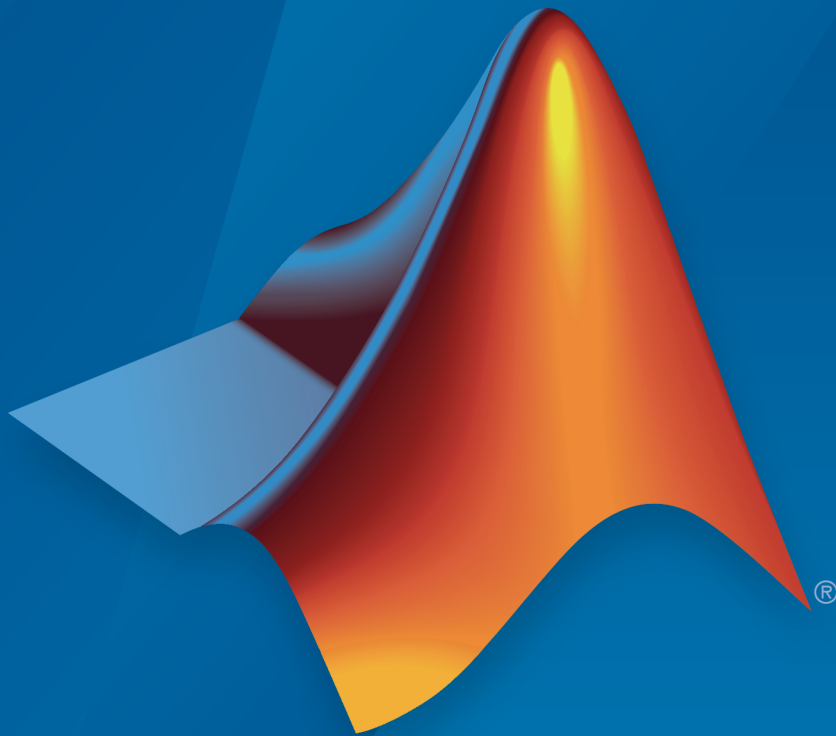# Phased Array System Toolbox™

Getting Started Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Revision History

April 2011	Online only	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)
September 2013	Online only	Revised for Version 2.1 (R2013b)
March 2014	Online only	Revised for Version 2.2 (R2014a)
October 2014	Online only	Revised for Version 2.3 (R2014b)
March 2015	Online only	Revised for Version 3.0 (R2015a)
September 2015	Online only	Revised for Version 3.1 (R2015b)
March 2016	Online only	Revised for Version 3.2 (R2016a)
September 2016	Online only	Revised for Version 3.3 (R2016b)

Contents

1

Getting Started with Phased Array System Toolbox Software

2

Phased Array Systems

3

Radar Data Cube, Units, and Physical Constants

4

System Objects

Basic Radar Workflow

5

Getting Started with Phased Array System Toolbox Software

Phased Array System Toolbox Product Description
Design and simulate phased array signal processing systems

Phased Array System Toolbox™ provides algorithms and apps for the design, simulation, and analysis of sensor array systems in radar, sonar, wireless communications, and medical imaging applications. The system toolbox includes pulsed and continuous waveforms and signal processing algorithms for beamforming, matched filtering, direction of arrival (DOA) estimation, and target detection. It also includes models for transmitters and receivers, propagation, targets, jammers, and clutter.

The system toolbox lets you model the dynamics of ground-based, airborne, or ship-borne multifunction radar systems with moving targets and platforms. You can design end-to-end phased array systems and analyze their performance under different scenarios using synthetic or acquired data. The toolbox apps let you explore the characteristics of sensor arrays and waveforms and perform link budget analysis. In-product examples provide a starting point for implementing custom phased array systems.

Toolbox algorithms are available as MATLAB® System objects and Simulink® blocks.

Key Features

- Multifunction radar system modeling, including actively electronically scanned array (AESA) and passively electronically scanned array (PESA)
- Radar theater models with moving targets, propagation channels, and interferences such as clutter and jammer
- URA, ULA, UCA, and conformal sensor arrays with perturbation and polarization effects
- Continuous and pulsed waveforms, including frequency modulated and phased coded pulses
- Digital beamforming algorithms for broadband and narrowband waveforms, including Capon and Frost
- Direction of arrival (DOA) algorithms, including monopulse, beamscan, MVDR, Root MUSIC, and ESPRIT
- Range and Doppler estimation and detection algorithms, including CFAR processing
- Space-time adaptive processing (STAP) algorithms, including sample matrix inversion (SMI) and adaptive DPCA pulse canceller

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

While the Phased Array System Toolbox software supports automatic generation of C code using MATLAB Coder™, there are several limitations. See "Code Generation" for more information about limitations on the use of MATLAB Coder with the Phased Array System 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.

Phased Array Systems

System Overviews

In this section...

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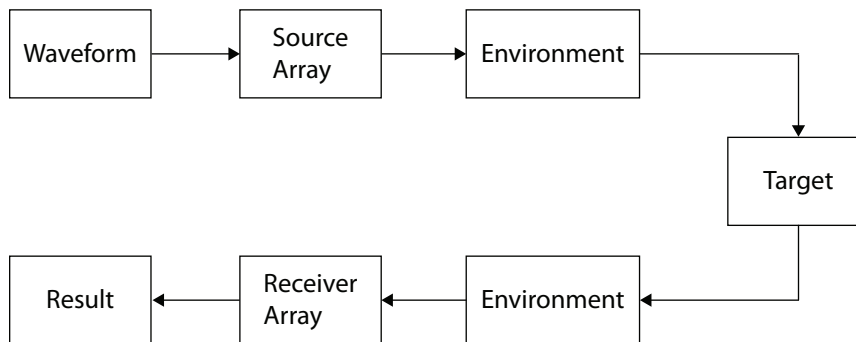
Phased array systems in diverse applications, such as radar, sonar, medical ultrasonography, medical imaging, and cellular phone communication share many common elements including:

- **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.
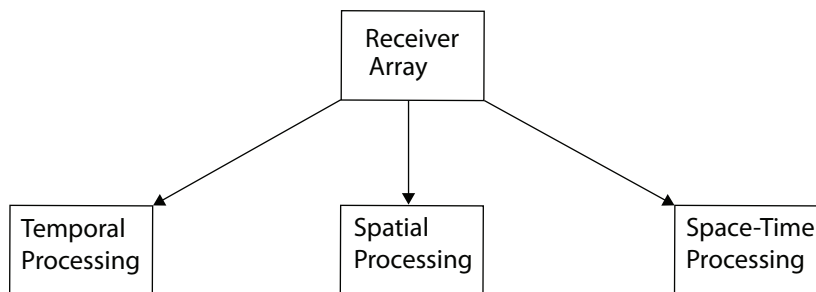
Brief descriptions of the three categories are:

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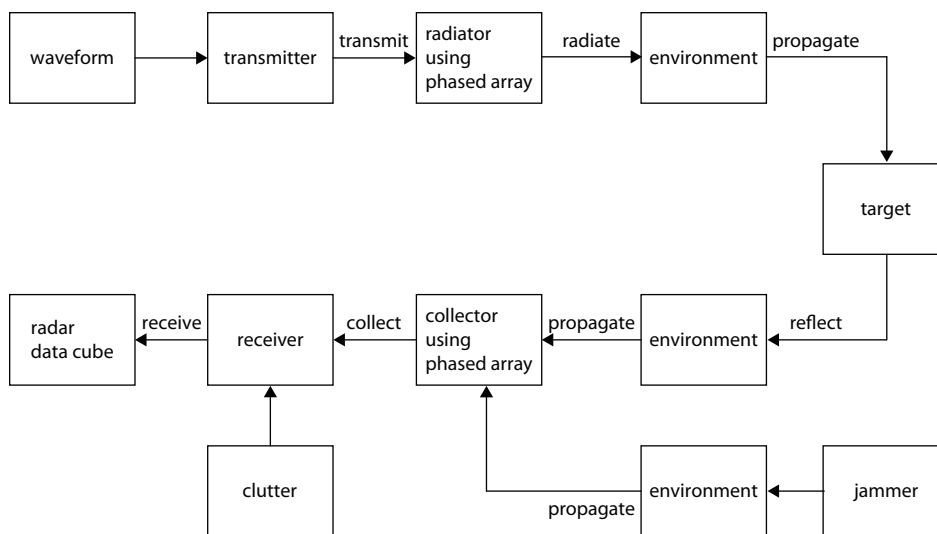
To exploit the advantages of array processing, you must first understand how to model and optimize the performance of each component and operation in a phased array system. This software provides models for all the components of the phased array system illustrated in the preceding figure from signal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

```
                        ┌─────────────────┐
                        │    Receiver     │
                        └─────────────────┘
             ┌───────────┐  ┌───────────┐  ┌──────────┐
             │    DOA    │  │  Matched  │  │   STAP   │
             │           │  │ Filtering │  │          │
             └───────────┘  └───────────┘  └──────────┘
          ┌─────────────┐ ┌─────────────┐ ┌──────────┐
          │ Beamforming │ │Time-varying │ │  Pulse   │
          │             │ │    Gain     │ │ Doppler  │
          └─────────────┘ └─────────────┘ └──────────┘
             ┌─────────────┐   ┌──────────────┐
             │  Coherent   │   │ Noncoherent  │
             │ Integration │   │ Integration  │
             └─────────────┘   └──────────────┘
                      ┌──────────────┐
                      │     NP       │
                      │  Detector    │
                      └──────────────┘
                      ┌──────────────┐
                      │    Range     │
                      │  Detection   │
                      └──────────────┘
```
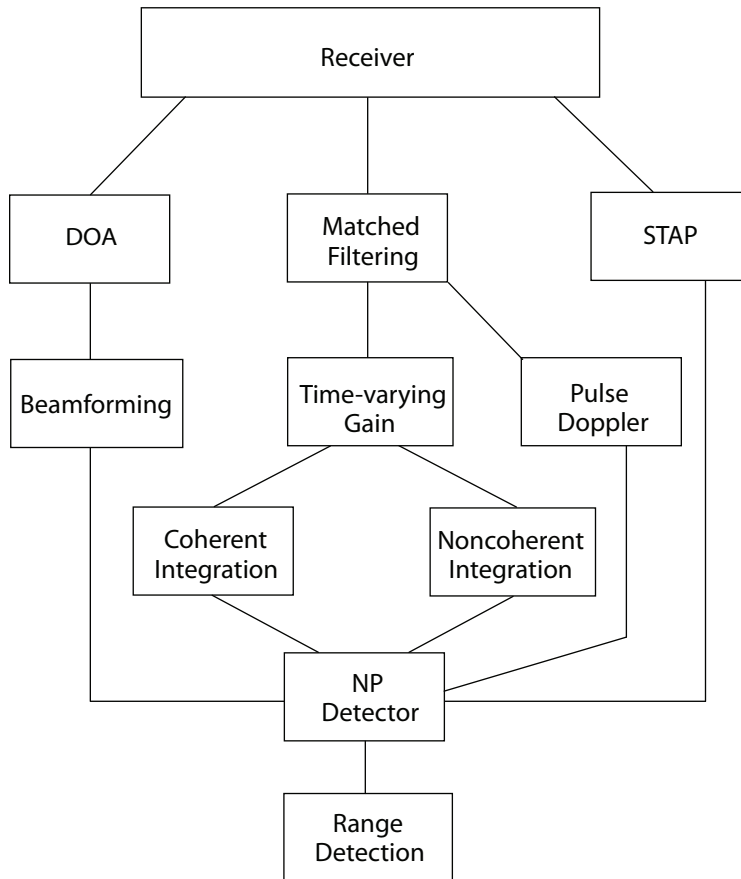
The preceding figure only presents an overview of the array signal processing operations supported by the software rather than predetermined orders of operation. For example, direction of arrival (DOA) estimation, beamforming, and space-time adaptive processing (STAP) often follow operations that improve 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.

Radar Data Cube, Units, and Physical Constants

Radar Data Cube

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

Slow Time

Spatial Sampling
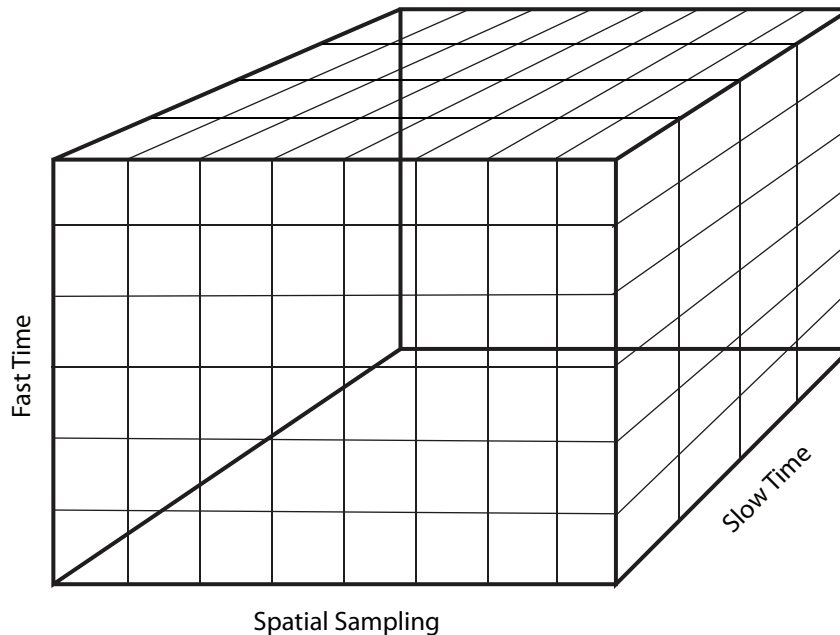
Fast Time Samples

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: `rad2deg` and `deg2rad`.

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?

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 in segments, such as video and audio processing systems. 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.

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 ensure that the filtering is performed correctly. This diagram represents a single loop of the system.
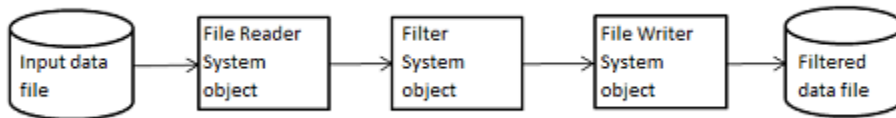
These advantages make System objects particularly well suited for processing streaming data

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.

System objects use a minimum of two commands to process data—a constructor to create the object (such as, `fft256 = dsp.FFT`) and a function call (such as, `fft256(x)`) 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.

In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See "Define System Objects".

System Objects vs. MATLAB Functions

In this section...

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, you can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB® function-only code for reading audio data.

The code reads audio data from a file, filters it, and then plays 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 in `processloop`.

```
type processloop
```

```
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 how to write System objects code for reading audio data.

The code uses 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.

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

Define the System object to filter the data.

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

Define the System object to play the filtered audio data.

```matlab
audioOut = audioDeviceWriter('SampleRate',audioIn.SampleRate);
```

Define the while loop to process the audio data.

```matlab
while ~isDone(audioIn)
    audio = audioIn();     % Read audio source file
    y = filtLP(audio);     % Filter the data
    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 device writer 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 in MATLAB as shown in this diagram.
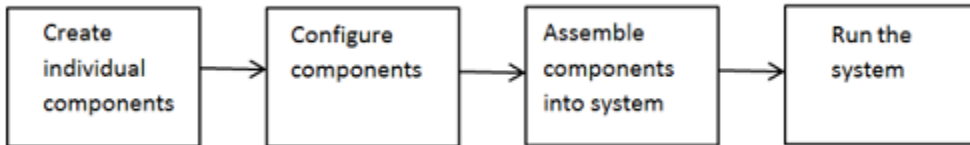
1 Create individual components — Create the System objects to use in your system. See "Create Components for Your System" on page 4-19. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See "Define System Objects".

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 "Component Properties" on page 4-20.

3 Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variables as inputs and outputs to simulate your system. See "Connecting System Objects" on page 4-21.

4 Run the system — Run your program. You can change tunable properties while your system is running. See "Run Your System" on page 4-22 and "Reconfiguring Objects" on page 4-22.

System Objects in MATLAB Code Generation

In this section...
"System Objects in Generated Code" on page 4-9
"System Objects in codegen" on page 4-12
"System Objects in the MATLAB Function Block" on page 4-12
"System Objects and MATLAB Compiler Software" on page 4-12

System Objects in Generated Code

You can generate C/C++ code in MATLAB from your system that contains System objects by using MATLAB Coder. You can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms.

System Objects Code with Persistent Objects for Code Generation

For another detailed code generation example, see "Generate Code for MATLAB Handle Classes and System Objects" in the MATLAB Coder product documentation.

Usage Rules and Limitations for System Objects for Generating Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty()`.
- Set arguments to System object constructors as compile-time constants.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- System objects accept a maximum of 1024 inputs. A maximum of 8 dimensions per input is supported.
- The data type of the inputs should not change.

- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

> **Note:** Variable-size properties in `MATLAB Function` block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the `MATLAB Function` block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a `MATLAB Function` block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.

  - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
  - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation. For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.

- Objects cannot be used as default values for properties.

Global Variables

- Global variables are allowed in a System object, unless you will be using that System object in Simulink via the MATLAB System block. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

```
f = coder.MEXConfig;
f.GlobalSyncMethod = 'NoSync'
```
Then, include '-config f' in your codegen command.

Methods

- Code generation support is available only for these System object methods:

  - get
  - getNumInputs
  - getNumOutputs
  - isDone (for sources only)
  - isLocked
  - release
  - reset
  - set (for tunable properties)
  - step

- For System objects that you define,

  Code generation support is available only for these methods:

  - getDiscreteStateImpl
  - getNumInputsImpl
  - getNumOutputsImpl
  - infoImpl
  - isDoneImpl
  - isInputDirectFeedThroughImpl
  - outputImpl
  - processTunedPropertiesImpl
  - releaseImpl — Code is not generated automatically for this method. To release an object, you must explicitly call the release method in your code.
  - resetImpl
  - setupImpl
  - stepImpl

- `updateImpl`
- `validateInputsImpl`
- `validatePropertiesImpl`

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See "Getting Started with MATLAB Coder" and "MATLAB Classes" for more information.

Note: Most, but not all, System objects support code generation. Refer to the particular object's reference page for information.

System Objects in the MATLAB Function Block

Using the `MATLAB Function` block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see "What Is a MATLAB Function Block?" in the Simulink documentation.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

System Objects in Simulink

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using the `MATLAB Function` block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. 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

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. The syntax for using methods is `<method>(<system object name>)`, plus possible extra input arguments. For example, for `txfourier = dsp.FFT`, where `txfourier` is a name you assign, you call the `reset` method using `reset(txfourier)`. For more information about methods, see the descriptions in "Common Methods" on page 4-16.

Running a System Object

To run a System object and perform the operation defined by its algorithm, you call the object as if it were a function. For example, to create an FFT object that uses the `dsp.FFT` System object, specifies a length of 1024, and names it `dft`, use

```
dft = dsp.FFT;
```
To run this object on input `x`, use

```
dft(x);
```
If you run a System object without any input arguments, you must include empty parentheses. For example, `asysobj()`.

When you run a System object, it also performs other important tasks related to data processing, such as initialization and handling object states.

Note: An alternative way to run a System object is to use the `step` method. For example, for an object created using `dft = dsp.FFT`, you can run it using `step(dft,x)`.

Calculate Propagation of Signal in Free Space

This example how to use two different `step` methods. The first `step` method is associated with the phased.LinearFMWaveform System object™. The second `step` method is associated with the phased.FreeSpace System 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.

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

Obtain the transmitted waveform.

```
sig = sLFM();
```

Construct a `phased.Freespace` System object™ having a propagation speed equal to the speed of light, an operating frequency of 3 GHz, and a sample rate of 1 MHz. The Freespace object models one-way propagation. Then, propagate the transmitted signal from coordinates *(0,0,0)* to *(500,1000,20)* meters.

```
sFS = phased.FreeSpace(...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',3e9,'TwoWayPropagation',false,...
    'SampleRate',1e6);
propsig = sFS(sig,[0; 0; 0],[500; 1e3; 20],...
    [0;0;0],[0;0;0]);
```

Plot the real part of the transmitted signal and the real part of the propagated signal for comparison.

```
t = 1000000*unigrid(0,1/sFS.SampleRate,length(sig)*1/sFS.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('Time (\mu sec)')
ylabel('Amplitude')
```
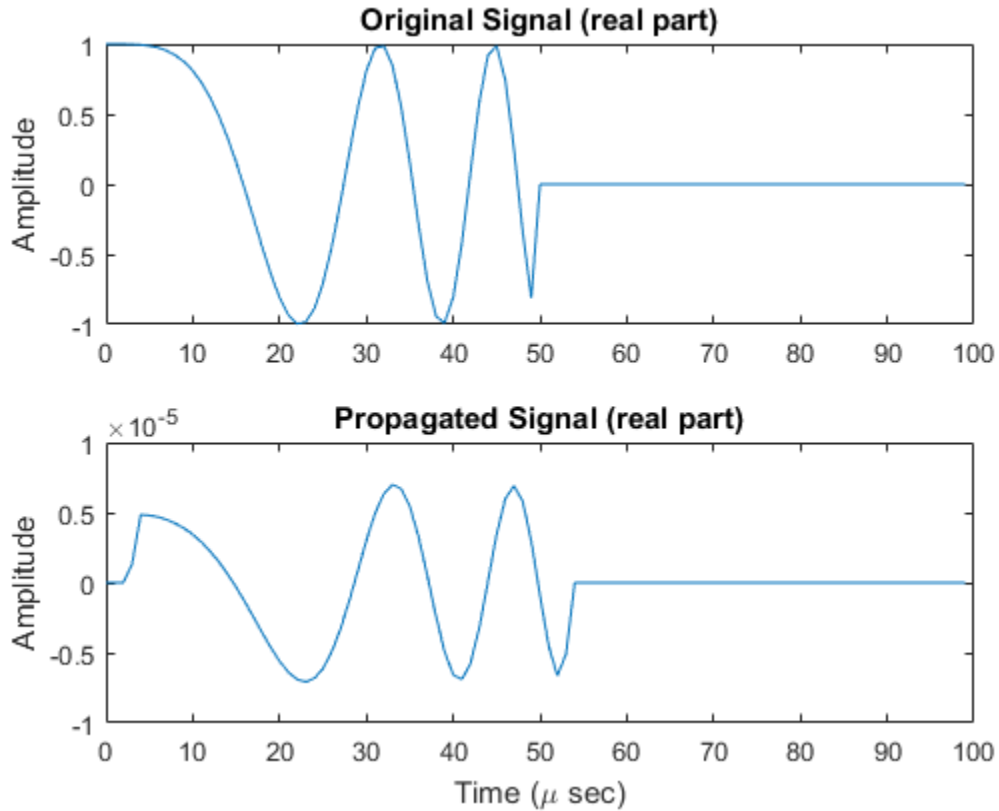
Common Methods

All System objects support the following methods, each of which is linked from each object's reference page. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
Run System object like a function, or `step`	Run the object to process data using the algorithm defined by that object.

Method	Description
	Example: For `dft = dsp.FFT;`, the standard way to run the object is `y = dft(x)`. The alternative way is `y = step(dft,x)`.
	As part of this processing, the object initializes needed resources, returns outputs, and updates the object states. After you start running the object, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. Both ways of running a System object return regular MATLAB variables.
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the `release` method instead of a destructor.
`reset`	Resets the internal states of a locked object to the initial values for that object and leaves the object locked
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected running the object. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from running the object. 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 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 end of the data file has been reached. If a particular object does not have end-of-data capability, this method value returns `false`.

Method	Description
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 in MATLAB Using System Objects

In this section...

Predefined Components

The example in the next section shows how to use System objects that are predefined in the software. If you use a function to create and use a System object, specify the object creation using conditional code. This will prevent errors if that function is called within a loop.

You can also create your own System objects (see "Define System Objects").

Create Components for Your System

This section of the example shows how to set up your system. The predefined components you need are:

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

First, create the component objects, using default property settings.

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

```
audioOut = audioDeviceWriter;
```

Component Properties

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 "Reconfiguring Objects" on page 4-22 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 section of the 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 separately from configuring them. You can also create and configure your components at the same time, as described in a later example.

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

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

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

```
audioIn.Filename = 'speech_dft_8kHz.wav';
audioIn.OutputDataType = 'single';
filtLP.Numerator = fir1(160,.15);
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. Specify each property with a `'Name'`, `Value` argument pair.

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 set the sample rate to the same rate as the input data.

```
audioOut = audioDeviceWriter('SampleRate',audioIn.SampleRate);
```

Connecting System Objects

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 that System objects use a two-step process. First you create the object and set its parameters and then, you run the object. Running the object initializes it and controls the data flow and state management of your system. You typically call a System object within a code loop.

You use the output from an object as the input to another object. 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 4-14.

Connect Components in a System

This section shows how to connect the components together to read, filter, and play a file of audio data. The while loop uses the `isDone` method to read through the entire file.

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

Run Your System

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, data is processed through your objects.

What You Cannot Change While Your System Is Running

The first call to a System object initializes and then locks the 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 "Reconfiguring Objects" on page 4-22.

Reconfiguring Objects

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 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.

Change a Tunable Property in Your System

You can change the filter type to a high-pass filter as your code is running by modifying the `Numerator` property of the filter object. The change takes effect at the next iteration of the while loop.

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

Basic Radar Workflow

Overview of Basic Workflow

The scenario and code examples contained in "End-to-End Radar System" on page 5-3 serve 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

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 μs 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.

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

Waveform Model

To create the waveform, use the `phased.RectangularWaveform` System object™ and set the properties to the desired values.

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

See "Rectangular Pulse Waveforms" for more detailed examples on creating waveform.

Antenna Model

To model the antenna, use the `phased.IsotropicAntennaElement` System object. 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° and elevation angles from -°90 to 90°.

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

Target Model

To model the target, use the `phased.RadarTarget` System object. 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.

```
target = 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, use the `phased.Platform` System object. 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)*.

```
antennaplatform = phased.Platform('InitialPosition',[0;0;0],'Velocity',[0;0;0]);
targetplatform = phased.Platform('InitialPosition',[7000; 5000; 0],...
    'Velocity',[-15;-10;0]);
```

For definitions and conventions regarding coordinate systems, see "Global and Local Coordinate Systems".

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

```
[tgtrng,tgtang] = rangeangle(targetplatform.InitialPosition,...
    antennaplatform.InitialPosition);
```

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

Modeling Transmitter

To model the transmitter specifications, use the `phased.Transmitter` System object. 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 the `albersheim` function 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. Use the `radareqpow` function to determine the required peak transmit power.

```
maxrange = 1.5e4;
lambda = physconst('LightSpeed')/4e9;
tau = waveform.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 record the transmitter status.

```
transmitter = 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 that use the radar equation.

Modeling Waveform Radiation and Collection

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

In this example, the pulse satisfies the narrowband signal assumption. The carrier frequency is 4 GHz. For the value of the Sensor property, insert use the handle for the isotropic antenna. In the phased.Collector System object, set the Wavefront property to 'Plane' to specify that the incident waveform on the antenna is a plane wave.

```
radiator = phased.Radiator('Sensor',antenna,...
    'PropagationSpeed',physconst('LightSpeed'),'OperatingFrequency',4e9);
collector = phased.Collector('Sensor',antenna,...
    'PropagationSpeed',physconst('LightSpeed'),'Wavefront','Plane',...
    'OperatingFrequency',4e9);
```

Modeling Receiver

To model the receiver, use the phased.ReceiverPreamp System object. 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 Kelvin. Seed the random number generator to a fixed value for reproducible results.

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

See "Receiver Preamp" for more details.

Modeling Propagation

To model the propagation environment, use the `phased.FreeSpace` System object. You can model one-way or two-propagation by setting the `TwoWayPropagation` property. In this example, set this property to `false` to model one-way propagation.

```matlab
channel = 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 scenario, 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

```matlab
T = 1/waveform.PRF;
% Get antenna position
txpos = antennaplatform.InitialPosition;
% Allocate array for received echoes
rxsig = zeros(waveform.SampleRate*T,numpulses);
```

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

```matlab
for n = 1:numpulses
    % Update the target position
    [tgtpos,tgtvel] = targetplatform(T);
    % Get the range and angle to the target
    [tgtrng,tgtang] = rangeangle(tgtpos,txpos);
    % Generate the pulse
    sig = waveform();
    % Transmit the pulse. Output transmitter status
    [sig,txstatus] = transmitter(sig);
```

```matlab
    % Radiate the pulse toward the target
    sig = radiator(sig,tgtang);
    % Propagate the pulse to the target in free space
    sig = channel(sig,txpos,tgtpos,[0;0;0],tgtvel);
    % Reflect the pulse off the target
    sig = target(sig);
    % Propagate the echo to the antenna in free space
    sig = channel(sig,tgtpos,txpos,tgtvel,[0;0;0]);
    % Collect the echo from the incident angle at the antenna
    sig = collector(sig,tgtang);
    % Receive the echo at the antenna when not transmitting
    rxsig(:,n) = receiver(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.

```matlab
rxsig = pulsint(rxsig,'noncoherent');
t = unigrid(0,1/receiver.SampleRate,T,'[)');
rangegates = (physconst('LightSpeed')*t)/2;
plot(rangegates/1e3,rxsig)
hold on
xlabel('range (km)')
ylabel('Power');
ylim = get(gca,'YLim');
plot([tgtrng/1e3,tgtrng/1e3],[0 ylim(2)],'r')
hold off
```