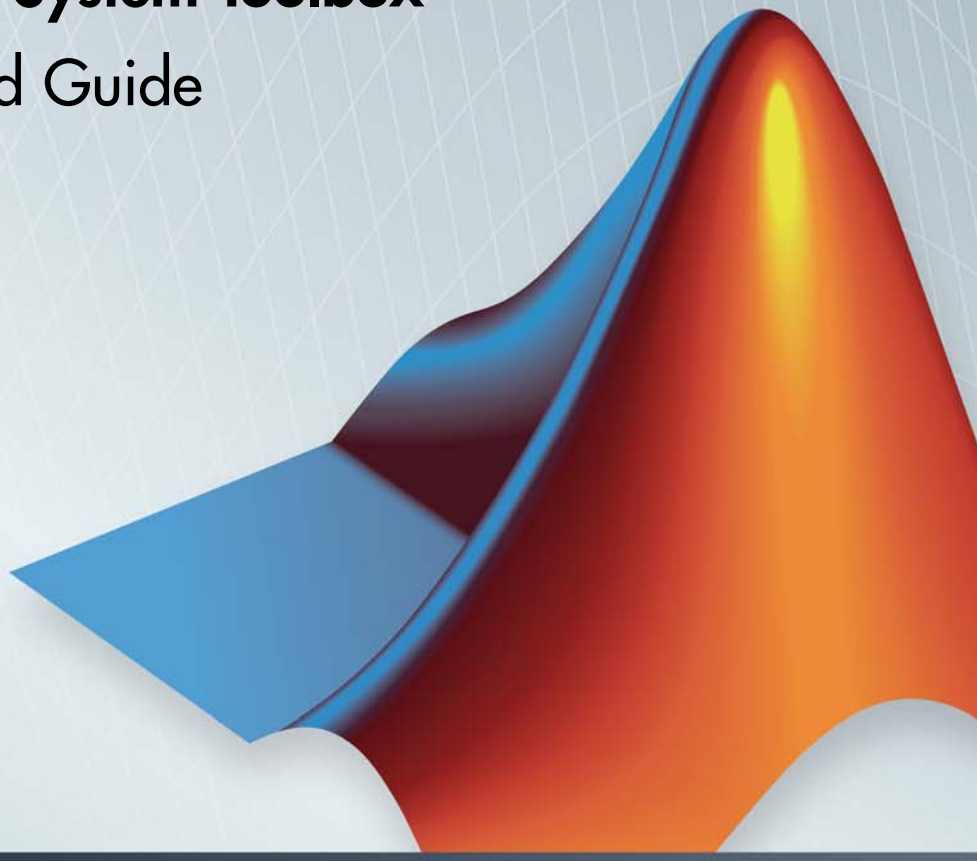
Phased Array System Toolbox™

Getting Started Guide

R2013a



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™ Getting Started 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	New for Version 1.0 (R2011a)
September 2011	Online only	Revised for Version 1.1 (R2011b)
March 2012	Online only	Revised for Version 1.2 (R2012a)
September 2012	Online only	Revised for Version 1.3 (R2012b)
March 2013	Online only	Revised for Version 2.0 (R2013a)

Contents

1 Getting Started with Phased Array System Toolbox Software

Product Description 1-2
Key Features 1-2

Limitations 1-3
MATLAB Compiler Support 1-3
Code Generation Support 1-3

Standards and Conventions 1-4
Scope of Standards and Conventions 1-4
Complex-Valued Baseband Signals 1-4
Data Organization of Baseband Signals 1-5
Spatial Coordinates 1-5
Physical Quantities 1-5
Supported Data Types 1-5

2 Phased Array Systems

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

3 Radar Data Cube, Units, and Physical Constants

Radar Data Cube 3-2

Radar Data Cube Concept 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 3-7

4 System Objects

What Is a System Toolbox? 4-2

What Are System Objects? 4-3

When to Use System Objects Instead of MATLAB Functions 4-5
System Objects vs. MATLAB Functions 4-5
Process Audio Data Using Only MATLAB Functions Code 4-5
Process Audio Data Using System Objects 4-6

System Design and Simulation in MATLAB 4-8

System Objects in Simulink 4-9

System Object Methods 4-10
What Are System Object Methods? 4-10
The Step Method 4-10
Common Methods 4-12

System Design Using System Objects 4-14
Create Components for Your System 4-14
Configure Components for Your System 4-15

Assemble Components to Create Your System 4-16
Run Your System 4-18
Reconfigure Your System During Runtime 4-18

5 Basic Radar Workflow

Overview of Basic Workflow 5-2

End-to-End Radar System 5-3
Radar Scenario 5-3
Waveform 5-3
Antenna 5-4
Target Model 5-4
Antenna and Target Platforms 5-4
Modeling the Transmitter 5-5
Modeling Waveform Radiation and Collection 5-6
Modeling the Receiver 5-6
Modeling the Propagation Environment 5-7
Implementing the Basic Radar Model 5-7

1

Getting Started with Phased Array System Toolbox Software

- "Product Description" on page 1-2
- "Limitations" on page 1-3
- "Standards and Conventions" on page 1-4

Product Description

Design and simulate phased array signal processing systems

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 apps 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

- Monostatic and multistatic radar system modeling, including point targets, free-space propagation, surface clutter, and barrage jammer
- Modeling of sensor arrays and subarrays with arbitrary geometries
- Polarization and platform motion specification for arrays and targets
- Synthesis and analysis of continuous and pulsed waveforms
- Broadband and narrowband digital beamforming, including MVDR/Capon, LCMV, time delay, Frost, and subband phase shift
- Direction of arrival algorithms, including monopulse, beamscan, MVDR, root MUSIC, and ESPRIT
- Algorithms for TVG, pulse compression, coherent and noncoherent integration, CFAR processing, plotting ROC curves, and estimating range and Doppler
- Space-time adaptive processing (STAP), including sample matrix inversion (SMI), and angle-Doppler response visualization

Limitations

In this section...
“MATLAB® Compiler™ Support” on page 1-3
“Code Generation Support” on page 1-3

MATLAB Compiler Support

Phased Array System Toolbox supports the MATLAB Compiler™ for all functions and System objects. Compiler support does not extend to any of the toolbox apps.

Code Generation Support

Phased Array System Toolbox software does not support automatic generation of C code. You cannot generate code from the functions or System objects in the toolbox.

Standards and Conventions

In this section...

"Scope of Standards and Conventions" on page 1-4

"Complex-Valued Baseband Signals" on page 1-4

"Data Organization of Baseband Signals" on page 1-5

"Spatial Coordinates" on page 1-5

"Physical Quantities" on page 1-5

"Supported Data Types" on page 1-5

Scope of Standards and Conventions

Phased Array System Toolbox software uses consistent conventions with respect to units of measure, data representations, and coordinate systems. You must understand these conventions to use the toolbox.

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. You accomplish this task 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 demodulation 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, $I+jQ$.

This software uses the complex-valued baseband representation to represent both transmitted and received signals. Actual phased array systems transmit real-valued signals and create complex-valued baseband signals only at the receiver. However, you can use a complex-valued representation at all stages.

Doing so enables you to accurately model the effect of system gains, losses, and interference on the received signal samples.

Data Organization of Baseband Signals

You can use this software to efficiently implement space-time processing of complex-valued baseband samples by organizing the data in a three-dimensional matrix. See "Radar Data Cube" on page 3-2 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. This software specifies rectangular and spherical coordinates as column vectors with respect to both global and local origins. For a detailed explanation of the conventions, see:

- "Rectangular Coordinates"
- "Spherical Coordinates"
- "Global and Local Coordinate Systems"

Physical Quantities

This software uses the International System of Units (SI) almost exclusively for measurement. 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 the conventions.

Supported Data Types

This software supports only double-precision data types.

2

Phased Array Systems

System Overviews

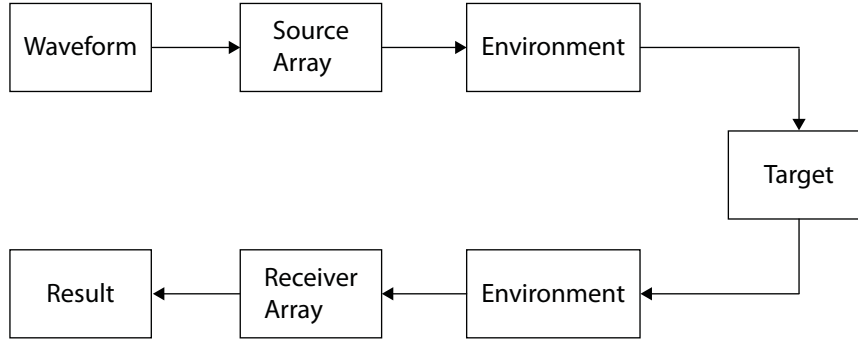
In this section...
"Phased Array System Overview" on page 2-2
"Phased Array Radar Overview" on page 2-4

Phased Array System Overview

Phased array systems use the spatial and temporal characteristics of propagating space-time wavefields to extract information about any 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. These areas include, but are not limited to:

- Signal detectability
- Spatial selectivity
- Source identification and localization

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



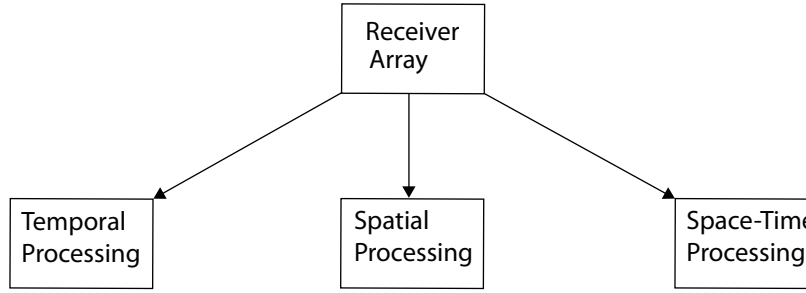
Phased array systems in diverse applications, such as radar, sonar, medical ultrasonography, medical imaging, and cellular phone communication share many common elements including:

- **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:

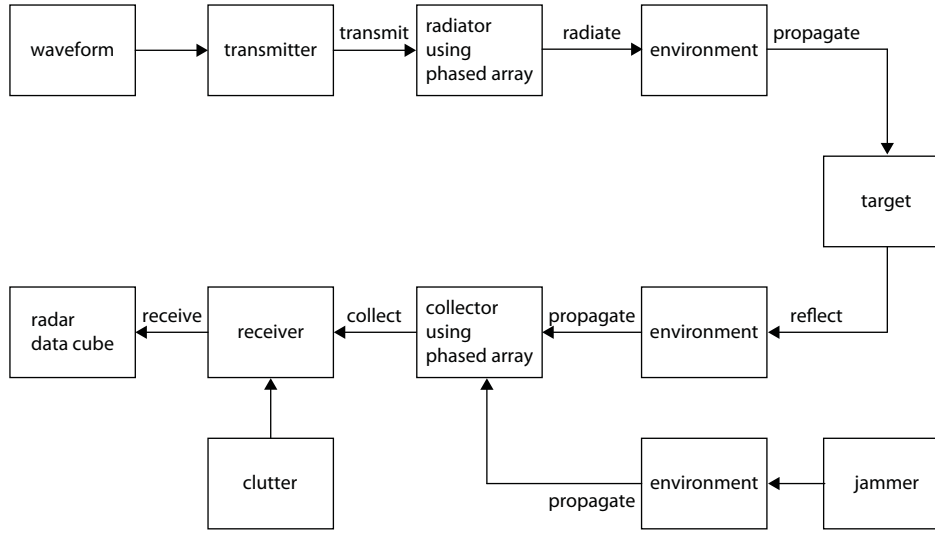
- **Temporal Processing** — Phased arrays often operate in poor signal-to-noise (SNR) ratios. Employing temporal integration and matched filtering improves the SNR. Knowing the propagation speed of the transmitted waveform and measuring the time it takes for a pulse to travel to and from a target allows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

Phased Array Radar Overview

The following figure presents an overview of a radar phased array system. The figure expands on the high-level overview shown in "Phased Array System Overview" on page 2-2.



To exploit the advantages of array processing, you must first understand how to model and optimize the performance of each component and operation in a phased array system. This software provides models for all the components

of the phased array system illustrated in the preceding figure from signal synthesis to signal analysis.

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

Waveform Synthesis

Phased Array System Toolbox software supports the design of rectangular, linear frequency-modulated, and linear stepped-frequency pulsed waveforms. To create such waveforms, you use `phased.RectangularWaveform`, `phased.LinearFMWaveform`, and `phased.SteppedFMWaveform`.

Physical Components and Environment Modeling

The software 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. These response patterns can encompass 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:

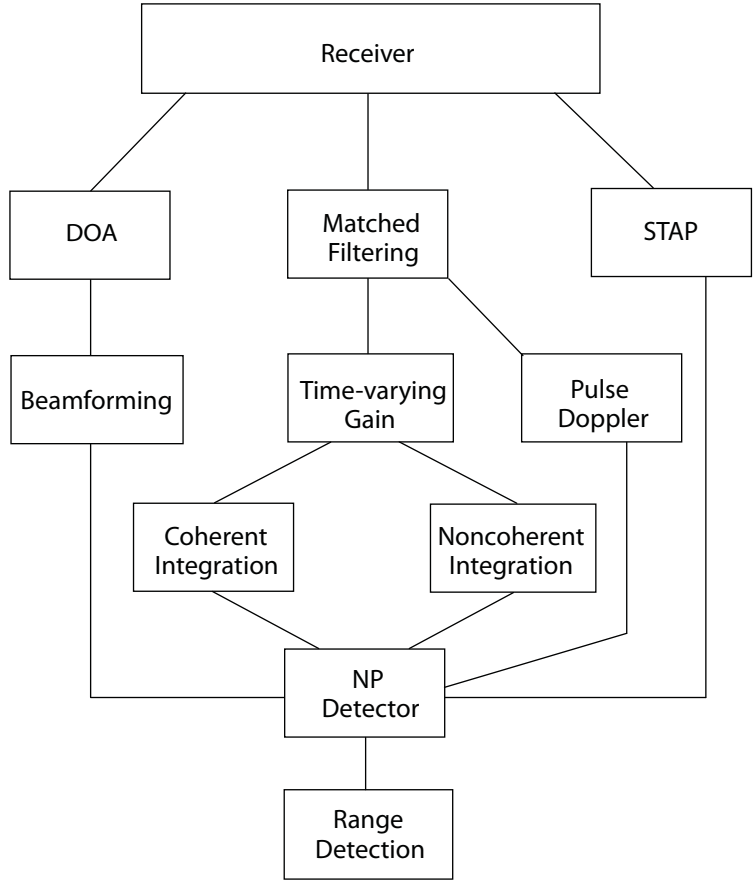
  - Uniform linear array (ULA) — `phased.ULA` enables you to model a uniform linear array consisting of sensor elements with isotropic or custom radiation patterns. You can specify the number of elements and element spacing.

  - 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, element spacing along two orthogonal axes, and lattice geometry.
  - Conformal array — `phased.ConformalArray` enables you to model a conformal array of sensor elements with isotropic or custom radiation patterns. To do so, specify 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** — You can simulate wideband interference with a user-specified radiated power, using `phased.BarrageJammer`.
- **Clutter** — You can simulate surface clutter using `phased.ConstantGammaClutter`.
- **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, Phased Array System Toolbox software 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 software rather than 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`. This object 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.
- **Detection** — A number of utility functions 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.
- **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™ product 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.

3

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

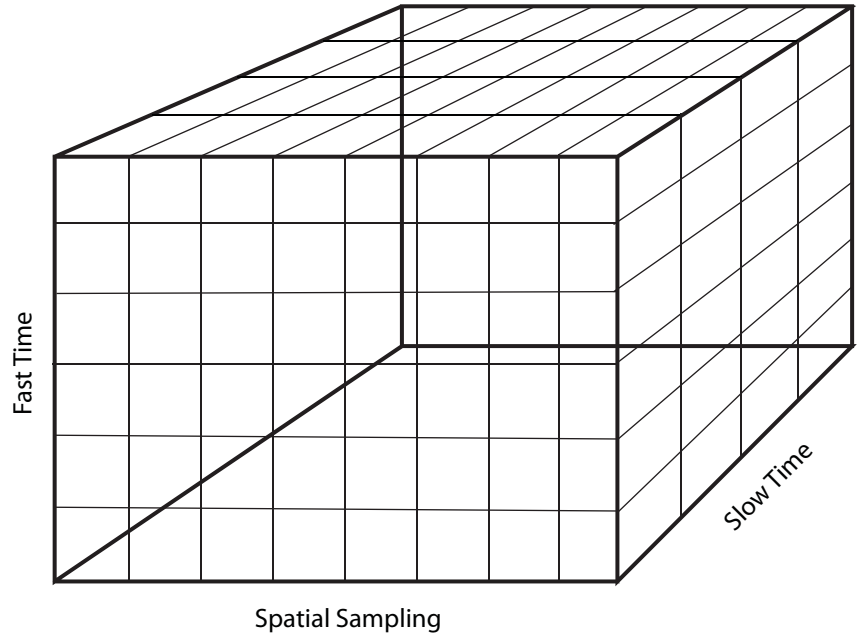
In this section...

"Radar Data Cube Concept" on page 3-2
"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

Radar Data Cube Concept

The radar data cube is a convenient way to conceptually represent space-time processing. To construct the radar data cube, assume that preprocessing 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 array of size *M*-by-*N*-by-*L*. Many radar signal processing operations in Phased Array System Toolbox software correspond to processing lower-dimensional subsets of the radar data cube. The subset could be a one-dimensional subvector or a two-dimensional submatrix.

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



Fast Time Samples

Consider an *M*-by-1 subvector of the radar data cube along the Fast Time axis in the above diagram. Each column vector represents a set of complex-valued baseband samples from a single pulse at one array element sampled at the rate F_s. This is the highest sampling rate of the system and leads to the designation *fast time*. F_s should be chosen to avoid aliasing. The corresponding sampling interval is given by $T_s = 1/F_s$. The fast time dimension is also referred to as the *range* dimension and the fast time sample intervals, when converted to distance using the signal propagation speed, are often 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. Because of the long sampling intervals, samples taken across multiple pulses are referred to 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 3-7

Units of Measure

Phased Array System Toolbox software almost exclusively uses SI base and derived units to measure physical quantities. The software does not provide any utilities for converting 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 Phased Array System Toolbox software are specified in degrees. See "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

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 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

Modeling and simulating phased array systems requires that you specify 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
Speed of light in a vacuum	299,792,458 m/s. Most commonly denoted by *c*.
Boltzmann constant relating energy to temperature.	1.38×10^{-23} J/K. Most commonly denoted by *k*.
Mean radius of the Earth	6,371,000 m

4

System Objects

- "What Is a System Toolbox?" on page 4-2
- "What Are System Objects?" on page 4-3
- "When to Use System Objects Instead of MATLAB Functions" on page 4-5
- "System Design and Simulation in MATLAB" on page 4-8
- "System Objects in Simulink" on page 4-9
- "System Object Methods" on page 4-10
- "System Design Using System Objects" on page 4-14

What Is a System Toolbox?

System Toolbox products provide algorithms and tools for designing, simulating, and deploying dynamic systems in MATLAB and Simulink®. These toolboxes contain MATLAB functions, System objects, and Simulink blocks that deliver the same design and verification capabilities across MATLAB and Simulink, enabling more effective collaboration among system designers. Available System Toolbox products include:

- DSP System Toolbox
- Communications System Toolbox
- Computer Vision System Toolbox
- Phased Array System Toolbox

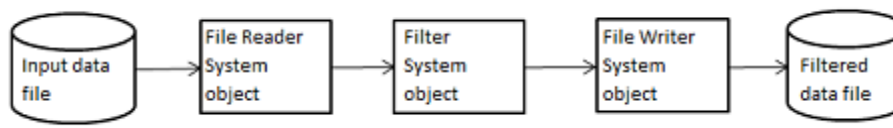
System Toolboxes support floating-point and fixed-point streaming data simulation for both sample- and frame-based data. They provide a programming environment for defining and executing code for various aspects of a system, such as initialization and reset. System Toolboxes also support code generation for a range of system development tasks and workflows, such as:

- Rapid development of reusable IP and test benches
- Sharing of component libraries and systems models across teams
- Large system simulation
- C-code generation for embedded processors
- Finite wordlength effects modeling and optimization
- Ability to prototype and test on real-time hardware

What Are System Objects?

A System object™ is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer™ license)
- C code generation (requires a MATLAB Coder™ or Simulink Coder license)
- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler license)

Note Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

When to Use System Objects Instead of MATLAB Functions

In this section...

"System Objects vs. MATLAB Functions" on page 4-5

"Process Audio Data Using Only MATLAB Functions Code" on page 4-5

"Process Audio Data Using System Objects" on page 4-6

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, your can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows code using only MATLAB functions to read audio data from a file, filter it, and then play the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);
maxSamples = audioInfo.TotalSamples;
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160,.15);
```

Initialize the filter states.

```
z = zeros(1,numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1
    audio = audioread(fname,[nIdx nIdx+frameSize-1]);
    [y,z] = filter(b,1,audio,z);
    sound(y,fs);
    nIdx = nIdx+frameSize;
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows code using System objects from the DSP System Toolbox™ software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname,'OutputDataType','single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

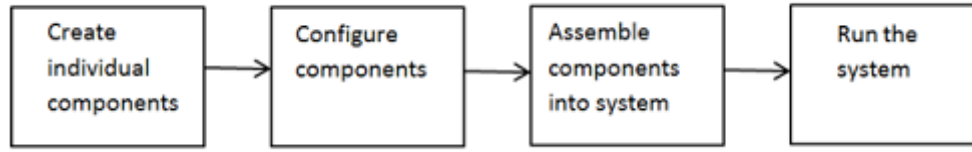
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects as shown in this diagram.



1. Create individual components — Create the System objects to use in your system. See "Create Components for Your System" on page 4-14 for information.

2. Configure components — If necessary, change the objects' property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See "Configure Components for Your System" on page 4-15 for information.

3. Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variable as inputs and outputs to simulate your system. See "Assemble Components to Create Your System" on page 4-16 for information.

4. Run the system — Run your program, which uses the `step` method to run your system's System objects. You can change tunable properties while your system is running. See "Run Your System" on page 4-18 and "Reconfigure Your System During Runtime" on page 4-18 for information.

System Objects in Simulink

You can also include System object code in Simulink models using the MATLAB Function block. This ability to include MATLAB code in Simulink. However, portions of the system are easier to implement in the MATLAB environment. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Object Methods

In this section...

"What Are System Object Methods?" on page 4-10

"The Step Method" on page 4-10

"Common Methods" on page 4-12

What Are System Object Methods?

After you create a System object, you use various 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 `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

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 objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

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.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that 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 Methods" on page 4-12.

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);
```

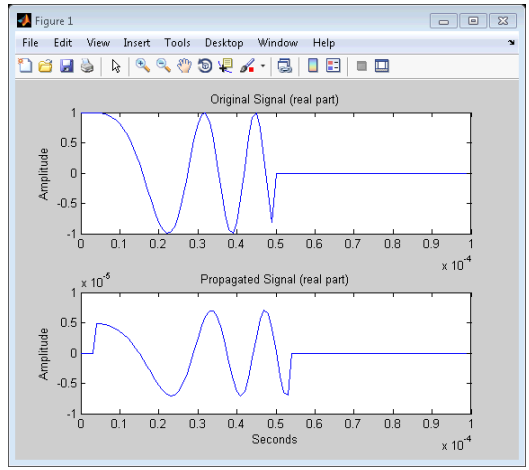
Construct a free space object with a propagation speed equal to the speed of light, an operating frequency of 3 GHz, and a sample rate of 1 MHz. The free space object is constructed to model one way propagation.

```
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],...
   [0;0;0],[0;0;0]);
% compare the original signal to the propagated waveform
t = unigrid(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 Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	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 `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks

Method	Description
	the object. For System objects, use the `release` method instead of a destructor.
`reset`	Resets the internal states of the object to the initial values for that object
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, `getDiscreteState` returns an empty structure.
`clone`	Creates another object of the same type with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked.
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design Using System Objects

In this section...
"Create Components for Your System" on page 4-14
"Configure Components for Your System" on page 4-15
"Assemble Components to Create Your System" on page 4-16
"Run Your System" on page 4-18
"Reconfigure Your System During Runtime" on page 4-18

Create Components for Your System

A System object is a component you can use to create your system in MATLAB. System objects support fixed- or variable-size data. *Variable-size data* is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at initialization time, and therefore, cannot change at run time.

Many System objects are predefined in the software. You can also define your own System objects (see "Define New System Objects").

This example shows the first step in designing a system that processes a long stream of audio data. The data is read from a file, filtered, and then played. The particular predefined components you need are:

- `dsp.AudioFileReader` — Read the file of audio data
- `dsp.FIRFilter` — Filter the audio data
- `dsp.AudioPlayer` — Play the filtered audio data

First, you create the component objects, using default property settings:

```
audioIn = dsp.AudioFileReader;
filtLP = dsp.FIRFilter;
audioOut = dsp.AudioPlayer;
```

Next, you configure each System object for your system. See "Configure Components for Your System" on page 4-15.

Note Alternately, if desired, you can "Create and Configure Components at the Same Time" on page 4-16.

Configure Components for Your System

When to Configure Components

If you did not set an object's properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See "Reconfigure Your System During Runtime" on page 4-18 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object's handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure the components for your system by setting the component objects' properties. Use this procedure if you have created your components as described in "Create Components for Your System" on page 4-14.

Note If you have not yet created your components, use the procedure in "Create and Configure Components at the Same Time" on page 4-16.

For the file reader object, specify the file to read and set the output data type.

```
audioIn.Filename = 'speech_dft_8kHz.wav';
audioIn.OutputDataType = 'single';
```

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

```
filtLP.Numerator = fir1(160,.15);
```

For the audio player object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut.SampleRate = audioIn.SampleRate;
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
    'OutputDataType','single')
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

After you create the components, you can assemble them in your system. See "Assemble Components to Create Your System" on page 4-16.

Assemble Components to Create Your System

- "Connect Inputs and Outputs" on page 4-17

• "Code for the Whole System" on page 4-17

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object's `step` method as the input to another object's `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see "System Object Methods" on page 4-10.

Code for the Whole System

This example shows the full code for reading, filtering, and playing a file of audio data.

You can type this code on the command line or put it into a program file.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
   'OutputDataType','single');
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);

while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system. See "Run Your System" on page 4-18.

Run Your System

- "How to Run Your System" on page 4-18
- "What You Cannot Change While Your System Is Running" on page 4-18

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see "Reconfigure Your System During Runtime" on page 4-18.

Reconfigure Your System During Runtime

- "When Can You Change Component Properties?" on page 4-19

- "Change a Tunable Property in Your System" on page 4-19
- "Change Input Complexity or Dimensions" on page 4-19

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object's reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change a Tunable Property in Your System

You can change the filter type to a high-pass filter as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as at the next iteration of the while loop).

```
reset(audioIn);               % Reset audio file
filtLP.Numerator = fir1(160,0.15,'high');
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

5

Basic Radar Workflow

- "Overview of Basic Workflow" on page 5-2
- "End-to-End Radar System" on page 5-3

Overview of Basic Workflow

The scenario and code examples contained in "End-to-End Radar System" on page 5-3 are intended as an introduction to the fundamental workflow used in Phased Array System Toolbox software. 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 many other 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. For information about the object properties, see 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 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.

End-to-End Radar System

In this section...

"Radar Scenario" on page 5-3
"Waveform" on page 5-3
"Antenna" on page 5-4
"Target Model" on page 5-4
"Antenna and Target Platforms" on page 5-4
"Modeling the Transmitter" on page 5-5
"Modeling Waveform Radiation and Collection" on page 5-6
"Modeling the Receiver" on page 5-6
"Modeling the Propagation Environment" on page 5-7
"Implementing the Basic Radar Model" on page 5-7

Radar Scenario

This example shows how to apply the basic toolbox workflow to 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 "Radar 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 “Rectangular Pulse Waveforms” for more detailed examples on building waveform models.

Antenna

To model the antenna described in “Radar 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 degees and elevation angles from –90 to 90 degrees.

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

Target Model

To model the target described in “Radar 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 “Radar 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. The target is initially located at `[7000; 5000; 0]` and moves with a constant velocity vector of `[-15;-10;0]`.

```
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]);
```

For definitions and conventions regarding *global* and *local* coordinates, see “Global and Local Coordinate Systems”.

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

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

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 “Radar 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 "Radar 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 "Radar 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,tgtvel] = 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,[0;0;0],tgtvel);
    % 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,tgtvel,[0;0;0]);
    % 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');
```