

Phased Array System Toolbox™ 1

Getting Started Guide

MATLAB®

How to Contact MathWorks



www.mathworks.com
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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™ Getting Started Guide

© COPYRIGHT 2011 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

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)

Getting Started with Phased Array System Toolbox Software

1

Overview	1-2
Product Overview	1-2
Conventions	1-3
Required Products	1-4
MATLAB® Compiler Support	1-4

Phased Array Systems

2

System Overviews	2-2
Phased Array System Overview	2-2
Radar Phased Array Overview	2-4

Radar Data Cube, Units, and Physical Constants

3

Radar Data Cube	3-2
Fast Time Samples	3-3
Slow Time Samples	3-4
Spatial Sampling	3-4
Space-Time Processing	3-5
Organizing Data in the Radar Data Cube	3-5
Units of Measure and Physical Constants	3-7
Units of Measure	3-7
Physical Constants	3-8

System Objects in the Phased Array System Toolbox

4

System Objects	4-2
What are System Objects?	4-2
Advantages of Using System Objects	4-2
Creating a System Object	4-3
Changing System Object Properties	4-3
Modes	4-5
Changing Properties While Running System Objects	4-5
Processing Data with System Objects	4-6
What are System Object Methods?	4-6
The Step Method	4-6
Step Method Examples	4-6
Common System Object Methods	4-8
Custom System Object Methods	4-9

Basic Radar Workflow

5

Overview of Basic Workflow	5-2
Building The Basic Radar Workflow Model	5-3

Getting Started with Phased Array System Toolbox Software

Overview

In this section...
“Product Overview” on page 1-2
“Conventions” on page 1-3
“Required Products” on page 1-4
“MATLAB® Compiler Support” on page 1-4

Product Overview

Phased Array System Toolbox™ provides algorithms and tools for the design, simulation, and analysis of phased array signal processing systems. These capabilities are provided as MATLAB® functions and MATLAB System objects. The system toolbox includes algorithms for waveform generation, beamforming, direction of arrival estimation, target detection, and space-time adaptive processing. The system toolbox lets you build monostatic, bistatic, and multistatic architectures for a variety of array geometries. You can model these architectures on stationary or moving platforms. Array analysis and visualization tools help you evaluate spatial, spectral, and temporal performance. The system toolbox lets you model an end-to-end phased array system or use individual algorithms to process acquired data.

Key features of this product include:

- Algorithms available as MATLAB functions and MATLAB System objects
- Monostatic, bistatic, and multistatic phased array system modeling
- Array analysis and 3D visualization; physical array modeling for uniform linear arrays, uniform rectangular arrays, and arbitrary conformal arrays on platforms with motion
- Broadband and narrowband digital beamforming functions, including MVDR/Capon, LCMV, time delay, Frost, time delay LCMV, and subband phase shift
- Space-time adaptive processing algorithms, including displaced phase center array (DPCA), adaptive DPCA, sample matrix inversion (SMI) beamforming, and angle-Doppler response visualization

- Direction of arrival algorithms, including MVDR, ESPRIT, Beamscan, Root MUSIC, and monopulse tracking
- Waveform synthesis functions for pulsed CW, linear FM, stepped FM, and staggered PRF signals, and waveform visualization tools for ambiguity function and matched filter response
- Algorithms for TVG, pulse compression, coherent and non-coherent integration, CFAR processing, plotting ROC curves, and estimating range and Doppler

Conventions

The Phased Array System Toolbox uses consistent conventions with respect to units of measure, data representations, and coordinate systems. Familiarity with these conventions will facilitate use of the toolbox functionality. The following sections provide brief descriptions of these conventions with links to more detailed explanations where needed.

Complex-valued Baseband Signals

In phased array signal processing, it is common to shift the frequency content of a waveform to support effective radiation and propagation in the medium. This is accomplished by modulating a *baseband* signal with nonzero spectral magnitudes in the vicinity of zero frequency to create a *bandpass* signal with nonzero spectral magnitudes centered around a carrier frequency. Typically, the bandwidth of the baseband signal is small compared to the carrier frequency resulting in a *narrowband* signal. To process returned signals, the receiver demodulates the bandpass signal to the baseband. The demodulation involves local oscillators both in phase and 90 degrees out of phase with the modulating carrier frequency. This results in in-phase (I) and quadrature (Q) baseband signals, or channels. For processing, it is convenient to create a complex-valued baseband signal by assigning the I channel to be the real part and the Q channel to be the imaginary part, $I+jQ$. The Phased Array System Toolbox uses the complex-valued baseband representation to represent both transmitted and received signals. While actual phased array systems transmit real-valued signals and create complex-valued baseband signals at the receiver, utilizing a complex-valued representation at all stages allows you to accurately model the impact of system gains, losses, and interference on the received signal samples.

Data Organization of Baseband Signals

Space-time processing of the complex-valued baseband samples is efficiently implemented by organizing the data in a three-dimensional matrix. See Radar Data Cube for an explanation of how the software organizes space-time data.

Spatial Coordinates

Representation of position in three dimensions is a fundamental aspect of array signal processing. The Phased Array System Toolbox specifies rectangular and spherical coordinates as column vectors with respect to both *global* and *local* origins. See “Coordinate Systems and Motion Modeling” for a detailed explanation of the conventions used in the toolbox.

Physical Quantities

The Phased Array System Toolbox almost exclusively uses the International System of Units (SI) for units of measure. In addition, there are physical constants declared and used in calculations. See “Units of Measure and Physical Constants” on page 3-7 for a detailed explanation of toolbox conventions.

Supported Data Types

The Phased Array System Toolbox only supports double-precision data types. Inputting a data type that is not double precision can produce an error or incorrect results.

Required Products

The Phased Array System Toolbox requires the Signal Processing Toolbox™ and the DSP System Toolbox™ software. You must install these products to use the Phased Array System Toolbox software.

MATLAB Compiler Support

Version 1.0 of the Phased Array System Toolbox software does not support the MATLAB® Compiler™. You cannot compile any functionality in the toolbox.

Phased Array Systems

System Overviews

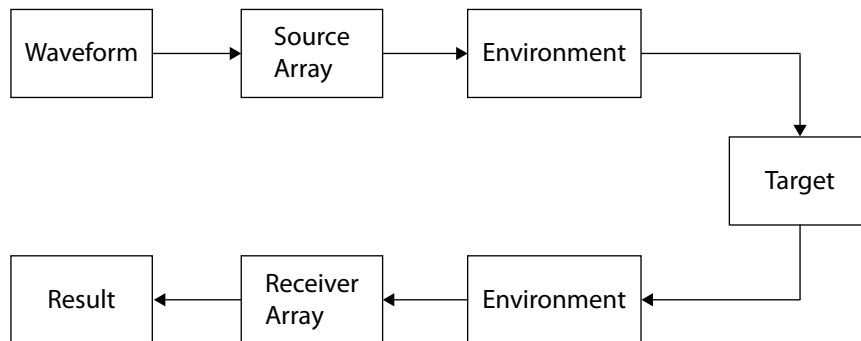
In this section...
“Phased Array System Overview” on page 2-2
“Radar Phased Array Overview” on page 2-4

Phased Array System Overview

Phased array systems use the spatial and temporal characteristics of propagating space-time wavefields to extract information about the source, or sources of the wavefields. By processing data collected over a spatiotemporal aperture using an array of sensors, you can significantly improve performance over a single sensor in a number of areas including, but not limited to:

- signal detectability
- spatial selectivity
- source identification and localization

The following figure shows a high-level overview of a phased array system:



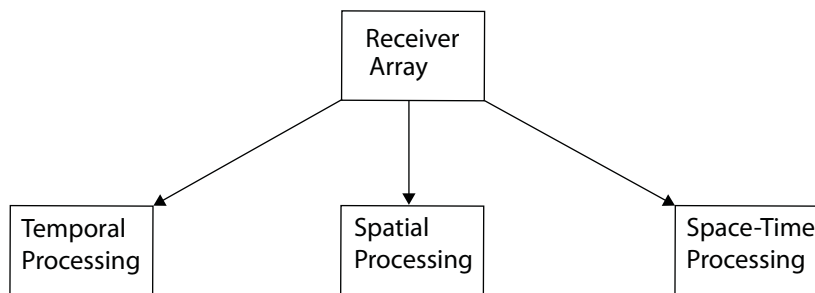
Phased array systems in diverse applications, such as radar, sonar, medical ultrasonography, medical imaging, and cellular phone communication share many common elements including:

- **Source Array** — The source array transmits a waveform through an environment. The waveform often consists of repeating pulses modulated

by a carrier frequency. Depending on the application, the wave may be an acoustic (mechanical), or electromagnetic wave. The source array is often electronically or mechanically steered to transmit in preferred directions.

- **Environment** — The medium in which the waveform travels to and from the target affects a number of system parameters including propagation speed, absorption loss, and wave dispersion.
- **Target** — The target reflects a portion of the incident waveform energy from the source array. Some percentage of the reflected energy is backscattered in the direction of the receiver array. In some applications, the target is the source of the waveform energy.
- **Receiver Array** — The receiver array collects energy from the target representing the *signal* along with external and internal sources of *noise*. The receiver implements algorithms to improve the signal-to-noise ratio and extract space-time information from the signal.

At the receiver, phased array systems implement algorithms to extract temporal and spatial information about the source, or sources of energy. The following figure shows a high-level overview of array signal processing algorithms common to a significant number of phased array systems.



Brief descriptions of the three categories are:

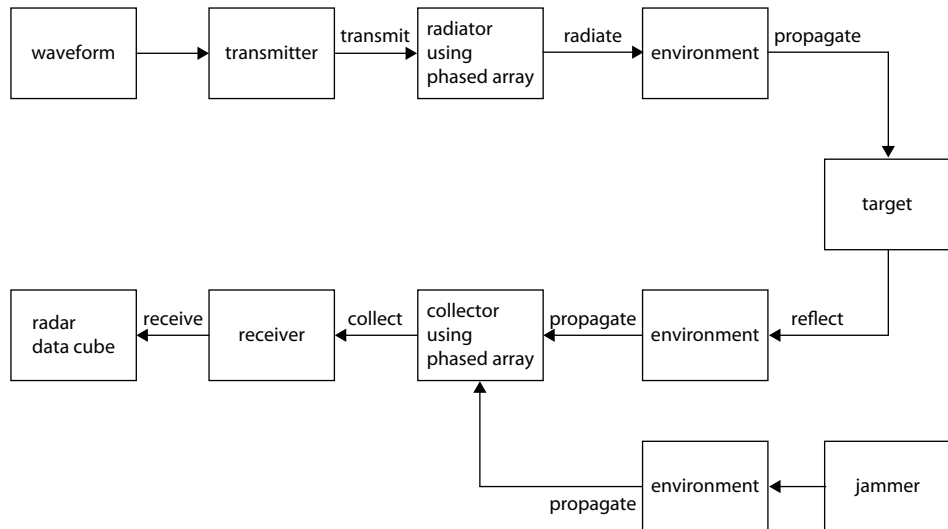
- **Temporal Processing** — Phased arrays often operate in poor signal-to-noise (SNR) ratios. Employing temporal integration and matched filtering improves the SNR. Knowing the propagation speed of the transmitted waveform and measuring the time it takes for a pulse to travel to and from a target allows phased array systems to estimate range.

Performing Fourier analysis on a time series of pulses enables the phased array to extract Doppler information from moving targets.

- **Spatial Processing** — Combining weighted information across multiple sensor elements with a known geometry enables phased array systems to spatially filter incoming waveforms. Phased arrays can also estimate the direction of arrival and the number of source waveforms incident on the array.
- **Space-Time Processing** — Simultaneously analyzing both spatial and temporal information enables phased array systems to produce joint angle-Doppler measurements of incident waveforms. Space-time processing enables phased array systems to distinguish moving targets from stationary targets when the phased array is in motion.

Radar Phased Array Overview

The following figure presents an overview of a radar phased array system. The figure is an expanded version of the high-level overview in “Phased Array System Overview” on page 2-2.



In order to exploit the advantages of array processing, it is important to understand how to model and optimize the performance of each component and operation in a phased array system. The Phased Array System Toolbox

provides models for all the components of the phased array system illustrated in the preceding figure from signal synthesis to signal analysis.

The toolbox supports models in which the transmitter and receiver are collocated or spatially separated. The toolbox also supports models in which both the targets and phased array are in motion.

Waveform Synthesis

The Phased Array System Toolbox supports the design of rectangular, linear frequency-modulated, and linear stepped-frequency pulsed waveforms with `phased.RectangularWaveform`, `phased.LinearFMWaveform`, and `phased.SteppedFMWaveform`.

Physical Components and Environment Modeling

The Phased Array System Toolbox enables you to simulate the physical components of a phased array system including:

- **Transmitter** — You can specify the transmitter peak power, gain, and loss factor. See `phased.Transmitter` for details.
- **Antenna elements** — You can create antenna elements with isotropic response patterns or antenna elements with user-specified response patterns over the entire range of azimuth ($[-180,180]$ degrees) and elevation ($[-90,90]$ degrees) angles. See `phased.IsotropicAntennaElement`, `phased.CosineAntennaElement`, and `phased.CustomAntennaElement` for details.
- **Microphone elements** — For acoustic applications, you can model an omnidirectional or custom microphone with `phased.OmnidirectionalMicrophoneElement` or `phased.CustomMicrophoneElement`.

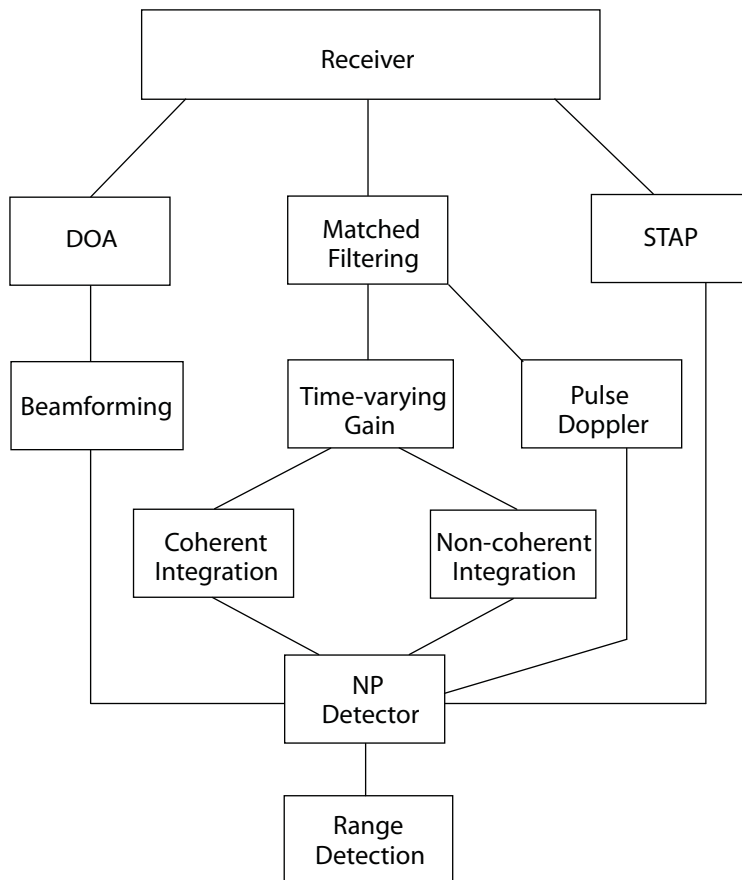
Phased arrays — There are System objects for three phased array geometries:

- **Uniform linear array (ULA)** — `phased.ULA` enables you to model a uniform linear array consisting of sensor elements with isotropic or custom radiation patterns. You can specify the number of elements and element spacing.

- Uniform rectangular array — `phased.URA` enables you to model a uniform rectangular array of sensor elements with isotropic or custom radiation patterns. You can specify the number of elements and element spacing along two orthogonal axes.
- Conformal array — `phased.ConformalArray` enables you to model a conformal array of sensor elements with isotropic or custom radiation patterns by specifying the antenna element positions and normal directions.
- **Radiator** — You can model waveform radiation through an antenna element, microphone, or array with the `phased.Radiator` object.
- **Environment**— You can model the propagation of an electromagnetic (EM) wave in free space with `phased.FreeSpace`. You can simulate one-way or two-way propagation of a narrowband EM signal by applying range-dependent attenuation and time delays, or phase shifts.
- **Target** — You can simulate a target with a specified radar cross section (RCS) using `phased.RadarTarget`. `phased.RadarTarget` supports both nonfluctuating and fluctuating (random) models of the RCS. The toolbox supports a family of random models based on the chi-square distribution known as *Swerling target models*.
- **Interference** — `phased.BarrageJammer` enables you to simulate wideband interference with a user-specified radiated power.
- **Signal collection** — You can simulate far-field or near-field narrowband and wideband signal reception from specified directions using `phased.Collector` and `phased.WidebandCollector`.
- **Receiver** — `phased.ReceiverPreamp` enables you to simulate the gain, loss factor, and internal noise characteristics of your receiver.

Array Signal Processing

For the processing of received data, the Phased Array System Toolbox supports a wide-range of array signal processing algorithms. The following figure presents a more detailed view of the general concepts discussed in “Phased Array System Overview” on page 2-2.



The preceding figure only presents an overview of the array signal processing operations supported by the Phased Array System Toolbox software. The figure does not purport to show predetermined orders of operation. For example, direction of arrival (DOA) estimation, beamforming, and space-time adaptive processing (STAP) often follow operations that improve the signal-to-noise ratio such as matched filtering. You can implement the supported algorithms in the manner best-suited to your application.

- **Matched Filtering** — You can perform matched filtering on your data with `phased.MatchedFilter`. See “Matched Filtering” for examples.

- **Time-varying gain** — You can equalize the power level of the incident waveform across samples from different ranges using `phased.TimeVaryingGain`. `phased.TimeVaryingGain` compensates for signal power loss due to range.
- **Beamforming and direction-of-arrival (DOA) estimation** — The Phased Array System Toolbox provides a number of algorithms for beamforming and direction of arrival estimation. See “Beamformers” and “Direction of Arrival (DOA)” for a list of supported beamforming and DOA algorithms. You can find example workflows for each in “Beamforming” and Direction of Arrival (DOA) Estimation.
- **Detection** — The Phased Array System Toolbox has a number of utility functions to implement and evaluate Neyman-Pearson detectors using both coherent and noncoherent pulse integration.

The toolbox also provides routines for evaluating detector performance through the construction of receiver operating characteristic curves.

To model fluctuating noise characteristics, `phased.CFARDetector` object adaptively estimates the noise characteristics from the data to maintain a constant false-alarm rate.

You can find example workflows in the “Detection” section of the User’s Guide.

- **Pulse Doppler** — The Phased Array System Toolbox has utility functions for estimating Doppler shift based on speed (`speed2dop`) and to estimate speed based on the Doppler shift (`dop2speed`). You can implement pulse-Doppler processing by using the spectrum estimation algorithms in the Signal Processing Toolbox on the slow-time data. See “Radar Data Cube” on page 3-2 for an explanation of the slow-time data.

See “Doppler Shift and Pulse-Doppler Processing” for examples of Doppler processing.

To calculate the joint angle-Doppler response of the input data, use `phased.AngleDopplerResponse`.

Example workflows for computing the angle-Doppler response can be found in “Angle-Doppler Response”.

- **Space-time adaptive processing** — You can implement displaced phase center antenna techniques with `phased.DPCACanceller` and `phased.ADPCACanceller`. `phased.STAPSMIBeamformer` implements an

adaptive beamformer by calculating the beamformer weights using the estimated space-time interference covariance matrix.

See “Space-Time Adaptive Processing (STAP)” for examples.

Radar Data Cube, Units, and Physical Constants

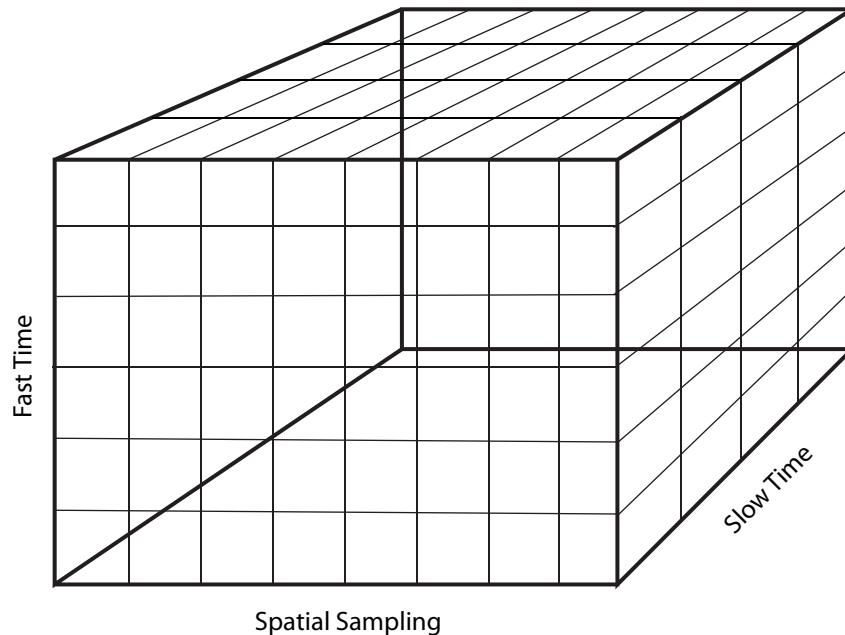
- “Radar Data Cube” on page 3-2
- “Units of Measure and Physical Constants” on page 3-7

Radar Data Cube

In this section...
“Fast Time Samples” on page 3-3
“Slow Time Samples” on page 3-4
“Spatial Sampling” on page 3-4
“Space-Time Processing” on page 3-5
“Organizing Data in the Radar Data Cube” on page 3-5

The radar data cube is a convenient way to conceptually represent space-time processing. To construct the radar data cube, assume that pre-processing converts the RF signals received from multiple pulses across multiple array elements to complex-valued baseband samples. Arrange the complex-valued baseband samples in a three-dimensional M -by- N -by- L matrix. Many of the radar signal processing operations supported by the Phased Array System Toolbox correspond to processing one-dimensional subvectors, or two-dimensional submatrices of the radar data cube.

The following figure shows the organization of the radar data cube in the Phased Array System Toolbox. Subsequent sections explain each of the dimensions and which aspect of space-time processing they represent.



Fast Time Samples

Consider each M -by-1 subvector of the radar data cube. Each of these M -by-1 column vectors represent complex-valued baseband samples from one pulse at one array element. Because pulse bandwidths can be on the order of a few hundred kHz, you require high sampling rates to avoid aliasing. This is the basis for the designation *fast time*.

The fast time dimension is also referred to as the *range* dimension. If the two-way range to a target from the phased array is $2R$, the difference in range, ΔR , represented by two samples acquired with sampling interval, T is:

$$\Delta R = \frac{cT}{2}$$

Each sample in the fast time dimension represents an incremental change in range of ΔR in range. For this reason, fast time samples are also referred to as *range bins*, or *range gates*.

Pulse compression is an example of a signal processing operation performed on the fast time samples.

Slow Time Samples

Consider each M -by- L submatrix of the radar data cube. In the submatrix there are M row vectors with dimension 1-by- L . Each of these row vectors contains complex-valued baseband samples from L different pulses corresponding to the same range bin. There is a M -by- L matrix for each of the N array elements. The sampling interval between the L samples is the *pulse repetition interval* (PRI). Typical PRIs are much longer than the fast-time sampling interval. This is the motivation for designating samples taken across multiple pulses as *slow time*.

Processing data in the slow time dimension allows you to estimate the Doppler spectrum at a given range bin.

The Nyquist criterion applies equally to the slow-time dimension. The reciprocal of the PRI is the *pulse repetition frequency* (PRF). The PRF gives the width of the unambiguous Doppler spectrum.

Spatial Sampling

Phased arrays consist of multiple array elements. Consider each M -by- N submatrix of the radar data cube. Each column vector consists of M fast-time samples for a single pulse received at a single array element. The N column vectors represent the same pulse sampled across N array elements. The sampled data in the N column vectors is a spatial sampling of the incident waveform. Analysis of the data across the array elements allows you to examine the spatial frequency content of each received pulse.

It is also possible to spatially sample a wavefield by mechanically steering a single antenna, but the more common scenario is to sample the wavefield by multiple array elements. The Nyquist criterion for spatial sampling dictates that array elements must not be separated by more than one half the wavelength of the carrier frequency.

Beamforming is a spatial filtering operation that combines data across the array elements to selectively enhance and suppress wavefields incident on the array from particular directions.

Space-Time Processing

Space-time adaptive processing operates on the two-dimensional angle-Doppler data for each range bin. Consider the M -by- N -by- L radar data cube. Each of the M samples is data from the same range. This range is sampled across N array elements, and L PRIs. Collapsing the three-dimensional matrix at each range bin into N -by- L submatrices allows the simultaneous two-dimensional analysis of angle of arrival and Doppler frequency.

Organizing Data in the Radar Data Cube

If you have M complex-valued baseband data samples collected from L pulses received at N sensors, you can organize your data in a format compatible with the Phased Array System Toolbox conventions using `permute`. After processing your data, you can convert back to your original data cube format with `ipermute`.

Reordering the Data Cube

Assume you have a data set consisting of 200 samples per pulse for ten pulses collected at 6 sensor elements. Assume that your data are organized as a 6-by-10-by-200 matrix. Simulate this data structure using complex-valued white Gaussian noise samples.

```
OrigData = randn(6,10,200)+1j*randn(6,10,200);
```

The first dimension of `OrigData` is the number of sensors (spatial sampling), the second dimension is the number of pulses (slow-time), and the third dimension contains the fast-time samples. This format is not compatible with the radar data cube conventions of the Phased Array System Toolbox.

The Phased Array System Toolbox expects the first dimension to contain the fast-time samples, the second dimension to represent individual sensors in the array, and the third dimension to contain the slow-time samples.

To reorganize `OrigData` in a format compatible with the toolbox conventions, enter:

```
NewData = permute(OrigData,[3 1 2]);
```

The preceding line of code moves the third dimension of `OrigData` to be the first dimension of `NewData`. The first dimension of `OrigData` becomes the second dimension of `NewData` and the second dimension of `OrigData` becomes the third dimension of `NewData`. This results in `NewData` being organized as *fast-time samples-by-sensors-by-slow-time samples*. You can now process `NewData` with the Phased Array System Toolbox software.

After you process your data, you can use `ipermute` to return your data format to the original structure.

```
Data = ipermute(NewData,[3 1 2]);  
% Data is equal to OrigData
```


Units of Measure and Physical Constants

In this section...

“Units of Measure” on page 3-7

“Physical Constants” on page 3-8

Units of Measure

The Phased Array System Toolbox almost exclusively uses SI base and derived units to measure physical quantities. The following table gives the physical quantity and corresponding SI base and derived units used in the software.

Physical Quantity	SI Base/Derived Unit	Abbreviation
Frequency	hertz	Hz
Length	meter	m
Power	watt	w
Time	second	s
Temperature	kelvin	k

The Phased Array System Toolbox does not provide any utilities for converting the preceding SI base or derived units to other systems of measurement.

Angles

Angles are an exception to the use of SI base and derived units. All angles in the Phased Array System Toolbox software are specified in degrees. See “Rectangular and Spherical Coordinates” for an explanation of the angles used in the software. There are two utility functions for converting angles from radians to degrees and degrees to radians: `radtodeg` and `degtorad`.

Decibels

In order to accurately model and simulate phased array systems, it is necessary to account for gains and losses in power incurred at various stages of processing. In the Phased Array System Toolbox software, these gains and

losses are specified in decibels (dB). Signal to noise ratios (SNRs) and the receiver noise figure are also expressed in dB. A power of P watts in dB is:

$$10\log_{10}(P)$$

There are two utility functions for converting between dB and power: `db2pow` and `pow2db`, and two utility functions for converting between magnitude and dB: `db2mag` and `mag2db`.

Physical Constants

In modeling and simulating phased array systems, you require values for a number of physical constants. For example, the distribution of thermal noise power per unit bandwidth depends on the Boltzmann constant. To measure Doppler shift and range in radar, you have to specify a value for the speed of light. The following table summarizes the three physical constants specified in the toolbox. See `physconst` for additional information.

Description	Value
The speed of light in a vacuum	299,792,458 meters/second. Most commonly denoted by c .
The Boltzmann constant relating energy to temperature.	1.38×10^{-23} joules/degree kelvin. Most commonly denoted by k .
Mean radius of the Earth	6,137,000 meters

System Objects in the Phased Array System Toolbox

- “System Objects” on page 4-2
- “Processing Data with System Objects” on page 4-6

System Objects

In this section...

“What are System Objects?” on page 4-2

“Advantages of Using System Objects” on page 4-2

“Creating a System Object” on page 4-3

“Changing System Object Properties” on page 4-3

“ Modes” on page 4-5

“Changing Properties While Running System Objects” on page 4-5

What are System Objects?

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets. For general information on MATLAB objects, see *Object-Oriented Programming*. For a brief description of the advantages of using System objects over procedural functions see “Advantages of Using System Objects” on page 4-2.

Advantages of Using System Objects

System objects use a minimum of two commands to process data: a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable instances of an object, each with different settings. Also, using the System object API avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

Creating a System Object

To use a System object, you must first create the object. To create an object, use the syntax:

```
ObjectHandle = phased.ObjectName
```

For example, to create a `phased.LinearFMWaveform` object, enter:

```
HLFMwav = phased.LinearFMWaveform
```

HLFMwav is an object handle. System objects are *handle class* objects. For more detailed information on handle class objects, see “Comparing Handle and Value Classes”.

Changing System Object Properties

Each System object has one or more properties, which determine how the object operates. To view the property names and their current values, enter the object handle at the command prompt without a semicolon. The object property names and values are also displayed in the command window when you create a System object without a terminating semicolon.

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

When you create an object using `ObjectHandle = phased.ObjectName`, the properties are assigned their default values. If a property is not read-only, you can modify the property values. To modify property values, you can either:

- Change the property values after construction using the `ObjectHandle.Property` syntax.
- Assign the property values at construction using name-value pairs. You can enter name-value pairs in any order.

- Assign the property values using value-only arguments at construction in the specified order. The reference page for each object describes which value-only arguments, if any, the object supports.

The following examples create `phased.LinearFMWaveform` objects with the same property values.

Change property values after construction.

```
HLFMwav = phased.LinearFMWaveform
% Change the value of the SweepBandwidth property
HLFMwav.SweepBandwidth = 2e5
% Change the value of the SweepDirection property
HLFMwav.SweepDirection = 'down'
```

Use name-value pairs at construction.

```
HLFMwav = phased.LinearFMWaveform('SweepBandwidth',2e5,...
'SweepDirection','down');
```

The following code creates `phased.URA` objects with the same property values.

Change property values after construction.

```
hURA = phased.URA
hURA.Size = [2 3]
hURA.ElementSpacing = 0.25
```

Use name-value pairs at construction.

```
hURA = phased.URA('Size',[2 3],'ElementSpacing',0.25)
% equivalent to
% hURA = phased.URA('ElementSpacing',0.25,'Size',[2 3]);
```

Use value-only arguments in a supported order. See `phased.URA` for a description of the value-only syntax.

```
hURA = phased.URA([2 3],0.25)
```

To obtain command-line help on an object property, enter `help phased.ObjectName/PropertyName` at the MATLAB command prompt. For example:

```
help phased.LinearFMWaveform/SweepBandwidth
```

System object properties are also documented on the class reference page in the help documentation. See `phased.LinearFMWaveform` for an example.

Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an instance of an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

When the object begins processing data, it initializes and is locked. The typical way in which an object becomes locked is when the `step` method is called on that object. To determine if an object is locked, use the `isLocked` method. When the object is locked, you cannot change the number of inputs, the number of outputs, or the value of any property unless the property is *tunable*. These restrictions allow the object to maintain states and allocate memory appropriately. See “Changing Properties While Running System Objects” on page 4-5 for information on tunable and nontunable properties.

Changing Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding object reference page or use a command of the form:

```
help phased.FrostBeamformer/DiagonalLoadingFactor
```

where `DiagonalLoadingFactor` is the property name. For information on locked and unlocked modes, see “Modes” on page 4-5.

Changing the data type or dimension of a nontunable property value is not allowed without first calling the `release` method.

Processing Data with System Objects

In this section...

“What are System Object Methods?” on page 4-6

“The Step Method” on page 4-6

“Step Method Examples” on page 4-6

“Common System Object Methods” on page 4-8

“Custom System Object Methods” on page 4-9

What are System Object Methods?

After you create a System object and assign property values, you use object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `methodName(ObjectHandle, ...)`, such as `step(H, ...)` where the ellipsis denotes a place holder for additional input arguments. For example, see the reference page for the `plotResponse` method of the uniform linear array, `phased.ULA`.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by the object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the `step` method and other available methods, see the descriptions in “Common System Object Methods” on page 4-8.

Step Method Examples

This section presents examples of using the `step` method. The action of the `step` method differs for each object. See the object reference page for a detailed description.

Calculate the steering vector for a uniform linear array

Construct a uniform linear array (ULA) consisting of 8 elements spaced 0.5 meters apart.

```
hULA = phased.ULA(8,0.5);
```

Construct the steering vector for the specified ULA. First construct the steering vector object with the ULA as the value of the `SensorArray` property.

```
hSV = phased.SteeringVector('SensorArray',hULA,...
    'PropagationSpeed',physconst('lightspeed'));
```

Calculate the steering vector assuming an operating frequency of 1 GHz and azimuth and elevation angles of 45 degrees.

```
SteerVec = step(hSV,1e9,[45 45]);
```

Calculate the effect of propagating a signal in free space

This example uses two different step methods. The first step method is associated with the `phased.LinearFMWaveform` object and the second step method is associated with the `phased.Freespace` object.

Construct a linear FM waveform with a pulse duration of 50 microseconds, a sweep bandwidth of 100 kHz, an increasing instantaneous frequency, and a pulse repetition frequency (PRF) of 10 kHz..

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

Obtain the waveform using the step method. Note that the input to the step method is a handle to a `phased.LinearFMWaveform` object.

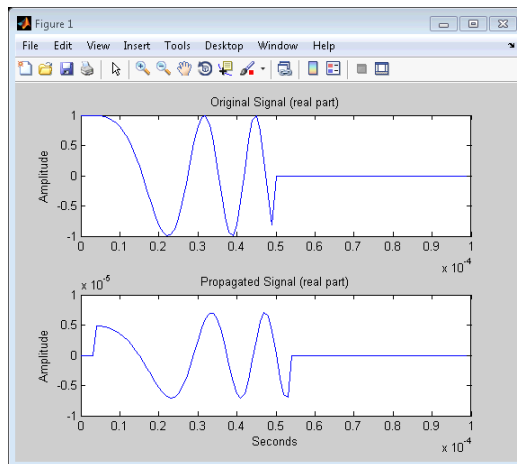
```
Sig = step(hFM);
```

Construct a free space object with a propagation speed equal to the speed of light, an operating frequency of 3 GHz, and a sample rate of 1 MHz. The free space object is constructed to model one way propagation.

```
hFS = phased.FreeSpace('PropagationSpeed',physconst('lightspeed'),...
    'OperatingFrequency',3e9,'TwoWayPropagation',false,'SampleRate',1e6);
```

Calculate the effect on the waveform of one-way propagation in free space from coordinates $[0;0;0]$ to $[500; 1e3; 20]$ and plot the results for comparison.

```
PropSig = step(hFS,Sig,[0; 0; 0],[500; 1e3; 20]);  
% compare the original signal to the propagated waveform  
t = unigrd(0,1/hFS.SampleRate,length(Sig)*1/hFS.SampleRate, '[]');  
subplot(211)  
plot(t,real(Sig)); title('Original Signal (real part)');  
ylabel('Amplitude');  
subplot(212)  
plot(t,real(PropSig)); title('Propagated Signal (real part)');  
xlabel('Seconds'); ylabel('Amplitude');
```



Common System Object Methods

All Phased Array System Toolbox System objects have the following methods, each of which is described in a method reference page associated with each specific object.

Method	Description
<code>step</code>	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the <code>step</code> method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The <code>step</code> method returns regular MATLAB variables. Example: <code>Y = step(H,X)</code>
<code>release</code>	<code>release</code> releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. See “Modes” on page 4-5.
<code>clone</code>	Creates another object with the same property values
<code>isLocked</code>	Returns a logical value indicating whether the object is locked. See “Modes” on page 4-5.
<code>getNumInputs</code>	Returns the number of inputs expected by the <code>step</code> method. This number varies for an object depending on whether any properties enable additional inputs.
<code>getNumOutputs</code>	Returns the number of outputs expected from the <code>step</code> method. This number varies for an object depending on whether any properties enable additional outputs.

Additionally, select System objects have a `reset` method. For example, both `phased.Platform` and `phased.RadarTarget` have `reset` methods.

Custom System Object Methods

In addition to common System object methods, each System object in the Phased Array System Toolbox may have one or more methods specific to its functionality. You can find these methods listed on the object reference page with links to the method reference page. You can view command-line help for these objects by entering `help phased.ObjectName.method`. Examples include:

The `plotResponse` method for `phased.URA`.

```
hURA = phased.URA([4,4],0.5); %Construct URA
plotResponse(hURA,3e8,physconst('lightspeed'));
```

The `collectPlaneWave` method for `phased.ULA`.

```
hULA = phased.ULA(8,0.5); %Construct ULA
Fc = 3e8; % Carrier Frequency 300 MHz
t = (0:1e-3:1)';
Sig = cos(2*pi*200*t); %Incident signal
% Obtain the plane wave response at the 8 array elements
RecSig = collectPlaneWave(hULA,Sig,[10;25],Fc);
```

The `bandwidth` and `plot` methods for `phased.LinearFMWaveform`

```
HLFMwav = phased.LinearFMWaveform('SampleRate',50e6,'PulseWidth',1e-6,'S
BWidth = bandwidth(HLFMwav);
plot(HLFMwav)
```

Basic Radar Workflow

- “Overview of Basic Workflow” on page 5-2
- “Building The Basic Radar Workflow Model” on page 5-3

Overview of Basic Workflow

The scenario and code examples contained in “Building The Basic Radar Workflow Model” on page 5-3 are intended as an introduction to the fundamental workflow utilized in the Phased Array System Toolbox. The example is intentionally simplified in order to familiarize you with the basic theme that extends throughout the toolbox. You will find the core elements of this workflow in all the demos, reference page examples, and *Phased Array System Toolbox User’s Guide* examples.

The basic workflow consists of:

- constructing objects that represent the physical components and algorithms of your model. The objects have modifiable properties that enable you to parameterize your model. The object properties are described in both the command line help and on the object reference page.
- using the object’s `step` method to perform the action of your parameterized object on inputs. The action of `step` is specific to each algorithm. For example, the `step` method for the linear FM waveform, `phased.LinearFMWaveform`, performs a different action than the `step` method for the steering vector, `phased.SteeringVector`. The specific action and syntax of each `step` method are documented in the command line help and on the reference page. You can access the documentation for an object’s `step` method by entering:

```
>>doc phased.ObjectName/step
```

at the MATLAB command prompt, or via the hyperlink in the `Methods` section of the object’s reference page.

Building The Basic Radar Workflow Model

Basic Radar Workflow Scenario

The basic toolbox workflow is illustrated with the following scenario: Assume you have a single isotropic antenna operating at 4 GHz. Assume the antenna is located at the origin of your global coordinate system. There is a target with a nonfluctuating radar cross section of 0.5 square meters initially located at [7000; 5000; 0]. The target moves with a constant velocity vector of [-15; -10; 0]. Your antenna transmits ten rectangular pulses with a duration of 1 microsecond at a pulse repetition frequency (PRF) of 5 kHz. The pulses propagate to the target, reflect off the target, propagate back to the antenna, and are collected by the antenna. The antenna operates in a monostatic mode, receiving only when the transmitter is inactive.

Waveform

To build the waveform described in Basic Radar Workflow Scenario on page 5-3, use `phased.RectangularWaveform` and set the properties to the desired values.

```
hwav = phased.RectangularWaveform('PulseWidth',1e-6,'PRF',5e3,...  
    'OutputFormat','Pulses','NumPulses',1)
```

See “Waveforms” for more detailed examples on building waveform models.

Antenna

To model the antenna described in Basic Radar Workflow Scenario on page 5-3, use `phased.IsotropicAntennaElement`. Set the operating frequency range of the antenna to [1,10] GHz. The isotropic antenna radiates equal energy for azimuth angles from -180 to 180 degrees and elevation angles from -90 to 90 degrees.

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

Target Model

To model the target described in Basic Radar Workflow Scenario on page 5-3, use `phased.RadarTarget`. The target has a nonfluctuating RCS of 0.5 square

meters and the waveform incident on the target has a carrier frequency of 4 GHz. The waveform reflecting off the target propagates at the speed of light. Parameterize this information in defining your target.

```
htgt = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',0.5,...  
    'PropagationSpeed',physconst('lightspeed'),...  
    'OperatingFrequency',4e9)
```

Antenna and Target Platforms

To model the location and movement of the antenna and target in Basic Radar Workflow Scenario on page 5-3, use `phased.Platform`.

The antenna is stationary in this scenario and is located at the origin of the global coordinate system. See “Global and Local Coordinate Systems” for definitions and conventions regarding *global* and *local* coordinates.

The target is initially located at [7000; 5000; 0] and moves with a constant velocity vector of [-15; -10; 0].

```
% Antenna location and velocity  
htxplat = phased.Platform('InitialPosition',[0;0;0],...  
    'Velocity',[0;0;0],'OrientationAxes',[1 0 0;0 1 0;0 0 1])  
% Target location and velocity  
htgtplat = phased.Platform('InitialPosition',[7000; 5000; 0],...  
    'Velocity',[-15;-10;0])
```

Use `rangeangle` to determine the range and angle between the antenna and the target.

```
[tgtrng,tgtang] = rangeangle(htgtplat.InitialPosition,htxplat.InitialPos
```

See “Motion Modeling in Phased Array Systems” for more details on modeling motion.

Modeling the Transmitter

To model the transmitter specifications, use `phased.Transmitter`. A key parameter in modeling a transmitter is the peak transmit power. To determine the peak transmit power, assume that the desired probability of

detection is 0.9 and the maximum tolerable false-alarm probability is 10^{-6} . Assume that the ten rectangular pulses are noncoherently integrated at the receiver. You can use `albersheim` to determine the required signal-to-noise ratio (SNR).

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

The required SNR is approximately 5 dB. Assume you want to set the peak transmit power in order to achieve the required SNR for your target at a range of up to 15 km. Assume that the transmitter has a 20 dB gain. You can use `radareqpow` to determine the required peak transmit power.

```
maxrange = 1.5e4;
lambda = physconst('lightspeed')/4e9;
tau = hwav.PulseWidth;
Pt = radareqpow(lambda,maxrange,SNR,tau,'rcs',0.5,'Gain',20)
```

The required peak transmit power is approximately 45 kilowatts. To be conservative, use a peak power of 50 kilowatts in modeling your transmitter. To maintain a constant phase in the pulse waveforms, set the `CoherentOnTransmit` property to `true`. Because you are operating the transmitter in a monostatic (transmit-receive) mode, set the `InUseOutputPort` property to `true` to keep a record of the transmitter status.

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

See “Transmitter” for more examples on modeling transmitters and “Radar Equation” for examples involving the radar equation.

Modeling Waveform Radiation and Collection

To model waveform radiation from the array, use `phased.Radiator`. To model narrowband signal collection at the array, use `phased.Collector`. For wideband signal collection, use `phased.WidebandCollector`.

In this example, the pulse satisfies the narrowband assumption around the carrier frequency of 4 GHz. For the value of the `Sensor` property, use

the handle for the isotropic antenna. In `phased.Collector`, setting the `Wavefront` property to `'Plane'` assumes the waveform incident on the antenna is a plane wave.

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

Modeling the Receiver

To model the receiver in Basic Radar Workflow Scenario on page 5-3, use `phased.ReceiverPreamp`. In the receiver, you specify the noise figure and reference temperature, which are key contributors to the internal noise of your system. In this example, set the noise figure to 2 dB and the reference temperature to 290 degrees kelvin. Seed the random number generator for reproducible results.

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

See “Receiver Preamp” for more details.

Modeling the Propagation Environment

To model the propagation environment in Basic Radar Workflow Scenario on page 5-3, use `phased.FreeSpace`. You can model one-way and two-propagation by setting the `TwoWayPropagation` property. In this example, set this property to `false` to model one-way propagation.

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

See “Free Space Path Loss” for more details.

Implementing the Basic Radar Model

Having parameterized all the necessary components for the model outlined in Basic Radar Workflow Scenario on page 5-3, you are ready to generate the pulses, propagate the pulses to and from the target, and collect the echoes.

The following code prepares for the main simulation loop.

```
% 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);
```

You can execute the main simulation loop with the following code:

```
for n = 1:numpulses
    % Update the target position
    tgtpos = step(htgtplat,T);
    % Get the range and angle to the target
    [tgrng,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
```

Noncoherently integrate the received echoes, create a vector of range gates, and plot the result. The red vertical line on the plot marks the range of the target.

```
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');
```

