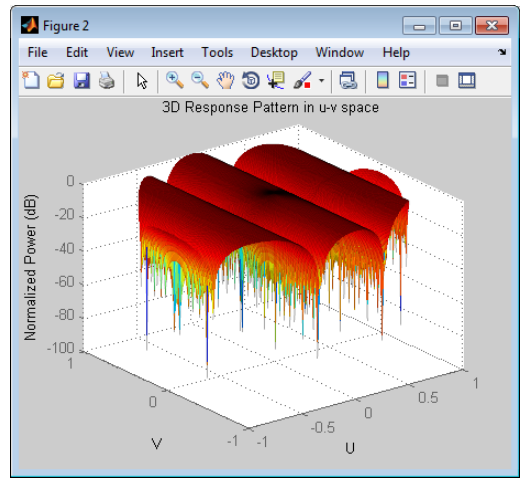
Construct a two-element ULA with elements spaced at 0.5 m. Obtain the array's magnitude response (absolute value of the complex-valued array response) for azimuth angles `-180:180` and plot the normalized magnitude response in decibels.

```
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
azangles = -180:180;
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 (degrees)');
```



Visualize the array response using the `plotResponse` method. This example uses options to create a 3-D plot of the response in *u*/*v* space; other plotting options are available.

```
figure;
plotResponse(hula,1e9,physconst('LightSpeed'),...
   'Format','UV','RespCut','3D')
```



Reception of Plane Wave Across Array

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

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

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 preceding code is equivalent to the following.

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

```
y1 = x*sv.';
```

Uniform Rectangular Array

In this section...

Support for Uniform Rectangular Arrays

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

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

Uniform Rectangular Array with Isotropic Antenna Elements

This example shows how to create a URA, get information about its element positions, response, and delays, and simulate its reception of two sine waves.

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

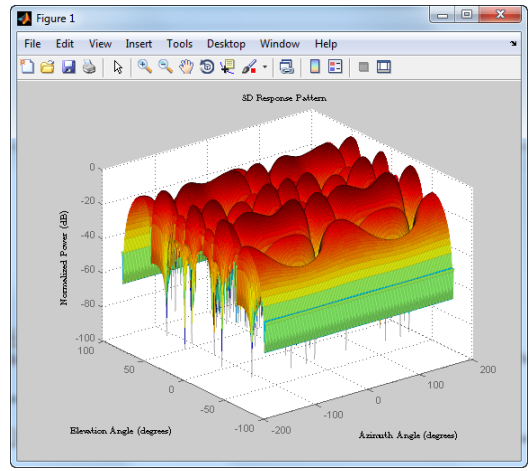
```
hura = phased.URA([2 3]);
pos = getElementPosition(hura);
```

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')
```



Calculate the element delays for signals arriving from +/– 45 degrees azimuth and 0 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 following code 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

In this section...

Support for Arrays with Custom Geometry

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 nonrectangular geometry, such as a circular array. You can also use `phased.ConformalArray` to design nonplanar arrays.

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

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

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

```
hcon = phased.ConformalArray
```

This default conformal array consists 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.

Array with Custom Geometry

This example shows how to construct and visualize a three-element conformal array.

Define the locations and normal directions of the elements. In this case, the elements are located at `[1;0;0]`, `[0;1;0]`, and `[0;-1;0]`. The element normal azimuth angles are 0, 120, and –120 degrees respectively. All normal elevation angles are 0 degrees.

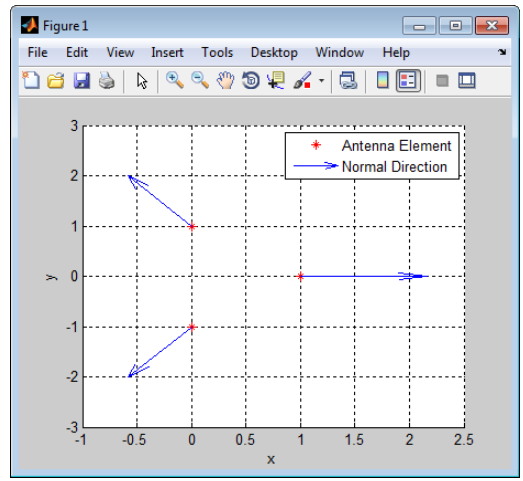
```
xpos = [1 0 0];
ypos = [0 1 -1];
zpos = [0 0 0];
normal_az = [0 120 -120];
normal_el = [0 0 0];
```

Define a conformal array with those elements.

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

Plot the positions and normal directions of the elements.

```
plot3(xpos,ypos,zpos,'r*')
hold on;
[xnorm,ynorm,znorm] = sph2cart(degtorad(normal_az),...
   degtorad(normal_el),[1 1 1]);
quiver3(xpos,ypos,zpos,xnorm,ynorm,znorm)
grid on;
xlabel('x'); ylabel('y'); zlabel('z');
legend('Antenna Element','Normal Direction',...
   'Location','NorthEast')
view(0,90)
```

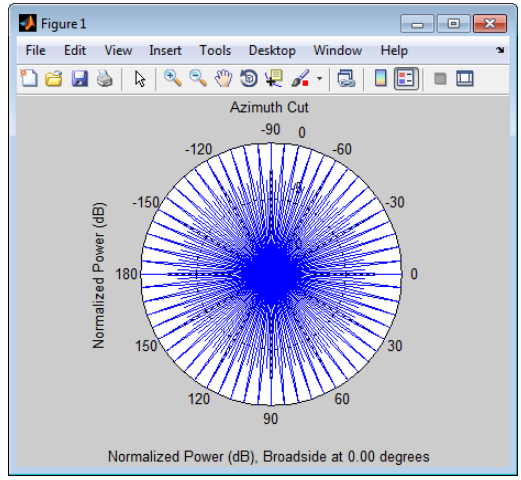


Uniform Circular Array

This example shows how to 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')
```



Subarrays Within Arrays

In this section...

Definition of Subarrays

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

Benefits of Using Subarrays

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

Support for Subarrays Within Arrays

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

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

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

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

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

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

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

- `phased.SubbandPhaseShiftBeamformer`
- `phased.WidebandCollector`

Rectangular Array Partitioned into Linear Subarrays

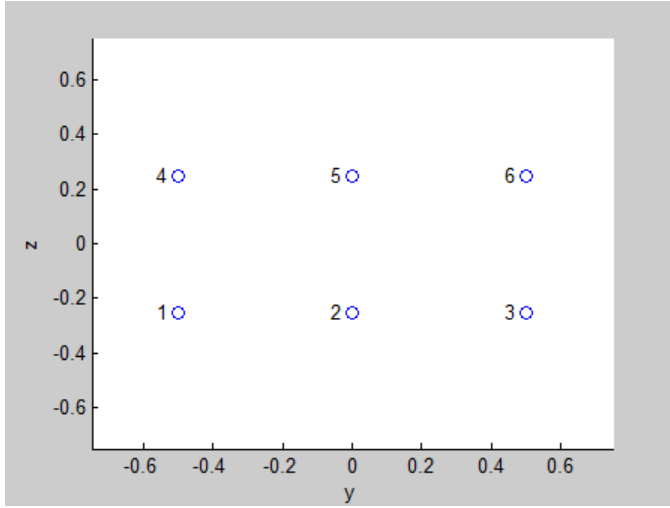
This example shows how to set up a rectangular array containing linear subarrays. The example also finds the phase centers of the subarrays.

Create a 3-by-2 rectangular array.

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

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

```
pos = getElementPosition(ha);
y = pos(2,:); z = pos(3,:);
scatter(y,z)
axis([-0.75 0.75 -0.75 0.75])
xlabel('y'); ylabel('z')

hold on
for jj = 1:length(y)
   text(y(jj)-0.06,z(jj),num2str(jj));
end
hold off
```



Create an array consisting of three 2-element linear subarrays parallel to the *z*-axis. Use the numbers in the plot to help form the matrix for the `SubarraySelection` property. Also, find the phase centers of the three subarrays.

```
subarray1 = [1 0 0 1 0 0; 0 1 0 0 1 0; 0 0 1 0 0 1];
hpa1 = phased.PartitionedArray('Array',ha,...
   'SubarraySelection',subarray1);
subarraypos1 = getSubarrayPosition(hpa1);
```

Create another array consisting of two 3-element linear subarrays parallel to the *y*-axis. Find the phase centers of the two subarrays.

```
subarray2 = [1 1 1 0 0 0; 0 0 0 1 1 1];
hpa2 = phased.PartitionedArray('Array',ha,...
   'SubarraySelection',subarray2);
subarraypos2 = getSubarrayPosition(hpa2);
```

Linear Subarray Replicated to Form Rectangular Array

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

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

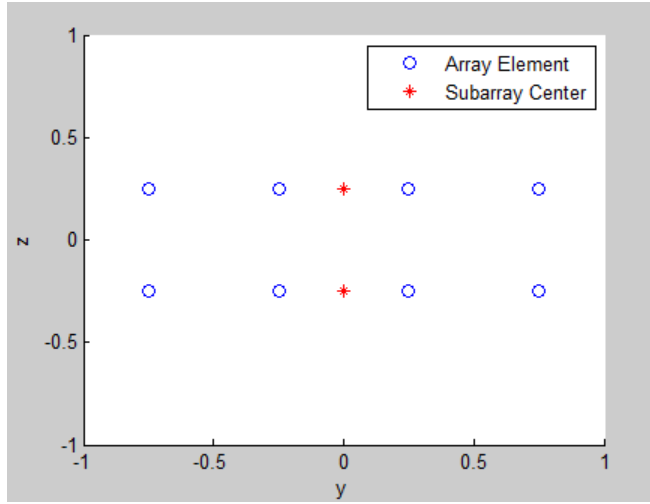
```
ha = phased.ULA('NumElements',4);
```

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

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

Plot the positions of the array elements and the phase centers of the subarrays. The plot is in the *yz* plane because all the *x* coordinates are zero.

```
pos = getElementPosition(hrs);
y = pos(2,:); z = pos(3,:);
scatter(y,z,'bo')
hold on;

subarraypos = getSubarrayPosition(hrs);
sy = subarraypos(2,:); sz = subarraypos(3,:);
scatter(sy,sz,'r*');

axis([-1 1 -1 1])
xlabel('y'); ylabel('z')
legend('Array Element','Subarray Center')
hold off
```



Linear Subarray Replicated in a Custom Grid

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

Create a 4-element linear array.

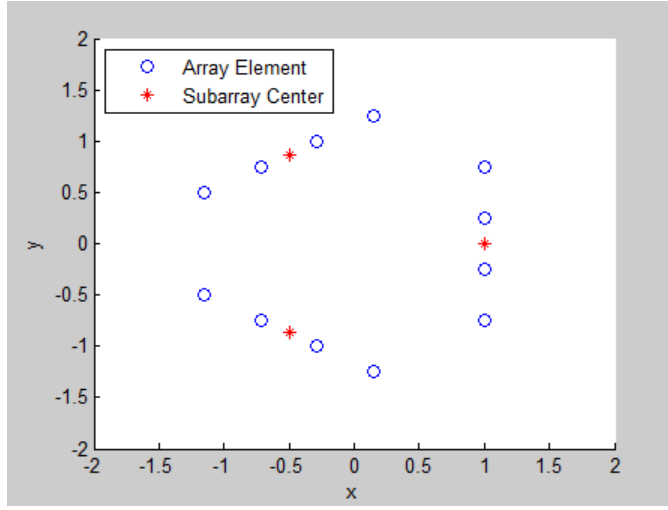
```
hele = phased.CosineAntennaElement('CosinePower',1);
ha = phased.ULA('NumElements',4,'Element',hele);
```

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

```
vertex_ang = [60 180 -60];
vertex = 2*[cosd(vertex_ang); sind(vertex_ang); zeros(1,3)];
subarray_pos = 1/2*[...
   (vertex(:,1)+vertex(:,2)) ...
   (vertex(:,2)+vertex(:,3)) ...
   (vertex(:,3)+vertex(:,1))];
hrs = phased.ReplicatedSubarray('Subarray',ha,...
   'Layout','Custom',...
   'SubarrayPosition',subarray_pos,...
   'SubarrayNormal',[120 0;-120 0;0 0].');
```

Plot the positions of the array elements and the phase centers of the subarrays. The plot is in the *xy* plane because all the *z* coordinates are zero.

```
element_pos = getElementPosition(hrs);
scatter(element_pos(1,:),element_pos(2,:),'bo')
hold on;
scatter(subarray_pos(1,:),subarray_pos(2,:),'r*');
axis([-2 2 -2 2])
xlabel('x'); ylabel('y')
legend('Array Element','Subarray Center',...
   'Location','NorthWest')
hold off
```



Related Examples

- Subarrays in Phased Array Antennas

3

Signal Radiation and Collection

Signal Radiation

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

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

- Whether the output of all sensor elements is combined
- Operating frequency of the array
- Propagation speed of the wave
- Sensor (single element) or sensor array
- Whether to apply weights to signals radiated by different elements in the array. If you want to apply weights, you specify them 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 4-27 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.

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

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

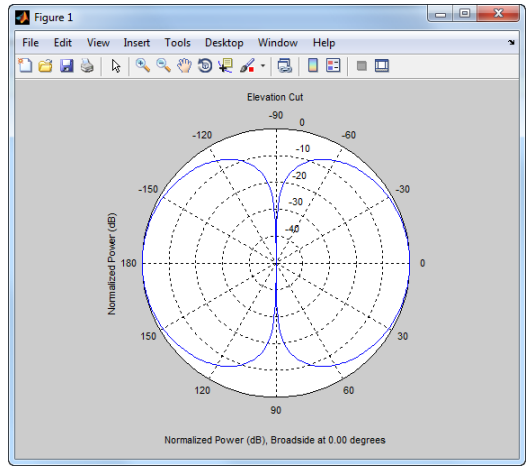
```
% default ULA
hula = phased.ULA('NumElements',2,'ElementSpacing',0.5);
% Construct steering vector
hsv = phased.SteeringVector('SensorArray',hula);
sv = step(hsv,3e8,[45;0]);
x =[1 -1 1 -1]';
y1 = x*sv.';
% compare y1 to y
```

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.

When you use the wideband collector, you must specify these aspects of the collector:

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

4

Waveforms, Transmitter, and Receiver

Rectangular Pulse Waveforms

In this section...

Definition of Rectangular Pulse Waveform

Define the following function of time:

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

Assume that a radar transmits a signal of the form:

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

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

How to Create Rectangular Pulse Waveforms

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

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

Rectangular Waveform Plot

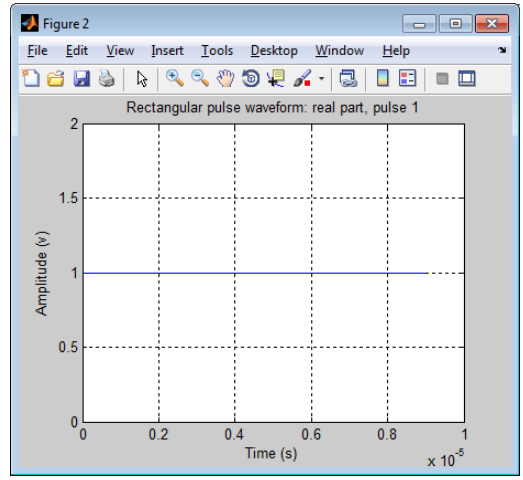
This example shows how to create a rectangular pulse waveform variable using `phased.RectangularWaveform`. The example also plots the pulse and finds the bandwidth of the pulse.

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

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

Plot a single rectangular pulse by calling `plot` directly on the rectangular waveform variable.

```
figure;
plot(hrect)
```



`plot` is a method of `phased.RectangularWaveform`. This `plot` method produces an annotated graph of your pulse waveform.

Find the bandwidth of the rectangular pulse.

```
bw = bandwidth(hrect);
```

The bandwidth of a rectangular pulse in hertz is approximately the reciprocal of that pulse's duration. That is, `bw` is approximately `1/hrect.PulseWidth`.

Pulses of Rectangular Waveform

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

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

```
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 µs.

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

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

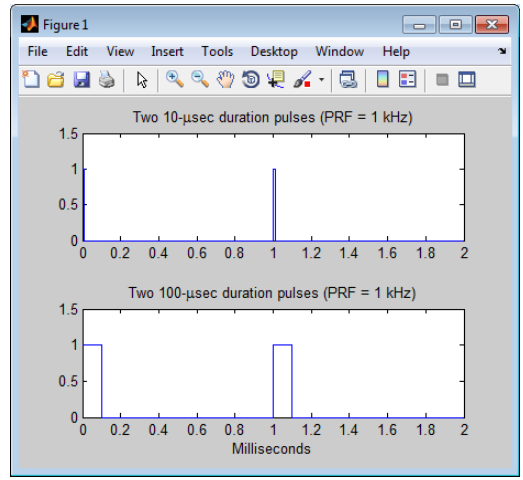
Use the `step` method to return two pulses of your rectangular pulse waveforms.

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

Plot the real part of 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,real(y)); axis([0 totaldur*1e3 0 1.5]);
title('Two 10-\musec duration pulses (PRF = 1 kHz)');
set(gca,'XTick',0:0.2:totaldur*1e3)
subplot(2,1,2);
plot(t.*1000,real(y1)); axis([0 totaldur*1e3 0 1.5]);
xlabel('Milliseconds');
```

```
title('Two 100-\musec duration pulses (PRF = 1 kHz)');
set(gca,'XTick',0:0.2:totaldur*1e3)
```



Linear Frequency Modulated Pulse Waveforms

In this section...

Benefits of Using Linear FM Pulse Waveform

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

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

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

Definition of Linear FM Pulse Waveform

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

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

where ß is the bandwidth and τ is the pulse duration.

If you denote the phase by 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.

How to Create Linear FM Pulse Waveforms

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

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

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

  - The Gaussian envelope is:

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

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

Configuration of Linear FM Pulse Waveform

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

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

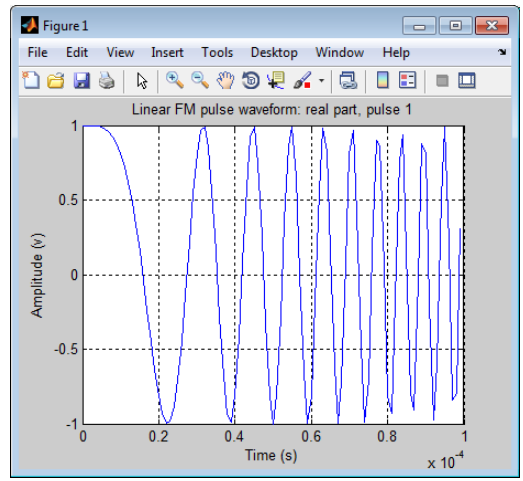
```
hfm1 = phased.LinearFMWaveform('SampleRate',1e6,...
    'PulseWidth',5e-5,'PRF',1e4,...
    'SweepBandwidth',1e5,'SweepDirection','Up',...
    'Envelope','Rectangular',...
    'OutputFormat','Pulses','NumPulses',1);
```

Linear FM Pulse Waveform Plots

This example shows how to design a linear FM pulse waveform, plot the real part of the waveform, and plot one pulse repetition interval.

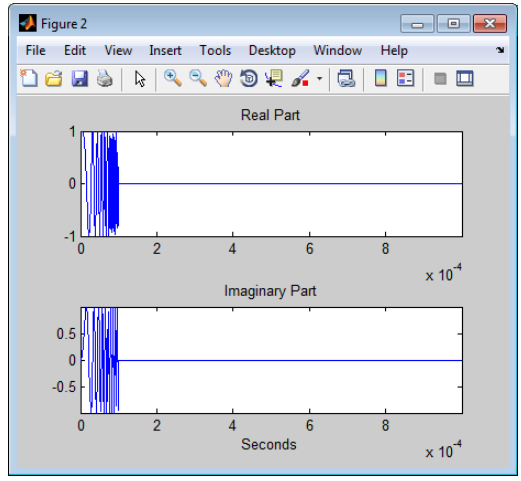
Design a linear FM pulse waveform with a duration of 100 μs, a bandwidth of 200 kHz, and a PRF of 1 kHz. Use the default values for the other properties. Compute the time-bandwidth product and plot the real part of the pulse waveform.

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



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

```
y = step(hfm);
t = unigrid(0,1/hfm.SampleRate,1/hfm.PRF,'[)');
figure;
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;
```



Ambiguity Function of Linear FM Waveform

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

Define and set up the linear FM waveform.

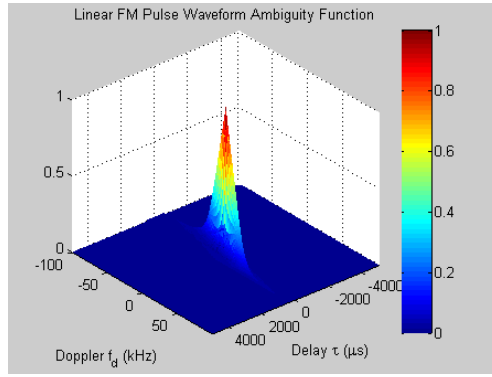
```
hlfm = phased.LinearFMWaveform('PulseWidth',100e-6,...
    'SweepBandwidth',2e5,'PRF',1e3);
```

Generate samples of the waveform.

```
x = step(hlfm);
```

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

```
[afmag_lfm,delay_lfm,doppler_lfm] = ambgfun(x,...
   hlfm.SampleRate,hlfm.PRF);
surf(delay_lfm*1e6,doppler_lfm/1e3,afmag_lfm,...
   'LineStyle','none');
axis tight; grid on; view([140,35]); colorbar;
xlabel('Delay \tau (\mus)');
ylabel('Doppler f_d (kHz)');
title('Linear FM Pulse Waveform Ambiguity Function');
```



The surface has a narrow ridge that is slightly tilted. The tilt indicates better resolution in the zero delay cut. For a more detailed analysis of waveforms using the ambiguity function, see Waveform Analysis Using the Ambiguity Function.

Comparing Autocorrelation for Rectangular and Linear FM Waveforms

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

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

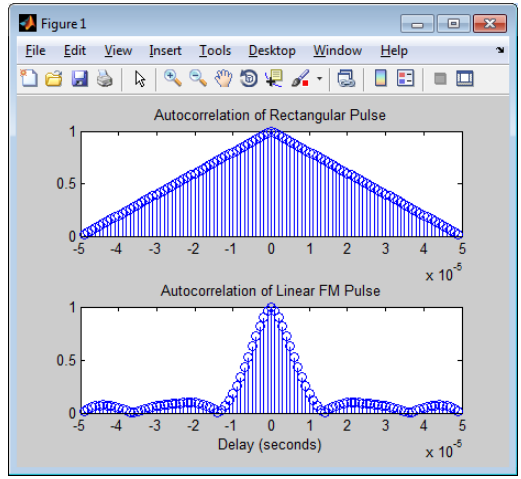
```
hrect = phased.RectangularWaveform('PRF',2e4);
hfm = phased.LinearFMWaveform('PRF',2e4);
xrect = step(hrect);
xfm = step(hfm);
```

Compute the ambiguity function magnitudes for each waveform.

```
[ambrect,delay] = ambgfun(xrect,hrect.SampleRate,hrect.PRF,...
    'Cut','Doppler');
ambfm = ambgfun(xfm,hfm.SampleRate,hfm.PRF,...
    'Cut','Doppler');
```

Plot the ambiguity function magnitudes.

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



Related Examples

- Waveform Analysis Using the Ambiguity Function

Stepped FM Pulse Waveforms

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

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

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

The stepped frequency pulse waveform has the following modifiable properties:

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

Enter the following to construct a stepped FM pulse waveform with a pulse duration (width) of 50 μs, 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, and 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 successively higher frequency pulses. Plot the pulses one by one in the same figure window. Pause the loop to visualize the increment in frequency with each successive call to `step`. Make an additional 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);
```

The next figure shows the plot in the final iteration of the loop.

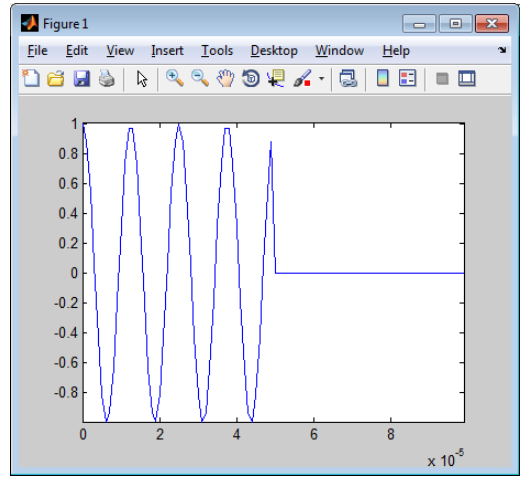


Figure 1
File Edit View Insert Tools Desktop Window Help
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
2
4
6
8
x 10^-5

Phase-Coded Waveforms

In this section...

When to Use Phase-Coded Waveforms

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

- When a rectangular pulse cannot provide both of these characteristics:
  - Short enough pulse for good range resolution
  - Enough energy in the signal to detect the reflected echo at the receiver
- When two or more radar systems are close to each other and you want to reduce interference among them.
- When digital processing suggests using a waveform with a discrete set of phases. For example, a Barker-coded waveform is a bi-phase waveform.

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

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

  Phase-coded waveforms tend to perform poorly when signals have Doppler shifts.
- When the hardware requirements for phase-coded waveforms are prohibitively expensive

How to Create Phase-Coded Waveforms

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

- Type of phase code

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

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

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

Basic Radar Using Phase-Coded Waveform

In the example in "End-to-End Radar System", you can use a phase-coded waveform in place of a rectangular waveform. To do so:

1. Replace the definition of `hwav` with the following definition.

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

2. Redefine the pulse width, `tau`, based on the properties of the new waveform.

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

For convenience, the complete code appears here. For a detailed explanation of the code, see the original example, "End-to-End Radar System".

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

hant = phased.IsotropicAntennaElement('FrequencyRange',...
    [1e9 10e9]);

htgt = phased.RadarTarget('Model','Nonfluctuating',...
```

```
    'MeanRCS',0.5,'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);

htxplat = phased.Platform('InitialPosition',[0;0;0],...
    'Velocity',[0;0;0],'OrientationAxes',[1 0 0;0 1 0;0 0 1]);
htgtplat = phased.Platform('InitialPosition',[7000; 5000; 0],...
    'Velocity',[-15;-10;0]);

[tgtrng,tgtang] = rangeangle(htgtplat.InitialPosition,...
    htxplat.InitialPosition);

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

maxrange = 1.5e4;
lambda = physconst('LightSpeed')/4e9;
tau = hwav.ChipWidth * hwav.NumChips;
Pt = radareqpow(lambda,maxrange,SNR,tau,'RCS',0.5,'Gain',20);

htx = phased.Transmitter('PeakPower',50e3,'Gain',20,...
    'LossFactor',0,'InUseOutputPort',true,...
    'CoherentOnTransmit',true);

hrad = phased.Radiator('Sensor',hant,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9);
hcol = phased.Collector('Sensor',hant,...
    'PropagationSpeed',physconst('LightSpeed'),...
    'Wavefront','Plane','OperatingFrequency',4e9);

hrec = phased.ReceiverPreamp('Gain',20,'NoiseFigure',2,...
    'ReferenceTemperature',290,'SampleRate',1e6,...
    'EnableInputPort',true,'SeedSource','Property','Seed',1e3);

hspace = phased.FreeSpace(...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',4e9,'TwoWayPropagation',false,...
    'SampleRate',1e6);
```

```
% Time step between pulses
T = 1/hwav.PRF;
% Get antenna position
txpos = htxplat.InitialPosition;
% Allocate array for received echoes
rxsig = zeros(hwav.SampleRate*T,numpulses);

for n = 1:numpulses
    % Update the target position
    tgtpos = step(htgtplat,T);
    % Get the range and angle to the target
    [tgtrng,tgtang] = rangeangle(tgtpos,txpos);
    % Generate the pulse
    sig = step(hwav);
    % Transmit the pulse. Output transmitter status
    [sig,txstatus] = step(htx,sig);
    % Radiate the pulse toward the target
    sig = step(hrad,sig,tgtang);
    % Propagate the pulse to the target in free space
    sig = step(hspace,sig,txpos,tgtpos);
    % Reflect the pulse off the target
    sig = step(htgt,sig);
    % Propagate the echo to the antenna in free space
    sig = step(hspace,sig,tgtpos,txpos);
    % Collect the echo from the incident angle at the antenna
    sig = step(hcol,sig,tgtang);
    % Receive the echo at the antenna when not transmitting
    rxsig(:,n) = step(hrec,sig,~txstatus);
end

rxsig = pulsint(rxsig,'noncoherent');
t = unigrid(0,1/hrec.SampleRate,T,'[)');
rangegates = (physconst('LightSpeed')*t)/2;
plot(rangegates,rxsig); hold on;
xlabel('Meters'); ylabel('Power');
ylim = get(gca,'YLim');
plot([tgtrng,tgtrng],[0 ylim(2)],'r');
```

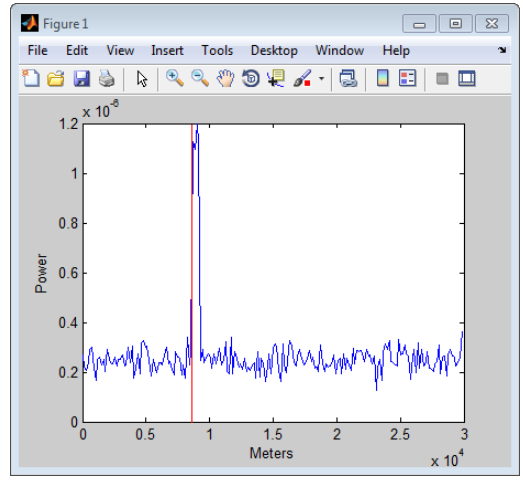


Figure 1
File Edit View Insert Tools Desktop Window Help
x 10^-6
1.2
1
0.8
0.6
0.4
0.2
0
Power
0 0.5 1 1.5 2 2.5 3
Meters
x 10^4

Waveforms with Staggered PRFs

In this section...

When to Use Staggered PRFs

Using a nonconstant PRF has important applications in radar. This approach is called *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, configure your waveform object with a vector instead of a scalar as the `PRF` property value.

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 µs. 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');
```

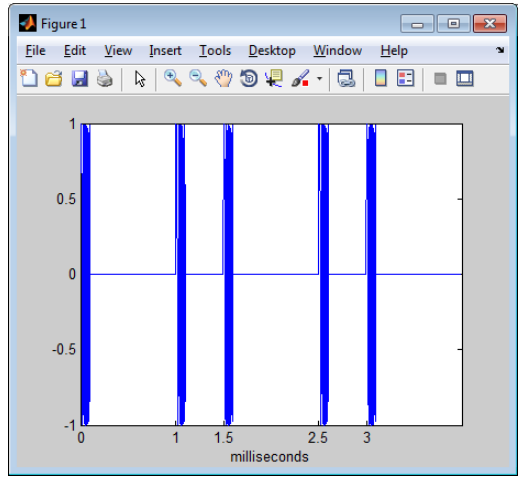


Figure 1
File Edit View Insert Tools Desktop Window Help
1
0.5
0
-0.5
-1
0
1
1.5
2.5
3
milliseconds

Transmitter

In this section...

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 transmitter object has the following modifiable properties:

- `PeakPower` — Peak transmit power in watts
- `Gain` — Transmit gain in decibels
- `LossFactor` — Loss factor in decibels
- `InUseOutputPort` — Track transmitter's status. Setting this property to `true` outputs a vector of 1s and 0s indicating when transmitter is on and

off. In a monostatic radar, the transmitter and receiver cannot operate simultaneously.

- `CoherentOnTransmit` — Preserve *coherence* among transmitter pulses. Setting this property to `true` (the default) models the operation of a fully coherent transmitter where the pulse-to-pulse phase is constant. Setting this property to `false` introduces random phase noise from pulse to pulse and models the operation of a non-coherent transmitter.
- `PhaseNoiseOutputPort` — Output the random pulse phases introduced by non-coherent operation of the transmitter. This property only applies if the `CoherentOnTransmit` property is `false`. By keeping a record of the random pulse phases, you can create a *pseudo-coherent*, or *coherent on receive* radar.

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)
```

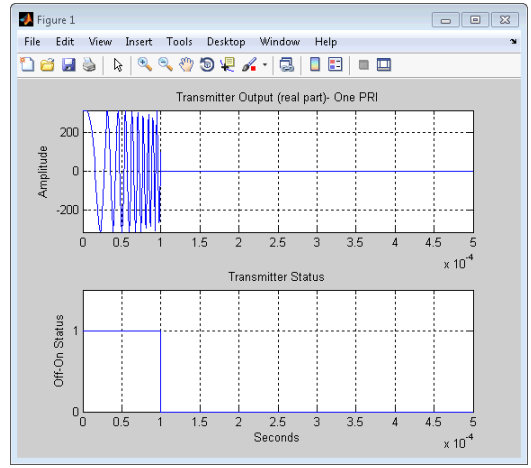
Construct a pulse waveform for transmission. In this example, use a 100-microsecond linear FM pulse with a bandwidth of 200 kHz. Use the default sweep direction and sample rate. Set the PRF to 2 kHz.

```
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

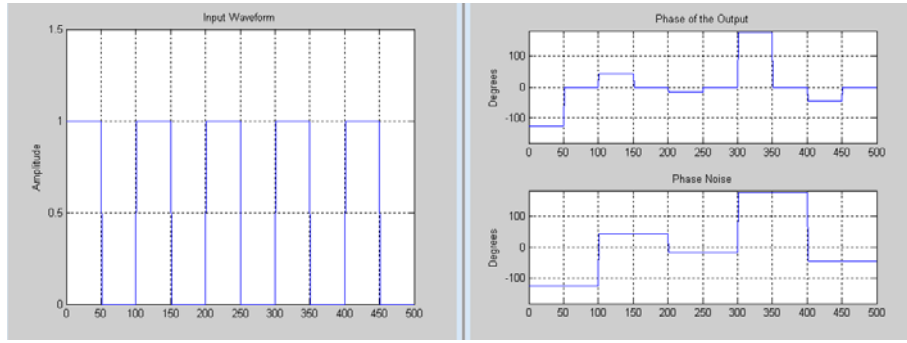
To model a coherent on receive radar, you can set the `CoherentOnTransmit` property to `false` and the `PhaseNoiseOutputPort` property to `true`. You can output the random phase added to each sample with `step`.

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 W, 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 4-23.
- `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 dimensionless quantity that 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 μs. 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');
```

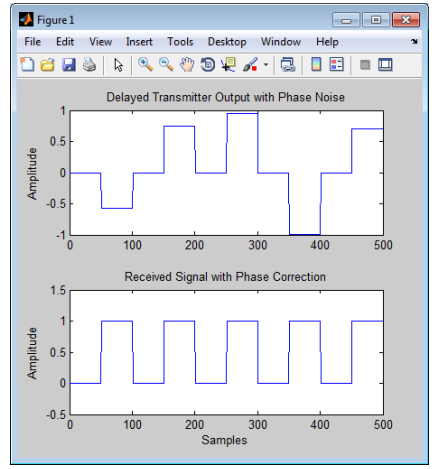


Figure 1
File Edit View Insert Tools Desktop Window Help
Delayed Transmitter Output with Phase Noise
Amplitude
1
0.5
0
-0.5
-1
0
100
200
300
400
500
Received Signal with Phase Correction
Amplitude
1.5
1
0.5
0
-0.5
0
100
200
300
400
500
Samples

Radar Equation

In this section...

Radar Equation Theory

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. 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.

Link Budget Calculation Using the Radar Equation

This example shows how to compute the required peak transmit power using the radar equation. You implement a noncoherent detector with a monostatic radar operating at 5 GHz. Based on the noncoherent integration of 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 m^2 at 30 km. The transmitter gain is 30 dB. Determine the required SNR at the receiver and use the radar equation to calculate the required peak transmit power.

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

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

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 kW.

Maximum Detectable Range for a Monostatic Radar

Assume that the minimum detectable SNR at the receiver of a monostatic radar operating at 1 GHz is 13 dB. Use the radar equation to determine the maximum detectable range for a target with a nonfluctuating RCS of 0.5 m^2 if the radar has a peak transmit power of 1 MW. Assume the transmitter gain is 40 dB and the radar transmits a pulse that is 0.5 μs 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.

Output SNR at the Receiver in a Bistatic Radar

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

```
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.

5

Beamforming

Conventional Beamforming

In this section...

Uses for Beamformers

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

Support for Conventional Beamforming

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

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

For wideband beamformers, see "Wideband Beamforming" on page 5-11.

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 one-half 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.

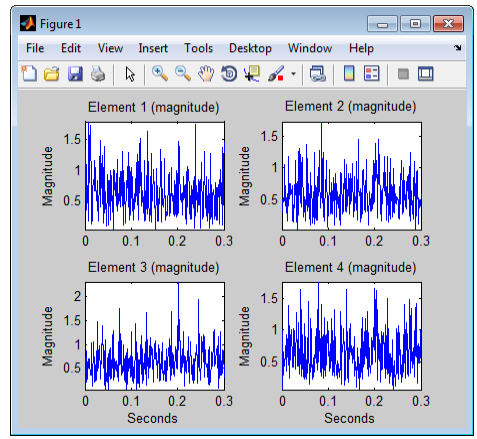
```
angle_of_arrival = [30;0];
x = collectPlaneWave(hula,testsig,angle_of_arrival,fc);
```

`x` is a matrix with ten columns. Each column represents 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))+1i*randn(size(x)));
subplot(221)
plot(t,abs(x(:,1))); title('Element 1 (magnitude)');
axis tight;  ylabel('Magnitude');
subplot(222)
plot(t,abs(x(:,2))); title('Element 2 (magnitude)');
axis tight; ylabel('Magnitude');
subplot(223)
plot(t,abs(x(:,3))); title('Element 3 (magnitude)');
axis tight; xlabel('Seconds'); ylabel('Magnitude');
subplot(224)
```

```
plot(t,abs(x(:,4))); title('Element 4 (magnitude)');
axis tight; xlabel('Seconds'); ylabel('Magnitude');
```



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

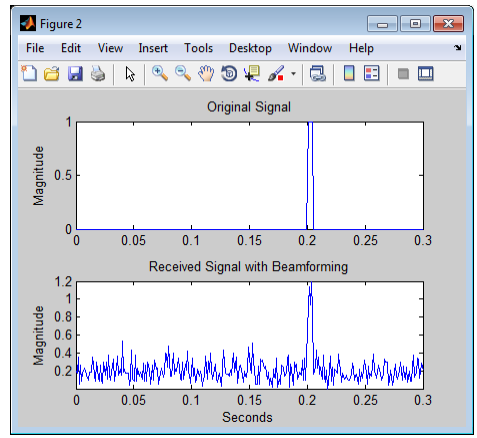
```
hbf = phased.PhaseShiftBeamformer('SensorArray',hula,...
    'OperatingFrequency',1e9,'Direction',angle_of_arrival,...
    'WeightsOutputPort',true);
```

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 then summed.

```
[y,w] = step(hbf,x);
```

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

```
figure;
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 both with—and without—the beamforming weights.

```
azang = -180:30:180;
figure;
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');
```

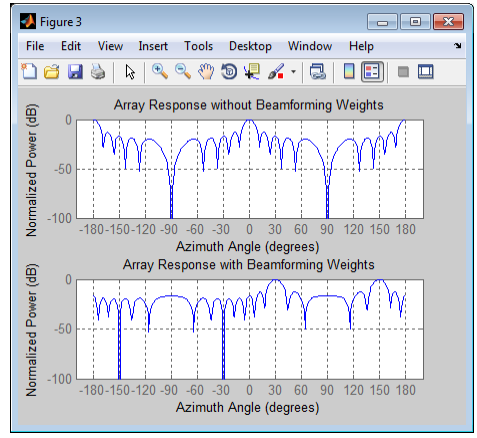


Figure 3
File Edit View Insert Tools Desktop Window Help
Array Response without Beamforming Weights
Normalized Power (dB)
0
-50
-100
-180 -150 -120 -90 -60 -30 0 30 60 90 120 150 180
Azimuth Angle (degrees)
Array Response with Beamforming Weights
Normalized Power (dB)
0
-50
-100
-180 -150 -120 -90 -60 -30 0 30 60 90 120 150 180
Azimuth Angle (degrees)

Adaptive Beamforming

In this section...

Benefits of Adaptive Beamforming

"Narrowband Phase Shift Beamformer with a ULA" on page 5-2 uses weights chosen independent of any data received by the array. The weights in the narrowband phase shift beamformer steer the array response in a specified direction. However, they do not account for any interference scenarios. As a result, these conventional beamformers are susceptible to interference signals. Such interference signals can be a particular problem if they occur at sidelobes of the array response.

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

Support for Adaptive Beamforming

Phased Array System Toolbox software provides these adaptive beamformers:

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

LCMV Beamformer

This example uses code from the "Narrowband Phase Shift Beamformer with a ULA" on page 5-2 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 W. The interference signal from the barrage jammer is incident on the ULA at an angle of 120 degrees azimuth and 0 degrees elevation.

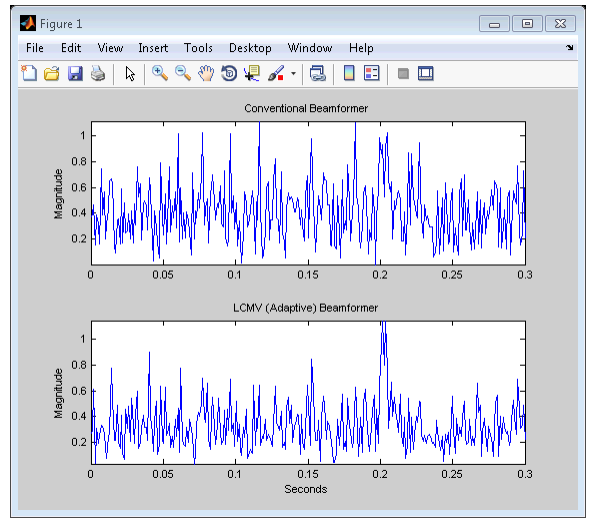
```
hjammer = phased.BarrageJammer('ERP',10,'SamplesPerFrame',300);
jamsig = step(hjammer);
jammer_angle = [120;0];
jamsig = collectPlaneWave(hula,jamsig,jammer_angle,fc);
```

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.

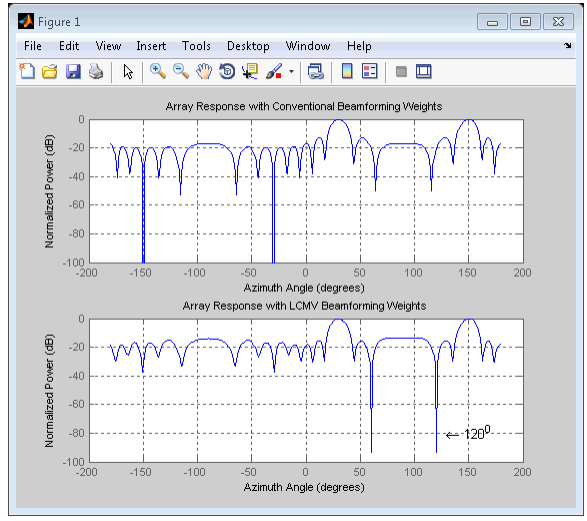
```
hstv = phased.SteeringVector('SensorArray',hula,...
       'PropagationSpeed',physconst('LightSpeed'));
hLCMV = phased.LCMVBeamformer('DesiredResponse',1,...
        'TrainingInputPort',true,'WeightsOutputPort',true);
hLCMV.Constraint = step(hstv,fc,angle_of_arrival);
hLCMV.DesiredResponse = 1;
[yLCMV,wLCMV]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

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

```
figure;
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 product 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

In this section...

Support for Wideband Beamforming

Beamforming achieved by multiplying the sensor input by a complex exponential with the appropriate phase shift only applies for narrowband signals. In the case of wideband, or *broadband*, signals, the steering vector is not a function of a single frequency.

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

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

Time-Delay Beamforming

This example shows how to perform wideband conventional time-delay beamforming with a microphone array.

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

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

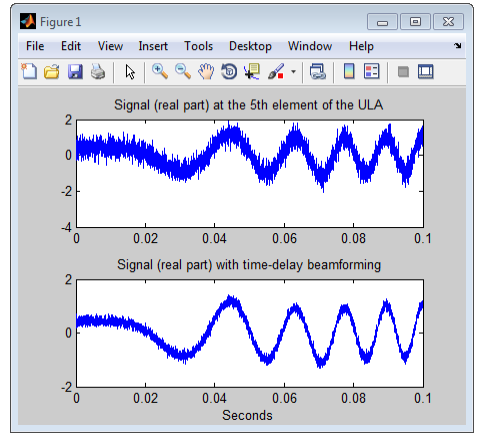
Collect the acoustic chirp with a ten-element ULA. Use omnidirectional microphone elements spaced less than one-half the wavelength of the 50 kHz

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 5th 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.

Visualization of Wideband Beamformer Performance

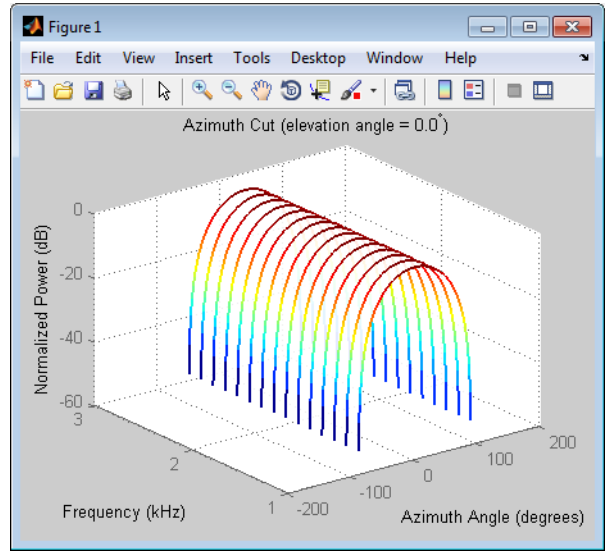
This example shows how to plot the response of an antenna element and an array, to help validate the performance of a beamformer. The array must maintain an acceptable array pattern throughout the bandwidth.

Create a uniform linear array of cosine antenna elements.

```
c = 340;
freq = [1e3 2.75e3];
fc = 2e3;
numels = 11;
h = phased.CosineAntennaElement('FrequencyRange',freq);
ha = phased.ULA('NumElements',numels,...
   'ElementSpacing',0.5*c/fc,'Element',h);
```

Plot the response pattern of the antenna element over a series of frequencies.

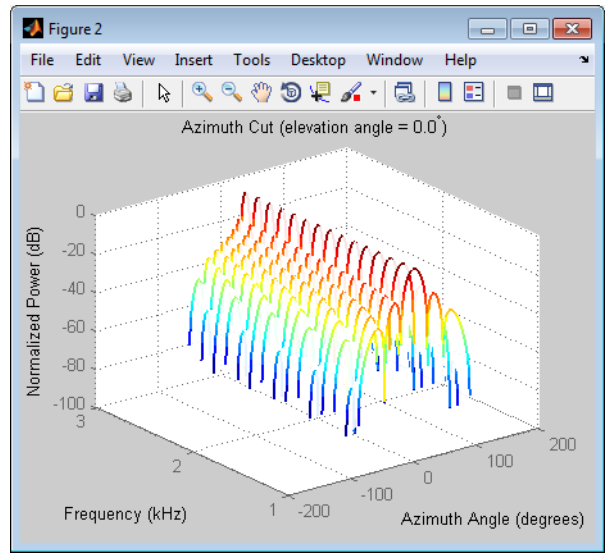
```
plotFreq = linspace(min(freq),max(freq),15);
figure;
plotResponse(h,plotFreq,'OverlayFreq',false);
```



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

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

```
figure;
plotResponse(ha,plotFreq,c,'OverlayFreq',false);
```

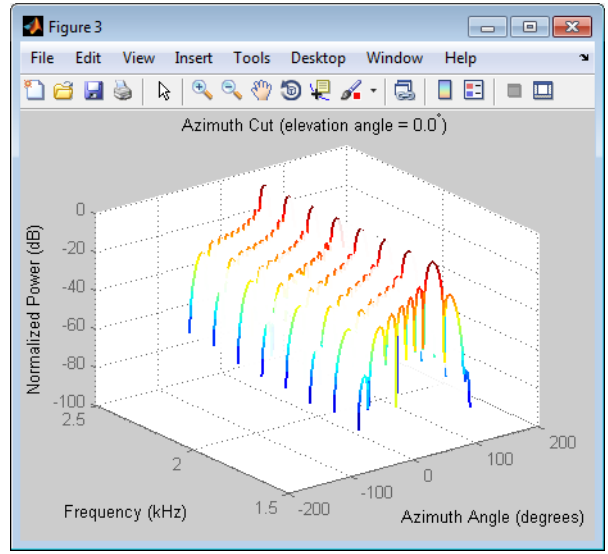


Apply a subband phase shift beamformer to the array. The direction of interest is 30 degrees azimuth and 0 degrees elevation.

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

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

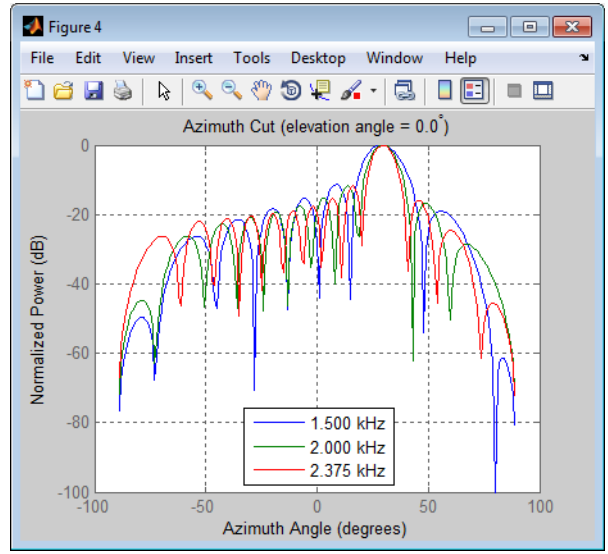
```
figure;
plotResponse(ha,centerFreq',c,'Weights',w,'OverlayFreq',false);
```



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

Plot the response pattern at selected frequencies using a two-dimensional format.

```
centerFreq = fftshift(centerFreq);
w = fftshift(w,2);
idx = [1 5 8];
figure;
plotResponse(ha,centerFreq(idx)',c,'Weights',w(:,idx));
legend('Location','South')
```



This plot shows that the main beam direction remains constant, but the beam width decreases with frequency.

6

Direction-of-Arrival (DOA) Estimation

Beamscan Direction-of-Arrival Estimation

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

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

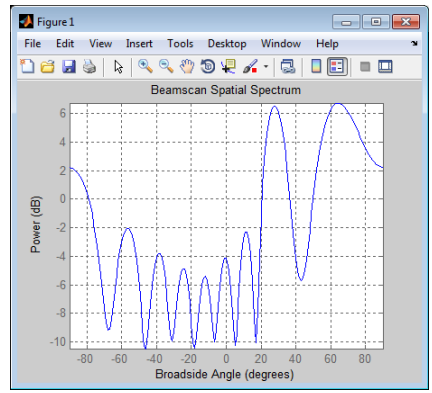
```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
hula = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
hula.Element.FrequencyRange = [8e8 1.2e9];
```

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))+1i*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);
```



Related Examples

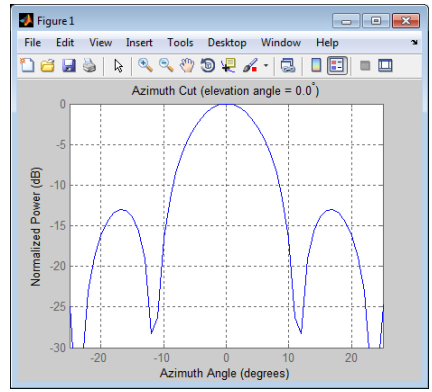
- Direction of Arrival Estimation with Beamscan and MVDR

Super-resolution DOA Estimation

The beamscan DOA estimator illustrated in "Beamscan Direction-of-Arrival Estimation" on page 6-2 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.

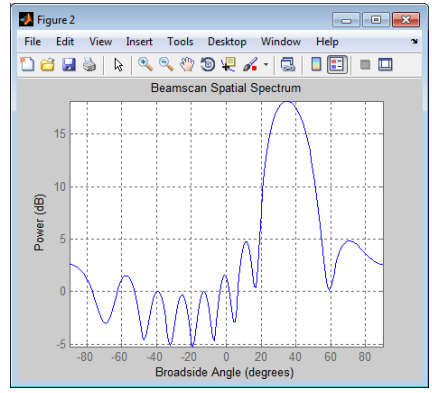
```
fc = 1e9;
lambda = physconst('LightSpeed')/fc;
hula = phased.ULA('NumElements',10,'ElementSpacing',lambda/2);
hula.Element.FrequencyRange = [8e8 1.2e9];
plotResponse(hula,fc,physconst('LightSpeed'));
axis([-25 25 -30 0]);
```



Assume that you have two signal sources with DOAs separated by ten degrees. Because both DOAs fall inside the main lobe of the array response, 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))+1i*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);
figure;
plotSpectrum(hbeam);
```



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

The Phased Array System Toolbox product provides super-resolution 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))+1i*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 =

   39.9697   30.0060
```

Related Examples

- High Resolution Direction of Arrival Estimation

7

Space-Time Adaptive Processing (STAP)

Angle-Doppler Response

In this section...

Benefits of Visualizing Angle-Doppler Response

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

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

Angle-Doppler Response of 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 one-half 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);
```

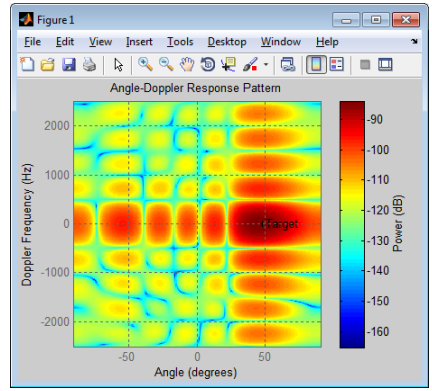
Propagate ten rectangular pulses to and from the target, and collect the responses at the array.

```
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
    % get angle to target
    [~,tgtang] = rangeangle(tgtloc,txloc);
    % transmit pulse
```

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

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

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

The scenario in this example is identical to that of "Angle-Doppler Response of a Stationary Target at a Stationary Array" on page 7-2, except that the ULA is moving at a constant velocity. For convenience, the MATLAB® code to set up the objects is repeated. Notice that the `InitialPosition` and `Velocity` properties of the `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 "Applicability of DPCA Pulse Canceller" on page 7-9.

```
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. Then, 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
    % move transmitter
    [txloc,~] = step(htxplat,1/PRF);
    % get angle to target
    [~,tgtang] = rangeangle(tgtloc,txloc);
    % transmit pulse
    [txsig,txstatus] = step(htx,wav);
    % radiate pulse
```

```
    txsig = step(hrad,txsig,tgtang);
    % propagate pulse to target
    txsig = step(hspace,txsig,txloc,tgtloc);
    % reflect pulse off stationary target
    txsig = step(htgt,txsig);
    % propagate pulse to array
    txsig = step(hspace,txsig,tgtloc,txloc);
    % collect pulse
    rxsig(:,:,n) = step(hcol,txsig,tgtang);
    % receive pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n),~txstatus);
end
```

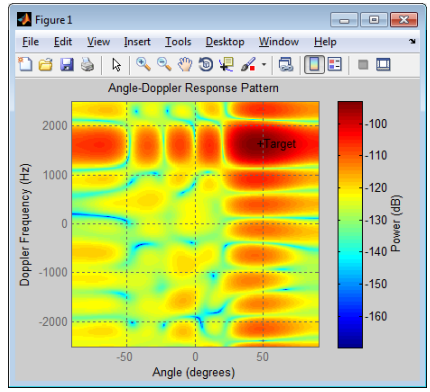
Calculate the target angles and range with respect to the ULA. Then, calculate the Doppler shift induced by the motion of the phased array.

```
sp = radialspeed(tgtloc, 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

In this section...

When to Use the DPCA Pulse Canceller

In a *moving target indication* (MTI) radar, clutter returns can make it more difficult to detect and track the targets of interest. A rudimentary way to mitigate the effects of clutter returns in such a system is to implement a displaced phase center antenna (DPCA) pulse canceller on the slow-time data.

You can implement a DPCA pulse canceller with `phased.DPCACanceller`. This implementation assumes that the entire array is used on transmit. On receive, the array is divided into two subarrays. The phase centers of the subarrays are separated by twice the distance the platform moves in one pulse repetition interval.

Applicability of DPCA Pulse Canceller

The DPCA pulse canceller is applicable when both these conditions are true:

- Clutter is stationary across pulses.
- The motion satisfies

$$vT = d / 2 \qquad \textbf{(7-1)}$$

  where:

  - v indicates the speed of the platform
  - T represents the pulse repetition interval
  - d indicates the inter-element spacing of the array

Example: DPCA Pulse Canceller for Clutter Rejection

This example implements a DPCA pulse canceller for clutter rejection. Assume you have an airborne radar platform modeled by a six-element ULA

operating at 4 GHz. The array elements are spaced at one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses two microseconds in duration with a PRF of 5 kHz. The platform moves along the array axis with a speed equal to one-half the product of the element spacing and the PRF. As a result, the condition in Equation 7-1 applies. The target has a nonfluctuating RCS of 1 square meter and moves 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.

```
PRF = 5e3;
fc = 4e9; fs = 1e6;
c = physconst('LightSpeed');
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth',2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (hula.ElementSpacing * PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
hclutter = phased.ConstantGammaClutter('Sensor',hula,...
    'PropagationSpeed',hrad.PropagationSpeed,...
    'OperatingFrequency',hrad.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',hrad.OperatingFrequency),...
    'EarthModel','Flat',...
    'BroadsideDepressionAngle',0,...
    'MaximumRange',hrad.PropagationSpeed/(2*PRF),...
    'PlatformHeight',htxplat.InitialPosition(3),...
    'PlatformSpeed',norm(htxplat.Velocity),...
    'PlatformDirection',[90;0]);
```

```
htgt = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hspace = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
hrx = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'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. Also, compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain.

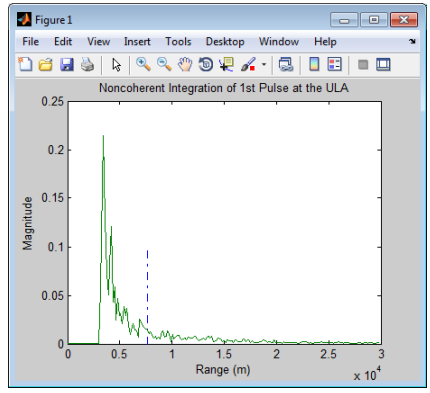
```
NumPulses = 10;
wav = step(hwav);
M = fs/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
csig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c * fasttime)/2;
hclutter.SeedSource = 'Property';
hclutter.Seed = 5;

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
    [txsig1,txstatus] = step(htx,wav);  % transmit pulse
    csig(:,:,n) = step(hclutter,txsig1(abs(txsig1)>0)); % collect clutter
    txsig = step(hrad,txsig1,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate to target
    txsig = step(htgt,txsig);  % reflect off target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang); % collect pulse
    rxsig(:,:,n) = step(hrx,rxsig(:,:,n) + csig(:,:,n),...
  ~txstatus); % receive pulse plus clutter return
end
```

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,c/(2 * fs));
```

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

```
firstpulse = pulsint(rxsig(:,:,1),'noncoherent');
figure;
plot([tgtrng tgtrng],[0 0.1],'-.',rangebins,firstpulse);
title('Noncoherent Integration of 1st Pulse at the ULA');
xlabel('Range (m)'); ylabel('Magnitude');
```



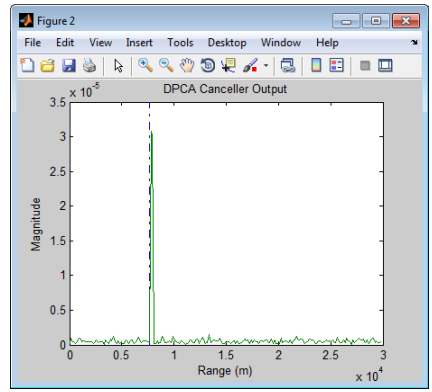
The large-magnitude clutter returns obscure the presence of the target. Apply the DPCA pulse canceller to reject the clutter.

```
hstap = phased.DPCACanceller('SensorArray',hula,'PRF',PRF,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
    'Direction',[0;0],'Doppler',tgtdoppler,...
```

```
    'WeightsOutputPort',true);
[y,w] = step(hstap,rxsig,tgtcell);
```

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

```
figure;
plot([tgtrng,tgtrng],[0 3.5e-5],'-.',rangebins,abs(y));
title('DPCA Canceller Output');
xlabel('Range (m)'), ylabel('Magnitude');
```



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

Adaptive Displaced Phase Center Antenna (ADPCA) Pulse Canceller

In this section...

When to Use the Adaptive DPCA Pulse Canceller

Consider an airborne radar system that needs to suppress clutter returns and possibly jammer interference. Under any of the following conditions, you might choose an adaptive DPCA (ADPCA) pulse canceller for suppressing these effects.

- Jamming and other interference effects are substantial. The DPCA pulse canceller is susceptible to interference because the DPCA pulse canceller does not use the received data.
- The sample matrix inversion (SMI) algorithm is inapplicable because of computational expense or a rapidly changing environment.

The `phased.ADPCAPulseCanceller` object implements an ADPCA pulse canceller. This pulse canceller uses the data received from two consecutive pulses to estimate the space-time interference covariance matrix. In particular, the object lets you specify:

- The number of training cells. The algorithm uses training cells to estimate the interference. In general, a larger number of training cells leads to a better estimate of interference.
- The number of guard cells close to the target cells. The algorithm recognizes guard cells to prevent target returns from contaminating the estimate of the interference.

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 for Clutter Rejection" on page 7-9 except that a stationary

broadband barrage jammer is added at `[3.5e3; 1e3; 0]`. The jammer has an effective radiated power of 1 kw.

To repeat the scenario for convenience, the airborne radar platform is a six-element ULA operating at 4 GHz. The array elements are spaced at one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses two μs in duration with a PRF of 5 kHz. The platform is moving along the array axis with a speed equal to one-half the product of the element spacing and the PRF. As a result, the condition in Equation 7-1 applies. The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of `[15;15;0]`.

The following commands construct the required System objects to simulate the scenario.

```
PRF = 5e3;
fc = 4e9; fs = 1e6;
c = physconst('LightSpeed');
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (hula.ElementSpacing * PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
hclutter = phased.ConstantGammaClutter('Sensor',hula,...
    'PropagationSpeed',hrad.PropagationSpeed,...
    'OperatingFrequency',hrad.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',hrad.OperatingFrequency),...
```

```
    'EarthModel','Flat',...
    'BroadsideDepressionAngle',0,...
    'MaximumRange',hrad.PropagationSpeed/(2*PRF),...
    'PlatformHeight',htxplat.InitialPosition(3),...
    'PlatformSpeed',norm(htxplat.Velocity),...
    'PlatformDirection',[90;0]);
htgt = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hjammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200);
hjammerplat = phased.Platform(...
    'InitialPosition',[3.5e3; 1e3; 0],'Velocity',[0;0;0]);
hspace = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
hrx = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'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. Compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain. Also, propagate the jamming signal from the jammer location to the airborne ULA.

```
NumPulses = 10;
wav = step(hwav);
M = fs/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
csig = zeros(M,N,NumPulses);
jsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c * fasttime)/2;
hclutter.SeedSource = 'Property';
hclutter.Seed = 40543;
hjammer.SeedSource = 'Property';
hjammer.Seed = 96703;
hrx.SeedSource = 'Property';
hrx.Seed = 56113;
```

```
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
    csig(:,:,n) = step(hclutter,txsig(abs(txsig)>0)); % collect clutter

    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig); % reflect off target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang); % collect pulse

    jamsig = step(hjammer); % generate jammer signal
    [~,jamang] = rangeangle(jamloc,txloc); % angle from jammer to transmitter
    jamsig = step(hspace,jamsig,jamloc,txloc); % propagate jammer signal
    jsig(:,:,n) = step(hcol,jamsig,jamang); % collect jammer signal

    rxsig(:,:,n) = step(hrx,...
        rxsig(:,:,n) + csig(:,:,n) + jsig(:,:,n),...
        ~txstatus); % receive pulse plus clutter return plus jammer signal
end
```

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

```
sp = radialspeed(tgtloc, 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,c/(2 * fs));
```

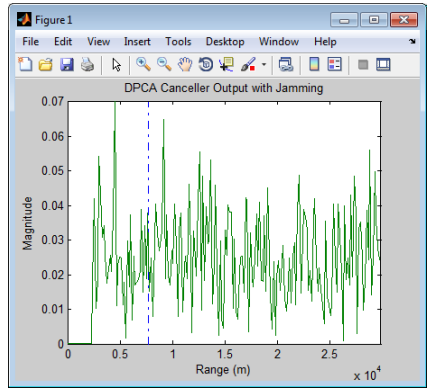
Process the array responses using the nonadaptive DPCA pulse canceller. To do so, construct the DPCA object, and apply it to the received signals.

```
hstap = phased.DPCACanceller('SensorArray',hula,'PRF',PRF,...
    'PropagationSpeed',c,...
```

```
    'OperatingFrequency',fc,...
    'Direction',[0;0],'Doppler',tgtdoppler,...
    'WeightsOutputPort',true);
[y,w] = step(hstap,rxsig,tgtcell);
```

Plot the DPCA result with the target range marked by a vertical dashed line. Notice how the presence of the interference signal has obscured the target.

```
figure;
plot([tgtrng,tgtrng],[0 7e-2],'-.',rangebins,abs(y));
axis tight;
xlabel('Range (m)'), ylabel('Magnitude');
title('DPCA Canceller Output with Jamming')
```

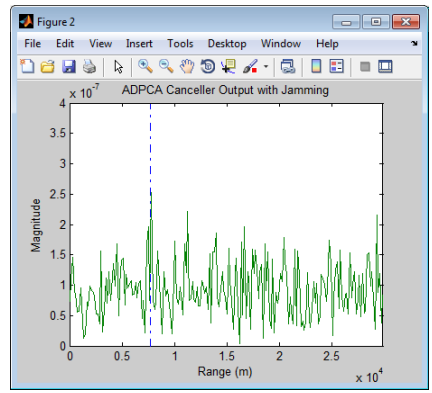


Apply the adaptive DPCA pulse canceller. Use 100 training cells and 4 guard cells, two on each side of the target range gate.

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

Plot the result with the target range marked by a vertical dashed line. Notice how the adaptive DPCA pulse canceller enables you to detect the target in the presence of the jamming signal.

```
figure;
plot([tgtrng,tgtrng],[0 4e-7],'-.',rangebins,abs(y_adpca));
axis tight;
title('ADPCA Canceller Output with Jamming');
xlabel('Range (m)'), ylabel('Magnitude');
```



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

```
hadresp = phased.AngleDopplerResponse('SensorArray',hula,...
    'OperatingFrequency',fc,...
    'PropagationSpeed',c,...
    'PRF',PRF,'ElevationAngle',tgtelang);
figure;
plotResponse(hadresp,w_adpca);
title('Angle-Doppler Response with ADPCA Pulse Cancellation');
text(az2broadside(jamang(1),jamang(2)) + 10,...
    0,'\leftarrow Interference Null')
```

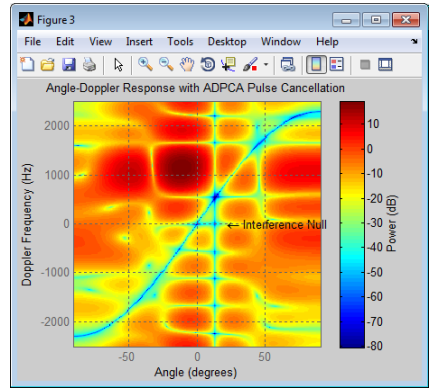


Figure 3
File Edit View Insert Tools Desktop Window Help
Angle-Doppler Response with ADPCA Pulse Cancellation
Doppler Frequency (Hz)
2000
1000
0
-1000
-2000
← Interference Null
-50
0
50
Angle (degrees)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
Power (dB)

Sample Matrix Inversion (SMI) Beamformer

In this section...

When to Use the SMI Beamformer

In situations where an airborne radar system needs to suppress clutter returns and jammer interference, the system needs a more sophisticated algorithm than a DPCA pulse canceller can provide. One option is the sample matrix inversion (SMI) algorithm. SMI is the optimum STAP algorithm and is often used as a baseline for comparison with other algorithms.

The SMI algorithm is computationally expensive and assumes a stationary environment across many pulses. If you need to suppress clutter returns and jammer interference with less computation, or in a rapidly changing environment, consider using an ADPCA pulse canceller instead.

The `phased.STAPSMIBeamformer` object implements the SMI algorithm. In particular, the object lets you specify:

- The number of training cells. The algorithm uses training cells to estimate the interference. In general, a larger number of training cells leads to a better estimate of interference.
- The number of guard cells close to the target cells. The algorithm recognizes guard cells to prevent target returns from contaminating the estimate of the interference.

Example: Sample Matrix Inversion (SMI) Beamformer

This scenario is identical to the one presented in "Example: Adaptive DPCA Pulse Canceller" on page 7-14. 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.

To repeat the scenario for convenience, the airborne radar platform is a six-element ULA operating at 4 GHz. The array elements are spaced at

one-half the wavelength of the 4 GHz carrier frequency. The radar emits ten rectangular pulses two μs in duration with a PRF of 5 kHz. The platform is moving along the array axis with a speed equal to one-half the product of the element spacing and the PRF. The target has a nonfluctuating RCS of 1 square meter and is moving with a constant velocity vector of `[15;15;0]`. A stationary broadband barrage jammer is located at `[3.5e3; 1e3; 0]`. The jammer has an effective radiated power of 1 kw.

The following commands construct the required System objects to simulate the scenario.

```
PRF = 5e3;
fc = 4e9; fs = 1e6;
c = physconst('LightSpeed');
hant = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e8 5e9],'BackBaffled',true);
lambda = c/fc;
hula = phased.ULA(6,'Element',hant,'ElementSpacing',lambda/2);
hwav = phased.RectangularWaveform('PulseWidth', 2e-6,...
    'PRF',PRF,'SampleRate',fs,'NumPulses',1);
hrad = phased.Radiator('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
hcol = phased.Collector('Sensor',hula,...
    'PropagationSpeed',c,...
    'OperatingFrequency',fc);
vy = (hula.ElementSpacing * PRF)/2;
htxplat = phased.Platform('InitialPosition',[0;0;3e3],...
    'Velocity',[0;vy;0]);
hclutter = phased.ConstantGammaClutter('Sensor',hula,...
    'PropagationSpeed',hrad.PropagationSpeed,...
    'OperatingFrequency',hrad.OperatingFrequency,...
    'SampleRate',fs,...
    'TransmitSignalInputPort',true,...
    'PRF',PRF,...
    'Gamma',surfacegamma('woods',hrad.OperatingFrequency),...
    'EarthModel','Flat',...
    'BroadsideDepressionAngle',0,...
    'MaximumRange',hrad.PropagationSpeed/(2*PRF),...
    'PlatformHeight',htxplat.InitialPosition(3),...
```

```
    'PlatformSpeed',norm(htxplat.Velocity),...
    'PlatformDirection',[90;0]);
htgt = phased.RadarTarget('MeanRCS',1,...
    'Model','Nonfluctuating','OperatingFrequency',fc);
htgtplat = phased.Platform('InitialPosition',[5e3; 5e3; 0],...
    'Velocity',[15;15;0]);
hjammer = phased.BarrageJammer('ERP',1e3,'SamplesPerFrame',200);
hjammerplat = phased.Platform(...
    'InitialPosition',[3.5e3; 1e3; 0],'Velocity',[0;0;0]);
hspace = phased.FreeSpace('OperatingFrequency',fc,...
    'TwoWayPropagation',false,'SampleRate',fs);
hrx = phased.ReceiverPreamp('NoiseFigure',0,...
    'EnableInputPort',true,'SampleRate',fs,'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. Compute clutter echoes using the constant gamma model with a gamma value corresponding to wooded terrain. Also, propagate the jamming signal from the jammer location to the airborne ULA.

```
NumPulses = 10;
wav = step(hwav);
M = fs/PRF;
N = hula.NumElements;
rxsig = zeros(M,N,NumPulses);
csig = zeros(M,N,NumPulses);
jsig = zeros(M,N,NumPulses);
fasttime = unigrid(0,1/fs,1/PRF,'[)');
rangebins = (c * fasttime)/2;
hclutter.SeedSource = 'Property';
hclutter.Seed = 40543;
hjammer.SeedSource = 'Property';
hjammer.Seed = 96703;
hrx.SeedSource = 'Property';
hrx.Seed = 56113;
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
    csig(:,:,n) = step(hclutter,txsig(abs(txsig)>0)); % collect clutter

    txsig = step(hrad,txsig,tgtang); % radiate pulse
    txsig = step(hspace,txsig,txloc,tgtloc); % propagate pulse to target
    txsig = step(htgt,txsig); % reflect off target
    txsig = step(hspace,txsig,tgtloc,txloc); % propagate to array
    rxsig(:,:,n) = step(hcol,txsig,tgtang); % collect pulse

    jamsig = step(hjammer); % generate jammer signal
    [~,jamang] = rangeangle(jamloc,txloc); % angle from jammer to transmitter
    jamsig = step(hspace,jamsig,jamloc,txloc); % propagate jammer signal
    jsig(:,:,n) = step(hcol,jamsig,jamang); % collect jammer signal

    rxsig(:,:,n) = step(hrx,...
        rxsig(:,:,n) + csig(:,:,n) + jsig(:,:,n),...
        ~txstatus); % receive pulse plus clutter return plus jammer signal
end
```

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

```
sp = radialspeed(tgtloc, 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,c/(2 * fs));
```

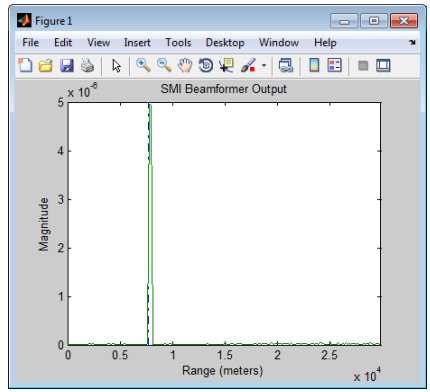
Construct an SMI beamformer object. Use 100 training cells, 50 on each side of the target range gate. Use four guard cells, two range gates in front of the target cell and two range gates beyond the target cell. Obtain the beamformer response and weights.

```
tgtang = [tgtazang; tgtelang];
hstap = phased.STAPSMIBeamformer('SensorArray',hula,...
    'PRF',PRF,'PropagationSpeed',c,...
    'OperatingFrequency',fc,...
```

```
    'Direction',tgtang,'Doppler',tgtdoppler,...
    'WeightsOutputPort',true,...
    'NumGuardCells',4,'NumTrainingCells',100);
[y,weights] = step(hstap,rxsig,tgtcell);
```

Plot the resulting array output after beamforming.

```
figure;
plot([tgtrng,tgtrng],[0 5e-6],'-.',rangebins,abs(y));
axis tight;
title('SMI Beamformer Output');
xlabel('Range (meters)'); ylabel('Magnitude');
```



Plot the angle-Doppler response with the beamforming weights.

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

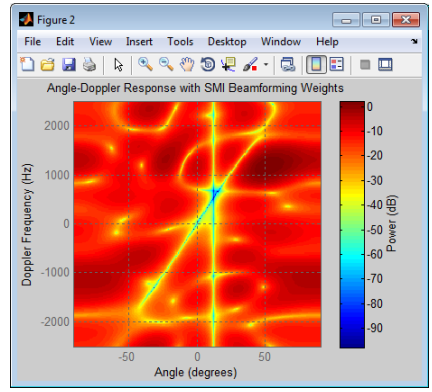


Figure 2
File Edit View Insert Tools Desktop Window Help
Angle-Doppler Response with SMI Beamforming Weights
Doppler Frequency (Hz)
2000
1000
0
-1000
-2000
-50
0
50
Angle (degrees)
Power (dB)
0
-10
-20
-30
-40
-50
-60
-70
-80
-90

8

Detection

Neyman-Pearson Hypothesis Testing

In this section...

Purpose of Hypothesis Testing

In phased-array applications, you sometimes need to decide between two competing hypotheses to determine the reality underlying the data the array receives. For example, suppose one hypothesis, called the *null hypothesis*, states that the observed data consists of noise only. Suppose another hypothesis, called the *alternative hypothesis*, states that the observed data consists of a deterministic signal plus noise. To decide, you must formulate a decision rule that uses specified criteria to choose between the two hypotheses.

Support for Neyman-Pearson Hypothesis Testing

When you use Phased Array System Toolbox software for applications such as radar and sonar, you typically use the Neyman-Pearson (NP) optimality criterion to formulate your hypothesis test.

When you choose the NP criterion, you can use `npwgnthresh` to determine the threshold for the detection of deterministic signals in white Gaussian noise. The optimal decision rule derives from a *likelihood ratio test* (LRT). An LRT chooses between the null and alternative hypotheses based on a ratio of conditional probabilities.

`npwgnthresh` enables you to specify the maximum false-alarm probability as a constraint. A *false alarm* means determining that the data consists of a signal plus noise, when only noise is present.

For details about the statistical assumptions the `npwgnthresh` function makes, see the reference page for that function.

Threshold for Real-Valued Signal in White Gaussian Noise

This example shows how to verify the probability of false alarm empirically for a real-valued signal in white Gaussian noise.

Determine the required signal-to-noise (SNR) in decibels for the NP detector when the maximum tolerable false-alarm probability is 10^{-3}.

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

Determine the actual threshold corresponding to the desired false-alarm probability, assuming the variance is 1.

```
variance = 1;
threshold = sqrt(variance * db2pow(T));
```

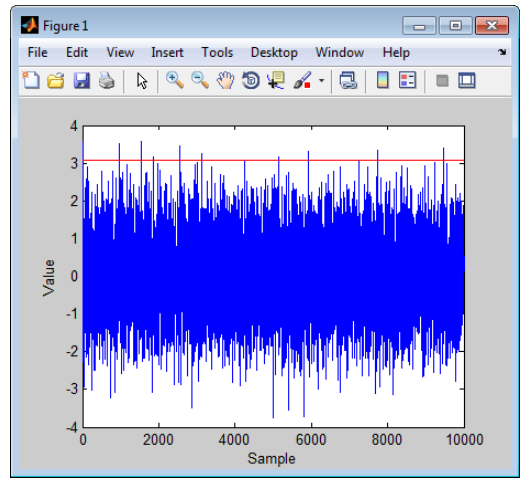
Verify empirically that the threshold results in the desired false-alarm probability under the null hypothesis. To do so, generate 1 million samples of a Gaussian random variable, and determine the proportion of samples that exceed the threshold.

```
rng default
N = 1e6;
x = sqrt(variance) * randn(N,1);
falsealarmrate = sum(x > threshold)/N
```

```
falsealarmrate =

   9.9500e-04
```

Plot the first 10,000 samples and a line showing the threshold.

```
x1 = x(1:1e4);
plot(x1)
line([1 length(x1)],[threshold threshold],'Color','red');
xlabel('Sample'); ylabel('Value');
```



The red horizontal line in the plot indicates the threshold. You can see that few sample values exceed the threshold. This result is expected because of the small false-alarm probability.

Threshold for Two Pulses of Real-Valued Signal in White Gaussian Noise

This example shows how to verify the probability of false alarm empirically in a system that uses pulse integration with two real-valued pulses. In this scenario, each sample is the sum of two samples, one from each pulse.

Determine the required SNR for the NP detector when the maximum tolerable false-alarm probability is 10^{-3}.

```
Pfa = 1e-3;
T = npwgnthresh(Pfa,2,'real');
```

Generate two sets of 10,000 samples of a Gaussian random variable.

```
rng default
variance = 1;
N = 1e6;
puls1 = sqrt(variance)*randn(N,1);
puls2 = sqrt(variance)*randn(N,1);
```

```
intpuls = puls1 + puls2;
```

Compute the proportion of samples that exceed the threshold.

```
threshold = sqrt(variance*db2pow(T));
falsealarmrate = sum(intpuls > threshold)/N
```

```
falsealarmrate =

   9.8900e-04
```

Threshold for Complex-Valued Signals in Complex White Gaussian Noise

This example shows how to verify the probability of false alarm empirically in a system that uses complex-valued signals. The system uses *coherent detection*, which means that the system has information about the phase of the complex-valued signals.

Determine the required SNR for the NP detector in a coherent detection scheme with one sample. Use a maximum tolerable false-alarm probability of 10^{-3}.

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

Test that this threshold empirically results in the correct false-alarm rate The sufficient statistic in the complex-valued case is the real part of received sample.

```
rng default
variance = 1;
N = 1e6;
x = sqrt(variance/2)*(randn(N,1)+1j*randn(N,1));
threshold = sqrt(variance*db2pow(T));
falsealarmrate = sum(real(x)>threshold)/length(x)
```

```
falsealarmrate =

   9.9500e-04
```

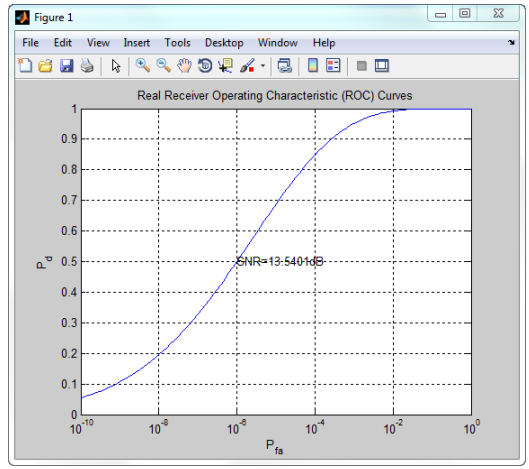
Receiver Operating Characteristic (ROC) Curves

ROC curves present graphical summaries of a detector's performance. You can generate ROC curves using the functions `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')
```

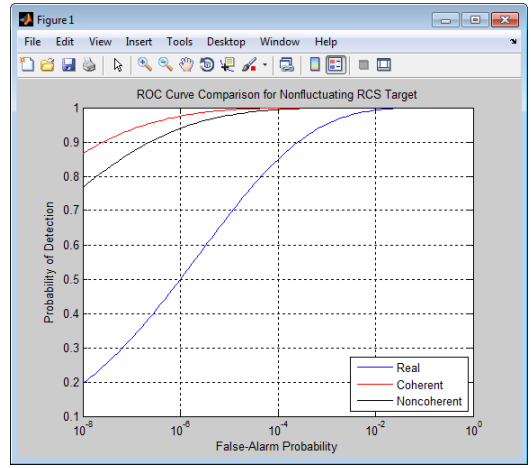


The ROC curve enables you to easily read off the probability of detection for a given false-alarm rate.

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);
```

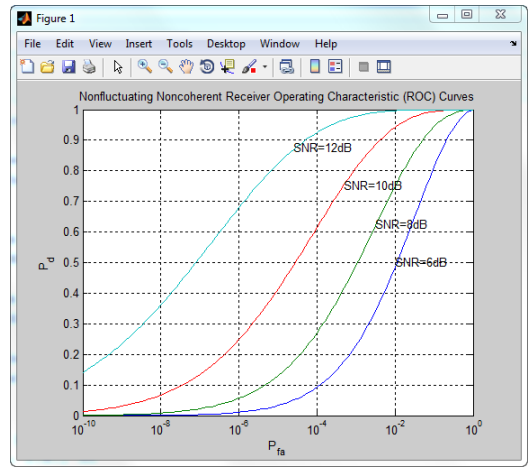
```
[Pd_coh,Pfa_coh] = rocsnr(SNR,...
    'SignalType','NonfluctuatingCoherent',...
    'MinPfa',1e-8);
[Pd_noncoh,Pfa_noncoh] = rocsnr(SNR,'SignalType',...
    'NonfluctuatingNoncoherent','MinPfa',1e-8);
figure;
semilogx(Pfa_real,Pd_real); hold on; grid on;
semilogx(Pfa_coh,Pd_coh,'r');
semilogx(Pfa_noncoh,Pd_noncoh,'k');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

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

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

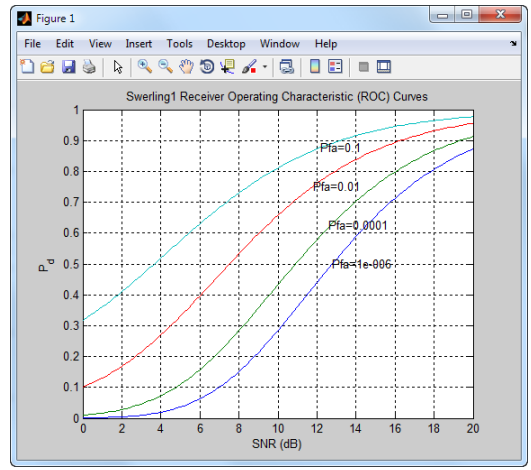


The graph shows that—as the SNR increases—the supports of the probability distributions under the null and alternative hypotheses become more disjoint. Therefore, for a given false-alarm probability, the probability of detection increases.

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

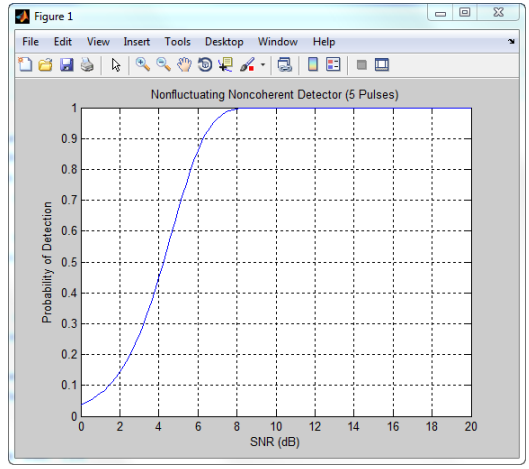
To obtain ROC curves for a Swerling I target model at false-alarm probabilities of `[1e-6 1e-4 1e-2 1e-1]`, enter:

```
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)');
```



Related Examples

- Detector Performance Analysis using ROC Curves

Matched Filtering

In this section...

Reasons for Using Matched Filtering

You can see from the results in "Receiver Operating Characteristic (ROC) Curves" on page 8-6 that the probability of detection increases with increasing SNR. For a deterministic signal in white Gaussian noise, you can maximize the SNR at the receiver by using a filter matched to the signal. The matched filter is a time-reversed and conjugated version of the signal. The matched filter is shifted to be causal.

Support for Matched Filtering

Use `phased.MatchedFilter` to implement a matched filter.

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

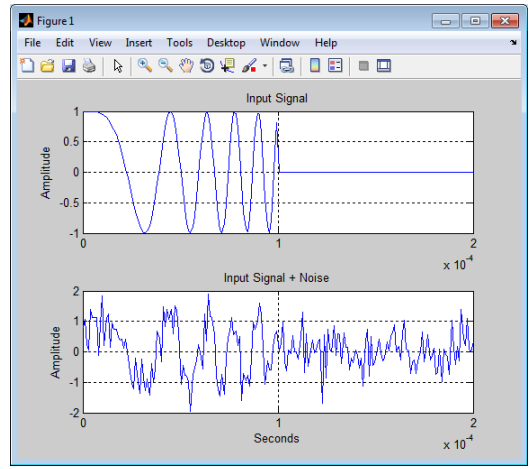
Matched Filtering of Linear FM Waveform

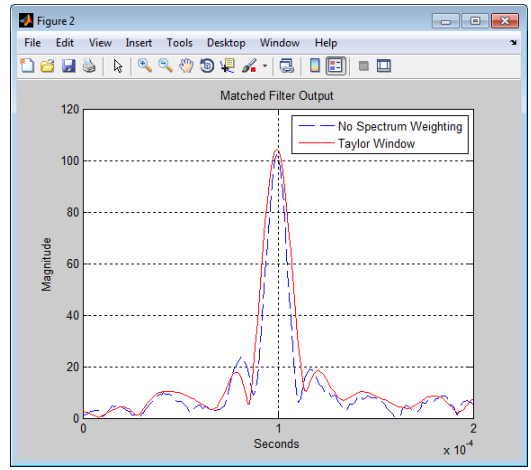
Create a linear FM waveform with a duration of 0.1 milliseconds, a sweep bandwidth of 100 kHz, and a pulse repetition frequency of 5 kHz. Add noise to the linear FM pulse and filter the noisy signal using the matched filter. Spectrum weighting is often used with linear FM waveform to reduce the sidelobes 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 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 example demonstrates the SNR improvement obtained after matched filtering.

Place an isotropic antenna element at the global origin `[0;0;0]`. Then, 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. This target has an RCS of 1 square meter at the maximum unambiguous range for the transmitter operating frequency and gain. The SNR is based on a desired false-alarm rate of `1e-6` for a noncoherent detector.

```
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. Then, create an ideal receiver and a matched filter for the rectangular waveform.

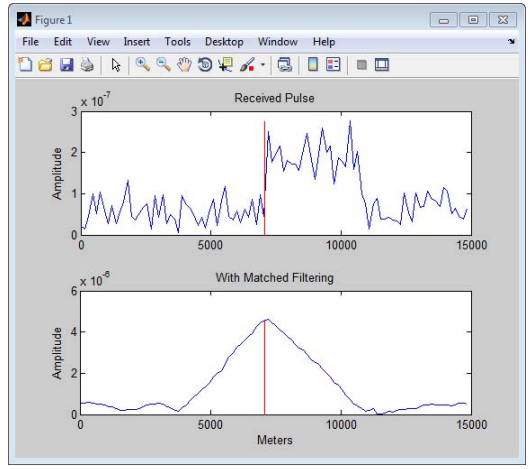
```
hrad = phased.Radiator(...
```

```
    '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);
```

After you create all the objects that define your model, you can propagate the pulse to and from the target. Collect the echo at the receiver, and implement the matched filter to improve the SNR.

```
% Generate waveform
wf = 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');
```



Stretch Processing

In this section...

Reasons for Using Stretch Processing

The linear FM waveform is popular in radar systems because its large time-bandwidth product can provide good range resolution. However, the large bandwidth of this waveform makes digital matched filtering difficult because it requires expensive, high-quality analog-to-digital converters. *Stretch processing*, also known as *deramping*, or *dechirping*, is an alternative to matched filtering. Stretch processing provides pulse compression by looking for the return within a predefined range interval of interest. Stretch processing occurs in the analog domain. Unlike matched filtering, stretch processing reduces the bandwidth requirement of subsequent processing.

Support for Stretch Processing

The `phased.StretchProcessor` System object implements stretch processing. You can use this object as part of a simulation that uses `phased.LinearFMWaveform` or directly with your own data.

Stretch Processing Procedure

The typical procedure for stretch processing is as follows:

1. Choose a range interval of interest, centered on a reference range. Stretch processing focuses on this interval instead of the entire range span that the pulse can cover.

2. Define and configure a stretch processor object. The configuration includes the reference range, length of the range interval of interest, characteristics of the linear FM waveform, and signal propagation speed.

- If you are using a `phased.LinearFMWaveform` object to implement the linear FM waveform, use the `getStretchProcessor` method to define and automatically configure a stretch processor object.
- Otherwise, create a `phased.StretchProcessor` object directly, and set its properties as needed.

3. Perform stretch processing by calling the `step` method on your stretch processor object. You provide your received signal as an input argument. The `step` method generates a reference signal and correlates it with your received signal.

4. Compute a periodogram of the output from `step`, and identify the peak frequencies. You can use the following features to help you perform this step:
   - `spectrum.periodogram`
   - `psd`
   - `findpeaks`

5. Convert each peak frequency to the corresponding range value, using the `stretchfreq2rng` function.

See Also

`phased.StretchProcessor` | `phased.LinearFMWaveform` | `stretchfreq2rng` | `spectrum.periodogram` | `findpeaks`

Related Examples

- Range Estimation Using Stretch Processing

Constant False-Alarm Rate (CFAR) Detectors

In this section...

Reasons for Using CFAR Detectors

In the Neyman-Pearson framework, the probability of detection is maximized subject to the constraint that the false-alarm probability does not exceed a specified level. The false-alarm probability depends on the noise variance. Therefore, to calculate the false-alarm probability, you must first estimate the noise variance. If the noise variance changes, you must adjust the threshold to maintain a constant false-alarm rate. *Constant false-alarm rate detectors* implement adaptive procedures that enable you to update the threshold level of your test when the power of the interference changes.

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

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

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.

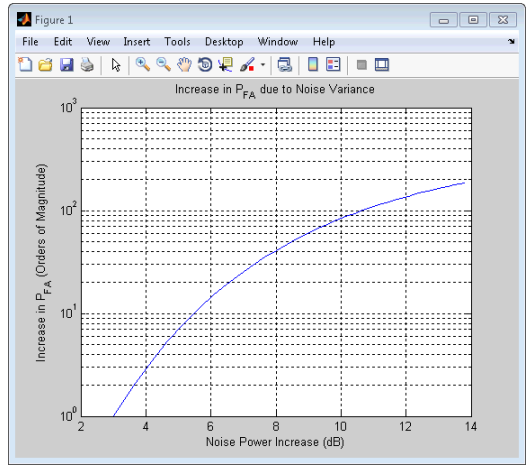
```
% check false-alarm probability
Pfa = 0.5*erfc(threshold/sqrt(2))
% compute probability of detection
Pd = 0.5*erfc((threshold-4)/sqrt(2))
```

Next, assume that the noise power increases by 6.02 dB, doubling the noise variance. If your detector does not adapt to this increase in variance by determining a new threshold, your false-alarm rate increases significantly.

```
Pfa = 0.5*erfc(threshold/2)
```

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)');
```



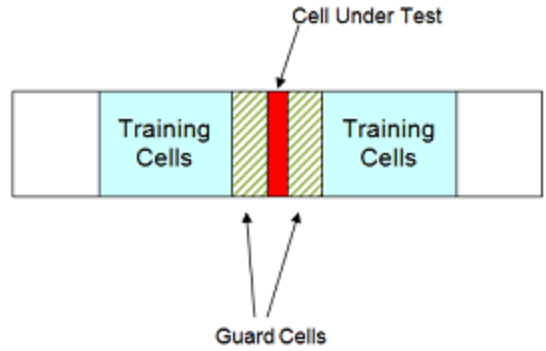
Cell-Averaging CFAR Detector

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 noise characteristics in the training cells are assumed to be identical to the noise characteristics in the cell under test (CUT).

This assumption is key in justifying the use of the training cells to estimate the noise variance in the CUT. Additionally, the cell-averaging CFAR detector assumes that the training cells do not contain any signals from targets. Thus, the data in the training cells are assumed to consist of noise only.

To make these assumptions realistic:

- It is preferable to have some buffer, or *guard cells*, between the CUT and the training cells. The buffer provided by the guard cells *guards* against signal leaking into the training cells and adversely affecting the estimation of the noise variance.
- The training cells should not represent range cells too distant in range from the CUT, as the following figure illustrates.



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

1. You are using a square-law detector.
2. You have a Gaussian, complex-valued, random variable (RV) with independent real and imaginary parts.
3. The real and imaginary parts each have mean zero and variance equal to $\sigma^2/2$.

Note If you denote this RV by Z=U+jV, the squared magnitude $|Z|^2$ follows an exponential distribution with mean σ^2.

If the samples in training cells are the squared magnitudes of such complex Gaussian RVs, you can use the sample mean as an estimator of the noise variance.

To implement cell-averaging CFAR detection, use `phased.CFARDetector`. You can customize characteristics of the detector such as the numbers of training cells and guard cells, and the probability of false alarm.

Testing CFAR Detector Adaption to Noisy Input Data

This example shows how to create a CFAR detector and test its ability to adapt to the statistics of input data. The test uses noise-only trials. By using the default square-law detector, you can determine how close the empirical false-alarm rate is to the desired false-alarm probability.

Create a CFAR detector object with two guard cells, 20 training cells, and a false-alarm probability of 0.001. By default, this object assumes a square-law detector with no pulse integration.

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

There are 10 training cells and 1 guard cell on each side of the cell under test (CUT). Set the CUT index to 12.

```
CUTidx = 12;
```

Seed the random number generator for a reproducible set of input data.

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

Set the noise variance to 0.25. This value corresponds to an approximate –6 dB SNR. Generate a 23-by-10000 matrix of complex-valued, white Gaussian 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 the example implements a square-law detector, take the squared magnitudes of the elements in the data matrix.

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

Provide the output 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, which corresponds closely to the desired false-alarm rate of 0.001.

9

Environment and Target Models

Free Space Path Loss

Propagation environments have significant effects on the amplitude, phase, and shape of propagating space-time wavefields. If you are simulating a system that propagates narrowband signals through free space, you can use the `phased.FreeSpace` object to model the range-dependent time delay, phase shift, and gain effects.

The free space path loss is:

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

R represents 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 Decibels

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 decibels 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 decibels is approximately 105 dB. You can express this value as:

```
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 decibels, 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 the following modifiable properties:

- `PropagationSpeed` — The propagation speed of the wave.
- `OperatingFrequency` — The system operating frequency.
- `TwoWayPropagation` — Perform two-way propagation. To model two-way propagation, set this property value to `true`.
- `SampleRate` — The sampling rate in hertz.

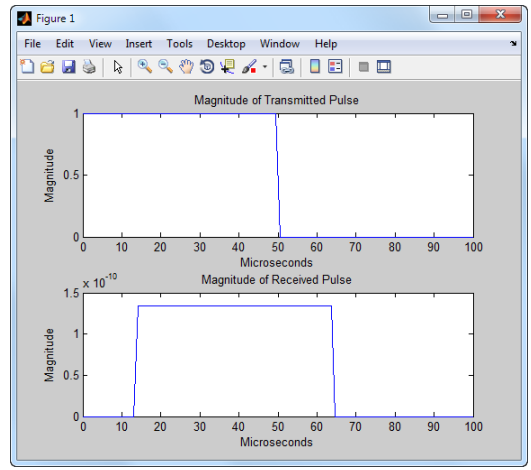
Propagate a Linear FM Pulse Waveform to and from a Target

Construct a linear FM pulse waveform 50 ms in duration with a bandwidth of 100 kHz. Model the range-dependent time delay and amplitude loss incurred during two-way propagation. The pulse propagates between the transmitter located at `[1000; 250; 10]` and a target location of `[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 µs, 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 how to use this property 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. The following code calls the `step` after the pulse is radiated from the antenna and before the pulse 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 so 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). This object has the following modifiable properties:

- `MeanRCSSource` — Source of the target's mean radar cross section
- `MeanRCS` — Target's mean RCS
- `Model` — Statistical model for the target's RCS
- `PropagationSpeed` — Signal propagation speed
- `OperatingFrequency` — Operating frequency
- `SeedSource` — Source of the seed for the random number generator to generate the target's random RCS values
- `Seed` — Seed for the random number generator

Create a radar target with a nonfluctuating RCS of 1 square meter and an operating frequency of 300 MHz. Specify 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:

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

Here, σ represents 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 equal to `11.8245*ones(10,1)`. The amplitude scaling factor equals:

```
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. These fluctuations can occur rapidly on pulse-to-pulse, or more slowly, on scan-to-scan time scales:

- **Several small randomly distributed reflectors with no dominant reflector** — This target, at close range or when the radar uses pulse-to-pulse frequency agility, can exhibit large magnitude rapid (pulse-to-pulse) fluctuations in the RCS. That same complex reflector at long range with no frequency agility can exhibit large magnitude fluctuations in the RCS over a longer time scale (scan-to-scan).
- **Dominant reflector along with several small reflectors** — The reflectors in this target can exhibit small magnitude fluctuations on pulse-to-pulse or scan-to-scan time scales, subject to:
  - How rapidly the aspect changes
  - Whether the radar uses frequency agility

To account for significant fluctuations in the RCS, you need to use statistical models. The four *Swerling* models, described in the following table, are widely used to cover these kinds of fluctuating-RCS cases.

Swerling Case Number	Description
I	Scan-to-scan decorrelation. Rayleigh/exponential PDF — A number of randomly distributed scatterers with no dominant scatterer.
II	Pulse-to-pulse decorrelation. Rayleigh/exponential PDF — A number of randomly distributed scatterers with no dominant scatterer.
III	Scan-to-scan decorrelation — Chi-square PDF with 4 degrees of freedom. A number of scatterers with one scatterer dominant.
IV	Pulse-to-pulse decorrelation — Chi-square PDF with 4 degrees of freedom. A number of scatterers with one scatterer dominant.

You can simulate a Swerling target model by setting the `Model` property. Use the `step` method and set the `UPDATERCS` input argument to `true` or `false`. Setting `UPDATERCS` to `true` updates the RCS value according to the specified probability model each time you call `step`. If you set `UPDATERCS` to `false`, the previous RCS value is used.

Model Pulse Reflection from 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
```

Clutter Modeling

In this section...

Surface Clutter Overview

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

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

Approaches for Clutter Simulation or Analysis

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

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

- `grazingang`
- `horizonrange`
- `surfclutterrcs`
- `surfacegamma`

Considerations for Setting Up a Constant Gamma Clutter Simulation

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

Physical Configuration Properties

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

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

Clutter-Related Properties

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

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

  In the simulation, you can use identical random numbers over a time interval or uncorrelated random numbers. Simulation behavior slightly

differs from reality, where a moving platform produces clutter returns that are correlated with each other over small time intervals.

Working with Samples or Pulses

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

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

Assumptions

The clutter simulation that `ConstantGammaClutter` provides makes these assumptions:

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

Related Examples

- Ground Clutter Mitigation with Moving Target Indication (MTI) Radar

- Introduction to Space-Time Adaptive Processing
- “Example: DPCA Pulse Canceller for Clutter Rejection” on page 7-9
- “Example: Adaptive DPCA Pulse Canceller” on page 7-14
- “Example: Sample Matrix Inversion (SMI) Beamformer” on page 7-21

Barrage Jammer

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

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

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

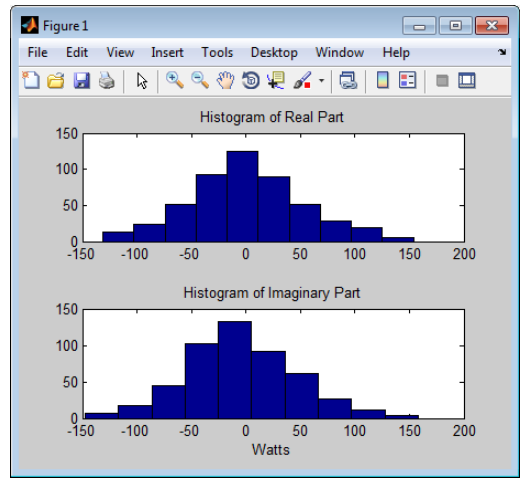
$$w[n] = \sqrt{\frac{P}{2}}x[n] + j\sqrt{\frac{P}{2}}y[n]$$

In this equation, *x[n]* and *y[n]* are 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 W. 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 Real Part');
subplot(2,1,2)
hist(imag(y)); title('Histogram of Imaginary Part');
xlabel('Watts');
```



Model Effect of Barrage Jammer on Target Echo

This example demonstrates how to simulate the effect of a barrage jammer on a target echo.

First, create the required objects. You need an array, a transmitter, a radiator, a target, a jammer, a collector, and a receiver. Additionally, you need to define two propagation paths: one from the array to the target and back, and the other path from the jammer to the array.

```
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);
```

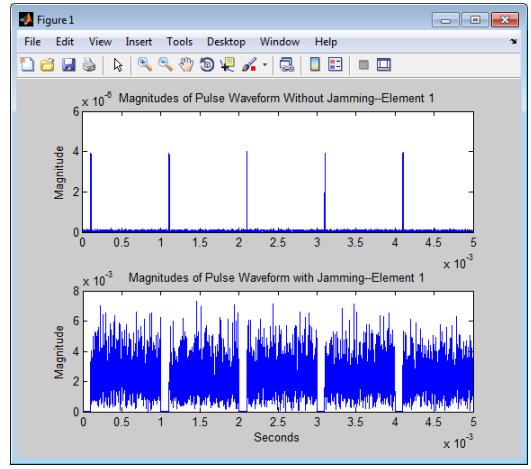
Finally, transmit the rectangular pulse waveform to the target, reflect it off the target, and collect the echo at the array. Simultaneously, the jammer transmits a jamming signal toward the array. The jamming signal and echo are mixed at the receiver.

```
% 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 it 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');
```



10

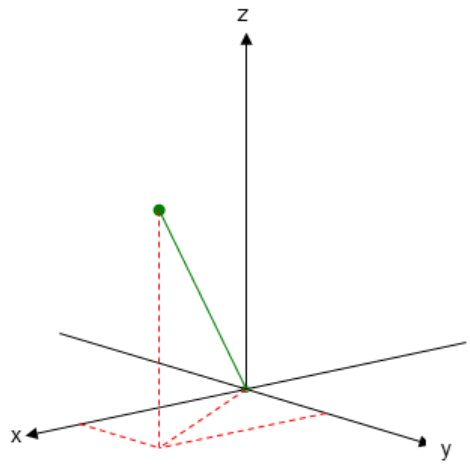
Coordinate Systems and Motion Modeling

Rectangular Coordinates

In this section...

Definitions of Coordinates

Construct a rectangular, or Cartesian, coordinate system for three-dimensional space by specifying three mutually orthogonal coordinate axes. The following figure shows one possible specification of the coordinate axes.



Rectangular coordinates specify a position in space in a given coordinate system as an ordered 3-tuple of real numbers, (x,y,z), with respect to the origin (0,0,0). Considerations for choosing the origin are discussed in "Global and Local Coordinate Systems" on page 10-15.

You can view the 3-tuple as a point in space, or equivalently as a vector in three-dimensional Euclidean space. Viewed as a vector space, the coordinate

axes are basis vectors and the vector gives the direction to a point in space from the origin. Every vector in space is uniquely determined by a linear combination of the basis vectors. The most common set of basis vectors for three-dimensional Euclidean space are the standard unit basis vectors:

{[1 0 0],[0 1 0],[0 0 1]}

Notation for Vectors and Points

In Phased Array System Toolbox software, you specify both coordinate axes and points as column vectors.

Note In this software, all coordinate vectors are column vectors. For convenience, the documentation represents column vectors in the format [*x y z*] without transpose notation.

Both the vector notation [*x y z*] and point notation (*x*,*y*,*z*) are used interchangeably. The interpretation of the column vector as a vector or point depends on the context. If the column vector specifies the axes of a coordinate system or direction, it is a vector. If the column vector specifies coordinates, it is a point.

Orthogonal Basis and Euclidean Norm

Any three linearly independent vectors define a basis for three-dimensional space. However, this software assumes that the basis vectors you use are orthogonal.

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

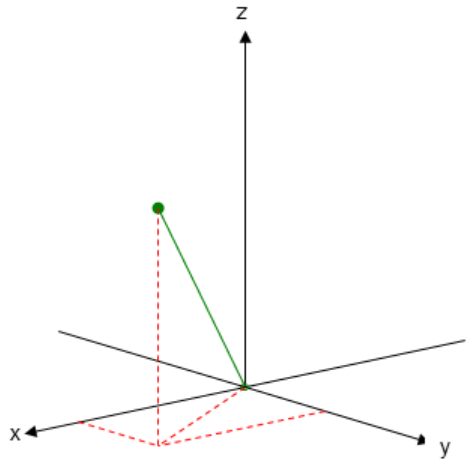
$$\sqrt{x^2 + y^2 + z^2}$$

The Euclidean norm gives the length of the vector measured from the origin as the hypotenuse of a right triangle. The distance between two vectors [x_0 y_0 z_0] and [x_1 y_1 z_1] is:

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

Orientation of Coordinate Axes

Given an orthonormal set of basis vectors representing the coordinate axes, there are multiple ways to orient the axes. The following figure illustrates one such orientation, called a *right-handed* coordinate system. The arrows on the coordinate axes indicate the positive directions.



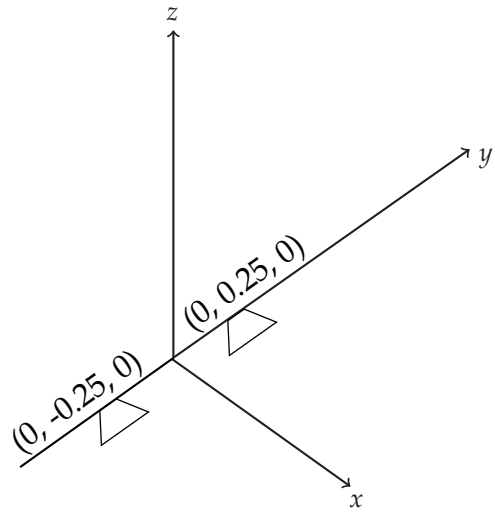
If you take your right hand and point it along the positive *x*-axis with your palm facing the positive *y*-axis and extend your thumb, your thumb indicates the positive direction of the *z*-axis.

Phase Center of Array

For some 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:

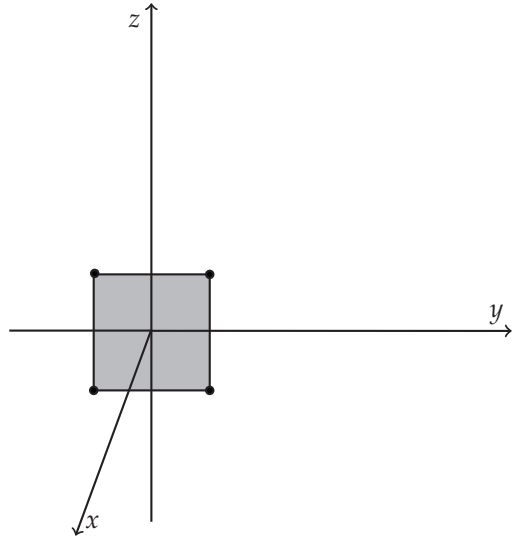
```
H = phased.ULA('NumElements',2,'ElementSpacing',0.5)
getElementPosition(H)
```

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



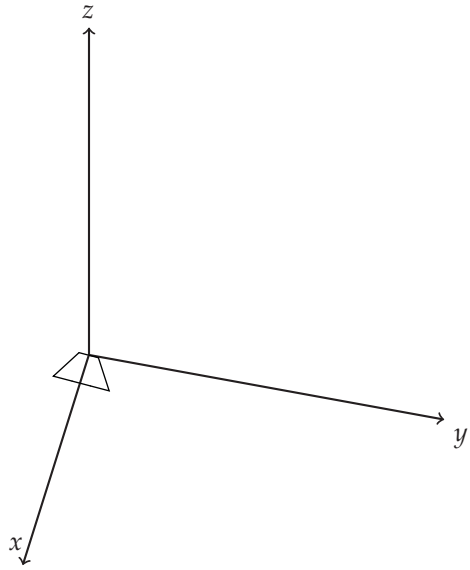
The next example creates a uniform rectangular array (URA) with four elements:

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



Boresight Direction

The direction that an antenna is facing when transmitting and receiving a signal is referred to as the *boresight*, or *look* direction. The convention in this software for specifying coordinate axes designates the positive *x*-axis as the boresight direction. The following figure shows a right-handed coordinate system with the positive *x*-axis oriented along the sensor boresight direction.



Common Boresight Direction

In ULAs and URAs, the boresight 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 alignment is referred to as the *array axis*. The array elements have a common boresight direction, which the toolbox designates as the *x*-axis. The *x*-axis is 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 boresight direction. The *x*-axis is normal to the plane containing the array elements. This alignment illustrates the software 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 boresight direction and the direction normal to each element is a property of the individual array elements.

More About

- "Global and Local Coordinate Systems" on page 10-15

Spherical Coordinates

In this section...

Support for Spherical Coordinates

Spherical coordinates describe a vector or point in space with a distance and two angles. The distance, R, is the usual Euclidean norm. There are multiple conventions regarding the specification of the two angles. They include:

- Azimuth and elevation angles
- Phi and theta angles
- u and v coordinates

Phased Array System Toolbox software natively supports the azimuth/elevation representation. The software also provides functions for converting between the azimuth/elevation representation and the other representations. See "Phi and Theta Angles" on page 10-9 and "U and V Coordinates" on page 10-10.

Azimuth and Elevation Angles

In Phased Array System Toolbox software, the predominant convention for spherical coordinates is as follows:

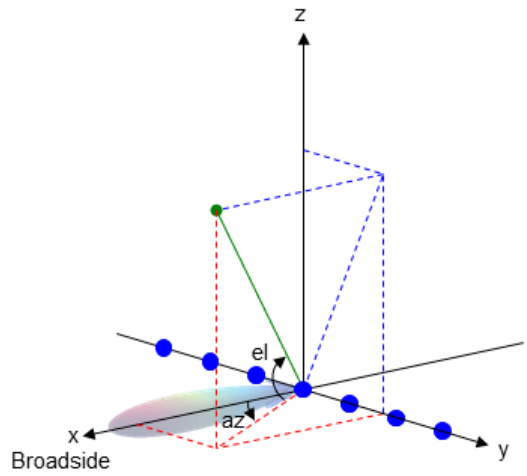
- Use the azimuth angle, *az*, and the elevation angle, *el*, to define the location of a point on the unit sphere.
- Specify all angles in degrees.

- List coordinates in the sequence (*az*,*el*,*R*).

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

Note The elevation angle is sometimes defined in the literature as the angle a vector makes with the positive *z*-axis. The MATLAB and Phased Array System Toolbox products do not use this definition.

This figure illustrates the azimuth angle and elevation angle for a vector that appears as a green solid line. The coordinate system is relative to the center of a uniform linear array, whose elements appear as blue circles.



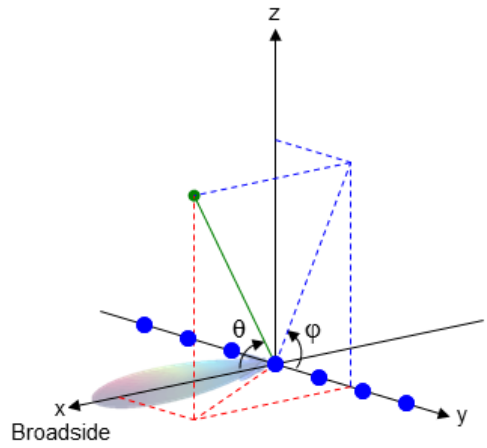
Phi and Theta Angles

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

representation, use coordinate conversion functions, `phitheta2azel` and `azel2phitheta`.

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

The figure illustrates φ and θ for a vector that appears as a green solid line. The coordinate system is relative to the center of a uniform linear array, whose elements appear as blue circles.



U and V Coordinates

In radar applications, it is often useful to parameterize the hemisphere $x \geq 0$ using coordinates denoted by *u* and *v*.

- To convert the φ/θ representation to and from the corresponding *u*/*v* representation, use coordinate conversion functions `phitheta2uv` and `uv2phitheta`.
- To convert the azimuth/elevation representation to and from the corresponding *u*/*v* representation, use coordinate conversion functions `azel2uv` and `uv2azel`.

You can define u and v in terms of φ and θ:

$u = \sin(\theta)\cos(\varphi)$

$v = \sin(\theta)\sin(\varphi)$

In these expressions, φ and θ are the phi and theta angles, respectively.

The values of u and v satisfy these inequalities:

$-1 \le u \le 1$

$-1 \le v \le 1$

$u^2 + v^2 \le 1$

Conversion Between Rectangular and Spherical Coordinates

The following equations define the relationships between rectangular coordinates and the (*az*,*el*,*R*) representation used in Phased Array System Toolbox software.

To convert rectangular coordinates to (*az*,*el*,*R*):

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

To convert (*az*,*el*,*R*) to rectangular coordinates:

$$x = R\cos(el)\cos(az)$$
$$y = R\cos(el)\sin(az)$$
$$z = R\sin(el)$$

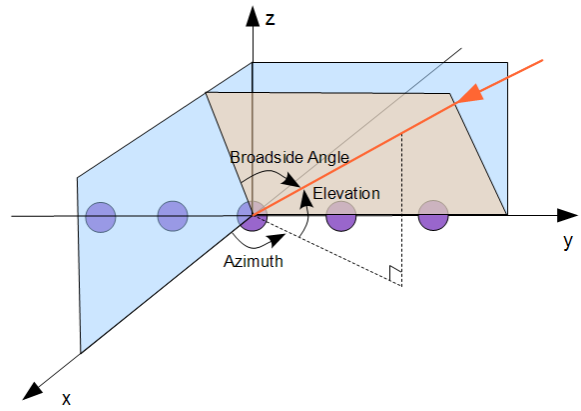
When specifying a target's location with respect to a phased array, it is common to refer to its distance and direction from the array. The distance

from the array corresponds to R in spherical coordinates. The direction corresponds to the azimuth and elevation angles.

Tip To convert between rectangular coordinates and (az,el,R), use the MATLAB functions `cart2sph` and `sph2cart`. These functions specify angles in radians. To convert between degrees and radians, use `degtorad` and `radtodeg`.

Broadside Angle

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



The shaded gray area in the figure is the plane determined by the signal incident direction and the array axis. The broadside angle is positive when measured toward the positive direction of the array axis. A number of algorithms for ULAs use the broadside angle instead of the azimuth and elevation angles. The algorithms do so because the broadside angle more accurately describes the ability to discern direction of arrival with this geometry.

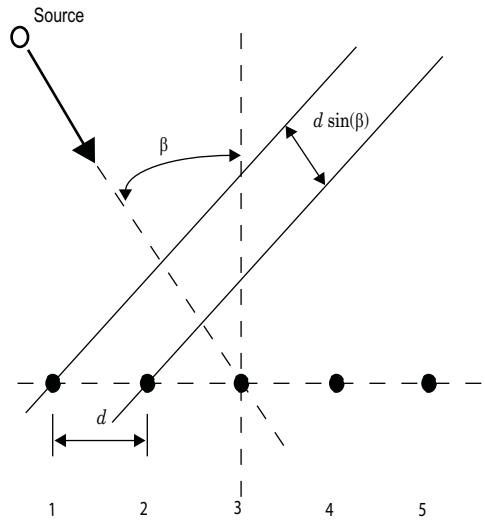
Phased Array System Toolbox software provides functions `az2broadside` and `broadside2az` for converting between azimuth and broadside angles. The following equation determines the broadside angle, ß, from the azimuth and elevation angles, *az* and *el*:

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

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

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

The following figure depicts a ULA with elements spaced *d* meters apart. The ULA is illuminated by a plane wave emitted from a point source in the far field. For convenience, the elevation angle is zero degrees. The plane determined by the signal incident direction and the array axis is the *xy* plane. The broadside angle reduces to the azimuth angle.



Because of the angle of arrival, the array elements are not simultaneously illuminated by the plane wave. The additional distance the incident wave travels between array elements is d sin(ß) where d is the distance between array elements. Therefore, the constant time delay between array elements is:

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

where c is the speed of the wave.

For broadside angles of ±90 degrees, the plane wave is incident on the array along the array axis and the time delay between sensors reduces to $\pm d/c$. 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

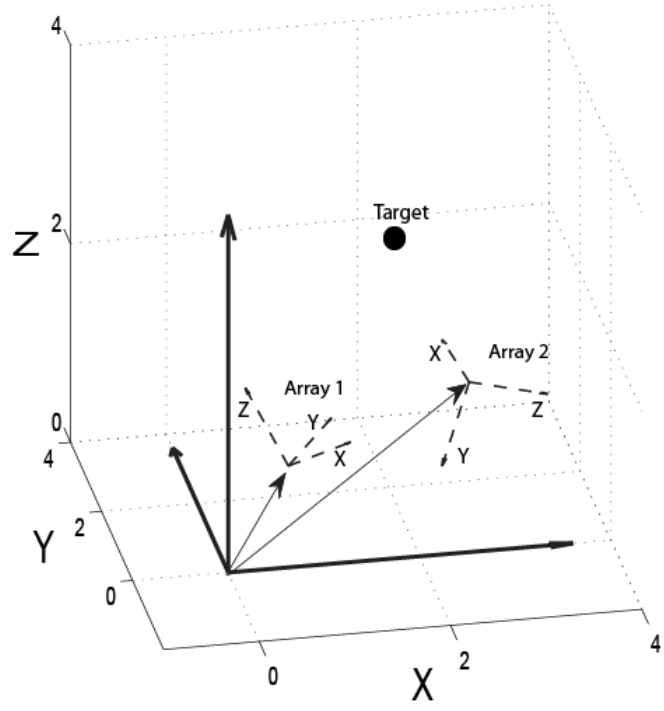
In this section...

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 want to ignore as well as stationary and mobile sources that are actively attempting to interfere with your phased arrays (jammers).

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. 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.

You can 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. The direction of arrival corresponds to the azimuth and elevation angles. Phased Array System Toolbox software follows the MATLAB convention and lists spherical coordinates in the order: (az,el,R).

The positions of all array elements in this software are in local coordinates. The following examples illustrate local coordinate systems for uniform linear, uniform rectangular, and conformal arrays.

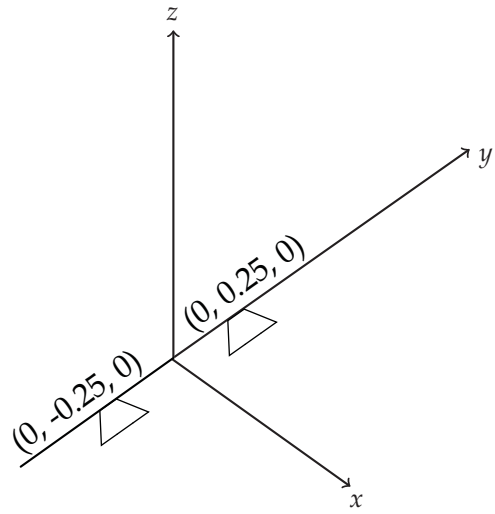
Local Coordinate System for a Uniform Linear Array

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 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 eight 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)
```

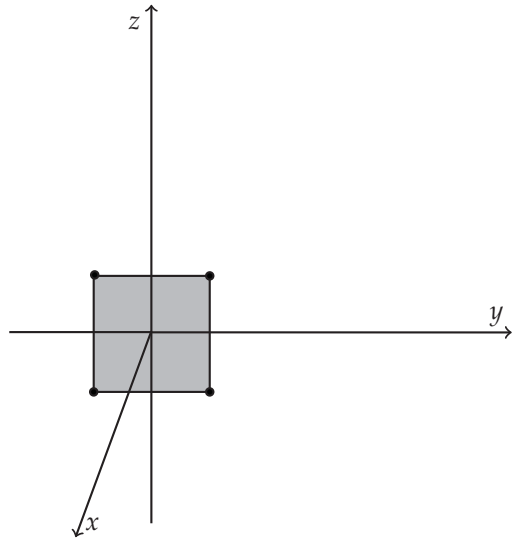
Local Coordinate System of a Uniform Rectangular Array

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. In the *yz* plane, the array elements have even row spacing and even column spacing.

Construct a URA:

```
H = phased.URA('Size',[2 2],'ElementSpacing',[0.5 0.5])
ElementLocs = 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])
ElementLocs2by3 = 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. Two utility functions, `global2localcoord` and `local2globalcoord`, enable you to do this conversion.

Convert Local Spherical Coordinates to Global Rectangular Coordinates

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

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

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

Convert Global Rectangular Coordinates to Local Spherical Coordinates

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

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

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

Motion Modeling in Phased Array Systems

A critical component in phased array system applications is the ability to model motion in space. Such modeling includes the motion of arrays, targets, and sources of interference. For convenience, you can ignore the distinction between these objects and collectively model the motion of a platform.

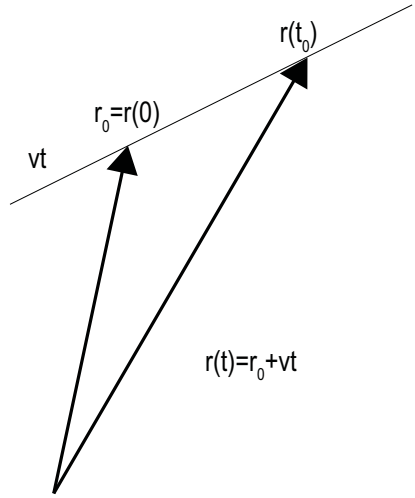
Extended bodies can undergo both translational and rotational motion in space. Phased Array System Toolbox software supports modeling of translational motion.

Modeling translational platform motion requires the specification of a position and velocity vector. Specification of a position vector implies a coordinate system. In the Phased Array System Toolbox, platform position and velocity are specified in a "Global Coordinate System" on page 10-15. You can think of the platform position as the displacement vector from the global origin or as the coordinates of a point with respect to the global origin.

Let r_0 denote the position vector at time 0 and v denote the velocity vector. The position vector of a platform as a function of time, *r(t)*, is:

$$r(t) = r_0 + vt$$

The following figure depicts the vector interpretation of translational motion.



When the platform represents a sensor element or array, it is important to know the orientation of the element or array *local coordinate axes*. For example, the orientation of the local coordinate axes is necessary to extract angle information from incident waveforms. See "Global and Local Coordinate Systems" on page 10-15 for a description of global and local coordinate systems in the software. Finally, for platforms with nonconstant velocity, you must be able to update the velocity vector over time.

You can model platform position, velocity, and local axes orientation with the `phased.Platform` object.

Platform Motion with Constant Velocity

Beginning with a simple example, model the motion of a platform over ten time steps. To determine the time step, assume that you have a pulse transmitter with a pulse repetition frequency (PRF) of 1 kilohertz. Accordingly, the time interval between each pulse is 1 millisecond. Set the time step equal to pulse repetition interval.

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

Next, construct a platform object specifying the platform's initial position and velocity. Assume that the initial position of the platform is 100 meters (m) from the origin at (60,80,0). Assume the speed is approximately 30 meters per second (m/s) with the constant velocity vector given by (15, 25.98, 0).

```
hplat = phased.Platform('InitialPosition',[60;80;0], ...
    'Velocity', [15;25.98;0]);
```

The orientation of the local coordinate axes of the platform is the value of the `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 that position. Accordingly, you expect the final position to differ from the initial position by 0.30 meters. Confirm this difference by examining the value of `DistTravel`.

Platform Motion with Nonconstant Velocity

Most platforms in phased array applications do not move with constant velocity. If the time interval described by the number of time steps is small with respect to the platform's speed, you can often approximate the velocity as constant. However, there are situations where you must update the

platform's velocity over time. You can do so with `phased.Platform` because the `Velocity` property is *tunable*. See "What are System Object Locking and Property Tunability?" for details.

In this example, assume you model a target initially at rest. The initial velocity vector is (0,0,0). Assume the time step is 1 millisecond. After 500 milliseconds, the platform begins to move with a speed of approximately 10 m/s. The velocity vector is (7.07,7.07,0). The platform continues at this velocity for an additional 500 milliseconds.

```
Tstep = 1e-3;
Nsteps = 1/Tstep;
hplat = phased.Platform('InitialPosition',[100;100;0]);
for k = 1:Nsteps/2
    [pos,vel] = step(hplat,Tstep);
end
hplat.Velocity = [7.07; 7.07; 0];
for k=Nsteps/2+1:Nsteps
    [pos,vel] = step(hplat,Tstep);
end
```

Track Range and Angle Changes Between Platforms

This example uses the `phased.Platform` object to model the changes in range between a stationary radar and a moving target. The radar is located at (1000,1000,0) and has a velocity of (0,0,0). The target has an initial position of (5000,8000,0) and moves with a constant velocity of (–30,–45,0). The pulse repetition frequency (PRF) is 1 kHz. Assume that the radar emits ten pulses.

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

```
PRF = 1e3;
Tstep = 1/PRF;
hradar = phased.Platform('InitialPosition',[1000;1000;0]);
htgt = phased.Platform('InitialPosition',[5000;8000;0],...
    'Velocity',[-30;-45;0]);
% Calculate initial target range and angle
[InitRng, InitAng] = rangeangle(htgt.InitialPosition,...
```

```
    hradar.InitialPosition);
% Calculate relative radial speed
v = radialspeed(htgt.InitialPosition,htgt.Velocity,...
    hradar.InitialPosition);
% Simulate target motion
Npulses = 10;                           % Number of pulses
for num = 1:Npulses
    tgtpos = step(htgt,Tstep);
end
tgtpos = tgtpos+htgt.Velocity*Tstep;
% Calculate final target range and angle
[FinalRng,FinalAng] = rangeangle(tgtpos,...
    hradar.InitialPosition);
DeltaRng = FinalRng-InitRng;
```

The constant velocity of the target is approximately 54 m/s. The total time elapsed is 0.01 seconds. The range between the target and the radar should decrease by approximately 54 centimeters. Compare the initial range of the target, `InitRng`, to the final range, `FinalRng`, to confirm that this decrease occurs.

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 (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 target
    txsig = step(hspace,txsig,transpos,...
        tgtpos); % propagate waveform to target
    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;
```

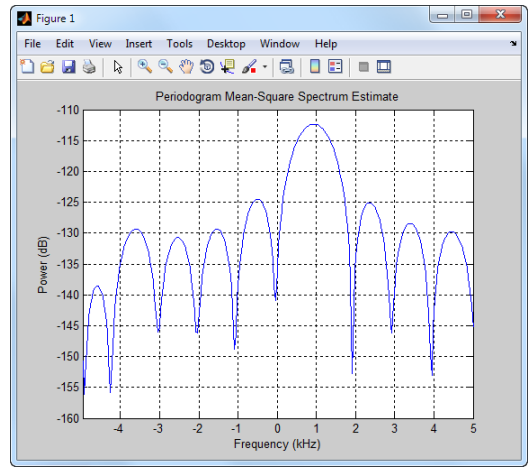
The next step is to detect range bins which contain targets. In this simple scenario, no matched filtering or time-varying gain compensation is utilized. See the Doppler Estimation demo for an example using matched filtering and range-dependent gain compensation to improve the SNR.

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_estimate);
```

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 transmitter.

```
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.

11

Define New System Objects

Define Basic System Objects

This example shows the structure of a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

1. Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

   ```
   edit AddOne.m
   ```

2. Subclass your object from `matlab.System`. Insert this line as the first line of your file.

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

3. Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access to `protected` because it is an internal method that users do not directly call or run.

   All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle.

   In this example, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables from occurring.

   By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

   ```
   methods (Access=protected)
      function y = stepImpl(~, x)
         y = x + 1;
      end
   end
   ```

Note Instead of manually creating your class definition file, you can use **File > New > System Object** to open a sample System object file in the editor. You then can edit that file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic System Object

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

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

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

See Also

`stepImpl` | `getNumInputsImpl` | `getNumOutputsImpl` | `matlab.System` |

Related Examples

- "Change Number of Step Method Inputs or Outputs" on page 11-4

More About

- "Process Data using System Objects"
- *Object-Oriented Programming*
- "Class Definition—Syntax Reference"
- "Methods — Defining Class Operations"

Change Number of Step Method Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you do not specify the `getNumInputsImpl` and `getNumOutputsImpl` methods, the object uses the default values of 1 input and 1 output. In this case, the user must provide an input to the `step` method.

To specify no inputs, you must explicitly set the number of inputs to 0 using the `getNumInputsImpl` method. To specify no outputs, you must explicitly return 0 in the `getNumOutputsImpl` method.

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

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In this example, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables from occurring.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to accept a second input and provide a second output.

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

Update the Associated Methods

Use `getNumInputsImpl` and `getNumOutputsImpl` to specify two inputs and two outputs, respectively.

```
methods (Access=protected)
   function numIn = getNumInputsImpl(~)
      numIn = 2;
   end

   function numOut = getNumOutputsImpl(~)
      numOut = 2;
   end
end
```

Complete Class Definition File with Multiple Inputs and Outputs

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

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

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

    % getNumInputsImpl method calculates number of inputs
    function num = getNumInputsImpl(~)
      num = 2;
    end

    % getNumOutputsImpl method calculates number of outputs
    function num = getNumOutputsImpl(~)
      num = 2;
    end
  end
end
```

See Also

`getNumInputsImpl` | `getNumOutputsImpl` |

Related Examples

- "Validate Property and Input Values" on page 11-7
- "Define Basic System Objects" on page 11-2

More About

- "Class Definition—Syntax Reference"
- "Methods — Defining Class Operations"

Validate Property and Input Values

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

You use the `validateInputsImpl` and `validatePropertiesImpl` methods to perform this validation

Note All inputs default to variable-size inputs. See "Change System Object Input Complexity or Dimensions" for more information.

Validate Properties

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

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

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

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

```
 end
```

Validate Inputs

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

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

Complete Class Definition File with Property and Input Validation

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

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

  properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
  end

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

  methods (Access=protected)
```

```
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue < obj.Increment
          error('Wrap value must be less than increment value');
        end
    end

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

    function out = stepImpl(obj,in)
      if obj.UseIncrement
        y = x + obj.Increment;
      else
        y = x + 1;
      end
    end
  end
end
```

See Also

`validateInputsImpl` | `validatePropertiesImpl` |

Related Examples

- "Define Basic System Objects" on page 11-2
- "Validate Property and Input Values" on page 11-7

More About

- "Methods Timing" on page 11-36
- "Property Set Methods"

Initialize Properties and Setup One-Time Calculations

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

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

Define Properties to Initialize

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

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

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

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

Define Setup

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

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

```
    end
  end
end
```

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

Complete Class Definition File with Initialization and Setup

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

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

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

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

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

        % Use release to close the file to prevent the
```

```
            % file handle from being left open.
            function releaseImpl(obj)
                fclose(obj.pFileID);
            end

            % You indicate that no outputs are provided by returning
            % zero from getNumOutputsImpl
            function numOutputs = getNumOutputsImpl(~)
                numOutputs = 0;
            end
        end
    end
```

See Also

`setupImpl` | `releaseImpl` | `stepImpl` |

Related Examples

- "Release System Object Resources" on page 11-28
- "Define Property Attributes" on page 11-18

More About

- "Methods Timing" on page 11-36

Set Property Values at Construction from Name-Value Pairs

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

Set Properties to Use Name-Value Pair Input

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

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

Complete Class Definition File with Constructor Setup

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

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

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

    methods
```

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

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

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

            % Use release to close the file to prevent the
            % file handle from being left open.
            function releaseImpl(obj)
                fclose(obj.pFileID);
            end
            % You indicate that no outputs are provided by returning
            % zero from getNumOutputsImpl
            function numOutputs = getNumOutputsImpl(~)
                numOutputs = 0;
            end
        end
    end
```

See Also

narginsetProperties |

Related Examples

- "Define Property Attributes" on page 11-18
- "Release System Object Resources" on page 11-28

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

`pCount` is an internal counter property of the System object `obj`. The user calls the `reset` method, which calls the `resetImpl` method. In this example , `pCount` resets to 0. See "Methods Timing" on page 11-36 for more information.

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

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

Complete Class Definition File with State Reset

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

   properties(Access = private)
      pCount
   end

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

      % Reset the counter to zero.
      function resetImpl(obj)
```

```
            obj.pCount = 0;
        end

        % The step method takes no inputs
        function numIn = getNumInputsImpl(~)
          numIn = 0;
        end
    end
end
end
```

See Also

`resetImpl` |

More About

- "Methods Timing" on page 11-36

Define Property Attributes

This example shows how to specify property attributes.

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

Specify Property as Nontunable

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

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

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter `1` or `0`, but the value displays as `true` or `false`, respectively. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property.

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

Specify Property as Positive Integer

In this example, the private property `pCount` is constrained to accept only real, positive integers. You cannot use sparse values.

```
properties (PositiveInteger)
   Count
end
```

Specify Property as DiscreteState

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

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

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

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

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
%Counter Increment a counter starting at an initial value

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
      % The inital value of the counter
      InitialValue = 0
  end

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

```
    end

    % Count state variable
    properties (DiscreteState, PositiveInteger)
        Count
    end

    methods (Access=protected)
        % In step, increment the counter and return its value
        % as an output
        function c = stepImpl(obj)
            if obj.Increment
                obj.Count = obj.Count + 1;
            end
            c = obj.Count;
        end
        % Setup the Count state variable
        function setupImpl(obj)
            obj.Count = 0;
        end
        % Reset the counter to zero.
        function resetImpl(obj)
            obj.Count = obj.InitialValue;
        end
        % The step method takes no inputs
        function numIn = getNumInputsImpl(~)
            numIn = 0;
        end
    end
end
```

More About

- "Class Attributes"
- "What are System Object Locking and Property Tunability?"
- "Methods Timing" on page 11-36

Hide Inactive Properties

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

Hide an inactive property

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

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

Complete Class Definition File with Hidden Inactive Property

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

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

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

  methods (Access=protected)
```

```
        % In step, increment the counter and return its value
        % as an output
        function c = stepImpl(obj)
          obj.pCount = obj.pCount + 1;
          c = obj.pCount;
        end

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

        % The step method takes no inputs
        function numIn = getNumInputsImpl(~)
          numIn = 0;
        end

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

See Also `isInactivePropertyImpl` |

Limit Property Values to a Finite Set of Strings

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

Specify a Set of Valid String Values

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

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

```
properties
   Color = 'blue'
end

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

Complete Class Definition File with String Set

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

   properties
     Color = 'blue'
   end

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

```
    methods(Access = protected)
      function stepImpl(obj)
          h = Whiteboard.getWhiteboard();
          plot(h, ...
            randn([2,1]), randn([2,1]), ...
            'Color', obj.Color(1));
      end
      function releaseImpl(obj)
          cla(Whiteboard.getWhiteboard());
          hold('on');
      end
      function n = getNumInputsImpl(~)
           n = 0;
      end
      function n = getNumOutputsImpl(~)
          n = 0;
      end
    end

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

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

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

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

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

%% Clear the whiteboard
hBlueInk.release();

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

See Also

`matlab.system.StringSet` |

More About

- “Enumerations”

Process Tuned Properties

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

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

Control When a Lookup Table Is Generated

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

```
methods  (Access = protected)
   function processTunedPropertiesImpl(obj)
      obj.pLookupTable = obj.MiddleC * ...
        (1+log(1:obj.NumNotes)/log(12));
   end
end
```

Complete Class Definition File with Tuned Property Processing

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

  properties
    MiddleC = 440
    NumNotes = 12
  end

  properties (Access=private)
    pLookupTable
  end

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

```
            obj.pLookupTable = obj.MiddleC * ...
               (1+log(1:obj.NumNotes)/log(12));
          end

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

          function processTunedPropertiesImpl(obj)
            % Generate a lookup table of note frequencies
            obj.pLookupTable = obj.MiddleC * ...
               (1+log(1:obj.NumNotes)/log(12));
          end
        end
      end
```

See Also

`processTunedPropertiesImpl` |

Release System Object Resources

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

Release Memory by Clearing the Object

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

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

Complete Class Definition File with Released Resources

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

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

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

```
        end

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

        function n = getNumInputsImpl(~)
            n = 0;
        end
        function n = getNumOutputsImpl(~)
            n = 0;
        end
    end

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

See Also

`isInactivePropertyImpl` |

Related Examples

- “Initialize Properties and Setup One-Time Calculations” on page 11-10

Define Composite System Objects

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

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the `pFir` and `pIir` property values.

```
properties (Nontunable, Access = private)
   pFir % store the FIR filter
   pIir % store the IIR filter
end

methods
   function obj = Filter(varargin)
      setProperties(obj, nargin, varargin{:});
      obj.pFir = FIR(obj.zero);
      obj.pIir = IIR(obj.pole);
   end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
%Filter System object with a single pole and a single zero
%
%   This System object illustrates composition by
%   composing an instance of itself.
%

    properties (Nontunable)
      zero = 0.01
      pole = 0.5
```

```
    end

    properties (Nontunable,Access=private)
      pZero % store the FIR filter
      pPole % store the IIR filter
    end

    methods
        function obj = Filter(varargin)
            setProperties(obj,nargin, varargin{:});
            % Create instances of FIR and IIR as
            % private properties
            obj.pZero = Zero(obj.zero);
            obj.pPole = Pole(obj.pole);
        end
    end

    methods (Access=protected)
        function setupImpl(obj,x)
            setup(obj.pZero,x);
            setup(obj.pPole,x);
        end

        function resetImpl(obj)
            reset(obj.pZero);
            reset(obj.pPole);
        end

        function y = stepImpl(obj,x)
            y = step(obj.pZero,x) + step(obj.pPole,x);
        end
        function releaseImpl(obj)
            release(obj.pZero);
            release(obj.pPole);
        end
    end
end
```

Class Definition File for FIR Component of Filter

```
classdef Pole < matlab.System

  properties
    Den = 1
  end

  properties (Access=private)
    tap = 0
  end

  methods
    function obj = Pole(varargin)
      setProperties(obj,nargin,varargin{:},'Den');
    end
  end

  methods (Access=protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Den;
      obj.tap = y;
    end
  end

end
```

Class Definition File for IIR Component of Filter

```
classdef Zero < matlab.System

  properties
    Num = 1
  end

  properties (Access=private)
    tap = 0
  end

  methods
    function obj = Zero(varargin)
      setProperties(obj, nargin,varargin{:},'Num');
```

```
    end
  end

  methods (Access=protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Num;
      obj.tap = x;
    end
  end

end
```

See Also

`nargin`

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

1. Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
   matlab.system.mixin.FiniteSource
```

2. Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
  function bDone = isDoneImpl(obj)
     bDone = obj.NumSteps==2
  end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
  %RunTwice System object that runs exactly two times
  %
  properties (Access=private)
    NumSteps
  end

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

    function y = stepImpl(obj)
      if ~obj.isDone()
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
      else
```

```
            out = 0;
          end
        end

        function bDone = isDoneImpl(obj)
          bDone = obj.NumSteps==2;
        end
      end

      methods (Access=protected)
        function n = getNumInputsImpl(~)
          n = 0;
        end
        function n = getNumOutputsImpl(~)
          n = 1;
        end
      end

    end
```

See Also

`matlab.system.mixin.FiniteSource` |

More About

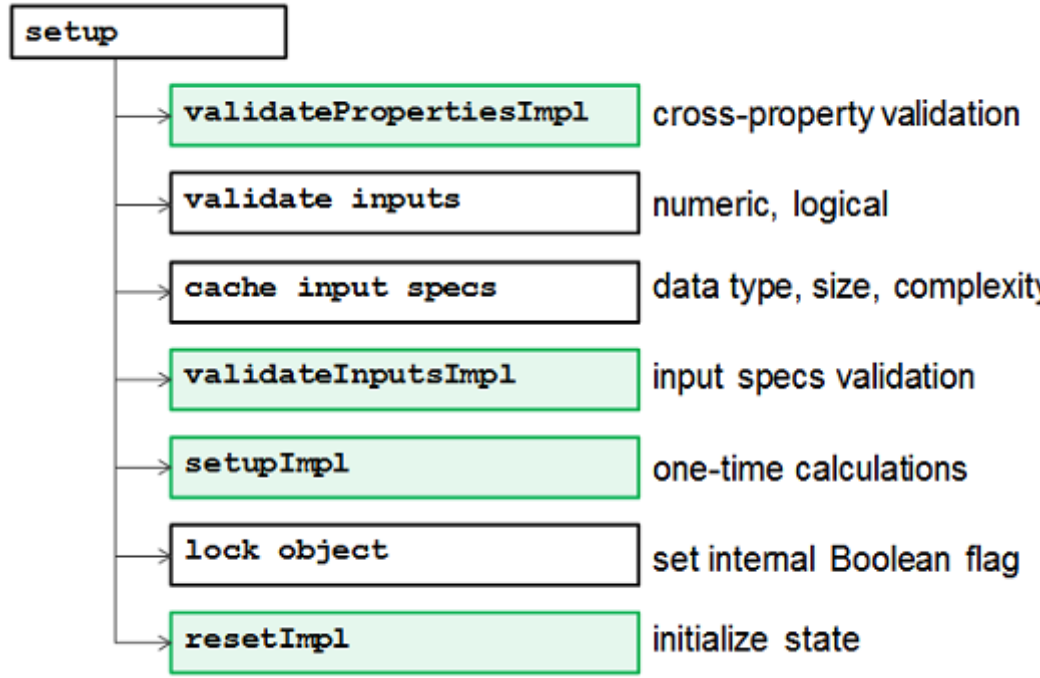
- "Subclassing Multiple Classes"

Methods Timing

In this section...

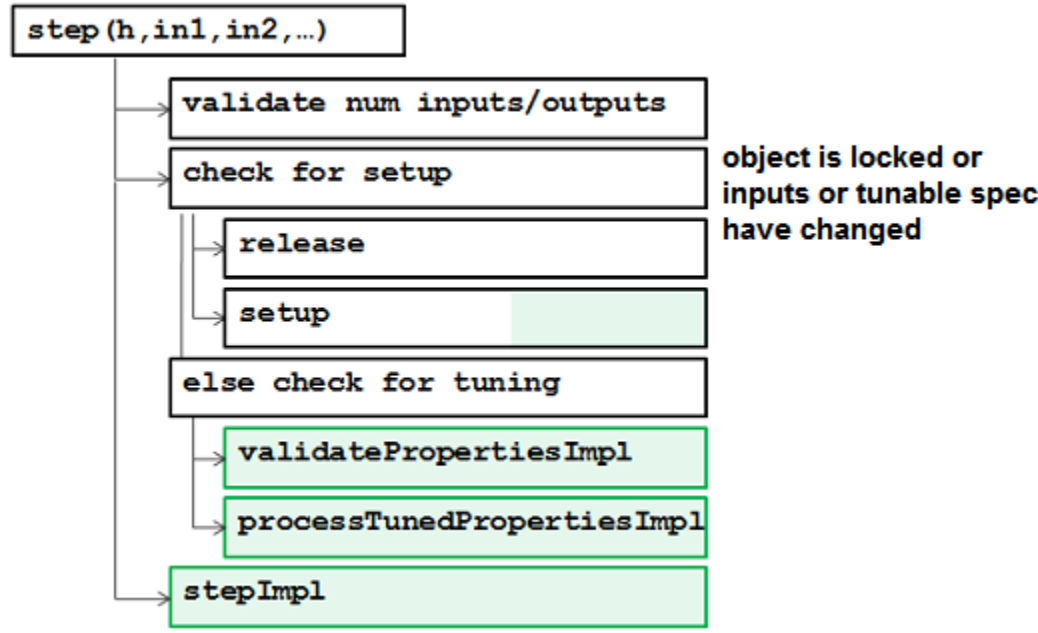
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.



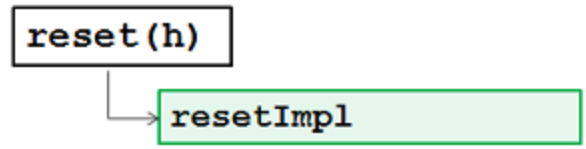
Step Method Call Sequence

This hierarchy shows the actions performed when you call the `step` method.



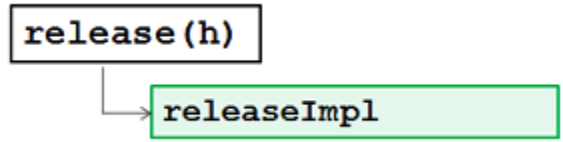
Reset Method Call Sequence

This hierarchy shows the actions performed when you call the `reset` method.



Release Method Call Sequence

This hierarchy shows the actions performed when you call the `release` method.



See Also

`setupImpl` | `stepImpl` | `releaseImpl` | `resetImpl` |

Related Examples

- "Release System Object Resources" on page 11-28
- "Reset Algorithm State" on page 11-16
- "Set Property Values at Construction from Name-Value Pairs" on page 11-13
- "Define Basic System Objects" on page 11-2

More About

- "What are System Object Methods?"
- "The Step Method"
- "Common Methods"